Magni Documentation

Version 1.0.0

Christian Schou Oxvig and Patrick Steffen Pedersen
in collaboration with
Jan Østergaard, Thomas Arildsen,
Tobias L. Jensen, and Torben Larsen

May 23, 2014

Magni 1.0.0 documentation »

Magni Documentation

magni is a Python package developed by Christian Schou Oxvig and Patrick Steffen Pedersen in collaboration with Jan Østergaard, Thomas Arildsen, Tobias L. Jensen, and Torben Larsen. The work was supported by 1) the Danish Council for Independent Research | Technology and Production Sciences - via grant DFF-1335-00278 for the project Enabling Fast Image Acquisition for Atomic Force Microscopy using Compressed Sensing and 2) the Danish e-Infrastructure Cooperation - via a grant for a high performance computing system for the project "High Performance Computing SMP Server for Signal Processing".

This page gives an *Introduction* to the package, briefly describes *How to Read the Documentation*, and explains how to actually use *The Package*.

Introduction

magni [4] is a Python package which provides functionality for increasing the speed of image acquisition using Atomic Force Microscopy (AFM) (see e.g. [1] for an introduction). The image acquisition algorithms of magni are based on the Compressed Sensing (CS) signal acquisition paradigm (see e.g. [2] or [3] for an introduction) and include both sensing and reconstruction. The sensing part of the acquisition generates sensed data from regular images possibly acquired using AFM. This is done by AFM hardware simulation. The reconstruction part of the acquisition reconstructs images from sensed data. This is done by CS reconstruction using well-known CS reconstruction algorithms modified for the purpose. The Python implementation of the above functionality uses the standard library, a number of third-party libraries, and additional utility functionality designed and implemented specifically for magni. The functionality provided by magni can thus be divided into five groups:

- **Atomic Force Microscopy** (magni.afm): AFM specific functionality including AFM image acquisition, AFM hardware simulation, and AFM data file handling.
- **Compressed Sensing** (magni.cs): General CS functionality including signal reconstruction and phase transition determination.
- **Imaging** (magni.imaging): General imaging functionality including measurement matrix and dictionary construction in addition to visualisation and evaluation.
- **Reproducibility** (magni.reproducibility): Tools that may aid in increasing the reproducibility of result obtained using magni.
- **Utilities** (magni.utils): General Python utilities including multiprocessing, tracing, and validation.

See *Other Resources* as well as *Notation* for further documentation related to the project and the *Examples* to draw inspiration from.

References

[1] B. Bhushan and O. Marti , "Scanning Probe Microscopy - Principle of Operation, Instrumentation, and Probes", *in Springer Handbook of Nanotechnology*, pp. 573-617, 2010.

[2] D.L. Donoho, "Compressed Sensing", *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.

[3] E.J. Candès, J. Romberg, and T. Tao, "Robust Uncertainty Principles: Exact Signal Reconstruction From Highly Incomplete Frequency Information", *IEEE Transactions on Information Theory*, vol. 52, no.2, pp. 489–509, Feb. 2010.

Footnotes

[4] In Norse mythology, Magni is son of Thor and the god of strength. However, the word MAGNI could as well be an acronym for almost anything including "Making AFM Grind the Normal Impatience".

How to Read the Documentation

The included subpackages, modules, classes and functions are documented through Python docstrings using the same format as the third-party library, numpy, i.e. using the numpydoc standard. A description of any entity can thus be found in the source code of `magni` in the docstring of that entity. For readability, the documentation has been compiled using `Sphinx` to produce this HTML page which can be found in the magni folder under '/doc/build/html/index.html'. The entire documentation is also available as a PDF file in the magni folder under '/doc/build/pdf/index.pdf'.

Building the Documentation

The HTML documentation may be built from source using the supplied Makefile in the magni folder under '/doc/'. Make sure the required *Dependencies* for building the documentaion are installed. The build process consists of running three commands:

```
make sourceclean
make docapi
make html
```

Note: In the *make docapi* command it is assumed that the python interpreter is available on the PATH under the name *python*. If the python interpreter is available under another name, the PYTHONINT variable may be set, e.g. "make PYTHONINT=python2 docapi" if the python interpreter is named python2.

Run "make clean" to remove all builds created by `Sphinx` under '/doc/build'.

The Package

The source code of `magni` is released under the BSD 2-Clause, see the *License* section. This is an Open Source Initiative (OSI) complient licence. To install `magni`, follow the instructions given under *Download and Installation*.

`magni` has been tested with Anaconda 1.9.0 (64-bit) on Linux. It may or may not work with other Python distributions and/or operating systems. See also the list of *Dependencies* for `magni`.

License

Primary developers
Christian Schou Oxvig and Patrick Steffen Pedersen.

Additional developers
Jan Østergaard, Thomas Arildsen, Tobias L. Jensen, and Torben Larsen.

Institution
Aalborg University, Department of Electronic Systems, Signal and Information Processing, Niels Jernes Vej 12, DK-9220 Aalborg, Denmark.

Download and Installation

All official releases of `magni` are available for download at doi:10.5278/VBN/MISC/Magni. The source code is hosted on GitHub at https://github.com/SIP-AAU/Magni.

To use Magni, extract the downloaded archive and include the extracted magni folder in your PYTHONPATH.

> **Note:** The `magni` package (excluding examples and documentation) is also available on an "as is" basis in source form at PyPi and as a conda package at Binstar.

Dependencies

`magni` has been designed for use with `Python 2` >= 2.7 or `Python 3` >= 3.3

Required third party dependencies for `magni` are:

- `PyTables` (Tested on version >=3.1)

- Numpy (Tested on version >= 1.8)
- Scipy (Tested on version >= 0.13)
- Matplotlib (Tested on version >= 1.3)

Optional third party dependencies for magni are:

- IPython (Tested on version >= 1.1) (For running the IPython notebook examples)
- Math Kernel Library (MKL) (Tested on version >= 11.1) (For accelerated vector operations)
- Sphinx (Tested on version >= 1.2) (For building the documentation from source)
- Napoleon (Tested on version >= 0.2.6) (For building the documentation from source)

> **Note:** When using the magni.utils.multiprocessing subpackage, it is generally a good idea to restrict acceleration libraries like MKL or OpenBLAS to use a single thread. If MKL is installed, this is done automatically at runtime in the magni.utils.multiprocessing subpackage. If other libraries than MKL are used, the user has to manually set an appropriate evironmental variable, e.g. OMP_NUM_THREADS.

You may use the *dep_check.py* script found in the Magni folder under '/magni/tests/' to check for missing dependencies for Magni. Simply run the script to print a dependency report, e.g.:

```
python dep_check.py
```

Bug Reports

Found a bug? Bug report may be submitted using the magni GitHub issue tracker. Please include all relevant details in the bug report, e.g. version of Magni, input/output data, stack traces, etc. If the supplied information does not entail preproducibility of the problem, there is no way we can fix it.

> **Note: Due to limited funds, we are unfortunately unable make any guarantees, whatsoever, that reported bugs will be fixed.**

Other Resources

Papers published in relation to the Enabling Fast Image Acquisition for Atomic Force Microscopy using Compressed Sensing project:

- T. L. Jensen, T. Arildsen, J. Østergaard, and T. Larsen, "Reconstruction of Undersampled Atomic Force Microscopy Images : Interpolation versus Basis Pursuit", in *International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, Kyoto, Japan, December 2 - 5, 2013, pp. 130-135, doi:10.1109/SITIS.2013.32.

Notation

To the extent possible, a consistent notation has been used in the documentation and implementation of algorithms that are part of magni. All the details are described in *A Note on Notation*.

A Note on Notation

A much as possible, a consistent notation is used in the **magni** package. This implies that variable names are shared between functions that are related. Furthermore a consistent coordinate system is used for the description of related surfaces.

The Compressed Sensing Reconstruction Problem

In the **magni.cs** subpackage, a consistent naming scheme is used for variables, i.e., vectors and matrices. This section gives an overview of the chosen notation. For the purpose of illustration, consider the Basis Pursuit CS reconstruction problem [1]:

$$\begin{aligned} &\text{minimize} && ||\alpha||_1 \\ &\text{subject to} && \mathbf{y} = \mathbf{A}\alpha \end{aligned}$$

Here $\mathbf{A} \in \mathbb{C}^{m \times n}$ is the matrix product of a sampling matrix $\mathbf{\Phi} \in \mathbb{C}^{m \times p}$ and a dictionary matrix $\mathbf{\Psi} \in \mathbb{C}^{p \times n}$. The dictionary coefficients are denoted $\alpha \in \mathbb{C}^n$ whereas the measurements are denoted $\mathbf{y} \in \mathbb{C}^m$.

Thus, the following relations are used:

$$\mathbf{A} = \mathbf{\Phi}\mathbf{\Psi}$$

$$\mathbf{x} = \mathbf{\Psi}\alpha$$

$$\begin{aligned} \mathbf{y} &= \mathbf{\Phi}\mathbf{x} \\ &= \mathbf{\Phi}\mathbf{\Psi}\alpha \\ &= \mathbf{A}\alpha \end{aligned}$$

Here the vector $\mathbf{x} \in \mathbb{C}^p$ represents the signal of interest. That is, it is the signal that is assumed to have a sparse representation in the dictionary $\mathbf{\Psi}$. The sparsity of the coefficient vector α, that is the size of the support set, is denoted $k = |\text{supp}(\alpha)|$.

> **Note:** Even though the above example involves complex vectors and matrices, the algorithms provided in `magni.cs` may be restricted to inputs and outputs that are real.

References

[1] S. Chen, D. L. Donoho, and M. A. Saunders, "Atomic Decomposition by Basis Pursuit", *Siam Review*, vol. 43, no. 1, pp. 129–159, Mar. 2001.

Handling Images as Matrices and Vectors

In **magni.imaging**, an image is considered a matrix $\mathbf{M} \in \mathbb{R}^{h \times w}$. That is, the image height is h whereas the width is w. In the **magni.cs** subpackage, the image must be represented as a vector. This is done by stacking the columns of $\mathbf{M}$ to form the vector $\mathbf{x}$. Thus, the dimension of the image vector representation is $n = h \cdot w$. The magni.imaging._util.vec2mat() (available as magni.imaging.vec2mat()) and magni.imaging._util.mat2vec() (available as magni.imaging.mat2vec()) may be use to convert between the matrix and vector notations.

When the matrix representation is used, the following coordinate system is used for it visual representation:

```
----------------> x (first axis - width w)
|
|
|
|
|
|
|
|
|
|
v
y (second axis - height h)
```

This way, a position on an AFM sample of size $h \times w$ is specified by a (x, y) coordinate pair.

Examples

The magni package includes a large number of examples showing its capabilities. See the dedicated *Examples* page for all the details.

Examples

All the examples are available as IPython Notebooks in the magni folder under '/examples/'. For an introduction to getting started with IPython Notebook see the official documentation.

Starting the IPython Notebook

Starting the IPython Notebook basically boils down to running:

```
ipython notebook
```

from a shell with the working directory set to the Magni '/examples/' folder. Remember to make sure that magni is available as described in *Download and Installation* prior to starting the IPython Notebook.

Examples overview

An overview of the available examples is given in the below table:

IPython Notebook Name	Example illustrates	Magni functionality used
afm-io	• Reading data from mi-file. • Handling the resulting buffers and images.	• magni.afm.io.read_mi_file • magni.afm.types.Buffer • magni.afm.types.Image
afm-reconstruction	• Reconstructing compressively sampled AFM images. • Simulating reconstruction of compressively sampled images from raster scanned images. • Calculating MSE and PSNR of reconstructed image.	• magni.afm.reconstruction • magni.imaging.evaluation
cs-phase_transition	• Estimating phase transitions using simulations. • Plotting phase transitions. • Plotting phase transition probability colormaps.	• magni.cs.phase_transition._util.determine • magni.cs.phase_transition.io • magni.cs.phase_transition.plotting
cs-phase_transition-config	• Using Magni configuration modules including setting and getting configuration values.	• magni.cs.phase_transition.config • magni.utils.config
cs-reconstruction	• Reconstruction of compressively sampled 1D signals.	• magni.cs.reconstruction
magni	• The typical work flow in compressively sampling and reconstructing AFM images using Magni.	• magni.afm
imaging-dictionaries	• Handling compressed sensing dictionaries using Magni.	• magni.imaging.dictionaries
imaging-domains	• Easy handling of an image in the three domains: image, measurement and sparse (dictionary).	• magni.imaging.domains.MultiDomainImage

IPython Notebook Name	Example illustrates	Magni functionality used
imaging-measurements	• Handling sampling/measurement patterns using Magni. • Sampling a surface. • Sampling an image. • Illustrating sampling patterns.	• magni.imaging.measurements
imaging-preprocessing	• Pre-processing an image prior to sampling • De-tilting AFM images.	• magni.imaging.preprocessing
reporducibility-io	• Annotating an HDF5 database to help in improving the reproducibility of the results it contains.	• magni.reproducibility.io
util-matrices	• Using the special Magni Matrix and MatrixCollection classes.	• magni.utils.matrices.Matrix • magni.utils.matrices.MatrixCollection
utils-plotting	• Using the predefined plotting options in Magni to create clearer and more visually pleasing plots.	• magni.utils.plotting
utils-validation	• Validation of function parameters • Disabling input validation to reduce computation overhead	• magni.utils.validation

API Overview

An overview of the high level magni API is given below:

magni package

Package providing a toolbox for compressed sensing for atomic force microscopy.

Routine listings

afm
: Subpackage providing atomic force miscroscopy specific functionality.

cs
: Subpackage providing generic compressed sensing functionality.

imaging
: Subpackage providing generic imaging functionality.

tests
: Subpackage providing unittesting of the other subpackages.

utils
: Subpackage providing support functionality for the other subpackages.

Notes

See the README file for additional information.

Subpackages

magni.afm package

Subpackage providing atomic force miscroscopy specific functionality.

The present subpackage includes functionality for handling AFM files and data and functionality for utilizing the other subpackages for such AFM data.

Routine listings

config
: Module providing configuration options for this subpackage.

io
: Module providing input/output functionality for MI files.

reconstruction
: Module providing reconstruction and analysis of reconstructed images.

types
: Module providing data container classes for MI files.

Submodules

magni.afm.config module

Module providing configuration options for the magni.afm subpackage.

Routine listings

get(key=None)
: Get the value of one or more configuration options.

set(dictionary={}, **kwargs)
: Set the value of one or more configuration options.

See also:

magni.utils.config
The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config** and assigns handles for the **get** and **set** methods of that class instance. The configuration options are the following:

algorithm : *{'iht', 'sl0'}*
The compressed sensing reconstruction algorithm to use (the default is 'iht').

magni.afm.io module

Module providing input/output functionality for MI files.

Routine listings

read_mi_file(path)
Read MI file and output an instance of an appropriate class.

See also:

magni.afm.types
Data container classes.

magni.afm.io.**read_mi_file**(*path*)

Read MI file and output an instance of an appropriate class.

Parameters:	**path** (*str*) – The path of the MI file.
Returns:	**obj** (*None*) – An instance of an appropriate class depending on the content of the MI file.

Notes

See the specification of the MI file format for an understanding of the steps performed in reading the MI file. An MI file can contain different types of data and thus the class of the output can vary.

Examples

An example of how to use read_mi_file to read the example MI file provided with the package:

```
>>> from magni.afm.io import read_mi_file
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> mi_file = read_mi_file(path)
```

magni.afm.io.**_convert_mi_image_data**(*buf*, *datatype*)

Convert the data part of an MI image file to a 1D numpy.ndarray.

Parameters:	• **buf** (*str*) – The raw data part of an MI image file. • **datatype** (*str*) – A string specifying how to interpret the data in the data buffer.
Returns:	**data** (*numpy.ndarray*) – The converted data part.

Notes

See the specification of the MI file format for a list of datatypes and an explanation of their meaning.

magni.afm.io.**_convert_mi_value**(*string*)

Convert the value of an MI header line to a meaningful Python type.

Parameters:	**string** (*str*) – The string representation of the MI header line value.
Returns:	**value** (*None*) – The converted value.

Notes

See the specification of the MI file format for an explanation of the different value types and the conversion from the string representation.

magni.afm.reconstruction module

Module providing AFM image reconstruction and analysis of reconstructed images.

Routine listings

analyse(x, Phi, Psi)
: Sample an image, reconstruct it, and analyse the reconstructed image.

reconstruct(y, Phi, Psi)
: Reconstruct an image from compressively sensed measurements.

magni.afm.reconstruction.**analyse**(*x*, *Phi*, *Psi*)

Sample an image, reconstruct it, and analyse the reconstructed image.

