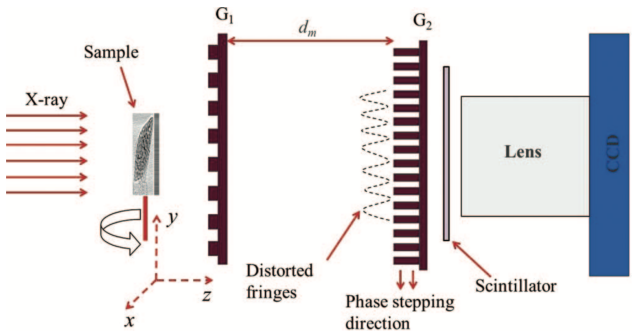
wavePy Documentation

Release 0.0.1

Argonne National Laboratory

Jul 31, 2017

Contents



wavePy is a python toolbox that provides ...

Here is how to add a link to your documentation Docs and here is how to add a reference *[A1]*.

You can include equations like:

$$V(x) = \int_{E_1}^{E_2} I(\theta, E) \cdot F(E)\, dE \cdot \left(\frac{1-\eta}{\sigma\sqrt{2\pi}} \right) \cdot exp\left(\frac{x^2}{2\sigma^2} \right) + \eta \cdot \frac{\sigma}{2\pi} \cdot \frac{1}{x^2 + \left(\frac{\sigma}{2}\right)^2}$$

and tables:

Member	Type	Example
first	ordinal	1st
second	ordinal	2nd
third	ordinal	3rd

CHAPTER 1

Note

Note that, according to Sphinx documentation, when you put math markup in Python docstrings read by autodoc, you have to double all backslashes. To obtain the equation above in this case we would use:

```
>>> V(x) = \\int_{E_{1}}^{E_{2}} I(\\theta,E) \\cdot F(E)\\,dE \\cdot \\left(\\frac{1-
↪\\eta}{\\sigma\\sqrt{2\\pi}}\\right) \\cdot exp\\left({\\frac{x^2}{2\\sigma^2}}
↪\\right) + \\eta \\cdot \\frac{\\sigma}{2\\pi} \\cdot \\frac{1}{x^2 + \\left(\\frac
↪{\\sigma}{2}\\right)^2}
```

CHAPTER 2

Features

- List here
- the wavePy package features

CHAPTER 3

Contribute

- Documentation: https://github.com/wavepy/wavepy/tree/master/doc
- Issue Tracker: https://github.com/wavepy/wavepy/docs/issues
- Source Code: https://github.com/wavepy/wavepy/wavepy

CHAPTER 4

Content

About

This section describes what the wavePy project is about.

Install

This section covers the basics of how to download and install wavePy. We recommend you to install the Anaconda Python distribution.

Contents:

Installing from source

Clone the wavePy from GitHub repository:

```
git clone https://github.com/wavepy/wavepy.git wavepy
```

then:

```
cd wavepy
python setup.py install
```

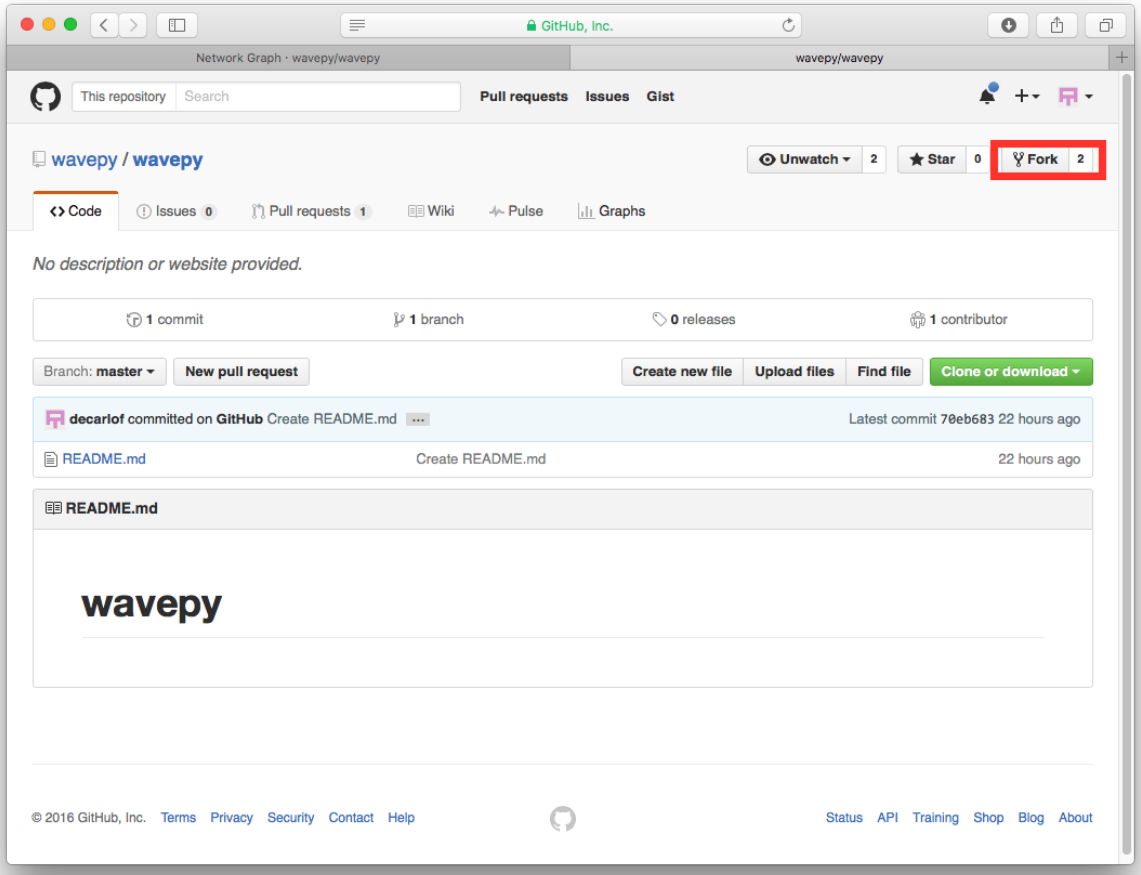
Development

This section explains the basics for developers who wish to contribute to the wavePy project. WavePy is mantained on GitHub and uses the fork / pull request mechanism for accepting developer contributions.

Contents:

Fork the repository

The project is maintained on GitHub, which is a version control and a collaboration platform for software developers. To start first register on GitHub and fork the wavePy repository by clicking the **Fork** button in the header of the wavePy repository:

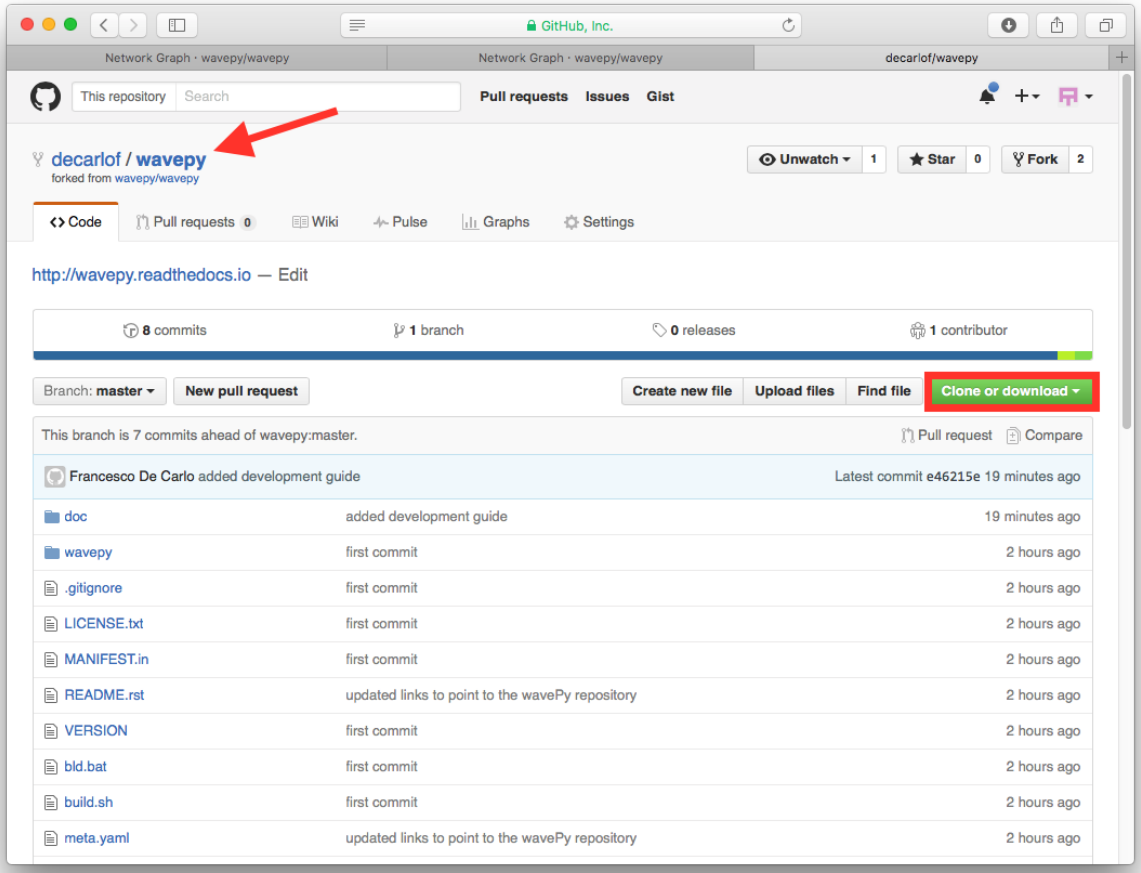


This successfully creates a copy of the project in your personal GitHub space.

