OpenPIV Documentation

Release 1

The OpenPIV Contributors

February 03, 2014

CONTENTS

OpenPiv is a effort of scientists to deliver a tool for the analysis of PIV images using state-of-the-art algorithms. Openpiv is released under the GPL Licence, which means that the source code is freely available for users to study, copy, modify and improve. Because of its permissive licence, you are welcome to download and try OpenPiv for whatever need you may have. Furthermore, you are encouraged to contribute to OpenPiv, with code, suggestions and critics.

OpenPiv exists in three forms: Matlab, C++ and Python. This is the home page of the Python implementation.

CHAPTER

ONE

CONTENTS:

1.1 Download OpenPIV Example

1.1.1 Tutorial files

These are zip files containing sample images and python scripts for analysing them with OpenPIV. These files are included in the source code if cloned from the Git.

Part 1: how to process an image pair. `source code and sample images`

Part 2: how to process in batch a list of image pairs. `source code and sample images`

1.2 Installation instruction

1.2.1 Dependencies

OpenPIV would not have been possible if other great open source projects did not exist. We make extensive use of code and tools that other people have created, so you should install them before you can use OpenPIV.

The dependencies are:

- python
- scipy
- numpy
- cython

On all platforms, the binary Enthought Python Distribution (EPD) is recommended. Visit http://www.enthought.com

How to install the dependencies on Linux

On a Linux platform installing these dependencies should be trick. Often, if not always, python is installed by default, while the other dependencies should appear in your package manager.

How to install the dependencies on Windows

On Windows all these dependencies, as well as several other useful packages, can be installed using the Python(x,y) distribution, available at http://www.pythonxy.com/. Note: Install it in Custom Directories, without spaces in the directory names (i.e. Program Files are prohibited), e.g. C:Pythonxy

How to install the dependencies on a Mac

The binary (32 or 64 bit) Enthought Python Distribution (EPD) is recommended. Visit http://www.enthought.com. However, if you use EPD Free distribution, you need to install Cython from http://www.cython.org

1.2.2 Get OpenPIV source code!

At this moment the only way to get OpenPIV's source code is using git. Git Git is a distributed revision control system and our code is hosted at GitHub.

Bleeding edge development version

If you are interested in the source code you are welcome to browse out git repository stored at https://github.com/alexlib/openpiv-python. If you want to download the source code on your machine, for testing, you need to set up git on your computer. Please look at http://help.github.com/ which provide extensive help for how to set up git.

To follow the development of OpenPIV, clone our repository with the command:

```
git clone http://github.com/alexlib/openpiv-python.git
```

and update from time to time. You can also download a tarball containing everything.

Then add the path where the OpenPIV source are to the PYTHONPATH environment variable, so that OpenPIV module can be imported and used in your programs. Remeber to build the extension with

```
python setup.py build
```

1.2.3 Having problems?

If you encountered some issues, found difficult to install OpenPIV following these instructions please drop us an email to openpiv-develop@lists.sourceforge.net , so that we can help you and improve this page!

1.3 Information for developers and contributors

OpenPiv need developers to improve further. Your support, code and contribution is very welcome and we are grateful you can provide some. Please send us an email to openpiv-develop@lists.sourceforge.net to get started, or for any kind of information.

We use git for development version control, and we have a main repository on github.

1.3.1 Development workflow

This is absolutely not a comprehensive guide of git development, and it is only an indication of our workflow.

1. Download and install git. Instruction can be found here.
2. Set up a github account.
3. Clone OpenPiv repository using:

   ```
   git clone http://github.com/alexlib/openpiv-python.git
   ```

4. create a branch *new_feature* where you implement your new feature.
5. Fix, change, implement, document code, ...
6. From time to time fetch and merge your master branch with that of the main repository.
7. Be sure that everything is ok and works in your branch.
8. Merge your master branch with your *new_feature* branch.
9. Be sure that everything is now ok and works in you master branch.
10. Send a pull request.
11. Create another branch for a new feature.

1.3.2 Which language can i use?

As a general rule, we use Python where it does not make any difference with code speed. In those situations where Python speed is the bottleneck, we have some possibilities, depending on your skills and background. If something has to be written from scratch use the first language from the following which you are confortable with: cython, c, c++, fortran. If you have existing, debugged, tested code that you would like to share, then no problem. We accept it, whichever language may be written in!

1.3.3 Things OpenPiv currently needs, (in order of importance)

- The implementation of advanced processing algorithms
- Good documentations
- Flow field filtering and validation functions
- Cython wrappers for c/c++ codes.
- a good graphical user interface

1.4 Tutorial

This is a series of examples and tutorials which focuses on showing features and capabilities of OpenPIV, so that after reading you should be able to set up scripts for your own analyses. If you are looking for a complete reference to the OpenPiv api, please look at *API reference*. It is assumed that you have Openpiv installed on your system along with a working python environment as well as the necessary *OpenPiv dependencies*. For installation details on various platforms see *Installation instruction*.

In this tutorial we are going to use some example data provided with the source distribution of OpenPIV. Altough it is not necessary, you may find helpful to actually run the code examples as the tutorial progresses. If you downloaded a tarball file, you should find these examples under the directory openpiv/docs/examples. Similarly if you cloned the git repository. If you cannot find them, dowload example images as well as the python source code from the *downloads* page.

1.4.1 First example: how to process an image pair

The first example shows how to process a single image pair. This is a common task and may be useful if you are studying how does a certain algorithm behaves. We assume that the current working directory is where the two image of the first example are located. Here is the code:

```
import openpiv.tools
import openpiv.process
import openpiv.scaling

frame_a  = openpiv.tools.imread( 'exp1_001_a.bmp' )
frame_b  = openpiv.tools.imread( 'exp1_001_b.bmp' )

u, v, sig2noise = openpiv.process.extended_search_area_piv( frame_a, frame_b, window_size=24, overlap

x, y = openpiv.process.get_coordinates( image_size=frame_a.shape, window_size=24, overlap=12 )

u, v, mask = openpiv.validation.sig2noise_val( u, v, sig2noise, threshold = 1.3 )

u, v = openpiv.filters.replace_outliers( u, v, method='localmean', n_iter=10, kernel_size=2)

x, y, u, v = openpiv.scaling.uniform(x, y, u, v, scaling_factor = 96.52 )

openpiv.tools.save(x, y, u, v, 'exp1_001.txt' )
```

This code can be executed as a script, or you can type each command in an Ipython console with pylab mode set, so that you can visualize result as they are available. I will follow the second option and i will present the results of each command.

