OpenPIV Documentation

Release 0.0.1

OpenPIV group

Jan 06, 2019

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 1

Contents:

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 2.7
- Scipy
- Numpy
- Cython

On all platforms, the following Python distributions arerecommended:

- Canopy <http://www.enthought.com>
- Anaconda <https://store.continuum.io/cshop/anaconda/>
- PythonXY <https://code.google.com/p/pythonxy/>
- WinPython <http://winpython.sourceforge.net/>

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

Missing package `progressbar`

Some distributions lack *progressbar* package. Install it separately using *pip*

> pip install progressbar

1.1.2 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_ext --inplace
```

Module import on Windows 7 64-bit

If you are working on a Windows 7 64-bit computer, you may face several problems while trying to import the OpenPIV modules. The following instructions provide a workaround and a short list of issues encountered while trying other alternatives than the one here presented.

Install Enthought Canopy 32-bit no matter if the OS is 64-bit.

Install Visual Studio 8 Express (32 bit) from

> http://download.microsoft.com/download/A/5/4/A54BADB6-9C3F-478D-8657-93B3FC9FE62D/vcsetup.exe

Install the full version of Windows SDK for Windows 7 and .NET Framework 3.5 SP1 from http://www.microsoft.com/en-us/download/details.aspx?id=3138

Remove in /openpiv/src/ the two .c files: lib.c and process.c - so they'll be regenerated by Cython from the .pyx files

In Command Prompt Interface, while executing Python, go to the directory containing OpenPIV and import the OpenPIV modules with:

python setup.py build_ext –inplace

PS. I have had a similar experience while working with a GIS system on a Windows 64-bit machine and trying to get Python modules to work. I started with a Python 64-bit MSI Installer and was not able to find the modules from the GIS system. I ended up installing the Python 32-bit version which worked. My uneducated (perhaps obvious) guess is that the problem lies on the flavour. Maybe a header line stating this (if proved) would be good.

Issues that led to this Workaround:

When using the visual Studio Redistributable Setup x64 the vcvarsall.bat file may not be available, which is needed to activate the C+ compiler.

When installing just the C+ compiler tools from SDK the fike basetsd.h may not be available or simply not found, causing cl.exe to not be properly executed, with following error message:

"Fatal error C1083: Cannot open include file: 'basetsd.h': No such file or directory error: command '"C:Program Files (x86)Microsoft Visual Studio 9.0VCBINcl.exe"

If the command line python setup.py build is used, the following error may appear: "ImportError: No module named lib"

The use of MinGW instead of Visual Studio for the C compiler has been tried and produces the same error as above:

"ImportError: No module named lib"

1.1.3 Having problems?

If you encountered some issues, found difficult to install OpenPIV following these instructions please drop us an email 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:

```
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.

1.2.2 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 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!

1.2.3 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. 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.

1.3.1 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:

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

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=12, dt=0.02, search_area_size=64, sig2noise_method='peak2peak' )

