
SLEPc for Python

Release 3.25.0

Lisandro Dalcin

May 01, 2026

Contents

1	Discussion and Support	1
2	Acknowledgments	2
3	Contents	2
3.1	Overview	2
3.2	Tutorial	4
3.3	Installation	5
3.4	Citations	6
3.5	CHANGES	6
3.6	Reference	8
3.7	slepc4py demos	392
	Python Module Index	431
	Index	432

Abstract

This document describes [slepc4py](#), a [Python](#) wrapper 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. It also provides more advanced functionality such as solvers for nonlinear eigenvalue problems (either polynomial or general) and matrix functions.

[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 Discussion and Support

- You can write to the [PETSc Users Mailing List](#)
- See also the [contact information for SLEPc](#)

- Issue Tracker: <https://gitlab.com/slepc/slepc/-/issues>
- Git Repository: <https://gitlab.com/slepc/slepc.git> (the source code is in `src/binding/slepc4py`)
- Current release in PyPI: <https://pypi.org/project/slepc4py/>

2 Acknowledgments

L. Dalcin was partially supported by the Extreme Computing Research Center (ECRC), Division of Computer, Electrical, and Mathematical Sciences & Engineering (CEMSE), King Abdullah University of Science and Technology (KAUST).

See additional acknowledgements related to [the SLEPc project](#).

3 Contents

3.1 Overview

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

SLEPc^{1,2} 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.

Features

Currently, the following types of eigenproblems can be addressed:

- Standard eigenvalue problem, $Ax = \lambda x$, either for Hermitian or non-Hermitian matrices.
- Generalized eigenvalue problem, $Ax = \lambda Bx$, either Hermitian positive-definite or not.
- Partial singular value decomposition of a rectangular matrix, $Au = \sigma v$.
- Generalized singular values of a matrix pair, $Ax = cu$, $Bx = sv$.
- Polynomial eigenvalue problem, $P(\lambda) = 0$.
- Nonlinear eigenvalue problem, $T(\lambda) = 0$.
- 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 methods such as LOBPCG.

¹ J. E. Roman, C. Campos, L. Dalcin, E. Romero, A. Tomas. SLEPc Users Manual. DSIC-II/24/02 - Revision 3.25. D. Sistemas Informaticos y Computacion, Universitat Politecnica de Valencia. 2026.

² 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. <https://doi.org/10.1145/1089014.1089019>

- A contour integral solver using high-order moments.

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

- Use an eigensolver via the cross-product matrix A^*A or the cyclic matrix $\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$.
- Explicitly restarted Lanczos bidiagonalization.
- Implicitly restarted Lanczos bidiagonalization (thick-restart Lanczos).
- A basic randomized solver.

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.
- Jacobi-Davidson for polynomial eigenproblems.
- A contour integral solver using high-order moments.

For general nonlinear eigenvalue problems, the following methods can be used:

- Solve a polynomial eigenproblem obtained via polynomial interpolation.
- Rational interpolation and linearization (NLEIGS).
- Newton-type methods such as SLP or RII.
- A subspace projection method (nonlinear Arnoldi).
- A contour integral solver using high-order moments.

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 $A - \sigma I$ in order to compute eigenvalues closest to a given target value, σ .
- Harmonic extraction, a cheap alternative to shift-and-invert that also tries to approximate eigenvalues closest to a target, σ , 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 setting at the command line.
- Ability to do the computation in parallel and/or using GPUs.

Components

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

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 behavior 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, $T(\lambda) = 0$. The user provides the parameter-dependent matrix T via the split form or by means of callback functions.

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 `MFNSolve(mfn,b,x)` computes $x = f(A)b$, where f is a function such as the exponential.

LME

This component provides the functionality for solving linear matrix equations such as Lyapunov or Sylvester where the solution has low rank.

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.

RG

A component used to define a region of the complex plane such as an ellipse or a rectangle. This is required by end-users in some cases such as contour-integral eigensolvers.

In addition to the above components, some extra functionality is provided in the `Sys` and `Util` sections.

3.2 Tutorial

Below we provide links to tutorial examples intended for learning the basics of `slepc4py`. For additional information, the reader is referred to:

- `slepc4py` [Reference](#) (manual pages for all `slepc4py` classes and methods).
- [SLEPc documentation](#).
- [petsc4py documentation](#).

Commented source of demo examples

- To get started, we recommend having a look at the source code of example `demo/ex1.py`, that has comments inserted inline. It is also [available online here](#). This example solves a standard symmetric eigenvalue problem.
- Demo examples for other problems, including linear and nonlinear eigenvalue problems and singular value problems, are available in the section *[slepc4py demos](#)*.

3.3 Installation

Install from PyPI using pip

You can use **pip** to install `slepc4py` and its dependencies.

If you have a working MPI implementation and the `mpicc` compiler wrapper is on your search path, it is highly recommended to install `mpi4py` first:

```
$ python -m pip install mpi4py
```

Ensure you have `NumPy` and `petsc4py` installed:

```
$ python -m pip install numpy petsc4py
```

Finally, install `slepc4py`:

```
$ python -m pip install slepc slepc4py
```

If you already have working PETSc and SLEPc installs, set environment variables `SLEPC_DIR` and `PETSC_DIR` (and perhaps `PETSC_ARCH` for non-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
$ python -m pip install petsc4py slepc4py
```

Install from the SLEPc source tree

If you also want to install `petsc4py` from the PETSc source tree, follow the instructions in the [petsc4py installation](#) page.

Set the `PETSC_DIR` and `PETSC_ARCH` environment variables, as well as `SLEPC_DIR`. Follow the instructions to [build SLEPc](#). Then `cd` to the top of the SLEPc source tree and run:

```
$ python -m pip install src/binding/slepc4py
```

The installation of `slepc4py` supports multiple `PETSC_ARCH` in the form of a colon separated list:

```
$ PETSC_ARCH='arch-0:...:arch-N' python -m pip install src/binding/slepc4py
```

If you are cross-compiling, and the `numpy` module cannot be loaded on your build host, then before invoking **pip**, set the `NUMPY_INCLUDE` environment variable to the path that would be returned by `import numpy; numpy.get_include()`:

```
$ export NUMPY_INCLUDE=/usr/lib/pythonX/site-packages/numpy/core/include
```

3.4 Citations

If SLEPc for Python has been significant to a project that leads to an academic publication, please acknowledge that fact by citing the project.

- L. Dalcin, P. Kler, R. Paz, and A. Cosimo, *Parallel Distributed Computing using Python*, Advances in Water Resources, 34(9):1124-1139, 2011. <https://doi.org/10.1016/j.advwatres.2011.04.013>
- V. Hernandez, J.E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Transactions on Mathematical Software, 31(3):351-362, 2005. <https://doi.org/10.1145/1089014.1089019>

There is also a list of [SLEPc-related references](#).

3.5 CHANGES

Release 3.25

- Update to SLEPc 3.25.

Release 3.24

- Update to SLEPc 3.24.
- Support (opt-in via setting the environment variable `SLEPC4PY_BUILD_PYSABI=1`) for building with `Py_LIMITED_API` (Python Stable ABI) under Python 3.10+ (requires Cython 3.1+).
- Add support for standard Python operators for BV and FN classes.
- Add new LME class.

Release 3.23

- Update to SLEPc 3.23.

Release 3.22

- Update to SLEPc 3.22.
- In `slepc4py` now `EPS.getEigenpair()` and `EPS.getEigenvalue()` will return a real value instead of a complex, if the problem is of Hermitian or generalized Hermitian type.

Release 3.21

- Update to SLEPc 3.21.

Release 3.20

- Update to SLEPc 3.20.

Release 3.19

- Update to SLEPc 3.19.

Release 3.18

- Update to SLEPc 3.18.

Release 3.17

- Update to SLEPc 3.17.

Release 3.16

- Update to SLEPc 3.16.

Release 3.15

- Update to SLEPc 3.15.
- Updates in installation scripts.

Release 3.14

- Update to SLEPc 3.14.

Release 3.13

- Update to SLEPc 3.13.

Release 3.12

- Update to SLEPc 3.12.

Release 3.11

- Update to SLEPc 3.11.

Release 3.10

- Update to SLEPc 3.10.

Release 3.9

- Update to SLEPc 3.9.

Release 3.8

- Update to SLEPc 3.8.

Release 3.7

- Update to SLEPc 3.7.

Release 3.6

- Update to SLEPc 3.6.

Release 3.5

- Update to SLEPc 3.5.
- Add RG class introduced in SLEPc 3.5 release.
- Add PySlepcXXX_New/Get C API functions.
- Fix compilation problem with complex scalars on OS X.
- Fix outdated SWIG interface file.

Release 3.4

- Update to SLEPc 3.4.

Release 3.3

- Update to SLEPc 3.3.
- Regenerate the wrappers using Cython 0.18 and fix binary compatibility issues with petsc4py 3.3.1.

Release 1.2

- Update to SLEPc 3.2.

Release 1.1

- Support for new QEP quadratic eigenproblem solver in SLEPc.
- Support for `pip install slepc4py` to download and install SLEPc.
- Support for PETSc/SLEPc static library builds (Linux-only).
- Preliminary support for Python 3.

Release 1.0.0

- This is the first release of the all-new, Cython-based, implementation of *SLEPc for Python*.

3.6 Reference

<code>slepc4py</code>	SLEPc for Python
<code>slepc4py.typing</code>	Typing support.
<code>slepc4py.SLEPc</code>	Scalable Library for Eigenvalue Problem Computations

slepc4py

SLEPc for Python

This package is an interface to [SLEPc](#) libraries.

[SLEPc](#) (the Scalable Library for Eigenvalue Problem Computations) is a software library for the solution of large scale sparse eigenvalue problems on parallel computers. It is an extension of [PETSc](#) and can be used for either standard or generalized eigenproblems, 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.

Functions

<code>get_config()</code>	Return a dictionary with information about SLEPc.
<code>get_include()</code>	Return the directory in the package that contains header files.
<code>init([args, arch, comm])</code>	Initialize SLEPc.

slepc4py.get_config

`slepc4py.get_config()`

Return a dictionary with information about SLEPc.

Return type

`dict[str, str]`

slepc4py.get_include

`slepc4py.get_include()`

Return the directory in the package that contains header files.

Extension modules that need to compile against slepc4py should use this function to locate the appropriate include directory.

Example

Using Python distutils or NumPy distutils:

```
import petsc4py, slepc4py
Extension('extension_name', ...
        include_dirs=[...,
                      petsc4py.get_include(),
                      slepc4py.get_include(),])
```

Return type

`str`

slepc4py.init

`slepc4py.init(args=None, arch=None, comm=None)`

Initialize SLEPc.

Parameters

- **args** (`str` | `list[str]` | `None`) – Command-line arguments, usually the `sys.argv` list.
- **arch** (`str` | `None`) – Specific configuration to use.
- **comm** (`Intracomm` | `None`) – MPI communicator.

Return type

`None`

Notes

This function should be called only once, typically at the very beginning of the bootstrap script of an application.

slepc4py.typing

Typing support.

Attributes

<code>Scalar</code>	Scalar type.
<code>ArrayBool</code>	Array of <code>bool</code> .
<code>ArrayInt</code>	Array of <code>int</code> .
<code>ArrayReal</code>	Array of <code>float</code> .
<code>ArrayComplex</code>	Array of <code>complex</code> .
<code>ArrayScalar</code>	Array of <code>Scalar</code> numbers.
<code>LayoutSizeSpec</code>	<code>int</code> or 2-tuple of <code>int</code> describing the layout sizes.
<code>EPSStoppingFunction</code>	<code>EPS</code> stopping test callback.
<code>EPSSArbitraryFunction</code>	<code>EPS</code> arbitrary selection callback.
<code>EPSEigenvalueComparison</code>	<code>EPS</code> eigenvalue comparison callback.
<code>EPSPMonitorFunction</code>	<code>EPS</code> monitor callback.
<code>PEPStoppingFunction</code>	<code>PEP</code> stopping test callback.
<code>PEPEigenvalueComparison</code>	<code>PEP</code> eigenvalue comparison callback.
<code>PEPMonitorFunction</code>	<code>PEP</code> monitor callback.
<code>NEPStoppingFunction</code>	<code>NEP</code> stopping test callback.
<code>NEPEigenvalueComparison</code>	<code>NEP</code> eigenvalue comparison callback.
<code>NEPMonitorFunction</code>	<code>NEP</code> monitor callback.
<code>NEPFunction</code>	<code>NEP</code> Function callback.
<code>NEPJacobian</code>	<code>NEP</code> Jacobian callback.
<code>SVDStoppingFunction</code>	<code>SVD</code> stopping test callback.
<code>SVDMonitorFunction</code>	<code>SVD</code> monitor callback.
<code>MFNMonitorFunction</code>	<code>MFN</code> monitor callback.
<code>LMEMonitorFunction</code>	<code>LME</code> monitor callback.

`slepc4py.typing.Scalar`

`slepc4py.typing.Scalar = float | complex`

Scalar type.

Scalars can be either `float` or `complex` (but not both) depending on how PETSc was configured (`./configure --with-scalar-type=real|complex`).

`slepc4py.typing.ArrayBool`

`slepc4py.typing.ArrayBool`

Array of `bool`.

alias of `ndarray[tuple[Any, ...], dtype[bool]]`

`slepc4py.typing.ArrayInt`

`slepc4py.typing.ArrayInt`

Array of `int`.

alias of `ndarray[tuple[Any, ...], dtype[integer]]`

`slepc4py.typing.ArrayReal`

`slepc4py.typing.ArrayReal`

Array of `float`.

alias of `ndarray[tuple[Any, ...], dtype[floating]]`

slepc4py.typing.ArrayComplex

`slepc4py.typing.ArrayComplex`

Array of `complex`.

alias of `ndarray[tuple[Any, ...], dtype[complexfloating]]`

slepc4py.typing.ArrayScalar

`slepc4py.typing.ArrayScalar`

Array of *Scalar* numbers.

alias of `ndarray[tuple[Any, ...], dtype[floating | complexfloating]]`

slepc4py.typing.LayoutSizeSpec

`slepc4py.typing.LayoutSizeSpec = int | tuple[int, int]`

`int` or 2-`tuple` of `int` describing the layout sizes.

A single `int` indicates global size. A `tuple` of `int` indicates (`local_size`, `global_size`).

slepc4py.typing.EPSStoppingFunction

`slepc4py.typing.EPSStoppingFunction`

EPS stopping test callback.

alias of `Callable[[EPS, int, int, int, int], ConvergedReason]`

slepc4py.typing.EPSArbitraryFunction

`slepc4py.typing.EPSArbitraryFunction`

EPS arbitrary selection callback.

alias of `Callable[[float | complex, float | complex, Vec, Vec, float | complex, float | complex], [float | complex, float | complex]]`

slepc4py.typing.EPSEigenvalueComparison

`slepc4py.typing.EPSEigenvalueComparison`

EPS eigenvalue comparison callback.

alias of `Callable[[float | complex, float | complex, float | complex, float | complex], int]`

slepc4py.typing.EPSMonitorFunction

`slepc4py.typing.EPSMonitorFunction`

EPS monitor callback.

alias of `Callable[[EPS, int, int, ndarray[tuple[Any, ...], dtype[floating | complexfloating]], ndarray[tuple[Any, ...], dtype[floating | complexfloating]], ndarray[tuple[Any, ...], dtype[floating]], int], None]`

slepc4py.typing.PEPStoppingFunction

`slepc4py.typing.PEPStoppingFunction`

PEP stopping test callback.

alias of `Callable[[PEP, int, int, int, int], ConvergedReason]`

slepc4py.typing.PEPEigenvalueComparison

`slepc4py.typing.PEPEigenvalueComparison`

PEP eigenvalue comparison callback.

alias of `Callable[[float | complex, float | complex, float | complex, float | complex], int]`

slepc4py.typing.PEPMonitorFunction

`slepc4py.typing.PEPMonitorFunction`

PEP monitor callback.

alias of `Callable[[PEP, int, int, ndarray[tuple[Any, ...], dtype[floating | complexfloating]], ndarray[tuple[Any, ...], dtype[floating | complexfloating]], ndarray[tuple[Any, ...], dtype[floating]], int], None]`

slepc4py.typing.NEPStoppingFunction

`slepc4py.typing.NEPStoppingFunction`

NEP stopping test callback.

alias of `Callable[[NEP, int, int, int, int], ConvergedReason]`

slepc4py.typing.NEPEigenvalueComparison

`slepc4py.typing.NEPEigenvalueComparison`

NEP eigenvalue comparison callback.

alias of `Callable[[float | complex, float | complex, float | complex, float | complex], int]`

slepc4py.typing.NEPMonitorFunction

`slepc4py.typing.NEPMonitorFunction`

NEP monitor callback.

alias of `Callable[[NEP, int, int, ndarray[tuple[Any, ...], dtype[floating | complexfloating]], ndarray[tuple[Any, ...], dtype[floating | complexfloating]], ndarray[tuple[Any, ...], dtype[floating]], int], None]`

slepc4py.typing.NEPFunction

`slepc4py.typing.NEPFunction`

NEP Function callback.

alias of `Callable[[NEP, float | complex, Mat, Mat], None]`

slepc4py.typing.NEPJacobian

slepc4py.typing.NEPJacobian

NEP Jacobian callback.

alias of `Callable[[NEP, float | complex, Mat], None]`

slepc4py.typing.SVDStoppingFunction

slepc4py.typing.SVDStoppingFunction

SVD stopping test callback.

alias of `Callable[[SVD, int, int, int, int], ConvergedReason]`

slepc4py.typing.SVDMonitorFunction

slepc4py.typing.SVDMonitorFunction

SVD monitor callback.

alias of `Callable[[SVD, int, int, ndarray[tuple[Any, ...], dtype[floating]], ndarray[tuple[Any, ...], dtype[floating]], int], None]`

slepc4py.typing.MFNMonitorFunction

slepc4py.typing.MFNMonitorFunction

MFN monitor callback.

alias of `Callable[[MFN, int, float], None]`

slepc4py.typing.LMEMonitorFunction

slepc4py.typing.LMEMonitorFunction

LME monitor callback.

alias of `Callable[[LME, int, float], None]`

slepc4py.SLEPc

Scalable Library for Eigenvalue Problem Computations

Classes

<i>BV</i>	Basis Vectors.
<i>DS</i>	Direct Solver (or Dense System).
<i>EPS</i>	Eigenvalue Problem Solver.
<i>FN</i>	Mathematical Function.
<i>LME</i>	Linear Matrix Equation.
<i>MFN</i>	Matrix Function.
<i>NEP</i>	Nonlinear Eigenvalue Problem Solver.
<i>PEP</i>	Polynomial Eigenvalue Problem Solver.
<i>RG</i>	Region.
<i>ST</i>	Spectral Transformation.
<i>SVD</i>	Singular Value Decomposition Solver.
<i>Sys</i>	System utilities.

continues on next page

Table 4 – continued from previous page

<i>Util</i>	Other utilities such as the creation of structured matrices.
-------------	--

slepc4py.SLEPc.BV**class** slepc4py.SLEPc.BVBases: `Object`

Basis Vectors.

The *BV* package provides the concept of a block of vectors that represent the basis of a subspace. It is a convenient way of handling a collection of vectors that often operate together, rather than working with an array of `petsc4py.PETSc.Vec`.

Enumerations

<i>MatMultType</i>	BV mat-mult types.
<i>OrthogBlockType</i>	BV block-orthogonalization types.
<i>OrthogRefineType</i>	BV orthogonalization refinement types.
<i>OrthogType</i>	BV orthogonalization types.
<i>SVDMethod</i>	BV methods for computing the SVD.
<i>Type</i>	BV type.

slepc4py.SLEPc.BV.MatMultType**class** slepc4py.SLEPc.BV.MatMultTypeBases: `object`

BV mat-mult types.

- *VECS*: Perform a matrix-vector multiply per each column.
- *MAT*: Carry out a Mat-Mat product with a dense matrix.

See also`BVMatMultType`**Attributes Summary**

<i>MAT</i>	Constant MAT of type <code>int</code>
<i>VECS</i>	Constant VECS of type <code>int</code>

Attributes Documentation**MAT:** `int` = *MAT*Constant MAT of type `int`**VECS:** `int` = *VECS*Constant VECS of type `int``__init__()`

```
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.BV.OrthogBlockType

class slepc4py.SLEPc.BV.OrthogBlockType

Bases: `object`

BV block-orthogonalization types.

- *GS*: Gram-Schmidt, column by column.
- *CHOL*: Cholesky QR method.
- *TSQR*: Tall-skinny QR method.
- *TSQRCHOL*: Tall-skinny QR, but computing the triangular factor only.
- *SVQB*: SVQB method.

See also

[BVOrthogBlockType](#)

Attributes Summary

<i>CHOL</i>	Constant CHOL of type <code>int</code>
<i>GS</i>	Constant GS of type <code>int</code>
<i>SVQB</i>	Constant SVQB of type <code>int</code>
<i>TSQR</i>	Constant TSQR of type <code>int</code>
<i>TSQRCHOL</i>	Constant TSQRCHOL of type <code>int</code>

Attributes Documentation

CHOL: `int` = CHOL

Constant CHOL of type `int`

GS: `int` = GS

Constant GS of type `int`

SVQB: `int` = SVQB

Constant SVQB of type `int`

TSQR: `int` = TSQR

Constant TSQR of type `int`

TSQRCHOL: `int` = TSQRCHOL

Constant TSQRCHOL of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.BV.OrthogRefineType

class slepc4py.SLEPc.BV.OrthogRefineType

Bases: `object`

BV orthogonalization refinement types.

- *IFNEEDED*: Reorthogonalize if a criterion is satisfied.
- *NEVER*: Never reorthogonalize.
- *ALWAYS*: Always reorthogonalize.

See also

`BVOrthogRefineType`

Attributes Summary

<i>ALWAYS</i>	Constant <i>ALWAYS</i> of type <code>int</code>
<i>IFNEEDED</i>	Constant <i>IFNEEDED</i> of type <code>int</code>
<i>NEVER</i>	Constant <i>NEVER</i> of type <code>int</code>

Attributes Documentation

ALWAYS: `int` = *ALWAYS*

Constant *ALWAYS* of type `int`

IFNEEDED: `int` = *IFNEEDED*

Constant *IFNEEDED* of type `int`

NEVER: `int` = *NEVER*

Constant *NEVER* of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.BV.OrthogType

class slepc4py.SLEPc.BV.OrthogType

Bases: `object`

BV orthogonalization types.

- *CGS*: Classical Gram-Schmidt.
- *MGS*: Modified Gram-Schmidt.

See also

`BVOrthogType`

Attributes Summary

<i>CGS</i>	Constant CGS of type <code>int</code>
<i>MGS</i>	Constant MGS of type <code>int</code>

Attributes Documentation

CGS: `int` = CGS

Constant CGS of type `int`

MGS: `int` = MGS

Constant MGS of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.BV.SVDMethod

class slepc4py.SLEPc.BV.SVDMethod

Bases: `object`

BV methods for computing the SVD.

- [*REFINE*](#): Based on the SVD of the cross product matrix S^*S , with refinement.
- [*QR*](#): Based on the SVD of the triangular factor of `qr(S)`.
- [*QR_CAA*](#): Variant of QR intended for use in communication-avoiding. Arnoldi.

See also

[*BVSVDMethod*](#)

Attributes Summary

<i>QR</i>	Constant QR of type <code>int</code>
<i>QR_CAA</i>	Constant QR_CAA of type <code>int</code>
<i>REFINE</i>	Constant REFINE of type <code>int</code>

Attributes Documentation

QR: `int` = QR

Constant QR of type `int`

QR_CAA: `int` = QR_CAA

Constant QR_CAA of type `int`

REFINE: `int` = REFINE

Constant REFINE of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.BV.Type

class slepc4py.SLEPc.BV.Type

Bases: `object`

BV type.

- **MAT**: A *BV* stored as a dense **Mat**.
- **SVEC**: A *BV* stored as a single **Vec**.
- **VECS**: A *BV* stored as an array of independent **Vec**.
- **CONTIGUOUS**: A *BV* stored as an array of **Vec** sharing a contiguous array of scalars.
- **TENSOR**: A special *BV* represented in compact form as $V = (I \otimes U)S$.

See also

BVType

Attributes Summary

<i>CONTIGUOUS</i>	Object CONTIGUOUS of type <code>str</code>
<i>MAT</i>	Object MAT of type <code>str</code>
<i>SVEC</i>	Object SVEC of type <code>str</code>
<i>TENSOR</i>	Object TENSOR of type <code>str</code>
<i>VECS</i>	Object VECS of type <code>str</code>

Attributes Documentation

CONTIGUOUS: `str` = CONTIGUOUS

Object CONTIGUOUS of type `str`

MAT: `str` = MAT

Object MAT of type `str`

SVEC: `str` = SVEC

Object SVEC of type `str`

TENSOR: `str` = TENSOR

Object TENSOR of type `str`

VECS: `str` = VECS

Object VECS of type `str`

`__init__()`

classmethod `__new__(*args, **kwargs)`

Methods Summary

<i>appendOptionsPrefix</i> ([prefix])	Append to the prefix used for searching for all BV options in the database.
---------------------------------------	---

continues on next page

Table 12 – continued from previous page

<code>applyMatrix(x, y)</code>	Multiply a vector with the matrix associated to the bilinear form.
<code>copy([result])</code>	Copy a basis vector object into another one.
<code>copyColumn(j, i)</code>	Copy the values from one of the columns to another one.
<code>copyVec(j, v)</code>	Copy one of the columns of a basis vectors object into a vector.
<code>create([comm])</code>	Create the BV object.
<code>createFromMat(A)</code>	Create a basis vectors object from a dense matrix.
<code>createMat()</code>	Create a new dense matrix and copy the contents of the BV.
<code>createVec()</code>	Create a vector with the type and dimensions of the columns of the BV.
<code>destroy()</code>	Destroy the BV object.
<code>dot(Y)</code>	Compute the 'block-dot' product of two basis vectors objects.
<code>dotColumn(j)</code>	Dot products of a column against all the column vectors of a BV.
<code>dotVec(v)</code>	Dot products of a vector against all the column vectors of the BV.
<code>duplicate()</code>	Duplicate the BV object with the same type and dimensions.
<code>duplicateResize(m)</code>	Create a BV object of the same type and dimensions as an existing one.
<code>getActiveColumns()</code>	Get the current active dimensions.
<code>getArray([readonly])</code>	Return the array where the data is stored.
<code>getColumn(j)</code>	Get a vector with the entries of the column of the BV object.
<code>getDefiniteTolerance()</code>	Get the tolerance to be used when checking a definite inner product.
<code>getLeadingDimension()</code>	Get the leading dimension.
<code>getMat()</code>	Get a matrix of dense type that shares the memory of the BV object.
<code>getMatMultMethod()</code>	Get the method used for the <code>matMult()</code> operation.
<code>getMatrix()</code>	Get the matrix representation of the inner product.
<code>getNumConstraints()</code>	Get the number of constraints.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all BV options in the database.
<code>getOrthogonalization()</code>	Get the orthogonalization settings from the BV object.
<code>getRandomContext()</code>	Get the <code>petsc4py.PETSc.Random</code> object associated with the BV.
<code>getSizes()</code>	Get the local and global sizes, and the number of columns.
<code>getType()</code>	Get the BV type of this object.
<code>getVecType()</code>	Get the vector type used when creating vectors via <code>createVec()</code> .
<code>insertConstraints(C)</code>	Insert a set of vectors as constraints.
<code>insertVec(j, w)</code>	Insert a vector into the specified column.
<code>insertVecs(s, W[, orth])</code>	Insert a set of vectors into the specified columns.
<code>matMult(A[, Y])</code>	Compute the matrix-vector product for each column, $Y = AV$.

continues on next page

Table 12 – continued from previous page

<code>matMultColumn(A, j)</code>	Mat-vec product for a column, storing the result in the next column.
<code>matMultHermitianTranspose(A[, Y])</code>	Pre-multiplication with the conjugate transpose of a matrix.
<code>matMultHermitianTransposeColumn(A, j)</code>	Conjugate-transpose matrix-vector product for a specified column.
<code>matMultTranspose(A[, Y])</code>	Pre-multiplication with the transpose of a matrix.
<code>matMultTransposeColumn(A, j)</code>	Transpose matrix-vector product for a specified column.
<code>matProject(A, Y)</code>	Compute the projection of a matrix onto a subspace.
<code>mult(delta, gamma, X, Q)</code>	Compute $Y = \gamma Y + \delta X Q$.
<code>multColumn(delta, gamma, j, q)</code>	Compute $y = \gamma y + \delta X q$.
<code>multInPlace(Q, s, e)</code>	Update a set of vectors as $V(:, s : e - 1) = V Q(:, s : e - 1)$.
<code>multVec(delta, gamma, y, q)</code>	Compute $y = \gamma y + \delta X q$.
<code>norm([norm_type])</code>	Compute the matrix norm of the BV.
<code>normColumn(j[, norm_type])</code>	Compute the vector norm of a selected column.
<code>orthogonalize([R])</code>	Orthogonalize all columns (except leading ones) (QR decomposition).
<code>orthogonalizeColumn(j)</code>	Orthogonalize a column vector with respect to the previous ones.
<code>orthogonalizeVec(v)</code>	Orthogonalize a vector with respect to all active columns.
<code>orthonormalizeColumn(j[, replace])</code>	Orthonormalize a column vector with respect to the previous ones.
<code>resize(m[, copy])</code>	Change the number of columns.
<code>restoreColumn(j, v)</code>	Restore a column obtained with <code>getColumn()</code> .
<code>restoreMat(A)</code>	Restore the matrix obtained with <code>getMat()</code> .
<code>scale(alpha)</code>	Multiply the entries by a scalar value.
<code>scaleColumn(j, alpha)</code>	Scale a column of a BV.
<code>setActiveColumns(l, k)</code>	Set the columns that will be involved in operations.
<code>setDefiniteTolerance(deftol)</code>	Set the tolerance to be used when checking a definite inner product.
<code>setFromOptions()</code>	Set BV options from the options database.
<code>setLeadingDimension(ld)</code>	Set the leading dimension.
<code>setMatMultMethod(method)</code>	Set the method used for the <code>matMult()</code> operation.
<code>setMatrix(B[, indef])</code>	Set the bilinear form to be used for inner products.
<code>setNumConstraints(nc)</code>	Set the number of constraints.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all BV options in the database.
<code>setOrthogonalization([otype, refine, eta, block])</code>	Set the method used for the (block-)orthogonalization of vectors.
<code>setRandom()</code>	Set the active columns of the BV to random numbers.
<code>setRandomColumn(j)</code>	Set one column of the BV to random numbers.
<code>setRandomCond(condn)</code>	Set the columns of a BV to random numbers.
<code>setRandomContext(rnd)</code>	Set the <code>petsc4py.PETSc.Random</code> object associated with the BV.
<code>setRandomNormal()</code>	Set the active columns of the BV to normal random numbers.
<code>setRandomSign()</code>	Set the entries of a BV to values 1 or -1 with equal probability.

continues on next page

Table 12 – continued from previous page

<code>setSizes(sizes, m)</code>	Set the local and global sizes, and the number of columns.
<code>setSizesFromVec(w, m)</code>	Set the local and global sizes, and the number of columns.
<code>setType(bv_type)</code>	Set the type for the BV object.
<code>setVecType(vec_type)</code>	Set the vector type to be used when creating vectors via <code>createVec()</code> .
<code>view([viewer])</code>	Print the BV data structure.

Attributes Summary

<code>column_size</code>	Basis vectors column size.
<code>local_size</code>	Basis vectors local size.
<code>size</code>	Basis vectors global size.
<code>sizes</code>	Basis vectors local and global sizes, and the number of columns.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching for all BV options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all BV option requests.

Return type

None

See also

`setOptionsPrefix`, `getOptionsPrefix`, `BVAppendOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:618 <slepc4py/SLEPc/BV.pyx#L618>`

`applyMatrix(x, y)`

Multiply a vector with the matrix associated to the bilinear form.

Neighbor-wise collective.

Parameters

- **x** (*Vec*) – The input vector.
- **y** (*Vec*) – The result vector.

Return type

None

Notes

If the bilinear form has no associated matrix this function copies the vector.

See also

`setMatrix`, `BVApplyMatrix`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:863 <slepc4py/SLEPc/BV.pyx#L863>`

copy(*result=None*)

Copy a basis vector object into another one.

Logically collective.

Returns

The copy.

Return type

`BV`

Parameters

result (`BV` / `None`) – The copy.

Notes

Both objects must be distributed in the same manner; local copies are done. Only active columns (excluding the leading ones) are copied. In the destination BV, columns are overwritten starting from the leading ones. Constraints are not copied.

See also

`BVCopy`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:354 <slepc4py/SLEPc/BV.pyx#L354>`

copyColumn(*j, i*)

Copy the values from one of the columns to another one.

Logically collective.

Parameters

- **j** (`int`) – The index of the source column.
- **i** (`int`) – The index of the destination column.

Return type

`None`

See also

`copy`, `copyVec`, `BVCopyColumn`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1235 <slepc4py/SLEPc/BV.pyx#L1235>`

copyVec(*j*, *v*)

Copy one of the columns of a basis vectors object into a vector.

Logically collective.

Parameters

- **j** (*int*) – The column index to be copied.
- **v** (*Vec*) – A vector.

Return type

None

Notes

The BV and v must be distributed in the same manner; local copies are done.

See also

copy, *copyColumn*, *BVCopyVec*

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1210 <slepc4py/SLEPc/BV.pyx#L1210>](#)

create(*comm*=*None*)

Create the BV object.

Collective.

Parameters

comm (*Comm* | *None*) – MPI communicator; if not provided, it defaults to all processes.

Return type

Self

See also

createFromMat, *BVCreate*

:sources: [Source code at slepc4py/SLEPc/BV.pyx:225 <slepc4py/SLEPc/BV.pyx#L225>](#)

createFromMat(*A*)

Create a basis vectors object from a dense matrix.

Collective.

Parameters

A (*Mat*) – A dense tall-skinny matrix.

Return type

Self

Notes

The matrix values are copied to the *BV* data storage, memory is not shared.

The communicator of the *BV* object will be the same as *A*, and so will be the dimensions.

See also

[*create*](#), [*createMat*](#), [*BVCreateFromMat*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:247 <slepc4py/SLEPc/BV.pyx#L247>`](#)

createMat()

Create a new dense matrix and copy the contents of the BV.

Collective.

Returns

The new matrix.

Return type

`petsc4py.PETSc.Mat`

Notes

The matrix contains all columns of the *BV*, not just the active columns.

See also

[*createFromMat*](#), [*createVec*](#), [*getMat*](#), [*BVCreateMat*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:275 <slepc4py/SLEPc/BV.pyx#L275>`](#)

createVec()

Create a vector with the type and dimensions of the columns of the BV.

Collective.

Returns

New vector.

Return type

`petsc4py.PETSc.Vec`

See also

[*createMat*](#), [*setVecType*](#), [*BVCreateVec*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:1147 <slepc4py/SLEPc/BV.pyx#L1147>`](#)

destroy()

Destroy the BV object.

Collective.

See also

[*BVDestroy*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:211 <slepc4py/SLEPc/BV.pyx#L211>`](#)

Return type*Self***dot(Y)**

Compute the ‘block-dot’ product of two basis vectors objects.

Collective.

$$M = Y^* X \text{ (} m_{ij} = y_i^* x_j \text{) or } M = Y^* B X$$

Parameters

Y (BV) – Left basis vectors, can be the same as self, giving $M = X^* X$.

Returns

The resulting matrix.

Return type*petsc4py.PETSc.Mat***Notes**

This is the generalization of `Vec.dot()` for a collection of vectors, $M = Y^* X$. The result is a matrix M whose entry m_{ij} is equal to $y_i^* x_j$ (where y_i^* denotes the conjugate transpose of y_i).

X and Y can be the same object.

If a non-standard inner product has been specified with `setMatrix()`, then the result is $M = Y^* B X$. In this case, both X and Y must have the same associated matrix.

Only rows (resp. columns) of M starting from l_y (resp. l_x) are computed, where l_y (resp. l_x) is the number of leading columns of Y (resp. X).

See also

dotVec, dotColumn, setActiveColumns, setMatrix, BVDot

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1500 <slepc4py/SLEPc/BV.pyx#L1500>`

dotColumn(j)

Dot products of a column against all the column vectors of a BV.

Collective.

Parameters

j (int) – The index of the column.

Returns

The computed values.

Return type*ArrayScalar***Notes**

This operation is equivalent to `dotVec()` but it uses column j of the BV rather than taking a vector as an argument. The number of active columns of the BV is set to j before the computation, and restored afterwards. If the BV has leading columns specified, then these columns do not participate in the computation. Therefore, the length of the returned array will be j minus the number of leading columns.

See also*dot*, *dotVec*, *BVDotColumn***:sources:** `Source code at slepc4py/SLEPc/BV.pyx:1344 <slepc4py/SLEPc/BV.pyx#L1344>`**dotVec(*v*)**

Dot products of a vector against all the column vectors of the BV.

Collective.

Parameters***v*** (*Vec*) – A vector.**Returns**

The computed values.

Return type*ArrayScalar***Notes**

This is analogue to `Vec.mDot()`, but using *BV* to represent a collection of vectors X . The result is $m = X^*v$, so m_i is equal to x_j^*v . Note that here X is transposed as opposed to `dot()`.

If a non-standard inner product has been specified with `setMatrix()`, then the result is $m = X^*Bv$.

See also*dot*, *dotColumn*, *setMatrix*, *BVDotVec***:sources:** `Source code at slepc4py/SLEPc/BV.pyx:1304 <slepc4py/SLEPc/BV.pyx#L1304>`**duplicate()**

Duplicate the BV object with the same type and dimensions.

Collective.

Returns

The new object.

Return type*BV***Notes**

This function does not copy the entries, it just allocates the storage for the new *BV*. Use `copy()` to copy the content.

See also*duplicateResize*, *BVDuplicate***:sources:** `Source code at slepc4py/SLEPc/BV.pyx:299 <slepc4py/SLEPc/BV.pyx#L299>`

duplicateResize(*m*)

Create a BV object of the same type and dimensions as an existing one.

Collective.

Parameters

m (*int*) – The number of columns.

Returns

The new object.

Return type

BV

Notes

This is equivalent to a call to *duplicate()* followed by *resize()* with possibly different number of columns. The contents of this *BV* are not copied to the new one.

See also

duplicate, *resize*, *BVDuplicateResize*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:323 <slepc4py/SLEPc/BV.pyx#L323>`

getActiveColumns()

Get the current active dimensions.

Not collective.

Returns

- **l** (*int*) – The leading number of columns.
- **k** (*int*) – The active number of columns.

Return type

tuple[*int*, *int*]

See also

setActiveColumns, *BVGetActiveColumns*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:924 <slepc4py/SLEPc/BV.pyx#L924>`

getArray(*readonly=False*)

Return the array where the data is stored.

Not collective.

Parameters

readonly (*bool*) – Enable to obtain a read only array.

Returns

The array.

Return type

ArrayScalar

See also

[BVGetArray](#), [BVGetArrayRead](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:546](#) <[slepc4py/SLEPc/BV.pyx#L546](#)>

getColumn(*j*)

Get a vector with the entries of the column of the BV object.

Logically collective.

Parameters

j (*int*) – The index of the requested column.

Returns

The vector containing the *j*-th column.

Return type

`petsc4py.PETSc.Vec`

Notes

Modifying the returned vector will change the BV entries as well.

The returned vector must not be destroyed, [restoreColumn\(\)](#) must be called when it is no longer needed. At most, two columns can be fetched, that is, this function can only be called twice before the corresponding [restoreColumn\(\)](#) is invoked.

A negative index *j* selects the *i*-th constraint, where *i*=-*j*. Constraints should not be modified.

See also

[restoreColumn](#), [insertConstraints](#), [BVGetColumn](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1384](#) <[slepc4py/SLEPc/BV.pyx#L1384](#)>

getDefiniteTolerance()

Get the tolerance to be used when checking a definite inner product.

Not collective.

Returns

The tolerance.

Return type

`float`

See also

[setDefiniteTolerance](#), [BVGetDefiniteTolerance](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1285](#) <[slepc4py/SLEPc/BV.pyx#L1285](#)>

getLeadingDimension()

Get the leading dimension.

Not collective.

Returns

The leading dimension.

Return type

`int`

Notes

The returned value may be different in different processes.

The leading dimension must be used when accessing the internal array via `getArray()`.

See also

`setLeadingDimension`, `BVGetLeadingDimension`

:sources: [Source code at slepc4py/SLEPc/BV.pyx:520](#) <slepc4py/SLEPc/BV.pyx#L520>

getMat()

Get a matrix of dense type that shares the memory of the BV object.

Collective.

Returns

The matrix.

Return type

`petsc4py.PETSc.Mat`

Notes

The returned matrix contains only the active columns. If the content of the matrix is modified, these changes are also done in the BV object. The user must call `restoreMat()` when no longer needed.

This operation implies a call to `getArray()`, which may result in data copies.

See also

`restoreMat`, `createMat`, `getArray`, `BVGetMat`

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1447](#) <slepc4py/SLEPc/BV.pyx#L1447>

getMatMultMethod()

Get the method used for the `matMult()` operation.

Not collective.

Returns

The method for the `matMult()` operation.

Return type

`MatMultType`

See also

matMult, *setMatMultMethod*, *BVGetMatMultMethod*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:760 <slepc4py/SLEPc/BV.pyx#L760>`

getMatrix()

Get the matrix representation of the inner product.

Not collective.

Returns

- **B** (`petsc4py.PETSc.Mat`) – The matrix of the inner product.
- **indef** (`bool`) – Whether the matrix is indefinite.

Return type

`tuple[Mat, bool] | tuple[None, bool]`

See also

setMatrix, *BVGetMatrix*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:799 <slepc4py/SLEPc/BV.pyx#L799>`

getNumConstraints()

Get the number of constraints.

Not collective.

Returns

The number of constraints.

Return type

`int`

See also

insertConstraints, *setNumConstraints*, *BVGetNumConstraints*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1128 <slepc4py/SLEPc/BV.pyx#L1128>`

getOptionsPrefix()

Get the prefix used for searching for all BV options in the database.

Not collective.

Returns

The prefix string set for this BV object.

Return type

`str`

See also

[`setOptionsPrefix`](#), [`appendOptionsPrefix`](#), [`BVGetOptionsPrefix`](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:637 <slepc4py/SLEPc/BV.pyx#L637>`

getOrthogonalization()

Get the orthogonalization settings from the BV object.

Not collective.

Returns

- **type** (*OrthogType*) – The type of orthogonalization technique.
- **refine** (*OrthogRefineType*) – The type of refinement.
- **eta** (*float*) – Parameter for selective refinement (used when the refinement type is *IFNEEDED*).
- **block** (*OrthogBlockType*) – The type of block orthogonalization.

Return type

`tuple[OrthogType, OrthogRefineType, float, OrthogBlockType]`

See also

[`setOrthogonalization`](#), [`BVGetOrthogonalization`](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:675 <slepc4py/SLEPc/BV.pyx#L675>`

getRandomContext()

Get the `petsc4py.PETSc.Random` object associated with the BV.

Collective.

Returns

The random number generator context.

Return type

`petsc4py.PETSc.Random`

See also

[`setRandomContext`](#), [`BVGetRandomContext`](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2133 <slepc4py/SLEPc/BV.pyx#L2133>`

getSizes()

Get the local and global sizes, and the number of columns.

Not collective.

Returns

- **(n, N)** (*tuple of int*) – The local and global sizes.
- **m** (*int*) – The number of columns.

Return type
tuple[LayoutSizeSpec, int]

See also

[setSizes](#), [setSizesFromVec](#), [BVGetSizes](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:477](#) <[slepc4py/SLEPc/BV.pyx#L477](#)>

getType()

Get the BV type of this object.

Not collective.

Returns
The basis vectors type currently being used.

Return type
str

See also

[setType](#), [BVGetType](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:407](#) <[slepc4py/SLEPc/BV.pyx#L407](#)>

getVecType()

Get the vector type used when creating vectors via [createVec\(\)](#).

Not collective.

Returns
The vector type.

Return type
str

See also

[createVec](#), [setVecType](#), [BVGetVecType](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1191](#) <[slepc4py/SLEPc/BV.pyx#L1191](#)>

insertConstraints(C)

Insert a set of vectors as constraints.

Collective.

Parameters
`C (Vec | list[Vec])` – Set of vectors to be inserted as constraints.

Returns
Number of linearly independent constraints.

Return type
int

Notes

The constraints are relevant only during orthogonalization. Constraint vectors span a subspace that is deflated in every orthogonalization operation, so they are intended for removing those directions from the orthogonal basis computed in regular BV columns.

Constraints are not stored in regular columns, but in a special part of the storage. They can be accessed with negative indices in `getColumn()`.

This operation is DESTRUCTIVE, meaning that all data contained in the columns of the BV is lost. This is typically invoked just after creating the BV. Once a set of constraints has been set, it is not allowed to call this function again.

The vectors are copied one by one and then orthogonalized against the previous ones. If any of them is linearly dependent then it is discarded and not counted in the return value. The behavior is similar to `insertVecs()`.

See also

`insertVecs`, `setNumConstraints`, `BVInsertConstraints`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1051 <slepc4py/SLEPc/BV.pyx#L1051>`

`insertVec(j, w)`

Insert a vector into the specified column.

Logically collective.

Parameters

- **j** (`int`) – The column to be overwritten.
- **w** (`Vec`) – The vector to be copied.

Return type

`None`

See also

`insertVecs`, `BVInsertVec`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:988 <slepc4py/SLEPc/BV.pyx#L988>`

`insertVecs(s, W, orth=False)`

Insert a set of vectors into the specified columns.

Collective.

Parameters

- **s** (`int`) – The first column to be overwritten.
- **W** (`Vec` | `list[Vec]`) – Set of vectors to be copied.
- **orth** (`bool`) – Flag indicating if the vectors must be orthogonalized.

Returns

Number of linearly independent vectors.

Return type

`int`

Notes

Copies the contents of vectors W into the BV columns $s:s+n$, where n is the length of W . If `orth` is set, then the vectors are copied one by one and then orthogonalized against the previous one. If any of them is linearly dependent then it is discarded and the not counted in the return value.

See also

[`insertVec`](#), [`orthogonalizeColumn`](#), [`BVInsertVecs`](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1008 <slepc4py/SLEPc/BV.pyx#L1008>](#)

matMult(A , $Y=None$)

Compute the matrix-vector product for each column, $Y = AV$.

Neighbor-wise collective.

Parameters

- **A** ([`Mat`](#)) – The matrix.
- **Y** ([`BV`](#) / `None`)

Returns

The result.

Return type

[`BV`](#)

Notes

Only active columns (excluding the leading ones) are processed. If Y is `None` a new `BV` is created.

It is possible to choose whether the computation is done column by column or as a dense matrix-matrix product with [`setMatMultMethod\(\)`](#).

See also

[`copy`](#), [`matMultColumn`](#), [`matMultTranspose`](#), [`setMatMultMethod`](#), [`BVMatMult`](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1598 <slepc4py/SLEPc/BV.pyx#L1598>](#)

matMultColumn(A, j)

Mat-vec product for a column, storing the result in the next column.

Neighbor-wise collective.

$$v_{j+1} = Av_j.$$

Parameters

- **A** ([`Mat`](#)) – The matrix.
- **j** ([`int`](#)) – Index of column.

Return type

[`None`](#)

See also

`matMult`, `BVMatMultColumn`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1743 <slepc4py/SLEPc/BV.pyx#L1743>`

matMultHermitianTranspose(*A*, *Y=None*)

Pre-multiplication with the conjugate transpose of a matrix.

Neighbor-wise collective.

$Y = A^*V$.

Parameters

- **A** (*Mat*) – The matrix.
- **Y** (*BV* | *None*)

Returns

The result.

Return type

BV

Notes

Only active columns (excluding the leading ones) are processed. If *Y* is *None* a new *BV* is created.

See also

`matMult`, `matMultHermitianTransposeColumn`, `BVMatMultHermitianTranspose`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1695 <slepc4py/SLEPc/BV.pyx#L1695>`

matMultHermitianTransposeColumn(*A*, *j*)

Conjugate-transpose matrix-vector product for a specified column.

Neighbor-wise collective.

Store the result in the next column: $v_{j+1} = A^*v_j$.

Parameters

- **A** (*Mat*) – The matrix.
- **j** (*int*) – Index of column.

Return type

None

See also

`matMultColumn`, `BVMatMultHermitianTransposeColumn`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1787 <slepc4py/SLEPc/BV.pyx#L1787>`

matMultTranspose(*A*, *Y=None*)

Pre-multiplication with the transpose of a matrix.

Neighbor-wise collective.

$$Y = A^T V.$$

Parameters

- **A** (*Mat*) – The matrix.
- **Y** (*BV* / *None*)

Returns

The result.

Return type

BV

Notes

Only active columns (excluding the leading ones) are processed. If *Y* is *None* a new *BV* is created.

See also

matMult, *matMultTransposeColumn*, *BVMatMultTranspose*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1647 <slepc4py/SLEPc/BV.pyx#L1647>`

matMultTransposeColumn(*A*, *j*)

Transpose matrix-vector product for a specified column.

Neighbor-wise collective.

Store the result in the next column: $v_{j+1} = A^T v_j$.

Parameters

- **A** (*Mat*) – The matrix.
- **j** (*int*) – Index of column.

Return type

None

See also

matMultColumn, *BVMatMultTransposeColumn*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1765 <slepc4py/SLEPc/BV.pyx#L1765>`

matProject(*A*, *Y*)

Compute the projection of a matrix onto a subspace.

Collective.

$$M = Y^* A X$$

Parameters

- **A** (*Mat* / *None*) – Matrix to be projected.

- **Y (BV)** – Left basis vectors, can be the same as self, giving $M = X^*AX$.

Returns

Projection of the matrix A onto the subspace.

Return type

`petsc4py.PETSc.Mat`

Notes

If A is None, then it is assumed that the BV already contains AX .

This operation is similar to `dot()`, with important differences. The goal is to compute the matrix resulting from the orthogonal projection of A onto the subspace spanned by the columns of the BV, $M = X^*AX$, or the oblique projection onto the BV along the second one Y, $M = Y^*AX$.

A difference with respect to `dot()` is that the standard inner product is always used, regardless of a non-standard inner product being specified with `setMatrix()`.

See also

`dot`, `setActiveColumns`, `setMatrix`, `BVMatProject`

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1549 <slepc4py/SLEPc/BV.pyx#L1549>](#)

mult(delta, gamma, X, Q)

Compute $Y = \gamma Y + \delta XQ$.

Logically collective.

Parameters

- **delta** (**Scalar**) – Coefficient that multiplies X.
- **gamma** (**Scalar**) – Coefficient that multiplies self (Y).
- **X (BV)** – Input basis vectors.
- **Q (Mat | None)** – Input matrix, if not given the identity matrix is assumed.

Return type

None

Notes

X must be different from self (Y). The case X=Y can be addressed with `multInPlace()`.

See also

`multVec`, `multColumn`, `multInPlace`, `BVMult`

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1809 <slepc4py/SLEPc/BV.pyx#L1809>](#)

multColumn(delta, gamma, j, q)

Compute $y = \gamma y + \delta Xq$.

Logically collective.

Compute $y = \gamma y + \delta Xq$, where y is the j-th column.

Parameters

- **delta** ([Scalar](#)) – Coefficient that multiplies self (X).
- **gamma** ([Scalar](#)) – Coefficient that multiplies y .
- **j** ([int](#)) – The column index.
- **q** ([Sequence](#)[[Scalar](#)]) – Input coefficients.

Return type

[None](#)

See also

[mult](#), [multVec](#), [multInPlace](#), [BVMultColumn](#)

:sources:[`Source code at slepc4py/SLEPc/BV.pyx:1863 <slepc4py/SLEPc/BV.pyx#L1863>`](#)

multInPlace(Q, s, e)

Update a set of vectors as $V(:, s : e - 1) = VQ(:, s : e - 1)$.

Logically collective.

Parameters

- **Q** ([Mat](#)) – A sequential dense matrix.
- **s** ([int](#)) – First column to be overwritten.
- **e** ([int](#)) – Last column to be overwritten.

Return type

[None](#)

See also

[mult](#), [multVec](#), [BVMultInPlace](#)

:sources:[`Source code at slepc4py/SLEPc/BV.pyx:1840 <slepc4py/SLEPc/BV.pyx#L1840>`](#)

multVec(δ, γ, y, q)

Compute $y = \gamma y + \delta Xq$.

Logically collective.

Parameters

- **delta** ([Scalar](#)) – Coefficient that multiplies self (X).
- **gamma** ([Scalar](#)) – Coefficient that multiplies y .
- **y** ([Vec](#)) – Input/output vector.
- **q** ([Sequence](#)[[Scalar](#)]) – Input coefficients.

Return type

[None](#)

See also

`mult`, `multColumn`, `multInPlace`, `BVMultVec`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1898` `<slepc4py/SLEPc/BV.pyx#L1898>`

norm(*norm_type=None*)

Compute the matrix norm of the BV.

Collective.

Parameters

norm_type (*NormType* / *None*) – The norm type.

Returns

The norm.

Return type

`float`

Notes

All active columns (except the leading ones) are considered as a matrix. The allowed norms are `NORM_1`, `NORM_FROBENIUS`, and `NORM_INFINITY`.

This operation fails if a non-standard inner product has been specified with `setMatrix()`.

See also

`normColumn`, `setMatrix`, `BVNorm`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1967` `<slepc4py/SLEPc/BV.pyx#L1967>`

normColumn(*j*, *norm_type=None*)

Compute the vector norm of a selected column.

Collective.

Parameters

- **j** (*int*) – Index of column.
- **norm_type** (*NormType* / *None*) – The norm type.

Returns

The norm.

Return type

`float`

Notes

The norm of v_j is computed (`NORM_1`, `NORM_2`, or `NORM_INFINITY`).

If a non-standard inner product has been specified with `setMatrix()`, then the returned value is $\sqrt{v_j^* B v_j}$, where B is the inner product matrix (argument ‘norm_type’ is ignored).

See also

[*norm*](#), [*setMatrix*](#), [*BVNormColumn*](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1929 <slepc4py/SLEPc/BV.pyx#L1929>](#)

orthogonalize(*R=None*, ***kargs*)

Orthogonalize all columns (except leading ones) (QR decomposition).

Collective.

Parameters

- **R** (*Mat* | *None*) – A sequential dense matrix.
- **kargs** (*Any*)

Return type

None

Notes

The output satisfies $V_0 = VR$ (where V_0 represent the input V) and $V^*V = I$ (or $V^*BV = I$ if an inner product matrix B has been specified with [*setMatrix\(\)*](#)).

See also

[*orthogonalizeColumn*](#), [*setMatrix*](#), [*setOrthogonalization*](#), [*BVOrthogonalize*](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2263 <slepc4py/SLEPc/BV.pyx#L2263>](#)

orthogonalizeColumn(*j*)

Orthogonalize a column vector with respect to the previous ones.

Collective.

Parameters

- **j** (*int*) – Index of the column to be orthogonalized.

Returns

- **norm** (*float*) – The norm of the resulting vector.
- **lindep** (*bool*) – Flag indicating that refinement did not improve the quality of orthogonalization.

Return type

tuple[*float*, *bool*]

Notes

This function applies an orthogonal projector to project vector v_j onto the orthogonal complement of the span of the columns $V[0..j-1]$, where $V[.]$ are the vectors of the BV. The columns $V[0..j-1]$ are assumed to be mutually orthonormal.

This routine does not normalize the resulting vector.

See also

[*orthogonalizeVec*](#), [*setOrthogonalization*](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2189 <slepc4py/SLEPc/BV.pyx#L2189>`

orthogonalizeVec(*v*)

Orthogonalize a vector with respect to all active columns.

Collective.

Parameters

v (*Vec*) – Vector to be orthogonalized, modified on return.

Returns

- **norm** (*float*) – The norm of the resulting vector.
- **lindep** (*bool*) – Flag indicating that refinement did not improve the quality of orthogonalization.

Return type

`tuple[float, bool]`

Notes

This function applies an orthogonal projector to project vector *v* onto the orthogonal complement of the span of the columns of the BV.

This routine does not normalize the resulting vector.

See also

[*orthogonalizeColumn*](#), [*setOrthogonalization*](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2153 <slepc4py/SLEPc/BV.pyx#L2153>`

orthonormalizeColumn(*j*, *replace=False*)

Orthonormalize a column vector with respect to the previous ones.

Collective.

This is equivalent to a call to [*orthogonalizeColumn\(\)*](#) followed by a call to [*scaleColumn\(\)*](#) with the reciprocal of the norm.

Parameters

- **j** (*int*) – Index of the column to be orthonormalized.
- **replace** (*bool*) – Whether it is allowed to set the vector randomly.

Returns

- **norm** (*float*) – The norm of the resulting vector.
- **lindep** (*bool*) – Flag indicating that refinement did not improve the quality of orthogonalization.

Return type

`tuple[float, bool]`

See also

[`orthogonalizeColumn`](#), [`setOrthogonalization`](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2227](#) <slepc4py/SLEPc/BV.pyx#L2227>

resize(*m*, *copy*=*True*)

Change the number of columns.

Collective.

Parameters

- **m** (*int*) – The new number of columns.
- **copy** (*bool*) – A flag indicating whether current values should be kept.

Return type

None

Notes

Internal storage is reallocated. If *copy* is *True*, then the contents are copied to the leading part of the new space.

See also

[`setSize`](#), [`setSizeFromVec`](#), [`BVResize`](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2002](#) <slepc4py/SLEPc/BV.pyx#L2002>

restoreColumn(*j*, *v*)

Restore a column obtained with [`getColumn\(\)`](#).

Logically collective.

Parameters

- **j** (*int*) – The index of the requested column.
- **v** (*Vec*) – The vector obtained with [`getColumn\(\)`](#).

Return type

None

Notes

The arguments must match the corresponding call to [`getColumn\(\)`](#).

See also

[`getColumn`](#), [`BVRestoreColumn`](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1422](#) <slepc4py/SLEPc/BV.pyx#L1422>

restoreMat(A)

Restore the matrix obtained with *getMat()*.

Logically collective.

Parameters

A (*Mat*) – The matrix obtained with *getMat()*.

Return type

None

Notes

A call to this function must match a previous call of *getMat()*. The effect is that the contents of the matrix are copied back to the BV internal data structures.

See also

getMat, *BVRestoreMat*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1476 <slepc4py/SLEPc/BV.pyx#L1476>`

scale(alpha)

Multiply the entries by a scalar value.

Logically collective.

Parameters

alpha (*Scalar*) – scaling factor.

Return type

None

Notes

All active columns (except the leading ones) are scaled.

See also

scaleColumn, *setActiveColumns*, *BVScale*

:sources: `Source code at slepc4py/SLEPc/BV.pyx:966 <slepc4py/SLEPc/BV.pyx#L966>`

scaleColumn(j, alpha)

Scale a column of a BV.

Logically collective.

Parameters

- **j** (*int*) – column index to be scaled.
- **alpha** (*Scalar*) – scaling factor.

Return type

None

See also

`scale`, `BVScaleColumn`

:sources: ``Source code at slepc4py/SLEPc/BV.pyx:945 <slepc4py/SLEPc/BV.pyx#L945>``

setActiveColumns(*l*, *k*)

Set the columns that will be involved in operations.

Logically collective.

Parameters

- **l** (*int*) – The leading number of columns.
- **k** (*int*) – The active number of columns.

Return type

`None`

Notes

In operations such as `mult()` or `dot()`, only the first *k* columns are considered. This is useful when the BV is filled from left to right, so the last *m*–*k* columns do not have relevant information.

Also in operations such as `mult()` or `dot()`, the first *l* columns are normally not included in the computation.

In orthogonalization operations, the first *l* columns are treated differently, they participate in the orthogonalization but the computed coefficients are not stored.

Use `CURRENT` to leave any of the values unchanged. Use `DETERMINE` to set *l* to the minimum value (0) and *k* to the maximum (*m*).

See also

`getActiveColumns`, `setSize`, `BVSetActiveColumns`

:sources: ``Source code at slepc4py/SLEPc/BV.pyx:887 <slepc4py/SLEPc/BV.pyx#L887>``

setDefiniteTolerance(*deftol*)

Set the tolerance to be used when checking a definite inner product.

Logically collective.

Parameters

deftol (*float*) – The tolerance.

Return type

`None`

Notes

When using a non-standard inner product, see `setMatrix()`, the solver needs to compute $\sqrt{z^*Bz}$ for various vectors *z*. If the inner product has not been declared indefinite, the value z^*Bz must be positive, but due to rounding error a tiny value may become negative. A tolerance is used to detect this situation. Likewise, in complex arithmetic z^*Bz should be real, and we use the same tolerance to check whether a nonzero imaginary part can be considered negligible.

See also

[`setMatrix`](#), [`getDefiniteTolerance`](#), [`BVSetDefiniteTolerance`](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:1256 <slepc4py/SLEPc/BV.pyx#L1256>`

setFromOptions()

Set BV options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

See also

[`setOptionsPrefix`](#), [`BVSetFromOptions`](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:656 <slepc4py/SLEPc/BV.pyx#L656>`

Return type

`None`

setLeadingDimension(*ld*)

Set the leading dimension.

Not collective.

Parameters

ld (`int`) – The leading dimension.

Return type

`None`

Notes

This parameter is relevant for a BV of `BV.Type.MAT`.

See also

[`getLeadingDimension`](#), [`BVSetLeadingDimension`](#)

:sources: `Source code at slepc4py/SLEPc/BV.pyx:498 <slepc4py/SLEPc/BV.pyx#L498>`

setMatMultMethod(*method*)

Set the method used for the `matMult()` operation.

Logically collective.

Parameters

method (`MatMultType`) – The method for the `matMult()` operation.

Return type

`None`

See also

`matMult`, `getMatMultMethod`, `BVSetMatMultMethod`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:779` `<slepc4py/SLEPc/BV.pyx#L779>`

setMatrix(*B*, *indef*=*False*)

Set the bilinear form to be used for inner products.

Collective.

Parameters

- **B** (*Mat* / *None*) – The matrix of the inner product.
- **indef** (*bool*) – Whether the matrix is indefinite.

Return type

None

Notes

This is used to specify a non-standard inner product, whose matrix representation is given by *B*. Then, all inner products required during orthogonalization are computed as $(x, y)_B = y^* B x$ rather than the standard form $(x, y) = y^* x$.

Matrix *B* must be real symmetric (or complex Hermitian). A genuine inner product requires that *B* is also positive (semi-)definite. However, we also allow for an indefinite *B* (setting *indef*=*True*), in which case the orthogonalization uses an indefinite inner product.

This affects operations `dot()`, `norm()`, `orthogonalize()`, and variants.

Omitting *B* has the same effect as if the identity matrix was passed.

See also

`getMatrix`, `BVSetMatrix`

:sources: `Source code at slepc4py/SLEPc/BV.pyx:825` `<slepc4py/SLEPc/BV.pyx#L825>`

setNumConstraints(*nc*)

Set the number of constraints.

Logically collective.

Parameters

- **nc** (*int*) – The number of constraints.

Return type

None

Notes

This function sets the number of constraints to *nc* and marks all remaining columns as regular. Normal usage would be to call `insertConstraints()` instead.

If *nc* is smaller than the previously set value, then some of the constraints are discarded. In particular, using *nc*=0 removes all constraints preserving the content of regular columns.

See also

[insertConstraints](#), [getNumConstraints](#), [BVSetNumConstraints](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1101](#) <[slepc4py/SLEPc/BV.pyx#L1101](#)>

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all BV options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all BV option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

See also

[appendOptionsPrefix](#), [getOptionsPrefix](#), [BVGetOptionsPrefix](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:593](#) <[slepc4py/SLEPc/BV.pyx#L593](#)>

setOrthogonalization(*otype=None, refine=None, eta=None, block=None*)

Set the method used for the (block-)orthogonalization of vectors.

Logically collective.

Orthogonalization of vectors (classical or modified Gram-Schmidt with or without refinement), and for the block-orthogonalization (simultaneous orthogonalization of a set of vectors).

Parameters

- **otype** (*OrthogType* / *None*) – The type of orthogonalization technique.
- **refine** (*OrthogRefineType* / *None*) – The type of refinement.
- **eta** (*float* / *None*) – Parameter for selective refinement.
- **block** (*OrthogBlockType* / *None*) – The type of block orthogonalization.

Return type

None

Notes

The default settings work well for most problems.

The parameter *eta* should be a real value between 0 and 1 (or *DETERMINE*). The value of *eta* is used only when the refinement type is *IFNEEDED*.

When using several processes, *MGS* is likely to result in bad scalability.

If the method set for block orthogonalization is *GS*, then the computation is done column by column with the vector orthogonalization.

See also

[*getOrthogonalization*](#), [*BVSetOrthogonalization*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:704 <slepc4py/SLEPc/BV.pyx#L704>`](#)

setRandom()

Set the active columns of the BV to random numbers.

Logically collective.

Notes

All active columns (except the leading ones) are modified.

See also

[*setRandomContext*](#), [*setRandomColumn*](#), [*setRandomNormal*](#), [*BVSetRandom*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:2028 <slepc4py/SLEPc/BV.pyx#L2028>`](#)

Return type

[*None*](#)

setRandomColumn(*j*)

Set one column of the BV to random numbers.

Logically collective.

Parameters

j ([*int*](#)) – Column index to be set.

Return type

[*None*](#)

See also

[*setRandomContext*](#), [*setRandom*](#), [*setRandomNormal*](#), [*BVSetRandomColumn*](#)

:sources: [`Source code at slepc4py/SLEPc/BV.pyx:2076 <slepc4py/SLEPc/BV.pyx#L2076>`](#)

setRandomCond(*condn*)

Set the columns of a BV to random numbers.

Logically collective.

The generated matrix has a prescribed condition number.

Parameters

condn ([*float*](#)) – Condition number.

Return type

[*None*](#)

See also

[*setRandomContext*](#), [*setRandomSign*](#), [*setRandomNormal*](#), [*BVSetRandomCond*](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2094 <slepc4py/SLEPc/BV.pyx#L2094>](#)

setRandomContext(*rnd*)

Set the `petsc4py.PETSc.Random` object associated with the BV.

Collective.

To be used in operations that need random numbers.

Parameters

rnd (*Random*) – The random number generator context.

Return type

None

See also

[*getRandomContext*](#), [*setRandom*](#), [*setRandomColumn*](#), [*BVSetRandomContext*](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2114 <slepc4py/SLEPc/BV.pyx#L2114>](#)

setRandomNormal()

Set the active columns of the BV to normal random numbers.

Logically collective.

Notes

All active columns (except the leading ones) are modified.

See also

[*setRandomContext*](#), [*setRandom*](#), [*setRandomSign*](#), [*BVSetRandomNormal*](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2044 <slepc4py/SLEPc/BV.pyx#L2044>](#)

Return type

None

setRandomSign()

Set the entries of a BV to values 1 or -1 with equal probability.

Logically collective.

Notes

All active columns (except the leading ones) are modified.

See also

[setRandomContext](#), [setRandom](#), [setRandomNormal](#), [BVSetRandomSign](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:2060](#) <[slepc4py/SLEPc/BV.pyx#L2060](#)>

Return type

None

setSize(*sizes*, *m*)

Set the local and global sizes, and the number of columns.

Collective.

Parameters

- **sizes** ([LayoutSizeSpec](#)) – The global size *N* or a two-tuple (*n*, *N*) with the local and global sizes.
- **m** (*int*) – The number of columns.

Return type

None

Notes

Either *n* or *N* (but not both) can be [DETERMINE](#) or None to have it automatically set.

See also

[setSizeFromVec](#), [getSizes](#), [BVSetSizes](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:426](#) <[slepc4py/SLEPc/BV.pyx#L426](#)>

setSizeFromVec(*w*, *m*)

Set the local and global sizes, and the number of columns.

Collective.

Local and global sizes are specified indirectly by passing a template vector.

Parameters

- **w** ([Vec](#)) – The template vector.
- **m** (*int*) – The number of columns.

Return type

None

See also

[setSize](#), [getSizes](#), [BVSetSizesFromVec](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:454](#) <[slepc4py/SLEPc/BV.pyx#L454](#)>

setType(*bv_type*)

Set the type for the BV object.

Logically collective.

Parameters

bv_type (*Type* / *str*) – The basis vectors type to be used.

Return type

None

See also

[getType](#), [BVSetType](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:388 <slepc4py/SLEPc/BV.pyx#L388>](#)

setVecType(*vec_type*)

Set the vector type to be used when creating vectors via [createVec\(\)](#).

Collective.

Parameters

vec_type (*petsc4py.PETSc.Vec.Type* / *str*) – Vector type used when creating vectors with [createVec](#).

Return type

None

Notes

This is not needed if the BV object is set up with [setSizeFromVec\(\)](#), but may be required in the case of [setSize\(\)](#) if one wants to work with non-standard vectors.

See also

[createVec](#), [getVecType](#), [setSize](#), [setSizeFromVec](#), [BVSetVecType](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:1166 <slepc4py/SLEPc/BV.pyx#L1166>](#)

view(*viewer=None*)

Print the BV data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

[BVView](#)

:sources: [Source code at slepc4py/SLEPc/BV.pyx:192 <slepc4py/SLEPc/BV.pyx#L192>](#)

Attributes Documentation

`column_size`

Basis vectors column size.

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2305 <slepc4py/SLEPc/BV.pyx#L2305>`

`local_size`

Basis vectors local size.

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2300 <slepc4py/SLEPc/BV.pyx#L2300>`

`size`

Basis vectors global size.

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2295 <slepc4py/SLEPc/BV.pyx#L2295>`

`sizes`

Basis vectors local and global sizes, and the number of columns.

:sources: `Source code at slepc4py/SLEPc/BV.pyx:2290 <slepc4py/SLEPc/BV.pyx#L2290>`

`__init__()`

classmethod `__new__(*args, **kwargs)`

`slepc4py.SLEPc.DS`

class `slepc4py.SLEPc.DS`

Bases: `Object`

Direct Solver (or Dense System).

The *DS* package provides auxiliary routines that are internally used by the different slepc4py solvers. It is used to represent low-dimensional eigenproblems that must be solved within iterative solvers with direct methods. It can be seen as a structured wrapper to LAPACK functionality.

Enumerations

<i>MatType</i>	To refer to one of the matrices stored internally in DS.
<i>ParallelType</i>	Indicates the parallel mode that the direct solver will use.
<i>StateType</i>	DS state types.
<i>Type</i>	DS type.

`slepc4py.SLEPc.DS.MatType`

class `slepc4py.SLEPc.DS.MatType`

Bases: `object`

To refer to one of the matrices stored internally in DS.

- *A*: first matrix of eigenproblem/singular value problem.
- *B*: second matrix of a generalized eigenproblem.
- *C*: third matrix of a quadratic eigenproblem.
- *T*: tridiagonal matrix.

- *D*: diagonal matrix.
- *Q*: orthogonal matrix of (right) Schur vectors.
- *Z*: orthogonal matrix of left Schur vectors.
- *X*: right eigenvectors.
- *Y*: left eigenvectors.
- *U*: left singular vectors.
- *V*: right singular vectors.
- *W*: workspace matrix.

See also

DSMatType

Attributes Summary

<i>A</i>	Constant A of type int
<i>B</i>	Constant B of type int
<i>C</i>	Constant C of type int
<i>D</i>	Constant D of type int
<i>Q</i>	Constant Q of type int
<i>T</i>	Constant T of type int
<i>U</i>	Constant U of type int
<i>V</i>	Constant V of type int
<i>W</i>	Constant W of type int
<i>X</i>	Constant X of type int
<i>Y</i>	Constant Y of type int
<i>Z</i>	Constant Z of type int

Attributes Documentation

A: [int](#) = A

Constant A of type [int](#)

B: [int](#) = B

Constant B of type [int](#)

C: [int](#) = C

Constant C of type [int](#)

D: [int](#) = D

Constant D of type [int](#)

Q: [int](#) = Q

Constant Q of type [int](#)

T: [int](#) = T

Constant T of type [int](#)

```

U: int = U
    Constant U of type int
V: int = V
    Constant V of type int
W: int = W
    Constant W of type int
X: int = X
    Constant X of type int
Y: int = Y
    Constant Y of type int
Z: int = Z
    Constant Z of type int
__init__()
classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.DS.ParallelType

class slepc4py.SLEPc.DS.ParallelType

Bases: `object`

Indicates the parallel mode that the direct solver will use.

- *REDUNDANT*: Every process performs the computation redundantly.
- *SYNCHRONIZED*: The first process sends the result to the rest.
- *DISTRIBUTED*: Used in some cases to distribute the computation among processes.

See also

DSParallelType

Attributes Summary

<i>DISTRIBUTED</i>	Constant DISTRIBUTED of type int
<i>REDUNDANT</i>	Constant REDUNDANT of type int
<i>SYNCHRONIZED</i>	Constant SYNCHRONIZED of type int

Attributes Documentation

DISTRIBUTED: **int** = **DISTRIBUTED**

Constant DISTRIBUTED of type **int**

REDUNDANT: **int** = **REDUNDANT**

Constant REDUNDANT of type **int**

SYNCHRONIZED: **int** = **SYNCHRONIZED**

Constant SYNCHRONIZED of type **int**

```
__init__()
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.DS.StateType

class slepc4py.SLEPc.DS.StateType

Bases: `object`

DS state types.

- *RAW*: Not processed yet.
- *INTERMEDIATE*: Reduced to Hessenberg or tridiagonal form (or equivalent).
- *CONDENSED*: Reduced to Schur or diagonal form (or equivalent).
- *TRUNCATED*: Condensed form truncated to a smaller size.

See also

DSSStateType

Attributes Summary

<i>CONDENSED</i>	Constant CONDENSED of type <code>int</code>
<i>INTERMEDIATE</i>	Constant INTERMEDIATE of type <code>int</code>
<i>RAW</i>	Constant RAW of type <code>int</code>
<i>TRUNCATED</i>	Constant TRUNCATED of type <code>int</code>

Attributes Documentation

CONDENSED: `int` = CONDENSED

Constant CONDENSED of type `int`

INTERMEDIATE: `int` = INTERMEDIATE

Constant INTERMEDIATE of type `int`

RAW: `int` = RAW

Constant RAW of type `int`

TRUNCATED: `int` = TRUNCATED

Constant TRUNCATED of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.DS.Type

class slepc4py.SLEPc.DS.Type

Bases: `object`

DS type.

- *HEP*: Dense Hermitian Eigenvalue Problem.

- *NHEP*: Dense Non-Hermitian Eigenvalue Problem.
- *GHEP*: Dense Generalized Hermitian Eigenvalue Problem.
- *GHIEP*: Dense Generalized Hermitian Indefinite Eigenvalue Problem.
- *GNHEP*: Dense Generalized Non-Hermitian Eigenvalue Problem.
- *NHEPTS*: Dense Non-Hermitian Eigenvalue Problem (special variant intended for two-sided Krylov solvers).
- *SVD*: Dense Singular Value Decomposition.
- *HSVD*: Dense Hyperbolic Singular Value Decomposition.
- *GSVD*: Dense Generalized Singular Value Decomposition.
- *PEP*: Dense Polynomial Eigenvalue Problem.
- *NEP*: Dense Nonlinear Eigenvalue Problem.

See also

DSType

Attributes Summary

<i>GHEP</i>	Object GHEP of type <i>str</i>
<i>GHIEP</i>	Object GHIEP of type <i>str</i>
<i>GNHEP</i>	Object GNHEP of type <i>str</i>
<i>GSVD</i>	Object GSVD of type <i>str</i>
<i>HEP</i>	Object HEP of type <i>str</i>
<i>HSVD</i>	Object HSVD of type <i>str</i>
<i>NEP</i>	Object NEP of type <i>str</i>
<i>NHEP</i>	Object NHEP of type <i>str</i>
<i>NHEPTS</i>	Object NHEPTS of type <i>str</i>
<i>PEP</i>	Object PEP of type <i>str</i>
<i>SVD</i>	Object SVD of type <i>str</i>

Attributes Documentation

GHEP: *str* = *GHEP*

Object GHEP of type *str*

GHIEP: *str* = *GHIEP*

Object GHIEP of type *str*

GNHEP: *str* = *GNHEP*

Object GNHEP of type *str*

GSVD: *str* = *GSVD*

Object GSVD of type *str*

HEP: *str* = *HEP*

Object HEP of type *str*

HSVD: `str = HSVD`
Object HSVD of type `str`

NEP: `str = NEP`
Object NEP of type `str`

NHEP: `str = NHEP`
Object NHEP of type `str`

NHEPTS: `str = NHEPTS`
Object NHEPTS of type `str`

PEP: `str = PEP`
Object PEP of type `str`

SVD: `str = SVD`
Object SVD of type `str`

`__init__()`

`classmethod __new__(*args, **kwargs)`

Methods Summary

<code>allocate(ld)</code>	Allocate memory for internal storage or matrices in DS.
<code>appendOptionsPrefix([prefix])</code>	Append to the prefix used for searching for all DS options in the database.
<code>cond()</code>	Compute the inf-norm condition number of the first matrix.
<code>create([comm])</code>	Create the DS object.
<code>destroy()</code>	Destroy the DS object.
<code>duplicate()</code>	Duplicate the DS object with the same type and dimensions.
<code>getArray(matname)</code>	Return the array where the data is stored.
<code>getBlockSize()</code>	Get the block size.
<code>getCompact()</code>	Get the compact storage flag.
<code>getDimensions()</code>	Get the current dimensions.
<code>getExtraRow()</code>	Get the extra row flag.
<code>getGSVDDimensions()</code>	Get the number of columns and rows of a <i>DS</i> of type <i>GSVD</i> .
<code>getHSVDDimensions()</code>	Get the number of columns of a <i>DS</i> of type <i>HSVD</i> .
<code>getLeadingDimension()</code>	Get the leading dimension of the allocated matrices.
<code>getMat(matname)</code>	Get the requested matrix as a sequential dense Mat object.
<code>getMethod()</code>	Get the method currently used in the DS.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all DS options in the database.
<code>getPEPCoefficients()</code>	Get the polynomial basis coefficients of a <i>DS</i> of type <i>PEP</i> .
<code>getPEPDegree()</code>	Get the polynomial degree of a <i>DS</i> of type <i>PEP</i> .
<code>getParallel()</code>	Get the mode of operation in parallel runs.
<code>getRefined()</code>	Get the refined vectors flag.
<code>getSVDDimensions()</code>	Get the number of columns of a <i>DS</i> of type <i>SVD</i> .

continues on next page

Table 19 – continued from previous page

<code>getState()</code>	Get the current state.
<code>getType()</code>	Get the DS type of this object.
<code>reset()</code>	Reset the DS object.
<code>restoreMat(matname, mat)</code>	Restore the previously seized matrix.
<code>setBlockSize(bs)</code>	Set the block size.
<code>setCompact(comp)</code>	Set the compact flag for storage of matrices.
<code>setDimensions([n, l, k])</code>	Set the matrix sizes in the DS object.
<code>setExtraRow(ext)</code>	Set a flag to indicate that the matrix has one extra row.
<code>setFromOptions()</code>	Set DS options from the options database.
<code>setGSVDDimensions(m, p)</code>	Set the number of columns and rows of a <i>DS</i> of type <i>GSVD</i> .
<code>setHSVDDimensions(m)</code>	Set the number of columns of a <i>DS</i> of type <i>HSVD</i> .
<code>setIdentity(matname)</code>	Set the identity on the active part of a matrix.
<code>setMethod(meth)</code>	Set the method to be used to solve the problem.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all DS options in the database.
<code>setPEPCoefficients(pbc)</code>	Set the polynomial basis coefficients of a <i>DS</i> of type <i>PEP</i> .
<code>setPEPDegree(deg)</code>	Set the polynomial degree of a <i>DS</i> of type <i>PEP</i> .
<code>setParallel(pmode)</code>	Set the mode of operation in parallel runs.
<code>setRefined(ref)</code>	Set a flag to indicate that refined vectors must be computed.
<code>setSVDDimensions(m)</code>	Set the number of columns of a <i>DS</i> of type <i>SVD</i> .
<code>setState(state)</code>	Set the state of the DS object.
<code>setType(ds_type)</code>	Set the type for the DS object.
<code>solve()</code>	Solve the problem.
<code>truncate(n[, trim])</code>	Truncate the system represented in the DS object.
<code>updateExtraRow()</code>	Ensure that the extra row gets up-to-date after a call to <i>DS.solve()</i> .
<code>vectors([matname])</code>	Compute vectors associated to the dense system such as eigenvectors.
<code>view([viewer])</code>	Print the DS data structure.

Attributes Summary

<code>block_size</code>	The block size.
<code>compact</code>	Compact storage of matrices.
<code>extra_row</code>	If the matrix has one extra row.
<code>method</code>	The method to be used to solve the problem.
<code>parallel</code>	The mode of operation in parallel runs.
<code>refined</code>	If refined vectors must be computed.
<code>state</code>	The state of the DS object.

Methods Documentation

`allocate(ld)`

Allocate memory for internal storage or matrices in DS.

Logically collective.

Parameters

ld (*int*) – Leading dimension (maximum allowed dimension for the matrices, including the

extra row if present).

Return type

`None`

Notes

If the leading dimension is different from a previously set value, then all matrices are destroyed with `reset()`.

See also

`getLeadingDimension`, `setDimensions`, `setExtraRow`, `reset`, `DSAllocate`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:338 <slepc4py/SLEPc/DS.pyx#L338>`

appendOptionsPrefix(*prefix=None*)

Append to the prefix used for searching for all DS options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all DS option requests.

Return type

`None`

See also

`setOptionsPrefix`, `getOptionsPrefix`, `DSSetOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:256 <slepc4py/SLEPc/DS.pyx#L256>`

cond()

Compute the inf-norm condition number of the first matrix.

Logically collective.

Returns

Condition number.

Return type

`float`

See also

`DSCond`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:905 <slepc4py/SLEPc/DS.pyx#L905>`

create(*comm=None*)

Create the DS object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator; if not provided, it defaults to all processes.

Return type*Self***See also***duplicate*, *DSCreate*

:sources: [Source code at slepc4py/SLEPc/DS.pyx:172](#) <slepc4py/SLEPc/DS.pyx#L172>

destroy()

Destroy the DS object.

Collective.

See also*DSDestroy*

:sources: [Source code at slepc4py/SLEPc/DS.pyx:146](#) <slepc4py/SLEPc/DS.pyx#L146>

Return type*Self***duplicate()**

Duplicate the DS object with the same type and dimensions.

Collective.

Returns

The new object.

Return type*DS***Notes**

This method does not copy the matrices, and the new object does not even have internal arrays allocated. Use *allocate()* to use the new *DS*.

See also*create*, *allocate*, *DSDuplicate*

:sources: [Source code at slepc4py/SLEPc/DS.pyx:311](#) <slepc4py/SLEPc/DS.pyx#L311>

getArray(matname)

Return the array where the data is stored.

Not collective.

Parameters

matname (*MatType*) – The selected matrix.

Returns

The array.

Return type
ArrayScalar

See also

DSGetArray

:sources: `Source code at slepc4py/SLEPc/DS.pyx:845 <slepc4py/SLEPc/DS.pyx#L845>`

getBlockSize()

Get the block size.

Not collective.

Returns
The block size.

Return type
int

See also

setBlockSize, DSGetBlockSize

:sources: `Source code at slepc4py/SLEPc/DS.pyx:555 <slepc4py/SLEPc/DS.pyx#L555>`

getCompact()

Get the compact storage flag.

Not collective.

Returns
The flag.

Return type
bool

See also

setCompact, DSGetCompact

:sources: `Source code at slepc4py/SLEPc/DS.pyx:639 <slepc4py/SLEPc/DS.pyx#L639>`

getDimensions()

Get the current dimensions.

Not collective.

- Returns**
- **n** (*int*) – The new size.
 - **l** (*int*) – Number of locked (inactive) leading columns.
 - **k** (*int*) – Intermediate dimension (e.g., position of arrow).
 - **t** (*int*) – Truncated length.

Return type

tuple[int, int, int, int]

Notes

The `t` value makes sense only if `truncate()` has been called. Otherwise it is equal to `n`.

See also

`setDimensions`, `truncate`, `getLeadingDimension`, `DSGetDimensions`

`:sources:` [Source code at slepc4py/SLEPc/DS.pyx:504](#) <slepc4py/SLEPc/DS.pyx#L504>

getExtraRow()

Get the extra row flag.

Not collective.

Returns

The flag.

Return type

bool

See also

`setExtraRow`, `DSGetExtraRow`

`:sources:` [Source code at slepc4py/SLEPc/DS.pyx:686](#) <slepc4py/SLEPc/DS.pyx#L686>

getGSVDDimensions()

Get the number of columns and rows of a *DS* of type *GSVD*.

Not collective.

Returns

- `m` (int) – The number of columns.
- `p` (int) – The number of rows for the second matrix.

Return type

tuple[int, int]

See also

`setGSVDDimensions`, `DSGSVDGetDimensions`

`:sources:` [Source code at slepc4py/SLEPc/DS.pyx:1082](#) <slepc4py/SLEPc/DS.pyx#L1082>

getHSVDDimensions()

Get the number of columns of a *DS* of type *HSVD*.

Not collective.

Returns

The number of columns.

Return type
`int`

See also

`setHSVDDimensions`, `DSHSVDGetDimensions`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1037 <slepc4py/SLEPc/DS.pyx#L1037>`

getLeadingDimension()

Get the leading dimension of the allocated matrices.

Not collective.

Returns

Leading dimension (maximum allowed dimension for the matrices).

Return type

`int`

See also

`allocate`, `setDimensions`, `DSGetLeadingDimension`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:362 <slepc4py/SLEPc/DS.pyx#L362>`

getMat(matname)

Get the requested matrix as a sequential dense Mat object.

Not collective.

Parameters

matname (`MatType`) – The requested matrix.

Returns

The matrix.

Return type

`petsc4py.PETSc.Mat`

Notes

The returned matrix has sizes equal to the current *DS* dimensions (see `setDimensions()`), and contains the values that would be obtained with `getArray()`. If the *DS* was truncated, then the number of rows is equal to the dimension prior to truncation, see `truncate()`.

When no longer needed the user must call `restoreMat()`.

See also

`restoreMat`, `setDimensions`, `getArray`, `truncate`, `DSGetMat`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:789 <slepc4py/SLEPc/DS.pyx#L789>`

getMethod()

Get the method currently used in the DS.

Not collective.

Returns

Identifier of the method.

Return type

`int`

See also

`setMethod`, `DSGetMethod`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:592 <slepc4py/SLEPc/DS.pyx#L592>`

getOptionsPrefix()

Get the prefix used for searching for all DS options in the database.

Not collective.

Returns

The prefix string set for this DS object.

Return type

`str`

See also

`appendOptionsPrefix`, `setOptionsPrefix`, `DSSetOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:275 <slepc4py/SLEPc/DS.pyx#L275>`

getPEPCoefficients()

Get the polynomial basis coefficients of a *DS* of type *PEP*.

Not collective.

Returns

Coefficients.

Return type

`ArrayReal`

See also

`setPEPCoefficients`, `DSPEPGetCoefficients`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1176 <slepc4py/SLEPc/DS.pyx#L1176>`

getPEPDegree()

Get the polynomial degree of a *DS* of type *PEP*.

Not collective.

Returns

The polynomial degree.

Return type

`int`

See also

`setPEPDegree`, `DSPEPGetDegree`

:sources: [Source code at slepc4py/SLEPc/DS.pyx:1122](#) <slepc4py/SLEPc/DS.pyx#L1122>

getParallel()

Get the mode of operation in parallel runs.

Not collective.

Returns

The parallel mode.

Return type

`ParallelType`

See also

`setParallel`, `DSGetParallel`

:sources: [Source code at slepc4py/SLEPc/DS.pyx:450](#) <slepc4py/SLEPc/DS.pyx#L450>

getRefined()

Get the refined vectors flag.

Not collective.

Returns

The flag.

Return type

`bool`

See also

`setRefined`, `DSGetRefined`

:sources: [Source code at slepc4py/SLEPc/DS.pyx:734](#) <slepc4py/SLEPc/DS.pyx#L734>

getSVDDimensions()

Get the number of columns of a *DS* of type *SVD*.

