## Contents

1 Contents:
- 1.1 Installation instruction ............................................................... 3
- 1.2 Information for developers and contributors ........................................ 4
- 1.3 Tutorial ...................................................................................... 5
- 1.4 Download OpenPIV Example ......................................................... 12
- 1.5 API reference ............................................................................. 12
- 1.6 Frequently Asked Questions about PIV parameters ......................... 13

2 Indices and tables ............................................................................. 15
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.
1.1 Installation instruction

1.1.1 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 following Python distribution is recommended:

- Anaconda <https://store.continuum.io/cshop/anaconda/>

1.1.2 Installation

Use conda

```
conda install -c conda-forge openpiv
```

Or use pip

```
pip install numpy cython
pip install openpiv --pre
```
1.1.3 Get OpenPIV source code!

At this moment the only way to get OpenPIV’s source code is using 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:

```bash
$ git clone http://github.com/openpiv/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. Remember to build the extension with

```bash
python setup.py build_ext --inplace
```

1.1.4 Having problems?

If you encountered some issues, found difficult to install OpenPIV following these instructions please register and write to openpiv-users@googlegroups.com, so that we can help you and improve this page!

1.2 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@googlegroups.com to get started, or for any kind of information.

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

1.2.1 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:

```bash
$ 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 your master branch.

10) Send a pull request.

11) Create another branch for a new feature.

### 1.2.2 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 comfortable 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!

### 1.2.3 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

### 1.3 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. Although 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, download example images as well as the python source code from the downloads page.

#### 1.3.1 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:

```python
import openpiv.tools
import openpiv.process
import openpiv.scaling
import openpiv.validation
import openpiv.filters
```

(continues on next page)
We first import some of the openpiv modules:

```python
import openpiv.tools
import openpiv.process
import openpiv.scaling
import openpiv.validation
import openpiv.filters
```

Module `openpiv.tools` contains mostly contains utilities and tools, such as file I/O and multiprocessing facilities. 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:

```python
frame_a = openpiv.tools.imread( 'expl_001_a.bmp' )
frame_b = openpiv.tools.imread( 'expl_001_b.bmp' )
```

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

```python
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:

```python
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:

```python
u, v, sig2noise = openpiv.process.extended_search_area_piv( frame_a, frame_b, window_size=24, overlap=12, dt=0.02, search_area_size=64, sig2noise_method='peak2peak' )
```

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 and size in pixels of the extended search area on `frame_b`. `sig2noise_method` specifies which method to use for the evaluation 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 height of the first and second peaks.

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

```python
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:

1.3. Tutorial
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, classifying a vector as an outlier if its signal to noise ratio exceeds a certain threshold. To accomplish this task we use the function:

```python
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:

```python
figure()
quiver( x, y, u, v )
```
The final step is to replace the missing vector. This is done with 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.

\[
x, y, u, v = \text{openpiv.scaling.uniform}(x, y, u, v, \text{scaling_factor} = 96.52)
\]

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

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

### 1.3.2 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 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 embarrassingly 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:

```python
import openpiv.tools
import openpiv.scaling
import openpiv.process
import openpiv.validation
import openpiv.filters
```

We then define a python function which will be executed 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::

```python
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=0.02, search_area_size=64 )

    # 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='	' )
```

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:

```python
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::

```python
task = openpiv.tools.Multiprocesser( data_dir = '.', pattern_a='2image_*0.tif',
                        pattern_b='2image_*1.tif' )
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 parallel. 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.

### 1.4 Download OpenPIV Example

#### 1.4.1 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 see `openpiv/examples/examples/tutorials/tutorial1.py`

Part 2: how to process in batch a list of image pairs. `openpiv/examples/examples/tutorials/tutorial2.py`


### 1.5 API reference

This is a complete api reference to the opnepiv python module.
1.5.1 The openpiv.preprocess module

1.5.2 The openpiv.tools module

1.5.3 The openpiv.pyprocess module

1.5.4 The openpiv.process module

1.5.5 The openpiv.lib module

1.5.6 The openpiv.filters module

1.5.7 The openpiv.validation module

1.5.8 The openpiv.scaling module

1.6 Frequently Asked Questions about PIV parameters

1. Can you please elaborate on the \texttt{sclt} parameter which is passed to the openpiv function. E.g. if the time between the two consecutive image is 0.5 seconds and 1 pixel in the image corresponds to 50 cms, what would be the value of \texttt{sclt}.