Clone the repository

The next thing you want to do is to clone the repository you just created in your personal GitHub space to your local machine.

You can do this by clicking the **Clone in Desktop** button in the bottom of the right hand side bar:



This will launch the GitHub desktop application (available for both Mac and Win) and ask you where you want to save it. Select a location in your computer and feel comfortable with making modifications in the code.

Coding conventions

We try to keep a consistent and readable code. So, please keep in mind the following style and syntax guidance before you start coding.

First of all the code should be well documented, easy to understand, and integrate well into the rest of the project. For example, when you are writing a new function always describe the purpose and the parameters:

```
def my_awesome_func(a, b):
    """
    Adds two numbers.

    Parameters
    ----------
    a : scalar (float)
        First number to add

    b : scalar (float)
        Second number to add
```

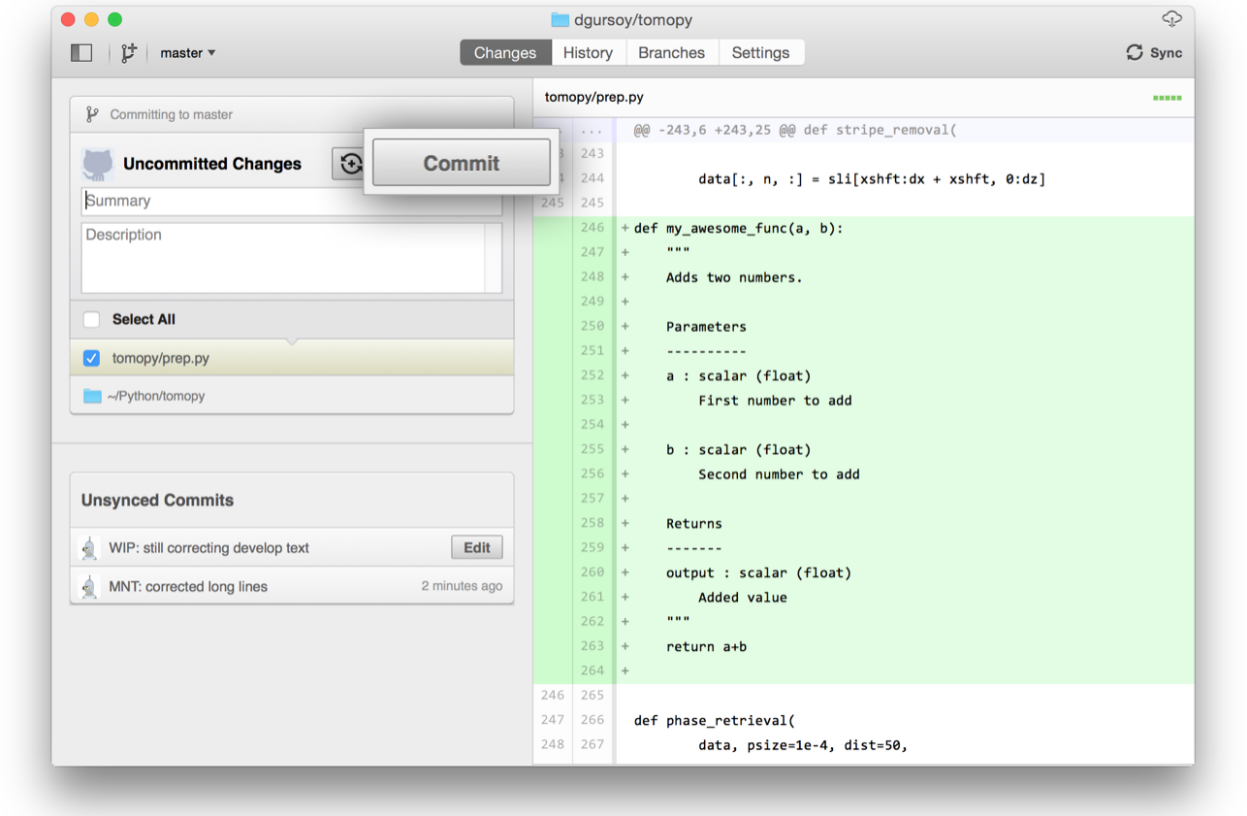
```
    Returns
    -------
    output : scalar (float)
        Added value
    """
    return a+b
```

Package versioning

We follow the X.Y.Z (Major.Minor.Patch) semantic for package versioning. The version should be updated before each pull request accordingly. The patch number is incremented for minor changes and bug fixes which do not change the software's API. The minor version is incremented for releases which add new, but backward-compatible, API features, and the major version is incremented for API changes which are not backward-compatible. For example, software which relies on version 2.1.5 of an API is compatible with version 2.2.3, but not necessarily with 3.2.4.

Commiting changes

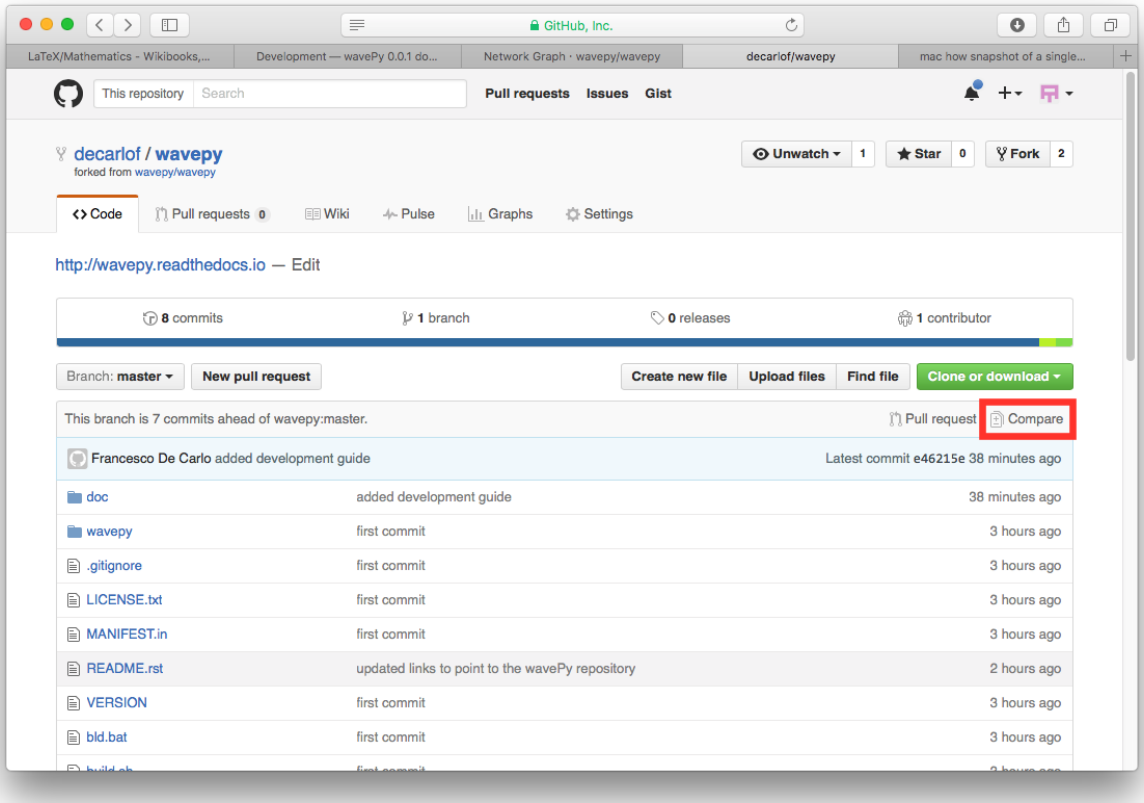
After making some changes in the code, you may want to take a *snapshot* of the edits you made. That's when you make a *commit*. To do this, launch the GitHub desktop application and it should provide you all the changes in your code since your last commit. Write a brief *Summary* and *Description* about the changes you made and click the **Commit** button:



