Magni Documentation

Version 1.5.0

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

June 30, 2016

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 [5] 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 Tests and 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

[5] 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 documentation 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 license, see the License

section. To install **magni**, follow the instructions given under Download and Installation.

magni has been tested with Anaconda (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

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 Anaconda.org.

Dependencies

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

Note: The below listed dependency version requirements are the absolute minimum tested versions. Some of these libraries may not work with newer versions of Python unless a newer version of those libraries are used.

Required third party dependencies for magni are:

- Matplotlib (Tested on version >= 1.3)
- Numpy (Tested on version >= 1.8)
- PyTables (Tested on version >= 3.1)
- Scipy (Tested on version >= 0.14)
- Setuptools (Tested on version >= 11.3)

Optional third party dependencies for magni are:

- Bottleneck (Tested on version >=1.0.0) (For speed-up of some algorithms)
- Conda (Tested on version >= 3.7.0) (For automatic metadata capture of a Conda managed Python environment)
- Coverage (Tested on version >= 3.7) (For running the test suite script)
- IPython (Tested on version >= 2.1) or Jupyter [4] (Tested on version >= 1.0) (For running the IPython notebook examples)
- Math Kernel Library (MKL) (Tested on version >= 11.1) (For accelerated vector operations)
- Nose (Tested on version >= 1.3) (For running unittests and doctests)
- PEP8 (Tested on version >= 1.5) (For running style check tests)
- PIL (or Pillow) (Tested on version >= 1.1.7) (For running the IPython notebook examples as tests)
- Pyflakes (Tested on version >= 0.8) (For running style check tests)
- Radon (Tested on version >= 1.2) (For running style check tests)
- Sphinx (Tested on version >= 1.3.1) (For building the documentation from source)

[4] The IPython project has evolved into Jupyter. When using Jupyter make sure to install the "full" Jupyter metapackage which includes all Jupyter and IPython components.

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
```

Tests

A test suite consisting of unittests, doctests, the IPython notebook examples, and several style checks is included in **magni**. The tests are organized in python modules found in the Magni folder under '/magni/tests/'. Each module features one or more **unittest.TestCase** classes containing the tests. Thus, the tests may be invoked using any test runner that supports the **unittest.TestCase**. E.g. running the wrapper for the doctests using **Nose** is done by issuing:

```
$ nosetests magni/tests/wrap_doctests.py
```

The entire test suite may be run by executing the convenience script `run_tests.py`:

```
$ magni/tests/run_tests.py
```

Note: This convenience script assumes that **magni** is available on the PYTHONPATH as explained under Download and Installation.

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 reproducibility 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:

- P. S. Pedersen, J. Østergaard and T. Larsen, "Modelling reconstruction quality of Lissajous undersampled atomic force microscopy images," *2016 IEEE 13th*

International Symposium on Biomedical Imaging (ISBI), Prague, Czech Republic, 2016, pp. 245-248, doi:10.1109/ISBI.2016.7493255.

- T. Arildsen, C. S. Oxvig, P. S. Pedersen, J. Østergaard, and T. Larsen, "Reconstruction Algorithms in Undersampled AFM Imaging", *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 1, pp. 31-46, Feb. 2016, doi:10.1109/JSTSP.2015.2500363.
- P. S. Pedersen, J. Østergaard, and T. Larsen, "Predicting reconstruction quality within compressive sensing for atomic force microscopy," *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, Orlando, FL, 2015, pp. 418-422, doi:10.1109/GlobalSIP.2015.7418229.
- C. S. Oxvig, P. S. Pedersen, T. Arildsen, J. Østergaard, and T. Larsen, "Magni: A Python Package for Compressive Sampling and Reconstruction of Atomic Force Microscopy Images", *Journal of Open Research Software*, vol. 2, no. 1, p. e29, Oct. 2014, doi:10.5334/jors.bk.
- 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:

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

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

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

Here the vector $\mathbf{x} \in \mathbb{C}^p$ represents the signal of interrest. 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 $k = |\mathrm{supp}(\alpha)|$.

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.

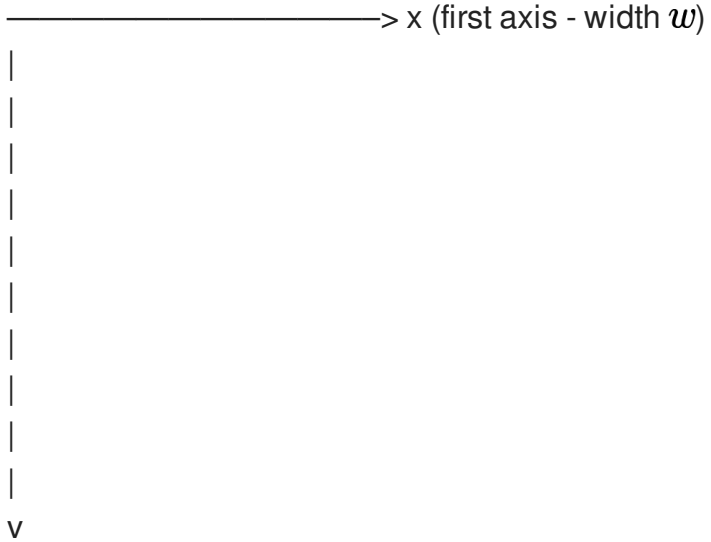
References

[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 parts of **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 its visual representation:



y (second axis - height h)

This way, a position on an AFM sample of size $w \times h$ 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 a mi-file. • Handling the resulting buffers and images.	• **magni.afm.io.read_mi_file**
cs-phase_transition-config	• Using Magni configuration modules including setting and getting configuration values.	• **magni.cs.phase_transition.config** • **magni.utils.config**

IPython Notebook Name	Example illustrates	Magni functionality used
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-reconstruction	• Reconstruction of compressively sampled 1D signals.	• magni.cs.reconstruction
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
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
magni	• The typical work flow in compressively sampling and reconstructing AFM images using Magni.	• magni.afm • magni.imaging
reporducibility-data	• Obtaining various platform and runtime data for annotations and chases.	• magni.reproducibility.data

IPython Notebook Name	Example illustrates	Magni functionality used
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-multiprocessing	• Doing multiprocessing using Magni.	• magni.utils.multiprocessing
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
: Configger providing configuration options for this subpackage.

io
: Subpackage providing input/output functionality for .mi files.

reconstruction
: Module providing reconstruction and analysis of reconstructed images.

types
: Subpackage providing data container classes for .mi files.

Subpackages

magni.afm.io package

Subpackage providing input/output functionality for .mi files.

Currently, only input functionality is provided.

Routine listings

read_mi_file(path)
: Read an .mi file given by a path.

Submodules
magni.afm.io._data_attachment module

Module providing functionality for attaching data.

The reading of an .mi file is logically separated into four steps of which the functionality provided by this module performs the third step.

Routine listings

attach_data(obj, data)

Attach data to a hierarchical object-structure.

magni.afm.io._data_attachment.**attach_data**(*obj*, *data*)

Attach data to a hierachical object-structure.

In the case of .mi image files, the data should be attached to the buffers of the file. In the case of .mi spectroscopy files, the data should be attached to the chunks of the file.

Parameters:
- **obj** (*object*) – The hierarchical object-structure to attach data to.
- **data** (*numpy.ndarray*) – The 1-dimensional data.

See also:

magni.afm.io.read_mi_file()

Function using the present function.

magni.afm.io._data_attachment.**_attach_image_data**(*obj*, *data*)

Attach data to a hierarchical object-structure representing an image.

The data should be attached to the buffers of the file.

Parameters:
- **obj** (*object*) – The hierarchical object-structure to attach data to.
- **data** (*numpy.ndarray*) – The 1-dimensional data.

See also:

attach_data()

Function using the present function.

magni.afm.io._data_attachment.**_attach_spectroscopy_data**(*obj*, *data*)

Attach data to a hierarchical object-structure representing a spectroscopy.

The data should be attached to the chunks of the file.

Parameters:
- **obj** (*object*) – The hierarchical object-structure to attach data to.
- **data** (*numpy.ndarray*) – The 1-dimensional data.

See also:

attach_data()

Function using the present function.

magni.afm.io._data_attachment.**_handle_format_inconsistency**(*obj*, *data*)

Handle format inconsistency.

The inconsistency is the optional precense of an undocumented thumbnail. If the thumbnail is present, the file header contains a thumbnail parameter specifying the resolution of the thumbnail image.

Parameters:
- **obj** (*object*) – The hierarchical object-structure to attach data to.
- **data** (*numpy.ndarray*) – The 1-dimensional data.

Returns:
- **obj** (*object*) – The hierarchical object-structure to attach data to.
- **data** (*numpy.ndarray*) – The 1-dimensional data excluding any thumbnail data.
- **thumbnail** (*list*) – A list of a thumbnail object-structure if any.

See also:

_attach_image_data()

Function using the present function.

Notes

The thumbnail is stored as 8-bit unsigned integers with a red, a green, and a blue channel. If the thumbnail is present, it is represented by three buffer object-structures; one for each channel.

magni.afm.io._object_building module

Module providing functionality for building a hierarchical object-structure.

The reading of an .mi file is logically separated into four steps of which the functionality provided by this module performs the second step.

Routine listings

build_object(params)

Build a hierarchical object-structure from header parameters.

magni.afm.io._object_building.**build_object**(*params*)

Build a hierarchical object-structure from header parameters.

The hierarchical object-structure mimics that of **magni.afm.types**.

Parameters: **params** (*list or tuple*) – The header parameters.

Returns: **obj** (*object*) – A hierarchical object-structure.

See also:

magni.afm.io.read_mi_file()

Function using the present function.

Notes

This function splits the header parameters into file-related parameters and buffer-related parameters.

magni.afm.io._object_building.**_build_buffer**(*file_type*, *index*, *params*)

Build a buffer-like object-structure from buffer parameters.

For spectroscopy buffers, this function converts parameters which contain "sub-parameters" to objects with attributes.

Parameters:
- **file_type** (*str*) – The .mi file type.
- **index** (*int*) – The index of the buffer.
- **params** (*list or tuple*) – The buffer parameters.

Returns: **obj** (*object*) – The buffer-like object-structure.

See also:

build_object()

Function using the present function.

magni.afm.io._object_building.**_expand_buffers**(*buffers*)

Expand the buffer-like object-structures.

Grid parameters specify implicit point parameters. Furthermore, some chunk parameters may be implicitly specified. These implicit parameters should be made explicit.

Parameters: **buffers** (*list or tuple*) – The buffer-like object-structures.

See also:

build_object()

Function using the present function.

_generate_grid_points()

Make implicit point parameters explicit.

_generate_implicit_chunks()

Make implicit chunk parameters explicit.

Notes

This function relies on the two functions, **_generate_grid_points** and **_generate_implicit_chunks** to make the implicit parameters explicit.

magni.afm.io._object_building.**_generate_grid_points**(*attrs*, *index*)

Make implicit grid point parameters explicit.

Parameters:
- **attrs** (*list or tuple*) – The attributes of the grid.
- **index** (*int*) – The index of the first grid point.

Returns: **points** (*tuple*) – The grid points.

See also:

_expand_buffers()
Function using the present function.

magni.afm.io._object_building.**_generate_implicit_chunks**(*npoints*, *explicit_chunks*)

Make implicit chunk parameters explicit.

Parameters:
- **npoints** (*int*) – The number of points.
- **explicit_chunks** (*list*) – The explicit chunks.

Returns: **chunks** (*list*) – The chunks including both explicit and implicit chunks.

See also:

_expand_buffer()
Function using the present function.

magni.afm.io._object_building.**_handle_format_inconsistency**(*obj*)

Handle format inconsistency.

The inconsistency is the usage of string values for the 'trace' header parameter which is specified to have a boolean value.

Parameters: **obj** (*object*) – The hierarchical object-structure.

Returns: **obj** (*object*) – The updated hierarchical object-structure.

See also:

build_object()
Function using the present function.

magni.afm.io._stream_conversion module

Module providing functionality for converting a file stream.

The reading of an .mi file is logically separated into four steps of which the functionality provided by this module performs the first step.

Routine listings

convert_stream(stream)

Convert a file stream to flat, basic variables.

magni.afm.io._stream_conversion.**convert_stream**(*stream*)

Convert a file stream to flat, basic variables.

The flat, basic variables are separated into header parameters and the data. The header parameters are name, value pairs whereas the data is a 1-dimensional numeric array.

Parameters: **stream** (*str*) – The file stream.

Returns:
- **params** (*tuple*) – The header parameters.
- **data** (*numpy.ndarray*) – The data.

See also:

magni.afm.io.read_mi_file()

Function using the present function.

Notes

This function splits the file stream into header and data, and splits the header into parameter name, value pairs.

magni.afm.io._stream_conversion.**_convert_image_data**(*buffer_*, *data_type*)

Convert the file stream image data to a 1-dimensional numeric array.

Parameters:
- **buffer_** (*str*) – The file stream image data.
- **data_type** (*str*) – The data type of the file stream image data.

Returns: **data** (*numpy.ndarray*) – The data as a 1-dimensional numeric array.

See also:

convert_stream()

Function using the present function.

magni.afm.io._stream_conversion.**_convert_parameter_value**(*string*)

Convert a file stream parameter value to a basic python value.

The converted value is either a boolean, string, floating point, or integer value or a list containing a mix of these.

Parameters: **string** (*str*) – The file stream parameter value.

Returns: **value** (*None*) – The converted, basic python value.

See also:

convert_stream()

Function using the present function.

magni.afm.io._stream_conversion.**_convert_spectroscopy_data**(*buffer_*, *data_type*)

Convert the file stream spectroscopy data to a 1-dimensional numeric array.

Parameters:
- **buffer_** (*str*) – The file stream spectroscopy data.
- **data_type** (*str*) – The data type of the file stream spectroscopy data.

Returns: **data** (*numpy.ndarray*) – The data as a 1-dimensional numeric array.

See also:

convert_stream()

Function using the present function.

magni.afm.io._util module

Module providing the public functionality of the magni.afm.io subpackage.

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

Read an .mi file given by a path.

The supported .mi file types are Image and Spectroscopy.

Parameters: **path** (*str*) – The path of the .mi file to read.

Returns: **file_** (*magni.afm.types.File*) – The read .mi file.

See also:

magni.afm.io._stream_conversion()

First step of the reading.

magni.afm.io._object_building()

Second step of the reading.

magni.afm.io._data_attachment()

Third step of the reading.

Notes

Depending on the .mi file type, the returned value will be an instance of a subclass of **magni.afm.types.File**: **magni.afm.types.image.Image** or **magni.afm.types.spectroscopy.Spectroscopy**.

The reading of an .mi file is logically separated into four steps:

1. Converting the file stream to flat, basic variables.
2. Converting the header parameters to a hierarchical object-structure mimicking that of **magni.afm.types**.
3. Attaching the actual data to the hierarchical object-structure.

4. Instantiating the classes in **magni.afm.types** from the hierarchical object structure.

The functionality needed for each step is grouped in separate modules with the functionality needed for the fourth step being grouped in the this module.

Examples

An example of how to use this function to read the example .mi file provided with the package:

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

magni.afm.io._util.**_instantiate_spectroscopy**(*obj*)

Instantiate a Spectroscopy object from a hierarchical object-structure.

Parameters: **obj** (*object*) – The hierarchical object-structure.
Returns: **spectroscopy** (*magni.afm.types.spectroscopy.Spectroscopy*) – The instantiated Spectroscopy object.

See also:

read_mi_file()
Function using the present function.

Notes

This function instantiates the sweep buffer and the spectroscopy file.

magni.afm.io._util.**_instantiate_spectroscopy_buffer**(*data_buffer*, *sweep_buffer*)

Instantiate a spectroscopy Buffer object from a hierarchical object-structure.

Parameters: **obj** (*object*) – The hierarchical object-structure.
Returns: **buffer_** (*magni.afm.types.spectroscopy.Buffer*) – The instantiated spectroscopy Buffer.

See also:

_instantiate_spectroscopy()
Function using the present function.

magni.afm.types package

Subpackage providing data container classes for .mi files.

Routine listings

image
Module providing data container classes for .mi image files.

spectroscopy
Module providing data container classes for .mi spectroscopy files.

BaseClass(object)
Base class of every **magni.afm.types** data class.

File(BaseClass)
Base class of the **magni.afm.types** file classes.

FileCollection(BaseClass)
Data class for collections of File instances with identical settings.

Submodules
magni.afm.types._util module

Module providing the common functionality of the subpackage.

class magni.afm.types._util.**BaseClass**(*attrs*)

Bases: **object**

Base class of every **magni.afm.types** data class.

The present class validates the attributes passed to its constructor against the allowed attributes of the class of the given instance and exposes these attributes through a read-only dictionary property, **attrs**. Furthermore, the present class exposes the allowed attributes of the class of the given instance through a read-only dictionary static property, **params**.

Parameters: **attrs** (*dict*) – The desired attributes of the instance.

attrs
magni.utils.types.ReadOnlyDict

The attributes of the instance.

params
magni.utils.types.ReadOnlyDict

The allowed attributes of the instance.

Examples

An example could be the subclass, ‘Person’ which allows only the string attribute, name:

```
>>> from magni.afm.types._util import BaseClass
>>> class Person(BaseClass):
...     def __init__(self, attrs):
...         BaseClass.__init__(self, attrs)
...     _params = {'name': str}
```

This class can then be initiated with a name:

```
>>> person = Person({'name': 'Murphy'})
>>> print('{!r}'.format(person.attrs['name']))
'Murphy'
```

Any other attributes, than 'name', passed to the class are ignored:

```
>>> person = Person({'name': 'Murphy', 'age': 42})
>>> for name in person.attrs.keys():
...     print('{!r}'.format(name))
'name'
```

attrs

params = *ReadOnlyDict()*

class magni.afm.types._util.**File**(*attrs*, *buffers*)

Bases: **magni.afm.types._util.BaseClass**

Base class of the **magni.afm.types** file classes.

The present class specifies the allowed file-level attributes and exposes the read-only property, buffers which all .mi files have. Furthermore, the present class provides a method for accessing buffers by their 'bufferLabel' attribute.

Parameters:
- **attrs** (*list or tuple*) – The desired attributes of the file.
- **buffers** (*list or tuple*) – The buffers of the file.

buffers

tuple

The buffers of the file.

See also:

BaseClass

Superclass of the present class.

Examples

A subclass of the present class is implicitly instantiated when loading, for example, the .mi file provided with the package:

```
>>> import os, magni
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> if os.path.isfile(path):
...     file_ = magni.afm.io.read_mi_file(path)
```

This file has a number of buffers which each has the 'bufferLabel' attribute:

```
>>> if os.path.isfile(path):
...     for buffer_ in file_.buffers[::2][:3]:
...         label = buffer_.attrs['bufferLabel']
...         print("Buffer with 'bufferLabel': {!r}".format(label))
... else:
...     for label in ('Topography', 'Deflection', 'Friction'):
...         print("Buffer with 'bufferLabel': {!r}".format(label))
Buffer with 'bufferLabel': 'Topography'
Buffer with 'bufferLabel': 'Deflection'
Buffer with 'bufferLabel': 'Friction'
```

If only, for example, buffers with 'bufferLabel' equal to 'Topography' are desired, the method, **get_buffer** can be called:

```
>>> if os.path.isfile(path):
...     buffers = len(file_.get_buffer('Topography'))
...     print("Buffers with 'bufferLabel' == 'Topography': {}"
...           .format(buffers))
... else:
...     print("Buffers with 'bufferLabel' == 'Topography': 2")
Buffers with 'bufferLabel' == 'Topography': 2
```

buffers

get_buffer(*label*)

Get the buffers that have the specified buffer label.

Parameters: **label** (*str*) – The desired buffer label of the buffer.
Returns: **buffers** (*list*) – The buffers that have the desired buffer label.

class magni.afm.types._util.**FileCollection**(*files*, *paths*)

Bases: **magni.afm.types._util.BaseClass**

Data class for collections of File instances with identical settings.

The settings are the following attributes: 'fileType', 'mode', 'xPixels', 'yPixels', 'xOffset', 'yOffset', 'xLength', 'yLength', 'scanSpeed', 'acMac', 'acACMode', 'plotType'.

The present class exposes the files of the collection through a read-only tuple property, **files** and the paths of these files through a read-only tuple property, **paths**.

Parameters:
- **files** (*list or tuple*) – The files of the collection.
- **paths** (*list or tuple*) – The paths of the files of the collection.

files

tuple

The files of the collection.

paths

tuple

The paths of the files of the collection.

See also:

BaseClass

Superclass of the present class.

Examples

No example .mi file collection is distributed with **magni**.

files

paths

magni.afm.types.image module

Module providing data container classes for .mi image files.

The classes of this module can be used either directly or indirectly through the **magni.afm.io** subpackage by loading an .mi image file.

Routine listings

Buffer(magni.afm.types.BaseClass)

Data class of the .mi image file buffers.

Image(magni.afm.types.File)

Data class of the .mi image files.

See also:

magni.afm.io

.mi file loading.

class magni.afm.types.image.**Buffer**(*attrs*, *data*)

Bases: **magni.afm.types._util.BaseClass**

Data class of the .mi image file buffers.

Parameters:
- **attrs** (*dict*) – The attributes of the buffer.
- **data** (*numpy.ndarray*) – The 2D data of the buffer.

apply_clipping

bool

A flag indicating if clipping should be applied to the data.

data

numpy.ndarray

The 2D data of the buffer.

See also:

magni.utils.types.BaseClass

Superclass of the present class.

Examples

A subclass of the present class is implicitly instantiated when loading, for example, the .mi file provided with the package:

```
>>> import os, magni
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> if os.path.isfile(path):
...     image = magni.afm.io.read_mi_file(path)
...     buffer_ = image.buffers[0]
```

This buffer can have a number of attributes including 'bufferLabel':

```
>>> if os.path.isfile(path):
...     print('{!r}'.format(buffer_.attrs['bufferLabel']))
... else:
...     print("'Topography'")
'Topography'
```

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