\texttt{sclt} is a shortcut for \textit{scaling factor from displacement to velocity units}. It’s also called the \_scale\_, or \_scaling\_.

PIV provides the local displacement in pixel units. In order to know the displacement in the real physical units you multiply it by the scaling of \textit{cm/pixel}, i.e. by 50 cm/pixel. To know the speed, the displacement is divided by the time separation, i.e. by 0.5 seconds, then we get: \( \text{scaling} = \text{sclt} = 50 \text{ cm/pixels} / 0.5 = 100 \text{ [cm/seconds/pixels]} \)

For example, if the vector is 10 pixels, then the result will be \( 100 \times 10 = 1000 \text{ cm/s} \)

2. What’s the purpose of the local and global filtering?

\textbf{global filtering} supposingly removes the obvious \textbf{outliers}, i.e. the vectors which length is larger than the mean of the flow field plus 3 times its standard deviation. These are global outliers in the statistical sense.

\textbf{local filtering} is performed on small neighborhoods of vectors, e.g. 3 x 3 or 5 x 5, in order to find \textbf{local outliers} - the vectors that are dissimilar from the close neighbors. Typically there are about 5 per-cent of erroneous vectors and these are removed and later the missing values are interpolated from the neighbor vector values. This is also a reason for the Matlab version to generate three lists of files: \texttt{raw - noflt.txt filtered (after global and local filters) - flt.txt final (after filtering and interpolation) - .txt}

3. Why, while taking the FFT, we use the Nfft parameter?

\( \text{fft}_a = \text{fft2}(a2,Nfft,Nfft); \text{ fft}_b = \text{fft2}(b2,Nfft,Nfft); \)

and why the size has been specified as Nfft which is twice the interrogation window size.

In the FFT-based correlation analysis, we have to pad the window with zeros and get correlation map of the right size and avoid aliasing problem (see Raffel et al. 2007)

4. Also in the same function why sub image \texttt{b2} is rotated before taking the correlation. \( b2 = b2(\text{end}:-1:1,\text{end}:-1:1); \)

Without rotation the result will be convolution, not correlation. The definition is \( \text{ifft}(\text{fft(a)*fft(conj(b))}) \). \texttt{conj()} is replaced by rotation in the case of real values. It is more computationally efficient.

5. In the find_displacement(c,s2nm) function for finding peak2, why neighbourhood pixels around peak1 are removed? %line no:352
These peaks might appear as ‘false second peak’, but they are the part of the same peak. Think about a top of a mountain. You want to remove not only the single point, but cut out the top part in order to search for the second peak.

6. In the read_pair _of_images( ) function why A = double(A(:,:,1))/255; %line no:259 B = double(B(:,:,1))/255; 
In order to convert RGB to gray scale. Not always true.

7. After the program is executed, the variable vel contains all the parameters for all the velocity vectors. Here what are the units of u & v. Is it in metres/second?
It is not, the result depends on the SCLT variable. if it SCLT is 1, then it is in pixels/dt (dt is the interval between two images).

8. What is the “Outlier Filter Value” in OpenPIV?
The outlier filter value is the threshold of the global outlier filter and is says how many times the standard deviation of the whole vector field is exceeded before the vector is considered as outlier. See above discussion on the filters.

9. What is the fifth column in the Output data *_.txt,*flt.txt or *_noflt.txt?
The fifth column is the value of the Signal-To-Noise (s2n) ration. Note that the value is different (numerically) if the user choses Peak-to-Second-Peak ratio as the s2n parameter or Peak-to-Mean ratio as s2n parameter. The value of Peak-to-Second-Peak or Peak-to-Mean ratio is stored for the further processing.
CHAPTER 2

Indices and tables

- genindex
- modindex
- search