We first import some of the openpiv modules.:

```
import openpiv.tools
import openpiv.process
import openpiv.scaling
```

Module `openpiv.tools` contains mostly contains utilities and tools, such as file I/O and multiprocessingvfacilities. Module `openpiv.process` contains advanced algorithms for PIV analysis and several helper functions. Last, module `openpiv.scaling` contains functions for field scaling.

We then load the two image files into numpy arrays:

```
frame_a  = openpiv.tools.imread( 'exp1_001_a.bmp' )
frame_b  = openpiv.tools.imread( 'exp1_001_b.bmp' )
```

Inspecting the attributes of one of the two images we can see that:

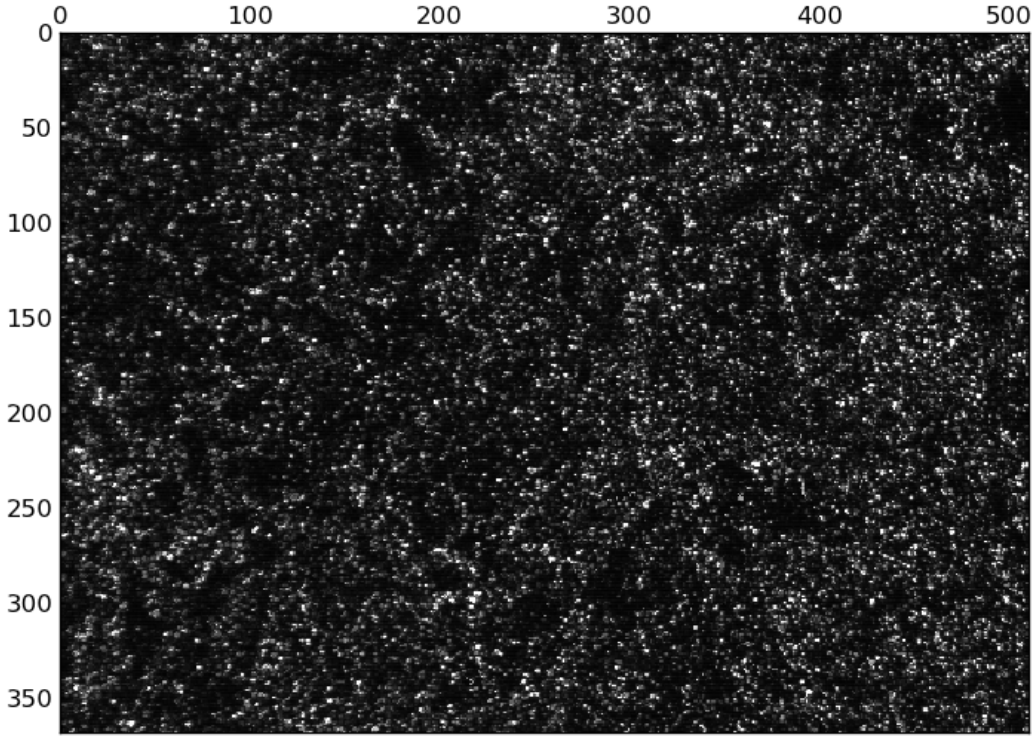
```
frame_a.shape
(369, 511)

frame_a.dtype
dtype('int32')
```

image has a size of 369x511 pixels and are contained in 32 bit integer arrays. Using pylab graphical capabilities it is easy to visualize one of the two frames::

```
matshow ( frame_a, cmap=cm.Greys _r )
```

which results in this figure.



In this example we are going to use the function `openpiv.process.extended_search_area_piv()` to process the image pair.:

```
u, v, sig2noise = openpiv.process.extended_search_area_piv( frame_a, frame_b, window_size=24, overlap
```

This method is a zero order displacement predictor cross-correlation algorithm, which cope with the problem of loss of pairs when the interrogation window is small, by increasing the search area on the second image. We also provide some options to the function, namely the `window_size`, i.e. the size of the interrogation window on `frame_a`, the `overlap` in pixels between adjacent windows, the time delay in seconds `dt` between the two image frames an te size in pixels of the extended search area on `frame_b`. `sig2noise_method` specifies which method to use for the evalutaion of the signal/noise ratio. The function also returns a third array, `sig2noise` which contains the signal to noise ratio obtained from each cross-correlation function, intended as the ratio between the heigth of the first and second peaks.

We then compute the coordinates of the centers of the interrogation windows using `openpiv.process.get_coordinates()`.:

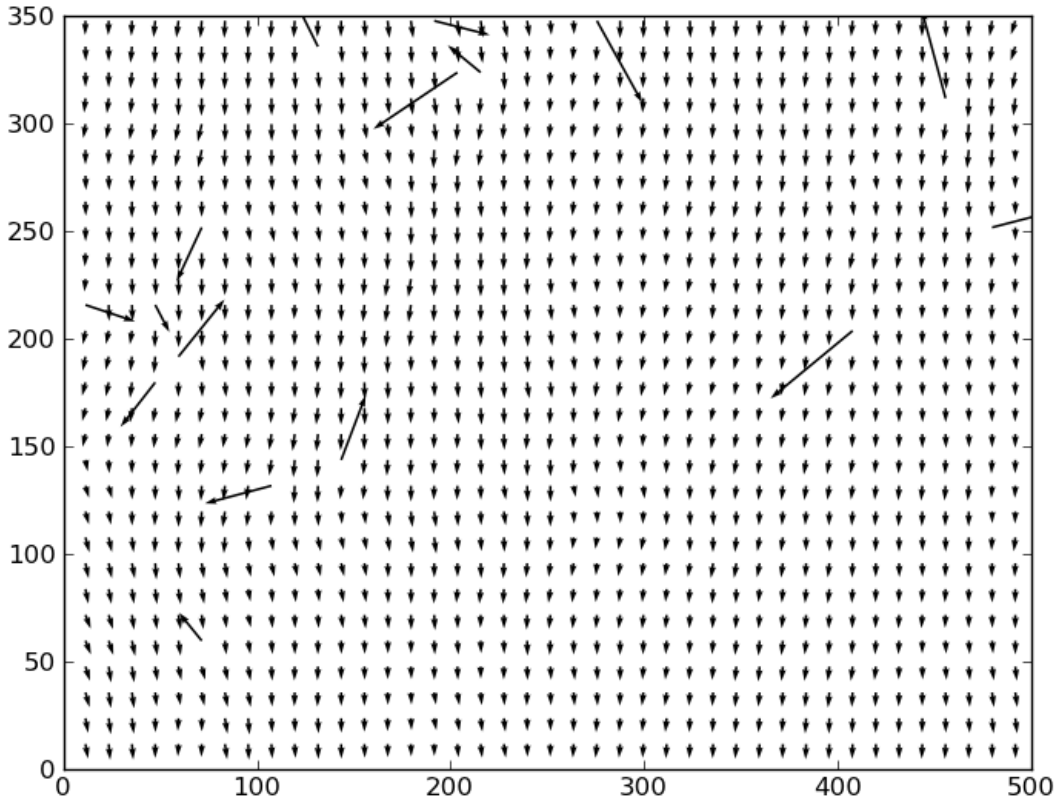
```
x, y = openpiv.process.get_coordinates( image_size=frame_a.shape, window_size=48, overlap=32 )
```

Note that we have provided some the same options we have given in the previous command to the processing function.