Parameters:	• **x** (*numpy.ndarray*) – The original image vector. • **Phi** (*magni.utils.matrices.Matrix or numpy.matrix*) – The measurement matrix. • **Psi** (*magni.utils.matrices.Matrix or numpy.matrix*) – The dictionary.
Returns:	**x** (*numpy.ndarray*) – The reconstructed image vector.

See also:

magni.afm.config()
: Configuration options.

magni.imaging.evaluation()
: Image reconstruction quality evaluation.

Examples

Prior to the actual example, data is loaded and a measurement matrix and a dictionary are defined. First, the example MI file provided with the package is loaded:

```
>>> from magni.afm.reconstruction import analyse
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> mi_file = magni.afm.io.read_mi_file(path)
>>> mi_buffer = mi_file.get_buffer('Topography')[0]
>>> mi_data = mi_buffer.get_data()
>>> x = magni.imaging.mat2vec(mi_data)
```

Next, a measurement matrix is defined. This matrix is equal to the matrix generated by running *np.eye(len(x))[::2, :]* but for speed, the matrix is instead defined with fast operations:

```
>>> def Phi_A(x):
...     y = x[::2]
...     return y
>>> def Phi_T(y):
...     x = np.zeros((2 * len(y), 1))
...     x[::2] = y
...     return x
>>> Phi = magni.utils.matrices.Matrix(Phi_A, Phi_T, (),
...                                   (int(len(x) / 2), len(x)))
```

Next, a dictionary is defined. This dictionary is the DCT basis likewise defined with fast operations:

```
>>> Psi = magni.imaging.dictionaries.get_DCT(mi_data.shape)
```

Finally, the actual example:

```
>>> print('MSE: {:.2f}, PSNR: {:.2f}'.format(*analyse(x, Phi, Psi)))
MSE: 0.23, PSNR: 5.92
```

magni.afm.reconstruction.**reconstruct**(*y*, *Phi*, *Psi*)

Reconstruct an image from compressively sensed measurements.

Parameters:
- **y** (*numpy.ndarray*) – The measurement vector.
- **Phi** (*magni.utils.matrices.Matrix or numpy.matrix*) – The measurement matrix.
- **Psi** (*magni.utils.matrices.Matrix or numpy.matrix*) – The dictionary.

Returns: **x** (*numpy.ndarray*) – The reconstructed image vector.

See also:

magni.afm.config()
Configuration options.

magni.cs.reconstruction()
Compressed sensing reconstruction algorithms.

Examples

Prior to the actual example, data is loaded and a measurement matrix and a dictionary are defined. First, the example MI file provided with the package is loaded:

```
>>> from magni.afm.reconstruction import reconstruct
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> mi_file = magni.afm.io.read_mi_file(path)
>>> mi_buffer = mi_file.get_buffer('Topography')[0]
>>> mi_data = mi_buffer.get_data()
>>> x = magni.imaging.mat2vec(mi_data)
```

Next, a measurement matrix is defined. This matrix is equal to the matrix generated by running *np.eye(len(x))[::2, :]* but for speed, the matrix is instead defined with fast operations:

```
>>> def Phi_A(x):
...     y = x[::2]
...     return y
>>> def Phi_T(y):
...     x = np.zeros((2 * len(y), 1))
...     x[::2] = y
...     return x
>>> Phi = magni.utils.matrices.Matrix(Phi_A, Phi_T, (),
...                                   (int(len(x) / 2), len(x)))
```

Next, a dictionary is defined. This dictionary is the DCT basis likewise defined with fast operations:

```
>>> Psi = magni.imaging.dictionaries.get_DCT(mi_data.shape)
```

Finally, the actual example:

```
>>> y = Phi.dot(x)
>>> print('Maximum absolute pixel error: {:.3f}'
...       .format(np.abs(reconstruct(y, Phi, Psi) - x).max()))
Maximum absolute pixel error: 0.960
```

magni.afm.types module

Module providing data container classes for MI files.

The classes of this module can be used either directly or indirectly through the **io** module by loading an MI file.

Routine listings

Buffer()
: Data class for MI image buffer.

Image()
: Data class for MI image.

See also:

magni.afm.io
: MI file loading.

class magni.afm.types.**Buffer**(*data*, *hdrs*, *width*, *height*)

Data class for MI image buffer.

This class contains both buffer specific header lines and the 2D data of the buffer.

Parameters:
- **data** (*numpy.ndarray*) – The data of the buffer represented as a 1D numpy.ndarray.
- **hdrs** (*list or tuple*) – The buffer specific header lines.
- **width** (*int*) – The width in pixels of the area covered by the buffer.
- **height** (*int*) – The height in pixels of the area covered by the buffer.

Notes

See **_image_headers** for a description of the header lines.

Examples

The `__init__` function is implicitly called when loading, for example, the MI file provided with the package:

```
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> mi_file = magni.afm.io.read_mi_file(path)
>>> mi_buffer = mi_file.get_buffer()[0]
```

This buffer can have a number of attributes (stored as header lines in the MI file) including the 'bufferLabel' attribute:

```
>>> print(mi_buffer.get_attr('bufferLabel'))
Topography
```

The primary purpose of this class is, however, to contain the 2D data of a buffer:

```
>>> data = mi_buffer.get_data()
>>> print('Buffer, Type: {}, Shape: {}'.format(str(type(data))[-15:-2],
... data.shape))
Buffer, Type: numpy.ndarray, Shape: (256, 256)
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*data*, *hdrs*, *width*, *height*)

_validate_get_attr(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

get_attr(*key=None*)

Get a copy of all header lines or a specific header line.

Parameters:	**key** (*str or None, optional*) – The name of the header line to retrieve (the default is None, which implies retrieving a copy of all header lines).
Returns:	**value** (*dict or None*) – The value of the specified key, if key is not None. Otherwise, a copy of the header lines.

_validate_get_data(*intensity_func*, *intensity_args*)

Validate the get_data function.

> **See also:**
>
> Buffer.get_data()
>
> The validated function.
>
> **fat.utils.validation.validate()**
>
> Validation.

get_data(*intensity_func=None*, *intensity_args=()*)

Get the 2D data of the buffer.

The optional *intensity_func* and *intensity_args* can be used to manipulate the intensity of the image before getting the data.

Parameters:	• **intensity_func** (*FunctionType, optional*) – The handle to the function used to manipulate the image intensity (the default is None, which implies that no intensity manipulation is used). • **intensity_args** (*list or tuple, optional*) – The arguments that are passed to the *intensity_func* (the default is (), which implies that no arguments are passed).
Returns:	**data** (*numpy.ndarray*) – The 2D data of the buffer.

class magni.afm.types.**Image**(*data*, *hdrs*)

Data class for MI image.

This class contains both image specific header lines and the buffers of the image.

Parameters:	• **data** (*numpy.ndarray*) – The data part of the MI image. • **hdrs** (*list or tuple*) – The image specific header lines.

Notes

See **_mi_headers** for a description of the header lines.

Examples

The __init__ function is implicitly called when loading, for example, the MI file provided with the package:

```
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> image = magni.afm.io.read_mi_file(path)
```

This image can have a number of attributes (stored as header lines in the MI file) including the ‘scanSpeed’ attribute:

```
>>> print('{:5.2f}'.format(image.get_attr('scanSpeed')))
 1.01
```

The primary purpose of this class is, however, to contain the buffers of an MI image file:

```
>>> buffers = image.get_buffer()
>>> for b in buffers[0:5:2]:
...     print('Buffer: {}'.format(b.get_attr('bufferLabel')))
Buffer: Topography
Buffer: Deflection
Buffer: Friction
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*data*, *hdrs*)

_validate_get_attr(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

get_attr(*key=None*)

Get a copy of all header lines or a specific header line.

Parameters: **key** (*str or None, optional*) – The name of the header line to retrieve (the default is None, which implies retrieving a copy of all header lines).

Returns: **value** (*dict or None*) – The value of the specified key, if key is not None. Otherwise, a copy of the header lines.

_validate_get_buffer(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

get_buffer(*key=None*)

Get all buffers or a specific buffer.

Parameters:	**key** (*str or None, optional*) – The name of the buffer to retrieve (the default is None, which implies retrieving all buffers).
Returns:	**value** (*dict or None*) – The buffer of the specified key, if key is not None. Otherwise, a dict with all the buffers.

magni.cs package

Subpackage providing generic compressed sensing functionality.

Routine listings

phase_transition
: Subpackage providing phase transition determination functionality.

reconstruction
: Subpackage providing implementations of generic reconstruction algorithms.

Subpackages

magni.cs.phase_transition package

Subpackage providing phase transition determination.

Routine listings

config
: Module providing configuration options for this subpackage.

io
: Module providing input/output functionality for stored phase transitions.

plotting
: Module providing plotting for this subpackage.

determine(algorithm, path, label='default', overwrite=False)
: Determine the phase transition of a reconstruction algorithm.

Notes

See _util for documentation of determine.

The phase transition of a reconstruction algorithm describes the reconstruction capabilities of that reconstruction algorithm. For a description of the concept of phase transition, see [1].

References

[1] C. S. Oxvig, P. S. Pedersen, T. Arildsen, and T. Larsen, "Surpassing the Theoretical 1-norm Phase Transition in Compressive Sensing by Tuning the Smoothed l0 Algorithm", *in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Vancouver, Canada, May 26-31, 2013, pp. 6019-6023.

Submodules

magni.cs.phase_transition._analysis module

Module providing functionality for analysing the simulation results.

Routine listings

run(path, label)
: Determine the phase transition from the simulation results.

> **See also:**
>
> **magni.cs.phase_transition.config**
> Configuration options.

Notes

For a description of the concept of phase transition, see [1].

References

[1] C. S. Oxvig, P. S. Pedersen, T. Arildsen, and T. Larsen, "Surpassing the Theoretical 1-norm Phase Transition in Compressive Sensing by Tuning the Smoothed l0 Algorithm", *in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Vancouver, Canada, May 26-31, 2013, pp. 6019-6023.

magni.cs.phase_transition._analysis.**run**(*path*, *label*)

Determine the phase transition from the simulation results.

The simulation results should be present in the HDF5 database specified by *path* in the pytables group specified by *label* in an array named 'dist'. The determined phase transition is stored in the same HDF5 database, in the same pytables group in an array named 'phase_transition'.

Parameters:
- **path** (*str*) – The path of the HDF5 database.
- **label** (*str*) – The path of the pytables group in the HDF5 database.

> **See also:**
>
> **_estimate_PT()**
> The actual phase transition estimation.

Notes

A simulation is considered successful if the simulation result is less than 10 to the power of -4.

magni.cs.phase_transition._analysis.**_estimate_PT**(*rho*, *success*)

Estimate the phase transition location for a given delta.

The phase transition location is estimated using logistic regression. The algorithm used for this is Newton's method.

Parameters:	• **rho** (*ndarray*) – The rho values. • **success** (*ndarray*) – The success indicators.
Returns:	**rho** (*float*) – The estimated phase transition location.

Notes

The function includes a number of non-standard ways of handling numerical and convergence related issues. This will be changed in a future version of the code.

magni.cs.phase_transition._backup module

Module providing backup capabilities for the monte carlo simulations.

The backup stores the simulation results and the simulation timings pointwise for the points in the delta-rho simulation grid. The set function targets a specific point while the get function targets the entire grid in order to keep the overhead low.

Routine listings

create(path)
    Create the HDF5 backup database with the required arrays.

get(path)
    Return which of the results have been stored.

set(path, ij_tuple, stat_time, stat_dist)
    Store the simulation data of a specified point.

See also:

magni.cs.phase_transition.config
    Configuration options.

Notes

In practice, the backup database includes an additional array for tracking for which points data has been stored. By first storing the data and then modifying this array, the data is guaranteed to have been stored, when the array is modified.

magni.cs.phase_transition._backup.**create**(*path*)

Create the HDF5 backup database with the required arrays.

The required arrays are an array for the simulation results, an array for the simulation timings, and an array for tracking the status.

Parameters:	**path** (*str*) – The path of the HDF5 backup database.

See also:

magni.cs.phase_transition.config()
Configuration options.

magni.cs.phase_transition._backup.**get**(*path*)

Return which of the results have been stored.

The returned value is a copy of the boolean status array indicating which of the results have been stored.

Parameters:	**path** (*str*) – The path of the HDF5 backup database.
Returns:	**status** (*ndarray*) – The boolean status array.

magni.cs.phase_transition._backup.**set**(*path*, *ij_tuple*, *stat_time*, *stat_dist*)

Store the simulation data of a specified point.

Parameters:

- **path** (*str*) – The path of the HDF5 backup database.
- **ij_tuple** (*tuple*) – A tuple (i, j) containing the parameters i, j as listed below.
- **i** (*int*) – The delta-index of the point in the delta-rho grid.
- **j** (*int*) – The rho-index of the point in the delta-rho grid.
- **stat_dist** (*ndarray*) – The simulation results of the specified point.
- **stat_time** (*ndarray*) – The simulation timings of the specified point.

magni.cs.phase_transition._data module

Module providing problem suite instance generation functionality.

The problem suite instances consist of a matrix, A, and a coefficient vector, alpha, with which the measurement vector, y, can be generated.

Routine listings

generate_matrix(m, n)
Generate a matrix belonging to a specific problem suite.

generate_vector(n, k)
Generate a vector belonging to a specific problem suite.

Notes

The matrices and vectors generated in this module use the numpy.random submodule. Consequently, the calling script or function should control the seed to ensure reproducibility.

Examples

Generate a problem suite instance:

```
>>> from magni.cs.phase_transition import _data
>>> m, n, k = 400, 800, 100
>>> A = _data.generate_matrix(m, n)
>>> alpha = _data.generate_vector(n, k)
>>> y = np.dot(A, alpha)
```

magni.cs.phase_transition._data.**generate_matrix**(*m*, *n*)

Generate a matrix belonging to a specific problem suite.

The available ensemble is the Uniform Spherical Ensemble. See Notes for a description of the ensemble.

Parameters:
- **m** (*int*) – The number of rows.
- **n** (*int*) – The number of columns.

Returns: **A** (*ndarray*) – The generated matrix.

Notes

The Uniform Spherical Ensemble:

The matrices of this ensemble have i.i.d. Gaussian entries and its columns are normalised to have unit length.

magni.cs.phase_transition._data.**generate_vector**(*n*, *k*)

Generate a vector belonging to a specific problem suite.

The available ensembles are the Gaussian ensemble and the Rademacher ensemble. See Notes for a description of the ensembles. Which of the available ensembles is used, is specified as a configuration option. Note, that the non-zero *k* non-zero coefficients are the *k* first entries.

Parameters:
- **n** (*int*) – The length of the vector.
- **k** (*int*) – The number of non-zero coefficients.

Returns: **alpha** (*ndarray*) – The generated vector.

See also:

magni.cs.phase_transition.config()

Configuration options.

Notes

The Gaussian ensemble :

The non-zero coefficients are drawn from the normal Gaussian distribution.

The Rademacher ensemble:

The non-zero coefficients are drawn from the constant amplitude with random signs ensemble.

magni.cs.phase_transition._simulation module

Module providing the actual simulation functionality.

Routine listings

run(algorithm, path, label)

Simulate a reconstruction algorithm.

See also:

magni.cs.phase_transition.config

Configuration options.

Notes

The results of the simulation are backed up throughout the simulation. In case the simulation is interrupted during execution, the simulation will resume from the last backup point when run again.

magni.cs.phase_transition._simulation.**run**(*algorithm*, *path*, *label*)

Simulate a reconstruction algorithm.

The simulation results are stored in a HDF5 database rather than returned by the function.

Parameters:
- **algorithm** (*function*) – A function handle to the reconstruction algorithm.
- **path** (*str*) – The path of the HDF5 database where the results should be stored.
- **label** (*str*) – The label assigned to the simulation results.

magni.cs.phase_transition._simulation.**_simulate**(*algorithm*, *ij_tuple*, *seeds*, *path*)

Run a number of monte carlo simulations in a single delta-rho point.

The result of a simulation is the simulation error distance, i.e., the ratio between the energy of the coefficient residual and the energy of the coefficient vector. The time of the simulation is the execution time of the reconstruction attempt.

Parameters:
- **algorithm** (*function*) – A function handle to the reconstruction algorithm.
- **ij_tuple** (*tuple*) – A tuple (i, j) containing the parameters i, j as listed below.
- **i** (*int*) – The delta-index of the point in the delta-rho grid.
- **j** (*int*) – The rho-index of the point in the delta-rho grid.
- **seeds** (*ndarray*) – The seeds to pass to numpy.random when generating the problem suite instances.
- **path** (*str*) – The path of the HDF5 backup database.

