pentapy Documentation

Release 1.2.1.dev11

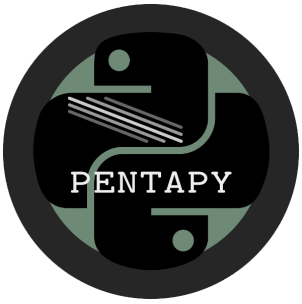
Sebastian Müller

Apr 10, 2024

CONTENTS

CHAPTER 1

PENTAPY QUICKSTART



pentapy is a toolbox to deal with pentadiagonal matrices in Python and solve the corresponding linear equation systems.

1.1 Installation

The package can be installed via pip. On Windows you can install WinPython to get Python and pip running.

```
pip install pentapy
```

There are pre-built wheels for Linux, MacOS and Windows for most Python versions.

To get the scipy solvers running, you have to install scipy or you can use the extra argument:

```
pip install pentapy[all]
```

Instead of “all” you can also typ “scipy” or “umfpack”.

1.2 References

The solver is based on the algorithms `PTRANS-I` and `PTRANS-II` presented by Askar et al. 2015.

1.3 Examples

Solving a pentadiagonal linear equation system

This is an example of how to solve a LES with a pentadiagonal matrix.

```
import numpy as np
import pentapy as pp

size = 1000
# create a flattened pentadiagonal matrix
M_flat = (np.random.random((5, size)) - 0.5) * 1e-5
V = np.random.random(size) * 1e5
# solve the LES with M_flat as row-wise flattened matrix
X = pp.solve(M_flat, V, is_flat=True)

# create the corresponding matrix for checking
M = pp.create_full(M_flat, col_wise=False)
# calculate the error
print(np.max(np.abs(np.dot(M, X) - V)))
```

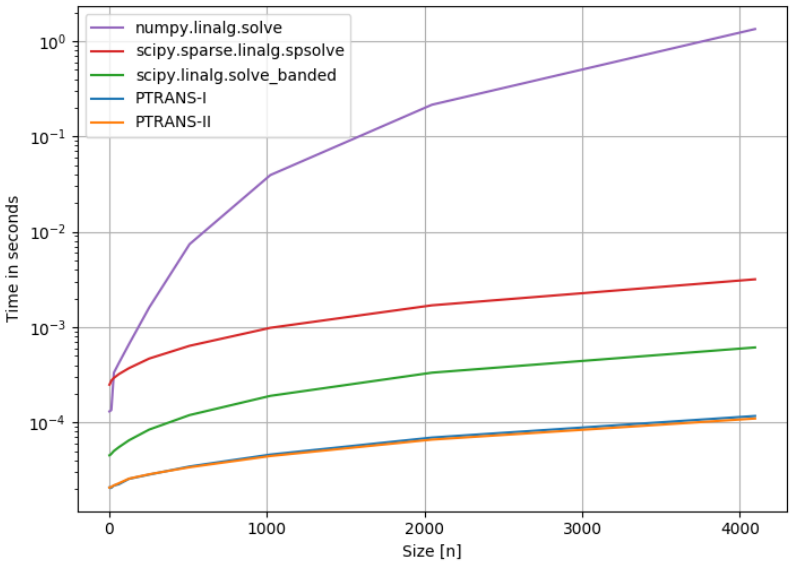
This should give something like:

```
4.257890395820141e-08
```

Performance

In the following, a couple of solvers for pentadiagonal systems are compared:

- Solver 1: Standard linear algebra solver of Numpy `np.linalg.solve` (link)
- Solver 2: `scipy.sparse.linalg.spsolve` (link)
- Solver 3: Scipy banded solver [`scipy.linalg.solve_banded`](scipy.github.io/devdocs/generated/scipy.linalg.solve_banded.
- Solver 4: pentapy.solve with `solver=1`
- Solver 5: pentapy.solve with `solver=2`



The performance plot was created with `perfplot` (link).

1.4 Requirements

- Numpy >= 1.14.5

Optional

- SciPy
- scikit-umfpack

1.5 License

MIT

CHAPTER 2

PENTAPY TUTORIALS

In the following you will find Tutorials on how to use pentapy.

