# OpenPIV Documentation

**OpenPIV group**

**Dec 05, 2022**

# Contents

**1 Contents: 3**

1.1 Basics of the PIV algorithms . . . 3
1.2 Installation instruction . . . 13
1.3 OpenPIV tutorial . . . 14
1.4 Multi-grid window deformation algorithm tutorial . . . 17
1.5 OpenPIV masking tutorial . . . 21
1.6 Information for developers and contributors . . . 28
1.7 API reference . . . 30
1.8 Frequently Asked Questions about PIV parameters . . . 49

**2 Indices and tables 51**

**Index 53**

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 Basics of the PIV algorithms

Using open source PIV software, OpenPIV (http://www.openpiv.net) written with the great help of Python, Numpy, Scipy (http://www.scipy.org) and runs online thanks to the great MyBinder project.

### 1.1.1 What is it about? Particle Image Velocimetry (PIV)

From Wikipedia: "Particle image velocimetry (PIV) is an optical method of flow visualization used in education and research. It is used to obtain instantaneous velocity measurements and related properties in fluids. The fluid is seeded with tracer particles which, for sufficiently small particles, are assumed to faithfully follow the flow dynamics (the degree to which the particles faithfully follow the flow is represented by the Stokes number). The fluid with entrained particles is illuminated so that particles are visible. The motion of the seeding particles is used to calculate speed and direction (the velocity field) of the flow being studied." Read more at http://en.wikipedia.org/wiki/Particle_image_velocimetry.

Particle Image Velocimetry (PIV) is a non-intrusive state-of-the-art technique for flow measurements (e.g.: Raffel et al., 2007, Adrian, 1991). The PIV technique is based on image recording of the illuminated flow field using seeding particles. The technique is based on illuminating the particles in a plane by forming a coherent light sheet. The light scattered by the particles is recorded on a sequence of image frames. The displacement of the particle images between two consecutive light pulses is determined through evaluation of the PIV recordings and by applying a spatial cross-correlation function as implemented by the OpenPIV resulting with a two dimensional two component velocity field.

In practice, small tracer particles, common sized are in the order of 10-100 microns, are introduced to the flow. The flow is illuminated twice by means of a laser light sheet forming a plane where the camera is focused on. The time delay between the pulses depends on the mean velocity and the image magnification. It is assumed that the tracer particles follow the local flow velocity between the two consecutive illuminations. The light scattered from the tracer particles is imaged via an optical lens on a digital camera. The images, acquired as pairs correspond to the two laser pulses, are than correlated using a cross-correlation function and image processing tools in order to provide the velocity field.

The effectiveness of the measurement results strongly depends on a large number of parameters such as particles concentration, size distribution and shape, illumination source, recording device, and synchronization between the illumination, acquisition and recording systems (Huang et al., 1997). An appropriate choice of the different parameters of the cross correlation analysis (e.g., interrogation area, time between pulses, scaling) will influence the results accuracy. Read more about PIV in the following chapters: Gurka and Kit, in Handbook of Environmental Fluid Mechanics, CRC Press, 2014 http://www.crcnetbase.com/doi/abs/10.1201/b13691-39 or Taylor, Gurka and Liberzon "Particle Image Velocimetry for Biological Mechanics" in the Handbook of Imaging in Biological Mechanics, CRC Press, 2015, http://www.crcpress.com/product/isbn/9781466588134.

## Open source software to learn the basics

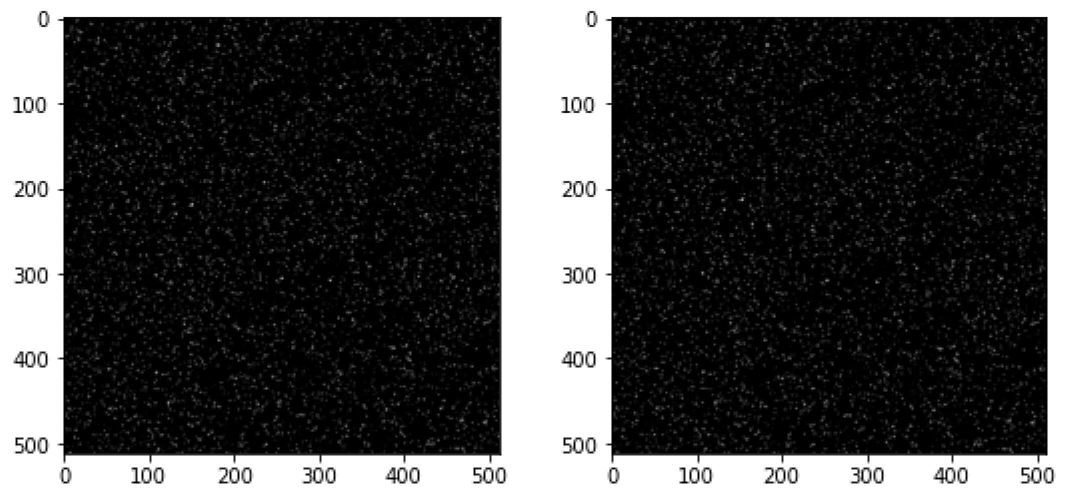
In principle velocimetry is a method to find out the velocity field of the moving fluid. "Particle" image velocimetry is the way to get velocity field from images of small particles, called tracers. The basic principle is to use two images of the same particles with a small time delay between them. For that purpose, typically two laser shots are created and two images are taken.