See also:

magni.cs.phase_transition._data.generate_matrix()

Matrix generation.

magni.cs.phase_transition._data.generate_vector()

Coefficient vector generation.

magni.cs.phase_transition._util module

Module providing the public function of the magni.cs.phase_transition subpackage.

magni.cs.phase_transition._util.**determine**(*algorithm*, *path*, *label='default'*, *overwrite=False*)

Determine the phase transition of a reconstruction algorithm.

The phase transition is determined from a number of monte carlo simulations on a delta-rho grid for a given problem suite.

Parameters:

- **algorithm** (*function*) – A function handle to the reconstruction algorithm.
- **path** (*str*) – The path of the HDF5 database where the results should be stored.
- **label** (*str*) – The label assigned to the phase transition (the default is 'default').
- **overwrite** (*bool*) – A flag indicating if an existing phase transition should be overwritten if it has the same path and label (the default is False).

See also:

magni.cs.phase_transition.config()
: Configuration options.

magni.cs.phase_transition._simulation.run()
: The actual simulation.

magni.cs.phase_transition._analysis.run()
: The actual phase determination.

Examples

An example of how to use this function is provided in the *examples* folder in the *cs-phase_transition.ipynb* ipython notebook file.

magni.cs.phase_transition.config module

Module providing configuration options for the phase_transition subpackage.

Routine listings

get(key=None)
: Get the value of one or more configuration options.

set(dictionary={}, **kwargs)
: Set the value of one or more configuration options.

See also:

magni.utils.config
: The Configger class used.

Notes

This module instantiates the *Configger* class provided by magni.utils.config and assigns handles for the **get** and **set** methods of that class instance. The configuration options are the following:

seed : *int*
: The seed used when picking seeds for generating data for the monte carlo simulations (the default is None, which implies an arbitrary seed).

n : *int*
: The length of the coefficient vector (the default is 800).

delta : *list or tuple*
: The delta values of the delta-rho grid whose points are used for the monte carlo simulations (the default is [0., 1.]).

rho : *list or tuple*
: The rho values of the delta-rho grid whose points are used for the monte carlo simulations (the default is [0., 1.]).

monte_carlo : *int*
: The number of monte carlo simulations to run in each point of the delta-rho grid (the default is 1).

coefficients : *{'rademacher', 'gaussian'}*
: The distribution which the non-zero coefficients in the coefficient vector are drawn from.

magni.cs.phase_transition.io module

Module providing input/output functionality for stored phase transitions.

Routine listings

load_phase_transition(path, label='default')
: Load the coordinates of a phase transition from a HDF5 file.

magni.cs.phase_transition.io.**load_phase_transition**(*path*, *label='default'*)

Load the coordinates of a phase transition from a HDF5 file.

This function is used to load the phase transition from the output file generated by **magni.cs.phase_transition.determine**.

Parameters:	• **path** (*str*) – The path of the HDF5 file where the phase transition is stored. • **label** (*str*) – The label assigned to the phase transition (the default is 'default').
Returns:	• **delta** (*np.ndarray*) – The delta values of the phase transition points. • **rho** (*np.ndarray*) – The rho values of the phase transition points.

See also:

magni.cs.phase_transition.determine()
: Phase transition determination.

magni.cs.phase_transition.plotting()
: Phase transition plotting.

Examples

An example of how to use this function is provided in the *examples* folder in the *cs-phase_transition.ipynb* ipython notebook file.

magni.cs.phase_transition.plotting module

Module providing plotting for the `magni.cs.phase_transition` subpackage.

Routine listings

plot_phase_transitions(curves, plot_l1=True, output_path=None)
: Function for plotting phase transition boundary curves.

plot_phase_transition_colormap(dist, delta, rho, plot_l1=True, output_path=None)
: Function for plotting reconstruction probabilities in the phase space.

magni.cs.phase_transition.plotting.**plot_phase_transitions**(*curves*, *plot_l1=True*, *output_path=None*)

Plot of a set of phase transition boundary curves.

The set of phase transition boundary curves are plotted an saved under the *output_path*, if specified. The *curves* must be a list of dictionaries each having keys *delta*, *rho*, and *label*. *delta* must be an *ndarray* of δ values in the phase space. *rho* must be an *ndarray* of the corresponding ρ values in the phase space. *label* must be a *str* describing the curve.

Parameters:
- **curves** (*list*) – A list of dicts describing the curves to plot.
- **plot_l1** (*bool, optional*) – Whether or not to plot the theoretical ℓ_1 curve (the default is True).
- **output_path** (*str, optional*) – Path (including file type extension) under which the plot is saved (the default value is None which implies, that the plot is not saved).

Notes

The plotting is done using `matplotlib`, which implies that an open figure containing the plot will result from using this function.

Tabulated values of the theoretical ℓ_1 phase transition boundary is available at http://people.maths.ox.ac.uk/tanner/polytopes.shtml

Examples

For example,

```
>>> from magni.cs.phase_transition.plotting import plot_phase_transitions
>>> delta = np.array([0.1, 0.2, 0.9])
>>> rho = np.array([0.1, 0.3, 0.8])
>>> curves = [{'delta': delta, 'rho': rho, 'label': 'data1'}]
>>> output_path = 'phase_transitions.pdf'
>>> plot_phase_transitions(curves, output_path=output_path)
```

magni.cs.phase_transition.plotting.**plot_phase_transition_colormap**(*dist*, *delta*, *rho*, *plot_l1=True*, *output_path=None*)

Create a colormap of the phase space reconstruction probabilities.

The *delta* and *rho* values span a 2D grid in the phase space. Reconstruction probabilities are then calculated from the *dist* 3D array of reconstruction error distances. The resulting 2D grid of reconstruction probabilites is visualised over the square centers in this 2D grid using a colormap. Values in this grid at lower indicies correspond to lower values of δ and ρ. If *plot_l1* is True, then the theoretical l1 curve is overlayed the colormap. The colormap is saved under the *output_path*, if specified.

Parameters:
- **dist** (*ndarray*) – A 3D array of reconstruction error distances.
- **delta** (*ndarray*) – δ values used in the 2D grid.
- **rho** (*ndarray*) – ρ values used in the 2D grid.
- **plot_l1** (*bool*) – Whether or not to plot the theoretical ℓ_1 curve. (the default is True)
- **output_path** (*str, optional*) – Path (including file type extension) under which the plot is saved (the default value is None which implies, that the plot is not saved).

See also:

magni.cs.phase_transition.io.load_phase_transition()
Loading phase transitions from an HDF database.

Notes

The plotting is done using **matplotlib**, which implies that an open figure containing the plot will result from using this function.

The values in *delta* and *rho* are assumed to be equally spaced.

Due to the *centering* of the color coded rectangles, they are not necessarily square towards the ends of the intervals defined by *delta* and *rho*.

Tabulated values of the theoretical ℓ_1 phase transition boundary is available at http://people.maths.ox.ac.uk/tanner/polytopes.shtml

Examples

For example,

```
>>> from magni.cs.phase_transition.plotting import (
...     plot_phase_transition_colormap)
>>> delta = np.array([0.2, 0.5, 0.8])
>>> rho = np.array([0.3, 0.6])
>>> dist = np.array([[[1.35e-08, 1.80e-08], [1.08, 1.11]],
... [[1.40e-12, 8.32e-12], [8.57e-01, 7.28e-01]], [[1.92e-13, 1.17e-13],
... [2.10e-10,   1.12e-10]]])
>>> out_path = 'phase_transition_cmap.pdf'
>>> plot_phase_transition_colormap(dist, delta, rho, output_path=out_path)
```

magni.cs.phase_transition.plotting.**_plot_theoretical_l1**(*axes*)

Plot the theoretical l1 phase transition on the *axes*.

Parameters: **axes** (*matplotlib.axes.Axes*) – The matplotlib Axes instance on which the theoretical l1 phase transition should be plotted.

Notes

The plotted theoretical $\ell 1$ phase transition is based on tabulated values of available at http://people.maths.ox.ac.uk/tanner/polytopes.shtml

magni.cs.reconstruction package

Subpackage providing implementations of generic reconstruction algorithms.

Each subpackage provides a family of generic reconstruction algorithms. Thus each subpackage has a config module and a run function which provide the interface of the given family of reconstruction algorithms.

Routine listings

iht
: Subpackage providing an implementation of Iterative Hard Thresholding (IHT)

sl0
: Subpackage providing implementations of Smoothed l0 Norm (SL0).

Subpackages

magni.cs.reconstruction.iht package

Subpackage providing an implementation of Iterative Hard Thresholding (IHT).

Routine listings

config
: Module providing configuration options for this subpackage.

run(y, A)
: Run the IHT reconstruction algorithm.

Notes

See _original for documentation of **run**.

The IHT reconstruction algorithm is described in [1].

References

[1] A. Maleki and D.L. Donoho, “Optimally Tuned Iterative Reconstruction Algorithms for Compressed Sensing”, *IEEE Journal Selected Topics in Signal Processing*, vol. 3, no. 2, pp. 330–341, Apr. 2010.

Submodules

magni.cs.reconstruction.iht._original module

Module providing the actual reconstruction algorithm.

Routine listings

run(y, A)
: Run the IHT reconstruction algorithm.

See also:

magni.cs.reconstruction.iht.config

Configuration options.

Notes

The IHT reconstruction algorithm is described in [1].

References

[1] A. Maleki and D.L. Donoho, "Optimally Tuned Iterative Reconstruction Algorithms for Compressed Sensing", *IEEE Journal Selected Topics in Signal Processing*, vol. 3, no. 2, pp. 330–341, Apr. 2010.

magni.cs.reconstruction.iht._original.**run**(*y*, *A*)

Run the IHT reconstruction algorithm.

Parameters:	• **y** (*ndarray*) – The m x 1 measurement vector. • **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.
Returns:	**alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

See also:

_calculate_far()

Optimal False Acceptance Rate calculation.

_normalise()

Matrix normalisation.

Notes

In each iteration, the threshold is robustly calculated as a fixed multiple of the standard deviation of the calculated correlations. The fixed multiple is based on the False Acceptance Rate (FAR) assuming a Gaussian distribution of the correlations.

The algorithm terminates after a fixed number of iterations or if the ratio between the 2-norm of the residual and the 2-norm of the measurements falls below the specified *tolerance*.

Examples

For example, recovering a vector from random measurements

```
>>> from magni.cs.reconstruction.iht._original import run
>>> np.random.seed(seed=6021)
>>> A = 1 / np.sqrt(80) * np.random.randn(80, 200)
>>> x = np.zeros((200, 1))
>>> x[:10] = 1
>>> y = A.dot(x)
>>> x_hat = run(y, A)
>>> x_hat[:12]
array([[ 0.99836297],
       [ 1.00029086],
       [ 0.99760224],
       [ 0.99927175],
       [ 0.99899124],
       [ 0.99899434],
       [ 0.9987368 ],
       [ 0.99801849],
       [ 1.00059408],
       [ 0.9983772 ],
       [ 0.        ],
       [ 0.        ]])
>>> (np.abs(x_hat) > 1e-2).sum()
10
```

magni.cs.reconstruction.iht._original.**_calculate_far**(*delta*)

Calculate the optimal False Acceptance Rate for a given indeterminacy.

Parameters:	**delta** (*float*) – The indeterminacy, m / n, of a system of equations of size m x n.
Returns:	**FAR** (*float*) – The optimal False Acceptance Rate for the given indeterminacy.

Notes

The optimal False Acceptance Rate to be used in connection with the interference heuristic presented in the paper “Optimally Tuned Iterative Reconstruction Algorithms for Compressed Sensing” [2] is calculated from a set of optimal values presented in the same paper. The calculated value is found from a linear interpolation or extrapolation on the known set of optimal values.

References

[2] A. Maleki and D.L. Donoho, “Optimally Tuned Iterative Reconstruction Algorithms for Compressed Sensing”, *IEEE Journal Selected Topics in Signal Processing*, vol. 3, no. 2, pp. 330–341, Apr. 2010.

magni.cs.reconstruction.iht.config module

Module providing configuration options for the **magni.cs.reconstruction.iht** subpackage.

Routine listings

get(key=None)
: Get the value of one or more configuration options.

set(dictionary={}, **kwargs)
: Set the value of one or more configuration options.

See also:

magni.utils.config

The Configger class used.

Notes

This module instantiates the *Configger* class provided by **magni.utils.config** and assigns handles for the **get** and **set** methods of that class instance. The configuration options are the following:

iterations : *int*
: The maximum number of iterations to do (the default is 300).

kappa : *float*
: The relaxation parameter used in the algorithm (the default is 0.65).

precision_float : *dtype*
: The floating point precision used for the computations (the default is float64).

threshold : *['far', 'oracle']*
: The method for selecting the threshold value.

threshold_rho : *float*
: The assumed rho value used for selecting the threshold value if using the oracle method.

tolerance : *float*
: The least acceptable ratio of residual to measurements (in 2-norm) to break the interations (the default is 0.001).

magni.cs.reconstruction.iht.config.**get**(*key=None*)

Get the value of one or more configuration options.

This function wraps 'Configger.get' in order to convert any float options to the specified precision before returning the option(s).

See also:

magni.utils.config.Configger.get()

The wrapped function.

magni.cs.reconstruction.sl0 package

Subpackage providing implementations of Smoothed l0 Norm SL0).

The implementations provided are the original SL0 reconstruction algorithm and a modified SL0 reconstruction algorithm. The algorithm used depends on the 'algorithm' configuration option: 'std' refers to the original algorithm while 'mod' (default) refers to the modified algorithm.

Routine listings

config
: Module providing configuration options for this subpackage.

run(y, A)

Run the specified SL0 reconstruction algorithm.

Notes

See _util for documentation of run.

The original SL0 reconstruction algorithm by Mohimani et. al is described in [1] whereas the constraint elimiation intepretation of the original SL0 algorithm by Cui et. al. is described in [2]. The modified SL0 reconstruction algorithm by Oxvig et. al. is described in [3]. Specifically, the provided sequential implementations are:

std : The standard SL0 algorithm
  For delta < 0.55: Standard projection algorithm by Mohimani et. al.
  For delta >= 0.55: Standard constraint elimination algorithm by Cui et. al.
mod : The modified SL0 algorithm (the default)
  For delta < 0.55: Modified projection algorithm
  For delta >= 0.55: Modified constraint elimination algorithm

References

[1] H. Mohimani, M. Babaie-Zadeh, and C. Jutten, "A Fast Approach for Overcomplete Sparse Decomposition Based on Smoothed l0 Norm", *IEEE Transactions on Signal Processing*, vol. 57, no. 1, pp. 289-301, Jan. 2009.

[2] Z. Cui, H. Zhang, and W. Lu, "An Improved Smoothed l0-norm Algorithm Based on Multiparameter Approximation Function", *in 12th IEEE International Conference on Communication Technology (ICCT)*, Nanjing, China, Nov. 11-14, 2011, pp. 942-945.

[3] C. S. Oxvig, P. S. Pedersen, T. Arildsen, and T. Larsen, "Surpassing the Theoretical 1-norm Phase Transition in Compressive Sensing by Tuning the Smoothed l0 Algorithm", *in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Vancouver, Canada, May 26-31, 2013, pp. 6019-6023.

Submodules

magni.cs.reconstruction.sl0._modified module

Module providing the modified SL0 reconstruction algorithm.

Routine listings

run(y, A)
  Run the modified SL0 reconstruction algorithm.

See also:

magni.cs.reconstruction.sl0.config
  Configuration options.

Notes

The modified SL0 reconstruction algorithm is described in [1].
  For delta < 0.55: Modified projection algorithm
  For delta >= 0.55: Modified constraint elimination algorithm

References

[1] C. S. Oxvig, P. S. Pedersen, T. Arildsen, and T. Larsen, "Surpassing the Theoretical 1-norm Phase Transition in Compressive Sensing by Tuning the Smoothed l0 Algorithm", *in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Vancouver, Canada, May 26-31, 2013, pp. 6019-6023.

magni.cs.reconstruction.sl0._modified.**run**(*y*, *A*)

Run the modified SL0 reconstruction algorithm.

Parameters:	• **y** (*ndarray*) – The m x 1 measurement vector. • **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.
Returns:	**alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

See also:

_run_proj()

The original projection algorithm.

_run_feas()

The original constraint elimination algorithm.