2.1 Introduction: Solving a pentadiagonal system

Pentadiagonal systems arise in many areas of science and engineering, for example in solving differential equations with a finite difference sceme.

Theoretical Background

A pentadiagonal system is given by the equation: $M \cdot X = Y$, where M is a quadratic $n \times n$ matrix given by:

$$M = \begin{pmatrix} d_1 & d_1^{(1)} & d_1^{(2)} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ d_2^{(-1)} & d_2 & d_2^{(1)} & d_2^{(2)} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ d_3^{(-2)} & d_3^{(-1)} & d_3 & d_3^{(1)} & d_3^{(2)} & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & d_4^{(-2)} & d_4^{(-1)} & d_4 & d_4^{(1)} & d_4^{(2)} & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & d_{n-2}^{(-2)} & d_{n-2}^{(-1)} & d_{n-2} & d_{n-2}^{(1)} & d_{n-2}^{(2)} \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & d_{n-1}^{(-2)} & d_{n-1}^{(-1)} & d_{n-1} & d_{n-1}^{(1)} \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & d_n^{(-2)} & d_n^{(-1)} & d_n \end{pmatrix}$$

The aim is now, to solve this equation for X.

Memory efficient storage

To store a pentadiagonal matrix memory efficient, there are two options:

1. row-wise storage:

$$M_{\text{row}} = \begin{pmatrix} d_1^{(2)} & d_2^{(2)} & d_3^{(2)} & \cdots & d_{n-2}^{(2)} & 0 & 0 \\ d_1^{(1)} & d_2^{(1)} & d_3^{(1)} & \cdots & d_{n-2}^{(1)} & d_{n-1}^{(1)} & 0 \\ d_1 & d_2 & d_3 & \cdots & d_{n-2} & d_{n-1} & d_n \\ 0 & d_2^{(-1)} & d_3^{(-1)} & \cdots & d_{n-2}^{(-1)} & d_{n-1}^{(-1)} & d_n^{(-1)} \\ 0 & 0 & d_3^{(-2)} & \cdots & d_{n-2}^{(-2)} & d_{n-1}^{(-2)} & d_n^{(-2)} \end{pmatrix}$$

Here we see, that the numbering in the above given matrix was aiming at the row-wise storage. That means, the indices were taken from the row-indices of the entries.

2. column-wise storage:

$$M_{\text{col}} = \begin{pmatrix} 0 & 0 & d_1^{(2)} & \cdots & d_{n-4}^{(2)} & d_{n-3}^{(2)} & d_{n-2}^{(2)} \\ 0 & d_1^{(1)} & d_2^{(1)} & \cdots & d_{n-3}^{(1)} & d_{n-2}^{(1)} & d_{n-1}^{(1)} \\ d_1 & d_2 & d_3 & \cdots & d_{n-2} & d_{n-1} & d_n \\ d_2^{(-1)} & d_3^{(-1)} & d_4^{(-1)} & \cdots & d_{n-1}^{(-1)} & d_n^{(-1)} & 0 \\ d_3^{(-2)} & d_4^{(-2)} & d_5^{(-2)} & \cdots & d_n^{(-2)} & 0 & 0 \end{pmatrix}$$

The numbering here is a bit confusing, but in the column-wise storage, all entries written in one column were in the same column in the original matrix.

Solving the system using pentapy

To solve the system you can either provide M as a full matrix or as a flattened matrix in row-wise resp. col-wise flattened form.

If M is a full matrix, you call the following:

```
import pentapy as pp

M = ...  # your matrix
Y = ...  # your right hand side

X = pp.solve(M, Y)
```

If M is flattend in row-wise order you have to set the keyword argument `is_flat=True`:

```
import pentapy as pp

M = ...  # your flattened matrix
Y = ...  # your right hand side

X = pp.solve(M, Y, is_flat=True)
```

If you got a col-wise flattend matrix you have to set `index_row_wise=False`:

```
X = pp.solve(M, Y, is_flat=True, index_row_wise=False)
```

Tools

pentapy provides some tools to convert a pentadiagonal matrix.

diag_indices(n[, offset])	Get indices for the main or minor diagonals of a matrix.
shift_banded(mat[, up, low, col_to_row, copy])	Shift rows of a banded matrix.
create_banded(mat[, up, low, col_wise, dtype])	Create a banded matrix from a given quadratic Matrix.
create_full(mat[, up, low, col_wise])	Create a (n x n) Matrix from a given banded matrix.

