1 What is KryPy and where is the code? 1

2 Contents 3
   2.1 krypy Package ........................................... 3
       2.1.1 krypy.linsys - Linear Algebraic Systems Solver ........ 3
       2.1.2 krypy.deflation - Linear Systems Solvers with Deflation .. 7
       2.1.3 krypy.recycling - Recycling Linear Systems Solvers ........ 10
       2.1.4 krypy.utils - Krylov Subspace Utilities ................. 14
       2.1.5 Subpackages ............................................. 24
   2.2 Bibliography .................................................. 25

3 Getting started 27
   3.1 Installation .................................................. 27
   3.2 Solve a linear system ......................................... 27
   3.3 Deflation ...................................................... 27
   3.4 Recycling ...................................................... 28

4 Indices and tables 29

Bibliography 31

Python Module Index 33
What is KryPy and where is the code?

KryPy is a Krylov subspace methods package for Python. If you’re looking for the source code or bug reports, take a look at KryPy’s github page. These pages provide the documentation of KryPy’s API. The project was initiated by André Gaul while researching Krylov subspace methods. The theoretical background as well as applications of this software package can be found in the PhD thesis [Gau14].

KryPy allows Python users to easily use Krylov subspace methods, e.g., for solving linear systems or eigenvalue problems. With its built-in deflation and recycling capabilities it is suitable for advanced applications of Krylov subspace methods (see krypy.deflation - Linear Systems Solvers with Deflation and krypy.recycling - Recycling Linear Systems Solvers). It is also ideal for experimenting with Krylov subspaces since you have access to all data that is generated (e.g., Arnoldi/Lanczos relations), you can use different orthogonalization algorithms (Lanczos short recurrences, modified Gram-Schmidt, double modified Gram-Schmidt, Householder), compare subspaces via angles, and much more. And if you need more: KryPy is free software, it’s easy to extend, and pull requests are more than welcome!
2.1 krypy Package

2.1.1 krypy.linsys - Linear Algebraic Systems Solver

The linsys module provides functions for the solution of linear algebraic systems.

```python
class krypy.linsys.LinearSystem(A, b, M=None, Minv=None, Ml=None, Mr=None,
ip_B=None, normal=None, self_adjoint=False, positive_definite=False, exact_solution=None)
```

Bases: `object`

Representation of a (preconditioned) linear system.

Represents a linear system

\[ Ax = b \]

or a preconditioned linear system

\[ MM_l A M_r y = MM_l b \quad \text{with} \quad x = M_r y. \]

**Parameters**

- **A** – a linear operator on \( \mathbb{C}^N \) (has to be compatible with `get_linearoperator()`).
- **b** – the right hand side in \( \mathbb{C}^N \), i.e., `b.shape == (N, 1)`.
- **M** – (optional) a self-adjoint and positive definite preconditioner, linear operator on \( \mathbb{C}^N \) with respect to the inner product defined by `ip_B`. This preconditioner changes the inner product to \( \langle x, y \rangle_M = \langle Mx, y \rangle \) where \( \langle \cdot, \cdot \rangle \) is the inner product defined by the parameter `ip_B`. Defaults to the identity.
- **Minv** – (optional) the inverse of the preconditioner provided by `M`. This operator is needed, e.g., for orthonormalizing vectors for the computation of Ritz vectors in deflated methods.
- **Ml** – (optional) left preconditioner, linear operator on \( \mathbb{C}^N \). Defaults to the identity.
- **Mr** – (optional) right preconditioner, linear operator on \( \mathbb{C}^N \). Defaults to the identity.
- **ip_B** – (optional) defines the inner product, see `inner()`.
• **normal** – (bool, optional) Is $M_lAM_r$ normal in the inner product defined by $ip_B$? Defaults to False.

• **self_adjoint** – (bool, optional) Is $M_lAM_r$ self-adjoint in the inner product defined by $ip_B$? Also sets normal=True. Defaults to False.

• **positive_definite** – (bool, optional) Is $M_lAM_r$ positive (semi-)definite with respect to the inner product defined by $ip_B$? Defaults to False.

• **exact_solution** – (optional) If an exact solution $x$ is known, it can be provided as a numpy.array with exact_solution.shape == (N,1). Then error norms can be computed (for debugging or research purposes). Defaults to None.

\[ MM_l b \|_{M^{-1}} \]

\[ N = \text{None} \]

Dimension $N$ of the space $\mathbb{C}^N$ where the linear system is defined.

\[ \text{get_ip_Minv_B}() \]

Returns the inner product that is implicitly used with the positive definite preconditioner $M$.

\[ \text{get_residual}(z, \text{compute_norm=False}) \]

Compute residual.

For a given $z \in \mathbb{C}^N$, the residual

\[ r = MM_l (b - Az) \]

is computed. If compute_norm == True, then also the absolute residual norm

\[ \| MM_l (b - Az) \|_{M^{-1}} \]

is computed.

Parameters

• **z** – approximate solution with $z$.shape == (N, 1).

• **compute_norm** – (bool, optional) pass True if also the norm of the residual should be computed.

class krypy.linsys.CG(linear_system, **kwargs)

Bases: krypy.linsys._KrylovSolver

Preconditioned CG method.

The preconditioned conjugate gradient method can be used to solve a system of linear algebraic equations where the linear operator is self-adjoint and positive definite. Let the following linear algebraic system be given:

\[ MM_l AM_r y = MM_l b, \]

where $x = M_l y$ and $M_lAM_r$ is self-adjoint and positive definite with respect to the inner product $\langle \cdot, \cdot \rangle$ defined by $ip_B$. The preconditioned CG method then computes (in exact arithmetics!) iterates $x_k \in x_0 + M_r K_k$ with $K_k := K_k(MM_l AM_r, r_0)$ such that

\[ \| x - x_k \|_A = \min_{z \in x_0 + M_r K_k} \| x - z \|_A. \]

The Lanczos algorithm is used with the operator $MM_l AM_r$ and the inner product defined by $\langle x, y \rangle_{M^{-1}} = \langle M^{-1}x, y \rangle$. The initial vector for Lanczos is $r_0 = MM_l (b - Ax_0)$ - note that $M_r$ is not used for the initial vector.

Memory consumption is:
• if `store_arnoldi==False`: 3 vectors or 6 vectors if $M$ is used.

• if `store_arnoldi==True`: about maxiter+1 vectors for the Lanczos basis. If $M$ is used the memory consumption is $2^*(\maxiter+1)$.

**Caution:** CG’s convergence may be delayed significantly due to round-off errors, cf. chapter 5.9 in [LieS13].

All parameters of `_KrylovSolver` are valid in this solver. Note the restrictions on $M, M_l, A, Mr$ and ip_B above.

**static operations** *(nsteps)*

Returns the number of operations needed for nsteps of CG

```python
class krypy.linsys.Minres(linear_system, ortho='lanczos', **kwargs)
```

**Preconditioned MINRES method.**

The **preconditioned minimal residual method** can be used to solve a system of linear algebraic equations where the linear operator is self-adjoint. Let the following linear algebraic system be given:

$$MM_l AM_r y = MM_l b,$$

where $x = M_r y$ and $M_l A M_r$ is self-adjoint with respect to the inner product $\langle \cdot, \cdot \rangle$ defined by `inner_product`. The preconditioned MINRES method then computes (in exact arithmetics!) iterates $x_k \in x_0 + M_r K_k$ with $K_k := K_k(M M_l A M_r, r_0)$ such that

$$\| M M_l (b - A x_k) \|_{M^{-1}} = \min_{z \in x_0 + M_r K_k} \| M M_l (b - A z) \|_{M^{-1}}.$$

The Lanczos algorithm is used with the operator $MM_l AM_r$ and the inner product defined by $\langle x, y \rangle_{M^{-1}} = \langle M^{-1} x, y \rangle$. The initial vector for Lanczos is $r_0 = MM_l (b - A x_0)$ - note that $M_r$ is not used for the initial vector.

Memory consumption is:

• if `store_arnoldi==False`: 3 vectors or 6 vectors if $M$ is used.

• if `store_arnoldi==True`: about maxiter+1 vectors for the Lanczos basis. If $M$ is used the memory consumption is $2^*(\maxiter+1)$.

**Caution:** MINRES’ convergence may be delayed significantly or even stagnate due to round-off errors, cf. chapter 5.9 in [LieS13].

In addition to the attributes described in `_KrylovSolver`, the following attributes are available in an instance of this solver:

• `lanczos`: the Lanczos relation (an instance of Arnoldi).