Examples

For example, recovering a vector from random measurements

```
>>> from magni.cs.reconstruction.sl0._modified import run
>>> np.random.seed(seed=6021)
>>> A = 1 / np.sqrt(80) * np.random.randn(80, 200)
>>> x = np.zeros((200, 1))
>>> x[:10] = 1
>>> y = A.dot(x)
>>> x_hat = run(y, A)
>>> x_hat[:12]
array([[  9.99997941e-01],
       [  9.99999463e-01],
       [  1.00000090e+00],
       [  9.99998622e-01],
       [  1.00000078e+00],
       [  9.99998433e-01],
       [  1.00000025e+00],
       [  1.00000346e+00],
       [  1.00000088e+00],
       [  9.99995474e-01],
       [ -4.12075673e-07],
       [  2.17244596e-07]])
>>> (np.abs(x_hat) > 1e-2).sum()
10
```

magni.cs.reconstruction.sl0._modified.**_calc_sigma_start**(*delta*)

Calculate the initial sigma factor for a given indeterminacy.

Parameters:	**delta** (*float*) – The indeterminacy, m / n, of a system of equations of size m x n.
Returns:	**sigma_start** (*float*) – The initial sigma factor for the given indeterminacy.

magni.cs.reconstruction.sl0._modified.**_run_feas**(*y*, *A*)

Run the modified *feasibility* SL0 reconstruction algorithm.

This function implements the algorithm with a search on the feasible set.

Parameters:
- **y** (*ndarray*) – The m x 1 measurement vector.
- **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.

Returns: **alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

magni.cs.reconstruction.sl0._modified.**_run_proj**(*y*, *A*)

Run the original *projection* SL0 reconstruction algorithm.

This function implements the algorithm with an unconstrained gradient step followed by a projection back onto the feasible set.

Parameters:
- **y** (*ndarray*) – The m x 1 measurement vector.
- **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.

Returns: **alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

magni.cs.reconstruction.sl0._original module

Module providing the original SL0 reconstruction algorithm.

Routine listings

run(y, A)
  Run the original SL0 reconstruction algorithm.

See also:

magni.cs.reconstruction.sl0.config
  Configuration options.

Notes

The original SL0 reconstruction algorithm is described in [1] and [2].
  For delta < 0.55: Standard projection algorithm by Mohimani et. al [1]
  For delta >= 0.55: Standard constraint elimination algorithm by Cui et. al. [2]

References

[1] *(1, 2)* H. Mohimani, M. Babaie-Zadeh, and C. Jutten, “A Fast Approach for Overcomplete Sparse Decomposition Based on Smoothed l0 Norm”, *IEEE Transactions on Signal Processing*, vol. 57, no. 1, pp. 289-301, Jan. 2009.

[2] *(1, 2)* Z. Cui, H. Zhang, and W. Lu, “An Improved Smoothed l0-norm Algorithm Based on Multiparameter Approximation Function”, *in 12th IEEE International Conference on Communication Technology (ICCT)*, Nanjing, China, Nov. 11-14, 2011, pp. 942-945.

magni.cs.reconstruction.sl0._original.**run**(*y*, *A*)

Run the SL0 reconstruction algorithm.

Parameters:	• **y** (*ndarray*) – The m x 1 measurement vector. • **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.
Returns:	**alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

See also:

_run_proj()

The original projection algorithm.

_run_feas()

The original constraint elimination algorithm.

Examples

For example, recovering a vector from random measurements

```
>>> from magni.cs.reconstruction.sl0._original import run
>>> np.random.seed(seed=6021)
>>> A = 1 / np.sqrt(80) * np.random.randn(80, 200)
>>> x = np.zeros((200, 1))
>>> x[:10] = 1
>>> y = A.dot(x)
>>> x_hat = run(y, A)
>>> x_hat[:12]
array([[  9.99840757e-01],
       [  9.99849856e-01],
       [  9.99955438e-01],
       [  9.99966334e-01],
       [  1.00010956e+00],
       [  1.00000432e+00],
       [  9.99995701e-01],
       [  1.00016335e+00],
       [  9.99927317e-01],
       [  9.99841626e-01],
       [ -3.01131370e-05],
       [  4.10127956e-06]])
>>> (np.abs(x_hat) > 1e-2).sum()
10
```

magni.cs.reconstruction.sl0._original.**_run_feas**(*y*, *A*)

Run the original *feasibility* SL0 reconstruction algorithm.

This function implements the algorithm with a search on the feasible set.

Parameters:	• **y** (*ndarray*) – The m x 1 measurement vector. • **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.
Returns:	**alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

magni.cs.reconstruction.sl0._original.**_run_proj**(*y*, *A*)

Run the original *projection* SL0 reconstruction algorithm.

This function implements the algorithm with an unconstrained gradient step followed by a projection back onto the feasible set.

Parameters:
- **y** (*ndarray*) – The m x 1 measurement vector.
- **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.

Returns: **alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

magni.cs.reconstruction.sl0._util module

Module providing the public function of the magni.cs.reconstruction.sl0 subpackage.

magni.cs.reconstruction.sl0._util.**run**(*y*, *A*)

Run the specified SL0 reconstruction algorithm.

The available SL0 reconstruction algorithms are the original SL0 and the modified SL0. Which of the available SL0 reconstruction algorithms is used, is specified as a configuration option.

Parameters:
- **y** (*ndarray*) – The m x 1 measurement vector.
- **A** (*ndarray*) – The m x n matrix which is the product of the measurement matrix and the dictionary matrix.

Returns: **alpha** (*ndarray*) – The n x 1 reconstructed coefficient vector.

See also:

magni.cs.reconstruction.sl0.config()
Configuration options.

magni.cs.reconstruction.sl0._original.run()
The original SL0 reconstruction algorithm.

magni.cs.reconstruction.sl0._modified.run()
The modified SL0 reconstruction algorithm.

Examples

See the individual run functions in the implementations of the original and modified SL0 reconstruction algorithms.

magni.cs.reconstruction.sl0.config module

Module providing configuration options for the **magni.cs.reconstruction.sl0** subpackage.

Routine listings

get(key=None)
Get the value of one or more configuration options.

set(dictionary={}, **kwargs)
Set the value of one or more configuration options.

See also:

magni.utils.config
The Configger class used.

Notes

This module instantiates the *Configger* class provided by magni.utils.config and assigns handles for the get and set methods of that class instance. The configuration options are the following:

epsilon : *float*
: The precision parameter used in centering (the default is 0.01).

L : *float*
: The maximum number of gradient descent iterations for each sigma (the default is 2.0).

L_update : *float*
: The additive/multiplicative update of L when it is not fixed (the default is 2.0).

mu : *float*
: The relative step-size used in gradient descent iteration (the default is 1.0).

mu_start : *float*
: The relative step-size used in gradient descent iteration for the first iterations (the default is 0.001).

mu_end :float
: The relative step-size used in gradient descent iteration for the last iterations (the default is 1.5).

precision_float : *dtype*
: The floating point precision used for the computations (the default is float64).

seq_algorithm : *{'mod', 'std'}*
: The sequential implementation of SL0 used (the default is 'mod').

sigma_update : *float*
: The constant used in the geometric sequence of sigmas (the default is 0.7).

sigma_min : *float*
: The minimum value of std. dev. in Gaussian l0 approx (the default 0.01).

magni.cs.reconstruction.sl0.config.**get**(*key=None*)

Get the value of one or more configuration options.

This function wraps 'Configger.get' in order to convert any float options to the specified precision before returning the option(s).

> **See also:**
>
> magni.utils.config.Configger.get()
>
> The wrapped function.

magni.imaging package

Subpackage providing functionality for image manipulation.

Routine listings

dictionaries
    Module providing fast linear operations wrapped in matrix emulators.

domains
    Module providing a multi domain image class.

evaluation
    Module providing functions for evaluation of image reconstruction quality.

measurements
    Module providing functions for constructing scan patterns for measurements.

preprocessing
    Module providing functionality to remove tilt in images.

visualisation
    Module providing functionality for visualising images.

mat2vec(x)
    Function to reshape a matrix into vector by stacking columns.

vec2mat(x, mn_tuple)
    Function to reshape a vector into a matrix.

Notes

See _util for documentation of **mat2vec** and **vec2mat**.

Submodules

magni.imaging._fastops module

Module providing functionality related linear transformations.

Routine listings

dct2(x, m, n)
    2D discrete cosine transform.

idct2(x, m, n)
    2D inverse discrete cosine tranform.

dft2(x, m, n)
    2D discrete Fourier transform.

idft2(x, m, n)
    2D inverse discrete Fourier transform.

magni.imaging._fastops.**dct2**(*x*, *m*, *n*)

Apply the 2D Discrete Cosine Transform (DCT) to *x*.

x is assumed to be the column vector resulting from stacking the columns of the associated matrix which the transform is to be taken on.

Parameters:
- **x** (*ndarray*) – The m*n x 1 vector representing the associated column stacked matrix.
- **m** (*int*) – Number of rows in the associated matrix.
- **n** (*int*) – Number of columns in the associated matrix.

Returns: *ndarray* – A m*n x 1 vector of coefficients scaled such that x = idct2(dct2(x)).

See also:

scipy.fftpack.dct()
: 1D DCT

magni.imaging._fastops.**idct2**(*x*, *m*, *n*)

Apply the 2D Inverse Discrete Cosine Transform (iDCT) to *x*.

x is assumed to be the column vector resulting from stacking the columns of the associated matrix which the transform is to be taken on.

Parameters:
- **x** (*ndarray*) – The m*n x 1 vector representing the associated column stacked matrix.
- **m** (*int*) – Number of rows in the associated matrix.
- **n** (*int*) – Number of columns in the associated matrix.

Returns: *ndarray* – A m*n x 1 vector of coefficients scaled such that x = dct2(idct2(x)).

See also:

scipy.fftpack.idct()
: 1D inverse DCT

magni.imaging._fastops.**dft2**(*x*, *m*, *n*)

Apply the 2D Discrete Fourier Transform (DFT) to *x*.

x is assumed to be the column vector resulting from stacking the columns of the associated matrix which the transform is to be taken on.

Parameters:
- **x** (*ndarray*) – The m*n x 1 vector representing the associated column stacked matrix.
- **m** (*int*) – Number of rows in the associated matrix.
- **n** (*int*) – Number of columns in the associated matrix.

Returns: *ndarray* – A m*n x 1 vector of coefficients scaled such that x = dft2(idft2(x)).

See also:

numpy.fft.fft2()
: The underlying 2D FFT used to compute the 2D DFT.

magni.imaging._fastops.**idft2**(*x*, *m*, *n*)

Apply the 2D Inverse Discrete Fourier Transform (iDFT) to *x*.

x is assumed to be the column vector resulting from stacking the columns of the associated matrix which the transform is to be taken on.

Parameters:
- **x** (*ndarray*) – The m*n x 1 vector representing the associated column stacked matrix.
- **m** (*int*) – Number of rows in the associated matrix.
- **n** (*int*) – Number of columns in the associated matrix.

Returns: *ndarray* – A m*n x 1 vector of coefficients scaled such that x = idft2(dft2(x)).

See also:

numpy.fft.ifft2()
: The underlying 2D iFFT used to compute the 2D iDFT.

magni.imaging._util module

Module providing the public functions of the magni.imaging subpackage.

magni.imaging._util.**mat2vec**(*x*)

Reshape *x* from matrix to vector by stacking columns.

Parameters: **x** (*ndarray*) – Matrix that should be reshaped to vector.

Returns: *ndarray* – Column vector formed by stacking the columns of the matrix *x*.

See also:

vec2mat()
: The inverse operation

Notes

The returned column vector is C contiguous.

Examples

For example,

```
>>> from magni.imaging._util import mat2vec
>>> x = np.arange(4).reshape(2, 2)
>>> x
array([[0, 1],
       [2, 3]])
>>> mat2vec(x)
array([[0],
       [2],
       [1],
       [3]])
```

magni.imaging._util.**vec2mat**(*x*, *mn_tuple*)

Reshape *x* from column vector to matrix.

Parameters:
- **x** (*ndarray*) – Matrix that should be reshaped to vector.
- **mn_tuple** (*tuple*) – A tuple (m, n) containing the parameters m, n as listed below.
- **m** (*int*) – Number of rows in the resulting matrix.
- **n** (*int*) – Number of columns in the resulting matrix.

Returns: *ndarray* – Matrix formed by taking *n* columns of lenght *m* from the column vector *x*.

See also:

mat2vec()
The inverse operation

Notes

The returned matrix is C contiguous.

Examples

For example,

```
>>> from magni.imaging._util import vec2mat
>>> x = np.arange(4).reshape(4, 1)
>>> x
array([[0],
       [1],
       [2],
       [3]])
>>> vec2mat(x, (2, 2))
array([[0, 2],
       [1, 3]])
```

magni.imaging.dictionaries module

Module providing fast linear operations wrapped in matrix emulators.

Routine listings

get_DCT(shape)
Get the DCT fast operation dictionary for the given image shape.

get_DFT(shape)
Get the DFT fast operation dictionary for the given image shape.

See also:

magni.imaging._fastops
Fast linear operations.

magni.utils.matrices
Matrix emulators.

magni.imaging.dictionaries.**get_DCT**(*shape*)

Get the DCT fast operation dictionary for the given image shape.

Parameters: **shape** (*list or tuple*) – The shape of the image which the dictionary is the DCT dictionary.

Returns: **matrix** (*magni.utils.matrices.Matrix*) – The specified DCT dictionary.

See also:

magni.utils.matrices.Matrix()
The matrix emulator class.

Examples

Create a dummy image:

```
>>> img = np.random.randn(64, 64)
>>> vec = magni.imaging.mat2vec(img)
```

Perform DCT in the ordinary way:

```
>>> dct_normal = magni.imaging._fastops.dct2(vec, *img.shape)
```

Perform DCT using the present function:

```
>>> from magni.imaging.dictionaries import get_DCT
>>> matrix = get_DCT(img.shape)
>>> dct_matrix = matrix.T.dot(vec)
```

Check that the two ways produce the same result:

```
>>> np.allclose(dct_matrix, dct_normal)
True
```

magni.imaging.dictionaries.**get_DFT**(*shape*)

Get the DFT fast operation dictionary for the given image shape.

Parameters: **shape** (*list or tuple*) – The shape of the image which the dictionary is the DFT dictionary.

Returns: **matrix** (*magni.utils.matrices.Matrix*) – The specified DFT dictionary.

See also:

magni.utils.matrices.Matrix()
The matrix emulator class.

Examples

Create a dummy image:

```
>>> img = np.random.randn(64, 64)
>>> vec = magni.imaging.mat2vec(img)
```

Perform DFT in the ordinary way:

```
>>> dft_normal = magni.imaging._fastops.dft2(vec, *img.shape)
```

Perform DFT using the present function:

```
>>> from magni.imaging.dictionaries import get_DFT
>>> matrix = get_DFT(img.shape)
>>> dft_matrix = matrix.T.dot(vec)
```

Check that the two ways produce the same result:

```
>>> np.allclose(dft_matrix, dft_normal)
True
```

magni.imaging.domains module

Module providing a multi domain image class.

Routine listings

MultiDomainImage(object)
: Provide access to an image in the domains of a compressed sensing context.

class magni.imaging.domains.**MultiDomainImage**(*Phi*, *Psi*)

Bases: **object**

Provide access to an image in the domains of a compressed sensing context.

Given a measurement matrix and a dictionary, an image can be supplied in either the measurement domain, the image domain, or the coefficient domain. This class then provides access to the image in all three domains.

Parameters:

- **Phi** (*magni.utils.matrices.Matrix, magni.utils.matrices.MatrixCollection,*) – or numpy.ndarray The measurement matrix.
- **Psi** (*magni.utils.matrices.Matrix, magni.utils.matrices.MatrixCollection,*) – or numpy.ndarray The dictionary.

Notes

The image is only converted to other domains than the supplied when the the image is requested in another domain. The image is, however, stored in up to three versions internally in order to reduce computation overhead. This may introduce a memory overhead.