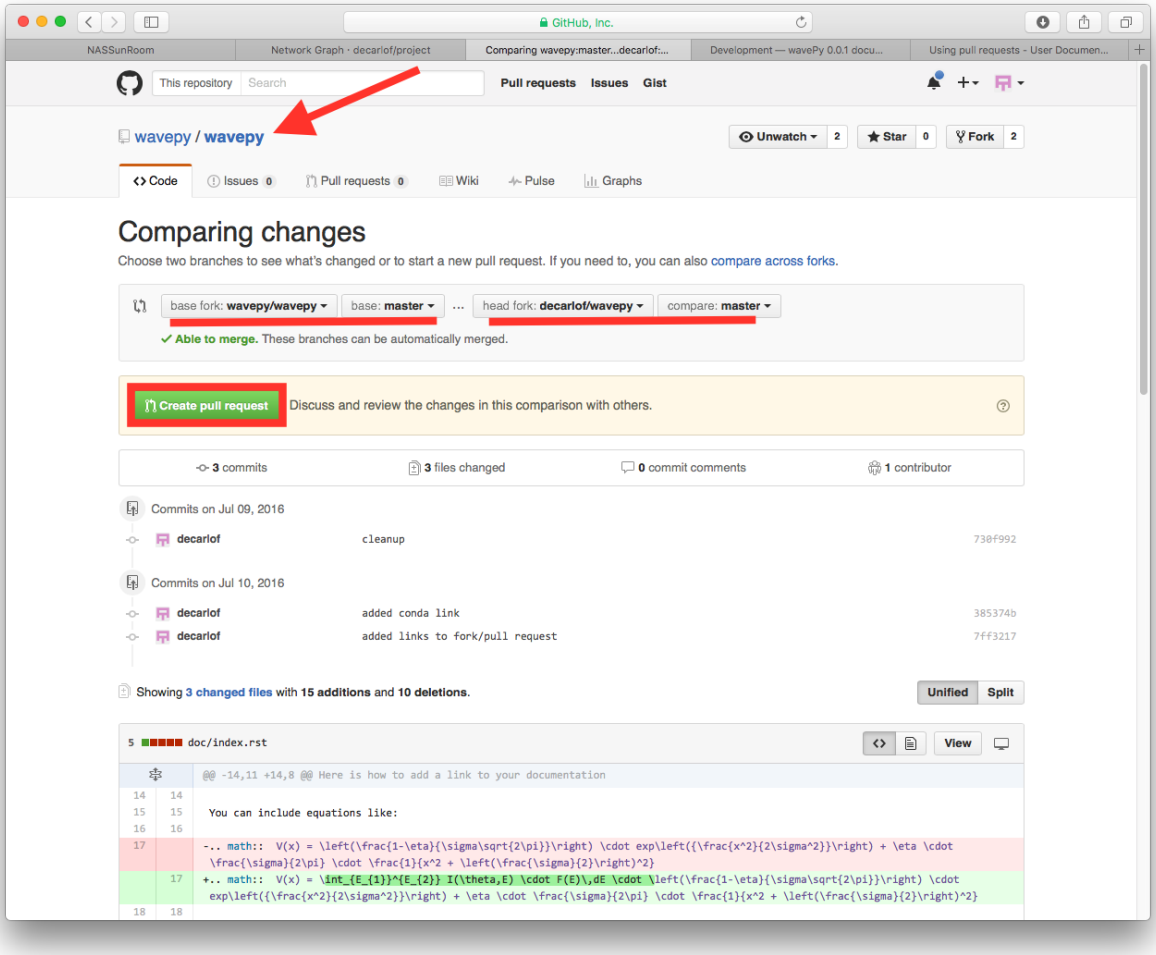
You can continue to make changes, add modules, write your own functions, and take more *Commit snapshots* of your code writing process.

Contributing back

Once you feel that the functionality you added would benefit the community, then you should consider contributing back to the wavePy project. For this, go to your online GitHub repository of the project and click on the *compare* button to compare, review and create a pull request.



After clicking on this button, you are presented with a review page where you can get a high-level overview of what exactly has changed between your forked branch and the original wavePy repository. When you're ready to submit your pull request, click **Create pull request**:



Clicking on **Create pull request** sends you to a discussion page, where you can enter a title and optional description. It's important to provide as much useful information and a rationale for why you're making this Pull Request in the first place.

When you're ready typing out your heartfelt argument, click on **Send pull request**.

You're done!

API reference

If the functions are derived from a publication add here the referenced citation as *[B10]*.

wavePy Modules:

`wavepy.utils`

Utility functions to help.

Functions:

crop_matrix_at_indexes(input_matrix, ...)	Alias for `np.copy(inputMatrix[i_min:i_max, j_min:j_max])`
date_now_str()	Returns the current date as a string in the format YYmmDD.
datetime_now_str()	Returns the current date and time as a string in the format YYmmDD_HHMMSS.
dummy_images([imagetype, shape])	Dummy images for simple tests.
find_nearest_value(input_array, value)	Alias for
find_nearest_value_index(input_array, value)	Similar to *wavepy.utils.find_nearest_value()*, but returns
`fouriercoordmatrix`(npointsx, deltax, ...)	For back compability
`fouriercoordvec`(npoints, delta)	For back compability
graphical_roi_idx(zmatrix[, verbose, ...])	Function to define a rectangular region of interest (ROI) in an image.
graphical_select_point_idx(zmatrix[, ...])	Plot a 2D array and allow to pick a point in the image.
h5_list_of_groups(h5file)	Get the names of all groups and subgroups in a hdf5 file.
load_ini_file(inifname)	**Parameters** **inifname** (*str*) – name of the `*.ini` file.
nan_mask_threshold(input_matrix[, threshold])	Calculate a square mask for array above OR below a threshold
plot_profile(xmatrix, ymatrix, zmatrix[, ...])	Plot contourf in the main graph plus profiles over vertical and horizontal lines defined with mouse.
print_blue(message)	Print with colored characters.
print_color(message[, color, highlights, attrs])	Print with colored characters.
print_red(message)	Print with colored characters.
progress_bar4pmap(res[, sleep_time])	Progress bar from `tqdm` to be used with the function `multiprocessing.starmap_async()`.
realcoordmatrix(npointsx, deltax, npointsy, ...)	Build a matrix (2D array) with real space coordinates based on the number of points and bin (pixels) size.
realcoordmatrix_fromvec(xvec, yvec)	Alias for `np.meshgrid(xvec, yvec)`
realcoordvec(npoints, delta)	Build a vector with real space coordinates based on the number of points and bin (pixels) size.
`rotate_img_graphical`(array2D[, order, mode, ...])	GUI to rotate an image
select_dir([message_to_print, pattern])	List subdirectories of the current working directory, and expected the user to choose one of them.
select_file([pattern, message_to_print])	List files under the subdirectories of the current working directory, and expected the user to choose one of them.
time_now_str()	Returns the current time as a string in the format HHMMSS.

`wavepy.utils.`**`print_color`**(*message*, *color='red'*, *highlights='on_white'*, *attrs=''*)

Print with colored characters. It is only a alias for colored print using the package `termcolor` and equals to:

```
print(termcolor.colored(message, color, highlights, attrs=attrs))
```

See options at https://pypi.python.org/pypi/termcolor

Parameters

- **message** (*str*) – Message to print.
- **color, highlights** (*str*)
- **attrs** (*list*)

wavepy.utils.**print_red**(*message*)
Print with colored characters. It is only a alias for colored print using the package `termcolor` and equals to:

```
print(termcolor.colored(message, color='red'))
```

Parameters **message** (*str*) – Message to print.

wavepy.utils.**print_blue**(*message*)
Print with colored characters. It is only a alias for colored print using the package `termcolor` and equals to:

```
print(termcolor.colored(message, color='blue'))
```

Parameters **message** (*str*) – Message to print.

wavepy.utils.**plot_profile**(*xmatrix*, *ymatrix*, *zmatrix*, *xlabel='x'*, *ylabel='y'*, *zlabel='z'*, *title='Title'*, *xo=None*, *yo=None*, *xunit=''*, *yunit=''*, *do_fwhm=True*, *arg4main=None*, *arg4top=None*, *arg4side=None*)
Plot contourf in the main graph plus profiles over vertical and horizontal lines defined with mouse.

Parameters

- **xmatrix, ymatrix** (*ndarray*) – *x* and *y* matrix coordinates generated with `numpy.meshgrid()`
- **zmatrix** (*ndarray*) – Matrix with the data. Note that `xmatrix`, `ymatrix` and `zmatrix` must have the same shape
- **xlabel, ylabel, zlabel** (*str, optional*) – Labels for the axes `x`, `y` and `z`.
- **title** (*str, optional*) – title for the main graph #BUG: sometimes this title disappear
- **xo, yo** (*float, optional*) – if equal to `None`, it allows to use the mouse to choose the vertical and horizontal lines for the profile. If not `None`, the profiles lines are are centered at `(xo, yo)`
- **xunit, yunit** (*str, optional*) – String to be shown after the values in the small text box
- **do_fwhm** (*Boolean, optional*) – Calculate and print the FWHM in the figure. The script to calculate the FWHM is not very robust, it works well if only one well defined peak is present. Turn this off by setting this var to `False`
- ***arg4main** – **args* for the main graph
- ***arg4top** – **args* for the top graph
- ***arg4side** – **args* for the side graph

Returns

- **ax_main, ax_top, ax_side** (*matplotlib.axes*) – return the axes in case one wants to modify them.
- **delta_x, delta_y** (*float*)

Example

```
>>> import numpy as np
>>> import wavepy.utils as wpu
>>> xx, yy = np.meshgrid(np.linspace(-1, 1, 101), np.linspace(-1, 1, 101))
>>> wpu.plot_profile(xx, yy, np.exp(-(xx**2+yy**2)/.2))
```

Animation of the example above:

wavepy.utils.**select_file**(*pattern='*'*, *message_to_print=None*)

List files under the subdirectories of the current working directory, and expected the user to choose one of them.

The list of files is of the form `number: filename`. The user choose the file by typing the number of the desired filename.

Parameters

- **pattern** (*str*) – list only files with this patter. Similar to pattern in the linux comands ls, grep, etc
- **message_to_print** (*str, optional*)

Returns **filename** (*str*) – path and name of the file

Example

```
>>>  select_file('*.dat')
```

wavepy.utils.**select_dir**(*message_to_print=None*, *pattern='**/'*)

List subdirectories of the current working directory, and expected the user to choose one of them.

The list of files is of the form `number: filename`. The user choose the file by typing the number of the desired filename.