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', max_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, mask, '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
import openpiv.validation
import openpiv.filters
```

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:

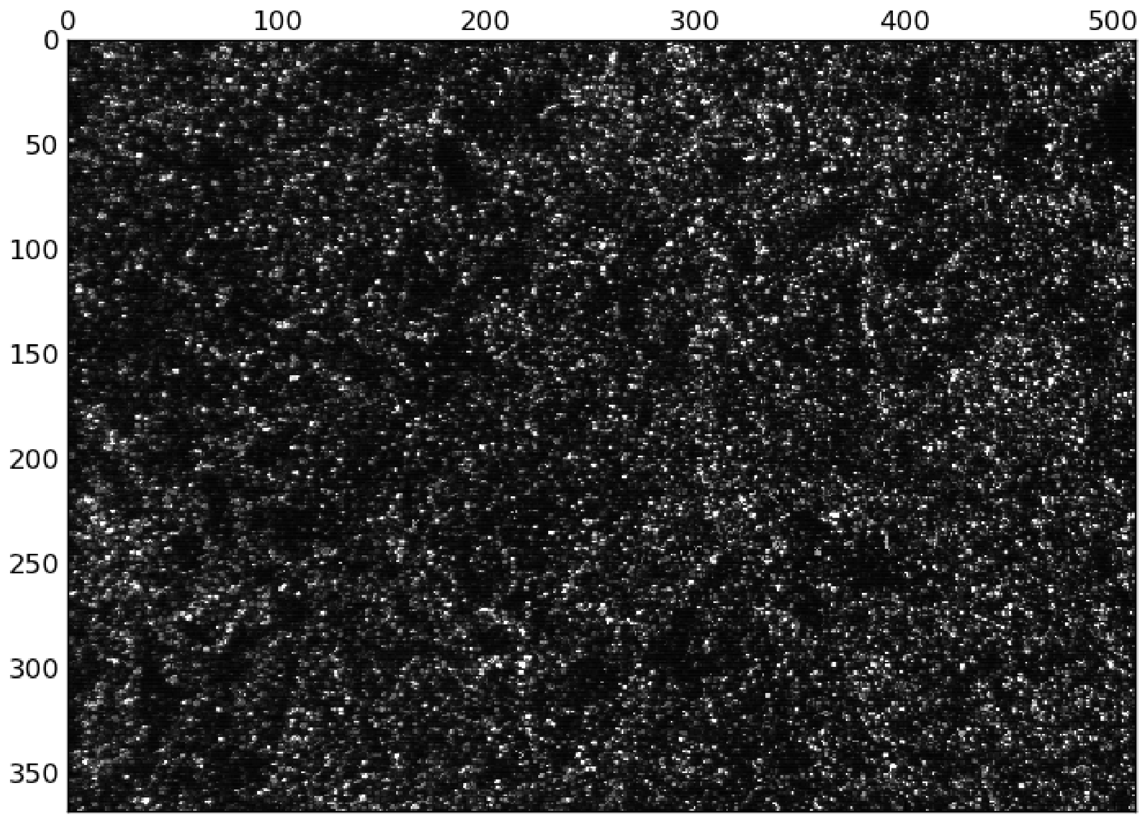
```
frame_a.shape
(369, 511)

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=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 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()`.:

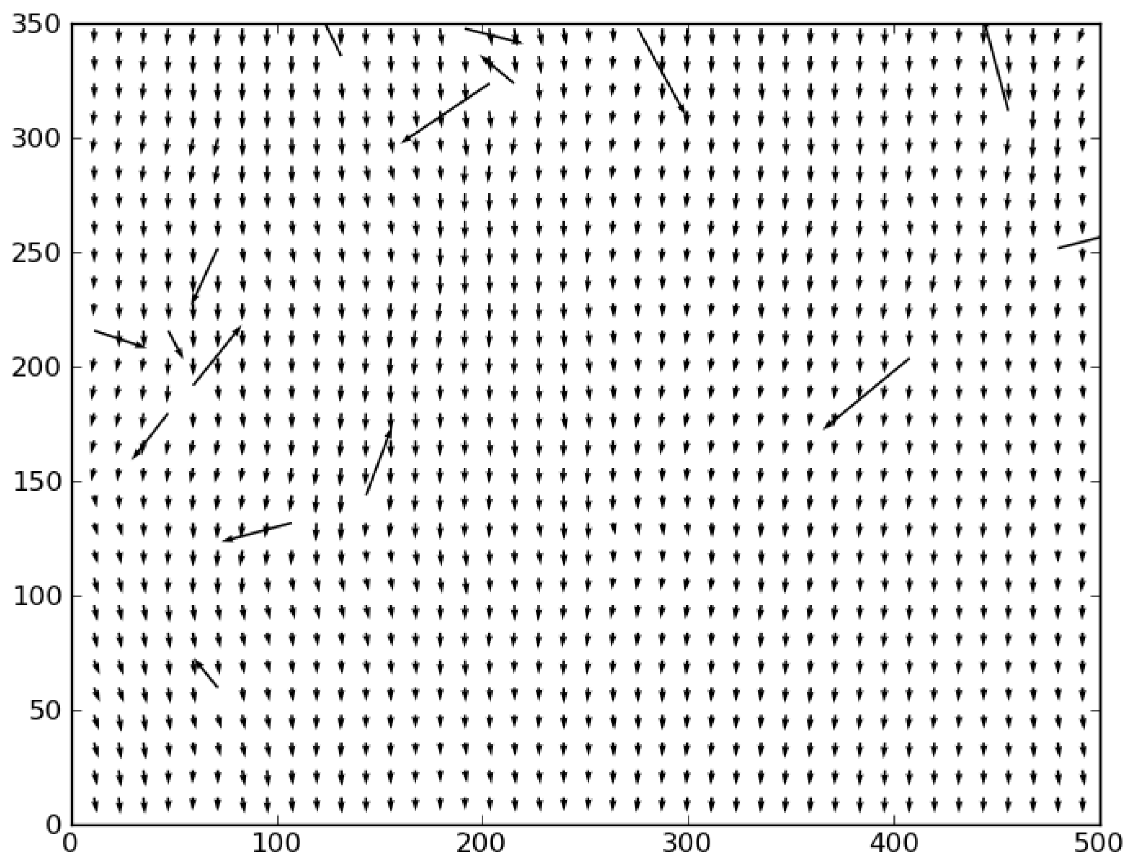
```
x, y = openpiv.process.get_coordinates( image_size=frame_a.shape, window_size=48,␣
↪overlap=32 )
```