Examples

Define a measurement matrix which skips every other sample:

```
>>> func = lambda vec: vec[::2]
>>> func_T = lambda vec: np.float64([vec[0], 0, vec[1]]).reshape(3, 1)
>>> Phi = magni.utils.matrices.Matrix(func, func_T, (), (2, 3))
```

Define a dictionary which is simply a rotated identity matrix:

```
>>> v = np.sqrt(0.5)
>>> Psi = np.float64([[ v, -v,  0],
...                   [ v,  v,  0],
...                   [ 0,  0,  1]])
```

Instantiate the current class to handle domains:

```
>>> from magni.imaging.domains import MultiDomainImage
>>> domains = MultiDomainImage(Phi, Psi)
```

An image can the be supplied in any domain and likewise retrieved in any domain. For example, the image:

```
>>> domains.image = np.ones(3).reshape(3, 1)
```

Can be retrieved both as measurements:

```
>>> domains.measurements
array([[ 1.],
       [ 1.]])
```

And as coefficients:

```
>>> domains.coefficients
array([[ 1.41421356],
       [ 0.        ],
       [ 1.        ]])
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*Phi*, *Psi*)

_validate_coefficients(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

coefficients

Get the image in the coefficient domain.

Returns:	**coefficients** (*numpy.ndarray*) – The dictionary coefficients of the image.

_validate_image(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

image

Get the image in the image domain.

Returns:	**image** (*numpy.ndarray*) – The image.

_validate_measurements(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

measurements

Get the image in the measurement domain.

Returns:	**measurements** (*numpy.ndarray*) – The measurements of the image.

magni.imaging.evaluation module

Module providing functions for evaluation of image reconstruction quality.

Routine listings

calculate_mse(x_org, x_recons)
: Function to calcualte Mean Squared Error (MSE).

calculate_psnr(x_org, x_recons, peak)
: Function to calculate Peak Signal to Noise Ratio (PSNR).

calculate_retained_energy(x_org, x_recons)
: Function to calculate the percentage of energy retained in reconstruction.

magni.imaging.evaluation.**calculate_mse**(*x_org*, *x_recons*)

Calculate Mean Squared Error (MSE) between *x_recons* and *x_org*.

Parameters:	• **x_org** (*ndarray*) – Array of original values. • **x_recons** (*ndarray*) – Array of reconstruction values.
Returns:	**mse** (*float*) – Mean Squared Error (MSE).

Notes

The Mean Squared Error (MSE) is calculated as:

$$\frac{1}{N} \cdot \sum (x_{org} - x_{recons})^2$$

where N is the number of entries in *x_org*.

Examples

For example,

```
>>> from magni.imaging.evaluation import calculate_mse
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_recons = np.ones((2,2))
>>> calculate_mse(x_org, x_recons)
1.5
```

magni.imaging.evaluation.**calculate_psnr**(*x_org*, *x_recons*, *peak*)

Calculate Peak Signal to Noise Ratio (PSNR) between *x_recons* and *x_org*.

Parameters:	• **x_org** (*ndarray*) – Array of original values. • **x_recons** (*ndarray*) – Array of reconstruction values. • **peak** (*int or float*) – Peak value.
Returns:	**psnr** (*float*) – Peak Signal to Noise Ratio (PSNR) in dB.

Notes

The PSNR is as calculated as

$$10 \cdot \log_{10}\left(\frac{peak^2}{1/N \cdot \sum(x_{org} - x_{recons})^2}\right)$$

where *N* is the number of entries in *x_org*.

Examples

For example,

```
>>> from magni.imaging.evaluation import calculate_psnr
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_recons = np.ones((2,2))
>>> peak = 3
>>> calculate_psnr(x_org, x_recons, peak)
7.7815125038364368
```

magni.imaging.evaluation.**calculate_retained_energy**(*x_org*, *x_recons*)

Calculate percentage of energy retained in reconstruction.

Parameters:	• **x_org** (*ndarray*) – Array of original values. • **x_recons** (*ndarray*) – Array of reconstruction values.
Returns:	**energy** (*float*) – Percentage of retained energy in reconstruction.

Notes

The retained energy is as calculated as

$$\frac{\sum x_{recons}^2}{\sum x_{org}^2} \cdot 100\%$$

Examples

For example,

```
>>> from magni.imaging.evaluation import calculate_retained_energy
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_recons = np.ones((2,2))
>>> calculate_retained_energy(x_org, x_recons)
28.571428571428569
```

magni.imaging.measurements module

Module providing functions for constructing scan patterns for measurements.

This module provides several pairs of scan pattern functions. The first function, named *_sample_surface, is used for sampling a given surface. The second function, named *_sample_image, is a wrapper that provides a pixel-oriented interface to the first function. In a addition to these pairs of scan pattern functions, the module provides auxillary functions that may be used to visualise the scan patterns.

Routine listings

construct_measurement_matrix(coords, h, w)
    Function for constructing a measurement matrix.

plot_pattern(l, w, coords, mode, output_path=None)
    Function for visualising a scan pattern.

plot_pixel_mask(h, w, pixels, output_path=None)
    Function for visualising a pixel mask obtained from a scan pattern.

random_line_sample_image(h, w, scan_length, num_points, discrete=None,
    seed=None) Function for random line sampling an image.

random_line_sample_surface(l, w, speed, sample_rate, time, discrete=None,
    seed=None) Function for random line sampling a surface.

spiral_sample_image(h, w, scan_length, num_points)
    Function for spiral sampling an image.

spiral_sample_surface(l, w, speed, sample_rate, time)
    Function for spiral sampling a surface.

square_spiral_sample_image(h, w, scan_length, num_points)
    Function for square spiral sampling an image.

square_spiral_sample_surface(l, w, speed, sample_rate, time)
    Function for square spiral sampling a surface.

uniform_line_sample_image(h, w, scan_length, num_points)
    Function for uniform line sampling an image.

uniform_line_sample_surface(l, w, speed, sample_rate, time)
    Function for uniform line sampling a surface.

unique_pixels(coords)
    Function for determining unique pixels from a set of coordinates.

Notes

In principle, most of the scan pattern related parameters need only be positive.

However, it is assumed that the following requirements are fulfilled:

Minimum length of scan area:
1 nm

Minimum width of scan area:
1 nm

Minimum scan speed:
1 nm/s

Minimum sample_rate:
1 Hz

Minimum scan time:
1 s

Minimum scan length:
1 nm

Minimum number of scan points:
1

Examples

Sample a surface using a spiral pattern:

```
>>> from magni.imaging.measurements import spiral_sample_surface
>>> l = 13.0; w = 13.0; speed = 4.0; time = 27.0; sample_rate = 3.0;
>>> coords = spiral_sample_surface(l, w, speed, sample_rate, time)
```

Display the resulting pattern:

```
>>> from magni.imaging.measurements import plot_pattern
>>> plot_pattern(l, w, coords, 'surface')
```

Sample a 128x128 pixel image using random lines and a fixed seed:

```
>>> from magni.imaging.measurements import random_line_sample_image
>>> h = 128; w = 128; scan_length = 1000.0; num_points = 200; seed=6021;
>>> coords = random_line_sample_image(h, w, scan_length, num_points, seed=seed)
```

Display the resulting pattern:

```
>>> plot_pattern(h, w, coords, 'image')
```

Find the corresponding unique pixels and plot the pixel mask:

```
>>> from magni.imaging.measurements import unique_pixels, plot_pixel_mask
>>> unique_pixels = unique_pixels(coords)
>>> plot_pixel_mask(h, w, unique_pixels)
```

magni.imaging.measurements.**construct_measurement_matrix**(*coords*, *h*, *w*)

Construct a measurement matrix extracting the specified measurements.

Parameters:	**coords** (*ndarray*) – The *k* floating point coordinates arranged into a 2D array where each row is a coordinate pair (x, y), such that *coords* has size *k* x 2.
Returns:	**Phi** (*magni.utils.matrices.Matrix*) – The constructed measurement matrix.

See also:

magni.utils.matrices.Matrix()
The matrix emulator class.

Notes

The function construct two functions: one for extracting pixels at the coordinates specified and one for the transposed operation. These functions are then wrapped by a matrix emulator which is returned.

Examples

Create a dummy 5 by 5 pixel image and an example sampling pattern:

```
>>> img = np.arange(25, dtype=np.float).reshape(5, 5)
>>> vec = magni.imaging.mat2vec(img)
>>> coords = magni.imaging.measurements.uniform_line_sample_image(
...             5, 5, 16., 17)
```

Sample the image in the ordinary way:

```
>>> unique = magni.imaging.measurements.unique_pixels(coords)
>>> samples_normal = img[unique[:, 1], unique[:, 0]]
>>> samples_normal = samples_normal.reshape((len(unique), 1))
```

Sample the image using the present function:

```
>>> from magni.imaging.measurements import construct_measurement_matrix
>>> matrix = construct_measurement_matrix(coords, *img.shape)
>>> samples_matrix = matrix.dot(vec)
```

Check that the two ways produce the same result:

```
>>> np.allclose(samples_matrix, samples_normal)
True
```

magni.imaging.measurements.**plot_pattern**(*l*, *w*, *coords*, *mode*, *output_path=None*)

Display a plot that shows the pattern given by a set of coordinates.

The pattern given by the *coords* is displayed on an *w* x *l* area. If *mode* is 'surface', *l* and *w* are regarded as measured in meters. If *mode* is 'image', *l* and *w* are regarded as measured in pixels. The *coords* are marked by filled circles and connected by straight dashed lines.

Parameters:
- **l** (*float or int*) – The length/height of the area. If *mode* is 'surface', it must be a float. If *mode* is 'image', it must be an integer.
- **w** (*float or int*) – The width of the area. If *mode* is 'surface', it must be a float. If *mode* is 'image', it must be an integer.
- **coords** (*ndarray*) – The 2D array of pixels that make up the mask. Each row is a coordinate pair (x, y).
- **mode** (*{'surface', 'image'}*) – The display mode that dertermines the axis labeling and the type of *l* and *w*.
- **output_path** (*str, optional*) – Path (including file type extension) under which the plot is saved (the default value is None which implies, that the plot is not saved).

Notes

The resulting plot is displayed in a figure using **matplotlib**'s *pyplot.plot*.

Examples

For example,

```
>>> from magni.imaging.measurements import plot_pattern
>>> l = 3
>>> w = 3
>>> coords = np.array([[0, 0], [1, 1], [2, 1]], dtype=np.float)
>>> mode = 'image'
>>> plot_pattern(l, w, coords, mode)
```

magni.imaging.measurements.**plot_pixel_mask**(*h*, *w*, *pixels*, *output_path=None*)

Display a binary image that shows the given pixel mask.

A black image with *w* x *h* pixels is created and the *pixels* are marked with white.

Parameters:
- **h** (*int*) – The height of the image in pixels.
- **w** (*int*) – The width of the image in pixels.
- **pixels** (*ndarray*) – The 2D array of pixels that make up the mask. Each row is a coordinate pair (x, y), such that *coords* has size len(*pixels*) x 2.
- **output_path** (*str, optional*) – Path (including file type extension) under which the plot is saved (the default value is None which implies, that the plot is not saved).

Notes

The resulting image is displayed in a figure using **magni.imaging.visualisation.imshow**.

Examples

For example,

```
>>> from magni.imaging.measurements import plot_pixel_mask
>>> h = 3
>>> w = 3
>>> pixels = np.array([[0, 0], [1, 1], [2, 1]])
>>> plot_pixel_mask(h, w, pixels)
```

magni.imaging.measurements.**random_line_sample_image**(*h*, *w*, *scan_length*, *num_points*, *discrete=None*, *seed=None*)

Sample an image using a set of random straight lines.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using a pattern based on a set of random straight lines are determined. The *scan_length* determines the length of the path scanned whereas *num_points* indicates the number of samples taken on that path. If *discrete* is set, it specifies the finite number of equally spaced lines from which the scan lines are be chosen at random. For reproducible results, the *seed* may be used to specify a fixed seed of the random number generator.

Parameters:

- **h** (*int*) – The height of the area to scan in units of pixels.
- **w** (*int*) – The width of the area to scan in units of pixels.
- **scan_length** (*float*) – The length of the path to scan in units of pixels.
- **num_points** (*int*) – The number of samples to take on the scanned path.
- **discrete** (*int or None, optional*) – The number of equally spaced lines from which the scan lines are chosen (the default is None, which implies that no discritisation is used).
- **seed** (*int or None, optional*) – The seed used for the random number generator (the defaul is None, which implies that the random number generator is not seeded).

Returns: **coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the height *h* is measured along the y-axis.

Each of the scanned lines span the entire width of the image with the exception of the last line that may only be partially scanned if the *scan_length* implies this. The top and bottom lines of the image are always included in the scan.

Examples

For example,

```
>>> from magni.imaging.measurements import random_line_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> seed = 6021
>>> random_line_sample_image(h, w, scan_length, num_points, seed=seed)
array([[ 0.5       ,  0.5       ],
       [ 5.04545455,  0.5       ],
       [ 9.5       ,  0.59090909],
       [ 5.65110302,  1.28746666],
       [ 1.10564847,  1.28746666],
       [ 0.5       ,  5.22727273],
       [ 3.29121172,  6.98151556],
       [ 7.83666626,  6.98151556],
       [ 7.23606783,  7.59970419],
       [ 2.69061328,  7.59970419],
       [ 0.95454545,  9.5       ],
       [ 5.5       ,  9.5       ]])
```

magni.imaging.measurements.**random_line_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*, *discrete=None*, *seed=None*)

Sample a surface area using a set of random straight lines.

The coordinates (in units of meters) resulting from sampling an image of size *l* times *w* using a pattern based on a set of random straight lines are determined. The scanned path is determined from the probe *speed* and the scan `time`. If *discrete* is set, it specifies the finite number of equally spaced lines from which the scan lines are be chosen at random. For reproducible results, the *seed* may be used to specify a fixed seed of the random number generator.

Parameters:
- **l** (*float*) – The length of the area to scan in units of meters.
- **w** (*float*) – The width of the area to scan in units of meters.
- **speed** (*float*) – The probe speed in units of meters/second.
- **sample_rate** (*float*) – The sample rate in units of Hertz.
- **time** (*float*) – The scan time in units of seconds.
- **discrete** (*int or None, optional*) – The number of equally spaced lines from which the scan lines are chosen (the default is None, which implies that no discritisation is used).
- **seed** (*int or None, optional*) – The seed used for the random number generator (the defaul is None, which implies that the random number generator is not seeded).

Returns: **coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the length *l* is measured along the y-axis.

Each of the scanned lines span the entire width of the image with the exception of the last line that may only be partially scanned if the *speed* and `time` implies this. The top and bottom lines of the image are always included in the scan and are not included in the *discrete* number of lines.

Examples

For example,

```
>>> from magni.imaging.measurements import random_line_sample_surface
>>> l = 2e-6
>>> w = 2e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> seed = 6021
>>> random_line_sample_surface(l, w, speed, sample_rate, time, seed=seed)
array([[  0.00000000e+00,   0.00000000e+00],
       [  7.00000000e-07,   0.00000000e+00],
       [  1.40000000e-06,   0.00000000e+00],
       [  2.00000000e-06,   1.00000000e-07],
       [  1.37499259e-06,   1.74992590e-07],
       [  6.74992590e-07,   1.74992590e-07],
       [  0.00000000e+00,   2.00000000e-07],
       [  0.00000000e+00,   9.00000000e-07],
       [  2.22879572e-08,   1.57771204e-06],
       [  7.22287957e-07,   1.57771204e-06],
       [  1.42228796e-06,   1.57771204e-06],
       [  2.00000000e-06,   1.70000000e-06],
       [  1.60000000e-06,   2.00000000e-06]])
```

magni.imaging.measurements.**spiral_sample_image**(*h*, *w*, *scan_length*, *num_points*)

Sample an image using an archimedean spiral pattern.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using an archimedean spiral pattern are determined. The *scan_length* determines the length of the path scanned whereas *num_points* indicates the number of samples taken on that path.

Parameters:	• **h** (*int*) – The height of the area to scan in units of pixels. • **w** (*int*) – The width of the area to scan in units of pixels. • **scan_length** (*float*) – The length of the path to scan in units of pixels. • **num_points** (*int*) – The number of samples to take on the scanned path.
Returns:	**coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the height *h* is measured along the y-axis. The width must equal the height for an archimedian spiral to make sense.

Examples

For example,

```
>>> from magni.imaging.measurements import spiral_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> spiral_sample_image(h, w, scan_length, num_points)
array([[ 6.28776846,  5.17074073],
       [ 3.13304898,  5.24133767],
       [ 6.07293751,  2.93873701],
       [ 6.99638041,  6.80851189],
       [ 2.89868434,  7.16724999],
       [ 2.35773914,  3.00320067],
       [ 6.41495385,  1.71018152],
       [ 8.82168896,  5.27557847],
       [ 6.34932919,  8.83624957],
       [ 2.04885699,  8.11199373],
       [ 0.6196052 ,  3.96939755]])
```

magni.imaging.measurements.**spiral_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*)

Sample a surface area using an archimedean spiral pattern.

The coordinates (in units of meters) resulting from sampling an area of size *l* times *w* using an archimedean spiral pattern are determined. The scanned path is determined from the probe *speed* and the scan `time`.

Parameters:	• **l** (*float*) – The length of the area to scan in units of meters. • **w** (*float*) – The width of the area to scan in units of meters. • **speed** (*float*) – The probe speed in units of meters/second. • **sample_rate** (*float*) – The sample rate in units of Hertz. • **time** (*float*) – The scan time in units of seconds.
Returns:	**coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the length *l* is measured along the y-axis. The width must equal the length for an archimedian sprial to make sense.

Examples

For example,