This tutorial will follow the most simple analysis path from the two images to the velocity field and some post-analysis. We will use one of many open source packages, the open source particle image velocimetry http://www.openpiv.net

[1]:
```
# import the standard numerical and plotting packages
import matplotlib.pyplot as plt
import numpy as np
from skimage.io import imread
```

We have downloaded some sample images from PIV challenge, see http://www.pivchallenge.org/pub/#b or another standard PIV images project: http://www.piv.jp/down/image05e.html

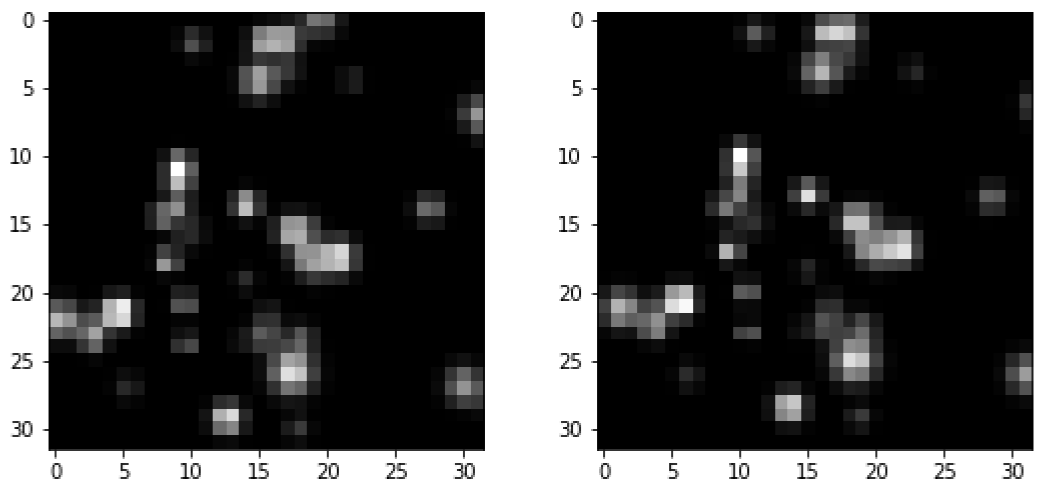
[2]:
```
# load the images
a = imread("../images/B005_1.tif")
b = imread("../images/B005_2.tif")

fig, axs = plt.subplots(1, 2, figsize=(9, 4))
axs[0].imshow(a, cmap=plt.cm.gray)
axs[1].imshow(b, cmap=plt.cm.gray)
plt.show()
```



The two images show the positions of the particles at two different times. We can analyze small regions of interest, called interrogation windows. Typically we can start with a size of 32 x 32 pixels or smaller. Until recently, the fast algorithms used powers of 2, so the historical sizes are always powers of 2: 8, 16, 32, 64, 128, …

Let's take the first top left windows from each image.

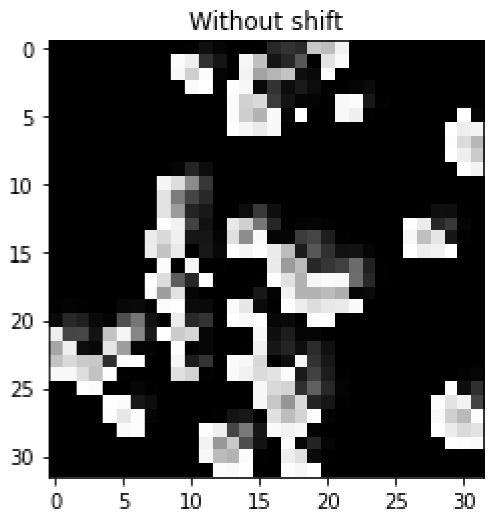
```
[3]: win_size = 32

a_win = a[:win_size, :win_size].copy()
b_win = b[:win_size, :win_size].copy()

fig, axs = plt.subplots(1, 2, figsize=(9, 4))
axs[0].imshow(a_win, cmap=plt.cm.gray)
axs[1].imshow(b_win, cmap=plt.cm.gray)
plt.show()
```



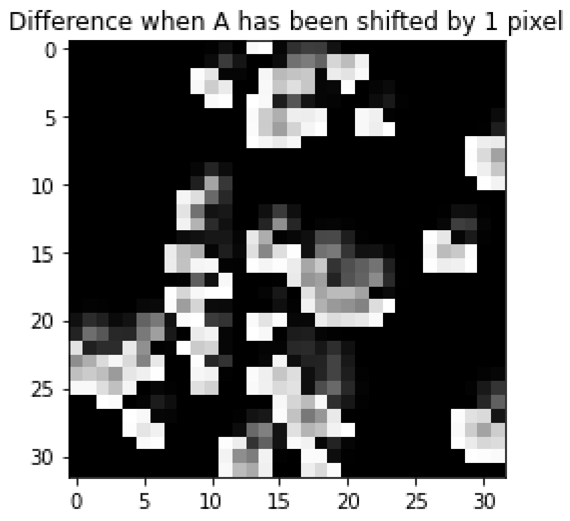
We can see that the bright pixels moved between the two frames. We can find out the distance that all the particles moved between frame A and frame B using the principles of least squares or correlations, but let's first try to get it manually.