```
>>> if os.path.isfile(path):
...     print('Buffer data shape: {!r}'.format(
...     tuple(int(s) for s in buffer_.data.shape)))
... else:
...     print('Buffer data shape: (256, 256)')
Buffer data shape: (256, 256)
```

apply_clipping

Get the apply_clipping property of the buffer.

The clipping is specified by the 'DisplayOffset', 'DisplayRange', and 'filter' attributes of the buffer.

Returns: **apply_clipping** (*bool*) – A flag indicating if clipping should be applied to the data.

data

Get the data property of the buffer.

If apply_clipping is False, the data is returned as-is. Otherwise, the filtering specified by the 'filter' attribute of the buffer and the clipping specified by the 'DisplayOffset' and 'DisplayRange' attributes of the buffer are applied to the data before it is returned.

Returns: **data** (*numpy.ndarray*) – The 2D data of the buffer.

class magni.afm.types.image.**Image**(*attrs*, *buffers*)

Bases: magni.afm.types._util.File

Data class of the .mi image files.

Parameters:
- **attrs** (*dict*) – The attributes of the image.
- **buffers** (*list or tuple*) – The buffers of the image.

See also:

magni.utils.types.File

Superclass of the present class.

Examples

The present class is implicitly instantiated when loading, for example, the .mi file provided with the package:

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

This file has a number of buffers which each has the 'bufferLabel' attribute:

```
>>> if os.path.isfile(path):
...     for buffer_ in image.buffers[::2][:3]:
...         label = buffer_.attrs['bufferLabel']
...         print("Buffer with 'bufferLabel': {!r}".format(label))
... else:
...     for label in ('Topography', 'Deflection', 'Friction'):
...         print("Buffer with 'bufferLabel': {!r}".format(label))
Buffer with 'bufferLabel': 'Topography'
Buffer with 'bufferLabel': 'Deflection'
Buffer with 'bufferLabel': 'Friction'
```

magni.afm.types.spectroscopy module

Module providing data container classes for .mi spectroscopy files.

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

Routine listings

Buffer(magni.afmtypes.BaseClass)
Data class for .mi spectroscopy buffer.

Chunk(magni.afmtypes.BaseClass)
Data class for .mi spectroscopy chunk.

Grid(magni.afmtypes.BaseClass)
Data class for .mi spectroscopy grid.

Point(magni.afmtypes.BaseClass)
Data class for .mi spectroscopy point.

Spectroscopy(magni.afm.types.File)
Data class for .mi spectroscopy.

See also:

magni.afm.io
.mi file loading.

class magni.afm.types.spectroscopy.**Buffer**(*attrs*, *data*)

Bases: **magni.afm.types._util.BaseClass**

Data class of the .mi spectroscopy buffers.

Parameters:
- **attrs** (*dict*) – The attributes of the buffer.
- **data** (*list or tuple*) – The grids, points, or chunks of the buffer.

data

numpy.ndarray

The grids, points, or chunks of the buffer.

See also:

magni.utils.types.BaseClass
Superclass of the present class.

Examples

No example .mi spectroscopy file is distributed with magni.

data

class magni.afm.types.spectroscopy.**Chunk**(*attrs*, *data*)

Bases: **magni.afm.types._util.BaseClass**

Data class of the .mi spectroscopy chunks.

Parameters:
- **attrs** (*dict*) – The attributes of the chunk.
- **data** (*numpy.ndarray*) – The data of the chunk.

data

numpy.ndarray

The data of the chunk.

See also:

magni.utils.types.BaseClass

Superclass of the present class.

Examples

No example .mi spectroscopy file is distributed with magni.

data

sweep

Get the sweep property of the chunk.

The sweep property is the series of the entity which was swept.

Returns: **sweep** (*numpy.ndarray*) – The sweep property.

Notes

To reduce the memory footprint of chunks, the series does not exist explicitly, until it is requested.

time

Get the time property of the chunk.

Returns: **time** (*numpy.ndarray*) – The time property.

Notes

To reduce the memory footprint of chunks, the series does not exist explicitly, until it is requested.

class magni.afm.types.spectroscopy.**Grid**(*attrs*, *points*)

Bases: **magni.afm.types._util.BaseClass**

Data class of the .mi spectroscopy grids.

Parameters:
- **attrs** (*dict*) – The attributes of the grid.
- **points** (*list or tuple*) – The points of the grid.

points

tuple

The points of the grid.

See also:

magni.utils.types.BaseClass

Superclass of the present class.

Notes

The points are input and output as a 2D tuple of Point instances.

Examples

No example .mi spectroscopy file is distributed with magni.

points

class magni.afm.types.spectroscopy.**Point**(*attrs*, *chunks=()*)

Bases: **magni.afm.types._util.BaseClass**

Data class of the .mi spectroscopy points.

Parameters:
- **attrs** (*dict*) – The attributes of the point.
- **chunks** (*list or tuple, optional*) – The chunks of the point. (the default is (), which implies no chunks)

chunks

tuple

The chunks of the point.

See also:

magni.utils.types.BaseClass

Superclass of the present class.

Examples

No example .mi spectroscopy file is distributed with magni.

chunks

class magni.afm.types.spectroscopy.**Spectroscopy**(*attrs*, *buffers*)

Bases: **magni.afm.types._util.File**

Data class of the .mi spectroscopy files.

Parameters:
- **attrs** (*dict*) – The attributes of the image.
- **buffers** (*list or tuple*) – The buffers of the image.

See also:

magni.utils.types.File
Superclass of the present class.

Examples

No example .mi spectroscopy file is distributed with magni.

Submodules

magni.afm._config module

Module providing configuration options for the **magni.afm** subpackage.

See also:

magni.utils.config.Configger
The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config**. The configuration options are the following:

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

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:

```
>>> import os, numpy as np, magni
>>> from magni.afm.reconstruction import analyse
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> if os.path.isfile(path):
...     mi_file = magni.afm.io.read_mi_file(path)
...     mi_buffer = mi_file.get_buffer('Topography')[0]
...     mi_data = mi_buffer.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
>>> if os.path.isfile(path):
...     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:

```
>>> if os.path.isfile(path):
...     Psi = magni.imaging.dictionaries.get_DCT(mi_data.shape)
```

Finally, the actual example:

```
>>> if os.path.isfile(path):
...     print('MSE: {:.2f}, PSNR: {:.2f}'.format(*analyse(x, Phi, Psi)))
... else:
...     print('MSE: 0.24, PSNR: 6.22')
MSE: 0.24, PSNR: 6.22
```

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:

```
>>> import os, numpy as np, magni
>>> from magni.afm.reconstruction import reconstruct
>>> path = magni.utils.split_path(magni.__path__[0])[0]
>>> path = path + 'examples' + os.sep + 'example.mi'
>>> if os.path.isfile(path):
...     mi_file = magni.afm.io.read_mi_file(path)
...     mi_buffer = mi_file.get_buffer('Topography')[0]
...     mi_data = mi_buffer.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
>>> if os.path.isfile(path):
...     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:

```
>>> if os.path.isfile(path):
...     Psi = magni.imaging.dictionaries.get_DCT(mi_data.shape)
```

Finally, the actual example:

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

magni.cs package

Subpackage providing generic compressed sensing functionality.

Routine listings

indicators
: Module prodiving performance indicator determination functionality.

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 functionality.

Routine listings

config
: Configger 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._config module

Module providing configuration options for the phase_transition subpackage.

See also:

magni.utils.config.Configger
The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config**. The configuration options are the following:

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

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.]).

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

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

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.]).

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).

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.

See also:

magni.cs.phase_transition._config
Configuration options.

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:

```
>>> import numpy as np
>>> 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._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.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*, *legend_loc='upper left'*)

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).
- **legend_loc** (*str*) – Location of legend as a **matplotlib** legend location string (the default is 'upper left', which implies that the legend is placed in the upper left corner of the plot.)

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,

```
>>> import numpy as np
>>> 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,

```
>>> import numpy as np
>>> 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

it
: Subpackage providing implementations of Iterative Thresholding (IT).

iht
: Subpackage providing implementations of Iterative Hard Thresholding (IHT). (Deprecated)

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

Subpackages

magni.cs.reconstruction.iht package

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

Note: Deprecated in Magni 1.3.0. magni.cs.reconstruction.iht will be removed in a future version. Use the more general magni.cs.reconstruction.it instead.

Warning: Change of variable interpretation. In magni.cs.reconstruction.it the config variable *threshold_fixed* has a different interpretation than in magni.cs.reconstruction.iht.

Routine listings

config
: Configger providing configuration options for this subpackage.

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

Notes

The IHT reconstruction algorithm is described in [1]. The default configuration uses the False Alarm Rate heuristic described in [2].

References

[1] T. Blumensath and M.E. Davies, “Iterative Thresholding for Sparse Approximations”, *Journal of Fourier Analysis and Applications*, vol. 14, pp. 629-654, Sep. 2008.

[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.**run**(*y*, *A*)

Run the IHT reconstruction algorithm.

Note: Deprecated in Magni 1.3.0 **magni.cs.reconstruction.iht** will be removed in a future version. Use the more general **magni.cs.reconstruction.it** instead.

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.

Examples

For example, recovering a vector from random measurements

```
>>> import warnings
>>> import numpy as np
>>> from magni.cs.reconstruction.iht import run, config
>>> np.random.seed(seed=6021)
>>> A = 1 / np.sqrt(80) * np.random.randn(80, 200)
>>> alpha = np.zeros((200, 1))
>>> alpha[:10] = 1
>>> y = A.dot(alpha)
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     alpha_hat = run(y, A)
...
>>> np.set_printoptions(suppress=True)
>>> alpha_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(alpha_hat) > 1e-2).sum()
10
```

Or recovering the same only using a fixed threshold level:

```
>>> config.update({'threshold': 'oracle', 'threshold_fixed': 10./80})
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     alpha_hat_2 = run(y, A)
...
>>> alpha_hat_2[:12]
array([[ 0.99877706],
       [ 0.99931441],
       [ 0.9978366 ],
       [ 0.99944973],
       [ 1.00052762],
       [ 1.00033436],
       [ 0.99943286],
       [ 0.99952526],
       [ 0.99941578],
       [ 0.99942908],
       [ 0.        ],
       [ 0.        ]])
```

```
>>> (np.abs(alpha_hat_2) > 1e-2).sum()
10
```

Submodules

magni.cs.reconstruction.iht._config module

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

See also:

magni.cs.reconstruction._config.Configger

The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config**. The configuration options are the following:

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

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

precision_float : *{np.float, np.float16, np.float32, np.float64, np.float128}*
The floating point precision used for the computations (the default is np.float64).

threshold : *{'far', 'oracle'}*
The method for selecting the threshold value.

threshold_fixed : *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.it package

Subpackage providing implementations of Iterative Thresholding (IT).

Routine listings

config
Configger providing configuration options for this subpackage.

run(y, A)
Run the IT reconstruction algorithm.

Notes

Implementations of Iterative Hard Thresholding (IHT) [1], [2] as well as implementations of Iterative Soft Thresholding (IST) [3], [4] are available. It is also possible to configure the subpackage to use a model based approach as described in [5].

References

[1] T. Blumensath and M.E. Davies, "Iterative Thresholding for Sparse Approximations", *Journal of Fourier Analysis and Applications*, vol. 14, pp. 629-654, Sep. 2008.

[2] T. Blumensath and M.E. Davies, “Normalized Iterative Hard Thresholding: Guaranteed Stability and Performance”, *IEEE Journal Selected Topics in Signal Processing*, vol. 4, no. 2, pp. 298-309, Apr. 2010.
[3] I. Daubechies, M. Defrise, and C. D. Mol, “An Iterative Thresholding Algorithm for Linear Inverse Problems with a Sparsity Constraint”, *Communications on Pure and Applied Mathematics*, vol. 57, no. 11, pp. 1413-1457, Nov. 2004.
[4] 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.
[5] R.G. Baraniuk, V. Cevher, M.F. Duarte, and C. Hedge, “Model-Based Compressive Sensing”, *IEEE Transactions on Information Theory*, vol. 56, no. 4, pp. 1982-2001, Apr. 2010.

Submodules

magni.cs.reconstruction.it._algorithm module

Module providing the core Iterative Thresholding (IT) algorithm.

Routine listings

run(y, A)

Run the IT reconstruction algorithm.

See also:

magni.cs.reconstruction.it._config

Configuration options.

Notes

The default configuration of the IT algorithm provides the Iterative Hard Thresholding (IHT) algorithm [1] using the False Alarm Rate (FAR) heuristic from [2]. The algorithm may also be configured to act as the Iterative Soft Thresholding (IST) [3] algorithm with the soft threshold from [4].

References

[1] T. Blumensath and M.E. Davies, “Iterative Thresholding for Sparse Approximations”, *Journal of Fourier Analysis and Applications*, vol. 14, pp. 629-654, Sep. 2008.
[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.
[3] I. Daubechies, M. Defrise, and C. D. Mol, “An Iterative Thresholding Algorithm for Linear Inverse Problems with a Sparsity Constraint”, *Communications on Pure and Applied Mathematics*, vol. 57, no. 11, pp. 1413-1457, Nov. 2004.

[4] D.L. Donoho, “De-Noising by Soft-Thresholding”, *IEEE Transactions on Information Theory*, vol. 41, no. 3, pp. 613-627, May. 1995.

magni.cs.reconstruction.it._algorithm.**run**(*y*, *A*)

Run the IT reconstruction algorithm.

Parameters:
- **y** (*ndarray*) – The m x 1 measurement vector.
- **A** (*ndarray or magni.utils.matrices.{Matrix, MatrixCollection}*) – 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.it._config()
Configuration options.

magni.cs.reconstruction.it._step_size()
Step-size calculation.

magni.cs.reconstruction.it._threshold()
Threshold calculation.

magni.cs.reconstruction.it._threshold_operators()
Thresholding operators.

Notes

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 using IHT with the FAR heuristic

```
>>> import numpy as np, magni
>>> from magni.cs.reconstruction.it import run
>>> np.random.seed(seed=6021)
>>> A = 1 / np.sqrt(90) * np.random.randn(90, 200)
>>> alpha = np.zeros((200, 1))
>>> alpha[:10] = 1
>>> y = A.dot(alpha)
>>> alpha_hat = run(y, A)
>>> alpha_hat[:12]
array([[ 0.99748945],
       [ 1.00082074],
       [ 0.99726507],
       [ 0.99987834],
       [ 1.00025857],
       [ 1.00003266],
       [ 1.00021666],
       [ 0.99838884],
       [ 1.00018356],
       [ 0.99859105],
       [ 0.        ],
       [ 0.        ]])
>>> (np.abs(alpha_hat) > 1e-2).sum()
10
```

Or recover the same vector as above using IST with a fixed number of non-thresholded coefficients

```
>>> it_config = {'threshold_operator': 'soft', 'threshold': 'fixed',
... 'threshold_fixed': 10}
>>> magni.cs.reconstruction.it.config.update(it_config)
>>> alpha_hat = run(y, A)
>>> alpha_hat[:12]
array([[ 0.99822443],
       [ 0.99888724],
       [ 0.9982493 ],
       [ 0.99928642],
       [ 0.99964131],
       [ 0.99947346],
       [ 0.9992809 ],
       [ 0.99872624],
       [ 0.99916842],
       [ 0.99903033],
       [ 0.        ],
       [ 0.        ]])
>>> (np.abs(alpha_hat) > 1e-2).sum()
10
```

Or use a weighted IHT method with a fixed number of non-thresholded coefficietns to recover the above signal

```
>>> weights = np.linspace(1.0, 0.9, 200).reshape(200, 1)
>>> it_config = {'threshold_operator': 'weighted_hard',
... 'threshold_weights': weights}
>>> magni.cs.reconstruction.it.config.update(it_config)
>>> alpha_hat = run(y, A)
>>> alpha_hat[:12]
array([[ 0.99853888],
       [ 0.99886104],
       [ 0.99843149],
       [ 1.0000333 ],
       [ 1.00060273],
       [ 1.00035539],
       [ 0.99966219],
       [ 0.99912209],
       [ 0.99961541],
       [ 0.99952029],
       [ 0.        ],
       [ 0.        ]])
>>> (np.abs(alpha_hat) > 1e-2).sum()
10
>>> magni.cs.reconstruction.it.config.reset()
```

magni.cs.reconstruction.it._config module

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

See also:

magni.cs.reconstruction._config.Configger
The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config**. The configuration options are the following:

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

kappa : *{'fixed', 'adaptive'}*
The method used to calculate the step-size (relaxation) parameter kappa.

kappa_fixed : *float*
The step-size (relaxation parameter) used in the algorithm when a fixed step-size is used (the default is 0.65).

precision_float : *{np.float, np.float16, np.float32, np.float64, np.float128}*
The floating point precision used for the computations (the default is np.float64).

threshold : *{'far', 'fixed'}*
The method used for calculating the threshold value.

threshold_fixed : *int*
The number of non-zero coefficients in the signal vector when this number is assumed fixed.

threshold_operator : *{'hard', 'soft', 'weighted_hard', 'weighted_soft', 'none'}*

The threshold operator used in the backprojection step.

threshold_weights : *ndarray*

Array of weights to be used in one of the weighted threshold operators (the default is array([[1]]), which implies that all coefficients are weighted equally).

tolerance : *float*

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

warm_start : *ndarray*

The initial guess of the solution vector (the default is None, which implies that a vector of zeros is used).

magni.cs.reconstruction.it._step_size module

Module providing functions for calculating the step-size (relaxation parameter) used in the Iterative Thresholding algorithms.

Routine listings

calculate_using_adaptive(var)

Calculate a step-size value in an 'adaptive' way.

calculate_using_fixed(var)

Calculate the fixed step-size value.

get_function_handle(method)

Return a function handle to a given calculation method.

magni.cs.reconstruction.it._step_size.**wrap_calculate_using_adaptive**(*var*)

Arguments wrapper for culculate_using_adaptive.

Calculate a step-size value in an 'adaptive' way.

magni.cs.reconstruction.it._step_size.**wrap_calculate_using_fixed**(*var*)

Arguments wrapper for calculate_using_fixed.

Calculate a fixed step-size value.

magni.cs.reconstruction.it._step_size.**get_function_handle**(*method*, *var*)

Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the step-size method.

Returns: **f_handle** (*function*) – Handle to calculation *method* defined in this globals scope.

magni.cs.reconstruction.it._threshold module

Module providing functions for calculating a threshold (level) used in Iterative Threshold algorithms.

Routine listings

calculate_far(delta)

Calculate the optimal False Acceptance Rate for a given indeterminacy.

calculate_using_far(var)

Calculate a threshold level using the FAR heuristic.

calculate_using_fixed(var)

Calculate a threshold level using a given fixed support size.

get_function_handle(method)

Return a function handle to a given calculation method.

magni.cs.reconstruction.it._threshold.**calculate_far**(*delta*, *it_algorithm*)

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.
- **it_algorithm** (*{IHT, ITS}*) – The iterative thresholding algorithm to calculate the FAR for.

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” [1] 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

[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.it._threshold.**wrap_calculate_using_far**(*var*)

Arguments wrapper for calculate_using_far.

Calculate a threshold level using the FAR heuristic.

magni.cs.reconstruction.it._threshold.**wrap_calculate_using_fixed**(*var*)

Arguments wrapper for calculate_using_fixed.

Calculate a threshold level using a given fixed support size.

magni.cs.reconstruction.it._threshold.**get_function_handle**(*method*, *var*)

Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the threshold method.

Returns: **f_handle** (*function*) – Handle to calculation *method* defined in this globals scope.

magni.cs.reconstruction.it._threshold_operators module

Module providing thresholding operators used in Iterative Thresholding algorithms.

Routine listings

get_function_handle(method)

Return a function handle to a given threshold operator.

threshold_hard(var)

The hard threshold operator.

threshold_none(var)

The “no” threshold operator.

threshold_soft(var)

The soft threshold operator.

threshold_weighted_hard(var)

The weighted hard threshold operator.

threshold_weighted_soft(var)

The weighted soft threshold operator.

magni.cs.reconstruction.it._threshold_operators.**get_function_handle**(*method*)

Return a function handle to a given threshold operator method.

Parameters: **method** (*str*) – Identifier of the threshold operator to return a handle to.

Returns: **f_handle** (*function*) – Handle to threshold method defined in this globals scope.

magni.cs.reconstruction.it._threshold_operators.**threshold_hard**(*var*)

Threshold the entries of a vector using the hard threshold.

Parameters: **var** (*dict*) – Local variables used in the threshold operation.

Notes

This threshold operation works “in-line” on the variables in *var*. Hence, this function does not return anything.

Examples

For example, thresholding a vector of values between -1 and 1

```
>>> import copy, numpy as np, magni
>>> from magni.cs.reconstruction.it._threshold_operators import (
... threshold_hard)
>>> var = {'alpha': np.linspace(-1, 1, 10), 'threshold': 0.4}
>>> threshold_hard(copy.copy(var))
>>> var['alpha']
array([-1.        , -0.77777778, -0.55555556,  0.        ,  0.        ,
        0.        ,  0.        ,  0.55555556,  0.77777778,  1.        ])
```

magni.cs.reconstruction.it._threshold_operators.**threshold_none**(*var*)

Do not threshold the entries of a vector.

Parameters: **var** (*dict*) – Local variables used in the threshold operation.

Notes

This is a dummy threshold operation that does nothing.

magni.cs.reconstruction.it._threshold_operators.**threshold_soft**(*var*)

Threshold the entries of a vector using the soft threshold.

Parameters: **var** (*dict*) – Local variables used in the threshold operation.

Notes

This threshold operation works “in-line” on the variables in *var*. Hence, this function does not return anything.

Examples

For example, thresholding a vector of values between -1 and 1

```
>>> import copy, numpy as np, magni
>>> from magni.cs.reconstruction.it._threshold_operators import (
... threshold_soft)
>>> var = {'alpha': np.linspace(-1, 1, 10), 'threshold': 0.4}
>>> threshold_soft(copy.copy(var))
>>> var['alpha']
array([-0.6       , -0.37777778, -0.15555556,  0.        ,  0.        ,
        0.        ,  0.        ,  0.15555556,  0.37777778,  0.6       ])
```

magni.cs.reconstruction.it._threshold_operators.**threshold_weighted_hard**(*var*)

Threshold the entries of a vector using a weighted hard threshold.

Parameters: **var** (*dict*) – Local variables used in the threshold operation.