All parameters of `_KrylovSolver` are valid in this solver. Note the restrictions on $M, M_l, A, Mr$ and ip_B above.

**static operations** *(nsteps)*

Returns the number of operations needed for nsteps of MINRES

```python
class krypy.linsys.Gmres(linear_system, ortho='mgs', **kwargs)
```

**Preconditioned GMRES method.**

The **preconditioned generalized minimal residual method** can be used to solve a system of linear algebraic equations. Let the following linear algebraic system be given:

$$MM_l AM_r y = MM_l b,$$

where $x = M_r y$. The preconditioned GMRES method then computes (in exact arithmetics!) iterates $x_k \in x_0 + M_r K_k$ with $K_k := K_k(M M_l A M_r, r_0)$ such that

$$\| M M_l (b - A x_k) \|_{M^{-1}} = \min_{z \in x_0 + M_r K_k} \| M M_l (b - A z) \|_{M^{-1}}.$$

2.1. krypy Package 5
The Arnoldi algorithm is used with the operator $M M_l A M_r$ and the inner product defined by $\langle x, y \rangle_{M^{-1}} = \langle M^{-1} x, y \rangle$. The initial vector for Arnoldi is $r_0 = M M_l (b - A x_0)$ - note that $M_r$ is not used for the initial vector.

Memory consumption is about maxiter+1 vectors for the Arnoldi basis. If $M$ is used the memory consumption is 2*(maxiter+1).

If the operator $M_l A M_r$ is self-adjoint then consider using the MINRES method Minres.

All parameters of _KrylovSolver are valid in this solver.

**static operations** (nsteps)

Returns the number of operations needed for nsteps of GMRES

```python
class krypy.linsys._KrylovSolver(linear_system, x0=None, tol=1e-05, maxiter=None, explicit_residual=False, store_arnoldi=False, dtype=None)
```

Prototype of a Krylov subspace method for linear systems.

Init standard attributes and perform checks.

All Krylov subspace solvers in this module are applied to a LinearSystem. The specific methods may impose further restrictions on the operators

**Parameters**

- **linear_system** – a LinearSystem.
- **x0** – (optional) the initial guess to use. Defaults to zero vector. Unless you have a good reason to use a nonzero initial guess you should use the zero vector, cf. chapter 5.8.3 in *Liesen, Strakos. Krylov subspace methods. 2013*. See also hegedus().
- **tol** – (optional) the tolerance for the stopping criterion with respect to the relative residual norm:
  \[
  \frac{\| M M_l (b - A (x_0 + M_r y_k)) \|_{M^{-1}}}{\| M M_l b \|_{M^{-1}}} \leq \texttt{tol}
  \]
- **maxiter** – (optional) maximum number of iterations. Defaults to N.
- **explicit_residual** – (optional) if set to False (default), the updated residual norm from the used method is used in each iteration. If set to True, the residual is computed explicitly in each iteration and thus requires an additional application of $M$, $M_l$, $A$ and $M_r$ in each iteration.
- **store_arnoldi** – (optional) if set to True then the computed Arnoldi basis and the Hessenberg matrix are set as attributes $V$ and $H$ on the returned object. If $M$ is not None, then also $P$ is set where $V=M*P$. Defaults to False. If the method is based on the Lanczos method (e.g., Cg or Minres), then $H$ is real, symmetric and tridiagonal.
- **dtype** – (optional) an optional dtype that is used to determine the dtype for the Arnoldi/Lanczos basis and matrix.

Upon convergence, the instance contains the following attributes:

- **xk**: the approximate solution $x_k$.
- **resnorms**: relative residual norms of all iterations, see parameter tol.
- **errnorms**: the error norms of all iterations if exact_solution was provided.
- **V, H and P if store_arnoldi==True, see store_arnoldi**

If the solver does not converge, a ConvergenceError is thrown which can be used to examine the misconvergence.

```python
errnorms = None
```

Error norms.
iter = None
Iteration number.

**static operations** (*nsteps*)
Returns the number of operations needed for *nsteps* of the solver.

**Parameters**

- **nsteps** – number of steps.

**Returns**
a dictionary with the same keys as the timings parameter. Each value is the number of operations of the corresponding type for *nsteps* iterations of the method.

resnorms = None
Relative residual norms as described for parameter *tol*.

xk = None
Approximate solution.

## 2.1.2 krypy.deflation - Linear Systems Solvers with Deflation

The *deflation* module provides functions for deflated Krylov subspace methods. With deflation, a linear system is multiplied with a suitable projection with the goal of accelerating the overall solution process for the linear system. This module provides tools that are needed in deflated Krylov subspace methods such as involved projections and computations of Ritz or harmonic Ritz pairs from a deflated Krylov subspace.

class krypy.deflation.DeflatedCg(*args, **kwargs)
Bases: krypy.deflation._DeflationMixin, krypy.linsys.Cg
Deflated preconditioned CG method.
See _DeflationMixin and Cg for the documentation of the available parameters.

_class_ krypy.deflation._apply_projection(*Av*)
Computes \( \langle C, M_l A M_r V_n \rangle \) efficiently with a three-term recurrence.

class krypy.deflation.DeflatedMinres(*linear_system, U=None, projection_kwargs=None, *args, **kwargs*)
Bases: krypy.deflation._DeflationMixin, krypy.linsys.Minres
Deflated preconditioned MINRES method.
See _DeflationMixin and Minres for the documentation of the available parameters.

class krypy.deflation.DeflatedGmres(*linear_system, U=None, projection_kwargs=None, *args, **kwargs*)
Bases: krypy.deflation._DeflationMixin, krypy.linsys.Gmres
Deflated preconditioned GMRES method.
See _DeflationMixin and Gmres for the documentation of the available parameters.

class krypy.deflation._DeflationMixin(*linear_system, U=None, projection_kwargs=None, *args, **kwargs*)
Bases: object
Mixin class for deflation in Krylov subspace methods.
Can be used to add deflation functionality to any solver from linsys.

**Parameters**

- **linear_system** – the LinearSystem that should be solved.
- **U** – a basis of the deflation space with \( U\text{.shape} == (N, k) \).

All other parameters are passed through to the underlying solver from linsys.

\[
B = (V_{n+1}, M_l A M_r U).
\]
This property is obtained from $C$ if the operator is self-adjoint. Otherwise, the inner products have to be formed explicitly.

\[ C = \text{None} \]

\[ C = \langle U, M_t A M_r V_n \rangle. \]

This attribute is updated while the Arnoldi/Lanczos method proceeds. See also \_apply_projection().

\[ E = \text{None} \]

\[ E = \langle U, M_t A M_r U \rangle. \]

\_apply_projection(Av)

Apply the projection and store inner product.

Parameters
\[ v \] – the vector resulting from an application of $M_t A M_r$ to the current Arnoldi vector. (CG needs special treatment, here).

\_get_initial_residual(x0)

Return the projected initial residual.

Returns $MPM_t(b - A x_0)$.

\_get_xk(yk)

\_solve()

\estimate_time(nsteps, ndefl, deflweight=1.0)

Estimate time needed to run nsteps iterations with deflation

Uses timings from \linear_system if it is an instance of \TimedLinearSystem. Otherwise, an OtherError is raised.

Parameters
- \nsteps – number of iterations.
- \ndefl – number of deflation vectors.
- \deflweight – (optional) the time for the setup and application of the projection for deflation is multiplied by this factor. This can be used as a counter measure for the evaluation of Ritz vectors. Defaults to 1.

\projection = \text{None}

Projection that is used for deflation.

class krypy.deflation.ObliqueProjection(linear_system, U, qr_reorthos=0, **kwargs)

Bases: krypy.deflation._Projection

Oblique projection for left deflation.

\AU = \text{None}

Result of application of operator to deflation space, i.e., $M_t A M_r U$.

\MAU

Result of preconditioned operator to deflation space, i.e., $M M_t A M_r U$.

\U = \text{None}

An orthonormalized basis of the deflation space $U$ with respect to provided inner product.

correct(z)

Correct the given approximate solution $z$ with respect to the linear system \linear_system and the deflation space defined by $U$.

class krypy.deflation._Projection(linear_system, U, **kwargs)

Bases: krypy.utils.Projection

Abstract base class of a projection for deflation.

Parameters
• A – the LinearSystem.
• U – basis of the deflation space with U.shape == (N, d).

All parameters of Projection are valid except X and Y.

**class** krypy.deflation.Ritz(deflated_solver, mode='ritz')

Bases: object

Compute Ritz pairs from a deflated Krylov subspace method.