Not collective.

Returns

The number of columns.

Return type

`int`

See also

[*setSVDDimensions*](#), [*DSSVDGetDimensions*](#)

:sources: [`Source code at slepc4py/SLEPc/DS.pyx:995 <slepc4py/SLEPc/DS.pyx#L995>`](#)

getState()

Get the current state.

Not collective.

Returns

The current state.

Return type

[*StateType*](#)

See also

[*setState*](#), [*DSGetState*](#)

:sources: [`Source code at slepc4py/SLEPc/DS.pyx:413 <slepc4py/SLEPc/DS.pyx#L413>`](#)

getType()

Get the DS type of this object.

Not collective.

Returns

The direct solver type currently being used.

Return type

[*str*](#)

See also

[*setType*](#), [*DSGetType*](#)

:sources: [`Source code at slepc4py/SLEPc/DS.pyx:212 <slepc4py/SLEPc/DS.pyx#L212>`](#)

reset()

Reset the DS object.

Collective.

See also

[*allocate*](#), [*DSReset*](#)

:sources: [`Source code at slepc4py/SLEPc/DS.pyx:160 <slepc4py/SLEPc/DS.pyx#L160>`](#)

Return type

[*None*](#)

restoreMat(*matname*, *mat*)

Restore the previously seized matrix.

Not collective.

Parameters

- **matname** (*MatType*) – The selected matrix.
- **mat** (*petsc4py.PETSc.Mat*) – The matrix previously obtained with *getMat()*.

Return type

None

See also

getMat, *DSRestoreMat*

:sources: [Source code at slepc4py/SLEPc/DS.pyx:824 <slepc4py/SLEPc/DS.pyx#L824>](#)

setBlockSize(*bs*)

Set the block size.

Logically collective.

Parameters

bs (*int*) – The block size.

Return type

None

See also

getBlockSize, *DSSetBlockSize*

:sources: [Source code at slepc4py/SLEPc/DS.pyx:537 <slepc4py/SLEPc/DS.pyx#L537>](#)

setCompact(*comp*)

Set the compact flag for storage of matrices.

Logically collective.

Parameters

comp (*bool*) – True means compact storage.

Return type

None

Notes

Compact storage is used in some *DS* types such as *DS.Type.HEP* when the matrix is tridiagonal. This flag can be used to indicate whether the user provides the matrix entries via the compact form (the tridiagonal *DS.MatType.T*) or the non-compact one (*DS.MatType.A*).

The default is False.

See also

[`getCompact`](#), [`DSSetCompact`](#)

:sources: [Source code at slepc4py/SLEPc/DS.pyx:611 <slepc4py/SLEPc/DS.pyx#L611>](#)

setDimensions(*n=None, l=None, k=None*)

Set the matrix sizes in the DS object.

Logically collective.

Parameters

- **n** (*int* / *None*) – The new size.
- **l** (*int* / *None*) – Number of locked (inactive) leading columns.
- **k** (*int* / *None*) – Intermediate dimension (e.g., position of arrow).

Return type

None

Notes

The internal arrays are not reallocated.

Some *DS* types have additional dimensions, e.g., the number of columns in *DS.Type.SVD*. For these, you should call a specific interface function.

See also

[`getDimensions`](#), [`allocate`](#), [`DSSetDimensions`](#)

:sources: [Source code at slepc4py/SLEPc/DS.pyx:469 <slepc4py/SLEPc/DS.pyx#L469>](#)

setExtraRow(*ext*)

Set a flag to indicate that the matrix has one extra row.

Logically collective.

Parameters

- **ext** (*bool*) – True if the matrix has extra row.

Return type

None

Notes

In Krylov methods it is useful that the matrix representing the direct solver has one extra row, i.e., has $(n + 1)$ rows and $(n + 1)$ columns. If this flag is activated, all transformations applied to the right of the matrix also affect this additional row. In that case, $(n + 1)$ must be less or equal than the leading dimension.

The default is `False`.

See also

[`getExtraRow`](#), [`solve`](#), [`allocate`](#), [`DSSetExtraRow`](#)

:sources: `Source code at slepc4py/SLEPc/DS.pyx:658 <slepc4py/SLEPc/DS.pyx#L658>`

setFromOptions()

Set DS options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

See also

[`setOptionsPrefix`](#), [`DSSetFromOptions`](#)

:sources: `Source code at slepc4py/SLEPc/DS.pyx:294 <slepc4py/SLEPc/DS.pyx#L294>`

Return type

`None`

setGSVDDimensions(*m*, *p*)

Set the number of columns and rows of a *DS* of type *GSVD*.

Logically collective.

Parameters

- **m** (`int`) – The number of columns.
- **p** (`int`) – The number of rows for the second matrix.

Return type

`None`

Notes

This call is complementary to [`setDimensions\(\)`](#), to provide dimensions that are specific to this *DS*. *Type*.

See also

[`setDimensions`](#), [`getGSVDDimensions`](#), [`DSGSVDSetDimensions`](#)

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1056 <slepc4py/SLEPc/DS.pyx#L1056>`

setHSVDDimensions(*m*)

Set the number of columns of a *DS* of type *HSVD*.

Logically collective.

Parameters

- **m** (`int`) – The number of columns.

Return type

`None`

Notes

This call is complementary to `setDimensions()`, to provide a dimension that is specific to this *DS.Type*.

See also

`setDimensions`, `getHSVDDimensions`, `DSHSVDSetDimensions`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1014 <slepc4py/SLEPc/DS.pyx#L1014>`

`setIdentity(matname)`

Set the identity on the active part of a matrix.

Logically collective.

Parameters

matname (*MatType*) – The matrix to be changed.

Return type

None

See also

`DSSetIdentity`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:885 <slepc4py/SLEPc/DS.pyx#L885>`

`setMethod(meth)`

Set the method to be used to solve the problem.

Logically collective.

Parameters

meth (*int*) – An index identifying the method.

Return type

None

See also

`getMethod`, `DSSetMethod`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:574 <slepc4py/SLEPc/DS.pyx#L574>`

`setOptionsPrefix(prefix=None)`

Set the prefix used for searching for all DS options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all DS option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

See also

[appendOptionsPrefix](#), [getOptionsPrefix](#), [DSSetOptionsPrefix](#)

:sources: `Source code at slepc4py/SLEPc/DS.pyx:231 <slepc4py/SLEPc/DS.pyx#L231>`

setPEPCoefficients(*pbc*)

Set the polynomial basis coefficients of a *DS* of type *PEP*.

Logically collective.

Parameters

pbc (*Sequence*[*float*]) – Coefficients.

Return type

None

Notes

This function is required only in the case of a polynomial specified in a non-monomial basis, to provide the coefficients that will be used during the linearization, multiplying the identity blocks on the three main diagonal blocks. Depending on the polynomial basis (Chebyshev, Legendre, ...) the coefficients must be different.

There must be a total of $3(d+1)$ coefficients, where d is the degree of the polynomial. The coefficients are arranged in three groups, a_i, b_i, c_i , according to the definition of the three-term recurrence. In the case of the monomial basis, $a_i = 1$ and $b_i = c_i = 0$, in which case it is not necessary to invoke this function.

See also

[getPEPCoefficients](#), [DSPEPSetCoefficients](#)

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1141 <slepc4py/SLEPc/DS.pyx#L1141>`

setPEPDegree(*deg*)

Set the polynomial degree of a *DS* of type *PEP*.

Logically collective.

Parameters

deg (*int*) – The polynomial degree.

Return type

None

See also

[getPEPDegree](#), [DSPEPSetDegree](#)

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1104 <slepc4py/SLEPc/DS.pyx#L1104>`

setParallel(*pmode*)

Set the mode of operation in parallel runs.

Logically collective.

Parameters

pmode (`ParallelType`) – The parallel mode.

Return type

None

See also

`getParallel`, `DSSetParallel`

:sources: [Source code at slepc4py/SLEPc/DS.pyx:432 <slepc4py/SLEPc/DS.pyx#L432>](#)

setRefined(*ref*)

Set a flag to indicate that refined vectors must be computed.

Logically collective.

Parameters

ref (`bool`) – True if refined vectors must be used.

Return type

None

Notes

Normally the vectors returned in `DS.MatType.X` are eigenvectors of the projected matrix. With this flag activated, `vectors()` will return the right singular vector of the smallest singular value of matrix $\hat{A} - \eta I$, where \hat{A} is the extended matrix (with extra row) and η is the Ritz value. This is used in the refined Ritz approximation.

The default is False.

See also

`getRefined`, `vectors`, `setExtraRow`, `DSSetRefined`

:sources: [Source code at slepc4py/SLEPc/DS.pyx:705 <slepc4py/SLEPc/DS.pyx#L705>](#)

setSVDDimensions(*m*)

Set the number of columns of a `DS` of type `SVD`.

Logically collective.

Parameters

m (`int`) – The number of columns.

Return type

None

Notes

This call is complementary to `setDimensions()`, to provide a dimension that is specific to this *DS.Type*.

See also

`setDimensions`, `getSVDDimensions`, `DSSVDSetDimensions`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:972 <slepc4py/SLEPc/DS.pyx#L972>`

setState(*state*)

Set the state of the DS object.

Logically collective.

Parameters

state (*StateType*) – The new state.

Return type

None

Notes

The state indicates that the dense system is in an initial state (raw), in an intermediate state (such as tridiagonal, Hessenberg or Hessenberg-triangular), in a condensed state (such as diagonal, Schur or generalized Schur), or in a truncated state.

The state is automatically changed in functions such as `solve()` or `truncate()`. This function is normally used to return to the raw state when the condensed structure is destroyed, or to indicate that `solve()` must start with a problem that already has an intermediate form.

See also

`getState`, `solve`, `truncate`, `DSSetState`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:381 <slepc4py/SLEPc/DS.pyx#L381>`

setType(*ds_type*)

Set the type for the DS object.

Logically collective.

Parameters

ds_type (*Type* / *str*) – The direct solver type to be used.

Return type

None

See also

`getType`, `DSSetType`

:sources: `Source code at slepc4py/SLEPc/DS.pyx:193 <slepc4py/SLEPc/DS.pyx#L193>`

solve()

Solve the problem.

Logically collective.

Returns

Eigenvalues or singular values.

Return type

ArrayScalar

See also

DSSolve

:sources: [Source code at slepc4py/SLEPc/DS.pyx:924 <slepc4py/SLEPc/DS.pyx#L924>](#)

truncate(*n*, *trim*=False)

Truncate the system represented in the DS object.

Logically collective.

Parameters

- **n** (*int*) – The new size.
- **trim** (*bool*) – A flag to indicate if the factorization must be trimmed.

Return type

None

See also

setDimensions, *setExtraRow*, *DSTruncate*

:sources: [Source code at slepc4py/SLEPc/DS.pyx:753 <slepc4py/SLEPc/DS.pyx#L753>](#)

updateExtraRow()

Ensure that the extra row gets up-to-date after a call to *DS.solve()*.

Logically collective.

Perform all necessary operations so that the extra row gets up-to-date after a call to *DS.solve()*.

See also

DSUpdateExtraRow

:sources: [Source code at slepc4py/SLEPc/DS.pyx:774 <slepc4py/SLEPc/DS.pyx#L774>](#)

Return type

None

vectors(*matname*=X)

Compute vectors associated to the dense system such as eigenvectors.

Logically collective.

Parameters

matname – The matrix, used to indicate which vectors are required.

Return type

None

See also

DSVectors

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:952 <slepc4py/SLEPc/DS.pyx#L952>``

view(viewer=None)

Print the DS data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

DSView

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:127 <slepc4py/SLEPc/DS.pyx#L127>``

Attributes Documentation

block_size

The block size.

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:1218 <slepc4py/SLEPc/DS.pyx#L1218>``

compact

Compact storage of matrices.

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:1232 <slepc4py/SLEPc/DS.pyx#L1232>``

extra_row

If the matrix has one extra row.

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:1239 <slepc4py/SLEPc/DS.pyx#L1239>``

method

The method to be used to solve the problem.

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:1225 <slepc4py/SLEPc/DS.pyx#L1225>``

parallel

The mode of operation in parallel runs.

:sources: ``Source code at slepc4py/SLEPc/DS.pyx:1211 <slepc4py/SLEPc/DS.pyx#L1211>``

refined

If refined vectors must be computed.

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1246 <slepc4py/SLEPc/DS.pyx#L1246>`

state

The state of the DS object.

:sources: `Source code at slepc4py/SLEPc/DS.pyx:1204 <slepc4py/SLEPc/DS.pyx#L1204>`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.EPS

class slepc4py.SLEPc.EPS

Bases: `Object`

Eigenvalue Problem Solver.

The Eigenvalue Problem Solver (*EPS*) is the object provided by slepc4py for specifying a linear eigenvalue problem, either in standard or generalized form. It provides uniform and efficient access to all of the linear eigensolvers included in the package.

Enumerations

<i>Balance</i>	EPS type of balancing used for non-Hermitian problems.
<i>CISSExtraction</i>	EPS CISS extraction technique.
<i>CISSQuadRule</i>	EPS CISS quadrature rule.
<i>Conv</i>	EPS convergence test.
<i>ConvergedReason</i>	EPS convergence reasons.
<i>ErrorType</i>	EPS error type to assess accuracy of computed solutions.
<i>Extraction</i>	EPS extraction technique.
<i>KrylovSchurBSEType</i>	EPS Krylov-Schur method for BSE problems.
<i>LanczosReorthogType</i>	EPS Lanczos reorthogonalization type.
<i>PowerShiftType</i>	EPS Power shift type.
<i>ProblemType</i>	EPS problem type.
<i>Stop</i>	EPS stopping test.
<i>Type</i>	EPS type.
<i>Which</i>	EPS desired part of spectrum.

slepc4py.SLEPc.EPS.Balance

class slepc4py.SLEPc.EPS.Balance

Bases: `object`

EPS type of balancing used for non-Hermitian problems.

- *NONE*: None.
- *ONESIDE*: One-sided balancing.
- *TWOSIDE*: Two-sided balancing.
- *USER*: User-provided balancing matrices.

See also
EPSPBalance

Attributes Summary

<i>NONE</i>	Constant NONE of type <code>int</code>
<i>ONESIDE</i>	Constant ONESIDE of type <code>int</code>
<i>TWOSIDE</i>	Constant TWOSIDE of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

NONE: `int` = NONE

Constant NONE of type `int`

ONESIDE: `int` = ONESIDE

Constant ONESIDE of type `int`

TWOSIDE: `int` = TWOSIDE

Constant TWOSIDE of type `int`

USER: `int` = USER

Constant USER of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

`slepc4py.SLEPc.EPS.CISSExtraction`

class `slepc4py.SLEPc.EPS.CISSExtraction`

Bases: `object`

EPS CISS extraction technique.

- *RITZ*: Ritz extraction.
- *HANKEL*: Extraction via Hankel eigenproblem.

See also
EPSCISSExtraction

Attributes Summary

<i>HANKEL</i>	Constant HANKEL of type <code>int</code>
<i>RITZ</i>	Constant RITZ of type <code>int</code>

Attributes Documentation

HANKEL: `int` = HANKEL

Constant HANKEL of type `int`

RITZ: `int` = RITZ

Constant RITZ of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.EPS.CISSQuadRule

class `slepc4py.SLEPc.EPS.CISSQuadRule`

Bases: `object`

EPS CISS quadrature rule.

- *TRAPEZOIDAL*: Trapezoidal rule.
- *CHEBYSHEV*: Chebyshev points.

See also

`EPSCISSQuadRule`

Attributes Summary

CHEBYSHEV

Constant CHEBYSHEV of type `int`

TRAPEZOIDAL

Constant TRAPEZOIDAL of type `int`

Attributes Documentation

CHEBYSHEV: `int` = CHEBYSHEV

Constant CHEBYSHEV of type `int`

TRAPEZOIDAL: `int` = TRAPEZOIDAL

Constant TRAPEZOIDAL of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.EPS.Conv

class `slepc4py.SLEPc.EPS.Conv`

Bases: `object`

EPS convergence test.

- *ABS*: Absolute convergence test.
- *REL*: Convergence test relative to the eigenvalue.
- *NORM*: Convergence test relative to the matrix norms.
- *USER*: User-defined convergence test.

See also
EPSCConv

Attributes Summary

<i>ABS</i>	Constant ABS of type <code>int</code>
<i>NORM</i>	Constant NORM of type <code>int</code>
<i>REL</i>	Constant REL of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

ABS: `int` = ABS

Constant ABS of type `int`

NORM: `int` = NORM

Constant NORM of type `int`

REL: `int` = REL

Constant REL of type `int`

USER: `int` = USER

Constant USER of type `int`

`__init__()`

classmethod `__new__`(*args, **kwargs)

slepc4py.SLEPc.EPS.ConvergedReason

class slepc4py.SLEPc.EPS.ConvergedReason

Bases: `object`

EPS convergence reasons.

- *CONVERGED_TOL*: All eigenpairs converged to requested tolerance.
- *CONVERGED_USER*: User-defined convergence criterion satisfied.
- *DIVERGED_ITS*: Maximum number of iterations exceeded.
- *DIVERGED_BREAKDOWN*: Solver failed due to breakdown.
- *DIVERGED_SYMMETRY_LOST*: Lanczos-type method could not preserve symmetry.
- *CONVERGED_ITERATING*: Iteration not finished yet.

See also
EPSCConvergedReason

Attributes Summary

<i>CONVERGED_ITERATING</i>	Constant CONVERGED_ITERATING of type <code>int</code>
<i>CONVERGED_TOL</i>	Constant CONVERGED_TOL of type <code>int</code>
<i>CONVERGED_USER</i>	Constant CONVERGED_USER of type <code>int</code>
<i>DIVERGED_BREAKDOWN</i>	Constant DIVERGED_BREAKDOWN of type <code>int</code>
<i>DIVERGED_ITS</i>	Constant DIVERGED_ITS of type <code>int</code>
<i>DIVERGED_SYMMETRY_LOST</i>	Constant DIVERGED_SYMMETRY_LOST of type <code>int</code>
<i>ITERATING</i>	Constant ITERATING of type <code>int</code>

Attributes Documentation

CONVERGED_ITERATING: `int` = CONVERGED_ITERATING

Constant CONVERGED_ITERATING of type `int`

CONVERGED_TOL: `int` = CONVERGED_TOL

Constant CONVERGED_TOL of type `int`

CONVERGED_USER: `int` = CONVERGED_USER

Constant CONVERGED_USER of type `int`

DIVERGED_BREAKDOWN: `int` = DIVERGED_BREAKDOWN

Constant DIVERGED_BREAKDOWN of type `int`

DIVERGED_ITS: `int` = DIVERGED_ITS

Constant DIVERGED_ITS of type `int`

DIVERGED_SYMMETRY_LOST: `int` = DIVERGED_SYMMETRY_LOST

Constant DIVERGED_SYMMETRY_LOST of type `int`

ITERATING: `int` = ITERATING

Constant ITERATING of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.EPS.ErrorType

class slepc4py.SLEPc.EPS.**ErrorType**

Bases: `object`

EPS error type to assess accuracy of computed solutions.

- *ABSOLUTE*: Absolute error.
- *RELATIVE*: Relative error.
- *BACKWARD*: Backward error.

See also

`EPSErrorType`

Attributes Summary

<i>ABSOLUTE</i>	Constant ABSOLUTE of type <code>int</code>
<i>BACKWARD</i>	Constant BACKWARD of type <code>int</code>
<i>RELATIVE</i>	Constant RELATIVE of type <code>int</code>

Attributes Documentation

ABSOLUTE: `int` = **ABSOLUTE**

Constant ABSOLUTE of type `int`

BACKWARD: `int` = **BACKWARD**

Constant BACKWARD of type `int`

RELATIVE: `int` = **RELATIVE**

Constant RELATIVE of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.EPS.Extraction

class slepc4py.SLEPc.EPS.Extraction

Bases: `object`

EPS extraction technique.

- *RITZ*: Standard Rayleigh-Ritz extraction.
- *HARMONIC*: Harmonic extraction.
- *HARMONIC_RELATIVE*: Harmonic extraction relative to the eigenvalue.
- *HARMONIC_RIGHT*: Harmonic extraction for rightmost eigenvalues.
- *HARMONIC_LARGEST*: Harmonic extraction for largest magnitude (without target).
- *REFINED*: Refined extraction.
- *REFINED_HARMONIC*: Refined harmonic extraction.

See also

`EPSExtraction`

Attributes Summary

<i>HARMONIC</i>	Constant HARMONIC of type <code>int</code>
<i>HARMONIC_LARGEST</i>	Constant HARMONIC_LARGEST of type <code>int</code>
<i>HARMONIC_RELATIVE</i>	Constant HARMONIC_RELATIVE of type <code>int</code>
<i>HARMONIC_RIGHT</i>	Constant HARMONIC_RIGHT of type <code>int</code>
<i>REFINED</i>	Constant REFINED of type <code>int</code>
<i>REFINED_HARMONIC</i>	Constant REFINED_HARMONIC of type <code>int</code>
<i>RITZ</i>	Constant RITZ of type <code>int</code>

Attributes Documentation

HARMONIC: `int` = **HARMONIC**

Constant HARMONIC of type `int`

HARMONIC_LARGEST: `int` = **HARMONIC_LARGEST**

Constant HARMONIC_LARGEST of type `int`

HARMONIC_RELATIVE: `int` = **HARMONIC_RELATIVE**

Constant HARMONIC_RELATIVE of type `int`

HARMONIC_RIGHT: `int` = **HARMONIC_RIGHT**

Constant HARMONIC_RIGHT of type `int`

REFINED: `int` = **REFINED**

Constant REFINED of type `int`

REFINED_HARMONIC: `int` = **REFINED_HARMONIC**

Constant REFINED_HARMONIC of type `int`

RITZ: `int` = **RITZ**

Constant RITZ of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

`slepc4py.SLEPc.EPS.KrylovSchurBSEType`

class `slepc4py.SLEPc.EPS.KrylovSchurBSEType`

Bases: `object`

EPS Krylov-Schur method for BSE problems.

- *SHAO*: Lanczos recurrence for H square.
- *GRUNING*: Lanczos recurrence for H.
- *PROJECTEDBSE*: Lanczos where the projected problem has BSE structure.

See also

`EPKrylovSchurBSEType`

Attributes Summary

<i>GRUNING</i>	Constant GRUNING of type <code>int</code>
<i>PROJECTEDBSE</i>	Constant PROJECTEDBSE of type <code>int</code>
<i>SHAO</i>	Constant SHAO of type <code>int</code>

Attributes Documentation

GRUNING: `int` = **GRUNING**

Constant GRUNING of type `int`

PROJECTEDBSE: `int` = **PROJECTEDBSE**
 Constant PROJECTEDBSE of type `int`

SHAO: `int` = **SHAO**
 Constant SHAO of type `int`

__init__()

classmethod **__new__**(*args, **kwargs)

slepc4py.SLEPc.EPS.LanczosReorthogType

class slepc4py.SLEPc.EPS.**LanczosReorthogType**

Bases: `object`

EPS Lanczos reorthogonalization type.

- *LOCAL*: Local reorthogonalization only.
- *FULL*: Full reorthogonalization.
- *SELECTIVE*: Selective reorthogonalization.
- *PERIODIC*: Periodic reorthogonalization.
- *PARTIAL*: Partial reorthogonalization.
- *DELAYED*: Delayed reorthogonalization.

See also

EPSLanczosReorthogType

Attributes Summary

<i>DELAYED</i>	Constant DELAYED of type <code>int</code>
<i>FULL</i>	Constant FULL of type <code>int</code>
<i>LOCAL</i>	Constant LOCAL of type <code>int</code>
<i>PARTIAL</i>	Constant PARTIAL of type <code>int</code>
<i>PERIODIC</i>	Constant PERIODIC of type <code>int</code>
<i>SELECTIVE</i>	Constant SELECTIVE of type <code>int</code>

Attributes Documentation

DELAYED: `int` = **DELAYED**
 Constant DELAYED of type `int`

FULL: `int` = **FULL**
 Constant FULL of type `int`

LOCAL: `int` = **LOCAL**
 Constant LOCAL of type `int`

PARTIAL: `int` = **PARTIAL**
 Constant PARTIAL of type `int`

```

PERIODIC: int = PERIODIC
    Constant PERIODIC of type int
SELECTIVE: int = SELECTIVE
    Constant SELECTIVE of type int
__init__()

classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.EPS.PowerShiftType

```

class slepc4py.SLEPc.EPS.PowerShiftType
    Bases: object
    EPS Power shift type.
    • CONSTANT: Constant shift.
    • RAYLEIGH: Rayleigh quotient.
    • WILKINSON: Wilkinson shift.

```

See also
<code>EPSPowerShiftType</code>

Attributes Summary

<i>CONSTANT</i>	Constant <i>CONSTANT</i> of type <code>int</code>
<i>RAYLEIGH</i>	Constant <i>RAYLEIGH</i> of type <code>int</code>
<i>WILKINSON</i>	Constant <i>WILKINSON</i> of type <code>int</code>

Attributes Documentation

```

CONSTANT: int = CONSTANT
    Constant CONSTANT of type int
RAYLEIGH: int = RAYLEIGH
    Constant RAYLEIGH of type int
WILKINSON: int = WILKINSON
    Constant WILKINSON of type int
__init__()

classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.EPS.ProblemType

```

class slepc4py.SLEPc.EPS.ProblemType
    Bases: object
    EPS problem type.
    • HEP: Hermitian eigenproblem.

```

- *NHEP*: Non-Hermitian eigenproblem.
- *GHEP*: Generalized Hermitian eigenproblem.
- *GNHEP*: Generalized Non-Hermitian eigenproblem.
- *PGNHEP*: Generalized Non-Hermitian eigenproblem with positive definite B .
- *GHIEP*: Generalized Hermitian-indefinite eigenproblem.
- *BSE*: Structured Bethe-Salpeter eigenproblem.
- *HAMILT*: Hamiltonian eigenproblem.
- *LREP*: Structured Linear Response eigenvalue problem.

See also

EPSProblemType

Attributes Summary

<i>BSE</i>	Constant BSE of type <code>int</code>
<i>GHEP</i>	Constant GHEP of type <code>int</code>
<i>GHIEP</i>	Constant GHIEP of type <code>int</code>
<i>GNHEP</i>	Constant GNHEP of type <code>int</code>
<i>HAMILT</i>	Constant HAMILT of type <code>int</code>
<i>HEP</i>	Constant HEP of type <code>int</code>
<i>LREP</i>	Constant LREP of type <code>int</code>
<i>NHEP</i>	Constant NHEP of type <code>int</code>
<i>PGNHEP</i>	Constant PGNHEP of type <code>int</code>

Attributes Documentation

BSE: `int` = BSE

Constant BSE of type `int`

GHEP: `int` = GHEP

Constant GHEP of type `int`

GHIEP: `int` = GHIEP

Constant GHIEP of type `int`

GNHEP: `int` = GNHEP

Constant GNHEP of type `int`

HAMILT: `int` = HAMILT

Constant HAMILT of type `int`

HEP: `int` = HEP

Constant HEP of type `int`

LREP: `int` = LREP

Constant LREP of type `int`

NHEP: `int` = NHEP

Constant NHEP of type `int`

PGNHEP: `int` = PGNHEP
 Constant PGNHEP of type `int`

`__init__()`

classmethod `__new__`(*args, **kwargs)

slepc4py.SLEPc.EPS.Stop

class slepc4py.SLEPc.EPS.Stop
 Bases: `object`

EPS stopping test.

- *BASIC*: Default stopping test.
- *USER*: User-defined stopping test.
- *THRESHOLD*: Threshold stopping test.

See also

EPSSStop

Attributes Summary

<i>BASIC</i>	Constant BASIC of type <code>int</code>
<i>THRESHOLD</i>	Constant THRESHOLD of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

BASIC: `int` = BASIC
 Constant BASIC of type `int`

THRESHOLD: `int` = THRESHOLD
 Constant THRESHOLD of type `int`

USER: `int` = USER
 Constant USER of type `int`

`__init__()`

classmethod `__new__`(*args, **kwargs)

slepc4py.SLEPc.EPS.Type

class slepc4py.SLEPc.EPS.Type
 Bases: `object`

EPS type.

Native eigenvalue solvers.

- *POWER*: Power Iteration, Inverse Iteration, RQI.
- *SUBSPACE*: Subspace Iteration.

- *ARNOLDI*: Arnoldi.
- *LANCZOS*: Lanczos.
- *KRYLOV SCHUR*: Krylov-Schur (default).
- *GD*: Generalized Davidson.
- *JD*: Jacobi-Davidson.
- *RQCG*: Rayleigh Quotient Conjugate Gradient.
- *LOBPCG*: Locally Optimal Block Preconditioned Conjugate Gradient.
- *CISS*: Contour Integral Spectrum Slicing.
- *LYAPII*: Lyapunov inverse iteration.

Wrappers to external eigensolvers (should be enabled during installation of SLEPc).

- *LAPACK*: Sequential dense eigensolver.
- *ARPACK*: Iterative Krylov-based eigensolver.
- *BLOPEX*: Implementation of LOBPCG.
- *PRIMME*: Iterative eigensolvers of Davidson type.
- *FEAST*: Contour integral eigensolver.
- *SCALAPACK*: Parallel dense eigensolver for symmetric problems.
- *ELPA*: Parallel dense eigensolver for symmetric problems.
- *ELEMENTAL*: Parallel dense eigensolver for symmetric problems.
- *EVSL*: Iterative eigensolver based on polynomial filters.
- *CHASE*: Subspace iteration accelerated with polynomials.

See also

EPSType

Attributes Summary

<i>ARNOLDI</i>	Object ARNOLDI of type <i>str</i>
<i>ARPACK</i>	Object ARPACK of type <i>str</i>
<i>BLOPEX</i>	Object BLOPEX of type <i>str</i>
<i>CHASE</i>	Object CHASE of type <i>str</i>
<i>CISS</i>	Object CISS of type <i>str</i>
<i>ELEMENTAL</i>	Object ELEMENTAL of type <i>str</i>
<i>ELPA</i>	Object ELPA of type <i>str</i>
<i>EVSL</i>	Object EVSL of type <i>str</i>
<i>FEAST</i>	Object FEAST of type <i>str</i>
<i>GD</i>	Object GD of type <i>str</i>
<i>JD</i>	Object JD of type <i>str</i>
<i>KRYLOV SCHUR</i>	Object KRYLOV SCHUR of type <i>str</i>
<i>LANCZOS</i>	Object LANCZOS of type <i>str</i>
<i>LAPACK</i>	Object LAPACK of type <i>str</i>
<i>LOBPCG</i>	Object LOBPCG of type <i>str</i>

continues on next page

Table 34 – continued from previous page

<i>LYAPII</i>	Object LYAPII of type <i>str</i>
<i>POWER</i>	Object POWER of type <i>str</i>
<i>PRIMME</i>	Object PRIMME of type <i>str</i>
<i>RQCG</i>	Object RQCG of type <i>str</i>
<i>SCALAPACK</i>	Object SCALAPACK of type <i>str</i>
<i>SUBSPACE</i>	Object SUBSPACE of type <i>str</i>

Attributes Documentation

ARNOLDI: *str* = ARNOLDI

Object ARNOLDI of type *str*

ARPACK: *str* = ARPACK

Object ARPACK of type *str*

BLOPEX: *str* = BLOPEX

Object BLOPEX of type *str*

CHASE: *str* = CHASE

Object CHASE of type *str*

CISS: *str* = CISS

Object CISS of type *str*

ELEMENTAL: *str* = ELEMENTAL

Object ELEMENTAL of type *str*

ELPA: *str* = ELPA

Object ELPA of type *str*

EVSL: *str* = EVSL

Object EVSL of type *str*

FEAST: *str* = FEAST

Object FEAST of type *str*

GD: *str* = GD

Object GD of type *str*

JD: *str* = JD

Object JD of type *str*

KRYLOVSHUR: *str* = KRYLOVSHUR

Object KRYLOVSHUR of type *str*

LANCZOS: *str* = LANCZOS

Object LANCZOS of type *str*

LAPACK: *str* = LAPACK

Object LAPACK of type *str*

LOBPCG: *str* = LOBPCG

Object LOBPCG of type *str*

LYAPII: *str* = LYAPII

Object LYAPII of type *str*

POWER: `str` = **POWER**
 Object POWER of type `str`

PRIMME: `str` = **PRIMME**
 Object PRIMME of type `str`

RQCG: `str` = **RQCG**
 Object RQCG of type `str`

SCALAPACK: `str` = **SCALAPACK**
 Object SCALAPACK of type `str`

SUBSPACE: `str` = **SUBSPACE**
 Object SUBSPACE of type `str`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.EPS.Which

class slepc4py.SLEPc.EPS.**Which**

Bases: `object`

EPS desired part of spectrum.

- `LARGEST_MAGNITUDE`: Largest magnitude (default).
- `SMALLEST_MAGNITUDE`: Smallest magnitude.
- `LARGEST_REAL`: Largest real parts.
- `SMALLEST_REAL`: Smallest real parts.
- `LARGEST_IMAGINARY`: Largest imaginary parts in magnitude.
- `SMALLEST_IMAGINARY`: Smallest imaginary parts in magnitude.
- `TARGET_MAGNITUDE`: Closest to target (in magnitude).
- `TARGET_REAL`: Real part closest to target.
- `TARGET_IMAGINARY`: Imaginary part closest to target.
- `ALL`: All eigenvalues in an interval.
- `USER`: User defined selection.

See also

`EPSWhich`

Attributes Summary

<code>ALL</code>	Constant ALL of type <code>int</code>
<code>LARGEST_IMAGINARY</code>	Constant LARGEST_IMAGINARY of type <code>int</code>
<code>LARGEST_MAGNITUDE</code>	Constant LARGEST_MAGNITUDE of type <code>int</code>
<code>LARGEST_REAL</code>	Constant LARGEST_REAL of type <code>int</code>

continues on next page

Table 35 – continued from previous page

<i>SMALLEST_IMAGINARY</i>	Constant <i>SMALLEST_IMAGINARY</i> of type <i>int</i>
<i>SMALLEST_MAGNITUDE</i>	Constant <i>SMALLEST_MAGNITUDE</i> of type <i>int</i>
<i>SMALLEST_REAL</i>	Constant <i>SMALLEST_REAL</i> of type <i>int</i>
<i>TARGET_IMAGINARY</i>	Constant <i>TARGET_IMAGINARY</i> of type <i>int</i>
<i>TARGET_MAGNITUDE</i>	Constant <i>TARGET_MAGNITUDE</i> of type <i>int</i>
<i>TARGET_REAL</i>	Constant <i>TARGET_REAL</i> of type <i>int</i>
<i>USER</i>	Constant <i>USER</i> of type <i>int</i>

Attributes Documentation

ALL: *int* = **ALL**

Constant **ALL** of type *int*

LARGEST_IMAGINARY: *int* = **LARGEST_IMAGINARY**

Constant **LARGEST_IMAGINARY** of type *int*

LARGEST_MAGNITUDE: *int* = **LARGEST_MAGNITUDE**

Constant **LARGEST_MAGNITUDE** of type *int*

LARGEST_REAL: *int* = **LARGEST_REAL**

Constant **LARGEST_REAL** of type *int*

SMALLEST_IMAGINARY: *int* = **SMALLEST_IMAGINARY**

Constant **SMALLEST_IMAGINARY** of type *int*

SMALLEST_MAGNITUDE: *int* = **SMALLEST_MAGNITUDE**

Constant **SMALLEST_MAGNITUDE** of type *int*

SMALLEST_REAL: *int* = **SMALLEST_REAL**

Constant **SMALLEST_REAL** of type *int*

TARGET_IMAGINARY: *int* = **TARGET_IMAGINARY**

Constant **TARGET_IMAGINARY** of type *int*

TARGET_MAGNITUDE: *int* = **TARGET_MAGNITUDE**

Constant **TARGET_MAGNITUDE** of type *int*

TARGET_REAL: *int* = **TARGET_REAL**

Constant **TARGET_REAL** of type *int*

USER: *int* = **USER**

Constant **USER** of type *int*

__init__()

classmethod **__new__**(*args, **kwargs)

Methods Summary

<i>appendOptionsPrefix</i> ([prefix])	Append to the prefix used for searching for all EPS options in the database.
<i>cancelMonitor</i> ()	Clear all monitors for an <i>EPS</i> object.
<i>computeError</i> (i[, etype])	Compute the error associated with the i-th computed eigenpair.

continues on next page

Table 36 – continued from previous page

<code>create([comm])</code>	Create the EPS object.
<code>destroy()</code>	Destroy the EPS object.
<code>errorView([etype, viewer])</code>	Display the errors associated with the computed solution.
<code>getArbitrarySelection()</code>	Get the arbitrary selection function.
<code>getArnoldiDelayed()</code>	Get the type of reorthogonalization used during the Arnoldi iteration.
<code>getBV()</code>	Get the basis vectors object associated to the eigensolver.
<code>getBalance()</code>	Get the balancing type used by the EPS, and the associated parameters.
<code>getCISSExtraction()</code>	Get the extraction technique used in the CISS solver.
<code>getCISSKSPs()</code>	Get the array of linear solver objects associated with the CISS solver.
<code>getCISSQuadRule()</code>	Get the quadrature rule used in the CISS solver.
<code>getCISSRefinement()</code>	Get the values of various refinement parameters in the CISS solver.
<code>getCISSSizes()</code>	Get the values of various size parameters in the CISS solver.
<code>getCISSThreshold()</code>	Get the values of various threshold parameters in the CISS solver.
<code>getCISSUseST()</code>	Get the flag indicating the use of the ST object in the CISS solver.
<code>getConverged()</code>	Get the number of converged eigenpairs.
<code>getConvergedReason()</code>	Get the reason why the <code>solve()</code> iteration was stopped.
<code>getConvergenceTest()</code>	Get how to compute the error estimate used in the convergence test.
<code>getDS()</code>	Get the direct solver associated to the eigensolver.
<code>getDimensions()</code>	Get number of eigenvalues to compute and the dimension of the subspace.
<code>getEigenpair(i[, Vr, Vi])</code>	Get the i-th solution of the eigenproblem as computed by <code>solve()</code> .
<code>getEigenvalue(i)</code>	Get the i-th eigenvalue as computed by <code>solve()</code> .
<code>getEigenvalueComparison()</code>	Get the eigenvalue comparison function.
<code>getEigenvector(i[, Vr, Vi])</code>	Get the i-th right eigenvector as computed by <code>solve()</code> .
<code>getErrorEstimate(i)</code>	Get the error estimate associated to the i-th computed eigenpair.
<code>getExtraction()</code>	Get the extraction type used by the EPS object.
<code>getGDBOrth()</code>	Get the orthogonalization used in the search subspace.
<code>getGDBBlockSize()</code>	Get the number of vectors to be added to the searching space.
<code>getGDDoubleExpansion()</code>	Get a flag indicating whether the double expansion variant is active.
<code>getGDInitialSize()</code>	Get the initial size of the searching space.
<code>getGDKrylovStart()</code>	Get a flag indicating if the search subspace is started with a Krylov basis.
<code>getGDRestart()</code>	Get the number of vectors of the search space after restart.
<code>getInterval()</code>	Get the computational interval for spectrum slicing.

continues on next page

Table 36 – continued from previous page

<code>getInvariantSubspace()</code>	Get an orthonormal basis of the computed invariant subspace.
<code>getIterationNumber()</code>	Get the current iteration number.
<code>getJDBOrth()</code>	Get the orthogonalization used in the search subspace.
<code>getJDBlockSize()</code>	Get the number of vectors to be added to the searching space.
<code>getJDConstCorrectionTol()</code>	Get the flag indicating if the dynamic stopping is being used.
<code>getJDFix()</code>	Get the threshold for changing the target in the correction equation.
<code>getJDInitialSize()</code>	Get the initial size of the searching space.
<code>getJDKrylovStart()</code>	Get a flag indicating if the search subspace is started with a Krylov basis.
<code>getJDRestart()</code>	Get the number of vectors of the search space after restart.
<code>getKrylovSchurBSEType()</code>	Get the method used for BSE structured eigenproblems (Krylov-Schur).
<code>getKrylovSchurDetectZeros()</code>	Get the flag that enforces zero detection in spectrum slicing.
<code>getKrylovSchurDimensions()</code>	Get the dimensions used for each subsolve step (spectrum slicing).
<code>getKrylovSchurInertias()</code>	Get the values of the shifts and their corresponding inertias.
<code>getKrylovSchurKSP()</code>	Get the linear solver object associated with the internal EPS object.
<code>getKrylovSchurLocking()</code>	Get the locking flag used in the Krylov-Schur method.
<code>getKrylovSchurPartitions()</code>	Get the number of partitions of the communicator (spectrum slicing).
<code>getKrylovSchurRestart()</code>	Get the restart parameter used in the Krylov-Schur method.
<code>getKrylovSchurSubcommInfo()</code>	Get information related to the case of doing spectrum slicing.
<code>getKrylovSchurSubcommMats()</code>	Get the eigenproblem matrices stored in the subcommunicator.
<code>getKrylovSchurSubcommPairs(i[, v])</code>	Get the i-th eigenpair stored in the multi-communicator of the process.
<code>getKrylovSchurSubintervals()</code>	Get the points that delimit the subintervals.
<code>getLOBPCGBlockSize()</code>	Get the block size used in the LOBPCG method.
<code>getLOBPCGLocking()</code>	Get the locking flag used in the LOBPCG method.
<code>getLOBPCGRestart()</code>	Get the restart parameter used in the LOBPCG method.
<code>getLanczosReorthogType()</code>	Get the type of reorthogonalization used during the Lanczos iteration.
<code>getLeftEigenvector(i[, Wr, Wi])</code>	Get the i-th left eigenvector as computed by solve() .
<code>getLyapIIRanks()</code>	Get the rank values used for the Lyapunov step.
<code>getMonitor()</code>	Get the list of monitor functions.
<code>getOperators()</code>	Get the matrices associated with the eigenvalue problem.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all EPS options in the database.
<code>getPowerShiftType()</code>	Get the type of shifts used during the power iteration.

continues on next page

Table 36 – continued from previous page

<i>getProblemType()</i>	Get the problem type from the EPS object.
<i>getPurify()</i>	Get the flag indicating whether purification is activated or not.
<i>getRG()</i>	Get the region object associated to the eigensolver.
<i>getRQCGReset()</i>	Get the reset parameter used in the RQCG method.
<i>getST()</i>	Get the spectral transformation object associated to the eigensolver.
<i>getStoppingTest()</i>	Get the stopping test function.
<i>getTarget()</i>	Get the value of the target.
<i>getThreshold()</i>	Get the threshold used in the threshold stopping test.
<i>getTolerances()</i>	Get the tolerance and max.
<i>getTrackAll()</i>	Get the flag indicating if all residual norms must be computed or not.
<i>getTrueResidual()</i>	Get the flag indicating if true residual must be computed explicitly.
<i>getTwoSided()</i>	Get the flag indicating if a two-sided variant of the algorithm is being used.
<i>getType()</i>	Get the EPS type of this object.
<i>getWhichEigenpairs()</i>	Get which portion of the spectrum is to be sought.
<i>isGeneralized()</i>	Tell if the EPS object corresponds to a generalized eigenproblem.
<i>isHermitian()</i>	Tell if the EPS object corresponds to a Hermitian eigenproblem.
<i>isPositive()</i>	Eigenproblem requiring a positive (semi-) definite matrix B .
<i>isStructured()</i>	Tell if the EPS object corresponds to a structured eigenvalue problem.
<i>reset()</i>	Reset the EPS object.
<i>setArbitrarySelection(arbitrary[, args, kargs])</i>	Set an arbitrary selection criterion function.
<i>setArnoldiDelayed(delayed)</i>	Set (toggle) delayed reorthogonalization in the Arnoldi iteration.
<i>setBV(bv)</i>	Set a basis vectors object associated to the eigensolver.
<i>setBalance([balance, iterations, cutoff])</i>	Set the balancing technique to be used by the eigensolver.
<i>setCISSExtraction(extraction)</i>	Set the extraction technique used in the CISS solver.
<i>setCISSQuadRule(quad)</i>	Set the quadrature rule used in the CISS solver.
<i>setCISSRefinement([inner, blsize])</i>	Set the values of various refinement parameters in the CISS solver.
<i>setCISSSizes([ip, bs, ms, npart, bsmax, ...])</i>	Set the values of various size parameters in the CISS solver.
<i>setCISSThreshold([delta, spur])</i>	Set the values of various threshold parameters in the CISS solver.
<i>setCISSUseST(usest)</i>	Set a flag indicating that the CISS solver will use the <i>ST</i> object.
<i>setConvergenceTest(conv)</i>	Set how to compute the error estimate used in the convergence test.
<i>setDS(ds)</i>	Set a direct solver object associated to the eigensolver.
<i>setDeflationSpace(space)</i>	Set vectors to form a basis of the deflation space.
<i>setDimensions([nev, ncv, mpd])</i>	Set number of eigenvalues to compute and the dimension of the subspace.

continues on next page

Table 36 – continued from previous page

<code>setEigenvalueComparison</code> (comparison[, args, ...])	Set an eigenvalue comparison function.
<code>setExtraction</code> (extraction)	Set the extraction type used by the eigensolver.
<code>setFromOptions</code> ()	Set EPS options from the options database.
<code>setGDBOrth</code> (borth)	Set the orthogonalization that will be used in the search subspace.
<code>setGDBlockSize</code> (bs)	Set the number of vectors to be added to the searching space.
<code>setGDDoubleExpansion</code> (doubleexp)	Set that the search subspace is expanded with double expansion.
<code>setGDInitialSize</code> (initialsize)	Set the initial size of the searching space.
<code>setGDKrylovStart</code> ([krylovstart])	Set (toggle) starting the search subspace with a Krylov basis.
<code>setGDRestart</code> ([minv, plusk])	Set the number of vectors of the search space after restart.
<code>setInitialSpace</code> (space)	Set the initial space from which the eigensolver starts to iterate.
<code>setInterval</code> (inta, intb)	Set the computational interval for spectrum slicing.
<code>setJDBOrth</code> (borth)	Set the orthogonalization that will be used in the search subspace.
<code>setJDBlockSize</code> (bs)	Set the number of vectors to be added to the searching space.
<code>setJDConstCorrectionTol</code> (constant)	Deactivate the dynamic stopping criterion.
<code>setJDFix</code> (fix)	Set the threshold for changing the target in the correction equation.
<code>setJDInitialSize</code> (initialsize)	Set the initial size of the searching space.
<code>setJDKrylovStart</code> ([krylovstart])	Set (toggle) starting the search subspace with a Krylov basis.
<code>setJDRestart</code> ([minv, plusk])	Set the number of vectors of the search space after restart.
<code>setKrylovSchurBSEType</code> (bse)	Set the Krylov-Schur variant used for BSE structured eigenproblems.
<code>setKrylovSchurDetectZeros</code> (detect)	Set the flag that enforces zero detection in spectrum slicing.
<code>setKrylovSchurDimensions</code> ([nev, ncv, mpd])	Set the dimensions used for each subsolve step (spectrum slicing).
<code>setKrylovSchurLocking</code> (lock)	Set (toggle) locking/non-locking variants of the Krylov-Schur method.
<code>setKrylovSchurPartitions</code> (npart)	Set the number of partitions of the communicator (spectrum slicing).
<code>setKrylovSchurRestart</code> (keep)	Set the restart parameter for the Krylov-Schur method.
<code>setKrylovSchurSubintervals</code> (subint)	Set the subinterval boundaries.
<code>setLOBPCGBlockSize</code> (bs)	Set the block size of the LOBPCG method.
<code>setLOBPCGLocking</code> (lock)	Toggle between locking and non-locking (LOBPCG method).
<code>setLOBPCGRestart</code> (restart)	Set the restart parameter for the LOBPCG method.
<code>setLanczosReorthogType</code> (reorthog)	Set the type of reorthogonalization used during the Lanczos iteration.
<code>setLeftInitialSpace</code> (space)	Set a left initial space from which the eigensolver starts to iterate.

continues on next page

Table 36 – continued from previous page

<code>setLyapIIRanks([rkc, rkl])</code>	Set the ranks used in the solution of the Lyapunov equation.
<code>setMonitor(monitor[, args, kargs])</code>	Append a monitor function to the list of monitors.
<code>setOperators(A[, B])</code>	Set the matrices associated with the eigenvalue problem.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all EPS options in the database.
<code>setPowerShiftType(shift)</code>	Set the type of shifts used during the power iteration.
<code>setProblemType(problem_type)</code>	Set the type of the eigenvalue problem.
<code>setPurify([purify])</code>	Set (toggle) eigenvector purification.
<code>setRG(rg)</code>	Set a region object associated to the eigensolver.
<code>setRQCGReset(nreset)</code>	Set the reset parameter of the RQCG iteration.
<code>setST(st)</code>	Set a spectral transformation object associated to the eigensolver.
<code>setStoppingTest(stopping[, args, kargs])</code>	Set a function to decide when to stop the outer iteration of the eigensolver.
<code>setTarget(target)</code>	Set the value of the target.
<code>setThreshold(thres[, rel])</code>	Set the threshold used in the threshold stopping test.
<code>setTolerances([tol, max_it])</code>	Set the tolerance and max.
<code>setTrackAll(trackall)</code>	Set if the solver must compute the residual of all approximate eigenpairs.
<code>setTrueResidual(trueres)</code>	Set if the solver must compute the true residual explicitly or not.
<code>setTwoSided(twosided)</code>	Set to use a two-sided variant that also computes left eigenvectors.
<code>setType(eps_type)</code>	Set the particular solver to be used in the EPS object.
<code>setUp()</code>	Set up all the internal data structures.
<code>setWhichEigenpairs(which)</code>	Set which portion of the spectrum is to be sought.
<code>solve()</code>	Solve the eigensystem.
<code>updateKrylovSchurSubcommMats([s, a, Au, t, ...])</code>	Update the eigenproblem matrices stored internally in the communicator.
<code>valuesView([viewer])</code>	Display the computed eigenvalues in a viewer.
<code>vectorsView([viewer])</code>	Output computed eigenvectors to a viewer.
<code>view([viewer])</code>	Print the EPS data structure.

Attributes Summary

<code>bv</code>	The basis vectors (<i>BV</i>) object associated.
<code>ds</code>	The direct solver (<i>DS</i>) object associated.
<code>extraction</code>	The type of extraction technique to be employed.
<code>max_it</code>	The maximum iteration count.
<code>problem_type</code>	The type of the eigenvalue problem.
<code>purify</code>	Eigenvector purification.
<code>rg</code>	The region (<i>RG</i>) object associated.
<code>st</code>	The spectral transformation (<i>ST</i>) object associated.
<code>target</code>	The value of the target.
<code>tol</code>	The tolerance.
<code>track_all</code>	Compute the residual norm of all approximate eigenpairs.
<code>true_residual</code>	Compute the true residual explicitly.
<code>two_sided</code>	Two-sided that also computes left eigenvectors.

continues on next page

Table 37 – continued from previous page

<i>which</i>	The portion of the spectrum to be sought.
--------------	---

Methods Documentation

appendOptionsPrefix(*prefix=None*)

Append to the prefix used for searching for all EPS options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all EPS option requests.

Return type

None

See also

setOptionsPrefix, *getOptionsPrefix*, *EPSAppendOptionsPrefix*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:515 <slepc4py/SLEPc/EPS.pyx#L515>](#)

cancelMonitor()

Clear all monitors for an *EPS* object.

Logically collective.

See also

EPSMonitorCancel

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:1862 <slepc4py/SLEPc/EPS.pyx#L1862>](#)

Return type

None

computeError(*i, etype=None*)

Compute the error associated with the *i*-th computed eigenpair.

Collective.

Compute the error (based on the residual norm) associated with the *i*-th computed eigenpair.

Parameters

- **i** (*int*) – Index of the solution to be considered.
- **etype** (*ErrorType* / *None*) – The error type to compute.

Returns

The error bound, computed in various ways from the residual norm $\|Ax - \lambda Bx\|_2$ where λ is the eigenvalue and x is the eigenvector.

Return type

float

Notes

The index `i` should be a value between 0 and `nconv-1` (see `getConverged()`).

If the computation of left eigenvectors was enabled with `setTwoSided()`, then the error will be computed using the maximum of the value above and the left residual norm $\|y^*A - \lambda y^*B\|_2$, where y is the approximate left eigenvector.

See also

`getErrorEstimate`, `setTwoSided`, `EPSComputeError`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2215 <slepc4py/SLEPc/EPS.pyx#L2215>`

create(*comm=None*)

Create the EPS object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator; if not provided, it defaults to all processes.

Return type

Self

See also

`EPSCreate`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:398 <slepc4py/SLEPc/EPS.pyx#L398>`

destroy()

Destroy the EPS object.

Collective.

See also

`EPSDestroy`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:372 <slepc4py/SLEPc/EPS.pyx#L372>`

Return type

Self

errorView(*etype=None*, *viewer=None*)

Display the errors associated with the computed solution.

Collective.

Display the errors and the eigenvalues.

Parameters

- **etype** (*ErrorType* / *None*) – The error type to compute.
- **viewer** (*petsc4py.PETSc.Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

Notes

By default, this function checks the error of all eigenpairs and prints the eigenvalues if all of them are below the requested tolerance. If the viewer has format ASCII_INFO_DETAIL then a table with eigenvalues and corresponding errors is printed.

See also

solve, *valuesView*, *vectorsView*, *EPSErrorView*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2258 <slepc4py/SLEPc/EPS.pyx#L2258>](#)

getArbitrarySelection()

Get the arbitrary selection function.

Not collective.

Returns

The arbitrary selection function.

Return type

EPsArbitraryFunction

See also

setArbitrarySelection

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:1756 <slepc4py/SLEPc/EPS.pyx#L1756>](#)

getArnoldiDelayed()

Get the type of reorthogonalization used during the Arnoldi iteration.

Not collective.

Returns

True if delayed reorthogonalization is to be used.

Return type

bool

See also

setArnoldiDelayed, *EPsArnoldiGetDelayed*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2413 <slepc4py/SLEPc/EPS.pyx#L2413>](#)

getBV()

Get the basis vectors object associated to the eigensolver.

Not collective.

Returns

The basis vectors context.

Return type*BV***See also***setBV*, *EPSGetBV***:sources:** `Source code at slepc4py/SLEPc/EPS.pyx:1409 <slepc4py/SLEPc/EPS.pyx#L1409>`**getBalance()**

Get the balancing type used by the EPS, and the associated parameters.

Not collective.

Returns

- **balance** (*Balance*) – The balancing method.
- **iterations** (*int*) – Number of iterations of the balancing algorithm.
- **cutoff** (*float*) – Cutoff value.

Return type*tuple[Balance, int, float]***See also***setBalance*, *EPSGetBalance***:sources:** `Source code at slepc4py/SLEPc/EPS.pyx:695 <slepc4py/SLEPc/EPS.pyx#L695>`**getCISSExtraction()**

Get the extraction technique used in the CISS solver.

Not collective.

Returns

The extraction technique.

Return type*CISSExtraction***See also***setCISSExtraction*, *EPSCISSGetExtraction***:sources:** `Source code at slepc4py/SLEPc/EPS.pyx:3959 <slepc4py/SLEPc/EPS.pyx#L3959>`**getCISSKSPs()**

Get the array of linear solver objects associated with the CISS solver.

Not collective.

Returns

The linear solver objects.

Return type*list of petsc4py.PETSc.KSP*

Notes

The number of `petsc4py.PETSc.KSP` solvers is equal to the number of integration points divided by the number of partitions. This value is halved in the case of real matrices with a region centered at the real axis.

See also

`setCISSSizes`, `EPSCISSGetKSPs`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4240 <slepc4py/SLEPc/EPs.pyx#L4240>`

`getCISSQuadRule()`

Get the quadrature rule used in the CISS solver.

Not collective.

Returns

The quadrature rule.

Return type

`CISSQuadRule`

See also

`setCISSQuadRule`, `EPSCISSGetQuadRule`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:3996 <slepc4py/SLEPc/EPs.pyx#L3996>`

`getCISSRefinement()`

Get the values of various refinement parameters in the CISS solver.

Not collective.

Returns

- **inner** (`int`) – Number of iterative refinement iterations (inner loop).
- **blsize** (`int`) – Number of iterative refinement iterations (blocksize loop).

Return type

`tuple[int, int]`

See also

`setCISSRefinement`, `EPSCISSGetRefinement`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4171 <slepc4py/SLEPc/EPs.pyx#L4171>`

`getCISSSizes()`

Get the values of various size parameters in the CISS solver.

Not collective.

Returns

- **ip** (`int`) – Number of integration points.

- **bs** (`int`) – Block size.
- **ms** (`int`) – Moment size.
- **npart** (`int`) – Number of partitions when splitting the communicator.
- **bsmax** (`int`) – Maximum block size.
- **realmats** (`bool`) – True if A and B are real.

Return type

`tuple[int, int, int, int, int, bool]`

See also

`setCISSSizes`, `EPSCISSGetSizes`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4069 <slepc4py/SLEPc/EPs.pyx#L4069>`

getCISSThreshold()

Get the values of various threshold parameters in the CISS solver.

Not collective.

Returns

- **delta** (`float`) – Threshold for numerical rank.
- **spur** (`float`) – Spurious threshold (to discard spurious eigenpairs).

Return type

`tuple[float, float]`

See also

`setCISSThreshold`, `EPSCISSGetThreshold`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4126 <slepc4py/SLEPc/EPs.pyx#L4126>`

getCISSUseST()

Get the flag indicating the use of the *ST* object in the CISS solver.

Not collective.

Returns

Whether to use the *ST* object or not.

Return type

`bool`

See also

`setCISSUseST`, `EPSCISSGetUseST`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4221 <slepc4py/SLEPc/EPs.pyx#L4221>`

getConverged()

Get the number of converged eigenpairs.

Not collective.

Returns

nconv – Number of converged eigenpairs.

Return type

int

Notes

This function should be called after *solve()* has finished.

The value *nconv* may be different from the number of requested solutions *nev*, but not larger than *ncv*, see *setDimensions()*.

See also

setDimensions, *solve*, *getEigenpair*, *EPSGetConverged*

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1962 <slepc4py/SLEPc/EPS.pyx#L1962>`

getConvergedReason()

Get the reason why the *solve()* iteration was stopped.

Not collective.

Returns

Negative value indicates diverged, positive value converged.

Return type

ConvergedReason

See also

setTolerances, *solve*, *EPSGetConvergedReason*

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1943 <slepc4py/SLEPc/EPS.pyx#L1943>`

getConvergenceTest()

Get how to compute the error estimate used in the convergence test.

Not collective.

Returns

The method used to compute the error estimate used in the convergence test.

Return type

Conv

See also

setConvergenceTest, *EPSGetConvergenceTest*

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1179 <slepc4py/SLEPc/EPS.pyx#L1179>`

getDS()

Get the direct solver associated to the eigensolver.

Not collective.

Returns

The direct solver context.

Return type

DS

See also

setDS, *EPSGetDS*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1446 <slepc4py/SLEPc/EPs.pyx#L1446>](#)

getDimensions()

Get number of eigenvalues to compute and the dimension of the subspace.

Not collective.

Returns

- **nev** (*int*) – Number of eigenvalues to compute.
- **ncv** (*int*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int*) – Maximum dimension allowed for the projected problem.

Return type

tuple[*int*, *int*, *int*]

See also

setDimensions, *EPSGetDimensions*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1292 <slepc4py/SLEPc/EPs.pyx#L1292>](#)

getEigenpair(*i*, *Vr*=None, *Vi*=None)

Get the *i*-th solution of the eigenproblem as computed by *solve()*.

Collective.

The solution consists of both the eigenvalue and the eigenvector.

Parameters

- **i** (*int*) – Index of the solution to be obtained.
- **Vr** (*Vec* / *None*) – Placeholder for the returned eigenvector (real part).
- **Vi** (*Vec* / *None*) – Placeholder for the returned eigenvector (imaginary part).

Returns

e – The computed eigenvalue. It will be a real variable in case of a Hermitian or generalized Hermitian eigenproblem. Otherwise it will be a complex variable (possibly with zero imaginary part).

Return type

Scalar

Notes

The index `i` should be a value between 0 and `nconv-1` (see `getConverged()`). Eigenpairs are indexed according to the ordering criterion established with `setWhichEigenpairs()`.

The 2-norm of the eigenvector is one unless the problem is generalized Hermitian. In this case the eigenvector is normalized with respect to the norm defined by the B matrix.

See also

`solve`, `getConverged`, `setWhichEigenpairs`, `EPSGetEigenpair`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2092 <slepc4py/SLEPc/EPS.pyx#L2092>`

`getEigenvalue(i)`

Get the `i`-th eigenvalue as computed by `solve()`.

Not collective.

Parameters

`i` (*int*) – Index of the solution to be obtained.

Returns

The computed eigenvalue. It will be a real variable in case of a Hermitian or generalized Hermitian eigenproblem. Otherwise it will be a complex variable (possibly with zero imaginary part).

Return type

Scalar

Notes

The index `i` should be a value between 0 and `nconv-1` (see `getConverged()`). Eigenpairs are indexed according to the ordering criterion established with `setWhichEigenpairs()`.

See also

`getConverged`, `setWhichEigenpairs`, `getEigenpair`, `EPSGetEigenvalue`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1988 <slepc4py/SLEPc/EPS.pyx#L1988>`

`getEigenvalueComparison()`

Get the eigenvalue comparison function.

Not collective.

Returns

The eigenvalue comparison function.

Return type

EPSEigenvalueComparison

See also

[`setEigenvalueComparison`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1803 <slepc4py/SLEPc/EPS.pyx#L1803>`

getEigenvector(*i*, *Vr*=None, *Vi*=None)

Get the *i*-th right eigenvector as computed by [`solve\(\)`](#).

Collective.

Parameters

- **i** (*int*) – Index of the solution to be obtained.
- **Vr** (*Vec* / *None*) – Placeholder for the returned eigenvector (real part).
- **Vi** (*Vec* / *None*) – Placeholder for the returned eigenvector (imaginary part).

Return type

None

Notes

The index *i* should be a value between 0 and `nconv-1` (see [`getConverged\(\)`](#)). Eigenpairs are indexed according to the ordering criterion established with [`setWhichEigenpairs\(\)`](#).

The 2-norm of the eigenvector is one unless the problem is generalized Hermitian. In this case the eigenvector is normalized with respect to the norm defined by the B matrix.

See also

[`getConverged`](#), [`setWhichEigenpairs`](#), [`getEigenpair`](#), [`EPSGetEigenvector`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2026 <slepc4py/SLEPc/EPS.pyx#L2026>`

getErrorEstimate(*i*)

Get the error estimate associated to the *i*-th computed eigenpair.

Not collective.

Parameters

- **i** (*int*) – Index of the solution to be considered.

Returns

Error estimate.

Return type

float

Notes

This is the error estimate used internally by the eigensolver. The actual error bound can be computed with [`computeError\(\)`](#).

See also

[*computeError*](#), [*EPSGetErrorEstimate*](#)

`:sources:`Source code at slepc4py/SLEPc/EPS.pyx:2186 <slepc4py/SLEPc/EPS.pyx#L2186>``

getExtraction()

Get the extraction type used by the EPS object.

Not collective.

Returns

The method of extraction.

Return type

[*Extraction*](#)

See also

[*setExtraction*](#), [*EPSGetExtraction*](#)

`:sources:`Source code at slepc4py/SLEPc/EPS.pyx:770 <slepc4py/SLEPc/EPS.pyx#L770>``

getGDBOrth()

Get the orthogonalization used in the search subspace.

Not collective.

Get the orthogonalization used in the search subspace in case of generalized Hermitian problems.

Returns

Whether to B-orthogonalize the search subspace.

Return type

[*bool*](#)

See also

[*setGDBOrth*](#), [*EPSGDGetBOrth*](#)

`:sources:`Source code at slepc4py/SLEPc/EPS.pyx:3336 <slepc4py/SLEPc/EPS.pyx#L3336>``

getGDBlockSize()

Get the number of vectors to be added to the searching space.

Not collective.

Get the number of vectors to be added to the searching space in every iteration.

Returns

The number of vectors added to the search space in every iteration.

Return type

[*int*](#)

See also

[*setGDBlockSize*](#), [*EPSCGGetBlockSize*](#)

:sources: [Source code at slepc4py/SLEPc/EPSC.pyx:3195 <slepc4py/SLEPc/EPSC.pyx#L3195>](#)

getGDDoubleExpansion()

Get a flag indicating whether the double expansion variant is active.

Not collective.

Get a flag indicating whether the double expansion variant has been activated or not.

Returns

True if using double expansion.

Return type

`bool`

See also

[*setGDDoubleExpansion*](#), [*EPSCGGetDoubleExpansion*](#)

:sources: [Source code at slepc4py/SLEPc/EPSC.pyx:3384 <slepc4py/SLEPc/EPSC.pyx#L3384>](#)

getGDInitialSize()

Get the initial size of the searching space.

Not collective.

Returns

The number of vectors of the initial searching subspace.

Return type

`int`

See also

[*setGDInitialSize*](#), [*EPSCGGetInitialSize*](#)

:sources: [Source code at slepc4py/SLEPc/EPSC.pyx:3296 <slepc4py/SLEPc/EPSC.pyx#L3296>](#)

getGDKrylovStart()

Get a flag indicating if the search subspace is started with a Krylov basis.

Not collective.

Returns

True if starting the search subspace with a Krylov basis.

Return type

`bool`

See also

`setGDKrylovStart`, `EPSCGDGetKrylovStart`

:sources: `Source code at slepc4py/SLEPc/EPSC.pyx:3155 <slepc4py/SLEPc/EPSC.pyx#L3155>`

getGDRestart()

Get the number of vectors of the search space after restart.

Not collective.

Get the number of vectors of the search space after restart and the number of vectors saved from the previous iteration.

Returns

- `minv (int)` – The number of vectors of the search subspace after restart.
- `plusk (int)` – The number of vectors saved from the previous iteration.

Return type

`tuple[int, int]`

See also

`setGDRestart`, `EPSCGDGetRestart`

:sources: `Source code at slepc4py/SLEPc/EPSC.pyx:3243 <slepc4py/SLEPc/EPSC.pyx#L3243>`

getInterval()

Get the computational interval for spectrum slicing.

Not collective.

Returns

- `inta (float)` – The left end of the interval.
- `intb (float)` – The right end of the interval.

Return type

`tuple[float, float]`

Notes

If the interval was not set by the user, then zeros are returned.

See also

`setInterval`, `EPSCGetInterval`

:sources: `Source code at slepc4py/SLEPc/EPSC.pyx:974 <slepc4py/SLEPc/EPSC.pyx#L974>`

getInvariantSubspace()

Get an orthonormal basis of the computed invariant subspace.

Collective.

Returns

Basis of the invariant subspace.

Return type

list of `petsc4py.PETSc.Vec`

Notes

This function should be called after `solve()` has finished.

The returned vectors span an invariant subspace associated with the computed eigenvalues. An invariant subspace X of A satisfies $Ax \in X$, for all $x \in X$ (a similar definition applies for generalized eigenproblems).

See also

`getEigenpair`, `getConverged`, `solve`, `EPSGetInvariantSubspace`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2142 <slepc4py/SLEPc/EPs.pyx#L2142>`

getIterationNumber()

Get the current iteration number.

Not collective.

If the call to `solve()` is complete, then it returns the number of iterations carried out by the solution method.

Returns

Iteration number.

Return type

`int`

See also

`getConvergedReason`, `setTolerances`, `EPSGetIterationNumber`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:1921 <slepc4py/SLEPc/EPs.pyx#L1921>`

getJDBOrth()

Get the orthogonalization used in the search subspace.

Not collective.

Get the orthogonalization used in the search subspace in case of generalized Hermitian problems.

Returns

Whether to B-orthogonalize the search subspace.

Return type

`bool`

See also

[*setJDBOrth*](#), [*EPSJDGetBOrth*](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3695 <slepc4py/SLEPc/EPS.pyx#L3695>](#)

getJDBlockSize()

Get the number of vectors to be added to the searching space.

Not collective.

Get the number of vectors to be added to the searching space in every iteration.

Returns

The number of vectors added to the search space in every iteration.

Return type

`int`

See also

[*setJDBlockSize*](#), [*EPSJDGetBlockSize*](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3466 <slepc4py/SLEPc/EPS.pyx#L3466>](#)

getJDConstCorrectionTol()

Get the flag indicating if the dynamic stopping is being used.

Not collective.

Returns

True if the dynamic stopping criterion is not being used.

Return type

`bool`

See also

[*setJDConstCorrectionTol*](#), [*EPSJDGetConstCorrectionTol*](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3655 <slepc4py/SLEPc/EPS.pyx#L3655>](#)

getJDFix()

Get the threshold for changing the target in the correction equation.

Not collective.

Returns

The threshold for changing the target.

Return type

`float`

See also

[`setJDFix`](#), [`EPSJDGetFix`](#)

:sources: [`Source code at slepc4py/SLEPc/EPS.pyx:3610 <slepc4py/SLEPc/EPS.pyx#L3610>`](#)

getJDInitialSize()

Get the initial size of the searching space.

Not collective.

Returns

The number of vectors of the initial searching subspace.

Return type

`int`

See also

[`setJDInitialSize`](#), [`EPSJDGetInitialSize`](#)

:sources: [`Source code at slepc4py/SLEPc/EPS.pyx:3567 <slepc4py/SLEPc/EPS.pyx#L3567>`](#)

getJDKrylovStart()

Get a flag indicating if the search subspace is started with a Krylov basis.

Not collective.

Returns

True if starting the search subspace with a Krylov basis.

Return type

`bool`

See also

[`setJDKrylovStart`](#), [`EPSJDGetKrylovStart`](#)

:sources: [`Source code at slepc4py/SLEPc/EPS.pyx:3426 <slepc4py/SLEPc/EPS.pyx#L3426>`](#)

getJDRestart()

Get the number of vectors of the search space after restart.

Not collective.

Get the number of vectors of the search space after restart and the number of vectors saved from the previous iteration.

Returns

- **minv** (`int`) – The number of vectors of the search subspace after restart.
- **plusk** (`int`) – The number of vectors saved from the previous iteration.

Return type

`tuple[int, int]`

See also

[*setJDRestart*](#), [*EPSJDGetRestart*](#)

:sources: [`Source code at slepc4py/SLEPc/EPS.pyx:3514 <slepc4py/SLEPc/EPS.pyx#L3514>`](#)

getKrylovSchurBSEType()

Get the method used for BSE structured eigenproblems (Krylov-Schur).

Not collective.

Returns

The BSE method.

Return type

KrylovSchurBSEType

See also

[*setKrylovSchurBSEType*](#), [*EPSKrylovSchurGetBSEType*](#)

:sources: [`Source code at slepc4py/SLEPc/EPS.pyx:2501 <slepc4py/SLEPc/EPS.pyx#L2501>`](#)

getKrylovSchurDetectZeros()

Get the flag that enforces zero detection in spectrum slicing.

Not collective.

Returns

The zero detection flag.

Return type

bool

See also

[*setKrylovSchurDetectZeros*](#), [*EPSKrylovSchurGetDetectZeros*](#)

:sources: [`Source code at slepc4py/SLEPc/EPS.pyx:2695 <slepc4py/SLEPc/EPS.pyx#L2695>`](#)

getKrylovSchurDimensions()

Get the dimensions used for each subsolve step (spectrum slicing).

Not collective.

Returns

- **nev** (*int*) – Number of eigenvalues to compute.
- **ncv** (*int*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int*) – Maximum dimension allowed for the projected problem.

Return type

tuple[*int*, *int*, *int*]

See also

[setKrylovSchurDimensions](#), [EPKrylovSchurGetDimensions](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2754 <slepc4py/SLEPc/EPS.pyx#L2754>`

getKrylovSchurInertias()

Get the values of the shifts and their corresponding inertias.

Not collective.

Get the values of the shifts and their corresponding inertias in case of doing spectrum slicing for a computational interval.

Returns

- **shifts** ([ArrayReal](#)) – The values of the shifts used internally in the solver.
- **inertias** ([ArrayInt](#)) – The values of the inertia in each shift.

Return type

[tuple](#)[[ArrayReal](#), [ArrayInt](#)]

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [setInterval\(\)](#) and [SINVERT](#) is set.

If called after [solve\(\)](#), all shifts used internally by the solver are returned (including both endpoints and any intermediate ones). If called before [solve\(\)](#) and after [setUp\(\)](#) then only the information of the endpoints of subintervals is available.

See also

[setInterval](#), [setKrylovSchurSubintervals](#), [EPKrylovSchurGetInertias](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3052 <slepc4py/SLEPc/EPS.pyx#L3052>`

getKrylovSchurKSP()

Get the linear solver object associated with the internal [EPS](#) object.

Collective.

Get the linear solver object associated with the internal [EPS](#) object in case of doing spectrum slicing for a computational interval.

Returns

The linear solver object.

Return type

[petsc4py.PETSc.KSP](#)

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [setInterval\(\)](#) and [SINVERT](#) is set.

When invoked to compute all eigenvalues in an interval with spectrum slicing, *KRYLOV SCHUR* creates another *EPS* object internally that is used to compute eigenvalues by chunks near selected shifts. This function allows access to the KSP object associated to this internal *EPS* object.

In case of having more than one partition, the returned KSP will be different in MPI processes belonging to different partitions. Hence, if required, *setKrylovSchurPartitions()* must be called BEFORE this function.

See also

setInterval, *setKrylovSchurPartitions*, *EPSPKrylovSchurGetKSP*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:3096 <slepc4py/SLEPc/EPs.pyx#L3096>`

getKrylovSchurLocking()

Get the locking flag used in the Krylov-Schur method.

Not collective.

Returns

The locking flag.

Return type

`bool`

See also

setKrylovSchurLocking, *EPSPKrylovSchurGetLocking*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2588 <slepc4py/SLEPc/EPs.pyx#L2588>`

getKrylovSchurPartitions()

Get the number of partitions of the communicator (spectrum slicing).

Not collective.

Returns

The number of partitions.

Return type

`int`

See also

setKrylovSchurPartitions, *EPSPKrylovSchurGetPartitions*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2642 <slepc4py/SLEPc/EPs.pyx#L2642>`

getKrylovSchurRestart()

Get the restart parameter used in the Krylov-Schur method.

Not collective.

Returns

The number of vectors to be kept at restart.

Return type
float

See also

[setKrylovSchurRestart](#), [EPKrylovSchurGetRestart](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2544 <slepc4py/SLEPc/EPs.pyx#L2544>`

getKrylovSchurSubcommInfo()

Get information related to the case of doing spectrum slicing.

Collective on the subcommunicator.

Get information related to the case of doing spectrum slicing for a computational interval with multiple communicators.

Returns

- **k** (`int`) – Index of the subinterval for the calling process.
- **n** (`int`) – Number of eigenvalues found in the k-th subinterval.
- **v** (`petsc4py.PETSc.Vec`) – A vector owned by processes in the subcommunicator with dimensions compatible for locally computed eigenvectors.

Return type

`tuple[int, int, Vec]`

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [setInterval\(\)](#) and *SINVERT* is set.

See also

[getKrylovSchurSubcommPairs](#), [EPKrylovSchurGetSubcommInfo](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2779 <slepc4py/SLEPc/EPs.pyx#L2779>`

getKrylovSchurSubcommMats()

Get the eigenproblem matrices stored in the subcommunicator.

Collective on the subcommunicator.

Get the eigenproblem matrices stored internally in the subcommunicator to which the calling process belongs.

Returns

- **A** (`petsc4py.PETSc.Mat`) – The matrix associated with the eigensystem.
- **B** (`petsc4py.PETSc.Mat`) – The second matrix in the case of generalized eigenproblems.

Return type

`tuple[Mat, Mat] | tuple[Mat, None]`

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with `setInterval()` and `SINVERT` is set. And is relevant only when the number of partitions (`setKrylovSchurPartitions()`) is larger than one.

This is the analog of `getOperators()`, but returns the matrices distributed differently (in the subcommunicator rather than in the parent communicator).

These matrices should not be modified by the user.

See also

`setInterval`, `setKrylovSchurPartitions`, `EPSKrylovSchurGetSubcommMats`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2857 <slepc4py/SLEPc/EPs.pyx#L2857>`

getKrylovSchurSubcommPairs(*i*, *v=None*)

Get the *i*-th eigenpair stored in the multi-communicator of the process.

Collective on the subcommunicator (if *v* is given).

Get the *i*-th eigenpair stored internally in the multi-communicator to which the calling process belongs.

Parameters

- **i** (*int*) – Index of the solution to be obtained.
- **v** (*Vec* | *None*) – Placeholder for the returned eigenvector.

Returns

The computed eigenvalue.

Return type

Scalar

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with `setInterval()` and `SINVERT` is set. And is relevant only when the number of partitions (`setKrylovSchurPartitions()`) is larger than one.

Argument *v* must be a valid `Vec` object, created by calling `getKrylovSchurSubcommInfo()`.

The index *i* should be a value between 0 and *n*-1, where *n* is the number of vectors in the local subinterval, see `getKrylovSchurSubcommInfo()`.

See also

`getKrylovSchurSubcommMats`, `EPSKrylovSchurGetSubcommPairs`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2813 <slepc4py/SLEPc/EPs.pyx#L2813>`

getKrylovSchurSubintervals()

Get the points that delimit the subintervals.

Not collective.

Get the points that delimit the subintervals used in spectrum slicing with several partitions.

Returns

Real values specifying subintervals.

Return type

ArrayReal

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with *setInterval()* and *SINVERT* is set.

If the user passed values with *setKrylovSchurSubintervals()*, then the same values are returned here. Otherwise, the values computed internally are obtained.

See also

setKrylovSchurSubintervals, *EPKrylovSchurGetSubintervals*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:3014 <slepc4py/SLEPc/EPs.pyx#L3014>`

getLOBPCGBlockSize()

Get the block size used in the LOBPCG method.

Not collective.

Returns

The block size.

Return type

int

See also

setLOBPCGBlockSize, *EPsLOBPCGGetBlockSize*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:3779 <slepc4py/SLEPc/EPs.pyx#L3779>`

getLOBPCGLocking()

Get the locking flag used in the LOBPCG method.

Not collective.

Returns

The locking flag.

Return type

bool

See also

setLOBPCGLocking, *EPsLOBPCGGetLocking*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:3866 <slepc4py/SLEPc/EPs.pyx#L3866>`

getLOBPCGRestart()

Get the restart parameter used in the LOBPCG method.

Not collective.

Returns

The restart parameter.

Return type

`float`

See also

`setLOBPCGRestart`, `EPsLOBPCGGetRestart`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:3823 <slepc4py/SLEPc/EPs.pyx#L3823>`

getLanczosReorthogType()

Get the type of reorthogonalization used during the Lanczos iteration.

Not collective.

Returns

The type of reorthogonalization.

Return type

`LanczosReorthogType`

See also

`setLanczosReorthogType`, `EPsLanczosGetReorthog`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2455 <slepc4py/SLEPc/EPs.pyx#L2455>`

getLeftEigenvector(*i*, *Wr*=None, *Wi*=None)

Get the *i*-th left eigenvector as computed by `solve()`.

Collective.

Parameters

- ***i*** (`int`) – Index of the solution to be obtained.
- ***Wr*** (`Vec` / `None`) – Placeholder for the returned left eigenvector (real part).
- ***Wi*** (`Vec` / `None`) – Placeholder for the returned left eigenvector (imaginary part).

Return type

`None`

Notes

The index *i* should be a value between 0 and `nconv-1` (see `getConverged()`). Eigensolutions are indexed according to the ordering criterion established with `setWhichEigenpairs()`.

Left eigenvectors are available only if the twosided flag was set with `setTwoSided()`.

See also

getConverged, setWhichEigenpairs, getEigenpair, EPSGetLeftEigenvector

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2060 <slepc4py/SLEPc/EPS.pyx#L2060>`

getLyapIIRanks()

Get the rank values used for the Lyapunov step.

Not collective.

Returns

- **rkc** (`int`) – The compressed rank.
- **rkl** (`int`) – The Lyapunov rank.

Return type

`tuple[int, int]`

See also

setLyapIIRanks, EPSLyapIIGetRanks

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3917 <slepc4py/SLEPc/EPS.pyx#L3917>`

getMonitor()

Get the list of monitor functions.

Not collective.

Returns

The list of monitor functions.

Return type

EPSPMonitorFunction

See also

setMonitor

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1845 <slepc4py/SLEPc/EPS.pyx#L1845>`

getOperators()

Get the matrices associated with the eigenvalue problem.

Collective.

Returns

- **A** (`petsc4py.PETSc.Mat`) – The matrix associated with the eigensystem.
- **B** (`petsc4py.PETSc.Mat`) – The second matrix in the case of generalized eigenproblems.

Return type

`tuple[Mat, Mat] | tuple[Mat, None]`

See also

setOperators, *EPSGetOperators*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:1520 <slepc4py/SLEPc/EPS.pyx#L1520>](#)

getOptionsPrefix()

Get the prefix used for searching for all EPS options in the database.

Not collective.

Returns

The prefix string set for this EPS object.

Return type

str

See also

setOptionsPrefix, *appendOptionsPrefix*, *EPSGetOptionsPrefix*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:465 <slepc4py/SLEPc/EPS.pyx#L465>](#)

getPowerShiftType()

Get the type of shifts used during the power iteration.

Not collective.

Returns

The type of shift.

Return type

PowerShiftType

See also

setPowerShiftType, *EPSPowerGetShiftType*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2366 <slepc4py/SLEPc/EPS.pyx#L2366>](#)

getProblemType()

Get the problem type from the EPS object.

Not collective.

Returns

The problem type that was previously set.

Return type

ProblemType

See also

setProblemType, *EPSGetProblemType*

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:555 <slepc4py/SLEPc/EPS.pyx#L555>](#)

getPurify()

Get the flag indicating whether purification is activated or not.

Not collective.

Returns

Whether purification is activated or not.

Return type

`bool`

See also

`setPurify`, `EPSGetPurify`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1134 <slepc4py/SLEPc/EPS.pyx#L1134>`

getRG()

Get the region object associated to the eigensolver.

Not collective.

Returns

The region context.

Return type

`RG`

See also

`setRG`, `EPSGetRG`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1483 <slepc4py/SLEPc/EPS.pyx#L1483>`

getRQCGReset()

Get the reset parameter used in the RQCG method.

Not collective.

Returns

The number of iterations between resets.

Return type

`int`

See also

`setRQCGReset`, `EPSRQCGGetReset`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3742 <slepc4py/SLEPc/EPS.pyx#L3742>`

getST()

Get the spectral transformation object associated to the eigensolver.

Not collective.

Returns

The spectral transformation.

Return type

ST

See also

setST, *EPSGetST*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1372 <slepc4py/SLEPc/EPs.pyx#L1372>](#)

getStoppingTest()

Get the stopping test function.

Not collective.

Returns

The stopping test function.

Return type

EPsStoppingFunction

See also

setStoppingTest

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1710 <slepc4py/SLEPc/EPs.pyx#L1710>](#)

getTarget()

Get the value of the target.

Not collective.

Returns

The value of the target.

Return type

Scalar

Notes

If the target was not set by the user, then zero is returned.

See also

setTarget, *EPSGetTarget*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:924 <slepc4py/SLEPc/EPs.pyx#L924>](#)

getThreshold()

Get the threshold used in the threshold stopping test.

Not collective.

Returns

- **thres** (`float`) – The threshold.
- **rel** (`bool`) – Whether the threshold is relative or not.

Return type

`tuple[float, bool]`

See also

`setThreshold`, `EPSGetThreshold`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:873 <slepc4py/SLEPc/EPS.pyx#L873>`

getTolerances()

Get the tolerance and max. iter. count used for convergence tests.

Not collective.

Get the tolerance and iteration limit used by the default EPS convergence tests.

Returns

- **tol** (`float`) – The convergence tolerance.
- **max_it** (`int`) – The maximum number of iterations.

Return type

`tuple[float, int]`

See also

`setTolerances`, `EPSGetTolerances`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1034 <slepc4py/SLEPc/EPS.pyx#L1034>`

getTrackAll()

Get the flag indicating if all residual norms must be computed or not.

Not collective.

Returns

Whether the solver computes all residuals or not.

Return type

`bool`

See also

`setTrackAll`, `EPSGetTrackAll`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1255 <slepc4py/SLEPc/EPS.pyx#L1255>`

getTrueResidual()

Get the flag indicating if true residual must be computed explicitly.

Not collective.

Returns

Whether the solver computes true residuals or not.

Return type

`bool`

See also

`setTrueResidual`, `EPSGetTrueResidual`

:sources: ``Source code at slepc4py/SLEPc/EPS.pyx:1218 <slepc4py/SLEPc/EPS.pyx#L1218>``

getTwoSided()

Get the flag indicating if a two-sided variant of the algorithm is being used.

Not collective.

Returns

Whether the two-sided variant is to be used or not.

Return type

`bool`

See also

`setTwoSided`, `EPSGetTwoSided`

:sources: ``Source code at slepc4py/SLEPc/EPS.pyx:1087 <slepc4py/SLEPc/EPS.pyx#L1087>``

getType()

Get the EPS type of this object.

Not collective.

Returns

The solver currently being used.

Return type

`str`

See also

`setType`, `EPSGetType`

:sources: ``Source code at slepc4py/SLEPc/EPS.pyx:446 <slepc4py/SLEPc/EPS.pyx#L446>``

getWhichEigenpairs()

Get which portion of the spectrum is to be sought.

Not collective.

Returns

The portion of the spectrum to be sought by the solver.

Return type

Which

See also

setWhichEigenpairs, *EPSGetWhichEigenpairs*

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:817 <slepc4py/SLEPc/EPS.pyx#L817>`

isGeneralized()

Tell if the EPS object corresponds to a generalized eigenproblem.

Not collective.

Returns

True if the problem is generalized.

Return type

bool

See also

isHermitian, *isPositive*, *isStructured*, *EPSIsGeneralized*

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:610 <slepc4py/SLEPc/EPS.pyx#L610>`

isHermitian()

Tell if the EPS object corresponds to a Hermitian eigenproblem.

Not collective.

Returns

True if the problem is Hermitian.

Return type

bool

See also

isGeneralized, *isPositive*, *isStructured*, *EPSIsHermitian*

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:629 <slepc4py/SLEPc/EPS.pyx#L629>`

isPositive()

Eigenproblem requiring a positive (semi-) definite matrix B .

Not collective.

Tell if the EPS corresponds to an eigenproblem requiring a positive (semi-) definite matrix B .

Returns

True if the problem is positive (semi-) definite.

Return type

bool

See also

isGeneralized, isHermitian, isStructured, EPSIsPositive

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:648 <slepc4py/SLEPc/EPS.pyx#L648>`

isStructured()

Tell if the EPS object corresponds to a structured eigenvalue problem.

Not collective.

Returns

True if the problem is structured.

Return type

bool

Notes

The result will be True if the problem type has been set to some structured type such as *BSE*. This is independent of whether the input matrix has been built with a certain structure with a helper function.

See also

isGeneralized, isHermitian, isPositive, EPSIsStructured

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:670 <slepc4py/SLEPc/EPS.pyx#L670>`

reset()

Reset the EPS object.

Collective.

See also

EPSReset

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:386 <slepc4py/SLEPc/EPS.pyx#L386>`

Return type

None

setArbitrarySelection(arbitrary, args=None, kargs=None)

Set an arbitrary selection criterion function.

Logically collective.

Set a function to look for eigenvalues according to an arbitrary selection criterion. This criterion can be based on a computation involving the current eigenvector approximation.

See also

getArbitrarySelection, EPSSetArbitrarySelection

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1727 <slepc4py/SLEPc/EPS.pyx#L1727>`

Parameters

- **arbitrary** ([EPsArbitraryFunction](#) / *None*)
- **args** (*tuple*[*Any*, ...] / *None*)
- **kargs** (*dict*[*str*, *Any*] / *None*)

Return type

None

setArnoldiDelayed(*delayed*)

Set (toggle) delayed reorthogonalization in the Arnoldi iteration.

Logically collective.

Parameters

delayed (*bool*) – True if delayed reorthogonalization is to be used.

Return type

None

Notes

This call is only relevant if the type was set to [EPS.Type.ARNOLDI](#) with [setType\(\)](#).

