fastmat

Release 0.2.2

Sebastian Semper, Christoph Wagner

Apr 21, 2023

CONTENTS

CHAPTER

ONE

BLOCK DIAGONAL MATRIX

class fastmat.**BlockDiag**

Bases: *Matrix*

$$M = \operatorname{diag}\left\{\left(A_i\right)_i\right\},$$

where the A_i can be fast transforms of *any* type.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the blocks
>>> A = fm.Circulant(x_A)
>>> B = fm.Circulant(x_B)
>>> C = fm.Fourier(n)
>>> D = fm.Diag(x_D)
>>>
>>> # define the block
>>> # diagonal matrix
>>> M = fm.BlockDiag(A, B, C, D)
```

Assume we have two circulant matrices A and B, an N-dimensional Fourier matrix C and a diagonal matrix D. Then we define

$$M = \begin{bmatrix} A & & & \\ & B & & \\ & & C & \\ & & & D \end{bmatrix}.$$

Meta types can also be nested, so that a block diagonal matrix can contain products of block matrices as its entries. Note that the efficiency of the fast transforms decreases the more building blocks they have.

```
>>> import fastmat as fm
>>> # import the package
>>>
>>> # define the blocks
>>> A = fm.Circulant(x_A)
>>> B = fm.Circulant(x_B)
>>> F = fm.Fourier(n)
>>> D = fm.Diag(x_D)
>>>
>>> # define a product
```

(continues on next page)

(continued from previous page)

```
>>> P = fm.Product(A.H, B)
>>>
>>> # define the block
>>> # diagonal matrix
>>> M = fm.BlockDiag(P, F, D)
```

Assume we have a product P of two matrices A^{H} and B, an N-dimensional Fourier matrix $\mathcal{F}$ and a diagonal matrix D. Then we define

$$M = \begin{bmatrix} A^{\mathrm{H}} \cdot B & & \\ & \mathcal{F} & \\ & & D \end{bmatrix}.$$

Todo:

- BlockDiag should simply skip all Zero Matrices (flag them as "None")?

`__init__()`

Initialize a BlockDiag matrix instance.

Parameters

***matrices**
[*`fastmat.Matrix`*] The matrix instances to be put along the main diagonal of the block diagonal matrix, beginning at index (0, 0) with the first matrix.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

CHAPTER

TWO

BLOCK MATRIX

class fastmat.**Blocks**

Bases: *Matrix*

$$M = (A_{i,j})_{i,j},$$

where the $A_{i,j}$ can be a fast transforms of *any* type.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the blocks
>>> A = fm.Circulant(x_A)
>>> B = fm.Circulant(x_B)
>>> C = fm.Fourier(n)
>>> D = fm.Diag(x_D
>>>
>>> # define the block
>>> # matrix row-wise
>>> M = fm.Blocks([[A,B],[C,D]])
```

Assume we have two circulant matrices A and B, an N-dimensional Fourier matrix C and a diagonal matrix D. Then we define

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}.$$

Todo:

- Blocks should simply skip all Zero Matrices (flag them as “None”)?

__init__()

Initialize a Blocks matrix instance.

Parameters

arrMatrices
[iterable] A 2d iterable of py:class:*fastmat.Matrix* instances. All matrices must form a consistent grid over all instances of the 2d iterable. The inner iterable defines one row of the block matrix whereas the outer iterable defines the stacking of these rows. All inner iterables must be of same length. Further, all matrix instances in a row must have equal height and all instances in a column must have equal width.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

CHAPTER

THREE

(MULTILEVEL) CIRCULANT CLASS

class fastmat.**Circulant**

Bases: *Partial*

This class provides a very general implementation of circulant matrices, which essentially realize a (possibly multidimensional) circular convolution.

This type of matrix is highly structured. A two-level circulant Matrix looks like:

```
>>> c_00 c_02 c_01   c_20 c_22 c_21   c_10 c_12 c_11
>>> c_01 c_00 c_02   c_21 c_20 c_22   c_11 c_10 c_12
>>> c_02 c_01 c_00   c_22 c_21 c_20   c_12 c_11 c_10
>>>
>>> c_10 c_12 c_11   c_00 c_02 c_01   c_20 c_22 c_21
>>> c_11 c_10 c_12   c_01 c_00 c_02   c_21 c_20 c_22
>>> c_12 c_11 c_10   c_02 c_01 c_00   c_22 c_21 c_20
>>>
>>> c_20 c_22 c_21   c_10 c_12 c_11   c_00 c_02 c_01
>>> c_21 c_20 c_22   c_11 c_10 c_12   c_01 c_00 c_02
>>> c_22 c_21 c_20   c_12 c_11 c_10   c_02 c_01 c_00
```

This shows that one can define an L-level Circulant matrix by a tensor of order L. By design circulant matrices are always square matrices.

__init__()

Initialize Multilevel Circulant matrix instance.

Also see the special options of `fastmat.Fourier`, which are also supported by this matrix and the general options offered by `fastmat.Matrix.__init__`.

Parameters

tenC

[`numpy.ndarray`] The generating nd-array tensor defining the circulant matrix. The matrix data type is determined by the data type of this array.

****options**

[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix* and *fastmat.Fourier*.

tenC

Return the matrix-defining column vector of the circulant matrix

CHAPTER

FOUR

DIAGONAL MATRIX

class fastmat.**Diag**

Bases: *Matrix*

$$x \mapsto \mathrm{diag}(d_1, \dots, d_n) \cdot x$$

A diagonal matrix is uniquely defined by the entries of its diagonal.

```
>>> # import the package
>>> import fastmat as fm
>>> import numpy as np
>>>
>>> # build the parameters
>>> n = 4
>>> d = np.array([1, 0, 3, 6])
>>>
>>> # construct the matrix
>>> D = fm.Diag(d)
```

This yields

$$d = (1, 0, 3, 6)^{\mathrm{T}}$$

$$D = \begin{bmatrix} 1 & & & \\ & 0 & & \\ & & 3 & \\ & & & 6 \end{bmatrix}$$

__init__()

Initialize a Diag matrix instance.

Parameters

vecD
[`numpy.ndarray`] The generating vector of the diagonal entries of this matrix.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

vecD

Return the matrix-defining vector of diagonal entries.

(read-only)

CHAPTER

FIVE

DIAGONAL BLOCK MATRIX

class fastmat.**DiagBlocks**

Bases: *Matrix*

For given $n, m \in \mathbb{N}$ this class allows to define a block matrix $M \in \mathbb{C}^{nm \times nm}$, where each block is a diagonal matrix $D_{ij} \in \mathbb{C}^{m \times m}$. This obviously allows efficient storage and computations.

```
>>> # import the package
>>> import fastmat as fm
>>> # define the sizes
>>> n,m = 2,
>>> # define the diagonals
>>> d = np.random.randn(
>>>        n,
>>>        n,
>>>        m)
>>> # define the block
>>> # matrix diagonal-wise
>>> M = fm.DiagBlocks(d)
```

We have randomly drawn the defining elements d from a standard Gaussian distribution, which results in

$$M = \begin{bmatrix} d_{1,1,1} & & & d_{1,2,1} & & \\ & d_{1,1,2} & & & d_{1,2,2} & \\ & & d_{1,1,3} & & & d_{1,2,3} \\ d_{2,1,1} & & & d_{2,2,1} & & \\ & d_{2,1,2} & & & d_{2,2,2} & \\ & & d_{2,1,3} & & & d_{2,2,3} \end{bmatrix}.$$

__init__()

Initialize DiagBlocks matrix instance.

Parameters

tenDiags
[`numpy.ndarray`] The generating 3d-array of the flattened diagonal tensor this matrix describes. The matrix data type is determined by the data type of this array.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

CHAPTER

SIX

IDENTITY MATRIX

class fastmat.**Eye**

Bases: *Matrix*

For $x \in \mathbb{C}^n$ we

map .. math::
x mapsto x.

note::
Eye.forward(x) returns the exact same object as the given input array x. Make sure to issue an explicit .copy() in case you need it!

The identity matrix only needs the dimension n of the vectors it acts on.

```
>>> # import the package
>>> import fastmat
>>> # set the parameter
>>> n = 10
>>> # construct the identity
>>> I = fastmat.Eye(n)
```

This yields the identity matrix I_{10} with dimension 10.

__init__()

Initialize Identity (Eye) matrix instance.

Parameters

order
[int] Size of the desired identity matrix [order x order].

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

CHAPTER

SEVEN

FOURIER MATRIX

class fastmat.**Fourier**

Bases: *Matrix*

The Fourier Transform realizes the mapping

$$x \mapsto \mathcal{F}_n \cdot x,$$

where the Fourier matrix $\mathcal{F}_n$ is uniquely defined by the size of the vectors it acts on.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define parameter
>>> n = 4
>>>
>>> # construct the matrix
>>> F = fm.Fourier(n)
```

This yields a Fourier $\mathcal{F}_4$ matrix of size 4. As a library to provide the Fast Fourier Transform we used the one provided by NumPy.

Todo:

- real valued transforms

__init__()

Initialize Fourier matrix instance.

Parameters

order
[int] The order of the DFT matrix represented by this matrix instance.

optimize
[bool, optional] Allow application of the Bluestein algorithm for badly conditioned fourier transform orders.

Defaults to True.

maxStage
[int, optional] Specify the maximum butterfly element size for the FFT. Larger values can reduce the required order for the FFTs computed in the Bluestein case. However, increasing only makes sense as long as an efficient implementation of the butterfly structures exist in your BLAS.

Defaults to 4, which is safe to assume on all architectures. However, most implementations support sizes of 5 and on some cpu architectures, also 7.

****options**
[optional] Additional optional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

`order`

Return the Order of the Fourier matrix.

(read-only)

CHAPTER

EIGHT

HADAMARD MATRIX

class fastmat.**Hadamard**

Bases: *Matrix*

A Hadamard Matrix is recursively defined as

$$H_n = H_1 \otimes H_{n-1},$$

where

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

and $H_0 = (1)$. Obviously the dimension of H_n is 2^n. The transform is realized with the Fast Hadamard Transform (FHT).

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the parameter
>>> n = 4
>>>
>>> # construct the matrix
>>> H = fm.Hadamard(n)
```

This yields a Hadamard matrix $\mathcal{H}_4$ of order 4, i.e. with 16 rows and columns.

The algorithm we used is described in *[2]* and was implemented in Cython *[3]*.

__init__()

Initialize Hadamard matrix instance.

Parameters

order
[int] The order of the Hadamard matrix to generate. The matrix data type is `numpy.int8`

****options**
[optional] Optional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

order

Return the order of the hadamard matrix.

CHAPTER

NINE

KRONECKER PRODUCT

class fastmat.**Kron**

Bases: *Matrix*

For matrices $A_i \in \mathbb{C}^{n_i \times n_i}$ for $i = 1, \dots, k$ the Kronecker product

$$A_1 \otimes A_2 \otimes \dots \otimes A_k$$

can be defined recursively because of associativity from the Kronecker product of $A \in \mathbb{C}^{n \times m}$ and $B \in \mathbb{C}^{r \times s}$ defined as

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \dots & a_{nm}B \end{bmatrix}.$$

We make use of a decomposition into a standard matrix product to speed up the matrix-vector multiplication which is introduced in *[4]*. This then yields multiple benefits:

- It already brings down the complexity of the forward and backward projection if the factors A_i have no fast transformations.
- It is not necessary to compute the matrix representation of the product, which saves *a lot* of memory.
- When fast transforms of the factors are available the calculations can be sped up further.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the factors
>>> C = fm.Circulant(x_C)
>>> H = fm.Hadamard(n)
>>>
>>> # define the Kronecker
>>> # product
>>> P = fm.Kron(C.H, H)
```

Assume we have a circulant matrix C with first column x_c and a Hadamard matrix $\mathcal{H}_n$ of order n. Then we define

$$P = C^{\mathrm{H}} \otimes H_n.$$

`__init__()`

Initialize a Kron matrix instance.

Parameters

***matrices**
[*fastmat.Matrix*] The matrix instances to form a kronecker product of. Currently only square matrices are supported as kronecker product terms.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

CHAPTER

TEN

LFSR CIRCULANT MATRIX

class fastmat.**LFSRCirculant**

Bases: *Matrix*

Linear Feedback Shift Registers (LFSR) as implemented in this class are finite state machines generating sequences of symbols from the finite field $F = [-1, +1]$. A shift register of size N is a cascade of N storage elements a_n for $n = 0, \dots, N-1$, each holding one symbol of F. The state of the shift register is defined by the states of $a_0, \dots, a_{N-1}$. [5]

The next state of the register is generated from the current state by moving the contents of each storage element to the next lower index by setting $a_{n-1} = a_n$ for $n \geq 1$, hence the name shift register. The element a_0 of the current state is discarded completely in the next state. A subset T of all storage elements with cardinality of 1 or greater is used for generating the next symbol a_{N-1} by multiplication within F. T is called the tap configuration of the shift register.

The output sequence of the register is the sequence of symbols a_0 for each state of the register. When the shift register repeats one of its previous states after L state transistions, the output sequence also repeats and thus is periodic with a length L. Evaluation of the sequence starts with all storage elements set to an initial state I. Only periodic sequences of length $L > 1$ are considered if they also repeat all states including the initial state and thus form a hamilton circle an the graph corresponding to the chosen shift register size N and tap configuration T.

Instanciation of this matrix class requires supplying the requested register size N, the tap configuration and the initial state. The latter two are required to be supplied as binary words of up to N bits. A one bit on position i in the tap configuration adds a_i as *feedback tap* to T. At least one feedback tap must be supplied. The bits in the given initial state word r will be mapped to the initial register state, where $r_n = 0$ sets $a_n = +1$ and $r_n = 1$ sets $a_n = -1$. If no r is given, it is assumed to be all-ones.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # construct the parameter
>>> polynomial = 0b11001
>>> start = 0b1010
>>>
>>> # construct the matrix
>>> L = fm.LFSRCirculant(polynomial, start)
>>> s = L.vecC
```