2.2 Gallery

1. Example: Solving

Here we create a random row wise flattened matrix M_flat and a random right hand side for the pentadiagonal equation system.

After solving we calculate the absolute difference between the right hand side and the product of the matrix (which is transformed to a full quadratic one) and the solution of the system.

```
1.141211214417126e-08
```

```
import numpy as np

import pentapy as pp

size = 1000
# create a flattened pentadiagonal matrix
M_flat = (np.random.random((5, size)) - 0.5) * 1e-5
V = np.random.random(size) * 1e5
# solve the LES with M_flat as row-wise flattened matrix
X = pp.solve(M_flat, V, is_flat=True)

# create the corresponding matrix for checking
M = pp.create_full(M_flat, col_wise=False)
# calculate the error
print(np.max(np.abs(np.dot(M, X) - V)))
```

Total running time of the script: (0 minutes 0.009 seconds)

2. Example: Comparison

Here we compare the outcome of the PTRANS-I and PTRANS-II algorithm for a random input.

```
rel. diff X1 X2:  8.70816612593307e-15
max X1:  8466819268689.504
max X2:  8466819268689.651
max |M*X1 - V|:  2.5858753360807896e-08
max |M*X2 - V|:  2.3494067136198282e-08
```

```
import numpy as np

import pentapy as pp

size = 1000
# create a flattened pentadiagonal matrix
M_flat = (np.random.random((5, size)) - 0.5) * 1e-5
V = np.random.random(size) * 1e5
# compare the two solvers
X1 = pp.solve(M_flat, V, is_flat=True, solver=1)
X2 = pp.solve(M_flat, V, is_flat=True, solver=2)

# calculate the error
print("rel. diff X1 X2: ", np.max(np.abs(X1 - X2)) / np.max(np.abs(X1 + X2)))
print("max X1: ", np.max(np.abs(X1)))
print("max X2: ", np.max(np.abs(X2)))

M = pp.create_full(M_flat, col_wise=False)
# calculate the error
print("max |M*X1 - V|: ", np.max(np.abs(np.dot(M, X1) - V)))
print("max |M*X2 - V|: ", np.max(np.abs(np.dot(M, X2) - V)))
```

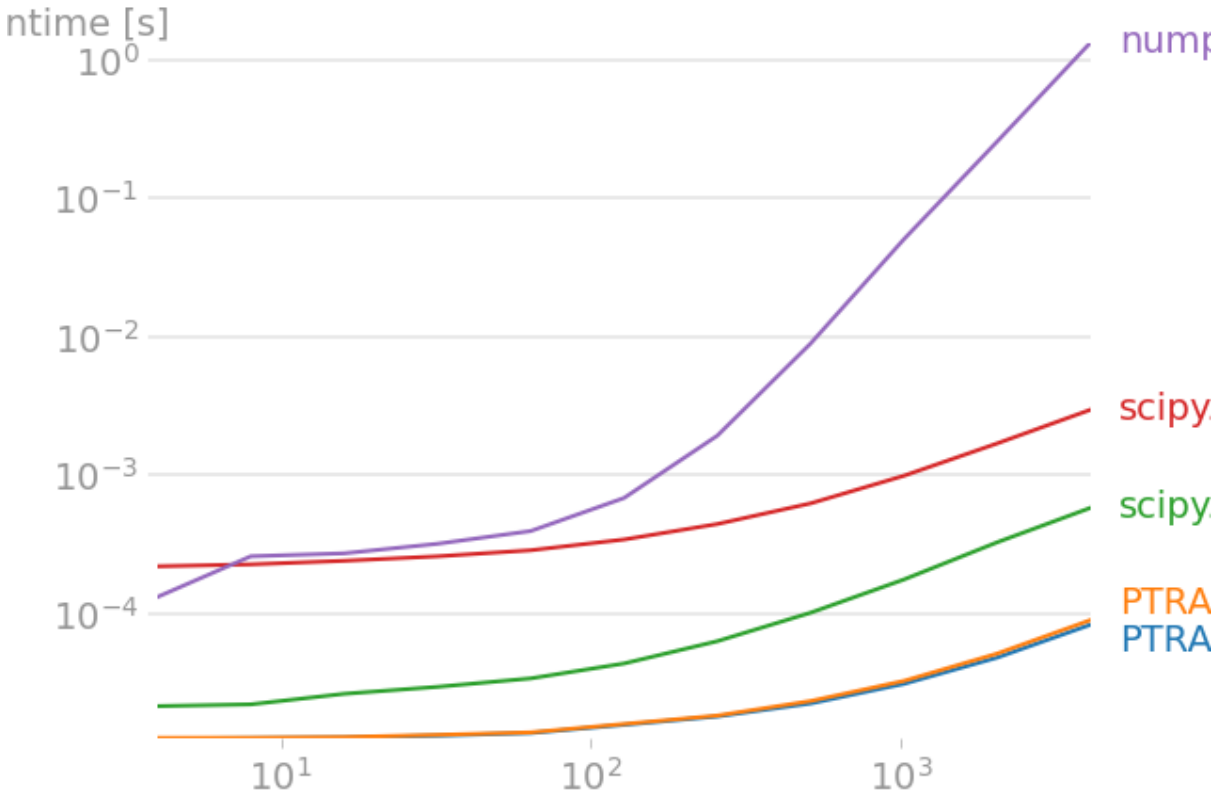
Total running time of the script: (0 minutes 0.009 seconds)