Similar to *wavepy.utils.select_file()*

Parameters **message_to_print** (*str, optional*)

Returns *str* – directory path

See also:

wavepy.utils.select_file()

wavepy.utils.**nan_mask_threshold**(*input_matrix*, *threshold=0.0*)

Calculate a square mask for array above OR below a threshold

Parameters

- **input_matrix** (*ndarray*) – 2 dimensional (or n-dimensional?) numpy.array to be masked
- **threshold** (*float*) – threshold for masking. If real (imaginary) value, values below(above) the threshold are set to NAN

Returns *ndarray* – array with values either equal to 1 or NAN.

Example

To use as a mask for array use:

```
>>> mask = nan_mask_threshold(input_array, threshold)
>>> masked_array = input_array*mask
```

Notes

- Note that `array[mask]` will return only the values where `mask == 1`.
- Also note that this is NOT the same as the masked arrays in numpy.

`wavepy.utils.`**`crop_matrix_at_indexes`**(*input_matrix*, *list_of_indexes*)

Alias for `np.copy(inputMatrix[i_min:i_max, j_min:j_max])`

Parameters

- **input_matrix** (*ndarray*) – 2 dimensional array
- **list_of_indexes** (*list*) – list in the format `[i_min, i_max, j_min, j_max]`

Returns *ndarray* – copy of the sub-region `inputMatrix[i_min:i_max, j_min:j_max]` of the inputMatrix.

Warning: Note the difference of copy and view in Numpy.

`wavepy.utils.`**`crop_graphic`**(*xvec=None*, *yvec=None*, *zmatrix=None*, *verbose=False*, *kargs4graph={}*)

Function to crop an image to the ROI selected using the mouse.

`wavepy.utils.graphical_roi_idx()` is first used to plot and select the ROI. The function then returns the croped version of the matrix, the cropped coordinate vectors `x` and `y`, and the indexes `[i_min, i_max, j_min,_j_max]`

Parameters

- **xvec, yvec** (*1D ndarray*) – vector with the coordinates `x` and `y`. See below how the returned variables change dependnding whether these vectors are provided.
- **zmatrix** (*2D numpy array*) – image to be croped, as an 2D ndarray
- ****kargs4graph** – kargs for main graph

Returns

- *1D ndarray, 1D ndarray* – cropped coordinate vectors `x` and `y`. These two vectors are only returned the input vectors `xvec` and `xvec` are provided
- *2D ndarray* – cropped image
- *list* – indexes of the crop `[i_min, i_max, j_min,_j_max]`. Useful when the same crop must be applies to other images

Examples

```
>>> import numpy as np
>>> import matplotlib as plt
>>> xVec = np.arange(0.,101)
>>> yVec = np.arange(0.,101)
>>> img = dummy_images('Shapes', size=(101,101), FWHM_x = .5, FWHM_y=1.0)
>>> (imgCroped, idx4crop) = crop_graphic(zmatrix=img)
>>> plt.imshow(imgCroped, cmap='Spectral')
>>> (xVecCroped,
>>>  yVecCroped,
>>>  imgCroped, idx4crop) = crop_graphic(xVec, yVec, img)
>>> plt.imshow(imgCroped, cmap='Spectral',
>>>            extent=np.array([xVecCroped[0], xVecCroped[-1],
>>>                             yVecCroped[0], yVecCroped[-1]])
```

See also:

wavepy.utils.crop_graphic_image() wavepy.utils.graphical_roi_idx()

wavepy.utils.**crop_graphic_image**(*image*, *verbose=False*, ***kargs4graph*)

Similar to *wavepy.utils.crop_graphic()*, but only for the main matrix (and not for the x and y vectors). The function then returns the croped version of the image and the indexes `[i_min, i_max, j_min, _j_max]`

Parameters

- **zmatrix** (*2D numpy array*) – image to be croped, as an 2D ndarray
- ****kargs4graph** – kargs for main graph

Returns

- *2D ndarray* – cropped image
- *list* – indexes of the crop `[i_min, i_max, j_min,_j_max]`. Useful when the same crop must be applies to other images

See also:

wavepy.utils.crop_graphic() wavepy.utils.graphical_roi_idx()

wavepy.utils.**crop_graphic_image**(*image*, *verbose=False*, ***kargs4graph*)

Similar to *wavepy.utils.crop_graphic()*, but only for the main matrix (and not for the x and y vectors). The function then returns the croped version of the image and the indexes `[i_min, i_max, j_min, _j_max]`

Parameters

- **zmatrix** (*2D numpy array*) – image to be croped, as an 2D ndarray
- ****kargs4graph** – kargs for main graph

Returns

- *2D ndarray* – cropped image
- *list* – indexes of the crop `[i_min, i_max, j_min,_j_max]`. Useful when the same crop must be applies to other images

See also:

wavepy.utils.crop_graphic() wavepy.utils.graphical_roi_idx()

wavepy.utils.**graphical_select_point_idx**(*zmatrix*, *verbose=False*, *kargs4graph={}*)

Plot a 2D array and allow to pick a point in the image. Returns the last selected position x and y of the choosen point

Parameters

- **zmatrix** (*2D numpy array*) – main image
- **verbose** (*Boolean*) – verbose mode
- ****kargs4graph** – kargs for main graph

Returns *int, int* – two integers with the point indexes `x` and `y`

Example

```
>>> jo, io = graphical_select_point_idx(array2D)
>>> value = array2D[io, jo]
```

See also:

wavepy.utils.graphical_roi_idx()

wavepy.utils.**find_nearest_value**(*input_array*, *value*)

Alias for `input_array.flatten()[np.argmin(np.abs(input_array.flatten() - value))]`

In a array of float numbers, due to the precision, it is impossible to find exact values. For instance something like `array1[array2==0.0]` might fail because the zero values in the float array `array2` are actually something like 0.0004324235 (fictious value).

This function will return the value in the array that is the nearest to the parameter `value`.

Parameters

- **input_array** (*ndarray*)
- **value** (*float*)

Returns *ndarray*

Example

```
>>> foo = dummy_images('NormalDist')
>>> find_nearest_value(foo, 0.5000)
0.50003537554879007
```

See also:

`wavepy:utils:find_nearest_value_index()`

wavepy.utils.**find_nearest_value_index**(*input_array*, *value*)

Similar to *wavepy.utils.find_nearest_value()*, but returns the index of the nearest value (instead of the value itself)

Parameters

- **input_array** (*ndarray*)
- **value** (*float*)

Returns *tuple of ndarray* – each array have the index of the nearest value in each dimension

---

Note: In principle it has no limit of the number of dimensions.

Example

```
>>> foo = dummy_images('NormalDist')
>>> find_nearest_value(foo, 0.5000)
0.50003537554879007
>>> (i_index, j_index) = find_nearest_value_index(foo, 0.500)
>>> foo[i_index[:], j_index[:]]
array([ 0.50003538,  0.50003538,  0.50003538,  0.50003538])
```

See also:

`wavepy:utils:find_nearest_value()`

`wavepy.utils.`**`dummy_images`**(*imagetype=None*, *shape=(100, 100)*, ***kwargs*)

Dummy images for simple tests.

Parameters

- **imagetype** (*str*) – See options Below
- **shape** (*tuple*) – Shape of the image. Similar to `numpy.shape`
- **kwargs** – keyword arguments depending on the image type.

Image types

•Noise (default): alias for `np.random.random(shape)`

•Stripes: `kwargs:  nLinesH, nLinesV`

•SumOfHarmonics: image is defined by: .. math:: sum_{ij} Amp_{ij} cos (2 pi i y) cos (2 pi j x).

–Note that `x` and `y` are assumed to in the range [-1, 1]. The keyword `kwargs:  harmAmpl` is a 2D list that can be used to set the values for Amp_ij, see **Examples**.

•Shapes: see **Examples**. `kwargs=noise`, amplitude of noise to be added to the image

•NormalDist: Normal distribution where it is assumed that `x` and `y` are in the interval *[-1,1]*. `keywords: FWHM_x, FWHM_y`