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:

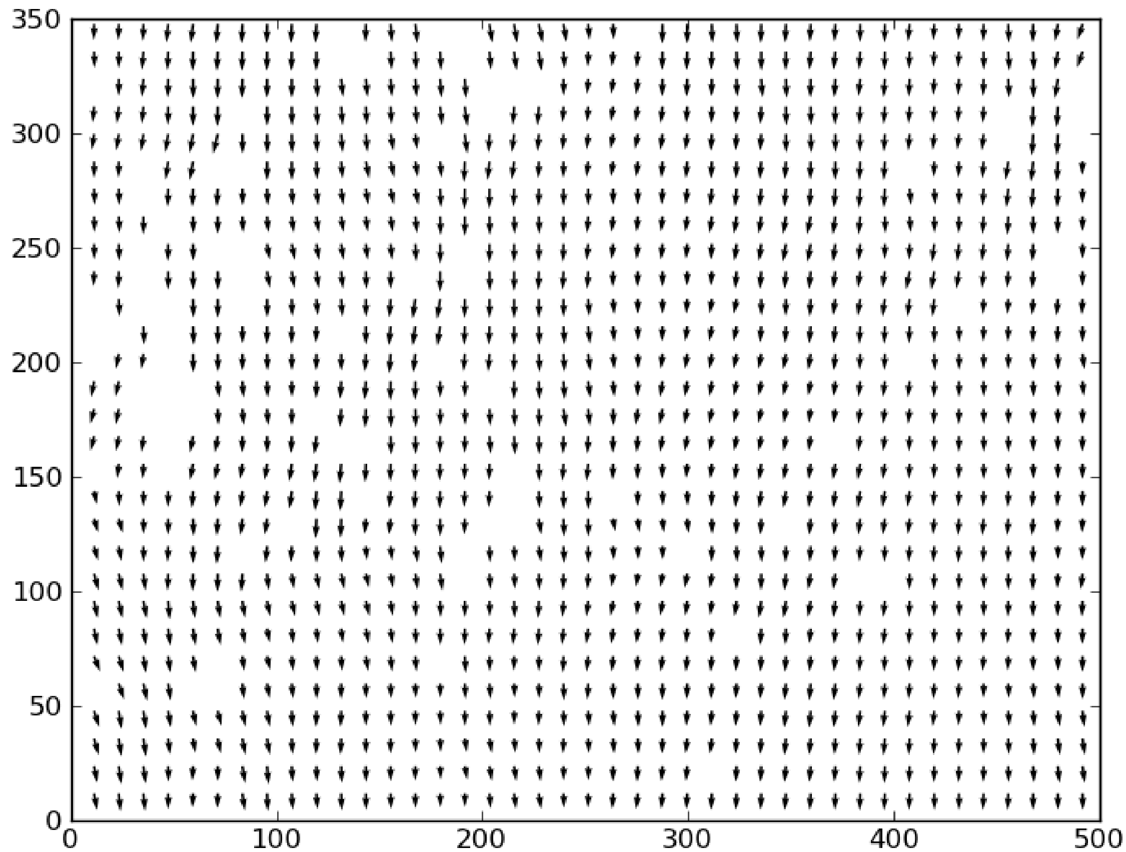


Several outliers vectors can be observed as a result of the small interrogation window size and we need to apply a validation scheme. Since we have information about the signal to noise ratio of the cross-correlation function we can apply a well know filtering scheme, classifing a vector as an outlier 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 )
```

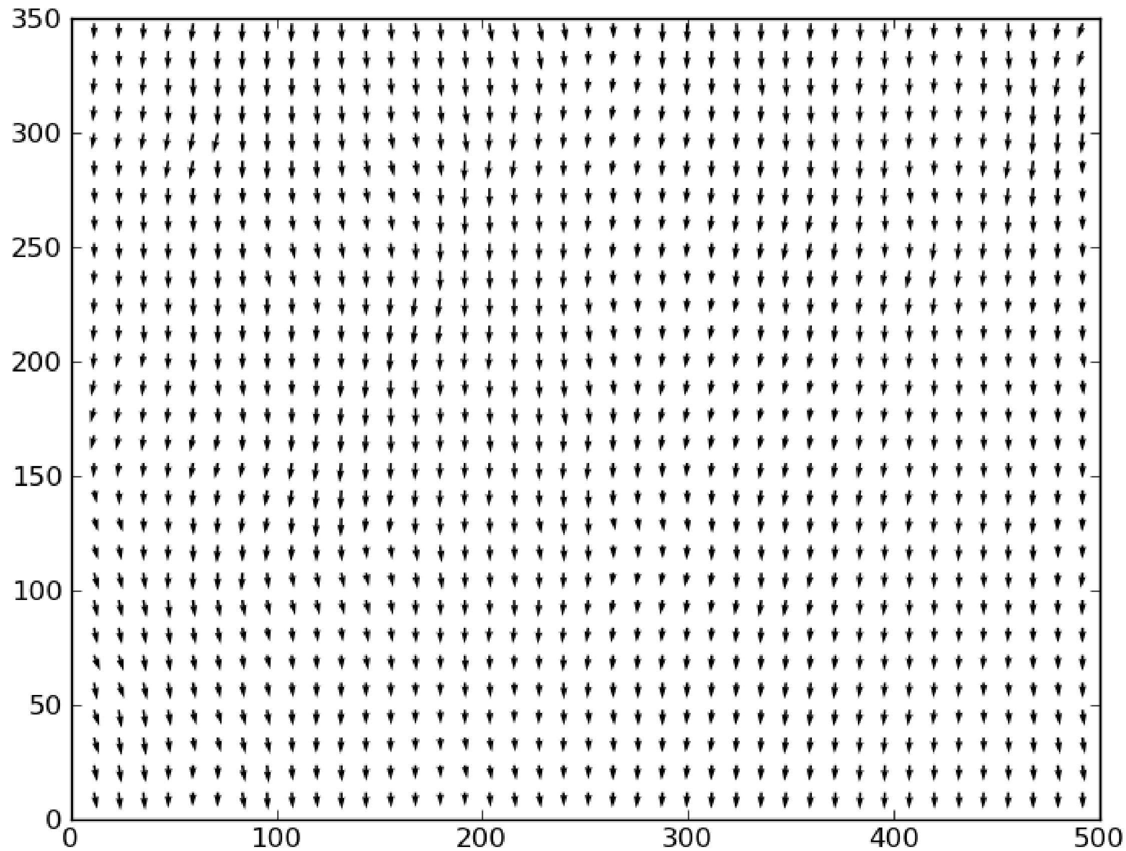


The final step is to replace the missing vector. This is done which the function `openpiv.filters.replace_outliers()`, which implements an iterative image inpainting algorithm with a specified kernel. We pass to this function the two velocity components arrays, a method type `localmean`, the number of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

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=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=
↪'\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.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 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.

1.4 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. `source code and sample images`

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

Part 3: how to process RGB JPEG images in IPython notebook using OpenPIV tools: http://nbviewer.ipython.org/gist/alexlib/8920253

1.5 API reference

This is a complete api reference to the openpiv 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

CHAPTER 2

Indices and tables

- genindex
- modindex
- search