Delayed reorthogonalization is an aggressive optimization for the Arnoldi eigensolver than may provide better scalability, but sometimes makes the solver converge more slowly compared to the default algorithm.

See also

[getArnoldiDelayed](#), [EPSArnoldiSetDelayed](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2385 <slepc4py/SLEPc/EPS.pyx#L2385>](#)

setBV(*bv*)

Set a basis vectors object associated to the eigensolver.

Collective.

Parameters

bv (*BV*) – The basis vectors context.

Return type

None

See also

[getBV](#), [EPSSetBV](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:1429 <slepc4py/SLEPc/EPS.pyx#L1429>](#)

setBalance(*balance=None*, *iterations=None*, *cutoff=None*)

Set the balancing technique to be used by the eigensolver.

Logically collective.

Parameters

- **balance** (*Balance* / *None*) – The balancing method.

- **iterations** (*int* / *None*) – Number of iterations of the balancing algorithm.
- **cutoff** (*float* / *None*) – Cutoff value.

Return type

None

Notes

When balancing is enabled, the solver works implicitly with matrix DAD^{-1} , where D is an appropriate diagonal matrix. This improves the accuracy of the computed results in some cases.

Balancing makes sense only for non-Hermitian problems when the required precision is high (i.e., with a small tolerance).

By default, balancing is disabled. The two-sided method is much more effective than the one-sided counterpart, but it requires the system matrices to have the `Mat.multTranspose()` operation defined.

The parameter `iterations` is the number of iterations performed by the method. The `cutoff` value is used only in the two-side variant.

See also

`setBalance`, `EPSGetBalance`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:720 <slepc4py/SLEPc/EPS.pyx#L720>`

setCISSExtraction(*extraction*)

Set the extraction technique used in the CISS solver.

Logically collective.

Parameters

extraction (`CISSExtraction`) – The extraction technique.

Return type

None

See also

`getCISSExtraction`, `EPSCISSSetExtraction`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3941 <slepc4py/SLEPc/EPS.pyx#L3941>`

setCISSQuadRule(*quad*)

Set the quadrature rule used in the CISS solver.

Logically collective.

Parameters

quad (`CISSQuadRule`) – The quadrature rule.

Return type

None

See also

[getCISSQuadRule](#), [EPSCISSSetQuadRule](#)

:sources: ``Source code at slepc4py/SLEPc/EPs.pyx:3978 <slepc4py/SLEPc/EPs.pyx#L3978>``

setCISSRefinement(*inner=None, blsize=None*)

Set the values of various refinement parameters in the CISS solver.

Logically collective.

Parameters

- **inner** (*int* / *None*) – Number of iterative refinement iterations (inner loop).
- **blsize** (*int* / *None*) – Number of iterative refinement iterations (blocksize loop).

Return type

None

See also

[getCISSRefinement](#), [EPSCISSSetRefinement](#)

:sources: ``Source code at slepc4py/SLEPc/EPs.pyx:4148 <slepc4py/SLEPc/EPs.pyx#L4148>``

setCISSSizes(*ip=None, bs=None, ms=None, npart=None, bsmax=None, realmats=False*)

Set the values of various size parameters in the CISS solver.

Logically collective.

Parameters

- **ip** (*int* / *None*) – Number of integration points.
- **bs** (*int* / *None*) – Block size.
- **ms** (*int* / *None*) – Moment size.
- **npart** (*int* / *None*) – Number of partitions when splitting the communicator.
- **bsmax** (*int* / *None*) – Maximum block size.
- **realmats** (*bool*) – True if A and B are real.

Return type

None

Notes

The default number of partitions is 1. This means the internal `petsc4py.PETSc.KSP` object is shared among all processes of the `EPS` communicator. Otherwise, the communicator is split into `npart` communicators, so that `npart` `petsc4py.PETSc.KSP` solves proceed simultaneously.

See also

[getCISSSizes](#), [setCISSThreshold](#), [setCISSRefinement](#), [EPSCISSSetSizes](#)

:sources: ``Source code at slepc4py/SLEPc/EPs.pyx:4015 <slepc4py/SLEPc/EPs.pyx#L4015>``

setCISSThreshold(*delta=None, spur=None*)

Set the values of various threshold parameters in the CISS solver.

Logically collective.

Parameters

- **delta** (*float* / *None*) – Threshold for numerical rank.
- **spur** (*float* / *None*) – Spurious threshold (to discard spurious eigenpairs).

Return type

None

See also

getCISSThreshold, *EPSCISSSetThreshold*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4103 <slepc4py/SLEPc/EPs.pyx#L4103>`

setCISSUseST(*usest*)

Set a flag indicating that the CISS solver will use the *ST* object.

Logically collective.

Parameters

usest (*bool*) – Whether to use the *ST* object or not.

Return type

None

Notes

When this option is set, the linear solves can be configured by setting options for the `petsc4py.PETSc.KSP` object obtained with `ST.getKSP()`. Otherwise, several `petsc4py.PETSc.KSP` objects are created, which can be accessed with `getCISSKSPs()`.

The default is to use the *ST*, unless several partitions have been specified, see `setCISSSizes()`.

See also

getCISSUseST, *getCISSKSPs*, *setCISSSizes*, *EPSCISSSetUseST*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4193 <slepc4py/SLEPc/EPs.pyx#L4193>`

setConvergenceTest(*conv*)

Set how to compute the error estimate used in the convergence test.

Logically collective.

Parameters

conv (*Conv*) – The method used to compute the error estimate used in the convergence test.

Return type

None

See also

[`getConvergenceTest`](#), [`EPSSetConvergenceTest`](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:1199 <slepc4py/SLEPc/EPs.pyx#L1199>`

setDS(*ds*)

Set a direct solver object associated to the eigensolver.

Collective.

Parameters

ds (DS) – The direct solver context.

Return type

None

See also

[`getDS`](#), [`EPSSetDS`](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:1466 <slepc4py/SLEPc/EPs.pyx#L1466>`

setDeflationSpace(*space*)

Set vectors to form a basis of the deflation space.

Collective.

Parameters

space (*Vec* | *list*[*Vec*]) – Set of basis vectors of the deflation space.

Return type

None

Notes

When a deflation space is given, the eigensolver seeks the eigensolution in the restriction of the problem to the orthogonal complement of this space. This can be used for instance in the case that an invariant subspace is known beforehand (such as the nullspace of the matrix).

These vectors do not persist from one [`solve\(\)`](#) call to the other, so the deflation space should be set every time.

The vectors do not need to be mutually orthonormal, since they are explicitly orthonormalized internally.

See also

[`setInitialSpace`](#), [`EPSSetDeflationSpace`](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:1578 <slepc4py/SLEPc/EPs.pyx#L1578>`

setDimensions(*nev=None*, *ncv=None*, *mpd=None*)

Set number of eigenvalues to compute and the dimension of the subspace.

Logically collective.

Parameters

- **nev** (*int* / *None*) – Number of eigenvalues to compute.
- **ncv** (*int* / *None*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int* / *None*) – Maximum dimension allowed for the projected problem.

Return type

None

Notes

Use *DETERMINE* for **ncv** and **mpd** to assign a reasonably good value, which is dependent on the solution method.

The parameters **ncv** and **mpd** are intimately related, so that the user is advised to set one of them at most. Normal usage is the following:

- In cases where **nev** is small, the user sets **ncv** (a reasonable default is $2 * \text{nev}$).
- In cases where **nev** is large, the user sets **mpd**.

The value of **ncv** should always be between **nev** and $(\text{nev} + \text{mpd})$, typically $\text{ncv} = \text{nev} + \text{mpd}$. If **nev** is not too large, $\text{mpd} = \text{nev}$ is a reasonable choice, otherwise a smaller value should be used.

When computing all eigenvalues in an interval, see *setInterval()*, these parameters lose relevance, and tuning must be done with *setKrylovSchurDimensions()*.

See also

getDimensions, *setKrylovSchurDimensions*, *EPSSetDimensions*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1317 <slepc4py/SLEPc/EPs.pyx#L1317>](#)

setEigenvalueComparison(*comparison*, *args=None*, *kargs=None*)

Set an eigenvalue comparison function.

Logically collective.

Notes

This eigenvalue comparison function is used when *setWhichEigenpairs()* is set to *EPS.Which.USER*.

See also

getEigenvalueComparison, *EPSSetEigenvalueComparison*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1773 <slepc4py/SLEPc/EPs.pyx#L1773>](#)

Parameters

- **comparison** (*EPSEigenvalueComparison* / *None*)
- **args** (*tuple*[*Any*, ...] / *None*)
- **kargs** (*dict*[*str*, *Any*] / *None*)

Return type

None

setExtraction(*extraction*)

Set the extraction type used by the eigensolver.

Logically collective.

Parameters

extraction ([Extraction](#)) – The extraction method to be used by the solver.

Return type

None

Notes

Not all eigensolvers support all types of extraction.

By default, a standard Rayleigh-Ritz extraction is used. Other extractions may be useful when computing interior eigenvalues.

Harmonic-type extractions are used in combination with a *target*, see [setTarget\(\)](#).

See also

[getExtraction](#), [setTarget](#), [EPSSetExtraction](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:789 <slepc4py/SLEPc/EPs.pyx#L789>](#)

setFromOptions()

Set EPS options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before [setUp\(\)](#) if the user is to be allowed to set the solver type.

See also

[setOptionsPrefix](#), [EPSSetFromOptions](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:534 <slepc4py/SLEPc/EPs.pyx#L534>](#)

Return type

None

setGDBOrth(*borth*)

Set the orthogonalization that will be used in the search subspace.

Logically collective.

Set the orthogonalization that will be used in the search subspace in case of generalized Hermitian problems.

Parameters

borth ([bool](#)) – Whether to B-orthogonalize the search subspace.

Return type

None

See also

[`getGDBOrth`](#), [`EPSCGSetBOrth`](#)

:sources: [Source code at slepc4py/SLEPc/EPSCG.pyx:3315 <slepc4py/SLEPc/EPSCG.pyx#L3315>](#)

setGDBlockSize(*bs*)

Set the number of vectors to be added to the searching space.

Logically collective.

Set the number of vectors to be added to the searching space in every iteration.

Parameters

bs (*int*) – The number of vectors added to the search space in every iteration.

Return type

None

See also

[`getGDBlockSize`](#), [`EPSCGSetBlockSize`](#)

:sources: [Source code at slepc4py/SLEPc/EPSCG.pyx:3174 <slepc4py/SLEPc/EPSCG.pyx#L3174>](#)

setGDDoubleExpansion(*doubleexp*)

Set that the search subspace is expanded with double expansion.

Logically collective.

Parameters

doubleexp (*bool*) – True if using double expansion.

Return type

None

Notes

In the double expansion variant the search subspace is expanded with $K[Ax, Bx]$ (double expansion) instead of the classic Kr , where K is the preconditioner, x the selected approximate eigenvector and r its associated residual vector.

See also

[`getGDDoubleExpansion`](#), [`EPSCGSetDoubleExpansion`](#)

:sources: [Source code at slepc4py/SLEPc/EPSCG.pyx:3358 <slepc4py/SLEPc/EPSCG.pyx#L3358>](#)

setGDInitialSize(*initialsize*)

Set the initial size of the searching space.

Logically collective.

Parameters

initialsize (*int*) – The number of vectors of the initial searching subspace.

Return type

None

Notes

If the flag in `setGDKrylovStart()` is set to `False` and the user provides vectors with `setInitialSpace()`, up to `initialsize` vectors will be used; and if the provided vectors are not enough, the solver completes the subspace with random vectors. In case the `setGDKrylovStart()` flag is `True`, the solver gets the first vector provided by the user or, if not available, a random vector, and expands the Krylov basis up to `initialsize` vectors.

See also

`setGDKrylovStart`, `getGDInitialSize`, `EPSCGSetInitialSize`

:sources: `Source code at slepc4py/SLEPc/EPSCG.pyx:3268 <slepc4py/SLEPc/EPSCG.pyx#L3268>`

setGDKrylovStart (*krylovstart=True*)

Set (toggle) starting the search subspace with a Krylov basis.

Logically collective.

Parameters

krylovstart (*bool*) – True if starting the search subspace with a Krylov basis.

Return type

None

See also

`setGDInitialSize`, `getGDKrylovStart`, `EPSCGSetKrylovStart`

:sources: `Source code at slepc4py/SLEPc/EPSCG.pyx:3137 <slepc4py/SLEPc/EPSCG.pyx#L3137>`

setGDRestart (*minv=None, plusk=None*)

Set the number of vectors of the search space after restart.

Logically collective.

Set the number of vectors of the search space after restart and the number of vectors saved from the previous iteration.

Parameters

- **minv** (*int*) – The number of vectors of the search subspace after restart.
- **plusk** (*int*) – The number of vectors saved from the previous iteration.

Return type

None

See also

`getGDRestart`, `EPSCGSetRestart`

:sources: `Source code at slepc4py/SLEPc/EPSCG.pyx:3217 <slepc4py/SLEPc/EPSCG.pyx#L3217>`

setInitialSpace(*space*)

Set the initial space from which the eigensolver starts to iterate.

Collective.

Parameters

space (*Vec* | *list*[*Vec*]) – Set of basis vectors of the initial space.

Return type

None

Notes

Some solvers start to iterate on a single vector (initial vector). In that case, only the first vector is taken into account and the other vectors are ignored. But other solvers such as [SUBSPACE](#) are able to make use of the whole initial subspace as an initial guess.

These vectors do not persist from one [solve\(\)](#) call to the other, so the initial space should be set every time.

The vectors do not need to be mutually orthonormal, since they are explicitly orthonormalized internally.

Common usage of this function is when the user can provide a rough approximation of the wanted eigenspace. Then, convergence may be faster.

See also

[setDeflationSpace](#), [setLeftInitialSpace](#), [EPSSetInitialSpace](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1616](#) <[slepc4py/SLEPc/EPs.pyx#L1616](#)>

setInterval(*inta*, *intb*)

Set the computational interval for spectrum slicing.

Logically collective.

Parameters

- **inta** (*float*) – The left end of the interval.
- **intb** (*float*) – The right end of the interval.

Return type

None

Notes

Spectrum slicing is a technique employed for computing all eigenvalues of symmetric eigenproblems in a given interval. This function provides the interval to be considered. It must be used in combination with [EPS.Which.ALL](#), see [setWhichEigenpairs\(\)](#).

A computational interval is also needed when using polynomial filters, see [STFILTER](#).

See also

[getInterval](#), [setWhichEigenpairs](#), [EPSSetInterval](#), [STFILTER](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1000](#) <[slepc4py/SLEPc/EPs.pyx#L1000](#)>

setJDBOrth(*borth*)

Set the orthogonalization that will be used in the search subspace.

Logically collective.

Set the orthogonalization that will be used in the search subspace in case of generalized Hermitian problems.

Parameters

borth (*bool*) – Whether to B-orthogonalize the search subspace.

Return type

None

See also

[getJDBOrth](#), [EPSJDSetBOrth](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3674 <slepc4py/SLEPc/EPS.pyx#L3674>`

setJDBlockSize(*bs*)

Set the number of vectors to be added to the searching space.

Logically collective.

Set the number of vectors to be added to the searching space in every iteration.

Parameters

bs (*int*) – The number of vectors added to the search space in every iteration.

Return type

None

See also

[getJDBlockSize](#), [EPSJDSetBlockSize](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3445 <slepc4py/SLEPc/EPS.pyx#L3445>`

setJDConstCorrectionTol(*constant*)

Deactivate the dynamic stopping criterion.

Logically collective.

Parameters

constant (*bool*) – If False, the `petsc4py.PETSc.KSP` relative tolerance is set to 0.5^{**i} .

Return type

None

Notes

If this flag is set to False, then the `petsc4py.PETSc.KSP` relative tolerance is dynamically set to 0.5^{**i} , where *i* is the number of *EPS* iterations since the last converged value. By the default, a constant tolerance is used.

See also

[`getJDConstCorrectionTol`](#), [`EPSJDSetConstCorrectionTol`](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3629 <slepc4py/SLEPc/EPS.pyx#L3629>](#)

setJDFix(*fix*)

Set the threshold for changing the target in the correction equation.

Logically collective.

Parameters

fix (*float*) – The threshold for changing the target.

Return type

None

Notes

The target in the correction equation is fixed at the first iterations. When the norm of the residual vector is lower than the **fix** value, the target is set to the corresponding eigenvalue.

See also

[`getJDFix`](#), [`EPSJDSetFix`](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3586 <slepc4py/SLEPc/EPS.pyx#L3586>](#)

setJDInitialSize(*initialsize*)

Set the initial size of the searching space.

Logically collective.

Parameters

initialsize (*int*) – The number of vectors of the initial searching subspace.

Return type

None

Notes

If the flag in [`setJDKrylovStart\(\)`](#) is set to `False` and the user provides vectors with [`setInitialSpace\(\)`](#), up to *initialsize* vectors will be used; and if the provided vectors are not enough, the solver completes the subspace with random vectors. In case the [`setJDKrylovStart\(\)`](#) flag is `True`, the solver gets the first vector provided by the user or, if not available, a random vector, and expands the Krylov basis up to *initialsize* vectors.

See also

[`setJDKrylovStart`](#), [`getJDInitialSize`](#), [`EPSJDSetInitialSize`](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3539 <slepc4py/SLEPc/EPS.pyx#L3539>](#)

setJDKrylovStart (*krylovstart=True*)

Set (toggle) starting the search subspace with a Krylov basis.

Logically collective.

Parameters

krylovstart (*bool*) – True if starting the search subspace with a Krylov basis.

Return type

None

See also

[setJDInitialSize](#), [getJDKrylovStart](#), [EPSJDSetKrylovStart](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3408 <slepc4py/SLEPc/EPS.pyx#L3408>](#)

setJDRestart (*minv=None, plusk=None*)

Set the number of vectors of the search space after restart.

Logically collective.

Set the number of vectors of the search space after restart and the number of vectors saved from the previous iteration.

Parameters

- **minv** (*int* / *None*) – The number of vectors of the search subspace after restart.
- **plusk** (*int* / *None*) – The number of vectors saved from the previous iteration.

Return type

None

See also

[getJDRestart](#), [EPSJDSetRestart](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:3488 <slepc4py/SLEPc/EPS.pyx#L3488>](#)

setKrylovSchurBSEType (*bse*)

Set the Krylov-Schur variant used for BSE structured eigenproblems.

Logically collective.

Parameters

bse ([KrylovSchurBSEType](#)) – The BSE method.

Return type

None

Notes

This call is only relevant if the type was set to [EPS.Type.KRYLOV SCHUR](#) with [setType\(\)](#) and the problem type to [EPS.ProblemType.BSE](#) with [setProblemType\(\)](#).

See also

[createMatBSE](#), [getKrylovSchurBSEType](#), [EPSKrylovSchurSetBSEType](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2477 <slepc4py/SLEPc/EPS.pyx#L2477>`

setKrylovSchurDetectZeros(*detect*)

Set the flag that enforces zero detection in spectrum slicing.

Logically collective.

Set a flag to enforce the detection of zeros during the factorizations throughout the spectrum slicing computation.

Parameters

detect (*bool*) – True if zeros must checked for.

Return type

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [setInterval\(\)](#) and *SINVERT* is set.

A zero in the factorization indicates that a shift coincides with an eigenvalue.

This flag is turned off by default, and may be necessary in some cases, especially when several partitions are being used. This feature currently requires an external package for factorizations with support for zero detection, e.g., MUMPS.

See also

[setInterval](#), [getKrylovSchurDetectZeros](#), [EPSKrylovSchurSetDetectZeros](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2661 <slepc4py/SLEPc/EPS.pyx#L2661>`

setKrylovSchurDimensions(*nev=None*, *ncv=None*, *mpd=None*)

Set the dimensions used for each subsolve step (spectrum slicing).

Logically collective.

Parameters

- **nev** (*int* / *None*) – Number of eigenvalues to compute.
- **ncv** (*int* / *None*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int* / *None*) – Maximum dimension allowed for the projected problem.

Return type

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [setInterval\(\)](#) and *SINVERT* is set.

The meaning of the parameters is the same as in [setDimensions\(\)](#), but the ones here apply to every subsolve done by the child *EPS* object.

See also

[`setInterval`](#), [`getKrylovSchurDimensions`](#), [`EPSKrylovSchurSetDimensions`](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2714](#) <[slepc4py/SLEPc/EPS.pyx#L2714](#)>

setKrylovSchurLocking(*lock*)

Set (toggle) locking/non-locking variants of the Krylov-Schur method.

Logically collective.

Parameters

lock (*bool*) – True if the locking variant must be selected.

Return type

None

Notes

The default is to lock converged eigenpairs when the method restarts. This behavior can be changed so that all directions are kept in the working subspace even if already converged to working accuracy (the non-locking variant).

See also

[`getKrylovSchurLocking`](#), [`EPSKrylovSchurSetLocking`](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2563](#) <[slepc4py/SLEPc/EPS.pyx#L2563](#)>

setKrylovSchurPartitions(*npart*)

Set the number of partitions of the communicator (spectrum slicing).

Logically collective.

Set the number of partitions for the case of doing spectrum slicing for a computational interval with the communicator split in several sub-communicators.

Parameters

npart (*int*) – The number of partitions.

Return type

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [`setInterval\(\)`](#) and *SINVERT* is set.

By default, *npart*=1 so all processes in the communicator participate in the processing of the whole interval. If *npart*>1 then the interval is divided into *npart* subintervals, each of them being processed by a subset of processes.

The interval is split proportionally unless the separation points are specified with [`setKrylovSchurSubintervals\(\)`](#).

See also

[`setInterval`](#), [`getKrylovSchurPartitions`](#), [`EPSKrylovSchurSetPartitions`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2607` `<slepc4py/SLEPc/EPS.pyx#L2607>`

setKrylovSchurRestart(*keep*)

Set the restart parameter for the Krylov-Schur method.

Logically collective.

It is the proportion of basis vectors that must be kept after restart.

Parameters

keep (*float*) – The number of vectors to be kept at restart.

Return type

None

Notes

Allowed values are in the range [0.1,0.9]. The default is 0.5.

See also

[`getKrylovSchurRestart`](#), [`EPSKrylovSchurSetRestart`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2520` `<slepc4py/SLEPc/EPS.pyx#L2520>`

setKrylovSchurSubintervals(*subint*)

Set the subinterval boundaries.

Logically collective.

Set the subinterval boundaries for spectrum slicing with a computational interval with several partitions.

Parameters

subint (*Sequence*[*float*]) – Real values specifying subintervals.

Return type

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [`setInterval\(\)`](#) and `SINVERT` is set.

This function must be called after [`setKrylovSchurPartitions\(\)`](#). For `npart` partitions, the argument `subint` must contain `npart+1` real values sorted in ascending order: `subint_0`, `subint_1`, ..., `subint_npart`, where the first and last values must coincide with the interval endpoints set with [`setInterval\(\)`](#). The subintervals are then defined by two consecutive points: `[subint_0,subint_1]`, `[subint_1,subint_2]`, and so on.

See also

[`setInterval`](#), [`setKrylovSchurPartitions`](#), [`EPSKrylovSchurSetSubintervals`](#)

:sources: ``Source code at slepc4py/SLEPc/EPS.pyx:2970 <slepc4py/SLEPc/EPS.pyx#L2970>``

setLOBPCGBlockSize(*bs*)

Set the block size of the LOBPCG method.

Logically collective.

Parameters

bs (*int*) – The block size.

Return type

`None`

See also

[`getLOBPCGBlockSize`](#), [`EPSLOBPCGSetBlockSize`](#)

:sources: ``Source code at slepc4py/SLEPc/EPS.pyx:3761 <slepc4py/SLEPc/EPS.pyx#L3761>``

setLOBPCGLocking(*lock*)

Toggle between locking and non-locking (LOBPCG method).

Logically collective.

Parameters

lock (*bool*) – True if the locking variant must be selected.

Return type

`None`

Notes

This flag refers to soft locking (converged vectors within the current block iterate), since hard locking is always used (when `nev` is larger than the block size).

See also

[`getLOBPCGLocking`](#), [`EPSLOBPCGSetLocking`](#)

:sources: ``Source code at slepc4py/SLEPc/EPS.pyx:3842 <slepc4py/SLEPc/EPS.pyx#L3842>``

setLOBPCGRestart(*restart*)

Set the restart parameter for the LOBPCG method.

Logically collective.

Parameters

restart (*float*) – The percentage of the block of vectors to force a restart.

Return type

`None`

Notes

The meaning of this parameter is the proportion of vectors within the current block iterate that must have converged in order to force a restart with hard locking. Allowed values are in the range [0.1,1.0]. The default is 0.9.

See also

[`getLOBPCGRestart`](#), [`EPSLOBPCGSetRestart`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:3798 <slepc4py/SLEPc/EPS.pyx#L3798>`

setLanczosReorthogType(*reorthog*)

Set the type of reorthogonalization used during the Lanczos iteration.

Logically collective.

Parameters

reorthog ([`LanczosReorthogType`](#)) – The type of reorthogonalization.

Return type

[`None`](#)

Notes

This call is only relevant if the type was set to [`EPS.Type.LANCZOS`](#) with [`setType\(\)`](#).

See also

[`getLanczosReorthogType`](#), [`EPSLanczosSetReorthog`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2432 <slepc4py/SLEPc/EPS.pyx#L2432>`

setLeftInitialSpace(*space*)

Set a left initial space from which the eigensolver starts to iterate.

Collective.

Parameters

space ([`Vec`](#) | [`list\[Vec\]`](#)) – Set of basis vectors of the left initial space.

Return type

[`None`](#)

Notes

Left initial vectors are used to initiate the left search space in two-sided eigensolvers. Users should pass here an approximation of the left eigenspace, if available.

The same comments in [`setInitialSpace\(\)`](#) are applicable here.

See also

[`setInitialSpace`](#), [`setTwoSided`](#), [`EPSSetLeftInitialSpace`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1654 <slepc4py/SLEPc/EPS.pyx#L1654>`

setLyapIIRanks(*rkc=None, rkl=None*)

Set the ranks used in the solution of the Lyapunov equation.

Logically collective.

Parameters

- **rkc** (*int* / *None*) – The compressed rank.
- **rkl** (*int* / *None*) – The Lyapunov rank.

Return type

None

Notes

Lyapunov inverse iteration needs to solve a large-scale Lyapunov equation at each iteration of the eigen-solver. For this, an iterative solver (*LME*) is used, which requires to prescribe the rank of the solution matrix *X*. This is the meaning of parameter *rkl*. Later, this matrix is compressed into another matrix of rank *rkc*. If not provided, *rkl* is a small multiple of *rkc*.

See also

getLyapIIRanks, *EPSLyapIISetRanks*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:3885 <slepc4py/SLEPc/EPs.pyx#L3885>](#)

setMonitor(*monitor, args=None, kargs=None*)

Append a monitor function to the list of monitors.

Logically collective.

See also

getMonitor, *cancelMonitor*, *EPsMonitorSet*

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1820 <slepc4py/SLEPc/EPs.pyx#L1820>](#)

Parameters

- **monitor** (*EPsMonitorFunction* / *None*)
- **args** (*tuple*[*Any*, ...] / *None*)
- **kargs** (*dict*[*str*, *Any*] / *None*)

Return type

None

setOperators(*A, B=None*)

Set the matrices associated with the eigenvalue problem.

Collective.

Parameters

- **A** (*Mat*) – The matrix associated with the eigensystem.
- **B** (*Mat* / *None*) – The second matrix in the case of generalized eigenproblems; if not provided, a standard eigenproblem is assumed.

Return type

None

Notes

It must be called before `setUp()`. If it is called again after `setUp()` and the matrix sizes have changed then the `EPS` object is reset.

For structured eigenproblem types such as `BSE`, see `setProblemType()`, the provided matrices must have been created with the corresponding helper function, i.e., `createMatBSE()`.

See also

`getOperators`, `solve`, `setUp`, `reset`, `setProblemType`, `EPSSetOperators`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1547 <slepc4py/SLEPc/EPS.pyx#L1547>`

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all EPS options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all EPS option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

For example, to distinguish between the runtime options for two different EPS contexts, one could call:

```
E1.setOptionsPrefix("eig1_")
E2.setOptionsPrefix("eig2_")
```

See also

`appendOptionsPrefix`, `getOptionsPrefix`, `EPSGetOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:484 <slepc4py/SLEPc/EPS.pyx#L484>`

setPowerShiftType(*shift*)

Set the type of shifts used during the power iteration.

Logically collective.

This can be used to emulate the Rayleigh Quotient Iteration (RQI) method.

Parameters

shift (*PowerShiftType*) – The type of shift.

Return type

None

Notes

This call is only relevant if the type was set to `EPS.Type.POWER` with `setType()`.

By default, shifts are constant (`EPS.PowerShiftType.CONSTANT`) and the iteration is the simple power method (or inverse iteration if a shift-and-invert transformation is being used).

A variable shift can be specified (`EPS.PowerShiftType.RAYLEIGH` or `EPS.PowerShiftType.WILKINSON`). In this case, the iteration behaves rather like a cubic converging method as RQI.

See also

`getPowerShiftType`, `EPSPowerSetShiftType`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:2330 <slepc4py/SLEPc/EPS.pyx#L2330>`

setProblemType(*problem_type*)

Set the type of the eigenvalue problem.

Logically collective.

Parameters

problem_type (`ProblemType`) – The problem type to be set.

Return type

`None`

Notes

This function must be used to instruct SLEPc to exploit symmetry or other kind of structure. If no problem type is specified, by default a non-Hermitian problem is assumed (either standard or generalized). If the user knows that the problem is Hermitian (i.e., $A = A^*$) or generalized Hermitian (i.e., $A = A^*$, $B = B^*$, and B positive definite) then it is recommended to set the problem type so that eigensolver can exploit these properties.

If the user does not call this function, the solver will use a reasonable guess.

For structured problem types such as `BSE`, the matrices passed in via `setOperators()` must have been created with the corresponding helper function, i.e., `createMatBSE()`.

See also

`setOperators`, `createMatBSE`, `getProblemType`, `EPSSetProblemType`

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:574 <slepc4py/SLEPc/EPS.pyx#L574>`

setPurify(*purify=True*)

Set (toggle) eigenvector purification.

Logically collective.

Parameters

purify (`bool`) – True to activate purification (default).

Return type

`None`

Notes

By default, eigenvectors of generalized symmetric eigenproblems are purified in order to purge directions in the nullspace of matrix B . If the user knows that B is non-singular, then purification can be safely deactivated and some computational cost is avoided (this is particularly important in interval computations).

See also

[`getPurify`](#), [`setInterval`](#), [`EPSSetPurify`](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1153 <slepc4py/SLEPc/EPs.pyx#L1153>](#)

setRG(*rg*)

Set a region object associated to the eigensolver.

Collective.

Parameters

rg (*RG*) – The region context.

Return type

None

See also

[`getRG`](#), [`EPSSetRG`](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:1503 <slepc4py/SLEPc/EPs.pyx#L1503>](#)

setRQCGReset(*nrest*)

Set the reset parameter of the RQCG iteration.

Logically collective.

Parameters

nrest (*int*) – The number of iterations between resets.

Return type

None

Notes

Every *nrest* iterations the solver performs a Rayleigh-Ritz projection step.

See also

[`getRQCGReset`](#), [`EPSRQCGSetReset`](#)

:sources: [Source code at slepc4py/SLEPc/EPs.pyx:3719 <slepc4py/SLEPc/EPs.pyx#L3719>](#)

setST(*st*)

Set a spectral transformation object associated to the eigensolver.

Collective.

Parameters

st (*ST*) – The spectral transformation.

Return type

None

See also[getST](#), [EPSSetST](#)**:sources:** `Source code at slepc4py/SLEPc/EPs.pyx:1392 <slepc4py/SLEPc/EPs.pyx#L1392>`**setStoppingTest**(*stopping*, *args*=None, *kargs*=None)

Set a function to decide when to stop the outer iteration of the eigensolver.

Logically collective.

See also[getStoppingTest](#), [EPSSetStoppingTestFunction](#)**:sources:** `Source code at slepc4py/SLEPc/EPs.pyx:1686 <slepc4py/SLEPc/EPs.pyx#L1686>`**Parameters**

- **stopping** ([EPSSettingFunction](#) | None)
- **args** ([tuple](#)[Any, ...] | None)
- **kargs** ([dict](#)[str, Any] | None)

Return type

None

setTarget(*target*)

Set the value of the target.

Logically collective.

Parameters**target** ([Scalar](#)) – The value of the target.**Return type**

None

Notes

The target is a scalar value used to determine the portion of the spectrum of interest. It is used in combination with [setWhichEigenpairs\(\)](#).

When PETSc is built with real scalars, it is not possible to specify a complex target.

See also[getTarget](#), [EPSSetTarget](#)**:sources:** `Source code at slepc4py/SLEPc/EPs.pyx:947 <slepc4py/SLEPc/EPs.pyx#L947>`

setThreshold(*thres*, *rel=False*)

Set the threshold used in the threshold stopping test.

Logically collective.

Parameters

- **thres** (*float*) – The threshold.
- **rel** (*bool*) – Whether the threshold is relative or not.

Return type

None

Notes

This function internally sets a special stopping test based on the threshold, where eigenvalues are computed in sequence until one of the computed eigenvalues is below/above the threshold (depending on whether largest or smallest eigenvalues are computed). The details are given in [EPSSetThreshold](#).

See also

[setStoppingTest](#), [getThreshold](#), [EPSSetThreshold](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:895 <slepc4py/SLEPc/EPs.pyx#L895>`

setTolerances(*tol=None*, *max_it=None*)

Set the tolerance and max. iter. used by the default EPS convergence tests.

Logically collective.

Parameters

- **tol** (*float* | *None*) – The convergence tolerance.
- **max_it** (*int* | *None*) – The maximum number of iterations.

Return type

None

Notes

Use [DETERMINE](#) for *max_it* to assign a reasonably good value, which is dependent on the solution method.

See also

[getTolerances](#), [EPSSetTolerances](#)

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:1059 <slepc4py/SLEPc/EPs.pyx#L1059>`

setTrackAll(*trackall*)

Set if the solver must compute the residual of all approximate eigenpairs.

Logically collective.

Parameters

- **trackall** (*bool*) – Whether to compute all residuals or not.

Return type

None

See also

[`getTrackAll`](#), [`EPSSetTrackAll`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1274 <slepc4py/SLEPc/EPS.pyx#L1274>`

setTrueResidual(*trueres*)

Set if the solver must compute the true residual explicitly or not.

Logically collective.

Parameters

trueres (*bool*) – Whether the solver computes true residuals or not.

Return type

None

See also

[`getTrueResidual`](#), [`EPSSetTrueResidual`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1237 <slepc4py/SLEPc/EPS.pyx#L1237>`

setTwoSided(*twosided*)

Set to use a two-sided variant that also computes left eigenvectors.

Logically collective.

Parameters

twosided (*bool*) – Whether the two-sided variant is to be used or not.

Return type

None

Notes

If the user sets `twosided` to `True` then the solver uses a variant of the algorithm that computes both right and left eigenvectors. This is usually much more costly. This option is not available in all solvers.

When using two-sided solvers, the problem matrices must have both the `Mat.mult` and `Mat.multTranspose` operations defined.

See also

[`getTwoSided`](#), [`getLeftEigenvector`](#), [`EPSSetTwoSided`](#)

:sources: `Source code at slepc4py/SLEPc/EPS.pyx:1106 <slepc4py/SLEPc/EPS.pyx#L1106>`

setType(*eps_type*)

Set the particular solver to be used in the EPS object.

Logically collective.

Parameters

eps_type (*Type* / *str*) – The solver to be used.

Return type

None

Notes

The default is *KRYLOV SCHUR*. Normally, it is best to use *setFromOptions()* and then set the EPS type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also*getType*, *EPSSetType*

:sources: *Source code at slepc4py/SLEPc/EPs.pyx:419 <slepc4py/SLEPc/EPs.pyx#L419>*

setUp()

Set up all the internal data structures.

Collective.

Notes

Sets up all the internal data structures necessary for the execution of the eigensolver. This includes the setup of the internal *ST* object.

This function need not be called explicitly in most cases, since *solve()* calls it. It can be useful when one wants to measure the set-up time separately from the solve time.

See also*solve*, *setInitialSpace*, *setDeflationSpace*, *EPSSetUp*

:sources: *Source code at slepc4py/SLEPc/EPs.pyx:1877 <slepc4py/SLEPc/EPs.pyx#L1877>*

Return type

None

setWhichEigenpairs(*which*)

Set which portion of the spectrum is to be sought.

Logically collective.

Parameters

which (*Which*) – The portion of the spectrum to be sought by the solver.

Return type

None

Notes

Not all eigensolvers implemented in EPS account for all the possible values. Also, some values make sense only for certain types of problems. If SLEPc is compiled for real numbers *EPs.Which.LARGEST_IMAGINARY* and *EPs.Which.SMALLEST_IMAGINARY* use the absolute value of the imaginary part for eigenvalue selection.

The target is a scalar value provided with *setTarget()*.

The criterion *EPS.Which.TARGET_IMAGINARY* is available only in case PETSc and SLEPc have been built with complex scalars.

EPS.Which.ALL is intended for use in combination with an interval (see *setInterval()*), when all eigenvalues within the interval are requested, or in the context of the *EPS.Type.CISS* solver for computing all eigenvalues in a region.

See also

setTarget, *setInterval*, *getWhichEigenpairs*, *EPSSetWhichEigenpairs*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:836 <slepc4py/SLEPc/EPs.pyx#L836>`

solve()

Solve the eigensystem.

Collective.

Notes

The problem matrices are specified with *setOperators()*.

solve() will return without generating an error regardless of whether all requested solutions were computed or not. Call *getConverged()* to get the actual number of computed solutions, and *getConvergedReason()* to determine if the solver converged or failed and why.

See also

setUp, *setOperators*, *getConverged*, *getConvergedReason*, *EPSSolve*

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:1899 <slepc4py/SLEPc/EPs.pyx#L1899>`

Return type

None

updateKrylovSchurSubcommMats(*s=1.0*, *a=1.0*, *Au=None*, *t=1.0*, *b=1.0*, *Bu=None*, *structure=None*, *globalup=False*)

Update the eigenproblem matrices stored internally in the communicator.

Collective.

Update the eigenproblem matrices stored internally in the subcommunicator to which the calling process belongs.

Parameters

- **s** (*Scalar*) – Scalar that multiplies the existing A matrix.
- **a** (*Scalar*) – Scalar used in the axpy operation on A.
- **Au** (*petsc4py.PETSc.Mat* / *None*) – The matrix used in the axpy operation on A.
- **t** (*Scalar*) – Scalar that multiplies the existing B matrix.
- **b** (*Scalar*) – Scalar used in the axpy operation on B.
- **Bu** (*petsc4py.PETSc.Mat* / *None*) – The matrix used in the axpy operation on B.
- **structure** (*petsc4py.PETSc.Mat.Structure* / *None*) – Either same, different, or a subset of the non-zero sparsity pattern.

- **globalup** (*bool*) – Whether global matrices must be updated or not.

Return type

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with `setInterval()` and `SINVERT` is set. And is relevant only when the number of partitions (`setKrylovSchurPartitions()`) is larger than one.

This function modifies the eigenproblem matrices at subcommunicator level, and optionally updates the global matrices in the parent communicator. The updates are expressed as $A \leftarrow sA + aAu$, $B \leftarrow tB + bBu$.

It is possible to update one of the matrices, or both.

The matrices Au and Bu must be equal in all subcommunicators.

The structure flag is passed to the `petsc4py.PETSc.Mat.axpy` operations to perform the updates.

If `globalup` is `True`, communication is carried out to reconstruct the updated matrices in the parent communicator.

See also

`setInterval`, `setKrylovSchurPartitions`, `EPSCrylovSchurUpdateSubcommMats`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2899 <slepc4py/SLEPc/EPs.pyx#L2899>`

valuesView(*viewer=None*)

Display the computed eigenvalues in a viewer.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

`solve`, `vectorsView`, `errorView`, `EPsValuesView`

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:2290 <slepc4py/SLEPc/EPs.pyx#L2290>`

vectorsView(*viewer=None*)

Output computed eigenvectors to a viewer.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

[solve](#), [valuesView](#), [errorView](#), [EPSVectorsView](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:2309 <slepc4py/SLEPc/EPS.pyx#L2309>](#)

view(viewer=None)

Print the EPS data structure.

Collective.

Parameters

viewer ([Viewer](#) / [None](#)) – Visualization context; if not provided, the standard output is used.

Return type

[None](#)

See also

[EPSView](#)

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:353 <slepc4py/SLEPc/EPS.pyx#L353>](#)

Attributes Documentation

bv

The basis vectors ([BV](#)) object associated.

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:4345 <slepc4py/SLEPc/EPS.pyx#L4345>](#)

ds

The direct solver ([DS](#)) object associated.

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:4359 <slepc4py/SLEPc/EPS.pyx#L4359>](#)

extraction

The type of extraction technique to be employed.

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:4275 <slepc4py/SLEPc/EPS.pyx#L4275>](#)

max_it

The maximum iteration count.

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:4303 <slepc4py/SLEPc/EPS.pyx#L4303>](#)

problem_type

The type of the eigenvalue problem.

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:4268 <slepc4py/SLEPc/EPS.pyx#L4268>](#)

purify

Eigenvector purification.

:sources: [Source code at slepc4py/SLEPc/EPS.pyx:4324 <slepc4py/SLEPc/EPS.pyx#L4324>](#)

rg

The region (*RG*) object associated.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4352 <slepc4py/SLEPc/EPs.pyx#L4352>`

st

The spectral transformation (*ST*) object associated.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4338 <slepc4py/SLEPc/EPs.pyx#L4338>`

target

The value of the target.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4289 <slepc4py/SLEPc/EPs.pyx#L4289>`

tol

The tolerance.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4296 <slepc4py/SLEPc/EPs.pyx#L4296>`

track_all

Compute the residual norm of all approximate eigenpairs.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4331 <slepc4py/SLEPc/EPs.pyx#L4331>`

true_residual

Compute the true residual explicitly.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4317 <slepc4py/SLEPc/EPs.pyx#L4317>`

two_sided

Two-sided that also computes left eigenvectors.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4310 <slepc4py/SLEPc/EPs.pyx#L4310>`

which

The portion of the spectrum to be sought.

:sources: `Source code at slepc4py/SLEPc/EPs.pyx:4282 <slepc4py/SLEPc/EPs.pyx#L4282>`

__init__()

classmethod __new__(*args, **kwargs)

slepc4py.SLEPc.FN

class slepc4py.SLEPc.FN

Bases: *Object*

Mathematical Function.

The *FN* package provides the functionality to represent a simple mathematical function such as an exponential, a polynomial or a rational function. This is used as a building block for defining the function associated to the nonlinear eigenproblem, as well as for specifying which function to use when computing the action of a matrix function on a vector.

continues on next page

Table 38 – continued from previous page

Enumerations

<i>CombineType</i>	FN type of combination of child functions.
<i>ParallelType</i>	FN parallel types.
<i>Type</i>	FN type.

slepc4py.SLEPc.FN.CombineType

class slepc4py.SLEPc.FN.CombineType

Bases: `object`

FN type of combination of child functions.

- *ADD*: Addition $f(x) = f_1(x) + f_2(x)$
- *MULTIPLY*: Multiplication $f(x) = f_1(x)f_2(x)$
- *DIVIDE*: Division $f(x) = f_1(x)/f_2(x)$
- *COMPOSE*: Composition $f(x) = f_2(f_1(x))$

See also

FNCombineType

Attributes Summary

<i>ADD</i>	Constant ADD of type <code>int</code>
<i>COMPOSE</i>	Constant COMPOSE of type <code>int</code>
<i>DIVIDE</i>	Constant DIVIDE of type <code>int</code>
<i>MULTIPLY</i>	Constant MULTIPLY of type <code>int</code>

Attributes Documentation

ADD: `int` = **ADD**

Constant ADD of type `int`

COMPOSE: `int` = **COMPOSE**

Constant COMPOSE of type `int`

DIVIDE: `int` = **DIVIDE**

Constant DIVIDE of type `int`

MULTIPLY: `int` = **MULTIPLY**

Constant MULTIPLY of type `int`

__init__()

classmethod **__new__**(*args, **kwargs)

slepc4py.SLEPc.FN.ParallelType

class slepc4py.SLEPc.FN.ParallelType

Bases: `object`

FN parallel types.

- *REDUNDANT*: Every process performs the computation redundantly.
- *SYNCHRONIZED*: The first process sends the result to the rest.

See also

`FNParallelType`

Attributes Summary

<i>REDUNDANT</i>	Constant REDUNDANT of type <code>int</code>
<i>SYNCHRONIZED</i>	Constant SYNCHRONIZED of type <code>int</code>

Attributes Documentation

REDUNDANT: `int` = REDUNDANT

Constant REDUNDANT of type `int`

SYNCHRONIZED: `int` = SYNCHRONIZED

Constant SYNCHRONIZED of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.FN.Type

class slepc4py.SLEPc.FN.Type

Bases: `object`

FN type.

- *COMBINE*: A math function defined by combining two functions.
- *RATIONAL*: A rational function $f(x) = p(x)/q(x)$.
- *EXP*: The exponential function $f(x) = e^x$.
- *LOG*: The logarithm function $f(x) = \log x$.
- *PHI*: One of the Phi_k functions with index k.
- *SQRT*: The square root function $f(x) = \sqrt{x}$.
- *INVSQRT*: The inverse square root function.

See also

`FNType`

Attributes Summary

<i>COMBINE</i>	Object COMBINE of type <i>str</i>
<i>EXP</i>	Object EXP of type <i>str</i>
<i>INVSQRT</i>	Object INVSQRT of type <i>str</i>
<i>LOG</i>	Object LOG of type <i>str</i>
<i>PHI</i>	Object PHI of type <i>str</i>
<i>RATIONAL</i>	Object RATIONAL of type <i>str</i>
<i>SQRT</i>	Object SQRT of type <i>str</i>

Attributes Documentation

COMBINE: *str* = COMBINE

Object COMBINE of type *str*

EXP: *str* = EXP

Object EXP of type *str*

INVSQRT: *str* = INVSQRT

Object INVSQRT of type *str*

LOG: *str* = LOG

Object LOG of type *str*

PHI: *str* = PHI

Object PHI of type *str*

RATIONAL: *str* = RATIONAL

Object RATIONAL of type *str*

SQRT: *str* = SQRT

Object SQRT of type *str*

__init__()

classmethod __new__(*args, **kwargs)

Methods Summary

<i>appendOptionsPrefix</i> ([prefix])	Append to the prefix used for searching for all FN options in the database.
<i>create</i> ([comm])	Create the FN object.
<i>destroy</i> ()	Destroy the FN object.
<i>duplicate</i> ([comm])	Duplicate the FN object copying all parameters.
<i>evaluateDerivative</i> (x)	Compute the value of the derivative $f'(x)$ for a given x.
<i>evaluateFunction</i> (x)	Compute the value of the function $f(x)$ for a given x.
<i>evaluateFunctionMat</i> (A[, B])	Compute the value of the function $f(A)$ for a given matrix A.
<i>evaluateFunctionMatVec</i> (A[, v])	Compute the first column of the matrix $f(A)$.
<i>getCombineChildren</i> ()	Get the two child functions that constitute this combined function.
<i>getMethod</i> ()	Get the method currently used for matrix functions.

continues on next page

Table 42 – continued from previous page

<code>getOptionsPrefix()</code>	Get the prefix used for searching for all FN options in the database.
<code>getParallel()</code>	Get the mode of operation in parallel runs.
<code>getPhiIndex()</code>	Get the index of the phi-function.
<code>getRationalDenominator()</code>	Get the coefficients of the denominator of the rational function.
<code>getRationalNumerator()</code>	Get the coefficients of the numerator of the rational function.
<code>getScale()</code>	Get the scaling parameters that define the mathematical function.
<code>getType()</code>	Get the FN type of this object.
<code>setCombineChildren(comb, f1, f2)</code>	Set the two child functions that constitute this combined function.
<code>setFromOptions()</code>	Set FN options from the options database.
<code>setMethod(meth)</code>	Set the method to be used to evaluate functions of matrices.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all FN options in the database.
<code>setParallel(pmode)</code>	Set the mode of operation in parallel runs.
<code>setPhiIndex(k)</code>	Set the index of the phi-function.
<code>setRationalDenominator(alpha)</code>	Set the coefficients of the denominator of the rational function.
<code>setRationalNumerator(alpha)</code>	Set the coefficients of the numerator of the rational function.
<code>setScale([alpha, beta])</code>	Set the scaling parameters that define the mathematical function.
<code>setType(fn_type)</code>	Set the type for the FN object.
<code>view([viewer])</code>	Print the FN data structure.

Attributes Summary

<code>method</code>	The method to be used to evaluate functions of matrices.
<code>parallel</code>	The mode of operation in parallel runs.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching for all FN options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all FN option requests.

Return type

None

See also
setOptionsPrefix , getOptionsPrefix , FNAppendOptionsPrefix

:sources: ``Source code at slepc4py/SLEPc/FN.pyx:266 <slepc4py/SLEPc/FN.pyx#L266>``

create(*comm=None*)

Create the FN object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator; if not provided, it defaults to all processes.

Return type

Self

See also
FNCreate

:sources: ``Source code at slepc4py/SLEPc/FN.pyx:182 <slepc4py/SLEPc/FN.pyx#L182>``

destroy()

Destroy the FN object.

Collective.

See also
FNDestroy

:sources: ``Source code at slepc4py/SLEPc/FN.pyx:168 <slepc4py/SLEPc/FN.pyx#L168>``

Return type

Self

duplicate(*comm=None*)

Duplicate the FN object copying all parameters.

Collective.

Duplicate the FN object copying all parameters, possibly with a different communicator.

Parameters

comm (*Comm* / *None*) – MPI communicator; if not provided, it defaults to the object's communicator.

Returns

The new object.

Return type

FN

See also

[create](#), [FNDuplicate](#)

:sources: [Source code at slepc4py/SLEPc/FN.pyx:321 <slepc4py/SLEPc/FN.pyx#L321>](#)

evaluateDerivative(x)

Compute the value of the derivative $f'(x)$ for a given x .

Not collective.

Parameters

\mathbf{x} ([Scalar](#)) – Value where the derivative must be evaluated.

Returns

The result of $f'(x)$.

Return type

[Scalar](#)

Notes

Scaling factors are taken into account, so the actual derivative evaluation will return $abf'(ax)$.

See also

[evaluateFunction](#), [setScale](#), [FNEvaluateDerivative](#)

:sources: [Source code at slepc4py/SLEPc/FN.pyx:382 <slepc4py/SLEPc/FN.pyx#L382>](#)

evaluateFunction(x)

Compute the value of the function $f(x)$ for a given x .

Not collective.

Parameters

\mathbf{x} ([Scalar](#)) – Value where the function must be evaluated.

Returns

The result of $f(x)$.

Return type

[Scalar](#)

Notes

Scaling factors are taken into account, so the actual function evaluation will return $bf(ax)$.

See also

[evaluateDerivative](#), [evaluateFunctionMat](#), [setScale](#), [FNEvaluateFunction](#)

:sources: [Source code at slepc4py/SLEPc/FN.pyx:352 <slepc4py/SLEPc/FN.pyx#L352>](#)

evaluateFunctionMat(*A*, *B=None*)

Compute the value of the function $f(A)$ for a given matrix *A*.

Logically collective.

Parameters

- **A** (*Mat*) – Matrix on which the function must be evaluated.
- **B** (*Mat* | *None*) – Placeholder for the result.

Returns

The result of $f(A)$.

Return type

`petsc4py.PETSc.Mat`

Notes

Scaling factors are taken into account, so the actual function evaluation will return $bf(aA)$.

See also

`evaluateFunction`, `evaluateFunctionMatVec`, `FNEvaluateFunctionMat`

:sources: `Source code at slepc4py/SLEPc/FN.pyx:412 <slepc4py/SLEPc/FN.pyx#L412>`

evaluateFunctionMatVec(*A*, *v=None*)

Compute the first column of the matrix $f(A)$.

Logically collective.

Parameters

- **A** (*Mat*) – Matrix on which the function must be evaluated.
- **v** (*Vec* | *None*)

Returns

The first column of the result $f(A)$.

Return type

`petsc4py.PETSc.Vec`

Notes

This operation is similar to `evaluateFunctionMat()` but returns only the first column of $f(A)$, hence saving computations in most cases.

See also

`evaluateFunctionMat`, `FNEvaluateFunctionMatVec`

:sources: `Source code at slepc4py/SLEPc/FN.pyx:443 <slepc4py/SLEPc/FN.pyx#L443>`

getCombineChildren()

Get the two child functions that constitute this combined function.

Not collective.

Get the two child functions that constitute this combined function, and the way they must be combined.

Returns

- **comb** (*CombineType*) – How to combine the functions (addition, multiplication, division, composition).
- **f1** (*FN*) – First function.
- **f2** (*FN*) – Second function.

Return type

tuple[CombineType, FN, FN]

See also

setCombineChildren, FNCombineGetChildren

:sources: [Source code at slepc4py/SLEPc/FN.pyx:725 <slepc4py/SLEPc/FN.pyx#L725>](#)

getMethod()

Get the method currently used for matrix functions.

Not collective.

Returns

An index identifying the method.

Return type

int

See also

setMethod, FNGetMethod

:sources: [Source code at slepc4py/SLEPc/FN.pyx:546 <slepc4py/SLEPc/FN.pyx#L546>](#)

getOptionsPrefix()

Get the prefix used for searching for all FN options in the database.

Not collective.

Returns

The prefix string set for this FN object.

Return type

str

See also

setOptionsPrefix, appendOptionsPrefix, FNGetOptionsPrefix

:sources: [Source code at slepc4py/SLEPc/FN.pyx:285 <slepc4py/SLEPc/FN.pyx#L285>](#)

getParallel()

Get the mode of operation in parallel runs.

Not collective.

Returns

The parallel mode.

Return type

ParallelType

See also

setParallel, *FNGetParallel*

:sources: [Source code at slepc4py/SLEPc/FN.pyx:588 <slepc4py/SLEPc/FN.pyx#L588>](#)

getPhiIndex()

Get the index of the phi-function.

Not collective.

Returns

The index.

Return type

int

See also

setPhiIndex, *FNPhiGetIndex*

:sources: [Source code at slepc4py/SLEPc/FN.pyx:778 <slepc4py/SLEPc/FN.pyx#L778>](#)

getRationalDenominator()

Get the coefficients of the denominator of the rational function.

Not collective.

Returns

Coefficients.

Return type

ArrayScalar

See also

setRationalDenominator, *FN RationalGetDenominator*

:sources: [Source code at slepc4py/SLEPc/FN.pyx:674 <slepc4py/SLEPc/FN.pyx#L674>](#)

getRationalNumerator()

Get the coefficients of the numerator of the rational function.

Not collective.

Returns

Coefficients.

Return type

ArrayScalar

See also

[`setRationalNumerator`](#), [`FNrationalGetNumerator`](#)

:sources: `Source code at slepc4py/SLEPc/FN.pyx:629 <slepc4py/SLEPc/FN.pyx#L629>`

getScale()

Get the scaling parameters that define the mathematical function.

Not collective.

Returns

- **alpha** (*Scalar*) – Inner scaling (argument).
- **beta** (*Scalar*) – Outer scaling (result).

Return type

`tuple[Scalar, Scalar]`

See also

[`setScale`](#), [`FNGetScale`](#)

:sources: `Source code at slepc4py/SLEPc/FN.pyx:496 <slepc4py/SLEPc/FN.pyx#L496>`

getType()

Get the FN type of this object.

Not collective.

Returns

The math function type currently being used.

Return type

`str`

See also

[`setType`](#), [`FNGetType`](#)

:sources: `Source code at slepc4py/SLEPc/FN.pyx:222 <slepc4py/SLEPc/FN.pyx#L222>`

setCombineChildren(*comb*, *f1*, *f2*)

Set the two child functions that constitute this combined function.

Logically collective.

Set the two child functions that constitute this combined function, and the way they must be combined.

Parameters

- **comb** (*CombineType*) – How to combine the functions (addition, multiplication, division, composition).
- **f1** (*FN*) – First function.
- **f2** (*FN*) – Second function.

Return type

None

See also[*getCombineChildren*](#), [*FNCombineSetChildren*](#)**`:sources:`** `Source code at slepc4py/SLEPc/FN.pyx:699 <slepc4py/SLEPc/FN.pyx#L699>`**setFromOptions()**

Set FN options from the options database.

Collective.

NotesTo see all options, run your program with the `-help` option.**See also**[*setOptionsPrefix*](#), [*FNSetFromOptions*](#)**`:sources:`** `Source code at slepc4py/SLEPc/FN.pyx:304 <slepc4py/SLEPc/FN.pyx#L304>`**Return type**

None

setMethod(*meth*)

Set the method to be used to evaluate functions of matrices.

Logically collective.

Parameters**meth** (*int*) – An index identifying the method.**Return type**

None

NotesIn some *FN* types there are more than one algorithms available for computing matrix functions. In that case, this function allows choosing the wanted method.If *meth* is currently set to 0 and the input argument of *FN.evaluateFunctionMat()* is a symmetric/Hermitian matrix, then the computation is done via the eigendecomposition, rather than with the general algorithm.**See also**[*getMethod*](#), [*FNSetMethod*](#)**`:sources:`** `Source code at slepc4py/SLEPc/FN.pyx:517 <slepc4py/SLEPc/FN.pyx#L517>`

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all FN options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all FN option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

See also

appendOptionsPrefix, getOptionsPrefix, FNGetOptionsPrefix

:sources: *Source code at slepc4py/SLEPc/FN.pyx:241 <slepc4py/SLEPc/FN.pyx#L241>*

setParallel(*pmode*)

Set the mode of operation in parallel runs.

Logically collective.

Parameters

pmode (*ParallelType*) – The parallel mode.

Return type

None

Notes

This is relevant only when the function is evaluated on a matrix, with either *evaluateFunctionMat()* or *evaluateFunctionMatVec()*.

See also

evaluateFunctionMat, getParallel, FNSetParallel

:sources: *Source code at slepc4py/SLEPc/FN.pyx:565 <slepc4py/SLEPc/FN.pyx#L565>*

setPhiIndex(*k*)

Set the index of the phi-function.

Logically collective.

Parameters

k (*int*) – The index.

Return type

None

Notes

If not set, the default index is 1.

See also

[`getPhiIndex`](#), [`FNPhiSetIndex`](#)

:sources: [Source code at slepc4py/SLEPc/FN.pyx:756 <slepc4py/SLEPc/FN.pyx#L756>](#)

setRationalDenominator(*alpha*)

Set the coefficients of the denominator of the rational function.

Logically collective.

Parameters

alpha (*Sequence*[*Scalar*]) – Coefficients.

Return type

None

See also

[`setRationalNumerator`](#), [`FN RationalSetDenominator`](#)

:sources: [Source code at slepc4py/SLEPc/FN.pyx:654 <slepc4py/SLEPc/FN.pyx#L654>](#)

setRationalNumerator(*alpha*)

Set the coefficients of the numerator of the rational function.

Logically collective.

Parameters

alpha (*Sequence*[*Scalar*]) – Coefficients.

Return type

None

See also

[`setRationalDenominator`](#), [`FN RationalSetNumerator`](#)

:sources: [Source code at slepc4py/SLEPc/FN.pyx:609 <slepc4py/SLEPc/FN.pyx#L609>](#)

setScale(*alpha=None*, *beta=None*)

Set the scaling parameters that define the mathematical function.

Logically collective.

Parameters

- **alpha** (*Scalar* / *None*) – Inner scaling (argument), default is 1.0.
- **beta** (*Scalar* / *None*) – Outer scaling (result), default is 1.0.

Return type

None

See also

[getScale](#), [evaluateFunction](#), [FNSetScale](#)

:sources: `Source code at slepc4py/SLEPc/FN.pyx:473 <slepc4py/SLEPc/FN.pyx#L473>`

setType(*fn_type*)

Set the type for the FN object.

Logically collective.

Parameters

fn_type (*Type* / *str*) – The math function type to be used.

Return type

None

See also

[getType](#), [FNSetType](#)

:sources: `Source code at slepc4py/SLEPc/FN.pyx:203 <slepc4py/SLEPc/FN.pyx#L203>`

view(*viewer=None*)

Print the FN data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

[FNView](#)

:sources: `Source code at slepc4py/SLEPc/FN.pyx:149 <slepc4py/SLEPc/FN.pyx#L149>`

Attributes Documentation

method

The method to be used to evaluate functions of matrices.

:sources: `Source code at slepc4py/SLEPc/FN.pyx:799 <slepc4py/SLEPc/FN.pyx#L799>`

parallel

The mode of operation in parallel runs.

:sources: `Source code at slepc4py/SLEPc/FN.pyx:806 <slepc4py/SLEPc/FN.pyx#L806>`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.LME

class slepc4py.SLEPc.LME

Bases: `Object`

Linear Matrix Equation.

Linear Matrix Equation ([LME](#)) is the object provided by slepc4py for solving linear matrix equations such as Lyapunov or Sylvester where the solution has low rank.

Enumerations

<i>ConvergedReason</i>	LME convergence reasons.
<i>ProblemType</i>	LME problem type.
<i>Type</i>	LME type.

slepc4py.SLEPc.LME.ConvergedReason

class slepc4py.SLEPc.LME.ConvergedReason

Bases: `object`

LME convergence reasons.

- *CONVERGED_TOL*: All eigenpairs converged to requested tolerance.
- *DIVERGED_ITS*: Maximum number of iterations exceeded.
- *DIVERGED_BREAKDOWN*: Solver failed due to breakdown.
- *CONVERGED_ITERATING*: Iteration not finished yet.

See also

`LMEConvergedReason`

Attributes Summary

<i>CONVERGED_ITERATING</i>	Constant <code>CONVERGED_ITERATING</code> of type <code>int</code>
<i>CONVERGED_TOL</i>	Constant <code>CONVERGED_TOL</code> of type <code>int</code>
<i>DIVERGED_BREAKDOWN</i>	Constant <code>DIVERGED_BREAKDOWN</code> of type <code>int</code>
<i>DIVERGED_ITS</i>	Constant <code>DIVERGED_ITS</code> of type <code>int</code>
<i>ITERATING</i>	Constant <code>ITERATING</code> of type <code>int</code>

Attributes Documentation

CONVERGED_ITERATING: `int` = `CONVERGED_ITERATING`

Constant `CONVERGED_ITERATING` of type `int`

CONVERGED_TOL: `int` = `CONVERGED_TOL`

Constant `CONVERGED_TOL` of type `int`

DIVERGED_BREAKDOWN: `int` = `DIVERGED_BREAKDOWN`

Constant `DIVERGED_BREAKDOWN` of type `int`

```

DIVERGED_ITS: int = DIVERGED_ITS
    Constant DIVERGED_ITS of type int

ITERATING: int = ITERATING
    Constant ITERATING of type int

__init__()

classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.LME.ProblemType

class slepc4py.SLEPc.LME.ProblemType

Bases: `object`

LME problem type.

- *LYAPUNOV*: Continuous-time Lyapunov.
- *SYLVESTER*: Continuous-time Sylvester.
- *GEN_LYAPUNOV*: Generalized Lyapunov.
- *GEN_SYLVESTER*: Generalized Sylvester.
- *DT_LYAPUNOV*: Discrete-time Lyapunov.
- *STEIN*: Stein.

See also
<code>LMEProblemType</code>

Attributes Summary

<i>DT_LYAPUNOV</i>	Constant DT_LYAPUNOV of type <code>int</code>
<i>GEN_LYAPUNOV</i>	Constant GEN_LYAPUNOV of type <code>int</code>
<i>GEN_SYLVESTER</i>	Constant GEN_SYLVESTER of type <code>int</code>
<i>LYAPUNOV</i>	Constant LYAPUNOV of type <code>int</code>
<i>STEIN</i>	Constant STEIN of type <code>int</code>
<i>SYLVESTER</i>	Constant SYLVESTER of type <code>int</code>

Attributes Documentation

```

DT_LYAPUNOV: int = DT_LYAPUNOV
    Constant DT_LYAPUNOV of type int

GEN_LYAPUNOV: int = GEN_LYAPUNOV
    Constant GEN_LYAPUNOV of type int

GEN_SYLVESTER: int = GEN_SYLVESTER
    Constant GEN_SYLVESTER of type int

LYAPUNOV: int = LYAPUNOV
    Constant LYAPUNOV of type int

```

```

STEIN: int = STEIN
    Constant STEIN of type int

SYLVESTER: int = SYLVESTER
    Constant SYLVESTER of type int

__init__()

classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.LME.Type

```

class slepc4py.SLEPc.LME.Type
    Bases: object
    LME type.
    • KRYLOV: Restarted Krylov solver.

```

See also
LMEType

Attributes Summary

<i>KRYLOV</i>	Object KRYLOV of type <code>str</code>
---------------	--

Attributes Documentation

```

KRYLOV: str = KRYLOV
    Object KRYLOV of type str

__init__()

classmethod __new__(*args, **kwargs)

```

Methods Summary

<i>appendOptionsPrefix</i> ([prefix])	Append to the prefix used for searching in the database.
<i>cancelMonitor</i> ()	Clear all monitors for an <i>LME</i> object.
<i>computeError</i> ()	Compute the error associated with the last equation solved.
<i>create</i> ([comm])	Create the LME object.
<i>destroy</i> ()	Destroy the LME object.
<i>getBV</i> ()	Get the basis vector object associated to the LME object.
<i>getCoefficients</i> ()	Get the coefficient matrices of the matrix equation.
<i>getConvergedReason</i> ()	Get the reason why the <i>solve</i> () iteration was stopped.
<i>getDimensions</i> ()	Get the dimension of the subspace used by the solver.
<i>getErrorEstimate</i> ()	Get the error estimate obtained during the solve.

continues on next page

Table 48 – continued from previous page

<code>getErrorIfNotConverged()</code>	Get if <code>solve()</code> generates an error if the solver does not converge.
<code>getIterationNumber()</code>	Get the current iteration number.
<code>getMonitor()</code>	Get the list of monitor functions.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all LME options in the database.
<code>getProblemType()</code>	Get the LME problem type of this object.
<code>getRHS()</code>	Get the right-hand side of the matrix equation.
<code>getSolution()</code>	Get the solution of the matrix equation.
<code>getTolerances()</code>	Get the tolerance and maximum iteration count.
<code>getType()</code>	Get the LME type of this object.
<code>reset()</code>	Reset the LME object.
<code>setBV(bv)</code>	Set a basis vector object to the LME object.
<code>setCoefficients(A[, B, D, E])</code>	Set the coefficient matrices.
<code>setDimensions(ncv)</code>	Set the dimension of the subspace to be used by the solver.
<code>setErrorIfNotConverged([flag])</code>	Set <code>solve()</code> to generate an error if the solver has not converged.
<code>setFromOptions()</code>	Set LME options from the options database.
<code>setMonitor(monitor[, args, kargs])</code>	Append a monitor function to the list of monitors.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all LME options in the database.
<code>setProblemType(lme_problem_type)</code>	Set the LME problem type of this object.
<code>setRHS(C)</code>	Set the right-hand side of the matrix equation.
<code>setSolution([X])</code>	Set the placeholder for the solution of the matrix equation.
<code>setTolerances([tol, max_it])</code>	Set the tolerance and maximum iteration count.
<code>setType(lme_type)</code>	Set the particular solver to be used in the LME object.
<code>setUp()</code>	Set up all the internal necessary data structures.
<code>solve()</code>	Solve the linear matrix equation.
<code>view([viewer])</code>	Print the LME data structure.

Attributes Summary

<code>bv</code>	The basis vectors (<i>BV</i>) object associated to the LME object.
<code>fn</code>	The math function (<i>FN</i>) object associated to the LME object.
<code>max_it</code>	The maximum iteration count used by the LME convergence tests.
<code>tol</code>	The tolerance value used by the LME convergence tests.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching in the database.

Logically collective.

Append to the prefix used for searching for all LME options in the database.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all LME option requests.

Return type

None

See also

setOptionsPrefix, *getOptionsPrefix*, *LMEAppendOptionsPrefix*

:sources: `Source code at slepc4py/SLEPc/LME.pyx:512 <slepc4py/SLEPc/LME.pyx#L512>`

cancelMonitor()

Clear all monitors for an *LME* object.

Logically collective.

See also

LMEMonitorCancel

:sources: `Source code at slepc4py/SLEPc/LME.pyx:713 <slepc4py/SLEPc/LME.pyx#L713>`

Return type

None

computeError()

Compute the error associated with the last equation solved.

Collective.

Returns

The error.

Return type

float

Notes

The error is based on the residual norm.

This function is not scalable (in terms of memory or parallel communication), so it should not be called except in the case of small problem size. For large equations, use *getErrorEstimate()*.

See also

getErrorEstimate, *LMEComputeError*

:sources: `Source code at slepc4py/SLEPc/LME.pyx:435 <slepc4py/SLEPc/LME.pyx#L435>`

create(comm=None)

Create the LME object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator. If not provided, it defaults to all processes.

Return type*Self***See also**[LMCreate](#)**:sources:** `Source code at slepc4py/SLEPc/LME.pyx:121 <slepc4py/SLEPc/LME.pyx#L121>`**destroy()**

Destroy the LME object.

Collective.

See also[LMEDestroy](#)**:sources:** `Source code at slepc4py/SLEPc/LME.pyx:95 <slepc4py/SLEPc/LME.pyx#L95>`**Return type***Self***getBV()**

Get the basis vector object associated to the LME object.

Not collective.

Returns

The basis vectors context.

Return type*BV***See also**[setBV](#), [LMGetBV](#)**:sources:** `Source code at slepc4py/SLEPc/LME.pyx:638 <slepc4py/SLEPc/LME.pyx#L638>`**getCoefficients()**

Get the coefficient matrices of the matrix equation.

Collective.

Returns

- `A (petsc4py.PETSc.Mat)` – First coefficient matrix.
- `B (petsc4py.PETSc.Mat)` – Second coefficient matrix, if available.
- `D (petsc4py.PETSc.Mat)` – Third coefficient matrix, if available.
- `E (petsc4py.PETSc.Mat)` – Fourth coefficient matrix, if available.

Return type`tuple[Mat, Mat | None, Mat | None, Mat | None]`

See also

[*setCoefficients*](#), [*LMEGetCoefficients*](#)

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:264 <slepc4py/SLEPc/LME.pyx#L264>``

getConvergedReason()

Get the reason why the *solve()* iteration was stopped.

Not collective.

Returns

Negative value indicates diverged, positive value converged.

Return type

ConvergedReason

See also

[*setTolerances*](#), [*solve*](#), [*setErrorIfNotConverged*](#), [*LMEGetConvergedReason*](#)

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:780 <slepc4py/SLEPc/LME.pyx#L780>``

getDimensions()

Get the dimension of the subspace used by the solver.

Not collective.

Returns

Maximum dimension of the subspace to be used by the solver.

Return type

int

See also

[*setDimensions*](#), [*LMEGetDimensions*](#)

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:601 <slepc4py/SLEPc/LME.pyx#L601>``

getErrorEstimate()

Get the error estimate obtained during the solve.

Not collective.

Returns

The error estimate.

Return type

float

Notes

This is the error estimated internally by the solver. The actual error bound can be computed with [*computeError\(\)*](#). Note that some solvers may not be able to provide an error estimate.

See also

[*computeError*](#), [*LMEGetErrorEstimate*](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:410 <slepc4py/SLEPc/LME.pyx#L410>](#)

getErrorIfNotConverged()

Get if [*solve\(\)*](#) generates an error if the solver does not converge.

Not collective.

Get a flag indicating whether [*solve\(\)*](#) will generate an error if the solver does not converge.

Returns

True indicates you want the error generated.

Return type

[*bool*](#)

See also

[*setErrorIfNotConverged*](#), [*LMEGetErrorIfNotConverged*](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:823 <slepc4py/SLEPc/LME.pyx#L823>](#)

getIterationNumber()

Get the current iteration number.

Not collective.

If the call to [*solve\(\)*](#) is complete, then it returns the number of iterations carried out by the solution method.

Returns

Iteration number.

Return type

[*int*](#)

See also

[*getConvergedReason*](#), [*LMEGetIterationNumber*](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:758 <slepc4py/SLEPc/LME.pyx#L758>](#)

getMonitor()

Get the list of monitor functions.

Not collective.

Returns

The list of monitor functions.

Return type

[*LMEMonitorFunction*](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:700 <slepc4py/SLEPc/LME.pyx#L700>](#)

getOptionsPrefix()

Get the prefix used for searching for all LME options in the database.

Not collective.

Returns

The prefix string set for this LME object.

Return type

`str`

See also

`setOptionsPrefix`, `appendOptionsPrefix`, `LMEGetOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:462 <slepc4py/SLEPc/LME.pyx#L462>`

getProblemType()

Get the LME problem type of this object.

Not collective.

Returns

The problem type currently being used.

Return type

`ProblemType`

See also

`setProblemType`, `LMEGetProblemType`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:206 <slepc4py/SLEPc/LME.pyx#L206>`

getRHS()

Get the right-hand side of the matrix equation.

Collective.

Returns

`C` – The low-rank matrix.

Return type

`petsc4py.PETSc.Mat`

See also

`setRHS`, `LMEGetRHS`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:327 <slepc4py/SLEPc/LME.pyx#L327>`

getSolution()

Get the solution of the matrix equation.

Collective.

Returns

X – The low-rank matrix.

Return type

`petsc4py.PETSc.Mat`

Notes

If called after `solve()`, **X** will contain the solution of the equation.

The matrix **X** may have been passed by the user via `setSolution()`, although this is not required.

See also

`solve`, `setSolution`, `LMEGetSolution`

`:sources:` [Source code at slepc4py/SLEPc/LME.pyx:382 <slepc4py/SLEPc/LME.pyx#L382>](#)

getTolerances()

Get the tolerance and maximum iteration count.

Not collective.

Returns

- **tol** (`float`) – The convergence tolerance.
- **max_it** (`int`) – The maximum number of iterations.

Return type

`tuple[float, int]`

See also

`setTolerances`, `LMEGetTolerances`

`:sources:` [Source code at slepc4py/SLEPc/LME.pyx:553 <slepc4py/SLEPc/LME.pyx#L553>](#)

getType()

Get the LME type of this object.

Not collective.

Returns

The solver currently being used.

Return type

`str`

See also

`setType`, `LMEGetType`

`:sources:` [Source code at slepc4py/SLEPc/LME.pyx:169 <slepc4py/SLEPc/LME.pyx#L169>](#)

reset()

Reset the LME object.

Collective.

See also

[LMEReset](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:109](#) <[slepc4py/SLEPc/LME.pyx#L109](#)>

Return type

[None](#)

setBV(bv)

Set a basis vector object to the LME object.

Collective.

Parameters

bv ([BV](#)) – The basis vectors context.

Return type

[None](#)

See also

[getBV](#), [LMESetBV](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:658](#) <[slepc4py/SLEPc/LME.pyx#L658](#)>

setCoefficients(A, B=None, D=None, E=None)

Set the coefficient matrices.

Collective.

Set the coefficient matrices that define the linear matrix equation to be solved.

Parameters

- **A** ([Mat](#)) – First coefficient matrix
- **B** ([Mat](#) | [None](#)) – Second coefficient matrix, optional
- **D** ([Mat](#) | [None](#)) – Third coefficient matrix, optional
- **E** ([Mat](#) | [None](#)) – Fourth coefficient matrix, optional

Return type

[None](#)

Notes

The matrix equation takes the general form $AXE + DXB = C$, where matrix C is not provided here but with [setRHS\(\)](#). Not all four matrices must be passed.

This must be called before [setUp\(\)](#). If called again after [setUp\(\)](#) then the [LME](#) object is reset.

See also

[`getCoefficients`](#), [`solve`](#), [`setRHS`](#), [`setUp`](#), [`LMESetCoefficients`](#)

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:225 <slepc4py/SLEPc/LME.pyx#L225>``

setDimensions(*ncv*)

Set the dimension of the subspace to be used by the solver.

Logically collective.

Parameters

ncv (*int*) – Maximum dimension of the subspace to be used by the solver.

Return type

None

See also

[`getDimensions`](#), [`LMESetDimensions`](#)

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:620 <slepc4py/SLEPc/LME.pyx#L620>``

setErrorIfNotConverged(*flag=True*)

Set [`solve\(\)`](#) to generate an error if the solver has not converged.

Logically collective.

Parameters

flag (*bool*) – True indicates you want the error generated.

Return type

None

Notes

Normally SLEPc continues if the solver fails to converge, you can call [`getConvergedReason\(\)`](#) after a [`solve\(\)`](#) to determine if it has converged.

See also

[`getConvergedReason`](#), [`solve`](#), [`LMEsetErrorIfNotConverged`](#)

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:799 <slepc4py/SLEPc/LME.pyx#L799>``

setFromOptions()

Set LME options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before [`setUp\(\)`](#) if the user is to be allowed to set the solver type.

See also

[setOptionsPrefix](#), [LMESetFromOptions](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:534 <slepc4py/SLEPc/LME.pyx#L534>](#)

Return type

None

setMonitor(*monitor*, *args*=None, *kargs*=None)

Append a monitor function to the list of monitors.

Logically collective.

See also

[getMonitor](#), [cancelMonitor](#), [LMEMonitorSet](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:675 <slepc4py/SLEPc/LME.pyx#L675>](#)

Parameters

- **monitor** ([LMEMonitorFunction](#) | None)
- **args** ([tuple](#)[Any, ...] | None)
- **kargs** ([dict](#)[str, Any] | None)

Return type

None

setOptionsPrefix(*prefix*=None)

Set the prefix used for searching for all LME options in the database.

Logically collective.

Parameters

prefix ([str](#) | None) – The prefix string to prepend to all LME option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

For example, to distinguish between the runtime options for two different LME contexts, one could call:

```
L1.setOptionsPrefix("lme1_")
L2.setOptionsPrefix("lme2_")
```

See also

[appendOptionsPrefix](#), [getOptionsPrefix](#), [LMEGetOptionsPrefix](#)

:sources: [Source code at slepc4py/SLEPc/LME.pyx:481 <slepc4py/SLEPc/LME.pyx#L481>](#)

setProblemType(*lme_problem_type*)

Set the LME problem type of this object.

Logically collective.

Parameters

lme_problem_type (`ProblemType` / *str*) – The problem type to be used.

Return type

`None`

See also

`getProblemType`, `LMESetProblemType`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:188 <slepc4py/SLEPc/LME.pyx#L188>`

setRHS(*C*)

Set the right-hand side of the matrix equation.

Collective.

Parameters

C (*Mat*) – The right-hand side matrix

Return type

`None`

Notes

The matrix equation takes the general form $AXE + DXB = C$, where matrix C is given with this function. It must be a low-rank matrix of type `petsc4py.PETSc.Mat.Type.LRC`, that is, $C = UDV^*$ where D is diagonal of order k , and U, V are dense tall-skinny matrices with k columns. No sparse matrix must be provided when creating the LRC matrix.

In equation types that require C to be symmetric, such as Lyapunov, C must be created with $V = U$.

See also

`getRHS`, `setSolution`, `LMESetRHS`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:298 <slepc4py/SLEPc/LME.pyx#L298>`

setSolution(*X=None*)

Set the placeholder for the solution of the matrix equation.

Collective.

Parameters

X (*Mat* / *None*) – The solution matrix

Return type

`None`

Notes

The matrix equation takes the general form $AXE + DXB = C$, where the solution matrix is of low rank and is written in factored form $X = UDV^*$. This function provides a matrix object of type `petsc4py.PETSc.Mat.Type.LRC` that stores U, V and (optionally) D . These factors will be computed during `solve()`.

In equation types whose solution X is symmetric, such as Lyapunov, X must be created with $V = U$.

If the user provides X with this function, then the solver will return a solution with rank at most the number of columns of U . Alternatively, it is possible to let the solver choose the rank of the solution, by passing `None` and then calling `getSolution()` after `solve()`.

See also

`solve`, `setRHS`, `getSolution`, `LMESetSolution`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:347 <slepc4py/SLEPc/LME.pyx#L347>`

setTolerances(*tol=None, max_it=None*)

Set the tolerance and maximum iteration count.

Logically collective.

Set the tolerance and maximum iteration count used by the default LME convergence tests.

Parameters

- **tol** (*float* / *None*) – The convergence tolerance.
- **max_it** (*int* / *None*) – The maximum number of iterations.

Return type

None

See also

`getTolerances`, `LMESetTolerances`

:sources: `Source code at slepc4py/SLEPc/LME.pyx:575 <slepc4py/SLEPc/LME.pyx#L575>`

setType(*lme_type*)

Set the particular solver to be used in the LME object.

Logically collective.

Parameters

- **lme_type** (*Type* / *str*) – The solver to be used.

Return type

None

Notes

The default is KRYLOV. Normally, it is best to use `setFromOptions()` and then set the LME type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also

[getType](#), [LMESetType](#)

:sources: `Source code at slepc4py/SLEPc/LME.pyx:142 <slepc4py/SLEPc/LME.pyx#L142>`

setUp()

Set up all the internal necessary data structures.

Collective.

Set up all the internal data structures necessary for the execution of the eigensolver.

See also

[solve](#), [LMESetUp](#)

:sources: `Source code at slepc4py/SLEPc/LME.pyx:726 <slepc4py/SLEPc/LME.pyx#L726>`

Return type

`None`

solve()

Solve the linear matrix equation.

Collective.

Notes

The matrix coefficients are specified with [setCoefficients\(\)](#). The right-hand side is specified with [setRHS\(\)](#). The placeholder for the solution is specified with [setSolution\(\)](#).

See also

[setCoefficients](#), [setRHS](#), [setSolution](#), [LMEsolve](#)

:sources: `Source code at slepc4py/SLEPc/LME.pyx:741 <slepc4py/SLEPc/LME.pyx#L741>`

Return type

`None`

view(viewer=None)

Print the LME data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

`None`

See also
LMEView

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:76 <slepc4py/SLEPc/LME.pyx#L76>``

Attributes Documentation

bv

The basis vectors (*BV*) object associated to the LME object.

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:868 <slepc4py/SLEPc/LME.pyx#L868>``

fn

The math function (*FN*) object associated to the LME object.

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:861 <slepc4py/SLEPc/LME.pyx#L861>``

max_it

The maximum iteration count used by the LME convergence tests.

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:854 <slepc4py/SLEPc/LME.pyx#L854>``

tol

The tolerance value used by the LME convergence tests.

:sources: ``Source code at slepc4py/SLEPc/LME.pyx:847 <slepc4py/SLEPc/LME.pyx#L847>``

__init__()

classmethod **__new__**(*args, **kwargs)

slepc4py.SLEPc.MFN

class slepc4py.SLEPc.MFN

Bases: `Object`

Matrix Function.

Matrix Function (*MFN*) is the object provided by slepc4py for computing the action of a matrix function on a vector. Given a matrix A and a vector b , the call `mf.solve(b, x)` computes $x = f(A)b$, where f is a function such as the exponential.

Enumerations

<i>ConvergedReason</i>	MFN convergence reasons.
<i>Type</i>	MFN type.

slepc4py.SLEPc.MFN.ConvergedReason

class slepc4py.SLEPc.MFN.ConvergedReason

Bases: `object`

MFN convergence reasons.

- *CONVERGED_TOL*: All eigenpairs converged to requested tolerance.

- *CONVERGED_ITS*: Solver completed the requested number of steps.
- *DIVERGED_ITS*: Maximum number of iterations exceeded.
- *DIVERGED_BREAKDOWN*: Generic breakdown in method.

See also

`MFNConvergedReason`

Attributes Summary

<i>CONVERGED_ITERATING</i>	Constant <i>CONVERGED_ITERATING</i> of type <code>int</code>
<i>CONVERGED_ITS</i>	Constant <i>CONVERGED_ITS</i> of type <code>int</code>
<i>CONVERGED_TOL</i>	Constant <i>CONVERGED_TOL</i> of type <code>int</code>
<i>DIVERGED_BREAKDOWN</i>	Constant <i>DIVERGED_BREAKDOWN</i> of type <code>int</code>
<i>DIVERGED_ITS</i>	Constant <i>DIVERGED_ITS</i> of type <code>int</code>
<i>ITERATING</i>	Constant <i>ITERATING</i> of type <code>int</code>

Attributes Documentation

CONVERGED_ITERATING: `int` = *CONVERGED_ITERATING*

Constant *CONVERGED_ITERATING* of type `int`

CONVERGED_ITS: `int` = *CONVERGED_ITS*

Constant *CONVERGED_ITS* of type `int`

CONVERGED_TOL: `int` = *CONVERGED_TOL*

Constant *CONVERGED_TOL* of type `int`

DIVERGED_BREAKDOWN: `int` = *DIVERGED_BREAKDOWN*

Constant *DIVERGED_BREAKDOWN* of type `int`

DIVERGED_ITS: `int` = *DIVERGED_ITS*

Constant *DIVERGED_ITS* of type `int`

ITERATING: `int` = *ITERATING*

Constant *ITERATING* of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

`slepc4py.SLEPc.MFN.Type`

class `slepc4py.SLEPc.MFN.Type`

Bases: `object`

MFN type.

- *KRYLOV*: Restarted Krylov solver.
- *EXPOKIT*: Implementation of the method in Expokit.

See also
MFNType

Attributes Summary

<i>EXPOKIT</i>	Object EXPOKIT of type <i>str</i>
<i>KRYLOV</i>	Object KRYLOV of type <i>str</i>

Attributes Documentation

EXPOKIT: *str* = EXPOKIT

Object EXPOKIT of type *str*

KRYLOV: *str* = KRYLOV

Object KRYLOV of type *str*

__init__()

classmethod **__new__**(*args, **kwargs)

Methods Summary

<i>appendOptionsPrefix</i> ([prefix])	Append to the prefix used for searching for all MFN options in the database.
<i>cancelMonitor</i> ()	Clear all monitors for an <i>MFN</i> object.
<i>create</i> ([comm])	Create the MFN object.
<i>destroy</i> ()	Destroy the MFN object.
<i>getBV</i> ()	Get the basis vector object associated to the MFN object.
<i>getConvergedReason</i> ()	Get the reason why the <i>solve()</i> iteration was stopped.
<i>getDimensions</i> ()	Get the dimension of the subspace used by the solver.
<i>getErrorIfNotConverged</i> ()	Get if <i>solve()</i> generates an error if the solver does not converge.
<i>getFN</i> ()	Get the math function object associated to the MFN object.
<i>getIterationNumber</i> ()	Get the current iteration number.
<i>getMonitor</i> ()	Get the list of monitor functions.
<i>getOperator</i> ()	Get the matrix associated with the MFN object.
<i>getOptionsPrefix</i> ()	Get the prefix used for searching for all MFN options in the database.
<i>getTolerances</i> ()	Get the tolerance and maximum iteration count.
<i>getType</i> ()	Get the MFN type of this object.
<i>reset</i> ()	Reset the MFN object.
<i>setBV</i> (bv)	Set a basis vector object associated to the MFN object.
<i>setDimensions</i> (ncv)	Set the dimension of the subspace to be used by the solver.
<i>setErrorIfNotConverged</i> ([flg])	Set <i>solve()</i> to generate an error if the solver does not converge.

continues on next page

Table 53 – continued from previous page

<code>setFN(fn)</code>	Set a math function object associated to the MFN object.
<code>setFromOptions()</code>	Set MFN options from the options database.
<code>setMonitor(monitor[, args, kargs])</code>	Append a monitor function to the list of monitors.
<code>setOperator(A)</code>	Set the matrix associated with the MFN object.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all MFN options in the database.
<code>setTolerances([tol, max_it])</code>	Set the tolerance and maximum iteration count.
<code>setType(mfn_type)</code>	Set the particular solver to be used in the MFN object.
<code>setUp()</code>	Set up all the necessary internal data structures.
<code>solve(b, x)</code>	Solve the matrix function problem.
<code>solveTranspose(b, x)</code>	Solve the transpose matrix function problem.
<code>view([viewer])</code>	Print the MFN data structure.

Attributes Summary

<code>bv</code>	The basis vectors (<i>BV</i>) object associated to the MFN object.
<code>fn</code>	The math function (<i>FN</i>) object associated to the MFN object.
<code>max_it</code>	The maximum iteration count used by the MFN convergence tests.
<code>tol</code>	The tolerance count used by the MFN convergence tests.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching for all MFN options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all MFN option requests.