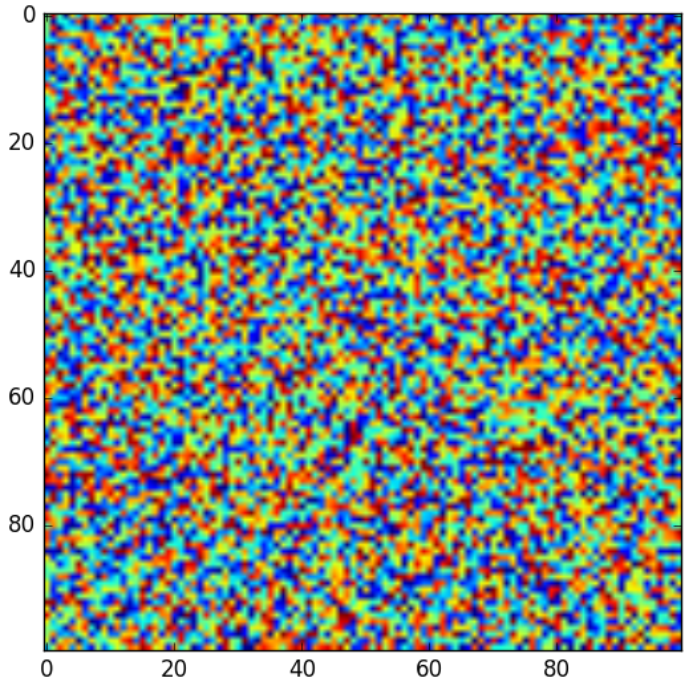
Returns *2D ndarray*

Examples

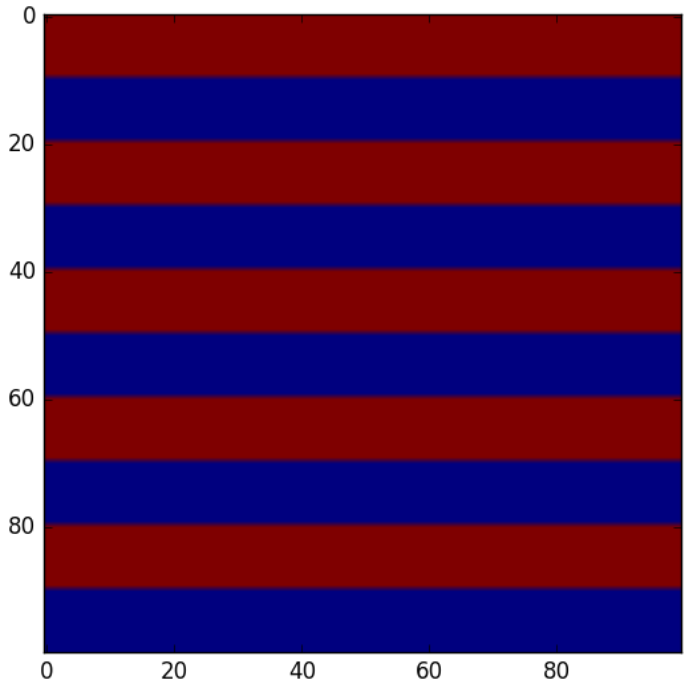
```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(dummy_images())
```

is the same than

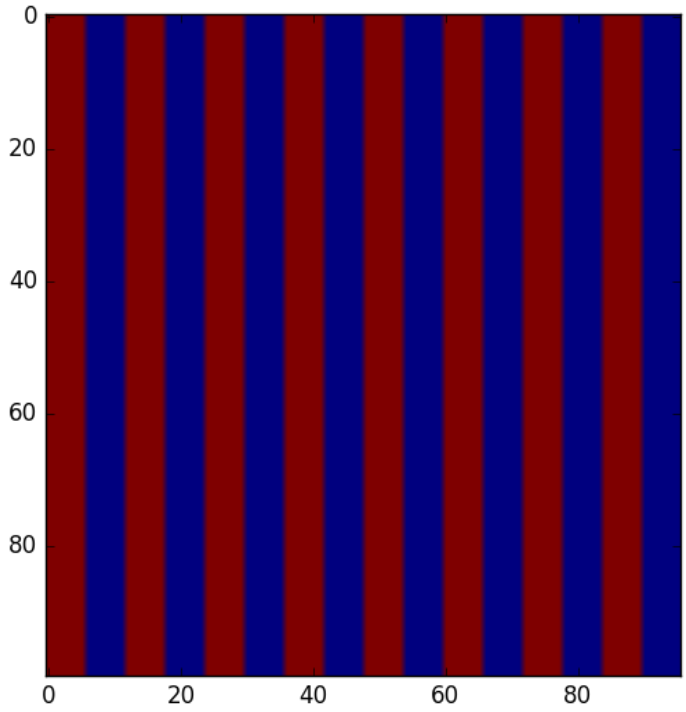
```
>>> plt.imshow(dummy_images('Noise'))
```



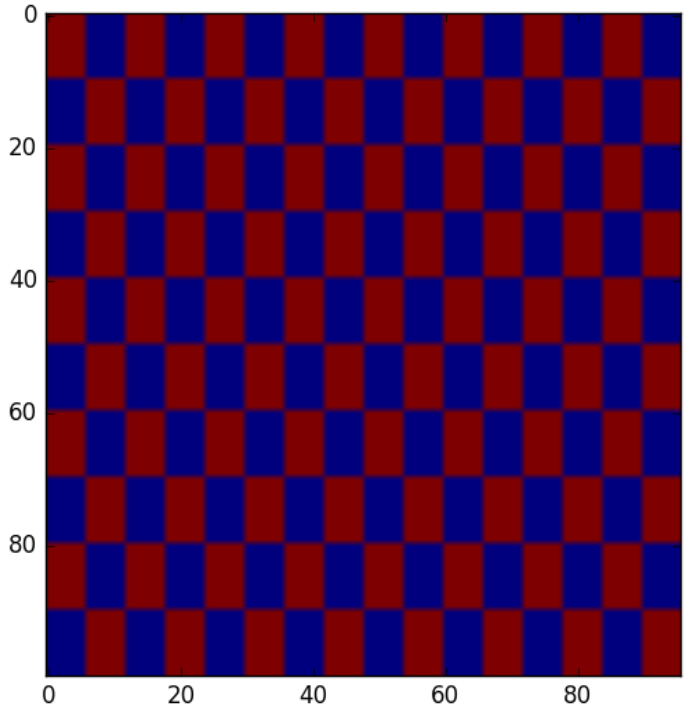

```
>>> plt.imshow(dummy_images('Stripes', nLinesV=5))
```



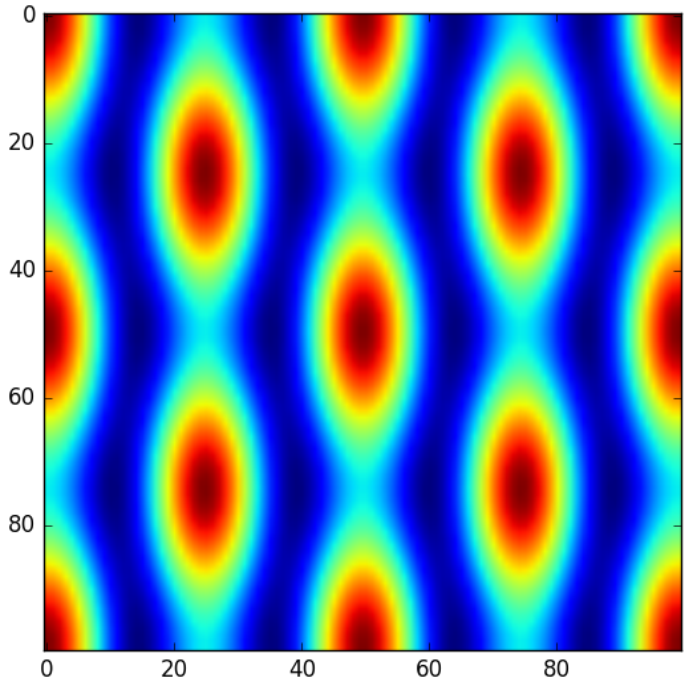

```
>>> plt.imshow(dummy_images('Stripes', nLinesH=8))
```



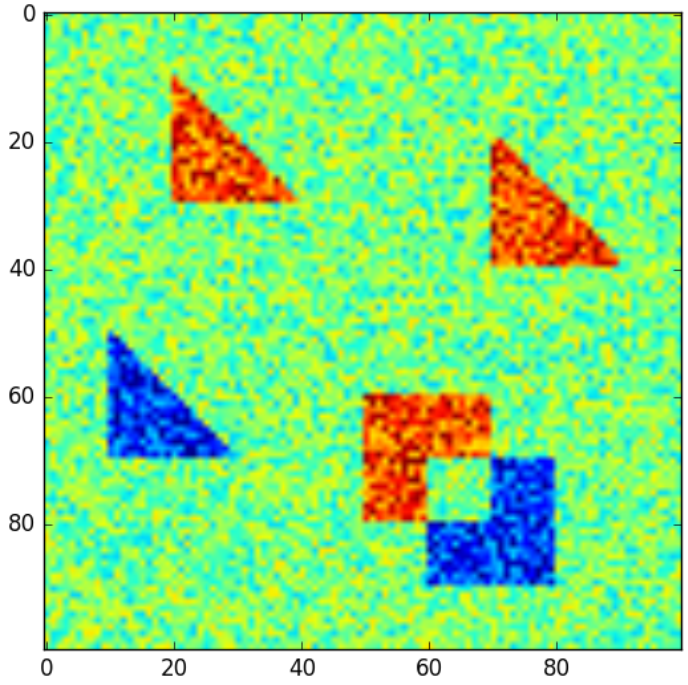

```
>>> plt.imshow(dummy_images('Checked', nLinesH=8, nLinesV=5))
```



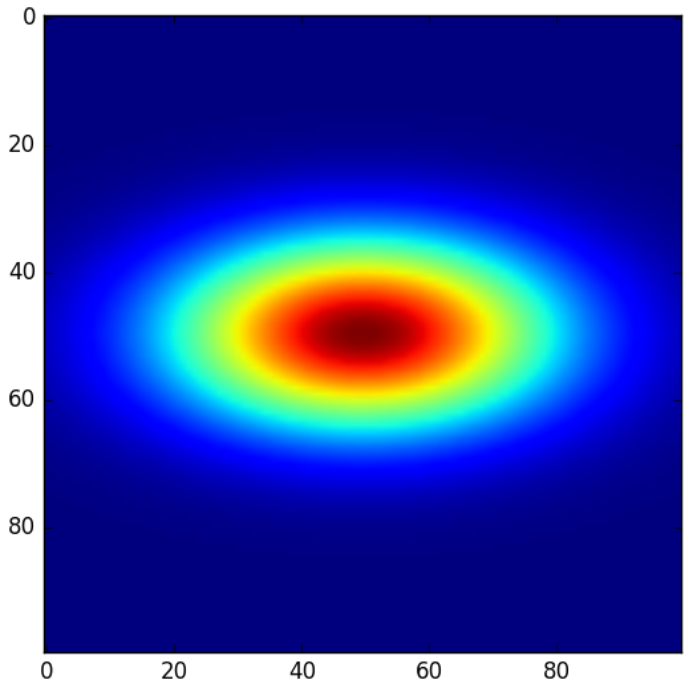

```
>>> plt.imshow(dummy_images('SumOfHarmonics', harmAmpl=[[1,0,1],[0,1,0]]))
```




```
>>> plt.imshow(dummy_images('Shapes', noise = 1))
```




```
>>> plt.imshow(dummy_images('NormalDist', FWHM_x = .5, FWHM_y=1.0))
```



wavepy.utils.**graphical_roi_idx**(*zmatrix*, *verbose=False*, *kargs4graph={}*)

Function to define a rectangular region of interest (ROI) in an image.

