



# **pentapy Documentation**

***Release 1.1.1***

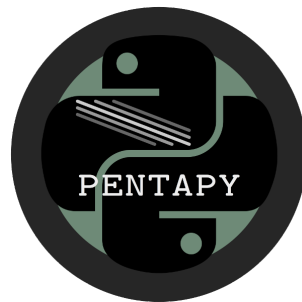
**Sebastian Mueller**

**Jun 19, 2021**



<b>1</b>	<b>pentapy Quickstart</b>	<b>1</b>
1.1	Installation . . . . .	1
1.2	References . . . . .	1
1.3	Examples . . . . .	2
	Solving a pentadiagonal linear equation system . . . . .	2
	Performance . . . . .	2
1.4	Requirements . . . . .	3
	Optional . . . . .	3
1.5	License . . . . .	3
<b>2</b>	<b>pentapy Tutorials</b>	<b>5</b>
2.1	Tutorial 1: Solving a pentadiagonal system . . . . .	5
	Theoretical Background . . . . .	5
	Memory efficient storage . . . . .	5
	Solving the system using pentapy . . . . .	6
	Tools . . . . .	6
	Example . . . . .	6
<b>3</b>	<b>pentapy API</b>	<b>9</b>
3.1	Purpose . . . . .	9
	Solver . . . . .	9
	Tools . . . . .	9
3.2	pentapy.core . . . . .	10
3.3	pentapy.tools . . . . .	11
	<b>Python Module Index</b>	<b>15</b>
	<b>Index</b>	<b>17</b>





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 (2.7, 3.4-3.7).

If your system is not supported and you want to have the Cython routines of pentapy installed, you have to provide a c-compiler and run:

```
pip install numpy cython  
pip install pentapy
```

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)))
```

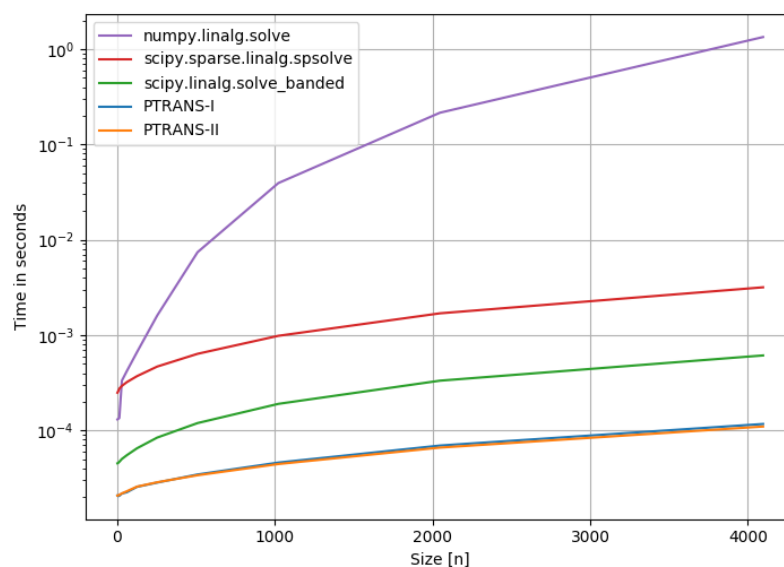
This should give something like:

```
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](https://scipy.github.io/devdocs/generated/scipy.linalg.solve_banded.html))
- 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  $\geq$  1.14.5

### Optional

- SciPy
- scikit-umfpack

## 1.5 License

MIT





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

### 2.1 Tutorial 1: Solving a pentadiagonal system

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

#### 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:

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

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:

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

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 flattened 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 flattened 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.

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

## Example

This is an example of how to solve a LES with a pentadiagonal matrix. The matrix is given as a row-wise flattened matrix, that is filled with random numbers. Afterwards the matrix is transformed to the full quadratic matrix to check the result.

```
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 small like:

```
4.257890395820141e-08
```



## CHAPTER 3

### 3.1 Purpose

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

#### Solver

Solver for a pentadiagonal equations system.

<code>solve(mat, rhs[, is_flat, index_row_wise, ...])</code>	Solver for a pentadiagonal system.
--	------------------------------------

#### Tools

The following tools are provided:

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

## 3.2 pentapy.core

The core module of pentapy.

The following functions are provided

---

<code>solve(mat, rhs[, is_flat, index_row_wise, ...])</code>	Solver for a pentadiagonal system.
--	------------------------------------

---

`pentapy.core.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  $n \times n$  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`

### 3.3 pentapy.tools

The tools module of pentapy.

The following functions are provided

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

`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(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]  ]]
```

(continues on next page)

(continued from previous page)

```
[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`

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

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` (*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] ]]
```

(continues on next page)



(continued from previous page)

```
[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 flattened 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_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`



**p**

`pentapy`, 9

`pentapy.core`, 10

`pentapy.tools`, 11



## C

`create_banded()` (*in module pentapy.tools*), 11  
`create_full()` (*in module pentapy.tools*), 11

## D

`diag_indices()` (*in module pentapy.tools*), 12

## P

`pentapy` (*module*), 9  
`pentapy.core` (*module*), 10  
`pentapy.tools` (*module*), 11

## S

`shift_banded()` (*in module pentapy.tools*), 12  
`solve()` (*in module pentapy.core*), 10