Return type

None

See also

`setOptionsPrefix`, `getOptionsPrefix`, `MFNAppendOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:219 <slepc4py/SLEPc/MFN.pyx#L219>`

`cancelMonitor()`

Clear all monitors for an *MFN* object.

Logically collective.

See also
<code>MFNMonitorCancel</code>

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:498 <slepc4py/SLEPc/MFN.pyx#L498>`

Return type

`None`

create(*comm=None*)

Create the MFN object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator. If not provided, it defaults to all processes.

Return type

Self

See also
<code>MFNCreate</code>

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:102 <slepc4py/SLEPc/MFN.pyx#L102>`

destroy()

Destroy the MFN object.

Logically collective.

See also
<code>MFNDestroy</code>

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:76 <slepc4py/SLEPc/MFN.pyx#L76>`

Return type

Self

getBV()

Get the basis vector object associated to the MFN object.

Not collective.

Returns

The basis vectors context.

Return type

BV

See also
<code>setBV</code> , <code>MFNGetBV</code>

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:379 <slepc4py/SLEPc/MFN.pyx#L379>`

getConvergedReason()

Get the reason why the *solve()* iteration was stopped.

Not collective.

Returns

Negative value indicates diverged, positive value converged.

Return type

ConvergedReason

See also

setTolerances, *solve*, *setErrorIfNotConverged*, *MFNGetConvergedReason*

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:607 <slepc4py/SLEPc/MFN.pyx#L607>](#)

getDimensions()

Get the dimension of the subspace used by the solver.

Not collective.

Returns

Maximum dimension of the subspace to be used by the solver.

Return type

int

See also

setDimensions, *MFNGetDimensions*

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:305 <slepc4py/SLEPc/MFN.pyx#L305>](#)

getErrorIfNotConverged()

Get if *solve()* generates an error if the solver does not converge.

Not collective.

Get a flag indicating whether *solve()* will generate an error if the solver does not converge.

Returns

True indicates you want the error generated.

Return type

bool

See also

setErrorIfNotConverged, *MFNGetErrorIfNotConverged*

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:650 <slepc4py/SLEPc/MFN.pyx#L650>](#)

getFN()

Get the math function object associated to the MFN object.

Not collective.

Returns

The math function context.

Return type

FN

See also

setFN, *MFNGetFN*

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:342 <slepc4py/SLEPc/MFN.pyx#L342>`

getIterationNumber()

Get the current iteration number.

Not collective.

Get the current iteration number. If the call to *solve()* is complete, then it returns the number of iterations carried out by the solution method.

Returns

Iteration number.

Return type

int

See also

getConvergedReason, *MFNGetIterationNumber*

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:584 <slepc4py/SLEPc/MFN.pyx#L584>`

getMonitor()

Get the list of monitor functions.

Not collective.

Returns

The list of monitor functions.

Return type

MFNMonitorFunction

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:485 <slepc4py/SLEPc/MFN.pyx#L485>`

getOperator()

Get the matrix associated with the MFN object.

Collective.

Returns

The matrix for which the matrix function is to be computed.

Return type

petsc4py.PETSc.Mat

See also

`setOperator`, `MFNGetOperator`

`:sources:`Source code at slepc4py/SLEPc/MFN.pyx:416 <slepc4py/SLEPc/MFN.pyx#L416>``

getOptionsPrefix()

Get the prefix used for searching for all MFN options in the database.

Not collective.

Returns

The prefix string set for this MFN object.

Return type

`str`

See also

`setOptionsPrefix`, `appendOptionsPrefix`, `MFNGetOptionsPrefix`

`:sources:`Source code at slepc4py/SLEPc/MFN.pyx:169 <slepc4py/SLEPc/MFN.pyx#L169>``

getTolerances()

Get the tolerance and maximum iteration count.

Not collective.

Returns

- `tol` (`float`) – The convergence tolerance.
- `max_it` (`int`) – The maximum number of iterations.

Return type

`tuple[float, int]`

See also

`setTolerances`, `MFNGetTolerances`

`:sources:`Source code at slepc4py/SLEPc/MFN.pyx:257 <slepc4py/SLEPc/MFN.pyx#L257>``

getType()

Get the MFN type of this object.

Not collective.

Returns

The solver currently being used.

Return type

`str`

See also
setType , MFNGetType

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:150 <slepc4py/SLEPc/MFN.pyx#L150>](#)

reset()

Reset the MFN object.

Collective.

See also
MFNReset

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:90 <slepc4py/SLEPc/MFN.pyx#L90>](#)

Return type

None

setBV(*bv*)

Set a basis vector object associated to the MFN object.

Collective.

Parameters

bv (*BV*) – The basis vectors context.

Return type

None

See also
getBV , MFNSetBV

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:399 <slepc4py/SLEPc/MFN.pyx#L399>](#)

setDimensions(*ncv*)

Set the dimension of the subspace to be used by the solver.

Logically collective.

Parameters

ncv (*int*) – Maximum dimension of the subspace to be used by the solver.

Return type

None

See also
getDimensions , MFNSetDimensions

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:324 <slepc4py/SLEPc/MFN.pyx#L324>](#)

setErrorIfNotConverged(*flag=True*)

Set `solve()` to generate an error if the solver does not converge.

Logically collective.

Parameters

flag (*bool*) – True indicates you want the error generated.

Return type

None

Notes

Normally SLEPc continues if the solver fails to converge, you can call `getConvergedReason()` after a `solve()` to determine if it has converged.

See also

`getConvergedReason`, `solve`, `MFNSetErrorIfNotConverged`

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:626 <slepc4py/SLEPc/MFN.pyx#L626>`

setFN(*fn*)

Set a math function object associated to the MFN object.

Collective.

Parameters

fn (*FN*) – The math function context.

Return type

None

See also

`getFN`, `MFNSetFN`

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:362 <slepc4py/SLEPc/MFN.pyx#L362>`

setFromOptions()

Set MFN options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before `setUp()` if the user is to be allowed to set the solver type.

See also

`setOptionsPrefix`, `MFNSetFromOptions`

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:238 <slepc4py/SLEPc/MFN.pyx#L238>`

Return type

None

setMonitor(*monitor*, *args=None*, *kargs=None*)

Append a monitor function to the list of monitors.

Logically collective.

See also

[*getMonitor*](#), [*cancelMonitor*](#), [*MFNMonitorSet*](#)

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:460](#) <[slepc4py/SLEPc/MFN.pyx#L460](#)>

Parameters

- **monitor** ([*MFNMonitorFunction*](#) / *None*)
- **args** ([*tuple*](#)[*Any*, ...] / *None*)
- **kargs** ([*dict*](#)[*str*, *Any*] / *None*)

Return type

None

setOperator(*A*)

Set the matrix associated with the MFN object.

Collective.

Parameters

A ([*Mat*](#)) – The problem matrix.

Return type

None

Notes

This must be called before [*setUp\(\)*](#). If called again after [*setUp\(\)*](#) then the *MFN* object is reset.

See also

[*getOperator*](#), [*MFNSetOperator*](#)

:sources: [Source code at slepc4py/SLEPc/MFN.pyx:436](#) <[slepc4py/SLEPc/MFN.pyx#L436](#)>

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all MFN options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all MFN option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

For example, to distinguish between the runtime options for two different MFN contexts, one could call:

```
M1.setOptionsPrefix("mfn1_")
M2.setOptionsPrefix("mfn2_")
```

See also

[appendOptionsPrefix](#), [getOptionsPrefix](#), [MFNGetOptionsPrefix](#)

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:188 <slepc4py/SLEPc/MFN.pyx#L188>`

setTolerances(*tol=None, max_it=None*)

Set the tolerance and maximum iteration count.

Logically collective.

Set the tolerance and maximum iteration count used by the default MFN convergence tests.

Parameters

- **tol** (*float* / *None*) – The convergence tolerance.
- **max_it** (*int* / *None*) – The maximum number of iterations.

Return type

None

See also

[getTolerances](#), [MFNSetTolerances](#)

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:279 <slepc4py/SLEPc/MFN.pyx#L279>`

setType(*mfn_type*)

Set the particular solver to be used in the MFN object.

Logically collective.

Parameters

- **mfn_type** (*Type* / *str*) – The solver to be used.

Return type

None

Notes

The default is KRYLOV. Normally, it is best to use [setFromOptions\(\)](#) and then set the MFN type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also[*getType*](#), [*MFNSetType*](#)**:sources:** ``Source code at slepc4py/SLEPc/MFN.pyx:123 <slepc4py/SLEPc/MFN.pyx#L123>``**setUp()**

Set up all the necessary internal data structures.

Collective.

Set up all the internal data structures necessary for the execution of the eigensolver.

See also[*solve*](#), [*MFNSetUp*](#)**:sources:** ``Source code at slepc4py/SLEPc/MFN.pyx:513 <slepc4py/SLEPc/MFN.pyx#L513>``**Return type**[*None*](#)**solve(*b*, *x*)**

Solve the matrix function problem.

Collective.

Given a vector *b*, the vector $x = f(A)b$ is returned.

Parameters

- **b** ([*Vec*](#)) – The right hand side vector.
- **x** ([*Vec*](#)) – The solution.

Return type[*None*](#)**Notes**

The matrix *A* is specified with [*setOperator\(\)*](#). The function *f* is specified via the [*FN*](#) object obtained with [*getFN\(\)*](#) or set with [*setFN\(\)*](#).

See also[*setOperator*](#), [*getFN*](#), [*solveTranspose*](#), [*MFNSolve*](#)**:sources:** ``Source code at slepc4py/SLEPc/MFN.pyx:528 <slepc4py/SLEPc/MFN.pyx#L528>``**solveTranspose(*b*, *x*)**

Solve the transpose matrix function problem.

Collective.

Given a vector *b*, the vector $x = f(A^T)b$ is returned.

Parameters

- **b** ([*Vec*](#)) – The right hand side vector.

- \mathbf{x} (*Vec*) – The solution.

Return type

None

Notes

The matrix A is specified with `setOperator()`. The function f is specified via the *FN* object obtained with `getFN()` or set with `setFN()`.

See also

`setOperator`, `getFN`, `solve`, `MFNSolveTranspose`

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:556 <slepc4py/SLEPc/MFN.pyx#L556>`

view(*viewer=None*)

Print the MFN data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

`MFNView`

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:57 <slepc4py/SLEPc/MFN.pyx#L57>`

Attributes Documentation

bv

The basis vectors (*BV*) object associated to the MFN object.

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:695 <slepc4py/SLEPc/MFN.pyx#L695>`

fn

The math function (*FN*) object associated to the MFN object.

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:688 <slepc4py/SLEPc/MFN.pyx#L688>`

max_it

The maximum iteration count used by the MFN convergence tests.

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:681 <slepc4py/SLEPc/MFN.pyx#L681>`

tol

The tolerance count used by the MFN convergence tests.

:sources: `Source code at slepc4py/SLEPc/MFN.pyx:674 <slepc4py/SLEPc/MFN.pyx#L674>`

__init__()

classmethod `__new__`(*args, **kwargs)

slepc4py.SLEPc.NEP

class `slepc4py.SLEPc.NEP`

Bases: `Object`

Nonlinear Eigenvalue Problem Solver.

The Nonlinear Eigenvalue Problem ([NEP](#)) solver is the object provided by `slepc4py` for specifying an eigenvalue problem that is nonlinear with respect to the eigenvalue (not the eigenvector). This is intended for general nonlinear problems (rather than polynomial eigenproblems) described as $T(\lambda)x = 0$.

Enumerations

CISSExtraction	NEP CISS extraction technique.
Conv	NEP convergence test.
ConvergedReason	NEP convergence reasons.
ErrorType	NEP error type to assess accuracy of computed solutions.
ProblemType	NEP problem type.
Refine	NEP refinement strategy.
RefineScheme	NEP scheme for solving linear systems during iterative refinement.
Stop	NEP stopping test.
Type	NEP type.
Which	NEP desired part of spectrum.

slepc4py.SLEPc.NEP.CISSExtraction

class `slepc4py.SLEPc.NEP.CISSExtraction`

Bases: `object`

NEP CISS extraction technique.

- [RITZ](#): Ritz extraction.
- [HANKEL](#): Extraction via Hankel eigenproblem.
- [CAA](#): Communication-avoiding Arnoldi.

See also

[NEPCISSExtraction](#)

Attributes Summary

CAA	Constant CAA of type <code>int</code>
HANKEL	Constant HANKEL of type <code>int</code>
RITZ	Constant RITZ of type <code>int</code>

Attributes Documentation

CAA: `int` = CAA
Constant CAA of type `int`

HANKEL: `int` = HANKEL
Constant HANKEL of type `int`

RITZ: `int` = RITZ
Constant RITZ of type `int`

`__init__()`

`classmethod` `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.Conv

class `slepc4py.SLEPc.NEP.Conv`
Bases: `object`
NEP convergence test.

- *ABS*: Absolute convergence test.
- *REL*: Convergence test relative to the eigenvalue.
- *NORM*: Convergence test relative to the matrix norms.
- *USER*: User-defined convergence test.

See also

NEPConv

Attributes Summary

<i>ABS</i>	Constant ABS of type <code>int</code>
<i>NORM</i>	Constant NORM of type <code>int</code>
<i>REL</i>	Constant REL of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

ABS: `int` = ABS
Constant ABS of type `int`

NORM: `int` = NORM
Constant NORM of type `int`

REL: `int` = REL
Constant REL of type `int`

USER: `int` = USER
Constant USER of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.ConvergedReason

class `slepc4py.SLEPc.NEP.ConvergedReason`

Bases: `object`

NEP convergence reasons.

- *CONVERGED_TOL*: All eigenpairs converged to requested tolerance.
- *CONVERGED_USER*: User-defined convergence criterion satisfied.
- *DIVERGED_ITS*: Maximum number of iterations exceeded.
- *DIVERGED_BREAKDOWN*: Solver failed due to breakdown.
- *DIVERGED_LINEAR_SOLVE*: Inner linear solve failed.
- *DIVERGED_SUBSPACE_EXHAUSTED*: Run out of space for the basis in an unrestarted solver.
- *CONVERGED_ITERATING*: Iteration not finished yet.

See also

`NEPConvergedReason`

Attributes Summary

<i>CONVERGED_ITERATING</i>	Constant <i>CONVERGED_ITERATING</i> of type <code>int</code>
<i>CONVERGED_TOL</i>	Constant <i>CONVERGED_TOL</i> of type <code>int</code>
<i>CONVERGED_USER</i>	Constant <i>CONVERGED_USER</i> of type <code>int</code>
<i>DIVERGED_BREAKDOWN</i>	Constant <i>DIVERGED_BREAKDOWN</i> of type <code>int</code>
<i>DIVERGED_ITS</i>	Constant <i>DIVERGED_ITS</i> of type <code>int</code>
<i>DIVERGED_LINEAR_SOLVE</i>	Constant <i>DIVERGED_LINEAR_SOLVE</i> of type <code>int</code>
<i>DIVERGED_SUBSPACE_EXHAUSTED</i>	Constant <i>DIVERGED_SUBSPACE_EXHAUSTED</i> of type <code>int</code>
<i>ITERATING</i>	Constant <i>ITERATING</i> of type <code>int</code>

Attributes Documentation

CONVERGED_ITERATING: `int` = *CONVERGED_ITERATING*

Constant *CONVERGED_ITERATING* of type `int`

CONVERGED_TOL: `int` = *CONVERGED_TOL*

Constant *CONVERGED_TOL* of type `int`

CONVERGED_USER: `int` = *CONVERGED_USER*

Constant *CONVERGED_USER* of type `int`

DIVERGED_BREAKDOWN: `int` = *DIVERGED_BREAKDOWN*

Constant *DIVERGED_BREAKDOWN* of type `int`

DIVERGED_ITS: `int` = *DIVERGED_ITS*

Constant *DIVERGED_ITS* of type `int`

DIVERGED_LINEAR_SOLVE: `int` = **DIVERGED_LINEAR_SOLVE**
Constant **DIVERGED_LINEAR_SOLVE** of type `int`

DIVERGED_SUBSPACE_EXHAUSTED: `int` = **DIVERGED_SUBSPACE_EXHAUSTED**
Constant **DIVERGED_SUBSPACE_EXHAUSTED** of type `int`

ITERATING: `int` = **ITERATING**
Constant **ITERATING** of type `int`

`__init__()`

`classmethod` `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.ErrorType

class `slepc4py.SLEPc.NEP.ErrorType`
Bases: `object`
NEP error type to assess accuracy of computed solutions.

- *ABSOLUTE*: Absolute error.
- *RELATIVE*: Relative error.
- *BACKWARD*: Backward error.

See also
<code>NEPErrorType</code>

Attributes Summary

<i>ABSOLUTE</i>	Constant ABSOLUTE of type <code>int</code>
<i>BACKWARD</i>	Constant BACKWARD of type <code>int</code>
<i>RELATIVE</i>	Constant RELATIVE of type <code>int</code>

Attributes Documentation

ABSOLUTE: `int` = **ABSOLUTE**
Constant **ABSOLUTE** of type `int`

BACKWARD: `int` = **BACKWARD**
Constant **BACKWARD** of type `int`

RELATIVE: `int` = **RELATIVE**
Constant **RELATIVE** of type `int`

`__init__()`

`classmethod` `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.ProblemType

class slepc4py.SLEPc.NEP.ProblemType

Bases: `object`

NEP problem type.

- *GENERAL*: General nonlinear eigenproblem.
- *RATIONAL*: NEP defined in split form with all f_i rational.

See also
<code>NEPProblemType</code>

Attributes Summary

<i>GENERAL</i>	Constant <i>GENERAL</i> of type <code>int</code>
<i>RATIONAL</i>	Constant <i>RATIONAL</i> of type <code>int</code>

Attributes Documentation

GENERAL: `int` = *GENERAL*
Constant *GENERAL* of type `int`

RATIONAL: `int` = *RATIONAL*
Constant *RATIONAL* of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.Refine

class slepc4py.SLEPc.NEP.Refine

Bases: `object`

NEP refinement strategy.

- *NONE*: No refinement.
- *SIMPLE*: Refine eigenpairs one by one.
- *MULTIPLE*: Refine all eigenpairs simultaneously (invariant pair).

See also
<code>NEPRefine</code>

Attributes Summary

<i>MULTIPLE</i>	Constant <i>MULTIPLE</i> of type <code>int</code>
<i>NONE</i>	Constant <i>NONE</i> of type <code>int</code>

continues on next page

Table 61 – continued from previous page

<i>SIMPLE</i>	Constant SIMPLE of type <code>int</code>
---------------	--

Attributes Documentation

MULTIPLE: `int` = **MULTIPLE**

Constant MULTIPLE of type `int`

NONE: `int` = **NONE**

Constant NONE of type `int`

SIMPLE: `int` = **SIMPLE**

Constant SIMPLE of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

`slepc4py.SLEPc.NEP.RefineScheme`

class `slepc4py.SLEPc.NEP.RefineScheme`

Bases: `object`

NEP scheme for solving linear systems during iterative refinement.

- *SCHUR*: Schur complement.
- *MBE*: Mixed block elimination.
- *EXPLICIT*: Build the explicit matrix.

See also

`NEPRefineScheme`

Attributes Summary

<i>EXPLICIT</i>	Constant EXPLICIT of type <code>int</code>
<i>MBE</i>	Constant MBE of type <code>int</code>
<i>SCHUR</i>	Constant SCHUR of type <code>int</code>

Attributes Documentation

EXPLICIT: `int` = **EXPLICIT**

Constant EXPLICIT of type `int`

MBE: `int` = **MBE**

Constant MBE of type `int`

SCHUR: `int` = **SCHUR**

Constant SCHUR of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.Stop

class slepc4py.SLEPc.NEP.Stop

Bases: `object`

NEP stopping test.

- *BASIC*: Default stopping test.
- *USER*: User-defined stopping test.

See also

NEPStop

Attributes Summary

<i>BASIC</i>	Constant BASIC of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

BASIC: `int` = BASIC

Constant BASIC of type `int`

USER: `int` = USER

Constant USER of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.NEP.Type

class slepc4py.SLEPc.NEP.Type

Bases: `object`

NEP type.

- *RII*: Residual inverse iteration.
- *SLP*: Successive linear problems.
- *NARNOLDI*: Nonlinear Arnoldi.
- *NLEIGS*: Fully rational Krylov method for nonlinear eigenproblems.
- *CISS*: Contour integral spectrum slice.
- *INTERPOL*: Polynomial interpolation.

See also

NEPType

Attributes Summary

<i>CISS</i>	Object CISS of type <i>str</i>
<i>INTERPOL</i>	Object INTERPOL of type <i>str</i>
<i>NARNOLDI</i>	Object NARNOLDI of type <i>str</i>
<i>NLEIGS</i>	Object NLEIGS of type <i>str</i>
<i>RII</i>	Object RII of type <i>str</i>
<i>SLP</i>	Object SLP of type <i>str</i>

Attributes Documentation

CISS: *str* = CISS

Object CISS of type *str*

INTERPOL: *str* = INTERPOL

Object INTERPOL of type *str*

NARNOLDI: *str* = NARNOLDI

Object NARNOLDI of type *str*

NLEIGS: *str* = NLEIGS

Object NLEIGS of type *str*

RII: *str* = RII

Object RII of type *str*

SLP: *str* = SLP

Object SLP of type *str*

__init__()

classmethod **__new__**(*args, **kwargs)

slepc4py.SLEPc.NEP.Which

class slepc4py.SLEPc.NEP.**Which**

Bases: *object*

NEP desired part of spectrum.

- *LARGEST_MAGNITUDE*: Largest magnitude (default).
- *SMALLEST_MAGNITUDE*: Smallest magnitude.
- *LARGEST_REAL*: Largest real parts.
- *SMALLEST_REAL*: Smallest real parts.
- *LARGEST_IMAGINARY*: Largest imaginary parts in magnitude.
- *SMALLEST_IMAGINARY*: Smallest imaginary parts in magnitude.
- *TARGET_MAGNITUDE*: Closest to target (in magnitude).
- *TARGET_REAL*: Real part closest to target.
- *TARGET_IMAGINARY*: Imaginary part closest to target.
- *ALL*: All eigenvalues in a region.
- *USER*: User defined selection.

See also
NEPWhich

Attributes Summary

<i>ALL</i>	Constant ALL of type <code>int</code>
<i>LARGEST_IMAGINARY</i>	Constant LARGEST_IMAGINARY of type <code>int</code>
<i>LARGEST_MAGNITUDE</i>	Constant LARGEST_MAGNITUDE of type <code>int</code>
<i>LARGEST_REAL</i>	Constant LARGEST_REAL of type <code>int</code>
<i>SMALLEST_IMAGINARY</i>	Constant SMALLEST_IMAGINARY of type <code>int</code>
<i>SMALLEST_MAGNITUDE</i>	Constant SMALLEST_MAGNITUDE of type <code>int</code>
<i>SMALLEST_REAL</i>	Constant SMALLEST_REAL of type <code>int</code>
<i>TARGET_IMAGINARY</i>	Constant TARGET_IMAGINARY of type <code>int</code>
<i>TARGET_MAGNITUDE</i>	Constant TARGET_MAGNITUDE of type <code>int</code>
<i>TARGET_REAL</i>	Constant TARGET_REAL of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

ALL: `int` = ALL

Constant ALL of type `int`

LARGEST_IMAGINARY: `int` = LARGEST_IMAGINARY

Constant LARGEST_IMAGINARY of type `int`

LARGEST_MAGNITUDE: `int` = LARGEST_MAGNITUDE

Constant LARGEST_MAGNITUDE of type `int`

LARGEST_REAL: `int` = LARGEST_REAL

Constant LARGEST_REAL of type `int`

SMALLEST_IMAGINARY: `int` = SMALLEST_IMAGINARY

Constant SMALLEST_IMAGINARY of type `int`

SMALLEST_MAGNITUDE: `int` = SMALLEST_MAGNITUDE

Constant SMALLEST_MAGNITUDE of type `int`

SMALLEST_REAL: `int` = SMALLEST_REAL

Constant SMALLEST_REAL of type `int`

TARGET_IMAGINARY: `int` = TARGET_IMAGINARY

Constant TARGET_IMAGINARY of type `int`

TARGET_MAGNITUDE: `int` = TARGET_MAGNITUDE

Constant TARGET_MAGNITUDE of type `int`

TARGET_REAL: `int` = TARGET_REAL

Constant TARGET_REAL of type `int`

USER: `int` = USER

Constant USER of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

Methods Summary

<code>appendOptionsPrefix([prefix])</code>	Append to the prefix used for searching for all NEP options in the database.
<code>applyResolvent(omega, v, r[, rg])</code>	Apply the resolvent $T^{-1}(z)$ to a given vector.
<code>cancelMonitor()</code>	Clear all monitors for a NEP object.
<code>computeError(i[, etype])</code>	Compute the error associated with the i-th computed eigenpair.
<code>create([comm])</code>	Create the NEP object.
<code>destroy()</code>	Destroy the NEP object.
<code>errorView([etype, viewer])</code>	Display the errors associated with the computed solution.
<code>getBV()</code>	Get the basis vectors object associated to the eigensolver.
<code>getCISSExtraction()</code>	Get the extraction technique used in the CISS solver.
<code>getCISSKSPs()</code>	Get the list of linear solver objects associated with the CISS solver.
<code>getCISSRefinement()</code>	Get the values of various refinement parameters in the CISS solver.
<code>getCISSSizes()</code>	Get the values of various size parameters in the CISS solver.
<code>getCISSThreshold()</code>	Get the values of various threshold parameters in the CISS solver.
<code>getConverged()</code>	Get the number of converged eigenpairs.
<code>getConvergedReason()</code>	Get the reason why the solve() iteration was stopped.
<code>getConvergenceTest()</code>	Get the method used to compute the error estimate used in the convergence test.
<code>getDS()</code>	Get the direct solver associated to the eigensolver.
<code>getDimensions()</code>	Get the number of eigenvalues to compute.
<code>getEigenpair(i[, Vr, Vi])</code>	Get the i-th solution of the eigenproblem as computed by solve() .
<code>getEigenvalueComparison()</code>	Get the eigenvalue comparison function.
<code>getErrorEstimate(i)</code>	Get the error estimate associated to the i-th computed eigenpair.
<code>getFunction()</code>	Get the function to compute the nonlinear Function $T(\lambda)$.
<code>getInterpolInterpolation()</code>	Get the tolerance and maximum degree for the interpolation polynomial.
<code>getInterpolPEP()</code>	Get the associated polynomial eigensolver object.
<code>getIterationNumber()</code>	Get the current iteration number.
<code>getJacobian()</code>	Get the function to compute the Jacobian $T'(\lambda)$ and J.
<code>getLeftEigenvector(i, Wr[, Wi])</code>	Get the i-th left eigenvector as computed by solve() .
<code>getMonitor()</code>	Get the list of monitor functions.
<code>getArnoldiKSP()</code>	Get the linear solver object associated with the nonlinear eigensolver.
<code>getArnoldiLagPreconditioner()</code>	Get how often the preconditioner is rebuilt.
<code>getNLEIGSEPS()</code>	Get the linear eigensolver object associated with the nonlinear eigensolver.
<code>getNLEIGSFullBasis()</code>	Get the flag that indicates if NLEIGS is using the full-basis variant.

continues on next page

Table 66 – continued from previous page

<i>getNLEIGSInterpolation()</i>	Get the tolerance and maximum degree for the interpolation polynomial.
<i>getNLEIGSKSPs()</i>	Get the list of linear solver objects associated with the NLEIGS solver.
<i>getNLEIGSLocking()</i>	Get the locking flag used in the NLEIGS method.
<i>getNLEIGSRKShifts()</i>	Get the list of shifts used in the Rational Krylov method.
<i>getNLEIGSRestart()</i>	Get the restart parameter used in the NLEIGS method.
<i>getOptionsPrefix()</i>	Get the prefix used for searching for all NEP options in the database.
<i>getProblemType()</i>	Get the problem type from the NEP object.
<i>getRG()</i>	Get the region object associated to the eigensolver.
<i>getRIIConstCorrectionTol()</i>	Get the constant tolerance flag.
<i>getRIIDeflationThreshold()</i>	Get the threshold value that controls deflation.
<i>getRIIHermitian()</i>	Get if the Hermitian version must be used by the solver.
<i>getRIIKSP()</i>	Get the linear solver object associated with the non-linear eigensolver.
<i>getRIILagPreconditioner()</i>	Get how often the preconditioner is rebuilt.
<i>getRIIMaximumIterations()</i>	Get the maximum number of inner iterations of RII.
<i>getRefine()</i>	Get the refinement strategy used by the NEP object.
<i>getRefineKSP()</i>	Get the KSP object used by the eigensolver in the refinement phase.
<i>getSLPDeflationThreshold()</i>	Get the threshold value that controls deflation.
<i>getSLPEPS()</i>	Get the linear eigensolver object associated with the nonlinear eigensolver.
<i>getSLPEPSLeft()</i>	Get the left eigensolver.
<i>getSLPKSP()</i>	Get the linear solver object associated with the non-linear eigensolver.
<i>getSplitOperator()</i>	Get the operator of the nonlinear eigenvalue problem in split form.
<i>getSplitPreconditioner()</i>	Get the operator of the split preconditioner.
<i>getStoppingTest()</i>	Get the stopping test function.
<i>getTarget()</i>	Get the value of the target.
<i>getTolerances()</i>	Get the tolerance and maximum iteration count.
<i>getTrackAll()</i>	Get the flag indicating whether all residual norms must be computed.
<i>getTwoSided()</i>	Get the flag indicating if a two-sided variant is being used.
<i>getType()</i>	Get the NEP type of this object.
<i>getWhichEigenpairs()</i>	Get which portion of the spectrum is to be sought.
<i>reset()</i>	Reset the NEP object.
<i>setBV(bv)</i>	Set the basis vectors object associated to the eigensolver.
<i>setCISSExtraction(extraction)</i>	Set the extraction technique used in the CISS solver.
<i>setCISSRefinement([inner, blsize])</i>	Set the values of various refinement parameters in the CISS solver.
<i>setCISSSizes([ip, bs, ms, npart, bsmx, ...])</i>	Set the values of various size parameters in the CISS solver.
<i>setCISSThreshold([delta, spur])</i>	Set the values of various threshold parameters in the CISS solver.

continues on next page

Table 66 – continued from previous page

<code>setConvergenceTest(conv)</code>	Set how to compute the error estimate used in the convergence test.
<code>setDS(ds)</code>	Set a direct solver object associated to the eigensolver.
<code>setDimensions([nev, ncv, mpd])</code>	Set the number of eigenvalues to compute.
<code>setEigenvalueComparison(comparison[, args, ...])</code>	Set an eigenvalue comparison function.
<code>setFromOptions()</code>	Set NEP options from the options database.
<code>setFunction(function[, F, P, args, kargs])</code>	Set the function to compute the nonlinear Function $T(\lambda)$.
<code>setInitialSpace(space)</code>	Set the initial space from which the eigensolver starts to iterate.
<code>setInterpolInterpolation([tol, deg])</code>	Set the tolerance and maximum degree for the interpolation polynomial.
<code>setInterpolPEP(pep)</code>	Set a polynomial eigensolver object associated to the nonlinear eigensolver.
<code>setJacobian(jacobian[, J, args, kargs])</code>	Set the function to compute the Jacobian $T'(\lambda)$.
<code>setMonitor(monitor[, args, kargs])</code>	Append a monitor function to the list of monitors.
<code>setArnoldiKSP(ksp)</code>	Set a linear solver object associated to the nonlinear eigensolver.
<code>setArnoldiLagPreconditioner(lag)</code>	Set when the preconditioner is rebuilt in the nonlinear solve.
<code>setNLEIGSEPS(eps)</code>	Set a linear eigensolver object associated to the nonlinear eigensolver.
<code>setNLEIGSFULLBasis([fullbasis])</code>	Set TOAR-basis (default) or full-basis variants of the NLEIGS method.
<code>setNLEIGSInterpolation([tol, deg])</code>	Set the tolerance and maximum degree for the interpolation polynomial.
<code>setNLEIGSLocking(lock)</code>	Toggle between locking and non-locking variants of the NLEIGS method.
<code>setNLEIGSRKShifts(shifts)</code>	Set a list of shifts to be used in the Rational Krylov method.
<code>setNLEIGSRestart(keep)</code>	Set the restart parameter for the NLEIGS method.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all NEP options in the database.
<code>setProblemType(problem_type)</code>	Set the type of the eigenvalue problem.
<code>setRG(rg)</code>	Set a region object associated to the eigensolver.
<code>setRIIConstCorrectionTol(cct)</code>	Set a flag to keep the tolerance used in the linear solver constant.
<code>setRIIDeflationThreshold(deftol)</code>	Set the threshold used to switch between deflated and non-deflated.
<code>setRIIHermitian(herm)</code>	Set a flag to use the Hermitian version of the solver.
<code>setRIIKSP(ksp)</code>	Set a linear solver object associated to the nonlinear eigensolver.
<code>setRIILagPreconditioner(lag)</code>	Set when the preconditioner is rebuilt in the nonlinear solve.
<code>setRIIMaximumIterations(its)</code>	Set the max.
<code>setRefine(ref[, npart, tol, its, scheme])</code>	Set the refinement strategy used by the NEP object.
<code>setSLPDeflationThreshold(deftol)</code>	Set the threshold used to switch between deflated and non-deflated.
<code>setSLPEPS(eps)</code>	Set a linear eigensolver object associated to the nonlinear eigensolver.

continues on next page

Table 66 – continued from previous page

<code>setSLPEPSLeft(eps)</code>	Set a linear eigensolver object associated to the non-linear eigensolver.
<code>setSLPKSP(ksp)</code>	Set a linear solver object associated to the nonlinear eigensolver.
<code>setSplitOperator(A, f[, structure])</code>	Set the operator of the nonlinear eigenvalue problem in split form.
<code>setSplitPreconditioner(P[, structure])</code>	Set the operator in split form.
<code>setStoppingTest(stopping[, args, kargs])</code>	Set a function to decide when to stop the outer iteration of the eigensolver.
<code>setTarget(target)</code>	Set the value of the target.
<code>setTolerances([tol, maxit])</code>	Set the tolerance and max.
<code>setTrackAll(trackall)</code>	Set if the solver must compute the residual of all approximate eigenpairs.
<code>setTwoSided(twosided)</code>	Set the solver to use a two-sided variant.
<code>setType(nep_type)</code>	Set the particular solver to be used in the NEP object.
<code>setUp()</code>	Set up all the internal data structures.
<code>setWhichEigenpairs(which)</code>	Set which portion of the spectrum is to be sought.
<code>solve()</code>	Solve the nonlinear eigenproblem.
<code>valuesView([viewer])</code>	Display the computed eigenvalues in a viewer.
<code>vectorsView([viewer])</code>	Output computed eigenvectors to a viewer.
<code>view([viewer])</code>	Print the NEP data structure.

Attributes Summary

<code>bv</code>	The basis vectors (<i>BV</i>) object associated.
<code>ds</code>	The direct solver (<i>DS</i>) object associated.
<code>max_it</code>	The maximum iteration count used by the NEP convergence tests.
<code>problem_type</code>	The problem type from the NEP object.
<code>rg</code>	The region (<i>RG</i>) object associated.
<code>target</code>	The value of the target.
<code>tol</code>	The tolerance used by the NEP convergence tests.
<code>track_all</code>	Compute the residual of all approximate eigenpairs.
<code>which</code>	The portion of the spectrum to be sought.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching for all NEP options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all NEP option requests.

Return type

None

See also

[setOptionsPrefix](#), [getOptionsPrefix](#), [NEPAppendOptionsPrefix](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:385 <slepc4py/SLEPc/NEP.pyx#L385>](#)

applyResolvent(*omega*, *v*, *r*, *rg*=None)

Apply the resolvent $T^{-1}(z)$ to a given vector.

Collective.

Parameters

- **omega** ([Scalar](#)) – Value where the resolvent must be evaluated.
- **v** ([Vec](#)) – Input vector.
- **r** ([Vec](#)) – Placeholder for the result vector.
- **rg** ([RG](#) | [None](#)) – Region.

Return type

[None](#)

Notes

The resolvent $T^{-1}(z) = \sum_i (z - \lambda_i)^{-1} x_i y_i^*$ is evaluated at $z = \omega$ and the matrix-vector product $r = T^{-1}(\omega)v$ is computed. Vectors x_i, y_i are right and left eigenvectors, respectively, normalized so that $y_i^* T'(\lambda_i) x_i = 1$. The sum contains only eigenvectors that have been previously computed with [solve\(\)](#), and if a region *rg* is given then only those corresponding to eigenvalues inside the region are considered.

See also

[solve](#), [getLeftEigenvector](#), [NEPApplyResolvent](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1834 <slepc4py/SLEPc/NEP.pyx#L1834>](#)

cancelMonitor()

Clear all monitors for a [NEP](#) object.

Logically collective.

See also

[NEPMonitorCancel](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1159 <slepc4py/SLEPc/NEP.pyx#L1159>](#)

Return type

[None](#)

computeError(*i*, *etype*=None)

Compute the error associated with the *i*-th computed eigenpair.

Collective.

Compute the error (based on the residual norm) associated with the *i*-th computed eigenpair.

Parameters

- **i** (*int*) – Index of the solution to be considered.
- **etype** (*ErrorType* / *None*) – The error type to compute.

Returns

The error bound, computed in various ways from the residual norm $\|T(\lambda)x\|_2$ where λ is the eigenvalue and x is the eigenvector.

Return type

float

Notes

The index *i* should be a value between 0 and *nconv*-1 (see *getConverged()*).

If the computation of left eigenvectors was enabled with *setTwoSided()*, then the error will be computed using the maximum of the value above and the left residual norm $\|y^*T(\lambda)\|_2$, where y is the approximate left eigenvector.

See also

getErrorEstimate, *setTwoSided*, *NEPComputeError*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1385 <slepc4py/SLEPc/NEP.pyx#L1385>`

create(*comm=None*)

Create the NEP object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator. If not provided, it defaults to all processes.

Return type

Self

See also

NEPCreate

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:268 <slepc4py/SLEPc/NEP.pyx#L268>`

destroy()

Destroy the NEP object.

Collective.

See also

NEPDestroy

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:242 <slepc4py/SLEPc/NEP.pyx#L242>`

Return type

Self

errorView(*etype=None, viewer=None*)

Display the errors associated with the computed solution.

Collective.

Display the errors and the eigenvalues.

Parameters

- **etype** (*ErrorType* / *None*) – The error type to compute.
- **viewer** (*petsc4py.PETSc.Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

Notes

By default, this function checks the error of all eigenpairs and prints the eigenvalues if all of them are below the requested tolerance. If the viewer has format ASCII_INFO_DETAIL then a table with eigenvalues and corresponding errors is printed.

See also

solve, valuesView, vectorsView, NEPErrorView

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1428 <slepc4py/SLEPc/NEP.pyx#L1428>``

getBV()

Get the basis vectors object associated to the eigensolver.

Not collective.

Returns

The basis vectors context.

Return type

BV

See also

setBV, NEPGetBV

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:877 <slepc4py/SLEPc/NEP.pyx#L877>``

getCISSExtraction()

Get the extraction technique used in the CISS solver.

Not collective.

Returns

The extraction technique.

Return type

CISSExtraction

See also

[`setCISSExtraction`](#), [`NEPCISSGetExtraction`](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2792](#) <[slepc4py/SLEPc/NEP.pyx#L2792](#)>

getCISSKSPs()

Get the list of linear solver objects associated with the CISS solver.

Collective.

Returns

The linear solver objects.

Return type

list of `petsc4py.PETSc.KSP`

Notes

The number of `petsc4py.PETSc.KSP` solvers is equal to the number of integration points divided by the number of partitions. This value is halved in the case of real matrices with a region centered at the real axis.

See also

[`setCISSSizes`](#), [`NEPCISSGetKSPs`](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2989](#) <[slepc4py/SLEPc/NEP.pyx#L2989](#)>

getCISSRefinement()

Get the values of various refinement parameters in the CISS solver.

Not collective.

Returns

- **inner** (`int`) – Number of iterative refinement iterations (inner loop).
- **blsize** (`int`) – Number of iterative refinement iterations (blocksize loop).

Return type

tuple[int, int]

See also

[`setCISSRefinement`](#), [`NEPCISSGetRefinement`](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2967](#) <[slepc4py/SLEPc/NEP.pyx#L2967](#)>

getCISSSizes()

Get the values of various size parameters in the CISS solver.

Not collective.

Returns

- **ip** (`int`) – Number of integration points.

- **bs** (`int`) – Block size.
- **ms** (`int`) – Moment size.
- **npart** (`int`) – Number of partitions when splitting the communicator.
- **bsmax** (`int`) – Maximum block size.
- **realmats** (`bool`) – True if A and B are real.

Return type

`tuple[int, int, int, int, int, bool]`

See also

`setCISSSizes`, `NEPCISSGetSizes`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2865 <slepc4py/SLEPc/NEP.pyx#L2865>`

getCISSThreshold()

Get the values of various threshold parameters in the CISS solver.

Not collective.

Returns

- **delta** (`float`) – Threshold for numerical rank.
- **spur** (`float`) – Spurious threshold (to discard spurious eigenpairs).

Return type

`tuple[float, float]`

See also

`setCISSThreshold`, `NEPCISSGetThreshold`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2922 <slepc4py/SLEPc/NEP.pyx#L2922>`

getConverged()

Get the number of converged eigenpairs.

Not collective.

Returns

nconv – Number of converged eigenpairs.

Return type

`int`

Notes

This function should be called after `solve()` has finished.

The value `nconv` may be different from the number of requested solutions `nev`, but not larger than `ncv`, see `setDimensions()`.

See also

setDimensions, *solve*, *getEigenpair*, *NEPGetConverged*

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1256 <slepc4py/SLEPc/NEP.pyx#L1256>``

getConvergedReason()

Get the reason why the *solve()* iteration was stopped.

Not collective.

Returns

Negative value indicates diverged, positive value converged.

Return type

ConvergedReason

See also

setTolerances, *solve*, *NEPGetConvergedReason*

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1237 <slepc4py/SLEPc/NEP.pyx#L1237>``

getConvergenceTest()

Get the method used to compute the error estimate used in the convergence test.

Not collective.

Returns

The method used to compute the error estimate used in the convergence test.

Return type

Conv

See also

setConvergenceTest, *NEPGetConvergenceTest*

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:624 <slepc4py/SLEPc/NEP.pyx#L624>``

getDS()

Get the direct solver associated to the eigensolver.

Not collective.

Returns

The direct solver context.

Return type

DS

See also

setDS, *NEPGetDS*

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:951 <slepc4py/SLEPc/NEP.pyx#L951>``

getDimensions()

Get the number of eigenvalues to compute.

Not collective.

Get the number of eigenvalues to compute, and the dimension of the subspace.

Returns

- **nev** (*int*) – Number of eigenvalues to compute.
- **ncv** (*int*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int*) – Maximum dimension allowed for the projected problem.

Return type

`tuple[int, int, int]`

See also

`setDimensions`, `NEPGetDimensions`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:795 <slepc4py/SLEPc/NEP.pyx#L795>`

getEigenpair(*i*, *Vr*=None, *Vi*=None)

Get the *i*-th solution of the eigenproblem as computed by `solve()`.

Collective.

The solution consists of both the eigenvalue and the eigenvector.

Parameters

- **i** (*int*) – Index of the solution to be obtained.
- **Vr** (*Vec* | *None*) – Placeholder for the returned eigenvector (real part).
- **Vi** (*Vec* | *None*) – Placeholder for the returned eigenvector (imaginary part).

Returns

The computed eigenvalue.

Return type

`complex`

Notes

The index *i* should be a value between 0 and `nconv-1` (see `getConverged()`). Eigenpairs are indexed according to the ordering criterion established with `setWhichEigenpairs()`.

The eigenvector is normalized to have unit norm.

See also

`solve`, `getConverged`, `setWhichEigenpairs`, `NEPGetEigenpair`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1282 <slepc4py/SLEPc/NEP.pyx#L1282>`

getEigenvalueComparison()

Get the eigenvalue comparison function.

Not collective.

Returns

The eigenvalue comparison function.

Return type

NEPEigenvalueComparison

See also

setEigenvalueComparison

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1100 <slepc4py/SLEPc/NEP.pyx#L1100>`

getErrorEstimate(i)

Get the error estimate associated to the i-th computed eigenpair.

Not collective.

Parameters

i (*int*) – Index of the solution to be considered.

Returns

Error estimate.

Return type

float

Notes

This is the error estimate used internally by the eigensolver. The actual error bound can be computed with *computeError()*.

See also

computeError, *NEPGetErrorEstimate*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1356 <slepc4py/SLEPc/NEP.pyx#L1356>`

getFunction()

Get the function to compute the nonlinear Function $T(\lambda)$.

Collective.

Get the function to compute the nonlinear Function $T(\lambda)$ and the matrix.

Returns

- **F** (*petsc4py.PETSc.Mat*) – Function matrix.
- **P** (*petsc4py.PETSc.Mat*) – Preconditioner matrix (usually the same as the F).
- **function** (*NEPFunction*) – Function evaluation routine.

Return type

tuple[*petsc4py.PETSc.Mat*, *petsc4py.PETSc.Mat*, *NEPFunction*]

See also

[*setFunction*](#), [*NEPGetFunction*](#)

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1540 <slepc4py/SLEPc/NEP.pyx#L1540>``

getInterpolInterpolation()

Get the tolerance and maximum degree for the interpolation polynomial.

Not collective.

Returns

- **tol** ([*float*](#)) – The tolerance to stop computing polynomial coefficients.
- **deg** ([*int*](#)) – The maximum degree of interpolation.

Return type

[*tuple*](#)[[*float*](#), [*int*](#)]

See also

[*setInterpolInterpolation*](#), [*NEPInterpolGetInterpolation*](#)

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:2444 <slepc4py/SLEPc/NEP.pyx#L2444>``

getInterpolPEP()

Get the associated polynomial eigensolver object.

Collective.

Returns

The polynomial eigensolver.

Return type

[*PEP*](#)

See also

[*setInterpolPEP*](#), [*NEPInterpolGetPEP*](#)

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:2401 <slepc4py/SLEPc/NEP.pyx#L2401>``

getIterationNumber()

Get the current iteration number.

Not collective.

If the call to [*solve\(\)*](#) is complete, then it returns the number of iterations carried out by the solution method.

Returns

Iteration number.

Return type

[*int*](#)

See also

[`getConvergedReason`](#), [`setTolerances`](#), [`NEPGetIterationNumber`](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1215 <slepc4py/SLEPc/NEP.pyx#L1215>`

`getJacobian()`

Get the function to compute the Jacobian $T'(\lambda)$ and J.

Collective.

Get the function to compute the Jacobian $T'(\lambda)$ and the matrix.

Returns

- **J** (`petsc4py.PETSc.Mat`) – Jacobian matrix.
- **jacobian** (`NEPJacobian`) – Jacobian evaluation routine.

Return type

`tuple[petsc4py.PETSc.Mat, NEPJacobian]`

See also

[`setJacobian`](#), [`NEPGetJacobian`](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1606 <slepc4py/SLEPc/NEP.pyx#L1606>`

`getLeftEigenvector(i, Wr, Wi=None)`

Get the i-th left eigenvector as computed by [`solve\(\)`](#).

Collective.

Parameters

- **i** (`int`) – Index of the solution to be obtained.
- **Wr** (`Vec`) – Placeholder for the returned eigenvector (real part).
- **Wi** (`Vec` / `None`) – Placeholder for the returned eigenvector (imaginary part).

Return type

`None`

Notes

The index `i` should be a value between 0 and `nconv-1` (see [`getConverged\(\)`](#)). Eigensolutions are indexed according to the ordering criterion established with [`setWhichEigenpairs\(\)`](#).

Left eigenvectors are available only if the `twosided` flag was set with [`setTwoSided\(\)`](#).

See also

[`getEigenpair`](#), [`getConverged`](#), [`setTwoSided`](#), [`NEPGetLeftEigenvector`](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1323 <slepc4py/SLEPc/NEP.pyx#L1323>`

getMonitor()

Get the list of monitor functions.

Not collective.

Returns

The list of monitor functions.

Return type

NEPMonitorFunction

See also

setMonitor

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1142 <slepc4py/SLEPc/NEP.pyx#L1142>](#)

getNArnoldiKSP()

Get the linear solver object associated with the nonlinear eigensolver.

Collective.

Returns

The linear solver object.

Return type

petsc4py.PETSc.KSP

See also

setNArnoldiKSP, *NEPNArnoldiGetKSP*

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2318 <slepc4py/SLEPc/NEP.pyx#L2318>](#)

getNArnoldiLagPreconditioner()

Get how often the preconditioner is rebuilt.

Not collective.

Returns

The lag parameter.

Return type

int

See also

setNArnoldiLagPreconditioner, *NEPNArnoldiGetLagPreconditioner*

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2363 <slepc4py/SLEPc/NEP.pyx#L2363>](#)

getNLEIGSEPS()

Get the linear eigensolver object associated with the nonlinear eigensolver.

Collective.

Returns

The linear eigensolver.

Return type*EPS***See also***setNLEIGSEPS, NEPNLEIGSGetEPS*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2675 <slepc4py/SLEPc/NEP.pyx#L2675>`

getNLEIGSFULLBASIS()

Get the flag that indicates if NLEIGS is using the full-basis variant.

Not collective.

Returns

True if the full-basis variant is selected.

Return type*bool***See also***setNLEIGSFULLBASIS, NEPNLEIGSGetFullBasis*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2639 <slepc4py/SLEPc/NEP.pyx#L2639>`

getNLEIGSINTERPOLATION()

Get the tolerance and maximum degree for the interpolation polynomial.

Not collective.

Get the tolerance and maximum degree when building the interpolation via divided differences.

Returns

- **tol** (*float*) – The tolerance to stop computing divided differences.
- **deg** (*int*) – The maximum degree of interpolation.

Return type*tuple[*float*, *int*]***See also***setNLEIGSINTERPOLATION, NEPNLEIGSGetInterpolation*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2581 <slepc4py/SLEPc/NEP.pyx#L2581>`

getNLEIGSKSPs()

Get the list of linear solver objects associated with the NLEIGS solver.

Collective.

Returns

The linear solver objects.

Return type*list of *petsc4py.PETSc.KSP**

Notes

The number of `petsc4py.PETSc.KSP` solvers is equal to the number of shifts provided by the user, or 1 if the user did not provide shifts.

See also

setNLEIGSRKShifts, *NEPNLEIGSGetKSPs*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2747 <slepc4py/SLEPc/NEP.pyx#L2747>`

`getNLEIGSLocking()`

Get the locking flag used in the NLEIGS method.

Not collective.

Returns

The locking flag.

Return type

bool

See also

setNLEIGSLocking, *NEPNLEIGSGetLocking*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2536 <slepc4py/SLEPc/NEP.pyx#L2536>`

`getNLEIGSRKShifts()`

Get the list of shifts used in the Rational Krylov method.

Not collective.

Returns

The shift values.

Return type

ArrayScalar

See also

setNLEIGSRKShifts, *NEPNLEIGSGetRKShifts*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2722 <slepc4py/SLEPc/NEP.pyx#L2722>`

`getNLEIGSRestart()`

Get the restart parameter used in the NLEIGS method.

Not collective.

Returns

The number of vectors to be kept at restart.

Return type

float

See also

setNLEIGSRestart, *NEPNLEIGSGetRestart*

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2492](#) <slepc4py/SLEPc/NEP.pyx#L2492>

getOptionsPrefix()

Get the prefix used for searching for all NEP options in the database.

Not collective.

Returns

The prefix string set for this NEP object.

Return type

str

See also

setOptionsPrefix, *appendOptionsPrefix*, *NEPGetOptionsPrefix*

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:335](#) <slepc4py/SLEPc/NEP.pyx#L335>

getProblemType()

Get the problem type from the *NEP* object.

Not collective.

Returns

The problem type that was previously set.

Return type

ProblemType

See also

setProblemType, *NEPGetProblemType*

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:423](#) <slepc4py/SLEPc/NEP.pyx#L423>

getRG()

Get the region object associated to the eigensolver.

Not collective.

Returns

The region context.

Return type

RG

See also

setRG, *NEPGetRG*

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:914](#) <slepc4py/SLEPc/NEP.pyx#L914>

getRIIConstCorrectionTol()

Get the constant tolerance flag.

Not collective.

Returns

If True, the `petsc4py.PETSc.KSP` relative tolerance is constant.

Return type

`bool`

See also

`setRIIConstCorrectionTol`, `NEPRIIGetConstCorrectionTol`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1942 <slepc4py/SLEPc/NEP.pyx#L1942>`

getRIIDeflationThreshold()

Get the threshold value that controls deflation.

Not collective.

Returns

The threshold value.

Return type

`float`

See also

`setRIIDeflationThreshold`, `NEPRIIGetDeflationThreshold`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2081 <slepc4py/SLEPc/NEP.pyx#L2081>`

getRIIHermitian()

Get if the Hermitian version must be used by the solver.

Not collective.

Returns

If True, the Hermitian version is used.

Return type

`bool`

See also

`setRIIHermitian`, `NEPRIIGetHermitian`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2031 <slepc4py/SLEPc/NEP.pyx#L2031>`

getRIIKSP()

Get the linear solver object associated with the nonlinear eigensolver.

Collective.

Returns

The linear solver object.

Return type

`petsc4py.PETSc.KSP`

See also

`setRIIKSP`, `NEPRIIGetKSP`

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2117](#) <slepc4py/SLEPc/NEP.pyx#L2117>

getRIILagPreconditioner()

Get how often the preconditioner is rebuilt.

Not collective.

Returns

The lag parameter.

Return type

`int`

See also

`setRIILagPreconditioner`, `NEPRIIGetLagPreconditioner`

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1898](#) <slepc4py/SLEPc/NEP.pyx#L1898>

getRIIMaximumIterations()

Get the maximum number of inner iterations of RII.

Not collective.

Returns

Maximum inner iterations.

Return type

`int`

See also

`setRIIMaximumIterations`, `NEPRIIGetMaximumIterations`

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1983](#) <slepc4py/SLEPc/NEP.pyx#L1983>

getRefine()

Get the refinement strategy used by the NEP object.

Not collective.

Returns

- **ref** (*Refine*) – The refinement type.
- **npart** (`int`) – The number of partitions of the communicator.
- **tol** (`float`) – The convergence tolerance.

- **its** (`int`) – The maximum number of refinement iterations.
- **scheme** (`RefineScheme`) – Scheme for solving linear systems.

Return type

`tuple[Refine, int, float, int, RefineScheme]`

See also

`setRefine`, `NEPGetRefine`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:663 <slepc4py/SLEPc/NEP.pyx#L663>`

getRefineKSP()

Get the KSP object used by the eigensolver in the refinement phase.

Collective.

Returns

The linear solver object.

Return type

`petsc4py.PETSc.KSP`

See also

`setRefine`, `NEPRefineGetKSP`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:738 <slepc4py/SLEPc/NEP.pyx#L738>`

getSLPDeflationThreshold()

Get the threshold value that controls deflation.

Not collective.

Returns

The threshold value.

Return type

`float`

See also

`setSLPDeflationThreshold`, `NEPSLPGetDeflationThreshold`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2167 <slepc4py/SLEPc/NEP.pyx#L2167>`

getSLPEPS()

Get the linear eigensolver object associated with the nonlinear eigensolver.

Collective.

Returns

The linear eigensolver.

Return type

`EPS`

See also

`setSLPEPS`, `NEPSLPGetEPS`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2203` `<slepc4py/SLEPc/NEP.pyx#L2203>`

getSLPEPSLeft()

Get the left eigensolver.

Collective.

Returns

The linear eigensolver.

Return type

`EPS`

See also

`setSLPEPSLeft`, `NEPSLPGetEPSLeft`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2242` `<slepc4py/SLEPc/NEP.pyx#L2242>`

getSLPKSP()

Get the linear solver object associated with the nonlinear eigensolver.

Collective.

Returns

The linear solver object.

Return type

`petsc4py.PETSc.KSP`

See also

`setSLPKSP`, `NEPSLPGetKSP`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2279` `<slepc4py/SLEPc/NEP.pyx#L2279>`

getSplitOperator()

Get the operator of the nonlinear eigenvalue problem in split form.

Collective.

Returns

- **A** (list of `petsc4py.PETSc.Mat`) – Coefficient matrices of the split form.
- **f** (list of `FN`) – Scalar functions of the split form.
- **structure** (`petsc4py.PETSc.Mat.Structure`) – Structure flag for matrices.

Return type

`tuple[list[petsc4py.PETSc.Mat], list[FN], petsc4py.PETSc.Mat.Structure]`

See also

setSplitOperator, *NEPGetSplitOperatorInfo*, *NEPGetSplitOperatorTerm*

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1683 <slepc4py/SLEPc/NEP.pyx#L1683>``

getSplitPreconditioner()

Get the operator of the split preconditioner.

Not collective.

Returns

- **P** (list of `petsc4py.PETSc.Mat`) – Coefficient matrices of the split preconditioner.
- **structure** (`petsc4py.PETSc.Mat.Structure`) – Structure flag for matrices.

Return type

`tuple[list[petsc4py.PETSc.Mat], petsc4py.PETSc.Mat.Structure]`

See also

setSplitPreconditioner, *NEPGetSplitPreconditionerTerm*

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1752 <slepc4py/SLEPc/NEP.pyx#L1752>``

getStoppingTest()

Get the stopping test function.

Not collective.

Returns

The stopping test function.

Return type

NEPStoppingFunction

See also

setStoppingTest

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1053 <slepc4py/SLEPc/NEP.pyx#L1053>``

getTarget()

Get the value of the target.

Not collective.

Returns

The value of the target.

Return type

Scalar

Notes

If the target was not set by the user, then zero is returned.

See also

`setTarget`, `NEPGetTarget`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:521 <slepc4py/SLEPc/NEP.pyx#L521>`

`getTolerances()`

Get the tolerance and maximum iteration count.

Not collective.

Get the tolerance and maximum iteration count used by the default NEP convergence tests.

Returns

- `tol` (`float`) – The convergence tolerance.
- `maxit` (`int`) – The maximum number of iterations.

Return type

`tuple[float, int]`

See also

`setTolerances`, `NEPGetTolerances`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:571 <slepc4py/SLEPc/NEP.pyx#L571>`

`getTrackAll()`

Get the flag indicating whether all residual norms must be computed.

Not collective.

Returns

Whether the solver computes all residuals or not.

Return type

`bool`

See also

`setTrackAll`, `NEPGetTrackAll`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:758 <slepc4py/SLEPc/NEP.pyx#L758>`

`getTwoSided()`

Get the flag indicating if a two-sided variant is being used.

Not collective.

Get the flag indicating whether a two-sided variant of the algorithm is being used or not.

Returns

Whether the two-sided variant is to be used or not.

Return type
`bool`

See also

`setTwoSided`, `NEPGetTwoSided`

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1781 <slepc4py/SLEPc/NEP.pyx#L1781>``

getType()

Get the NEP type of this object.

Not collective.

Returns

The solver currently being used.

Return type

`str`

See also

`setType`, `NEPGetType`

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:316 <slepc4py/SLEPc/NEP.pyx#L316>``

getWhichEigenpairs()

Get which portion of the spectrum is to be sought.

Not collective.

Returns

The portion of the spectrum to be sought by the solver.

Return type

`Which`

See also

`setWhichEigenpairs`, `NEPGetWhichEigenpairs`

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:467 <slepc4py/SLEPc/NEP.pyx#L467>``

reset()

Reset the NEP object.

Collective.

See also

`NEPReset`

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:256 <slepc4py/SLEPc/NEP.pyx#L256>``

Return type

None

setBV(*bv*)

Set the basis vectors object associated to the eigensolver.

Collective.

Parameters

bv (*BV*) – The basis vectors context.

Return type

None

See also

[getBV](#), [NEPSetBV](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:897 <slepc4py/SLEPc/NEP.pyx#L897>`

setCISSExtraction(*extraction*)

Set the extraction technique used in the CISS solver.

Logically collective.

Parameters

extraction (*CISSExtraction*) – The extraction technique.

Return type

None

See also

[getCISSExtraction](#), [NEPCISSSetExtraction](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2774 <slepc4py/SLEPc/NEP.pyx#L2774>`

setCISSRefinement(*inner=None, blsize=None*)

Set the values of various refinement parameters in the CISS solver.

Logically collective.

Parameters

- **inner** (*int* / *None*) – Number of iterative refinement iterations (inner loop).
- **blsize** (*int* / *None*) – Number of iterative refinement iterations (blocksize loop).

Return type

None

See also

[getCISSRefinement](#), [NEPCISSSetRefinement](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2944 <slepc4py/SLEPc/NEP.pyx#L2944>`

setCISSSizes(*ip=None, bs=None, ms=None, npart=None, bsmax=None, realmats=False*)

Set the values of various size parameters in the CISS solver.

Logically collective.

Parameters

- **ip** (*int* / *None*) – Number of integration points.
- **bs** (*int* / *None*) – Block size.
- **ms** (*int* / *None*) – Moment size.
- **npart** (*int* / *None*) – Number of partitions when splitting the communicator.
- **bsmax** (*int* / *None*) – Maximum block size.
- **realmats** (*bool*) – True if A and B are real.

Return type

None

Notes

The default number of partitions is 1. This means the internal `petsc4py.PETSc.KSP` object is shared among all processes of the *NEP* communicator. Otherwise, the communicator is split into `npart` communicators, so that `npart` `petsc4py.PETSc.KSP` solves proceed simultaneously.

See also

[`getCISSSizes`](#), [`setCISSThreshold`](#), [`setCISSRefinement`](#), [`NEPCISSSetSizes`](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2811](#) <[slepc4py/SLEPc/NEP.pyx#L2811](#)>

setCISSThreshold(*delta=None, spur=None*)

Set the values of various threshold parameters in the CISS solver.

Logically collective.

Parameters

- **delta** (*float* / *None*) – Threshold for numerical rank.
- **spur** (*float* / *None*) – Spurious threshold (to discard spurious eigenpairs).

Return type

None

See also

[`getCISSThreshold`](#), [`NEPCISSSetThreshold`](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2899](#) <[slepc4py/SLEPc/NEP.pyx#L2899](#)>

setConvergenceTest(*conv*)

Set how to compute the error estimate used in the convergence test.

Logically collective.

Parameters

conv (*Conv*) – The method used to compute the error estimate used in the convergence test.

Return type

None

See also[`getConvergenceTest`](#), [`NEPSetConvergenceTest`](#)**:sources:**``Source code at slepc4py/SLEPc/NEP.pyx:644 <slepc4py/SLEPc/NEP.pyx#L644>``**setDS(ds)**

Set a direct solver object associated to the eigensolver.

Collective.

Parameters**ds** (DS) – The direct solver context.**Return type**

None

See also[`getDS`](#), [`NEPSetDS`](#)**:sources:**``Source code at slepc4py/SLEPc/NEP.pyx:971 <slepc4py/SLEPc/NEP.pyx#L971>``**setDimensions(nev=None, ncv=None, mpd=None)**

Set the number of eigenvalues to compute.

Logically collective.

Set the number of eigenvalues to compute and the dimension of the subspace.

Parameters

- **nev** (*int* / *None*) – Number of eigenvalues to compute.
- **ncv** (*int* / *None*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int* / *None*) – Maximum dimension allowed for the projected problem.

Return type

None

Notes

Use [`DETERMINE`](#) for `ncv` and `mpd` to assign a reasonably good value, which is dependent on the solution method.

The parameters `ncv` and `mpd` are intimately related, so that the user is advised to set one of them at most. Normal usage is the following:

- In cases where `nev` is small, the user sets `ncv` (a reasonable default is `2 * nev`).
- In cases where `nev` is large, the user sets `mpd`.

The value of `ncv` should always be between `nev` and `(nev + mpd)`, typically `ncv = nev + mpd`. If `nev` is not too large, `mpd = nev` is a reasonable choice, otherwise a smaller value should be used.

See also

[`getDimensions`](#), [`NEPSetDimensions`](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:823 <slepc4py/SLEPc/NEP.pyx#L823>`

setEigenvalueComparison(*comparison*, *args=None*, *kargs=None*)

Set an eigenvalue comparison function.

Logically collective.

Notes

This eigenvalue comparison function is used when `setWhichEigenpairs()` is set to `NEP.Which.USER`.

See also

[`getEigenvalueComparison`](#), [`NEPSetEigenvalueComparison`](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1070 <slepc4py/SLEPc/NEP.pyx#L1070>`

Parameters

- **comparison** ([`NEPEigenvalueComparison`](#) / `None`)
- **args** (`tuple`[`Any`, ...] / `None`)
- **kargs** (`dict`[`str`, `Any`] / `None`)

Return type

`None`

setFromOptions()

Set NEP options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before `setUp()` if the user is to be allowed to set the solver type.

See also

[`setOptionsPrefix`](#), [`NEPSetFromOptions`](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:404 <slepc4py/SLEPc/NEP.pyx#L404>`

Return type

`None`

setFunction(*function*, *F=None*, *P=None*, *args=None*, *kargs=None*)

Set the function to compute the nonlinear Function $T(\lambda)$.

Collective.

Set the function to compute the nonlinear Function $T(\lambda)$ as well as the location to store the matrix.

Parameters

- **function** ([NEPFunction](#)) – Function evaluation routine.
- **F** ([petsc4py.PETSc.Mat](#) | *None*) – Function matrix.
- **P** ([petsc4py.PETSc.Mat](#) | *None*) – Preconditioner matrix (usually the same as F).
- **args** ([tuple](#)[*Any*, ...] | *None*)
- **kargs** ([dict](#)[*str*, *Any*] | *None*)

Return type

None

See also

[setJacobian](#), [getFunction](#), [NEPSetFunction](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1500 <slepc4py/SLEPc/NEP.pyx#L1500>`

setInitialSpace(*space*)

Set the initial space from which the eigensolver starts to iterate.

Collective.

Parameters

space ([Vec](#)) – The initial space.

Return type

None

Notes

Some solvers start to iterate on a single vector (initial vector). In that case, only the first vector is taken into account and the other vectors are ignored.

These vectors do not persist from one [solve\(\)](#) call to the other, so the initial space should be set every time.

The vectors do not need to be mutually orthonormal, since they are explicitly orthonormalized internally.

Common usage of this function is when the user can provide a rough approximation of the wanted eigenspace. Then, convergence may be faster.

See also

[setUp](#), [NEPSetInitialSpace](#)

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:990 <slepc4py/SLEPc/NEP.pyx#L990>`

setInterpolInterpolation(*tol=None, deg=None*)

Set the tolerance and maximum degree for the interpolation polynomial.

Collective.

Parameters

- **tol** ([float](#) | *None*) – The tolerance to stop computing polynomial coefficients.
- **deg** ([int](#) | *None*) – The maximum degree of interpolation.

Return type

None

See also[*getInterpolInterpolation*](#), [*NEPInterpolSetInterpolation*](#)**:sources:** ``Source code at slepc4py/SLEPc/NEP.pyx:2421 <slepc4py/SLEPc/NEP.pyx#L2421>``**setInterpolPEP**(*pep*)

Set a polynomial eigensolver object associated to the nonlinear eigensolver.

Collective.

Parameters**pep** ([*PEP*](#)) – The polynomial eigensolver.**Return type**

None

See also[*getInterpolPEP*](#), [*NEPInterpolSetPEP*](#)**:sources:** ``Source code at slepc4py/SLEPc/NEP.pyx:2384 <slepc4py/SLEPc/NEP.pyx#L2384>``**setJacobian**(*jacobian*, *J=None*, *args=None*, *kargs=None*)Set the function to compute the Jacobian $T'(\lambda)$.

Collective.

Set the function to compute the Jacobian $T'(\lambda)$ as well as the location to store the matrix.**Parameters**

- **jacobian** ([*NEPJacobian*](#)) – Jacobian evaluation routine.
- **J** ([*petsc4py.PETSc.Mat*](#) | *None*) – Jacobian matrix.
- **args** ([*tuple*](#)[*Any*, ...] | *None*)
- **kargs** ([*dict*](#)[*str*, *Any*] | *None*)

Return type

None

See also[*setFunction*](#), [*getJacobian*](#), [*NEPSetJacobian*](#)**:sources:** ``Source code at slepc4py/SLEPc/NEP.pyx:1570 <slepc4py/SLEPc/NEP.pyx#L1570>``**setMonitor**(*monitor*, *args=None*, *kargs=None*)

Append a monitor function to the list of monitors.

Logically collective.

See also

[*getMonitor*](#), [*cancelMonitor*](#), [*NEPMonitorSet*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1117](#) <[slepc4py/SLEPc/NEP.pyx#L1117](#)>

Parameters

- **monitor** ([*NEPMonitorFunction*](#) | *None*)
- **args** ([*tuple*](#)[*Any*, ...] | *None*)
- **kargs** ([*dict*](#)[*str*, *Any*] | *None*)

Return type

None

setNArnoldiKSP(*ksp*)

Set a linear solver object associated to the nonlinear eigensolver.

Collective.

Parameters

ksp ([*KSP*](#)) – The linear solver object.

Return type

None

See also

[*getNArnoldiKSP*](#), [*NEPNArnoldiSetKSP*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2301](#) <[slepc4py/SLEPc/NEP.pyx#L2301](#)>

setNArnoldiLagPreconditioner(*lag*)

Set when the preconditioner is rebuilt in the nonlinear solve.

Logically collective.

Parameters

lag ([*int*](#)) – 0 indicates NEVER rebuild, 1 means rebuild every time the Jacobian is computed within the nonlinear iteration, 2 means every second time the Jacobian is built, etc.

Return type

None

Notes

The default is 1. The preconditioner is ALWAYS built in the first iteration of a nonlinear solve.

See also

[*getNArnoldiLagPreconditioner*](#), [*NEPNArnoldiSetLagPreconditioner*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2338](#) <[slepc4py/SLEPc/NEP.pyx#L2338](#)>

setNLEIGSEPS(*eps*)

Set a linear eigensolver object associated to the nonlinear eigensolver.

Collective.

Parameters

eps ([EPS](#)) – The linear eigensolver.

Return type

[None](#)

See also

[getNLEIGSEPS](#), [NEPNLEIGSSetEPS](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2658](#) <[slepc4py/SLEPc/NEP.pyx#L2658](#)>

setNLEIGSFullBasis(*fullbasis=True*)

Set TOAR-basis (default) or full-basis variants of the NLEIGS method.

Logically collective.

Toggle between TOAR-basis (default) and full-basis variants of the NLEIGS method.

Parameters

fullbasis ([bool](#)) – True if the full-basis variant must be selected.

Return type

[None](#)

Notes

The default is to use a compact representation of the Krylov basis, that is, $V = (I \otimes U)S$, with a [BV](#) of type [TENSOR](#). This behavior can be changed so that the full basis V is explicitly stored and operated with. This variant is more expensive in terms of memory and computation, but is necessary in some cases, particularly for two-sided computations, see [setTwoSided\(\)](#).

In the full-basis variant, the NLEIGS solver uses an [EPS](#) object to explicitly solve the linearized eigenproblem, see [getNLEIGSEPS\(\)](#).

See also

[setTwoSided](#), [getNLEIGSFullBasis](#), [getNLEIGSEPS](#), [NEPNLEIGSSetFullBasis](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2606](#) <[slepc4py/SLEPc/NEP.pyx#L2606](#)>

setNLEIGSInterpolation(*tol=None, deg=None*)

Set the tolerance and maximum degree for the interpolation polynomial.

Collective.

Set the tolerance and maximum degree when building the interpolation via divided differences.

Parameters

- **tol** ([float](#) / [None](#)) – The tolerance to stop computing divided differences.
- **deg** ([int](#) / [None](#)) – The maximum degree of interpolation.

Return type

None

See also`getNLEIGSInterpolation`, `NEPNLEIGSSetInterpolation`**:sources:** `Source code at slepc4py/SLEPc/NEP.pyx:2555 <slepc4py/SLEPc/NEP.pyx#L2555>`**setNLEIGSLocking**(*lock*)

Toggle between locking and non-locking variants of the NLEIGS method.

Logically collective.

Parameters**lock** (*bool*) – True if the locking variant must be selected.**Return type**

None

Notes

The default is to lock converged eigenpairs when the method restarts. This behavior can be changed so that all directions are kept in the working subspace even if already converged to working accuracy (the non-locking variant).

See also`getNLEIGSLocking`, `NEPNLEIGSSetLocking`**:sources:** `Source code at slepc4py/SLEPc/NEP.pyx:2511 <slepc4py/SLEPc/NEP.pyx#L2511>`**setNLEIGSRKShifts**(*shifts*)

Set a list of shifts to be used in the Rational Krylov method.

Collective.

Parameters**shifts** (*Sequence*[*Scalar*]) – Values specifying the shifts.**Return type**

None

Notes

If only one shift is provided, the built subspace is equivalent to shift-and-invert Krylov-Schur (provided that the absolute convergence criterion is used). Otherwise, the rational Krylov variant is run.

See also`getNLEIGSRKShifts`, `getNLEIGSKSPs`, `NEPNLEIGSSetRKShifts`**:sources:** `Source code at slepc4py/SLEPc/NEP.pyx:2695 <slepc4py/SLEPc/NEP.pyx#L2695>`

setNLEIGSRestart(*keep*)

Set the restart parameter for the NLEIGS method.

Logically collective.

The proportion of basis vectors that must be kept after restart.

Parameters

keep (*float*) – The number of vectors to be kept at restart.

Return type

None

Notes

Allowed values are in the range [0.1,0.9]. The default is 0.5.

See also

[*getNLEIGSRestart*](#), [*NEPNLEIGSSetRestart*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2468](#) <[slepc4py/SLEPc/NEP.pyx#L2468](#)>

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all NEP options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all NEP option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

For example, to distinguish between the runtime options for two different NEP contexts, one could call:

```
N1.setOptionsPrefix("nep1_")
N2.setOptionsPrefix("nep2_")
```

See also

[*appendOptionsPrefix*](#), [*getOptionsPrefix*](#), [*NEPGetOptionsPrefix*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:354](#) <[slepc4py/SLEPc/NEP.pyx#L354](#)>

setProblemType(*problem_type*)

Set the type of the eigenvalue problem.

Logically collective.

Parameters

problem_type (*ProblemType*) – The problem type to be set.

Return type

None

Notes

This function is used to provide a hint to the *NEP* solver to exploit certain properties of the nonlinear eigenproblem. This hint may be used or not, depending on the solver. By default, no particular structure is assumed.

See also
[*getProblemType*](#), [*NEPSetProblemType*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:442 <slepc4py/SLEPc/NEP.pyx#L442>](#)

setRG(*rg*)

Set a region object associated to the eigensolver.

Collective.

Parameters

rg (*RG*) – The region context.

Return type

None

See also
[*getRG*](#), [*NEPSetRG*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:934 <slepc4py/SLEPc/NEP.pyx#L934>](#)

setRIIConstCorrectionTol(*cct*)

Set a flag to keep the tolerance used in the linear solver constant.

Logically collective.

Parameters

cct (*bool*) – If True, the `petsc4py.PETSc.KSP` relative tolerance is constant.

Return type

None

Notes

By default, an exponentially decreasing tolerance is set in the KSP used within the nonlinear iteration, so that each Newton iteration requests better accuracy than the previous one. The constant correction tolerance flag stops this behavior.

See also
[*getRIIConstCorrectionTol*](#), [*NEPRIISetConstCorrectionTol*](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:1917 <slepc4py/SLEPc/NEP.pyx#L1917>](#)

setRIIDeflationThreshold(*deftol*)

Set the threshold used to switch between deflated and non-deflated.

Logically collective.

Set the threshold value used to switch between deflated and non-deflated iteration.

Parameters

deftol (*float*) – The threshold value.

Return type

None

Notes

Normally, the solver iterates on the extended problem in order to deflate previously converged eigenpairs. If this threshold is set to a nonzero value, then once the residual error is below this threshold the solver will continue the iteration without deflation. The intention is to be able to improve the current eigenpair further, despite having previous eigenpairs with somewhat bad precision.

See also

getRIIDeflationThreshold, *NEPRIISetDeflationThreshold*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2050 <slepc4py/SLEPc/NEP.pyx#L2050>`

setRIIHermitian(*herm*)

Set a flag to use the Hermitian version of the solver.

Logically collective.

Set a flag to indicate if the Hermitian version of the scalar nonlinear equation must be used by the solver.

Parameters

herm (*bool*) – If True, the Hermitian version is used.

Return type

None

Notes

By default, the scalar nonlinear equation $x^*T(\sigma)^{-1}T(z)x = 0$ is solved at each step of the nonlinear iteration. When this flag is set the simpler form $x^*T(z)x = 0$ is used, which is supposed to be valid only for Hermitian problems.

See also

getRIIHermitian, *NEPRIISetHermitian*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2002 <slepc4py/SLEPc/NEP.pyx#L2002>`

setRIIKSP(*ksp*)

Set a linear solver object associated to the nonlinear eigensolver.

Collective.

Parameters

ksp (*KSP*) – The linear solver object.

Return type

None

See also`getRIIKSP`, `NEPRIISetKSP`**:sources:** `Source code at slepc4py/SLEPc/NEP.pyx:2100` `<slepc4py/SLEPc/NEP.pyx#L2100>`**setRIILagPreconditioner**(*lag*)

Set when the preconditioner is rebuilt in the nonlinear solve.

Logically collective.

Parameters**lag** (*int*) – 0 indicates NEVER rebuild, 1 means rebuild every time the Jacobian is computed within the nonlinear iteration, 2 means every second time the Jacobian is built, etc.**Return type**

None

See also`getRIILagPreconditioner`, `NEPRIISetLagPreconditioner`**:sources:** `Source code at slepc4py/SLEPc/NEP.pyx:1878` `<slepc4py/SLEPc/NEP.pyx#L1878>`**setRIIMaximumIterations**(*its*)

Set the max. number of inner iterations to be used in the RII solver.

Logically collective.

These are the Newton iterations related to the computation of the nonlinear Rayleigh functional.

Parameters**its** (*int*) – Maximum inner iterations.**Return type**

None

See also`getRIIMaximumIterations`, `NEPRIISetMaximumIterations`**:sources:** `Source code at slepc4py/SLEPc/NEP.pyx:1962` `<slepc4py/SLEPc/NEP.pyx#L1962>`**setRefine**(*ref*, *npart*=None, *tol*=None, *its*=None, *scheme*=None)

Set the refinement strategy used by the NEP object.

Logically collective.

Set the refinement strategy used by the NEP object and the associated parameters.

Parameters

- **ref** (*Refine*) – The refinement type.
- **npart** (*int* / *None*) – The number of partitions of the communicator.

- **tol** (*float* / *None*) – The convergence tolerance.
- **its** (*int* / *None*) – The maximum number of refinement iterations.
- **scheme** (*RefineScheme* / *None*) – Scheme for solving linear systems.

Return type

None

See also

getRefine, *NEPSetRefine*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:694 <slepc4py/SLEPc/NEP.pyx#L694>`

setSLPDeflationThreshold(*deftol*)

Set the threshold used to switch between deflated and non-deflated.

Logically collective.

Parameters

deftol (*float*) – The threshold value.

Return type

None

Notes

Normally, the solver iterates on the extended problem in order to deflate previously converged eigenpairs. If this threshold is set to a nonzero value, then once the residual error is below this threshold the solver will continue the iteration without deflation. The intention is to be able to improve the current eigenpair further, despite having previous eigenpairs with somewhat bad precision.

See also

getSLPDeflationThreshold, *NEPSLPSetDeflationThreshold*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2139 <slepc4py/SLEPc/NEP.pyx#L2139>`

setSLPEPS(*eps*)

Set a linear eigensolver object associated to the nonlinear eigensolver.

Collective.

Parameters

eps (*EPS*) – The linear eigensolver.

Return type

None

See also

getSLPEPS, *NEPSLPSetEPS*

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:2186 <slepc4py/SLEPc/NEP.pyx#L2186>`

setSLPEPSLeft(*eps*)

Set a linear eigensolver object associated to the nonlinear eigensolver.

Collective.

Used to compute left eigenvectors in the two-sided variant of SLP.

Parameters

eps (*EPS*) – The linear eigensolver.

Return type

None

See also

[setTwoSided](#), [setSLPEPS](#), [getSLPEPSLeft](#), [NEPSLPSetEPSLeft](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2223](#) <[slepc4py/SLEPc/NEP.pyx#L2223](#)>

setSLPKSP(*ksp*)

Set a linear solver object associated to the nonlinear eigensolver.

Collective.

Parameters

ksp (*KSP*) – The linear solver object.

Return type

None

See also

[getSLPKSP](#), [NEPSLPSetKSP](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:2262](#) <[slepc4py/SLEPc/NEP.pyx#L2262](#)>

setSplitOperator(*A*, *f*, *structure=None*)

Set the operator of the nonlinear eigenvalue problem in split form.

Collective.

Parameters

- **A** (*petsc4py.PETSc.Mat* / *list*[*petsc4py.PETSc.Mat*]) – Coefficient matrices of the split form.
- **f** (*FN* / *list*[*FN*]) – Scalar functions of the split form.
- **structure** (*petsc4py.PETSc.Mat.Structure* / *None*) – Structure flag for matrices.

Return type

None

Notes

The nonlinear operator is written as $T(\lambda) = \sum_i A_i f_i(\lambda)$, for $i = 1, \dots, n$. The derivative $T'(\lambda)$ can be obtained using the derivatives of f_i .

The structure flag provides information about A_i 's nonzero pattern.

This function must be called before `setUp()`. If it is called again after `setUp()` then the `NEP` object is reset.

See also

`getSplitOperator`, `NEPSetSplitOperator`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1632 <slepc4py/SLEPc/NEP.pyx#L1632>`

setSplitPreconditioner(*P*, *structure=None*)

Set the operator in split form.

Collective.

Set the operator in split form from which to build the preconditioner to be used when solving the nonlinear eigenvalue problem in split form.

Parameters

- **P** (`petsc4py.PETSc.Mat` / `list[petsc4py.PETSc.Mat]`) – Coefficient matrices of the split preconditioner.
- **structure** (`petsc4py.PETSc.Mat.Structure` / `None`) – Structure flag for matrices.

Return type

`None`

See also

`getSplitPreconditioner`, `NEPSetSplitPreconditioner`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1719 <slepc4py/SLEPc/NEP.pyx#L1719>`

setStoppingTest(*stopping*, *args=None*, *kargs=None*)

Set a function to decide when to stop the outer iteration of the eigensolver.

Logically collective.

See also

`getStoppingTest`, `NEPSetStoppingTestFunction`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1029 <slepc4py/SLEPc/NEP.pyx#L1029>`

Parameters

- **stopping** (`NEPStoppingFunction` / `None`)
- **args** (`tuple[Any, ...]` / `None`)
- **kargs** (`dict[str, Any]` / `None`)

Return type

`None`

setTarget(*target*)

Set the value of the target.

Logically collective.

Parameters

target ([Scalar](#)) – The value of the target.

Return type

[None](#)

Notes

The target is a scalar value used to determine the portion of the spectrum of interest. It is used in combination with [setWhichEigenpairs\(\)](#).

When PETSc is built with real scalars, it is not possible to specify a complex target.

See also

[getTarget](#), [setWhichEigenpairs](#), [NEPSetTarget](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:544 <slepc4py/SLEPc/NEP.pyx#L544>](#)

setTolerances(*tol=None, maxit=None*)

Set the tolerance and max. iteration count used in convergence tests.

Logically collective.

Parameters

- **tol** ([float](#) / [None](#)) – The convergence tolerance.
- **maxit** ([int](#) / [None](#)) – The maximum number of iterations.

Return type

[None](#)

Notes

Use [DETERMINE](#) for `max_it` to assign a reasonably good value, which is dependent on the solution method.

See also

[getTolerances](#), [NEPSetTolerances](#)

:sources: [Source code at slepc4py/SLEPc/NEP.pyx:596 <slepc4py/SLEPc/NEP.pyx#L596>](#)

setTrackAll(*trackall*)

Set if the solver must compute the residual of all approximate eigenpairs.

Logically collective.

Parameters

trackall ([bool](#)) – Whether to compute all residuals or not.

Return type

[None](#)

See also

[`getTrackAll`](#), [`NEPSetTrackAll`](#)

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:777 <slepc4py/SLEPc/NEP.pyx#L777>``

setTwoSided(*twosided*)

Set the solver to use a two-sided variant.

Logically collective.

Set the solver to use a two-sided variant so that left eigenvectors are also computed.

Parameters

twosided (*bool*) – Whether the two-sided variant is to be used or not.

Return type

`None`

Notes

If the user sets `twosided` to `True` then the solver uses a variant of the algorithm that computes both right and left eigenvectors. This is usually much more costly. This option is not available in all solvers.

When using two-sided solvers, the problem matrices must have both the `Mat.mult` and `Mat.multTranspose` operations defined.

See also

[`getTwoSided`](#), [`getLeftEigenvector`](#), [`NEPSetTwoSided`](#)

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:1803 <slepc4py/SLEPc/NEP.pyx#L1803>``

setType(*nep_type*)

Set the particular solver to be used in the NEP object.

Logically collective.

Parameters

nep_type (*Type* / *str*) – The solver to be used.

Return type

`None`

Notes

The default is `RII`. Normally, it is best to use `setFromOptions()` and then set the NEP type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also

[`getType`](#), [`NEPSetType`](#)

:sources: ``Source code at slepc4py/SLEPc/NEP.pyx:289 <slepc4py/SLEPc/NEP.pyx#L289>``

setUp()

Set up all the internal data structures.

Collective.

Notes

Sets up all the internal data structures necessary for the execution of the eigensolver.

This function need not be called explicitly in most cases, since `solve()` calls it. It can be useful when one wants to measure the set-up time separately from the solve time.

See also

`solve`, `NEPSetUp`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1174 <slepc4py/SLEPc/NEP.pyx#L1174>`

Return type

None

setWhichEigenpairs(which)

Set which portion of the spectrum is to be sought.

Logically collective.

Parameters

which (*Which*) – The portion of the spectrum to be sought by the solver.

Return type

None

Notes

Not all eigensolvers implemented in NEP account for all the possible values. Also, some values make sense only for certain types of problems. If SLEPc is compiled for real numbers *NEP.Which.LARGEST_IMAGINARY* and *NEP.Which.SMALLEST_IMAGINARY* use the absolute value of the imaginary part for eigenvalue selection.

The target is a scalar value provided with `setTarget()`.

The criterion *NEP.Which.TARGET_IMAGINARY* is available only in case PETSc and SLEPc have been built with complex scalars.

NEP.Which.ALL is intended for use in the context of the *PEP.Type.CISS* solver for computing all eigenvalues in a region.

See also

`getWhichEigenpairs`, `setTarget`, `PEPSetWhichEigenpairs`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:486 <slepc4py/SLEPc/NEP.pyx#L486>`

solve()

Solve the nonlinear eigenproblem.

Collective.

Notes

`solve()` will return without generating an error regardless of whether all requested solutions were computed or not. Call `getConverged()` to get the actual number of computed solutions, and `getConvergedReason()` to determine if the solver converged or failed and why.

See also

`setUp`, `getConverged`, `getConvergedReason`, `NEPSolve`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1195 <slepc4py/SLEPc/NEP.pyx#L1195>`

Return type

`None`

valuesView(*viewer=None*)

Display the computed eigenvalues in a viewer.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

`None`

See also

`solve`, `vectorsView`, `errorView`, `NEPValuesView`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1460 <slepc4py/SLEPc/NEP.pyx#L1460>`

vectorsView(*viewer=None*)

Output computed eigenvectors to a viewer.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

`None`

See also

`solve`, `valuesView`, `errorView`, `NEPVectorsView`

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:1479 <slepc4py/SLEPc/NEP.pyx#L1479>`

view(*viewer=None*)

Print the NEP data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

NEPView

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:223 <slepc4py/SLEPc/NEP.pyx#L223>`

Attributes Documentation

bv

The basis vectors (*BV*) object associated.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3058 <slepc4py/SLEPc/NEP.pyx#L3058>`

ds

The direct solver (*DS*) object associated.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3072 <slepc4py/SLEPc/NEP.pyx#L3072>`

max_it

The maximum iteration count used by the NEP convergence tests.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3044 <slepc4py/SLEPc/NEP.pyx#L3044>`

problem_type

The problem type from the NEP object.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3016 <slepc4py/SLEPc/NEP.pyx#L3016>`

rg

The region (*RG*) object associated.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3065 <slepc4py/SLEPc/NEP.pyx#L3065>`

target

The value of the target.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3030 <slepc4py/SLEPc/NEP.pyx#L3030>`

tol

The tolerance used by the NEP convergence tests.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3037 <slepc4py/SLEPc/NEP.pyx#L3037>`

track_all

Compute the residual of all approximate eigenpairs.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3051 <slepc4py/SLEPc/NEP.pyx#L3051>`

which

The portion of the spectrum to be sought.