The image is plotted and, using the mouse, the user select the region of interest (ROI). The ROI is ploted as an transparent rectangular region. When the image is closed the function returns the indexes `[i_min, i_max, j_min, j_max]` of the ROI.

Parameters

- **input_array** (*ndarray*)
- **verbose** (*Boolean*) – In the verbose mode it is printed some additional infomations, like the ROI indexes, as the user select different ROI's
- ****kargs4graph** – Options for the main graph. **WARNING:** not tested very well

Returns *list* – indexes of the ROI `[i_min, i_max, j_min,_j_max]`. Useful when the same crop must be applies to other images

---

Note: In principle it has no limit of the number of dimensions.

Example

See example at `wavepy:utils:crop_graphic()`

See also:

`wavepy:utils:crop_graphic()`

wavepy.utils.**crop_graphic**(*xvec=None*, *yvec=None*, *zmatrix=None*, *verbose=False*, *kargs4graph={}*)

Function to crop an image to the ROI selected using the mouse.

wavepy.utils.graphical_roi_idx() is first used to plot and select the ROI. The function then returns the croped version of the matrix, the cropped coordinate vectors `x` and `y`, and the indexes `[i_min, i_max, j_min,_j_max]`

Parameters

- **xvec, yvec** (*1D ndarray*) – vector with the coordinates `x` and `y`. See below how the returned variables change dependnding whether these vectors are provided.
- **zmatrix** (*2D numpy array*) – image to be croped, as an 2D ndarray
- ****kargs4graph** – kargs for main graph

Returns

- *1D ndarray, 1D ndarray* – cropped coordinate vectors `x` and `y`. These two vectors are only returned the input vectors `xvec` and `xvec` are provided
- *2D ndarray* – cropped image
- *list* – indexes of the crop `[i_min, i_max, j_min,_j_max]`. Useful when the same crop must be applies to other images

Examples

```
>>> import numpy as np
>>> import matplotlib as plt
>>> xVec = np.arange(0.,101)
>>> yVec = np.arange(0.,101)
>>> img = dummy_images('Shapes', size=(101,101), FWHM_x = .5, FWHM_y=1.0)
>>> (imgCroped, idx4crop) = crop_graphic(zmatrix=img)
>>> plt.imshow(imgCroped, cmap='Spectral')
>>> (xVecCroped,
>>>  yVecCroped,
>>>  imgCroped, idx4crop) = crop_graphic(xVec, yVec, img)
>>> plt.imshow(imgCroped, cmap='Spectral',
>>>            extent=np.array([xVecCroped[0], xVecCroped[-1],
>>>                             yVecCroped[0], yVecCroped[-1]])
```

See also:

wavepy.utils.crop_graphic_image() *wavepy.utils.graphical_roi_idx()*

`wavepy.utils.`**`choose_unit`**`(`*`array`*`)`

Script to choose good(best) units in engineering notation for a `ndarray`.

For a given input array, the function returns `factor` and `unit` according to

$$10^n < \max(array) < 10^{n+3}$$

n	factor (float)	unit(str)
0	1.0	`''` empty string
-9	10^-9	`n`
-6	10^-6	`r'\mu'`
-3	10^-3	`m`
+3	10^-6	`k`
+6	10^-9	`M`
+9	10^-6	`G`

n=-6 returns \mu since this is the latex syntax for micro. See Example.

Parameters **array** (*ndarray*) – array from where to choose proper unit.

Returns *float, unit* – Multiplication Factor and strig for unit

Example

```
>>> array1 = np.linspace(0,100e-6,101)
>>> array2 = array1*1e10
>>> factor1, unit1 = choose_unit(array1)
>>> factor2, unit2 = choose_unit(array2)
>>> plt.plot(array1*factor1,array2*factor2)
>>> plt.xlabel(r'${0} m$'.format(unit1))
>>> plt.ylabel(r'${0} m$'.format(unit2))
```

The syntax r'$ string $ ' is necessary to use latex commands in the matplotlib labels.

wavepy.utils.**datetime_now_str**()
Returns the current date and time as a string in the format YYmmDD_HHMMSS. Alias for time.strftime("%Y%m%d_%H%M%S").

Returns *str*

wavepy.utils.**time_now_str**()
Returns the current time as a string in the format HHMMSS. Alias for time.strftime("%H%M%S").

Returns *str*

wavepy.utils.**date_now_str**()
Returns the current date as a string in the format YYmmDD. Alias for time.strftime("%Y%m%d").

Returns *str*

wavepy.utils.**realcoordvec**(*npoints*, *delta*)
Build a vector with real space coordinates based on the number of points and bin (pixels) size.

Alias for np.mgrid[-npoints/2*delta:npoints/2*delta-delta:npoints*1j]

Parameters

- **npoints** (*int*) – number of points
- **delta** (*float*) – vector with the values of x

Returns *ndarray* – vector (1D array) with real coordinates

Example

```
>>> realcoordvec(10,1)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

See also:

wavepy.utils.realcoordmatrix_fromvec() wavepy.utils.realcoordmatrix()

wavepy.utils.**realcoordmatrix_fromvec**(*xvec*, *yvec*)
Alias for np.meshgrid(xvec, yvec)

Parameters **xvec, yvec** (*ndarray*) – vector (1D array) with real coordinates

Returns *ndarray* – 2 matrices (1D array) with real coordinates

Example

```
>>> vecx = realcoordvec(3,1)
>>> vecy = realcoordvec(4,1)
>>> realcoordmatrix_fromvec(vecx, vecy)
[array([[-1.5, -0.5,  0.5],
[-1.5, -0.5,  0.5],
[-1.5, -0.5,  0.5],
[-1.5, -0.5,  0.5]]),
array([[-2., -2., -2.],
[-1., -1., -1.],
[ 0.,  0.,  0.],
[ 1.,  1.,  1.]])]
```

See also:

wavepy.utils.realcoordvec() wavepy.utils.realcoordmatrix()

wavepy.utils.**realcoordmatrix**(*npointsx*, *deltax*, *npointsy*, *deltay*)

Build a matrix (2D array) with real space coordinates based on the number of points and bin (pixels) size.

Alias for `realcoordmatrix_fromvec(realcoordvec(nx, delx), realcoordvec(ny, dely))`

Parameters

- **npointsx, npointsy** (*int*) – number of points in the x and y directions
- **deltax, deltay** (*float*) – step size in the x and y directions

Returns *ndarray, ndarray* – 2 matrices (2D array) with real coordinates

Example

```
>>> realcoordmatrix(3,1,4,1)
[array([[-1.5, -0.5,  0.5], [-1.5, -0.5,  0.5], [-1.5, -0.5,  0.5],
[-1.5, -0.5,  0.5]]), array([[-2., -2., -2.], [-1., -1., -1.],
[ 0.,  0.,  0.], [ 1.,  1.,  1.]])]
```

See also:

wavepy.utils.realcoordvec() wavepy.utils.realcoordmatrix_fromvec()

wavepy.utils.**reciprocalcoordvec**(*npoints*, *delta*)

Create coordinates in the (spatial) frequency domain based on the number of points `n` and the step (binning) `\Delta x` in the **REAL SPACE**. It returns a vector of frequencies with values in the interval

$$\left[\frac{-1}{2\Delta x} : \frac{1}{2\Delta x} - \frac{1}{n\Delta x}\right],$$

with the same number of points.

Parameters

- **npoints** (*int*) – number of points
- **delta** (*float*) – step size in the **REAL SPACE**

Returns *ndarray*

Example

```
>>> reciprocalcoordvec(10,1e-3)
array([-500., -400., -300., -200., -100., 0., 100., 200., 300., 400.])
```

See also:

wavepy.utils.realcoordvec() wavepy.utils.reciprocalcoordmatrix()

wavepy.utils.**reciprocalcoordmatrix**(*npointsx*, *deltax*, *npointsy*, *deltay*)

Similar to *wavepy.utils.reciprocalcoordvec()*, but for matrices (2D arrays).

Parameters

- **npointsx, npointsy** (*int*) – number of points in the x and y directions
- **deltax, deltay** (*float*) – step size in the x and y directions

Returns *ndarray, ndarray* – 2 matrices (2D array) with coordinates in the frequencies domain

Example