If we shift the window IA by some pixels to the right and subtract from IB the shifted IA, we shall see how good the shift predicts the real displacement between the two.

```
[4]: fig = plt.imshow(b_win - a_win, cmap=plt.cm.gray)
plt.title("Without shift")
plt.show()
```



```
[5]: plt.imshow(b_win - np.roll(a_win, (1, 0), axis=(0, 1)), cmap=plt.cm.gray)
     plt.title("Difference when A has been shifted by 1 pixel")
     plt.show()
```



Let's try to find the best shift algorithmically: shift and calculated the sum of squared differences the minimum is the best shift

```
[6]: def match_template(img, template, maxroll=8):
         best_dist = np.inf
         best_shift = (-1, -1)
         for y in range(maxroll):
             for x in range(maxroll):
                 # calculate Euclidean distance
                 dist = np.sqrt(np.sum((img - np.roll(template, (y, x), axis=(0, 1))) **␣
     ↪2))
                 if dist < best_dist:
                     best_dist = dist
                     best_shift = (y, x)
         return (best_dist, best_shift)
```

```
[7]: # let's test that it works by manually rolling (shifting circurlarly) the same
     # image
     match_template(np.roll(a_win, (2, 0), axis=(0, 1)), a_win)
```

```
[7]: (0.0, (2, 0))
```

```
[8]: # indeed, when we find the correct shift, we got zero distance. it's not so in real␣
     ↪images:
     best_dist, best_shift = match_template(b_win, a_win)
     print(f"{best_dist=}")
     print(f"{best_shift=}")
```

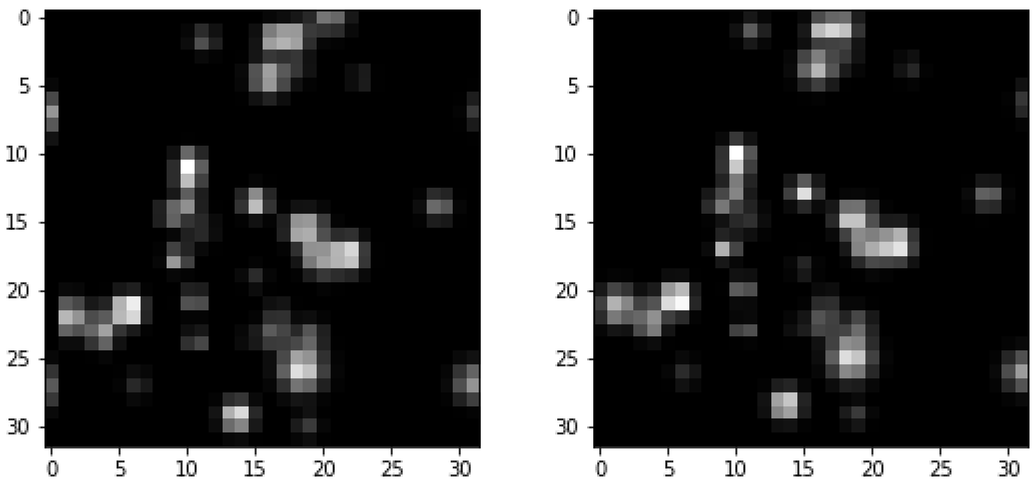
```
best_dist=170.43180454363556
best_shift=(0, 1)
```

We can draw this as a vector of velocity

$$u = \frac{\Delta x \text{ pixels}}{\Delta t}, \qquad v = \frac{\Delta y \text{ pixels}}{\Delta t}$$

where $\Delta t$ is the time interval (delay) between the two images (or two laser pulses).

```
[9]: fig, axs = plt.subplots(1, 2, figsize=(9, 4))
     axs[0].imshow(np.roll(a_win, best_shift, axis=(0, 1)), cmap='gray')
     axs[1].imshow(b_win, cmap='gray')
     plt.show()
```



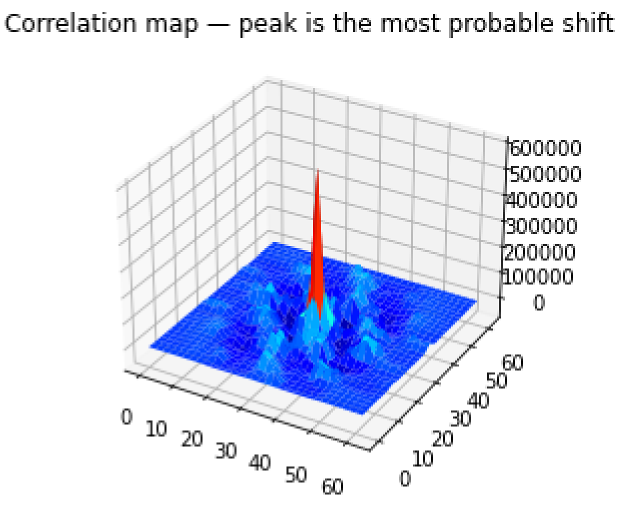
Well, maybe it's not the best match, but it is already better than nothing.