We can now plot the vector plot on a new figure to inspect the result of the analysis, using:

```
close()
quiver( x, y, u, v )
```

and we obtain:

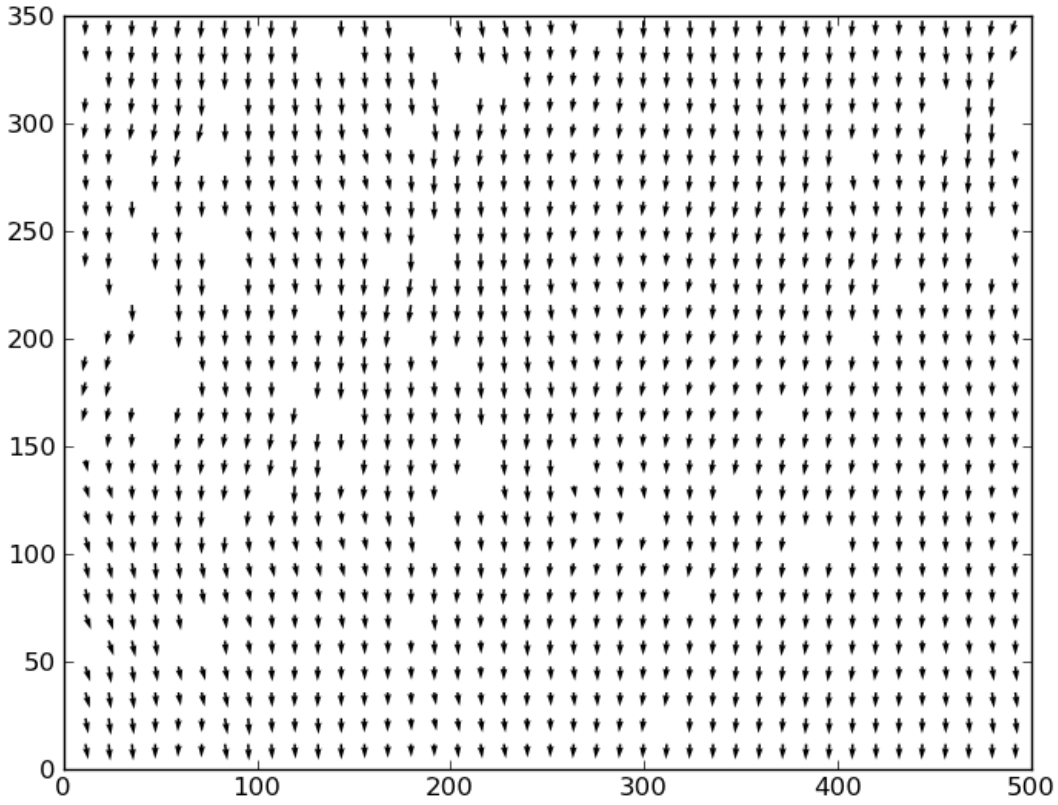


Several outliers vectors can be observed as a result of the small interrogation window size and we need to apply a validation scheme. Since we have information about the signal to noise ratio of the cross-correlation function we can apply a well know filtering scheme, classifing a vector as an outlier if its signal to noise ratio exceeds a certain threshold. To accomplish this task we use the function:

```
u, v, mask = openpiv.validation.sig2noise_val( u, v, sig2noise, threshold = 1.3 )
```

with a threshold value set to `1.3`. This function actually sets to NaN all those vector for which the signal to noise ratio is below 1.3. Therefore, the arrays `u` and `v` contains some np.nan elements. Furthermore, we get in output a third variable `mask`, which is a boolean array where elements corresponding to invalid vectors have been replace by Nan. The result of the filtering is shown in the following image, which we obtain with the two commands:

```
figure()
quiver( x, y, u, v )
```