**Parameters**

• **deflated_solver** – an instance of a deflated solver.
• **mode** – (optional)
  - ritz (default): compute Ritz pairs.
  - harmonic: compute harmonic Ritz pairs.

**coeffs = None**

Coefficients for Ritz vectors in the basis \([V_n, U]\).

**get_explicit_residual(indices=None)**

Explicitly computes the Ritz residual.

**get_explicit_resnorms(indices=None)**

Explicitly computes the Ritz residual norms.

**get_vectors(indices=None)**

Compute Ritz vectors.

**resnorms = None**

Residual norms of Ritz pairs.

**values = None**

Ritz values.

**class** krypy.deflation.Arnoldifyer(deflated_solver)

Bases: object

Obtain Arnoldi relations for approximate deflated Krylov subspaces.

**Parameters**

• **deflated_solver** – an instance of a deflated solver.

**get(Wt, full=False)**

Get Arnoldi relation for a deflation subspace choice.

**Parameters**

• **Wt** – the coefficients \(\tilde{W}\) of the deflation vectors in the basis \([V_n, U]\) with \(\tilde{W}.shape == (n+d, k)\), i.e., the deflation vectors are \(W = [V_n, U]\tilde{W}\). Must fulfill \(\tilde{W}^*\tilde{W} = I_k\).
• **full** – (optional) should the full Arnoldi basis and the full perturbation be returned? Defaults to False.

**Returns**

• **Hh**: the Hessenberg matrix with \(Hh.shape == (n+d-k, n+d-k)\).
• **Rh**: the perturbation core matrix with \(Rh.shape == (l, n+d-k)\).
• **q_norm**: norm \(\|q\|_2\).
• **vdiff_norm**: the norm of the difference of the initial vectors \(\tilde{v} - \hat{v}\).
• **PWAW_norm**: norm of the projection \(P_{W^+,AW^+}\).
• **Vh**: (if full == True) the Arnoldi basis with \(Vh.shape == (N, n+d-k)\).
• **F**: (if full == True) the perturbation matrix \(F = -Z\tilde{R}_n^* - \hat{V}_n\hat{R}^*Z^*\).
krypy.deflation.bound_pseudo(arnoldifyer, Wt, g_norm=0.0, G_norm=0.0, GW_norm=0.0, WGW_norm=0.0, tol=1e-06, pseudo_type='auto', pseudo_kwargs=None, delta_n=20, terminate_factor=1.0)

Bound residual norms of next deflated system.

Parameters

• arnoldifyer – an instance of Arnoldifyer.

• Wt – coefficients \( \tilde{W} \in \mathbb{C}^{n+d,k} \) of the considered deflation vectors \( W \) for the basis \([V, U]\) where \( V=\text{last_solver.V} \) and \( U=\text{last_P.U} \) i.e., \( W = [V, U] \tilde{W} \) and \( \mathcal{W} = \text{colspan}(W) \). Must fulfill \( \tilde{W}^* \tilde{W} = I_k \).

• g_norm – norm \( \|g\| \) of difference \( g = c - b \) of right hand sides. Has to fulfill \( \|g\| < \|b\| \).

• G_norm – norm \( \|G\| \) of difference \( G = B - A \) of operators.

• GW_norm – Norm \( \|G|\mathcal{W}\| \) of difference \( G = B - A \) of operators restricted to \( \mathcal{W} \).

• WGW_norm – Norm \( \|\langle W, GW \rangle\|_2 \).

• pseudo_type – One of
  - 'auto': determines if \( \hat{H} \) is non-normal, normal or Hermitian and uses the corresponding mode (see other options below).
  - 'nonnormal': the pseudospectrum of the Hessenberg matrix \( \hat{H} \) is used (involves one computation of a pseudospectrum)
  - 'normal': the pseudospectrum of \( \hat{H} \) is computed efficiently by the union of circles around the eigenvalues.
  - 'hermitian': the pseudospectrum of \( \hat{H} \) is computed efficiently by the union of intervals around the eigenvalues.
  - 'contain': the pseudospectrum of the extended Hessenberg matrix \( \begin{bmatrix} \hat{H} \\ S_1 \end{bmatrix} \) is used (pseudospectrum has to be re computed for each iteration).
  - 'omit': do not compute the pseudospectrum at all and just use the residual bounds from the approximate Krylov subspace.

• pseudo_kwargs – (optional) arguments that are passed to the method that computes the pseudospectrum.

• terminate_factor – (optional) terminate the computation if the ratio of two subsequent residual norms is larger than the provided factor. Defaults to 1.

2.1.3 krypy.recycling - Recycling Linear Systems Solvers

The recycling module provides functions for the solution of sequences of linear algebraic systems. Once a linear system has been solved, the generated data is examined and a deflation space is determined automatically for the solution of the next linear system. Several selection strategies are available.

krypy.recycling.factories - deflation vector factories

class krypy.recycling.factories.RitzFactory(subset_evaluator, subsets_generator=None, mode='ritz', print_results=None)

Bases: krypy.recycling.factories._DeflationVectorFactory

Factory of Ritz vectors for automatic recycling.

Parameters
• **subset_evaluator** – an instance of `_RitzSubsetEvaluator` that evaluates a proposed subset of Ritz vectors for deflation.

• **subsets_generator** – (optional) an instance of `_RitzSubsetsGenerator` that generates lists of subsets of Ritz vectors for deflation.

• **print_results** – (optional) may be one of the following:
  - `None`: nothing is printed.
  - `'number'`: the number of selected deflation vectors is printed.
  - `'values'`: the Ritz values corresponding to the selected Ritz vectors are printed.
  - `'timings'`: the timings of all evaluated subsets of Ritz vectors are printed.

```python
get (deflated_solver)
```

```python
class krypy.recycling.factories.RitzFactorySimple (mode='ritz', n_vectors=0, which='sm')
Bases: krypy.recycling.factories._DeflationVectorFactory
```

Selects a fixed number of Ritz or harmonic Ritz vectors with respect to a prescribed criterion.

**Parameters**

• **mode** – See mode parameter of `Ritz`.

• **n_vectors** – number of vectors that are chosen. Actual number of deflation vectors may be lower if the number of Ritz pairs is less than `n_vectors`.

• **which** – the `n_vectors` Ritz vectors are chosen such that the corresponding Ritz values are the ones with
  - `lm`: largest magnitude.
  - `sm`: smallest magnitude.
  - `lr`: largest real part.
  - `sr`: smallest real part.
  - `li`: largest imaginary part.
  - `si`: smallest imaginary part.
  - `smallest_res`: smallest Ritz residual norms.

```python
get (solver)
```

```python
class krypy.recycling.factories.UnionFactory (factories)
Bases: krypy.recycling.factories._DeflationVectorFactory
```

Combine a list of factories.

**Parameters**

**factories** – a list of factories derived from `_DeflationVectorFactory`.

```python
get (solver)
```

```python
class krypy.recycling.factories._DeflationVectorFactory
```

Abstract base class for selectors.

**get (solver)**

Get deflation vectors.

**Returns**

numpy.array of shape \((N, k)\)
**krypy.recycling.generators** - generators for deflation vector candidates

**class** krypy.recycling.generators.RitzExtremal\( (max\_vectors=\infty) \)

Bases: krypy.recycling.generators._RitzSubsetsGenerator

Successively returns the extremal Ritz values.

For self-adjoint problems, the indices of the minimal negative, maximal negative, minimal positive and maximal positive Ritz values are returned.

For non-self-adjoint problems, only the indices of the Ritz values of smallest and largest magnitude are returned.

**generate**(ritz, remaining_subset)

**class** krypy.recycling.generators.RitzSmall\( (max\_vectors=\infty) \)

Bases: krypy.recycling.generators._RitzSubsetsGenerator

Successively returns the Ritz value of smallest magnitude.

**generate**(ritz, remaining_subset)

**class** krypy.recycling.generators._RitzSubsetsGenerator

Abstract base class for the generation of subset generation.

**generate**(ritz, remaining_subset)

Returns a list of subsets with indices of Ritz vectors that are considered for deflation.

**krypy.recycling.evaluators** - evaluators for deflation vector candidates

**class** krypy.recycling.evaluators.RitzApproxKrylov\( (mode='extrapolate', \text{tol}=\text{None}, \text{pseudospectra}=\text{False}, \text{bound\_pseudo\_kwargs}=\text{None}, \text{deflweight}=1.0) \)

Bases: krypy.recycling.evaluators._RitzSubsetEvaluator

Evaluates a choice of Ritz vectors with a tailored approximate Krylov subspace method.