The problem now is that manually shifting each image and repeating the loop many times is impractical. However, based on the same principle of finding the right shift, one can get by using a different template matching principle, based on the property called cross-correlation (cross because we use two different images). In short this is an efficient computational algorithm to find out the right shift. You can see more details here: http://paulbourke.net/miscellaneous/correlate/.

```
[10]: from scipy.signal import correlate

      cross_corr = correlate(b_win - b_win.mean(), a_win - a_win.mean(), method="fft")
      # Note that it's approximately twice as large than the original windows, as we
      # can shift a_win by a maximum of it's size - 1 horizontally and vertically
      # while still maintaining some overlap between the two windows.
      print("Size of the correlation map: %d x %d" % cross_corr.shape)
```

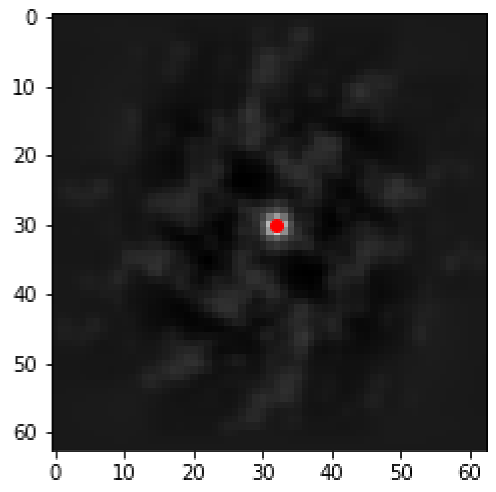
```
Size of the correlation map: 63 x 63
```

```
[11]: # let's see what the cross-correlation looks like
      from mpl_toolkits.mplot3d import Axes3D

      fig = plt.figure()
      ax = fig.add_subplot(projection="3d")
      Y, X = np.meshgrid(np.arange(cross_corr.shape[0]), np.arange(cross_corr.shape[1]))

      ax.plot_surface(Y, X, cross_corr, cmap='jet', linewidth=0.2)  # type: ignore
      plt.title("Correlation map -- peak is the most probable shift")
      plt.show()
```



[12]:
```
# let's see the same correlation map, from above
plt.imshow(cross_corr, cmap=plt.cm.gray)

y, x = np.unravel_index(cross_corr.argmax(), cross_corr.shape)
print(f"{y=}, {x=}")

plt.plot(x, y, "ro")
plt.show()
```

```
y=30, x=32
```



The image of the correlation map shows the same result that we got manually looping. We need to shift `a_win` to give the best possible correlation between the two windows. If there best correlation would come from no shift, the result would be `(31, 31)`—the center of symmetry.

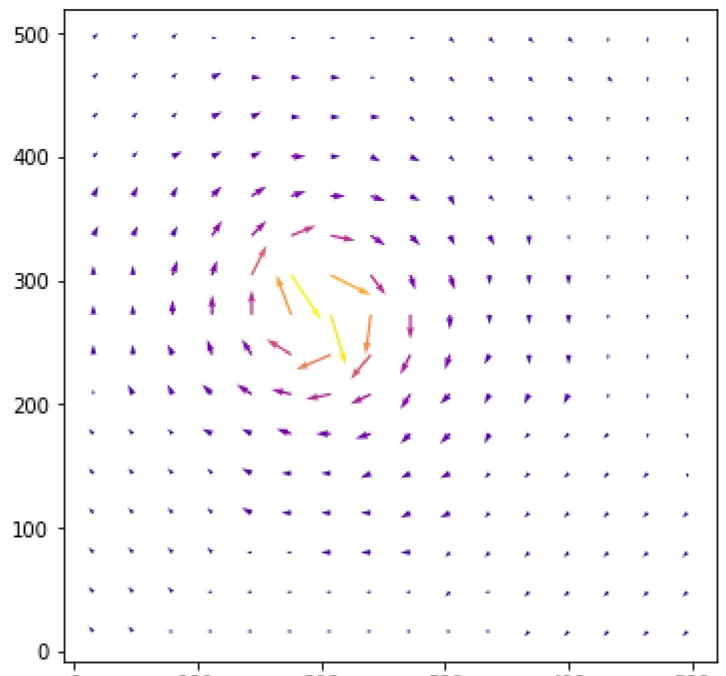
[13]:
```
dy, dx = y - 31, x - 31
print(f"{dy=}, {dx=}")
```

```
dy=-1, dx=1
```

We can get the first velocity field by repeating this analysis for all small windows. Let's take 32 x 32 pixels windows from each image and do the loop:

[14]:
```
def vel_field(curr_frame, next_frame, win_size):
    ys = np.arange(0, curr_frame.shape[0], win_size)
    xs = np.arange(0, curr_frame.shape[1], win_size)
    dys = np.zeros((len(ys), len(xs)))
    dxs = np.zeros((len(ys), len(xs)))
    for iy, y in enumerate(ys):
        for ix, x in enumerate(xs):
            int_win = curr_frame[y : y + win_size, x : x + win_size]
            search_win = next_frame[y : y + win_size, x : x + win_size]
            cross_corr = correlate(
                search_win - search_win.mean(), int_win - int_win.mean(), method="fft"
            )
            dys[iy, ix], dxs[iy, ix] = (
                np.unravel_index(np.argmax(cross_corr), cross_corr.shape)
                - np.array([win_size, win_size])
                + 1
            )
    # draw velocity vectors from the center of each window
    ys = ys + win_size / 2
    xs = xs + win_size / 2
    return xs, ys, dxs, dys
```

[15]:
```
xs, ys, dxs, dys = vel_field(a, b, 32)
norm_drs = np.sqrt(dxs ** 2 + dys ** 2)

fig, ax = plt.subplots(figsize=(6, 6))
# we need these flips on y since quiver uses a bottom-left origin, while our
# arrays use a top-right origin
ax.quiver(
    xs,
    ys[::-1],
    dxs,
    -dys,
    norm_drs,
    cmap="plasma",
    angles="xy",
    scale_units="xy",
    scale=0.25,
)
ax.set_aspect("equal")
plt.show()
```



If you've followed along this far, great! Now you understand the basics.