```
>>> from magni.imaging.measurements import spiral_sample_surface
>>> l = 1e-6
>>> w = 1e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> spiral_sample_surface(l, w, speed, sample_rate, time)
array([[  3.61074393e-07,   4.60846340e-07],
       [  5.16981905e-07,   7.08474495e-07],
       [  5.18627260e-07,   2.42473901e-07],
       [  5.90470818e-07,   7.85303986e-07],
       [  2.13777354e-07,   3.25138348e-07],
       [  8.38392713e-07,   3.55407335e-07],
       [  4.85640136e-07,   8.97652018e-07],
       [  9.79367276e-08,   3.60017412e-07],
       [  7.24145327e-07,   1.07666546e-07],
       [  8.91388500e-07,   7.71842556e-07],
       [  2.12224761e-07,   9.08883127e-07]])
```

magni.imaging.measurements.**square_spiral_sample_image**(*h*, *w*, *scan_length*, *num_points*)

Sample an image using a square spiral pattern.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using a square spiral pattern are determined. The *scan_length* determines the length of the path scanned whereas *num_points* indicates the number of samples taken on that path.

Parameters:
- **h** (*int*) – The height of the area to scan in units of pixels.
- **w** (*int*) – The width of the area to scan in units of pixels.
- **scan_length** (*float*) – The length of the path to scan in units of pixels.
- **num_points** (*int*) – The number of samples to take on the scanned path.

Returns: **coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the height *h* is measured along the y-axis.

Examples

For example,

```
>>> from magni.imaging.measurements import square_spiral_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> square_spiral_sample_image(h, w, scan_length, num_points)
array([[ 5.        ,  5.        ],
       [ 6.28571429,  5.97619048],
       [ 4.38095238,  3.71428571],
       [ 2.42857143,  5.92857143],
       [ 4.95238095,  7.57142857],
       [ 7.57142857,  6.02380952],
       [ 7.        ,  2.42857143],
       [ 2.83333333,  2.42857143],
       [ 1.14285714,  4.9047619 ],
       [ 1.35714286,  8.85714286],
       [ 5.52380952,  8.85714286],
       [ 8.85714286,  8.02380952]])
```

magni.imaging.measurements.**square_spiral_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*)

Sample a surface area using a square spiral pattern.

The coordinates (in units of meters) resulting from sampling an area of size *l* times *w* using a square spiral pattern are determined. The scanned path is determined from the probe *speed* and the scan **time**.

Parameters:
- **l** (*float*) – The length of the area to scan in units of meters.
- **w** (*float*) – The width of the area to scan in units of meters.
- **speed** (*float*) – The probe speed in units of meters/second.
- **sample_rate** (*float*) – The sample rate in units of Hertz.
- **time** (*float*) – The scan time in units of seconds.

Returns: **coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the length *l* is measured along the y-axis.

Examples

For example,

```
>>> from magni.imaging.measurements import square_spiral_sample_surface
>>> l = 1e-6
>>> w = 1e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> square_spiral_sample_surface(l, w, speed, sample_rate, time)
array([[  5.00000000e-07,   5.00000000e-07],
       [  4.00000000e-07,   4.00000000e-07],
       [  6.00000000e-07,   7.00000000e-07],
       [  5.00000000e-07,   3.00000000e-07],
       [  2.00000000e-07,   7.00000000e-07],
       [  8.00000000e-07,   7.00000000e-07],
       [  6.00000000e-07,   2.00000000e-07],
       [  1.00000000e-07,   4.00000000e-07],
       [  3.00000000e-07,   9.00000000e-07],
       [  9.00000000e-07,   8.00000000e-07],
       [  9.00000000e-07,   1.00000000e-07],
       [  2.00000000e-07,   1.00000000e-07]])
```

magni.imaging.measurements.**uniform_line_sample_image**(*h*, *w*, *scan_length*, *num_points*)

Sample an image using a set of uniformly distributed straight lines.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using a pattern based on a set of uniformly distributed straight lines are determined. The *scan_length* determines the length of the path scanned whereas *num_points* indicates the number of samples taken on that path.

Parameters:

- **h** (*int*) – The height of the area to scan in units of pixels.
- **w** (*int*) – The width of the area to scan in units of pixels.
- **scan_length** (*float*) – The length of the path to scan in units of pixels.
- **num_points** (*int*) – The number of samples to take on the scanned path.

Returns: **coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the height *h* is measured along the y-axis.

Each of the scanned lines span the entire width of the image with the exception of the last line that may only be partially scanned if the *scan_length* implies this. The top and bottom lines of the image are always included in the scan.

Examples

For example,

```
>>> from magni.imaging.measurements import uniform_line_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> uniform_line_sample_image(h, w, scan_length, num_points)
array([[ 0.5       ,  0.5       ],
       [ 5.04545455,  0.5       ],
       [ 9.5       ,  0.59090909],
       [ 7.11363636,  2.75      ],
       [ 2.56818182,  2.75      ],
       [ 0.72727273,  5.        ],
       [ 5.27272727,  5.        ],
       [ 9.5       ,  5.31818182],
       [ 6.88636364,  7.25      ],
       [ 2.34090909,  7.25      ],
       [ 0.95454545,  9.5       ],
       [ 5.5       ,  9.5       ]])
```

magni.imaging.measurements.**uniform_line_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*)

Sample aa surface area using a set of uniformly distributed straight lines.

The coordinates (in units of meters) resulting from sampling an area of size *l* times *w* using a pattern based on a set of uniformly distributed straight lines are determined. The scanned path is determined from the probe *speed* and the scan `time`.

Parameters:	• **l** (*float*) – The length of the area to scan in units of meters. • **w** (*float*) – The width of the area to scan in units of meters. • **speed** (*float*) – The probe speed in units of meters/second. • **sample_rate** (*float*) – The sample rate in units of Hertz. • **time** (*float*) – The scan time in units of seconds.
Returns:	**coords** (*ndarray*) – The coordinates of the samples arranged into a 2D array, such that each row is a coordinate pair (x, y).

Notes

The orientation of the coordinate system is such that the width *w* is measured along the x-axis whereas the height *l* is measured along the y-axis.

Each of the scanned lines span the entire width of the image with the exception of the last line that may only be partially scanned if the *scan_length* implies this. The top and bottom lines of the image are always included in the scan.

Examples

For example,

```
>>> from magni.imaging.measurements import uniform_line_sample_surface
>>> l = 2e-6
>>> w = 2e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> uniform_line_sample_surface(l, w, speed, sample_rate, time)
array([[  0.00000000e+00,   0.00000000e+00],
       [  7.00000000e-07,   0.00000000e+00],
       [  1.40000000e-06,   0.00000000e+00],
       [  2.00000000e-06,   1.00000000e-07],
       [  1.86666667e-06,   6.66666667e-07],
       [  1.16666667e-06,   6.66666667e-07],
       [  4.66666667e-07,   6.66666667e-07],
       [  0.00000000e+00,   9.00000000e-07],
       [  2.66666667e-07,   1.33333333e-06],
       [  9.66666667e-07,   1.33333333e-06],
       [  1.66666667e-06,   1.33333333e-06],
       [  2.00000000e-06,   1.70000000e-06],
       [  1.60000000e-06,   2.00000000e-06]])
```

magni.imaging.measurements.**unique_pixels**(*coords*)

Identify unique pixels from a set of coordinates.

The floating point *coords* are reduced to a unique set of integer pixels by flooring the floating point values.

Parameters:	**coords** (*ndarray*) – The *k* floating point coordinates arranged into a 2D array where each row is a coordinate pair (x, y), such that *coords* has size *k* x 2.
Returns:	**unique_pixels** (*ndarray*) – The *l* <= *k* unique (integer) pixels, such that `unique_pixels` is a 2D array and has size *l* x 2.

Examples

For example,

```
>>> from magni.imaging.measurements import unique_pixels
>>> coords = np.array([[1.7, 1.0], [1.0, 1.2], [3.3, 4.3]])
>>> unique_pixels(coords)
array([[1, 1],
       [3, 4]])
```

magni.imaging.measurements.**_get_line_scan_coords**(*lines*, *samples*, *sample_dist*, *l*, *w*)

Determine the coordinates of the sampled lines in a line scanning.

Parameters:	• **lines** (*ndarray*) – The position (vertical distance from top of scan area) of the lines to scan. • **samples** (*int*) – The number of samples on the scan path. • **sample_dist** (*float*) – The distance (path length) between samples. • **l** (*float*) – The length of the area to scan in units of meters. • **w** (*float*) – The width of the area to scan in units of meters.
Returns:	**coords** (*ndarray*) – The w and l coordinates arranged into a N_samples-by-2 array.

magni.imaging.preprocessing module

Module providing functionality to remove tilt in images.

Routine listings

detilt(img, mask=None, mode='plane_flatten', degree=1, return_tilt=False)
Function to remove tilt from an image.

magni.imaging.preprocessing.**detilt**(*img*, *mask=None*, *mode='plane_flatten'*, *degree=1*, *return_tilt=False*)

Estimate the tilt in an image and return the detilted image.

Parameters:
- **img** (*ndarray*) – The image that is to be detilted.
- **mask** (*ndarray, optional*) – Bool array of the same size as *img* indicating the pixels to use in detilt (the default is None, which implies, that the the entire image is used)
- **mode** (*{'line_flatten', 'plane_flatten'}, optional*) – The type of detilting applied (the default is plane_flatten).
- **degree** (*int, optional*) – The degree of the polynomial used in line flattening (the default is 1).
- **return_tilt** (*bool, optional*) – If True, the detilted image and the estimated tilt is returned (the default is False).

Returns:
- **img_detilt** (*ndarray*) – Detilted image.
- **tilt** (*ndarray, optional*) – The estimated tilt (image). Only returned if return_tilt is True.

Notes

If *mode* is line flatten, the tilt in each horizontal line of pixels in the image is estimated by a polynomial fit independently of all other lines. If *mode* is plane flatten, the tilt is estimated by fitting a plane to all pixels.

If a custom *mask* is specified, only the masked (True) pixels are used in the estimation of the tilt.

Examples

For example, line flatten an image using a degree 1 polynomial

```
>>> from magni.imaging.preprocessing import detilt
>>> img = np.array([[0, 2, 3], [1, 5, 7], [3, 6, 8]], dtype=np.float)
>>> detilt(img, mode='line_flatten', degree=1)
array([[-0.16666667,  0.33333333, -0.16666667],
       [-0.33333333,  0.66666667, -0.33333333],
       [-0.16666667,  0.33333333, -0.16666667]])
```

Or plane flatten the image based on a mask and return the tilt

```
>>> mask = np.array([[1, 1, 0], [1, 0, 1], [0, 1, 1]], dtype=np.bool)
>>> im, ti = detilt(img, mask=mask, mode='plane_flatten', return_tilt=True)
>>> im
array([[  3.33333333e-01,  -3.33333333e-01,  -2.00000000e+00],
       [ -3.33333333e-01,   1.00000000e+00,   3.33333333e-01],
       [ -2.66453526e-15,   3.33333333e-01,  -3.33333333e-01]])
>>> ti
array([[-0.33333333,  2.33333333,  5.        ],
       [ 1.33333333,  4.        ,  6.66666667],
       [ 3.        ,  5.66666667,  8.33333333]])
```

magni.imaging.preprocessing.**_line_flatten_tilt**(*img*, *mask*, *degree*)

Estimate tilt using the line flatten method.

Parameters:	• **img** (*ndarray*) – The image from which the tilt is estimated. • **mask** (*ndarray, or None*) – If not None, a bool ndarray of the the shape as *img* indicating which pixels should be used in estimate of tilt. • **degree** (*int*) – The degree of the polynomial in the estimated line tilt.
Returns:	**tilt** (*ndarray*) – The estimated tilt.

magni.imaging.preprocessing.**_plane_flatten_tilt**(*img*, *mask*)

Estimate tilt using the plane flatten method.

Parameters:	• **img** (*ndarray*) – The image from which the tilt is estimated. • **mask** (*ndarray, or None*) – If not None, a bool ndarray of the the shape as *img* indicating which pixels should be used in estimate of tilt.
Returns:	**tilt** (*ndarray*) – The estimated tilt.

magni.imaging.visualisation module

Module providing functionality for visualising images.

The module provides functionality for adjusting the intensity of an image. Furthermore, it provides a wrapper of the **matplotlib.pyplot.imshow** function that may exploit the provided functions for adjusting the image intensity.

Routine listings

imshow(X, ax=None, intensity_func=None, intensity_args=(), **kwargs)
: Function that may be used to display an image.

shift_mean(x_mod, x_org)
: Function for shifting mean intensity of an image based on another image.

stretch_image(img, max_val)
: Function for stretching the intensity of an image.

magni.imaging.visualisation.**imshow**(*X*, *ax=None*, *intensity_func=None*, *intensity_args=()*, *show_axis='none'*, ***kwargs*)

Display an image.

Wrap **matplotlib.pyplot.imshow** to display a possibly intensity manipulated version of the image *X*.

Parameters:
- **X** (*ndarray*) – The image to be displayed.
- **ax** (*matplotlib.axes.Axes, optional*) – The axes on which the image is displayed (the default is None, which implies that the current axes is used).
- **intensity_func** (*FunctionType, optional*) – The handle to the function used to manipulate the image intensity before the image is displayed (the default is None, which implies that no intensity manipulation is used).
- **intensity_args** (*list or tuple, optional*) – The arguments that are passed to the *intensity_func* (the default is (), which implies that no arguments are passed).
- **show_axis** (*{'none', 'top', 'inherit'}*) – How the x- and y-axis are display. If 'none', no axis are displayed. If 'top', the x-axis is displayed at the top of the image. If 'inherit', the axis display is inherited from `matplotlib.pyplot.imshow`.

Returns: **im_out** (*matplotlib.image.AxesImage*) – The AxesImage returned by matplotlibs imshow.

See also:

`matplotlib.pyplot.imshow()`
: Matplotlib's imshow function.

Examples

For example,

```
>>> from magni.imaging.visualisation import imshow
>>> X = np.arange(4).reshape(2, 2)
>>> add_k = lambda X, k: X + k
>>> im_out = imshow(X, intensity_func=add_k, intensity_args=(2,))
```

magni.imaging.visualisation.**shift_mean**(*x_mod*, *x_org*)

Shift the mean value of *x_mod* such that it equals the mean of *x_org*.

Parameters:
- **x_org** (*ndarray*) – The array which hold the "true" mean value.
- **x_mod** (*ndarray*) – The modified copy of *x_org* which must have its mean value shifted.

Returns: **shifted_x_mod** (*ndarray*) – A copy of *x_mod* with the same mean value as *x_org*.

Examples

For example,

```
>>> from magni.imaging.visualisation import shift_mean
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_mod = np.ones((2, 2))
>>> x_org.mean()
1.5
>>> x_mod.mean()
1.0
>>> shifted_x_mod = shift_mean(x_mod, x_org)
>>> shifted_x_mod.mean()
1.5
>>> shifted_x_mod
array([[ 1.5,  1.5],
       [ 1.5,  1.5]])
```

magni.imaging.visualisation.**stretch_image**(*img*, *max_val*)

Stretch image such that pixels values are in the range [0, *max_val*].

Parameters:	• **img** (*ndarray*) – The (float) image that is to be stretched. • **max_val** (*int or float*) – The maximum value in the stretched image.
Returns:	**stretched_img** (*ndarray*) – A stretched copy of the input image.

Notes

The pixel values in the input image are scaled to lie in the interval [0, *max_val*] using a linear stretch.

Examples

For example,