3. Example: Performance for flattened matrices

Here we compare all algorithms for solving pentadiagonal systems provided by pentapy (except umf).

To use this script you need to have the following packages installed:

- scipy
- perfplot
- matplotlib




```
Overall  100% 0:00:00
Kernels  100% 0:00:00
```

```python
import numpy as np
import perfplot

from pentapy import solve, tools


def get_les(size):
    mat = (np.random.random((5, size)) - 0.5) * 1e-5
    V = np.array(np.random.random(size) * 1e5)
    return mat, V


def solve_1(in_val):
    """PTRANS-I"""
    mat, V = in_val
    return solve(mat, V, is_flat=True, index_row_wise=False, solver=1)


def solve_2(in_val):
    """PTRANS-II"""
```

(continues on next page)

(continued from previous page)

```
    mat, V = in_val
    return solve(mat, V, is_flat=True, index_row_wise=False, solver=2)


def solve_3(in_val):
    mat, V = in_val
    return solve(mat, V, is_flat=True, index_row_wise=False, solver=3)


def solve_4(in_val):
    mat, V = in_val
    return solve(mat, V, is_flat=True, index_row_wise=False, solver=4)


def solve_5(in_val):
    mat, V = in_val
    M = tools.create_full(mat)
    return np.linalg.solve(M, V)


perfplot.show(
    setup=get_les,
    kernels=[solve_1, solve_2, solve_3, solve_4, solve_5],
    labels=[
        "PTRANS-I",
        "PTRANS-II",
        "scipy.linalg.solve_banded",
        "scipy.sparse.linalg.spsolve",
        "numpy.linalg.solve",
    ],
    n_range=[2**k for k in range(2, 13)],
    xlabel="Size [n]",
    logy=True,
)
```