We can also try out a variant of this that uses a search window larger than the interrogation window instead of relying on zero-padding. By avoiding using zero-padding around the search window, movement detection should theoretically be a bit better, assuming that the window sizes are chosen well.

```
[16]: def vel_field_asymmetric_wins(
          curr_frame, next_frame, half_int_win_size, half_search_win_size
      ):
          ys = np.arange(half_int_win_size[0], curr_frame.shape[0], 2 * half_int_win_
      ↪size[0])
          xs = np.arange(half_int_win_size[1], curr_frame.shape[1], 2 * half_int_win_
      ↪size[1])
          dys = np.zeros((len(ys), len(xs)))
          dxs = np.zeros((len(ys), len(xs)))
          for iy, y in enumerate(ys):
              for ix, x in enumerate(xs):
                  int_win = curr_frame[
                      y - half_int_win_size[0] : y + half_int_win_size[0],
                      x - half_int_win_size[1] : x + half_int_win_size[1],
                  ]
                  search_win_y_min = y - half_search_win_size[0]
                  search_win_y_max = y + half_search_win_size[0]
                  search_win_x_min = x - half_search_win_size[1]
                  search_win_x_max = x + half_search_win_size[1]
                  truncated_search_win = next_frame[
                      max(0, search_win_y_min) : min(b.shape[0], search_win_y_max),
                      max(0, search_win_x_min) : min(b.shape[1], search_win_x_max),
                  ]
                  cross_corr = correlate(
                      truncated_search_win - np.mean(truncated_search_win),
                      int_win - np.mean(int_win),
```

(continues on next page)

(continued from previous page)

```
                mode="valid",
                method="fft",
            )
            dy, dx = np.unravel_index(np.argmax(cross_corr), cross_corr.shape)
            # if the top of the search window got truncated, shift the origin
            # up to the top edge of the (non-truncated) search window
            if search_win_y_min < 0:
                dy += -search_win_y_min
            # if the left of the search window got truncated, shift the origin
            # over to the left edge of the (non-truncated) search window
            if search_win_x_min < 0:
                dx += -search_win_x_min
            # shift origin to the center of the search window
            dy -= half_search_win_size[0] - half_int_win_size[0]
            dx -= half_search_win_size[1] - half_int_win_size[1]
            dys[iy, ix] = dy
            dxs[iy, ix] = dx
    return xs, ys, dxs, dys
```

[17]:
```
int_win_size = np.array([32, 32])
print(f"{int_win_size=}")
assert np.all(np.array(a.shape) % int_win_size == 0)
assert np.all(int_win_size % 2 == 0)
half_int_win_size = int_win_size // 2

search_win_size = int_win_size * 2
print(f"{search_win_size=}")
assert np.all(search_win_size % 2 == 0)
half_search_win_size = search_win_size // 2
assert np.all(search_win_size > int_win_size)
print(
    "max velocity that can be detected with these window sizes: "
    + f"{half_search_win_size - half_int_win_size}"
)
```

```
int_win_size=array([32, 32])
search_win_size=array([64, 64])
max velocity that can be detected with these window sizes: [16 16]
```

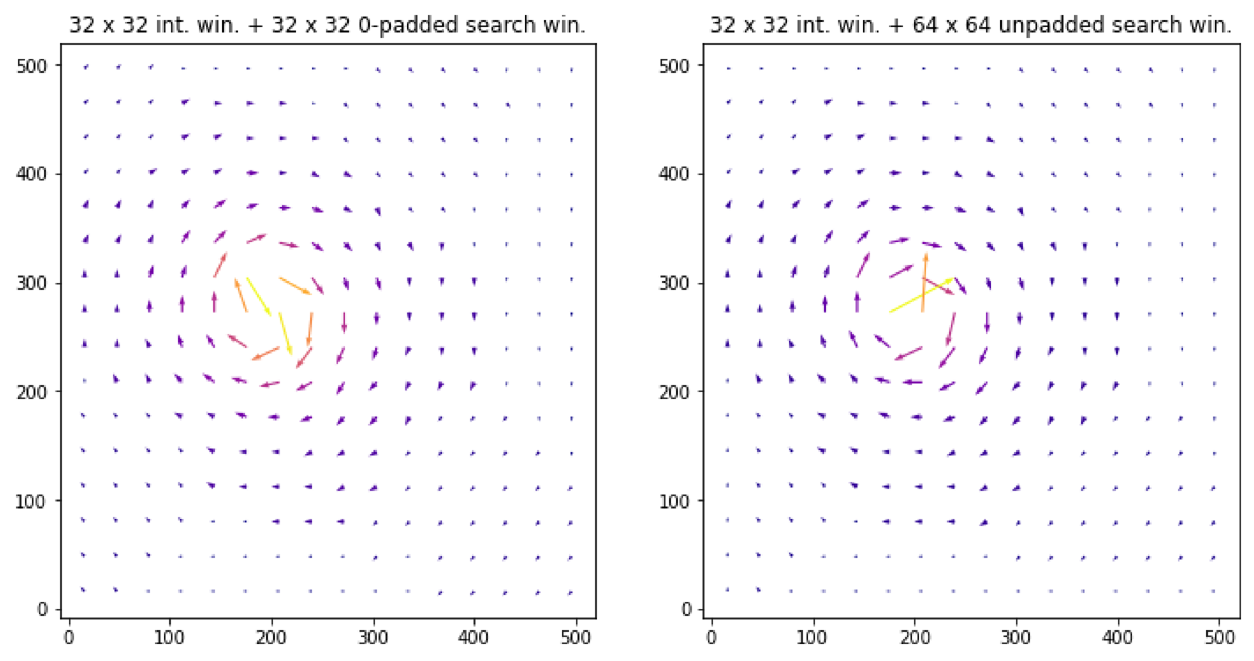
Making the search window larger compared to the interrogation window would allow for larger velocities to be detected.