Notes

This threshold operation works “in-line” on the variables in *var*. Hence, this function does not return anything.

Examples

For example, thresholding a vector of values between -1 and 1

```
>>> import copy, numpy as np, magni
>>> from magni.cs.reconstruction.it._threshold_operators import (
... threshold_weighted_hard)
>>> var = {'alpha': np.linspace(-1, 1, 10), 'threshold': 0.4,
... 'threshold_weights': 0.7 * np.ones(10)}
>>> threshold_weighted_hard(copy.copy(var))
>>> var['alpha']
array([-1.        , -0.77777778,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.77777778,  1.        ])
```

magni.cs.reconstruction.it._threshold_operators.**threshold_weighted_soft**(*var*)

Threshold the entries of a vector using a weighted soft threshold.

Parameters: **var** (*dict*) – Local variables used in the threshold operation.

Notes

This threshold operation works “in-line” on the variables in *var*. Hence, this function does not return anything.

Examples

For example, thresholding a vector of values between -1 and 1

```
>>> import copy, numpy as np, magni
>>> from magni.cs.reconstruction.it._threshold_operators import (
... threshold_weighted_soft)
>>> var = {'alpha': np.linspace(-1, 1, 10), 'threshold': 0.4,
... 'threshold_weights': 0.7 * np.ones(10)}
>>> threshold_weighted_soft(copy.copy(var))
>>> var['alpha']
array([-0.42857143, -0.20634921,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.20634921,  0.42857143])
```

magni.cs.reconstruction.it._util module

Module providing miscellaneous utility functions for the different implementations of Iterative Thresholding (IT).

Routine listings

get_methods(module)

Extract relevant methods from module.

def _get_operators(module)

Extract relevant operators from *module*.

magni.cs.reconstruction.it._util.**_get_methods**(*module*)

Extract relevant methods from *module*.

Parameters:	**module** (*str*) – The name of the module from which the methods should be extracted.
Returns:	**methods** (*tuple*) – The names of the relevant methods from *module*.

Notes

Looks for functions with names starting with 'wrap_calculate_using_' in the module: _"module".

magni.cs.reconstruction.it._util.**_get_operators**(*module*)

Extract relevant operators from *module*.

Parameters:	**module** (*str*) – The name of the module from which the operators should be extracted.
Returns:	**methods** (*tuple*) – The names of the relevant operators from *module*.

Notes

Looks for functions with names starting with 'threshold_' in the module _"module".

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

Configger providing configuration options for this subpackage.

run(y, A)

Run the specified SL0 reconstruction algorithm.

Notes

See **_algorithm** for documentation of **run**.

Implementations of the original SL0 reconstruction algorithm [1] and a modified Sl0 reconstruction algorithm [3] are available. It is also possible to configure the subpackage to provide customised versions of the SL0 reconstruction algorithm. The projection algorithm [1] is used for small delta (< 0.55) whereas the contraint elimination algorithm [2] is used for large delta (>= 0.55) which merely affects the computation time.

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] 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._L_start module

Module providing functions for calculating the starting value of L (the number of gradient descent iterations for each sigma) used in the Smoothed l0 algorithms.

Routine listings

calculate_using_fixed(var)

Calculate the fixed L value.

calculate_using_geometric(var)

Calculate an L value in a ‘geometric’ way.

get_function_handle(method)

Return a function handle to a given calculation method.

magni.cs.reconstruction.sl0._L_start.**wrap_calculate_using_fixed**(*var*)

Arguments wrapper for calculate_using_fixed.

magni.cs.reconstruction.sl0._L_start.**wrap_calculate_using_geometric**(*var*)

Arguments wrapper for calculate_using_geometric.

magni.cs.reconstruction.sl0._L_start.**get_function_handle**(*method*, *var*)

Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the L method.

Returns: **f_handle** (*function*) – Handle to the calculation method defined in this globals scope.

magni.cs.reconstruction.sl0._L_update module

Module providing functions for calculating the updated value of L (the number of gradient descent iterations for each sigma) used in the Smoothed l0 algorithms.

Routine listings

calculate_using_fixed(var)
Calculate the updated, fixed L value.

calculate_using_geometric(var)
Calculate an updated L value in a ‘geometric’ way.

get_function_handle(method)
Return a function handle to a given calculation method.

magni.cs.reconstruction.sl0._L_update.**wrap_calculate_using_fixed**(*var*)
Arguments wrapper for calculate_using_fixed.

magni.cs.reconstruction.sl0._L_update.**wrap_calculate_using_geometric**(*var*)
Arguments wrapper for calculate_using_geometric.

magni.cs.reconstruction.sl0._L_update.**get_function_handle**(*method*, *var*)
Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the L update method.

Returns: **f_handle** (*function*) – Handle to the calculation method defined in this globals scope.

magni.cs.reconstruction.sl0._algorithm module

Module providing the core Smoothed l0 (SL0) algorithm.

Routine listings

run(y, A)
Run the SL0 reconstruction algoritm.

See also:

magni.cs.reconstruction.sl0._config
Configuration options.

Notes

Implementations of the original SL0 reconstruction algorithm [1] and a modified Sl0 reconstruction algorithm [3] are available. It is also possible to configure the subpackage to provide customised versions of the SL0 reconstruction algorithm. The projection algorithm [1] is used for small delta (< 0.55) whereas the contraint elimination algorithm [2] is used for large delta (>= 0.55) which merely affects the computation time.

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] 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.

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

Run the SL0 reconstruction algorithm.

Parameters:
- **y** (*ndarray*) – The m x 1 measurement vector.
- **A** (*ndarray or magni.utils.matrices.{Matrix, MatrixCollection}*) – 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.

Notes

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 using the original SL0 reconstruction algorithm

```
>>> import numpy as np, magni
>>> from magni.cs.reconstruction.sl0 import run
>>> np.set_printoptions(suppress=True)
>>> magni.cs.reconstruction.sl0.config['L'] = 'fixed'
>>> magni.cs.reconstruction.sl0.config['mu'] = 'fixed'
>>> magni.cs.reconstruction.sl0.config['sigma_start'] = 'fixed'
>>> np.random.seed(seed=6021)
>>> A = 1 / np.sqrt(80) * np.random.randn(80, 200)
>>> alpha = np.zeros((200, 1))
>>> alpha[:10] = 1
>>> y = A.dot(alpha)
>>> alpha_hat = run(y, A)
>>> alpha_hat[:12]
array([[ 0.99993202],
       [ 0.99992793],
       [ 0.99998107],
       [ 0.99998105],
       [ 1.00005882],
       [ 1.00000843],
       [ 0.99999138],
       [ 1.00009479],
       [ 0.99995889],
       [ 0.99992509],
       [-0.00001509],
       [ 0.00000275]])
>>> (np.abs(alpha_hat) > 1e-2).sum()
10
```

Or recover the same vector as above using the modified SL0 reconstruction algorithm

```
>>> magni.cs.reconstruction.sl0.config['L'] = 'geometric'
>>> magni.cs.reconstruction.sl0.config['mu'] = 'step'
>>> magni.cs.reconstruction.sl0.config['sigma_start'] = 'reciprocal'
>>> alpha_hat = run(y, A)
>>> alpha_hat[:12]
array([[ 0.9999963 ],
       [ 1.00000119],
       [ 1.00000293],
       [ 0.99999661],
       [ 1.00000021],
       [ 0.9999951 ],
       [ 1.00000103],
       [ 1.00000662],
       [ 1.00000404],
       [ 0.99998937],
       [-0.00000075],
       [ 0.00000037]])
>>> (np.abs(alpha_hat) > 1e-2).sum()
10
```

magni.cs.reconstruction.sl0._config module

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

See also:

magni.cs.reconstruction._config.Configger
The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config**. The configuration options are the following:

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

L : *{'geometric', 'fixed'}*
: The method for selecting the maximum number of gradient descent iterations for each sigma.

L_fixed : *int*
: The value used for L if using the 'fixed' method (the default is 2.0).

L_geometric_ratio : *float*
: The common ratio used for the L update if using the 'geometric' method (the default is 2.0).

L_geometric_start : *float*
: The starting value used for the L if using the 'geometric' method (the default is 2.0).

mu : *{'step', 'fixed'}*
: The method for selecting the relative step-size used in gradient descent iteration.

mu_fixed : *float*
: The value used for mu if using the 'fixed' method (the default is 1.0).

mu_step_end : *float*
: The value used for mu for the last iterations if using the 'step' method (the default is 1.5).

mu_step_iteration : *int*
: The iteration where the value used for mu changes if using the 'step' method

mu_step_start : *float*
: The value used for mu for the first iterations if using the 'step' method (the default is 0.001).

precision_float : *{np.float, np.float16, np.float32, np.float64, np.float128}*
: The floating point precision used for the computations (the default is np.float64).

sigma_geometric : *float*
: The common ratio used for the sigma update (the default is 0.7).

sigma_start : *{'reciprocal', 'fixed'}*
: The method for selecting the starting value of sigma.

sigma_start_fixed : *float*
: The fixed factor multiplied onto the maximum coefficient of a least squares solution to obtain the value if using the 'fixed' method (the default is 2.0).

sigma_start_reciprocal : *float*
: The constant in the factor $\frac{1}{constant \cdot \delta}$ multiplied onto the maximum coefficient of a least squares solution to obtain the value if using the 'reciprocal' method (the default is

2.75).

sigma_stop_fixed : *float*

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

magni.cs.reconstruction.sl0._mu_start module

Module providing functions for calculating the starting value of mu (the relative step-size used in the gradient descent iteration) used in the Smoothed l0 algorithms.

Routine listings

calculate_using_fixed(var)

Calculate the fixed mu value.

calculate_using_step(var)

Calculate an mu value in a 'step' way.

get_function_handle(method)

Return a function handle to a given calculation method.

magni.cs.reconstruction.sl0._mu_start.**wrap_calculate_using_fixed**(*var*)

Arguments wrapper for calculate_using_fixed.

magni.cs.reconstruction.sl0._mu_start.**wrap_calculate_using_step**(*var*)

Arguments wrapper for calculate_using_step.

magni.cs.reconstruction.sl0._mu_start.**get_function_handle**(*method*, *var*)

Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the mu method.

Returns: **f_handle** (*function*) – Handle to the calculation method defined in this globals scope.

magni.cs.reconstruction.sl0._mu_update module

Module providing functions for calculating the updated value of mu (the relative step-size used in the gradient descent iteration) used in the Smoothed l0 algorithms.

Routine listings

calculate_using_fixed(var)

Calculate the updated, fixed mu value.

calculate_using_step(var)

Calculate an updated mu value in a 'step' way.

get_function_handle(method)

Return a function handle to a given calculation method.

magni.cs.reconstruction.sl0._mu_update.**wrap_calculate_using_fixed**(*var*)

Arguments wrapper for calculate_using_fixed.

magni.cs.reconstruction.sl0._mu_update.**wrap_calculate_using_step**(*var*)

Arguments wrapper for calculate_using_step.

magni.cs.reconstruction.sl0._mu_update.**get_function_handle**(*method*, *var*)

Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the mu update method.

Returns: **f_handle** (*function*) – Handle to the calculation method defined in this globals scope.

magni.cs.reconstruction.sl0._sigma_start module

Module providing functions for calculating the starting value of sigma used in the Smoothed l0 algorithms.

Routine listings

calculate_using_fixed(var)

Calculate the fixed sigma value.

calculate_using_reciprocal(var)

Calculate a sigma value in a ‘reciprocal’ way.

get_function_handle(method)

Return a function handle to a given calculation method.

magni.cs.reconstruction.sl0._sigma_start.**wrap_calculate_using_fixed**(*var*)

Arguments wrapper for calculate_using_fixed.

magni.cs.reconstruction.sl0._sigma_start.**wrap_calculate_using_reciprocal**(*var*)

Arguments wrapper for calculate_using_reciprocal.

magni.cs.reconstruction.sl0._sigma_start.**get_function_handle**(*method*, *var*)

Return a function handle to a given calculation method.

Parameters:
- **method** (*str*) – Identifier of the calculation method to return a handle to.
- **var** (*dict*) – Local variables needed in the sigma method.

Returns: **f_handle** (*function*) – Handle to the calculation method defined in this globals scope.

Submodules

magni.cs.reconstruction._config module

Module providing a CS reconstruction algorithm adapted configger subclass.

Routine listings

Configger(magni.utils.config.Configger)
Provide functionality to access a set of configuration options.

Notes

This module does not itself contain any configuration options and thus has no access to any configuration options unlike the other config modules of **magni**.

class magni.cs.reconstruction._config.**Configger**(*params*, *valids*)

Bases: **magni.utils.config.Configger**

Provide functionality to access a set of configuration options.

The present class redefines the methods for retrieving configuration parameters in order to ensure the desired precision of the floating point parameter values.

Parameters:
- **params** (*dict*) – The configuration options and their default values.
- **valids** (*dict*) – The validation schemes of the configuration options.

property

See also:

magni.utils.config.Configger
Superclass of the present class.

__getitem__(*name*)
Get the value of a configuration parameter.

Parameters: **name** (*str*) – The name of the parameter.
Returns: **value** (*None*) – The value of the parameter.

Notes

If the value is a floating point value then that value is typecast to the desired precision.

Submodules

magni.cs.indicators module

Module prodiving performance indicator determination functionality.

Routine listings

calculate_coherence(Phi, Psi, norm=None)

Calculate the coherence of the Phi Psi matrix product.

calculate_mutual_coherence(Phi, Psi, norm=None)

Calculate the mutual coherence of Phi and Psi.

calculate_relative_energy(Phi, Psi, method=None)

Calculate the relative energy of Phi Psi matrix product atoms.

magni.cs.indicators.**calculate_coherence**(*Phi*, *Psi*, *norm=None*)

Calculate the coherence of the Phi Psi matrix product.

In the context of Compressive Sensing, coherence usually refers to the maximum absolute correlation between two columns of the Phi Psi matrix product. This function allows the usage of a different normalised norm where the infinity-norm yields the usual case.

Parameters:
- **Phi** (*magni.utils.matrices.Matrix or numpy.ndarray*) – The measurement matrix.
- **Psi** (*magni.utils.matrices.Matrix or numpy.ndarray*) – The dictionary matrix.
- **norm** (*int or float*) – The normalised norm used for the calculation (the default value is None which implies that the 0-, 1-, 2-, and infinity-norms are returned).

Returns: **coherence** (*float or dict*) – The coherence value(s).

Notes

If *norm* is None, the function returns a dict containing the coherence using the 0-, 1-, 2-, and infinity-norms. Otherwise, the function returns the coherence using the specified norm.

The coherence is calculated as:

$$\left(\frac{1}{n^2 - n} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(\frac{|\Psi_{:,i}^T \Phi^T \Phi \Psi_{:,j}|}{||\Phi \Psi_{:,i}||_2 ||\Phi \Psi_{:,j}||_2} \right)^{\text{norm}} \right)^{\frac{1}{\text{norm}}}$$

where *n* is the number of columns in *Psi*. In the case of the 0-norm, the coherence is calculated as:

$$\frac{1}{n^2 - n} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(\frac{|\Psi_{:,i}^T \Phi^T \Phi \Psi_{:,j}|}{||\Phi \Psi_{:,i}||_2 ||\Phi \Psi_{:,j}||_2} \right)$$

where $(\ )$ is 1 if *a* is non-zero and 0 otherwise. In the case of the infinity-norm, the coherence is calculated as:

$$\underset{i,j}{\mathrm{m}} \underset{i \neq j}{1,} ,n \left(\frac{|\Psi_{:,i}^T \Phi^T \Phi \Psi_{:,j}|}{||\Phi \Psi_{:,i}||_2 ||\Phi \Psi_{:,j}||_2} \right)$$

Examples

For example,

```
>>> import numpy as np
>>> import magni
>>> from magni.cs.indicators import calculate_coherence
>>> Phi = np.zeros((5, 9))
>>> Phi[0, 0] = Phi[1, 2] = Phi[2, 4] = Phi[3, 6] = Phi[4, 8] = 1
>>> Psi = magni.imaging.dictionaries.get_DCT((3, 3))
>>> for item in sorted(calculate_coherence(Phi, Psi).items()):
...     print('{}-norm: {:.3f}'.format(*item))
0-norm: 0.222
1-norm: 0.141
2-norm: 0.335
inf-norm: 1.000
```

The above values can be calculated individually by specifying a norm:

```
>>> for norm in (0, 1, 2, np.inf):
...     value = calculate_coherence(Phi, Psi, norm=norm)
...     print('{}-norm: {:.3f}'.format(norm, value))
0-norm: 0.222
1-norm: 0.141
2-norm: 0.335
inf-norm: 1.000
```

magni.cs.indicators.**calculate_mutual_coherence**(*Phi*, *Psi*, *norm=None*)

Calculate the mutual coherence of Phi and Psi.

In the context of Compressive Sensing, mutual coherence usually refers to the maximum absolute correlation between two columns of Phi and Psi. This function allows the usage of a different normalised norm where the infinity-norm yields the usual case.

Parameters:
- **Phi** (*magni.utils.matrices.Matrix or numpy.ndarray*) – The measurement matrix.
- **Psi** (*magni.utils.matrices.Matrix or numpy.ndarray*) – The dictionary matrix.
- **norm** (*int or float*) – The normalised norm used for the calculation (the default value is None which implies that the 0-, 1-, 2-, and infinity-norms are returned).

Returns: **mutual_coherence** (*float or dict*) – The mutual_coherence value(s).

Notes

If *norm* is None, the function returns a dict containing the mutual coherence using the 0-, 1-, 2-, and infinity-norms. Otherwise, the function returns the mutual coherence using the specified norm.

The mutual coherence is calculated as:

$$\left(\frac{1}{n} \sum_{i=1}^{m} \sum_{j=1}^{n} |\Phi_{i,:} \Psi_{:,j}|^{\text{norm}} \right)^{\frac{1}{\text{norm}}}$$

where *m* is the number of rows in *Phi* and *n* is the number of columns in *Psi*. In the case of the 0-norm, the mutual coherence is calculated as:

$$\frac{1}{n} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(|\Phi_{i,:} \Psi_{:,j}| \right)$$

where $\left(\right)$ is 1 if *a* is non-zero and 0 otherwise. In the case of the infinity-norm, the mutual coherence is calculated as:

$$\max_{i = 1, \ldots, m} \max_{j = 1, \ldots, n} |\Phi_{i,:} \Psi_{:,j}|$$

Examples

For example,

```
>>> import numpy as np
>>> import magni
>>> from magni.cs.indicators import calculate_mutual_coherence
>>> Phi = np.zeros((5, 9))
>>> Phi[0, 0] = Phi[1, 2] = Phi[2, 4] = Phi[3, 6] = Phi[4, 8] = 1
>>> Psi = magni.imaging.dictionaries.get_DCT((3, 3))
>>> for item in sorted(calculate_mutual_coherence(Phi, Psi).items()):
...     print('{}-norm: {:.3f}'.format(*item))
0-norm: 0.889
1-norm: 0.298
2-norm: 0.333
inf-norm: 0.667
```

The above values can be calculated individually by specifying a norm:

```
>>> for norm in (0, 1, 2, np.inf):
...     value = calculate_mutual_coherence(Phi, Psi, norm=norm)
...     print('{}-norm: {:.3f}'.format(norm, value))
0-norm: 0.889
1-norm: 0.298
2-norm: 0.333
inf-norm: 0.667
```

magni.cs.indicators.**calculate_relative_energy**(*Phi*, *Psi*, *method=None*)

Calculate the relative energy of Phi Psi matrix product atoms.

Parameters:

- **Phi** (*magni.utils.matrices.Matrix or numpy.ndarray*) – The measurement matrix.
- **Psi** (*magni.utils.matrices.Matrix or numpy.ndarray*) – The dictionary matrix.
- **method** (*str*) – The method used for summarising the relative energies of the Phi Psi matrix product atoms.

Returns: **relative_energy** (*float or dict*) – The relative_energy summary value(s).

Notes

The summary *method* used is either 'mean' for mean value, 'std' for standard deviation, 'min' for minimum value, or 'diff' for difference between the maximum and minimum values.

If *method* is None, the function returns a dict containing all of the above summaries. Otherwise, the function returns the specified summary.

The relative energies, which are summarised by the given *method*, are calculated as:

$$\left[\frac{||\mathbf{\Phi\Psi}_{:,1}||_2}{||\mathbf{\Psi}_{:,1}||_2}, \quad , \frac{||\mathbf{\Phi\Psi}_{:,n}||_2}{||\mathbf{\Psi}_{:,n}||_2} \right]^T$$

where *n* is the number of columns in *Psi*.

Examples

For example,

```
>>> import numpy as np
>>> import magni
>>> from magni.cs.indicators import calculate_relative_energy
>>> Phi = np.zeros((5, 9))
>>> Phi[0, 0] = Phi[1, 2] = Phi[2, 4] = Phi[3, 6] = Phi[4, 8] = 1
>>> Psi = magni.imaging.dictionaries.get_DCT((3, 3))
>>> for item in sorted(calculate_relative_energy(Phi, Psi).items()):
...     print('{}: {:.3f}'.format(*item))
diff: 0.423
mean: 0.735
min: 0.577
std: 0.126
```

The above values can be calculated individually by specifying a norm:

```
>>> for method in ('mean', 'std', 'min', 'diff'):
...     value = calculate_relative_energy(Phi, Psi, method=method)
...     print('{}: {:.3f}'.format(method, value))
mean: 0.735
std: 0.126
min: 0.577
diff: 0.423
```

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.

double_mirror(img, fftstyle=True)
: Function to mirror image in both the vertical and horisontal axes.

get_inscribed_masks(img, as_vec=False)
: Function to get inscribed masks covering the image.

Notes

See **_util** for documentation of **mat2vec**, **vec2mat**, **double_mirror**, and **get_inscribed_masks**.

Subpackages

magni.imaging.dictionaries package

Subpackage providing functionality for image dictionary manipulations.

Routine listings

get_DCT(shape, overcomplete_shape=None)
: Get the DCT fast operation dictionary for the given image shape.

get_DFT(shape, overcomplete_shape=None)
: Get the DFT fast operation dictionary for the given image shape.

analysis
: Module providing functionality to analyse dictionaries.

utils

Module providing utility functions for the dictionaries subpackage.