:sources: `Source code at slepc4py/SLEPc/NEP.pyx:3023 <slepc4py/SLEPc/NEP.pyx#L3023>`

```
__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP

class slepc4py.SLEPc.PEP

Bases: [Object](#)

Polynomial Eigenvalue Problem Solver.

The Polynomial Eigenvalue Problem ([PEP](#)) solver is the object provided by slepc4py for specifying a polynomial eigenvalue problem. Apart from the specific solvers for this type of problems, there is an [EPS](#)-based solver, i.e., it uses a solver from [EPS](#) to solve a generalized eigenproblem obtained after linearization.

Enumerations

Basis	PEP basis type for the representation of the polynomial.
CISSExtraction	PEP CISS extraction technique.
Conv	PEP convergence test.
ConvergedReason	PEP convergence reasons.
ErrorType	PEP error type to assess accuracy of computed solutions.
Extract	PEP extraction strategy used.
JDProjection	PEP type of projection to be used in the Jacobi-Davidson solver.
ProblemType	PEP problem type.
Refine	PEP refinement strategy.
RefineScheme	PEP scheme for solving linear systems during iterative refinement.
Scale	PEP scaling strategy.
Stop	PEP stopping test.
Type	PEP type.
Which	PEP desired part of spectrum.

slepc4py.SLEPc.PEP.Basis

class slepc4py.SLEPc.PEP.Basis

Bases: [object](#)

PEP basis type for the representation of the polynomial.

- [MONOMIAL](#): Monomials (default).
- [CHEBYSHEV1](#): Chebyshev polynomials of the 1st kind.
- [CHEBYSHEV2](#): Chebyshev polynomials of the 2nd kind.
- [LEGENDRE](#): Legendre polynomials.
- [LAGUERRE](#): Laguerre polynomials.
- [HERMITE](#): Hermite polynomials.

See also
PEPBasis

Attributes Summary

CHEBYSHEV1	Constant CHEBYSHEV1 of type <code>int</code>
CHEBYSHEV2	Constant CHEBYSHEV2 of type <code>int</code>
HERMITE	Constant HERMITE of type <code>int</code>
LAGUERRE	Constant LAGUERRE of type <code>int</code>
LEGENDRE	Constant LEGENDRE of type <code>int</code>
MONOMIAL	Constant MONOMIAL of type <code>int</code>

Attributes Documentation

CHEBYSHEV1: `int` = **CHEBYSHEV1**
Constant CHEBYSHEV1 of type `int`

CHEBYSHEV2: `int` = **CHEBYSHEV2**
Constant CHEBYSHEV2 of type `int`

HERMITE: `int` = **HERMITE**
Constant HERMITE of type `int`

LAGUERRE: `int` = **LAGUERRE**
Constant LAGUERRE of type `int`

LEGENDRE: `int` = **LEGENDRE**
Constant LEGENDRE of type `int`

MONOMIAL: `int` = **MONOMIAL**
Constant MONOMIAL of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

`slepc4py.SLEPc.PEP.CISSExtraction`

class `slepc4py.SLEPc.PEP.CISSExtraction`

Bases: `object`

PEP CISS extraction technique.

- [RITZ](#): Ritz extraction.
- [HANKEL](#): Extraction via Hankel eigenproblem.
- [CAA](#): Communication-avoiding Arnoldi.

See also
PEPCISSExtraction

Attributes Summary

<i>CAA</i>	Constant CAA of type <code>int</code>
<i>HANKEL</i>	Constant HANKEL of type <code>int</code>
<i>RITZ</i>	Constant RITZ of type <code>int</code>

Attributes Documentation

CAA: `int` = CAA

Constant CAA of type `int`

HANKEL: `int` = HANKEL

Constant HANKEL of type `int`

RITZ: `int` = RITZ

Constant RITZ of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.PEP.Conv

class `slepc4py.SLEPc.PEP.Conv`

Bases: `object`

PEP convergence test.

- *ABS*: Absolute convergence test.
- *REL*: Convergence test relative to the eigenvalue.
- *NORM*: Convergence test relative to the matrix norms.
- *USER*: User-defined convergence test.

See also

PEPConv

Attributes Summary

<i>ABS</i>	Constant ABS of type <code>int</code>
<i>NORM</i>	Constant NORM of type <code>int</code>
<i>REL</i>	Constant REL of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

ABS: `int` = ABS

Constant ABS of type `int`

NORM: `int` = NORM

Constant NORM of type `int`

```
REL: int = REL
    Constant REL of type int

USER: int = USER
    Constant USER of type int

__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.ConvergedReason

class slepc4py.SLEPc.PEP.ConvergedReason

Bases: `object`

PEP convergence reasons.

- *CONVERGED_TOL*: All eigenpairs converged to requested tolerance.
- *CONVERGED_USER*: User-defined convergence criterion satisfied.
- *DIVERGED_ITS*: Maximum number of iterations exceeded.
- *DIVERGED_BREAKDOWN*: Solver failed due to breakdown.
- *DIVERGED_SYMMETRY_LOST*: Lanczos-type method could not preserve symmetry.
- *CONVERGED_ITERATING*: Iteration not finished yet.

See also

`PEPConvergedReason`

Attributes Summary

<i>CONVERGED_ITERATING</i>	Constant <i>CONVERGED_ITERATING</i> of type <code>int</code>
<i>CONVERGED_TOL</i>	Constant <i>CONVERGED_TOL</i> of type <code>int</code>
<i>CONVERGED_USER</i>	Constant <i>CONVERGED_USER</i> of type <code>int</code>
<i>DIVERGED_BREAKDOWN</i>	Constant <i>DIVERGED_BREAKDOWN</i> of type <code>int</code>
<i>DIVERGED_ITS</i>	Constant <i>DIVERGED_ITS</i> of type <code>int</code>
<i>DIVERGED_SYMMETRY_LOST</i>	Constant <i>DIVERGED_SYMMETRY_LOST</i> of type <code>int</code>
<i>ITERATING</i>	Constant <i>ITERATING</i> of type <code>int</code>

Attributes Documentation

CONVERGED_ITERATING: int = CONVERGED_ITERATING

Constant *CONVERGED_ITERATING* of type `int`

CONVERGED_TOL: int = CONVERGED_TOL

Constant *CONVERGED_TOL* of type `int`

CONVERGED_USER: int = CONVERGED_USER

Constant *CONVERGED_USER* of type `int`

DIVERGED_BREAKDOWN: int = DIVERGED_BREAKDOWN

Constant *DIVERGED_BREAKDOWN* of type `int`

```
DIVERGED_ITS: int = DIVERGED_ITS
    Constant DIVERGED_ITS of type int

DIVERGED_SYMMETRY_LOST: int = DIVERGED_SYMMETRY_LOST
    Constant DIVERGED_SYMMETRY_LOST of type int

ITERATING: int = ITERATING
    Constant ITERATING of type int

__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.ErrorType

```
class slepc4py.SLEPc.PEP.ErrorType
    Bases: object

    PEP error type to assess accuracy of computed solutions.

    • ABSOLUTE: Absolute error.
    • RELATIVE: Relative error.
    • BACKWARD: Backward error.
```

See also

PEPErrorType

Attributes Summary

ABSOLUTE	Constant ABSOLUTE of type int
BACKWARD	Constant BACKWARD of type int
RELATIVE	Constant RELATIVE of type int

Attributes Documentation

```
ABSOLUTE: int = ABSOLUTE
    Constant ABSOLUTE of type int

BACKWARD: int = BACKWARD
    Constant BACKWARD of type int

RELATIVE: int = RELATIVE
    Constant RELATIVE of type int

__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.Extract

class slepc4py.SLEPc.PEP.Extract

Bases: `object`

PEP extraction strategy used.

PEP extraction strategy used to obtain eigenvectors of the PEP from the eigenvectors of the linearization.

- *NONE*: Use the first block.
- *NORM*: Use the first or last block depending on norm of H.
- *RESIDUAL*: Use the block with smallest residual.
- *STRUCTURED*: Combine all blocks in a certain way.

See also

PEPExtract

Attributes Summary

<i>NONE</i>	Constant NONE of type <code>int</code>
<i>NORM</i>	Constant NORM of type <code>int</code>
<i>RESIDUAL</i>	Constant RESIDUAL of type <code>int</code>
<i>STRUCTURED</i>	Constant STRUCTURED of type <code>int</code>

Attributes Documentation

NONE: `int` = NONE

Constant NONE of type `int`

NORM: `int` = NORM

Constant NORM of type `int`

RESIDUAL: `int` = RESIDUAL

Constant RESIDUAL of type `int`

STRUCTURED: `int` = STRUCTURED

Constant STRUCTURED of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.PEP.JDProjection

class slepc4py.SLEPc.PEP.JDProjection

Bases: `object`

PEP type of projection to be used in the Jacobi-Davidson solver.

- *HARMONIC*: Harmonic projection.
- *ORTHOGONAL*: Orthogonal projection.

See also
PEPJDDProjection

Attributes Summary

<i>HARMONIC</i>	Constant HARMONIC of type <code>int</code>
<i>ORTHOGONAL</i>	Constant ORTHOGONAL of type <code>int</code>

Attributes Documentation

HARMONIC: `int` = HARMONIC

Constant HARMONIC of type `int`

ORTHOGONAL: `int` = ORTHOGONAL

Constant ORTHOGONAL of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.PEP.ProblemType

class slepc4py.SLEPc.PEP.ProblemType

Bases: `object`

PEP problem type.

- *GENERAL*: No structure.
- *HERMITIAN*: Hermitian structure.
- *HYPERBOLIC*: QEP with Hermitian matrices, $M > 0$, $(x^T C x)^2 > 4(x^T M x)(x^T K x)$.
- *GYROSCOPIC*: QEP with M , K Hermitian, $M > 0$, C skew-Hermitian.

See also
PEPProblemType

Attributes Summary

<i>GENERAL</i>	Constant GENERAL of type <code>int</code>
<i>GYROSCOPIC</i>	Constant GYROSCOPIC of type <code>int</code>
<i>HERMITIAN</i>	Constant HERMITIAN of type <code>int</code>
<i>HYPERBOLIC</i>	Constant HYPERBOLIC of type <code>int</code>

Attributes Documentation

GENERAL: `int` = GENERAL

Constant GENERAL of type `int`

```
GYROSCOPIC: int = GYROSCOPIC
    Constant GYROSCOPIC of type int
HERMITIAN: int = HERMITIAN
    Constant HERMITIAN of type int
HYPERBOLIC: int = HYPERBOLIC
    Constant HYPERBOLIC of type int
__init__()
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.Refine

```
class slepc4py.SLEPc.PEP.Refine
    Bases: object
    PEP refinement strategy.
    • NONE: No refinement.
    • SIMPLE: Refine eigenpairs one by one.
    • MULTIPLE: Refine all eigenpairs simultaneously (invariant pair).
```

See also
PEPRefine

Attributes Summary

<i>MULTIPLE</i>	Constant MULTIPLE of type int
<i>NONE</i>	Constant NONE of type int
<i>SIMPLE</i>	Constant SIMPLE of type int

Attributes Documentation

```
MULTIPLE: int = MULTIPLE
    Constant MULTIPLE of type int
NONE: int = NONE
    Constant NONE of type int
SIMPLE: int = SIMPLE
    Constant SIMPLE of type int
__init__()
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.RefineScheme

class slepc4py.SLEPc.PEP.RefineScheme

Bases: `object`

PEP scheme for solving linear systems during iterative refinement.

- *SCHUR*: Schur complement.
- *MBE*: Mixed block elimination.
- *EXPLICIT*: Build the explicit matrix.

See also

`PEPRefineScheme`

Attributes Summary

<i>EXPLICIT</i>	Constant EXPLICIT of type <code>int</code>
<i>MBE</i>	Constant MBE of type <code>int</code>
<i>SCHUR</i>	Constant SCHUR of type <code>int</code>

Attributes Documentation

EXPLICIT: `int` = **EXPLICIT**

Constant EXPLICIT of type `int`

MBE: `int` = **MBE**

Constant MBE of type `int`

SCHUR: `int` = **SCHUR**

Constant SCHUR of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.PEP.Scale

class slepc4py.SLEPc.PEP.Scale

Bases: `object`

PEP scaling strategy.

- *NONE*: No scaling.
- *SCALAR*: Parameter scaling.
- *DIAGONAL*: Diagonal scaling.
- *BOTH*: Both parameter and diagonal scaling.

See also

`PEPScale`

Attributes Summary

<i>BOTH</i>	Constant BOTH of type <code>int</code>
<i>DIAGONAL</i>	Constant DIAGONAL of type <code>int</code>
<i>NONE</i>	Constant NONE of type <code>int</code>
<i>SCALAR</i>	Constant SCALAR of type <code>int</code>

Attributes Documentation

BOTH: `int` = **BOTH**

Constant BOTH of type `int`

DIAGONAL: `int` = **DIAGONAL**

Constant DIAGONAL of type `int`

NONE: `int` = **NONE**

Constant NONE of type `int`

SCALAR: `int` = **SCALAR**

Constant SCALAR of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.PEP.Stop

class `slepc4py.SLEPc.PEP.Stop`

Bases: `object`

PEP stopping test.

- *BASIC*: Default stopping test.
- *USER*: User-defined stopping test.

See also

`PEPStop`

Attributes Summary

<i>BASIC</i>	Constant BASIC of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

BASIC: `int` = **BASIC**

Constant BASIC of type `int`

USER: `int` = **USER**

Constant USER of type `int`

```
__init__()
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.Type

class slepc4py.SLEPc.PEP.Type

Bases: `object`

PEP type.

- *TOAR*: Two-level orthogonal Arnoldi.
- *STOAR*: Symmetric TOAR.
- *QARNOLDI*: Q-Arnoldi for quadratic problems.
- *LINEAR*: Linearization via EPS.
- *JD*: Polynomial Jacobi-Davidson.
- *CISS*: Contour integral spectrum slice.

See also
PEPType

Attributes Summary

<i>CISS</i>	Object CISS of type <code>str</code>
<i>JD</i>	Object JD of type <code>str</code>
<i>LINEAR</i>	Object LINEAR of type <code>str</code>
<i>QARNOLDI</i>	Object QARNOLDI of type <code>str</code>
<i>STOAR</i>	Object STOAR of type <code>str</code>
<i>TOAR</i>	Object TOAR of type <code>str</code>

Attributes Documentation

CISS: `str` = CISS

Object CISS of type `str`

JD: `str` = JD

Object JD of type `str`

LINEAR: `str` = LINEAR

Object LINEAR of type `str`

QARNOLDI: `str` = QARNOLDI

Object QARNOLDI of type `str`

STOAR: `str` = STOAR

Object STOAR of type `str`

TOAR: `str` = TOAR

Object TOAR of type `str`

```
__init__()  
  
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.PEP.Which

class slepc4py.SLEPc.PEP.**Which**

Bases: `object`

PEP desired part of spectrum.

- *LARGEST_MAGNITUDE*: Largest magnitude (default).
- *SMALLEST_MAGNITUDE*: Smallest magnitude.
- *LARGEST_REAL*: Largest real parts.
- *SMALLEST_REAL*: Smallest real parts.
- *LARGEST_IMAGINARY*: Largest imaginary parts in magnitude.
- *SMALLEST_IMAGINARY*: Smallest imaginary parts in magnitude.
- *TARGET_MAGNITUDE*: Closest to target (in magnitude).
- *TARGET_REAL*: Real part closest to target.
- *TARGET_IMAGINARY*: Imaginary part closest to target.
- *ALL*: All eigenvalues in an interval.
- *USER*: User-defined criterion.

See also

PEPWhich

Attributes Summary

<i>ALL</i>	Constant ALL of type <code>int</code>
<i>LARGEST_IMAGINARY</i>	Constant LARGEST_IMAGINARY of type <code>int</code>
<i>LARGEST_MAGNITUDE</i>	Constant LARGEST_MAGNITUDE of type <code>int</code>
<i>LARGEST_REAL</i>	Constant LARGEST_REAL of type <code>int</code>
<i>SMALLEST_IMAGINARY</i>	Constant SMALLEST_IMAGINARY of type <code>int</code>
<i>SMALLEST_MAGNITUDE</i>	Constant SMALLEST_MAGNITUDE of type <code>int</code>
<i>SMALLEST_REAL</i>	Constant SMALLEST_REAL of type <code>int</code>
<i>TARGET_IMAGINARY</i>	Constant TARGET_IMAGINARY of type <code>int</code>
<i>TARGET_MAGNITUDE</i>	Constant TARGET_MAGNITUDE of type <code>int</code>
<i>TARGET_REAL</i>	Constant TARGET_REAL of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

ALL: `int` = ALL
Constant ALL of type `int`

LARGEST_IMAGINARY: int = LARGEST_IMAGINARY
 Constant LARGEST_IMAGINARY of type int

LARGEST_MAGNITUDE: int = LARGEST_MAGNITUDE
 Constant LARGEST_MAGNITUDE of type int

LARGEST_REAL: int = LARGEST_REAL
 Constant LARGEST_REAL of type int

SMALLEST_IMAGINARY: int = SMALLEST_IMAGINARY
 Constant SMALLEST_IMAGINARY of type int

SMALLEST_MAGNITUDE: int = SMALLEST_MAGNITUDE
 Constant SMALLEST_MAGNITUDE of type int

SMALLEST_REAL: int = SMALLEST_REAL
 Constant SMALLEST_REAL of type int

TARGET_IMAGINARY: int = TARGET_IMAGINARY
 Constant TARGET_IMAGINARY of type int

TARGET_MAGNITUDE: int = TARGET_MAGNITUDE
 Constant TARGET_MAGNITUDE of type int

TARGET_REAL: int = TARGET_REAL
 Constant TARGET_REAL of type int

USER: int = USER
 Constant USER of type int

__init__()

classmethod __new__(*args, **kwargs)

Methods Summary

<i>appendOptionsPrefix</i> ([prefix])	Append to the prefix used for searching for all PEP options in the database.
<i>cancelMonitor</i> ()	Clear all monitors for a <i>PEP</i> object.
<i>computeError</i> (i[, etype])	Compute the error associated with the i-th computed eigenpair.
<i>create</i> ([comm])	Create the PEP object.
<i>destroy</i> ()	Destroy the PEP object.
<i>errorView</i> ([etype, viewer])	Display the errors associated with the computed solution.
<i>getBV</i> ()	Get the basis vectors object associated to the eigen-solver.
<i>getBasis</i> ()	Get the type of polynomial basis used.
<i>getCISSExtraction</i> ()	Get the extraction technique used in the CISS solver.
<i>getCISSKSPs</i> ()	Get the array of linear solver objects associated with the CISS solver.
<i>getCISSRefinement</i> ()	Get the values of various refinement parameters in the CISS solver.
<i>getCISSSizes</i> ()	Get the values of various size parameters in the CISS solver.

continues on next page

Table 83 – continued from previous page

<code>getCISSThreshold()</code>	Get the values of various threshold parameters in the CISS solver.
<code>getConverged()</code>	Get the number of converged eigenpairs.
<code>getConvergedReason()</code>	Get the reason why the <code>solve()</code> iteration was stopped.
<code>getConvergenceTest()</code>	Get the method used to compute the error estimate used in the convergence test.
<code>getDS()</code>	Get the direct solver associated to the eigensolver.
<code>getDimensions()</code>	Get the number of eigenvalues to compute and the dimension of the subspace.
<code>getEigenpair(i[, Vr, Vi])</code>	Get the i-th solution of the eigenproblem as computed by <code>solve()</code> .
<code>getEigenvalueComparison()</code>	Get the eigenvalue comparison function.
<code>getErrorEstimate(i)</code>	Get the error estimate associated to the i-th computed eigenpair.
<code>getExtract()</code>	Get the extraction technique used by the <code>PEP</code> object.
<code>getInterval()</code>	Get the computational interval for spectrum slicing.
<code>getIterationNumber()</code>	Get the current iteration number.
<code>getJDFix()</code>	Get threshold for changing the target in the correction equation.
<code>getJDMINimalityIndex()</code>	Get the maximum allowed value of the minimality index.
<code>getJDProjection()</code>	Get the type of projection to be used in the Jacobi-Davidson solver.
<code>getJDRestart()</code>	Get the restart parameter used in the Jacobi-Davidson method.
<code>getJDReusePreconditioner()</code>	Get the flag for reusing the preconditioner.
<code>getLinearEPS()</code>	Get the eigensolver object associated to the polynomial eigenvalue solver.
<code>getLinearExplicitMatrix()</code>	Get if the matrices for the linearization are built explicitly.
<code>getLinearLinearization()</code>	Get the coeffs.
<code>getMonitor()</code>	Get the list of monitor functions.
<code>getOperators()</code>	Get the matrices associated with the eigenvalue problem.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all PEP options in the database.
<code>getProblemType()</code>	Get the problem type from the PEP object.
<code>getQArnoldiLocking()</code>	Get the locking flag used in the Q-Arnoldi method.
<code>getQArnoldiRestart()</code>	Get the restart parameter used in the Q-Arnoldi method.
<code>getRG()</code>	Get the region object associated to the eigensolver.
<code>getRefine()</code>	Get the refinement strategy used by the PEP object.
<code>getRefineKSP()</code>	Get the KSP object used by the eigensolver in the refinement phase.
<code>getST()</code>	Get the spectral transformation object associated to the eigensolver.
<code>getSTOARCheckEigenvalueType()</code>	Get the flag for the eigenvalue type check in spectrum slicing.
<code>getSTOARDetectZeros()</code>	Get the flag that enforces zero detection in spectrum slicing.
<code>getSTOARDimensions()</code>	Get the dimensions used for each subsolve step.

continues on next page

Table 83 – continued from previous page

<i>getSTOARInertias()</i>	Get the values of the shifts and their corresponding inertias.
<i>getSTOARLinearization()</i>	Get the coefficients that define the linearization of a quadratic eigenproblem.
<i>getSTOARLocking()</i>	Get the locking flag used in the STOAR method.
<i>getScale([DI, Dr])</i>	Get the strategy used for scaling the polynomial eigenproblem.
<i>getStoppingTest()</i>	Get the stopping test function.
<i>getTOARLocking()</i>	Get the locking flag used in the TOAR method.
<i>getTOARRestart()</i>	Get the restart parameter used in the TOAR method.
<i>getTarget()</i>	Get the value of the target.
<i>getTolerances()</i>	Get the tolerance and maximum iteration count.
<i>getTrackAll()</i>	Get the flag indicating whether all residual norms must be computed.
<i>getType()</i>	Get the PEP type of this object.
<i>getWhichEigenpairs()</i>	Get which portion of the spectrum is to be sought.
<i>reset()</i>	Reset the PEP object.
<i>setBV(bv)</i>	Set a basis vectors object associated to the eigensolver.
<i>setBasis(basis)</i>	Set the type of polynomial basis used.
<i>setCISSExtraction(extraction)</i>	Set the extraction technique used in the CISS solver.
<i>setCISSRefinement([inner, blsize])</i>	Set the values of various refinement parameters in the CISS solver.
<i>setCISSSizes([ip, bs, ms, npart, bsmax, ...])</i>	Set the values of various size parameters in the CISS solver.
<i>setCISSThreshold([delta, spur])</i>	Set the values of various threshold parameters in the CISS solver.
<i>setConvergenceTest(conv)</i>	Set how to compute the error estimate used in the convergence test.
<i>setDS(ds)</i>	Set a direct solver object associated to the eigensolver.
<i>setDimensions([nev, ncv, mpd])</i>	Set the number of eigenvalues to compute and the dimension of the subspace.
<i>setEigenvalueComparison(comparison[, args, ...])</i>	Set an eigenvalue comparison function.
<i>setExtract(extract)</i>	Set the extraction strategy to be used.
<i>setFromOptions()</i>	Set PEP options from the options database.
<i>setInitialSpace(space)</i>	Set the initial space from which the eigensolver starts to iterate.
<i>setInterval(inta, intb)</i>	Set the computational interval for spectrum slicing.
<i>setJDFix(fix)</i>	Set the threshold for changing the target in the correction equation.
<i>setJDMINimalityIndex(flag)</i>	Set the maximum allowed value for the minimality index.
<i>setJDProjection(proj)</i>	Set the type of projection to be used in the Jacobi-Davidson solver.
<i>setJDRestart(keep)</i>	Set the restart parameter for the Jacobi-Davidson method.
<i>setJDReusePreconditioner(flag)</i>	Set a flag indicating whether the preconditioner must be reused or not.
<i>setLinearEPS(eps)</i>	Set an eigensolver object associated to the polynomial eigenvalue solver.

continues on next page

Table 83 – continued from previous page

<code>setLinearExplicitMatrix(flag)</code>	Set flag to explicitly build the matrices for the linearization.
<code>setLinearLinearization([alpha, beta])</code>	Set the coefficients that define the linearization of a quadratic eigenproblem.
<code>setMonitor(monitor[, args, kargs])</code>	Append a monitor function to the list of monitors.
<code>setOperators(operators)</code>	Set the matrices associated with the eigenvalue problem.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all PEP options in the database.
<code>setProblemType(problem_type)</code>	Set the type of the polynomial eigenvalue problem.
<code>setQArnoldiLocking(lock)</code>	Toggle between locking and non-locking variants of the Q-Arnoldi method.
<code>setQArnoldiRestart(keep)</code>	Set the restart parameter for the Q-Arnoldi method.
<code>setRG(rg)</code>	Set a region object associated to the eigensolver.
<code>setRefine(ref[, npart, tol, its, scheme])</code>	Set the refinement strategy used by the PEP object.
<code>setST(st)</code>	Set a spectral transformation object associated to the eigensolver.
<code>setSTOARCheckEigenvalueType(flag)</code>	Set flag to check if all eigenvalues have the same definite type.
<code>setSTOARDetectZeros(detect)</code>	Set flag to enforce detection of zeros during the factorizations.
<code>setSTOARDimensions([nev, ncv, mpd])</code>	Set the dimensions used for each subsolve step.
<code>setSTOARLinearization([alpha, beta])</code>	Set the coefficients that define the linearization of a quadratic eigenproblem.
<code>setSTOARLocking(lock)</code>	Toggle between locking and non-locking variants of the STOAR method.
<code>setScale(scale[, alpha, DI, Dr, its, lbda])</code>	Set the scaling strategy to be used.
<code>setStoppingTest(stopping[, args, kargs])</code>	Set a function to decide when to stop the outer iteration of the eigensolver.
<code>setTOARLocking(lock)</code>	Toggle between locking and non-locking variants of the TOAR method.
<code>setTOARRestart(keep)</code>	Set the restart parameter for the TOAR method.
<code>setTarget(target)</code>	Set the value of the target.
<code>setTolerances([tol, max_it])</code>	Set the tolerance and maximum iteration count.
<code>setTrackAll(trackall)</code>	Set flag to compute the residual of all approximate eigenpairs.
<code>setType(pep_type)</code>	Set the particular solver to be used in the PEP object.
<code>setUp()</code>	Set up all the internal data structures.
<code>setWhichEigenpairs(which)</code>	Set which portion of the spectrum is to be sought.
<code>solve()</code>	Solve the polynomial eigenproblem.
<code>valuesView([viewer])</code>	Display the computed eigenvalues in a viewer.
<code>vectorsView([viewer])</code>	Output computed eigenvectors to a viewer.
<code>view([viewer])</code>	Print the PEP data structure.

Attributes Summary

<code>bv</code>	The basis vectors (<i>BV</i>) object associated.
<code>ds</code>	The direct solver (<i>DS</i>) object associated.
<code>extract</code>	The type of extraction technique to be employed.
<code>max_it</code>	The maximum iteration count.
<code>problem_type</code>	The type of the eigenvalue problem.

continues on next page

Table 84 – continued from previous page

<i>rg</i>	The region (<i>RG</i>) object associated.
<i>st</i>	The spectral transformation (<i>ST</i>) object associated.
<i>target</i>	The value of the target.
<i>tol</i>	The tolerance.
<i>track_all</i>	Compute the residual norm of all approximate eigenpairs.
<i>which</i>	The portion of the spectrum to be sought.

Methods Documentation

appendOptionsPrefix(*prefix=None*)

Append to the prefix used for searching for all PEP options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all PEP option requests.

Return type

None

See also

setOptionsPrefix, *getOptionsPrefix*, *PEPAppendOptionsPrefix*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:467](#) <[slepc4py/SLEPc/PEP.pyx#L467](#)>

cancelMonitor()

Clear all monitors for a *PEP* object.

Logically collective.

See also

PEPMonitorCancel

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1589](#) <[slepc4py/SLEPc/PEP.pyx#L1589](#)>

Return type

None

computeError(*i, etype=None*)

Compute the error associated with the *i*-th computed eigenpair.

Collective.

Compute the error (based on the residual norm) associated with the *i*-th computed eigenpair.

Parameters

- **i** (*int*) – Index of the solution to be considered.
- **etype** (*ErrorType* / *None*) – The error type to compute.

Returns

The error bound, computed in various ways from the residual norm $\|P(\lambda)x\|_2$ where λ is the eigenvalue and x is the eigenvector.

Return type
`float`

Notes

The index `i` should be a value between 0 and `nconv-1` (see `getConverged()`).

See also

`getErrorEstimate`, `PEPComputeError`

`:sources:` [Source code at slepc4py/SLEPc/PEP.pyx:1785](#) <slepc4py/SLEPc/PEP.pyx#L1785>

`create(comm=None)`

Create the PEP object.

Collective.

Parameters

`comm` (`Comm` / `None`) – MPI communicator. If not provided, it defaults to all processes.

Return type

Self

See also

`PEPCreate`

`:sources:` [Source code at slepc4py/SLEPc/PEP.pyx:350](#) <slepc4py/SLEPc/PEP.pyx#L350>

`destroy()`

Destroy the PEP object.

Collective.

See also

`PEPDestroy`

`:sources:` [Source code at slepc4py/SLEPc/PEP.pyx:324](#) <slepc4py/SLEPc/PEP.pyx#L324>

Return type

Self

`errorView(etype=None, viewer=None)`

Display the errors associated with the computed solution.

Collective.

Display the errors and the eigenvalues.

Parameters

- **`etype`** (`ErrorType` / `None`) – The error type to compute.
- **`viewer`** (`petsc4py.PETSc.Viewer` / `None`) – Visualization context; if not provided, the standard output is used.

Return type

None

Notes

By default, this function checks the error of all eigenpairs and prints the eigenvalues if all of them are below the requested tolerance. If the viewer has format ASCII_INFO_DETAIL then a table with eigenvalues and corresponding errors is printed.

See also

solve, *valuesView*, *vectorsView*, *PEPErrorView*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1823](#) <slepc4py/SLEPc/PEP.pyx#L1823>

getBV()

Get the basis vectors object associated to the eigensolver.

Not collective.

Returns

The basis vectors context.

Return type

BV

See also

setBV, *PEPGetBV*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1250](#) <slepc4py/SLEPc/PEP.pyx#L1250>

getBasis()

Get the type of polynomial basis used.

Not collective.

Returns

The basis that was previously set.

Return type

Basis

See also

setBasis, *PEPGetBasis*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:505](#) <slepc4py/SLEPc/PEP.pyx#L505>

getCISSExtraction()

Get the extraction technique used in the CISS solver.

Not collective.

Returns

The extraction technique.

Return type
CISSExtraction

See also

setCISSExtraction, *PEPCISSGetExtraction*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2729 <slepc4py/SLEPc/PEP.pyx#L2729>`

getCISSKSPs()

Get the array of linear solver objects associated with the CISS solver.

Collective.

Returns
The linear solver objects.

Return type
list of petsc4py.PETSc.KSP

Notes

The number of *petsc4py.PETSc.KSP* solvers is equal to the number of integration points divided by the number of partitions. This value is halved in the case of real matrices with a region centered at the real axis.

See also

setCISSSizes, *PEPCISSGetKSPs*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2926 <slepc4py/SLEPc/PEP.pyx#L2926>`

getCISSRefinement()

Get the values of various refinement parameters in the CISS solver.

Not collective.

Returns

- **inner** (*int*) – Number of iterative refinement iterations (inner loop).
- **blsize** (*int*) – Number of iterative refinement iterations (blocksize loop).

Return type
tuple[int, int]

See also

setCISSRefinement, *PEPCISSGetRefinement*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2904 <slepc4py/SLEPc/PEP.pyx#L2904>`

getCISSSizes()

Get the values of various size parameters in the CISS solver.

Not collective.

Returns

- **ip** ([int](#)) – Number of integration points.
- **bs** ([int](#)) – Block size.
- **ms** ([int](#)) – Moment size.
- **npart** ([int](#)) – Number of partitions when splitting the communicator.
- **bsmax** ([int](#)) – Maximum block size.
- **realmats** ([bool](#)) – True if A and B are real.

Return type

[tuple](#)[[int](#), [int](#), [int](#), [int](#), [int](#), [bool](#)]

See also

[setCISSSizes](#), [PEPCISSGetSizes](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2802 <slepc4py/SLEPc/PEP.pyx#L2802>](#)

getCISSThreshold()

Get the values of various threshold parameters in the CISS solver.

Not collective.

Returns

- **delta** ([float](#)) – Threshold for numerical rank.
- **spur** ([float](#)) – Spurious threshold (to discard spurious eigenpairs).

Return type

[tuple](#)[[float](#), [float](#)]

See also

[setCISSThreshold](#), [PEPCISSGetThreshold](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2859 <slepc4py/SLEPc/PEP.pyx#L2859>](#)

getConverged()

Get the number of converged eigenpairs.

Not collective.

Returns

nconv – Number of converged eigenpairs.

Return type

[int](#)

Notes

This function should be called after [solve\(\)](#) has finished.

The value `nconv` may be different from the number of requested solutions `nev`, but not larger than `ncv`, see [setDimensions\(\)](#).

See also

setDimensions, *solve*, *getEigenpair*, *PEPGetConverged*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1689](#) <[slepc4py/SLEPc/PEP.pyx#L1689](#)>

getConvergedReason()

Get the reason why the *solve()* iteration was stopped.

Not collective.

Returns

Negative value indicates diverged, positive value converged.

Return type

ConvergedReason

See also

setTolerances, *solve*, *PEPGetConvergedReason*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1670](#) <[slepc4py/SLEPc/PEP.pyx#L1670](#)>

getConvergenceTest()

Get the method used to compute the error estimate used in the convergence test.

Not collective.

Returns

The method used to compute the error estimate used in the convergence test.

Return type

Conv

See also

setConvergenceTest, *PEPGetConvergenceTest*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:820](#) <[slepc4py/SLEPc/PEP.pyx#L820](#)>

getDS()

Get the direct solver associated to the eigensolver.

Not collective.

Returns

The direct solver context.

Return type

DS

See also

setDS, *PEPGetDS*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1324](#) <[slepc4py/SLEPc/PEP.pyx#L1324](#)>

getDimensions()

Get the number of eigenvalues to compute and the dimension of the subspace.

Not collective.

Returns

- **nev** (*int*) – Number of eigenvalues to compute.
- **ncv** (*int*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int*) – Maximum dimension allowed for the projected problem.

Return type

`tuple[int, int, int]`

See also

[`setDimensions`](#), [`PEPGetDimensions`](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1038 <slepc4py/SLEPc/PEP.pyx#L1038>`

getEigenpair(*i*, *Vr*=None, *Vi*=None)

Get the *i*-th solution of the eigenproblem as computed by [`solve\(\)`](#).

Collective.

The solution consists of both the eigenvalue and the eigenvector.

Parameters

- **i** (*int*) – Index of the solution to be obtained.
- **Vr** (*Vec* | *None*) – Placeholder for the returned eigenvector (real part).
- **Vi** (*Vec* | *None*) – Placeholder for the returned eigenvector (imaginary part).

Returns

The computed eigenvalue.

Return type

`complex`

Notes

The index *i* should be a value between 0 and `nconv-1` (see [`getConverged\(\)`](#)). Eigenpairs are indexed according to the ordering criterion established with [`setWhichEigenpairs\(\)`](#).

The eigenvector is normalized to have unit norm.

See also

[`solve`](#), [`getConverged`](#), [`setWhichEigenpairs`](#), [`PEPGetEigenpair`](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1715 <slepc4py/SLEPc/PEP.pyx#L1715>`

getEigenvalueComparison()

Get the eigenvalue comparison function.

Not collective.

Returns

The eigenvalue comparison function.

Return type

PEPEigenvalueComparison

See also

setEigenvalueComparison

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1530 <slepc4py/SLEPc/PEP.pyx#L1530>](#)

getErrorEstimate(*i*)

Get the error estimate associated to the *i*-th computed eigenpair.

Not collective.

Parameters

i (*int*) – Index of the solution to be considered.

Returns

Error estimate.

Return type

float

Notes

This is the error estimate used internally by the eigensolver. The actual error bound can be computed with *computeError()*.

See also

computeError, *PEPGetErrorEstimate*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1756 <slepc4py/SLEPc/PEP.pyx#L1756>](#)

getExtract()

Get the extraction technique used by the *PEP* object.

Not collective.

Returns

The extraction strategy.

Return type

Extract

See also

setExtract, *PEPGetExtract*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:979 <slepc4py/SLEPc/PEP.pyx#L979>](#)

getInterval()

Get the computational interval for spectrum slicing.

Not collective.

Returns

- **inta** (`float`) – The left end of the interval.
- **intb** (`float`) – The right end of the interval.

Return type

`tuple[float, float]`

Notes

If the interval was not set by the user, then zeros are returned.

See also

`setInterval`, `PEPGetInterval`

:sources: ``Source code at slepc4py/SLEPc/PEP.pyx:764 <slepc4py/SLEPc/PEP.pyx#L764>``

getIterationNumber()

Get the current iteration number.

Not collective.

If the call to `solve()` is complete, then it returns the number of iterations carried out by the solution method.

Returns

Iteration number.

Return type

`int`

See also

`getConvergedReason`, `setTolerances`, `PEPGetIterationNumber`

:sources: ``Source code at slepc4py/SLEPc/PEP.pyx:1648 <slepc4py/SLEPc/PEP.pyx#L1648>``

getJDFix()

Get threshold for changing the target in the correction equation.

Not collective.

Returns

The threshold for changing the target.

Return type

`float`

See also

setJDFix, *PEPJGetFix*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2566 <slepc4py/SLEPc/PEP.pyx#L2566>](#)

getJDMinimalityIndex()

Get the maximum allowed value of the minimality index.

Not collective.

Returns

The maximum minimality index.

Return type

int

See also

setJDMinimalityIndex, *PEPJGetMinimalityIndex*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2653 <slepc4py/SLEPc/PEP.pyx#L2653>](#)

getJDProjection()

Get the type of projection to be used in the Jacobi-Davidson solver.

Not collective.

Returns

The type of projection.

Return type

JDProjection

See also

setJDProjection, *PEPJGetProjection*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2690 <slepc4py/SLEPc/PEP.pyx#L2690>](#)

getJDRestart()

Get the restart parameter used in the Jacobi-Davidson method.

Not collective.

Returns

The number of vectors to be kept at restart.

Return type

float

See also

setJDRestart, *PEPJGetRestart*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2523 <slepc4py/SLEPc/PEP.pyx#L2523>](#)

getJDReusePreconditioner()

Get the flag for reusing the preconditioner.

Not collective.

Returns

The reuse flag.

Return type

`bool`

See also

`setJDReusePreconditioner`, `PEPJDGetReusePreconditioner`

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2610 <slepc4py/SLEPc/PEP.pyx#L2610>](#)

getLinearEPS()

Get the eigensolver object associated to the polynomial eigenvalue solver.

Collective.

Returns

The linear eigensolver.

Return type

`EPS`

See also

`setLinearEPS`, `PEPLinearGetEPS`

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1912 <slepc4py/SLEPc/PEP.pyx#L1912>](#)

getLinearExplicitMatrix()

Get if the matrices for the linearization are built explicitly.

Not collective.

Returns

Boolean flag indicating if the matrices are built explicitly.

Return type

`bool`

See also

`getLinearExplicitMatrix`, `PEPLinearSetExplicitMatrix`

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1993 <slepc4py/SLEPc/PEP.pyx#L1993>](#)

getLinearLinearization()

Get the coeffs. defining the linearization of a quadratic eigenproblem.

Not collective.

Returns

- **alpha** (`float`) – First parameter of the linearization.
- **beta** (`float`) – Second parameter of the linearization.

Return type

`tuple[float, float]`

See also

`setLinearLinearization`, `PEPLinearGetLinearization`

:sources: ``Source code at slepc4py/SLEPc/PEP.pyx:1953 <slepc4py/SLEPc/PEP.pyx#L1953>``

getMonitor()

Get the list of monitor functions.

Not collective.

Returns

The list of monitor functions.

Return type

`PEPMonitorFunction`

See also

`setMonitor`

:sources: ``Source code at slepc4py/SLEPc/PEP.pyx:1572 <slepc4py/SLEPc/PEP.pyx#L1572>``

getOperators()

Get the matrices associated with the eigenvalue problem.

Collective.

Returns

The matrices associated with the eigensystem.

Return type

`list of petsc4py.PETSc.Mat`

See also

`setOperators`, `PEPGetOperators`

:sources: ``Source code at slepc4py/SLEPc/PEP.pyx:1361 <slepc4py/SLEPc/PEP.pyx#L1361>``

getOptionsPrefix()

Get the prefix used for searching for all PEP options in the database.

Not collective.

Returns

The prefix string set for this PEP object.

Return type

`str`

See also

setOptionsPrefix, *appendOptionsPrefix*, *PEPGetOptionsPrefix*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:417](#) <slepc4py/SLEPc/PEP.pyx#L417>

getProblemType()

Get the problem type from the PEP object.

Not collective.

Returns

The problem type that was previously set.

Return type

ProblemType

See also

setProblemType, *PEPGetProblemType*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:553](#) <slepc4py/SLEPc/PEP.pyx#L553>

getQArnoldiLocking()

Get the locking flag used in the Q-Arnoldi method.

Not collective.

Returns

The locking flag.

Return type

bool

See also

setQArnoldiLocking, *PEPQArnoldiGetLocking*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2084](#) <slepc4py/SLEPc/PEP.pyx#L2084>

getQArnoldiRestart()

Get the restart parameter used in the Q-Arnoldi method.

Not collective.

Returns

The number of vectors to be kept at restart.

Return type

float

See also

setQArnoldiRestart, *PEPQArnoldiGetRestart*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2040](#) <slepc4py/SLEPc/PEP.pyx#L2040>

getRG()

Get the region object associated to the eigensolver.

Not collective.

Returns

The region context.

Return type

RG

See also

setRG, *PEPGetRG*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1287 <slepc4py/SLEPc/PEP.pyx#L1287>`

getRefine()

Get the refinement strategy used by the PEP object.

Not collective.

Returns

- **ref** (*Refine*) – The refinement type.
- **npart** (*int*) – The number of partitions of the communicator.
- **tol** (*float*) – The convergence tolerance.
- **its** (*int*) – The maximum number of refinement iterations.
- **scheme** (*RefineScheme*) – Scheme for solving linear systems.

Return type

tuple[*Refine*, *int*, *float*, *int*, *RefineScheme*]

See also

setRefine, *PEPGetRefine*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:859 <slepc4py/SLEPc/PEP.pyx#L859>`

getRefineKSP()

Get the KSP object used by the eigensolver in the refinement phase.

Collective.

Returns

The linear solver object.

Return type

petsc4py.PETSc.KSP

See also

`setRefine`, `PEPRefineGetKSP`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:934 <slepc4py/SLEPc/PEP.pyx#L934>`

getST()

Get the spectral transformation object associated to the eigensolver.

Not collective.

Returns

The spectral transformation.

Return type

`ST`

See also

`setST`, `PEPGetST`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1114 <slepc4py/SLEPc/PEP.pyx#L1114>`

getSTOARCheckEigenvalueType()

Get the flag for the eigenvalue type check in spectrum slicing.

Not collective.

Returns

Whether the eigenvalue type is checked or not.

Return type

`bool`

See also

`setSTOARCheckEigenvalueType`, `PEPSTOARGetCheckEigenvalueType`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2476 <slepc4py/SLEPc/PEP.pyx#L2476>`

getSTOARDetectZeros()

Get the flag that enforces zero detection in spectrum slicing.

Not collective.

Returns

The zero detection flag.

Return type

`bool`

See also

`setSTOARDetectZeros`, `PEPSTOARGetDetectZeros`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2316 <slepc4py/SLEPc/PEP.pyx#L2316>`

getSTOARDimensions()

Get the dimensions used for each subsolve step.

Not collective.

Returns

- **nev** ([int](#)) – Number of eigenvalues to compute.
- **ncv** ([int](#)) – Maximum dimension of the subspace to be used by the solver.
- **mpd** ([int](#)) – Maximum dimension allowed for the projected problem.

Return type

[tuple](#)[[int](#), [int](#), [int](#)]

See also

[setSTOARDimensions](#), [PEPSTOARGetDimensions](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2375 <slepc4py/SLEPc/PEP.pyx#L2375>`

getSTOARInertias()

Get the values of the shifts and their corresponding inertias.

Not collective.

Get the values of the shifts and their corresponding inertias in case of doing spectrum slicing for a computational interval.

Returns

- **shifts** ([ArrayReal](#)) – The values of the shifts used internally in the solver.
- **inertias** ([ArrayInt](#)) – The values of the inertia in each shift.

Return type

[tuple](#)[[ArrayReal](#), [ArrayInt](#)]

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with [setInterval\(\)](#) and *SINVERT* is set.

If called after [solve\(\)](#), all shifts used internally by the solver are returned (including both endpoints and any intermediate ones). If called before [solve\(\)](#) and after [setUp\(\)](#) then only the information of the endpoints of subintervals is available.

See also

[setInterval](#), [PEPSTOARGetInertias](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2400 <slepc4py/SLEPc/PEP.pyx#L2400>`

getSTOARLinearization()

Get the coefficients that define the linearization of a quadratic eigenproblem.

Not collective.

Returns

- **alpha** (`float`) – First parameter of the linearization.
- **beta** (`float`) – Second parameter of the linearization.

Return type

`tuple[float, float]`

See also

`setSTOARLinearization`, `PEPSTOARGetLinearization`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2217 <slepc4py/SLEPc/PEP.pyx#L2217>`

getSTOARLocking()

Get the locking flag used in the STOAR method.

Not collective.

Returns

The locking flag.

Return type

`bool`

See also

`setSTOARLocking`, `PEPSTOARGetLocking`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2264 <slepc4py/SLEPc/PEP.pyx#L2264>`

getScale(Dl=None, Dr=None)

Get the strategy used for scaling the polynomial eigenproblem.

Not collective.

Parameters

- **Dl** (`petsc4py.PETSc.Vec` / `None`) – Placeholder for the returned left diagonal matrix.
- **Dr** (`petsc4py.PETSc.Vec` / `None`) – Placeholder for the returned right diagonal matrix.

Returns

- **scale** (`Scale`) – The scaling strategy.
- **alpha** (`float`) – The scaling factor.
- **its** (`int`) – The number of iterations of diagonal scaling.
- **lbda** (`float`) – Approximation of the wanted eigenvalues (modulus).

Return type

`tuple[Scale, float, int, float]`

See also

`setScale`, `PEPGetScale`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1151 <slepc4py/SLEPc/PEP.pyx#L1151>`

getStoppingTest()

Get the stopping test function.

Not collective.

Returns

The stopping test function.

Return type

PEPStoppingFunction

See also

setStoppingTest

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1483 <slepc4py/SLEPc/PEP.pyx#L1483>](#)

getTOARLocking()

Get the locking flag used in the TOAR method.

Not collective.

Returns

The locking flag.

Return type

bool

See also

setTOARLocking, *PEPTOARGetLocking*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2175 <slepc4py/SLEPc/PEP.pyx#L2175>](#)

getTOARRestart()

Get the restart parameter used in the TOAR method.

Not collective.

Returns

The number of vectors to be kept at restart.

Return type

float

See also

setTOARRestart, *PEPTOARGetRestart*

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2131 <slepc4py/SLEPc/PEP.pyx#L2131>](#)

getTarget()

Get the value of the target.

Not collective.

Returns

The value of the target.

Return type

Scalar

Notes

If the target was not set by the user, then zero is returned.

See also

setTarget, *PEPGetTarget*

`:sources:`Source code at slepc4py/SLEPc/PEP.pyx:658 <slepc4py/SLEPc/PEP.pyx#L658>``

getTolerances()

Get the tolerance and maximum iteration count.

Not collective.

Get the tolerance and maximum iteration count used by the default PEP convergence tests.

Returns

- **tol** (*float*) – The convergence tolerance.
- **max_it** (*int*) – The maximum number of iterations.

Return type

tuple[*float*, *int*]

See also

setTolerances, *PEPGetTolerances*

`:sources:`Source code at slepc4py/SLEPc/PEP.pyx:708 <slepc4py/SLEPc/PEP.pyx#L708>``

getTrackAll()

Get the flag indicating whether all residual norms must be computed.

Not collective.

Returns

Whether the solver computes all residuals or not.

Return type

bool

See also

setTrackAll, *PEPGetTrackAll*

`:sources:`Source code at slepc4py/SLEPc/PEP.pyx:998 <slepc4py/SLEPc/PEP.pyx#L998>``

getType()

Get the PEP type of this object.

Not collective.

Returns

The solver currently being used.

Return type

`str`

See also

`setType`, `PEPGetType`

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:398](#) <[slepc4py/SLEPc/PEP.pyx#L398](#)>

getWhichEigenpairs()

Get which portion of the spectrum is to be sought.

Not collective.

Returns

The portion of the spectrum to be sought by the solver.

Return type

`Which`

See also

`setWhichEigenpairs`, `PEPGetWhichEigenpairs`

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:602](#) <[slepc4py/SLEPc/PEP.pyx#L602](#)>

reset()

Reset the PEP object.

Collective.

See also

`PEPReset`

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:338](#) <[slepc4py/SLEPc/PEP.pyx#L338](#)>

Return type

`None`

setBV(bv)

Set a basis vectors object associated to the eigensolver.

Collective.

Parameters

bv (`BV`) – The basis vectors context.

Return type

`None`

See also

[getBV](#), [PEPSetBV](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1270](#) <[slepc4py/SLEPc/PEP.pyx#L1270](#)>

setBasis(*basis*)

Set the type of polynomial basis used.

Logically collective.

Set the type of polynomial basis used to describe the polynomial eigenvalue problem.

Parameters

basis ([Basis](#)) – The basis to be set.

Return type

[None](#)

Notes

By default, the coefficient matrices passed via [setOperators\(\)](#) are expressed in the monomial basis, i.e. $P(\lambda) = A_0 + \lambda A_1 + \lambda^2 A_2 + \dots + \lambda^d A_d$. Other polynomial bases may have better numerical behavior, but the user must then pass the coefficient matrices accordingly.

See also

[getBasis](#), [setOperators](#), [PEPSetBasis](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:524](#) <[slepc4py/SLEPc/PEP.pyx#L524](#)>

setCISSExtraction(*extraction*)

Set the extraction technique used in the CISS solver.

Logically collective.

Parameters

extraction ([CISSExtraction](#)) – The extraction technique.

Return type

[None](#)

See also

[getCISSExtraction](#), [PEPCISSSetExtraction](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2711](#) <[slepc4py/SLEPc/PEP.pyx#L2711](#)>

setCISSRefinement(*inner=None*, *blsize=None*)

Set the values of various refinement parameters in the CISS solver.

Logically collective.

Parameters

- **inner** ([int](#) / [None](#)) – Number of iterative refinement iterations (inner loop).
- **blsize** ([int](#) / [None](#)) – Number of iterative refinement iterations (blocksize loop).

Return type

None

See also[getCISSRefinement](#), [PEPCISSSetRefinement](#)**:sources:** `Source code at slepc4py/SLEPc/PEP.pyx:2881 <slepc4py/SLEPc/PEP.pyx#L2881>`**setCISSSizes**(*ip=None, bs=None, ms=None, npart=None, bsmax=None, realmats=False*)

Set the values of various size parameters in the CISS solver.

Logically collective.

Parameters

- **ip** (*int* / *None*) – Number of integration points.
- **bs** (*int* / *None*) – Block size.
- **ms** (*int* / *None*) – Moment size.
- **npart** (*int* / *None*) – Number of partitions when splitting the communicator.
- **bsmax** (*int* / *None*) – Maximum block size.
- **realmats** (*bool*) – True if A and B are real.

Return type

None

Notes

The default number of partitions is 1. This means the internal `petsc4py.PETSc.KSP` object is shared among all processes of the `PEP` communicator. Otherwise, the communicator is split into `npart` communicators, so that `npart` `petsc4py.PETSc.KSP` solves proceed simultaneously.

See also[getCISSSizes](#), [setCISSThreshold](#), [setCISSRefinement](#), [PEPCISSSetSizes](#)**:sources:** `Source code at slepc4py/SLEPc/PEP.pyx:2748 <slepc4py/SLEPc/PEP.pyx#L2748>`**setCISSThreshold**(*delta=None, spur=None*)

Set the values of various threshold parameters in the CISS solver.

Logically collective.

Parameters

- **delta** (*float* / *None*) – Threshold for numerical rank.
- **spur** (*float* / *None*) – Spurious threshold (to discard spurious eigenpairs).

Return type

None

See also

[`getCISSThreshold`](#), [`PEPCISSetThreshold`](#)

:sources: [`Source code at slepc4py/SLEPc/PEP.pyx:2836 <slepc4py/SLEPc/PEP.pyx#L2836>`](#)

setConvergenceTest(*conv*)

Set how to compute the error estimate used in the convergence test.

Logically collective.

Parameters

conv (*Conv*) – The method used to compute the error estimate used in the convergence test.

Return type

None

See also

[`getConvergenceTest`](#), [`PEPSetConvergenceTest`](#)

:sources: [`Source code at slepc4py/SLEPc/PEP.pyx:840 <slepc4py/SLEPc/PEP.pyx#L840>`](#)

setDS(*ds*)

Set a direct solver object associated to the eigensolver.

Collective.

Parameters

ds (*DS*) – The direct solver context.

Return type

None

See also

[`getDS`](#), [`PEPSetDS`](#)

:sources: [`Source code at slepc4py/SLEPc/PEP.pyx:1344 <slepc4py/SLEPc/PEP.pyx#L1344>`](#)

setDimensions(*nev=None, ncv=None, mpd=None*)

Set the number of eigenvalues to compute and the dimension of the subspace.

Logically collective.

Parameters

- **nev** (*int* / *None*) – Number of eigenvalues to compute.
- **ncv** (*int* / *None*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int* / *None*) – Maximum dimension allowed for the projected problem.

Return type

None

Notes

Use *DETERMINE* for `ncv` and `mpd` to assign a reasonably good value, which is dependent on the solution method.

The parameters `ncv` and `mpd` are intimately related, so that the user is advised to set one of them at most. Normal usage is the following:

- In cases where `nev` is small, the user sets `ncv` (a reasonable default is $2 * nev$).
- In cases where `nev` is large, the user sets `mpd`.

The value of `ncv` should always be between `nev` and $(nev + mpd)$, typically $ncv = nev + mpd$. If `nev` is not too large, $mpd = nev$ is a reasonable choice, otherwise a smaller value should be used.

See also

[*getDimensions*](#), [*PEPSetDimensions*](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1063 <slepc4py/SLEPc/PEP.pyx#L1063>`

setEigenvalueComparison(*comparison*, *args=None*, *kargs=None*)

Set an eigenvalue comparison function.

Logically collective.

Notes

This eigenvalue comparison function is used when *setWhichEigenpairs()* is set to *PEP.Which.USER*.

See also

[*getEigenvalueComparison*](#), [*PEPSetEigenvalueComparison*](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1500 <slepc4py/SLEPc/PEP.pyx#L1500>`

Parameters

- **comparison** ([*PEPEigenvalueComparison*](#) / *None*)
- **args** (*tuple*[*Any*, ...] / *None*)
- **kargs** (*dict*[*str*, *Any*] / *None*)

Return type

None

setExtract(*extract*)

Set the extraction strategy to be used.

Logically collective.

Parameters

extract ([*Extract*](#)) – The extraction strategy.

Return type

None

Notes

This is relevant for solvers based on linearization. Once the solver has converged, the polynomial eigenvectors can be extracted from the eigenvectors of the linearized problem in different ways.

See also

`getExtract`, `PEPSetExtract`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:954 <slepc4py/SLEPc/PEP.pyx#L954>`

`setFromOptions()`

Set PEP options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before `setUp()` if the user is to be allowed to set the solver type.

See also

`setOptionsPrefix`, `PEPSetFromOptions`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:486 <slepc4py/SLEPc/PEP.pyx#L486>`

Return type

`None`

`setInitialSpace(space)`

Set the initial space from which the eigensolver starts to iterate.

Collective.

Parameters

space (`Vec` / `list[Vec]`) – The initial space.

Return type

`None`

Notes

Some solvers start to iterate on a single vector (initial vector). In that case, only the first vector is taken into account and the other vectors are ignored.

These vectors do not persist from one `solve()` call to the other, so the initial space should be set every time.

The vectors do not need to be mutually orthonormal, since they are explicitly orthonormalized internally.

Common usage of this function is when the user can provide a rough approximation of the wanted eigenspace. Then, convergence may be faster.

See also

`setUp`, `PEPSetInitialSpace`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1420 <slepc4py/SLEPc/PEP.pyx#L1420>`

setInterval(*inta*, *intb*)

Set the computational interval for spectrum slicing.

Logically collective.

Parameters

- **inta** (*float*) – The left end of the interval.
- **intb** (*float*) – The right end of the interval.

Return type

None

Notes

Spectrum slicing is a technique employed for computing all eigenvalues of symmetric quadratic eigenproblems in a given interval. This function provides the interval to be considered. It must be used in combination with `PEP.Which.ALL`, see `setWhichEigenpairs()`. Note that in polynomial eigenproblems spectrum slicing is implemented in `STOAR` only.

See also

`getInterval`, `PEPSetInterval`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:790 <slepc4py/SLEPc/PEP.pyx#L790>`

setJDFix(*fix*)

Set the threshold for changing the target in the correction equation.

Logically collective.

Parameters

- **fix** (*float*) – Threshold for changing the target.

Return type

None

Notes

The target in the correction equation is fixed at the first iterations. When the norm of the residual vector is lower than the fix value, the target is set to the corresponding eigenvalue.

See also

`getJDFix`, `PEPJDSetsFix`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2542 <slepc4py/SLEPc/PEP.pyx#L2542>`

setJDMinimalityIndex(*flag*)

Set the maximum allowed value for the minimality index.

Logically collective.

Parameters

flag (*int*) – The maximum minimality index.

Return type

None

Notes

The default value is equal to the degree of the polynomial. A smaller value can be used if the wanted eigenvectors are known to be linearly independent.

See also

[*getJDMinimalityIndex*](#), [*PEPJDSetsMinimalityIndex*](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2629 <slepc4py/SLEPc/PEP.pyx#L2629>](#)

setJDProjection(*proj*)

Set the type of projection to be used in the Jacobi-Davidson solver.

Logically collective.

Parameters

proj (*JDProjection*) – The type of projection.

Return type

None

See also

[*getJDProjection*](#), [*PEPJDSetsProjection*](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2672 <slepc4py/SLEPc/PEP.pyx#L2672>](#)

setJDRestart(*keep*)

Set the restart parameter for the Jacobi-Davidson method.

Logically collective.

Set the restart parameter for the Jacobi-Davidson method, in particular the proportion of basis vectors that must be kept after restart.

Parameters

keep (*float*) – The number of vectors to be kept at restart.

Return type

None

Notes

Allowed values are in the range [0.1,0.9]. The default is 0.5.

See also

[`getJDRestart`](#), [`PEPJDSetsRestart`](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2497 <slepc4py/SLEPc/PEP.pyx#L2497>](#)

setJDReusePreconditioner(*flag*)

Set a flag indicating whether the preconditioner must be reused or not.

Logically collective.

Parameters

flag (*bool*) – The reuse flag.

Return type

None

Notes

The default value is `False`. If set to `True`, the preconditioner is built only at the beginning, using the target value. Otherwise, it may be rebuilt (depending on the `fix` parameter) at each iteration from the Ritz value.

See also

[`setJDFix`](#), [`getJDReusePreconditioner`](#), [`PEPJDSetsReusePreconditioner`](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2585 <slepc4py/SLEPc/PEP.pyx#L2585>](#)

setLinearEPS(*eps*)

Set an eigensolver object associated to the polynomial eigenvalue solver.

Collective.

Parameters

eps (*EPS*) – The linear eigensolver.

Return type

None

See also

[`getLinearEPS`](#), [`PEPLinearSetEPS`](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1895 <slepc4py/SLEPc/PEP.pyx#L1895>](#)

setLinearExplicitMatrix(*flag*)

Set flag to explicitly build the matrices for the linearization.

Logically collective.

Parameters

flag (*bool*) – Boolean flag indicating if the matrices are built explicitly.

Return type

None

See also[*getLinearExplicitMatrix*](#), [*PEPLinearSetExplicitMatrix*](#)**:sources:**``Source code at slepc4py/SLEPc/PEP.pyx:1975 <slepc4py/SLEPc/PEP.pyx#L1975>``**setLinearLinearization**(*alpha=1.0, beta=0.0*)

Set the coefficients that define the linearization of a quadratic eigenproblem.

Logically collective.

Parameters

- **alpha** (*float*) – First parameter of the linearization.
- **beta** (*float*) – Second parameter of the linearization.

Return type

None

See also[*getLinearLinearization*](#), [*PEPLinearSetLinearization*](#)**:sources:**``Source code at slepc4py/SLEPc/PEP.pyx:1932 <slepc4py/SLEPc/PEP.pyx#L1932>``**setMonitor**(*monitor, args=None, kargs=None*)

Append a monitor function to the list of monitors.

Logically collective.

See also[*getMonitor*](#), [*cancelMonitor*](#), [*PEPMonitorSet*](#)**:sources:**``Source code at slepc4py/SLEPc/PEP.pyx:1547 <slepc4py/SLEPc/PEP.pyx#L1547>``**Parameters**

- **monitor** ([*PEPMonitorFunction*](#) | *None*)
- **args** ([*tuple*](#)[*Any*, ...] | *None*)
- **kargs** ([*dict*](#)[*str*, *Any*] | *None*)

Return type

None

setOperators(*operators*)

Set the matrices associated with the eigenvalue problem.

Collective.

Parameters

- **operators** ([*list*](#)[[*Mat*](#)]) – The matrices associated with the eigensystem.

Return type

None

Notes

The polynomial eigenproblem is defined as $P(\lambda)x = 0$, where λ is the eigenvalue, x is the eigenvector, and P is defined as $P(\lambda) = A_0 + \lambda A_1 + \dots + \lambda^d A_d$, with $d = \text{pmat}-1$ (the degree of P). For non-monomial bases, this expression is different.

See also`getOperators`, `PEPSetOperators`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1387 <slepc4py/SLEPc/PEP.pyx#L1387>`

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all PEP options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all PEP option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

For example, to distinguish between the runtime options for two different PEP contexts, one could call:

```
P1.setOptionsPrefix("pep1_")
P2.setOptionsPrefix("pep2_")
```

See also`appendOptionsPrefix`, `getOptionsPrefix`, `PEPGetOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:436 <slepc4py/SLEPc/PEP.pyx#L436>`

setProblemType(*problem_type*)

Set the type of the polynomial eigenvalue problem.

Logically collective.

Parameters

problem_type (*ProblemType*) – The problem type to be set.

Return type

None

Notes

This function is used to instruct SLEPc to exploit certain structure in the polynomial eigenproblem. By default, no particular structure is assumed.

If the problem matrices are Hermitian (symmetric in the real case) or Hermitian/skew-Hermitian then the solver can exploit this fact to perform less operations or provide better stability. Hyperbolic problems are a particular case of Hermitian problems, some solvers may treat them simply as Hermitian.

See also

[setOperators](#), [setType](#), [getProblemType](#), [PEPSetProblemType](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:572 <slepc4py/SLEPc/PEP.pyx#L572>](#)

setQArnoldiLocking(lock)

Toggle between locking and non-locking variants of the Q-Arnoldi method.

Logically collective.

Parameters

lock (*bool*) – True if the locking variant must be selected.

Return type

None

Notes

The default is to lock converged eigenpairs when the method restarts. This behavior can be changed so that all directions are kept in the working subspace even if already converged to working accuracy (the non-locking variant).

See also

[getQArnoldiLocking](#), [PEPQArnoldiSetLocking](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2059 <slepc4py/SLEPc/PEP.pyx#L2059>](#)

setQArnoldiRestart(keep)

Set the restart parameter for the Q-Arnoldi method.

Logically collective.

Set the restart parameter for the Q-Arnoldi method, in particular the proportion of basis vectors that must be kept after restart.

Parameters

keep (*float*) – The number of vectors to be kept at restart.

Return type

None

Notes

Allowed values are in the range [0.1,0.9]. The default is 0.5.

See also

[`getQArnoldiRestart`](#), [`PEPQArnoldiSetRestart`](#)

:sources: [`Source code at slepc4py/SLEPc/PEP.pyx:2014 <slepc4py/SLEPc/PEP.pyx#L2014>`](#)

setRG(*rg*)

Set a region object associated to the eigensolver.

Collective.

Parameters

rg (*RG*) – The region context.

Return type

None

See also

[`getRG`](#), [`PEPSetRG`](#)

:sources: [`Source code at slepc4py/SLEPc/PEP.pyx:1307 <slepc4py/SLEPc/PEP.pyx#L1307>`](#)

setRefine(*ref*, *npart*=*None*, *tol*=*None*, *its*=*None*, *scheme*=*None*)

Set the refinement strategy used by the PEP object.

Logically collective.

Set the refinement strategy used by the PEP object, and the associated parameters.

Parameters

- **ref** (*Refine*) – The refinement type.
- **npart** (*int* / *None*) – The number of partitions of the communicator.
- **tol** (*float* / *None*) – The convergence tolerance.
- **its** (*int* / *None*) – The maximum number of refinement iterations.
- **scheme** (*RefineScheme* / *None*) – Scheme for solving linear systems.

Return type

None

See also

[`getRefine`](#), [`PEPSetRefine`](#)

:sources: [`Source code at slepc4py/SLEPc/PEP.pyx:890 <slepc4py/SLEPc/PEP.pyx#L890>`](#)

setST(*st*)

Set a spectral transformation object associated to the eigensolver.

Collective.

Parameters

st (*ST*) – The spectral transformation.

Return type

None

See also`getST`, `PEPSetST`**:sources:** `Source code at slepc4py/SLEPc/PEP.pyx:1134 <slepc4py/SLEPc/PEP.pyx#L1134>`**setSTOARCheckEigenvalueType**(*flag*)

Set flag to check if all eigenvalues have the same definite type.

Logically collective.

Set a flag to check that all the eigenvalues obtained throughout the spectrum slicing computation have the same definite type.

Parameters**flag** (*bool*) – Whether the eigenvalue type is checked or not.**Return type**

None

NotesThis option is relevant only for spectrum slicing computations, but is ignored in `slepc4py.SLEPc.PEP.ProblemType.HYPERBOLIC` problems.

This flag is turned on by default, to guarantee that the computed eigenvalues have the same type (otherwise the computed solution might be wrong). But since the check is computationally quite expensive, the check may be turned off if the user knows for sure that all eigenvalues in the requested interval have the same type.

See also`getSTOARCheckEigenvalueType`, `PEPSTOARSetCheckEigenvalueType`**:sources:** `Source code at slepc4py/SLEPc/PEP.pyx:2444 <slepc4py/SLEPc/PEP.pyx#L2444>`**setSTOARDetectZeros**(*detect*)

Set flag to enforce detection of zeros during the factorizations.

Logically collective.

Set a flag to enforce detection of zeros during the factorizations throughout the spectrum slicing computation.

Parameters**detect** (*bool*) – True if zeros must be checked for.**Return type**

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with `setInterval()` and `SINVERT` is set.

A zero in the factorization indicates that a shift coincides with an eigenvalue.

This flag is turned off by default, and may be necessary in some cases. This feature currently requires an external package for factorizations with support for zero detection, e.g. MUMPS.

See also

`setInterval`, `getSTOARDetectZeros`, `PEPSTOARSetDetectZeros`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2283 <slepc4py/SLEPc/PEP.pyx#L2283>`

setSTOARDimensions(*nev=None, ncv=None, mpd=None*)

Set the dimensions used for each subsolve step.

Logically collective.

Parameters

- **nev** (*int* / *None*) – Number of eigenvalues to compute.
- **ncv** (*int* / *None*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int* / *None*) – Maximum dimension allowed for the projected problem.

Return type

None

Notes

This call makes sense only for spectrum slicing runs, that is, when an interval has been given with `setInterval()` and `SINVERT` is set.

The meaning of the parameters is the same as in `setDimensions()`, but the ones here apply to every subsolve done by the child `PEP` object.

See also

`setInterval`, `setDimensions`, `getSTOARDimensions`, `PEPSTOARSetDimensions`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2335 <slepc4py/SLEPc/PEP.pyx#L2335>`

setSTOARLinearization(*alpha=1.0, beta=0.0*)

Set the coefficients that define the linearization of a quadratic eigenproblem.

Logically collective.

Parameters

- **alpha** (*float*) – First parameter of the linearization.
- **beta** (*float*) – Second parameter of the linearization.

Return type

None

See also

[getSTOARLinearization](#), [PEPSTOARSetLinearization](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2196 <slepc4py/SLEPc/PEP.pyx#L2196>`

setSTOARLocking(*lock*)

Toggle between locking and non-locking variants of the STOAR method.

Logically collective.

Parameters

lock (*bool*) – True if the locking variant must be selected.

Return type

None

Notes

The default is to lock converged eigenpairs when the method restarts. This behavior can be changed so that all directions are kept in the working subspace even if already converged to working accuracy (the non-locking variant).

See also

[getSTOARLocking](#), [PEPSTOARSetLocking](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2239 <slepc4py/SLEPc/PEP.pyx#L2239>`

setScale(*scale*, *alpha*=*None*, *Dl*=*None*, *Dr*=*None*, *its*=*None*, *lbda*=*None*)

Set the scaling strategy to be used.

Collective.

Parameters

- **scale** (*Scale*) – The scaling strategy.
- **alpha** (*float* | *None*) – The scaling factor.
- **Dl** (*petsc4py.PETSc.Vec* | *None*) – The left diagonal matrix.
- **Dr** (*petsc4py.PETSc.Vec* | *None*) – The right diagonal matrix.
- **its** (*int* | *None*) – The number of iterations of diagonal scaling.
- **lbda** (*float* | *None*) – Approximation of the wanted eigenvalues (modulus).

Return type

None

See also

[getScale](#), [PEPSetScale](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1204 <slepc4py/SLEPc/PEP.pyx#L1204>`

setStoppingTest(*stopping*, *args*=None, *kargs*=None)

Set a function to decide when to stop the outer iteration of the eigensolver.

Logically collective.

See also

[getStoppingTest](#), [PEPSetStoppingTestFunction](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:1459](#) <[slepc4py/SLEPc/PEP.pyx#L1459](#)>

Parameters

- **stopping** ([PEPStoppingFunction](#) | None)
- **args** ([tuple](#)[Any, ...] | None)
- **kargs** ([dict](#)[str, Any] | None)

Return type

None

setTOARLocking(*lock*)

Toggle between locking and non-locking variants of the TOAR method.

Logically collective.

Parameters

lock ([bool](#)) – True if the locking variant must be selected.

Return type

None

Notes

The default is to lock converged eigenpairs when the method restarts. This behavior can be changed so that all directions are kept in the working subspace even if already converged to working accuracy (the non-locking variant).

See also

[getTOARLocking](#), [PEPTOARSetLocking](#)

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2150](#) <[slepc4py/SLEPc/PEP.pyx#L2150](#)>

setTOARRestart(*keep*)

Set the restart parameter for the TOAR method.

Logically collective.

Set the restart parameter for the TOAR method, in particular the proportion of basis vectors that must be kept after restart.

Parameters

keep ([float](#)) – The number of vectors to be kept at restart.

Return type

None

Notes

Allowed values are in the range [0.1,0.9]. The default is 0.5.

See also

`getTOARRestart`, `PEPTOARSetRestart`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:2105 <slepc4py/SLEPc/PEP.pyx#L2105>`

setTarget(*target*)

Set the value of the target.

Logically collective.

Parameters

target (*Scalar*) – The value of the target.

Return type

None

Notes

The target is a scalar value used to determine the portion of the spectrum of interest. It is used in combination with `setWhichEigenpairs()`.

When PETSc is built with real scalars, it is not possible to specify a complex target.

See also

`getTarget`, `setWhichEigenpairs`, `PEPSetTarget`

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:681 <slepc4py/SLEPc/PEP.pyx#L681>`

setTolerances(*tol=None*, *max_it=None*)

Set the tolerance and maximum iteration count.

Logically collective.

Set the tolerance and maximum iteration count used by the default PEP convergence tests.

Parameters

- **tol** (*float* / *None*) – The convergence tolerance.
- **max_it** (*int* / *None*) – The maximum number of iterations

Return type

None

Notes

Use `DETERMINE` for `max_it` to assign a reasonably good value, which is dependent on the solution method.

See also

[`getTolerances`](#), [`PEPSetTolerances`](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:733 <slepc4py/SLEPc/PEP.pyx#L733>`

setTrackAll(*trackall*)

Set flag to compute the residual of all approximate eigenpairs.

Logically collective.

Set if the solver must compute the residual of all approximate eigenpairs or not.

Parameters

trackall (*bool*) – Whether to compute all residuals or not.

Return type

`None`

See also

[`getTrackAll`](#), [`PEPSetTrackAll`](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1017 <slepc4py/SLEPc/PEP.pyx#L1017>`

setType(*pep_type*)

Set the particular solver to be used in the PEP object.

Logically collective.

Parameters

pep_type (*Type* / *str*) – The solver to be used.

Return type

`None`

Notes

The default is `TOAR`. Normally, it is best to use `setFromOptions()` and then set the PEP type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also

[`getType`](#), [`PEPSetType`](#)

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:371 <slepc4py/SLEPc/PEP.pyx#L371>`

setUp()

Set up all the internal data structures.

Collective.

Notes

Sets up all the internal data structures necessary for the execution of the eigensolver. This includes the setup of the internal *ST* object.

This function need not be called explicitly in most cases, since *solve()* calls it. It can be useful when one wants to measure the set-up time separately from the solve time.

See also

solve, *PEPSetUp*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:1604 <slepc4py/SLEPc/PEP.pyx#L1604>`

Return type

None

setWhichEigenpairs(*which*)

Set which portion of the spectrum is to be sought.

Logically collective.

Parameters

which (*Which*) – The portion of the spectrum to be sought by the solver.

Return type

None

Notes

Not all eigensolvers implemented in PEP account for all the possible values. Also, some values make sense only for certain types of problems. If SLEPc is compiled for real numbers *PEP.Which.LARGEST_IMAGINARY* and *PEP.Which.SMALLEST_IMAGINARY* use the absolute value of the imaginary part for eigenvalue selection.

The target is a scalar value provided with *setTarget()*.

The criterion *PEP.Which.TARGET_IMAGINARY* is available only in case PETSc and SLEPc have been built with complex scalars.

PEP.Which.ALL is intended for use in combination with an interval (see *setInterval()*), when all eigenvalues within the interval are requested, or in the context of the *PEP.Type.CISS* solver for computing all eigenvalues in a region.

See also

getWhichEigenpairs, *setTarget*, *setInterval*, *PEPSetWhichEigenpairs*

:sources: `Source code at slepc4py/SLEPc/PEP.pyx:621 <slepc4py/SLEPc/PEP.pyx#L621>`

solve()

Solve the polynomial eigenproblem.

Collective.

Notes

The problem matrices are specified with `setOperators()`.

`solve()` will return without generating an error regardless of whether all requested solutions were computed or not. Call `getConverged()` to get the actual number of computed solutions, and `getConvergedReason()` to determine if the solver converged or failed and why.

See also

`setUp`, `setOperators`, `getConverged`, `getConvergedReason`, `PEPSolve`

`:sources:``Source code at slepc4py/SLEPc/PEP.pyx:1626 <slepc4py/SLEPc/PEP.pyx#L1626>`

Return type

`None`

`valuesView`(*viewer=None*)

Display the computed eigenvalues in a viewer.

Collective.

Parameters

`viewer` (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

`None`

See also

`solve`, `vectorsView`, `errorView`, `PEPValuesView`

`:sources:``Source code at slepc4py/SLEPc/PEP.pyx:1855 <slepc4py/SLEPc/PEP.pyx#L1855>`

`vectorsView`(*viewer=None*)

Output computed eigenvectors to a viewer.

Collective.

Parameters

`viewer` (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

`None`

See also

`solve`, `valuesView`, `errorView`, `PEPVectorsView`

`:sources:``Source code at slepc4py/SLEPc/PEP.pyx:1874 <slepc4py/SLEPc/PEP.pyx#L1874>`

`view`(*viewer=None*)

Print the PEP data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

PEPView

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:305 <slepc4py/SLEPc/PEP.pyx#L305>](#)

Attributes Documentation

bv

The basis vectors (*BV*) object associated.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:3011 <slepc4py/SLEPc/PEP.pyx#L3011>](#)

ds

The direct solver (*DS*) object associated.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:3025 <slepc4py/SLEPc/PEP.pyx#L3025>](#)

extract

The type of extraction technique to be employed.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2976 <slepc4py/SLEPc/PEP.pyx#L2976>](#)

max_it

The maximum iteration count.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2990 <slepc4py/SLEPc/PEP.pyx#L2990>](#)

problem_type

The type of the eigenvalue problem.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2955 <slepc4py/SLEPc/PEP.pyx#L2955>](#)

rg

The region (*RG*) object associated.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:3018 <slepc4py/SLEPc/PEP.pyx#L3018>](#)

st

The spectral transformation (*ST*) object associated.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:3004 <slepc4py/SLEPc/PEP.pyx#L3004>](#)

target

The value of the target.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2969 <slepc4py/SLEPc/PEP.pyx#L2969>](#)

tol

The tolerance.

:sources: [Source code at slepc4py/SLEPc/PEP.pyx:2983 <slepc4py/SLEPc/PEP.pyx#L2983>](#)


```
track_all
    Compute the residual norm of all approximate eigenpairs.

:sources:`Source code at slepc4py/SLEPc/PEP.pyx:2997 <slepc4py/SLEPc/PEP.pyx#L2997>`

which
    The portion of the spectrum to be sought.

:sources:`Source code at slepc4py/SLEPc/PEP.pyx:2962 <slepc4py/SLEPc/PEP.pyx#L2962>`

__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.RG

```
class slepc4py.SLEPc.RG
    Bases: Object
    Region.

    The RG package provides a way to define a region of the complex plane. This is used in various eigensolvers to
    specify where the wanted eigenvalues are located.
```

Enumerations

<i>QuadRule</i>	RG quadrature rule for contour integral methods.
<i>Type</i>	RG type.

slepc4py.SLEPc.RG.QuadRule

```
class slepc4py.SLEPc.RG.QuadRule
    Bases: object
    RG quadrature rule for contour integral methods.

    • TRAPEZOIDAL: Trapezoidal rule.
    • CHEBYSHEV: Chebyshev points.
```

See also	
<i>RGQuadRule</i>	

Attributes Summary

<i>CHEBYSHEV</i>	Constant CHEBYSHEV of type <i>int</i>
<i>TRAPEZOIDAL</i>	Constant TRAPEZOIDAL of type <i>int</i>

Attributes Documentation

```
CHEBYSHEV: int = CHEBYSHEV
    Constant CHEBYSHEV of type int
```

```

TRAPEZOIDAL: int = TRAPEZOIDAL
    Constant TRAPEZOIDAL of type int

__init__()

classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.RG.Type

```

class slepc4py.SLEPc.RG.Type
    Bases: object

    RG type.

    • INTERVAL: A (generalized) interval.

    • POLYGON: A polygonal region defined by its vertices.

    • ELLIPSE: An ellipse defined by its center, radius and vertical scale.

    • RING: A ring region.

```

See also
RGType

Attributes Summary

<i>ELLIPSE</i>	Object ELLIPSE of type <code>str</code>
<i>INTERVAL</i>	Object INTERVAL of type <code>str</code>
<i>POLYGON</i>	Object POLYGON of type <code>str</code>
<i>RING</i>	Object RING of type <code>str</code>

Attributes Documentation

```

ELLIPSE: str = ELLIPSE
    Object ELLIPSE of type str

INTERVAL: str = INTERVAL
    Object INTERVAL of type str

POLYGON: str = POLYGON
    Object POLYGON of type str

RING: str = RING
    Object RING of type str

__init__()

classmethod __new__(*args, **kwargs)

```

Methods Summary

<code>appendOptionsPrefix([prefix])</code>	Append to the prefix used for searching for all RG options in the database.
<code>canUseConjugates([realmats])</code>	Half of integration points can be avoided (use their conjugates).
<code>checkInside(a)</code>	Determine if a set of given points are inside the region or not.
<code>computeBoundingBox()</code>	Compute box containing the region.
<code>computeContour(n)</code>	Compute points on the contour of the region.
<code>computeQuadrature(quad, n)</code>	Compute the values of the parameters used in a quadrature rule.
<code>create([comm])</code>	Create the RG object.
<code>destroy()</code>	Destroy the RG object.
<code>getComplement()</code>	Get the flag indicating whether the region is complemented or not.
<code>getEllipseParameters()</code>	Get the parameters that define the ellipse region.
<code>getIntervalEndpoints()</code>	Get the parameters that define the interval region.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all RG options in the database.
<code>getPolygonVertices()</code>	Get the parameters that define the interval region.
<code>getRingParameters()</code>	Get the parameters that define the ring region.
<code>getScale()</code>	Get the scaling factor.
<code>getType()</code>	Get the RG type of this object.
<code>isAxisymmetric([vertical])</code>	Determine if the region is axisymmetric.
<code>isTrivial()</code>	Tell whether it is the trivial region (whole complex plane).
<code>setComplement([comp])</code>	Set a flag to indicate that the region is the complement of the specified one.
<code>setEllipseParameters(center, radius[, vscale])</code>	Set the parameters defining the ellipse region.
<code>setFromOptions()</code>	Set RG options from the options database.
<code>setIntervalEndpoints(a, b, c, d)</code>	Set the parameters defining the interval region.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all RG options in the database.
<code>setPolygonVertices(v)</code>	Set the vertices that define the polygon region.
<code>setRingParameters(center, radius, vscale, ...)</code>	Set the parameters defining the ring region.
<code>setScale([sfactor])</code>	Set the scaling factor to be used.
<code>setType(rg_type)</code>	Set the type for the RG object.
<code>view([viewer])</code>	Print the RG data structure.

Attributes Summary

<code>complement</code>	If the region is the complement of the specified one.
<code>scale</code>	The scaling factor to be used.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching for all RG options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all RG option requests.

Return type

None

See also

[setOptionsPrefix](#), [getOptionsPrefix](#), [RGAppendOptionsPrefix](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:190 <slepc4py/SLEPc/RG.pyx#L190>](#)

canUseConjugates (*realmats=True*)

Half of integration points can be avoided (use their conjugates).

Not collective.

Used in contour integral methods to determine whether half of integration points can be avoided (use their conjugates).

Parameters

realmats (*bool*) – True if the problem matrices are real.

Returns

Whether it is possible to use conjugates.

Return type

bool

Notes

If some integration points are the conjugates of other points, then the associated computational cost can be saved. This depends on the problem matrices being real and also the region being symmetric with respect to the horizontal axis. The result is `false` if using real arithmetic or in the case of a flat region (height equal to zero).

See also

[isAxisymmetric](#), [RGCanUseConjugates](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:458 <slepc4py/SLEPc/RG.pyx#L458>](#)

checkInside (*a*)

Determine if a set of given points are inside the region or not.

Not collective.

Parameters

a (*Sequence[complex]*) – The coordinates of the points.

Returns

Computed result for each point (1=inside, 0=on the contour, -1=outside).

Return type

ArrayInt

Notes

If a scaling factor was set, the points are scaled before checking.

See also

[`setScale`](#), [`setComplement`](#), [`RGCheckInside`](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:355 <slepc4py/SLEPc/RG.pyx#L355>](#)

computeBoundingBox()

Compute box containing the region.

Not collective.

Determine the endpoints of a rectangle in the complex plane that contains the region.

Returns

- **a** ([`float`](#)) – The left endpoint of the bounding box in the real axis.
- **b** ([`float`](#)) – The right endpoint of the bounding box in the real axis.
- **c** ([`float`](#)) – The bottom endpoint of the bounding box in the imaginary axis.
- **d** ([`float`](#)) – The top endpoint of the bounding box in the imaginary axis.

Return type

[`tuple`](#)[[`float`](#), [`float`](#), [`float`](#), [`float`](#)]

See also

[`computeContour`](#), [`setScale`](#), [`RGComputeBoundingBox`](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:430 <slepc4py/SLEPc/RG.pyx#L430>](#)

computeContour(*n*)

Compute points on the contour of the region.

Not collective.

Compute the coordinates of several points lying on the contour of the region.

Parameters

n ([`int`](#)) – The number of points to compute.

Returns

Computed points.

Return type

[`list`](#) of [`complex`](#)

See also

[`computeBoundingBox`](#), [`setScale`](#), [`RGComputeContour`](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:395 <slepc4py/SLEPc/RG.pyx#L395>](#)

computeQuadrature(*quad*, *n*)

Compute the values of the parameters used in a quadrature rule.

Not collective.

Compute the values of the parameters used in a quadrature rule for a contour integral around the boundary of the region.

Parameters

- **quad** ([QuadRule](#)) – The type of quadrature.
- **n** ([int](#)) – The number of quadrature points to compute.

Returns

- **z** ([ArrayScalar](#)) – Quadrature points.
- **zn** ([ArrayScalar](#)) – Normalized quadrature points.
- **w** ([ArrayScalar](#)) – Quadrature weights.

Return type

[tuple](#)[[ArrayScalar](#), [ArrayScalar](#), [ArrayScalar](#)]

Notes

In complex scalars, the values returned in *z* are often the same as those computed by [computeContour\(\)](#), but this is not the case in real scalars where all output arguments are real.

The computed values change for different quadrature rules.

See also

[computeContour](#), [RGComputeQuadrature](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:494 <slepc4py/SLEPc/RG.pyx#L494>](#)

create(*comm*=None)

Create the RG object.

Collective.

Parameters

comm ([Comm](#) | [None](#)) – MPI communicator; if not provided, it defaults to all processes.

Return type

[Self](#)

See also

[RGCreate](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:87 <slepc4py/SLEPc/RG.pyx#L87>](#)

destroy()

Destroy the RG object.

Collective.

See also
RGDestroy

:sources: [Source code at slepc4py/SLEPc/RG.pyx:73 <slepc4py/SLEPc/RG.pyx#L73>](#)

Return type

Self

getComplement()

Get the flag indicating whether the region is complemented or not.

Not collective.

Returns

Whether the region is complemented or not.

Return type

`bool`

See also
setComplement , RGGetComplement

:sources: [Source code at slepc4py/SLEPc/RG.pyx:277 <slepc4py/SLEPc/RG.pyx#L277>](#)

getEllipseParameters()

Get the parameters that define the ellipse region.

Not collective.

Returns

- **center** (*Scalar*) – The center.
- **radius** (`float`) – The radius.
- **vscale** (`float`) – The vertical scale.

Return type

`tuple[Scalar, float, float]`

See also
setEllipseParameters , RGEllipseGetParameters

:sources: [Source code at slepc4py/SLEPc/RG.pyx:572 <slepc4py/SLEPc/RG.pyx#L572>](#)

getIntervalEndpoints()

Get the parameters that define the interval region.

Not collective.