[18]:
```
xs_asym, ys_asym, dxs_asym, dys_asym = vel_field_asymmetric_wins(
    a, b, half_int_win_size, half_search_win_size
)
norm_drs_asym = np.sqrt(dxs_asym ** 2 + dys_asym ** 2)

fig, axs = plt.subplots(1, 2, figsize=(12, 6))
axs[0].quiver(
    xs,
    ys[::-1],
    dxs,
    -dys,
    norm_drs,
    cmap="plasma",
    angles="xy",
    scale_units="xy",
```

(continues on next page)

(continued from previous page)

```
    scale=0.25,
)
axs[1].quiver(
    xs_asym,
    ys_asym[::-1],
    dxs_asym,
    -dys_asym,
    norm_drs_asym,
    cmap="plasma",
    angles="xy",
    scale_units="xy",
    scale=0.25,
)
axs[0].set_title(
    f"{win_size} x {win_size} int. win. + "
    f"{win_size} x {win_size} 0-padded search win."
)
axs[1].set_title(
    f"{int_win_size[0]} x {int_win_size[1]} int. win. + "
    f"{search_win_size[0]} x {search_win_size[0]} unpadded search win."
)
ax.set_aspect("equal")
plt.show()
```



## 1.1.2 Additional examples

### Run this Jupyter notebook without installation:

Now you can take our online Jupyter notebook and process any pairs of images yourself, on the cloud, http://github.com/openpiv/openpiv-python-example.

Download the OpenPIV and try it yourself: https://github.com/OpenPIV

### See multiple examples

Including how to work with movies, different series of files, all kind of tips and tricks in our repository of Jupyter notebooks in the https://github.com/OpenPIV/openpiv-python-examples

## 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
- scikit-image

On all platforms, the following Python distribution is recommended:

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

### 1.2.2 Installation

Use *conda*

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

Or use *pip*

```
pip install numpy cython
pip install openpiv --pre
```

### 1.2.3 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/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. Remeber to build the extension with

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

### 1.2.4 Experience problems?

If you encountered some issues, found difficult to install OpenPIV following these instructions please register and write on our Google groups forum https://groups.google.com/g/openpiv-users , so that we can help you and improve this page!

## 1.3 OpenPIV tutorial

In this tutorial we read a pair of images and perform the PIV using a standard algorithm. At the end, the velocity vector field is plotted.

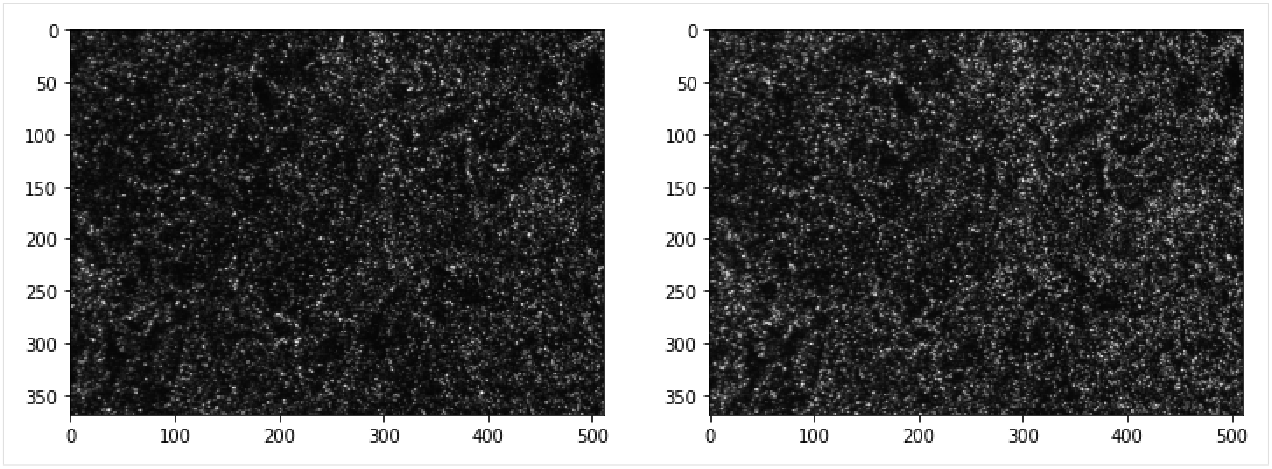
[21]:
```
from openpiv import tools, pyprocess, validation, filters, scaling

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import imageio
import importlib_resources
import pathlib
```

### 1.3.1 Reading images:

The images can be read using the `imread` function, and diplayed with matplotlib.