Submodules

magni.imaging.dictionaries._fastops module

Module providing functionality related to linear transformations.

Routine listings

dct2(x, mn_tuple, overcomplete_mn_tuple=None)

2D discrete cosine transform.

idct2(x, mn_tuple, overcomplete_mn_tuple=None)

2D inverse discrete cosine tranform.

dft2(x, mn_tuple, overcomplete_mn_tuple=None)

2D discrete Fourier transform.

idft2(x, mn_tuple, overcomplete_mn_tuple=None)

2D inverse discrete Fourier transform.

magni.imaging.dictionaries._fastops.**dct2**(*x*, *mn_tuple*, *overcomplete_mn_tuple=None*)

Apply the 2D Discrete Cosine Transform (DCT).

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.
- **mn_tuple** (*tuple of int*) – *(m, n) - m* number of rows in the matrix represented by *x*; *n* number of columns in the matrix.
- **overcomplete_mn_tuple** (*tuple of int, optional*) – *(mo, no) - mo* number of rows in the matrix represented by the function's output; *no* number of columns in the matrix.

Returns: **alpha** (*ndarray*) – When *overcomplete_mn_tuple* is omitted: an m*n x 1 vector of coefficients scaled such that x = idct2(dct2(x)). When *overcomplete_mn_tuple* is supplied: an mo*no x 1 vector of coefficients scaled such that x = idct2(dct2(x)).

See also:

scipy.fftpack.dct()

1D DCT

Notes

The overcomplete transform feature invoked by supplying *overcomplete_mn_tuple* is only available for SciPy >= 0.16.1.

The 2D DCT uses the seperability property of the 1D DCT.

http://stackoverflow.com/questions/14325795/scipys-fftpack-dct-and-idct Including the reshape operation, the full transform is: dct(dct(x.reshape(n, m).T).T.T.reshape(m * n, 1)

magni.imaging.dictionaries._fastops.**idct2**(*x*, *mn_tuple*, *overcomplete_mn_tuple=None*)

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

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 vector representing the associated column-stacked matrix. When *overcomplete_mn_tuple* is omitted: the shape must be m*n x 1. When *overcomplete_mn_tuple* is supplied: the shape must be mo*no x 1.
- **mn_tuple** (*tuple of int*) – *(m, n)* - *m* number of rows in the associated matrix, *n* number of columns in the associated matrix. When *overcomplete_mn_tuple* is supplied, this is the shape of the function's output.
- **overcomplete_mn_tuple** (*tuple of int, optional*) – *(mo, no)* - *mo* number of rows in the associated matrix, *no* number of columns in the associated matrix. When supplied, this is the shape of the function's input.

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

See also:

scipy.fftpack.idct()

1D inverse DCT

Notes

The overcomplete transform feature invoked by supplying *overcomplete_mn_tuple* is only available for SciPy >= 0.16.1.

magni.imaging.dictionaries._fastops.**dft2**(*x*, *mn_tuple*, *overcomplete_mn_tuple=None*)

Apply the 2D Discrete Fourier Transform (DFT).

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.
- **mn_tuple** (*tuple of int*) – *(m, n)* - *m* number of rows in the matrix represented by *x*; *n* number of columns in the matrix.
- **overcomplete_mn_tuple** (*tuple of int, optional*) – *(mo, no)* - *mo* number of rows in the matrix represented by the function's output; *no* number of columns in the matrix.

Returns: *ndarray* – When *overcomplete_mn_tuple* is omitted: an m*n x 1 vector of coefficients scaled such that x = idft2(dft2(x)). When *overcomplete_mn_tuple* is supplied: an mo*no x 1 vector of coefficients scaled such that x = idft2(dft2(x)).

See also:

numpy.fft.fft2()

The underlying 2D FFT used to compute the 2D DFT.

Notes

This is a normalised DFT, i.e. a normalisation constant of $\sqrt{n}$ is used.

magni.imaging.dictionaries._fastops.**idft2**(*x*, *mn_tuple*, *overcomplete_mn_tuple=None*)

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

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.
- **mn_tuple** (*tuple of int*) – *(m, n)* - *m* number of rows in the associated matrix, *n* number of columns in the associated matrix. When *overcomplete_mn_tuple* is supplied, this is the shape of the function's output.
- **overcomplete_mn_tuple** (*tuple of int, optional*) – *(mo, no)* - *mo* number of rows in the associated matrix, *no* number of columns in the associated matrix. When supplied, this is the shape of the function's input.

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

See also:

numpy.fft.ifft2()

The underlying 2D iFFT used to compute the 2D iDFT.

Notes

This is a normalised iDFT, i.e. a normalisation constant of $\sqrt{n}$ is used.

magni.imaging.dictionaries._fastops.**_validate_transform_fwd**(*x*, *mn_tuple*, *overcomplete_mn_tuple=None*)

Validate a 2D transform.

Parameters:
- **x** (*ndarray*) – The m*n x 1 vector representing the associated column stacked matrix.
- **mn_tuple** (*tuple of int*) – *(m, n)* - *m* number of rows in the matrix represented by *x*; *n* number of columns in the matrix.
- **overcomplete_mn_tuple** (*tuple of int, optional*) – *(mo, no)* - *mo* number of rows in the matrix represented by the function's output; *no* number of columns in the matrix.

magni.imaging.dictionaries._fastops.**_validate_transform_bwd**(*x*, *mn_tuple*, *overcomplete_mn_tuple=None*)

Validate a 2D transform.

Parameters:
- **x** (*ndarray*) – The m*n x 1 vector representing the associated column stacked matrix.
- **mn_tuple** (*tuple of int*) – *(m, n)* - *m* number of rows in the associated matrix, *n* number of columns in the associated matrix.
- **overcomplete_mn_tuple** (*tuple of int, optional*) – *(mo, no)* - *mo* number of rows in the associated matrix, *no* number of columns in the associated matrix.

magni.imaging.dictionaries._matrices module

Module providing fast linear operations wrapped in matrix emulators.

Routine listings

get_DCT(shape, overcomplete_shape=None)

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

get_DFT(shape, overcomplete_shape=None)

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

See also:

magni.imaging.dictionaries._fastops

Fast linear operations.

magni.utils.matrices

Matrix emulators.

magni.imaging.dictionaries._matrices.**get_DCT**(*shape*, *overcomplete_shape=None*)

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

Parameters:
- **shape** (*list or tuple*) – The shape of the image for which the dictionary is the DCT dictionary.
- **overcomplete_shape** (*list or tuple, optional*) – The shape of the (overcomplete) frequency domain for the DCT dictionary. The entries must be greater than or equal to the corresponding entries in *shape*.

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:

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

Perform DCT in the ordinary way:

```
>>> dct_normal = magni.imaging.dictionaries._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
```

Compute the overcomplete transform (and back again) and check that the resulting image is identical to the original. Notice how this example first ensures that the necessary version of SciPy is available:

```
>>> from pkg_resources import parse_version
>>> from scipy import __version__ as _scipy_version
>>> if parse_version(_scipy_version) >= parse_version('0.16.0'):
...     matrix = get_DCT(img.shape, img.shape)
...     dct_matrix = matrix.T.dot(vec)
...     vec_roundtrip = matrix.dot(dct_matrix)
...     np.allclose(vec, vec_roundtrip)
... else:
...     True
True
```

magni.imaging.dictionaries._matrices.**get_DFT**(*shape*, *overcomplete_shape=None*)

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

Parameters:
- **shape** (*list or tuple*) – The shape of the image for which the dictionary is the DFT dictionary.
- **overcomplete_shape** (*list or tuple, optional*) – The shape of the (overcomplete) frequency domain for the DFT dictionary. The entries must be greater than or equal to the corresponding entries in *shape*.

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:

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

Perform DFT in the ordinary way:

```
>>> dft_normal = magni.imaging.dictionaries._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.conj().T.dot(vec)
```

Check that the two ways produce the same result:

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

Compute the overcomplete transform (and back again):

```
>>> matrix = get_DFT(img.shape, img.shape)
>>> dft_matrix = matrix.conj().T.dot(vec)
>>> vec_roundtrip = matrix.dot(dft_matrix)
```

Check that the twice transformed image is identical to the original:

```
>>> np.allclose(vec, vec_roundtrip)
True
```

magni.imaging.dictionaries._visualisations module

Module providing functionality for visualising dictionary coefficients.

Routine listings

visualise_DCT(shape)

Function for visualising DCT coefficients.

visualise_DFT(shape)

Function for visualising DFT coefficients.

magni.imaging.dictionaries._visualisations.**visualise_DCT**(*shape*)

Return utilities for visualising DCT coefficients.

A handle to a function to transform the coefficients into a 'displayable' format is returned along with a tuple of ranges of the axes in the 2D coefficient plane.

Parameters: **shape** (*tuple*) – The shape of the 2D DCT being visualised.

Returns:
- **display_coefficients** (*Function*) – The function used to transform coefficients into a 'displayable' format.
- **axes_extent** (*tuple*) – The ranges of the axes in the 2D coefficient plane.

Notes

The display_coefficients function takes log10 to the absolute value of the transform cofficient vector given to it as an argument. The returned displayable coefficients is a matrix.

The axes_extent consists of (abcissa_min, abcissa_max, ordinate_min, ordinate_max).

magni.imaging.dictionaries._visualisations.**visualise_DFT**(*shape*)

Return utilities for visualising DFT coefficients.

A handle to a function to transform the coefficients into a 'displayable' format is returned along with a tuple of ranges of the axes in the 2D coefficient plane.

Parameters: **shape** (*tuple*) – The shape of the 2D DFT being visualised.

Returns:
- **display_coefficients** (*Function*) – The function used to transform coefficients into a 'displayable' format.
- **axes_extent** (*tuple*) – The ranges of the axes in the 2D coefficient plane.

Notes

The display_coefficients function takes log10 to the absolute value of the transform cofficient vector given to it as an argument. The returned displayable coefficients is a matrix that is flipped up/down and fftshifted.

The axes_extent consists of (abcissa_min, abcissa_max, ordinate_min, ordinate_max).

magni.imaging.dictionaries.analysis module

Module providing functionality for the analysis of dictionaries.

Routine listings

get_reconstructions(img, transform, fractions)
Function to get a set of transform reconstructions.

show_coefficient_histogram(img, transforms, bins=None, range=None,
output_path=None, fig_ext='pdf') Function to show a transform coefficient histogram.

show_psnr_energy_rolloff(img, reconstructions, fractions, return_vals=False,
output_path=None, fig_ext='pdf') Function to show PSNR and retained energy rolloff of reconstructions.

show_reconstructions(coefficients, reconstructions, transform, fractions,
output_path=None, fig_ext='pdf') Function to show tranforms reconstructions and coefficients.

show_sorted_coefficients(img, transforms, output_path=None, fig_ext='pdf')
Function to show a plot of transform coefficients in sorted order.

show_transform_coefficients(img, transforms, output_path=None, fig_ext='pdf')
Function to show transform coefficients.

show_transform_quantiles(img, transform, fraction=1.0, area_mask=None)
Function to show quantiles of transform coefficients.

magni.imaging.dictionaries.analysis.**get_reconstructions**(*img*, *transform*, *fractions*)

Return transform reconstructions with different fractions of coefficents.

The image *img* is transform coded using the specified *transform*. Reconstructions for the **fractions** of transform coefficients kept are returned along with the coefficients used in the reconstructions.

Parameters:
- **img** (*ndarray*) – The image to get reconstructions of.
- **transform** (*str*) – The transform to use to obtain the reconstructions.
- **fractions** (*list or tuple*) – The fractions of coefficents to be used in the reconstructions.

Returns:
- **coefficients** (*list*) – The list of coefficients (each an ndarray) used in the reconstructions.
- **reconstructions** (*list*) – The list of reconstructions.

Examples

Get reconstructions from DCT based on 20% and 40% of the coefficients:

```
>>> import numpy as np
>>> from magni.imaging.dictionaries.analysis import get_reconstructions
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> transform = 'DCT'
>>> fractions = (0.2, 0.4)
>>> coefs, recons = get_reconstructions(img, transform, fractions)
>>> len(recons)
2
>>> tuple(int(s) for s in coefs[0].shape)
(8, 8)
>>> tuple(int(s) for s in recons[0].shape)
(8, 8)
```

magni.imaging.dictionaries.analysis.**show_coefficient_histogram**(*img*, *transforms*, *bins=None*, *range=None*, *output_path=None*, *fig_ext='pdf'*)

Show a histogram of coefficient values for different transforms.

A histogram of the transform coefficient values for *img* using the different *transforms* are shown. If *output_path* is not None, the resulting figure and data used in the figure are saved.

Parameters:

- **img** (*ndarray*) – The image to get transform coefficients for.
- **transforms** (*list or tuple*) – The names as strings of the transforms to use.
- **bins** (*int*) – The number of bins to use in the histogram (the default is None, which implies that the number of bins is determined based on the size of *img*).
- **range** (*tuple*) – The lower and upper range of the bins to use in the histogram (the default is None, which implies that the min and max values of *img* are used).
- **output_path** (*str*) – The output path (see notes below) to save the figure and data to (the default is None, which implies that the figure and data are not saved).
- **fig_ext** (*str*) – The figure extension determining the format of the saved figure (the default is 'pdf' which implies that the figure is saved as a PDF).

See also:

matplotlib.pyplot.hist()

The underlying histogram plot function.

Notes

The *output_path* is specified as a path to a folder + an optional prefix to the file name. The remaining file name is fixed. If e.g, the fixed part of the file name was 'plot', then:

- output_path = '/home/user/' would save the figure under /home/user/ with the name plot.pdf.
- output_path = '/home/user/best' would save the figure under /home/user with the name best_plot.pdf.

In addition to the saved figures, an annotated and chased HDF database with the data used to create the figures are also saved. The name of the HDF database is the same as for the figure with the exception that the file extension is '.hdf5'.

Examples

Save a coefficient histogram using the DCT and the DFT as transforms

```
>>> import os, numpy as np
>>> from magni.imaging.dictionaries import analysis as _a
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> transforms = ('DCT', 'DFT')
>>> output_path = './histogram_test'
>>> _a.show_coefficient_histogram(img, transforms, output_path=output_path)
>>> current_dir = os.listdir('./')
>>> for file in sorted(current_dir):
...     if 'histogram_test' in file:
...         print(file)
histogram_test_coefficient_histogram.hdf5
histogram_test_coefficient_histogram.pdf
```

magni.imaging.dictionaries.analysis.**show_psnr_energy_rolloff**(*img*, *reconstructions*, *fractions*, *return_vals=False*, *output_path=None*, *fig_ext='pdf'*)

Show the PSNR and energy rolloff for the reconstructions.

A plot of the Peak Signal to Noise Ratio (PSNR) and retained energy in the *recontructions* versus the **fractions** of coefficients used in the reconstructions is shown. If return_vals is True, the data used in the plot is returned. If *output_path* is not None, the resulting figure and data used in the figure are saved.

Parameters:
- **img** (*ndarray*) – The image which the reconstructions are based on.
- **reconstructions** (*list or tuple*) – The reconstructions (each an ndarray) to show rolloff for.
- **fractions** (*list or tuple*) – The fractions of coefficents used in the reconstructions.
- **return_vals** (*bool*) – The flag indicating wheter or not to return the PSNR and energy values (the default is False, which indicate that the values are not returned).
- **output_path** (*str*) – The output path (see notes below) to save the figure and data to (the default is None, which implies that the figure and data are not saved).
- **fig_ext** (*str*) – The figure extension determining the format of the saved figure (the default is 'pdf' which implies that the figure is saved as a PDF).

Returns:
- **psnrs** (*ndarray*) – The PSNR values shown in the figure (only returned if return_vals=True).
- **energy** (*ndarray*) – The retained energy values shown in the figure (only returned if return_vals=True).

Notes

The *output_path* is specified as a path to a folder + an optional prefix to the file name. The remaining file name is fixed. If e.g, the fixed part of the file name was 'plot', then:

- output_path = '/home/user/' would save the figure under /home/user/ with the name plot.pdf.
- output_path = '/home/user/best' would save the figure under /home/user with the name best_plot.pdf.

In addition to the saved figures, an annotated and chased HDF database with the data used to create the figures are also saved. The name of the HDF database is the same as for the figure with the exception that the file extension is '.hdf5'.

Examples

Save a PSNR and energy rolloff plot for reconstructions bases on the DCT:

```
>>> import os, numpy as np
>>> from magni.imaging.dictionaries import analysis as _a
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> transform = 'DCT'
>>> fractions = (0.2, 0.4)
>>> coefs, recons = _a.get_reconstructions(img, transform, fractions)
>>> o_p = './rolloff_test'
>>> _a.show_psnr_energy_rolloff(img, recons, fractions, output_path=o_p)
>>> current_dir = os.listdir('./')
>>> for file in sorted(current_dir):
...     if 'rolloff_test' in file:
...         print(file)
rolloff_test_psnr_energy_rolloff.hdf5
rolloff_test_psnr_energy_rolloff.pdf
```

magni.imaging.dictionaries.analysis.**show_reconstructions**(*coefficients*, *reconstructions*, *transform*, *fractions*, *output_path=None*, *fig_ext='pdf'*)

Show reconstructions and corresponding coefficients.

Parameters:
- **coefficients** (*list or tuple*) – The coefficients (each an ndarray) used in the reconstructions.
- **reconstructions** (*list or tuple*) – The reconstructions (each an ndarray) to show.
- **transform** (*str*) – The transform used to obtain the reconstructions.
- **fractions** (*list or tuple*) – The fractions of coefficents used in the reconstructions.
- **output_path** (*str*) – The output path (see notes below) to save the figure and data to (the default is None, which implies that the figure and data are not saved).
- **fig_ext** (*str*) – The figure extension determining the format of the saved figure (the default is 'pdf' which implies that the figure is saved as a PDF).

Notes

The *output_path* is specified as a path to a folder + an optional prefix to the file name. The remaining file name is fixed. If e.g, the fixed part of the file name was 'plot', then:

- output_path = '/home/user/' would save the figure under /home/user/ with the name plot.pdf.
- output_path = '/home/user/best' would save the figure under /home/user with the name best_plot.pdf.

In addition to the saved figures, an annotated and chased HDF database with the data used to create the figures are also saved. The name of the HDF database is the same as for the figure with the exception that the file extension is '.hdf5'.

Examples

Save images of coefficients and reconstructions based on the DCT:

```
>>> import os, numpy as np
>>> from magni.imaging.dictionaries import analysis as _a
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> trans = 'DCT'
>>> fracs = (0.2, 0.4)
>>> coefs, recons = _a.get_reconstructions(img, trans, fracs)
>>> o_p = './reconstruction_test'
>>> _a.show_reconstructions(coefs, recons, trans, fracs, output_path=o_p)
>>> current_dir = os.listdir('./')
>>> for file in sorted(current_dir):
...     if 'reconstruction_test' in file:
...         print(file)
reconstruction_test_reconstruction_coefficients.hdf5
reconstruction_test_reconstruction_coefficients.pdf
reconstruction_test_reconstructions.hdf5
reconstruction_test_reconstructions.pdf
```

magni.imaging.dictionaries.analysis.**show_sorted_coefficients**(*img*, *transforms*, *output_path=None*, *fig_ext='pdf'*)

Show the transform coefficient values in sorted order.

A plot of the sorted coefficient values vs array index number is shown. If *output_path* is not None, the resulting figure and data used in the figure are saved.

Parameters:
- **img** (*ndarray*) – The image to show the sorted transform coefficients values for.
- **transforms** (*list or tuple*) – The names as strings of the transforms to use.
- **output_path** (*str*) – The output path (see notes below) to save the figure and data to (the default is None, which implies that the figure and data are not saved).
- **fig_ext** (*str*) – The figure extension determining the format of the saved figure (the default is 'pdf' which implies that the figure is saved as a PDF).

Notes

The *output_path* is specified as a path to a folder + an optional prefix to the file name. The remaining file name is fixed. If e.g, the fixed part of the file name was 'plot', then:

- output_path = '/home/user/' would save the figure under /home/user/ with the name plot.pdf.
- output_path = '/home/user/best' would save the figure under /home/user with the name best_plot.pdf.

In addition to the saved figures, an annotated and chased HDF database with the data used to create the figures are also saved. The name of the HDF database is the same as for the figure with the exception that the file extension is '.hdf5'.

Examples

Save a sorted transform coefficient plot for the DCT and DFT transforms:

```
>>> import os, numpy as np
>>> from magni.imaging.dictionaries import analysis as _a
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> transforms = ('DCT', 'DFT')
>>> output_path = './sorted_test'
>>> _a.show_sorted_coefficients(img, transforms, output_path=output_path)
>>> current_dir = os.listdir('./')
>>> for file in sorted(current_dir):
...     if 'sorted_test' in file:
...         print(file)
sorted_test_sorted_coefficients.hdf5
sorted_test_sorted_coefficients.pdf
```

magni.imaging.dictionaries.analysis.**show_transform_coefficients**(*img*, *transforms*, *output_path=None*, *fig_ext='pdf'*)

Show the transform coefficients.

The transform coefficient of *img* are shown for the *tranforms*. If *output_path* is not None, the resulting figure and data used in the figure are saved.

Parameters:
- **img** (*ndarray*) – The image to show the transform coefficients for.
- **transforms** (*list or tuple*) – The names as strings of the transforms to use.
- **output_path** (*str*) – The output path (see notes below) to save the figure and data to (the default is None, which implies that the figure and data are not saved).
- **fig_ext** (*str*) – The figure extension determining the format of the saved figure (the default is 'pdf' which implies that the figure is saved as a PDF).

Notes

The *output_path* is specified as a path to a folder + an optional prefix to the file name. The remaining file name is fixed. If e.g, the fixed part of the file name was 'plot', then:

- output_path = '/home/user/' would save the figure under /home/user/ with the name plot.pdf.
- output_path = '/home/user/best' would save the figure under /home/user with the name best_plot.pdf.

In addition to the saved figures, an annotated and chased HDF database with the data used to create the figures are also saved. The name of the HDF database is the same as for the figure with the exception that the file extension is '.hdf5'.