Returns

- **a** (`float`) – The left endpoint in the real axis.
- **b** (`float`) – The right endpoint in the real axis.

- **c** (*float*) – The bottom endpoint in the imaginary axis.
- **d** (*float*) – The top endpoint in the imaginary axis.

Return type

tuple[*float*, *float*, *float*, *float*]

See also

setIntervalEndpoints, *RGIntervalGetEndpoints*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:634 <slepc4py/SLEPc/RG.pyx#L634>`

getOptionsPrefix()

Get the prefix used for searching for all RG options in the database.

Not collective.

Returns

The prefix string set for this RG object.

Return type

str

See also

setOptionsPrefix, *appendOptionsPrefix*, *RGGetOptionsPrefix*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:171 <slepc4py/SLEPc/RG.pyx#L171>`

getPolygonVertices()

Get the parameters that define the interval region.

Not collective.

Returns

The vertices.

Return type

ArrayComplex

See also

setPolygonVertices, *RGPolygonGetVertices*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:690 <slepc4py/SLEPc/RG.pyx#L690>`

getRingParameters()

Get the parameters that define the ring region.

Not collective.

Returns

- **center** (*Scalar*) – The center.
- **radius** (*float*) – The radius.

- **vscale** (`float`) – The vertical scale.
- **start_ang** (`float`) – The right-hand side angle.
- **end_ang** (`float`) – The left-hand side angle.
- **width** (`float`) – The width of the ring.

Return type

`tuple[Scalar, float, float, float, float, float]`

See also

[`setRingParameters`](#), [`RGRingGetParameters`](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:775 <slepc4py/SLEPc/RG.pyx#L775>`

getScale()

Get the scaling factor.

Not collective.

Returns

The scaling factor.

Return type

`float`

See also

[`setScale`](#), [`RGGetScale`](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:336 <slepc4py/SLEPc/RG.pyx#L336>`

getType()

Get the RG type of this object.

Not collective.

Returns

The region type currently being used.

Return type

`str`

See also

[`setType`](#), [`RGGetType`](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:127 <slepc4py/SLEPc/RG.pyx#L127>`

isAxisymmetric(vertical=False)

Determine if the region is axisymmetric.

Not collective.

Determine if the region is symmetric with respect to the real or imaginary axis.

Parameters

vertical (*bool*) – True if symmetry must be checked against the vertical axis.

Returns

True if the region is axisymmetric.

Return type

bool

See also

canUseConjugates, *RGIsAxisymmetric*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:249 <slepc4py/SLEPc/RG.pyx#L249>`

isTrivial()

Tell whether it is the trivial region (whole complex plane).

Not collective.

Returns

True if the region is equal to the whole complex plane, e.g., an interval region with all four endpoints unbounded or an ellipse with infinite radius.

Return type

bool

See also

checkInside, *RGIsTrivial*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:228 <slepc4py/SLEPc/RG.pyx#L228>`

setComplement(comp=True)

Set a flag to indicate that the region is the complement of the specified one.

Logically collective.

Parameters

comp (*bool*) – Activate/deactivate the complementation of the region.

Return type

None

See also

getComplement, *RGSetComplement*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:296 <slepc4py/SLEPc/RG.pyx#L296>`

setEllipseParameters(center, radius, vscale=None)

Set the parameters defining the ellipse region.

Logically collective.

Parameters

- **center** (*Scalar*) – The center.
- **radius** (*float*) – The radius.
- **vscale** (*float* / *None*) – The vertical scale.

Return type

None

Notes

When PETSc is built with real scalars, the center is restricted to a real value.

See also

getEllipseParameters, *RGEllipseSetParameters*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:542 <slepc4py/SLEPc/RG.pyx#L542>`

setFromOptions()

Set RG options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

See also

setOptionsPrefix, *RGSetFromOptions*

:sources: `Source code at slepc4py/SLEPc/RG.pyx:209 <slepc4py/SLEPc/RG.pyx#L209>`

Return type

None

setIntervalEndpoints(*a*, *b*, *c*, *d*)

Set the parameters defining the interval region.

Logically collective.

Parameters

- **a** (*float*) – The left endpoint in the real axis.
- **b** (*float*) – The right endpoint in the real axis.
- **c** (*float*) – The bottom endpoint in the imaginary axis.
- **d** (*float*) – The top endpoint in the imaginary axis.

Return type

None

Notes

The region is defined as $[a, b]x[c, d]$. Particular cases are an interval on the real axis ($c = d = 0$), similarly for the imaginary axis ($a = b = 0$), the whole complex plane ($a = -\infty, b = \infty, c = -\infty, d = \infty$), and so on.

When PETSc is built with real scalars, the region must be symmetric with respect to the real axis.

See also

[`getIntervalEndpoints`](#), [`RGIntervalSetEndpoints`](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:597 <slepc4py/SLEPc/RG.pyx#L597>`

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all RG options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all RG option requests.

Return type

None

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

See also

[`appendOptionsPrefix`](#), [`getOptionsPrefix`](#), [`RGGetOptionsPrefix`](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:146 <slepc4py/SLEPc/RG.pyx#L146>`

setPolygonVertices(*v*)

Set the vertices that define the polygon region.

Logically collective.

Parameters

v (*Sequence[float]* / *Sequence[Scalar]*) – The vertices.

Return type

None

See also

[`getPolygonVertices`](#), [`RGPolygonSetVertices`](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:662 <slepc4py/SLEPc/RG.pyx#L662>`

setRingParameters(*center, radius, vscale, start_ang, end_ang, width*)

Set the parameters defining the ring region.

Logically collective.

Parameters

- **center** (*Scalar*) – The center.
- **radius** (*float*) – The radius.
- **vscale** (*float*) – The vertical scale.
- **start_ang** (*float*) – The right-hand side angle.
- **end_ang** (*float*) – The left-hand side angle.
- **width** (*float*) – The width of the ring.

Return type

None

Notes

The values of **center**, **radius** and **vscale** have the same meaning as in the ellipse region. The **start_ang** and **end_ang** define the span of the ring (by default it is the whole ring), while the **width** is the separation between the two concentric ellipses (above and below the radius by $\text{width}/2$).

The start and end angles are expressed as a fraction of the circumference. The allowed range is $[0, \dots, 1]$, with 0 corresponding to 0 radians, 0.25 to $\pi/2$ radians, and so on. It is allowed to have **start_ang** > **end_ang**, in which case the ring region crosses over the zero angle.

When PETSc is built with real scalars, the center is restricted to a real value, and the start and end angles must be such that the region is symmetric with respect to the real axis.

See also

[*getRingParameters*](#), [*RGRingSetParameters*](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:716](#) <[slepc4py/SLEPc/RG.pyx#L716](#)>

setScale(*sfactor=None*)

Set the scaling factor to be used.

Logically collective.

Set the scaling factor to be used when checking that a point is inside the region and when computing the contour.

Parameters

- **sfactor** (*float*) – The scaling factor (default=1).

Return type

None

See also

[*getScale*](#), [*checkInside*](#), [*computeContour*](#), [*RGSetScale*](#)

:sources: [Source code at slepc4py/SLEPc/RG.pyx:314](#) <[slepc4py/SLEPc/RG.pyx#L314](#)>

setType(*rg_type*)

Set the type for the RG object.

Logically collective.

Parameters

rg_type (*Type* / *str*) – The region type to be used.

Return type

None

See also

[getType](#), [RGSetType](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:108 <slepc4py/SLEPc/RG.pyx#L108>`

view(*viewer=None*)

Print the RG data structure.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

[RGView](#)

:sources: `Source code at slepc4py/SLEPc/RG.pyx:54 <slepc4py/SLEPc/RG.pyx#L54>`

Attributes Documentation

complement

If the region is the complement of the specified one.

:sources: `Source code at slepc4py/SLEPc/RG.pyx:811 <slepc4py/SLEPc/RG.pyx#L811>`

scale

The scaling factor to be used.

:sources: `Source code at slepc4py/SLEPc/RG.pyx:818 <slepc4py/SLEPc/RG.pyx#L818>`

__init__()

classmethod **__new__**(*args, **kwargs)

slepc4py.SLEPc.ST

class **slepc4py.SLEPc.ST**

Bases: [Object](#)

Spectral Transformation.

The Spectral Transformation (*ST*) class encapsulates the functionality required for acceleration techniques based on the transformation of the spectrum. The eigensolvers implemented 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. Polynomial eigensolvers in [PEP](#) also support spectral transformation.

Enumerations

<i>FilterDamping</i>	ST filter damping.
<i>FilterType</i>	ST filter type.
<i>MatMode</i>	ST matrix mode.
<i>Type</i>	ST type.

slepc4py.SLEPc.ST.FilterDamping

class slepc4py.SLEPc.ST.**FilterDamping**

Bases: `object`

ST filter damping.

- *NONE*: No damping
- *JACKSON*: Jackson damping
- *LANCZOS*: Lanczos damping
- *FEJER*: Fejer damping

See also

`STFilterDamping`

Attributes Summary

<i>FEJER</i>	Constant FEJER of type <code>int</code>
<i>JACKSON</i>	Constant JACKSON of type <code>int</code>
<i>LANCZOS</i>	Constant LANCZOS of type <code>int</code>
<i>NONE</i>	Constant NONE of type <code>int</code>

Attributes Documentation

FEJER: `int` = **FEJER**

Constant FEJER of type `int`

JACKSON: `int` = **JACKSON**

Constant JACKSON of type `int`

LANCZOS: `int` = **LANCZOS**

Constant LANCZOS of type `int`

NONE: `int` = **NONE**

Constant NONE of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.ST.FilterType

class slepc4py.SLEPc.ST.**FilterType**

Bases: `object`

ST filter type.

- **FILTLAN**: An adapted implementation of the Filtered Lanczos Package.
- **CHEBYSHEV**: A polynomial filter based on a truncated Chebyshev series.

See also
<code>STFilterType</code>

Attributes Summary

<code>CHEBYSHEV</code>	Constant CHEBYSHEV of type <code>int</code>
<code>FILTLAN</code>	Constant FILTLAN of type <code>int</code>

Attributes Documentation

CHEBYSHEV: `int` = **CHEBYSHEV**

Constant CHEBYSHEV of type `int`

FILTLAN: `int` = **FILTLAN**

Constant FILTLAN of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.ST.MatMode

class slepc4py.SLEPc.ST.**MatMode**

Bases: `object`

ST matrix mode.

- **COPY**: A working copy of the matrix is created.
- **INPLACE**: The operation is computed in-place.
- **SHELL**: The matrix $A - \sigma B$ is handled as an implicit matrix.

See also
<code>STMatMode</code>

Attributes Summary

<code>COPY</code>	Constant COPY of type <code>int</code>
<code>INPLACE</code>	Constant INPLACE of type <code>int</code>

continues on next page

Table 93 – continued from previous page

<i>SHELL</i>	Constant SHELL of type <code>int</code>
--------------	---

Attributes Documentation

COPY: `int` = **COPY**

Constant COPY of type `int`

INPLACE: `int` = **INPLACE**

Constant INPLACE of type `int`

SHELL: `int` = **SHELL**

Constant SHELL of type `int`

`__init__()`

classmethod **`__new__()`**(*args, **kwargs)

slepc4py.SLEPc.ST.Type

class slepc4py.SLEPc.ST.Type

Bases: `object`

ST type.

- *SHIFT*: Shift from origin.
- *SINVERT*: Shift-and-invert.
- *CAYLEY*: Cayley transform.
- *PRECOND*: Preconditioner.
- *FILTER*: Polynomial filter.
- *SHELL*: User-defined.

See also

STType

Attributes Summary

<i>CAYLEY</i>	Object CAYLEY of type <code>str</code>
<i>FILTER</i>	Object FILTER of type <code>str</code>
<i>PRECOND</i>	Object PRECOND of type <code>str</code>
<i>SHELL</i>	Object SHELL of type <code>str</code>
<i>SHIFT</i>	Object SHIFT of type <code>str</code>
<i>SINVERT</i>	Object SINVERT of type <code>str</code>

Attributes Documentation

CAYLEY: `str` = **CAYLEY**

Object CAYLEY of type `str`

FILTER: `str = FILTER`
 Object FILTER of type `str`

PRECOND: `str = PRECOND`
 Object PRECOND of type `str`

SHELL: `str = SHELL`
 Object SHELL of type `str`

SHIFT: `str = SHIFT`
 Object SHIFT of type `str`

SINVERT: `str = SINVERT`
 Object SINVERT of type `str`

`__init__()`

`classmethod __new__(*args, **kwargs)`

Methods Summary

<code>appendOptionsPrefix([prefix])</code>	Append to the prefix used for searching for all ST options in the database.
<code>apply(x, y)</code>	Apply the spectral transformation operator to a vector.
<code>applyHermitianTranspose(x, y)</code>	Apply the Hermitian-transpose of the operator to a vector.
<code>applyMat(X, Y)</code>	Apply the spectral transformation operator to a matrix.
<code>applyTranspose(x, y)</code>	Apply the transpose of the operator to a vector.
<code>create([comm])</code>	Create the ST object.
<code>destroy()</code>	Destroy the ST object.
<code>getCayleyAntishift()</code>	Get the value of the anti-shift for the Cayley spectral transformation.
<code>getFilterDamping()</code>	Get the type of damping used in the polynomial filter.
<code>getFilterDegree()</code>	Get the degree of the filter polynomial.
<code>getFilterInterval()</code>	Get the interval containing the desired eigenvalues.
<code>getFilterRange()</code>	Get the interval containing all eigenvalues.
<code>getFilterType()</code>	Get the method to be used to build the polynomial filter.
<code>getKSP()</code>	Get the KSP object associated with the spectral transformation.
<code>getMatMode()</code>	Get a flag that indicates how the matrix is being shifted.
<code>getMatStructure()</code>	Get the internal matrix structure attribute.
<code>getMatrices()</code>	Get the matrices associated with the eigenvalue problem.
<code>getOperator()</code>	Get a shell matrix that represents the operator of the spectral transformation.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all ST options in the database.
<code>getPreconditionerMat()</code>	Get the matrix previously set by <code>setPreconditionerMat()</code> .

continues on next page

Table 95 – continued from previous page

<i>getShift()</i>	Get the shift associated with the spectral transformation.
<i>getSplitPreconditioner()</i>	Get the matrices to be used to build the preconditioner.
<i>getTransform()</i>	Get the flag indicating whether the transformed matrices are computed or not.
<i>getType()</i>	Get the ST type of this object.
<i>reset()</i>	Reset the ST object.
<i>restoreOperator(op)</i>	Restore the previously seized operator matrix.
<i>setCayleyAntishift(mu)</i>	Set the value of the anti-shift for the Cayley spectral transformation.
<i>setFilterDamping(damping)</i>	Set the type of damping to be used in the polynomial filter.
<i>setFilterDegree(deg)</i>	Set the degree of the filter polynomial.
<i>setFilterInterval(inta, intb)</i>	Set the interval containing the desired eigenvalues.
<i>setFilterRange(left, right)</i>	Set the numerical range (or field of values) of the matrix.
<i>setFilterType(filter_type)</i>	Set the method to be used to build the polynomial filter.
<i>setFromOptions()</i>	Set ST options from the options database.
<i>setKSP(ksp)</i>	Set the KSP object associated with the spectral transformation.
<i>setMatMode(mode)</i>	Set a flag related to management of transformed matrices.
<i>setMatStructure(structure)</i>	Set the matrix structure attribute.
<i>setMatrices(operators)</i>	Set the matrices associated with the eigenvalue problem.
<i>setOptionsPrefix([prefix])</i>	Set the prefix used for searching for all ST options in the database.
<i>setPreconditionerMat([P])</i>	Set the matrix to be used to build the preconditioner.
<i>setShift(shift)</i>	Set the shift associated with the spectral transformation.
<i>setSplitPreconditioner(operators[, structure])</i>	Set the matrices to be used to build the preconditioner.
<i>setTransform([flag])</i>	Set a flag to indicate whether the transformed matrices are computed or not.
<i>setType(st_type)</i>	Set the particular spectral transformation to be used.
<i>setUp()</i>	Prepare for the use of a spectral transformation.
<i>view([viewer])</i>	Print the ST data structure.

Attributes Summary

<i>ksp</i>	KSP object associated with the spectral transformation.
<i>mat_mode</i>	How the transformed matrices are being stored in the ST.
<i>mat_structure</i>	Relation of the sparsity pattern of all ST matrices.
<i>shift</i>	Value of the shift.
<i>transform</i>	If the transformed matrices are computed.

Methods Documentation

appendOptionsPrefix(*prefix=None*)

Append to the prefix used for searching for all ST options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all ST option requests.

Return type

None

See also

setOptionsPrefix, *getOptionsPrefix*, *STAppendOptionsPrefix*

:sources: [Source code at slepc4py/SLEPc/ST.pyx:256 <slepc4py/SLEPc/ST.pyx#L256>](#)

apply(*x*, *y*)

Apply the spectral transformation operator to a vector.

Collective.

Apply the spectral transformation operator to a vector, for instance $y = (A - \sigma B)^{-1} Bx$ in the case of the shift-and-invert transformation and generalized eigenproblem.

Parameters

- **x** (*Vec*) – The input vector.
- **y** (*Vec*) – The result vector.

Return type

None

See also

applyTranspose, *applyHermitianTranspose*, *applyMat*, *STApply*

:sources: [Source code at slepc4py/SLEPc/ST.pyx:758 <slepc4py/SLEPc/ST.pyx#L758>](#)

applyHermitianTranspose(*x*, *y*)

Apply the Hermitian-transpose of the operator to a vector.

Collective.

Apply the Hermitian-transpose of the operator to a vector, for instance $y = B^*(A - \sigma B)^{-*} x$ in the case of the shift-and-invert transformation and generalized eigenproblem.

Parameters

- **x** (*Vec*) – The input vector.
- **y** (*Vec*) – The result vector.

Return type

None

See also[apply](#), [applyTranspose](#), [STApplyHermitianTranspose](#)**:sources:**``Source code at slepc4py/SLEPc/ST.pyx:804 <slepc4py/SLEPc/ST.pyx#L804>``**applyMat**(*X*, *Y*)

Apply the spectral transformation operator to a matrix.

Collective.

Apply the spectral transformation operator to a matrix, for instance $Y = (A - \sigma B)^{-1}BX$ in the case of the shift-and-invert transformation and generalized eigenproblem.

Parameters

- **X** (*Mat*) – The input matrix.
- **Y** (*Mat*) – The result matrix.

Return type

None

See also[apply](#), [STApplyMat](#)**:sources:**``Source code at slepc4py/SLEPc/ST.pyx:827 <slepc4py/SLEPc/ST.pyx#L827>``**applyTranspose**(*x*, *y*)

Apply the transpose of the operator to a vector.

Collective.

Apply the transpose of the operator to a vector, for instance $y = B^T(A - \sigma B)^{-T}x$ in the case of the shift-and-invert transformation and generalized eigenproblem.

Parameters

- **x** (*Vec*) – The input vector.
- **y** (*Vec*) – The result vector.

Return type

None

See also[apply](#), [applyHermitianTranspose](#), [STApplyTranspose](#)**:sources:**``Source code at slepc4py/SLEPc/ST.pyx:781 <slepc4py/SLEPc/ST.pyx#L781>``**create**(*comm=None*)

Create the ST object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator; if not provided, it defaults to all processes.

Return type*Self***See also**[STCreate](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:145 <slepc4py/SLEPc/ST.pyx#L145>](#)

destroy()

Destroy the ST object.

Collective.

See also[STDestroy](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:119 <slepc4py/SLEPc/ST.pyx#L119>](#)

Return type*Self***getCayleyAntishift()**

Get the value of the anti-shift for the Cayley spectral transformation.

Not collective.

Returns

The anti-shift.

Return type*Scalar***See also**[setCayleyAntishift](#), [STCayleyGetAntishift](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:942 <slepc4py/SLEPc/ST.pyx#L942>](#)

getFilterDamping()

Get the type of damping used in the polynomial filter.

Not collective.

Returns

The type of damping.

Return type*FilterDamping*

See also

setFilterDamping, *STFilterGetDamping*

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1164 <slepc4py/SLEPc/ST.pyx#L1164>`

getFilterDegree()

Get the degree of the filter polynomial.

Not collective.

Returns

The polynomial degree.

Return type

`int`

See also

setFilterDegree, *STFilterGetDegree*

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1123 <slepc4py/SLEPc/ST.pyx#L1123>`

getFilterInterval()

Get the interval containing the desired eigenvalues.

Not collective.

Returns

- **inta** (`float`) – The left end of the interval.
- **intb** (`float`) – The right end of the interval.

Return type

`tuple[float, float]`

See also

setFilterInterval, *STFilterGetInterval*

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1031 <slepc4py/SLEPc/ST.pyx#L1031>`

getFilterRange()

Get the interval containing all eigenvalues.

Not collective.

Returns

- **left** (`float`) – The left end of the spectral range.
- **right** (`float`) – The right end of the spectral range.

Return type

`tuple[float, float]`

See also

getFilterInterval, *STFilterGetRange*

:sources: ``Source code at slepc4py/SLEPc/ST.pyx:1083 <slepc4py/SLEPc/ST.pyx#L1083>``

getFilterType()

Get the method to be used to build the polynomial filter.

Not collective.

Returns

The type of filter.

Return type

FilterType

See also

setFilterType, *STFilterGetType*

:sources: ``Source code at slepc4py/SLEPc/ST.pyx:979 <slepc4py/SLEPc/ST.pyx#L979>``

getKSP()

Get the KSP object associated with the spectral transformation.

Collective.

Returns

The linear solver object.

Return type

`petsc4py.PETSc.KSP`

See also

setKSP, *STGetKSP*

:sources: ``Source code at slepc4py/SLEPc/ST.pyx:585 <slepc4py/SLEPc/ST.pyx#L585>``

getMatMode()

Get a flag that indicates how the matrix is being shifted.

Not collective.

Get a flag that indicates how the matrix is being shifted in the shift-and-invert and Cayley spectral transformations.

Returns

The mode flag.

Return type

MatMode

See also

[*setMatMode*](#), [*STGetMatMode*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:430 <slepc4py/SLEPc/ST.pyx#L430>](#)

getMatStructure()

Get the internal matrix structure attribute.

Not collective.

Get the internal `petsc4py.PETSc.Mat.Structure` attribute to indicate which is the relation of the sparsity pattern of the matrices.

Returns

The structure flag.

Return type

`petsc4py.PETSc.Mat.Structure`

See also

[*setMatStructure*](#), [*STGetMatStructure*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:545 <slepc4py/SLEPc/ST.pyx#L545>](#)

getMatrices()

Get the matrices associated with the eigenvalue problem.

Collective.

Returns

The matrices associated with the eigensystem.

Return type

`list of petsc4py.PETSc.Mat`

See also

[*setMatrices*](#), [*STGetNumMatrices*](#), [*STGetMatrix*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:486 <slepc4py/SLEPc/ST.pyx#L486>](#)

getOperator()

Get a shell matrix that represents the operator of the spectral transformation.

Collective.

Returns

Operator matrix.

Return type

`petsc4py.PETSc.Mat`

Notes

The operator is defined in linear eigenproblems only, not in polynomial ones, so the call will fail if more than 2 matrices were passed in `setMatrices()`.

The returned shell matrix is essentially a wrapper to the `apply()` and `applyTranspose()` operations. The operator can often be expressed as

$$Op = DK^{-1}MD^{-1}$$

where D is the balancing matrix, and M and K are two matrices corresponding to the numerator and denominator for spectral transformations that represent a rational matrix function.

The preconditioner matrix K typically depends on the value of the shift, and its inverse is handled via an internal KSP object. Normal usage does not require explicitly calling `getOperator()`, but it can be used to force the creation of K and M , and then K is passed to the KSP. This is useful for setting options associated with the PCFactor (to set MUMPS options, for instance).

The returned matrix must NOT be destroyed by the user. Instead, when no longer needed it must be returned with `restoreOperator()`. In particular, this is required before modifying the `ST` matrices or the shift.

See also

`apply`, `setMatrices`, `setShift`, `restoreOperator`, `STGetOperator`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:850 <slepc4py/SLEPc/ST.pyx#L850>`

`getOptionsPrefix()`

Get the prefix used for searching for all ST options in the database.

Not collective.

Returns

The prefix string set for this ST object.

Return type

`str`

See also

`setOptionsPrefix`, `appendOptionsPrefix`, `STGetOptionsPrefix`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:237 <slepc4py/SLEPc/ST.pyx#L237>`

`getPreconditionerMat()`

Get the matrix previously set by `setPreconditionerMat()`.

Not collective.

Returns

The matrix that will be used in constructing the preconditioner.

Return type

`petsc4py.PETSc.Mat`

See also

`setPreconditionerMat`, `STGetPreconditionerMat`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:649 <slepc4py/SLEPc/ST.pyx#L649>`

getShift()

Get the shift associated with the spectral transformation.

Not collective.

Returns

The value of the shift.

Return type

Scalar

See also

`setShift`, `STGetShift`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:323 <slepc4py/SLEPc/ST.pyx#L323>`

getSplitPreconditioner()

Get the matrices to be used to build the preconditioner.

Not collective.

Returns

- `list` of `petsc4py.PETSc.Mat` – The list of matrices associated with the preconditioner.
- `petsc4py.PETSc.Mat.Structure` – The structure flag.

Return type

`tuple[list[petsc4py.PETSc.Mat], petsc4py.PETSc.Mat.Structure]`

See also

`STGetSplitPreconditionerInfo`, `STGetSplitPreconditionerTerm`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:716 <slepc4py/SLEPc/ST.pyx#L716>`

getTransform()

Get the flag indicating whether the transformed matrices are computed or not.

Not collective.

Returns

This flag is intended for the case of polynomial eigenproblems solved via linearization. If this flag is `False` (default) the spectral transformation is applied to the linearization (handled by the eigensolver), otherwise it is applied to the original problem.

Return type

`bool`

See also

[*setTransform*](#), [*STGetTransform*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:364](#) <[slepc4py/SLEPc/ST.pyx#L364](#)>

getType()

Get the ST type of this object.

Not collective.

Returns

The spectral transformation currently being used.

Return type

`str`

See also

[*setType*](#), [*STGetType*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:193](#) <[slepc4py/SLEPc/ST.pyx#L193](#)>

reset()

Reset the ST object.

Collective.

See also

[*STReset*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:133](#) <[slepc4py/SLEPc/ST.pyx#L133](#)>

Return type

`None`

restoreOperator(*op*)

Restore the previously seized operator matrix.

Logically collective.

Parameters

op (*Mat*) – Operator matrix previously obtained with [*getOperator\(\)*](#).

Return type

`None`

See also

[*getOperator*](#), [*STRestoreOperator*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:898](#) <[slepc4py/SLEPc/ST.pyx#L898](#)>

setCayleyAntishift(*mu*)

Set the value of the anti-shift for the Cayley spectral transformation.

Logically collective.

Parameters

mu ([Scalar](#)) – The anti-shift.

Return type

[None](#)

Notes

In the generalized Cayley transform, the operator can be expressed as $(A - \sigma B)^{-1}(A + \mu B)$. This function sets the value of *mu*. Use [setShift\(\)](#) for setting σ .

See also

[setShift](#), [getCayleyAntishift](#), [STCayleySetAntishift](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:918 <slepc4py/SLEPc/ST.pyx#L918>](#)

setFilterDamping(*damping*)

Set the type of damping to be used in the polynomial filter.

Logically collective.

Parameter

damping

The type of damping.

Notes

Only used in [FilterType.CHEBYSHEV](#) filters.

See also

[getFilterDamping](#), [STFilterSetDamping](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:1142 <slepc4py/SLEPc/ST.pyx#L1142>](#)

Parameters

damping ([FilterDamping](#))

Return type

[None](#)

setFilterDegree(*deg*)

Set the degree of the filter polynomial.

Logically collective.

Parameters

deg ([int](#)) – The polynomial degree.

Return type

[None](#)

See also

[`getFilterDegree`](#), [`STFilterSetDegree`](#)

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1105 <slepc4py/SLEPc/ST.pyx#L1105>`

setFilterInterval(*inta*, *intb*)

Set the interval containing the desired eigenvalues.

Logically collective.

Parameters

- **inta** (*float*) – The left end of the interval.
- **intb** (*float*) – The right end of the interval.

Return type

`None`

Notes

The filter will be configured to emphasize eigenvalues contained in the given interval, and damp out eigenvalues outside it. If the interval is open, then the filter is low- or high-pass, otherwise it is mid-pass.

Common usage is to set the interval in *EPS* with *EPS.setInterval()*.

The interval must be contained within the numerical range of the matrix, see [`setFilterRange\(\)`](#).

See also

[`getFilterInterval`](#), [`setFilterRange`](#), [`STFilterSetInterval`](#)

:sources: `Source code at slepc4py/SLEPc/ST.pyx:998 <slepc4py/SLEPc/ST.pyx#L998>`

setFilterRange(*left*, *right*)

Set the numerical range (or field of values) of the matrix.

Logically collective.

Set the numerical range (or field of values) of the matrix, that is, the interval containing all eigenvalues.

Parameters

- **left** (*float*) – The left end of the spectral range.
- **right** (*float*) – The right end of the spectral range.

Return type

`None`

Notes

The filter will be most effective if the numerical range is tight, that is, **left** and **right** are good approximations to the leftmost and rightmost eigenvalues, respectively.

See also

[*setFilterInterval*](#), [*getFilterRange*](#), [*STFilterSetRange*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:1053 <slepc4py/SLEPc/ST.pyx#L1053>](#)

setFilterType(*filter_type*)

Set the method to be used to build the polynomial filter.

Logically collective.

Parameter

filter_type

The type of filter.

See also

[*getFilterType*](#), [*STFilterSetType*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:961 <slepc4py/SLEPc/ST.pyx#L961>](#)

Parameters

filter_type ([*FilterType*](#))

Return type

[*None*](#)

setFromOptions()

Set ST options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before [*setUp*](#)() if the user is to be allowed to set the solver type.

See also

[*setOptionsPrefix*](#), [*STSetFromOptions*](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:275 <slepc4py/SLEPc/ST.pyx#L275>](#)

Return type

[*None*](#)

setKSP(*ksp*)

Set the KSP object associated with the spectral transformation.

Collective.

Parameters

ksp ([*KSP*](#)) – The linear solver object.

Return type

None

See also[getKSP](#), [STSetKSP](#)**:sources:** [Source code at slepc4py/SLEPc/ST.pyx:568 <slepc4py/SLEPc/ST.pyx#L568>](#)**setMatMode(mode)**

Set a flag related to management of transformed matrices.

Logically collective.

The flag indicates how the transformed matrices are being stored in the spectral transformation.

Parameters**mode** ([MatMode](#)) – The mode flag.**Return type**

None

NotesBy default ([ST.MatMode.COPY](#)), a copy of matrix A is made and then this copy is modified explicitly, e.g., $A \leftarrow (A - \sigma B)$.With [ST.MatMode.INPLACE](#), the original matrix A is modified at [setUp\(\)](#) and reverted at the end of the computations. With respect to the previous one, this mode avoids a copy of matrix A . However, a drawback is that the recovered matrix might be slightly different from the original one (due to roundoff).With [ST.MatMode.SHELL](#), the solver works with an implicit shell matrix that represents the shifted matrix. This mode is the most efficient in creating the transformed matrix but it places serious limitations to the linear solves performed in each iteration of the eigensolver (typically, only iterative solvers with Jacobi preconditioning can be used).In the two first modes the efficiency of this computation can be controlled with [setMatStructure\(\)](#).**See also**[setMatrices](#), [setMatStructure](#), [getMatMode](#), [STSetMatMode](#)**:sources:** [Source code at slepc4py/SLEPc/ST.pyx:387 <slepc4py/SLEPc/ST.pyx#L387>](#)**setMatStructure(structure)**

Set the matrix structure attribute.

Logically collective.

Set an internal `petsc4py.PETSc.Mat.Structure` attribute to indicate which is the relation of the sparsity pattern of all the [ST](#) matrices.**Parameters****structure** (`petsc4py.PETSc.Mat.Structure`) – The matrix structure specification.**Return type**

None

Notes

By default, the sparsity patterns are assumed to be different. If the patterns are equal or a subset then it is recommended to set this attribute for efficiency reasons (in particular, for internal `Mat . axpy()` operations).

This function has no effect in the case of standard eigenproblems.

In case of polynomial eigenproblems, the flag applies to all matrices relative to the first one.

See also

`getMatStructure`, `setMatrices`, `STSetMatStructure`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:512 <slepc4py/SLEPc/ST.pyx#L512>`

setMatrices(*operators*)

Set the matrices associated with the eigenvalue problem.

Collective.

Parameters

operators (*list* [*Mat*]) – The matrices associated with the eigensystem.

Return type

`None`

Notes

It must be called before `setUp()`. If it is called again after `setUp()` then the *ST* object is reset.

In standard eigenproblems only one matrix is passed, while in generalized problems two matrices are provided. The number of matrices is larger in polynomial eigenproblems.

In normal usage, matrices are provided via the corresponding *EPS* of *PEP* interface function.

See also

`getMatrices`, `setUp`, `reset`, `STSetMatrices`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:452 <slepc4py/SLEPc/ST.pyx#L452>`

setOptionsPrefix(*prefix=*`None`)

Set the prefix used for searching for all ST options in the database.

Logically collective.

Parameters

prefix (*str* / `None`) – The prefix string to prepend to all ST option requests.

Return type

`None`

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

See also

[appendOptionsPrefix](#), [getOptionsPrefix](#), [STGetOptionsPrefix](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:212](#) <[slepc4py/SLEPc/ST.pyx#L212](#)>

setPreconditionerMat(*P=None*)

Set the matrix to be used to build the preconditioner.

Collective.

Parameters

P (*Mat* / *None*) – The matrix that will be used in constructing the preconditioner.

Return type

None

Notes

This matrix will be passed to the internal KSP object (via the last argument of `KSP.setOperators()`) as the matrix to be used when constructing the preconditioner. If no matrix is set then $A - \sigma B$ will be used to build the preconditioner, being σ the value set by [setShift\(\)](#).

More precisely, this is relevant for spectral transformations that represent a rational matrix function, and use a KSP object for the denominator. It includes also the [PRECOND](#) case. If the user has a good approximation to matrix that can be used to build a cheap preconditioner, it can be passed with this function. Note that it affects only the `Pmat` argument of `KSP.setOperators()`, not the `Amat` argument.

If a preconditioner matrix is set, the default is to use an iterative KSP rather than a direct method.

An alternative to pass an approximation of $A - \sigma B$ with this function is to provide approximations of A and B via [setSplitPreconditioner\(\)](#). The difference is that when σ changes the preconditioner is recomputed.

A call with no matrix argument will remove a previously set matrix.

See also

[getPreconditionerMat](#), [STSetPreconditionerMat](#)

:sources: [Source code at slepc4py/SLEPc/ST.pyx:605](#) <[slepc4py/SLEPc/ST.pyx#L605](#)>

setShift(*shift*)

Set the shift associated with the spectral transformation.

Collective.

Parameters

shift (*Scalar*) – The value of the shift.

Return type

None

Notes

In some spectral transformations, changing the shift may have associated a lot of work, for example recomputing a factorization.

This function is normally not directly called by users, since the shift is indirectly set by `EPS.setTarget()`.

See also

`getShift`, `STSetShift`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:296 <slepc4py/SLEPc/ST.pyx#L296>`

setSplitPreconditioner(*operators*, *structure=None*)

Set the matrices to be used to build the preconditioner.

Collective.

Parameters

- **operators** (`list[petsc4py.PETSc.Mat]`) – The matrices associated with the preconditioner.
- **structure** (`petsc4py.PETSc.Mat.Structure` / `None`) – The matrix structure specification.

Return type

`None`

Notes

The number of matrices passed here must be the same as in `setMatrices()`.

For linear eigenproblems, the preconditioner matrix is computed as $P(\sigma) = A_0 - \sigma B_0$, where A_0, B_0 are approximations of A, B (the eigenproblem matrices) provided via the `operators` argument in this function. Compared to `setPreconditionerMat()`, this function allows setting a preconditioner in a way that is independent of the shift σ . Whenever the value of σ changes the preconditioner is recomputed.

Similarly, for polynomial eigenproblems the matrix for the preconditioner is expressed as $P(\sigma) = \sum_i P_i \phi_i(\sigma)$, for $i = 1, \dots, n$, where P_i are given in `operators` and the ϕ_i 's are the polynomial basis functions.

The structure flag provides information about the relative nonzero pattern of the operators matrices, in the same way as in `setMatStructure()`.

See also

`getSplitPreconditioner`, `setPreconditionerMat`, `STSetSplitPreconditioner`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:669 <slepc4py/SLEPc/ST.pyx#L669>`

setTransform(*flag=True*)

Set a flag to indicate whether the transformed matrices are computed or not.

Logically collective.

Parameters

- **flag** (`bool`) – This flag is intended for the case of polynomial eigenproblems solved via

linearization. If this flag is `False` (default) the spectral transformation is applied to the linearization (handled by the eigensolver), otherwise it is applied to the original problem.

Return type

`None`

See also

`getTransform`, `STSetTransform`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:342 <slepc4py/SLEPc/ST.pyx#L342>`

setType(*st_type*)

Set the particular spectral transformation to be used.

Logically collective.

Parameters

st_type (`Type` / `str`) – The spectral transformation to be used.

Return type

`None`

Notes

The default is `SHIFT` with a zero shift. Normally, it is best to use `setFromOptions()` and then set the ST type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also

`getType`, `STSetType`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:166 <slepc4py/SLEPc/ST.pyx#L166>`

setUp()

Prepare for the use of a spectral transformation.

Collective.

See also

`apply`, `STSetUp`

:sources: `Source code at slepc4py/SLEPc/ST.pyx:746 <slepc4py/SLEPc/ST.pyx#L746>`

Return type

`None`

view(*viewer=None*)

Print the ST data structure.

Collective.

Parameters

viewer (`Viewer` / `None`) – Visualization context; if not provided, the standard output is used.

Return type
None

See also

STView

:sources: `Source code at slepc4py/SLEPc/ST.pyx:100 <slepc4py/SLEPc/ST.pyx#L100>`

Attributes Documentation

ksp

KSP object associated with the spectral transformation.

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1213 <slepc4py/SLEPc/ST.pyx#L1213>`

mat_mode

How the transformed matrices are being stored in the ST.

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1199 <slepc4py/SLEPc/ST.pyx#L1199>`

mat_structure

Relation of the sparsity pattern of all ST matrices.

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1206 <slepc4py/SLEPc/ST.pyx#L1206>`

shift

Value of the shift.

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1185 <slepc4py/SLEPc/ST.pyx#L1185>`

transform

If the transformed matrices are computed.

:sources: `Source code at slepc4py/SLEPc/ST.pyx:1192 <slepc4py/SLEPc/ST.pyx#L1192>`

__init__()

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.SVD

class `slepc4py.SLEPc.SVD`

Bases: `Object`

Singular Value Decomposition Solver.

The Singular Value Decomposition Solver (*SVD*) is very similar to the *EPS* object, but intended for the computation of the partial SVD of a rectangular matrix. With this type of object, the user can specify an SVD problem and solve it with any of the different solvers encapsulated by the package. Some of these solvers are actually implemented through calls to *EPS* eigensolvers.

Enumerations

<i>Conv</i>	SVD convergence test.
<i>ConvergedReason</i>	SVD convergence reasons.

continues on next page

Table 97 – continued from previous page

<i>ErrorType</i>	SVD error type to assess accuracy of computed solutions.
<i>ProblemType</i>	SVD problem type.
<i>Stop</i>	SVD stopping test.
<i>TRLanczosGBidiag</i>	SVD TRLanczos bidiagonalization choices for the GSVD case.
<i>Type</i>	SVD type.
<i>Which</i>	SVD desired part of spectrum.

slepc4py.SLEPc.SVD.Conv**class** slepc4py.SLEPc.SVD.ConvBases: `object`

SVD convergence test.

- *ABS*: Absolute convergence test.
- *REL*: Convergence test relative to the singular value.
- *NORM*: Convergence test relative to the matrix norms.
- *MAXIT*: No convergence until maximum number of iterations has been reached.
- *USER*: User-defined convergence test.

See also

SVDConv

Attributes Summary

<i>ABS</i>	Constant <i>ABS</i> of type <code>int</code>
<i>MAXIT</i>	Constant <i>MAXIT</i> of type <code>int</code>
<i>NORM</i>	Constant <i>NORM</i> of type <code>int</code>
<i>REL</i>	Constant <i>REL</i> of type <code>int</code>
<i>USER</i>	Constant <i>USER</i> of type <code>int</code>

Attributes Documentation**ABS:** `int` = **ABS**Constant *ABS* of type `int`**MAXIT:** `int` = **MAXIT**Constant *MAXIT* of type `int`**NORM:** `int` = **NORM**Constant *NORM* of type `int`**REL:** `int` = **REL**Constant *REL* of type `int`**USER:** `int` = **USER**Constant *USER* of type `int`

```
__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.SVD.ConvergedReason

class slepc4py.SLEPc.SVD.ConvergedReason

Bases: `object`

SVD convergence reasons.

- *CONVERGED_TOL*: All eigenpairs converged to requested tolerance.
- *CONVERGED_USER*: User-defined convergence criterion satisfied.
- *CONVERGED_MAXIT*: Maximum iterations completed in case MAXIT convergence criterion.
- *DIVERGED_ITS*: Maximum number of iterations exceeded.
- *DIVERGED_BREAKDOWN*: Solver failed due to breakdown.
- *DIVERGED_SYMMETRY_LOST*: Underlying indefinite eigensolver was not able to keep symmetry.
- *CONVERGED_ITERATING*: Iteration not finished yet.

See also

SVDConvergedReason

Attributes Summary

<i>CONVERGED_ITERATING</i>	Constant CONVERGED_ITERATING of type <code>int</code>
<i>CONVERGED_MAXIT</i>	Constant CONVERGED_MAXIT of type <code>int</code>
<i>CONVERGED_TOL</i>	Constant CONVERGED_TOL of type <code>int</code>
<i>CONVERGED_USER</i>	Constant CONVERGED_USER of type <code>int</code>
<i>DIVERGED_BREAKDOWN</i>	Constant DIVERGED_BREAKDOWN of type <code>int</code>
<i>DIVERGED_ITS</i>	Constant DIVERGED_ITS of type <code>int</code>
<i>DIVERGED_SYMMETRY_LOST</i>	Constant DIVERGED_SYMMETRY_LOST of type <code>int</code>
<i>ITERATING</i>	Constant ITERATING of type <code>int</code>

Attributes Documentation

CONVERGED_ITERATING: `int` = CONVERGED_ITERATING

Constant CONVERGED_ITERATING of type `int`

CONVERGED_MAXIT: `int` = CONVERGED_MAXIT

Constant CONVERGED_MAXIT of type `int`

CONVERGED_TOL: `int` = CONVERGED_TOL

Constant CONVERGED_TOL of type `int`

CONVERGED_USER: `int` = CONVERGED_USER

Constant CONVERGED_USER of type `int`

DIVERGED_BREAKDOWN: `int` = DIVERGED_BREAKDOWN

Constant DIVERGED_BREAKDOWN of type `int`

```
DIVERGED_ITS: int = DIVERGED_ITS
    Constant DIVERGED_ITS of type int

DIVERGED_SYMMETRY_LOST: int = DIVERGED_SYMMETRY_LOST
    Constant DIVERGED_SYMMETRY_LOST of type int

ITERATING: int = ITERATING
    Constant ITERATING of type int

__init__()

classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.SVD.ErrorType

```
class slepc4py.SLEPc.SVD.ErrorType
    Bases: object

    SVD error type to assess accuracy of computed solutions.

    • ABSOLUTE: Absolute error.
    • RELATIVE: Relative error.
    • NORM: Error relative to the matrix norm.
```

See also

SVDErrorType

Attributes Summary

ABSOLUTE	Constant ABSOLUTE of type int
NORM	Constant NORM of type int
RELATIVE	Constant RELATIVE of type int

Attributes Documentation

```
ABSOLUTE: int = ABSOLUTE
    Constant ABSOLUTE of type int

NORM: int = NORM
    Constant NORM of type int

RELATIVE: int = RELATIVE
    Constant RELATIVE of type int

__init__()

classmethod __new__(*args, **kwargs)
```


slepc4py.SLEPc.SVD.ProblemType

class slepc4py.SLEPc.SVD.ProblemType

Bases: `object`

SVD problem type.

- *STANDARD*: Standard SVD.
- *GENERALIZED*: Generalized singular value decomposition (GSVD).
- *HYPERBOLIC* : Hyperbolic singular value decomposition (HSVD).

See also

`SVDProblemType`

Attributes Summary

<i>GENERALIZED</i>	Constant <i>GENERALIZED</i> of type <code>int</code>
<i>HYPERBOLIC</i>	Constant <i>HYPERBOLIC</i> of type <code>int</code>
<i>STANDARD</i>	Constant <i>STANDARD</i> of type <code>int</code>

Attributes Documentation

GENERALIZED: `int` = *GENERALIZED*

Constant *GENERALIZED* of type `int`

HYPERBOLIC: `int` = *HYPERBOLIC*

Constant *HYPERBOLIC* of type `int`

STANDARD: `int` = *STANDARD*

Constant *STANDARD* of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

slepc4py.SLEPc.SVD.Stop

class slepc4py.SLEPc.SVD.Stop

Bases: `object`

SVD stopping test.

- *BASIC*: Default stopping test.
- *USER*: User-defined stopping test.
- *THRESHOLD*: Threshold stopping test.

See also

`SVDDStop`

Attributes Summary

<i>BASIC</i>	Constant BASIC of type <code>int</code>
<i>THRESHOLD</i>	Constant THRESHOLD of type <code>int</code>
<i>USER</i>	Constant USER of type <code>int</code>

Attributes Documentation

BASIC: `int` = BASIC

Constant BASIC of type `int`

THRESHOLD: `int` = THRESHOLD

Constant THRESHOLD of type `int`

USER: `int` = USER

Constant USER of type `int`

`__init__()`

`classmethod __new__(*args, **kwargs)`

`slepc4py.SLEPc.SVD.TRLanczosGBidiag`

class `slepc4py.SLEPc.SVD.TRLanczosGBidiag`

Bases: `object`

SVD TRLanczos bidiagonalization choices for the GSVD case.

- *SINGLE*: Single bidiagonalization (Q_A).
- *UPPER*: Joint bidiagonalization, both Q_A and Q_B in upper bidiagonal form.
- *LOWER*: Joint bidiagonalization, Q_A lower bidiagonal, Q_B upper bidiagonal.

See also

`SVDTRLanczosGBidiag`

Attributes Summary

<i>LOWER</i>	Constant LOWER of type <code>int</code>
<i>SINGLE</i>	Constant SINGLE of type <code>int</code>
<i>UPPER</i>	Constant UPPER of type <code>int</code>

Attributes Documentation

LOWER: `int` = LOWER

Constant LOWER of type `int`

SINGLE: `int` = SINGLE

Constant SINGLE of type `int`

UPPER: `int` = UPPER

Constant UPPER of type `int`

```
__init__()  
  
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.SVD.Type

class slepc4py.SLEPc.SVD.Type

Bases: `object`

SVD type.

Native singular value solvers.

- *CROSS*: Eigenproblem with the cross-product matrix.
- *CYCLIC*: Eigenproblem with the cyclic matrix.
- *LANCZOS*: Explicitly restarted Lanczos.
- *TRLANCZOS*: Thick-restart Lanczos.
- *RANDOMIZED*: Iterative RSVD for low-rank matrices.

Wrappers to external SVD solvers (should be enabled during installation of SLEPc).

- *LAPACK*: Sequential dense SVD solver.
- *SCALAPACK*: Parallel dense SVD solver.
- *KSVD*: Parallel dense SVD solver.
- *ELEMENTAL*: Parallel dense SVD solver.
- *PRIMME*: Iterative SVD solvers of Davidson type.

See also

SVDType

Attributes Summary

<i>CROSS</i>	Object CROSS of type <code>str</code>
<i>CYCLIC</i>	Object CYCLIC of type <code>str</code>
<i>ELEMENTAL</i>	Object ELEMENTAL of type <code>str</code>
<i>KSVD</i>	Object KSVD of type <code>str</code>
<i>LANCZOS</i>	Object LANCZOS of type <code>str</code>
<i>LAPACK</i>	Object LAPACK of type <code>str</code>
<i>PRIMME</i>	Object PRIMME of type <code>str</code>
<i>RANDOMIZED</i>	Object RANDOMIZED of type <code>str</code>
<i>SCALAPACK</i>	Object SCALAPACK of type <code>str</code>
<i>TRLANCZOS</i>	Object TRLANCZOS of type <code>str</code>

Attributes Documentation

CROSS: `str` = CROSS
Object CROSS of type `str`

```
CYCLIC: str = CYCLIC
    Object CYCLIC of type str
ELEMENTAL: str = ELEMENTAL
    Object ELEMENTAL of type str
KSVD: str = KSVD
    Object KSVD of type str
LANCZOS: str = LANCZOS
    Object LANCZOS of type str
LAPACK: str = LAPACK
    Object LAPACK of type str
PRIMME: str = PRIMME
    Object PRIMME of type str
RANDOMIZED: str = RANDOMIZED
    Object RANDOMIZED of type str
SCALAPACK: str = SCALAPACK
    Object SCALAPACK of type str
TRLANCZOS: str = TRLANCZOS
    Object TRLANCZOS of type str
__init__()
classmethod __new__(*args, **kwargs)
```

slepc4py.SLEPc.SVD.Which

class slepc4py.SLEPc.SVD.Which

Bases: `object`

SVD desired part of spectrum.

- *LARGEST*: Largest singular values.
- *SMALLEST*: Smallest singular values.

See also

SVDWhich

Attributes Summary

<i>LARGEST</i>	Constant LARGEST of type <code>int</code>
<i>SMALLEST</i>	Constant SMALLEST of type <code>int</code>

Attributes Documentation

LARGEST: `int` = LARGEST
Constant LARGEST of type `int`

SMALLEST: `int` = **SMALLEST**

Constant **SMALLEST** of type `int`

`__init__()`

classmethod `__new__(*args, **kwargs)`

Methods Summary

<code>appendOptionsPrefix([prefix])</code>	Append to the prefix used for searching for all SVD options in the database.
<code>cancelMonitor()</code>	Clear all monitors for an SVD object.
<code>computeError(i[, etype])</code>	Compute the error associated with the i-th singular triplet.
<code>create([comm])</code>	Create the SVD object.
<code>destroy()</code>	Destroy the SVD object.
<code>errorView([etype, viewer])</code>	Display the errors associated with the computed solution.
<code>getBV()</code>	Get the basis vectors objects associated to the SVD object.
<code>getConverged()</code>	Get the number of converged singular triplets.
<code>getConvergedReason()</code>	Get the reason why the solve() iteration was stopped.
<code>getConvergenceTest()</code>	Get the method used to compute the error estimate used in the convergence test.
<code>getCrossEPS()</code>	Get the eigensolver object associated to the singular value solver.
<code>getCrossExplicitMatrix()</code>	Get the flag indicating if A^*A is built explicitly.
<code>getCyclicEPS()</code>	Get the eigensolver object associated to the singular value solver.
<code>getCyclicExplicitMatrix()</code>	Get the flag indicating if $H(A)$ is built explicitly.
<code>getDS()</code>	Get the direct solver associated to the singular value solver.
<code>getDimensions()</code>	Get the number of singular values to compute and the dimension of the subspace.
<code>getImplicitTranspose()</code>	Get the mode used to handle the transpose of the associated matrix.
<code>getIterationNumber()</code>	Get the current iteration number.
<code>getLanczosOneSide()</code>	Get if the variant of the Lanczos method to be used is one-sided or two-sided.
<code>getMonitor()</code>	Get the list of monitor functions.
<code>getOperators()</code>	Get the matrices associated with the singular value problem.
<code>getOptionsPrefix()</code>	Get the prefix used for searching for all SVD options in the database.
<code>getProblemType()</code>	Get the problem type from the SVD object.
<code>getSignature([omega])</code>	Get the signature matrix defining a hyperbolic singular value problem.
<code>getSingularTriplet(i[, U, V])</code>	Get the i-th triplet of the singular value decomposition.
<code>getStoppingTest()</code>	Get the stopping test function.
<code>getTRLanczosExplicitMatrix()</code>	Get the flag indicating if $Z = [A^*, B^*]^*$ is built explicitly.

continues on next page

Table 106 – continued from previous page

<code>getTRLanczosGBidiag()</code>	Get bidiagonalization choice used in the GSVD TR-Lanczos solver.
<code>getTRLanczosKSP()</code>	Get the linear solver object associated with the SVD solver.
<code>getTRLanczosLocking()</code>	Get the locking flag used in the thick-restart Lanczos method.
<code>getTRLanczosOneSide()</code>	Get if the variant of the method to be used is one-sided or two-sided.
<code>getTRLanczosRestart()</code>	Get the restart parameter used in the thick-restart Lanczos method.
<code>getThreshold()</code>	Get the threshold used in the threshold stopping test.
<code>getTolerances()</code>	Get the tolerance and maximum iteration count.
<code>getTrackAll()</code>	Get the flag indicating if all residual norms must be computed or not.
<code>getType()</code>	Get the SVD type of this object.
<code>getValue(i)</code>	Get the i-th singular value as computed by <code>solve()</code> .
<code>getVectors(i, U, V)</code>	Get the i-th left and right singular vectors as computed by <code>solve()</code> .
<code>getWhichSingularTriplets()</code>	Get which singular triplets are to be sought.
<code>isGeneralized()</code>	Tell if the SVD corresponds to a generalized singular value problem.
<code>isHyperbolic()</code>	Tell whether the SVD object corresponds to a hyperbolic singular value problem.
<code>reset()</code>	Reset the SVD object.
<code>setBV(V[, U])</code>	Set basis vectors objects associated to the SVD solver.
<code>setConvergenceTest(conv)</code>	Set how to compute the error estimate used in the convergence test.
<code>setCrossEPS(eps)</code>	Set an eigensolver object associated to the singular value solver.
<code>setCrossExplicitMatrix([flag])</code>	Set if the eigensolver operator A^*A must be computed.
<code>setCyclicEPS(eps)</code>	Set an eigensolver object associated to the singular value solver.
<code>setCyclicExplicitMatrix([flag])</code>	Set if the eigensolver operator $H(A)$ must be computed explicitly.
<code>setDS(ds)</code>	Set a direct solver object associated to the singular value solver.
<code>setDimensions([nsv, ncv, mpd])</code>	Set the number of singular values to compute and the dimension of the subspace.
<code>setFromOptions()</code>	Set SVD options from the options database.
<code>setImplicitTranspose(mode)</code>	Set how to handle the transpose of the associated matrix.
<code>setInitialSpace([spaceright, spaceleft])</code>	Set the initial spaces from which the SVD solver starts to iterate.
<code>setLanczosOneSide([flag])</code>	Set if the variant of the Lanczos method to be used is one-sided or two-sided.
<code>setMonitor(monitor[, args, kargs])</code>	Append a monitor function to the list of monitors.
<code>setOperators(A[, B])</code>	Set the matrices associated with the singular value problem.
<code>setOptionsPrefix([prefix])</code>	Set the prefix used for searching for all SVD options in the database.
<code>setProblemType(problem_type)</code>	Set the type of the singular value problem.

continues on next page

Table 106 – continued from previous page

<code>setSignature([omega])</code>	Set the signature matrix defining a hyperbolic singular value problem.
<code>setStoppingTest(stopping[, args, kargs])</code>	Set a function to decide when to stop the outer iteration of the eigensolver.
<code>setTRLanczosExplicitMatrix([flag])</code>	Set if the matrix $Z = [A^*, B^*]^*$ must be built explicitly.
<code>setTRLanczosGBidiag(bidiag)</code>	Set the bidiagonalization choice to use in the GSVD TRLanczos solver.
<code>setTRLanczosKSP(ksp)</code>	Set a linear solver object associated to the SVD solver.
<code>setTRLanczosLocking(lock)</code>	Toggle between locking and non-locking variants of TRLanczos.
<code>setTRLanczosOneSide([flag])</code>	Set if the variant of the method to be used is one-sided or two-sided.
<code>setTRLanczosRestart(keep)</code>	Set the restart parameter for the thick-restart Lanczos method.
<code>setThreshold(thres[, rel])</code>	Set the threshold used in the threshold stopping test.
<code>setTolerances([tol, max_it])</code>	Set the tolerance and maximum iteration count used.
<code>setTrackAll(trackall)</code>	Set flag to compute the residual of all singular triplets.
<code>setType(svd_type)</code>	Set the particular solver to be used in the SVD object.
<code>setUp()</code>	Set up all the internal data structures.
<code>setWhichSingularTriplets(which)</code>	Set which singular triplets are to be sought.
<code>solve()</code>	Solve the singular value problem.
<code>valuesView([viewer])</code>	Display the computed singular values in a viewer.
<code>vectorsView([viewer])</code>	Output computed singular vectors to a viewer.
<code>view([viewer])</code>	Print the SVD data structure.

Attributes Summary

<code>ds</code>	The direct solver (<i>DS</i>) object associated.
<code>max_it</code>	The maximum iteration count.
<code>problem_type</code>	The type of the eigenvalue problem.
<code>tol</code>	The tolerance.
<code>track_all</code>	Compute the residual norm of all approximate eigenpairs.
<code>transpose_mode</code>	How to handle the transpose of the matrix.
<code>which</code>	The portion of the spectrum to be sought.

Methods Documentation

`appendOptionsPrefix(prefix=None)`

Append to the prefix used for searching for all SVD options in the database.

Logically collective.

Parameters

prefix (*str* / *None*) – The prefix string to prepend to all SVD option requests.

Return type

None

See also

[setOptionsPrefix](#), [getOptionsPrefix](#), [SVDAppendOptionsPrefix](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:357 <slepc4py/SLEPc/SVD.pyx#L357>``

`cancelMonitor()`

Clear all monitors for an *SVD* object.

Logically collective.

See also

[SVDMonitorCancel](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1149 <slepc4py/SLEPc/SVD.pyx#L1149>``

Return type

`None`

`computeError(i, etype=None)`

Compute the error associated with the *i*-th singular triplet.

Collective.

Compute the error (based on the residual norm) associated with the *i*-th singular triplet.

Parameters

- **i** (`int`) – Index of the solution to be considered.
- **etype** (`ErrorType` / `None`) – The error type to compute.

Returns

The error bound, computed in various ways from the residual norm $\sqrt{\eta_1^2 + \eta_2^2}$ where $\eta_1 = \|Av - \sigma u\|_2$, $\eta_2 = \|A^*u - \sigma v\|_2$, σ is the approximate singular value, *u* and *v* are the left and right singular vectors.

Return type

`float`

Notes

The index *i* should be a value between 0 and `nconv-1` (see [getConverged\(\)](#)).

In the case of the GSVD, the two components of the residual norm are $\eta_1 = \|s^2 A^*u - cB^*Bx\|_2$ and $\eta_2 = \|c^2 B^*v - sA^*Ax\|_2$, where (σ, u, v, x) is the approximate generalized singular quadruple, with $\sigma = c/s$.

See also

[solve](#), [SVDComputeError](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1377 <slepc4py/SLEPc/SVD.pyx#L1377>``

create(*comm=None*)

Create the SVD object.

Collective.

Parameters

comm (*Comm* / *None*) – MPI communicator; if not provided, it defaults to all processes.

Return type

Self

See also

SVDCreate

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:240 <slepc4py/SLEPc/SVD.pyx#L240>](#)

destroy()

Destroy the SVD object.

Collective.

See also

SVDDestroy

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:214 <slepc4py/SLEPc/SVD.pyx#L214>](#)

Return type

Self

errorView(*etype=None, viewer=None*)

Display the errors associated with the computed solution.

Collective.

Display the errors and the singular values.

Parameters

- **etype** (*ErrorType* / *None*) – The error type to compute.
- **viewer** (*petsc4py.PETSc.Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

Notes

By default, this function checks the error of all singular triplets and prints the singular values if all of them are below the requested tolerance. If the viewer has format ASCII_INFO_DETAIL then a table with singular values and corresponding errors is printed.

See also

[*solve*](#), [*valuesView*](#), [*vectorsView*](#), [*SVDErrorView*](#)

:sources: [`Source code at slepc4py/SLEPc/SVD.pyx:1424 <slepc4py/SLEPc/SVD.pyx#L1424>`](#)

getBV()

Get the basis vectors objects associated to the SVD object.

Not collective.

Returns

- **V** (*BV*) – The basis vectors context for right singular vectors.
- **U** (*BV*) – The basis vectors context for left singular vectors.

Return type

`tuple[BV, BV]`

See also

[*setBV*](#), [*SVDGetBV*](#)

:sources: [`Source code at slepc4py/SLEPc/SVD.pyx:829 <slepc4py/SLEPc/SVD.pyx#L829>`](#)

getConverged()

Get the number of converged singular triplets.

Not collective.

Returns

nconv – Number of converged singular triplets.

Return type

`int`

Notes

This function should be called after [*solve\(\)*](#) has finished.

The value `nconv` may be different from the number of requested solutions `nsv`, but not larger than `ncv`, see [*setDimensions\(\)*](#).

See also

[*setDimensions*](#), [*solve*](#), [*getValue*](#), [*SVDGetConverged*](#)

:sources: [`Source code at slepc4py/SLEPc/SVD.pyx:1248 <slepc4py/SLEPc/SVD.pyx#L1248>`](#)

getConvergedReason()

Get the reason why the [*solve\(\)*](#) iteration was stopped.

Not collective.

Returns

Negative value indicates diverged, positive value converged.

Return type
ConvergedReason

See also

setTolerances, *solve*, *SVDGetConvergedReason*

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1229 <slepc4py/SLEPc/SVD.pyx#L1229>``

getConvergenceTest()

Get the method used to compute the error estimate used in the convergence test.

Not collective.

Returns
The method used to compute the error estimate used in the convergence test.

Return type
Conv

See also

setConvergenceTest, *SVDGetConvergenceTest*

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:674 <slepc4py/SLEPc/SVD.pyx#L674>``

getCrossEPS()

Get the eigensolver object associated to the singular value solver.

Collective.

Returns
The eigensolver object.

Return type
EPS

See also

setCrossEPS, *SVDCrossGetEPS*

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1513 <slepc4py/SLEPc/SVD.pyx#L1513>``

getCrossExplicitMatrix()

Get the flag indicating if A^*A is built explicitly.

Not collective.

Returns
True if A^*A is built explicitly.

Return type
bool

See also

`setCrossExplicitMatrix`, `SVDCrossGetExplicitMatrix`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1560 <slepc4py/SLEPc/SVD.pyx#L1560>`

getCyclicEPS()

Get the eigensolver object associated to the singular value solver.

Collective.

Returns

The eigensolver object.

Return type

EPS

See also

`setCyclicEPS`, `SVDCyclicGetEPS`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1596 <slepc4py/SLEPc/SVD.pyx#L1596>`

getCyclicExplicitMatrix()

Get the flag indicating if $H(A)$ is built explicitly.

Not collective.

Get the flag indicating if $H(A) = [0 \ A; A^T \ 0]$ is built explicitly.

Returns

True if $H(A)$ is built explicitly.

Return type

`bool`

See also

`setCyclicExplicitMatrix`, `SVDCyclicGetExplicitMatrix`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1645 <slepc4py/SLEPc/SVD.pyx#L1645>`

getDS()

Get the direct solver associated to the singular value solver.

Not collective.

Returns

The direct solver context.

Return type

DS

See also

[*setDS*](#), [*SVDGetDS*](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:874 <slepc4py/SLEPc/SVD.pyx#L874>](#)

getDimensions()

Get the number of singular values to compute and the dimension of the subspace.

Not collective.

Returns

- **nsv** (*int*) – Number of singular values to compute.
- **ncv** (*int*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int*) – Maximum dimension allowed for the projected problem.

Return type

tuple[*int*, *int*, *int*]

See also

[*setDimensions*](#), [*SVDGetDimensions*](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:753 <slepc4py/SLEPc/SVD.pyx#L753>](#)

getImplicitTranspose()

Get the mode used to handle the transpose of the associated matrix.

Not collective.

Returns

How to handle the transpose (implicitly or not).

Return type

bool

See also

[*setImplicitTranspose*](#), [*SVDGetImplicitTranspose*](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:478 <slepc4py/SLEPc/SVD.pyx#L478>](#)

getIterationNumber()

Get the current iteration number.

Not collective.

If the call to [*solve\(\)*](#) is complete, then it returns the number of iterations carried out by the solution method.

Returns

Iteration number.

Return type

int

See also

getConvergedReason, *setTolerances*, *SVDGetIterationNumber*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1207 <slepc4py/SLEPc/SVD.pyx#L1207>`

getLanczosOneSide()

Get if the variant of the Lanczos method to be used is one-sided or two-sided.

Not collective.

Returns

True if the method is one-sided.

Return type

`bool`

See also

setLanczosOneSide, *SVDLanczosGetOneSide*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1693 <slepc4py/SLEPc/SVD.pyx#L1693>`

getMonitor()

Get the list of monitor functions.

Not collective.

Returns

The list of monitor functions.

Return type

SVDMonitorFunction

See also

setMonitor

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1132 <slepc4py/SLEPc/SVD.pyx#L1132>`

getOperators()

Get the matrices associated with the singular value problem.

Collective.

Returns

- **A** (`petsc4py.PETSc.Mat`) – The matrix associated with the singular value problem.
- **B** (`petsc4py.PETSc.Mat`) – The second matrix in the case of GSVD.

Return type

`tuple[Mat, Mat] | tuple[Mat, None]`

See also

[setOperators](#), [SVDGetOperators](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:911 <slepc4py/SLEPc/SVD.pyx#L911>``

getOptionsPrefix()

Get the prefix used for searching for all SVD options in the database.

Not collective.

Returns

The prefix string set for this SVD object.

Return type

`str`

See also

[setOptionsPrefix](#), [appendOptionsPrefix](#), [SVDGetOptionsPrefix](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:307 <slepc4py/SLEPc/SVD.pyx#L307>``

getProblemType()

Get the problem type from the SVD object.

Not collective.

Returns

The problem type that was previously set.

Return type

`ProblemType`

See also

[setProblemType](#), [SVDGetProblemType](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:395 <slepc4py/SLEPc/SVD.pyx#L395>``

getSignature(*omega*=None)

Get the signature matrix defining a hyperbolic singular value problem.

Collective.

Parameters

omega (`Vec` / `None`) – Optional vector to store the diagonal elements of the signature matrix.

Returns

A vector containing the diagonal elements of the signature matrix.

Return type

`petsc4py.PETSc.Vec`

See also

[setSignature](#), [SVDGetSignature](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:958](#) <[slepc4py/SLEPc/SVD.pyx#L958](#)>

getSingularTriplet(*i*, *U=None*, *V=None*)

Get the *i*-th triplet of the singular value decomposition.

Collective.

Get the *i*-th triplet of the singular value decomposition as computed by [solve\(\)](#). The solution consists of the singular value and its left and right singular vectors.

Parameters

- **i** (*int*) – Index of the solution to be obtained.
- **U** (*Vec* | *None*) – Placeholder for the returned left singular vector.
- **V** (*Vec* | *None*) – Placeholder for the returned right singular vector.

Returns

The computed singular value.

Return type

[float](#)

Notes

The index *i* should be a value between 0 and `nconv-1` (see [getConverged\(\)](#). Singular triplets are indexed according to the ordering criterion established with [setWhichSingularTriplets\(\)](#).

See also

[getConverged](#), [setWhichSingularTriplets](#), [SVDGetSingularTriplet](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1334](#) <[slepc4py/SLEPc/SVD.pyx#L1334](#)>

getStoppingTest()

Get the stopping test function.

Not collective.

Returns

The stopping test function.

Return type

[SVDDStoppingFunction](#)

See also

[setStoppingTest](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1088](#) <[slepc4py/SLEPc/SVD.pyx#L1088](#)>

getTRLanczosExplicitMatrix()

Get the flag indicating if $Z = [A^*, B^*]^*$ is built explicitly.

Not collective.

Returns

True if $Z = [A^*, B^*]^*$ is built explicitly.

Return type

`bool`

See also

`setTRLanczosExplicitMatrix`, `SVDTRLanczosGetExplicitMatrix`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1948 <slepc4py/SLEPc/SVD.pyx#L1948>`

getTRLanczosGBidiag()

Get bidiagonalization choice used in the GSVD TRLanczos solver.

Not collective.

Returns

The bidiagonalization choice.

Return type

`TRLanczosGBidiag`

See also

`setTRLanczosGBidiag`, `SVDTRLanczosGetGBidiag`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1780 <slepc4py/SLEPc/SVD.pyx#L1780>`

getTRLanczosKSP()

Get the linear solver object associated with the SVD solver.

Collective.

Returns

The linear solver object.

Return type

`petsc4py.PETSc.KSP`

See also

`setTRLanczosKSP`, `SVDTRLanczosGetKSP`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1905 <slepc4py/SLEPc/SVD.pyx#L1905>`

getTRLanczosLocking()

Get the locking flag used in the thick-restart Lanczos method.

Not collective.

Returns

The locking flag.

Return type

`bool`

See also

`setTRLanczosLocking`, `SVDTRLanczosGetLocking`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1869 <slepc4py/SLEPc/SVD.pyx#L1869>`

getTRLanczosOneSide()

Get if the variant of the method to be used is one-sided or two-sided.

Not collective.

Get if the variant of the thick-restart Lanczos method to be used is one-sided or two-sided.

Returns

True if the method is one-sided.

Return type

`bool`

See also

`setTRLanczosOneSide`, `SVDLanczosGetOneSide`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1740 <slepc4py/SLEPc/SVD.pyx#L1740>`

getTRLanczosRestart()

Get the restart parameter used in the thick-restart Lanczos method.

Not collective.

Returns

The number of vectors to be kept at restart.

Return type

`float`

See also

`setTRLanczosRestart`, `SVDTRLanczosGetRestart`

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1825 <slepc4py/SLEPc/SVD.pyx#L1825>`

getThreshold()

Get the threshold used in the threshold stopping test.

Not collective.

Returns

- **thres** (`float`) – The threshold.
- **rel** (`bool`) – Whether the threshold is relative or not.

Return type
tuple[float, bool]

See also

setThreshold, *SVDGetThreshold*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:561 <slepc4py/SLEPc/SVD.pyx#L561>`

getTolerances()

Get the tolerance and maximum iteration count.

Not collective.

Get the tolerance and maximum iteration count used by the default SVD convergence tests.

Returns

- **tol** (float) – The convergence tolerance.
- **max_it** (int) – The maximum number of iterations.

Return type
tuple[float, int]

See also

setTolerances, *SVDGetTolerances*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:618 <slepc4py/SLEPc/SVD.pyx#L618>`

getTrackAll()

Get the flag indicating if all residual norms must be computed or not.

Not collective.

Returns

Whether the solver computes all residuals or not.

Return type
bool

See also

setTrackAll, *SVDGetTrackAll*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:713 <slepc4py/SLEPc/SVD.pyx#L713>`

getType()

Get the SVD type of this object.

Not collective.

Returns

The solver currently being used.

Return type
str

See also

[setType](#), [SVDGetType](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:288](#) <[slepc4py/SLEPc/SVD.pyx#L288](#)>

getValue(*i*)

Get the *i*-th singular value as computed by [solve\(\)](#).

Collective.

Parameters

i ([int](#)) – Index of the solution to be obtained.

Returns

The computed singular value.

Return type

[float](#)

Notes

The index *i* should be a value between 0 and `nconv-1` (see [getConverged\(\)](#)). Singular triplets are indexed according to the ordering criterion established with [setWhichSingularTriplets\(\)](#).

See also

[getConverged](#), [setWhichSingularTriplets](#), [SVDGetSingularTriplet](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1274](#) <[slepc4py/SLEPc/SVD.pyx#L1274](#)>

getVectors(*i*, *U*, *V*)

Get the *i*-th left and right singular vectors as computed by [solve\(\)](#).

Collective.

Parameters

- **i** ([int](#)) – Index of the solution to be obtained.
- **U** ([Vec](#)) – Placeholder for the returned left singular vector.
- **V** ([Vec](#)) – Placeholder for the returned right singular vector.

Return type

[None](#)

Notes

The index *i* should be a value between 0 and `nconv-1` (see [getConverged\(\)](#)). Singular triplets are indexed according to the ordering criterion established with [setWhichSingularTriplets\(\)](#).

See also

[getConverged](#), [setWhichSingularTriplets](#), [SVDGetSingularTriplet](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1305](#) <[slepc4py/SLEPc/SVD.pyx#L1305](#)>

getWhichSingularTriplets()

Get which singular triplets are to be sought.

Not collective.

Returns

The singular values to be sought (either largest or smallest).

Return type

Which

See also

setWhichSingularTriplets, *SVDGetWhichSingularTriplets*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:524 <slepc4py/SLEPc/SVD.pyx#L524>`

isGeneralized()

Tell if the SVD corresponds to a generalized singular value problem.

Not collective.

Returns

True if two matrices were set with *setOperators()*.

Return type

bool

See also

setProblemType, *isHyperbolic*, *SVDIsGeneralized*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:438 <slepc4py/SLEPc/SVD.pyx#L438>`

isHyperbolic()

Tell whether the SVD object corresponds to a hyperbolic singular value problem.

Not collective.

Returns

True if the problem was specified as hyperbolic.

Return type

bool

See also

setProblemType, *isGeneralized*, *SVDIsHyperbolic*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:457 <slepc4py/SLEPc/SVD.pyx#L457>`

reset()

Reset the SVD object.

Collective.

See also
SVDReset

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:228 <slepc4py/SLEPc/SVD.pyx#L228>](#)

Return type

None

setBV(*V*, *U=None*)

Set basis vectors objects associated to the SVD solver.

Collective.

Parameters

- **V** ([BV](#)) – The basis vectors context for right singular vectors.
- **U** ([BV](#) / [None](#)) – The basis vectors context for left singular vectors.

Return type

None

See also
getBV , SVDSetBV

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:853 <slepc4py/SLEPc/SVD.pyx#L853>](#)

setConvergenceTest(*conv*)

Set how to compute the error estimate used in the convergence test.

Logically collective.

Parameters

- **conv** ([Conv](#)) – The method used to compute the error estimate used in the convergence test.

Return type

None

See also
getConvergenceTest , SVDSetConvergenceTest

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:694 <slepc4py/SLEPc/SVD.pyx#L694>](#)

setCrossEPS(*eps*)

Set an eigensolver object associated to the singular value solver.

Collective.

Parameters

- **eps** ([EPS](#)) – The eigensolver object.

Return type

None

See also

[`getCrossEPS`](#), [`SVDCrossSetEPS`](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1496](#) <[slepc4py/SLEPc/SVD.pyx#L1496](#)>

setCrossExplicitMatrix(*flag=True*)

Set if the eigensolver operator A^*A must be computed.

Logically collective.

Parameters

flag (*bool*) – True to build A^*A explicitly.

Return type

None

Notes

In GSVD there are two cross product matrices, A^*A and B^*B . In HSVD the expression for the cross product matrix is different, $A^*\Omega A$.

By default the matrices are not built explicitly, but handled as shell matrices

See also

[`getCrossExplicitMatrix`](#), [`SVDCrossSetExplicitMatrix`](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1533](#) <[slepc4py/SLEPc/SVD.pyx#L1533](#)>

setCyclicEPS(*eps*)

Set an eigensolver object associated to the singular value solver.

Collective.

Parameters

eps (*EPS*) – The eigensolver object.

Return type

None

See also

[`getCyclicEPS`](#), [`SVDCyclicSetEPS`](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1579](#) <[slepc4py/SLEPc/SVD.pyx#L1579](#)>

setCyclicExplicitMatrix(*flag=True*)

Set if the eigensolver operator $H(A)$ must be computed explicitly.

Logically collective.

Set if the eigensolver operator $H(A) = [0 \ A; A^T \ 0]$ must be computed explicitly.

Parameters

flag (*bool*) – True if $H(A)$ must be built explicitly.

Return type

None

Notes

In GSVD and HSVD the equivalent eigenvalue problem has generalized form, and hence two matrices are built.

By default the matrices are not built explicitly, but handled as shell matrices.

See also

getCyclicExplicitMatrix, *SVDCyclicSetExplicitMatrix*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1616 <slepc4py/SLEPc/SVD.pyx#L1616>`

setDS(ds)

Set a direct solver object associated to the singular value solver.

Collective.

Parameters

ds (DS) – The direct solver context.

Return type

None

See also

getDS, *SVDSetDS*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:894 <slepc4py/SLEPc/SVD.pyx#L894>`

setDimensions(nsv=None, ncv=None, mpd=None)

Set the number of singular values to compute and the dimension of the subspace.

Logically collective.

Parameters

- **nsv** (*int* / *None*) – Number of singular values to compute.
- **ncv** (*int* / *None*) – Maximum dimension of the subspace to be used by the solver.
- **mpd** (*int* / *None*) – Maximum dimension allowed for the projected problem.

Return type

None

Notes

Use *DETERMINE* for ncv and mpd to assign a reasonably good value, which is dependent on the solution method.

The parameters ncv and mpd are intimately related, so that the user is advised to set one of them at most. Normal usage is the following:

- In cases where nsv is small, the user sets ncv (a reasonable default is $2 * nsv$).
- In cases where nsv is large, the user sets mpd.

The value of `ncv` should always be between `nsv` and `(nsv + mpd)`, typically `ncv = nsv + mpd`. If `nsv` is not too large, `mpd = nsv` is a reasonable choice, otherwise a smaller value should be used.

See also

[`getDimensions`](#), [`SVDSetDimensions`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:778 <slepc4py/SLEPc/SVD.pyx#L778>`

setFromOptions()

Set SVD options from the options database.

Collective.

Notes

To see all options, run your program with the `-help` option.

This routine must be called before [`setUp\(\)`](#) if the user is to be allowed to set the solver type.

See also

[`setOptionsPrefix`](#), [`SVDSetFromOptions`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:376 <slepc4py/SLEPc/SVD.pyx#L376>`

Return type

`None`

setImplicitTranspose(mode)

Set how to handle the transpose of the associated matrix.

Logically collective.

Parameters

- **impl** – How to handle the transpose (implicitly or not).
- **mode** (`bool`)

Return type

`None`

Notes

By default, the transpose of the matrix is explicitly built (if the matrix has defined the `Mat.transpose()` operation).

If this flag is set to `True`, the solver does not build the transpose, but handles it implicitly via `Mat.multTranspose()` (or `Mat.multHermitianTranspose()` in the complex case).

See also

[`getImplicitTranspose`](#), [`SVDSetImplicitTranspose`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:497 <slepc4py/SLEPc/SVD.pyx#L497>`

setInitialSpace(*spaceright=None, spaceleft=None*)

Set the initial spaces from which the SVD solver starts to iterate.

Collective.

Parameters

- **spaceright** (*list[Vec] | None*) – The right initial space.
- **spaceleft** (*list[Vec] | None*) – The left initial space.

Return type

None

Notes

The initial right and left spaces are rough approximations to the right and/or left singular subspaces from which the solver starts to iterate. It is not necessary to provide both sets of vectors.

Some solvers start to iterate on a single vector (initial vector). In that case, the other vectors are ignored.

These vectors do not persist from one *solve()* call to the other, so the initial spaces should be set every time.

The vectors do not need to be mutually orthonormal, since they are explicitly orthonormalized internally.

Common usage of this function is when the user can provide a rough approximation of the wanted singular spaces. Then, convergence may be faster.

See also

SVDSetInitialSpaces

:sources: *Source code at slepc4py/SLEPc/SVD.pyx:1007 <slepc4py/SLEPc/SVD.pyx#L1007>*

setLanczosOneSide(*flag=True*)

Set if the variant of the Lanczos method to be used is one-sided or two-sided.

Logically collective.

Parameters

- **flag** (*bool*) – True if the method is one-sided.

Return type

None

Notes

By default, a two-sided variant is selected, which is sometimes slightly more robust. However, the one-sided variant is faster because it avoids the orthogonalization associated to left singular vectors. It also saves the memory required for storing such vectors.

See also

getLanczosOneSide, SVDLanczosSetOneSide

:sources: *Source code at slepc4py/SLEPc/SVD.pyx:1667 <slepc4py/SLEPc/SVD.pyx#L1667>*

setMonitor(*monitor*, *args=None*, *kargs=None*)

Append a monitor function to the list of monitors.

Logically collective.

See also

[getMonitor](#), [cancelMonitor](#), [SVDMonitorSet](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1107](#) <[slepc4py/SLEPc/SVD.pyx#L1107](#)>

Parameters

- **monitor** ([SVDMonitorFunction](#) | *None*)
- **args** ([tuple](#)[*Any*, ...] | *None*)
- **kargs** ([dict](#)[*str*, *Any*] | *None*)

Return type

[None](#)

setOperators(*A*, *B=None*)

Set the matrices associated with the singular value problem.

Collective.

Parameters

- **A** ([Mat](#)) – The matrix associated with the singular value problem.
- **B** ([Mat](#) | *None*) – The second matrix in the case of GSVD.

Return type

[None](#)

See also

[getOperators](#), [SVDSetOperators](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:938](#) <[slepc4py/SLEPc/SVD.pyx#L938](#)>

setOptionsPrefix(*prefix=None*)

Set the prefix used for searching for all SVD options in the database.

Logically collective.

Parameters

prefix (*str* | *None*) – The prefix string to prepend to all SVD option requests.

Return type

[None](#)

Notes

A hyphen (-) must NOT be given at the beginning of the prefix name. The first character of all runtime options is AUTOMATICALLY the hyphen.

For example, to distinguish between the runtime options for two different SVD contexts, one could call:

```
S1.setOptionsPrefix("svd1_")
S2.setOptionsPrefix("svd2_")
```

See also

[appendOptionsPrefix](#), [getOptionsPrefix](#), [SVDGetOptionsPrefix](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:326 <slepc4py/SLEPc/SVD.pyx#L326>`

setProblemType(*problem_type*)

Set the type of the singular value problem.

Logically collective.

Parameters

problem_type ([ProblemType](#)) – The problem type to be set.

Return type

None

Notes

The GSVD requires that two matrices have been passed via [setOperators\(\)](#). The HSVD requires that a signature matrix has been passed via [setSignature\(\)](#).

See also

[setOperators](#), [setSignature](#), [getProblemType](#), [SVDSetProblemType](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:414 <slepc4py/SLEPc/SVD.pyx#L414>`

setSignature(*omega=None*)

Set the signature matrix defining a hyperbolic singular value problem.

Collective.

Parameters

omega ([Vec](#) / *None*) – A vector containing the diagonal elements of the signature matrix.

Return type

None

See also

[getSignature](#), [SVDSetSignature](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:987 <slepc4py/SLEPc/SVD.pyx#L987>`

setStoppingTest(*stopping*, *args=None*, *kargs=None*)

Set a function to decide when to stop the outer iteration of the eigensolver.

Logically collective.

See also

[`getStoppingTest`](#), [`SVDSetStoppingTestFunction`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1064 <slepc4py/SLEPc/SVD.pyx#L1064>`

Parameters

- **stopping** ([`SVDStoppingFunction`](#) | `None`)
- **args** ([`tuple`](#)[`Any`, ...] | `None`)
- **kargs** ([`dict`](#)[`str`, `Any`] | `None`)

Return type

`None`

setTRLanczosExplicitMatrix(*flag=True*)

Set if the matrix $Z = [A^*, B^*]^*$ must be built explicitly.

Logically collective.

Parameters

flag (`bool`) – True if $Z = [A^*, B^*]^*$ is built explicitly.

Return type

`None`

Notes

This option is relevant for the GSVD case only. Z is the coefficient matrix of the least-squares solver used internally.

See also

[`getTRLanczosExplicitMatrix`](#), [`SVDTRLanczosSetExplicitMatrix`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1925 <slepc4py/SLEPc/SVD.pyx#L1925>`

setTRLanczosGBidiag(*bidiag*)

Set the bidiagonalization choice to use in the GSVD TRLanczos solver.

Logically collective.

Parameters

bidiag ([`TRLanczosGBidiag`](#)) – The bidiagonalization choice.

Return type

`None`

See also

[`getTRLanczosGBidiag`](#), [`SVDTRLanczosSetGBidiag`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1762 <slepc4py/SLEPc/SVD.pyx#L1762>`

setTRLanczosKSP(*ksp*)

Set a linear solver object associated to the SVD solver.

Collective.

Parameters

ksp (*KSP*) – The linear solver object.

Return type

None

See also

[getTRLanczosKSP](#), [SVDTRLanczosSetKSP](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1888](#) <slepc4py/SLEPc/SVD.pyx#L1888>

setTRLanczosLocking(*lock*)

Toggle between locking and non-locking variants of TRLanczos.

Logically collective.

Parameters

lock (*bool*) – True if the locking variant must be selected.

Return type

None

Notes

The default is to lock converged singular triplets when the method restarts. This behavior can be changed so that all directions are kept in the working subspace even if already converged to working accuracy (the non-locking variant).

See also

[getTRLanczosLocking](#), [SVDTRLanczosSetLocking](#)

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1844](#) <slepc4py/SLEPc/SVD.pyx#L1844>

setTRLanczosOneSide(*flag=True*)

Set if the variant of the method to be used is one-sided or two-sided.

Logically collective.

Set if the variant of the thick-restart Lanczos method to be used is one-sided or two-sided.

Parameters

flag (*bool*) – True if the method is one-sided.

Return type

None

Notes

By default, a two-sided variant is selected, which is sometimes slightly more robust. However, the one-sided variant is faster because it avoids the orthogonalization associated to left singular vectors.

See also

[`getTRLanczosOneSide`](#), [`SVDLanczosSetOneSide`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1712 <slepc4py/SLEPc/SVD.pyx#L1712>`

`setTRLanczosRestart(keep)`

Set the restart parameter for the thick-restart Lanczos method.

Logically collective.

Set the restart parameter for the thick-restart Lanczos method, in particular the proportion of basis vectors that must be kept after restart.

Parameters

keep (*float*) – The number of vectors to be kept at restart.

Return type

None

Notes

Allowed values are in the range [0.1,0.9]. The default is 0.5.

See also

[`getTRLanczosRestart`](#), [`SVDTRLanczosSetRestart`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1799 <slepc4py/SLEPc/SVD.pyx#L1799>`

`setThreshold(thres, rel=False)`

Set the threshold used in the threshold stopping test.

Logically collective.

Parameters

- **thres** (*float*) – The threshold.
- **rel** (*bool*) – Whether the threshold is relative or not.

Return type

None

Notes

This function internally sets a special stopping test based on the threshold, where singular values are computed in sequence until one of the computed singular values is below/above the threshold (depending on whether largest or smallest singular values are computed).

In the case of largest singular values, the threshold can be made relative with respect to the largest singular value (i.e., the matrix norm).

The details are given in [`SVDSetThreshold`](#).

See also

[`setStoppingTest`](#), [`getThreshold`](#), [`SVDSetThreshold`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:583 <slepc4py/SLEPc/SVD.pyx#L583>`

setTolerances(*tol=None, max_it=None*)

Set the tolerance and maximum iteration count used.

Logically collective.

Set the tolerance and maximum iteration count used by the default SVD convergence tests.

Parameters

- **tol** (*float* / *None*) – The convergence tolerance.
- **max_it** (*int* / *None*) – The maximum number of iterations

Return type

None

Notes

Use *DETERMINE* for *max_it* to assign a reasonably good value, which is dependent on the solution method.

See also

[`getTolerances`](#), [`SVDSetTolerances`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:643 <slepc4py/SLEPc/SVD.pyx#L643>`

setTrackAll(*trackall*)

Set flag to compute the residual of all singular triplets.

Logically collective.

Set if the solver must compute the residual of all approximate singular triplets or not.

Parameters

- **trackall** (*bool*) – Whether to compute all residuals or not.

Return type

None

See also

[`getTrackAll`](#), [`SVDSetTrackAll`](#)

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:732 <slepc4py/SLEPc/SVD.pyx#L732>`

setType(*svd_type*)

Set the particular solver to be used in the SVD object.

Logically collective.

Parameters

- **svd_type** (*Type* / *str*) – The solver to be used.

Return type

None

Notes

The default is *CROSS*. Normally, it is best to use *setFromOptions()* and then set the SVD type from the options database rather than by using this routine. Using the options database provides the user with maximum flexibility in evaluating the different available methods.

See also

getType, *SVDSetType*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:261 <slepc4py/SLEPc/SVD.pyx#L261>`

setUp()

Set up all the internal data structures.

Collective.

Notes

Sets up all the internal data structures necessary for the execution of the singular value solver.

This function need not be called explicitly in most cases, since *solve()* calls it. It can be useful when one wants to measure the set-up time separately from the solve time.

See also

solve, *SVDSetUp*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:1164 <slepc4py/SLEPc/SVD.pyx#L1164>`

Return type

None

setWhichSingularTriplets(*which*)

Set which singular triplets are to be sought.

Logically collective.

Parameters

which (*Which*) – The singular values to be sought (either largest or smallest).

Return type

None

See also

getWhichSingularTriplets, *SVDSetWhichSingularTriplets*

:sources: `Source code at slepc4py/SLEPc/SVD.pyx:543 <slepc4py/SLEPc/SVD.pyx#L543>`

solve()

Solve the singular value problem.

Collective.

Notes

The problem matrices are specified with *setOperators()*.

solve() will return without generating an error regardless of whether all requested solutions were computed or not. Call *getConverged()* to get the actual number of computed solutions, and *getConvergedReason()* to determine if the solver converged or failed and why.

See also

setUp, *setOperators*, *getConverged*, *getConvergedReason*, *SVDsolve*

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1185](#) <slepc4py/SLEPc/SVD.pyx#L1185>

Return type

None

valuesView(viewer=None)

Display the computed singular values in a viewer.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

solve, *vectorsView*, *errorView*, *SVDValuesView*

:sources: [Source code at slepc4py/SLEPc/SVD.pyx:1456](#) <slepc4py/SLEPc/SVD.pyx#L1456>

vectorsView(viewer=None)

Output computed singular vectors to a viewer.

Collective.

Parameters

viewer (*Viewer* / *None*) – Visualization context; if not provided, the standard output is used.

Return type

None

See also

[solve](#), [valuesView](#), [errorView](#), [SVDVectorsView](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1475 <slepc4py/SLEPc/SVD.pyx#L1475>``

view(viewer=None)

Print the SVD data structure.

Collective.

Parameters

viewer ([Viewer](#) / `None`) – Visualization context; if not provided, the standard output is used.

Return type

`None`

See also

[SVDView](#)

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:195 <slepc4py/SLEPc/SVD.pyx#L195>``

Attributes Documentation

ds

The direct solver (*DS*) object associated.

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:2013 <slepc4py/SLEPc/SVD.pyx#L2013>``

max_it

The maximum iteration count.

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1999 <slepc4py/SLEPc/SVD.pyx#L1999>``

problem_type

The type of the eigenvalue problem.

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1971 <slepc4py/SLEPc/SVD.pyx#L1971>``

tol

The tolerance.

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1992 <slepc4py/SLEPc/SVD.pyx#L1992>``

track_all

Compute the residual norm of all approximate eigenpairs.

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:2006 <slepc4py/SLEPc/SVD.pyx#L2006>``

transpose_mode

How to handle the transpose of the matrix.

:sources: ``Source code at slepc4py/SLEPc/SVD.pyx:1978 <slepc4py/SLEPc/SVD.pyx#L1978>``

which

The portion of the spectrum to be sought.

:sources:`Source code at slepc4py/SLEPc/SVD.pyx:1985 <slepc4py/SLEPc/SVD.pyx#L1985>`

`__init__()`

`classmethod __new__(*args, **kwargs)`

slepc4py.SLEPc.Sys

class slepc4py.SLEPc.Sys

Bases: `object`

System utilities.

Methods Summary

<code>getVersion([devel, date, author])</code>	Return SLEPc version information.
<code>getVersionInfo()</code>	Return SLEPc version information.
<code>hasExternalPackage(package)</code>	Return whether SLEPc has support for external package.
<code>isFinalized()</code>	Return whether SLEPc has been finalized.
<code>isInitialized()</code>	Return whether SLEPc has been initialized.

Methods Documentation

classmethod `getVersion(devel=False, date=False, author=False)`

Return SLEPc version information.

Not collective.

Parameters

- **devel** (`bool`) – Additionally, return whether using an in-development version.
- **date** (`bool`) – Additionally, return date information.
- **author** (`bool`) – Additionally, return author information.

Returns

- **major** (`int`) – Major version number.
- **minor** (`int`) – Minor version number.
- **micro** (`int`) – Micro (or patch) version number.

Return type

`tuple[int, int, int]`

See also

`SlepcGetVersion`, `SlepcGetVersionNumber`

:sources:`Source code at slepc4py/SLEPc/Sys.pyx:8 <slepc4py/SLEPc/Sys.pyx#L8>`

classmethod getVersionInfo()

Return SLEPc version information.

Not collective.

Returns

info – Dictionary with version information.

Return type

`dict`

See also

`SlepcGetVersion`, `SlepcGetVersionNumber`

:sources: `Source code at slepc4py/SLEPc/Sys.pyx:64 <slepc4py/SLEPc/Sys.pyx#L64>`

classmethod hasExternalPackage(package)

Return whether SLEPc has support for external package.

Not collective.

Parameters

package (`str`) – The external package name.

Return type

`bool`

See also

`SlepcHasExternalPackage`

:sources: `Source code at slepc4py/SLEPc/Sys.pyx:118 <slepc4py/SLEPc/Sys.pyx#L118>`

classmethod isFinalized()

Return whether SLEPc has been finalized.

Not collective.

See also

`isInitialized`

:sources: `Source code at slepc4py/SLEPc/Sys.pyx:103 <slepc4py/SLEPc/Sys.pyx#L103>`

Return type

`bool`

classmethod isInitialized()

Return whether SLEPc has been initialized.

Not collective.

See also
<i>isFinalized</i>

:sources: [Source code at slepc4py/SLEPc/Sys.pyx:90 <slepc4py/SLEPc/Sys.pyx#L90>](#)

```

    Return type
        bool

__init__()

classmethod __new__(*args, **kwargs)

```

slepc4py.SLEPc.Util

class slepc4py.SLEPc.Util
 Bases: `object`
 Other utilities such as the creation of structured matrices.

Methods Summary

<i>createMatBSE</i> (R, C)	Create a matrix that can be used to define a BSE type problem.
<i>createMatHamiltonian</i> (A, B, C)	Create matrix to be used for a structured Hamiltonian eigenproblem.
<i>createMatLREP</i> (AK, BM[, red])	Create a matrix that can be used to define a LREP type problem.

Methods Documentation

classmethod `createMatBSE`(R, C)
 Create a matrix that can be used to define a BSE type problem.
 Collective.
 Create a matrix that can be used to define a structured eigenvalue problem of type BSE (Bethe-Salpeter Equation).

Parameters

- **R** (*petsc4py.PETSc.Mat*) – The matrix for the diagonal block (resonant).
- **C** (*petsc4py.PETSc.Mat*) – The matrix for the off-diagonal block (coupling).

Returns
 The matrix with the block form $H = \begin{bmatrix} R & C \\ -C^* & -R^T \end{bmatrix}$.

Return type
petsc4py.PETSc.Mat

See also
<i>MatCreateBSE</i>

:sources: [Source code at slepc4py/SLEPc/Util.pyx:8 <slepc4py/SLEPc/Util.pyx#L8>](#)

classmethod `createMatHamiltonian(A, B, C)`

Create matrix to be used for a structured Hamiltonian eigenproblem.

Collective.

Parameters

- **A** (`petsc4py.PETSc.Mat`) – The matrix for (0,0) block.
- **B** (`petsc4py.PETSc.Mat`) – The matrix for (0,1) block, must be real symmetric or Hermitian.
- **C** (`petsc4py.PETSc.Mat`) – The matrix for (1,0) block, must be real symmetric or Hermitian.

Returns

The matrix with the block form $H = [A \ B; C \ -A^*]$.

Return type

`petsc4py.PETSc.Mat`

See also

`MatCreateHamiltonian`

:sources: `Source code at slepc4py/SLEPc/Util.pyx:38 <slepc4py/SLEPc/Util.pyx#L38>`

classmethod `createMatLREP(AK, BM, red=False)`

Create a matrix that can be used to define a LREP type problem.

Collective.

Create a matrix that can be used to define a structured Linear Response eigenvalue problem.

Parameters

- **AK** (`petsc4py.PETSc.Mat`) – The matrix for the diagonal block or the top block.
- **BM** (`petsc4py.PETSc.Mat`) – The matrix for the off-diagonal block or the bottom block.
- **red** (`bool`) – Whether the reduced form should be built.

Returns

The matrix with the block form $H = [A \ B; -B \ -A]$ (non-reduced) or $H = [0 \ K; M \ 0]$ (reduced).

Return type

`petsc4py.PETSc.Mat`

See also

`MatCreateLREP`

:sources: `Source code at slepc4py/SLEPc/Util.pyx:67 <slepc4py/SLEPc/Util.pyx#L67>`

__init__()

classmethod `__new__(*args, **kwargs)`

Attributes

<i>DECIDE</i>	Constant DECIDE of type <code>int</code>
<i>DEFAULT</i>	Constant DEFAULT of type <code>int</code>
<i>DETERMINE</i>	Constant DETERMINE of type <code>int</code>
<i>CURRENT</i>	Constant CURRENT of type <code>int</code>

`slepc4py.SLEPc.DECIDE`

```
slepc4py.SLEPc.DECIDE: int = DECIDE
```

Constant DECIDE of type `int`

`slepc4py.SLEPc.DEFAULT`

```
slepc4py.SLEPc.DEFAULT: int = DEFAULT
```

Constant DEFAULT of type `int`

`slepc4py.SLEPc.DETERMINE`

```
slepc4py.SLEPc.DETERMINE: int = DETERMINE
```

Constant DETERMINE of type `int`

`slepc4py.SLEPc.CURRENT`

```
slepc4py.SLEPc.CURRENT: int = CURRENT
```

Constant CURRENT of type `int`

3.7 slepc4py demos

ex1.py: Standard symmetric eigenproblem for the 1-D Laplacian

This example computes eigenvalues and eigenvectors of the discrete Laplacian on a one-dimensional domain with finite differences.

The full source code for this demo can be [downloaded here](#).

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 setting 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 below 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(); A.create()
A.setSizes([n, n])
A.setFromOptions()

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)

history = []
def monitor(eig, its, nconv, err):
    if nconv < len(err): history.append(err[nconv])
E.setMonitor(monitor)

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). In this example we know that both the eigenvalue and eigenvector are real, so only one vector `v` is needed. 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
    v, _ = A.createVecs()
    #
    Print()
    Print("          k          ||Ax-kx||/||kx|| ")
    Print("-----")
    for i in range(nconv):
        k = E.getEigenpair(i, v)
        error = E.computeError(i)
        Print( " %12f          %12g" % (k, error) )
    Print()

```

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

*****

```

(continues on next page)

(continued from previous page)

```
*** 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
```

<i>k</i>	$ 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 subspace
```

All the above settings can also be changed 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 process
  type: krylovschur
    50% of basis vectors kept after restart
    using the locking variant
  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 process
  type: mat
  17 columns of global length 30
  orthogonalization method: classical Gram-Schmidt
  orthogonalization refinement: if needed (eta: 0.7071)
  block orthogonalization method: GS
  doing matmult as a single matrix-matrix product
DS Object: 1 MPI process
  type: hep
  solving the problem with: Implicit QR method (_steqr)
```

(continues on next page)

(continued from previous page)

```
ST Object: 1 MPI process
  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
```

ex2.py: Standard symmetric eigenproblem for the 2-D Laplacian

This example computes eigenvalues and eigenvectors of the discrete Laplacian on a two-dimensional domain with finite differences.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

```
try: range = xrange
except: pass

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

from petsc4py import PETSc
from slepc4py import SLEPc

Print = PETSc.Sys.Print
```

In this example we have organized the code in several functions. This one builds the finite-difference Laplacian matrix by computing the indices of each entry. An alternative would be to use the functionality offered by [DMDA](#).