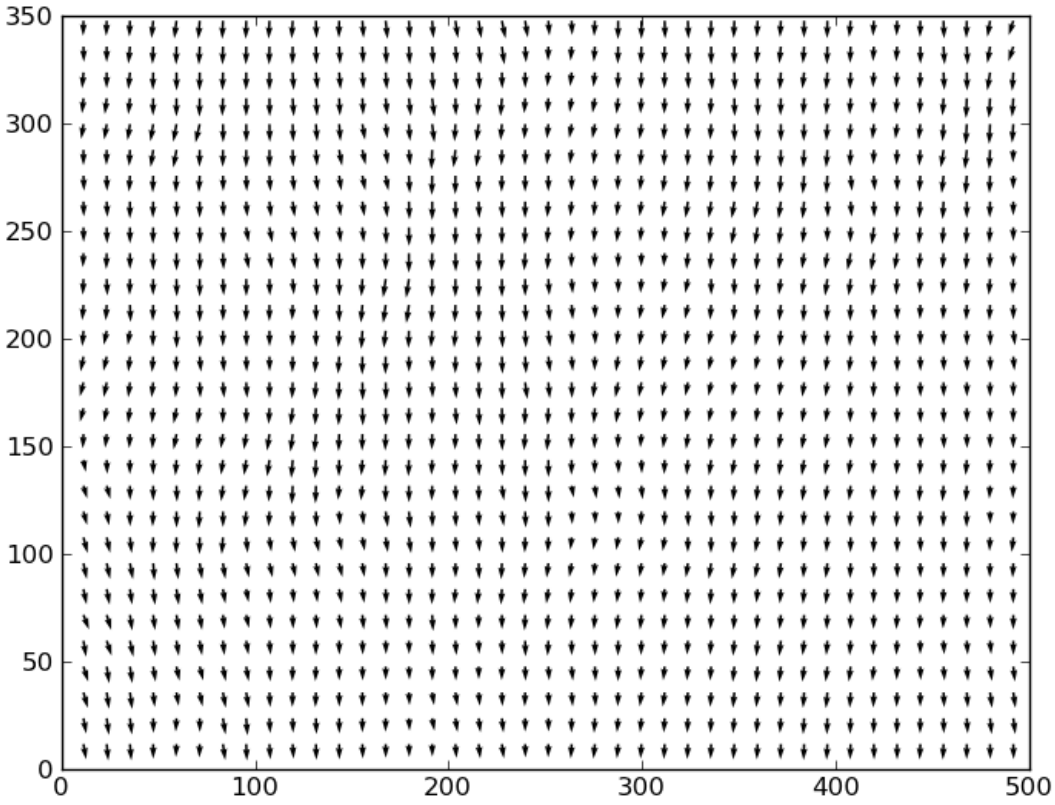


The final step is to replace the missing vector. This is done which the function `openpiv.filters.replace_outliers()`, which implements an iterative image inpainting algorithm with a specified kernel. We pass to this function the two velocity components arrays, a method type `localmean`, the number of passes and the size of the kernel.:

```
u, v = openpiv.filters.replace_outliers( u, v, method='localmean', n_iter=10, kernel_size=2 )
```

The flow field now appears much more smooth and the outlier vectors have been correctly replaced.

```
figure()
quiver( x, y, u, v )
```



The last step is to apply an uniform scaling to the flow field to get dimensional units. We use the function `openpiv.scaling.uniform()` providing the `scaling_factor` value, in pixels per meters if we want position and velocities in meters and meters/seconds or in pixels per millimeters if we want positions and velocities in millimeters and millimeters/seconds, respectively.

```
x, y, u, v = openpiv.scaling.uniform(x, y, u, v, scaling_factor = 96.52 )
```

Finally we save the data to an ascii file, for later processing, using::

```
openpiv.tools.save(x, y, u, v, 'exp1_001.txt')
```

1.4.2 Second example: how to process in batch a list of image pairs.

It if often the case, where several hundreds of image pairs have been sampled in an experiment and have to be processed. For these tasks it is easier to launch the analysis in batch and process all the image pairs with the same processing parameters. OpenPiv, with its powerful python scripting capabilities, provides a convenient way to accomplish this task and offers multiprocessing facilities for machines which have multiple cores, to speed up the computation. Since the analysis is an embarassingly parallel problem, the speed up that can be reached is quite high and almost equal to the number of core your machine has.

Compared to the previous example we have to setup some more things in the python script we will use for the batch processing.

Let's first import the needed modules.:

```
import openpiv.tools
import openpiv.scaling
import openpiv.process
```

We then define a python function which will be excecuted for each image pair. In this function we can specify any operation to execute on each single image pair, but here, for clarity we will setup a basic analysis, without a validation/replacement step.

Here is an example of valid python function::

```
def func( args ):
    """A function to process each image pair."""

    # this line is REQUIRED for multiprocessing to work
    # always use it in your custom function

    file_a, file_b, counter = args


    ######################
    # Here goes you code
    ######################

    # read images into numpy arrays
    frame_a  = openpiv.tools.imread( file_a )
    frame_b  = openpiv.tools.imread( file_b )

    # process image pair with extended search area piv algorithm.
    u, v = openpiv.process.extended_search_area_piv( frame_a, frame_b, window_size=32, overlap=16, dt

    # get window centers coordinates
    x, y = openpiv.process.get_coordinates( image_size=frame_a.shape, window_size=32, overlap=16 )

    # save to a file
    openpiv.tools.save(x, y, u, v, 'exp1_%03d.txt' % counter, fmt='%8.7f', delimiter='\t' )
```

The function we have specified *must* accept in input a single argument. This argument is a three element tuple, which you have to unpack inside the function body as we have done with:

```
file_a, file_b, counter = args
```

The tuple contains the two filenames of the image pair and a counter, which is needed to remember which image pair we are currently processing, (basically just for the output filename). After that you have unpacked the tuple into its three elements, you can use them to load the images and do the rest.

The *simple* processing function we wrote is just half of the job. We still need to specify which image pairs to process and where they are located. Therefore, in the same script we add the following two lines of code.:

```
task = openpiv.tools.Multiprocesser( data_dir = '.', pattern_a='2image_*0.tif', pattern_b='2image_*1
task.run( func = func, n_cpus=8 )
```

where we have set datadir to `.` because the script and the images are in the same folder. The first line creates an instance of the `openpiv.tools.Multiprocesser()` class. This class is responsible of sharing the processing work to multiple processes, so that the analysis can be executed in parallell. To construct the class you have to pass it three arguments:

- `data_dir`: the directory where image files are located
- `pattern_a` and `pattern_b`: the patterns for matching image files for frames *a* and *b*.

Note: Variables `pattern_a` and `pattern_b` are shell globbing patterns. Let 's say we have thousands of files for frame *a* in a sequence like file0001-a.tif, file0002-a.tif, file0003-a.tif, file0004-a.tif, ..., and the same for frames *b* file0001-b.tif, file0002-b.tif, file0003-b.tif, file0004-b.tif. To match these files we would set `pattern_a = file*-a.tif` and `pattern_b = file*-a.tif`. Basically, the * is a wildcard to match 0001, 0002, 0003, ...

The second line actually launches the batch process, using for each image pair the `func` function we have provided. Note that we have set the `n_cpus` option to be equal to `8` just because my machine has eight cores. You should not set `n_cpus` higher than the number of core your machine has, because you would not get any speed up.

1.5 API reference

This is a complete api reference to the openpiv python module.

1.5.1 The `openpiv.tools` module

The openpiv.tools module is a collection of utilities and tools.

`imread`(filename[, flatten])	Read an image file into a numpy array
`save`(x, y, u, v, mask, filename[, fmt, ...])	Save flow field to an ascii file.
`display`(message)	Display a message to standard output.
`display_vector_field`(filename, **kw)	Displays quiver plot of the data stored in the file
`Multiprocesser`(data_dir, pattern_a, pattern_b)	

openpiv.tools.imread

static `tools.`**`imread`**(*filename*, *flatten=0*)

Read an image file into a numpy array using scipy.misc.imread

Parameters **filename** : string

the absolute path of the image file

flatten : bool

True if the image is RGB color or False (default) if greyscale

Returns **frame** : np.ndarray

a numpy array with grey levels

Examples

```
>>> image = openpiv.tools.imread( 'image.bmp' )
>>> print image.shape
    (1280, 1024)
```

openpiv.tools.save

static `tools.`**`save`**(*x*, *y*, *u*, *v*, *mask*, *filename*, *fmt='%8.4f'*, *delimiter='t'*)

Save flow field to an ascii file.