Examples

Save a figure showing the coefficients for the DCT and DFT transforms:

```
>>> import os, numpy as np
>>> from magni.imaging.dictionaries import analysis as _a
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> transforms = ('DCT', 'DFT')
>>> o_p = './coefficient_test'
>>> _a.show_transform_coefficients(img, transforms, output_path=o_p)
>>> current_dir = os.listdir('./')
>>> for file in sorted(current_dir):
...     if 'coefficient_test' in file:
...         print(file)
coefficient_test_transform_coefficients.hdf5
coefficient_test_transform_coefficients.pdf
```

magni.imaging.dictionaries.analysis.**show_transform_quantiles**(*img*, *transform*, *fraction=1.0*, *area_mask=None*, *ax=None*)

Show a plot of the quantiles of the transform coefficients.

The *fraction* of *transform* coefficients holding the most energy for the image *img* is considered. The four quantiles within this fraction of coefficients are illustrated in the *transform* domain by showing coefficients between the different quantiles in different colours. If an *area_mask* is specified, only this area in the plot is higlighted whereas the rest is darkened.

Parameters:
- **img** (*ndarray*) – The image to show the transform quantiles for.
- **transform** (*str*) – The transform to use.
- **fraction** (*float*) – The fraction of coefficients used in the quantiles calculation (the default value is 1.0, which implies that all coefficients are used).
- **area_mask** (*ndarray*) – Bool array of the same shape as *img* which indicates the area of the image to highlight (the default value is None, which implies that no particular part of the image is highlighted).
- **ax** (*matplotlib.axes.Axes*) – The axes on which the image is displayed (the default is None, which implies that a separate figure is created).

Returns: **coef_count** (*dict*) – Different counts of coeffcients within the *area_mask* (only returned if *area_mask* is not None).

Notes

The ticks on the colorbar shown below the figure are the percentiles of the entire set of coefficients corresponding to the quantiles with fraction of coefficients. For instance, if fraction is 0.10, then the percentiles are 92.5, 95.0, 97.5, 100.0, corresponding to the four quantiles within the 10 percent coefficients holding the most

energy.

The *coef_count* dictionary holds the following keys:

- C_total : Total number of considered coefficients.
- Q_potential : Number of potential coefficients within *mask_area*.
- P_fraction : The fraction of Q_potential to the pixel count in *img*.
- Q_total : Total number of (considered) coefficients within *mask_area*.
- Q_fraction : The fraction of Q_total to Q_potential
- QC_fraction : The fraction of Q_total to C_total.
- Q0-Q1 : Number of coefficients smaller than the first quantile.
- Q1-Q2 : Number of coefficients between the first and second quantile.
- Q2-Q3 : Number of coefficients between the second and third quantile.
- Q3-Q4 : Number of coefficients between the third and fourth quantile.

Each of the QX-QY holds a tuple containing two values:

1. The number of coefficients.
2. The fraction of the number of coefficients to Q_total.

Examples

For example, show quantiles for a fraction of 0.2 of the DCT coefficients:

```
>>> import numpy as np
>>> from magni.imaging.dictionaries import analysis as _a
>>> img = np.arange(64).astype(np.float).reshape(8, 8)
>>> transforms = 'DCT'
>>> fraction = 0.2
>>> _a.show_transform_quantiles(img, transform, fraction=fraction)
```

magni.imaging.dictionaries.analysis.**_save_output**(*output_path*, *name*, *fig*, *fig_ext*, *datasets*)

Save figure and data output.

Parameters:
- **output_path** (*str*) – The output_path to save to.
- **name** (*str*) – The 'fixed' part of the file name saved to.
- **fig** (*matplotlib.figure.Figure*) – The figure instance to save.
- **fig_ext** (*str*) – The file extension to use for the saved figure.
- **datasets** (*dict*) – The dict of dicts for datasets to save in a HDF database.

magni.imaging.dictionaries.utils module

Module providing support functionality for the dictionaries subpackage.

Routine listings

get_function_handle(type_, transform)

Function to get a function handle to a transform method.

get_transform_names()

Function to get a tuple of names of the available transforms.

magni.imaging.dictionaries.utils.**get_function_handle**(*type_*, *transform*)

Return a function handle to a given transform method.

Parameters:
- **type_** (*{'matrix', 'visualisation'}*) – Identifier of the type of method to return a handle to.
- **transform** (*str*) – Identifier of the transform method to return a handle to.

Returns: **f_handle** (*function*) – Handle to *transform*.

Examples

For example, return a handle to the matrix method for a DCT:

```
>>> from magni.imaging.dictionaries.utils import get_function_handle
>>> get_function_handle('matrix', 'DCT').__name__
'get_DCT'
```

or return a handle to the visualisation method for a DFT:

```
>>> get_function_handle('visualisation', 'DFT').__name__
'visualise_DFT'
```

magni.imaging.dictionaries.utils.**get_transform_names**()

Return a tuple of names of the available transforms.

Returns: **names** (*tuple*) – The tuple of names of available transforms.

Examples

For example, get transform names and extract 'DCT' and 'DFT'

```
>>> from magni.imaging.dictionaries.utils import get_transform_names
>>> names = get_transform_names()
>>> tuple(sorted(name for name in names if name in ('DCT', 'DFT')))
('DCT', 'DFT')
```

and a handle to corresponding visualisation method for the DCT

```
>>> from magni.imaging.dictionaries.utils import get_function_handle
>>> f_handles = tuple(get_function_handle('visualisation', name)
... for name in names)
>>> tuple(f_h.__name__ for f_h in f_handles if 'DCT' in f_h.__name__)
('visualise_DCT',)
```

magni.imaging.measurements package

Subpackage providing functionality for constructing and visualising scan patterns for measurements.

This subpackage 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 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.

construct_pixel_mask(h, w, pixels)
Construct a binary pixel mask.

lissajous_sample_image(h, w, scan_length, num_points, f_y=1., f_x=1.,
theta_y=0., theta_x=np.pi / 2) Function for lissajous sampling an image.

lissajous_sample_surface(l, w, speed, sample_rate, time, f_y=1., f_x=1.,
theta_y=0., theta_x=np.pi / 2, speed_mode=0) Function for lissajous sampling a surface.

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, rect_area=False)
Function for spiral sampling an image.

spiral_sample_surface(l, w, speed, sample_rate, time, rect_area=False)
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.

uniform_rotated_line_sample_image(h, w, scan_length, num_points, angle=0.,
follow_edge=True) Function for uniform rotated line sampling an image.

uniform_rotated_line_sample_surface(l, w, speed, sample_rate, time, angle=0.,
follow_edge=True) Function for uniform rotated line sampling a surface.

unique_pixels(coords)

Function for determining unique pixels from a set of coordinates.

zigzag_sample_image(h, w, scan_length, num_points, angle=np.pi / 20)

Function for zigzag sampling an image.

zigzag_sample_surface(l, w, speed, sample_rate, time, angle=np.pi / 20)

Function for zigzag sampling a surface.

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 lissajous pattern:

```
>>> from magni.imaging.measurements import lissajous_sample_surface
>>> l = 13.0; w = 13.0; speed = 4.0; time = 27.0; sample_rate = 3.0;
>>> coords = lissajous_sample_surface(l, w, speed, sample_rate, time,
...                                   speed_mode=1)
```

Display the resulting pattern:

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

Sample a 128x128 pixel image using a spiral pattern:

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

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)
```

Submodules

magni.imaging.measurements._lissajous module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

lissajous_sample_image(h, w, scan_length, num_points, f_y=1., f_x=1., theta_y=0., theta_x=np.pi / 2) Function for lissajous sampling an image.

lissajous_sample_surface(l, w, speed, sample_rate, time, f_y=1., f_x=1., theta_y=0., theta_x=np.pi / 2, speed_mode=0) Function for lissajous sampling a surface.

magni.imaging.measurements._lissajous.**lissajous_sample_image**(*h*, *w*, *scan_length*, *num_points*, *f_y=1.0*, *f_x=1.0*, *theta_y=0.0*, *theta_x=1.5707963267948966*)

Sample an image using a lissajous pattern.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using a lissajous 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.
- **f_y** (*float*) – The frequency of the y-sinusoid (the default value is 1.0).
- **f_x** (*float*) – The frequency of the x-sinusoid (the default value is 1.0).
- **theta_y** (*float*) – The starting phase of the y-sinusoid (the default is 0.0).
- **theta_x** (*float*) – The starting phase of the x-sinusoid (the default is pi / 2).

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,

```
>>> import numpy as np
>>> from magni.imaging.measurements import lissajous_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> np.set_printoptions(suppress=True)
>>> lissajous_sample_image(h, w, scan_length, num_points)
array([[ 5.        ,  9.5       ],
       [ 1.40370042,  7.70492686],
       [ 0.67656563,  3.75183526],
       [ 3.39871123,  0.79454232],
       [ 7.39838148,  1.19240676],
       [ 9.48459832,  4.62800824],
       [ 7.99295651,  8.36038857],
       [ 4.11350322,  9.41181634],
       [ 0.94130617,  6.94345168],
       [ 1.0071768 ,  2.92458128],
       [ 4.25856283,  0.56150128],
       [ 8.10147506,  1.7395012 ],
       [ 9.4699986 ,  5.51876059]])
```

magni.imaging.measurements._lissajous.**lissajous_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*, *f_y=1.0*, *f_x=1.0*, *theta_y=0.0*, *theta_x=1.5707963267948966*, *speed_mode=0*)

Sample a surface area using a lissajous pattern.

The coordinates (in units of meters) resulting from sampling an area of size *l* times *w* using a lissajous 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.
- **f_y** (*float*) – The frequency of the y-sinusoid (the default value is 1.0).
- **f_x** (*float*) – The frequency of the x-sinusoid (the default value is 1.0).
- **theta_y** (*float*) – The starting phase of the y-sinusoid (the default is 0.0).
- **theta_x** (*float*) – The starting phase of the x-sinusoid (the default is pi / 2).
- **speed_mode** (*int*) – The speed mode used to select sampling points (the default is 0 which implies that the speed argument determines the speed, and f_y and f_x determine the ratio between the relative frequencies used).

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.

Generally, the lissajous sampling pattern does not provide constant speed, and this cannot be compensated for without violating f_y, f_x, or both. Therefore, *speed_mode* allows the user to determine how this issue is handled: In *speed_mode* 0, constant speed equal to *speed* is ensured by non-uniform sampling of a lissajous curve, whereby *f_y* and *f_x* are not constant frequencies. In *speed_mode* 1, average speed equal to *speed* is ensured by scaling *f_y* and *f_x* by the same constant. In *speed_mode* 2, *f_y* and *f_x* are kept constant and the *speed* is only used to determine the path length in combination with **time**.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import lissajous_sample_surface
>>> l = 1e-6
>>> w = 1e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> np.set_printoptions(suppress=True)
>>> lissajous_sample_surface(l, w, speed, sample_rate, time)
array([[ 0.0000005 ,  0.000001  ],
       [ 0.00000001,  0.00000058],
       [ 0.00000033,  0.00000003],
       [ 0.00000094,  0.00000025],
       [ 0.00000082,  0.00000089],
       [ 0.00000017,  0.00000088],
       [ 0.00000007,  0.00000024],
       [ 0.00000068,  0.00000003],
       [ 0.00000099,  0.0000006 ],
       [ 0.00000048,  0.000001  ],
       [ 0.        ,  0.00000057],
       [ 0.00000035,  0.00000002],
       [ 0.00000094,  0.00000027]])
```

magni.imaging.measurements._matrices module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

construct_measurement_matrix(coords, h, w)

Function for constructing a measurement matrix.

magni.imaging.measurements._matrices.**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.
- **h** (*int*) – The height of the image measured in pixels.
- **w** (*int*) – The width of the image measured in pixels.

Returns: **Phi** (*magni.utils.matrices.Matrix*) – The constructed measurement matrix.

See also:

magni.utils.matrices.Matrix()

The matrix emulator class.

Notes

The function constructs 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:

```
>>> import numpy as np, magni
>>> 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._random_line module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

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.

magni.imaging.measurements._random_line.**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,

```
>>> import numpy as np
>>> from magni.imaging.measurements import random_line_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> seed = 6021
>>> np.set_printoptions(suppress=True)
>>> random_line_sample_image(h, w, scan_length, num_points, seed=seed)
array([[ 0.5       ,  0.5       ],
       [ 4.59090909,  0.5       ],
       [ 8.68181818,  0.5       ],
       [ 7.01473938,  1.28746666],
       [ 2.92383029,  1.28746666],
       [ 0.5       ,  2.95454545],
       [ 0.5       ,  7.04545455],
       [ 4.03665944,  7.59970419],
       [ 8.12756853,  7.59970419],
       [ 8.68181818,  9.5       ],
       [ 4.59090909,  9.5       ],
       [ 0.5       ,  9.5       ]])
```

magni.imaging.measurements._random_line.**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,

```
>>> import numpy as np
>>> 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
>>> np.set_printoptions(suppress=True)
>>> random_line_sample_surface(l, w, speed, sample_rate, time, seed=seed)
array([[ 0.        ,  0.        ],
       [ 0.00000067,  0.        ],
       [ 0.00000133,  0.        ],
       [ 0.000002  ,  0.        ],
       [ 0.000002  ,  0.00000067],
       [ 0.000002  ,  0.00000133],
       [ 0.00000158,  0.00000158],
       [ 0.00000091,  0.00000158],
       [ 0.00000024,  0.00000158],
       [ 0.        ,  0.000002  ],
       [ 0.00000067,  0.000002  ],
       [ 0.00000133,  0.000002  ],
       [ 0.000002  ,  0.000002  ]])
```

magni.imaging.measurements._spiral module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

spiral_sample_image(h, w, scan_length, num_points, rect_area=False)

Function for spiral sampling an image.

spiral_sample_surface(l, w, speed, sample_rate, time, rect_area=False)

Function for spiral sampling a surface.

magni.imaging.measurements._spiral.**spiral_sample_image**(*h*, *w*, *scan_length*, *num_points*, *rect_area=False*)

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.
- **rect_area** (*bool*) – A flag indicating whether or not the full rectangular area is sampled (the default value is False which implies that the “corners” of the rectangular area are not sampled).

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.

If the *rect_area* flag is True, then it is assumed that the sampling continues outside of the rectangular area specified by *h* and *w* such that the "corners" of the rectangular area are also sampled. The sample points outside of the rectangular area are discarded and, hence, not returned.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import spiral_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> np.set_printoptions(suppress=True)
>>> 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.**spiral_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*, *rect_area=False*)

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.
- **rect_area** (*bool*) – A flag indicating whether or not the full rectangular area is sampled (the default value is False which implies that the "corners" of the rectangular area are not sampled).

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.

If the *rect_area* flag is True, then it is assumed that the sampling continues outside of the rectangular area specified by *l* and *w* such that the "corners" of the rectangular area are also sampled. The sample points outside of the rectangular area are discarded and, hence, not returned.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import spiral_sample_surface
>>> l = 1e-6
>>> w = 1e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> np.set_printoptions(suppress=True)
>>> spiral_sample_surface(l, w, speed, sample_rate, time)
array([[ 0.00000036,  0.00000046],
       [ 0.00000052,  0.00000071],
       [ 0.00000052,  0.00000024],
       [ 0.00000059,  0.00000079],
       [ 0.00000021,  0.00000033],
       [ 0.00000084,  0.00000036],
       [ 0.00000049,  0.0000009 ],
       [ 0.0000001 ,  0.00000036],
       [ 0.00000072,  0.00000011],
       [ 0.00000089,  0.00000077],
       [ 0.00000021,  0.00000091]])
```

magni.imaging.measurements._square_spiral module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

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.

magni.imaging.measurements._square_spiral.**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,

```
>>> import numpy as np
>>> from magni.imaging.measurements import square_spiral_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> np.set_printoptions(suppress=True)
>>> 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.**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,

```
>>> import numpy as np
>>> 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
>>> np.set_printoptions(suppress=True)
>>> square_spiral_sample_surface(l, w, speed, sample_rate, time)
array([[ 0.0000005,  0.0000005],
       [ 0.0000004,  0.0000004],
       [ 0.0000006,  0.0000007],
       [ 0.0000005,  0.0000003],
       [ 0.0000002,  0.0000007],
       [ 0.0000008,  0.0000007],
       [ 0.0000006,  0.0000002],
       [ 0.0000001,  0.0000004],
       [ 0.0000003,  0.0000009],
       [ 0.0000009,  0.0000008],
       [ 0.0000009,  0.0000001],
       [ 0.0000002,  0.0000001]])
```

magni.imaging.measurements._uniform_line module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

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.

magni.imaging.measurements._uniform_line.**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,

```
>>> import numpy as np
>>> from magni.imaging.measurements import uniform_line_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> np.set_printoptions(suppress=True)
>>> uniform_line_sample_image(h, w, scan_length, num_points)
array([[ 0.5       ,  0.5       ],
       [ 4.59090909,  0.5       ],
       [ 8.68181818,  0.5       ],
       [ 9.22727273,  3.5       ],
       [ 5.13636364,  3.5       ],
       [ 1.04545455,  3.5       ],
       [ 1.04545455,  6.5       ],
       [ 5.13636364,  6.5       ],
       [ 9.22727273,  6.5       ],
       [ 8.68181818,  9.5       ],
       [ 4.59090909,  9.5       ],
       [ 0.5       ,  9.5       ]])
```

magni.imaging.measurements._uniform_line.**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,

```
>>> import numpy as np
>>> 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
>>> np.set_printoptions(suppress=True)
>>> uniform_line_sample_surface(l, w, speed, sample_rate, time)
array([[ 0.        ,  0.        ],
       [ 0.00000067,  0.        ],
       [ 0.00000133,  0.        ],
       [ 0.000002  ,  0.        ],
       [ 0.000002  ,  0.00000067],
       [ 0.00000167,  0.000001  ],
       [ 0.000001  ,  0.000001  ],
       [ 0.00000033,  0.000001  ],
       [ 0.        ,  0.00000133],
       [ 0.        ,  0.000002  ],
       [ 0.00000067,  0.000002  ],
       [ 0.00000133,  0.000002  ],
       [ 0.000002  ,  0.000002  ]])
```

magni.imaging.measurements._uniform_rotated_line module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

uniform_rotated_line_sample_image(h, w, scan_length, num_points, angle=0., follow_edge=True) Function for uniform rotated line sampling an image.

uniform_rotated_line_sample_surface(l, w, speed, sample_rate, time, angle=0., follow_edge=True) Function for uniform rotated line sampling a surface.

magni.imaging.measurements._uniform_rotated_line.**uniform_rotated_line_sample_image**(*h*, *w*, *scan_length*, *num_points*, *angle=0.0*, *follow_edge=True*)

Sample an image using a uniform rotated line pattern.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using a uniform rotated line 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.
- **angle** (*float*) – The angle measured in radians by which the uniform lines are rotated (the default is 0.0 resulting in a pattern identical to that of uniform_line_sample_image).
- **follow_edge** (*bool*) – A flag indicating whether or not the pattern follows the edges of the rectangular area in-between lines (the default is True).

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 *angle* is limited to the interval). An *angle* of 0 results in the same behaviour as that of uniform_line_sample_image. An increase in the *angle* rotates the overall direction counterclockwise, i.e., at $\frac{}{2}$, the uniform_line_sample_image sampling pattern is rotated 90 degrees counterclockwise.

If the *follow_edge* flag is True, then the pattern follows the edges of the rectangular area when moving from one line to the next. If the flag is False, then the pattern follows a line perpendicular to the uniform lines when moving from one line to the next. In the latter case, some of the uniform lines are shortened to allow the described behaviour.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import \
... uniform_rotated_line_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> np.set_printoptions(suppress=True)
>>> uniform_rotated_line_sample_image(h, w, scan_length, num_points)
array([[ 0.5       ,  0.5       ],
       [ 4.59090909,  0.5       ],
       [ 8.68181818,  0.5       ],
       [ 9.22727273,  3.5       ],
       [ 5.13636364,  3.5       ],
       [ 1.04545455,  3.5       ],
       [ 1.04545455,  6.5       ],
       [ 5.13636364,  6.5       ],
       [ 9.22727273,  6.5       ],
       [ 8.68181818,  9.5       ],
       [ 4.59090909,  9.5       ],
       [ 0.5       ,  9.5       ]])
```

magni.imaging.measurements._uniform_rotated_line.**uniform_rotated_line_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*, *angle=0.0*, *follow_edge=True*)

Sample a surface area using a uniform rotated line pattern.

The coordinates (in units of meters) resulting from sampling an area of size *l* times *w* using uniform rotated line 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.
- **angle** (*float*) – The angle measured in radians by which the uniform lines are rotated (the default is 0.0 resulting in a pattern identical to that of uniform_line_sample_image).
- **follow_edge** (*bool*) – A flag indicating whether or not the pattern foolows the edges of the rectangular area in-between lines (the default is True).

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 *angle* is limited to the interval). An *angle* of 0 results in the same behaviour

as that of uniform_line_sample_surface. An increase in the *angle* rotates the overall direction counterclockwise, i.e., at $\frac{\pi}{2}$, the uniform_line_sample_surface sampling pattern is rotated 90 degrees counterclockwise.