```
def construct_operator(m, n):
    # Create matrix for 2D Laplacian operator
    A = PETSc.Mat().create()
    A.setSizes([m*n, m*n])
    A.setFromOptions()
    # Fill matrix
    hx = 1.0/(m-1) # x grid spacing
    hy = 1.0/(n-1) # y grid spacing
    diagv = 2.0*hy/hx + 2.0*hx/hy
    offdx = -1.0*hy/hx
    offdy = -1.0*hx/hy
    Istart, Iend = A.getOwnershipRange()
    for I in range(Istart, Iend) :
        A[I,I] = diagv
```

(continues on next page)

```

    i = I//n      # map row number to
    j = I - i*n  # grid coordinates
    if i > 0 : J = I-n; A[I,J] = offdx
    if i < m-1: J = I+n; A[I,J] = offdx
    if j > 0 : J = I-1; A[I,J] = offdy
    if j < n-1: J = I+1; A[I,J] = offdy
A.assemble()
return A

```

This function receives the matrix and the problem type, then solves the eigenvalue problem and prints information about the computed solution. Although we know that eigenvalues and eigenvectors are real in this example, the function is prepared to solve it as a non-symmetric problem, by passing `SLEPc.EPS.ProblemType.NHEP`, that is why the code handles possibly complex eigenvalues and eigenvectors.

```

def solve_eigensystem(A, problem_type=SLEPc.EPS.ProblemType.HEP):
    # Create the result vectors
    xr, xi = A.createVecs()

    # Setup the eigensolver
    E = SLEPc.EPS().create()
    E.setOperators(A, None)
    E.setDimensions(3, PETSc.DECIDE)
    E.setProblemType(problem_type)
    E.setFromOptions()

    # Solve the eigensystem
    E.solve()

    Print("")
    its = E.getIterationNumber()
    Print("Number of iterations of the method: %i" % its)
    sol_type = E.getType()
    Print("Solution method: %s" % sol_type)
    nev, ncv, mpd = E.getDimensions()
    Print("Number of requested eigenvalues: %i" % nev)
    tol, maxit = E.getTolerances()
    Print("Stopping condition: tol=%.4g, maxit=%d" % (tol, maxit))
    nconv = E.getConverged()
    Print("Number of converged eigenpairs: %d" % nconv)
    if nconv > 0:
        Print("")
        Print("          k          ||Ax-kx||/||kx|| ")
        Print("-----")
        for i in range(nconv):
            k = E.getEigenpair(i, xr, xi)
            error = E.computeError(i)
            if k.imag != 0.0:
                Print(" %9f%+9f j %12g" % (k.real, k.imag, error))
            else:
                Print(" %12f          %12g" % (k.real, error))
        Print("")

```

The main program simply processes three user-defined command-line options and calls the other two functions.

```
def main():
    opts = PETSc.Options()
    N = opts.getInt('N', 32)
    m = opts.getInt('m', N)
    n = opts.getInt('n', m)
    Print("Symmetric Eigenproblem (sparse matrix), "
          "N=%d (%dx%d grid)" % (m*n, m, n))
    A = construct_operator(m,n)
    solve_eigensystem(A)

if __name__ == '__main__':
    main()
```

ex3.py: Matrix-free eigenproblem for the 2-D Laplacian

This example solves the eigenproblem for the 2-D discrete Laplacian without building the matrix explicitly.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

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

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

In this case, the program cannot be run in parallel, so we check that the number of MPI processes is 1. In order to enable parallelism, we should implement a parallel matrix-vector operation ourselves, which is not done in this example.

```
assert PETSc.COMM_WORLD.getSize() == 1

Print = PETSc.Sys.Print
```

This function computes the matrix-vector product $f = L \cdot x$ where the Laplacian L is not built explicitly, and the vectors x, f are viewed as two-dimensional arrays associated to grid points.

```
def laplace2d(U, x, f):
    U[:, :] = 0
    U[1:-1, 1:-1] = x
    # Grid spacing
    m, n = x.shape
    hx = 1.0/(m-1) # x grid spacing
    hy = 1.0/(n-1) # y grid spacing
    # Setup 5-points stencil
    u = U[1:-1, 1:-1] # center
    uN = U[1:-1, :-2] # north
    uS = U[1:-1, 2:] # south
    uW = U[:-2, 1:-1] # west
    uE = U[2:, 1:-1] # east
    # Apply Laplacian
    f[:, :] = \
```

(continues on next page)

(continued from previous page)

```
(2*u - uE - uW) * (hy/hx) \
+ (2*u - uN - uS) * (hx/hy) \
```

For a matrix-free solution in slepc4py we have to create a class that wraps the matrix-vector operation and optionally other operations of the matrix. In this case, we provide the constructor and the `mult` operation, that simply calls the `laplace2d` function above.

```
class Laplacian2D(object):

    def __init__(self, m, n):
        self.m, self.n = m, n
        scalar = PETSc.ScalarType
        self.U = np.zeros([m+2, n+2], dtype=scalar)

    def mult(self, A, x, y):
        m, n = self.m, self.n
        xx = x.getArray(readonly=1).reshape(m,n)
        yy = y.getArray(readonly=0).reshape(m,n)
        laplace2d(self.U, xx, yy)
```

In this example, building the matrix amounts to creating an object of the class defined above, and passing it to a special petsc4py matrix with `createPython()`.

```
def construct_operator(m, n):
    # Create shell matrix
    context = Laplacian2D(m,n)
    A = PETSc.Mat().createPython([m*n,m*n], context)
    return A
```

This function receives the matrix and the problem type, then solves the eigenvalue problem and prints information about the computed solution. Although we know that eigenvalues and eigenvectors are real in this example, the function is prepared to solve it as a non-symmetric problem, by passing `SLEPc.EPS.ProblemType.NHEP`, that is why the code handles possibly complex eigenvalues and eigenvectors.

```
def solve_eigensystem(A, problem_type=SLEPc.EPS.ProblemType.HEP):
    # Create the result vectors
    xr, xi = A.createVecs()

    # Setup the eigensolver
    E = SLEPc.EPS().create()
    E.setOperators(A, None)
    E.setDimensions(3, PETSc.DECIDE)
    E.setProblemType(problem_type)
    E.setFromOptions()

    # Solve the eigensystem
    E.solve()
    Print("")
    its = E.getIterationNumber()
    Print("Number of iterations of the method: %i" % its)
    sol_type = E.getType()
    Print("Solution method: %s" % sol_type)
    nev, ncv, mpd = E.getDimensions()
```

(continues on next page)

(continued from previous page)

```
Print("Number of requested eigenvalues: %i" % nev)
tol, maxit = E.getTolerances()
Print("Stopping condition: tol=%.4g, maxit=%d" % (tol, maxit))
nconv = E.getConverged()
Print("Number of converged eigenpairs: %d" % nconv)
if nconv > 0:
    Print("")
    Print("          k          ||Ax-kx||/||kx|| ")
    Print("-----")
    for i in range(nconv):
        k = E.getEigenpair(i, xr, xi)
        error = E.computeError(i)
        if k.imag != 0.0:
            Print(" %9f%+9f j %12g" % (k.real, k.imag, error))
        else:
            Print(" %12f %12g" % (k.real, error))
    Print("")
```

The main program simply processes three user-defined command-line options and calls the other two functions.

```
def main():
    opts = PETSc.Options()
    N = opts.getInt('N', 32)
    m = opts.getInt('m', N)
    n = opts.getInt('n', m)
    Print("Symmetric Eigenproblem (matrix-free), "
          "N=%d (%dx%d grid)" % (m*n, m, n))
    A = construct_operator(m,n)
    solve_eigensystem(A)

if __name__ == '__main__':
    main()
```

ex4.py: Singular value decomposition of the Lauchli matrix

This example illustrates the use of the SVD solver in slepc4py. It computes singular values and vectors of the Lauchli matrix, whose condition number depends on a parameter μ .

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

```
try: range = xrange
except: pass

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

from petsc4py import PETSc
from slepc4py import SLEPc
```

This example takes two command-line arguments, the matrix size n and the μ parameter.


```

opts = PETSc.Options()
n = opts.getInt('n', 30)
mu = opts.getReal('mu', 1e-6)

PETSc.Sys.Print( "Lauchli singular value decomposition, (%d x %d) mu=%g\n" % (n+1,n,mu) )

```

Create the matrix and fill its nonzero entries. Every MPI process will insert its locally owned part only.

```

A = PETSc.Mat(); A.create()
A.setSizes([n+1, n])
A.setFromOptions()

rstart, rend = A.getOwnershipRange()

for i in range(rstart, rend):
    if i==0:
        for j in range(n):
            A[0,j] = 1.0
    else:
        A[i,i-1] = mu

A.assemble()

```

The singular value solver is similar to the eigensolver used in previous examples. In this case, we select the thick-restart Lanczos bidiagonalization method.

```

S = SLEPc.SVD(); S.create()

S.setOperator(A)
S.setType(S.Type.TRLANCZOS)
S.setFromOptions()

S.solve()

```

After solve, we print some informative data and extract the computed solution, showing the list of singular values and the corresponding residual errors.

```

Print = PETSc.Sys.Print

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

svd_type = S.getType()
Print( "Solution method: %s" % svd_type )

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

nsv, ncv, mpd = S.getDimensions()
Print( "Number of requested singular values: %d" % nsv )

tol, maxit = S.getTolerances()

```

(continues on next page)

(continued from previous page)

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

nconv = S.getConverged()
Print( "Number of converged approximate singular triplets %d" % nconv )

if nconv > 0:
    v, u = A.createVecs()
    Print()
    Print("      sigma      residual norm ")
    Print("-----")
    for i in range(nconv):
        sigma = S.getSingularTriplet(i, u, v)
        error = S.computeError(i)
        Print( "      %6f      %12g" % (sigma, error) )
    Print()
```

ex5.py: Simple quadratic eigenvalue problem

This example solves a polynomial eigenvalue problem of degree 2, which is the most commonly found in applications. Larger degree polynomials are handled similarly. The coefficient matrices in this example do not come from an application, they are just simple matrices that are easy to build, just to illustrate how it works. The eigenvalues in this example are purely imaginary and come in conjugate pairs.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

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

from petsc4py import PETSc
from slepc4py import SLEPc

Print = PETSc.Sys.Print
```

A function to build the matrices. The lowest degree coefficient is the 2-D Laplacian, the highest degree one is the identity matrix, and the other matrix is set to zero (which means that this problem could have been solved as a linear eigenproblem).

```
def construct_operators(m,n):
    Print("Quadratic Eigenproblem, N=%d (%dx%d grid)"% (m*n, m, n))
    # K is the 2-D Laplacian
    K = PETSc.Mat().create()
    K.setSizes([n*m, n*m])
    K.setFromOptions()
    Istart, Iend = K.getOwnershipRange()
    for I in range(Istart,Iend):
        v = -1.0; i = I//n; j = I-i*n;
        if i>0:
            J=I-n; K[I,J] = v
        if i<m-1:
            J=I+n; K[I,J] = v
        if j>0:
```

(continues on next page)

(continued from previous page)

```
        J=I-1; K[I,J] = v
    if j<n-1:
        J=I+1; K[I,J] = v
    v=4.0; K[I,I] = v
K.assemble()
# C is the zero matrix
C = PETSc.Mat().create()
C.setSizes([n*m, n*m])
C.setFromOptions()
C.assemble()
# M is the identity matrix
M = PETSc.Mat().createConstantDiagonal([n*m, n*m], 1.0)

return M, C, K
```

The polynomial eigenvalue solver is similar to the linear eigensolver used in previous examples. The main difference is that we must provide a list of matrices, from lowest to highest degree.

```
def solve_eigensystem(M, C, K):
    # Setup the eigensolver
    Q = SLEPc.PEP().create()
    Q.setOperators([K, C, M])
    Q.setDimensions(6)
    Q.setProblemType(SLEPc.PEP.ProblemType.GENERAL)
    Q.setFromOptions()
    # Solve the eigensystem
    Q.solve()
    # Create the result vectors
    xr, xi = K.createVecs()

    its = Q.getIterationNumber()
    Print("Number of iterations of the method: %i" % its)
    sol_type = Q.getType()
    Print("Solution method: %s" % sol_type)
    nev, ncv, mpd = Q.getDimensions()
    Print("")
    Print("Number of requested eigenvalues: %i" % nev)
    tol, maxit = Q.getTolerances()
    Print("Stopping condition: tol=%.4g, maxit=%d" % (tol, maxit))
    nconv = Q.getConverged()
    Print("Number of converged approximate eigenpairs: %d" % nconv)
    if nconv > 0:
        Print("")
        Print("          k          ||(k^2M+Ck+K)x||/||kx|| ")
        Print("-----")
        for i in range(nconv):
            k = Q.getEigenpair(i, xr, xi)
            error = Q.computeError(i)
            if k.imag != 0.0:
                Print("%9f%+9f j    %12g" % (k.real, k.imag, error))
            else:
                Print("%12f          %12g" % (k.real, error))
```

(continues on next page)

```
Print("")
```

The main program simply processes two user-defined command-line options (the dimensions of the mesh) and calls the other two functions.

```
if __name__ == '__main__':
    opts = PETSc.Options()
    m = opts.getInt('m', 32)
    n = opts.getInt('n', m)
    M, C, K = construct_operators(m,n)
    solve_eigensystem(M, C, K)
    M = C = K = None
```

ex6.py: Compute $\exp(tA)v$ for a matrix from a Markov model

This example illustrates the functionality in slepc4py for computing matrix functions, or more precisely, the application of a matrix function on a given vector. The example works with the exponential function, which is most commonly found in applications.

The main focus of slepc4py is eigenvalue and singular value problems, but it has some codes to deal with matrix functions, which sometimes are needed in the context of eigenproblems, but have interest on their own.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

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

from petsc4py import PETSc
from slepc4py import SLEPc

Print = PETSc.Sys.Print
```

This function builds a matrix that implements a Markov model of a random walk on a triangular grid. The entries of the matrix represent probabilities of moving to neighboring cells in the grid.

```
def build_matrix(m):
    N = m*(m+1)/2
    Print("Markov y=exp(t*A)*e_1, N=%d (m=%d)"% (N, m))
    A = PETSc.Mat().create()
    A.setSizes([N, N])
    A.setFromOptions()
    Istart, Iend = A.getOwnershipRange()
    ix = 0
    cst = 0.5/(m-1)
    for i in range(1,m+1):
        jmax = m-i+1
        for j in range(1,jmax+1):
            ix = ix + 1
            if ix-1<Istart or ix>Iend:
                continue # compute only owned rows
            if j!=jmax:
                pd = cst*(i+j-1)
```

(continues on next page)

(continued from previous page)

```
    # north
    if i==1:
        A[ix-1,ix] = 2*pd
    else:
        A[ix-1,ix] = pd
    # east
    if j==1:
        A[ix-1,ix+jmax-1] = 2*pd
    else:
        A[ix-1,ix+jmax-1] = pd
    # south
    pu = 0.5 - cst*(i+j-3)
    if j>1:
        A[ix-1,ix-2] = pu
    # west
    if i>1:
        A[ix-1,ix-jmax-2] = pu

A.assemble()
return A
```

The following function solves the problem. This case is quite different from eigenproblems, and is more similar to solving a linear system of equations with `KSP`. To configure the problem we must provide the matrix and the function (the exponential in this case). Note how the internal `FN` object is extracted from the `MFN` solver. Also, it is often necessary to specify a scale factor, which in this case represents the time for which we want to obtain the evolved state. Once the solver is set up, we call `solve()` passing the right-hand side vector `b` and the solution vector `x`.

```
def solve_exp(t, A, b, x):
    # Setup the solver
    M = SLEPc.MFN().create()
    M.setOperator(A)
    f = M.getFN()
    f.setType(SLEPc.FN.Type.EXP)
    f.setScale(t)
    M.setTolerances(1e-7)
    M.setFromOptions()
    # Solve the problem
    M.solve(b,x)

    its = M.getIterationNumber()
    Print("Number of iterations of the method: %i" % its)
    sol_type = M.getType()
    Print("Solution method: %s" % sol_type)
    ncv = M.getDimensions()
    Print("")
    Print("Subspace dimension: %i" % ncv)
    tol, maxit = M.getTolerances()
    Print("Stopping condition: tol=%.4g, maxit=%d" % (tol, maxit))
    Print("Computed vector at time t=%.4g has norm %g" % (t.real, x.norm()))
    Print("")
```

The main program processes the command-line option `m` (size of the grid), builds the matrix and calls the solver. Note how the vectors are created from the matrix. In this case, the right-hand side vector is the first element of the canonical

basis.

```
if __name__ == '__main__':
    opts = PETSc.Options()
    m = opts.getInt('m', 15)
    A = build_matrix(m) # transition probability matrix
    x, b = A.createVecs()
    x.set(0)
    b.set(0)
    b[0] = 1
    b.assemble()
    t = 2
    solve_exp(t, A, b, x) # compute x=exp(t*A)*b
    A = None
    b = x = None
```

ex7.py: Nonlinear eigenproblem with callback functions

This example solves a nonlinear eigenvalue problem arising from the discretization of a PDE on a one-dimensional domain with finite differences. The nonlinearity comes from the boundary conditions. The PDE is

$$-u'' = \lambda u$$

defined on the interval $[0,1]$ and subject to the boundary conditions

$$u(0) = 0, u'(1) = u(1)\lambda \frac{\kappa}{\kappa - \lambda},$$

where λ is the eigenvalue and κ is a parameter.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

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

from petsc4py import PETSc
from slepc4py import SLEPc
from numpy import sqrt, sin

Print = PETSc.Sys.Print
```

When implementing a nonlinear eigenproblem with callback functions we must provide code that builds the function matrix $T(\lambda)$ for a given λ and optionally the Jacobian matrix $T'(\lambda)$, i.e., the derivative with respect to the eigenvalue.

In slepc4py the callbacks are integrated in a class. In this example, apart from the constructor, we have three methods:

- `formFunction` to fill the function matrix `F`. Note that `F` is received as an argument and we just need to fill its entries using the value of the parameter `mu`. Matrix `B` is used to build the preconditioner, and is usually equal to `F`.
- `formJacobian` to fill the Jacobian matrix `J`. Some eigensolvers do not need this, but it is recommended to implement it.
- `checkSolution` is just a convenience method to check that a given solution satisfies the PDE.

```

class MyPDE(object):

    def __init__(self, kappa, h):
        self.kappa = kappa
        self.h      = h

    def formFunction(self, nep, mu, F, B):
        n, m = F.getSize()
        Istart, Iend = F.getOwnershipRange()
        i1 = Istart
        if Istart==0: i1 = i1 + 1
        i2 = Iend
        if Iend==n: i2 = i2 - 1
        h = self.h
        c = self.kappa/(mu-self.kappa)
        d = n

        # Interior grid points
        for i in range(i1,i2):
            val = -d-mu*h/6.0
            F[i,i-1] = val
            F[i,i]   = 2.0*(d-mu*h/3.0)
            F[i,i+1] = val

        # Boundary points
        if Istart==0:
            F[0,0] = 2.0*(d-mu*h/3.0)
            F[0,1] = -d-mu*h/6.0
        if Iend==n:
            F[n-1,n-2] = -d-mu*h/6.0
            F[n-1,n-1] = d-mu*h/3.0+mu*c

        F.assemble()
        if B != F: B.assemble()
        return PETSc.Mat.Structure.SAME_NONZERO_PATTERN

    def formJacobian(self, nep, mu, J):
        n, m = J.getSize()
        Istart, Iend = J.getOwnershipRange()
        i1 = Istart
        if Istart==0: i1 = i1 + 1
        i2 = Iend
        if Iend==n: i2 = i2 - 1
        h = self.h
        c = self.kappa/(mu-self.kappa)

        # Interior grid points
        for i in range(i1,i2):
            J[i,i-1] = -h/6.0
            J[i,i]   = -2.0*h/3.0
            J[i,i+1] = -h/6.0

        # Boundary points

```

(continues on next page)

(continued from previous page)

```
if Istart==0:
    J[0,0] = -2.0*h/3.0
    J[0,1] = -h/6.0
if Iend==n:
    J[n-1,n-2] = -h/6.0
    J[n-1,n-1] = -h/3.0-c*c

J.assemble()
return PETSc.Mat.Structure.SAME_NONZERO_PATTERN

def checkSolution(self, mu, y):
    nu = sqrt(mu)
    u = y.duplicate()
    n = u.getSize()
    Istart, Iend = J.getOwnershipRange()
    h = self.h
    for i in range(Istart,Iend):
        x = (i+1)*h
        u[i] = sin(nu*x);
    u.assemble()
    u.normalize()
    u.axy(-1.0,y)
    return u.norm()
```

We use an auxiliary function `FixSign` to force the computed eigenfunction to be real and positive, since some eigen-solvers may return the eigenvector multiplied by a complex number of modulus one.

```
def FixSign(x):
    comm = x.getComm()
    rank = comm.getRank()
    n = 1 if rank == 0 else 0
    aux = PETSc.Vec().createMPI((n, PETSc.DECIDE), comm=comm)
    if rank == 0: aux[0] = x[0]
    aux.assemble()
    x0 = aux.sum()
    sign = x0/abs(x0)
    x.scale(1.0/sign)
```

The main program processes two command-line options, `n` (size of the grid) and `kappa` (the parameter of the PDE), then creates an object of the class we have defined previously.

```
opts = PETSc.Options()
n = opts.getInt('n', 128)
kappa = opts.getReal('kappa', 1.0)
pde = MyPDE(kappa, 1.0/n)
```

In order to set up the solver we have to pass the two callback functions (methods of the class) together with the matrix objects that will be used every time these methods are called. In this simple example we can do a preallocation of the matrices, although this is not necessary.

```
nep = SLEPc.NEP().create()
F = PETSc.Mat().create()
F.setSizes([n, n])
```

(continues on next page)


```

F.setType('aij')
F.setPreallocationNNZ(3)
nep.setFunction(pde.formFunction, F)

J = PETSc.Mat().create()
J.setSizes([n, n])
J.setType('aij')
J.setPreallocationNNZ(3)
nep.setJacobian(pde.formJacobian, J)

```

After setting some options, we can solve the problem. Here we also illustrate how to pass an initial guess to the solver.

```

nep.setTolerances(tol=1e-9)
nep.setDimensions(1)
nep.setFromOptions()

x = F.createVecs('right')
x.set(1.0)
nep.setInitialSpace(x)
nep.solve()

```

Once the solver has finished, we print some information together with the computed solution. For each computed eigenpair, we print the residual norm and also the error estimated with the class method `checkSolution`.

```

its = nep.getIterationNumber()
Print("Number of iterations of the method: %i" % its)
sol_type = nep.getType()
Print("Solution method: %s" % sol_type)
nev, ncv, mpd = nep.getDimensions()
Print("")
Print("Subspace dimension: %i" % ncv)
tol, maxit = nep.getTolerances()
Print("Stopping condition: tol=%.4g" % tol)
Print("")

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

if nconv > 0:
    Print()
    Print("          k          ||T(k)x||          error ")
    Print("-----")
    for i in range(nconv):
        k = nep.getEigenpair(i, x)
        FixSign(x)
        res = nep.computeError(i)
        error = pde.checkSolution(k.real, x)
        if k.imag != 0.0:
            Print( " %9f%+9f j %12g          %12g" % (k.real, k.imag, res, error) )
        else:
            Print( " %12f          %12g          %12g" % (k.real, res, error) )
    Print()

```

ex8.py: Nonlinear eigenproblem with split form

This example solves a nonlinear eigenvalue problem where the nonlinear function is expressed in split form.

We want to solve the following parabolic partial differential equation with time delay τ

$$\begin{aligned}u_t &= u_{xx} + au(t) + bu(t - \tau) \\ u(0, t) &= u(\pi, t) = 0\end{aligned}$$

with $a = 20$ and $b(x) = -4.1 + x(1 - e^{x-\pi})$.

Discretization leads to a DDE of dimension n

$$-u' = Au(t) + Bu(t - \tau)$$

which results in the nonlinear eigenproblem

$$(-\lambda I + A + e^{-\tau\lambda}B)u = 0.$$

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples. In this case we also need to import some math symbols.

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

from petsc4py import PETSc
from slepc4py import SLEPc
from numpy import exp
from math import pi

Print = PETSc.Sys.Print
```

This script has two command-line options: the discretization size `n` and the time delay `tau`.

```
opts = PETSc.Options()
n = opts.getInt('n', 128)
tau = opts.getReal('tau', 0.001)
a = 20
h = pi/(n+1)
```

Next we have to set up the solver. In this case, we are going to represent the nonlinear problem in split form, i.e., as a sum of terms made of a constant matrix multiplied by a scalar nonlinear function.

```
nep = SLEPc.NEP().create()
```

The first term involves the identity matrix.

```
Id = PETSc.Mat().createConstantDiagonal([n, n], 1.0)
```

The second term has a tridiagonal matrix obtained from the discretization, $A = \frac{1}{h^2} \text{tridiag}(1, -2, 1) + aI$.

```
A = PETSc.Mat().create()
A.setSizes([n, n])
A.setFromOptions()
rstart, rend = A.getOwnershipRange()
vd = -2.0/(h*h)+a
```

(continues on next page)

(continued from previous page)

```
vo = 1.0/(h*h)
if rstart == 0:
    A[0, :2] = [vd, vo]
    rstart += 1
if rend == n:
    A[n-1, -2:] = [vo, vd]
    rend -= 1
for i in range(rstart, rend):
    A[i, i-1:i+2] = [vo, vd, vo]
A.assemble()
```

The third term includes a diagonal matrix $B = \text{diag}(b(x_i))$.

```
B = PETSc.Mat().create()
B.setSizes([n, n])
B.setFromOptions()
rstart, rend = B.getOwnershipRange()
for i in range(rstart, rend):
    xi = (i+1)*h
    B[i, i] = -4.1+xi*(1.0-exp(xi-pi));
B.assemble()
B.setOption(PETSc.Mat.Option.HERMITIAN, True)
```

Apart from the matrices, we have to create the functions, represented with `FN` objects: $f_1 = -\lambda, f_2 = 1, f_3 = \exp(-\tau\lambda)$.

```
f1 = SLEPc.FN().create()
f1.setType(SLEPc.FN.Type.RATIONAL)
f1.setRationalNumerator([-1, 0])
f2 = SLEPc.FN().create()
f2.setType(SLEPc.FN.Type.RATIONAL)
f2.setRationalNumerator([1])
f3 = SLEPc.FN().create()
f3.setType(SLEPc.FN.Type.EXP)
f3.setScale(-tau)
```

Put all the information together to define the split operator. Note that `A` is passed first so that `SUBSET nonzero_pattern` can be used.

```
nep.setSplitOperator([A, Id, B], [f2, f1, f3], PETSc.Mat.Structure.SUBSET)
```

Now we can set some options and call the solver.

```
nep.setTolerances(tol=1e-9)
nep.setDimensions(1)
nep.setFromOptions()

nep.solve()
```

Once the solver has finished, we print some information together with the computed solution. For each computed eigenpair, we print the eigenvalue and the residual norm.

```

its = nep.getIterationNumber()
Print("Number of iterations of the method: %i" % its)
sol_type = nep.getType()
Print("Solution method: %s" % sol_type)
nev, ncv, mpd = nep.getDimensions()
Print("")
Print("Subspace dimension: %i" % ncv)
tol, maxit = nep.getTolerances()
Print("Stopping condition: tol=%.4g" % tol)
Print("")

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

if nconv > 0:
    x = Id.createVecs('right')
    x.set(1.0)
    Print()
    Print("          k          ||T(k)x||")
    Print("-----")
    for i in range(nconv):
        k = nep.getEigenpair(i, x)
        res = nep.computeError(i)
        if k.imag != 0.0:
            Print( " %9f%+9f j %12g" % (k.real, k.imag, res) )
        else:
            Print( " %12f          %12g" % (k.real, res) )
    Print()

```

ex9.py: Generalized symmetric-definite eigenproblem

This example computes eigenvalues and eigenvectors of a generalized symmetric-definite eigenvalue problem, where the first matrix is the discrete Laplacian in two dimensions and the second matrix is quasi diagonal.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

```

try: range = xrange
except: pass

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

from petsc4py import PETSc
from slepc4py import SLEPc

Print = PETSc.Sys.Print

```

This function builds the discretized Laplacian operator in 2 dimensions.

```

def Laplacian2D(m, n):
    # Create matrix for 2D Laplacian operator
    A = PETSc.Mat().create()

```

(continues on next page)

(continued from previous page)

```
A.setSizes([m*n, m*n])
A.setFromOptions()
# Fill matrix
hx = 1.0/(m-1) # x grid spacing
hy = 1.0/(n-1) # y grid spacing
diagv = 2.0*hy/hx + 2.0*hx/hy
offdx = -1.0*hy/hx
offdy = -1.0*hx/hy
Istart, Iend = A.getOwnershipRange()
for I in range(Istart, Iend):
    A[I,I] = diagv
    i = I//n # map row number to
    j = I - i*n # grid coordinates
    if i> 0 : J = I-n; A[I,J] = offdx
    if i< m-1: J = I+n; A[I,J] = offdx
    if j> 0 : J = I-1; A[I,J] = offdy
    if j< n-1: J = I+1; A[I,J] = offdy
A.assemble()
return A
```

This function builds a quasi-diagonal matrix. It is two times the identity matrix except for the 2x2 leading submatrix [6 -1; -1 1].

```
def QuasiDiagonal(N):
    # Create matrix
    B = PETSc.Mat().create()
    B.setSizes([N, N])
    B.setFromOptions()
    # Fill matrix
    Istart, Iend = B.getOwnershipRange()
    for I in range(Istart, Iend):
        B[I,I] = 2.0
    if Istart==0:
        B[0,0] = 6.0
        B[0,1] = -1.0
        B[1,0] = -1.0
        B[1,1] = 1.0
    B.assemble()
    return B
```

The following function receives the two matrices and solves the eigenproblem. In this example we illustrate how to pass objects that have been created beforehand, instead of extracting the internal objects. We are using a spectral transformation of type *ST.Type.PRECOND* and a Block Jacobi preconditioner. We want to compute the leftmost eigenvalues. The selected eigensolver is LOBPCG, which is appropriate for this use case. After the solve, we print the computed solution.

```
def solve_eigensystem(A, B, problem_type=SLEPc.EPS.ProblemType.GHEP):
    # Create the results vectors
    xr, xi = A.createVecs()

    pc = PETSc.PC().create()
    # pc.setType(pc.Type.HYPRE)
```

(continues on next page)

```

pc.setType(pc.Type.BJACOBI)

ksp = PETSc.KSP().create()
ksp.setType(ksp.Type.PREONLY)
ksp.setPC( pc )

F = SLEPc.ST().create()
F.setType(F.Type.PRECOND)
F.setKSP( ksp )
F.setShift(0)

# Setup the eigensolver
E = SLEPc.EPS().create()
E.setST(F)
E.setOperators(A,B)
E.setType(E.Type.LOBPCG)
E.setDimensions(10,PETSc.DECIDE)
E.setWhichEigenpairs(E.Which.SMALLEST_REAL)
E.setProblemType( problem_type )
E.setFromOptions()

# Solve the eigensystem
E.solve()

Print("")
its = E.getIterationNumber()
Print("Number of iterations of the method: %i" % its)
sol_type = E.getType()
Print("Solution method: %s" % sol_type)
nev, ncv, mpd = E.getDimensions()
Print("Number of requested eigenvalues: %i" % nev)
tol, maxit = E.getTolerances()
Print("Stopping condition: tol=%.4g, maxit=%d" % (tol, maxit))
nconv = E.getConverged()
Print("Number of converged eigenpairs: %d" % nconv)
if nconv > 0:
    Print("")
    Print("          k          ||Ax-kx||/||kx|| ")
    Print("-----")
    for i in range(nconv):
        k = E.getEigenpair(i, xr, xi)
        error = E.computeError(i)
        if k.imag != 0.0:
            Print(" %9f%+9f j %12g" % (k.real, k.imag, error))
        else:
            Print(" %12f %12g" % (k.real, error))
    Print("")

```

The main program simply processes three user-defined command-line options and calls the other functions.

```

def main():
    opts = PETSc.Options()

```

(continues on next page)

(continued from previous page)

```
N = opts.getInt('N', 10)
m = opts.getInt('m', N)
n = opts.getInt('n', m)
Print("Symmetric-definite Eigenproblem, N=%d (%dx%d grid)" % (m*n, m, n))
A = Laplacian2D(m,n)
B = QuasiDiagonal(m*n)
solve_eigensystem(A,B)

if __name__ == '__main__':
    main()
```

ex10.py: Laplace problem using the Proper Orthogonal Decomposition

This example program solves the Laplace problem using the Proper Orthogonal Decomposition (POD) reduced-order modeling technique. For a full description of the technique the reader is referred to^{1,2}.

The method is split into an offline (computationally intensive) and an online (computationally cheap) phase. This has many applications including real-time simulation, uncertainty quantification and inverse problems, where similar models must be evaluated quickly and many times.

Offline phase:

1. A set of solution snapshots of the 1D Laplace problem in the full problem space are constructed and assembled into the columns of a dense matrix S .
2. A standard eigenvalue decomposition is performed on the matrix $S^T S$.
3. The eigenvectors and eigenvalues are projected back to the original eigenvalue problem S .
4. The leading eigenvectors then form the POD basis.

Online phase:

1. The operator corresponding to the discrete Laplacian is projected onto the POD basis.
2. The operator corresponding to the right-hand side is projected onto the POD basis.
3. The reduced (dense) problem expressed in the POD basis is solved.
4. The reduced solution is projected back to the full problem space.

Contributed by: Elisa Schenone, Jack S. Hale.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples, but importing some additional modules.

```
try: range = xrange
except: pass

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

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

(continues on next page)

¹ K. Kunisch and S. Volkwein. Galerkin proper orthogonal decomposition methods for a general equation in fluid dynamics. SIAM Journal on Numerical Analysis, 40(2):492-515, 2003.

² S. Volkwein, Optimal control of a phase-field model using the proper orthogonal decomposition, Z. Angew. Math. Mech., 81:83-97, 2001.

```
import random
import math
```

This function builds the discrete Laplacian operator in 1 dimension with homogeneous Dirichlet boundary conditions.

```
def construct_operator(m):
    # Create matrix for 1D Laplacian operator
    A = PETSc.Mat().create(PETSc.COMM_SELF)
    A.setSizes([m, m])
    A.setFromOptions()
    # Fill matrix
    hx = 1.0/(m-1) # x grid spacing
    diagv = 2.0/hx
    offdx = -1.0/hx
    Istart, Iend = A.getOwnershipRange()
    for i in range(Istart, Iend):
        if i != 0 and i != (m - 1):
            A[i, i] = diagv
            if i > 1: A[i, i - 1] = offdx
            if i < m - 2: A[i, i + 1] = offdx
        else:
            A[i, i] = 1.

    A.assemble()

    return A
```

Set bell-shape function as the solution of the Laplacian problem in 1 dimension with homogeneous Dirichlet boundary conditions and compute the associated discrete RHS.

```
def set_problem_rhs(m):
    # Create 1D mass matrix operator
    M = PETSc.Mat().create(PETSc.COMM_SELF)
    M.setSizes([m, m])
    M.setFromOptions()
    # Fill matrix
    hx = 1.0/(m-1) # x grid spacing
    diagv = hx/3
    offdx = hx/6
    Istart, Iend = M.getOwnershipRange()
    for i in range(Istart, Iend):
        if i != 0 and i != (m - 1):
            M[i, i] = 2*diagv
        else:
            M[i, i] = diagv
            if i > 1: M[i, i - 1] = offdx
            if i < m - 2: M[i, i + 1] = offdx

    M.assemble()

    x_0 = 0.3
```

(continues on next page)

(continued from previous page)

```
x_f = 0.7
mu = x_0 + (x_f - x_0)*random.random()
sigma = 0.1**2
uex, f = M.createVecs()
for j in range(Istart, Iend):
    value = 2/sigma * math.exp(-(hx*j - mu)**2/sigma) * (1 - 2/sigma * (hx*j -
    mu)**2)
    f.setValue(j, value)
    value = math.exp(-(hx*j - mu)**2/sigma)
    uex.setValue(j, value)
f.assemble()
uex.assemble()

RHS = f.duplicate()
M.mult(f, RHS)
RHS.setValue(0, 0.)
RHS.setValue(m-1, 0.)
RHS.assemble()

return RHS, uex
```

Solve 1D Laplace problem with FEM.

```
def solve_laplace_problem(A, RHS):
    u = A.createVecs('right')
    r, c = A.getOrdering("natural")
    A.factorILU(r, c)
    A.solve(RHS, u)
    A.setUnfactored()
    return u
```

Solve 1D Laplace problem with POD (dense matrix).

```
def solve_laplace_problem_pod(A, RHS, u):
    ksp = PETSc.KSP().create(PETSc.COMM_SELF)
    ksp.setOperators(A)
    ksp.setType('preonly')
    pc = ksp.getPC()
    pc.setType('none')
    ksp.setFromOptions()

    ksp.solve(RHS, u)

    return u
```

Set N solution of the 1D Laplace problem as columns of a matrix (snapshot matrix).

Note: For simplicity we do not perform a linear solve, but use some analytical solution: $z(x) = \exp(-(x - \mu)^2/\sigma)$.

```
def construct_snapshot_matrix(A, N, m):
    snapshots = PETSc.Mat().create(PETSc.COMM_SELF)
    snapshots.setSizes([m, N])
    snapshots.setType('seqdense')
```

(continues on next page)

```

Istart, Iend = snapshots.getOwnershipRange()
hx = 1.0/(m - 1)
x_0 = 0.3
x_f = 0.7
sigma = 0.1**2
for i in range(N):
    mu = x_0 + (x_f - x_0)*random.random()
    for j in range(Istart, Iend):
        value = math.exp(-(hx*j - mu)**2/sigma)
        snapshots.setValue(j, i, value)
snapshots.assemble()

return snapshots

```

Solve the eigenvalue problem: the eigenvectors of this problem form the POD basis.

```

def solve_eigenproblem(snapshots, N):
    print('Solving POD basis eigenproblem using eigensolver...')

    Es = SLEPc.EPS()
    Es.create(PETSc.COMM_SELF)
    Es.setDimensions(N)
    Es.setProblemType(SLEPc.EPS.ProblemType.NHEP)
    Es.setTolerances(1.0e-8, 500);
    Es.setKrylovSchurRestart(0.6)
    Es.setWhichEigenpairs(SLEPc.EPS.Which.LARGEST_REAL)
    Es.setOperators(snapshots)
    Es.setFromOptions()

    Es.solve()
    print('Solved POD basis eigenproblem.')
    return Es

```

Function to project $S^T S$ eigenvectors to S eigenvectors.

```

def project_STS_eigenvectors_to_S_eigenvectors(bvEs, S):
    sizes = S.getSizes()[0]
    N = bvEs.getActiveColumns()[1]
    bv = SLEPc.BV().create(PETSc.COMM_SELF)
    bv.setSizes(sizes, N)
    bv.setActiveColumns(0, N)
    bv.setFromOptions()

    tmpvec2 = S.createVecs('left')
    for i in range(N):
        tmpvec = bvEs.getColumn(i)
        S.mult(tmpvec, tmpvec2)
        bv.insertVec(i, tmpvec2)
        bvEs.restoreColumn(i, tmpvec)

    return bv

```

Function to project the reduced space to the full space.

```
def project_reduced_to_full_space(alpha, bv):
    uu = bv.getColumn(0)
    uPOD = uu.duplicate()
    bv.restoreColumn(0,uu)

    scatter, Wr = PETSc.Scatter.toAll(alpha)
    scatter.begin(alpha, Wr, PETSc.InsertMode.INSERT, PETSc.ScatterMode.FORWARD)
    scatter.end(alpha, Wr, PETSc.InsertMode.INSERT, PETSc.ScatterMode.FORWARD)
    PODcoeff = Wr.getArray(readonly=1)

    bv.multVec(1., 0., uPOD, PODcoeff)

    return uPOD
```

The main function realizes the full procedure.

```
def main():
    problem_dim = 200
    num_snapshots = 30
    num_pod_basis_functions = 8

    assert(num_pod_basis_functions <= num_snapshots)

    A = construct_operator(problem_dim)
    S = construct_snapshot_matrix(A, num_snapshots, problem_dim)

    # Instead of solving the SVD of S, we solve the standard
    # eigenvalue problem on S.T*S
    STS = S.transposeMatMult(S)

    Es = solve_eigenproblem(STS, num_pod_basis_functions)
    nconv = Es.getConverged()
    print('Number of converged eigenvalues: %i' % nconv)
    Es.view()

    # get the EPS solution in a BV object
    bvEs = Es.getBV()
    bvEs.setActiveColumns(0, num_pod_basis_functions)

    # set the bv POD basis
    bv = project_STS_eigenvectors_to_S_eigenvectors(bvEs, S)
    # rescale the eigenvectors
    for i in range(num_pod_basis_functions):
        ll = Es.getEigenvalue(i)
        print('Eigenvalue '+str(i)+' : '+str(ll.real))
        bv.scaleColumn(i, 1.0/math.sqrt(ll.real))

    print('-----')
    # Verify that the active columns of bv form an orthonormal subspace, i.e. that  $X^H X = Id$ 
    print('Check that bv.dot(bv) is close to the identity matrix')
    XtX = bv.dot(bv)
```

(continues on next page)

```

XtX.view()
XtX_array = XtX.getDenseArray()
n,m = XtX_array.shape
assert numpy.allclose(XtX_array, numpy.eye(n, m))
print('-----')
print('Solve the problem with POD')

# Project the linear operator A
Ared = bv.matProject(A,bv)

# Set the RHS and the exact solution
RHS, uex = set_problem_rhs(problem_dim)

# Project the RHS on the POD basis
RHSred = PETSc.Vec().createWithArray(bv.dotVec(RHS))

# Solve the problem with POD
alpha = Ared.createVecs('right')
alpha = solve_laplace_problem_pod(Ared,RHSred,alpha)

# Project the POD solution back to the FE space
uPOD = project_reduced_to_full_space(alpha, bv)

# Compute the L2 and Linf norm of the error
error = uex.copy()
error.axpy(-1,uPOD)
errorL2 = math.sqrt(error.dot(error).real)
print('The L2-norm of the error is: '+str(errorL2))

print("NORMAL END")

if __name__ == '__main__':
    main()

```

ex11.py: 2-D Laplacian eigenproblem solved with contour integral

This example is similar to ex2.py, but employs a contour integral solver. It illustrates how to define a region of the complex plane using an *RG* object.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

```

try: range = xrange
except: pass

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

from petsc4py import PETSc
from slepc4py import SLEPc

Print = PETSc.Sys.Print

```

Build the finite-difference 2-D Laplacian matrix.

```
def construct_operator(m, n):
    # Create matrix for 2D Laplacian operator
    A = PETSc.Mat().create()
    A.setSizes([m*n, m*n])
    A.setFromOptions()
    # Fill matrix
    hx = 1.0/(m-1) # x grid spacing
    hy = 1.0/(n-1) # y grid spacing
    diagv = 2.0*hy/hx + 2.0*hx/hy
    offdx = -1.0*hy/hx
    offdy = -1.0*hx/hy
    Istart, Iend = A.getOwnershipRange()
    for I in range(Istart, Iend) :
        A[I,I] = diagv
        i = I//n # map row number to
        j = I - i*n # grid coordinates
        if i> 0 : J = I-n; A[I,J] = offdx
        if i< m-1: J = I+n; A[I,J] = offdx
        if j> 0 : J = I-1; A[I,J] = offdy
        if j< n-1: J = I+1; A[I,J] = offdy
    A.assemble()
    return A
```

In the main function, first two command-line options are processed to set the grid dimensions. Then the matrix is built and passed to the solver object. In this case, the solver is configured to use the contour integral method. Next, the region of interest is defined, in this case an ellipse centered at the origin, with radius 0.2 and vertical scaling of 0.1. Finally, the solver is run. In this example, we illustrate how to print the solution using the solver method `errorView()`.

```
def main():
    opts = PETSc.Options()
    n = opts.getInt('n', 32)
    m = opts.getInt('m', 32)
    Print("2-D Laplacian Eigenproblem solved with contour integral, "
          "N=%d (%dx%d grid)\n" % (m*n, m, n))
    A = construct_operator(m,n)

    E = SLEPc.EPS().create()
    E.setOperators(A)
    E.setProblemType(SLEPc.EPS.ProblemType.HEP)
    E.setType(SLEPc.EPS.Type.CISS)

    R = E.getRG()
    R.setType(SLEPc.RG.Type.ELLIPSE)
    R.setEllipseParameters(0.0,0.2,0.1)
    E.setFromOptions()

    E.solve()

    vw = PETSc.Viewer.STDOUT()
    vw.pushFormat(PETSc.Viewer.Format.ASCII_INFO_DETAIL)
    E.errorView(viewer=vw)
    vw.popFormat()
```

(continues on next page)

```
if __name__ == '__main__':
    main()
```

ex12.py: Illustrate the use of arbitrary selection

This example solves a simple tridiagonal eigenproblem. It illustrates how to set up the arbitrary selection of eigenvalues, where the decision of which is the preferred eigenvalue is made based not only on the value of the approximate eigenvalue but also on the approximate eigenvector.

In this example, the selection criterion is based on the projection of the approximate eigenvector onto a precomputed eigenvector. That is why we solve the problem twice.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

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

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

The matrix size `n` can be specified at the command line.

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

Create the matrix `tridiag([-1 0 -1])`.

```
A = PETSc.Mat(); A.create()
A.setSizes([n, n])
A.setFromOptions()
rstart, rend = A.getOwnershipRange()
for i in range(rstart, rend):
    if i>0: A[i, i-1] = -1
    if i<n-1: A[i, i+1] = -1
A.assemble()
```

Configure the linear eigensolver initially to compute leftmost eigenvalues.

```
E = SLEPc.EPS(); E.create()
E.setOperators(A)
E.setProblemType(SLEPc.EPS.ProblemType.HEP)
E.setWhichEigenpairs(SLEPc.EPS.Which.SMALLEST_REAL)
E.setFromOptions()
```

Solve the eigenproblem and store the first computed eigenvector in `sx` to be used later. For the second solve, we configure the solver to select largest magnitude values with an arbitrary selection callback function `myArbitrarySel()`. It means that instead of sorting eigenvalues, the solver will sort the approximations according to the largest values of the result of `myArbitrarySel()` evaluated on the approximate eigenvectors. In this way, the same eigenvalue should be computed again.

```

E.solve()
nconv = E.getConverged()
Print = PETSc.Sys.Print
vw = PETSc.Viewer.STDOUT()
if nconv>0:
    sx, _ = A.createVecs()
    E.getEigenpair(0, sx)
    vw.pushFormat(PETSc.Viewer.Format.ASCII_INFO_DETAIL)
    E.errorView(viewer=vw)
    def myArbitrarySel(evalue, xr, xi, sx):
        return abs(xr.dot(sx))
    E.setArbitrarySelection(myArbitrarySel, sx)
    E.setWhichEigenpairs(SLEPc.EPS.Which.LARGEST_MAGNITUDE)
    E.solve()
    E.errorView(viewer=vw)
    vw.popFormat()
else:
    Print( "No eigenpairs converged" )

```

ex13.py: Nonlinear eigenproblem with contour integral

This example solves a nonlinear eigenvalue problem with the *NEP* module configured to apply a contour integral method.

The problem arises from the PDE

$$u_{xx}(x) + n_c^2 \lambda^2 u(x) + g(\lambda) D_0 \lambda^2 u(x) = 0$$

where

$$\begin{aligned}
 g(\lambda) &= g_t / (\lambda - k_a + i g_t), \\
 k_a &= 8.0, \\
 g_t &= 0.5, \\
 D_0 &= 0.5, \\
 n_c &= 1.2,
 \end{aligned}$$

and the boundary conditions are

$$\begin{aligned}
 u(0) &= 0 \\
 u_x(1) &= i\lambda u(1).
 \end{aligned}$$

For the discretization, n grid points are used, $x_1 = 0, \dots, x_n = 1$, with step size $h = 1/(n - 1)$. Hence,

$$\begin{aligned}
 u_{xx}(x_i) &= \frac{1}{h^2} (u_{i-1} - 2u_i + u_{i+1}) \\
 &= \frac{1}{h^2} [1, -2, 1] [u_{i-1}, u_i, u_{i+1}]^T.
 \end{aligned}$$

The boundary condition at $x = 0$ is $u_1 = 0$, and at $x = 1$:

$$\begin{aligned}
 u'(1) &= 1/2(u'(1 + h/2) + u'(1 - h/2)) \\
 &= 1/2((u_{n+1} - u_n)/h + (u_n - u_{n-1})/h) \\
 &= 1/(2h)(u_{n+1} - u_{n-1}) = i\lambda u_n,
 \end{aligned}$$

and therefore

$$u_{n+1} = 2ih\lambda u_n + u_{n-1}.$$

The Laplace term for u_n is

$$\begin{aligned}
 &= \frac{1}{h^2}(u_{n-1} - 2u_n + u_{n+1}) \\
 &= \frac{1}{h^2}(u_{n-1} - 2u_n + 2ih\lambda u_n + u_{n-1}) \\
 &= \frac{1}{h^2}(2u_{n-1} + (2ih\lambda - 2)u_n) \\
 &= \frac{1}{h^2}[2, 2ih\lambda - 2][u_{n-1}, u_n]^T.
 \end{aligned}$$

The above discretization allows us to write the nonlinear PDE in the following split-operator form

$$\{A + \lambda^2 n_c^2 I_d + g(\lambda)\lambda^2 D_0 I_d + 2i\lambda/hD\}u = 0$$

so $f_1 = 1$, $f_2 = n_c^2 \lambda^2$, $f_3 = g(\lambda)\lambda^2 D_0$, $f_4 = 2i\lambda/h$, with coefficient matrices

$$A = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & * & * & \dots & 0 \\ 0 & * & * & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & * \end{bmatrix}, I_d = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}.$$

Contributed by: Thomas Hisch.

The full source code for this demo can be [downloaded here](#).

Initialization by importing slepc4py, petsc4py, numpy and scipy.

```

import sys

import slepc4py

slepc4py.init(sys.argv) # isort:skip

import numpy as np

try:
    import scipy
    import scipy.optimize
except ImportError:
    scipy = None

from petsc4py import import PETSc
from slepc4py import import SLEPc

Print = PETSc.Sys.Print

```

Check that the selected PETSc/SLEPc are built with complex scalars.

```

if not np.issubdtype(PETSc.ScalarType, np.complexfloating):
    Print("Demo should only be executed with complex PETSc scalars")
    exit(0)

```

Long function that defines the nonlinear eigenproblem and solves it.


```

def solve(n):
    L = 1.0
    h = L / (n - 1)
    nc = 1.2
    ka = 10.0
    gt = 4.0
    D0 = 0.5

    A = PETSc.Mat().create()
    A.setSizes([n, n])
    A.setFromOptions()
    A.setOption(PETSc.Mat.Option.HERMITIAN, False)

    rstart, rend = A.getOwnershipRange()
    d0, d1, d2 = (
        1 / h**2,
        -2 / h**2,
        1 / h**2,
    )
    Print(f"dterms={d0, d1, d2}")

    if rstart == 0:
        # dirichlet boundary condition at the left lead
        A[0, 0] = 1.0
        A[0, 1] = 0.0
        A[1, 0] = 0.0

        A[1, 1] = d1
        A[1, 2] = d2
        rstart += 2
    if rend == n:
        # at x=1.0 neumann boundary condition (not handled here but in a
        # different matrix (D))
        A[n - 1, n - 2] = 2.0 / h**2
        A[n - 1, n - 1] = (-2) / h**2 # + 2j*k*h / h**2 (neumann)
        rend -= 1

    for i in range(rstart, rend):
        A[i, i - 1 : i + 2] = [d0, d1, d2]

    A.assemble()

    Id = PETSc.Mat().create()
    Id.setSizes([n, n])
    Id.setFromOptions()
    Id.setOption(PETSc.Mat.Option.HERMITIAN, True)
    rstart, rend = Id.getOwnershipRange()
    if rstart == 0:
        # due to dirichlet BC
        rstart += 1
    for i in range(rstart, rend):
        Id[i, i] = 1.0
    Id.assemble()

```

(continues on next page)

```

D = PETSc.Mat().create()
D.setSizes([n, n])
D.setFromOptions()
D.setOption(PETSc.Mat.Option.HERMITIAN, True)
_, rend = D.getOwnershipRange()
if rend == n:
    D[n - 1, n - 1] = 1
D.assemble()

Print(f"DOF: {A.getInfo()['nz_used']}, MEM: {A.getInfo()['memory']}")

f1 = SLEPc.FN().create()
f1.setType(SLEPc.FN.Type.RATIONAL)
f1.setRationalNumerator([1.0])
f2 = SLEPc.FN().create()
f2.setType(SLEPc.FN.Type.RATIONAL)
f2.setRationalNumerator([nc**2, 0.0, 0.0])
f3 = SLEPc.FN().create()
f3.setType(SLEPc.FN.Type.RATIONAL)
f3.setRationalNumerator([D0 * gt, 0.0, 0.0])
f3.setRationalDenominator([1.0, -ka + 1j * gt])
f4 = SLEPc.FN().create()
f4.setType(SLEPc.FN.Type.RATIONAL)
f4.setRationalNumerator([2j / h, 0])

# Setup the solver
nep = SLEPc.NEP().create()

nep.setSplitOperator(
    [A, Id, Id, D],
    [f1, f2, f3, f4],
    PETSc.Mat.Structure.SUBSET,
)

# Customize options
nep.setTolerances(tol=1e-7)

nep.setDimensions(nev=24)
nep.setType(SLEPc.NEP.Type.CISS)

# the rg params are chosen s.t. the singularity at  $k = ka - 1j*gt$  is
# outside of the contour.
radius = 3 * gt
vscale = 0.5 * gt / radius
rg_params = (ka, 3 * gt, vscale)
R = nep.getRG()
R.setType(SLEPc.RG.Type.ELLIPSE)
Print(f"RG params: {rg_params}")
R.setEllipseParameters(*rg_params)

nep.setFromOptions()

```

(continues on next page)

```

# Solve the problem
nep.solve()

its = nep.getIterationNumber()
Print("Number of iterations of the method: %i" % its)
sol_type = nep.getType()
Print("Solution method: %s" % sol_type)
nev, ncv, mpd = nep.getDimensions()
Print("")
Print("Subspace dimension: %i" % ncv)
tol, maxit = nep.getTolerances()
Print("Stopping condition: tol=%.4g" % tol)
Print("")

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

x = A.createVecs("right")

evals = []
modes = []
if nconv > 0:
    Print()
    Print("          lam          ||T(lam)x||   |lam-lam_exact|/|lam_exact| ")
    Print("-----")
    for i in range(nconv):
        lam = nep.getEigenpair(i, x)
        error = nep.computeError(i)

        def eigenvalue_error_term(k):
            gkmu = gt / (k - ka + 1j * gt)
            nceff = np.sqrt(nc**2 + gkmu * D0)
            return -1j / np.tan(nceff * k * L) - 1 / nceff

        # compute the expected eigenvalue

        # we assume that the numerically calculated eigenvalue is close to
        # the exact one, which we can determine using a Newton-Raphson
        # method.
        if scipy:
            expected_lam = scipy.optimize.newton(
                eigenvalue_error_term, np.complex128(lam), rtol=1e-11
            )
            rel_err = abs(lam - expected_lam) / abs(expected_lam)
            rel_err = "%6g" % rel_err
        else:
            rel_err = "scipy not installed"

    Print(" %9f%+9f j %12g   %s" % (lam.real, lam.imag, error, rel_err))

    evals.append(lam)

```

(continues on next page)

```

        modes.append(x.getArray().copy())
    Print()

    return np.asarray(evals), rg_params, ka, gt

```

The main function reads the problem size n from the command line, solves the problem with the above function, and then plots the computed eigenvalues.

```

def main():
    opts = PETSc.Options()
    n = opts.getInt("n", 256)
    Print(f"n={n}")

    evals, rg_params, ka, gt = solve(n)

    if not opts.getBool("ploteigs", True) or PETSc.COMM_WORLD.getRank():
        return

    try:
        import matplotlib.pyplot as plt
        from matplotlib.patches import Ellipse
    except ImportError:
        print("plot is not shown, because matplotlib is not installed")
    else:
        fig, ax = plt.subplots()
        ax.plot(evals.real, evals.imag, "x")

        height = 2 * rg_params[1] * rg_params[2]
        ellipse = Ellipse(
            xy=(rg_params[0], 0.0),
            width=rg_params[1] * 2,
            height=height,
            edgecolor="r",
            fc="None",
            lw=2,
        )
        ax.add_patch(ellipse)

        ax.grid()
        ax.legend()
        plt.show()

if __name__ == "__main__":
    main()

```

ex14.py: Lyapunov equation with the shifted 2-D Laplacian

This example illustrates the use of slepc4py linear matrix equations solver object. In particular, a Lyapunov equation is solved, where the coefficient matrix is the shifted 2-D Laplacian.

The full source code for this demo can be [downloaded here](#).

Initialization is similar to previous examples.

```

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

from petsc4py import PETSc
from slepc4py import SLEPc

Print = PETSc.Sys.Print

```

Once again, a function to build the discretized Laplacian operator in two dimensions. In this case, we build the shifted negative Laplacian because the Lyapunov equation requires the coefficient matrix being stable.

```

def Laplacian2D(m, n, sigma):
    # Create matrix for 2D Laplacian operator
    A = PETSc.Mat().create()
    A.setSizes([m*n, m*n])
    A.setFromOptions()
    # Fill matrix
    Istart, Iend = A.getOwnershipRange()
    for I in range(Istart, Iend):
        A[I,I] = -4.0-sigma
        i = I//n # map row number to
        j = I - i*n # grid coordinates
        if i> 0 : J = I-n; A[I,J] = 1.0
        if i< m-1: J = I+n; A[I,J] = 1.0
        if j> 0 : J = I-1; A[I,J] = 1.0
        if j< n-1: J = I+1; A[I,J] = 1.0
    A.assemble()
    return A

```

The following function solves the continuous-time Lyapunov equation

$$AX + XA^* = -C$$

where C is supposed to have low rank. The solution X also has low rank, which will be restricted to rk if it was provided as an argument of the function. After solving the equation, this function computes and prints a residual norm.

```

def solve_lyap(A, C, rk=0):
    # Setup the solver
    L = SLEPc.LME().create()
    L.setProblemType(SLEPc.LME.ProblemType.LYAPUNOV)
    L.setCoefficients(A)
    L.setRHS(C)

    N = C.size[0]
    if rk>0:
        X1 = PETSc.Mat().createDense((N,rk))
        X1.assemble()
        X = PETSc.Mat().createLRC(None, X1, None, None)
        L.setSolution(X)

    L.setTolerances(1e-7)
    L.setFromOptions()

```

(continues on next page)

```

# Solve the problem
L.solve()

if rk==0:
    X = L.getSolution()

its = L.getIterationNumber()
Print(f'Number of iterations of the method: {its}')
sol_type = L.getType()
Print(f'Solution method: {sol_type}')
tol, maxit = L.getTolerances()
Print(f'Stopping condition: tol={tol}, maxit={maxit}')
Print(f'Error estimate reported by the solver: {L.getErrorEstimate()}')
if N<500:
    Print(f'Computed residual norm: {L.computeError()}')
Print('')

return X

```

The main function processes several command-line arguments such as the grid dimension (n, m), the shift `sigma` and the rank of the solution, `rank`.

The coefficient matrix `A` is built with the function above, while the right-hand side matrix `C` must be built as a low-rank matrix using `Mat.createLRC()`.

```

if __name__ == '__main__':
    opts = PETSc.Options()
    n = opts.getInt('n', 15)
    m = opts.getInt('m', n)
    N = m*n
    sigma = opts.getScalar('sigma', 0.0)
    rk = opts.getInt('rank', 0)

    # Coefficient matrix A
    A = Laplacian2D(m, n, sigma)

    # Create a low-rank Mat to store the right-hand side C = C1*C1'
    C1 = PETSc.Mat().createDense((N,2))
    rstart, rend = C1.getOwnershipRange()
    v = C1.getDenseArray()
    for i in range(rstart,rend):
        if i<N//2: v[i-rstart,0] = 1.0
        if i==0: v[i-rstart,1] = -2.0
        if i==1 or i==2: v[i-rstart,1] = -1.0
    C1.assemble()
    C = PETSc.Mat().createLRC(None, C1, None, None)