This yields a Circulant matrix where the column-definition vector is the output of a LFSR of size 4, which is configured to generate a maximum length sequence of length 15 and a cyclic shift corresponding to the given initial state.

$$s = [+1, -1, +1, -1, -1, +1, +1, -1, +1, +1, +1, -1, -1, -1, -1]$$

$$L = \begin{bmatrix} +1 & -1 & -1 & & -1 \\ -1 & +1 & -1 & \dots & +1 \\ +1 & -1 & +1 & & -1 \\ & \vdots & & \ddots & \\ -1 & -1 & -1 & & +1 \end{bmatrix}$$

This class depends on `Hadamard`.

`__init__()`

Initialize a LFSR Circulant matrix instance.

The definition vector of the circulant matrix is defined by the output [+1/-1] of a binary Linear Feedback Shift Register (LFSR) with the given defining parameters over one period.

Parameters

polynomial
[int] The characteristic polynomial corresponding to the shift register sequence. Every set bit k in this value corresponds to one feedback tap at storage element k of the register or the monome x^k of the characterisctic polynomial that forms a cycle in the galois field GF2 of the order corresponding to the highest non-zero monome x^K in the polynomial.

start
[int] The initial value of the storage elements of the register.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

All optional arguments will be passed on to all *`fastmat.Matrix`* instances that are generated during initialization.

`order`

`period`

`polynomial`

`size`

Deprecated. Will be removed in future releases

`start`

`states`

Return the internal register states during the sequence.

(read-only)

`taps`

Deprecated. See .polynomial

`vecC`

Return the sequence defining the circular convolution.

(read only)

CHAPTER

ELEVEN

LOW RANK MATRIX

class fastmat.**LowRank**

Bases: *Product*

Generally one can consider the "complexity" of a matrix as the number of its rows n and columns m. The rank of a matrix $A \in \mathbb{C}^{n \times m}$ always obeys the bound

$$\mathrm{rk}(A) \leqslant \min\{n, m\}.$$

If one carries out the normal matrix vector multiplication, one assumes the rank to be essentially close to this upper bound. However if the rank of A is far lower than the minimum of its dimensions, then one carries out a lot of redundant tasks, when applying this matrix to a vector. But if one computes the singular value decomposition (SVD) of $A = U \Sigma V^{\mathrm{H}}$, then one can express A as a sum of rank-1 matrices as

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^{\mathrm{H}}.$$

If $r = \mathrm{rk}(A)$ is much smaller than the minimum of the dimensions, then one can save a lot of computational effort in applying A to a vector.

```
>>> # import the package
>>> import fastmat as fm
>>> import numpy as np
>>>
>>> # define all parameters
>>> S = np.random.randn(2)
>>> U = np.random.randn(20,2)
>>> V = np.random.randn(20,2)
>>>
>>> # define the matrix
>>> L = fm.LowRank(S, U, V)
```

We define a matrix $L = U S V^{\mathrm{H}} \in \mathbb{R}^{20 \times 20}$ with rank 2.

__init__()

Initialize a Low Rank matrix instance.

Parameters

vecS

[numpy.ndarray] The singular values as 1d vector corresponding to the singular value decomposition of the matrix.

arrU

[numpy.ndarray] A 2d array corresponding to U of the singular value decomposition of the matrix.

arrU
[`numpy.ndarray`] A 2d array corresponding to V of the singular value decomposition of the matrix.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

`arrU`

Return the array of left orthogonal vectors, i.e. the image

(read-only)

`arrV`

Return the array of right orthogonal vectors

the orthogonal complement of the kernel

(read-only)

`vecS`

Return the vector of non-zero singular values entries.

(read-only)

CHAPTER TWELVE

MATRIX BASE CLASS

`class fastmat.Matrix`

Bases: `object`

Matrix Base Class

Description: The baseclass of all matrix classes in fastmat. It also serves as wrapper around the standard Numpy Array *[1]*.

H

Return the hermitian transpose

(read-only)

T

Return the transpose of the matrix as fastmat class

(read-only)

`__init__()`

Initialize an instance of a fastmat matrix.

This is the baseclass for all fastmat matrices and serves as a wrapper to define a matrix based on a two dimensional ndarray. Any specialized matrix type in fastmat is derived from this base class and defines its own *__init__*.

Every *__init__* routine allows the specification of arbitrary keyworded arguments, which are passed in ***options*. Each specialized *__init__* routine processes the options it accepts and passes the rest on to the initialization routines in the base class to define the basic behaviour of the class.

Parameters

arrMatrix
[`numpy.ndarray`] A 2d array representing a dense matrix to be cast as a fastmat matrix.

forceContiguousInput
[bool, optional] If set, the input array is forced to be contiguous in the style as specified by *fortranStyle*. If the input array already fulfils the requirement nothing is done.

Defaults to False.

widenInputDatatype
[bool, optional] If set, the data type of the input array is promoted to at least match the output data type of the operation. Just like the *minType* option this parameter controls the accumulator width, however dynamically according to the output data type in this case.

Defaults to False.

fortranStyle
[bool, optional] Control the style of contiguousity to be enforced by forceConfiguousInput. If this option is set to True, Fortran-style ordering (contiguous along columns) is enforced, if False C-Style (contiguous along rows).

Defaults to True.

minType
[bool, optional] Specify a minimum data type for the input array to a transform. The input array data type will be promoted to at least the data type specified in this option before performing the actual transforms. Using this option is strongly advised for cases where small data types of both input array and matrix could cause range overflows otherwise, as the output data type promotion rules do not consider avoiding accumulator overflows due to performance reasons.

Defaults to `numpy.int8`.

bypassAllow
[bool, optional] Allow bypassing the implemented *`fastmat.Matrix.forward()`* and *`fastmat.Matrix.backward()`* transforms with dense matrix-vector products if runtime estimates suggest this is faster than using the implemented transforms. This requires valid calibration data to be available for the class of the to-be-created instance itself and the *`fastmat.Matrix`* base class at the time the new instance is created. If no valid performance calibration data exists this parameter is ignored and the implemented transforms will be used always.

Defaults to the value set in the package-wide `fastmat.flags` options.

bypassAutoArray
[bool, optional] Prevents the automatic generation of a dense matrix representation that would be used for bypassing the implemented transforms in case the performance profiles suggest this would be faster, if set to True. This is heavily advised if the matrix is unfeasibly large for a dense representation and does not feature fast transforms.

Defaults to the value as set in the package-wide :py:class`fastmat.flags` if no nested matrix of this instance has set this option to False. If just one has, this parameter defaults to False. If the matrix instance would disregard this, a nested instances' AutoArray function would be called implicitly through this instances' dense array constructur although this is disabled for the particular ndested matrix.

`array`

`backward()`

Backward Transform

Calculate the backward transform A^mathrm{H}*x where H is the hermitian transpose. Dimension-checking is performed to ensure valid fast transforms as these may succeed even when dimensions do not match. To support both single- and multidimensional input vectors x, single dimensional input will be reshaped to (n, 1) before processing and flattened to (n) after completion. This allows the use of both vectors and arrays. The actual transform code gets called by the callbacks specified in funcPython and funcCython, depending on the state of self._cythonCall.

Warning: Do not override this method

Note: The returned ndarray object may own its data, may be a view into another ndarray and may even be identical to the input array.

Parameters

arrX
[`numpy.ndarray`] The input data array of either 1d or 2d. 1d arrays will be reshaped to 2d during internal processing.

Returns

The result of the operation as `np.ndarray` with the same number of dimensions as *arrX*.

`bypassAllow`

`bypassAutoArray`

`colNormalized`

Return a column normalized matrix for this instance

(read-only)

`colNorms`

Return the column norms for this matrix instance

(read-only)

`complexity`

Complexity

(read-only)

Return the computational complexity of all functionality implemented in the class itself, not including calls of external code.

`conj`

Return the conjugate of the matrix as fastmat class

(read-only)

`content`

`dtype`

`estimateRuntime()`

Estimate the runtime of this matrix instances' transforms.

Parameters

numVectors
[int] Estimate the runtime for processing this number of vectors.

Returns

A tuple containing float estimates on the runtime of the *`fastmat.Matrix.forward()`* and the *`fastmat.Matrix.backward()`* transform if valid performance profiles are available to this matrix instance. If not, return (NaN, NaN)

`forward()`

Forward

Calculate the forward transform A * x. Dimension-checking is performed to ensure valid fast transforms as these may succeed even when dimensions do not match. To support both single- and multidimensional input vectors x, single dimensional input will be reshaped to (n, 1) before processing and flattened to (n) after completion. This allows the use of both vectors and arrays. The actual transform code gets called by the callbacks specified in funcPython and funcCython, depending on the state of self._cythonCall.

Warning: Do not override this method!

Note: The returned ndarray object may own its data, may be a view into another ndarray and may even be identical to the input array.

Parameters

arrX
[`numpy.ndarray`] The input data array of either 1d or 2d. 1d arrays will be reshaped to 2d during internal processing.

Returns

The result of the operation as `np.ndarray` with the same number of dimensions as *arrX*.

`fusedType`

`getArray()`

Return a dense array representation of this matrix.

`getCol()`

Return a column by index.

Parameters

idx
[int] Index of the column to return.

Returns

1d-`numpy.ndarray` holding the specified column.

`getColNormalized()`

Return a column normalized version of this matrix as fastmat matrix.

`getColNorms()`

Return a column normalized version of this matrix as fastmat matrix.

`getCols()`

Return a set of columns by index.

Parameters

indices
[int, slice or `numpy.ndarray`] If an integer is given, this is equal to the output of `` getCol`(indices). If a 1D vector or slice is given, a 2D ``

:py:class:`numpy.ndarray()` containing the columns as requested by *indices* is returned.

Returns

1D or 2D (depending on type of *indices*) `numpy.ndarray` holding the specified column(s).

`getComplexity()`

Return a transform complexity estimate for this matrix instance.

Returns a tuple containing the complexity estimates for the *`fastmat.Matrix.forward()`* and *`fastmat.Matrix.backward()`* transforms (in that order).

`getConj()`

Return the conjugate of this matrix as fastmat matrix.

`getGram()`

Return the gramian of this matrix as fastmat matrix.

`getH()`

Return the hermitian transpose of this matrix as fastmat matrix.

`getInverse()`

Return the hermitian transpose of this matrix as fastmat matrix.

`getLargestEigenValue()`

Largest Singular Value

For a given matrix $A \in \mathbb{C}^{n \times n}$, so A is square, we calculate the absolute value of the largest eigenvalue $\lambda \in \mathbb{C}$. The eigenvalues obey the equation

$$A \cdot v = \lambda \cdot v,$$

where v is a non-zero vector.