**Parameters**

- **mode** – (optional) determines how the number of iterations is estimated. Must be one of the following:
  - extrapolate (default): use the iteration count where the extrapolation of the smallest residual reduction over all steps drops below the tolerance.
  - direct: use the iteration count where the predicted residual bound drops below the tolerance. May result in severe underestimation if pseudospectra==False.
- **pseudospectra** – (optional) should pseudospectra be computed for the given problem? With pseudospectra=True, a prediction may not be possible due to unfulfilled assumptions for the computation of the pseudospectral bound.
- **bound\_pseudo\_kwargs** – (optional) a dictionary with arguments that are passed to bound_pseudo().
- **deflweight** – (optional) see estimate_time(). Defaults to 1.

**evaluate**(ritz, subset)

**class** krypy.recycling.evaluators.RitzApriori\( (Bound, \text{tol}=\text{None}, \text{strategy}='simple', \text{deflweight}=1.0) \)

Bases: krypy.recycling.evaluators._RitzSubsetEvaluator

Evaluates a choice of Ritz vectors with an a-priori bound for self-adjoint problems.

**Parameters**

- **Bound** – the a-priori bound which is used for estimating the convergence behavior.
tol – (optional) the tolerance for the stopping criterion, see \_KrylovSolver. If
None is provided (default), then the tolerance is retrieved from \_itz\_deflated\_solver.tol
in the call to evaluate().

strategy – (optional) the following strategies are available

– simple: (default) uses the Ritz values that are complementary to the deflated ones for
the evaluation of the bound.
– intervals: uses intervals around the Ritz values that are considered with simple. The
intervals incorporate possible changes in the operators.

evaluate (ritz, subset)

class krypy.recycling.RecyclingCg (*args, **kwargs)
    Bases: krypy.recycling.linsys._RecyclingSolver

Recycling preconditioned CG method.

See \_RecyclingSolver for the documentation of the available parameters.

class krypy.recycling.RecyclingMinres (*args, **kwargs)
    Bases: krypy.recycling.linsys._RecyclingSolver

Recycling preconditioned MINRES method.

See \_RecyclingSolver for the documentation of the available parameters.

class krypy.recycling.RecyclingGmres (*args, **kwargs)
    Bases: krypy.recycling.linsys._RecyclingSolver

Recycling preconditioned GMRES method.

See \_RecyclingSolver for the documentation of the available parameters.

class krypy.recycling.linsys._RecyclingSolver (DeflatedSolver,
    vector_factory=None)

Base class for recycling solvers.

Initialize recycling solver base.

Parameters

- DeflatedSolver – a deflated solver from deflation.

- vector_factory – (optional) An instance of a subclass of krypy.recycling.
    factories._DeflationVectorFactory that constructs deflation vectors for
    recycling. Defaults to None which means that no recycling is used.

Also the following strings are allowed as shortcuts:

- 'RitzApproxKrylov': uses the approximate Krylov subspace bound evaluator
  krypy.recycling.evaluators.RitzApproxKrylov.

- 'RitzAprioriCg': uses the CG k-bound (krypy.utils.BoundCG) as an a
  priori bound with krypy.recycling.evaluators.RitzApriori.

- 'RitzAprioriMinres': uses the MINRES bound (krypy.utils.
  BoundMinres) as an a priori bound with krypy.recycling.evaluators.
  RitzApriori.

After a run of the provided DeflatedSolver via solve(), the resulting instance of the
DeflatedSolver is available in the attribute last_solver.

last_solver = None
    DeflatedSolver instance from last run of solve().

    Instance of DeflatedSolver that resulted from the last call to solve(). Initialized with None
    before the first run.
solve (linear_system, vector_factory=None, *args, **kwargs)
Solve the given linear system with recycling.

The provided vector_factory determines which vectors are used for deflation.

Parameters

- **linear_system** – the LinearSystem that is about to be solved.
- **vector_factory** – (optional) see description in constructor.

All remaining arguments are passed to the DeflatedSolver.

Returns instance of DeflatedSolver which was used to obtain the approximate solution. The approximate solution is available under the attribute xk.

timings = None
Timings from last run of solve().
Timings of the vector factory runs and the actual solution processes.

2.1.4 krypy.utils - Krylov Subspace Utilities

The utils module provides helper functions for common tasks in the process of solving linear algebraic systems. Collection of standard functions.

This method provides functions like inner products, norms, ...

**exception** krypy.utils.ArgumentError

Raise when an argument is invalid.
Analogue to ValueError which is not used here in order to be able to distinguish between built-in errors and krypy errors.

**exception** krypy.utils.AssumptionError

Raise when an assumption is not satisfied.
Differs from ArgumentError in that all passed arguments are valid but computations reveal that assumptions are not satisfied and the result cannot be computed.

**exception** krypy.utils.ConvergenceError (msg, solver)

Raise when a method did not converge.
The ConvergenceError holds a message describing the error and the attribute solver through which the last approximation and other relevant information can be retrieved.

**exception** krypy.utils.LinearOperatorError

Raise when a LinearOperator cannot be applied.

**exception** krypy.utils.InnerProductError

Raise when the inner product is indefinite.

**exception** krypy.utils.RuntimeError

Raise for errors that do not fit in any other exception.
class krypy.utils.Arnnoldi(A, v, maxiter=None, ortho='mgs', M=None, Mv=None, Mv_norm=None, ip_B=None)
    Bases: object
Arnoldi algorithm.
Computes V and H such that \(AV_n = V_{n+1}H_n\). If the Krylov subspace becomes A-invariant then V and H are truncated such that \(AV_n = V_{n}H_n\).

Parameters

- **A** – a linear operator that can be used with scipy’s aslinearoperator with shape==(N, N).
- **v** – the initial vector with shape==(N,1).
- **maxiter** – (optional) maximal number of iterations. Default: N.
- **ortho** – (optional) orthogonalization algorithm: may be one of
  - 'mgs': modified Gram-Schmidt (default).
  - 'dmgs': double Modified Gram-Schmidt.
  - 'lanczos': Lanczos short recurrence.
  - 'house': Householder.
- **M** – (optional) a self-adjoint and positive definite preconditioner. If M is provided, then also a second basis \(P_n\) is constructed such that \(V_n = MP_n\). This is of importance in preconditioned methods. M has to be None if ortho=='house' (see B).
- **ip_B** – (optional) defines the inner product to use. See inner().
  ip_B has to be None if ortho=='house'. It’s unclear to me (andrenaranchy), how a variant of the Householder QR algorithm can be used with a non-Euclidean inner product. Compare http://math.stackexchange.com/questions/433644/is-householder-orthogonalization-qr-practicable-for-non-euclidean-inner-products

advance()
Carry out one iteration of Arnoldi.

get()

get_last()

class krypy.utils.BoundCG(evals, exclude_zeros=False)
    Bases: object
CG residual norm bound.
Computes the \(\kappa\)-bound for the CG error \(A\)-norm when the eigenvalues of the operator are given, see [LieS13].

Parameters

- **evals** – an array of eigenvalues \(\lambda_1, \ldots, \lambda_N \in \mathbb{R}\). The eigenvalues will be sorted internally such that \(0 = \lambda_1 = \ldots = \lambda_{t-1} < \lambda_t \leq \ldots \lambda_N\) for \(t \in \mathbb{N}\).
- **steps** – (optional) the number of steps \(k\) to compute the bound for. If steps is None (default), then \(k = N\) is used.

Returns

array \([\eta_0, \ldots, \eta_k]\) with

\[
\eta_n = 2 \left( \frac{\sqrt{\kappa_{\text{eff}}} - 1}{\sqrt{\kappa_{\text{eff}}} + 1} \right)^n \quad \text{for} \quad n \in \{0, \ldots, k\}
\]

where \(\kappa_{\text{eff}} = \frac{\lambda_N}{\lambda_1}\).
Initialize with array/list of eigenvalues or Intervals object.
**eval_step**(step)
Evaluate bound for given step.

**get_step**(tol)
Return step at which bound falls below tolerance.

### class krypy.utils.BoundMinres(evals)

**Bases:** object

MINRES residual norm bound.

Computes a bound for the MINRES residual norm when the eigenvalues of the operator are given, see [Gre97].