```
>>> from magni.imaging.visualisation import stretch_image
>>> img = np.arange(4, dtype=np.float).reshape(2, 2)
>>> stretched_img = stretch_image(img, 1.0)
>>> stretched_img
array([[ 0.        ,  0.33333333],
       [ 0.66666667,  1.        ]])
```

magni.reproducibility package

Module providing functionality for aiding in quest for more reproducible research.

Routine listings

io
    Module providing input/output functions to databases containing results from reproducible research.

Submodules

magni.reproducibility._annotation module

Module providing functions that may be used to annotate data.

Routine Listings

get_conda_info()
: Function that returns information about a Continnum Anaconda install.

get_datetime()
: Function that returns information about the current date and time.

get_git_revision()
: Function that returns information about the magni git revision.

get_magni_config()
: Function that returns information about the current configuration of Magni.

get_magni_info()
: Function that returns genral information about Magni.

get_platform_info()
: Function that returns information about the platform used to run the code.

Notes

The return annotations are any nested level of dicts of dicts of strings.

magni.reproducibility._annotation.**get_conda_info**()

Return a dictionary contianing information from Conda.

Conda is the package manager for the Continuum Anaconda scientific Python distribution. This function will return various information about the Anaconda installation on the system by quering the Conda package database.

Returns: **conda_info** (*dict*) – Various information from conda (see notes below for further details).

Notes

If the Python intepreter is unable to locate and import the conda package, an empty dicionary is returned.

The returned dictionary contains the same infomation that is returned by "conda info" in addition to an overview of the linked modules in the Anaconda installation. Specifically, the returned dictionary has the following keys:

- platform
- conda_version
- root_prefix
- default_prefix
- envs_dirs
- package_cache
- channels
- config_file
- is_foreign_system
- linked_modules

Additionally, the returned dictionary has a key named *status*, which can have either of the following values:

- 'Succeeded' (Everything seems to be OK)

- 'Failed' (Import of conda failed – nothing else is returned)

magni.reproducibility._annotation.**get_datetime**()

Return a dictionary holding the current date and time.

Returns: **date_time** (*dict*) – The dictionary holding the current date and time.

Notes

The returned dictionary has the following keys:

- today (date and time including timezone offset)
- utcnow (UTC date and time)
- pretty_utc (UTC date and time formatted according to current locale)
- status

The status entry informs about the success of the pretty_utc formatting. It has one of the follwing values:

- Succeeded (Everything seems OK)
- Failed (It was not possible to format the time)

magni.reproducibility._annotation.**get_git_revision**()

Return a dictionary containing information about the current git revision.

Returns: **git_revision** (*dict*) – Information about the current git revision.

Notes

If the git revision extract succeeded, the returned dictionary has the following keys:

- status (with value 'Succeeded')
- tag (output of "git describe")
- branch (output of "git describe –all")

If the git revision extract failed, the returned dictionary has the following keys:

- status (with value 'Failed')
- returncode (returncode from failing git command)
- output (output from failing git command)

The "git describe" commands are run in the directory in which magni is loaded from.

magni.reproducibility._annotation.**get_magni_config**()

Return a dictionary holding the current configuration of Magni.

Returns: **magni_config** (*dict*) – The dictionary holding the current configuration of Magni.

Notes

The returned dictionary has a key for each of the *config* modules in Magni and its subpackages. The value of a given key is a dictionary with the current configuration of the corresponding *config* module. Furthermore, the returned dictionary has a status key, which can have either of the following values:

- Succeeded (The entire configuration was extracted)
- Failed (It was not possible to get information from one or more modules)

magni.reproducibility._annotation.**get_magni_info**()

Return a string representation of the output of help(magni).

Returns: **magni_info** (*dict*) – Information about magni.

Notes

The returned dictionary has a single key:

- help_magni (a string representation of help(magni))

magni.reproducibility._annotation.**get_platform_info**()

Return a dictionary containing information about the system platform.

Returns: **platform_info** (*dict*) – Various information about the system platform.

See also:

platform()
    The Python module used to query information about the system.

Notes

The returned dictionary has the following keys:

- system
- node
- release
- version
- processor
- python
- libc
- linux
- mac_os
- win32
- status

The linux/mac_os/win32 entries are “empty” if they are not applicable.

If the processor information returned by **platform** is “empty”, a query of *lscpu* is attempted in order to provide the necessary information.

The status entry informs about the success of the queries. It has one of the follwing values:

- ‘All OK’ (everything seems to be OK)
- ‘Used lscpu in processor query’ (*lscpu* was used)

- 'Processor query failed' (failed to get processor information)

magni.reproducibility.io module

Module providing input/output functions to databases containing results from reproducible research.

Routine listings

annotate_database(h5file)
: Function for annotating an existing HDF5 database.

read_annotations(h5file)
: Function for reading annotations in an HDF5 database.

remove_annotations(h5file)
: Function for removing annotations in an HDF5 database.

See also:

magni.reproducibility._annotation.get_conda_info
: Conda annotation

magni.reproducibility._annotation.get_git_revision
: Git annotation

magni.reproducibility._annotation.get_platform_info
: Platform annotation

magni.reproducibility._annotation.get_datetime
: Date and time annotation

magni.reproducibility._annotation.get_magni_config
: Magni config annotation

magni.reproducibility._annotation.get_magni_info
: Magni info annotation

magni.reproducibility.io.**annotate_database**(*h5file*)

Annotate an HDF5 database with information about Magni and the platform.

The annotation consists of a group in the root of the *h5file* having nodes that each provide information about Magni or the platform on which this function is run.

Parameters:	**h5file** (*tables.file.File*) – The handle to the HDF5 database that should be annotated.

See also:

magni.reproducibility._annotation.get_conda_info()
: Conda annotation

magni.reproducibility._annotation.get_git_revision()
: Git annotation

magni.reproducibility._annotation.get_platform_info()
: Platform annotation

magni.reproducibility._annotation.get_datetime()

Date and time annotation

magni.reproducibility._annotation.get_magni_config()

Magni config annotation

magni.reproducibility._annotation.get_magni_info()

Magni info annotation

Notes

The annotations of the database includes the following:

- conda_info – Information about Continuum Anacononda install
- git_revision – Git revision and tag of Magni
- platform_info – Information about the current platform (system)
- datetime – The current date and time
- magni_config – Infomation about the current configuration of Magni
- magni_info – Information from *help(magni)*

Examples

Annotate the database named 'db.hdf5':

```
>>> from magni.reproducibility.io import annotate_database
>>> with magni.utils.multiprocessing.File('db.hdf5', mode='a') as h5file:
...     annotate_database(h5file)
```

magni.reproducibility.io.**read_annotations**(*h5file*)

Read the annotations to an HDF5 database.

Parameters:	**h5file** (*tables.file.File*) – The handle to the HDF5 database from which the annotations is read.
Returns:	**annotations** (*dict*) – The annotations read from the HDF5 database.
Raises:	**ValueError** – If the annotations to the HDF5 database does not conform to the Magni annotation standard.

Notes

The returned dict holds a key for each annotation in the database. The value corresponding to a given key is in itself a dict. See *magni.reproducibility.annotate_database* for examples of such annotations.

Examples

Read annotations from the database named 'db.hdf5':

```
>>> from magni.reproducibility.io import read_annotations
>>> with magni.utils.multiprocessing.File('db.hdf5', mode='r') as h5file:
...     annotations = read_annotations(h5file)
```

magni.reproducibility.io.**remove_annotations**(*h5file*)

Remove the annotations from an HDF5 database.

Parameters: **h5file** (*tables.file.File*) – The handle to the HDF5 database from which the annotations is removed.

Examples

Remove annotations from the database named 'db.hdf5':

```
>>> from magni.reproducibility.io import remove_annotations
>>> with magni.utils.multiprocessing.File('db.hdf5', mode='a') as h5file:
...     remove_annotations(h5file)
```

magni.utils package

Subpackage providing support functionality for the other subpackages.

Routine listings

multiprocessing
: Subpackage providing intuitive and extensive multiprocessing functionality.

config
: Module providing a robust configger class.

matrices
: Module providing matrix emulators.

plotting
: Module providing utilities for control of plotting using **matplotlib**.

validation
: Module providing validation capability.

split_path(path)
: Split a path into folder path, file name, and file extension.

Notes

See _util for documentation of **split_path**.

Subpackages

magni.utils.multiprocessing package

Subpackage providing intuitive and extensive multiprocessing functionality.

Routine listings

config
: Module providing configuration options for this subpackage.

File()
: Control pytables access to hdf5 files when using multiprocessing.

process(func, namespace={}, args_list=None, kwargs_list=None, maxtasks=None)

Map multiple function calls to multiple processors.

Notes

See _util for documentation of `File`. See _processing for documentation of `process`.

Submodules
magni.utils.multiprocessing._processing module

Module providing the process function.

Routine listings

process(func, namespace={}, args_list=None, kwargs_list=None, maxtasks=None)
Map multiple function calls to multiple processors.

See also:

magni.utils.multiprocessing.config
Configuration options.

magni.utils.multiprocessing._processing.**process**(*func*, *namespace={}*, *args_list=None*, *kwargs_list=None*, *maxtasks=None*)

Map multiple function calls to multiple processors.

For each entry in args_list and kwargs_list, a task is formed which is used for a function call of the type *func(*args, **kwargs)*.

Parameters:
- **func** (*function*) – A function handle to the function which the calls should be mapped to.
- **namespace** (*dict, optional*) – A dict whose keys and values should be globally available in func (the default is an empty dict).
- **args_list** (*list or tuple, optional*) – A sequence of argument lists for the function calls (the default is None, which implies that no arguments are used in the calls).
- **kwargs_list** (*list or tuple, optional*) – A sequence of keyword argument dicts for the function calls (the default is None, which implies that no keyword arguments are used in the calls).
- **maxtasks** (*int, optional*) – The maximum number of tasks of a process before it is replaced by a new process (the default is None, which implies that processes are not replaced).

Returns: **results** (*list*) – A list with the results from the function calls.

See also:

magni.utils.multiprocessing.config()
Configuration options.

Notes

If the *workers* configuration option is equal to 0, map is used. Otherwise, the map functionality of a multiprocessing worker pool is used.

Reasons for using this function over map or standard multiprocessing:

- Simplicity of the code over standard multiprocessing.
- Simplicity in switching between single- and multiprocessing.
- The use of both arguments and keyword arguments in the function calls.
- The reporting of exceptions before termination.
- The possibility of terminating multiprocessing with a single interrupt.

Examples

An example of how to use args_list, and kwargs_list:

```
>>> from magni.utils.multiprocessing._processing import process
>>> def calculate(a, b, op='+'):
...     if op == '+':
...         return a + b
...     elif op == '-':
...         return a - b
...
>>> args_list = [[5, 7], [9, 3]]
>>> kwargs_list = [{'op': '+'}, {'op': '-'}]
>>> process(calculate, args_list=args_list, kwargs_list=kwargs_list)
[12, 6]
```

magni.utils.multiprocessing._processing.**_process_init**(*func*, *namespace*)

Initialise the process by making global variables available to it.

Parameters:
- **func** (*function*) – A function handle to the function which the calls should be mapped to.
- **namespace** (*dict*) – A dict whose keys and values should be globally available in func.

magni.utils.multiprocessing._processing.**_process_worker**(*fak_tuple*)

Unpack and map a task to the function.

Parameters:
- **fak_tuple** (*tuple*) – A tuple (func, args, kwargs) containing the parameters listed below.
- **func** (*function*) – A function handle to the function which the calls should be mapped to.
- **args** (*list or tuple*) – The sequence of arguments that should be unpacked and passed.
- **kwargs** (*list or tuple*) – The sequence of keyword arguments that should be unpacked and passed.

Notes

If an exception is raised in *func*, the stacktrace of that exception is printed since the exception is otherwise silenced until every task has been executed when using multiple workers.

Also, a workaround has been implemented to allow KeyboardInterrupts to interrupt the current tasks and all remaining tasks. This is done by setting a global variable, when catching a KeyboardInterrupt, which is checked for every call.

magni.utils.multiprocessing._util module

Module providing the public class of the magni.utils.multiprocessing subpackage.

class magni.utils.multiprocessing._util.**File**(**args*, ***kwargs*)

Control pytables access to hdf5 files when using multiprocessing.

File retains the interface of **tables.open_file** and should only be used in 'with' statements (see Examples).

Parameters:
- **args** (*tuple*) – The arguments that are passed to 'tables.open_file'.
- **kwargs** (*dict*) – The keyword arguments that are passed to 'tables.open_file'.

See also:

tables.open_file
The wrapped function.

Notes

Internally the module uses a global lock which is shared amongst all files. This solution is simple and does not entail significant overhead. However, the wait time introduced when using multiple files at the same time can be significant.

Examples

The class is used in the following way:

```
>>> from magni.utils.multiprocessing._util import File
>>> with File('database.hdf5', 'a') as f:
...     pass # execute something involving the opened file
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

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

__enter__()

Acquire the global lock before opening and returning the file.

Returns: **file** (*tables.File*) – The file specified in the call to __init__.

__exit__(*type*, *value*, *traceback*)

Release the global lock after closing the file.

Parameters:	• **type** (*type*) – The type of the exception raised, if any. • **value** (*Exception*) – The exception rasied, if any. • **traceback** (*traceback*) – The traceback of the exception raised, if any.

magni.utils.multiprocessing.config module

Module providing configuration options for the multiprocessing subpackage.

Routine listings

get(key=None)
: Get the value of one or more configuration options.

set(dictionary={}, **kwargs)
: Set the value of one or more configuration options.

> **See also:**
>
> **magni.utils.config.Configger**
> The Configger class used.

Notes

This module instantiates the *Configger* class provided by **magni.utils.config** and assigns handles for the `get` and `set` methods of that class instance. The configuration options are the following:

workers : *int*
: The number of workers to use for multiprocessing (the default is 0, which implies no multiprocessing).

Submodules

magni.utils._util module

Module providing the public function of the magni.utils subpackage.

magni.utils._util.**split_path**(*path*)

Split a path into folder path, file name, and file extension.

The returned folder path ends with a folder separation character while the returned file extension starts with an extension separation character. The function is independent of the operating system and thus of the use of folder separation character and extension separation character.

Parameters:	**path** (*str*) – The path of the file either absolute or relative to the current working directory.
Returns:	• **path** (*str*) – The path of the containing folder of the input path. • **name** (*str*) – The name of the object which the input path points to. • **ext** (*str*) – The extension of the object which the input path points to (if any).

Examples

Concatenate a dummy path and split it using the present function:

```
>>> from magni.utils._util import split_path
>>> path = 'folder' + os.sep + 'file' + os.path.extsep + 'extension'
>>> parts = split_path(path)
>>> print(tuple((parts[0][-7:-1], parts[1], parts[2][1:])))
('folder', 'file', 'extension')
```

magni.utils.config module

Module providing a robust configger class.

Routine listings

Configger()
: Provide set and get functions to access a set of configuration options.

Notes

This module does not itself contain any configuration options and thus has no get or set functions unlike the other config modules of **magni**.

class magni.utils.config.**Configger**(*param*, *requirements*)

Provide set and get functions to access a set of configuration options.

The set of configuration options, their default values, and their validation schemes are specified upon initialisation.

Parameters:
- **param** (*dict*) – The configuration options along with their default values.
- **requirements** (*dict*) – The validation schemes of the configuration options.

See also:

magni.utils.validation
: Validation.

Notes

requirements must contain the same keys as *param*. For each key in *requirements*, the value is used as validation scheme – see **set** for further information.

Examples

Instantiate Configger with the parameter 'key' with default value 'default' which can only assume string values.

```
>>> from magni.utils.config import Configger
>>> config = Configger({'key': 'default'}, {'key': {'type': str}})
```

The parameter can either be retrieved by getting a copy of the entire parameter dictionary or by getting the specific key.

```
>>> config.get()
{'key': 'default'}
>>> config.get('key')
'default'
```

Likewise the parameter can either be changed by passing a dictionary with the parameter or by using a keyword argument.

```
>>> config.set({'key': 'value'})
>>> config.set(key='value')
>>> config.get('key')
'value'
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*param*, *requirements*)

_validate_get(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

get(*key=None*)

Retrieve a copy of all parameters or a specific parameter.

Parameters:	**key** (*str or None, optional*) – The name of the key to retrieve (the default is None, which implies retrieving a copy of all parameters)
Returns:	**value** (*dict or None*) – The value of the specified key, if a key is not None. Otherwise, a copy of the parameter dictionary.

_validate_set(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

set(*dictionary={}*, ***kwargs*)

Overwrite the value of one or more parameters.

Each value is validated according to the validation scheme of that parameter.

Parameters:
- **dictionary** (*dict, optional*) – A dictionary containing the key and value pairs to update.
- **kwargs** (*dict, optional*) – Keyword arguments being the key and value pairs to update.

See also:

magni.utils.validation.validate()
Validation.

magni.utils.matrices module

Module providing matrix emulators.

The matrix emulators of this module are wrappers of fast linear operations giving the fast linear operations the same basic interface as a numpy ndarray. Thereby allowing fast linear operations and numpy ndarrays to be used interchangably in other parts of the package.

Routine listings

Matrix()
Wrap fast linear operations in a matrix emulator.

MatrixCollection()
Wrap multiple matrix emulators in a single matrix emulator.

See also:

magni.imaging._fastops
Fast linear operations.

class magni.utils.matrices.**Matrix**(*func*, *trans*, *args*, *shape*)

Wrap fast linear operations in a matrix emulator.

Matrix defines a few attributes and internal methods which ensures that instances have the same basic interface as a numpy matrix instance without explicitly forming the matrix. This basic interface allows instances to be multiplied with vectors, to be transposed, and to assume a shape. Also, instances have an attribute which explicitly forms the matrix.

Parameters:
- **func** (*function*) – The fast linear operation applied to the vector when multiplying the matrix with a vector.
- **trans** (*function*) – The fast linear operation applied to the vector when multiplying the transposed matrix with a vector.
- **args** (*list or tuple*) – The arguments which should be passed to *func* and *trans* in addition to the vector.
- **shape** (*list or tuple*) – The shape of the emulated matrix.

Examples

For example, the negative identity matrix could be emulated as

```
>>> from magni.utils.matrices import Matrix
>>> func = lambda vec: -vec
>>> matrix = Matrix(func, func, (), (3, 3))
```

The example matrix will have the desired shape:

```
>>> matrix.shape
(3, 3)
```

The example matrix will behave just like an explicit matrix:

```
>>> vec = np.float64([1, 2, 3]).reshape(3, 1)
>>> matrix.dot(vec)
array([[-1.],
       [-2.],
       [-3.]])
```

If, at some point, an explicit representation of the matrix is required, this can easily be obtained:

```
>>> matrix.A
array([[-1., -0., -0.],
       [-0., -1., -0.],
       [-0., -0., -1.]])
```

Likewise, the transpose of the matrix can be obtained:

```
>>> matrix.T.A
array([[-1., -0., -0.],
       [-0., -1., -0.],
       [-0., -0., -1.]])
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*func*, *trans*, *args*, *shape*)

A

Explicitly form the matrix.