Parameters **x** : 2d np.ndarray

a two dimensional array containing the x coordinates of the interrogation window centers, in pixels.

y : 2d np.ndarray

a two dimensional array containing the y coordinates of the interrogation window centers, in pixels.

u : 2d np.ndarray

a two dimensional array containing the u velocity components, in pixels/seconds.

v : 2d np.ndarray

a two dimensional array containing the v velocity components, in pixels/seconds.

mask : 2d np.ndarray

a two dimensional boolen array where elements corresponding to invalid vectors are True.

filename : string

the path of the file where to save the flow field

fmt : string

a format string. See documentation of numpy.savetxt for more details.

delimiter : string

character separating columns

Examples

```
>>> openpiv.tools.save( x, y, u, v, 'field_001.txt', fmt='%6.3f', delimiter='       ')
```

openpiv.tools.display

static `tools.`**`display`**(*message*)

Display a message to standard output.

Parameters **message** : string

a message to be printed

openpiv.tools.display_vector_field

`openpiv.tools.`**`display_vector_field`**(*filename*, ***kw*)

Displays quiver plot of the data stored in the file

Parameters **filename** : string

the absolute path of the text file

Key arguments : (additional parameters, optional)

scale: [None | float] *width*: [None | float]

Examples

```
>>> openpiv.tools.display_vector_field('./exp1_0000.txt',scale=100, width=0.0025)
```

openpiv.tools.Multiprocesser

class openpiv.tools.**Multiprocesser**(*data_dir*, *pattern_a*, *pattern_b*)

Methods

run(func[, n_cpus])	Start to process images.

__init__(*data_dir*, *pattern_a*, *pattern_b*)

A class to handle and process large sets of images.

This class is responsible of loading image datasets and processing them. It has parallelization facilities to speed up the computation on multicore machines.

It currently support only image pair obtained from conventional double pulse piv acquisition. Support for continuos time resolved piv acquistion is in the future.

Parameters **data_dir** : str

the path where image files are located

pattern_a : str

a shell glob patter to match the first frames.

pattern_b : str

a shell glob patter to match the second frames.

Examples

```
>>> multi = openpiv.tools.Multiprocesser( '/home/user/images', 'image_*_a.bmp', 'image_*_b.b
```

Methods

__init__(data_dir, pattern_a, pattern_b)	A class to handle and process large sets of images.
run(func[, n_cpus])	Start to process images.

1.5.2 The openpiv.pyprocess module

This module contains a pure python implementation of the basic cross-correlation algorithm for PIV image processing.

normalize_intensity(window)	Normalize interrogation window by removing the mean value.
correlate_windows(window_a, window_b[, ...])	Compute correlation function between two interrogation windows.
get_coordinates(image_size, window_size, overlap)	Compute the x, y coordinates of the centers of the interrogation windows

Continued on next page

Table 1.4 – continued from previous page

`get_field_shape`(image_size, window_size, overlap)	Compute the shape of the resulting flow field.
`moving_window_array`(array, window_size, overlap)	This is a nice numpy trick.
`find_first_peak`(corr)	Find row and column indices of the first correlation peak.
`find_second_peak`(corr[, i, j, width])	Find the value of the second largest peak.
`find_subpixel_peak_position`(corr[, ...])	Find subpixel approximation of the correlation peak.
`piv`(frame_a, frame_b[, window_size, ...])	Standard PIV cross-correlation algorithm.

openpiv.pyprocess.normalize_intensity

`openpiv.pyprocess.`**`normalize_intensity`**(*window*)

Normalize interrogation window by removing the mean value.

Parameters **window** : 2d np.ndarray

the interrogation window array

Returns **window** : 2d np.ndarray

the interrogation window array, with mean value equal to zero.

openpiv.pyprocess.correlate_windows

static `pyprocess.`**`correlate_windows`**(*window_a*, *window_b*, *corr_method='fft'*, *nfftx=None*, *nffty=None*)

Compute correlation function between two interrogation windows.

The correlation function can be computed by using the correlation theorem to speed up the computation.

Parameters **window_a** : 2d np.ndarray

a two dimensions array for the first interrogation window.

window_b : 2d np.ndarray

a two dimensions array for the second interrogation window.

corr_method : string

one of the two methods currently implemented: 'fft' or 'direct'. Default is 'fft', which is much faster.

nfftx : int

the size of the 2D FFT in x-direction, [default: 2 x windows_a.shape[0] is recommended].

nffty : int

the size of the 2D FFT in y-direction, [default: 2 x windows_a.shape[1] is recommended].

Returns **corr** : 2d np.ndarray

a two dimensions array for the correlation function.

openpiv.pyprocess.get_coordinates

static `pyprocess.`**`get_coordinates`**(*image_size*, *window_size*, *overlap*)

Compute the x, y coordinates of the centers of the interrogation windows.

Parameters **image_size: two elements tuple** :

a two dimensional tuple for the pixel size of the image first element is number of rows, second element is the number of columns.

window_size: int :

the size of the interrogation windows.

overlap: int :

the number of pixel by which two adjacent interrogation windows overlap.

Returns **x** : 2d np.ndarray

a two dimensional array containing the x coordinates of the interrogation window centers, in pixels.

y : 2d np.ndarray

a two dimensional array containing the y coordinates of the interrogation window centers, in pixels.

openpiv.pyprocess.get_field_shape

static pyprocess.**get_field_shape**(*image_size*, *window_size*, *overlap*)

Compute the shape of the resulting flow field.

Given the image size, the interrogation window size and the overlap size, it is possible to calculate the number of rows and columns of the resulting flow field.

Parameters **image_size: two elements tuple** :

a two dimensional tuple for the pixel size of the image first element is number of rows, second element is the number of columns.

window_size: int :

the size of the interrogation window.

overlap: int :

the number of pixel by which two adjacent interrogation windows overlap.

Returns **field_shape** : two elements tuple

the shape of the resulting flow field

openpiv.pyprocess.moving_window_array

static pyprocess.**moving_window_array**(*array*, *window_size*, *overlap*)

This is a nice numpy trick. The concept of numpy strides should be clear to understand this code.