Input matrix A, parameter $0 < \varepsilon \ll 1$ as a stopping criterion Output largest eigenvalue $\sigma_{\max}(A)$

Note: This algorithm performs well if the two largest eigenvalues are not very close to each other on a relative scale with respect to their absolute value. Otherwise it might get trouble converging properly.

```
>>> # import the packages
>>> import numpy.linalg as npl
>>> import numpy as np
>>> import fastmat as fm
>>>
>>> # define the matrices
>>> n = 5
>>> H = fm.Hadamard(n)
>>> D = fm.Diag(np.linspace
>>>         1, 2 ** n, 2 ** n))
>>>
>>> K1 = fm.Product(H, D)
>>> K2 = K1.array
>>>
>>> # calculate the eigenvalue
```

(continues on next page)

(continued from previous page)

```
>>> x1 = K1.largestEigenValue
>>> x2 = npl.eigvals(K2)
>>> x2 = np.sort(np.abs(x2))[-1]
>>>
>>> # check if the solutions match
>>> print(x1 - x2)
```

We define a matrix-matrix product of a Hadamard matrix and a diagonal matrix. Then we also cast it into a numpy-array and use the integrated EVD. For demonstration, try to increase n>10`and see what happens.

getLargestEigenVec()

getLargestSingularValue()

Largest Singular Value

For a given matrix $A \in \mathbb{C}^{n \times m}$, we calculate the largest singular value $\sigma_{\max}(A) > 0$, which is the largest entry of the diagonal matrix $\Sigma \in \mathbb{C}^{n \times m}$ in the decomposition

$$A = U \Sigma V^{\mathrm{H}},$$

where U and V are matrices of the appropriate dimensions. This is done via the so called power iteration of $A^{\mathrm{H}} \cdot A$.

- Input matrix A, parameter $0 < \varepsilon \ll 1$ as a stopping criterion
- Output largest singular value $\sigma_{\max}(A)$

Note: This algorithm performs well if the two largest singular values are not very close to each other on a relative scale. Otherwise it might get trouble converging properly.

```
>>> # import packages
>>> import numpy.linalg as npl
>>> import numpy as np
>>> import fastmat
>>>
>>> # define involved matrices
>>> n = 5
>>> H = fm.Hadamard(n)
>>> F = fm.Fourier(2**n)
>>> K1 = fm.Kron(H, F)
>>> K2 = K1
>>>
>>> # calculate the largest SV
>>> # and a reference solution
>>> x1 = largestSingularValue(K1.largestSingularValue
>>> x2 = npl.svd(K2,compute_uv
>>> # check if they match
>>> print(x1-x2)
```

We define a Kronecker product of a Hadamard matrix and a Fourier matrix. Then we also cast it into a numpy-array and use the integrated SVD. For demonstration, try to increase n to >*10* and see what happens.

Returns

The largest singular value

`getLargestSingularVectors()`

`getPseudoInverse()`

Return the hermitian transpose of this matrix as fastmat matrix.

`getRow()`

Return a row by index.

Parameters

idx
[int] Index of the row to return.

Returns

1d-`numpy.ndarray` holding the specified row.

`getRowNormalized()`

Return a column normalized version of this matrix as fastmat matrix.

`getRowNorms()`

Return a row normalized version of this matrix as fastmat matrix.

`getRows()`

Return a set of rows by index.

Parameters

indices
[int, slice or `numpy.ndarray`] If an integer is given, this is equal to the output of `getRow`(indices). If a 1D vector or slice is given, a 2D :py:class:`numpy.ndarray()` containing the rows as requested by *indices* is returned.

Returns

1D or 2D (depending on type of *indices*) `numpy.ndarray` holding the specified row(s).

`getScipyLinearOperator()`

`getT()`

Return the transpose of this matrix as fastmat matrix.

`gram`

Return the gram matrix for this fastmat class

(read-only)

`inverse`

Return the inverse

(read-only)

`largestEV`

`largestEigenValue`

Return the largest eigenvalue for this matrix instance

(read-only)

`largestEigenVec`

Return the vector corresponding to the largest eigen value

(read-only)

`largestSV`

`largestSingularValue`

Return the largestSingularValue for this matrix instance

(read-only)

`largestSingularVectors`

Return the vectors corresponding to the largest singular value

This property returns a tuple (u, v) of the first columns of U and V in the singular value decomposition of

$$A = U \Sigma V^{\mathrm{H}},$$

which means that the tuple contains the leading left and right singular vectors of the matrix

(read-only)

`nbytes`

`nbytesReference`

`next()`

Stop iteration as __iter__ redirected here. Python2-Style.

`normalized`

`numCols`

`numM`

`numN`

`numRows`

`numpyType`

`profileBackward`

`profileForward`

`pseudoInverse`

Return the moore penrose inverse

(read-only)

`reference()`

Return explicit array reference of this matrix instance.

Return an explicit representation of the matrix without using any fastmat code. Provides type checks and raises errors if the matrix type (self.dtype) cannot hold the reference data. This implementation is meant to provide a reference version for testing and MUST not use any fastmat code for its implementation.

Returns

The array representation of this matrix instance as 2d `np.ndarray`.

`rowNormalized`

Return a column normalized matrix for this instance

(read-only)

`rowNorms`

Return the row norms for this matrix instance

(read-only)

`scipyLinearOperator`

Return a Representation as scipy's linear Operator

This property allows to make use of all the powerfull algorithms provided by scipy, that allow passing a linear operator to them, like optimization routines, system solvers or decomposition algorithms.

(read-only)

`shape`

`tag`

CHAPTER

THIRTEEN

OUTER PRODUCT

class fastmat.**Outer**

Bases: *Matrix*

The outer product is a special case of the Kronecker product of one-dimensional vectors. For given $a \in \mathbb{C}^n$ and $b \in \mathbb{C}^m$ it is defined as

$$x \mapsto a \cdot b^{\mathrm{T}} \cdot x.$$

It is clear, that this matrix has at most rank 1 and as such has a fast transformation.

```
>>> # import the package
>>> import fastmat as fm
>>> import numpy as np
>>>
>>> # define parameter
>>> n, m = 4, 5
>>> v = np.arange(n)
>>> h = np.arange(m)
>>>
>>> # construct the matrix
>>> M = fm.Outer(v, h)
```

This yields

$$v = (0, 1, 2, 3, 4)^{\mathrm{T}}$$

$$h = (0, 1, 2, 3, 4, 5)^{\mathrm{T}}$$

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 6 & 8 \\ 0 & 3 & 6 & 9 & 12 \end{bmatrix}$$

__init__()

Initialize a Outer product matrix instance.

Parameters

arrV
[`numpy.ndarray`] A 1d vector defining the column factors of the resulting matrix.

arrH
[`numpy.ndarray`] A 1d vector defining the row factors of the resulting matrix.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by `fastmat.Matrix`.

`vecH`

Return the matrix-defining vector of horizontal defining entries.

(read only)

`vecV`

Return the matrix-defining vector of vertical defining entries.

(read only)

CHAPTER

FOURTEEN

PARAMETRIC MATRIX

class fastmat.**Parametric**

Bases: *Matrix*

Let $f\mathbb{C}^2 \rightarrow \mathbb{C}$ be any function and two vectors $x \in \mathbb{C}^m$ and $y \in \mathbb{C}^n$ such that $(x_j, y_i) \in (f)$ for $i \in [n]$ and $j \in [m]$. Then the matrix $F \in \mathbb{C}^{n \times m}$ is defined as

$$F_{i,j} = f(x_j, y_i).$$

This class is not designed to be super fast, but memory efficient. This means, that everytime the forward or backward projections are called, the elements are generated according to the specified function on the fly.

Note: For small dimensions, where the matrix fits into memory, it is definately more efficient to cast the matrix to a regular Matrix object.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define parameter
>>> # function for the elements
>>> def f(x, y):
>>>     return x ** 2 - y ** 2
>>>
>>> # define the input array
>>> # for the function f
>>> x = np.linspace(1, 4, 4)
>>>
>>> # construct the transform
>>> F = fm.Parametric(x, x, f)
```

This yields

$$f : \mathbb{C} \rightarrow \mathbb{C}$$

$$(x_1, x_2)^{\mathrm{T}} \mapsto x_1^2 - x_2^2$$

$$x = (1, 2, 3, 4)^{\mathrm{T}}$$

$$F = \begin{bmatrix} 1 & 3 & 8 & 15 \\ -3 & 0 & 5 & 12 \\ -8 & -5 & 0 & 7 \\ -15 & -12 & -7 & 0 \end{bmatrix}$$

We used Cython [3] to get an efficient implementation in order to reduce computation time. Moreover, it is generally assumed the the defined function is able to use row and column broadcasting during evaluation. If this is not the case, one has to set the flag `rangeAccess` to `False`.

`__init__()`

Initialize a Parametric matrix instance.

Parameters

vecX
[`numpy.ndarray`] A 1d vector mapping the matrix column index to the x-values of funF.

vecY
[`numpy.ndarray`] A 1d vector mapping the matrix row index to the y-values of funF.

funF
[callable with arguments (x, y)] A function returning the element at index (x, y).

funDtype
[`numpy.dtype`, optional] Data type of the values returned by funF

Not specified by default (determine the datatype from the element at the first index funF(vecX[0], vecY[0]).

rangeAccess
[bool, optional] Allow passing row- and column vectors directly to funF. This can lead to significant speed-ups compared to single-element access.

Defaults to True.

****options**
[optional] Additional optional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

`fun`

Return the parameterizing function

(read only)

`vecX`

Return the support vector in X dimension.

(read only)

`vecY`

Return the support vector in Y dimension.

(read only)

CHAPTER

FIFTEEN

PARTIAL MATRIX

class fastmat.**Partial**

Bases: *Matrix*

Let $I \subset \{1, \dots, n\}$ and $J \subset \{1, \dots, m\}$ index sets and $M \in \mathbb{C}^{n \times m}$ a linear transform. Then the partial transform $M_{I,J}$ is defined as

$$x \in \mathbb{C}^m \mapsto (M_J \cdot x_J)_{i \in I}.$$

In other words, we select the rows I of M and columns J of M and rows J of x.

```
>>> # import the package
>>> import fastmat as fm
>>> import numpy as np
>>>
>>> # define the index set
>>> a = np.arange(n)
>>> am = np.mod(a, 2)
>>> b = np.array(am, dtype='bool')
>>> I = a[b]
>>>
>>> # construct the partial transform
>>> M = fm.Partial(F, I)
```

Let $\mathcal{F}$ be the n-dimensional Fourier matrix. And let I be the set of odd integers. Then we define a partial transform as

$$M = \mathcal{F}_I$$

__init__()

Initialize a Partial matrix instance.

Parameters

mat
[*fastmat.Matrix*] A fastmat matrix instance subject to partial access.

rows
[numpy.ndarray, optional] A 1d vector selecting rows of mat.

If *N* is of type bool it's size must match the height of mat and the values of *N* corresponds to taking/dumping the corresponding row.

If *N* is of type int it's values correspond to the indices of the rows of mat to select. The size of *N* then matches the height of the partialed matrix.

Defaults to selecting all rows.

cols
[`numpy.ndarray`, optional] A 1d vector selecting columns of mat. The behaviour is identical to *N*.

Defaults to selecting all columns.

****options**
[optional] Additional optional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

`colSelection`

Return the support of the base matrix which defines the partial

Subselected columns

(read only)

`indicesM`

Deprecated. See .colSelection

`indicesN`

Deprecated. See .rowSelection

`rowSelection`

Return the support of the base matrix which defines the partial

Subselected rows

(read only)

PERMUTATION MATRIX

class fastmat.**Permutation**

Bases: *Matrix*

For a given permutation $\sigma \in S_n$ and a vector $x \in \mathbb{C}^n$ we map

$$x \mapsto \left(x_{\sigma(i)}\right)_{i=1}^{n}.$$

```
>>> # import the package
>>> import fastmat
>>>
>>> # set the permutation
>>> sigma = np.array([3,1,2,0])
>>>
>>> # construct the identity
>>> P = fastmat.Permutation(sigma)
```

$$J = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

`__init__`()

Initialize a Permutation matrix instance.

Parameters

sigma
[`numpy.ndarray`] A 1d vector of type int mapping the row indices to column indices uniquely.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

sigma

Return the defining permutation

(read only)

CHAPTER

SEVENTEEN

MATRIX POLYNOMIAL

class fastmat.**Polynomial**

Bases: *Matrix*

For given coefficients $a_k, \ldots, a_0 \in \mathbb{C}$ and a linear mapping $A \in \mathbb{C}^{n \times n}$, we define

$$M = a_n A^n + a_{n-1} A^{n-1} + a_1 A + a_0 I.$$

The transform $M \cdot x$ can be calculated efficiently with Horner's method.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the transforms
>>> H = fm.Hadamard(n)
>>>
>>> # define the coefficient array
>>> arr_a = [1, 2 + 1j, -3.0, 0.0]
>>>
>>> # define the polynomial
>>> M = fm.Polynomial(H, arr_a)
```

Let H_n be the Hadamard matrix of order n. And let $a = (1, 2+i, -3, 0) \in \mathbb{C}^4$ be a coefficient vector, then the polynomial is defined as

$$M = H_n^3 + (2+i) H_n^2 - 3 H_n.$$

__init__()

Initialize a Polynomial matrix instance.

Parameters

mat
[*fastmat.Matrix*] A fastmat matrix instance subject to constructing the polynomial.

coeff
[numpy.ndarray] A 1d vector defining the polynomial coefficients.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

`coeff`

Return the polynomial coefficient vector.