**Parameters**
- **evals** – an array of eigenvalues \( \lambda_1, \ldots, \lambda_N \in \mathbb{R} \). The eigenvalues will be sorted internally such that \( \lambda_1 \leq \ldots \lambda_s < 0 = \lambda_{s+1} = \ldots = \lambda_{s+t-1} < \lambda_t \leq \ldots \lambda_N \) for \( s, t \in \mathbb{N} \) and \( s < t \).
- **steps** – (optional) the number of steps \( k \) to compute the bound for. If steps is None (default), then \( k = N \) is used.

**Returns**
array \([\eta_0, \ldots, \eta_k]\) with

\[
\eta_n = 2 \left( \sqrt{|\lambda_1 \lambda_N|} - \sqrt{|\lambda_s \lambda_t|} \right) \left( \frac{1}{\sqrt{|\lambda_1 \lambda_N|} + \sqrt{|\lambda_s \lambda_t|}} \right)
\]

for \( n \in \{0, \ldots, k\} \)

if \( s > 0 \). If \( s = 0 \), i.e., if the eigenvalues are non-negative, then the result of \( \text{bound_cg()} \) is returned.

Initialize with array/list of eigenvalues or Intervals object.

**static __new__(evals)**
Use BoundCG if all eigenvalues are non-negative.

**eval_step**(step)
Evaluate bound for given step.

**get_step**(tol)
Return step at which bound falls below tolerance.

### exception krypy.utils.ConvergenceError(msg, solver)

**Bases:** Exception

Raised when a method did not converge.

The ConvergenceError holds a message describing the error and the attribute solver through which the last approximation and other relevant information can be retrieved.

### class krypy.utils.Givens(x)

**Bases:** object

Compute Givens rotation for provided vector \( x \).

Computes Givens rotation \( G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \) such that \( Gx = \begin{bmatrix} r \\ 0 \end{bmatrix} \).

**apply**(x)
Apply Givens rotation to vector \( x \).

### class krypy.utils.House(x)

**Bases:** object

Compute Householder transformation for given vector.

Initialize Householder transformation \( H \) such that \( Hx = \alpha \|x\|_2 e_1 \) with \( |\alpha| = 1 \)
KryPy Documentation, Release 2.1.7


**apply** (x)

Apply Householder transformation to vector x.

Applies the Householder transformation efficiently to the given vector.

**matrix**()

Build matrix representation of Householder transformation.

Builds the matrix representation \( H = I - \beta vv^\ast \).

Use with care! This routine may be helpful for testing purposes but should not be used in production codes for high dimensions since the resulting matrix is dense.

```python
class krypy.utils.IdentityLinearOperator(shape)
    Bases: krypy.utils.LinearOperator

class krypy.utils.LinearOperator(shape, dtype, dot=None, dot_adj=None)
    Bases: object

    Linear operator.
    Is partly based on the LinearOperator from scipy (BSD License).

    __add__ (X)

    __mul__ (X)

    __neg__ ()

    __pow__ (X)

    __repr__ ()

    __rmul__ (X)

    __sub__ (X)

    adj

    dot (X)

    dot_adj (X)

class krypy.utils.MatrixLinearOperator(A)
    Bases: krypy.utils.LinearOperator

    __repr__ ()

class krypy.utils.NormalizedRootsPolynomial(roots)
    Bases: object

    A polynomial with specified roots and \( p(0)=1 \).

    Represents the polynomial

    \[
    p(\lambda) = \prod_{i=1}^{n} \left(1 - \frac{\lambda}{\theta_i}\right).
    \]

    Parameters roots -- array with roots \( \theta_1, \ldots, \theta_n \) of the polynomial and roots.

    shape==\( (n,) \).

    __call__ (points)

    Evaluate polynomial at given points.

    Parameters points -- a point \( x \) or array of points \( x_1, \ldots, x_m \) with points.

    shape==\( (m,) \).

    Returns \( p(x) \) or array of shape \( (m,) \) with \( p(x_1), \ldots, p(x_m) \).

```

2.1. krypy Package 17
minmax_candidates()
    Get points where derivative is zero.

Useful for computing the extrema of the polynomial over an interval if the polynomial has real roots.
In this case, the maximum is attained for one of the interval endpoints or a point from the result of this
function that is contained in the interval.

class krypy.utils.Projection(X=None, Y=None, ip_B=None, orthogonalize=True, iterations=2)
Bases: object

Generic projection.

This class can represent any projection (orthogonal and oblique) on a N-dimensional Hilbert space. A
projection is a linear operator \( P \) with \( P^2 = P \). A projection is uniquely defined by its range \( \mathcal{V} := \text{range}(P) \)
and its kernel \( \mathcal{W} := \ker(P) \); this projection is called \( P_{\mathcal{V},\mathcal{W}} \).

Let \( X \) and \( Y \) be two full rank arrays with shape\(=(N,k)\) and let \( \mathcal{X} \oplus \mathcal{Y} \perp = \mathbb{C}^N \) where \( \mathcal{X} := \text{colspan}(X) \)
and \( \mathcal{Y} := \text{colspan}(Y) \). Then this class constructs the projection \( P_{\mathcal{X},\mathcal{Y}^\perp} \). The requirement \( \mathcal{X} \oplus \mathcal{Y}^\perp = \mathbb{C}^N \)
is equivalent to \( \langle X,Y \rangle \) being nonsingular.

Parameters

- \( X \) – array with shape\(=(N,k)\) and rank\((X) = k.\)
- \( Y \) – (optional) None or array with shape\(=(N,k)\) and rank\((X) = k.\) If \( Y \) is None then \( Y \) is set to \( X \) which means that the resulting projection is orthogonal.
- \( \text{ip}_B \) – (optional) inner product, see \text{inner}(). None, a \text{numpy.array} or a \text{LinearOperator} is preferred due to the applicability of the proposed algorithms
  in [Ste11], see below.
- \( \text{orthogonalize} \) – (optional) \( True \) orthogonalizes the bases provided in \( X \) and \( Y \) with respect to the inner product defined by \text{ip}_B. \) Defaults to \( True \) as the orthogonalization
  is suggested by the round-off error analysis in [Ste11].
- \( \text{iterations} \) – (optional) number of applications of the projection. It was suggested in
  [Ste11] to use 2 iterations (default) in order to achieve high accuracy (“twice is enough” as
  in the orthogonal case).

This projection class makes use of the round-off error analysis of oblique projections in the work of Stew-
art [Ste11] and implements the algorithms that are considered as the most stable ones (e.g., the XQRY
representation in [Ste11]).

apply \((a, \text{return}_Ya=False)\)
Apply the projection to an array.

The computation is carried out without explicitly forming the matrix corresponding to the projection
(which would be an array with shape\(=(N,N)\)).

See also \_apply().

apply_adj \((a)\)

apply_complement \((a, \text{return}_Ya=False)\)
Apply the complementary projection to an array.

Parameters \( z \) – array with shape\(=(N,m)\).

Returns \( P_{\mathcal{Y}^\perp,\mathcal{X}} z = z - P_{\mathcal{X},\mathcal{Y}^\perp} z.\)

apply_complement_adj \((a)\)

matrix()
Builds matrix representation of projection.

Builds the matrix representation \( P = X(Y, X)^{-1}(Y, I_N).\)

Use with care! This routine may be helpful for testing purposes but should not be used in production
codes for high dimensions since the resulting matrix is dense.
operator()
Get a LinearOperator corresponding to apply().

Returns a LinearOperator that calls apply().

operator_complement()
Get a LinearOperator corresponding to apply_complement().

Returns a LinearOperator that calls apply_complement().

class krypy.utils.Timer
Bases: list

Measure execution time of multiple code blocks with `with`.

Example:

```python
t = Timer()
with t:
    print('time me!')
print('don\’t time me!')
with t:
    print('time me, too!')
print(t)
```

Result:

```
time me!
don’t time me!
time me, too!
[6.389617919921875e-05, 6.008148193359375e-05]
```

__enter__()

__exit__ (a, b, c)

krypy.utils.angles (F, G, ip_B=None, compute_vectors=False)
Principal angles between two subspaces.

This algorithm is based on algorithm 6.2 in Knyazev, Argentati. Principal angles between subspaces in an A-based scalar product: algorithms and perturbation estimates. 2002. This algorithm can also handle small angles (in contrast to the naive cosine-based svd algorithm).

Parameters

- `F` – array with shape==\((N, k)\).
- `G` – array with shape==\((N, l)\).
- `ip_B` – (optional) angles are computed with respect to this inner product. See `inner()`.
- `compute_vectors` – (optional) if set to `False` then only the angles are returned (default). If set to `True` then also the principal vectors are returned.