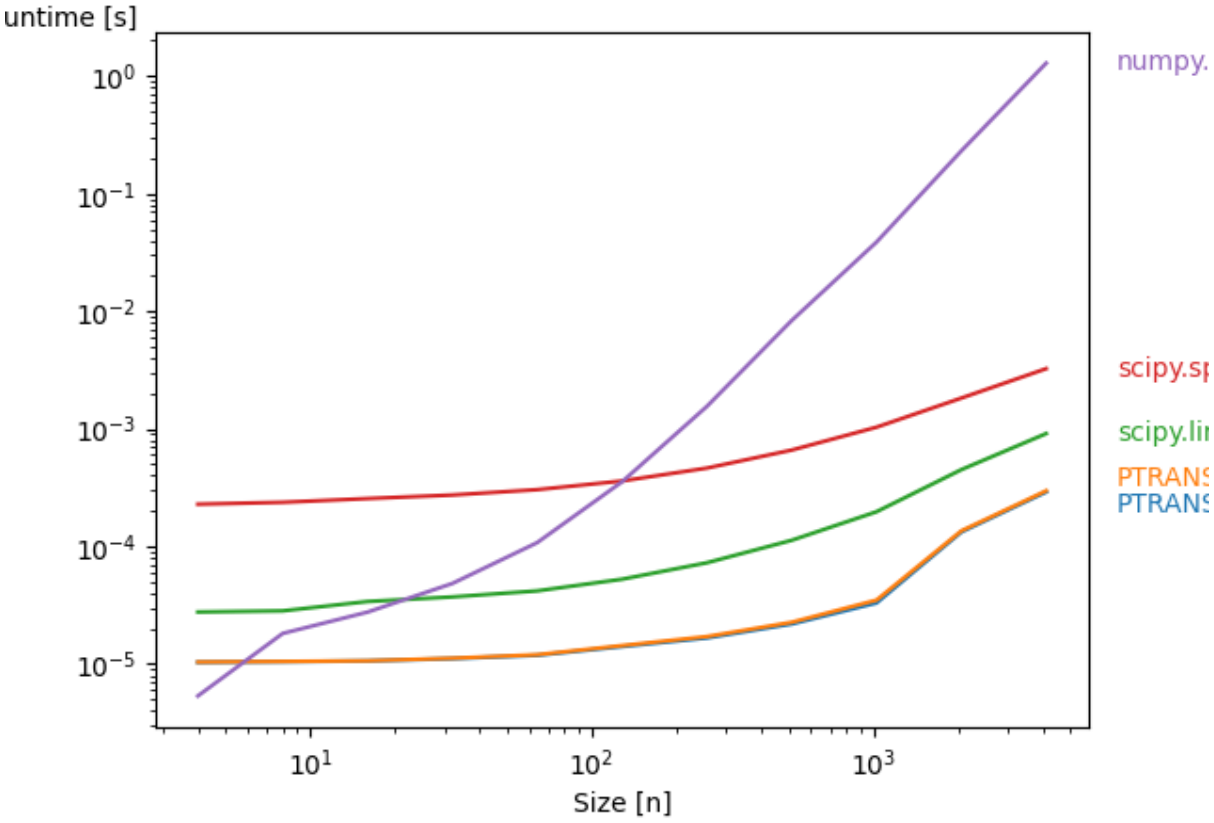
Total running time of the script: (0 minutes 38.795 seconds)

4. Example: Performance for full matrices

Here we compare all algorithms for solving pentadiagonal systems provided by pentapy (except umf) using a full quadratic matrix as input.

To use this script you need to have the following packages installed:

- scipy
- perfplot
- matplotlib




```
Overall  100% 0:00:00
Kernels  100% 0:00:00
```

```python
import numpy as np
import perfplot

from pentapy import solve, tools


def get_les(size):
    mat = (np.random.random((5, size)) - 0.5) * 1e-5
    V = np.array(np.random.random(size) * 1e5)
    M = tools.create_full(mat)
    return M, V


def solve_1(in_val):
    """PTRANS-I"""
    mat, V = in_val
    return solve(mat, V, is_flat=False, solver=1)


def solve_2(in_val):
```

(continues on next page)

(continued from previous page)

```
    """PTRANS-II"""
    mat, V = in_val
    return solve(mat, V, is_flat=False, solver=2)


def solve_3(in_val):
    mat, V = in_val
    return solve(mat, V, is_flat=False, solver=3)


def solve_4(in_val):
    mat, V = in_val
    return solve(mat, V, is_flat=False, solver=4)


def solve_5(in_val):
    mat, V = in_val
    return np.linalg.solve(mat, V)


perfplot.show(
    setup=get_les,
    kernels=[solve_1, solve_2, solve_3, solve_4, solve_5],
    labels=[
        "PTRANS-I",
        "PTRANS-II",
        "scipy.linalg.solve_banded",
        "scipy.sparse.linalg.spsolve",
        "numpy.linalg.solve",
    ],
    n_range=[2**k for k in range(2, 13)],
    xlabel="Size [n]",
    logy=True,
)
```

Total running time of the script: (0 minutes 37.397 seconds)