(read only)

CHAPTER

EIGHTEEN

MATRIX-MATRIX PRODUCT

class fastmat.**Product**

Bases: *Matrix*

$$M = \prod_i A_i$$

where the A_i can be fast transforms of *any* type.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the product terms
>>> A = fm.Circulant(x_A)
>>> B = fm.Circulant(x_B)
>>>
>>> # construct the product
>>> M = fm.Product(A.H, B)
```

Assume we have two circulant matrices A and B. Then we define

$$M = A_c^{\mathrm{H}} B_c.$$

__init__()

Initialize a Product matrix instance.

Parameters

***matrices**
[*fastmat.Matrix* or scalar value] The matrix instances to form a matrix-matrix product of. You may also specify scalar values.

typeExpansion
[bool, optional] Expand the data type of input data to the data type specified with this paramter.

Defaults to a floating-point expansion of the promoted type of all nested matrices' (and scalar values') data types.

****options**
[optional] Additional optional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

CHAPTER NINETEEN

SPARSE MATRIX

class fastmat.**Sparse**

Bases: *Matrix*

$$x \mapsto Sx,$$

where S is a `scipy.sparse` matrix. To provide a high level of generality, the user has to make use of the standard `scipy.sparse` matrix constructors and pass them to `fastmat` during construction. After that a `Sparse` matrix can be used like every other type in `fastmat`

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # import scipy to get
>>> # the constructor
>>> import scipy.sparse.rand as r
>>>
>>> # set the matrix size
>>> n = 100
>>>
>>> # construct the sparse matrix
>>> S = fm.Sparse(
>>>         r(
>>>             n,
>>>             n,
>>>             0.01,
>>>             format='csr'
>>>         ))
```

This yields a random sparse matrix with 1% of its entries occupied drawn from a random distribution.

It is also possible to directly cast SciPy sparse matrices into the *fastmat`* sparse matrix format as follows.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # import scipy to get
>>> # the constructor
>>> import scipy.sparse as ss
>>>
>>> # construct the SciPy sparse matrix
>>> S_scipy = ss.csr_matrix(
```

(continues on next page)

(continued from previous page)

```
>>>         [
>>>             [1, 0, 0],
>>>             [1, 0, 0],
>>>             [0, 0, 1]
>>>         ]
>>>     )
>>>
>>> # construct the fastmat sparse matrix
>>> S = fm.Sparse(S_scipy)
```

Note: The `format` specifier drastically influences performance during multiplication of these matrices. From our experience `'csr'` works best in these cases.

For this matrix class we used the already tried and tested routines of SciPy *[1]*, so we merely provide a convenient wrapper to integrate nicely into `fastmat`.

`__init__()`

Initialize a Sparse matrix instance.

Parameters

matSparse
[`scipy.sparse.spmatrix`] A 2d scipy sparse matrix to be cast as a fastmat matrix.

****options**
[optional] Additional optional keyworded arguments. Supports all optional arguments supported by *`fastmat.Matrix`*.

`spArray`

Return the scipy sparse matrix .

(read only)

`spArrayH`

Return the scipy sparse matrix' hermitian transpose.

(read only)

CHAPTER

TWENTY

MATRIX SUM

class fastmat.**Sum**

Bases: *Matrix*

For matrices $A_k \in \mathbb{C}^{n \times m}$ with $k = 1, \dots, N$ we define a new mapping M as the sum

$$M = \sum_{k=1}^{N} A_k,$$

which then also is a mapping in $\mathbb{C}^{n \times m}$.

```
>>> # import the package
>>> import fastmat as fm
>>>
>>> # define the components
>>> A = fm.Circulant(x_A)
>>> B = fm.Circulant(x_B)
>>> C = fm.Fourier(n)
>>> D = fm.Diag(x_D)
>>>
>>> # construct the sum of transformations
>>> M = fm.Sum(A, B, C, D)
```

Assume we have two circulant matrices A and B, an N-dimensional Fourier matrix C and a diagonal matrix D. Then we define

$$M = A + B + C + D.$$

__init__()

Initialize a Sum matrix instance.

Parameters

***matrices**
[*fastmat.Matrix*] The matrix instances to be summed.

****options**
[optional] Additional optional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

CHAPTER

TWENTYONE

(MULTILEVEL) TOEPLITZ CLASS

class fastmat.**Toeplitz**

Bases: *Partial*

This class provides a very general implementation of Toeplitz matrices, which essentially realize a (possibly multidimensional) non-circular convolution.

This type of matrix is highly structured. A two-level Toeplitz Matrix looks like:

```
>>> t_00 t_05 t_04 t_03   t_40 t_45 t_44 t_43   t_30 t_35 t_34 t_33
>>> t_01 t_00 t_05 t_04   t_41 t_40 t_45 t_44   t_31 t_30 t_35 t_34
>>> t_02 t_01 t_00 t_05   t_42 t_41 t_40 t_45   t_32 t_31 t_30 t_35
>>>
>>> t_10 t_15 t_14 t_13   t_00 t_05 t_04 t_03   t_40 t_45 t_44 t_43
>>> t_11 t_10 t_15 t_14   t_01 t_00 t_05 t_04   t_41 t_40 t_45 t_44
>>> t_12 t_11 t_10 t_15   t_02 t_01 t_00 t_05   t_42 t_41 t_40 t_45
>>>
>>> t_20 t_25 t_24 t_23   t_10 t_15 t_14 t_13   t_00 t_05 t_04 t_03
>>> t_21 t_20 t_25 t_24   t_11 t_10 t_15 t_14   t_01 t_00 t_05 t_04
>>> t_22 t_21 t_20 t_25   t_12 t_11 t_10 t_15   t_02 t_01 t_00 t_05
```

This shows that one can define an L-level Toeplitz matrix by a tensor of order L together with means of deciding the sizes `n_1,...,n_L` of the individual levels.

__init__()

Initialize Toeplitz matrix instance.

One either has to specify `(vecC, vecR)` or `tenT` with optinal `split` argument.

Parameters

tenT
[`numpy.ndarray`] This is the most general way to define a (multilevel) Toeplitz Matrix. The number of dimensions (length of .shape) determines the number of levels. If *split* is not defined then tenT needs to have odd size in each dimension, so that this results in a square matrix. The handling of the indexing in direction of columns follows the same reversed fashion as in the one-dimensional case with *vecR*, but here naturally for each level.

split
[`numpy.ndarray`, optional] This vector needs to have as many elements as the number of elements of *tenT.shape*. If it is specified it defines the number of elements which are used to determine the number of rows of each level. The rest of the elements are indexed in reverse order in the same fashion as without split.

****options**

[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix* and *fastmat.Fourier*.

All optional arguments will be passed on to all *fastmat.Matrix* instances that are generated during initialization.

Note:

For backward compatibility reasons it is still possible to substitute the *tenT* argument by two 1D `numpy.ndarray` arrays *vecC* and *vecR* that describe the column- and row-defining vectors in the single-level case respectively. The column-defining vector describes the forst column of the resulting matrix and the row-defining vector the first row except the (0,0) element (which is already specified by the column-defining vector). Note that this vector is indexed backwards in the sense that its first element is the last element in the defined Toeplitz matrix.

`tenT`

Return the defining Tensor of Toeplitz matrix.

`vecC`

Return the column-defining vector of Toeplitz matrix.

`vecR`

Return the row-defining vector of Toeplitz matrix.

CHAPTER

TWENTYTWO

TRANSPOSITION AND RELATED CLASSES

class fastmat.**Transpose**

Bases: *Hermitian*

Transpose of a Matrix

__init__()

Initialize an instance of a transposed matrix.

Parameters

matrix
[*fastmat.Matrix*] The matrix instance to be transposed.

class fastmat.**Hermitian**

Bases: *Matrix*

Hermitian Transpose of a Matrix

__init__()

Initialize an instance of a hermitian transposed matrix.

Parameters

matrix
[*fastmat.Matrix*] The matrix instance to be transposed.

class fastmat.**Conjugate**

Bases: *Matrix*

Conjugate of a Matrix

__init__()

Initialize an instance of a conjugated matrix.

Parameters

matrix
[*fastmat.Matrix*] The matrix instance to be conjugated.

CHAPTER

TWENTYTHREE

ZERO MATRIX

class fastmat.**Zero**

Bases: *Matrix*

$$x \mapsto 0$$

The zero matrix only needs the dimension n of the vectors it acts on. It is very fast and very good!

```
>>> import fastmat as fm
>>>
>>> # define the parameter
>>> n = 10
>>>
>>> # construct the matrix
>>> O = fm.Zero(n)
```

__init__()

Initialize Zero matrix instance.

Parameters

numRows
[int] Height (row count) of the desired zero matrix.

numCols
[int] Width (column count) of the desired zero matrix.

****options**
[optional] Additional keyworded arguments. Supports all optional arguments supported by *fastmat.Matrix*.

Here we list the classes in the package for easy referencing and access.

- *fastmat.Matrix* base class, the mother of all matrices.
- *fastmat.BlockDiag*
- *fastmat.Blocks*
- *fastmat.Circulant*
- *fastmat.Conjugate*
- *fastmat.Diag*
- *fastmat.DiagBlocks*
- *fastmat.Eye*

- *fastmat.Fourier*
- *fastmat.Hadamard*
- *fastmat.Hermitian*
- *fastmat.Kron*
- *fastmat.LFSRCirculant*
- *fastmat.LowRank*
- *fastmat.Outer*
- *fastmat.Parametric*
- *fastmat.Partial*
- *fastmat.Permutation*
- *fastmat.Polynomial*
- *fastmat.Product*
- *fastmat.Sparse*
- *fastmat.Sum*
- *fastmat.Toeplitz*
- *fastmat.Transpose*
- *fastmat.Zero*

CHAPTER TWENTYFOUR

ALGORITHM INDEX

24.1 Algorithm Base Class

class fastmat.algorithms.**Algorithm**

Bases: `object`

Algorithm Base Class

The baseclass of all algorithms that operate on Matrices. This abstract baseclass introduces general framework concepts such as interfaces for parameter specification, algorithm execution, logging and callbacks.

```
>>> algI = fma.ISTA(Fourier(10))
>>> algI.cbResult = lambda i: print(i.arrResult)
>>> algI.cbStep = lambda i: print(i.numStep)
>>> algI.cbTrace = fma.Algorithm.snapshot
>>> algI.process(np.ones(10) + np.random.randn(10))
>>> plt.imshow(np.hstack((np.abs(tt.arrX) for tt in algI.trace)))
>>> plt.show()
```

__init__(**args*, ***kwargs*)

cbResult

cbTrace

handleCallback()

Call the callback if it is not None.

nbytes

process()

Process an array of data by the algorithm.

This method also accepts passing additional parameters as keyworded arguments. These arguments will be applied to the algorithm instance using self.updateParameters().

If no additional parameters are required the self._process() method may also be called directly for slightly higher call performance.

snapshot()

Add the current instances' state (without the trace) to the trace.

trace

updateParameters()

Update the parameters of the algorithm instance with the supllied keyworded arguments.

Apply the set of parameters specified in kwargs by iteratively passing them to setattr(self, …). Specifying an parameter which does not have a mathing attribute in the algorithm class will cause an AttributeError to be raised.

24.2 FISTA Algorithm

class fastmat.algorithms.**FISTA**(*fmatA*, ***kwargs*)

Bases: *Algorithm*

Fast Iterative Shrinking-Thresholding Algorithm (FISTA)

Definition and Interface: For a given matrix $A \in \mathbb{C}^{m \times N}$ with $m \ll N$ and a vector $b \in \mathbb{C}^m$ we approximately solve

$$\min_{x \in \mathbb{C}^N} \|A \cdot x - b\|_2^2 + \lambda \cdot \|x\|_1,$$

where $\lambda > 0$ is a regularization parameter to steer the trade-off between data fidelity and sparsity of the solution.