[22]:
```

path = importlib_resources.files('openpiv')
```

[23]:
```

frame_a  = tools.imread( path / 'data/test1/exp1_001_a.bmp' )
frame_b  = tools.imread( path / 'data/test1/exp1_001_b.bmp' )

fig,ax = plt.subplots(1,2,figsize=(12,10))
ax[0].imshow(frame_a,cmap=plt.cm.gray);
ax[1].imshow(frame_b,cmap=plt.cm.gray);
```



### 1.3.2 Processing

In this tutorial, we are going to use the `extended_search_area_piv` function, wich is a standard PIV cross-correlation algorithm.

This function allows the search area (`search_area_size`) in the second frame to be larger than the interrogation window in the first frame (`window_size`). Also, the search areas can overlap (`overlap`).

The `extended_search_area_piv` function will return three arrays. 1. The `u` component of the velocity vectors 2. The `v` component of the velocity vectors 3. The signal-to-noise ratio (S2N) of the cross-correlation map of each vector. The higher the S2N of a vector, the higher the probability that its magnitude and direction are correct.

[24]:
```
winsize = 32 # pixels, interrogation window size in frame A
searchsize = 38  # pixels, search area size in frame B
overlap = 17 # pixels, 50% overlap
dt = 0.02 # sec, time interval between the two frames

u0, v0, sig2noise = pyprocess.extended_search_area_piv(
    frame_a.astype(np.int32),
    frame_b.astype(np.int32),
    window_size=winsize,
    overlap=overlap,
    dt=dt,
    search_area_size=searchsize,
    sig2noise_method='peak2peak',
)
```

The function `get_coordinates` finds the center of each interrogation window. This will be useful later on when plotting the vector field.

[25]:
```
x, y = pyprocess.get_coordinates(
    image_size=frame_a.shape,
    search_area_size=searchsize,
    overlap=overlap,
)
```

### 1.3.3 Post-processing

Strictly speaking, we are ready to plot the vector field. But before we do that, we can perform some convenient pos-processing.

To start, lets use the function `sig2noise_val` to get a mask indicating which vectors have a minimum amount of S2N. Vectors below a certain threshold are substituted by `NaN`. If you are not sure about which threshold value to use, try taking a look at the S2N histogram with:

`plt.hist(sig2noise.flatten())`

[26]:
```
invalid_mask = validation.sig2noise_val(
    sig2noise,
    threshold = 1.05,
)
```

Another useful function is `replace_outliers`, which will find outlier vectors, and substitute them by an average of neighboring vectors. The larger the `kernel_size` the larger is the considered neighborhood. This function uses an iterative image inpainting algorithm. The amount of iterations can be chosen via `max_iter`.

[27]:
```
u2, v2 = filters.replace_outliers(
    u0, v0,
    invalid_mask,
    method='localmean',
    max_iter=3,
    kernel_size=3,
)
```

Next, we are going to convert pixels to millimeters, and flip the coordinate system such that the origin becomes the bottom left corner of the image.

[28]:
```
# convert x,y to mm
# convert u,v to mm/sec

x, y, u3, v3 = scaling.uniform(
    x, y, u2, v2,
    scaling_factor = 96.52,  # 96.52 pixels/millimeter
)

# 0,0 shall be bottom left, positive rotation rate is counterclockwise
x, y, u3, v3 = tools.transform_coordinates(x, y, u3, v3)
```

### 1.3.4 Results

The function `save` is used to save the vector field to a ASCII tabular file. The coordinates and S2N mask are also saved.

[29]:
```
tools.save('exp1_001.txt' , x, y, u3, v3, invalid_mask)
```

Finally, the vector field can be plotted with `display_vector_field`.

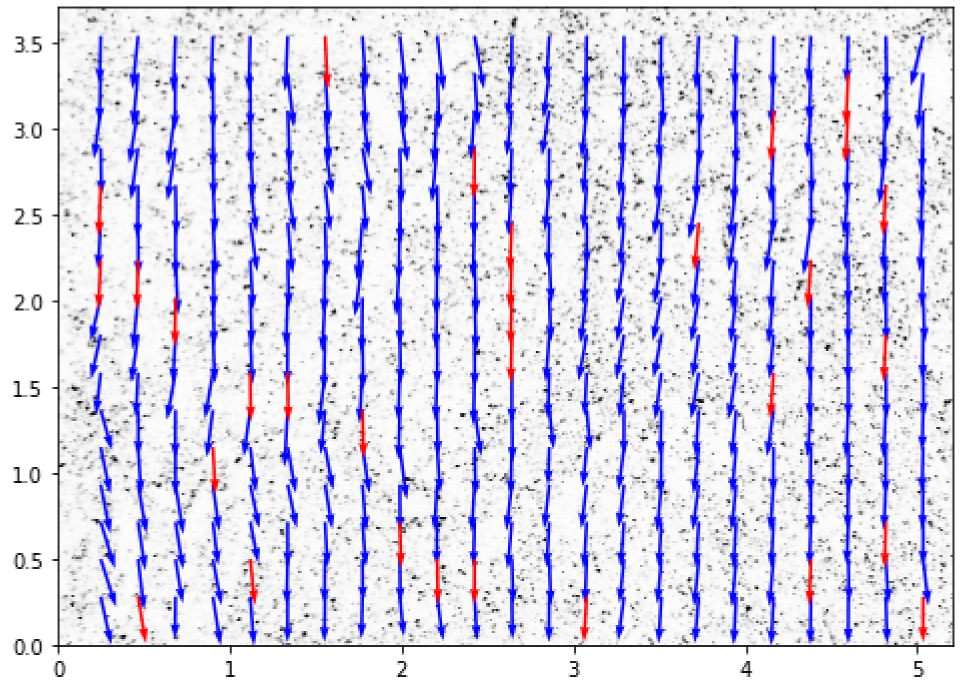
Vectors with S2N bellow the threshold are displayed in red.