If the *follow_edge* flag is True, then the pattern follows the edges of the rectangular area when moving from one line to the next. If the flag is False, then the pattern follows a line perpendicular to the uniform lines when moving from one line to the next. In the latter case, some of the uniform lines are shortened to allow the described behaviour.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import \
... uniform_rotated_line_sample_surface
>>> l = 1e-6
>>> w = 1e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> np.set_printoptions(suppress=True)
>>> uniform_rotated_line_sample_surface(l, w, speed, sample_rate, time)
array([[ 0.        ,  0.        ],
       [ 0.00000067,  0.        ],
       [ 0.00000083,  0.00000017],
       [ 0.00000017,  0.00000017],
       [ 0.00000033,  0.00000033],
       [ 0.000001  ,  0.00000033],
       [ 0.0000005 ,  0.0000005 ],
       [ 0.        ,  0.00000067],
       [ 0.00000067,  0.00000067],
       [ 0.00000083,  0.00000083],
       [ 0.00000017,  0.00000083],
       [ 0.00000033,  0.000001  ],
       [ 0.000001  ,  0.000001  ]])
```

magni.imaging.measurements._util module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

construct_pixel_mask(h, w, pixels)
: Construct a binary pixel mask.

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

magni.imaging.measurements._util.**construct_pixel_mask**(*h*, *w*, *pixels*)

Construct a binary pixel mask.

An image (2D array) of shape *w* x *h* is created where all *pixels* are marked True.

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.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import construct_pixel_mask
>>> h = 3
>>> w = 3
>>> pixels = np.array([[0, 0], [1, 1], [2, 1]])
>>> construct_pixel_mask(h, w, pixels)
array([[ True, False, False],
       [False,  True,  True],
       [False, False, False]], dtype=bool)
```

magni.imaging.measurements._util.**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,

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

magni.imaging.measurements._visualisation module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

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.

magni.imaging.measurements._visualisation.**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,

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

magni.imaging.measurements._visualisation.**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,

```
>>> import numpy as np
>>> 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._zigzag module

Module providing public functions for the magni.imaging.measurements subpackage.

Routine listings

zigzag_sample_image(h, w, scan_length, num_points, angle=np.pi / 20)
Function for zigzag sampling an image.

zigzag_sample_surface(l, w, speed, sample_rate, time, angle=np.pi / 20)
Function for zigzag sampling a surface.

magni.imaging.measurements._zigzag.**zigzag_sample_image**(*h*, *w*, *scan_length*, *num_points*, *angle=0.15707963267948966*)

Sample an image using a zigzag pattern.

The coordinates (in units of pixels) resulting from sampling an image of size *h* times *w* using a zigzag 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.
- **angle** (*float*) – The angle measured in radians by which the lines deviate from being horizontal (the default is pi / 20).

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 *angle* is measured clockwise relative to horizontal and is limited to the interval $\left[0, \arctan\left(\frac{h}{w}\right)\right]$.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import zigzag_sample_image
>>> h = 10
>>> w = 10
>>> scan_length = 50.0
>>> num_points = 12
>>> np.set_printoptions(suppress=True)
>>> zigzag_sample_image(h, w, scan_length, num_points)
array([[ 0.5       ,  0.5       ],
       [ 4.98949246,  1.21106575],
       [ 9.47898491,  1.9221315 ],
       [ 5.03152263,  2.63319725],
       [ 0.54203017,  3.344263  ],
       [ 4.94746229,  4.05532875],
       [ 9.43695474,  4.7663945 ],
       [ 5.0735528 ,  5.47746025],
       [ 0.58406034,  6.188526  ],
       [ 4.90543212,  6.89959175],
       [ 9.39492457,  7.6106575 ],
       [ 5.11558297,  8.32172325]])
```

magni.imaging.measurements._zigzag.**zigzag_sample_surface**(*l*, *w*, *speed*, *sample_rate*, *time*, *angle=0.15707963267948966*)

Sample a surface area using a zigzag pattern.

The coordinates (in units of meters) resulting from sampling an area of size *l* times *w* using a zigzag 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.
- **angle** (*float*) – The angle measured in radians by which the lines deviate from being horizontal (the default is pi / 20).

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 *angle* is measured clockwise relative to horizontal and is limited to the interval $\mathrm{ct}\ \mathrm{n}\ \frac{h}{w}$.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.measurements import zigzag_sample_surface
>>> l = 1e-6
>>> w = 1e-6
>>> speed = 7e-7
>>> sample_rate = 1.0
>>> time = 12.0
>>> np.set_printoptions(suppress=True)
>>> zigzag_sample_surface(l, w, speed, sample_rate, time)
array([[ 0.        ,  0.        ],
       [ 0.00000069,  0.00000011],
       [ 0.00000062,  0.00000022],
       [ 0.00000007,  0.00000033],
       [ 0.00000077,  0.00000044],
       [ 0.00000054,  0.00000055],
       [ 0.00000015,  0.00000066],
       [ 0.00000084,  0.00000077],
       [ 0.00000047,  0.00000088],
       [ 0.00000022,  0.00000099],
       [ 0.00000091,  0.0000009 ],
       [ 0.00000039,  0.0000008 ],
       [ 0.0000003 ,  0.00000069]])
```

Submodules

magni.imaging._util module

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

magni.imaging._util.**double_mirror**(*img*, *fftstyle=False*)

Mirror image in both the vertical and horisontal axes.

The image is mirrored around its upper left corner first in the horizontal axis and then in the vertical axis such that an image of four times the size of the original is returned. If *fftstyle* is True, the image is constructed such it would represent a fftshifted version of the mirrored *img* such that entry (0, 0) is the DC component.

Parameters:
- **img** (*ndarray*) – The image to mirror.
- **fftstyle** (*bool*) – The flag that indicates if the fftstyle mirrored image is returned.

Returns: **mirrored_img** (*ndarray*) – The mirrored image.

Examples

For example, mirror a very simple 2-by-3 pixel image.

```
>>> import numpy as np
>>> from magni.imaging._util import double_mirror
>>> img = np.arange(6).reshape(2, 3)
>>> img
array([[0, 1, 2],
       [3, 4, 5]])
>>> double_mirror(img)
array([[5, 4, 3, 3, 4, 5],
       [2, 1, 0, 0, 1, 2],
       [2, 1, 0, 0, 1, 2],
       [5, 4, 3, 3, 4, 5]])
>>> double_mirror(img, fftstyle=True)
array([[0, 0, 0, 0, 0, 0],
       [0, 5, 4, 3, 4, 5],
       [0, 2, 1, 0, 1, 2],
       [0, 5, 4, 3, 4, 5]])
```

magni.imaging._util.**get_inscribed_masks**(*img*, *as_vec=False*)

Return a set of inscribed masks covering the image.

Two masks are returned. One is the disc with radius equal to that of the inscribed circle for *img*. The other is the inscribed square of the first mask. If *as_vec* is True, the *img* must be a vector representation of the (matrix) image. In this case, the masks are also returned in vector representation.

Parameters:
- **img** (*ndarray*) – The square image of even height/width which the masks should cover.
- **as_vec** (*bool*) – The indicator of whether or not to treat *img* as a vector instead of an image (the default is False, which implies that *img* is treated as a matrix.

Returns:
- **cicle_mask** (*ndarray*) – The inscribed cicle mask.
- **square_mask** (*ndarray*) – The inscribed square mask.

Examples

For example, get the inscribed masks of an 8-by-8 image:

```
>>> import numpy as np
>>> from magni.imaging._util import get_inscribed_masks, mat2vec
>>> img = np.arange(64).reshape(8, 8)
>>> circle_mask, square_mask = get_inscribed_masks(img)
>>> np_printoptions = np.get_printoptions()
>>> np.set_printoptions(formatter={'bool': lambda x: str(int(x))})
>>> circle_mask
array([[0, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1],
       [0, 1, 1, 1, 1, 1, 1, 0]], dtype=bool)
```

```
>>> square_mask
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]], dtype=bool)
```

Or get the same masks based on a vector:

```
>>> img_vec = mat2vec(img)
>>> c_vec_mask, s_vec_mask = get_inscribed_masks(img_vec, as_vec=True)
>>> c_vec_mask
array([[0],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [0],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
```

```
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [0],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [0]], dtype=bool)
>>> np.set_printoptions(**np_printoptions)
```

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,

```
>>> import numpy as np
>>> 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,

```
>>> import numpy as np
>>> 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.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:

```
>>> import numpy as np, magni
>>> 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, setting the measurements and getting the coefficients:

```
>>> domains.measurements = np.ones(2).reshape(-1, 1)
>>> np.set_printoptions(suppress=True)
>>> print(domains.coefficients)
[[ 0.70710678]
 [-0.70710678]
 [ 1.        ]]
```

Or setting the coefficients and getting the image:

```
>>> domains.coefficients = np.ones(3).reshape(-1, 1)
>>> print(domains.image)
[[ 0.        ]
 [ 1.41421356]
 [ 1.        ]]
```

Or setting the image and getting the measurements:

```
>>> domains.image = np.ones(3).reshape(-1, 1)
>>> print(domains.measurements)
[[ 1.]
 [ 1.]]
```

coefficients

Get the image in the coefficient domain.

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

image

Get the image in the image domain.

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

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:

$$\frac{1}{} \cdot \sum (x_o \quad - x \quad _{cons})^2$$

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

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.evaluation import calculate_mse
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_recons = np.ones((2,2))
>>> print('{:.2f}'.format(calculate_mse(x_org, x_recons)))
1.50
```

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

$$1 \quad \cdot \mathrm{o}_{\ 1} \quad \frac{k^2}{1 \quad \cdot \quad (x_o \quad - x \quad _{cons})^2}$$

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

If $|x_o \quad - x \quad _{cons}| \quad = (1 \qquad 1 \qquad |x \quad _{cons}|)$ then **np.inf** is returned.

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.evaluation import calculate_psnr
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_recons = np.ones((2,2))
>>> peak = 3
>>> print('{:.2f}'.format(calculate_psnr(x_org, x_recons, peak)))
7.78
```

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 (must not be all zeros).
- **x_recons** (*ndarray*) – Array of reconstruction values.

Returns: **energy** (*float*) – Percentage of retained energy in reconstruction.

Notes

The retained energy is as calculated as

$$\frac{x^2{}_{cons}}{x_o^2} \cdot 1$$

Examples

For example,

```
>>> import numpy as np
>>> from magni.imaging.evaluation import calculate_retained_energy
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_recons = np.ones((2,2))
>>> print('{:.2f}'.format(calculate_retained_energy(x_org, x_recons)))
28.57
```

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

```
>>> import numpy as np
>>> from magni.imaging.preprocessing import detilt
>>> img = np.array([[0, 2, 3], [1, 5, 7], [3, 6, 8]], dtype=np.float)
>>> np.set_printoptions(suppress=True)
>>> 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, 0, 0], [1, 0, 1], [0, 1, 1]], dtype=np.bool)
>>> im, ti = detilt(img, mask=mask, mode='plane_flatten', return_tilt=True)
>>> np.set_printoptions(suppress=True)
>>> im
array([[ 0.11111111, -0.66666667, -2.44444444],
       [-0.33333333,  0.88888889,  0.11111111],
       [ 0.22222222,  0.44444444, -0.33333333]])
>>> ti
array([[-0.11111111,  2.66666667,  5.44444444],
       [ 1.33333333,  4.11111111,  6.88888889],
       [ 2.77777778,  5.55555556,  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. It provides a wrapper of the **matplotlib.pyplot.imshow** function that may exploit the provided functions for adjusting the image intensity. Also it include a function may be used to display a set of related images using a common colormapping.

Routine listings

imshow(X, ax=None, intensity_func=None, intensity_args=(), **kwargs)

Function that may be used to display an image.

imsubplot(imgs, rows, titles=None, x_labels=None, y_labels=None,

x_ticklabels=None, y_ticklabels=None, cbar_label=None, normalise=True, fixed_clim=None, **kwargs) Function that may be used to display a set of related images.

mask_img_from_coords(img, coords)

Function for masking certain parts of an image based on coordinates.

shift_mean(x_mod, x_org)

Function for shifting mean intensity of an image based on another image.

stretch_image(img, max_val, min_val=0)

Function for stretching the intensity of an image.

magni.imaging.visualisation.**imshow**(*X*, *ax=None*, *intensity_func=None*, *intensity_args=()*, *show_axis='frame'*, ***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', 'frame'}*) – 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**. If 'frame' only the frame is shown and not the ticks.

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

See also:

matplotlib.pyplot.imshow()

Matplotlib's imshow function.

Examples

For example,

```
>>> import numpy as np
>>> 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.**imsubplot**(*imgs*, *rows*, *titles=None*, *x_labels=None*, *y_labels=None*, *x_ticklabels=None*, *y_ticklabels=None*, *cbar_label=None*, *normalise=True*, *fixed_clim=None*, ***kwargs*)

Display a set of related images as subplots in a figure.

The images *imgs* are shown in a figure with a subplot layout based on the number of *rows*. The *titles*, *x_labels*, *y_labels*, *x_ticklabels*, and *y_ticklabels* are shown in the subplots. If *normalise* is True, all the images will share the same normalised colorbar/colormapping i.e. a particular colour will correspond to the same value across all images.

Parameters:
- **imgs** (*list or tuple*) – The images (as ndarrays) that is to be displayed.
- **rows** (*int*) – The number of rows to use in the subplot layout.
- **titles** (*list or tuple*) – The titles (as strings) to use for each of the subplots (the default is None, which implies that no titles are displayed).
- **x_labels** (*list or tuple*) – The x_labels (as strings) to use for each of the subplots (the default is None, which implies that no x_labels are displayed).
- **y_labels** (*list or tuple*) – The y_labels (as strings) to use for each of the subplots (the default is None, which implies that no y_labels are displayed).
- **x_ticklabels** (*list or tuple*) – The x_ticklabels (as strings or lists of strings) to use for the subplots (the default is None, which implies that no x_ticklabels are displayed).
- **y_ticklabels** (*list or tuple*) – The y_ticklabels (as strings or lists of strings) to use for the subplots (the default is None, which implies that no y_ticklabels are displayed).
- **cbar_label** (*str*) – The colorbar label to use with a normalised colormapping (the default is None, which implies that no colorbar label is displayed).
- **fixed_clim** (*list or tuple*) – The colorbar limits as a (min, max) sequence (the default is None, which implies that the colorbar limits are inferred from the data).
- **normalise** (*bool*) – The flag that indicates whether to use a normalised colormapping.

Returns: **fig** (*matplotlib.figure.Figure*) – The resulting figure instance.

See also:

matplotlib.pyplot.subplots()

Underlying subplot function.

Notes

The *x_ticklabels* and *y_ticklabels* may be either a collection of strings or a collections of collections of strings depending on wherther the labels should be shared across all subplots or different labels are to be used for each subplot.

Additional kwargs given to the function will be passed to the underlying suplot instantiation function **matplotlib.pyplot.subplots**.

If *normalise* is True, the common colorbar is shown below the subplots.

The implementation of the normalisation feature is based on the Pylab example: http://matplotlib.org/examples/pylab_examples/multi_image.html.

Examples

For example, show two images next to each other with a common colormapping:

```
>>> import numpy as np
>>> from magni.imaging.visualisation import imsubplot
>>> img1 = np.arange(4).reshape(2, 2)
>>> img2 = np.ones((4, 4))
>>> fig = imsubplot([img1, img2], 1, titles=['arange', 'ones'],
... x_labels=['x1', 'x2'], y_labels=['y1', 'y2'], cbar_label='Example',
... normalise=True)
```

or show the same images with shared ticklabels and fixed colorbar limits:

```
>>> common_x_ticklabels = ['a', 'b']
>>> common_y_ticklabels = ['c', 'd']
>>> fig = imsubplot([img1, img2], 1, x_ticklabels=common_x_ticklabels,
... y_ticklabels=common_y_ticklabels, fixed_clim=(0, 2), normalise=True)
```

or show the same images with different colormappings and ticklabels:

```
>>> x_ticklabels = [['a', 'b'], ['aa', 'bb']]
>>> y_ticklabels = [['c', 'd'], ['cc', 'dd']]
>>> fig = imsubplot([img1, img2], 1, x_ticklabels=x_ticklabels,
... y_ticklabels=y_ticklabels, normalise=False)
```

magni.imaging.visualisation.**mask_img_from_coords**(*img*, *coords*)

Mask coordinates in an image.

The coordinates *coords* in the image *img* are masked such that only the cooordinates are shown.

Parameters:
- **img** (*ndarray*) – The image to mask.
- **coords** (*ndarray*) – The coordinates arranged into a 2D array, such that each row is a coordinate pair (x, y).

Returns: **masked_img** (*numpy.ma.MaskedArray*) – The masked image.

See also:

magni.imaging.measurements()

Further description of the coordinate format.

Examples

For example, display only center pixel in a 3-by-3 image

```
>>> import numpy as np
>>> from magni.imaging.visualisation import mask_img_from_coords
>>> img = np.arange(9).reshape(3, 3)
>>> coords = np.array([[1, 1]])
>>> mask_img_from_coords(img, coords)
masked_array(data =
 [[-- -- --]
 [-- 4 --]
 [-- -- --]],
             mask =
 [[ True  True  True]
 [ True False  True]
 [ True  True  True]],
       fill_value = 999999)
```

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,

```
>>> import numpy as np
>>> from magni.imaging.visualisation import shift_mean
>>> x_org = np.arange(4).reshape(2, 2)
>>> x_mod = np.ones((2, 2))
>>> print('{:.1f}'.format(x_org.mean()))
1.5
>>> print('{:.1f}'.format(x_mod.mean()))
1.0
>>> shifted_x_mod = shift_mean(x_mod, x_org)
>>> print('{:.1f}'.format(shifted_x_mod.mean()))
1.5
>>> np.set_printoptions(suppress=True)
>>> shifted_x_mod
array([[ 1.5,  1.5],
       [ 1.5,  1.5]])
```

magni.imaging.visualisation.**stretch_image**(*img*, *max_val*, *min_val=0*)

Stretch image such that pixels values are in [*min_val*, *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.
- **min_val** (*int or float*) – The minimum 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 [*min_val*, *max_val*] using a linear stretch.

Examples

For example, stretch an image between 0 and 1

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

or stretch the image between -1 and 1

```
>>> stretched_img = stretch_image(img, 1.0, min_val=-1.0)
>>> stretched_img
array([[-1.        , -0.33333333],
       [ 0.33333333,  1.        ]])
```

or re-stretch the strecthed image between -3.0 and -1.5

```
>>> stretched_img = stretch_image(stretched_img, -1.5, min_val=-3.0)
>>> stretched_img
array([[-3. , -2.5],
       [-2. , -1.5]])
```

or re-stretch that image between 1.25 and 8.00

```
>>> stretched_img = stretch_image(stretched_img, 8.00, min_val=1.25)
>>> stretched_img
array([[ 1.25,  3.5 ],
       [ 5.75,  8.  ]])
```

magni.imaging.visualisation.**_handle_ticklabels**(*ax*, *k*, *x_ticklabels*, *y_ticklabels*)

Handle and format ticks and ticklabels for use in imsubplot.

The **imsubplot** function creates a figure showing an abitrary number of subplots. The **imsubplot** function allows for custom ticklabels along the x- and y-axes. This function handles the formatting of the ticklabels for a given subplot.

Parameters:
- **ax** (*matplotlib.axes.Axes*) – The matplotlib axes (subplot) to format ticklabels for.
- **k** (*int*) – The axes index, i.e. the subplot number out of the total number of subplots.
- **x_ticklabels** (*list or tuple*) – The x_ticklabels (as strings or lists of strings) to use for the subplots (the default is None, which implies that no x_ticklabels are displayed).
- **y_ticklabels** (*list or tuple*) – The y_ticklabels (as strings or lists of strings) to use for the subplots (the default is None, which implies that no y_ticklabels are displayed).

class magni.imaging.visualisation.**_ImageColourTracker**(*tracker*)

Track a common 'clim' in a set of matplotlib image subplots.

Parameters: **tracker** (*matplotlib.image.AxesImage*) – The image instance that must track a given 'clim'.

__call__(*tracked*)

magni.reproducibility package

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

Routine listings

data

Module providing functions that return various data about the system, magni, files, etc.

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 an Anaconda install.

get_datetime()

Function that returns information about the current date and time.

get_git_revision(git_root_dir=None)

Function that returns information about the current git revision.

get_file_hashes(path, blocksize=2**30)
: Function that returns the md5 and sha256 checksums of a file.

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 returned 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 Anaconda scientific Python distribution. This function will return various information about the Anaconda installation on the system by querying the Conda package database.

Warning: THIS IS HIGHLY EXPERIMENTAL AND MAY BREAK WITHOUT FURTHER NOTICE.

Note: Only infomation about the conda root environment is captured.

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

- env_export

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)

If “conda-env” is installed on the system, the *env_export* essentially holds the infomation from “conda env export -n root” as a dictionary. The information provided by this key partially overlaps with the infomation in the *linked_modules* and *modules_info* keys.

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**(*git_root_dir=None*)

Return a dictionary containing information about the current git revision.

Parameters: **git_root_dir** (*str*) – The path to the git root directory to get git revision for (the default is None, which implies that the git revision of the **magni** directory is returned).

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”)
- remote (output of “git remote -v”)

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" commands are run in the git root directory.

magni.reproducibility._annotation.**get_file_hashes**(*path*, *blocksize=1073741824*)

Return a dictionary with md5 and sha256 checksums of a file.

Parameters:
- **path** (*str*) – The path to the file to checksum.
- **blocksize** (*int*) – The chunksize (in bytes) to read from the file one at a time.

Returns: **file_hashes** (*dict*) – The dictionary holding the md5 and sha256 hexdigests of the file.

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._chase module

Module providing functions that may be used to chase data.

Routine listings

get_main_file_name()
: Function that returns the name of the main file/script.

get_main_file_source()
: Function that returns the source code of the main file/script.

get_main_source()
: Function that returns the ‘local’ source code of the main file/script.

get_stack_trace()
: Function that returns the complete stack trace.

magni.reproducibility._chase.**get_main_file_name**()

Return the name of the main file/script which called this function.

This will (if possible) return the name of the main file/script invoked by the Python interpreter i.e., the returned name is the name of the file, in which the bottom call on the stack is defined. Thus, the main file is still returned if the call to this function is buried deep in the code.

Returns: **source_file** (*str*) – The name of the main file/script which called this function.

Notes

It may not be possible to associate the call to this function with a main file. This is for instance the case if the call is handled by an IPython kernel. If no file is found, the returned string will indicate so.

magni.reproducibility._chase.**get_main_file_source**()

Return the source code of the main file/script which called this function.

This will (if possible) return the source code of the main file/script invoked by the Python interpreter i.e., the returned source code is the source code of the file, in which the bottom call on the stack is defined. Thus, the source code is still returned if the call to this function is buried deep in the code.

Returns: **source** (*str*) – The source code of the main file/script which called this function.

Notes

It may not be possible to associate the call to this function with a main file. This is for instance the case if the call is handled by an IPython kernel. If no file is found, the returned string will indicate so.

magni.reproducibility._chase.**get_main_source**()

Return the local source code of the main file which called this function.

This will (if possible) return the part of the source code of the main file/script surrounding (local to) the call to this function. The returned source code is a part of the source code of the file, in which the bottom call on the stack is defined. Thus, the source code is still returned if the call to this function is buried deep in the code.