Basically, we have a 2d array and we want to perform cross-correlation over the interrogation windows. An approach could be to loop over the array but loops are expensive in python. So we create from the array a new array with three dimension, of size (n_windows, window_size, window_size), in which each slice, (along the first axis) is an interrogation window.

openpiv.pyprocess.find_first_peak

static pyprocess.**find_first_peak**(*corr*)
Find row and column indices of the first correlation peak.

Parameters **corr** : np.ndarray

the correlation map

Returns **i** : int

the row index of the correlation peak

j : int

the column index of the correlation peak

corr_max1 : int

the value of the correlation peak

openpiv.pyprocess.find_second_peak

static pyprocess.**find_second_peak**(*corr*, *i=None*, *j=None*, *width=2*)
Find the value of the second largest peak.

The second largest peak is the height of the peak in the region outside a 3x3 submatrxi around the first correlation peak.

Parameters **corr: np.ndarray** :

the correlation map.

i,j : ints

row and column location of the first peak.

width : int

the half size of the region around the first correlation peak to ignore for finding the second peak.

Returns **i** : int

the row index of the second correlation peak.

j : int

the column index of the second correlation peak.

corr_max2 : int

the value of the second correlation peak.

openpiv.pyprocess.find_subpixel_peak_position

static pyprocess.**find_subpixel_peak_position**(*corr*, *subpixel_method='gaussian'*)
Find subpixel approximation of the correlation peak.

This function returns a subpixels approximation of the correlation peak by using one of the several methods available. If requested, the function also returns the signal to noise ratio level evaluated from the correlation map.

Parameters **corr** : np.ndarray

the correlation map.

subpixel_method : string

one of the following methods to estimate subpixel location of the peak: 'centroid' [replaces default if correlation map is negative], 'gaussian' [default if correlation map is positive], 'parabolic'.

Returns **subp_peak_position** : two elements tuple

the fractional row and column indices for the sub-pixel approximation of the correlation peak.

openpiv.pyprocess.piv

static pyprocess.**piv**(*frame_a*, *frame_b*, *window_size=64*, *overlap=32*, *dt=1.0*, *corr_method='fft'*, *subpixel_method='gaussian'*, *sig2noise_method=None*, *nfftx=None*, *nffty=None*, *width=2*)

Standard PIV cross-correlation algorithm.

This is a pure python implementation of the standard PIV cross-correlation algorithm. It is a zero order displacement predictor, and no iterative process is performed.

Parameters **frame_a** : 2d np.ndarray

an two dimensions array of integers containing grey levels of the first frame.

frame_b : 2d np.ndarray

an two dimensions array of integers containing grey levels of the second frame.

window_size : int

the size of the (square) interrogation window, [default: 32 pix].

overlap : int

the number of pixels by which two adjacent windows overlap [default: 16 pix].

dt : float

the time delay separating the two frames [default: 1.0].

corr_method : string

one of the two methods implemented: 'fft' or 'direct', [default: 'fft'].

subpixel_method : string

one of the following methods to estimate subpixel location of the peak: 'centroid' [replaces default if correlation map is negative], 'gaussian' [default if correlation map is positive], 'parabolic'.

sig2noise_method : string

defines the method of signal-to-noise-ratio measure, ('peak2peak' or 'peak2mean'. If None, no measure is performed.)

nfftx : int

the size of the 2D FFT in x-direction, [default: 2 x windows_a.shape[0] is recommended]

nffty : int

the size of the 2D FFT in y-direction, [default: 2 x windows_a.shape[1] is recommended]

width : int

the half size of the region around the first correlation peak to ignore for finding the second peak. [default: 2]. Only used if `sig2noise_method==peak2peak`.

Returns **u** : 2d np.ndarray

a two dimensional array containing the u velocity component, in pixels/seconds.

v : 2d np.ndarray

a two dimensional array containing the v velocity component, in pixels/seconds.

sig2noise : 2d np.ndarray, (optional: only if sig2noise_method is not None)

a two dimensional array the signal to noise ratio for each window pair.

1.5.3 The `openpiv.process` module

This module is dedicated to advanced algorithms for PIV image analysis.

`extended_search_area_piv`	The implementation of the one-step direct correlation with different size of the interrogation wi
`CorrelationFunction`	
`get_coordinates`	Compute the x, y coordinates of the centers of the interrogation windows.
`get_field_shape`	Compute the shape of the resulting flow field.
`correlate_windows`	Compute correlation function between two interrogation windows.
`normalize_intensity`	Normalize interrogation window by removing the mean value.

openpiv.process.extended_search_area_piv

`process.`**`extended_search_area_piv`**`()`

The implementation of the one-step direct correlation with different size of the interrogation window and the search area. The increased size of the search areas cope with the problem of loss of pairs due to in-plane motion, allowing for a smaller interrogation window size, without increasing the number of outlier vectors.

See:

Particle-Imaging Techniques for Experimental Fluid Mechanics

Annual Review of Fluid Mechanics Vol. 23: 261-304 (Volume publication date January 1991) DOI: 10.1146/annurev.fl.23.010191.001401

Parameters **frame_a** : 2d np.ndarray, dtype=np.int32

an two dimensions array of integers containing grey levels of the first frame.

frame_b : 2d np.ndarray, dtype=np.int32

an two dimensions array of integers containing grey levels of the second frame.

window_size : int

the size of the (square) interrogation window.

overlap : int

the number of pixels by which two adjacent windows overlap.

dt : float

the time delay separating the two frames.

search_area_size : int

the size of the (square) interrogation window from the second frame

subpixel_method : string

one of the following methods to estimate subpixel location of the peak: 'centroid' [replaces default if correlation map is negative], 'gaussian' [default if correlation map is positive], 'parabolic'.

sig2noise_method : string

defines the method of signal-to-noise-ratio measure, ('peak2peak' or 'peak2mean'. If None, no measure is performed.)

width : int

the half size of the region around the first correlation peak to ignore for finding the second peak. [default: 2]. Only used if `sig2noise_method==peak2peak`.

nfftx : int

the size of the 2D FFT in x-direction, [default: 2 x windows_a.shape[0] is recommended]

nffty : int

the size of the 2D FFT in y-direction, [default: 2 x windows_a.shape[1] is recommended]

Returns **u** : 2d np.ndarray

a two dimensional array containing the u velocity component, in pixels/seconds.

v : 2d np.ndarray

a two dimensional array containing the v velocity component, in pixels/seconds.

sig2noise : 2d np.ndarray, optional

a two dimensional array containing the signal to noise ratio from the cross correlation function. This array is returned if sig2noise_method is not None.