5. Example: Failing Corner Cases

Here we demonstrate that the solver PTRANS-I can fail to solve a given system. A warning is given in that case and the output will be a nan-array.

```
[nan nan nan nan]
```

```
import numpy as np

import pentapy as pp

# create a full pentadiagonal matrix
mat = np.array([[3, 2, 1, 0], [-3, -2, 7, 1], [3, 2, -1, 5], [0, 1, 2, 3]])
V = np.array([6, 3, 9, 6])
```

(continues on next page)

(continued from previous page)

```
# solve the LES with mat as a qudratic input matrix
X = pp.solve(mat, V, is_flat=False, solver=1)
print(X)
```

Total running time of the script: (0 minutes 0.001 seconds)

CHAPTER 3

PENTAPY API

3.1 Purpose

pentapy is a toolbox to deal with pentadiagonal matrizes in Python.

Subpackages

tools	The tools module of pentapy.

pentapy.tools

The tools module of pentapy.

The following functions are provided

diag_indices(n[, offset])	Get indices for the main or minor diagonals of a matrix.
shift_banded(mat[, up, low, col_to_row, copy])	Shift rows of a banded matrix.
create_banded(mat[, up, low, col_wise, dtype])	Create a banded matrix from a given quadratic Matrix.
create_full(mat[, up, low, col_wise])	Create a (n x n) Matrix from a given banded matrix.

pentapy.tools.diag_indices

pentapy.tools.**diag_indices**(*n*, *offset=0*)

Get indices for the main or minor diagonals of a matrix.

This returns a tuple of indices that can be used to access the main diagonal of an array *a* with `a.ndim == 2` dimensions and shape (n, n).

Parameters

- **n** (*int*) – The size, along each dimension, of the arrays for which the returned indices can be used.
- **offset** (*int, optional*) – The diagonal offset.

Returns

- **idx** (numpy.ndarray) – row indices
- **idy** (numpy.ndarray) – col indices

pentapy.tools.shift_banded

pentapy.tools.**shift_banded**(*mat*, *up=2*, *low=2*, *col_to_row=True*, *copy=True*)

Shift rows of a banded matrix.

Either from column-wise to row-wise storage or vice versa.

The Matrix has to be given as a flattend matrix. Either in a column-wise flattend form:

```
[[0        0        Dup2[2]  ... Dup2[N-2]  Dup2[N-1]  Dup2[N] ]
 [0        Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  Dup1[N] ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [Dlow1[0] Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] 0       ]
 [Dlow2[0] Dlow2[1] Dlow2[2] ... Dlow2[N-2] 0          0       ]]
```

Then use:

```
col_to_row=True
```

Or in a row-wise flattend form:

```
[[Dlow2[0]  Dlow2[1]  Dlow2[2]  ... Dlow2[N-2]  0          0       ]
 [Dlow1[0]  Dlow1[1]  Dlow1[2]  ... Dlow1[N-2]  Dlow1[N-1] 0       ]
 [Diag[0]   Diag[1]   Diag[2]   ... Diag[N-2]   Diag[N-1]  Diag[N] ]
 [0         Dlow1[1]  Dlow1[2]  ... Dlow1[N-2]  Dlow1[N-1] Dlow1[N]]
 [0         0         Dlow2[2]  ... Dlow2[N-2]  Dlow2[N-2] Dlow2[N]]]
```

Then use:

```
col_to_row=False
```

Dup1 and Dup2 are the first and second upper minor-diagonals and Dlow1 resp. Dlow2 are the lower ones. The number of upper and lower minor-diagonals can be altered.

Parameters

- **mat** (numpy.ndarray) – The Matrix or the flattened Version of the pentadiagonal matrix.
- **up** (int) – The number of upper minor-diagonals. Default: 2
- **low** (int) – The number of lower minor-diagonals. Default: 2
- **col_to_row** (bool, optional) – Shift from column-wise to row-wise storage or vice versa. Default: True
- **copy** (bool, optional) – Copy the input matrix or overwrite it. Default: True

Returns

Shifted bandend matrix

Return type

numpy.ndarray

pentapy.tools.create_banded

pentapy.tools.**create_banded**(*mat*, *up=2*, *low=2*, *col_wise=True*, *dtype=None*)

Create a banded matrix from a given quadratic Matrix.