```
>>> # import the packages
>>> import numpy.linalg as npl
>>> import numpy as np
>>> import fastmat as fm
>>> import fastmat.algorithms as fma
>>> # define the dimensions and the sparsity
>>> n, k = 512, 3
>>> # define the sampling positions
>>> t = np.linspace(0, 20 * np.pi, n)
>>> # construct the convolution matrix
>>> c = np.cos(2 * t) * np.exp(-t ** 2)
>>> C = fm.Circulant(c)
>>> # create the ground truth
>>> x = np.zeros(n)
>>> x[np.random.choice(range(n), k, replace=0)] = 1
>>> b = C * x
>>> # reconstruct it
>>> fista = fma.FISTA(C, numLambda=0.005, numMaxSteps=100)
>>> y = fista.process(b)
>>> # test if they are close in the
>>> # domain of C
>>> print(npl.norm(C * y - b))
```

We solve a sparse deconvolution problem, where the atoms are harmonics windowed by a gaussian envelope. The ground truth x is build out of three pulses at arbitrary locations.

Note: The proper choice of λ is crucial for good perfomance of this algorithm, but this is not an easy task. Unfortunately we are not in the place here to give you a rule of thumb what to do, since it highly depends on the application at hand. Again, consult [1] for any further considerations of this matter.

Parameters

fmatA
[fm.Matrix] the system matrix

arrB
[np.ndarray] the measurement vector

numLambda
[float, optional] the thresholding parameter; default is 0.1

numMaxSteps
[int, optional] maximum number of steps; default is 100

Returns

np.ndarray
solution array

`__init__`(*fmatA*, ***kwargs*)

`softThreshold`(*arrX*, *numAlpha*)

Do a soft thresholding step.

24.3 ISTA Algorithm

class `fastmat.algorithms.`**`ISTA`**(*fmatA*, ***kwargs*)

Bases: *`Algorithm`*

Iterative Soft Thresholding Algorithm

Definition and Interface: For a given matrix $A \in \mathbb{C}^{m \times N}$ with $m \ll N$ and a vector $b \in \mathbb{C}^m$ we approximately solve

$$\min_{x \in \mathbb{C}^N} \|A \cdot x - b\|_2^2 + \lambda \cdot \|x\|_1,$$

where $\lambda > 0$ is a regularization parameter to steer the trade-off between data fidelity and sparsity of the solution.

```
>>> # import the packages
>>> import numpy.linalg as npl
>>> import numpy as np
>>> import fastmat as fm
>>> import fastmat.algorithms as fma
>>> # define the dimensions and the sparsity
>>> n, k = 512, 3
>>> # define the sampling positions
>>> t = np.linspace(0, 20 * np.pi, n)
>>> # construct the convolution matrix
>>> c = np.cos(2 * t) * np.exp(-t ** 2)
>>> C = fm.Circulant(c)
>>> # create the ground truth
>>> x = np.zeros(n)
>>> x[np.random.choice(range(n), k, replace=0)] = 1
>>> b = C * x
>>> # reconstruct it
>>> ista = fma.ISTA(C, numLambda=0.005, numMaxSteps=100)
>>> y = ista.process(b)
```

(continues on next page)

(continued from previous page)

```
>>> # test if they are close in the
>>> # domain of C
>>> print(npl.norm(C * y - b))
```

We solve a sparse deconvolution problem, where the atoms are harmonics windowed by a gaussian envelope. The ground truth x is build out of three pulses at arbitrary locations.

Note: The proper choice of λ is crucial for good perfomance of this algorithm, but this is not an easy task. Unfortunately we are not in the place here to give you a rule of thumb what to do, since it highly depends on the application at hand. Again, consult [1] for any further considerations of this matter.

Parameters

fmatA
[fm.Matrix] the system matrix

arrB
[np.ndarray] the measurement vector

numLambda
[float, optional] the thresholding parameter; default is 0.1

numMaxSteps
[int, optional] maximum number of steps; default is 100

Returns

np.ndarray
solution array

`__init__`(*fmatA*, ***kwargs*)

`softThreshold`(*arrX*, *numAlpha*)

Do a soft thresholding step.

24.4 OMP Algorithm

`class fastmat.algorithms.OMP`

Bases: `Algorithm`

Orthogonal Matching Pursuit

Definition and Interface: For a given matrix $A \in \mathbb{C}^{m \times N}$ with $m \ll N$ and a vector $b \in \mathbb{C}^m$ we approximately solve

$$\min_{x \in \mathbb{C}^N} \|x\|_0 \quad \text{s.t.} \quad A \cdot x = x.$$

If it holds that $b = A \cdot x_0$ for some k-sparse x_0 and k is low enough, we can recover x_0 via OMP [1].

This type of problem as the one described above occurs in Compressed Sensing and Sparse Signal Recovery, where signals are approximated by sparse representations.

```
>>> # import the packages
>>> import numpy.linalg as npl
>>> import numpy as np
>>> import fastmat as fm
>>> import fastmat.algorithms as fma
>>> # define the dimensions
>>> # and the sparsity
>>> n, k = 512, 3
>>> # define the sampling positions
>>> t = np.linspace(0, 20 * np.pi, n)
>>> # construct the convolution matrix
>>> c = np.cos(2 * t)
>>> C = fm.Circulant(c)
>>> # create the ground truth
>>> x = np.zeros(n)
>>> x[np.random.choice(range(n), k, replace=0)] = 1
>>> b = C * x
>>> # reconstruct it
>>> omp = fma.OMP(C, numMaxSteps=100)
>>> y = omp.process(b)
>>> # test if they are close in the
>>> # domain of C
>>> print(npl.norm(C * y - b))
```

We describe a sparse deconvolution problem, where the signal is in $\mathbb{R}^{512}$ and consists of 3 windowed cosine pulses of the form c with circulant displacement. Then we take the convolution and try to recover the location of the pulses using the OMP algorithm.

Note: The algorithm exploits two mathematical shortcuts. First it obviously uses the fast transform of the involved system matrix during the correlation step and second it uses a method to calculate the pseudo inverse after a rank-1 update of the matrix.

Todo:

- optimize einsum-stuff

Parameters

fmatA
[fm.Matrix] the system matrix

arrB
[np.ndarray] the measurement vector

numMaxSteps
[int] the desired sparsity order

Returns

np.ndarray
solution array

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

`arrA`

`arrB`

`arrC`

`arrResidual`

`arrSupport`

`arrX`

`arrXtmp`

`fmatA`

`fmatC`

`matPinv`

`newCols`

`newIndex`

`numL`

`numM`

`numMaxSteps`

`numN`

`numStep`

`v2`

`v2n`

`v2y`

24.5 STELA Algorithm

class `fastmat.algorithms.STELA`(*fmatA*, ***kwargs*)

Bases: `Algorithm`

Soft-Thresholding with simplified Exact Line search Algorithm (STELA)

The algorithm is presented in [1] with derivation and convergence results.

Definition and Interface: For a given matrix $A \in \mathbb{C}^{m \times N}$ with $m \ll N$ and a vector $b \in \mathbb{C}^m$ we approximately solve

$$\min_{x \in \mathbb{C}^N} \|A \cdot x - b\|_2^2 + \lambda \cdot \|x\|_1,$$

where $\lambda > 0$ is a regularization parameter to steer the trade-off between data fidelity and sparsity of the solution.

```
>>> # import the packages
>>> import numpy.linalg as npl
>>> import numpy as np
>>> import fastmat as fm
>>> import fastmat.algorithms as fma
>>> # define the dimensions and the sparsity
>>> n, k = 512, 3
>>> # define the sampling positions
>>> t = np.linspace(0, 20 * np.pi, n)
>>> # construct the convolution matrix
>>> c = np.cos(2 * t) * np.exp(-t ** 2)
>>> C = fm.Circulant(c)
>>> # create the ground truth
>>> x = np.zeros(n)
>>> x[np.random.choice(range(n), k, replace=0)] = 1
>>> b = C * x
>>> # reconstruct it
>>> stela = fma.STELA(C, numLambda=0.005, numMaxSteps=100)
>>> y = stela.process(b)
>>> # test if they are close in the
>>> # domain of C
>>> print(npl.norm(C * y - b))
```

We solve a sparse deconvolution problem, where the atoms are harmonics windowed by a gaussian envelope. The ground truth x is build out of three pulses at arbitrary locations.

Note: The proper choice of λ is crucial for good perfomance of this algorithm, but this is not an easy task. Unfortunately we are not in the place here to give you a rule of thumb what to do, since it highly depends on the application at hand. Again, consult [1] for any further considerations of this matter.

Parameters

fmatA
[fm.Matrix] the system matrix

arrB
[np.ndarray] the measurement vector

numLambda
[float, optional] the thresholding parameter; default is 0.1

numMaxSteps
[int, optional] maximum number of steps; default is 100

numMaxError
[float, optional] maximum error tolerance; default is 1e-6

Returns

np.ndarray
solution array

`__init__`(*fmatA*, ***kwargs*)

`softThreshold`(*arrX*, *numAlpha*)

Do a soft thresholding step.

Here we list the specialized algorithms in the package for easy referencing and access.

- *fastmat.algorithms.Algorithm* base class, the mother of all algorithms.
- *fastmat.algorithms.FISTA*
- *fastmat.algorithms.ISTA*
- *fastmat.algorithms.OMP*
- *fastmat.algorithms.STELA*

CHAPTER

TWENTYFIVE

ARCHITECTURE

25.1 Matrix Class Model

25.2 Algorithm Class Model

25.3 SciPy Interface

25.3.1 Motivation

SciPy offers a large zoo of algorithms which exploit the possibility to pass a so called LinearOperator, which only provides methods for forward and backward transforms together with some simple properties like a datatype and shape parameters. This is exactly what we can provide for a specific instance of a fastmat Matrix. To this end, each fastmat Matrix has the (read only) property scipyLinearOperator, which provides a SciPy Linear operator realizing the transform specified by the fastmat object.

This allows to combine fastmat and SciPy in the most efficient manner possible. Here, fastmat provides the simple and efficient description of a huge variety of linear operators, which can then be used neatly and trouble free in SciPy.

25.3.2 Usage

The interface is provided by the factory *`fastmat.Matrix.scipyLinearOperator()`*, which returns an instance of `scipy.sparse.linalg.LinearOperator`. All fastmat matrices can be used from *scipy* methods supporting this interface, offering a wide range of functionality – by combining both worlds – to the user.

For a code example, see *Solve a System of Linear Equations with Preconditioning*.

25.4 Data Types in fastmat

To achieve high performance, fastmat is designed to support common data types only, namely

- Floating point with single and double precision (*float32* and *float64*)
- Complex floating point with single and double precision (*complex64* and *complex128*)
- Signed integer of selected fixed sizes (*int8*, *int16*, *int32*, *int64*)

Some implementation of fastmat matrices use numpy under the hood. Although those could technically be able to deal with other data types offered by `numpy` as well, using other types than those listed above is disencouraged. This is important to ensure consistency throughout the intestines of `fastmat`, which is important for being able to reliably test the package.

The following sections detail the organization and handling of types in `fastmat` and explain the mechanisms how fastmat handles type promotion. The final section references the internal type API of `fastmat`.

25.4.1 Type handling

ftype

To distinguish between the supported data types `fastmat` uses the `ftype` internally as type identifier. All data type checks within the package as well as the type promotion logic is internally handled via these type numbers, that correspond directly to the associated `numpy.dtype` given in this table:

Data Type	`numpy.dtype`	fast-mat `ftype`	Short
Signed Integer 8 bit Signed Integer 16 bit Signed Integer 32 bit Signed Integer 64 bit Single-precision Float Double-Precision Float Single-Precision Complex Double-Precision Complex	`int8_t int16_t int32_t int64_t float32_t float64_t complex64_t complex128_t`	0 1 2 3 4 5 6 7	i8 i16 i32 i64 f32 f64 c64 c128

25.4.2 Type promotion

Type promotion matrix of binary operators of kind `f(A, B)` as used throughout `fastmat`:

Type promotion		B							
		int				float		complex	
		i8	i16	i32	i64	f32	f64	c64	c128
A	int 8	i8	i16	i32	i64	f32	f64	c64	c128
	int 16	i16	i16	i32	i64	f32	f64	c64	c128
	int 32	i32	i32	i32	i64	f64	f64	c128	c128
	int 64	i64	i64	i64	i64	f64	f64	c128	c128
	float 32	f32	f32	f64	f64	f64	f64	c128	c128
	float 64	f64	f64	f64	f64	f64	f64	c128	c128
	complex 64	c64	c64	c128	c128	c64	c128	c64	c128
	complex 128	c128	c128	c128	c128	c128	c128	c128	c128

Example:

The forward operator of a *`fastmat.Matrix`* of type *float 32* will, if provided with an *int 32* input vector, yield an output vector of type *float 64*.

Note: The output data type will be expanded to fit the mantissa of any of the operands best. As *int 32* has a wider mantissa than *float 32* offers, the example type promotion will yield *float 64* to maintain accuracy.

Note: Data types will not be expanded automatically to the next larger data type for the sake of preventing overflows. You'll need to specifically expand the data type – where necessary – by specifying `minType=?` during the generation of your *`fastmat.Matrix`* instance.

25.4.3 `fastmat.core.types`

`fastmat.core.types.`**`getFusedType`**`()`

Return fastmat type number for a given data type (or array).

Parameters

obj
[object] The object type or *numpy.ndarray* to query for.

Returns

ftype
The fastmat type number for that type.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`getNumpyType`**`()`

Return numpy type number for a given data type (or array).

Parameters

obj
[object] The object type or *numpy.ndarray* to query for.

Returns

ntype
The numpy type number for that type.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`getTypeEps`**`()`

Return eps for a given data type (or array).

Parameters

obj
[object] The object type or *numpy.ndarray* to query for.

Returns

np.float64_t
The epsilon value for that type.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`getTypeMax`**`()`

Return the maximum representable value for a given data type (or array).

Parameters

obj
[object] The object type or *numpy.ndarray* to query for.

Returns

np.float64_t
The maximum representable value for that type.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`getTypeMin`**`()`

Return the minimum representable value for a given data type (or array).

Parameters

obj
[object] The object type or *numpy.ndarray* to query for.

Returns

np.float64_t
The minimum representable value for that type.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`isComplex`**`()`

Return whether a given data type or an array's data type is complex.

Parameters

obj
[object] The object type to query for.

Returns

bool
True if the data type is of complex kind.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`isFloat`**`()`

Return whether a given data type or an array's data type is floating point.

Parameters

obj
[object] The object type to query for.

Returns

type
True if the data type is of floating point kind.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`isInteger`**`()`

Return whether a given data type or an array's data type is integer.

Parameters

obj
[object] The object type to query for.

Returns

type
True if the data type is of integer kind.

Raises

TypeError
When the type is not supported by fastmat.

`fastmat.core.types.`**`safeTypeExpansion`**`()`

Return a floating type expanding the given type with full accuracy.

Parameters

dtype
[object] A type object to be expanded to float without numerical accuracy loss.

Returns

object
The safely expanded datatype

25.5 Performance Interface to numpy C-API

25.5.1 `fastmat.core.cmath`

`fastmat.core.cmath.`**`profileCall`**`()`

Measure the runtime of a function call with arguments by averaging the cumulated runtime of multiple calls.

To avoid unpacking arguments each time the function is called calls with one or two arguments get unpacked before the measurement, thus excluding argument unpacking in this case effectively.

Parameters

reps
[int] The number of repetitions of `call()` in one runtime measurement.

call
[callable] The function to be called

args
[iterable] The positional arguments to be passed to call

Returns

dict

The dictionary contains the following key:value pairs: avg
[float] The average runtime of a single call to `call(*args)`

time
[float] The accumulated runtime of *reps* calls to `call(*args)`

cnt
[int] The total count of calls to `call(*args)`

25.6 Low-Overhead Array Striding Interface

Fastmat offers a special C-level interface allowing the creation, processing and manipulation of views into the underlying data of `numpy.ndarray` objects without the overhead of creating view or memoryview objects of that array object. As the implementation is based on C structures, no interaction with the python object interface is necessary, thus increasing the efficiency of advanced linear operators from within cython code. By mimimizing memory operations occuring during view creation, structure- or object allocation or copying, this helps minimizing the already low overhead on using cython memoryviews further.

The main container for defining and using strides is the `STRIDE_s` structure:

```
ctypedef struct STRIDE_s:
    char *          base
    intsize         strideElement
    intsize         strideVector
    intsize         numElements
    intsize         numVectors
    np.uint8_t      sizeItem
    ftype           dtype
```

fastmat Type Identifier

The striding interface supports:

- Two-dimensional `numpy.ndarray` objects
- Non-contiguous (striding) access into the data
- Modifying views (substriding

25.6.1 `fastmat.core.strides`

25.7 Calibration and Runtime Optimization

25.7.1 `fastmat.core.calibration`

`fastmat.core.calibration.`**`calibrateAll`**`(**`*options*`)`

Calibrate all classes present in fastmat.

Parameters

****options**
[dict] Additional keyworded arguments that will be passed on to `calibrateClass()` calls. Note: The *verbose* option will be digested by this function and not passed on to `calibrateClass()`.

Returns

None

`fastmat.core.calibration.`**`calibrateClass`**`(`*target*, `**`*options*`)`

Calibrate a fastmat matrix baseclass using the specified benchmark.

The generated calibration data will be cached in *calData* and is then available during instantiation of upcoming fastmat classes and can be imported/exported to disk using the routines *loadCalibration* and *saveCalibration*.

Parameters

target
[`Matrix`] The Matrix class to be calibrated. Any existing calibration data will be overwritten when the calibration succeeded.

benchmarkOnly
[bool, optional] If true, only perform the benchmark evaluation and do not generate calibration data (or update the corresponding entries in *calData*).

Defaults to False.

verbose
[bool, optional] Controls the *BENCH.verbosity* flag of the `BENCH` instance, resulting in increased verbosity during the test.

Defaults to False.

maxIter
[float, optional] Additional benchmark option that will be passed on to the evaluation. Abort iteration if evaluation of one problem takes more than this amount of seconds.

Defaults to 0.1.

maxInit
[float, optional] Additional benchmark option that will be passed on to the evaluation. Abort iteration if preparation of one problem takes more than this amount of seconds.

Defaults to 0.1.

maxSize
[float, optional] Additional benchmark option that will be passed on to the evaluation. Abort iteration if this problem size is exceeded.

Defaults to 1000000 (one million).

maxMem
[float, optional] Additional benchmark option that will be passed on to the evaluation. Abort iteration if memory usage exceeds this amount of kiB.

Defaults to 100000 (100 MB).

minItems
[int, optional] Additional benchmark option that will be passed on to the evaluation. Require the evaluation of at least this number of different problem sizes.

Defaults to 3.

measMinTime
[float, optional] Additional benchmark option that will be passed on to the evaluation. Require the measurement interval to be at least this amount of seconds. Increase repetition count of the evaluation of one problem size is faster than that.

Defaults to 0.003.

meas_minReps
[int, optional] Additional benchmark option that will be passed on to the evaluation. Require at least this number of repetitions to be performed in one measurement interval.

Defaults to 3.

meas_minReps
[int, optional] Additional benchmark option that will be passed on to the evaluation. Require at least this number of independent measurements for one evaluation.

Defaults to 3.

funcStep
[int callable(int)] Additional benchmark option that will be passed on to the evaluation. Provision to increase problem size after each evaluation as lamba function returning the next problem size, based on the current.

Defaults to *lambda x: x + 1*.

****options**
[optional] Additional benchmark options that will be passed on to the evaluation.

Returns

tuple (`MatrixCalibration`, `BENCH`)
If the option *benchmarkOnly* is True, return the generated calibration data and the benchmark instance (containing all benchmark data collected) as a tuple

`BENCH`
If the option *benchmarkOnly* is False, return the benchmark instance.

`fastmat.core.calibration.`**`getMatrixCalibration`**`(`*target*`)`

Return a `MatrixCalibration` object with the calibration data for the fastmat baseclass target was instantiated from.

Parameters

target
[`Matrix`] The fastmat Matrix class for which a `MatrixCalibration` object shall be returned.

Returns

`MatrixCalibration`
If no calibration data exists, *None* will be returned.

`fastmat.core.calibration.`**`loadCalibration`**`(`*filename*`)`

Short summary.

Parameters

filename
[type] Description of parameter *filename*.

Returns

type
Description of returned object.

`fastmat.core.calibration.`**`saveCalibration`**`(`*filename*`)`

Save package calibration data in JSON format to file.

The top level is a dictionary containing calibration data for each class, as a `MatrixCalibration` object, and identified by the class object's basename as string. The `MatrixCalibration` object – being a `dict` itself – will be represented transparently by JSON.

Parameters

filename
[str] Filename to write the configuration data to.

Returns

None

CHAPTER

TWENTYSIX

USER DEFINED CLASSES

26.1 Developing Your own fastmat Matrix

To be delivered (somewhere in time).

26.2 Optimizing fastmat Class Implementations

To be delivered (somewhere in time).

In this section we will show what needs to be done in order to *implement* a new fastmat class and detail on how to *test* it properly once it is implemented using the built-in class test system. Further, we show how to evaluate the performance of your implementation by using the also-built-in *benchmarking* system and give examples on how to *optimize* a given implementation to improve its performance.

CHAPTER

TWENTYSEVEN

TESTING AND BENCHMARKING

27.1 Benchmarking fastmat Classes

To be delivered (somewhere in time).

27.2 Testing fastmat Classes and Unit Tests

27.2.1 Fastmat type identifiers

27.2.2 `fastmat.inspect.test`

class `fastmat.inspect.test.`**`TEST`**

Bases: `NAME`

`ALG = 'algorithm'`

`ALGORITHM = 'algorithm'`

`ALG_ARGS = 'algorithmArgs'`

`ALG_KWARGS = 'algorithmKwargs'`

`ALG_MATRIX = 'algorithmMatrix'`

`ARGS = 'testArgs'`

`CHECK_DATATYPE = 'checkDataType'`

`CHECK_PROXIMITY = 'checkProximity'`

`CLASS = 'class'`

`DATAALIGN = 'dataAlign'`

`DATAARRAY = 'arrData'`

`DATACENTER = 'dataDistCenter'`

`DATACOLS = 'numVectors'`

`DATAGEN = 'dataGenerator'`

`DATASHAPE = 'dataShape'`

`DATASHAPE_T = 'dataShapeBackward'`

`DATATYPE = 'dataType'`

`IGNORE = 'ignore'`

`INIT = 'init'`

`INITARGS = 'args'`

`INITKWARGS = 'kwargs'`

`INIT_VARIANT = 'initVariant'`

`INSTANCE = 'instance'`

`KWARGS = 'testKwargs'`

`NAMING = 'naming'`

`NAMINGARGS = 'namingArgs'`

`NUM_COLS = 'numCols'`

`NUM_ROWS = 'numRows'`

`OBJECT = 'object'`

`PARAMALIGN = 'alignment'`

`QUERY = 'query'`

`REFALG = 'refAlgorithm'`

`REFALG_ARGS = 'refAlgorithmArgs'`

`REFALG_KWARGS = 'refAlgorithmKwargs'`

`REFERENCE = 'reference'`

`RESULT_IGNORED = 'testResultIgnored'`

`RESULT_INFO = 'testInfo'`

`RESULT_INPUT = 'testInput'`

`RESULT_OUTPUT = 'testOutput'`

`RESULT_PROX = 'testResultProximity'`

`RESULT_REF = 'testReference'`

`RESULT_TOLERR = 'testTolError'`

`RESULT_TYPE = 'testResultType'`

`SKIP = 'skip'`

`TOL_MINEPS = 'tolMinEps'`

`TOL_POWER = 'tolPower'`

`TRANSFORMS = 'transform'`

`TYPE_EXPECTED = 'typeExpected'`

`TYPE_PROMOTION = 'typePromotion'`

`class fastmat.inspect.test.Test`(*targetClass*, ***options*)

Bases: `Worker`

Short summary.

Parameters

targetClass
[type] Description of parameter *targetClass*.

****options**
[type] Description of parameter ***options*.

Attributes

_verboseFull
[type] Description of attribute *_verboseFull*.

`__init__`(*targetClass*, ***options*)

Short summary.

Parameters

targetClass
[type] Description of parameter *targetClass*.

****options**
[type] Description of parameter ***options*.

Returns

type
Description of returned object.

`findProblems`(*nameTarget*, *targetResult*)

Short summary.

Parameters

nameTarget
[type] Description of parameter *nameTarget*.

targetResult
[type] Description of parameter *targetResult*.

Returns

type
Description of returned object.

`printStatus`(*nameTest*, *resultTest*, *lenName=-1*, *descrVariants=''*)

Short summary.

Parameters

nameTest
[type] Description of parameter *nameTest*.

resultTest
[type] Description of parameter *resultTest*.

lenName
[type] Description of parameter *lenName*.

descrVariants
[type] Description of parameter *descrVariants*.

Returns

type
Description of returned object.

`property verbosity`

`fastmat.inspect.test.`**`compareResults`**`(`*test*, *query*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

query
[type] Description of parameter *query*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`formatResult`**`(`*result*`)`

Short summary.

Parameters

result
[type] Description of parameter *result*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`initTest`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testAlgorithm`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testArray`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testArrays`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testBackward`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testColNorms`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testColNormsColNormalized`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testConjugate`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testFailDump`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testForward`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testGetColsMultiple`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testGetColsSingle`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testGetItem`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testGetRowsMultiple`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testGetRowsSingle`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testGram`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testHermitian`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testInterface`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testLargestSV`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testRowNormalized`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testRowNorms`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`testTranspose`**`(`*test*`)`

Short summary.

Parameters

test
[type] Description of parameter *test*.

Returns

type
Description of returned object.

`fastmat.inspect.test.`**`tryQuery`**`(`*nameTest*, *query*, *argument*`)`

In this section we will show two built-in systems that allow you to *test* and *benchmarking* any fastmat Matrix implementation. Testing is important to verify that a certain implementation actually does what you'd expect of it and is virtually *the* essential cornerstone to writing your own user defined matrix class implementation.

The scope of the implemented testing system expands to these areas (not complete):

- Testing of mathematical correctness
- Testing of computational accuracy
- Testing of various data types (and combinations thereof)
- Testing of parameter combinations
- Testing for different platforms ans versions (mostly done by our CI setup)

Benchmarking, however, is a valuable goodie to have in your toolbox to evaluate the performance of your implementation and to find culprits that impact the runtime or memory consumption of your implementation. Make sure to also tune in to out *optimization* section, where we detail on how to use the information gained from benchmarking your classes productively to improve your implementation.

CHAPTER

TWENTYEIGHT

EXAMPLES

28.1 Compressed Sensing example

We set up a linear forward model using a Fourier matrix as dictionary and reconstruct the underlying sparse vector from linear projections using a matrix with elements drawn randomly from a Gaussian distribution.