    X = solve_lyap(A, C, rk)

```

Python Module Index

S

`slepc4py`, 8
`slepc4py.SLEPc`, 13
`slepc4py.typing`, 9

Index

Symbols

`__init__()` (*slepc4py.SLEPc.BV* method), 52
`__init__()` (*slepc4py.SLEPc.BV.MatMultType* method), 14
`__init__()` (*slepc4py.SLEPc.BV.OrthogBlockType* method), 15
`__init__()` (*slepc4py.SLEPc.BV.OrthogRefineType* method), 16
`__init__()` (*slepc4py.SLEPc.BV.OrthogType* method), 17
`__init__()` (*slepc4py.SLEPc.BV.SVDMethod* method), 17
`__init__()` (*slepc4py.SLEPc.BV.Type* method), 18
`__init__()` (*slepc4py.SLEPc.DS* method), 76
`__init__()` (*slepc4py.SLEPc.DS.MatType* method), 54
`__init__()` (*slepc4py.SLEPc.DS.ParallelType* method), 54
`__init__()` (*slepc4py.SLEPc.DS.StateType* method), 55
`__init__()` (*slepc4py.SLEPc.DS.Type* method), 57
`__init__()` (*slepc4py.SLEPc.EPS* method), 156
`__init__()` (*slepc4py.SLEPc.EPS.Balance* method), 77
`__init__()` (*slepc4py.SLEPc.EPS.CISSExtraction* method), 78
`__init__()` (*slepc4py.SLEPc.EPS.CISSQuadRule* method), 78
`__init__()` (*slepc4py.SLEPc.EPS.Conv* method), 79
`__init__()` (*slepc4py.SLEPc.EPS.ConvergedReason* method), 80
`__init__()` (*slepc4py.SLEPc.EPS.ErrorType* method), 81
`__init__()` (*slepc4py.SLEPc.EPS.Extraction* method), 82
`__init__()` (*slepc4py.SLEPc.EPS.KrylovSchurBSEType* method), 83
`__init__()` (*slepc4py.SLEPc.EPS.LanczosReorthogType* method), 84
`__init__()` (*slepc4py.SLEPc.EPS.PowerShiftType* method), 84
`__init__()` (*slepc4py.SLEPc.EPS.ProblemType* method), 86
`__init__()` (*slepc4py.SLEPc.EPS.Stop* method), 86
`__init__()` (*slepc4py.SLEPc.EPS.Type* method), 89
`__init__()` (*slepc4py.SLEPc.EPS.Which* method), 90
`__init__()` (*slepc4py.SLEPc.FN* method), 170
`__init__()` (*slepc4py.SLEPc.FN.CombineType* method), 157
`__init__()` (*slepc4py.SLEPc.FN.ParallelType* method), 158
`__init__()` (*slepc4py.SLEPc.FN.Type* method), 159
`__init__()` (*slepc4py.SLEPc.LME* method), 187
`__init__()` (*slepc4py.SLEPc.LME.ConvergedReason* method), 172
`__init__()` (*slepc4py.SLEPc.LME.ProblemType* method), 173
`__init__()` (*slepc4py.SLEPc.LME.Type* method), 173
`__init__()` (*slepc4py.SLEPc.MFN* method), 200
`__init__()` (*slepc4py.SLEPc.MFN.ConvergedReason* method), 188
`__init__()` (*slepc4py.SLEPc.MFN.Type* method), 189
`__init__()` (*slepc4py.SLEPc.NEP* method), 255
`__init__()` (*slepc4py.SLEPc.NEP.CISSExtraction* method), 202
`__init__()` (*slepc4py.SLEPc.NEP.Conv* method), 202
`__init__()` (*slepc4py.SLEPc.NEP.ConvergedReason* method), 204
`__init__()` (*slepc4py.SLEPc.NEP.ErrorType* method), 204
`__init__()` (*slepc4py.SLEPc.NEP.ProblemType* method), 205
`__init__()` (*slepc4py.SLEPc.NEP.Refine* method), 206
`__init__()` (*slepc4py.SLEPc.NEP.RefineScheme* method), 206
`__init__()` (*slepc4py.SLEPc.NEP.Stop* method), 207
`__init__()` (*slepc4py.SLEPc.NEP.Type* method), 208
`__init__()` (*slepc4py.SLEPc.NEP.Which* method), 209
`__init__()` (*slepc4py.SLEPc.PEP* method), 313
`__init__()` (*slepc4py.SLEPc.PEP.Basis* method), 257
`__init__()` (*slepc4py.SLEPc.PEP.CISSExtraction* method), 258
`__init__()` (*slepc4py.SLEPc.PEP.Conv* method), 259
`__init__()` (*slepc4py.SLEPc.PEP.ConvergedReason* method), 260
`__init__()` (*slepc4py.SLEPc.PEP.ErrorType* method), 260
`__init__()` (*slepc4py.SLEPc.PEP.Extract* method), 261
`__init__()` (*slepc4py.SLEPc.PEP.JDProjection* method), 262
`__init__()` (*slepc4py.SLEPc.PEP.ProblemType* method), 263
`__init__()` (*slepc4py.SLEPc.PEP.Refine* method), 263
`__init__()` (*slepc4py.SLEPc.PEP.RefineScheme* method), 264
`__init__()` (*slepc4py.SLEPc.PEP.Scale* method), 265
`__init__()` (*slepc4py.SLEPc.PEP.Stop* method), 265
`__init__()` (*slepc4py.SLEPc.PEP.Type* method), 266
`__init__()` (*slepc4py.SLEPc.PEP.Which* method), 268
`__init__()` (*slepc4py.SLEPc.RG* method), 326
`__init__()` (*slepc4py.SLEPc.RG.QuadRule* method), 314
`__init__()` (*slepc4py.SLEPc.RG.Type* method), 314

`__init__()` (*slepc4py.SLEPc.ST* method), 349
`__init__()` (*slepc4py.SLEPc.ST.FilterDamping* method), 327
`__init__()` (*slepc4py.SLEPc.ST.FilterType* method), 328
`__init__()` (*slepc4py.SLEPc.ST.MatMode* method), 329
`__init__()` (*slepc4py.SLEPc.ST.Type* method), 330
`__init__()` (*slepc4py.SLEPc.SVD* method), 388
`__init__()` (*slepc4py.SLEPc.SVD.Conv* method), 350
`__init__()` (*slepc4py.SLEPc.SVD.ConvergedReason* method), 352
`__init__()` (*slepc4py.SLEPc.SVD.ErrorType* method), 352
`__init__()` (*slepc4py.SLEPc.SVD.ProblemType* method), 353
`__init__()` (*slepc4py.SLEPc.SVD.Stop* method), 354
`__init__()` (*slepc4py.SLEPc.SVD.TRLanczosGBidiag* method), 354
`__init__()` (*slepc4py.SLEPc.SVD.Type* method), 356
`__init__()` (*slepc4py.SLEPc.SVD.Which* method), 357
`__init__()` (*slepc4py.SLEPc.Sys* method), 390
`__init__()` (*slepc4py.SLEPc.Util* method), 391
`__new__()` (*slepc4py.SLEPc.BV* class method), 52
`__new__()` (*slepc4py.SLEPc.BV.MatMultType* class method), 14
`__new__()` (*slepc4py.SLEPc.BV.OrthogBlockType* class method), 15
`__new__()` (*slepc4py.SLEPc.BV.OrthogRefineType* class method), 16
`__new__()` (*slepc4py.SLEPc.BV.OrthogType* class method), 17
`__new__()` (*slepc4py.SLEPc.BV.SVDMethod* class method), 17
`__new__()` (*slepc4py.SLEPc.BV.Type* class method), 18
`__new__()` (*slepc4py.SLEPc.DS* class method), 76
`__new__()` (*slepc4py.SLEPc.DS.MatType* class method), 54
`__new__()` (*slepc4py.SLEPc.DS.ParallelType* class method), 55
`__new__()` (*slepc4py.SLEPc.DS.StateType* class method), 55
`__new__()` (*slepc4py.SLEPc.DS.Type* class method), 57
`__new__()` (*slepc4py.SLEPc.EPS* class method), 156
`__new__()` (*slepc4py.SLEPc.EPS.Balance* class method), 77
`__new__()` (*slepc4py.SLEPc.EPS.CISSExtraction* class method), 78
`__new__()` (*slepc4py.SLEPc.EPS.CISSQuadRule* class method), 78
`__new__()` (*slepc4py.SLEPc.EPS.Conv* class method), 79
`__new__()` (*slepc4py.SLEPc.EPS.ConvergedReason* class method), 80
`__new__()` (*slepc4py.SLEPc.EPS.ErrorType* class method), 81
`__new__()` (*slepc4py.SLEPc.EPS.Extraction* class method), 82
`__new__()` (*slepc4py.SLEPc.EPS.KrylovSchurBSEType* class method), 83
`__new__()` (*slepc4py.SLEPc.EPS.LanczosReorthogType* class method), 84
`__new__()` (*slepc4py.SLEPc.EPS.PowerShiftType* class method), 84
`__new__()` (*slepc4py.SLEPc.EPS.ProblemType* class method), 86
`__new__()` (*slepc4py.SLEPc.EPS.Stop* class method), 86
`__new__()` (*slepc4py.SLEPc.EPS.Type* class method), 89
`__new__()` (*slepc4py.SLEPc.EPS.Which* class method), 90
`__new__()` (*slepc4py.SLEPc.FN* class method), 170
`__new__()` (*slepc4py.SLEPc.FN.CombineType* class method), 157
`__new__()` (*slepc4py.SLEPc.FN.ParallelType* class method), 158
`__new__()` (*slepc4py.SLEPc.FN.Type* class method), 159
`__new__()` (*slepc4py.SLEPc.LME* class method), 187
`__new__()` (*slepc4py.SLEPc.LME.ConvergedReason* class method), 172
`__new__()` (*slepc4py.SLEPc.LME.ProblemType* class method), 173
`__new__()` (*slepc4py.SLEPc.LME.Type* class method), 173
`__new__()` (*slepc4py.SLEPc.MFN* class method), 200
`__new__()` (*slepc4py.SLEPc.MFN.ConvergedReason* class method), 188
`__new__()` (*slepc4py.SLEPc.MFN.Type* class method), 189
`__new__()` (*slepc4py.SLEPc.NEP* class method), 256
`__new__()` (*slepc4py.SLEPc.NEP.CISSExtraction* class method), 202
`__new__()` (*slepc4py.SLEPc.NEP.Conv* class method), 202
`__new__()` (*slepc4py.SLEPc.NEP.ConvergedReason* class method), 204
`__new__()` (*slepc4py.SLEPc.NEP.ErrorType* class method), 204
`__new__()` (*slepc4py.SLEPc.NEP.ProblemType* class method), 205
`__new__()` (*slepc4py.SLEPc.NEP.Refine* class method), 206
`__new__()` (*slepc4py.SLEPc.NEP.RefineScheme* class method), 206
`__new__()` (*slepc4py.SLEPc.NEP.Stop* class method), 207
`__new__()` (*slepc4py.SLEPc.NEP.Type* class method), 208
`__new__()` (*slepc4py.SLEPc.NEP.Which* class method), 209

`__new__()` (*slepc4py.SLEPc.PEP class method*), 313
`__new__()` (*slepc4py.SLEPc.PEP.Basis class method*), 257
`__new__()` (*slepc4py.SLEPc.PEP.CISSExttraction class method*), 258
`__new__()` (*slepc4py.SLEPc.PEP.Conv class method*), 259
`__new__()` (*slepc4py.SLEPc.PEP.ConvergedReason class method*), 260
`__new__()` (*slepc4py.SLEPc.PEP.ErrorType class method*), 260
`__new__()` (*slepc4py.SLEPc.PEP.Extract class method*), 261
`__new__()` (*slepc4py.SLEPc.PEP.JDProjection class method*), 262
`__new__()` (*slepc4py.SLEPc.PEP.ProblemType class method*), 263
`__new__()` (*slepc4py.SLEPc.PEP.Refine class method*), 263
`__new__()` (*slepc4py.SLEPc.PEP.RefineScheme class method*), 264
`__new__()` (*slepc4py.SLEPc.PEP.Scale class method*), 265
`__new__()` (*slepc4py.SLEPc.PEP.Stop class method*), 266
`__new__()` (*slepc4py.SLEPc.PEP.Type class method*), 267
`__new__()` (*slepc4py.SLEPc.PEP.Which class method*), 268
`__new__()` (*slepc4py.SLEPc.RG class method*), 326
`__new__()` (*slepc4py.SLEPc.RG.QuadRule class method*), 314
`__new__()` (*slepc4py.SLEPc.RG.Type class method*), 314
`__new__()` (*slepc4py.SLEPc.ST class method*), 349
`__new__()` (*slepc4py.SLEPc.ST.FilterDamping class method*), 327
`__new__()` (*slepc4py.SLEPc.ST.FilterType class method*), 328
`__new__()` (*slepc4py.SLEPc.ST.MatMode class method*), 329
`__new__()` (*slepc4py.SLEPc.ST.Type class method*), 330
`__new__()` (*slepc4py.SLEPc.SVD class method*), 388
`__new__()` (*slepc4py.SLEPc.SVD.Conv class method*), 351
`__new__()` (*slepc4py.SLEPc.SVD.ConvergedReason class method*), 352
`__new__()` (*slepc4py.SLEPc.SVD.ErrorType class method*), 352
`__new__()` (*slepc4py.SLEPc.SVD.ProblemType class method*), 353
`__new__()` (*slepc4py.SLEPc.SVD.Stop class method*), 354
`__new__()` (*slepc4py.SLEPc.SVD.TRLanczosGBidiag class method*), 355

`__new__()` (*slepc4py.SLEPc.SVD.Type class method*), 356
`__new__()` (*slepc4py.SLEPc.SVD.Which class method*), 357
`__new__()` (*slepc4py.SLEPc.Sys class method*), 390
`__new__()` (*slepc4py.SLEPc.Util class method*), 391

A

A (*slepc4py.SLEPc.DS.MatType attribute*), 53
ABS (*slepc4py.SLEPc.EPS.Conv attribute*), 79
ABS (*slepc4py.SLEPc.NEP.Conv attribute*), 202
ABS (*slepc4py.SLEPc.PEP.Conv attribute*), 258
ABS (*slepc4py.SLEPc.SVD.Conv attribute*), 350
ABSOLUTE (*slepc4py.SLEPc.EPS.ErrorType attribute*), 81
ABSOLUTE (*slepc4py.SLEPc.NEP.ErrorType attribute*), 204
ABSOLUTE (*slepc4py.SLEPc.PEP.ErrorType attribute*), 260
ABSOLUTE (*slepc4py.SLEPc.SVD.ErrorType attribute*), 352
ADD (*slepc4py.SLEPc.FN.CombineType attribute*), 157
ALL (*slepc4py.SLEPc.EPS.Which attribute*), 90
ALL (*slepc4py.SLEPc.NEP.Which attribute*), 209
ALL (*slepc4py.SLEPc.PEP.Which attribute*), 267
allocate() (*slepc4py.SLEPc.DS method*), 58
ALWAYS (*slepc4py.SLEPc.BV.OrthogRefineType attribute*), 16
appendOptionsPrefix() (*slepc4py.SLEPc.BV method*), 21
appendOptionsPrefix() (*slepc4py.SLEPc.DS method*), 59
appendOptionsPrefix() (*slepc4py.SLEPc.EPS method*), 96
appendOptionsPrefix() (*slepc4py.SLEPc.FN method*), 160
appendOptionsPrefix() (*slepc4py.SLEPc.LME method*), 174
appendOptionsPrefix() (*slepc4py.SLEPc.MFN method*), 190
appendOptionsPrefix() (*slepc4py.SLEPc.NEP method*), 213
appendOptionsPrefix() (*slepc4py.SLEPc.PEP method*), 272
appendOptionsPrefix() (*slepc4py.SLEPc.RG method*), 315
appendOptionsPrefix() (*slepc4py.SLEPc.ST method*), 332
appendOptionsPrefix() (*slepc4py.SLEPc.SVD method*), 359
apply() (*slepc4py.SLEPc.ST method*), 332
applyHermitianTranspose() (*slepc4py.SLEPc.ST method*), 332
applyMat() (*slepc4py.SLEPc.ST method*), 333
applyMatrix() (*slepc4py.SLEPc.BV method*), 21

applyResolvent() (*slepc4py.SLEPc.NEP method*), 214
 applyTranspose() (*slepc4py.SLEPc.ST method*), 333
 ARNOLDI (*slepc4py.SLEPc.EPS.Type attribute*), 88
 ARPACK (*slepc4py.SLEPc.EPS.Type attribute*), 88
 ArrayBool (*in module slepc4py.typing*), 10
 ArrayComplex (*in module slepc4py.typing*), 11
 ArrayInt (*in module slepc4py.typing*), 10
 ArrayReal (*in module slepc4py.typing*), 10
 ArrayScalar (*in module slepc4py.typing*), 11

B

B (*slepc4py.SLEPc.DS.MatType attribute*), 53
 BACKWARD (*slepc4py.SLEPc.EPS.ErrorType attribute*), 81
 BACKWARD (*slepc4py.SLEPc.NEP.ErrorType attribute*), 204
 BACKWARD (*slepc4py.SLEPc.PEP.ErrorType attribute*), 260
 Balance (*class in slepc4py.SLEPc.EPS*), 76
 BASIC (*slepc4py.SLEPc.EPS.Stop attribute*), 86
 BASIC (*slepc4py.SLEPc.NEP.Stop attribute*), 207
 BASIC (*slepc4py.SLEPc.PEP.Stop attribute*), 265
 BASIC (*slepc4py.SLEPc.SVD.Stop attribute*), 354
 Basis (*class in slepc4py.SLEPc.PEP*), 256
 block_size (*slepc4py.SLEPc.DS attribute*), 75
 BLOPEX (*slepc4py.SLEPc.EPS.Type attribute*), 88
 BOTH (*slepc4py.SLEPc.PEP.Scale attribute*), 265
 BSE (*slepc4py.SLEPc.EPS.ProblemType attribute*), 85
 BV (*class in slepc4py.SLEPc*), 14
 bv (*slepc4py.SLEPc.EPS attribute*), 155
 bv (*slepc4py.SLEPc.LME attribute*), 187
 bv (*slepc4py.SLEPc.MFN attribute*), 200
 bv (*slepc4py.SLEPc.NEP attribute*), 255
 bv (*slepc4py.SLEPc.PEP attribute*), 312

C

C (*slepc4py.SLEPc.DS.MatType attribute*), 53
 CAA (*slepc4py.SLEPc.NEP.CISSExtraction attribute*), 202
 CAA (*slepc4py.SLEPc.PEP.CISSExtraction attribute*), 258
 cancelMonitor() (*slepc4py.SLEPc.EPS method*), 96
 cancelMonitor() (*slepc4py.SLEPc.LME method*), 175
 cancelMonitor() (*slepc4py.SLEPc.MFN method*), 190
 cancelMonitor() (*slepc4py.SLEPc.NEP method*), 214
 cancelMonitor() (*slepc4py.SLEPc.PEP method*), 272
 cancelMonitor() (*slepc4py.SLEPc.SVD method*), 360
 canUseConjugates() (*slepc4py.SLEPc.RG method*), 316
 CAYLEY (*slepc4py.SLEPc.ST.Type attribute*), 329
 CGS (*slepc4py.SLEPc.BV.OrthogType attribute*), 17
 CHASE (*slepc4py.SLEPc.EPS.Type attribute*), 88
 CHEBYSHEV (*slepc4py.SLEPc.EPS.CISSQuadRule attribute*), 78
 CHEBYSHEV (*slepc4py.SLEPc.RG.QuadRule attribute*), 313

CHEBYSHEV (*slepc4py.SLEPc.ST.FilterType attribute*), 328
 CHEBYSHEV1 (*slepc4py.SLEPc.PEP.Basis attribute*), 257
 CHEBYSHEV2 (*slepc4py.SLEPc.PEP.Basis attribute*), 257
 checkInside() (*slepc4py.SLEPc.RG method*), 316
 CHOL (*slepc4py.SLEPc.BV.OrthogBlockType attribute*), 15
 CISS (*slepc4py.SLEPc.EPS.Type attribute*), 88
 CISS (*slepc4py.SLEPc.NEP.Type attribute*), 208
 CISS (*slepc4py.SLEPc.PEP.Type attribute*), 266
 CISSExtraction (*class in slepc4py.SLEPc.EPS*), 77
 CISSExtraction (*class in slepc4py.SLEPc.NEP*), 201
 CISSExtraction (*class in slepc4py.SLEPc.PEP*), 257
 CISSQuadRule (*class in slepc4py.SLEPc.EPS*), 78
 column_size (*slepc4py.SLEPc.BV attribute*), 52
 COMBINE (*slepc4py.SLEPc.FN.Type attribute*), 159
 CombineType (*class in slepc4py.SLEPc.FN*), 157
 compact (*slepc4py.SLEPc.DS attribute*), 75
 complement (*slepc4py.SLEPc.RG attribute*), 326
 COMPOSE (*slepc4py.SLEPc.FN.CombineType attribute*), 157
 computeBoundingBox() (*slepc4py.SLEPc.RG method*), 317
 computeContour() (*slepc4py.SLEPc.RG method*), 317
 computeError() (*slepc4py.SLEPc.EPS method*), 96
 computeError() (*slepc4py.SLEPc.LME method*), 175
 computeError() (*slepc4py.SLEPc.NEP method*), 214
 computeError() (*slepc4py.SLEPc.PEP method*), 272
 computeError() (*slepc4py.SLEPc.SVD method*), 360
 computeQuadrature() (*slepc4py.SLEPc.RG method*), 317
 cond() (*slepc4py.SLEPc.DS method*), 59
 CONDENSED (*slepc4py.SLEPc.DS.StateType attribute*), 55
 CONSTANT (*slepc4py.SLEPc.EPS.PowerShiftType attribute*), 84
 CONTIGUOUS (*slepc4py.SLEPc.BV.Type attribute*), 18
 Conv (*class in slepc4py.SLEPc.EPS*), 78
 Conv (*class in slepc4py.SLEPc.NEP*), 202
 Conv (*class in slepc4py.SLEPc.PEP*), 258
 Conv (*class in slepc4py.SLEPc.SVD*), 350
 CONVERGED_ITERATING (*slepc4py.SLEPc.EPS.ConvergedReason attribute*), 80
 CONVERGED_ITERATING (*slepc4py.SLEPc.LME.ConvergedReason attribute*), 171
 CONVERGED_ITERATING (*slepc4py.SLEPc.MFN.ConvergedReason attribute*), 188
 CONVERGED_ITERATING (*slepc4py.SLEPc.NEP.ConvergedReason attribute*), 203
 CONVERGED_ITERATING (*slepc4py.SLEPc.PEP.ConvergedReason attribute*), 203

attribute), 259
 CONVERGED_ITERATING (slepc4py.SLEPc.SVD.ConvergedReason attribute), 351
 CONVERGED_ITS (slepc4py.SLEPc.MFN.ConvergedReason attribute), 188
 CONVERGED_MAXIT (slepc4py.SLEPc.SVD.ConvergedReason attribute), 351
 CONVERGED_TOL (slepc4py.SLEPc.EPS.ConvergedReason attribute), 80
 CONVERGED_TOL (slepc4py.SLEPc.LME.ConvergedReason attribute), 171
 CONVERGED_TOL (slepc4py.SLEPc.MFN.ConvergedReason attribute), 188
 CONVERGED_TOL (slepc4py.SLEPc.NEP.ConvergedReason attribute), 203
 CONVERGED_TOL (slepc4py.SLEPc.PEP.ConvergedReason attribute), 259
 CONVERGED_TOL (slepc4py.SLEPc.SVD.ConvergedReason attribute), 351
 CONVERGED_USER (slepc4py.SLEPc.EPS.ConvergedReason attribute), 80
 CONVERGED_USER (slepc4py.SLEPc.NEP.ConvergedReason attribute), 203
 CONVERGED_USER (slepc4py.SLEPc.PEP.ConvergedReason attribute), 259
 CONVERGED_USER (slepc4py.SLEPc.SVD.ConvergedReason attribute), 351
 ConvergedReason (class in slepc4py.SLEPc.EPS), 79
 ConvergedReason (class in slepc4py.SLEPc.LME), 171
 ConvergedReason (class in slepc4py.SLEPc.MFN), 187
 ConvergedReason (class in slepc4py.SLEPc.NEP), 203
 ConvergedReason (class in slepc4py.SLEPc.PEP), 259
 ConvergedReason (class in slepc4py.SLEPc.SVD), 351
 COPY (slepc4py.SLEPc.ST.MatMode attribute), 329
 copy() (slepc4py.SLEPc.BV method), 22
 copyColumn() (slepc4py.SLEPc.BV method), 22
 copyVec() (slepc4py.SLEPc.BV method), 22
 create() (slepc4py.SLEPc.BV method), 23
 create() (slepc4py.SLEPc.DS method), 59
 create() (slepc4py.SLEPc.EPS method), 97
 create() (slepc4py.SLEPc.FN method), 161
 create() (slepc4py.SLEPc.LME method), 175
 create() (slepc4py.SLEPc.MFN method), 191
 create() (slepc4py.SLEPc.NEP method), 215
 create() (slepc4py.SLEPc.PEP method), 273
 create() (slepc4py.SLEPc.RG method), 318
 create() (slepc4py.SLEPc.ST method), 333
 create() (slepc4py.SLEPc.SVD method), 360
 createFromMat() (slepc4py.SLEPc.BV method), 23
 createMat() (slepc4py.SLEPc.BV method), 24
 createMatBSE() (slepc4py.SLEPc.Util class method), 390
 createMatHamiltonian() (slepc4py.SLEPc.Util class method), 390
 createMatLREP() (slepc4py.SLEPc.Util class method), 391
 createVec() (slepc4py.SLEPc.BV method), 24
 CROSS (slepc4py.SLEPc.SVD.Type attribute), 355
 CURRENT (in module slepc4py.SLEPc), 392
 CYCLIC (slepc4py.SLEPc.SVD.Type attribute), 355

D

D (slepc4py.SLEPc.DS.MatType attribute), 53
 DECIDE (in module slepc4py.SLEPc), 392
 DEFAULT (in module slepc4py.SLEPc), 392
 DELAYED (slepc4py.SLEPc.EPS.LanczosReorthogType attribute), 83
 destroy() (slepc4py.SLEPc.BV method), 24
 destroy() (slepc4py.SLEPc.DS method), 60
 destroy() (slepc4py.SLEPc.EPS method), 97
 destroy() (slepc4py.SLEPc.FN method), 161
 destroy() (slepc4py.SLEPc.LME method), 176
 destroy() (slepc4py.SLEPc.MFN method), 191
 destroy() (slepc4py.SLEPc.NEP method), 215
 destroy() (slepc4py.SLEPc.PEP method), 273
 destroy() (slepc4py.SLEPc.RG method), 318
 destroy() (slepc4py.SLEPc.ST method), 334
 destroy() (slepc4py.SLEPc.SVD method), 361
 DETERMINE (in module slepc4py.SLEPc), 392
 DIAGONAL (slepc4py.SLEPc.PEP.Scale attribute), 265
 DISTRIBUTED (slepc4py.SLEPc.DS.ParallelType attribute), 54
 DIVERGED_BREAKDOWN (slepc4py.SLEPc.EPS.ConvergedReason attribute), 80
 DIVERGED_BREAKDOWN (slepc4py.SLEPc.LME.ConvergedReason attribute), 171
 DIVERGED_BREAKDOWN (slepc4py.SLEPc.MFN.ConvergedReason attribute), 188
 DIVERGED_BREAKDOWN (slepc4py.SLEPc.NEP.ConvergedReason attribute), 203
 DIVERGED_BREAKDOWN (slepc4py.SLEPc.PEP.ConvergedReason attribute), 259
 DIVERGED_BREAKDOWN (slepc4py.SLEPc.SVD.ConvergedReason attribute), 351
 DIVERGED_ITS (slepc4py.SLEPc.EPS.ConvergedReason attribute), 80
 DIVERGED_ITS (slepc4py.SLEPc.LME.ConvergedReason attribute), 171
 DIVERGED_ITS (slepc4py.SLEPc.MFN.ConvergedReason attribute), 188
 DIVERGED_ITS (slepc4py.SLEPc.NEP.ConvergedReason attribute), 203
 DIVERGED_ITS (slepc4py.SLEPc.PEP.ConvergedReason attribute), 259
 DIVERGED_ITS (slepc4py.SLEPc.SVD.ConvergedReason attribute), 351

DIVERGED_LINEAR_SOLVE
 (*slepc4py.SLEPc.NEP.ConvergedReason*
 attribute), 203

DIVERGED_SUBSPACE_EXHAUSTED
 (*slepc4py.SLEPc.NEP.ConvergedReason*
 attribute), 204

DIVERGED_SYMMETRY_LOST
 (*slepc4py.SLEPc.EPS.ConvergedReason*
 attribute), 80

DIVERGED_SYMMETRY_LOST
 (*slepc4py.SLEPc.PEP.ConvergedReason*
 attribute), 260

DIVERGED_SYMMETRY_LOST
 (*slepc4py.SLEPc.SVD.ConvergedReason*
 attribute), 352

DIVIDE (*slepc4py.SLEPc.FN.CombineType attribute*),
 157

dot() (*slepc4py.SLEPc.BV method*), 25

dotColumn() (*slepc4py.SLEPc.BV method*), 25

dotVec() (*slepc4py.SLEPc.BV method*), 26

DS (*class in slepc4py.SLEPc*), 52

ds (*slepc4py.SLEPc.EPS attribute*), 155

ds (*slepc4py.SLEPc.NEP attribute*), 255

ds (*slepc4py.SLEPc.PEP attribute*), 312

ds (*slepc4py.SLEPc.SVD attribute*), 387

DT_LYAPUNOV (*slepc4py.SLEPc.LME.ProblemType at-*
 tribute), 172

duplicate() (*slepc4py.SLEPc.BV method*), 26

duplicate() (*slepc4py.SLEPc.DS method*), 60

duplicate() (*slepc4py.SLEPc.FN method*), 161

duplicateResize() (*slepc4py.SLEPc.BV method*), 26

E

ELEMENTAL (*slepc4py.SLEPc.EPS.Type attribute*), 88

ELEMENTAL (*slepc4py.SLEPc.SVD.Type attribute*), 356

ELLIPSE (*slepc4py.SLEPc.RG.Type attribute*), 314

ELPA (*slepc4py.SLEPc.EPS.Type attribute*), 88

environment variable
 NUMPY_INCLUDE, 5

PETSC_ARCH, 5

PETSC_DIR, 5

SLEPC_DIR, 5

EPS (*class in slepc4py.SLEPc*), 76

EPSSArbitraryFunction (*in module slepc4py.typing*),
 11

EPSEigenvalueComparison (*in module*
 slepc4py.typing), 11

EPSPMonitorFunction (*in module slepc4py.typing*), 11

EPSSStoppingFunction (*in module slepc4py.typing*), 11

ErrorType (*class in slepc4py.SLEPc.EPS*), 80

ErrorType (*class in slepc4py.SLEPc.NEP*), 204

ErrorType (*class in slepc4py.SLEPc.PEP*), 260

ErrorType (*class in slepc4py.SLEPc.SVD*), 352

errorView() (*slepc4py.SLEPc.EPS method*), 97

getArnoldiDelayed() (slepc4py.SLEPc.EPS method), 98	getConvergedReason() (slepc4py.SLEPc.EPS method), 102	(slepc4py.SLEPc.EPS
getArray() (slepc4py.SLEPc.BV method), 27	getConvergedReason() (slepc4py.SLEPc.LME method), 177	method), 177
getArray() (slepc4py.SLEPc.DS method), 60	getConvergedReason() (slepc4py.SLEPc.MFN method), 191	method), 191
getBalance() (slepc4py.SLEPc.EPS method), 99	getConvergedReason() (slepc4py.SLEPc.NEP method), 219	method), 219
getBasis() (slepc4py.SLEPc.PEP method), 274	getConvergedReason() (slepc4py.SLEPc.PEP method), 277	method), 277
getBlockSize() (slepc4py.SLEPc.DS method), 61	getConvergedReason() (slepc4py.SLEPc.SVD method), 362	method), 362
getBV() (slepc4py.SLEPc.EPS method), 98	getConvergenceTest() (slepc4py.SLEPc.EPS method), 102	method), 102
getBV() (slepc4py.SLEPc.LME method), 176	getConvergenceTest() (slepc4py.SLEPc.NEP method), 219	method), 219
getBV() (slepc4py.SLEPc.MFN method), 191	getConvergenceTest() (slepc4py.SLEPc.PEP method), 277	method), 277
getBV() (slepc4py.SLEPc.NEP method), 216	getConvergenceTest() (slepc4py.SLEPc.SVD method), 363	method), 363
getBV() (slepc4py.SLEPc.PEP method), 274	getCrossePS() (slepc4py.SLEPc.SVD method), 363	method), 363
getBV() (slepc4py.SLEPc.SVD method), 362	getCrossExplicitMatrix() (slepc4py.SLEPc.SVD method), 363	method), 363
getCayleyAntishift() (slepc4py.SLEPc.ST method), 334	getCyclicEPS() (slepc4py.SLEPc.SVD method), 364	method), 364
getCISSExtraction() (slepc4py.SLEPc.EPS method), 99	getCyclicExplicitMatrix() (slepc4py.SLEPc.SVD method), 364	method), 364
getCISSExtraction() (slepc4py.SLEPc.NEP method), 216	getDefiniteTolerance() (slepc4py.SLEPc.BV method), 28	method), 28
getCISSExtraction() (slepc4py.SLEPc.PEP method), 274	getDimensions() (slepc4py.SLEPc.DS method), 61	method), 61
getCISSKSPs() (slepc4py.SLEPc.EPS method), 99	getDimensions() (slepc4py.SLEPc.EPS method), 103	method), 103
getCISSKSPs() (slepc4py.SLEPc.NEP method), 217	getDimensions() (slepc4py.SLEPc.LME method), 177	method), 177
getCISSKSPs() (slepc4py.SLEPc.PEP method), 275	getDimensions() (slepc4py.SLEPc.MFN method), 192	method), 192
getCISSThreshold() (slepc4py.SLEPc.EPS method), 100	getDimensions() (slepc4py.SLEPc.NEP method), 220	method), 220
getCISSThreshold() (slepc4py.SLEPc.EPS method), 100	getDimensions() (slepc4py.SLEPc.PEP method), 278	method), 278
getCISSThreshold() (slepc4py.SLEPc.NEP method), 217	getDimensions() (slepc4py.SLEPc.SVD method), 365	method), 365
getCISSThreshold() (slepc4py.SLEPc.PEP method), 275	getDS() (slepc4py.SLEPc.EPS method), 102	method), 102
getCISSThreshold() (slepc4py.SLEPc.EPS method), 100	getDS() (slepc4py.SLEPc.NEP method), 219	method), 219
getCISSThreshold() (slepc4py.SLEPc.NEP method), 218	getDS() (slepc4py.SLEPc.PEP method), 277	method), 277
getCISSThreshold() (slepc4py.SLEPc.PEP method), 276	getDS() (slepc4py.SLEPc.SVD method), 364	method), 364
getCISSThreshold() (slepc4py.SLEPc.PEP method), 276	getEigenpair() (slepc4py.SLEPc.EPS method), 103	method), 103
getCISSThreshold() (slepc4py.SLEPc.EPS method), 101	getEigenpair() (slepc4py.SLEPc.NEP method), 220	method), 220
getCISSThreshold() (slepc4py.SLEPc.NEP method), 218	getEigenpair() (slepc4py.SLEPc.PEP method), 278	method), 278
getCISSThreshold() (slepc4py.SLEPc.PEP method), 276	getEigenvalue() (slepc4py.SLEPc.EPS method), 104	method), 104
getCISSThreshold() (slepc4py.SLEPc.EPS method), 101	getEigenvalueComparison() (slepc4py.SLEPc.EPS method), 104	method), 104
getCoefficients() (slepc4py.SLEPc.LME method), 176	getEigenvalueComparison() (slepc4py.SLEPc.NEP method), 220	method), 220
getColumn() (slepc4py.SLEPc.BV method), 28	getEigenvalueComparison() (slepc4py.SLEPc.PEP method), 278	method), 278
getCombineChildren() (slepc4py.SLEPc.FN method), 163	getEigenvector() (slepc4py.SLEPc.EPS method), 105	method), 105
getCompact() (slepc4py.SLEPc.DS method), 61	getEllipseParameters() (slepc4py.SLEPc.RG method), 319	method), 319
getComplement() (slepc4py.SLEPc.RG method), 319	getErrorEstimate() (slepc4py.SLEPc.EPS method), 105	method), 105
getConverged() (slepc4py.SLEPc.EPS method), 101		
getConverged() (slepc4py.SLEPc.NEP method), 218		
getConverged() (slepc4py.SLEPc.PEP method), 276		
getConverged() (slepc4py.SLEPc.SVD method), 362		

<code>getErrorEstimate()</code> (<i>slepc4py.SLEPc.LME method</i>), 177	<code>getIterationNumber()</code> (<i>slepc4py.SLEPc.PEP method</i>), 280
<code>getErrorEstimate()</code> (<i>slepc4py.SLEPc.NEP method</i>), 221	<code>getIterationNumber()</code> (<i>slepc4py.SLEPc.SVD method</i>), 365
<code>getErrorEstimate()</code> (<i>slepc4py.SLEPc.PEP method</i>), 279	<code>getJacobian()</code> (<i>slepc4py.SLEPc.NEP method</i>), 223
<code>getErrorIfNotConverged()</code> (<i>slepc4py.SLEPc.LME method</i>), 178	<code>getJDBlockSize()</code> (<i>slepc4py.SLEPc.EPS method</i>), 110
<code>getErrorIfNotConverged()</code> (<i>slepc4py.SLEPc.MFN method</i>), 192	<code>getJDBOrth()</code> (<i>slepc4py.SLEPc.EPS method</i>), 109
<code>getExtract()</code> (<i>slepc4py.SLEPc.PEP method</i>), 279	<code>getJDConstCorrectionTol()</code> (<i>slepc4py.SLEPc.EPS method</i>), 110
<code>getExtraction()</code> (<i>slepc4py.SLEPc.EPS method</i>), 106	<code>getJDFix()</code> (<i>slepc4py.SLEPc.EPS method</i>), 110
<code>getExtraRow()</code> (<i>slepc4py.SLEPc.DS method</i>), 62	<code>getJDFix()</code> (<i>slepc4py.SLEPc.PEP method</i>), 280
<code>getFilterDamping()</code> (<i>slepc4py.SLEPc.ST method</i>), 334	<code>getJDInitialSize()</code> (<i>slepc4py.SLEPc.EPS method</i>), 111
<code>getFilterDegree()</code> (<i>slepc4py.SLEPc.ST method</i>), 335	<code>getJDKrylovStart()</code> (<i>slepc4py.SLEPc.EPS method</i>), 111
<code>getFilterInterval()</code> (<i>slepc4py.SLEPc.ST method</i>), 335	<code>getJDMinimalityIndex()</code> (<i>slepc4py.SLEPc.PEP method</i>), 281
<code>getFilterRange()</code> (<i>slepc4py.SLEPc.ST method</i>), 335	<code>getJDProjection()</code> (<i>slepc4py.SLEPc.PEP method</i>), 281
<code>getFilterType()</code> (<i>slepc4py.SLEPc.ST method</i>), 336	<code>getJDRestart()</code> (<i>slepc4py.SLEPc.EPS method</i>), 111
<code>getFN()</code> (<i>slepc4py.SLEPc.MFN method</i>), 192	<code>getJDRestart()</code> (<i>slepc4py.SLEPc.PEP method</i>), 281
<code>getFunction()</code> (<i>slepc4py.SLEPc.NEP method</i>), 221	<code>getJDReusePreconditioner()</code> (<i>slepc4py.SLEPc.PEP method</i>), 282
<code>getGDBlockSize()</code> (<i>slepc4py.SLEPc.EPS method</i>), 106	<code>getKrylovSchurBSEType()</code> (<i>slepc4py.SLEPc.EPS method</i>), 112
<code>getGDBOrth()</code> (<i>slepc4py.SLEPc.EPS method</i>), 106	<code>getKrylovSchurDetectZeros()</code> (<i>slepc4py.SLEPc.EPS method</i>), 112
<code>getGDDoubleExpansion()</code> (<i>slepc4py.SLEPc.EPS method</i>), 107	<code>getKrylovSchurDimensions()</code> (<i>slepc4py.SLEPc.EPS method</i>), 112
<code>getGDInitialSize()</code> (<i>slepc4py.SLEPc.EPS method</i>), 107	<code>getKrylovSchurInertias()</code> (<i>slepc4py.SLEPc.EPS method</i>), 113
<code>getGDKrylovStart()</code> (<i>slepc4py.SLEPc.EPS method</i>), 107	<code>getKrylovSchurKSP()</code> (<i>slepc4py.SLEPc.EPS method</i>), 113
<code>getGDRestart()</code> (<i>slepc4py.SLEPc.EPS method</i>), 108	<code>getKrylovSchurLocking()</code> (<i>slepc4py.SLEPc.EPS method</i>), 114
<code>getGSVDDimensions()</code> (<i>slepc4py.SLEPc.DS method</i>), 62	<code>getKrylovSchurPartitions()</code> (<i>slepc4py.SLEPc.EPS method</i>), 114
<code>getHSVDDimensions()</code> (<i>slepc4py.SLEPc.DS method</i>), 62	<code>getKrylovSchurRestart()</code> (<i>slepc4py.SLEPc.EPS method</i>), 114
<code>getImplicitTranspose()</code> (<i>slepc4py.SLEPc.SVD method</i>), 365	<code>getKrylovSchurSubcommInfo()</code> (<i>slepc4py.SLEPc.EPS method</i>), 115
<code>getInterpolInterpolation()</code> (<i>slepc4py.SLEPc.NEP method</i>), 222	<code>getKrylovSchurSubcommMats()</code> (<i>slepc4py.SLEPc.EPS method</i>), 115
<code>getInterpolPEP()</code> (<i>slepc4py.SLEPc.NEP method</i>), 222	<code>getKrylovSchurSubcommPairs()</code> (<i>slepc4py.SLEPc.EPS method</i>), 116
<code>getInterval()</code> (<i>slepc4py.SLEPc.EPS method</i>), 108	<code>getKrylovSchurSubintervals()</code> (<i>slepc4py.SLEPc.EPS method</i>), 116
<code>getInterval()</code> (<i>slepc4py.SLEPc.PEP method</i>), 279	<code>getKSP()</code> (<i>slepc4py.SLEPc.ST method</i>), 336
<code>getIntervalEndpoints()</code> (<i>slepc4py.SLEPc.RG method</i>), 319	<code>getLanczosOneSide()</code> (<i>slepc4py.SLEPc.SVD method</i>), 366
<code>getInvariantSubspace()</code> (<i>slepc4py.SLEPc.EPS method</i>), 108	<code>getLanczosReorthogType()</code> (<i>slepc4py.SLEPc.EPS method</i>), 118
<code>getIterationNumber()</code> (<i>slepc4py.SLEPc.EPS method</i>), 109	<code>getLeadingDimension()</code> (<i>slepc4py.SLEPc.BV method</i>), 28
<code>getIterationNumber()</code> (<i>slepc4py.SLEPc.LME method</i>), 178	
<code>getIterationNumber()</code> (<i>slepc4py.SLEPc.MFN method</i>), 193	
<code>getIterationNumber()</code> (<i>slepc4py.SLEPc.NEP method</i>), 222	

getLeadingDimension() (*slepc4py.SLEPc.DS method*), 63
 getLeftEigenvector() (*slepc4py.SLEPc.EPS method*), 118
 getLeftEigenvector() (*slepc4py.SLEPc.NEP method*), 223
 getLinearEPS() (*slepc4py.SLEPc.PEP method*), 282
 getLinearExplicitMatrix() (*slepc4py.SLEPc.PEP method*), 282
 getLinearLinearization() (*slepc4py.SLEPc.PEP method*), 282
 getLOBPCGBlockSize() (*slepc4py.SLEPc.EPS method*), 117
 getLOBPCGLocking() (*slepc4py.SLEPc.EPS method*), 117
 getLOBPCGRestart() (*slepc4py.SLEPc.EPS method*), 117
 getLyapIIRanks() (*slepc4py.SLEPc.EPS method*), 119
 getMat() (*slepc4py.SLEPc.BV method*), 29
 getMat() (*slepc4py.SLEPc.DS method*), 63
 getMatMode() (*slepc4py.SLEPc.ST method*), 336
 getMatMultMethod() (*slepc4py.SLEPc.BV method*), 29
 getMatrices() (*slepc4py.SLEPc.ST method*), 337
 getMatrix() (*slepc4py.SLEPc.BV method*), 30
 getMatStructure() (*slepc4py.SLEPc.ST method*), 337
 getMethod() (*slepc4py.SLEPc.DS method*), 63
 getMethod() (*slepc4py.SLEPc.FN method*), 164
 getMonitor() (*slepc4py.SLEPc.EPS method*), 119
 getMonitor() (*slepc4py.SLEPc.LME method*), 178
 getMonitor() (*slepc4py.SLEPc.MFN method*), 193
 getMonitor() (*slepc4py.SLEPc.NEP method*), 223
 getMonitor() (*slepc4py.SLEPc.PEP method*), 283
 getMonitor() (*slepc4py.SLEPc.SVD method*), 366
 getNArnoldiKSP() (*slepc4py.SLEPc.NEP method*), 224
 getNArnoldiLagPreconditioner() (*slepc4py.SLEPc.NEP method*), 224
 getNLEIGSEPS() (*slepc4py.SLEPc.NEP method*), 224
 getNLEIGSFullBasis() (*slepc4py.SLEPc.NEP method*), 225
 getNLEIGSInterpolation() (*slepc4py.SLEPc.NEP method*), 225
 getNLEIGSKSPs() (*slepc4py.SLEPc.NEP method*), 225
 getNLEIGSLocking() (*slepc4py.SLEPc.NEP method*), 226
 getNLEIGSRestart() (*slepc4py.SLEPc.NEP method*), 226
 getNLEIGSRKShifts() (*slepc4py.SLEPc.NEP method*), 226
 getNumConstraints() (*slepc4py.SLEPc.BV method*), 30
 getOperator() (*slepc4py.SLEPc.MFN method*), 193
 getOperator() (*slepc4py.SLEPc.ST method*), 337
 getOperators() (*slepc4py.SLEPc.EPS method*), 119
 getOperators() (*slepc4py.SLEPc.PEP method*), 283
 getOperators() (*slepc4py.SLEPc.SVD method*), 366
 getOptionsPrefix() (*slepc4py.SLEPc.BV method*), 30
 getOptionsPrefix() (*slepc4py.SLEPc.DS method*), 64
 getOptionsPrefix() (*slepc4py.SLEPc.EPS method*), 120
 getOptionsPrefix() (*slepc4py.SLEPc.FN method*), 164
 getOptionsPrefix() (*slepc4py.SLEPc.LME method*), 178
 getOptionsPrefix() (*slepc4py.SLEPc.MFN method*), 194
 getOptionsPrefix() (*slepc4py.SLEPc.NEP method*), 227
 getOptionsPrefix() (*slepc4py.SLEPc.PEP method*), 283
 getOptionsPrefix() (*slepc4py.SLEPc.RG method*), 320
 getOptionsPrefix() (*slepc4py.SLEPc.ST method*), 338
 getOptionsPrefix() (*slepc4py.SLEPc.SVD method*), 367
 getOrthogonalization() (*slepc4py.SLEPc.BV method*), 31
 getParallel() (*slepc4py.SLEPc.DS method*), 65
 getParallel() (*slepc4py.SLEPc.FN method*), 164
 getPEPCoefficients() (*slepc4py.SLEPc.DS method*), 64
 getPEPDegree() (*slepc4py.SLEPc.DS method*), 64
 getPhiIndex() (*slepc4py.SLEPc.FN method*), 165
 getPolygonVertices() (*slepc4py.SLEPc.RG method*), 320
 getPowerShiftType() (*slepc4py.SLEPc.EPS method*), 120
 getPreconditionerMat() (*slepc4py.SLEPc.ST method*), 338
 getProblemType() (*slepc4py.SLEPc.EPS method*), 120
 getProblemType() (*slepc4py.SLEPc.LME method*), 179
 getProblemType() (*slepc4py.SLEPc.NEP method*), 227
 getProblemType() (*slepc4py.SLEPc.PEP method*), 284
 getProblemType() (*slepc4py.SLEPc.SVD method*), 367
 getPurify() (*slepc4py.SLEPc.EPS method*), 121
 getQArnoldiLocking() (*slepc4py.SLEPc.PEP method*), 284
 getQArnoldiRestart() (*slepc4py.SLEPc.PEP method*), 284
 getRandomContext() (*slepc4py.SLEPc.BV method*), 31
 getRationalDenominator() (*slepc4py.SLEPc.FN method*), 165
 getRationalNumerator() (*slepc4py.SLEPc.FN method*), 165
 getRefine() (*slepc4py.SLEPc.NEP method*), 229
 getRefine() (*slepc4py.SLEPc.PEP method*), 285
 getRefined() (*slepc4py.SLEPc.DS method*), 65

getRefineKSP() (*slepc4py.SLEPc.NEP method*), 230
 getRefineKSP() (*slepc4py.SLEPc.PEP method*), 285
 getRG() (*slepc4py.SLEPc.EPS method*), 121
 getRG() (*slepc4py.SLEPc.NEP method*), 227
 getRG() (*slepc4py.SLEPc.PEP method*), 285
 getRHS() (*slepc4py.SLEPc.LME method*), 179
 getRIIConstCorrectionTol() (*slepc4py.SLEPc.NEP method*), 228
 getRIIDeflationThreshold() (*slepc4py.SLEPc.NEP method*), 228
 getRIIHermitian() (*slepc4py.SLEPc.NEP method*), 228
 getRIIKSP() (*slepc4py.SLEPc.NEP method*), 228
 getRIILagPreconditioner() (*slepc4py.SLEPc.NEP method*), 229
 getRIIMaximumIterations() (*slepc4py.SLEPc.NEP method*), 229
 getRingParameters() (*slepc4py.SLEPc.RG method*), 320
 getRQCGReset() (*slepc4py.SLEPc.EPS method*), 121
 getScale() (*slepc4py.SLEPc.FN method*), 166
 getScale() (*slepc4py.SLEPc.PEP method*), 288
 getScale() (*slepc4py.SLEPc.RG method*), 321
 getShift() (*slepc4py.SLEPc.ST method*), 339
 getSignature() (*slepc4py.SLEPc.SVD method*), 367
 getSingularTriplet() (*slepc4py.SLEPc.SVD method*), 368
 getSizes() (*slepc4py.SLEPc.BV method*), 31
 getSLPDeflationThreshold() (*slepc4py.SLEPc.NEP method*), 230
 getSLPEPS() (*slepc4py.SLEPc.NEP method*), 230
 getSLPEPSLeft() (*slepc4py.SLEPc.NEP method*), 231
 getSLPKSP() (*slepc4py.SLEPc.NEP method*), 231
 getSolution() (*slepc4py.SLEPc.LME method*), 179
 getSplitOperator() (*slepc4py.SLEPc.NEP method*), 231
 getSplitPreconditioner() (*slepc4py.SLEPc.NEP method*), 232
 getSplitPreconditioner() (*slepc4py.SLEPc.ST method*), 339
 getST() (*slepc4py.SLEPc.EPS method*), 121
 getST() (*slepc4py.SLEPc.PEP method*), 286
 getState() (*slepc4py.SLEPc.DS method*), 66
 getSTOARCheckEigenvalueType() (*slepc4py.SLEPc.PEP method*), 286
 getSTOARDetectZeros() (*slepc4py.SLEPc.PEP method*), 286
 getSTOARDimensions() (*slepc4py.SLEPc.PEP method*), 287
 getSTOARInertias() (*slepc4py.SLEPc.PEP method*), 287
 getSTOARLinearization() (*slepc4py.SLEPc.PEP method*), 287
 getSTOARLocking() (*slepc4py.SLEPc.PEP method*), 288
 getStoppingTest() (*slepc4py.SLEPc.EPS method*), 122
 getStoppingTest() (*slepc4py.SLEPc.NEP method*), 232
 getStoppingTest() (*slepc4py.SLEPc.PEP method*), 288
 getStoppingTest() (*slepc4py.SLEPc.SVD method*), 368
 getSVDDimensions() (*slepc4py.SLEPc.DS method*), 65
 getTarget() (*slepc4py.SLEPc.EPS method*), 122
 getTarget() (*slepc4py.SLEPc.NEP method*), 232
 getTarget() (*slepc4py.SLEPc.PEP method*), 289
 getThreshold() (*slepc4py.SLEPc.EPS method*), 122
 getThreshold() (*slepc4py.SLEPc.SVD method*), 370
 getTOARLocking() (*slepc4py.SLEPc.PEP method*), 289
 getTOARRestart() (*slepc4py.SLEPc.PEP method*), 289
 getTolerances() (*slepc4py.SLEPc.EPS method*), 123
 getTolerances() (*slepc4py.SLEPc.LME method*), 180
 getTolerances() (*slepc4py.SLEPc.MFN method*), 194
 getTolerances() (*slepc4py.SLEPc.NEP method*), 233
 getTolerances() (*slepc4py.SLEPc.PEP method*), 290
 getTolerances() (*slepc4py.SLEPc.SVD method*), 371
 getTrackAll() (*slepc4py.SLEPc.EPS method*), 123
 getTrackAll() (*slepc4py.SLEPc.NEP method*), 233
 getTrackAll() (*slepc4py.SLEPc.PEP method*), 290
 getTrackAll() (*slepc4py.SLEPc.SVD method*), 371
 getTransform() (*slepc4py.SLEPc.ST method*), 339
 getTRLanczosExplicitMatrix() (*slepc4py.SLEPc.SVD method*), 368
 getTRLanczosGBidiag() (*slepc4py.SLEPc.SVD method*), 369
 getTRLanczosKSP() (*slepc4py.SLEPc.SVD method*), 369
 getTRLanczosLocking() (*slepc4py.SLEPc.SVD method*), 369
 getTRLanczosOneSide() (*slepc4py.SLEPc.SVD method*), 370
 getTRLanczosRestart() (*slepc4py.SLEPc.SVD method*), 370
 getTrueResidual() (*slepc4py.SLEPc.EPS method*), 123
 getTwoSided() (*slepc4py.SLEPc.EPS method*), 124
 getTwoSided() (*slepc4py.SLEPc.NEP method*), 233
 getType() (*slepc4py.SLEPc.BV method*), 32
 getType() (*slepc4py.SLEPc.DS method*), 66
 getType() (*slepc4py.SLEPc.EPS method*), 124
 getType() (*slepc4py.SLEPc.FN method*), 166
 getType() (*slepc4py.SLEPc.LME method*), 180
 getType() (*slepc4py.SLEPc.MFN method*), 194
 getType() (*slepc4py.SLEPc.NEP method*), 234
 getType() (*slepc4py.SLEPc.PEP method*), 290
 getType() (*slepc4py.SLEPc.RG method*), 321
 getType() (*slepc4py.SLEPc.ST method*), 340

getType() (*slepc4py.SLEPc.SVD method*), 371
 getValue() (*slepc4py.SLEPc.SVD method*), 372
 getVectors() (*slepc4py.SLEPc.SVD method*), 372
 getVecType() (*slepc4py.SLEPc.BV method*), 32
 getVersion() (*slepc4py.SLEPc.Sys class method*), 388
 getVersionInfo() (*slepc4py.SLEPc.Sys class method*), 388
 getWhichEigenpairs() (*slepc4py.SLEPc.EPS method*), 124
 getWhichEigenpairs() (*slepc4py.SLEPc.NEP method*), 234
 getWhichEigenpairs() (*slepc4py.SLEPc.PEP method*), 291
 getWhichSingularTriplets() (*slepc4py.SLEPc.SVD method*), 372
 GHEP (*slepc4py.SLEPc.DS.Type attribute*), 56
 GHEP (*slepc4py.SLEPc.EPS.ProblemType attribute*), 85
 GHIEP (*slepc4py.SLEPc.DS.Type attribute*), 56
 GHIEP (*slepc4py.SLEPc.EPS.ProblemType attribute*), 85
 GNHEP (*slepc4py.SLEPc.DS.Type attribute*), 56
 GNHEP (*slepc4py.SLEPc.EPS.ProblemType attribute*), 85
 GRUNING (*slepc4py.SLEPc.EPS.KrylovSchurBSEType attribute*), 82
 GS (*slepc4py.SLEPc.BV.OrthogBlockType attribute*), 15
 GSVD (*slepc4py.SLEPc.DS.Type attribute*), 56
 GYROSCOPIC (*slepc4py.SLEPc.PEP.ProblemType attribute*), 262

H

HAMILT (*slepc4py.SLEPc.EPS.ProblemType attribute*), 85
 HANKEL (*slepc4py.SLEPc.EPS.CISSExtraction attribute*), 78
 HANKEL (*slepc4py.SLEPc.NEP.CISSExtraction attribute*), 202
 HANKEL (*slepc4py.SLEPc.PEP.CISSExtraction attribute*), 258
 HARMONIC (*slepc4py.SLEPc.EPS.Extraction attribute*), 82
 HARMONIC (*slepc4py.SLEPc.PEP.JDProjection attribute*), 262
 HARMONIC_LARGEST (*slepc4py.SLEPc.EPS.Extraction attribute*), 82
 HARMONIC_RELATIVE (*slepc4py.SLEPc.EPS.Extraction attribute*), 82
 HARMONIC_RIGHT (*slepc4py.SLEPc.EPS.Extraction attribute*), 82
 hasExternalPackage() (*slepc4py.SLEPc.Sys class method*), 389
 HEP (*slepc4py.SLEPc.DS.Type attribute*), 56
 HEP (*slepc4py.SLEPc.EPS.ProblemType attribute*), 85
 HERMITE (*slepc4py.SLEPc.PEP.Basis attribute*), 257
 HERMITIAN (*slepc4py.SLEPc.PEP.ProblemType attribute*), 263
 HSVD (*slepc4py.SLEPc.DS.Type attribute*), 56

HYPERBOLIC (*slepc4py.SLEPc.PEP.ProblemType attribute*), 263
 HYPERBOLIC (*slepc4py.SLEPc.SVD.ProblemType attribute*), 353

I

IFNEEDED (*slepc4py.SLEPc.BV.OrthogRefineType attribute*), 16
 init() (*in module slepc4py*), 9
 INPLACE (*slepc4py.SLEPc.ST.MatMode attribute*), 329
 insertConstraints() (*slepc4py.SLEPc.BV method*), 32
 insertVec() (*slepc4py.SLEPc.BV method*), 33
 insertVecs() (*slepc4py.SLEPc.BV method*), 33
 INTERMEDIATE (*slepc4py.SLEPc.DS.StateType attribute*), 55
 INTERPOL (*slepc4py.SLEPc.NEP.Type attribute*), 208
 INTERVAL (*slepc4py.SLEPc.RG.Type attribute*), 314
 INVSQRT (*slepc4py.SLEPc.FN.Type attribute*), 159
 isAxisymmetric() (*slepc4py.SLEPc.RG method*), 321
 isFinalized() (*slepc4py.SLEPc.Sys class method*), 389
 isGeneralized() (*slepc4py.SLEPc.EPS method*), 125
 isGeneralized() (*slepc4py.SLEPc.SVD method*), 373
 isHermitian() (*slepc4py.SLEPc.EPS method*), 125
 isHyperbolic() (*slepc4py.SLEPc.SVD method*), 373
 isInitialized() (*slepc4py.SLEPc.Sys class method*), 389
 isPositive() (*slepc4py.SLEPc.EPS method*), 125
 isStructured() (*slepc4py.SLEPc.EPS method*), 126
 isTrivial() (*slepc4py.SLEPc.RG method*), 322
 ITERATING (*slepc4py.SLEPc.EPS.ConvergedReason attribute*), 80
 ITERATING (*slepc4py.SLEPc.LME.ConvergedReason attribute*), 172
 ITERATING (*slepc4py.SLEPc.MFN.ConvergedReason attribute*), 188
 ITERATING (*slepc4py.SLEPc.NEP.ConvergedReason attribute*), 204
 ITERATING (*slepc4py.SLEPc.PEP.ConvergedReason attribute*), 260
 ITERATING (*slepc4py.SLEPc.SVD.ConvergedReason attribute*), 352

J

JACKSON (*slepc4py.SLEPc.ST.FilterDamping attribute*), 327
 JD (*slepc4py.SLEPc.EPS.Type attribute*), 88
 JD (*slepc4py.SLEPc.PEP.Type attribute*), 266
 JDProjection (*class in slepc4py.SLEPc.PEP*), 261

K

KRYLOV (*slepc4py.SLEPc.LME.Type attribute*), 173
 KRYLOV (*slepc4py.SLEPc.MFN.Type attribute*), 189
 KRYLOVSCHUR (*slepc4py.SLEPc.EPS.Type attribute*), 88

KrylovSchurBSEType (class in *slepc4py.SLEPc.EPS*), 82
 ksp (*slepc4py.SLEPc.ST* attribute), 349
 KSVD (*slepc4py.SLEPc.SVD.Type* attribute), 356

L

LAGUERRE (*slepc4py.SLEPc.PEP.Basis* attribute), 257
 LANCZOS (*slepc4py.SLEPc.EPS.Type* attribute), 88
 LANCZOS (*slepc4py.SLEPc.ST.FilterDamping* attribute), 327
 LANCZOS (*slepc4py.SLEPc.SVD.Type* attribute), 356
 LanczosReorthogType (class in *slepc4py.SLEPc.EPS*), 83
 LAPACK (*slepc4py.SLEPc.EPS.Type* attribute), 88
 LAPACK (*slepc4py.SLEPc.SVD.Type* attribute), 356
 LARGEST (*slepc4py.SLEPc.SVD.Which* attribute), 356
 LARGEST_IMAGINARY (*slepc4py.SLEPc.EPS.Which* attribute), 90
 LARGEST_IMAGINARY (*slepc4py.SLEPc.NEP.Which* attribute), 209
 LARGEST_IMAGINARY (*slepc4py.SLEPc.PEP.Which* attribute), 267
 LARGEST_MAGNITUDE (*slepc4py.SLEPc.EPS.Which* attribute), 90
 LARGEST_MAGNITUDE (*slepc4py.SLEPc.NEP.Which* attribute), 209
 LARGEST_MAGNITUDE (*slepc4py.SLEPc.PEP.Which* attribute), 268
 LARGEST_REAL (*slepc4py.SLEPc.EPS.Which* attribute), 90
 LARGEST_REAL (*slepc4py.SLEPc.NEP.Which* attribute), 209
 LARGEST_REAL (*slepc4py.SLEPc.PEP.Which* attribute), 268
 LayoutSizeSpec (in module *slepc4py.typing*), 11
 LEGENDRE (*slepc4py.SLEPc.PEP.Basis* attribute), 257
 LINEAR (*slepc4py.SLEPc.PEP.Type* attribute), 266
 LME (class in *slepc4py.SLEPc*), 171
 LMEMonitorFunction (in module *slepc4py.typing*), 13
 LOBPCG (*slepc4py.SLEPc.EPS.Type* attribute), 88
 LOCAL (*slepc4py.SLEPc.EPS.LanczosReorthogType* attribute), 83
 local_size (*slepc4py.SLEPc.BV* attribute), 52
 LOG (*slepc4py.SLEPc.FN.Type* attribute), 159
 LOWER (*slepc4py.SLEPc.SVD.TRLanczosGBidiag* attribute), 354
 LREP (*slepc4py.SLEPc.EPS.ProblemType* attribute), 85
 LYAPII (*slepc4py.SLEPc.EPS.Type* attribute), 88
 LYAPUNOV (*slepc4py.SLEPc.LME.ProblemType* attribute), 172

M

MAT (*slepc4py.SLEPc.BV.MatMultType* attribute), 14
 MAT (*slepc4py.SLEPc.BV.Type* attribute), 18

mat_mode (*slepc4py.SLEPc.ST* attribute), 349
 mat_structure (*slepc4py.SLEPc.ST* attribute), 349
 MatMode (class in *slepc4py.SLEPc.ST*), 328
 matMult() (*slepc4py.SLEPc.BV* method), 34
 matMultColumn() (*slepc4py.SLEPc.BV* method), 34
 matMultHermitianTranspose() (*slepc4py.SLEPc.BV* method), 35
 matMultHermitianTransposeColumn() (*slepc4py.SLEPc.BV* method), 35
 matMultTranspose() (*slepc4py.SLEPc.BV* method), 35
 matMultTransposeColumn() (*slepc4py.SLEPc.BV* method), 36
 MatMultType (class in *slepc4py.SLEPc.BV*), 14
 matProject() (*slepc4py.SLEPc.BV* method), 36
 MatType (class in *slepc4py.SLEPc.DS*), 52
 max_it (*slepc4py.SLEPc.EPS* attribute), 155
 max_it (*slepc4py.SLEPc.LME* attribute), 187
 max_it (*slepc4py.SLEPc.MFN* attribute), 200
 max_it (*slepc4py.SLEPc.NEP* attribute), 255
 max_it (*slepc4py.SLEPc.PEP* attribute), 312
 max_it (*slepc4py.SLEPc.SVD* attribute), 387
 MAXIT (*slepc4py.SLEPc.SVD.Conv* attribute), 350
 MBE (*slepc4py.SLEPc.NEP.RefineScheme* attribute), 206
 MBE (*slepc4py.SLEPc.PEP.RefineScheme* attribute), 264
 method (*slepc4py.SLEPc.DS* attribute), 75
 method (*slepc4py.SLEPc.FN* attribute), 170
 MFN (class in *slepc4py.SLEPc*), 187
 MFNMonitorFunction (in module *slepc4py.typing*), 13
 MGS (*slepc4py.SLEPc.BV.OrthogType* attribute), 17
 module
 slepc4py, 8
 slepc4py.SLEPc, 13
 slepc4py.typing, 9
 MONOMIAL (*slepc4py.SLEPc.PEP.Basis* attribute), 257
 mult() (*slepc4py.SLEPc.BV* method), 37
 multColumn() (*slepc4py.SLEPc.BV* method), 37
 multInPlace() (*slepc4py.SLEPc.BV* method), 38
 MULTIPLE (*slepc4py.SLEPc.NEP.Refine* attribute), 206
 MULTIPLE (*slepc4py.SLEPc.PEP.Refine* attribute), 263
 MULTIPLY (*slepc4py.SLEPc.FN.CombineType* attribute), 157
 multVec() (*slepc4py.SLEPc.BV* method), 38

N

NARNOLDI (*slepc4py.SLEPc.NEP.Type* attribute), 208
 NEP (class in *slepc4py.SLEPc*), 201
 NEP (*slepc4py.SLEPc.DS.Type* attribute), 57
 NEPEigenvalueComparison (in module *slepc4py.typing*), 12
 NEPFunction (in module *slepc4py.typing*), 12
 NEPJacobian (in module *slepc4py.typing*), 13
 NEPMonitorFunction (in module *slepc4py.typing*), 12
 NEPStoppingFunction (in module *slepc4py.typing*), 12

NEVER (*slepc4py.SLEPc.BV.OrthogRefineType* attribute), 16
 NHEP (*slepc4py.SLEPc.DS.Type* attribute), 57
 NHEP (*slepc4py.SLEPc.EPS.ProblemType* attribute), 85
 NHEPTS (*slepc4py.SLEPc.DS.Type* attribute), 57
 NLEIGS (*slepc4py.SLEPc.NEP.Type* attribute), 208
 NONE (*slepc4py.SLEPc.EPS.Balance* attribute), 77
 NONE (*slepc4py.SLEPc.NEP.Refine* attribute), 206
 NONE (*slepc4py.SLEPc.PEP.Extract* attribute), 261
 NONE (*slepc4py.SLEPc.PEP.Refine* attribute), 263
 NONE (*slepc4py.SLEPc.PEP.Scale* attribute), 265
 NONE (*slepc4py.SLEPc.ST.FilterDamping* attribute), 327
 NORM (*slepc4py.SLEPc.EPS.Conv* attribute), 79
 NORM (*slepc4py.SLEPc.NEP.Conv* attribute), 202
 NORM (*slepc4py.SLEPc.PEP.Conv* attribute), 258
 NORM (*slepc4py.SLEPc.PEP.Extract* attribute), 261
 NORM (*slepc4py.SLEPc.SVD.Conv* attribute), 350
 NORM (*slepc4py.SLEPc.SVD.ErrorType* attribute), 352
 norm() (*slepc4py.SLEPc.BV* method), 39
 normColumn() (*slepc4py.SLEPc.BV* method), 39
 NUMPY_INCLUDE, 5

O

ONESIDE (*slepc4py.SLEPc.EPS.Balance* attribute), 77
 OrthogBlockType (class in *slepc4py.SLEPc.BV*), 15
 ORTHOGONAL (*slepc4py.SLEPc.PEP.IDProjection* attribute), 262
 orthogonalize() (*slepc4py.SLEPc.BV* method), 40
 orthogonalizeColumn() (*slepc4py.SLEPc.BV* method), 40
 orthogonalizeVec() (*slepc4py.SLEPc.BV* method), 41
 OrthogRefineType (class in *slepc4py.SLEPc.BV*), 16
 OrthogType (class in *slepc4py.SLEPc.BV*), 16
 orthonormalizeColumn() (*slepc4py.SLEPc.BV* method), 41

P

parallel (*slepc4py.SLEPc.DS* attribute), 75
 parallel (*slepc4py.SLEPc.FN* attribute), 170
 ParallelType (class in *slepc4py.SLEPc.DS*), 54
 ParallelType (class in *slepc4py.SLEPc.FN*), 158
 PARTIAL (*slepc4py.SLEPc.EPS.LanczosReorthogType* attribute), 83
 PEP (class in *slepc4py.SLEPc*), 256
 PEP (*slepc4py.SLEPc.DS.Type* attribute), 57
 PEPEigenvalueComparison (in module *slepc4py.typing*), 12
 PEPMonitorFunction (in module *slepc4py.typing*), 12
 PEPStoppingFunction (in module *slepc4py.typing*), 12
 PERIODIC (*slepc4py.SLEPc.EPS.LanczosReorthogType* attribute), 83
 PETSC_ARCH, 5
 PETSC_DIR, 5
 PGNHEP (*slepc4py.SLEPc.EPS.ProblemType* attribute), 85

PHI (*slepc4py.SLEPc.FN.Type* attribute), 159
 POLYGON (*slepc4py.SLEPc.RG.Type* attribute), 314
 POWER (*slepc4py.SLEPc.EPS.Type* attribute), 88
 PowerShiftType (class in *slepc4py.SLEPc.EPS*), 84
 PRECOND (*slepc4py.SLEPc.ST.Type* attribute), 330
 PRIMME (*slepc4py.SLEPc.EPS.Type* attribute), 89
 PRIMME (*slepc4py.SLEPc.SVD.Type* attribute), 356
 problem_type (*slepc4py.SLEPc.EPS* attribute), 155
 problem_type (*slepc4py.SLEPc.NEP* attribute), 255
 problem_type (*slepc4py.SLEPc.PEP* attribute), 312
 problem_type (*slepc4py.SLEPc.SVD* attribute), 387
 ProblemType (class in *slepc4py.SLEPc.EPS*), 84
 ProblemType (class in *slepc4py.SLEPc.LME*), 172
 ProblemType (class in *slepc4py.SLEPc.NEP*), 205
 ProblemType (class in *slepc4py.SLEPc.PEP*), 262
 ProblemType (class in *slepc4py.SLEPc.SVD*), 353
 PROJECTEDBSE (*slepc4py.SLEPc.EPS.KrylovSchurBSEType* attribute), 82
 purify (*slepc4py.SLEPc.EPS* attribute), 155

Q

Q (*slepc4py.SLEPc.DS.MatType* attribute), 53
 QARNOLDI (*slepc4py.SLEPc.PEP.Type* attribute), 266
 QR (*slepc4py.SLEPc.BV.SVDMethod* attribute), 17
 QR_CAA (*slepc4py.SLEPc.BV.SVDMethod* attribute), 17
 QuadRule (class in *slepc4py.SLEPc.RG*), 313

R

RANDOMIZED (*slepc4py.SLEPc.SVD.Type* attribute), 356
 RATIONAL (*slepc4py.SLEPc.FN.Type* attribute), 159
 RATIONAL (*slepc4py.SLEPc.NEP.ProblemType* attribute), 205
 RAW (*slepc4py.SLEPc.DS.StateType* attribute), 55
 RAYLEIGH (*slepc4py.SLEPc.EPS.PowerShiftType* attribute), 84
 REDUNDANT (*slepc4py.SLEPc.DS.ParallelType* attribute), 54
 REDUNDANT (*slepc4py.SLEPc.FN.ParallelType* attribute), 158
 Refine (class in *slepc4py.SLEPc.NEP*), 205
 Refine (class in *slepc4py.SLEPc.PEP*), 263
 REFINE (*slepc4py.SLEPc.BV.SVDMethod* attribute), 17
 refined (*slepc4py.SLEPc.DS* attribute), 75
 REFINED (*slepc4py.SLEPc.EPS.Extraction* attribute), 82
 REFINED_HARMONIC (*slepc4py.SLEPc.EPS.Extraction* attribute), 82
 RefineScheme (class in *slepc4py.SLEPc.NEP*), 206
 RefineScheme (class in *slepc4py.SLEPc.PEP*), 264
 REL (*slepc4py.SLEPc.EPS.Conv* attribute), 79
 REL (*slepc4py.SLEPc.NEP.Conv* attribute), 202
 REL (*slepc4py.SLEPc.PEP.Conv* attribute), 258
 REL (*slepc4py.SLEPc.SVD.Conv* attribute), 350
 RELATIVE (*slepc4py.SLEPc.EPS.ErrorType* attribute), 81

method), 375
 setDefiniteTolerance() (*slepc4py.SLEPc.BV method*), 44
 setDeflationSpace() (*slepc4py.SLEPc.EPS method*), 131
 setDimensions() (*slepc4py.SLEPc.DS method*), 68
 setDimensions() (*slepc4py.SLEPc.EPS method*), 131
 setDimensions() (*slepc4py.SLEPc.LME method*), 182
 setDimensions() (*slepc4py.SLEPc.MFN method*), 195
 setDimensions() (*slepc4py.SLEPc.NEP method*), 237
 setDimensions() (*slepc4py.SLEPc.PEP method*), 294
 setDimensions() (*slepc4py.SLEPc.SVD method*), 376
 setDS() (*slepc4py.SLEPc.EPS method*), 131
 setDS() (*slepc4py.SLEPc.NEP method*), 237
 setDS() (*slepc4py.SLEPc.PEP method*), 294
 setDS() (*slepc4py.SLEPc.SVD method*), 376
 setEigenvalueComparison() (*slepc4py.SLEPc.EPS method*), 132
 setEigenvalueComparison() (*slepc4py.SLEPc.NEP method*), 238
 setEigenvalueComparison() (*slepc4py.SLEPc.PEP method*), 295
 setEllipseParameters() (*slepc4py.SLEPc.RG method*), 322
 setErrorIfNotConverged() (*slepc4py.SLEPc.LME method*), 182
 setErrorIfNotConverged() (*slepc4py.SLEPc.MFN method*), 195
 setExtract() (*slepc4py.SLEPc.PEP method*), 295
 setExtraction() (*slepc4py.SLEPc.EPS method*), 132
 setExtraRow() (*slepc4py.SLEPc.DS method*), 68
 setFilterDamping() (*slepc4py.SLEPc.ST method*), 341
 setFilterDegree() (*slepc4py.SLEPc.ST method*), 341
 setFilterInterval() (*slepc4py.SLEPc.ST method*), 342
 setFilterRange() (*slepc4py.SLEPc.ST method*), 342
 setFilterType() (*slepc4py.SLEPc.ST method*), 343
 setFN() (*slepc4py.SLEPc.MFN method*), 196
 setFromOptions() (*slepc4py.SLEPc.BV method*), 45
 setFromOptions() (*slepc4py.SLEPc.DS method*), 69
 setFromOptions() (*slepc4py.SLEPc.EPS method*), 133
 setFromOptions() (*slepc4py.SLEPc.FN method*), 167
 setFromOptions() (*slepc4py.SLEPc.LME method*), 182
 setFromOptions() (*slepc4py.SLEPc.MFN method*), 196
 setFromOptions() (*slepc4py.SLEPc.NEP method*), 238
 setFromOptions() (*slepc4py.SLEPc.PEP method*), 296
 setFromOptions() (*slepc4py.SLEPc.RG method*), 323
 setFromOptions() (*slepc4py.SLEPc.ST method*), 343
 setFromOptions() (*slepc4py.SLEPc.SVD method*), 377
 setFunction() (*slepc4py.SLEPc.NEP method*), 238
 setGDBlockSize() (*slepc4py.SLEPc.EPS method*), 134
 setGDBOrth() (*slepc4py.SLEPc.EPS method*), 133
 setGDDoubleExpansion() (*slepc4py.SLEPc.EPS method*), 134
 setGDInitialSize() (*slepc4py.SLEPc.EPS method*), 134
 setGDKrylovStart() (*slepc4py.SLEPc.EPS method*), 135
 setGDRestart() (*slepc4py.SLEPc.EPS method*), 135
 setGSVDDimensions() (*slepc4py.SLEPc.DS method*), 69
 setHSVDDimensions() (*slepc4py.SLEPc.DS method*), 69
 setIdentity() (*slepc4py.SLEPc.DS method*), 70
 setImplicitTranspose() (*slepc4py.SLEPc.SVD method*), 377
 setInitialSpace() (*slepc4py.SLEPc.EPS method*), 135
 setInitialSpace() (*slepc4py.SLEPc.NEP method*), 239
 setInitialSpace() (*slepc4py.SLEPc.PEP method*), 296
 setInitialSpace() (*slepc4py.SLEPc.SVD method*), 377
 setInterpolInterpolation() (*slepc4py.SLEPc.NEP method*), 239
 setInterpolPEP() (*slepc4py.SLEPc.NEP method*), 240
 setInterval() (*slepc4py.SLEPc.EPS method*), 136
 setInterval() (*slepc4py.SLEPc.PEP method*), 297
 setIntervalEndpoints() (*slepc4py.SLEPc.RG method*), 323
 setJacobian() (*slepc4py.SLEPc.NEP method*), 240
 setJDBlockSize() (*slepc4py.SLEPc.EPS method*), 137
 setJDBOrth() (*slepc4py.SLEPc.EPS method*), 136
 setJDConstCorrectionTol() (*slepc4py.SLEPc.EPS method*), 137
 setJDFix() (*slepc4py.SLEPc.EPS method*), 138
 setJDFix() (*slepc4py.SLEPc.PEP method*), 297
 setJDInitialSize() (*slepc4py.SLEPc.EPS method*), 138
 setJDKrylovStart() (*slepc4py.SLEPc.EPS method*), 138
 setJDMinimalityIndex() (*slepc4py.SLEPc.PEP method*), 297
 setJDProjection() (*slepc4py.SLEPc.PEP method*), 298
 setJDRestart() (*slepc4py.SLEPc.EPS method*), 139
 setJDRestart() (*slepc4py.SLEPc.PEP method*), 298
 setJDReusePreconditioner() (*slepc4py.SLEPc.PEP method*), 299
 setKrylovSchurBSEType() (*slepc4py.SLEPc.EPS method*), 139
 setKrylovSchurDetectZeros() (*slepc4py.SLEPc.EPS method*), 140
 setKrylovSchurDimensions() (*slepc4py.SLEPc.EPS*

method), 140
 setKrylovSchurLocking() (*slepc4py.SLEPc.EPS method*), 141
 setKrylovSchurPartitions() (*slepc4py.SLEPc.EPS method*), 141
 setKrylovSchurRestart() (*slepc4py.SLEPc.EPS method*), 142
 setKrylovSchurSubintervals() (*slepc4py.SLEPc.EPS method*), 142
 setKSP() (*slepc4py.SLEPc.ST method*), 343
 setLanczosOneSide() (*slepc4py.SLEPc.SVD method*), 378
 setLanczosReorthogType() (*slepc4py.SLEPc.EPS method*), 144
 setLeadingDimension() (*slepc4py.SLEPc.BV method*), 45
 setLeftInitialSpace() (*slepc4py.SLEPc.EPS method*), 144
 setLinearEPS() (*slepc4py.SLEPc.PEP method*), 299
 setLinearExplicitMatrix() (*slepc4py.SLEPc.PEP method*), 299
 setLinearLinearization() (*slepc4py.SLEPc.PEP method*), 300
 setLOBPCGBlockSize() (*slepc4py.SLEPc.EPS method*), 143
 setLOBPCGLocking() (*slepc4py.SLEPc.EPS method*), 143
 setLOBPCGRestart() (*slepc4py.SLEPc.EPS method*), 143
 setLyapIIRanks() (*slepc4py.SLEPc.EPS method*), 144
 setMatMode() (*slepc4py.SLEPc.ST method*), 344
 setMatMultMethod() (*slepc4py.SLEPc.BV method*), 45
 setMatrices() (*slepc4py.SLEPc.ST method*), 345
 setMatrix() (*slepc4py.SLEPc.BV method*), 46
 setMatStructure() (*slepc4py.SLEPc.ST method*), 344
 setMethod() (*slepc4py.SLEPc.DS method*), 70
 setMethod() (*slepc4py.SLEPc.FN method*), 167
 setMonitor() (*slepc4py.SLEPc.EPS method*), 145
 setMonitor() (*slepc4py.SLEPc.LME method*), 183
 setMonitor() (*slepc4py.SLEPc.MFN method*), 197
 setMonitor() (*slepc4py.SLEPc.NEP method*), 240
 setMonitor() (*slepc4py.SLEPc.PEP method*), 300
 setMonitor() (*slepc4py.SLEPc.SVD method*), 378
 setNArnoldiKSP() (*slepc4py.SLEPc.NEP method*), 241
 setNArnoldiLagPreconditioner() (*slepc4py.SLEPc.NEP method*), 241
 setNLEIGSEPS() (*slepc4py.SLEPc.NEP method*), 241
 setNLEIGSFullBasis() (*slepc4py.SLEPc.NEP method*), 242
 setNLEIGSInterpolation() (*slepc4py.SLEPc.NEP method*), 242
 setNLEIGSLocking() (*slepc4py.SLEPc.NEP method*), 243
 setNLEIGSRestart() (*slepc4py.SLEPc.NEP method*), 243
 setNLEIGSRKShifts() (*slepc4py.SLEPc.NEP method*), 243
 setNumConstraints() (*slepc4py.SLEPc.BV method*), 46
 setOperator() (*slepc4py.SLEPc.MFN method*), 197
 setOperators() (*slepc4py.SLEPc.EPS method*), 145
 setOperators() (*slepc4py.SLEPc.PEP method*), 300
 setOperators() (*slepc4py.SLEPc.SVD method*), 379
 setOptionsPrefix() (*slepc4py.SLEPc.BV method*), 47
 setOptionsPrefix() (*slepc4py.SLEPc.DS method*), 70
 setOptionsPrefix() (*slepc4py.SLEPc.EPS method*), 146
 setOptionsPrefix() (*slepc4py.SLEPc.FN method*), 167
 setOptionsPrefix() (*slepc4py.SLEPc.LME method*), 183
 setOptionsPrefix() (*slepc4py.SLEPc.MFN method*), 197
 setOptionsPrefix() (*slepc4py.SLEPc.NEP method*), 244
 setOptionsPrefix() (*slepc4py.SLEPc.PEP method*), 301
 setOptionsPrefix() (*slepc4py.SLEPc.RG method*), 324
 setOptionsPrefix() (*slepc4py.SLEPc.ST method*), 345
 setOptionsPrefix() (*slepc4py.SLEPc.SVD method*), 379
 setOrthogonalization() (*slepc4py.SLEPc.BV method*), 47
 setParallel() (*slepc4py.SLEPc.DS method*), 71
 setParallel() (*slepc4py.SLEPc.FN method*), 168
 setPEPCoefficients() (*slepc4py.SLEPc.DS method*), 71
 setPEPDegree() (*slepc4py.SLEPc.DS method*), 71
 setPhiIndex() (*slepc4py.SLEPc.FN method*), 168
 setPolygonVertices() (*slepc4py.SLEPc.RG method*), 324
 setPowerShiftType() (*slepc4py.SLEPc.EPS method*), 146
 setPreconditionerMat() (*slepc4py.SLEPc.ST method*), 346
 setProblemType() (*slepc4py.SLEPc.EPS method*), 147
 setProblemType() (*slepc4py.SLEPc.LME method*), 183
 setProblemType() (*slepc4py.SLEPc.NEP method*), 244
 setProblemType() (*slepc4py.SLEPc.PEP method*), 301
 setProblemType() (*slepc4py.SLEPc.SVD method*), 380
 setPurify() (*slepc4py.SLEPc.EPS method*), 147
 setQArnoldiLocking() (*slepc4py.SLEPc.PEP method*), 302
 setQArnoldiRestart() (*slepc4py.SLEPc.PEP method*), 302

setRandom() (*slepc4py.SLEPc.BV method*), 48
 setRandomColumn() (*slepc4py.SLEPc.BV method*), 48
 setRandomCond() (*slepc4py.SLEPc.BV method*), 48
 setRandomContext() (*slepc4py.SLEPc.BV method*), 49
 setRandomNormal() (*slepc4py.SLEPc.BV method*), 49
 setRandomSign() (*slepc4py.SLEPc.BV method*), 49
 setRationalDenominator() (*slepc4py.SLEPc.FN method*), 169
 setRationalNumerator() (*slepc4py.SLEPc.FN method*), 169
 setRefine() (*slepc4py.SLEPc.NEP method*), 247
 setRefine() (*slepc4py.SLEPc.PEP method*), 303
 setRefined() (*slepc4py.SLEPc.DS method*), 72
 setRG() (*slepc4py.SLEPc.EPS method*), 148
 setRG() (*slepc4py.SLEPc.NEP method*), 245
 setRG() (*slepc4py.SLEPc.PEP method*), 303
 setRHS() (*slepc4py.SLEPc.LME method*), 184
 setRIIConstCorrectionTol() (*slepc4py.SLEPc.NEP method*), 245
 setRIIDeflationThreshold() (*slepc4py.SLEPc.NEP method*), 245
 setRIIHermitean() (*slepc4py.SLEPc.NEP method*), 246
 setRIIKSP() (*slepc4py.SLEPc.NEP method*), 246
 setRIILagPreconditioner() (*slepc4py.SLEPc.NEP method*), 247
 setRIIMaximumIterations() (*slepc4py.SLEPc.NEP method*), 247
 setRingParameters() (*slepc4py.SLEPc.RG method*), 324
 setRQCGReset() (*slepc4py.SLEPc.EPS method*), 148
 setScale() (*slepc4py.SLEPc.FN method*), 169
 setScale() (*slepc4py.SLEPc.PEP method*), 306
 setScale() (*slepc4py.SLEPc.RG method*), 325
 setShift() (*slepc4py.SLEPc.ST method*), 346
 setSignature() (*slepc4py.SLEPc.SVD method*), 380
 setSizes() (*slepc4py.SLEPc.BV method*), 50
 setSizesFromVec() (*slepc4py.SLEPc.BV method*), 50
 setSLPDeflationThreshold() (*slepc4py.SLEPc.NEP method*), 248
 setSLPEPS() (*slepc4py.SLEPc.NEP method*), 248
 setSLPEPSLeft() (*slepc4py.SLEPc.NEP method*), 248
 setSLPKSP() (*slepc4py.SLEPc.NEP method*), 249
 setSolution() (*slepc4py.SLEPc.LME method*), 184
 setSplitOperator() (*slepc4py.SLEPc.NEP method*), 249
 setSplitPreconditioner() (*slepc4py.SLEPc.NEP method*), 250
 setSplitPreconditioner() (*slepc4py.SLEPc.ST method*), 347
 setST() (*slepc4py.SLEPc.EPS method*), 148
 setST() (*slepc4py.SLEPc.PEP method*), 303
 setState() (*slepc4py.SLEPc.DS method*), 73
 setSTOARCheckEigenvalueType() (*slepc4py.SLEPc.PEP method*), 304
 setSTOARDetectZeros() (*slepc4py.SLEPc.PEP method*), 304
 setSTOARDimensions() (*slepc4py.SLEPc.PEP method*), 305
 setSTOARLinearization() (*slepc4py.SLEPc.PEP method*), 305
 setSTOARLocking() (*slepc4py.SLEPc.PEP method*), 306
 setStoppingTest() (*slepc4py.SLEPc.EPS method*), 149
 setStoppingTest() (*slepc4py.SLEPc.NEP method*), 250
 setStoppingTest() (*slepc4py.SLEPc.PEP method*), 306
 setStoppingTest() (*slepc4py.SLEPc.SVD method*), 380
 setSVDDimensions() (*slepc4py.SLEPc.DS method*), 72
 setTarget() (*slepc4py.SLEPc.EPS method*), 149
 setTarget() (*slepc4py.SLEPc.NEP method*), 250
 setTarget() (*slepc4py.SLEPc.PEP method*), 308
 setThreshold() (*slepc4py.SLEPc.EPS method*), 149
 setThreshold() (*slepc4py.SLEPc.SVD method*), 383
 setTOARLocking() (*slepc4py.SLEPc.PEP method*), 307
 setTOARRestart() (*slepc4py.SLEPc.PEP method*), 307
 setTolerances() (*slepc4py.SLEPc.EPS method*), 150
 setTolerances() (*slepc4py.SLEPc.LME method*), 185
 setTolerances() (*slepc4py.SLEPc.MFN method*), 198
 setTolerances() (*slepc4py.SLEPc.NEP method*), 251
 setTolerances() (*slepc4py.SLEPc.PEP method*), 308
 setTolerances() (*slepc4py.SLEPc.SVD method*), 384
 setTrackAll() (*slepc4py.SLEPc.EPS method*), 150
 setTrackAll() (*slepc4py.SLEPc.NEP method*), 251
 setTrackAll() (*slepc4py.SLEPc.PEP method*), 309
 setTrackAll() (*slepc4py.SLEPc.SVD method*), 384
 setTransform() (*slepc4py.SLEPc.ST method*), 347
 setTRLanczosExplicitMatrix() (*slepc4py.SLEPc.SVD method*), 381
 setTRLanczosGBidiag() (*slepc4py.SLEPc.SVD method*), 381
 setTRLanczosKSP() (*slepc4py.SLEPc.SVD method*), 381
 setTRLanczosLocking() (*slepc4py.SLEPc.SVD method*), 382
 setTRLanczosOneSide() (*slepc4py.SLEPc.SVD method*), 382
 setTRLanczosRestart() (*slepc4py.SLEPc.SVD method*), 383
 setTrueResidual() (*slepc4py.SLEPc.EPS method*), 151
 setTwoSided() (*slepc4py.SLEPc.EPS method*), 151
 setTwoSided() (*slepc4py.SLEPc.NEP method*), 252
 setType() (*slepc4py.SLEPc.BV method*), 50

setType() (slepc4py.SLEPc.DS method), 73
 setType() (slepc4py.SLEPc.EPS method), 151
 setType() (slepc4py.SLEPc.FN method), 170
 setType() (slepc4py.SLEPc.LME method), 185
 setType() (slepc4py.SLEPc.MFN method), 198
 setType() (slepc4py.SLEPc.NEP method), 252
 setType() (slepc4py.SLEPc.PEP method), 309
 setType() (slepc4py.SLEPc.RG method), 325
 setType() (slepc4py.SLEPc.ST method), 348
 setType() (slepc4py.SLEPc.SVD method), 384
 setUp() (slepc4py.SLEPc.EPS method), 152
 setUp() (slepc4py.SLEPc.LME method), 186
 setUp() (slepc4py.SLEPc.MFN method), 199
 setUp() (slepc4py.SLEPc.NEP method), 252
 setUp() (slepc4py.SLEPc.PEP method), 309
 setUp() (slepc4py.SLEPc.ST method), 348
 setUp() (slepc4py.SLEPc.SVD method), 385
 setVecType() (slepc4py.SLEPc.BV method), 51
 setWhichEigenpairs() (slepc4py.SLEPc.EPS method), 152
 setWhichEigenpairs() (slepc4py.SLEPc.NEP method), 253
 setWhichEigenpairs() (slepc4py.SLEPc.PEP method), 310
 setWhichSingularTriplets() (slepc4py.SLEPc.SVD method), 385
 SHAO (slepc4py.SLEPc.EPS.KrylovSchurBSEType attribute), 83
 SHELL (slepc4py.SLEPc.ST.MatMode attribute), 329
 SHELL (slepc4py.SLEPc.ST.Type attribute), 330
 shift (slepc4py.SLEPc.ST attribute), 349
 SHIFT (slepc4py.SLEPc.ST.Type attribute), 330
 SIMPLE (slepc4py.SLEPc.NEP.Refine attribute), 206
 SIMPLE (slepc4py.SLEPc.PEP.Refine attribute), 263
 SINGLE (slepc4py.SLEPc.SVD.TRLanczosGBidiag attribute), 354
 SINVERT (slepc4py.SLEPc.ST.Type attribute), 330
 size (slepc4py.SLEPc.BV attribute), 52
 sizes (slepc4py.SLEPc.BV attribute), 52
 slepc4py
 module, 8
 slepc4py.SLEPc
 module, 13
 slepc4py.typing
 module, 9
 SLEPC_DIR, 5
 SLP (slepc4py.SLEPc.NEP.Type attribute), 208
 SMALLEST (slepc4py.SLEPc.SVD.Which attribute), 356
 SMALLEST_IMAGINARY (slepc4py.SLEPc.EPS.Which attribute), 90
 SMALLEST_IMAGINARY (slepc4py.SLEPc.NEP.Which attribute), 209
 SMALLEST_IMAGINARY (slepc4py.SLEPc.PEP.Which attribute), 268
 SMALLEST_MAGNITUDE (slepc4py.SLEPc.EPS.Which attribute), 90
 SMALLEST_MAGNITUDE (slepc4py.SLEPc.NEP.Which attribute), 209
 SMALLEST_MAGNITUDE (slepc4py.SLEPc.PEP.Which attribute), 268
 SMALLEST_REAL (slepc4py.SLEPc.EPS.Which attribute), 90
 SMALLEST_REAL (slepc4py.SLEPc.NEP.Which attribute), 209
 SMALLEST_REAL (slepc4py.SLEPc.PEP.Which attribute), 268
 solve() (slepc4py.SLEPc.DS method), 73
 solve() (slepc4py.SLEPc.EPS method), 153
 solve() (slepc4py.SLEPc.LME method), 186
 solve() (slepc4py.SLEPc.MFN method), 199
 solve() (slepc4py.SLEPc.NEP method), 253
 solve() (slepc4py.SLEPc.PEP method), 310
 solve() (slepc4py.SLEPc.SVD method), 385
 solveTranspose() (slepc4py.SLEPc.MFN method), 199
 SQRT (slepc4py.SLEPc.FN.Type attribute), 159
 ST (class in slepc4py.SLEPc), 326
 st (slepc4py.SLEPc.EPS attribute), 156
 st (slepc4py.SLEPc.PEP attribute), 312
 STANDARD (slepc4py.SLEPc.SVD.ProblemType attribute), 353
 state (slepc4py.SLEPc.DS attribute), 76
 StateType (class in slepc4py.SLEPc.DS), 55
 STEIN (slepc4py.SLEPc.LME.ProblemType attribute), 172
 STOAR (slepc4py.SLEPc.PEP.Type attribute), 266
 Stop (class in slepc4py.SLEPc.EPS), 86
 Stop (class in slepc4py.SLEPc.NEP), 207
 Stop (class in slepc4py.SLEPc.PEP), 265
 Stop (class in slepc4py.SLEPc.SVD), 353
 STRUCTURED (slepc4py.SLEPc.PEP.Extract attribute), 261
 SUBSPACE (slepc4py.SLEPc.EPS.Type attribute), 89
 SVD (class in slepc4py.SLEPc), 349
 SVD (slepc4py.SLEPc.DS.Type attribute), 57
 SVDMethod (class in slepc4py.SLEPc.BV), 17
 SVDMonitorFunction (in module slepc4py.typing), 13
 SVDStoppingFunction (in module slepc4py.typing), 13
 SVEC (slepc4py.SLEPc.BV.Type attribute), 18
 SVQB (slepc4py.SLEPc.BV.OrthogBlockType attribute), 15
 SYLVESTER (slepc4py.SLEPc.LME.ProblemType attribute), 173
 SYNCHRONIZED (slepc4py.SLEPc.DS.ParallelType attribute), 54
 SYNCHRONIZED (slepc4py.SLEPc.FN.ParallelType attribute), 158
 Sys (class in slepc4py.SLEPc), 388

T

T (*slepc4py.SLEPc.DS.MatType attribute*), 53
target (*slepc4py.SLEPc.EPS attribute*), 156
target (*slepc4py.SLEPc.NEP attribute*), 255
target (*slepc4py.SLEPc.PEP attribute*), 312
TARGET_IMAGINARY (*slepc4py.SLEPc.EPS.Which attribute*), 90
TARGET_IMAGINARY (*slepc4py.SLEPc.NEP.Which attribute*), 209
TARGET_IMAGINARY (*slepc4py.SLEPc.PEP.Which attribute*), 268
TARGET_MAGNITUDE (*slepc4py.SLEPc.EPS.Which attribute*), 90
TARGET_MAGNITUDE (*slepc4py.SLEPc.NEP.Which attribute*), 209
TARGET_MAGNITUDE (*slepc4py.SLEPc.PEP.Which attribute*), 268
TARGET_REAL (*slepc4py.SLEPc.EPS.Which attribute*), 90
TARGET_REAL (*slepc4py.SLEPc.NEP.Which attribute*), 209
TARGET_REAL (*slepc4py.SLEPc.PEP.Which attribute*), 268
TENSOR (*slepc4py.SLEPc.BV.Type attribute*), 18
THRESHOLD (*slepc4py.SLEPc.EPS.Stop attribute*), 86
THRESHOLD (*slepc4py.SLEPc.SVD.Stop attribute*), 354
TOAR (*slepc4py.SLEPc.PEP.Type attribute*), 266
tol (*slepc4py.SLEPc.EPS attribute*), 156
tol (*slepc4py.SLEPc.LME attribute*), 187
tol (*slepc4py.SLEPc.MFN attribute*), 200
tol (*slepc4py.SLEPc.NEP attribute*), 255
tol (*slepc4py.SLEPc.PEP attribute*), 312
tol (*slepc4py.SLEPc.SVD attribute*), 387
track_all (*slepc4py.SLEPc.EPS attribute*), 156
track_all (*slepc4py.SLEPc.NEP attribute*), 255
track_all (*slepc4py.SLEPc.PEP attribute*), 312
track_all (*slepc4py.SLEPc.SVD attribute*), 387
transform (*slepc4py.SLEPc.ST attribute*), 349
transpose_mode (*slepc4py.SLEPc.SVD attribute*), 387
TRAPEZOIDAL (*slepc4py.SLEPc.EPS.CISSQuadRule attribute*), 78
TRAPEZOIDAL (*slepc4py.SLEPc.RG.QuadRule attribute*), 313
TRLANCZOS (*slepc4py.SLEPc.SVD.Type attribute*), 356
TRLanczosGBidiag (*class in slepc4py.SLEPc.SVD*), 354
true_residual (*slepc4py.SLEPc.EPS attribute*), 156
truncate() (*slepc4py.SLEPc.DS method*), 74
TRUNCATED (*slepc4py.SLEPc.DS.StateType attribute*), 55
TSQR (*slepc4py.SLEPc.BV.OrthogBlockType attribute*), 15
TSQRCHOL (*slepc4py.SLEPc.BV.OrthogBlockType attribute*), 15
two_sided (*slepc4py.SLEPc.EPS attribute*), 156
TWOSIDE (*slepc4py.SLEPc.EPS.Balance attribute*), 77
Type (*class in slepc4py.SLEPc.BV*), 18

Type (*class in slepc4py.SLEPc.DS*), 55
Type (*class in slepc4py.SLEPc.EPS*), 86
Type (*class in slepc4py.SLEPc.FN*), 158
Type (*class in slepc4py.SLEPc.LME*), 173
Type (*class in slepc4py.SLEPc.MFN*), 188
Type (*class in slepc4py.SLEPc.NEP*), 207
Type (*class in slepc4py.SLEPc.PEP*), 266
Type (*class in slepc4py.SLEPc.RG*), 314
Type (*class in slepc4py.SLEPc.ST*), 329
Type (*class in slepc4py.SLEPc.SVD*), 355

U

U (*slepc4py.SLEPc.DS.MatType attribute*), 53
updateExtraRow() (*slepc4py.SLEPc.DS method*), 74
updateKrylovSchurSubcommMats() (*slepc4py.SLEPc.EPS method*), 153
UPPER (*slepc4py.SLEPc.SVD.TRLanczosGBidiag attribute*), 354
USER (*slepc4py.SLEPc.EPS.Balance attribute*), 77
USER (*slepc4py.SLEPc.EPS.Conv attribute*), 79
USER (*slepc4py.SLEPc.EPS.Stop attribute*), 86
USER (*slepc4py.SLEPc.EPS.Which attribute*), 90
USER (*slepc4py.SLEPc.NEP.Conv attribute*), 202
USER (*slepc4py.SLEPc.NEP.Stop attribute*), 207
USER (*slepc4py.SLEPc.NEP.Which attribute*), 209
USER (*slepc4py.SLEPc.PEP.Conv attribute*), 259
USER (*slepc4py.SLEPc.PEP.Stop attribute*), 265
USER (*slepc4py.SLEPc.PEP.Which attribute*), 268
USER (*slepc4py.SLEPc.SVD.Conv attribute*), 350
USER (*slepc4py.SLEPc.SVD.Stop attribute*), 354
Util (*class in slepc4py.SLEPc*), 390

V

V (*slepc4py.SLEPc.DS.MatType attribute*), 54
valuesView() (*slepc4py.SLEPc.EPS method*), 154
valuesView() (*slepc4py.SLEPc.NEP method*), 254
valuesView() (*slepc4py.SLEPc.PEP method*), 311
valuesView() (*slepc4py.SLEPc.SVD method*), 386
VECS (*slepc4py.SLEPc.BV.MatMultiType attribute*), 14
VECS (*slepc4py.SLEPc.BV.Type attribute*), 18
vectors() (*slepc4py.SLEPc.DS method*), 74
vectorsView() (*slepc4py.SLEPc.EPS method*), 154
vectorsView() (*slepc4py.SLEPc.NEP method*), 254
vectorsView() (*slepc4py.SLEPc.PEP method*), 311
vectorsView() (*slepc4py.SLEPc.SVD method*), 386
view() (*slepc4py.SLEPc.BV method*), 51
view() (*slepc4py.SLEPc.DS method*), 75
view() (*slepc4py.SLEPc.EPS method*), 155
view() (*slepc4py.SLEPc.FN method*), 170
view() (*slepc4py.SLEPc.LME method*), 186
view() (*slepc4py.SLEPc.MFN method*), 200
view() (*slepc4py.SLEPc.NEP method*), 254
view() (*slepc4py.SLEPc.PEP method*), 311
view() (*slepc4py.SLEPc.RG method*), 326

`view()` (*slepc4py.SLEPc.ST method*), 348
`view()` (*slepc4py.SLEPc.SVD method*), 387

W

`W` (*slepc4py.SLEPc.DS.MatType attribute*), 54
`Which` (*class in slepc4py.SLEPc.EPS*), 89
`Which` (*class in slepc4py.SLEPc.NEP*), 208
`Which` (*class in slepc4py.SLEPc.PEP*), 267
`Which` (*class in slepc4py.SLEPc.SVD*), 356
`which` (*slepc4py.SLEPc.EPS attribute*), 156
`which` (*slepc4py.SLEPc.NEP attribute*), 255
`which` (*slepc4py.SLEPc.PEP attribute*), 313
`which` (*slepc4py.SLEPc.SVD attribute*), 387
`WILKINSON` (*slepc4py.SLEPc.EPS.PowerShiftType attribute*), 84

X

`X` (*slepc4py.SLEPc.DS.MatType attribute*), 54

Y

`Y` (*slepc4py.SLEPc.DS.MatType attribute*), 54

Z

`Z` (*slepc4py.SLEPc.DS.MatType attribute*), 54