The Matrix will to be returned as a flattend matrix. Either in a column-wise flattend form:

```
[[0        0        Dup2[2]  ... Dup2[N-2]  Dup2[N-1]  Dup2[N] ]
 [0        Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  Dup1[N] ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [Dlow1[0] Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] 0       ]
 [Dlow2[0] Dlow2[1] Dlow2[2] ... Dlow2[N-2] 0          0       ]]
```

Then use:

```
col_wise=True
```

Or in a row-wise flattend form:

```
[[Dup2[0]  Dup2[1]  Dup2[2]  ... Dup2[N-2]  0          0       ]
 [Dup1[0]  Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  0       ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [0        Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] Dlow1[N]]
 [0        0        Dlow2[2] ... Dlow2[N-2] Dlow2[N-2] Dlow2[N]]]
```

Then use:

```
col_wise=False
```

Dup1 and Dup2 or the first and second upper minor-diagonals and Dlow1 resp. Dlow2 are the lower ones. The number of upper and lower minor-diagonals can be altered.

Parameters

- **mat** (numpy.ndarray) – The full (n x n) Matrix.
- **up** (int) – The number of upper minor-diagonals. Default: 2
- **low** (int) – The number of lower minor-diagonals. Default: 2
- **col_wise** (bool, optional) – Use column-wise storage. If False, use row-wise storage. Default: True

Returns

Bandend matrix

Return type

numpy.ndarray

pentapy.tools.create_full

pentapy.tools.**create_full**(*mat*, *up=2*, *low=2*, *col_wise=True*)

Create a (n x n) Matrix from a given banded matrix.

The given Matrix has to be a flattend matrix. Either in a column-wise flattend form:

```
[[0        0        Dup2[2]  ... Dup2[N-2]  Dup2[N-1]  Dup2[N] ]
 [0        Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  Dup1[N] ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [Dlow1[0] Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] 0       ]
 [Dlow2[0] Dlow2[1] Dlow2[2] ... Dlow2[N-2] 0          0       ]]
```

Then use:

```
col_wise=True
```

Or in a row-wise flattend form:

```
[[Dup2[0]  Dup2[1]  Dup2[2]  ... Dup2[N-2]  0          0       ]
 [Dup1[0]  Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  0       ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [0        Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] Dlow1[N]]
 [0        0        Dlow2[2] ... Dlow2[N-2] Dlow2[N-2] Dlow2[N]]]
```

Then use:

```
col_wise=False
```

Dup1 and Dup2 or the first and second upper minor-diagonals and Dlow1 resp. Dlow2 are the lower ones. The number of upper and lower minor-diagonals can be altered.

Parameters

- **mat** (numpy.ndarray) – The flattened Matrix.
- **up** (int) – The number of upper minor-diagonals. Default: 2
- **low** (int) – The number of lower minor-diagonals. Default: 2
- **col_wise** (bool, optional) – Input is in column-wise storage. If False, use as row-wise storage. Default: True

Returns

Full matrix.

Return type

numpy.ndarray

Solver

Solver for a pentadiagonal equations system.

solve(mat, rhs[, is_flat, index_row_wise, ...])	Solver for a pentadiagonal system.

pentapy.solve

pentapy.**solve**(*mat*, *rhs*, *is_flat=False*, *index_row_wise=True*, *solver=1*)

Solver for a pentadiagonal system.

The matrix can be given as a full n x n matrix or as a flattend one. The flattend matrix can be given in a row-wise flattend form:

```
[[Dup2[0]  Dup2[1]  Dup2[2]  ... Dup2[N-2]  0          0       ]
 [Dup1[0]  Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  0       ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [0        Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] Dlow1[N]]
 [0        0        Dlow2[2] ... Dlow2[N-2] Dlow2[N-2] Dlow2[N]]]
```

Or a column-wise flattend form:

```
[[0        0        Dup2[2]  ... Dup2[N-2]  Dup2[N-1]  Dup2[N] ]
 [0        Dup1[1]  Dup1[2]  ... Dup1[N-2]  Dup1[N-1]  Dup1[N] ]
 [Diag[0]  Diag[1]  Diag[2]  ... Diag[N-2]  Diag[N-1]  Diag[N] ]
 [Dlow1[0] Dlow1[1] Dlow1[2] ... Dlow1[N-2] Dlow1[N-1] 0       ]
 [Dlow2[0] Dlow2[1] Dlow2[2] ... Dlow2[N-2] 0          0       ]]
```