Examples

```
>>> u, v = openpiv.process.extended_search_area_piv( frame_a, frame_b, window_size=16, overlap=8
```

openpiv.process.CorrelationFunction

class `openpiv.process.`**`CorrelationFunction`**

Methods

`sig2noise_ratio`	Computes the signal to noise ratio.
`subpixel_peak_position`	Find subpixel approximation of the correlation peak.

__init__()

A class representing a cross correlation function.

Parameters **corr** : 2d np.ndarray

the correlation function array

Methods

`__init__`	A class representing a cross correlation function.
`sig2noise_ratio`	Computes the signal to noise ratio.
`subpixel_peak_position`	Find subpixel approximation of the correlation peak.

openpiv.process.get_coordinates

openpiv.process.**get_coordinates**()

Compute the x, y coordinates of the centers of the interrogation windows.

Parameters **image_size: two elements tuple** :

a two dimensional tuple for the pixel size of the image first element is number of rows, second element is the number of columns.

window_size: int :

the size of the interrogation windows.

overlap: int :

the number of pixel by which two adjacent interrogation windows overlap.

Returns **x** : 2d np.ndarray

a two dimensional array containing the x coordinates of the interrogation window centers, in pixels.

y : 2d np.ndarray

a two dimensional array containing the y coordinates of the interrogation window centers, in pixels.

openpiv.process.get_field_shape

openpiv.process.**get_field_shape**()

Compute the shape of the resulting flow field.

Given the image size, the interrogation window size and the overlap size, it is possible to calculate the number of rows and columns of the resulting flow field.

Parameters **image_size: two elements tuple** :

a two dimensional tuple for the pixel size of the image first element is number of rows, second element is the number of columns.

window_size: int :

the size of the interrogation window.

overlap: int :

the number of pixel by which two adjacent interrogation windows overlap.

Returns **field_shape** : two elements tuple

the shape of the resulting flow field

openpiv.process.correlate_windows

openpiv.process.**correlate_windows**()

Compute correlation function between two interrogation windows.

The correlation function can be computed by using the correlation theorem to speed up the computation.

Parameters **window_a** : 2d np.ndarray

a two dimensions array for the first interrogation window.

window_b : 2d np.ndarray

a two dimensions array for the second interrogation window.

corr_method : string

one of the two methods currently implemented: 'fft' or 'direct'. Default is 'fft', which is much faster.

nfftx : int

the size of the 2D FFT in x-direction, [default: 2 x windows_a.shape[0] is recommended].

nffty : int

the size of the 2D FFT in y-direction, [default: 2 x windows_a.shape[1] is recommended].

Returns **corr** : 2d np.ndarray

a two dimensions array for the correlation function.

openpiv.process.normalize_intensity

openpiv.process.**normalize_intensity**()

Normalize interrogation window by removing the mean value.

Parameters **window** : 2d np.ndarray

the interrogation window array

Returns **window** : 2d np.ndarray

the interrogation window array, with mean value equal to zero.

1.5.4 The openpiv.lib module

A module for various utilities and helper functions

sinc interp	Re-sample an image at intermediate positions between pixels.
replace_nans	Replace NaN elements in an array using an iterative image inpainting algorithm.

openpiv.lib.sincinterp

openpiv.lib.**sincinterp**()

Re-sample an image at intermediate positions between pixels.

This function uses a cardinal interpolation formula which limits the loss of information in the resampling process. It uses a limited number of neighbouring pixels.

The new image im^+ at fractional locations x and y is computed as:

$$im^+(x,y) = \sum_{i=-\texttt{kernel\_size}}^{i=\texttt{kernel\_size}} \sum_{j=-\texttt{kernel\_size}}^{j=\texttt{kernel\_size}} \texttt{image}(i,j) sin[\pi(i-\texttt{x})] sin[\pi(j-\texttt{y})] / \pi(i-\texttt{x}) / \pi(j-\texttt{y})$$

Parameters **image** : np.ndarray, dtype np.int32

the image array.

x : two dimensions np.ndarray of floats

an array containing fractional pixel row positions at which to interpolate the image

y : two dimensions np.ndarray of floats

an array containing fractional pixel column positions at which to interpolate the image

kernel_size : int

interpolation is performed over a `(2*kernel_size+1)*(2*kernel_size+1)` submatrix in the neighbourhood of each interpolation point.

Returns **im** : np.ndarray, dtype np.float64

the interpolated value of `image` at the points specified by `x` and `y`

openpiv.lib.replace_nans

openpiv.lib.**replace_nans**()

Replace NaN elements in an array using an iterative image inpainting algorithm.

The algorithm is the following:

1. For each element in the input array, replace it by a weighted average of the neighbouring elements which are not NaN themselves. The weights depends of the method type. If `method=localmean` weight are equal to 1/((2*kernel_size+1)**2 -1)
2. Several iterations are needed if there are adjacent NaN elements. If this is the case, information is "spread" from the edges of the missing regions iteratively, until the variation is below a certain threshold.

Parameters **array** : 2d np.ndarray

an array containing NaN elements that have to be replaced

max_iter : int

the number of iterations

kernel_size : int

the size of the kernel, default is 1

method : str

the method used to replace invalid values. Valid options are *localmean*.

Returns **filled** : 2d np.ndarray

a copy of the input array, where NaN elements have been replaced.

1.5.5 The `openpiv.filters` module

The openpiv.filters module contains some filtering/smoothing routines.

`gaussian`(u, v, size)	Smooths the velocity field with a Gaussian kernel.
`_gaussian_kernel`(size)	A normalized 2D Gaussian kernel array
`replace_outliers`(u, v[, method, max_iter, ...])	Replace invalid vectors in an velocity field using an iterative image inpainting

openpiv.filters.gaussian

static `filters.`**`gaussian`**(*u*, *v*, *size*)

Smooths the velocity field with a Gaussian kernel.

Parameters **u** : 2d np.ndarray

the u velocity component field

v : 2d np.ndarray

the v velocity component field

size : int

the half width of the kernel. Kernel has shape 2*size+1

Returns **uf** : 2d np.ndarray

the smoothed u velocity component field

vf : 2d np.ndarray

the smoothed v velocity component field

openpiv.filters._gaussian_kernel

static `filters.`**`_gaussian_kernel`**(*size*)

A normalized 2D Gaussian kernel array

Parameters **size** : int

the half width of the kernel. Kernel has shape 2*size+1

Examples

```
>>> from openpiv.filters import _gaussian_kernel
>>> _gaussian_kernel(1)
array([[ 0.04491922,  0.12210311,  0.04491922],
   [ 0.12210311,  0.33191066,  0.12210311],
   [ 0.04491922,  0.12210311,  0.04491922]])
```

openpiv.filters.replace_outliers

openpiv.filters.**replace_outliers**(*u*, *v*, *method='localmean'*, *max_iter=5*, *tol=0.001*, *kernel_size=1*)

Replace invalid vectors in an velocity field using an iterative image inpainting algorithm.