Returns

- `theta` if `compute_vectors==False`
- `theta`, `U`, `V` if `compute_vectors==True`

where

- `theta` is the array with shape==\((\max(k, l),)\) containing the principal angles \(0 \leq \theta_1 \leq \ldots \leq \theta_{\max(k, l)} \leq \frac{\pi}{2}\).
- `U` are the principal vectors from `F` with \(\langle U, U \rangle = I_k\).
- `V` are the principal vectors from `G` with \(\langle V, V \rangle = I_l\).
The principal angles and vectors fulfill the relation $\langle U, V \rangle = \begin{bmatrix} \cos(\Theta) & 0_{m,l-m} \\ 0_{l-m,m} & 0_{l-m,l-m} \end{bmatrix}$ where $m = \min\{k, l\}$ and $\cos(\Theta) = \text{diag}(\cos(\theta_1), \ldots, \cos(\theta_m))$. Furthermore, $\theta_{m+1} = \ldots = \theta_{\max(k, l)} = \frac{\pi}{2}$.

**krypy.utils.arnoldi** (*args, **kwargs)

**krypy.utils.arnoldi_res** (*A, V, H, ip_B=None*)

Measure Arnoldi residual.

**Parameters**

- **A** – a linear operator that can be used with scipy’s aslinearoperator with shape== (N, N).
- **V** – Arnoldi basis matrix with shape== (N, n).
- **H** – Hessenberg matrix: either $H_{n-1}$ with shape==(n, n-1) or $H_n$ with shape==(n, n) (if the Arnoldi basis spans an A-invariant subspace).
- **ip_B** – (optional) the inner product to use, see inner().

**Returns** either $\|AV_{n-1} - V_n H_{n-1}\|$ or $\|AV_n - V_n H_n\|$ (in the invariant case).

**krypy.utils.arnoldi_projected** (*H, P, k, ortho='mgs'*)

Compute (perturbed) Arnoldi relation for projected operator.

Assume that you have computed an Arnoldi relation

$$AV_n = V_{n+1} H_n$$

where $V_{n+1} \in \mathbb{C}^{N,n+1}$ has orthogonal columns (with respect to an inner product $\langle \cdot, \cdot \rangle$) and $H_n \in \mathbb{C}^{n+1,n}$ is an extended upper Hessenberg matrix.

For $k < n$ you choose full rank matrices $X \in \mathbb{C}^{n-k, n}$ and $Y \in \mathbb{C}^{n,k}$ and define $\tilde{X} := AV_n X = V_n H_{n-1} X$ and $\tilde{Y} := V_n Y$ such that $\langle \tilde{Y}, \tilde{X} \rangle = Y^* H_{n-1} X$ is invertible. Then the projections $P$ and $\tilde{P}$ characterized by

- $\tilde{P} x = x - \tilde{X} \langle \tilde{Y}, \tilde{X} \rangle^{-1} \langle \tilde{Y}, x \rangle$
- $P = I - H_{n-1} X (Y^* H_{n-1} X)^{-1} Y^*$

are well defined and $\tilde{P} V_{n+1} = [V_n P, v_{n+1}]$ holds.

This method computes for $i < n - k$ the Arnoldi relation

$$(\tilde{P} A + E_i) W_i = W_{i+1} G_i$$

where $W_{i+1} = V_n U_{i+1}$ has orthogonal columns with respect to $\langle \cdot, \cdot \rangle$. $G_i$ is an extended upper Hessenberg matrix and $E_i x = v_{n+1} F_i(W_i, x)$ with $F_i = [f_1, \ldots, f_i] \in \mathbb{C}^{1,i}$.

The perturbed Arnoldi relation can also be generated with the operator $P V_n \tilde{P} A$:

$$P V_n \tilde{P} A W_i = W_{i+1} G_i.$$

In a sense the perturbed Arnoldi relation is the best prediction for the behavior of the Krylov subspace $K_i(\tilde{P} A, \tilde{P} v_1)$ that can be generated only with the data from $K_{n+1}(A, v_1)$ and without carrying out further matrix-vector multiplications with A.

**Parameters**

- **H** – the extended upper Hessenberg matrix $H_n$ with shape==(n+1, n).
- **P** – the projection $P : \mathbb{C}^n \rightarrow \mathbb{C}^n$ (has to be compatible with get_linearoperator())
- **k** – the dimension of the null space of P.

**Returns**

$U$, $G$, $F$ where
• U is the coefficient matrix $U_{i+1}$ with shape=$(n, i+1)$.
• G is the extended upper Hessenberg matrix $G_i$ with shape=$(i+1, i)$.
• F is the error matrix $F_i$ with shape=$(1, i)$.

```
krypy.utils.bound_perturbed_gmres(pseudo, p, epsilon, deltas)
```

Compute GMRES perturbation bound based on pseudospectum

```
krypy.utils.gap(lamda, sigma, mode='individual')
```

Compute spectral gap.

Useful for eigenvalue/eigenvector bounds. Computes the gap $\delta \geq 0$ between two sets of real numbers $\lambda$ and $\sigma$. The gap can be computed in several ways and may not exist, see the mode parameter.

**Parameters**

- **lamda** – a non-empty set $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ given as a single real number or a list or `numpy.array` with real numbers.
- **sigma** – a non-empty set $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$. See lamda.
- **mode** – (optional). Defines how the gap should be computed. May be one of
  - 'individual' (default): $\delta = \min_{i \in \{1, \ldots, n\}} |\lambda_i - \sigma_j|$. With this mode, the gap is always be defined.
  - 'interval': determine the maximal $\delta$ such that $\Sigma \subset \mathbb{R} \setminus [\min_{\lambda \in \Lambda} \lambda - \delta, \max_{\lambda \in \Lambda} \lambda + \delta]$. If the gap does not exists, None is returned.

**Returns**

$\delta$ or None.

```
krypy.utils.get_linearoperator(shape, A, timer=None)
```

Enhances aslinearoperator if $A$ is None.

```
krypy.utils.hegedus(A, b, x0, M=None, Mr=None, ip_B=None)
```

Rescale initial guess appropriately (Hegedüs trick). The Hegedüs trick rescales the initial guess to $\gamma_{\min} x_0$ such that

$$
\|r_0\|_{M^{-1}} = \|M M_1 (b - A \gamma_{\min} x_0)\|_{M^{-1}} = \min_{\gamma \in \mathbb{C}} \|M M_1 (b - A \gamma x_0)\|_{M^{-1}} \leq \|MM_1 b\|_{M^{-1}}.
$$

This is achieved by $\gamma_{\min} = \frac{\langle z, MM_1 b \rangle_{M^{-1}}}{\|z\|_{M^{-1}}^2}$ for $z = MM_1 A x_0$ because then $r_0 = P_{z^*} b$. (Note that the right hand side of formula (5.8.16) in [LieS13] has to be complex conjugated.)

The parameters are the parameters you want to pass to gmres(), minres() or cg().

**Returns**

the adapted initial guess with the above property.

```
krypy.utils.inner(X, Y, ip_B=None)
```

Euclidean and non-Euclidean inner product.

`numpy.vdot` only works for vectors and `numpy.dot` does not use the conjugate transpose.

**Parameters**

- **X** – `numpy.array` with shape=$(N, m)$
- **Y** – `numpy.array` with shape=$(N, n)$
- **ip_B** – (optional) May be one of the following
  - None: Euclidean inner product.
  - a self-adjoint and positive definite operator $B$ (as `numpy.array` or `LinearOperator`). Then $X^* BY$ is returned.
  - a callable which takes 2 arguments $X$ and $Y$ and returns $\langle X, Y \rangle$. 

21. **krypy Package**

---
Caution: a callable should only be used if necessary. The choice potentially has an impact on the round-off behavior, e.g. of projections.

Returns numpy array \((X, Y)\) with shape\(= (m, n)\).

\section*{ip_euclid(X, Y)}
Euclidean inner product.

numpy.vdot only works for vectors and numpy.dot does not use the conjugate transpose.

**Parameters**
- \(X\) – numpy array with shape\(= (N, m)\)
- \(Y\) – numpy array with shape\(= (N, n)\)

**Returns** numpy array \(X^* Y\) with shape\(= (m, n)\).

\section*{norm(x, y=None, ip_B=None)}
Compute norm (Euclidean and non-Euclidean).

**Parameters**
- \(x\) – a 2-dimensional numpy.array
- \(y\) – a 2-dimensional numpy.array
- \(ip_B\) – see \texttt{inner()}.