Dup1 and Dup2 are the first and second upper minor-diagonals and Dlow1 resp. Dlow2 are the lower ones. If you provide a column-wise flattend matrix, you have to set:

```
index_row_wise=False
```

Parameters

- **mat** (numpy.ndarray) – The Matrix or the flattened Version of the pentadiagonal matrix.
- **rhs** (numpy.ndarray) – The right hand side of the equation system.
- **is_flat** (bool, optional) – State if the matrix is already flattend. Default: `False`
- **index_row_wise** (bool, optional) – State if the flattend matrix is row-wise flattend. Default: `True`
- **solver** (int or str, optional) –

  Which solver should be used. The following are provided:

  - `[1, "1", "PTRANS-I"]` : The PTRANS-I algorithm
  - `[2, "2", "PTRANS-II"]` : The PTRANS-II algorithm
  - `[3, "3", "lapack", "solve_banded"]` : scipy.linalg.solve_banded
  - `[4, "4", "spsolve"]` : The scipy sparse solver without umf_pack
  - `[5, "5", "spsolve_umf", "umf", "umf_pack"]` : The scipy sparse solver with umf_pack

  Default: 1

Returns
result – Solution of the equation system

Return type
numpy.ndarray

Tools

The following tools are provided:

diag_indices(n[, offset])	Get indices for the main or minor diagonals of a matrix.
shift_banded(mat[, up, low, col_to_row, copy])	Shift rows of a banded matrix.
create_banded(mat[, up, low, col_wise, dtype])	Create a banded matrix from a given quadratic Matrix.
create_full(mat[, up, low, col_wise])	Create a (n x n) Matrix from a given banded matrix.

CHAPTER 4

CHANGELOG

All notable changes to **pentapy** will be documented in this file.

4.1 1.3.0 - 2024-04

See #21

Enhancements

- added support for python 3.12
- added support for numpy 2
- build extensions with numpy 2 and cython 3

Changes

- dropped python 3.7 support
- dropped 32bit builds
- linted cython files
- increase maximal line length to 88 (black default)

4.2 1.2.0 - 2023-04

See #19

Enhancements

- added support for python 3.10 and 3.11
- add wheels for arm64 systems
- created `solver.pxd` file to be able to cimport the solver module
- added a `CITATION.bib` file

Changes

- move to `src/` based package structure
- dropped python 3.6 support
- move meta-data to pyproject.toml
- simplified documentation

Bugfixes

- determine correct version when installing from archive

4.3 1.1.2 - 2021-07

Changes

- new package structure with `pyproject.toml` (#15)
- Sphinx-Gallery for Examples
- Repository restructuring: use a single `main` branch
- use `np.asarray` in `solve` to speed up computation (#17)

4.4 1.1.1 - 2021-02

Enhancements

- Python 3.9 support

Changes

- GitHub Actions for CI

4.5 1.1.0 - 2020-03-22

Enhancements

- Python 3.8 support

Changes

- python only builds are no longer available
- Python 2.7 and 3.4 support dropped

4.6 1.0.3 - 2019-11-10

Enhancements

- the algorithms `PTRANS-I` and `PTRANS-II` now raise a warning when they can not solve the given system
- there are now switches to install scipy and umf solvers as extra requirements

Bugfixes

- multiple minor bugfixes

4.7 1.0.0 - 2019-09-18

Enhancements

- the second algorithm `PTRANS-II` from *Askar et al. 2015* is now implemented and can be used by `solver=2`
- the package is now tested and a coverage is calculated
- there are now pre-built binaries for Python 3.7
- the documentation is now available under https://geostat-framework.readthedocs.io/projects/pentapy

Changes

- pentapy is now licensed under the MIT license

4.8 0.1.1 - 2019-03-08

Bugfixes

- MANIFEST.in was missing in the 0.1.0 version

4.9 0.1.0 - 2019-03-07

This is the first release of pentapy, a python toolbox for solving pentadiagonal linear equation systems. The solver is implemented in cython, which makes it really fast.

PYTHON MODULE INDEX

p

INDEX