The algorithm is the following:

1. For each element in the arrays of the u and v components, replace it by a weighted average of the neighbouring elements which are not invalid themselves. The weights depends of the method type. If method=localmean weight are equal to 1/((2*kernel_size+1)**2 -1)
2. Several iterations are needed if there are adjacent invalid elements. If this is the case, inforation is "spread" from the edges of the missing regions iteratively, until the variation is below a certain threshold.

Parameters **u** : 2d np.ndarray

the u velocity component field

v : 2d np.ndarray

the v velocity component field

max_iter : int

the number of iterations

fil :

kernel_size : int

the size of the kernel, default is 1

method : str

the type of kernel used for repairing missing vectors

Returns **uf** : 2d np.ndarray

the smoothed u velocity component field, where invalid vectors have been replaced

vf : 2d np.ndarray

the smoothed v velocity component field, where invalid vectors have been replaced

1.5.6 The openpiv.validation module

A module for spurious vector detection.

global_val(u, v, u_thresholds, v_thresholds)	Eliminate spurious vectors with a global threshold.
sig2noise_val(u, v, sig2noise[, threshold])	Eliminate spurious vectors from cross-correlation signal to noise ratio.
global_std(u, v[, std_threshold])	Eliminate spurious vectors with a global threshold defined by the standard d
local_median_val(u, v, u_threshold, v_threshold)	Eliminate spurious vectors with a local median threshold.

openpiv.validation.global_val

openpiv.validation.**global_val**(*u*, *v*, *u_thresholds*, *v_thresholds*)

Eliminate spurious vectors with a global threshold.

This validation method tests for the spatial consistency of the data and outliers vector are replaced with Nan (Not a Number) if at least one of the two velocity components is out of a specified global range.

Parameters **u** : 2d np.ndarray

a two dimensional array containing the u velocity component.

v : 2d np.ndarray

a two dimensional array containing the v velocity component.

u_thresholds: two elements tuple :

u_thresholds = (u_min, u_max). If `u<u_min` or `u>u_max` the vector is treated as an outlier.

v_thresholds: two elements tuple :

`v_thresholds = (v_min, v_max)`. If `v<v_min` or `v>v_max` the vector is treated as an outlier.

Returns **u** : 2d np.ndarray

a two dimensional array containing the u velocity component, where spurious vectors have been replaced by NaN.

v : 2d np.ndarray

a two dimensional array containing the v velocity component, where spurious vectors have been replaced by NaN.

mask : boolean 2d np.ndarray

a boolean array. True elements corresponds to outliers.

openpiv.validation.sig2noise_val

`openpiv.validation.`**`sig2noise_val`**(*u*, *v*, *sig2noise*, *threshold=1.3*)

Eliminate spurious vectors from cross-correlation signal to noise ratio.

Replace spurious vectors with zero if signal to noise ratio is below a specified threshold.

Parameters **u** : 2d np.ndarray

a two dimensional array containing the u velocity component.

v : 2d np.ndarray

a two dimensional array containing the v velocity component.

sig2noise : 2d np.ndarray

a two dimensional array containing the value of the signal to noise ratio from cross-correlation function.

threshold: float :

the signal to noise ratio threshold value.

Returns **u** : 2d np.ndarray

a two dimensional array containing the u velocity component, where spurious vectors have been replaced by NaN.

v : 2d np.ndarray

a two dimensional array containing the v velocity component, where spurious vectors have been replaced by NaN.

mask : boolean 2d np.ndarray

a boolean array. True elements corresponds to outliers.

References

18. (a)Keane and R. J. Adrian, Measurement Science & Technology,1990, 1, 1202-1215.

openpiv.validation.global_std

openpiv.validation.**global_std**(*u*, *v*, *std_threshold=3*)

Eliminate spurious vectors with a global threshold defined by the standard deviation

This validation method tests for the spatial consistency of the data and outliers vector are replaced with NaN (Not a Number) if at least one of the two velocity components is out of a specified global range.

Parameters **u** : 2d np.ndarray

a two dimensional array containing the u velocity component.

v : 2d np.ndarray

a two dimensional array containing the v velocity component.

std_threshold: float :

If the length of the vector (actually the sum of squared components) is larger than std_threshold times standard deviation of the flow field, then the vector is treated as an outlier. [default = 3]

Returns **u** : 2d np.ndarray

a two dimensional array containing the u velocity component, where spurious vectors have been replaced by NaN.

v : 2d np.ndarray

a two dimensional array containing the v velocity component, where spurious vectors have been replaced by NaN.

mask : boolean 2d np.ndarray

a boolean array. True elements corresponds to outliers.

openpiv.validation.local_median_val

openpiv.validation.**local_median_val**(*u*, *v*, *u_threshold*, *v_threshold*, *size=1*)

Eliminate spurious vectors with a local median threshold.

This validation method tests for the spatial consistency of the data. Vectors are classified as outliers and replaced with Nan (Not a Number) if the absolute difference with the local median is greater than a user specified threshold. The median is computed for both velocity components.

Parameters **u** : 2d np.ndarray

a two dimensional array containing the u velocity component.

v : 2d np.ndarray

a two dimensional array containing the v velocity component.

u_threshold : float

the threshold value for component u

v_threshold : float

the threshold value for component v

Returns **u** : 2d np.ndarray

a two dimensional array containing the u velocity component, where spurious vectors have been replaced by NaN.

v : 2d np.ndarray

a two dimensional array containing the v velocity component, where spurious vectors have been replaced by NaN.

mask : boolean 2d np.ndarray

a boolean array. True elements corresponds to outliers.

1.5.7 The `openpiv.scaling` module

Scaling utilities

`uniform`(x, y, u, v, scaling_factor)	Apply an uniform scaling

openpiv.scaling.uniform

static `scaling.`**`uniform`**(*x*, *y*, *u*, *v*, *scaling_factor*)

Apply an uniform scaling

Parameters **x** : 2d np.ndarray

y : 2d np.ndarray

u : 2d np.ndarray

v : 2d np.ndarray

scaling_factor : float

the image scaling factor in pixels per meter

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

o

PYTHON MODULE INDEX

o