Compute \(\sqrt{\langle x, y \rangle}\) where the inner product is defined via \(ip_B\).

\section*{norm_MMlr(M, Ml, A, Mr, b, x0, yk, inner_product=<function ip_euclid>)}

\section*{norm_squared(x, Mx=None, inner_product=<function ip_euclid>)}
Compute the norm\(^2\) w.r.t. to a given scalar product.

\section*{orthonormality(V, ip_B=None)}
Measure orthonormality of given basis.

**Parameters**
- \(V\) – a matrix \(V = [v_1, \ldots, v_n]\) with shape\(= (N, n)\).
- \(ip_B\) – (optional) the inner product to use, see \texttt{inner()}.

**Returns** \(\|I_n - \langle V, V \rangle\|_2\).

\section*{qr(X, ip_B=None, reorthos=1)}
QR factorization with customizable inner product.

**Parameters**
- \(X\) – array with shape\(= (N, k)\)
- \(ip_B\) – (optional) inner product, see \texttt{inner()}.
- \(reorthos\) – (optional) number of reorthogonalizations. Defaults to 1 (i.e. 2 runs of modified Gram-Schmidt) which should be enough in most cases (TODO: add reference).

**Returns** \(Q, R\) where \(X = QR\) with \(\langle Q, Q \rangle = I_k\) and \(R\) upper triangular.

\section*{ritz(H, V=None, hermitian=False, type='ritz')} Compute several kinds of Ritz pairs from an Arnoldi/Lanczos relation.

This function computes Ritz, harmonic Ritz or improved harmonic Ritz values and vectors with respect to the Krylov subspace \(K_n(A, v)\) from the extended Hessenberg matrix \(H_n\) generated with \(n\) iterations the Arnoldi algorithm applied to \(A\) and \(v\).

**Parameters**
- \(H\) – Hessenberg matrix from Arnoldi/Lanczos algorithm.


• **V** – (optional) Arnoldi/Lanczos vectors, \( V \in \mathbb{C}^{N,n+1} \). If provided, the Ritz vectors are also returned. The Arnoldi vectors have to form an orthonormal basis with respect to an inner product.

**Caution:** if you are using the Lanczos or Gram-Schmidt Arnoldi algorithm without reorthogonalization, then the orthonormality of the basis is usually lost. For accurate results it is advisable to use the Householder Arnoldi (\( \text{ortho}='\text{house}' \)) or modified Gram-Schmidt with reorthogonalization (\( \text{ortho}='\text{dmgs}' \)).

• **hermitian** – (optional) if set to \( \text{True} \) the matrix \( H_n \) must be Hermitian. A Hermitian matrix \( H_n \) allows for faster and often more accurate computation of Ritz pairs.

• **type** – (optional) type of Ritz pairs, may be one of 'ritz', 'harmonic' or 'harmonic_like'. Two choices of Ritz pairs fit in the following description:

Given two \( n \)-dimensional subspaces \( X, Y \subseteq \mathbb{C}^N \), find a basis \( z_1, \ldots, z_n \) of \( X \) and \( \theta_1, \ldots, \theta_n \in \mathbb{C} \) such that \( Az_i - \theta_i z_i \perp Y \) for all \( i \in \{1, \ldots, n\} \).

In this setting the choices are

- 'ritz': regular Ritz pairs, i.e. \( X = Y = K_n(A, v) \).
- 'harmonic': harmonic Ritz pairs, i.e. \( X = K_n(A, v) \) and \( Y = AK_n(A, v) \).
- 'harmonic_improved': the returned vectors \( U \) (and \( V \), if requested) are the same as with type='harmonic'. The \( \theta \) array contains the improved Ritz values \( \theta_i = u_i^* H_n u_i \), cf. section 2 in Morgan, Zeng. Harmonic Projection Methods for Large Non-symmetric Eigenvalue Problems. 1998. It can be shown that the residual norm of improved Ritz pairs is always less than or equal to the residual norm of the harmonic Ritz pairs. However, the improved Ritz pairs do not fit into the framework above since the orthogonality condition is lost.

**Returns**

- If \( V \) is not None then \( \theta \), \( U \), \( \text{resnorm} \), \( Z \) is returned.
- If \( V \) is None then \( \theta \), \( U \), \( \text{resnorm} \) is returned.

Where

- \( \theta \) are the Ritz values \( [\theta_1, \ldots, \theta_n] \).
- \( U \) are the coefficients of the Ritz vectors in the Arnoldi basis, i.e. \( z_i = V u_i \) where \( u_i \) is the \( i \)-th column of \( U \).
- \( \text{resnorm} \) is a residual norm vector.
- \( Z \) are the actual Ritz vectors, i.e. \( Z = \text{dot}(V, U) \).

The **krypy.utils.shape_vec** function:

Take a \( (n,) \) ndarray and return it as \( (n,1) \) ndarray.

The **krypy.utils.shape_vecs** function:

Reshape all ndarrays with \( \text{shape}==(n,) \) to \( \text{shape}==(n,1) \). Recognizes ndarrays and ignores all others.

The **krypy.utils.strakos** function:

Return the Strakoš matrix.

See [Str92].
2.1.5 Subpackages

tests Package

tests Module

krypy.tests.test_linsys.check_solver(sol, solver, ls, params)
krypy.tests.test_linsys.dictpick(d)
krypy.tests.test_linsys.dictproduct(d)
   enhance itertools product to process values of dicts
   example: d = {'a':[1,2], 'b':[3,4]} then list(dictproduct(d)) == [{a:1,b:3}, {a:1,b:4}, {a:2,b:3}, {a:2,b:4}]
krypy.tests.test_linsys.linear_systems_generator(A, **kwargs)
krypy.tests.test_linsys.run_solver(solver, ls, params)
krypy.tests.test_linsys.solver_params_generator(solver, ls)
krypy.tests.test_linsys.test_LinearSystem()
krypy.tests.test_linsys.test_solver()
krypy.tests.test_utils.assert_arnoldi(A, V, H, P, maxiter, ortho, M, ip_B, lanczos=False, arnoldi_const=1, ortho_const=1, proj_const=10, An=None)
krypy.tests.test_utils.get_ip_Bs()
krypy.tests.test_utils.get_matrices(spd=True, hpd=True, symm_indef=True, herm_indef=True, nonsymn=True, comp_nonsymn=True)
krypy.tests.test_utils.get_matrix_comp_nonsymn()
krypy.tests.test_utils.get_matrix_herm_indef()
krypy.tests.test_utils.get_matrix_hpd()
krypy.tests.test_utils.get_matrix_nonsymn()
krypy.tests.test_utils.get_matrix_spd()
krypy.tests.test_utils.get_matrix_symm_indef()
krypy.tests.test_utils.get_operators(A)
krypy.tests.test_utils.get_vecs(v)
krypy.tests.test_utils.run_NormalizedRootsPolynomial(roots)
krypy.tests.test_utils.run_angles(F, G, ip_B, compute_vectors)
krypy.tests.test_utils.run_arnoldi(A, v, maxiter, ortho, M, ip_B, An)
krypy.tests.test_utils.run_givens(x)
krypy.tests.test_utils.run_hegedus(A, b, x0, M, Ml, ip_B)
krypy.tests.test_utils.run_house(x)
krypy.tests.test_utils.run_projection(X, Y, ip_B, iterations)
krypy.tests.test_utils.run_qr(X, ip_B, reorthos)
krypy.tests.test_utils.run_ritz(A, v, maxiter, ip_B, Aevals, An, with_V, hermitian, type)
krypy.tests.test_utils.test_BoundCG()
krypy.tests.test_utils.test_BoundMinres()
krypy.tests.test_utils.test_Interval()
krypy.tests.test_utils.test_NormalizedRootsPolynomial()
krypy.tests.test_utils.test_arnoldi()
krypy.tests.test_utils.test_gap()
krypy.tests.test_utils.test_givens()
krypy.tests.test_utils.test_hegedus()
krypy.tests.test_utils.test_house()
krypy.tests.test_utils.test_projection()
krypy.tests.test_utils.test_qr()
krypy.tests.test_utils.test_ritz()

2.2 Bibliography
3.1 Installation

KryPy can be installed easily with the Python package installer by issuing `pip install krypy`. Alternatively, it can be installed by downloading the source from KryPy’s github page and then running `python setup.py install`.