The fast linear operations implicitly define a matrix which is usually not explicitly formed. However, some functionality might require a more advanced matrix interface than that provided by this class.

Returns: **matrix** (*numpy.ndarray*) – The explicit matrix.

Notes

The explicit matrix is formed by multiplying the matrix with the columns of an identity matrix and stacking the resulting vectors as columns in a matrix.

shape

Get the shape of the matrix.

Returns: **shape** (*tuple*) – The shape of the matrix.

T

Get the transpose of the matrix.

Returns: **matrix** (*Matrix*) – The transpose of the matrix.

Notes

The fast linear operation and the fast linear transposed operation of the resulting matrix are same as those of the current matrix except swapped. The shape is modified accordingly.

_validate_dot(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

dot(*vec*)

Multiply the matrix with a vector.

Parameters: **vec** (*numpy.ndarray*) – The vector which the matrix is multiplied with.

Returns: **vec** (*numpy.matrix*) – The result of the multiplication.

class magni.utils.matrices.**MatrixCollection**(*matrices*)

Wrap multiple matrix emulators in a single matrix emulator.

MatrixCollection defines a few attributes and internal methods which ensures that instances have the same basic interface as a numpy matrix instance without explicitly forming the matrix. This basic interface allows instances to be multiplied with vectors, to be transposed, and to assume a shape. Also, instances have an attribute which explicitly forms the matrix.

Parameters: **matrices** (*list or tuple*) – The collection of Matrix instances.

See also:

Matrix

Matrix emulator.

Examples

For example, two matrix emulators can be combined into one. That is, the matrix:

```
>>> func = lambda vec: -vec
>>> negate = magni.utils.matrices.Matrix(func, func, (), (3, 3))
>>> negate.A
array([[-1., -0., -0.],
       [-0., -1., -0.],
       [-0., -0., -1.]])
```

And the matrix:

```
>>> func = lambda vec: vec[::-1]
>>> reverse = magni.utils.matrices.Matrix(func, func, (), (3, 3))
>>> reverse.A
array([[ 0.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  0.]])
```

Can be combined into one matrix emulator using the present class:

```
>>> from magni.utils.matrices import MatrixCollection
>>> matrix = MatrixCollection((negate, reverse))
```

The example matrix will have the desired shape:

```
>>> matrix.shape
(3, 3)
```

The example matrix will behave just like an explicit matrix:

```
>>> vec = np.float64([1, 2, 3]).reshape(3, 1)
>>> matrix.dot(vec)
array([[-3.],
       [-2.],
       [-1.]])
```

If, at some point, an explicit representation of the matrix is required, this can easily be obtained:

```
>>> matrix.A
array([[-0., -0., -1.],
       [-0., -1., -0.],
       [-1., -0., -0.]])
```

Likewise, the transpose of the matrix can be obtained:

```
>>> matrix.T.A
array([[-0., -0., -1.],
       [-0., -1., -0.],
       [-1., -0., -0.]])
```

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	
	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*matrices*)

A

Explicitly form the matrix.

The collection of matrices implicitly defines a matrix which is usually not explicitly formed. However, some functionality might require a more advanced matrix interface than that provided by this class.

Returns: **matrix** (*numpy.ndarray*) – The explicit matrix.

Notes

The explicit matrix is formed by multiplying the matrix with the columns of an identity matrix and stacking the resulting vectors as columns in a matrix.

shape

Get the shape of the matrix.

Returns: **shape** (*tuple*) – The shape of the matrix.

Notes

The shape of the product of a number of matrices is the number of rows of the first matrix times the number of columns of the last matrix.

T

Get the transpose of the matrix.

Returns: **matrix** (*MatrixCollection*) – The transpose of the matrix.

Notes

The transpose of the product of the number of matrices is the product of the transpose of the matrices in reverse order.

_validate_dot(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:
- **args** (*tuple*) – The arguments passed to the decorated function.
- **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

dot(*vec*)

Multiply the matrix with a vector.

Parameters: **vec** (*numpy.matrix*) – The vector which the matrix is multiplied with.

Returns: **vec** (*numpy.matrix*) – The result of the multiplication.

magni.utils.plotting module

Module providing utilities for control of plotting using **matplotlib**.

The module has a number of public attributes which provide settings for colormap cycles, linestyle cycles, and marker cycles that may be used in combination with

matplotlib.

Routine listings

setup_matplotlib(settings={}, cmap='jet')
: Function that set the Magni default **matplotlib** configuration.

colour_collections : *dict*
: Collections of colours that may be used in e.g., a **matplotlib** color_cycle.

seq_cmaps : *list*
: Names of **matplotlib.cm** colormaps optimized for sequential data.

div_cmaps : *list*
: Names of **matplotlib.cm** colormaps optimized for diverging data.

linestyles : *list*
: A subset of linestyles from **matplotlib.lines**

markers : *list*
: A subset of markers from **matplotlib.markers**

Examples

Use the default Magni matplotlib settings.

```
>>> magni.utils.plotting.setup_matplotlib()
```

Get the normalised 'Blue' colour brew from the psp colour map:

```
>>> magni.utils.plotting.colour_collections['psp']['Blue']
((0.1255, 0.502, 0.8745),)
```

class magni.utils.plotting.**_ColourCollection**(*brews*)

Bases: **object**

A container for colour maps.

A single colour is stored as an RGB 3-tuple of integers in the interval [0,255]. A set of related colours is termed a colour brew and is stored as a list of colours. A set of related colour brews is termed a colour collection and is stored as a dictionary. The dictionary key identifies the name of the colour collection whereas the value is the list of colour brews.

The default colour collections named "cb*" are colorblind safe, print friendly, and photocopy-able. They have been created using the online ColorBrewer 2.0 tool [1].

Parameters:	**brews** (*dict*) – The dictionary of colour brews from which the colour collection is created.

Notes

Each colour brew is a list (or tuple) of length 3 lists (or tuples) of RGB values.

References

[1] M. Harrower and C. A. Brewer, "ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps", *The Cartographic Journal*, vol. 40, pp. 27-37, 2003 (See also: http://colorbrewer2.org/)

_validate_init(**args*, ***kwargs*)

Wrap a validation function (see module Notes).

Parameters:	• **args** (*tuple*) – The arguments passed to the decorated function. • **kwargs** (*dict*) – The keyword arguments passed to the decorated function.

__init__(*brews*)

_validate_getitem(*name*)

Validate the __getitem__ function.

See also:

_ColourCollection.__getitem__()
: The validated function.

magni.utils.validation.validate()
: Validation.

__getitem__(*name*)

Return a single colour brew.

The returned colour brew is normalised in the sense of matplotlib normalised rgb values, i.e., colours are 3-tuples of floats in the interval [0, 1].

Parameters:	**name** (*str*) – Name of the colour brew to return.
Returns:	**brew** (*tuple*) – A colour brew list.

magni.utils.plotting.**setup_matplotlib**(*settings={}*, *cmap=None*)

Adjust the configuration of `matplotlib`.

Sets the default configuration of `matplotlib` to optimize for producing high quality plots of the data produced by the functionality provided in the Magni.

Parameters:	• **settings** (*dict, optional*) – A dictionary of custom matplotlibrc settings. See examples for details about the structure of the dictionary. • **cmap** (*str or matplotlib.colors.Colormap, optional*) – Colormap to be used by matplotlib (the default is None, which implices that the 'jet' colormap is used).
Raises:	`UserWarning` – If the supplied custom settings are invalid.

Examples

For example, set lines.linewidth=2 and lines.color='r'.

```
>>> from magni.utils.plotting import setup_matplotlib
>>> custom_settings = {'lines': {'linewidth': 2, 'color': 'r'}}
>>> setup_matplotlib(custom_settings)
```

magni.utils.validation module

Module providing validation capability.

The intention is to validate all public functions of the package such that erroneous arguments in calls are reported in an informative fashion rather than causing arbitrary exceptions or unexpected results. To avoid performance impairments, the validation can be disabled globally.

Routine listings

decorate_validation(func)
: Decorate a validation function (see Notes).

disable_validation()
: Disable validation globally (see Notes).

validate(var, path, levels, ignore_none=False)
: Validate the value of a variable according to a validation scheme.

validate_ndarray(var, path, constraints)
: Validate a numpy.ndarray according to a validation scheme.

Notes

To be able to disable validation (and to ensure consistency), for every public function a validation function with the same name prefixed by '_validate_' should be defined. This function should be decorated by **decorate_validation**, be placed just above the function which it decorates, and be called as the first thing with all arguments.

Examples

If, for example, the following function is defined:

```
>>> def greet(person, greeting):
...     print('{}, {} {}.'.format(greeting, person['title'], person['name']))
```

This function probably expects its argument, 'person' to be a dictionary with keys 'title' and 'name' and its argument, 'greeting' to be a string. If, for example, a list is passed as the first argument, a TypeError is raised with the description 'list indices must be integers, not str'. While obviously correct, this message is not excessively informative to the user of the function. Instead, this module can be used to redefine the function as follows:

```
>>> from magni.utils.validation import validate, decorate_validation
>>> @decorate_validation
... def _validate_greet(person, greeting):
...     validate(person, 'person', {'type': dict, 'keys': ('title', 'name')})
...     validate(greeting, 'greeting', {'type': str})
>>> def greet(person, greeting):
...     _validate_greet(person, greeting)
...     print('{}, {} {}.'.format(greeting, person['title'], person['name']))
```

If, again, a list is passed as the first argument, a TypeError with the description ‘type(person) must be <type ‘dict’>’. Now, the user of the function can easily identify the mistake and correct the call to read:

```
>>> greet({'title': 'Mr.', 'name': 'Anderson'}, 'You look surprised to see me')
You look surprised to see me, Mr. Anderson.
```

magni.utils.validation.**decorate_validation**(*func*)

Decorate a validation function (see module Notes).

Parameters:	**func** (*function*) – The validation function which should be decorated.

Examples

An example of a function which accepts only an integer as argument:

```
>>> from magni.utils.validation import decorate_validation
>>> @decorate_validation
... def _validate_test(arg):
...     magni.utils.validation.validate(arg, 'arg', {'type': int})
>>> def test(arg):
...     _validate_test(arg)
...     return
```

If the function is called with anything but an integer, it fails:

```
>>> try:
...     test('string')
...     print('No exception occured')
... except BaseException:
...     print('An exception occured')
An exception occured
```

magni.utils.validation.**disable_validation**()

Disable validation globally (see module Notes).

There is no equivalent function to enable validation since either any or no function calls should be validated depending on the run mode.

Examples

An example of a function which accepts only an integer as argument:

```
>>> @magni.utils.validation.decorate_validation
... def _validate_test(arg):
...     magni.utils.validation.validate(arg, 'arg', {'type': int})
>>> def test(arg):
...     _validate_test(arg)
...     return
```

If the function is called with anything but an integer, it fails:

```
>>> try:
...     test('string')
...     print('No exception occured')
... except BaseException:
...     print('An exception occured')
An exception occured
```

However, if validation is disabled, the same call does not fail:

```
>>> from magni.utils.validation import disable_validation
>>> disable_validation()
>>> try:
...     test('string')
...     print('No exception occured')
... except BaseException:
...     print('An exception occured')
No exception occured
```

magni.utils.validation.**validate**(*var*, *path*, *levels*, *ignore_none=False*)

Validate the value of a variable according to a validation scheme.

Parameters:

- **var** (*None*) – The variable to validate which can take any type.
- **path** (*str*) – The path of the variable which is printed if the variable is invalid.
- **levels** (*list or tuple or dict*) – The validation scheme.
- **ignore_none** (*bool, optional*) – The flag indicating whether the variable is allowed to have the value None (the default is False).

Notes

The *levels* parameter is either a dict or a sequence of dicts. If it is a dict, it is considered a sequence of one dict. Each dict corresponds to the validation scheme of a 'level': the first level consists of *var* itself, the second level consists of any value in *var* if *var* is itself a list, tuple, or dict, and so on.

A level is validated according to the keys and values of the dict specifying the validation scheme of that level. See the source for details on the usable keys and their function.

Note: It is assumed that *var*, *path*, *levels*, and *ignore_none* are all valid arguments as these are not explicitly validated.

Examples

Define a function for reporting if an exception occurs when calling making some call:

```
>>> from magni.utils.validation import validate
>>> def report(call):
...     try:
...         call()
...         print('No exception occured')
...     except BaseException:
...         print('An exception occured')
```

An example of how to validate an integer:

```
>>> var = 1
>>> call = lambda: validate(var, 'var', {'type': int})
>>> report(call)
No exception occured
```

The above code fails when, e.g., a string is passed instead:

```
>>> var = 'string'
>>> report(call)
An exception occured
```

An example of how to validate a tuple expected to hold two integers:

```
>>> var = (42, 1337)
>>> call = lambda: validate(var, 'var', ({'type': tuple, 'len': 2},
...                                      {'type': int}))
>>> report(call)
No exception occured
```

The above code fails when, e.g., a tuple with three integers is passed instead:

```
>>> var = (1, 2, 3)
>>> report(call)
An exception occured
```

magni.utils.validation.**validate_ndarray**(*var*, *path*, *constraints={}*, *ignore_none=False*)

Validate a numpy.ndarray according to a validation scheme.

Parameters:
- **var** (*numpy.ndarray*) – The variable to validate.
- **path** (*str*) – The path of the variable which is printed if the variable is invalid.
- **constraints** (*dict*) – The validation scheme.
- **ignore_none** (*bool, optional*) – Boolean indicating whether the variable is allowed to have the value None (the default is False).

Notes

The variable is validated according to the keys and values of the dict specifying the validation scheme. See the source for details on the usable keys and their function.

> **Note:** It is assumed that *var*, *path*, *levels*, and *ignore_none* are all valid arguments as these are not explicitly validated.

Examples

Define a function for reporting if an exception occurs when calling making some call:

```
>>> from magni.utils.validation import validate_ndarray
>>> def report(call):
...     try:
...         call()
...         print('No exception occured')
...     except BaseException:
...         print('An exception occured')
```

An example of how to validate a float64 numpy.ndarray of shape (5,):

```
>>> var = np.float64([1, 2, 3, 4, 5])
>>> call = lambda: validate_ndarray(var, 'var',
...                                 {'dtype': np.float64, 'shape': (5,)})
>>> report(call)
No exception occured
```

The above code fails when, e.g., an int64 numpy.ndarray is passed instead:

```
>>> var = np.int64([1, 2, 3, 4, 5])
>>> report(call)
An exception occured
```

magni.utils.validation.**_error**(*error*, *message*, *args*)

Raise an error of a given type with a given message.

Parameters:

- **error** (*{'type', 'value', 'key'}*) – The type of the raised error.
- **message** (*str*) – The message of the raised error after being formatted using 'format'.
- **args** (*list or tuple*) – The args passed to 'format'.

Magni 1.0.0 documentation »