```
import numpy as np
import matplotlib.pyplot as plt

import fastmat as fm
import fastmat.algorithms as fma

# Problem dimensions
compression_factor = 4
N = 60  # Problem size
M = int(N / compression_factor)  # Number of observations
k = 3  # sparsity level
noise_power_db = -10.

# Ground truth of scenario
# We choose the ground_truth to be two dimensional here to align all vectors
# explicitly vertical, allowing easy stacking later on
ground_truth_positions = np.random.choice(N, k)
ground_truth_weights = np.random.randn(k, 1)
ground_truth = np.zeros((N, 1))
ground_truth[ground_truth_positions] = ground_truth_weights


# Set up the linear signal model and reconstruction method,
# consisting of Measurement Matrix `Phi` and Signal Base `Dict`
Phi = fm.Matrix(np.random.randn(M, N))
Dict = fm.Fourier(N)

A = Phi * Dict


# Now determine the actual (real-world) signal and its observation
# according to the specified Measurement matrix and plot the signals
# also allow for noise
```

(continues on next page)

(continued from previous page)

```
def add_noise(signal, pwr_db):
    return signal + 10**(pwr_db / 10.) * np.linalg.norm(signal) * (
        np.random.randn(*signal.shape) / np.sqrt(signal.size)
    )


x_clean = Dict * ground_truth
x = add_noise(x_clean, noise_power_db)
b = Phi * x

# Now reconstruct the original ground truth using
# * Orthogonal Matching Pursuit (OMP)
# * Fast Iterative Shrinkage Thresholding Algorithm (FISTA)
# * Soft-Thresholding with simplified Exact Line search Algorithm (STELA)
numLambda = 5
numSteps = 600
alg_omp = fma.OMP(A, numMaxSteps=k)
alg_fista = fma.FISTA(A, numMaxSteps=numSteps, numLambda=numLambda)
alg_stela = fma.STELA(A, numMaxSteps=numSteps, numLambda=numLambda)
y_omp = alg_omp.process(b)
y_fista = alg_fista.process(b)
y_stela = alg_stela.process(b)

# Setup a simple phase transition diagram for OMP, for a number of randomly
# chosen measurement matrices and another number of noise realizations for
# each measurement matrix.
trials = 15
M_phase_transition = np.arange(k, N)
true_support = (ground_truth == 0)
success_rate = np.zeros(len(M_phase_transition))
for index, m_phase_transition in enumerate(M_phase_transition):
    for _ in range(trials):
        # randomly choose a new measurement matrix
        Phi_pt = fm.Matrix(np.random.randn(m_phase_transition, N))
        alg_omp = fma.OMP(Phi_pt * Dict, numMaxSteps=k)

        # randomly choose `trials` different noise realizations
        x_pt = add_noise(np.tile(x_clean, (1, trials)), noise_power_db)
        b_pt = Phi_pt * x_pt

        # and process recovery all in one flush
        recovered_support = alg_omp.process(b_pt)

        # now determine the success of our recovery and update the success rate
        success = (recovered_support == 0.) == true_support
        success_rate[index] += np.mean(np.all(success, axis=0))

    print(success_rate[index])

# finally, normalize the success_rate to the amount of trials performed
success_rate = success_rate / trials
```

(continues on next page)

(continued from previous page)

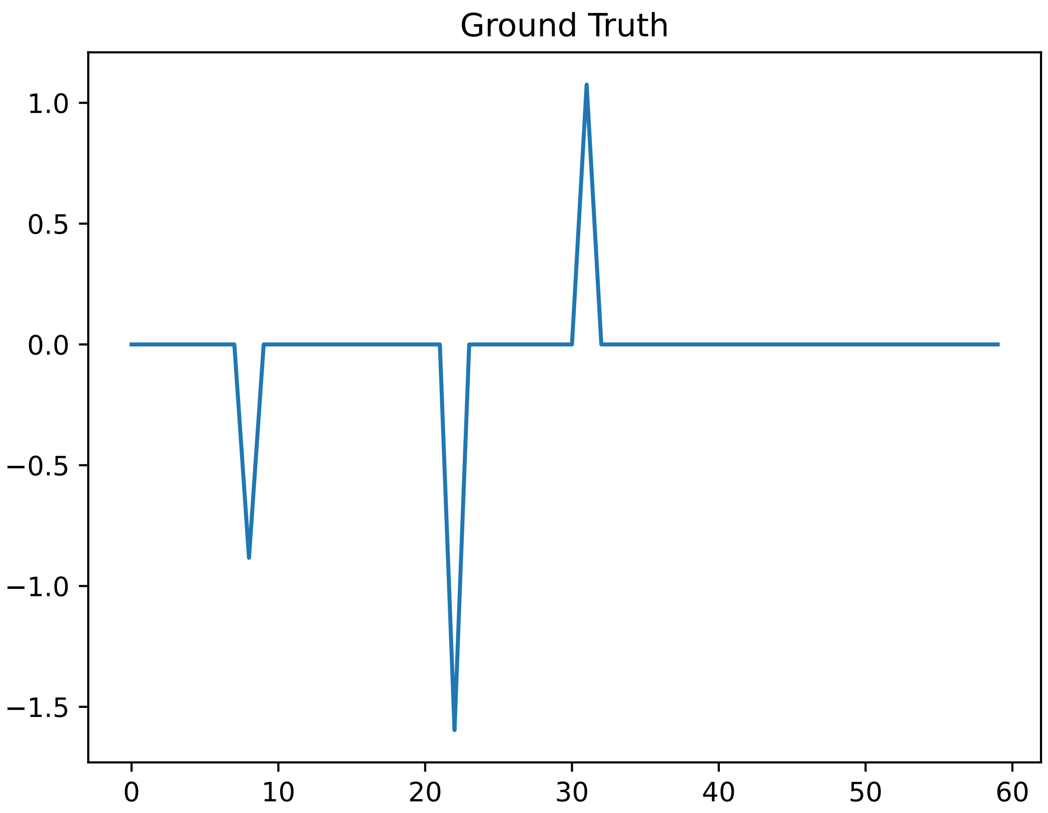
```

# Plot all results
plt.figure(1)
plt.clf()
plt.title('Ground Truth')
plt.plot(ground_truth)

plt.figure(2)
plt.clf()
plt.title('Actual Signal')
plt.plot(x_clean, label='Actual signal')
plt.plot(x, label='Actual signal with noise')
plt.legend()

plt.figure(3)
plt.clf()
plt.title('Observed Signal')
plt.plot(b)

plt.figure(4)
plt.clf()
plt.title("Reconstruction from M = " + str(M) + " measurements.")
plt.stem(ground_truth_positions, ground_truth_weights, label='Ground Truth')
plt.plot(y_omp, label='Reconstruction from OMP')
plt.plot(y_fista, label='Reconstruction from FISTA')
plt.plot(y_stela, label='Reconstruction from STELA')
plt.legend()
#
plt.figure(5)
plt.clf()
plt.title("Phase transition for sparsity k = " + str(k))
plt.plot(1. * M_phase_transition / N, success_rate, label='Sucess rate of OMP')
plt.xlabel('compression ratio M/N')
plt.ylabel('Sucess rate')
plt.legend()
plt.show()
```

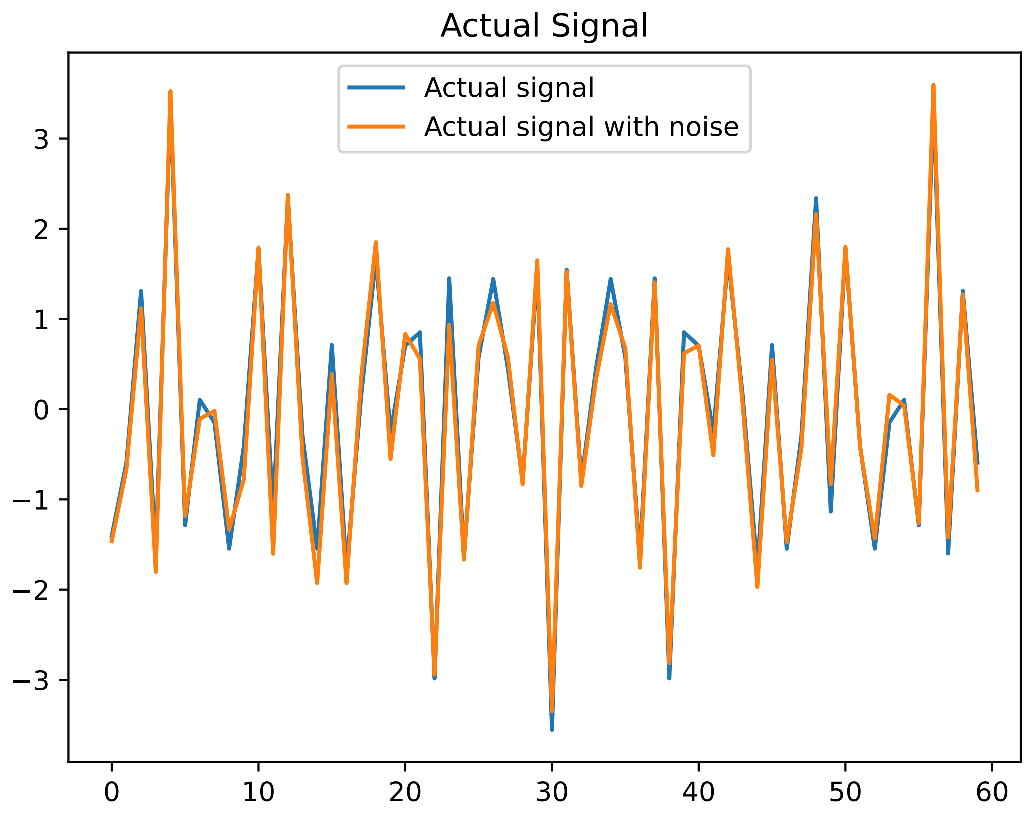
28.2 Solve a System of Linear Equations with Preconditioning

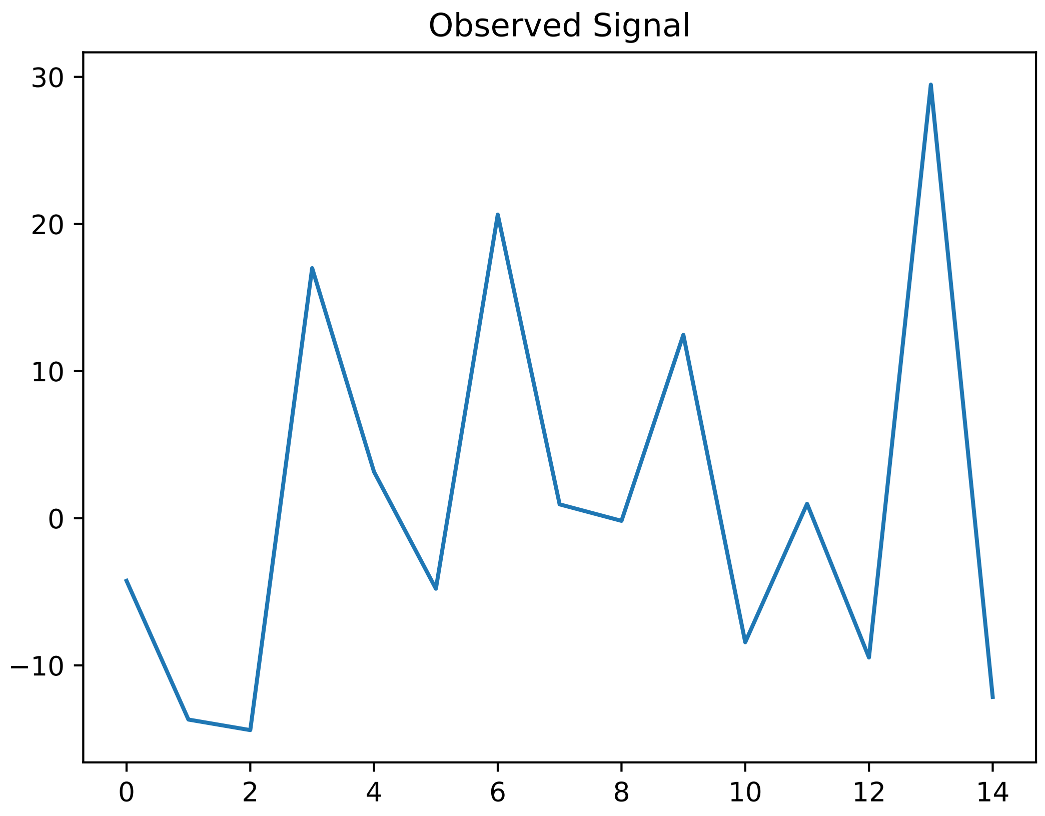
The preconditioner used for solving can also be provided as a `LinearOperator`.

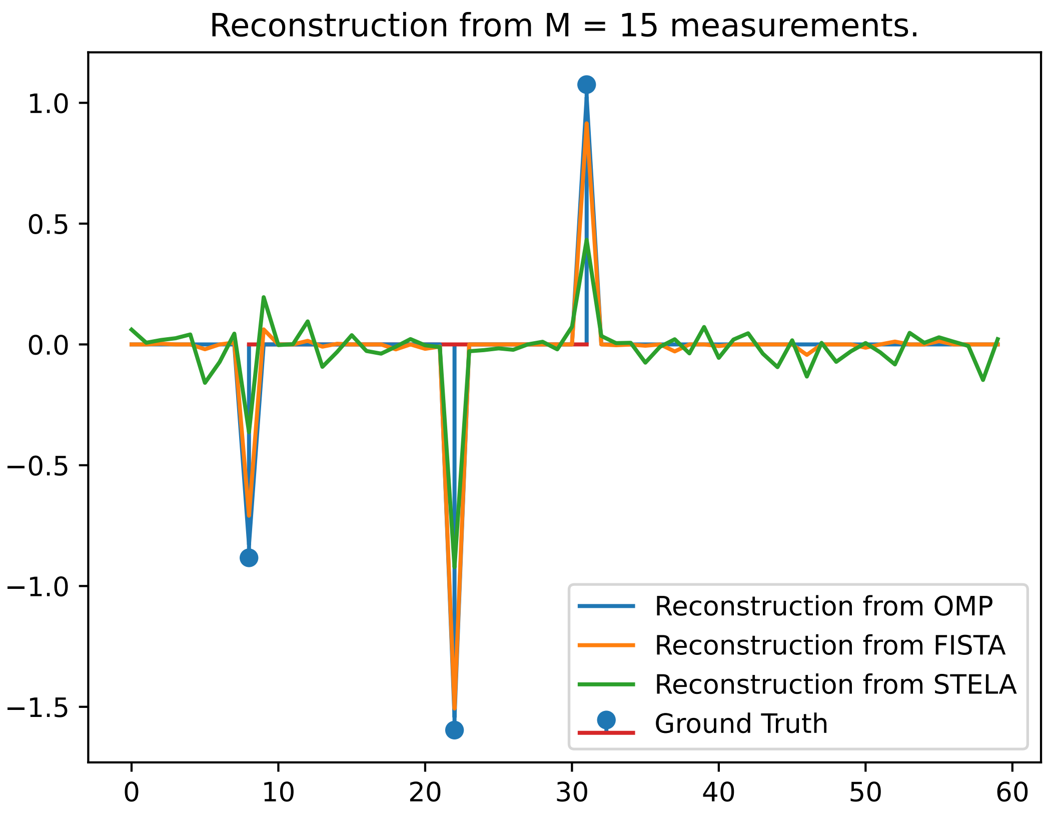
```
import fastmat as fm
import numpy as np
from scipy.sparse.linalg import cgs
# diagonal matrix with no zeros
d = np.random.uniform(1, 20, 2 ** 10)