3.2 Solve a linear system

The following code uses MINRES to solve a linear system with an indefinite diagonal matrix:

```python
from numpy import diag, linspace, ones, eye
from krypy.linalg import LinearSystem, Minres

# construct the linear system
A = diag(linspace(1, 2, 20))
A[0, 0] = -1e-5
b = ones(20)
linear_system = LinearSystem(A, b, self_adjoint=True)

# solve the linear system (approximate solution is solver.xk)
solver = Minres(linear_system)
```

3.3 Deflation

The vector $e_1$ can be used as a deflation vector to get rid of the small negative eigenvalue $-10^{-5}$:

```python
from krypy.deflation import DeflatedMinres
dsolver = DeflatedMinres(linear_system, U=eye(20, 1))
```
3.4 Recycling

The deflation subspace can also be determined automatically with a recycling strategy. Just for illustration, the same linear system is solved twice in the following code:

```python
from krypy.recycling import RecyclingMinres

# get recycling solver with approximate Krylov subspace strategy
rminres = RecyclingMinres(vector_factory='RitzApproxKrylov')

# solve twice
rsolver1 = rminres.solve(linear_system)
rsolver2 = rminres.solve(linear_system)
```

The convergence histories can be plotted by

```python
from matplotlib.pyplot import semilogy, show, legend
semilogy(solver.resnorms, label='original')
semilogy(dsolver.resnorms, label='exact deflation', ls='dotted')
semilogy(rsolver2.resnorms, label='automatic recycling', ls='dashed')
legend()
show()
```

which results in the following figure.
Indices and tables

• genindex
• modindex
• search


Python Module Index

k
krypy.deflation, 7
krypy.linsys, 3
krypy.recycling, 13
krypy.recycling.evaluators, 12
krypy.recycling.factories, 10
krypy.recycling.generators, 12
krypy.tests.test_linsys, 24
krypy.tests.test_utils, 24
krypy.utils, 14
Symbols

_ADeflationMixin (class in krypy.deflation), 7
_ADeflationVectorFactory (class in krypy.recycling.factories), 11
_AKrylovSolver (class in krypy.linsys), 6
_AProjection (class in krypy.deflation), 8
_ARecyclingSolver (class in krypy.recycling.linsys), 13
_ARitzSubsetsGenerator (class in krypy.recycling.generators), 12
__add__() (krypy.utils.LinearOperator method), 17
__call__() (krypy.utils.NormalizedRootsPolynomial method), 17
__enter__() (krypy.utils.Timer method), 19
__exit__() (krypy.utils.Timer method), 19
__mul__() (krypy.utils.LinearOperator method), 17
__neg__() (krypy.utils.LinearOperator method), 17
__new__() (krypy.utils.BoundMinres static method), 16
__pow__() (krypy.utils.LinearOperator method), 17
__repr__() (krypy.utils.LinearOperator method), 17
__repr__() (krypy.utils.MatrixLinearOperator method), 17
__rmul__() (krypy.utils.LinearOperator method), 17
__sub__() (krypy.utils.LinearOperator method), 17
_apply_projection() (krypy.deflation.DeflatedCg method), 7
_apply_projection() (krypy.deflation._DeflationMixin method), 8
_get_initial_residual() (krypy.deflation._DeflationMixin method), 8
_get_xk() (krypy.deflation._DeflationMixin method), 8
_solve() (krypy.deflation._DeflationMixin method), 8

A
adj (krypy.utils.LinearOperator attribute), 17
advance() (krypy.utils.Arnoldi method), 15
angles() (in module krypy.utils), 19
apply() (krypy.utils.Givens method), 16
apply() (krypy.utils.House method), 17
apply() (krypy.utils.Projection method), 18
apply_adj() (krypy.utils.Projection method), 18
apply_complement() (krypy.utils.Projection method), 18
apply_complement_adj() (krypy.utils.Projection method), 18

ArgumentError, 14
Arnoldi (class in krypy.utils), 14
arnoldit() (in module krypy.utils), 20
arnoldi_projected() (in module krypy.utils), 20
arnoldi_res() (in module krypy.utils), 20
Arnoldifier (class in krypy.deflation), 9
assert_arnoldi() (in module krypy.tests.test_utils), 24
AssumptionError, 14
AU (krypy.deflation.ObliqueProjection attribute), 8

B
B_ (krypy.deflation._DeflationMixin attribute), 7
bound_perturbed_gmres() (in module krypy.utils), 21
bound_pseudo() (in module krypy.deflation), 9
BoundCG (class in krypy.utils), 15
BoundMinres (class in krypy.utils), 16

C
C (krypy.deflation._DeflationMixin attribute), 8
Cg (class in krypy.linsys), 4
check_solver() (in module krypy.tests.test_linsys), 24
coeffs (krypy.deflation.Ritz attribute), 9
ConvergenceError, 14, 16
correct() (krypy.deflation.ObliqueProjection method), 8

D
DeflatedCg (class in krypy.deflation), 7
DeflatedGmres (class in krypy.deflation), 7
DeflatedMinres (class in krypy.deflation), 7
dictpick() (in module krypy.tests.test_linsys), 24
dictproduct() (in module krypy.tests.test_linsys), 24
dot() (krypy.utils.LinearOperator method), 17
dot_adj() (krypy.utils.LinearOperator method), 17

E
E (krypy.deflation._DeflationMixin attribute), 8
errnorms (krypy.linsys._KrylovSolver attribute), 6
estimate_time() (krypy.deflation._DeflationMixin method), 8
eval_step() (krypy.utils.BoundCG method), 15
eval_step() (krypy.utils.BoundMinres method), 16
evaluate() (krypy.recycling.evaluators.RitzApproxKrylov method), 12
RecyclingCg (class in krypy.recycling), 13
RecyclingGmres (class in krypy.recycling), 13
RecyclingMinres (class in krypy.recycling), 13
resnorms (krypy.deflation.Ritz attribute), 9
resnorms (krypy.linsys._KrylovSolver attribute), 7
Ritz (class in krypy.deflation), 9
ritz() (in module krypy.utils), 22
RitzApproxKrylov (class in krypy.recycling.evaluators), 12
RitzApriori (class in krypy.recycling.evaluators), 12
RitzExtremal (class in krypy.recycling.generators), 12
RitzFactory (class in krypy.recycling.factories), 10
RitzFactorySimple (class in krypy.recycling.factories), 11
RitzSmall (class in krypy.recycling.generators), 12
run_angles() (in module krypy.tests.test_utils), 24
run_arnoldi() (in module krypy.tests.test_utils), 24
run_givens() (in module krypy.tests.test_utils), 24
run_hegedus() (in module krypy.tests.test_utils), 24
run_house() (in module krypy.tests.test_utils), 24
run_NormalizedRootsPolynomial() (in module krypy.tests.test_utils), 24
run_projection() (in module krypy.tests.test_utils), 24
run_qr() (in module krypy.tests.test_utils), 24
run_ritz() (in module krypy.tests.test_utils), 24
run_solver() (in module krypy.tests.test_linsys), 24
TimeoutError, 14
shape_vec() (in module krypy.utils), 23
shape_vecs() (in module krypy.utils), 23
solve() (krypy.recycling.linsys._RecyclingSolver method), 13
solver_params_generator() (in module krypy.tests.test_linsys), 24
strakos() (in module krypy.utils), 23
test_angles() (in module krypy.tests.test_utils), 25
test_arnoldi() (in module krypy.tests.test_utils), 25
test_BoundCG() (in module krypy.tests.test_utils), 24
test_BoundMinres() (in module krypy.tests.test_utils), 24
test_gap() (in module krypy.tests.test_utils), 25
test_givens() (in module krypy.tests.test_utils), 25
test_hegedus() (in module krypy.tests.test_utils), 25
test_house() (in module krypy.tests.test_utils), 25
test_Interval() (in module krypy.tests.test_utils), 24
test_LinearSystem() (in module krypy.tests.test_linsys), 24
test_NormalizedRootsPolynomial() (in module krypy.tests.test_utils), 25
test_projection() (in module krypy.tests.test_utils), 25
test_qr() (in module krypy.tests.test_utils), 25
test_ritz() (in module krypy.tests.test_utils), 25
test_solver() (in module krypy.tests.test_linsys), 24
Timer (class in krypy.utils), 19
timings (krypy.recycling.linsys._RecyclingSolver attribute), 14
U
U (krypy.deflation.ObliqueProjection attribute), 8
UnionFactory (class in krypy.recycling.factories), 11
V
values (krypy.deflation.Ritz attribute), 9
X
xk (krypy.linsys._KrylovSolver attribute), 7