Returns: **source** (*str*) – The local source code of the main file/script.

Notes

It may not be possible to associate the call to this function with a main file. This is for instance the case if the call is handled by an IPython kernel. If no file is found, the returned string will indicate so.

magni.reproducibility._chase.**get_stack_trace**()

Return the complete stack trace that led to the call to this function.

A (pretty) print version of the stack trace is returned.

Returns: **printed_stack** (*str*) – The stack trace that led to the call to this function.

magni.reproducibility.data module

Module providing functions that return data about the system, magni, files, etc

Functions used to get various annotations and chases data are available from this module.

See also:

magni.reproducibility._annotation
Documentation of annotations

magni.reproducibility._chase
Documentation of chases

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.

chase_database(h5file)
Function for chasing an existing HDF5 database.

create_database(h5file)
Function for creating a new annotated and chased HDF5 database.

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

read_chases(h5file)
Function for reading chases in an HDF5 database.

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

remove_chases(h5file)
Function for removing chases in an HDF5 database.

write_custom_annotation(h5file, annotation_name, annotation_value,
annotations_sub_group=None) Write a custom annotation to 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._chase.get_main_file_name
Magni main file name chase

magni.reproducibility._chase.get_main_file_source
Magni source code chase

magni.reproducibility._chase.get_main_source
Magni main source code chase

magni.reproducibility._chase.get_stack_trace
Magni stack trace chase

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':

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

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

Chase an HDF5 database to track information about stack and source code.

The chase consist of a group in the root of the *h5file* having nodes that each profide information about the program execution that led to this chase of the database.

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

See also:

magni.reproducibility._chase.get_main_file_name()

Name of main file

magni.reproducibility._chase.get_main_file_source()

Main file source code

magni.reproducibility._chase.get_main_source()

Source code around main

magni.reproducibility._chase.get_stack_trace()

Complete stack trace

Notes

The chase include the following information:

- main_file_name - Name of the main file/script that called this function

- main_file_source - Full source code of the main file/script
- main_source - Extract of main file source code that called this function
- stack_trace - Complete stack trace up until the call to this function

Examples

Chase the database named 'db.hdf5':

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

magni.reproducibility.io.**create_database**(*path*, *overwrite=True*)

Create a new HDF database that is annotated and chased.

A new HDF database is created and it is annotated using **magni.reproducibility.io.annotate_database** and chased using **magni.reproducibility.io.annotate_database**. If the *overwrite* flag is true and existing database at *path* is overwritten.

Parameters:
- **path** (*str*) – The path to the HDF file that is to be created.
- **overwrite** (*bool*) – The flag that indicates if an existing database should be overwritten.

See also:

magni.reproducibility.io.annotate_database()
Database annotation

magni.reproducibility.io.chase_database()
Database chase

Examples

Create a new database named 'new_db.hdf5':

```
>>> from magni.reproducibility.io import create_database
>>> create_database('new_db.hdf5')
```

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 are 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’:

```
>>> import magni
>>> 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.**read_chases**(*h5file*)

Read the chases to an HDF5 database.

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

Notes

The returned dict holds a key for each chase in the database. The value corresponding to a given key is a string. See *magni.reproducibility.chase_database* for examples of such chases.

Examples

Read chases from the database named ‘db.hdf5’:

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

Examples

Remove annotations from the database named ‘db.hdf5’:

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

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

Remove the chases from an HDF5 database.

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

Examples

Remove chases from the database named ‘db.hdf5’:

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

magni.reproducibility.io.**write_custom_annotation**(*h5file*, *annotation_name*, *annotation_value*, *annotations_sub_group=None*)

Write a custom annotation to an HDF5 database.

The annotation is written to the *h5file* under the *annotation_name* such that it holds the *annotation_value*.

Parameters:

- **h5file** (*tables.file.File*) – The handle to the HDF5 database to which the annotation is written.
- **annotation_name** (*str*) – The name of the annotation to write.
- **annotation_value** (*a JSON serialisable object*) – The annotation value to write.
- **annotations_sub_group** (*str*) – The group node under “/annotations” to which the custom annotation is written (the default is None which implies that the custom annotation is written directly under “/annotations”).

Notes

The *annotation_value* must be a JSON seriablisable object.

Examples

Write a custom annotation to an HDF5 database.

```
>>> import magni
>>> from magni.reproducibility.io import write_custom_annotation
>>> annotation_name = 'custom_annotation'
>>> annotation_value = 'the value'
>>> with magni.utils.multiprocessing.File('db.hdf5', mode='a') as h5file:
...     write_custom_annotation(h5file, annotation_name, annotation_value)
...     annotations = magni.reproducibility.io.read_annotations(h5file)
>>> str(annotations['custom_annotation'])
'the value'
```

magni.reproducibility.io.**_recursive_annotation_read**(*h5_annotations*, *out_annotations_dict*)

Recursively read annotations from an annotation group

Parameters:
- **h5_annotations** (*tables.group.Group*) – The group to read annotations from.
- **out_annotations_dict** (*dict*) – The dictionary to store the read annotations in.

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
: Subpackage providing validation capability.

types
: Module providing custom data types.

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
: Configger 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._config module

Module providing configuration options for the multiprocessing subpackage.

See also:

magni.utils.config.Configger
: The Configger class used

Notes

This module instantiates the *Configger* class provided by **magni.utils.config**. The configuration options are the following:

max_broken_pool_restarts : *int or None*
: The maximum number of attempts at restarting the process pool upon a BrokenPoolError (the default is 0, which implies that the process pool may not be restarted). If set to None, the process pool may restart indefinitely.

prefer_futures : *bool*
: The indicator of whether or not to prefer the concurrent.futures module over the multiprocessing module (the default is False, which implies that the multiprocessing module is used).

re_raise_exceptions : *bool*
: A flag indicating if exceptions should be re-raised (the default is False, which implies that the exception are not re-raised). It is useful to set this to True if the processing pool supports proper exception handling as e.g. when using “futures”.

silence_exceptions : *bool*

A flag indicating if exceptions should be silenced (the default is False, which implies that exceptions are raised).

workers : *int*

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

See the notes for the **magni.utils.multiprocessing._processing.process** function for more details about the *prefer_futures* and *max_broken_pool_restarts* configuration parameters.

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 processes.

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)*. Each task is executed in a seperate process using the concept of a processing pool.

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.

Raises: **BrokenPoolError** – If using the *concurrent.futures* module and one or more workers terminate abrubtly with the automatic broken pool restart funtionality disabled.

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 processing 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.
- The option of automatically restarting a broken process pool.

As of Python 3.2, two different, though quite similar, modules exist in the standard library for managing processing pools: **multiprocessing** and *concurrent.futures*. According to Python core contributor Jesse Noller, the plan is to eventually make concurrent.futures the only interface to the high level processing pools (futures), whereas multiprocessing is supposed to serve more low level needs for individually handling processes, locks, queues, etc. (see https://bugs.python.org/issue9205#msg132661). As of Python 3.5, both the **multiprocessing.Pool** and *concurrent.futures.ProcessPoolExecutor* serve almost the same purpose and provide very similar interfaces. The main differences between the two are:

- The option of using a worker initialiser is only available in **multiprocessing**.
- The option of specifing a maximum number of tasks for a worker to execute before being replaced to free up ressources (the maxtasksperchild option) is only available in **multiprocessing**.
- The option of specifying a context is only available in **multiprocessing**
- “Reasonable” handling of abrubt worker termination and exceptions is only available in *concurrent.futures*.

Particularly, the “reasonable” handling of a broken process pool may be a strong argument to prefer *concurrent.futures* over **multiprocessing**. The matter of handling a broken process pool has been extensively discussed in https://bugs.python.org/issue9205 which led to the fix for *concurrent.futures*. A similar fix for **multiprocessing** has been proposed in https://bugs.python.org/issue22393.

Both the **multiprocessing** and *concurrent.futures* interfaces are available for use with this function. If the configuration parameter *prefer_futures* is set to True and the *concurrent.futures* module is available, this is used. Otherwise, the **multiprocessing** module is used. A Python 2 backport of *concurrent.futures* is available at https://pythonhosted.org/futures/.

When using *concurrent.futures*, the *maxtasks*, *namespace*, and *init_args* are ignored since these are not supported by that module. It seems that *concurrent.futures* works as if maxtasks==1, however this is based purely on emprical observations. The *init_args* functionality may be added later on if an initialiser is added to *concurrent.futures* - see http://bugs.python.org/issue21423. If the *max_broken_pool_restarts* configuration parameter is set to a value different from 0, the Pool is automatically restarted and the tasks are re-run should a broken pool be encountered. If *max_broken_pool_restarts* is set to 0, a BrokenPoolError is raised should a broken pool be encountered.

When using **multiprocessing**, the *max_broken_pool_restarts* is ignored since the BrokenPoolError handling has not yet been implemented for the **multiprocessing.Pool** - see https://bugs.python.org/issue22393 as well as https://bugs.python.org/issue9205.

Examples

An example of how to use args_list, and kwargs_list:

```
>>> import magni
>>> 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]
```

or the same example preferring *concurrent.futures* over **multiprocessing**:

```
>>> magni.utils.multiprocessing.config['prefer_futures'] = True
>>> process(calculate, args_list=args_list, kwargs_list=kwargs_list)
[12, 6]
```

magni.utils.multiprocessing._processing.**_get_tasks**(*func*, *args_list*, *kwargs_list*)

Prepare a list of tasks.

Parameters:
- **func** (*function*) – A function handle to the function which the calls should be mapped to.
- **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).

Returns: **tasks** (*list*) – The list of tasks.

magni.utils.multiprocessing._processing.**_map_using_futures**(*func*, *tasks*, *init_args*, *maxtasks*, *max_broken_pool_restarts*)

Map a set of *tasks* to *func* and run them in parallel using a futures.

I f *max_broken_pool_restarts* is different from 0, the tasks must be an explicit collection, e.g. a list or tuple, for the restart to work. If an exception occurs in one of the function calls, the process pool terminates ASAP and re-raises the first exception that occurred. All exceptions that may have occurred in the workers are available as the last element in the exceptions args.

Parameters:
- **func** (*function*) – A function handle to the function which the calls should be mapped to.
- **tasks** (*iterable*) – The list of tasks to use as arguments in the function calls.
- **maxtasks** (*int*) – The maximum number of tasks of a process before it is replaced by a new process. If set to None, the process is never replaced.
- **init_args** (*tuple*) – The (func, namespace) tuple for the _process_init initialisation function.
- **max_broken_pool_restarts** (*int or None*) – The maximum number of attempts at restarting the process pool upon a BrokenPoolError. If set to None, the process pool may restart indefinitely.

Returns: **results** (*list*) – The list of results from the map operation.

Notes

The *maxtasks*, *namespace*, and *init_args* are ignored since these are not supported b y *concurrent.futures*. It seems that *concurrent.futures* works as if maxtasks==1, however this is unclear. The *init_args* functionality may be added if an initialiser is added to *concurrent.futures* - see http://bugs.python.org/issue21423.

magni.utils.multiprocessing._processing.**_map_using_mppool**(*func*, *tasks*, *init_args*, *maxtasks*, *max_broken_pool_restarts*)

Map a set of *tasks* to *func* and run them in parallel via multiprocessing

Parameters:
- **func** (*function*) – A function handle to the function which the calls should be mapped to.
- **tasks** (*iterable*) – The list of tasks to use as arguments in the function calls.
- **maxtasks** (*int*) – The maximum number of tasks of a process before it is replaced by a new process. If set to None, the process is never replaced.
- **init_args** (*tuple*) – The (func, namespace) tuple for the _process_init initialisation function.
- **max_broken_pool_restarts** (*int or None*) – The maximum number of attempts at restarting the process pool upon a BrokenPoolError. If set to None, the process pool may restart indefinitely.

Returns: **results** (*list*) – The list of results from the map operation.

Notes

The *max_broken_pool_restarts* is ignored since the BrokenPoolError handling has not yet been implemented in the multiprocessing.Pool - see https://bugs.python.org/issue22393 and https://bugs.python.org/issue9205.

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
```

__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.validation package

Subpackage 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

types
: Module providing abstract superclasses for validation.

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

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

enable_validate_once()
: Enable validating inputs only once (see Notes).

validate_generic(name, type, value_in=None, len_=None, keys_in=None,
: has_keys=None, superclass=None, ignore_none=False, var=None) Validate non-numeric objects.

validate_levels(name, levels)
: Validate containers and mappings as well as contained objects.

validate_numeric(name, type, range_='[-inf;inf]', shape=(), precision=None,
: ignore_none=False, var=None) Validate numeric objects.

validate_once(func)
: Decorate a function to allow for a one-time input validation (see Notes).

Notes

To be able to disable validation (and to ensure consistency), every public function or method should define a nested validation function with the name 'validate_input' which takes no arguments. This function should be decorated by **decorate_validation**, be placed in the beginning of the parent function or method, and be called as the first thing after its definition.

Functions in magni may be decorated by **validate_once**. If the validate once functionality is enabled, these functions only validate their input arguments on the first call to the function.

Examples

If, for example, the following function is defined:

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

This function 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 decorate_validation, validate_generic
>>> def greet(person, greeting):
...     @decorate_validation
...     def validate_input():
...         validate_generic('person', 'mapping', has_keys=('title', 'name'))
...         validate_generic('greeting', 'string')
...     validate_input()
...     print('{}, {} {}.'.format(greeting, person['title'], person['name']))
```

If, again, a list is passed as the first argument, a TypeError with the description “The value(s) of >>type(person)<<, <type ‘list’>, must be in (‘mapping’,).” is raised. 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.
```

Submodules

magni.utils.validation._generic module

Module providing the **validate_generic** function.

Routine listings

validate_generic(name, type, value_in=None, len_=None, keys_in=None, has_keys=None, superclass=None, ignore_none=False, var=None) Validate non-numeric objects.

magni.utils.validation._generic.**validate_generic**(*name*, *type_*, *value_in=None*, *len_=None*, *keys_in=None*, *has_keys=None*, *superclass=None*, *ignore_none=False*, *var=None*)

Validate non-numeric objects.

The present function is meant to validate the type or class of an object. Furthermore, if the object may only take on a limited number of values, the object can be validated against this list. In the case of collections (for example lists and tuples) and mappings (for example dictionaries), a specific length can be required. Furthermore, in the case of mappings, the keys can be validated by requiring and/or only allowing certain keys.

If the present function is called with *name* set to None, an iterable with the value ‘generic’ followed by the remaining arguments passed in the call is returned. This is useful in combination with the validation function **magni.utils.validation.validate_levels**.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **type_** (*None*) – One or more references to groups of types, specific types, and/or specific classes.
- **value_in** (*set-like*) – The list of values accepted. (the default is None, which implies that all values are accepted)
- **len_** (*int*) – The length required. (the default is None, which implies that all lengths are accepted)
- **keys_in** (*set-like*) – The list of accepted keys. (the default is None, which implies that all keys are accepted)
- **has_keys** (*set-like*) – The list of required keys. (the default is None, which implies that no keys are required)
- **superclass** (*class*) – The required superclass. (the default is None, which implies that no superclass is required)
- **ignore_none** (*bool*) – A flag indicating if the variable is allowed to be none. (the default is False)
- **var** (*None*) – The value of the variable to be validated.

See also:

magni.utils.validation.validate_levels()

Validate contained objects.

magni.utils.validation.validate_numeric()

Validate numeric objects.

Notes

name must refer to a variable in the parent scope of the function or method decorated by **magni.utils.validation.decorate_validation** which is closest to the top of the call stack. If *name* is a string then there must be a variable of that name in that scope. If *name* is a set-like object then there must be a variable having the first value in that set-like object as name. The remaining values are used as keys/indices on the variable to obtain the variable to be validated. For example, the *name* ('name', 0, 'key') refers to the variable "name[0]['key']".

*type_* is either a single value treated as a list with one value or a set-like object containing at least one value. Each value is either a specific type or class, or it refers to one or more types by having one of the string values 'string', 'explicit collection', 'implicict collection', 'collection', 'mapping', 'function', 'class'.

- 'string' tests if the variable is a str.
- 'explicit collection' tests if the variable is a list or tuple.
- 'implicit collection' tests if the variable is iterable.
- 'collection' is a combination of the two above.
- 'mapping' tests if the variable is a dict.
- 'function' tests if the variable is a function.
- 'class' tests if the variable is a type.

var can be used to pass the value of the variable to be validated. This is useful either when the variable cannot be looked up by *name* (for example, if the variable is a

property of the argument of a function) or to remove the overhead of looking up the value.

Examples

Every public function and method of the present package (with the exception of the functions of this subpackage itself) validates every argument and keyword argument using the functionality of this subpackage. Thus, for examples of how to use the present function, browse through the code.

magni.utils.validation._generic.**_check_inheritance**(*name*, *var*, *superclass*)

Check the superclasses of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **var** (*None*) – The value of the variable to be validated.
- **superclass** (*class*) – The required superclass.

magni.utils.validation._generic.**_check_keys**(*name*, *var*, *keys_in*, *has_keys*)

Check the keys of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **var** (*None*) – The value of the variable to be validated.
- **keys_in** (*set-like*) – The allowed keys.
- **has_keys** (*set-like*) – The required keys.

magni.utils.validation._generic.**_check_len**(*name*, *var*, *len_*)

Check the length of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **var** (*None*) – The value of the variable to be validated.
- **len_** (*int*) – The required length.

magni.utils.validation._generic.**_check_type**(*name*, *var*, *types_*)

Check the type of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **var** (*None*) – The value of the variable to be validated.
- **types_** (*set-like*) – The allowed types.

magni.utils.validation._generic.**_check_value**(*name*, *var*, *value_in*)

Check the value of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **var** (*None*) – The value of the variable to be validated.
- **value_in** (*set-like*) – The allowed values.

magni.utils.validation._levels module

Module providing the **validate_levels** function.

Routine listings

validate_levels(name, levels)

Validate containers and mappings as well as contained objects.

magni.utils.validation._levels.**validate_levels**(*name*, *levels*)

Validate containers and mappings as well as contained objects

The present function is meant to valdiate the 'levels' of a variable. That is, the value of the variable itself, the values of the second level (in case the value is a list, tuple, or dict), the values of the third level, and so on.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **levels** (*list or tuple*) – The list of levels.

See also:

magni.utils.validation.validate_generic()

Validate non-numeric objects.

magni.utils.validation.validate_numeric()

Validate numeric objects.

Notes

name must refer to a variable in the parent scope of the function or method decorated by **magni.utils.validation.decorate_validation** which is closest to the top of the call stack. If *name* is a string then there must be a variable of that name in that scope. If *name* is a set-like object then there must be a variable having the first value in that set-like object as name. The remaining values are used as keys/indices on the variable to obtain the variable to be validated. For example, the *name* ('name', 0, 'key') refers to the variable "name[0]['key']".

levels is a list containing the levels. The value of the variable is validated against the first level. In case the value is a list, tuple, or dict, the values contained in this are validated against the second level and so on. Each level is itself a list with the first value being either 'generic' or 'numeric' followed by the arguments that should be passed to the respective function (with the exception of *name* which is automatically prepended by the present function).

Examples

Every public function and method of the present package (with the exception of the functions of this subpackage itself) validates every argument and keyword argument using the functionality of this subpackage. Thus, for examples of how to use the present function, browse through the code.

magni.utils.validation._levels.**_validate_level**(*name*, *var*, *levels*, *index=0*)

Validate a level.

Parameters:
- **name** (*None*) – The name of the variable.
- **var** (*None*) – The value of the variable.
- **levels** (*set-like*) – The levels.
- **index** (*int*) – The index of the current level. (the default is 0)

magni.utils.validation._numeric module

Module providing the **validate_numeric** function.

Routine listings

validate_numeric(name, type, range_='[-inf;inf]', shape=(), precision=None, ignore_none=False, var=None) Validate numeric objects.

magni.utils.validation._numeric.**validate_numeric**(*name*, *type_*, *range_='[-inf;inf]'*, *shape=()*, *precision=None*, *ignore_none=False*, *var=None*)

Validate numeric objects.

The present function is meant to valdiate the type or class of an object. Furthermore, if the object may only take on a connected range of values, the object can be validated against this range. Also, the shape of the object can be validated. Finally, the precision used to represent the object can be validated.

If the present function is called with *name* set to None, an iterable with the value 'numeric' followed by the remaining arguments passed in the call is returned. This is useful in combination with the validation function **magni.utils.validation.validate_levels**.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **type_** (*None*) – One or more references to groups of types.
- **range_** (*None*) – The range of accepted values. (the default is '[-inf;inf]', which implies that all values are accepted)
- **shape** (*list or tuple*) – The accepted shape. (the default is (), which implies that only scalar values are accepted)
- **precision** (*None*) – One or more precisions.
- **ignore_none** (*bool*) – A flag indicating if the variable is allowed to be none. (the default is False)
- **var** (*None*) – The value of the variable to be validated.

See also:

magni.utils.validation.validate_generic()
Validate non-numeric objects.

magni.utils.validation.validate_levels()
Validate contained objects.

Notes

name must refer to a variable in the parent scope of the function or method decorated

by **magni.utils.validation.decorate_validation** which is closest to the top of the call stack. If *name* is a string then there must be a variable of that name in that scope. If *name* is a set-like object then there must be a variable having the first value in that set-like object as name. The remaining values are used as keys/indices on the variable to obtain the variable to be validated. For example, the *name* ('name', 0, 'key') refers to the variable "name[0]['key']".

*type_* is either a single value treated as a list with one value or a set-like object containing at least one value. Each value refers to a number of data types depending if the string value is 'boolean', 'integer', 'floating', or 'complex'.

- 'boolean' tests if the variable is a bool or has the data type **numpy.bool8**.
- 'integer' tests if the variable is an int or has the data type **numpy.int8**, **numpy.int16**, **numpy.int32**, or **numpy.int64**.
- 'floating' tests if the variable is a float or has the data type **numpy.float16**, **numpy.float32**, **numpy.float64**, or **numpy.float128**.
- 'complex' tests if the variable is a complex or has the data type *numpy.complex32*, **numpy.complex64**, or **numpy.complex128**.