# fastmat object
H = fm.Diag(d)

```

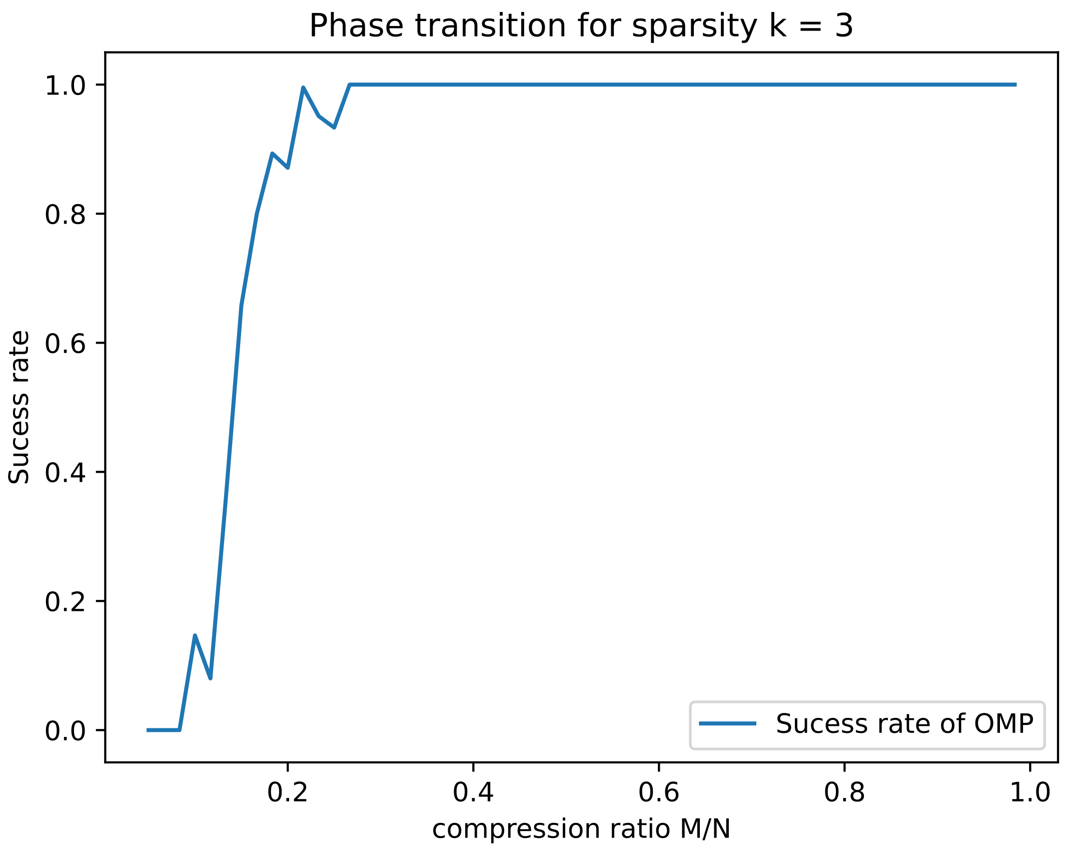
(continues on next page)


Ground Truth
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
10
20
30
40
50
60


Actual Signal
Actual signal
Actual signal with noise
3
2
1
0
−1
−2
−3
0
10
20
30
40
50
60


Observed Signal
30
20
10
0
−10
0
2
4
6
8
10
12
14


Reconstruction from M = 15 measurements.
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
10
20
30
40
50
60
Reconstruction from OMP
Reconstruction from FISTA
Reconstruction from STELA
Ground Truth


Phase transition for sparsity k = 3
Sucess rate
compression ratio M/N
Sucess rate of OMP
0.0
0.2
0.4
0.6
0.8
1.0
0.2
0.4
0.6
0.8
1.0

(continued from previous page)

```
# use the new property to generate a scipy linear operator
Hs = H.scipyLinearOperator

# also generate a Preconditioning linear operator,
# which in this case is the exact inverse
Ms = fm.Diag(1.0 / d).scipyLinearOperator

# get a baseline
x = np.random.uniform(1, 20, 2 ** 10)
y = np.linalg.solve(H.array, x)
cgs(Hs, x, tol=1e-10)
cgs(Hs, x, tol=1e-10, M=Ms)
```

In this section we will put some examples on the usage of fastmat (later on).

CHAPTER

TWENTYNINE

REFERENCES

12345

[1] Stefan van der Walt, S. Chris Colbert and Gael Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computations”, Computing in Science and Engineering, Volume 13, 2011.

[2] Rao K. Yarlagadda, John E. Hershey’, “Hadamard Matrix Analysis and Synthesis, With Applications to Communications and Signal/Image Processing”, The Springer International Series in Engineering and Computer Science, Volume 383, 1997

[3] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith, “Cython: The Best of Both Worlds”, Computing in Science and Engineering, Volume 13,2011

[4] Fernandes, Paulo and Plateau, Brigitte and Stewart, William J., “Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks”, Journal of the ACM, New York, Volume 45, 1998

[5] Simon W. Golomb, “Shift Register Sequences”, Holden-Day, Inc. 1967

CHAPTER

THIRTY

RELEASES

30.1 Rolling Stable Branch

- Github Link

30.2 Version 0.2

- Download Link

30.3 Version 0.1.2

- Download Link

30.4 Version 0.1.1

- Download Link
- Documentation Link

30.5 Version 0.1

- Download Link
- Documentation Link

CHAPTER

THIRTYONE

INTRODUCTION

In many fields of engineering linear transforms play a key role during modeling and solving real world problems. Often these linear transforms have an inherent structure which reduces the degrees of freedom in their parametrization. Moreover, this structure allows to describe the action of a linear mapping on a given vector more efficiently than the general one.

This structure can be exploited twofold. First, the storage of these transforms in form of matrices, on computers normally an array of numbers in $\mathbb{C}$ or $\mathbb{R}$, might be unnecessary. So for each structure there is a more concise way of representation, which leads to a benefit in memory consumption when using these linear transforms. Second, the structure allows more efficient calculations when applying the linear transform to a vector. This may result in a drop in algorithmic complexity which implies that computing time can be saved.

Still, these structural benefits have to be exploited and it is not often easy to accomplish this in a save and reuseable way. Moreover, in applications you often think of the linear transforms as a matrix and your way of working with it is streamlined to this way of thinking, which is only natural, but does not directly allow to exploit the structure.

So, there are different ways of thinking in what is natural and in what is efficient. This is the gap fastmat tries to bridge by allowing you to work with the provided objects as if they were common matrices represented as arrays of numbers, while the algorithms that make up the internals are highly adapted to the specific structure at hand. It provides you with a set of tools to work with linear transforms while hiding the algorithmic complexity and exposing the benefits in memory and calculation efficiency without too much overhead.

This way you can worry about really urgent matters to you, like research and development of algorithms and leave the internals to fastmat.

CHAPTER THIRTYTWO

PUBLICATIONS

Since we created a package for scientific computing, it makes sense to use it for science. Below we list all publications, which make use of our package with varying degree. If made use of fastmat in your publication, we are happy to reference it here:

- The White Paper: Fast Linear Transforms in Python
- Defect Detection from 3D Ultrasonic Measurements Using Matrix-free Sparse Recovery Algorithms
- GPU-accelerated Matrix-Free 3D Ultrasound Reconstruction for Nondestructive Testing

If you use fastmat in your own work we kindly ask you to cite the above mentioned white paper as an acknowledgement.

CHAPTER THIRTYTHREE

PUBLIC APPEARANCES

Sometimes we also get out in the wild and present the package. The talks we held can be found below.

- EuroScipy 2017 Erlangen: PDF, Youtube

CHAPTER THIRTYFOUR

CONTRIBUTIONS

There are many ways you as an individual can contribute. We are happy about feature requests, bug reports and of course contributions in form of additional features. To these ends, please step by at Github where we organize the work on the package.

CHAPTER

THIRTYFIVE

AFFILIATIONS AND CREDITS

Currently the project is jointly maintained by Sebastian Semper and Christoph Wagner at the EMS group at TU Ilmenau.

BIBLIOGRAPHY

[1] Amir Beck, Marc Teboulle, “A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems”, SIAM Journal on Imaging Sciences, 2009, Vol. 2, No. 1 : pp. 183-202

[1] Amir Beck, Marc Teboulle, “A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems”, SIAM Journal on Imaging Sciences, 2009, Vol. 2, No. 1 : pp. 183-202

[1] S. G. Mallat, Z. Zhang, “Matching pursuits with time-frequency dictionaries”, IEEE Transactions on Signal Processing, vol. 41, no. 12, pp. 3397-3415, Dec 1993

[1] Y. Yang, M. Pesavento, “A Unified Successive Pseudoconvex Approximation Framework”, IEEE Transactions on Signal Processing, vol. 65, no. 13, pp. 3313-3327, Dec 2017

PYTHON MODULE INDEX

INDEX