```
>>> reciprocalcoordmatrix(5, 1e-3, 4, 1e-3)
[array([[-500., -300., -100.,  100.,  300.],
[-500., -300., -100.,  100.,  300.],
[-500., -300., -100.,  100.,  300.],
[-500., -300., -100.,  100.,  300.]]),
array([[-500., -500., -500., -500., -500.],
[-250., -250., -250., -250., -250.],
[   0.,    0.,    0.,    0.,    0.],
[ 250.,  250.,  250.,  250.,  250.]])]
```

See also:

wavepy.utils.realcoordmatrix() wavepy.utils.reciprocalcoordvec()

wavepy.utils.**h5_list_of_groups**(*h5file*)

Get the names of all groups and subgroups in a hdf5 file.

Parameters **h5file** (*h5py file*)

Returns *list* – list of strings with group names

Example

```
>>> fh5 = h5py.File(filename,'r')
>>> listOfGoups = h5_list_of_groups(fh5)
>>> for group in listOfGoups: print(group)
```

wavepy.utils.**progress_bar4pmap**(*res*, *sleep_time=1.0*)

Progress bar from `tqdm` to be used with the function `multiprocessing.starmap_async()`.

It holds the program in a loop waiting `multiprocessing.starmap_async()` to finish

Parameters

- **res** (result object of the `multiprocessing.Pool` class)
- **sleep_time**

Example

```
>>> from multiprocessing import Pool
>>> p = Pool()
>>> res = p.starmap_async(...)  # use your function inside brackets
>>> p.close()  # No more work
>>> progress_bar4pmap(res)
```

wavepy.utils.**load_ini_file**(*inifname*)

Parameters **inifname** (*str*) – name of the `*.ini` file.

Returns *configparser objects*

Example

Example of `ini` file:

```
[Files]
image_filename = file1.tif
ref_filename = file2.tif

[Parameters]
par1 = 10.5e-5
par2 = 10, 100, 500, 600
par can have long name = 25
par3 = the value can be anything
```

If we create a file named `.temp.ini` with the example above, we can load it as:

```
>>> config = load_ini_file('.temp.ini')
>>> print(config['Parameters']['Par1'] )
```

See also:

`wavepy.utils.load_ini_file_terminal_dialog()`, `wavepy.utils.get_from_ini_file()`, `wavepy.utils.set_at_ini_file()`.

wavepy.utils.**rocking_3d_figure**(*ax*, *outfname='out.ogv'*, *elevAmp=50*, *azimAmpl=90*, *elevOffset=0*, *azimOffset=0*, *dpi=50*, *npoints=200*, *del_tmp_imgs=True*)

Saves an image at different view angles and join the images to make a short animation. The movement is defined by setting the elevation and azimut angles following a sine function. The frequency of the vertical movement (elevation) is twice of the horizontal (azimute), forming a figure eight movement.

Parameters

- **ax** (*3D axis object*) – See example below how to create this object. If *None*, this will use the temporary images from a previous run
- **outfname** (*str*) – output file name. Note that the extension defines the file format. This function has been tested for the formats *.gif* (not recomended, big files and poor quality), *.mp4*, *.mkv* and *.ogv*. For LibreOffice, *.ogv* is the recomend format.

- **elevAmp** (*float*) – amplitude of elevation movement, in degrees
- **azimAmpl** (*float*) – amplitude of azimutal movement, in degrees. If negative, the image will continually rotate around the azimute direction (no azimute rocking)
- **elevOffset** (*float*) – offset of elevation movement, in degrees
- **azimOffset** (*float*) – offset of azimutal movement, in degrees
- **dpi** (*float*) – resolution of the individual images
- **npoints** (*int*) – number of intermediary images to form the animation. More images will make the the movement slower and the animation longer.
- **remove_images** (*float*) – the program creates *npoints* temporary images, and this flag defines if these images are deleted or not

Example

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> xx = np.random.rand(npoints)*2-1
>>> yy = np.random.rand(npoints)*2-1
>>> zz = np.sinc(-(xx**2/.5**2+yy**2/1**2))
>>> ax.plot_trisurf(xx, yy, zz, cmap='viridis', linewidth=0.2, alpha = 0.8)
>>> plt.show()
>>> plt.pause(.5)
>>> # this pause is necessary to plot the first image in the main screen
>>> rocking_3d_figure(ax, 'out_080.ogv',
                      elevAmp=45, azimAmpl=45,
                      elevOffset=0, azimOffset=45, dpi=80)
>>> # example of use of the del_tmp_imgs flag
>>> rocking_3d_figure(ax, 'out_050.ogv',
                      elevAmp=60, azimAmpl=60,
                      elevOffset=10, azimOffset=45,
                      dpi=50, del_tmp_imgs=False)
>>> rocking_3d_figure(None, 'out2_050.gif',
                      del_tmp_imgs=True)
```

wavepy.speckletracking

Functions for speckle tracking analises

Functions:

speckleDisplacement(image, image_ref[, ...])	This function track the movements of speckle in an image (with sample) related to a reference image (whith no sample).

wavepy.speckletracking.**speckleDisplacement**(*image*, *image_ref*, *stride=1*, *npointsmax=None*, *halfsubwidth=10*, *halfTemplateSize=None*, *subpixelResolution=None*, *ncores=0.5*, *taskPerCore=100*, *verbose=False*)

This function track the movements of speckle in an image (with sample) related to a reference image (whith no sample). The function relies in two other functions that you are advised to check: register_translation and see match_template

References

see http://scikit-image.org/docs/dev/auto_examples/transform/plot_register_translation.html

see http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_template.html

wavepy.surface_from_grad

Surface from gradient data

In many x-ray imaging techniques we obtain the differential phase of the wavefront in two directions. This is the same as to say that we measured the gradient of the phase. Therefore to obtain the phase we need a method to integrate the differential data. Matematically we want to obtain the surface $s(x, y)$ from the (experimental) differential curves $s_x = s_x(x, y) = \frac{\partial s(x,y)}{\partial x}$ and $s_y = s_y(x, y) = \frac{\partial s(x,y)}{\partial y}$.

This is not as straight forward as it looks. The main reason is that, to be able to calculate the function from its gradient, the partial derivatives must be integrable. A function is integrable when the two cross partial derivative of the function are equal, that is

$$\frac{\partial^2 s(x,y)}{\partial x \partial y} = \frac{\partial^2 s(x,y)}{\partial y \partial x}$$

However, due to experimental errors and noises, there are no guarantees that the data is integrable (very likely they are not).

To obtain a signal/image from differential information is a broader topic with application in others topic of science. Few methods have been developed in different context, in special computer vision where this problem is refered as "Surface Reconstruction from Gradient Fields". For consistense, we will (try to) stick to this same term.

These methods try to find the best signal $s(x, y)$ that best describes the differential curves. For this reason, it is advised to use some kind of check for the integration, for instance by calculating the gradient from the result and comparing with the original gradient. It is better if this is done in the current library. See for instance the use of `wavepy.error_integration()` in the function `wavepy.surdace_from_grad.frankotchellappa()` below.

It is the goal for this library to add few different methods, since it is clear that different methods have different strenghts and weakness (precision, processing time, memory requirements, etc).

References

Frankot, R. T., & Chellappa, R. (1988). A method for enforcing integrability in shape from shading algorithms.

Agrawal, A. et al (2006). What Is the Range of Surface Reconstructions from a Gradient Field?.

Harker, M. et al (2008). Least squares surface reconstruction from measured gradient fields.

Sevcenco, I. S. et al (2015). A wavelet based method for image reconstruction from gradient data with applications.

Harker, M. et al (2015). MATLAB toolbox for the regularized surface reconstruction from gradients.

Huang, L .et al (2015). Comparison of two-dimensional integration methods for shape reconstruction from gradient data.

Functions:

frankotchellappa(del_f_del_x, del_f_del_y[, ...])	The simplest method is the so-called Frankot-Chelappa method.

wavepy.surface_from_grad.**frankotchellappa**(*del_f_del_x*, *del_f_del_y*, *reflec_pad=True*)

The simplest method is the so-called Frankot-Chelappa method. The idea behind this method is to search (calculate) an integrable gradient field that best fits the data. Luckly, Frankot Chelappa were able in they article to find a simple single (non interective) equation for that. We are even luckier since this equation makes use of FFT, which is very computationally efficient.

Considering a signal $s(x, y)$ with differential signal given by $s_x = s_x(x, y) = \frac{\partial s(x,y)}{\partial x}$ and $s_y = s_y(x, y) = \frac{\partial s(x,y)}{\partial y}$. The Fourier Transform of s_x and s_y are given by

$$\mathcal{F}[s_x] = \mathcal{F}[s_x](f_x, f_y) = \mathcal{F}\left[\frac{\partial s(x,y)}{\partial x}\right](f_x, f_y), \quad \mathcal{F}[s_y] = \mathcal{F}[s_y](f_x, f_y) = \mathcal{F}\left[\frac{\partial s(x,y)}{\partial y}\right](f_x, f_y).$$

Finally, Frankot-Chellappa method is base in solving the following equation:

$$\mathcal{F}[s] = \frac{-if_x\mathcal{F}[s_x] - if_y\mathcal{F}[s_y]}{2\pi(f_x^2 + f_y^2)}$$

where

$$\mathcal{F}[s] = \mathcal{F}[s(x, y)](f_x, f_y)$$

is the Fourier Transform of s.

To avoid the singularity in the denominator, it is added `numpy.finfo(float).eps()`, the smallest float number in the machine.

Keep in mind that Frankot-Chelappa is not the best method. More advanced alghorithms are available, where it is used more complex math and interactive methods. Unfortunatelly, these algorothims are only available for MATLAB.