Because *bool* is a subclass of *int*, a *bool* will pass validation as an 'integer'. This, however, is not the case for **numpy.bool8**.

*range_* is either a list with two strings or a single string. In the latter case, the default value of the argument is used as the second string. The first value represents the accepted range of real values whereas the second value represents the accepted range of imaginary values. Each string consists of the following parts:

- One of the following delimiters: '[', '(', ']'.
- A numeric value (or '-inf').
- A semi-colon.
- A numeric value (or 'inf').
- One of the following delimiters: ']', ')', '['.

shape is either None meaning that any shape is accepted or a list of integers. In the latter case, the integer -1 may be used to indicate that the given axis may have any length.

precision is either an integer treated as a list with one value or a list or tuple containing at least one integer. Each value refers to an accepted number of bits used to store each value of the variable.

var can be used to pass the value of the variable to be validated. This is useful either when the variable cannot be looked up by *name* (for example, if the variable is a property of the argument of a function) or to remove the overhead of looking up the value.

Examples

Every public function and method of the present package (with the exception of the functions of this subpackage itself) validates every argument and keyword argument using the functionality of this subpackage. Thus, for examples of how to use the present function, browse through the code.

magni.utils.validation._numeric.**_check_precision**(*name*, *dtype*, *types_*, *precision*)

Check the precision of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **dtype** (*type*) – The data type of the variable to be validated.
- **types_** (*set-like*) – The list of accepted types.
- **precision** (*None*) – The accepted precision(s).

magni.utils.validation._numeric.**_check_range**(*name*, *bounds*, *range_*)

Check the range of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **bounds** (*list or tuple*) – The bounds of the variable to be validated.
- **range_** (*None*) – The accepted range(s).

magni.utils.validation._numeric.**_check_shape**(*name*, *dshape*, *shape*)

Check the shape of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **dshape** (*list or tuple*) – The shape of the variable to be validated.
- **shape** (*list or tuple*) – The accepted shape.

magni.utils.validation._numeric.**_check_type**(*name*, *dtype*, *types_*)

Check the type of a variable.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **dtype** (*type*) – The data type of the variable to be validated.
- **types_** (*set-like*) – The accepted types.

magni.utils.validation._numeric.**_examine_var**(*name*, *var*)

Examine a variable.

The present function examines the data type, the bounds, and the shape of a variable. The variable can be of a built-in type, a numpy type, or a subclass of the **magni.utils.validation.types.MatrixBase** class.

Parameters:
- **name** (*None*) – The name of the variable to be validated.
- **var** (*None*) – The value of the variable to be examined.

Returns:
- **dtype** (*type*) – The data type of the examined variable.
- **bounds** (*tuple*) – The bounds of the examined variable.
- **dshape** (*tuple*) – The shape of the examined variable.

magni.utils.validation._util module

Module providing functionality for disabling validation.

The disabling functionality is provided through a decorator for validation functions and a function for disabling validation. Furthermore, the present module provides functionality which is internal to **magni.utils.validation**.

Routine listings

decorate_validation(func)
: Decorate a validation function to allow disabling of validation checks.

disable_validation()
: Disable validation checks in **magni**.

enable_validate_once()
: Enable validating inputs only once in certain functions in **magni**.

get_var(name)
: Retrieve the value of a variable through call stack inspection.

report(type, description, format_args=(), var_name=None, var_value=None,
: expr='{}', prepend='') Raise an exception.

validate_once(func)
: Decorate a function to allow for a one-time input validation only.

magni.utils.validation._util.**decorate_validation**(*func*)

Decorate a validation function to allow disabling of validation checks.

Parameters:	**func** (*function*) – The validation function to be decorated.
Returns:	**func** (*function*) – The decorated validation function.

See also:

disable_validation()
: Disabling of validation checks.

Notes

This decorater wraps the validation function in another function which checks if validation has been disabled. If validation has been disabled, the validation function is not called. Otherwise, the validation function is called.

Examples

See **disable_validation** for an example.

magni.utils.validation._util.**disable_validation**()

Disable validation checks in **magni**.

See also:

decorate_validation()
: Decoration of validation functions.

Notes

This function merely sets a global flag and relies on **decorate_validation** to perform the actual disabling.

Examples

An example of a function which accepts only an integer as argument:

```
>>> import magni
>>> def test(arg):
...     @magni.utils.validation.decorate_validation
...     def validate_input():
...         magni.utils.validation.validate_numeric('arg', 'integer')
...     validate_input()
```

If the function is called with anything but an integer, it fails:

```
>>> try:
...     test('string')
... except BaseException:
...     print('An exception occured')
... else:
...     print('No 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')
... except BaseException:
...     print('An exception occured')
... else:
...     print('No exception occured')
No exception occured
```

Although strongly discouraged, validation can be re-enabled after being disabled in the following way:

```
>>> import magni.utils.validation._util
>>> magni.utils.validation._util._disabled = False
```

As this interface is not public it may be subject to changes in future releases without further notice.

magni.utils.validation._util.**enable_validate_once**()

Enable validating inputs only once in certain functions in **magni**.

See also:

validate_once()

Decoration of functions.

Notes

This function merely sets a global flag and relies on **validate_once** to implement the actual “validate once” functionality.

Examples

An example of a function which accepts only an integer as argument:

```
>>> import magni
>>> @magni.utils.validation.validate_once
... def test(arg):
...     @magni.utils.validation.decorate_validation
...     def validate_input():
...         magni.utils.validation.validate_numeric('arg', 'integer')
...     validate_input()
```

If “validate once” is enabled, the function validation is only called on its first run:

```
>>> magni.utils.validation.enable_validate_once()
>>> try:
...     test('string')
... except BaseException:
...     print('An exception occured')
... else:
...     print('No exception occured')
An exception occured
```

```
>>> try:
...     test('string')
... except BaseException:
...     print('An exception occured')
... else:
...     print('No exception occured')
No exception occured
```

magni.utils.validation._util.**get_var**(*name*)

Retrieve the value of a variable through call stack inspection.

name must refer to a variable in the parent scope of the function or method decorated by **magni.utils.validation.decorate_validation** which is closest to the top of the call stack. If *name* is a string then there must be a variable of that name in that scope. If *name* is a set-like object then there must be a variable having the first value in that set-like object as name. The remaining values are used as keys/indices on the variable to obtain the variable to be validated. For example, the *name* (‘name’, 0, ‘key’) refers to the variable “name[0][‘key’]”.

Parameters:	**name** (*None*) – The name of the variable to be retrieved.
Returns:	**var** (*None*) – The value of the retrieved variable.

Notes

The present function searches the call stack from top to bottom until it finds a function named ‘wrapper’ defined in this file. That is, until it finds a decorated validation

function. The present function then looks up the variable indicated by *name* in the parent scope of that decorated validation function.

magni.utils.validation._util.**report**(*type_*, *description*, *format_args=()*, *var_name=None*, *var_value=None*, *expr='{}'*, *prepend=''*)

Raise an exception.

The type of the exception is given by *type_*, and the message can take on many forms. This ranges from a single description to a description formatted using a number of arguments, prepended by an expression using a variable name followed by the variable value, prepended by another description.

Parameters:
- **type_** (*type*) – The exception type.
- **description** (*str*) – The core description.
- **format_args** (*list or tuple*) – The arguments which *description* is formatted with. (the default is (), which implies no arguments)
- **var_name** (*None*) – The name of the variable which the description concerns. (the default is None, which implies that no variable name and value is prepended to the description)
- **var_value** (*None*) – The value of the variable which the description concerns. (the default is None, which implies that the value is looked up from the *var_name* in the call stack if *var_name* is not None)
- **expr** (*str*) – The expression to evaluate with the variable name. (the default is '{}', which implies that the variable value is used directly)
- **prepend** (*str*) – The text to prepend to the message generated by the previous arguments. (the default is '')

Notes

name must refer to a variable in the parent scope of the function or method decorated by **magni.utils.validation.decorate_validation** which is closest to the top of the call stack. If *name* is a string then there must be a variable of that name in that scope. If *name* is a set-like object then there must be a variable having the first value in that set-like object as name. The remaining values are used as keys/indices on the variable to obtain the variable to be validated. For example, the *name* ('name', 0, 'key') refers to the variable "name[0]['key']".

magni.utils.validation._util.**validate_once**(*func*)

Decorate a function to allow for a one-time input validation only.

Parameters: **func** (*function*) – The validation function to be decorated.
Returns: **func** (*function*) – The decorated validation function.

See also:

enable_validate_once()

Enabling allow validate once functionality.

Notes

This decorater wraps any function. It checks if the “validate once” has been enabled. If it has been enabled, the validation function is only called on the first run. Otherwise, the validation function is always called if not otherwise disabled.

magni.utils.validation.types module

Module providing abstract superclasses for validation.

Routine listings

MatrixBase(object)
Abstract base class of custom matrix classes.

class magni.utils.validation.types.**MatrixBase**(*dtype*, *bounds*, *shape*)

Bases: **object**

Abstract base class of custom matrix classes.

The **magni.utils.validation.validate_numeric** function accepts built-in numeric types, numpy built-in numeric types, and subclasses of the present class. In order to perform validation checks, the validation function needs to know the data type, the bounds, and the shape of the variable. Thus, subclasses must call the init function of the present class with these arguments.

Parameters:
- **dtype** (*type*) – The data type of the values of the instance.
- **bounds** (*list or tuple*) – The bounds of the values of the instance.
- **shape** (*list or tuple*) – The shape of the instance.

bounds
list or tuple

The bounds of the values of the instance.

dtype
type

The data type of the values of the instance.

shape
list or tuple

The shape of the instance.

Notes

dtype is either a built-in numeric type or a numpy built-in numeric type.

If the matrix has complex values, **bounds** is a list with two values; The bounds on the real values and the bounds on the imaginary values. If, on the other hand, the matrix

has real values, **bounds** has one value; The bounds on the real values. Each such bounds value is a list with two real, numeric values; The lower bound (that is, the minimum value) and the upper bound (that is, the maximum value).

bounds

dtype

shape

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:

```
>>> import os
>>> 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(object)

Provide functionality to access a set of configuration options.

Notes

This module does not itself contain any configuration options and thus has no access to any configuration options unlike the other config modules of **magni**.

class magni.utils.config.**Configger**(*params*, *valids*)

Bases: **object**

Provide functionality to access a set of configuration options.

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

Parameters:
- **params** (*dict*) – The configuration options and their default values.
- **valids** (*dict*) – The validation schemes of the configuration options.

See also:

magni.utils.validation
Validation.

Notes

valids must contain the same keys as *params*. For each key in 'valids', the first value is the validation function ('generic', 'levels', or 'numeric'), whereas the remaining values are passed to that validation function.

Examples

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

```
>>> import magni
>>> from magni.utils.config import Configger
>>> valid = magni.utils.validation.validate_generic(None, 'string')
>>> config = Configger({'key': 'default'}, {'key': valid})
```

The number of parameters can be retrieved as the length:

```
>>> len(config)
1
```

That parameter can be retrieved in a number of ways:

```
>>> config['key']
'default'
```

```
>>> for key, value in config.items():
...     print('key: {!r}, value: {!r}'.format(key, value))
key: 'key', value: 'default'
```

```
>>> for key in config.keys():
...     print('key: {!r}'.format(key))
key: 'key'
```

```
>>> for value in config.values():
...     print('value: {!r}'.format(value))
value: 'default'
```

Likewise, the parameter can be changed in a number of ways:

```
>>> config['key'] = 'value'
>>> config['key']
'value'
```

```
>>> config.update({'key': 'value changed by dict'})
>>> config['key']
'value changed by dict'
```

```
>>> config.update(key='value changed by keyword')
>>> config['key']
'value changed by keyword'
```

Finally, the parameter can be reset to the default value at any point:

```
>>> config.reset()
>>> config['key']
'default'
```

_funcs *= {'generic': <function validate_generic at 0x7f69a38515f0>, 'levels': <function validate_levels at 0x7f69a38519b0>, 'numeric': <function validate_numeric at 0x7f699b6f21b8>}*

__getitem__(*name*)

Get the value of a configuration parameter.

Parameters: **name** (*str*) – The name of the parameter.
Returns: **value** (*None*) – The value of the parameter.

__len__()

Get the number of configuration parameters.

Returns: **length** (*int*) – The number of parameters.

__setitem__(*name*, *value*)

Set the value of a configuration parameter.

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

Parameters:
- **name** (*str*) – The name of the parameter.
- **value** (*None*) – The new value of the parameter.

get(*key=None*)

Deprecated method.

See also:

Configger.__getitem__()
Replacing method.

Configger.items()
Replacing method.

Configger.keys()
Replacing method.

Configger.values()
Replacing method.

items()
Get the configuration parameters as key, value pairs.

Returns: **items** (*set-like*) – The list of parameters.

keys()
Get the configuration parameter keys.

Returns: **keys** (*set-like*) – The keys.

reset()
Reset the parameter values to the default values.

set(*dictionary={}*, ***kwargs*)
Deprecated method.

See also:

Configger.__setitem__()
Replacing function.

update(*params={}*, ***kwargs*)
Update the value of one or more configuration parameters.

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

Parameters:
- **params** (*dict, optional*) – A dictionary containing the key and values to update. (the default value is an empty dictionary)
- **kwargs** (*dict*) – Keyword arguments being the key and values to update.

values()
Get the configuration parameter values.

Returns: **values** (*set-like*) – The values.

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(magni.utils.validation.types.MatrixBase)
: Wrap fast linear operations in a matrix emulator.

MatrixCollection(magni.utils.validation.types.MatrixBase)
: Wrap multiple matrix emulators in a single matrix emulator.

See also:

magni.imaging._fastops
: Fast linear operations.

class magni.utils.matrices.**Matrix**(*func*, *conj_trans*, *args*, *shape*, *is_complex=False*, *is_valid=True*)

Bases: **magni.utils.validation.types.MatrixBase**

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 ndarray instance without explicitly forming the array. This basic interface allows instances to be multiplied with vectors, to be transposed, to be complex conjuagted, and to assume a shape. Also, instances have an attribute which explicitly forms the matrix as an ndarray.

Parameters:

- **func** (*function*) – The fast linear operation applied to the vector when multiplying the matrix with a vector.
- **conj_trans** (*function*) – The fast linear operation applied to the vector when multiplying the complex conjuated transposed matrix with a vector.
- **args** (*list or tuple*) – The arguments which should be passed to *func* and *conj_trans* in addition to the vector.
- **shape** (*list or tuple*) – The shape of the emulated matrix.
- **is_complex** (*bool*) – The indicator of whether or not the emulated matrix is defined for the complex numbers (the default is False, which implies that the emulated matrix is defined for the real numbers only).
- **is_valid** (*bool*) – The indicator of whether or not the fast linear operation corresponds to a valid matrix (se discussion below) (the default is True, which implies that the matrix is considered valid).

See also:

magni.utils.validation.types.MatrixBase
Superclass of the present class.

Notes

The *is_valid* indicator is an implementation detail used to control possibly missing operations in the fast linear operation. For instance, consider the Discrete Fourier Transform (DFT) and its inverse transform. The forward DFT transform is a matrix-vector product involving the corresponding DFT matrix. The inverse transform is a matrix-vetor product involving the complex conjugate transpose DFT matrix. That is, it involves complex conjugation and transposition, both of which are (individually) valid transformations of the DFT matrix. However, when implementing the DFT (and its inverse) using a Fast Fourier Transform (FFT), only the combined (complex conjugate, transpose) operation is available. Thus, when using an FFT based *magni.util.matrices.Matrix*, one can get, e.g., a **Matrix.T** object corresponding to its transpose. However, the operation of computing a matrix-vector product involving the tranpose DFT matrix is not available and the **Matrix.T** is, consequently, considered an invalid matrix. Only the combined *Matrix.T.conj()* or *Matrix.conj().T* is considered a valid matrix.

Warning: The tranpose operation changed in **magni** 1.5.0.
For complex matrices, the the **T** tranpose operation now yields an invalid matrix as described above. Prior to **magni** 1.5.0, the **T** would yield the “inverse” which would usually be the complex conjugated transpose for complex matrices.

Examples

For example, the negative identity matrix could be emulated as

```
>>> import numpy as np, magni
>>> 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)
>>> np.set_printoptions(suppress=True)
>>> 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.]])
```

__array__()

Return ndarray representation of the matrix.

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.

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 the same as those of the current matrix except swapped. The shape is modified accordingly. This returns an *invalid* matrix if the entries are complex numbers as only the complex conjugate transpose is considered valid.

conj()

Get the complex conjugate of the matrix.

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

Notes

The fast linear operation and the fast linear transposed operation of the resulting

matrix are the same as those of the current matrix. This returns an *invalid* matrix if the entries are complex numbers as only the complex conjugate transpose is considered valid.

dot(**args*, ***kwargs*)

Multiply the matrix with a vector.

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

Notes

This method honors **magni.utils.validation.enable_validate_once**.

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

Bases: **magni.utils.validation.types.MatrixBase**

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 ndarray instance without explicitly forming the ndarray. This basic interface allows instances to be multiplied with vectors, to be transposed, to be complex conjugated, 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:

magni.utils.validation.types.MatrixBase
: Superclass of the present class.

Matrix
: Matrix emulator.

Examples

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

```
>>> import numpy as np, magni
>>> func = lambda vec: -vec
>>> negate = magni.utils.matrices.Matrix(func, func, (), (3, 3))
>>> np.set_printoptions(suppress=True)
>>> 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.]])
```

__array__()

Return ndarray representation of the matrix.

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.

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.

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

T

Get the transpose of the matrix.

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

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

conj()

Get the complex conjugate of the matrix.

The complex conjugate of the product of the number of matrices is the product of the complex conjugates of the matrices.

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

dot(**args*, ***kwargs*)

Multiply the matrix with a vector.

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

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

Notes

This method honors **magni.utils.validation.enable_validate_once**.

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=None)

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 / prop_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.

```
>>> import magni
>>> 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/)

__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 tuple, optional*) – Colormap to be used by matplotlib (the default is None, which implices that the ‘coolwarm’ colormap is used). If a tuple is supplied it must be a (‘colormap_name’, matplotlib.colors.Colormap()) tuple.
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.types module

Module providing custom data types.

Routine listings

ClassProperty(property)

Class property.

ReadOnlyDict(collections.OrderedDict)

Read-only ordered dict.

class magni.utils.types.**ClassProperty**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Bases: **property**

Class property.

The present class is a combination of the built-in property type and the built-in classmethod type. That is, it is a property which is invoked on a class rather than an instance but is available to both the class and its instances.

Parameters:

- **fget** (*function, optional*) – The getter function of the property. (the default is None, which implies that the property cannot be read)
- **fset** (*function, optional*) – The setter function of the property. (the default is None, which implies that the property cannot be written)
- **fdel** (*function, optional*) – The deleter function of the property. (the default is None, which implies that the property cannot be deleted)
- **doc** (*string, optional*) – The docstring of the property. (the default is None, which implies that the docstring of the getter function, if any, is used)

See also:

property

Superclass from which all behaviour is inherited or extended.

Examples

The following example illustrates the difference between regular properties and class properties. First, the example class is defined and instantiated:

```
>>> from magni.utils.types import ClassProperty
>>> class Example(object):
...     _x_class = 0
...     def __init__(self, x):
...         self._x_instance = x
...     @ClassProperty
...     def x_class(class_):
...         return class_._x_class
...     @property
...     def x_instance(self):
...         return self._x_instance
>>> example = Example(1)
```

The regular read-only property works on the instance:

```
>>> print('{!r}'.format(example.x_instance))
1
>>> print('Is property? {!r}'.format(isinstance(Example.x_instance,
...                                     property)))
Is property? True
```

The class property, on the other hand, works on the class:

```
>>> print('{!r}'.format(example.x_class))
0
>>> print('Is property? {!r}'.format(isinstance(Example.x_class,
...                                     property)))
Is property? False
>>> print('{!r}'.format(Example.x_class))
0
```

__get__(*obj*, *class_=None*)

The get method of the property.

Parameters:
- **obj** (*object*) – The instance on which the property is requested.
- **class_** (*type, optional*) – The class on which the property is requested. (the default is None, which implies that the class is retrieved from *obj*)

Returns: **value** (*None*) – The value of the property.

See also:

property.__get__()

The method being extended.

__set__(*obj*, *value*)

The set method of the property.

Parameters:
- **obj** (*object*) – The instance on which the property is requested.
- **value** (*None*) – The value which should be assigned to the property.

See also:

property.__set__()

The method being extended.

__delete__(*obj*)

The delete method of the property.

Parameters: **obj** (*object*) – The instance on which the property is requested.

See also:

property.__delete__()

The method being extended.

class magni.utils.types.**ReadOnlyDict**(**args*, ***kwargs*)

Bases: **collections.OrderedDict**

Read-only ordered dict.

The present ordered dict subclass has its non-read-only methods disabled.

See also:

collections.OrderedDict
Superclass from which all read-only behaviour is inherited.

Examples

This ordered dict subclass exposes the same interface as the OrderedDict class when not using methods that alter the dict.

```
>>> from magni.utils.types import ReadOnlyDict
>>> d = ReadOnlyDict(key='value')
>>> for item in d.items():
...     print('{!r}'.format(item))
('key', 'value')
```

However, any attempt to assign another value to the property raises an exception:

```
>>> try:
...     d['key'] = 'new value'
... except Exception:
...     print('An exception occured.')
An exception occured.
```

__delitem__(*name*)
Prevent deletion of items.

Parameters: **name** (*str*) – The name of the item to be deleted.

__getattribute__(*name*)
Return the requested attribute unless it is a non-read-only method.

The purpose of this overwrite is to disable access to the following non-read-only dict methods: **clear**, **pop**, **popitem**, **setdefault**, and **update**. The first two methods are disabled otherwise.

Parameters: **name** (*str*) – The name of the requested attribute.
Returns: **attribute** (*None*) – The requested attribute.

Notes

__getattribute__ is implicitly called, when the attribute of an object is accessed as *object.attribute*.

__setitem__(*name*, *value*)
Prevent overwrite of items.

Parameters:
- **name** (*str*) – The name of the item to be overwritten.
- **value** (*None*) – The value to be written.

Magni 1.5.0 documentation »