Parameters

- **del_f_del_x, del_f_del_y** (*ndarrays*) – 2 dimensional gradient data
- **reflec_pad** (*bool*) – This flag pad the gradient field in order to obtain a 2-dimensional reflected function. See more in the Notes below.

Returns *ndarray* – Integrated data, as provided by the Frankt-Chellappa Algorithm. Note that the result are complex numbers. See below

Notes

•Padding

Frankt-Chellappa makes intensive use of the Discrete Fourier Transform (DFT), and due to the periodicity property of the DFT, the result of the integration will also be periodic (even though we only get one period of the answer). This property can result in a discontinuity at the edges, and Frankt-Chellappa method is badly affected by discontinuity,

In this sense the idea of this padding is that by reflecting the function at the edges we avoid discontinuity. This was inspired by the code of the function DfGBox, available in the MATLAB File Exchange website.

Note that, since we only have the gradient data, we need to consider how a reflection at the edges will affect the partial derivatives. We show it below without proof (but it is easy to see).

First lets consider the data for the x direction derivative $\Delta_x = \dfrac{\partial f}{\partial x}$ consisting of a 2D array of size $N \times M$. The padded matrix will be given by:

$$\begin{bmatrix} \Delta_x(x,y) & -\Delta_x(N-x,y) \\ \Delta_x(x,M-y) & -\Delta_x(N-x,M-y) \end{bmatrix}$$

and for the for the y direction derivative $\Delta_y = \dfrac{\partial f}{\partial y}$ we have

$$\begin{bmatrix} \Delta_y(x,y) & \Delta_y(N-x,y) \\ -\Delta_y(x,M-y) & -\Delta_y(N-x,M-y) \end{bmatrix}$$

Note that this padding increases the number of points from $N \times M$ to $2M \times 2N$. However, **the function only returns the $N \times M$ result**, since the other parts are only a repetion of the result. In other words, the padding is done only internally.

•Results are Complex Numbers

Again due to the use of DFT's, the results are complex numbers. In principle an ideal gradient field of real numbers results a real-only result. This "imaginary noise" is observerd even with theoretical functions, which leads to the conclusion that it is due to a numerical noise. It is left to the user to decide what to do with noise, for instance to use the modulus or the real part of the result. But it is recomended to use the real part.

wavepy.surface_from_grad.**error_integration**(*del_f_del_x*, *del_f_del_y*, *func*, *pixel-size*, *errors=False*, *shifthalfpixel=False*, *plot_flag=True*)

Examples

Here we describe what the examples are doing.

Example Speckle Analyses

The analyses of the speckle tracking data is split in three steps:

- **Pre-processing:** Consist of loading and cropping the raw image(s) and doing basic operations like normalization, filtering and subtraction of background.
- **Data analyses *per se*:** Here the physics of the method is used to convert the pre processed data to some meaninfull physical information. It will likely require experimental information like pixel size, photon energy and distance sample to detector. It should not require any information about the sample.

- **Post Processing:** Perform additional steps to the physical data, like integration, mask, filtering and unwrap. At this point some information about the sample may be required. It must produce graphics and results to be presented to others.

The two files below present examples of speckle tracking data analyses. The first is `speckleAnalyses.py` and it perfrom basic pre processing (interactive crop), data analyses, and save the result in `hfd5` format.

Then `speckleAnalyses_postProcessing.py` loads the results obtained with `speckleAnalyses.py`, and perform operations like undersampling, mask, integration and extra calculations to obtain the final desired result, in this case thisckness of the sample. It also plot the results in a meaninful manner.

Download file with pre-precessing and data analyses: `speckleAnalyses.py`

Download file with post processing and plot of results: `speckleAnalyses_postProcessing.py`

Code for Speckle Tracking Analyses.py

This section contains the speckleAnalyses script.

Download file: `speckleAnalyses.py`

Speckle Analyses Post Processing

This section contains the speckleAnalyses_postProcessing script.

Download file: `speckleAnalyses_postProcessing.py`

Credits

Citations

We kindly request that you cite the following article *[A1]* if you use wavePy.

References

Bibliography

[A1] Shashidhara Marathe, Lahsen Assoufid, Xianghui Xiao, Kyungmin Ham, Warren W. Johnson, and Leslie G. Butler. Improved algorithm for processing grating-based phase contrast interferometry image sets. *Review of Scientific Instruments*, 85(1):, 2014. URL: http://scitation.aip.org/content/aip/journal/rsi/85/1/10.1063/1.4861199, doi:http://dx.doi.org/10.1063/1.4861199.

[B1] F. Bayer, S. Zabler, C. Brendel, G. Pelzer, J. Rieger, A. Ritter, T. Weber, T. Michel, and G. Anton. "projection angle dependence in grating-based x-ray dark-field imaging of ordered structures". *Optics Express*, 21(17):19922–19933, 2013.

[B2] B. Betz, R. P. Harti, M. Strobl, J. Hovind, A. Kaestner, E. Lehmann, H. Van Swygenhoven, and C. Gruenzweig. "quantification of the sensitivity range in neutron dark-field imaging". *Review of Scientific Instruments*, 2015.

[B3] E. Eggl, S. Schleede, M. Bech, K. Achterhold, R. Loewen, R. D. Ruth, and F. Pfeiffer. "x-ray phase-contrast tomography with a compact laser-driven synchrotron source". *Proceedings of the National Academy of Sciences of the United States of America*, 112(18):5567–5572, 2015.

[B4] S. M. Gruner, M. W. Tate, and E. F. Eikenberry. "charge-coupled device area x-ray detectors". *Review of Scientific Instruments*, 73(8):2815–2842, 2002.

[B5] C. Grünzweig, F. Pfeiffer, O. Bunk, T. Donath, G. Kühne, G. Frei, M. Dierolf, and C. David. Design, fabrication, and characterization of diffraction gratings for neutron phase contrast imaging. *Review of Scientific Instruments*, 79(5):, 2008. URL: http://scitation.aip.org/content/aip/journal/rsi/79/5/10.1063/1.2930866, doi:http://dx.doi.org/10.1063/1.2930866.

[B6] T. Lauridsen, M. Willner, M. Bech, F. Pfeiffer, and R. Feidenhans'l. "detection of sub-pixel fractures in x-ray dark-field tomography". *Applied Physics a-Materials Science & Processing*, 121(3):1243–1250, 2015.

[B7] S. K. Lynch, V. Pai, J. Auxier, A. F. Stein, E. E. Bennett, C. K. Kemble, X. Xiao, W.-K. Lee, N. Y. Morgan, and H. H. Wen. "interpretation of dark-field contrast and particle-size selectivity in grating interferometers". *Applied Optics*, 50(22):4310–4319, 2011.

[B8] S. Marathe, L. Assoufid, X. Xiao, K. Ham, W. W. Johnson, and L. G. Butler. "improved algorithm for processing grating-based phase contrast interferometry image sets". *Review of Scientific Instruments*, 85(1):art. no. 013704, 2014.

[B9] S. Marathe, X. Xiao, M. J. Wojcik, R. Divan, L. G. Butler, K. Ham, K. Fezzaa, M. Erdmann, H. H. Wen, W.-K. Lee, A. T. Macrander, F. De Carlo, D. C. Mancini, and L. Assoufid. "development of grating-based x-ray talbot interferometry at the advanced photon source". *AIP Conference Proceedings*, 1466(1):249–254, 2012.

[B10] Shashidhara Marathe, Lahsen Assoufid, Xianghui Xiao, Kyungmin Ham, Warren W. Johnson, and Leslie G. Butler. Improved algorithm for processing grating-based phase contrast interferometry image sets. *Review of Scientific Instruments*, 85(1):, 2014. URL: http://scitation.aip.org/content/aip/journal/rsi/85/1/10.1063/1.4861199, doi:http://dx.doi.org/10.1063/1.4861199.

[B11] V. Revol, C. Kottler, R. Kaufmann, A. Neels, and A. Dommann. "orientation-selective x-ray dark field imaging of ordered systems". *Journal of Applied Physics*, 112(11):art. no. 114903, 2012.

[B12] M. Strobl. "general solution for quantitative dark-field contrast imaging with grating interferometers". *Scientific Reports*, 4:art. no. 7243, 2014.

[B13] A. Velroyen, A. Yaroshenko, D. Hahn, A. Fehringer, A. Tapfer, M. Muller, P. B. Noel, B. Pauwels, A. Sasov, A. O. Yildirim, O. Eickelberg, K. Hellbach, S. D. Auweter, F. G. Meinel, M. F. Reiser, M. Bech, and F. Pfeiffer. "grating-based x-ray dark-field computed tomography of living mice". *EBioMedicine*, 2(10):1500–6, 2015.

[B14] H. H. Wen, E. E. Bennett, R. Kopace, A. F. Stein, and V. Pai. "single-shot x-ray differential phase-contrast and diffraction imaging using two-dimensional transmission gratings". *Optics Letters*, 35(12):1932–1934, 2010.

[B15] M. Willner, G. Fior, M. Marschner, L. Birnbacher, J. Schock, C. Braun, A. A. Fingerle, P. B. Noel, E. J. Rummeny, F. Pfeiffer, and J. Herzen. "phase-contrast hounsfield units of fixated and non-fixated soft-tissue samples". *Plos One*, 10(8):art. no. e0137016, 2015.

Python Module Index

W

Index