[30]:
```
fig, ax = plt.subplots(figsize=(8,8))
tools.display_vector_field(
    pathlib.Path('exp1_001.txt'),
```

(continues on next page)

(continued from previous page)

```
    ax=ax, scaling_factor=96.52,
    scale=50, # scale defines here the arrow length
    width=0.0035, # width is the thickness of the arrow
    on_img=True, # overlay on the image
    image_name= str(path / 'data'/'test1'/'exp1_001_a.bmp'),
);
```



## 1.4 Multi-grid window deformation algorithm tutorial

[6]:
```
# import packages
```

[7]:
```
from openpiv import windef  # <---- see windef.py for details
from openpiv import tools, scaling, validation, filters, preprocess
import openpiv.pyprocess as process
from openpiv import pyprocess
import numpy as np
import pathlib
import importlib_resources
from time import time
import warnings


import matplotlib.pyplot as plt
%matplotlib inline
```

### 1.4.1 Set up all the settings:

1. where the images are

2. where to save the results
3. names of the image files
4. what is the region of interest
5. do you apply dynamic masking or a masking image
6. what kind of correlation to apply: circular vs linear
7. interrogation window sizes, overlap sizes, number of iterations
8. time interval, interpolation options, etc.

Read the tutorial by Theo Kaufer with all the details. See `windef.py` for more code details

[8]:
```
settings = windef.PIVSettings()

path = importlib_resources.files('openpiv')

'Data related settings'
# Folder with the images to process
settings.filepath_images = path / 'data' / 'test1'  # type: ignore
# Folder for the outputs
settings.save_path = path / 'data' / 'test1'  # type: ignore
# Root name of the output Folder for Result Files
settings.save_folder_suffix = 'Test_1'
# Format and Image Sequence (see below for more options)
settings.frame_pattern_a = 'exp1_001_a.bmp'
settings.frame_pattern_b = 'exp1_001_b.bmp'

# or if you have a sequence:
# settings.frame_pattern_a = '000*.tif'
# settings.frame_pattern_b = '(1+2),(2+3)'
# settings.frame_pattern_b = '(1+3),(2+4)'
# settings.frame_pattern_b = '(1+2),(3+4)'

'Region of interest'
# (50,300,50,300) #Region of interest: (xmin,xmax,ymin,ymax) or 'full' for full image
settings.roi = 'full'

'Image preprocessing'
# 'None' for no masking, 'edges' for edges masking, 'intensity' for intensity masking
# WARNING: This part is under development so better not to use MASKS
settings.dynamic_masking_method = 'None'
settings.dynamic_masking_threshold = 0.005
settings.dynamic_masking_filter_size = 7

settings.deformation_method = 'symmetric'

'Processing Parameters'
settings.correlation_method='circular'  # 'circular' or 'linear'
settings.normalized_correlation=False

settings.num_iterations = 2  # select the number of PIV passes
# add the interroagtion window size for each pass.
# For the moment, it should be a power of 2
settings.windowsizes = (64, 32, 16) # if longer than n iteration the rest is ignored
# The overlap of the interroagtion window for each pass.
settings.overlap = (32, 16, 8) # This is 50% overlap
# Has to be a value with base two. In general window size/2 is a good choice.
```
(continues on next page)

(continued from previous page)

```
# methode used for subpixel interpolation: 'gaussian','centroid','parabolic'
settings.subpixel_method = 'gaussian'
# order of the image interpolation for the window deformation
settings.interpolation_order = 3
settings.scaling_factor = 1  # scaling factor pixel/meter
settings.dt = 1  # time between to frames (in seconds)
'Signal to noise ratio options (only for the last pass)'
# It is possible to decide if the S/N should be computed (for the last pass) or not
# settings.extract_sig2noise = True  # 'True' or 'False' (only for the last pass)
# method used to calculate the signal to noise ratio 'peak2peak' or 'peak2mean'
settings.sig2noise_method = 'peak2peak'
# select the width of the masked to masked out pixels next to the main peak
settings.sig2noise_mask = 2
# If extract_sig2noise==False the values in the signal to noise ratio
# output column are set to NaN
'vector validation options'
# choose if you want to do validation of the first pass: True or False
settings.validation_first_pass = True
# only effecting the first pass of the interrogation the following passes
# in the multipass will be validated
'Validation Parameters'
# The validation is done at each iteration based on three filters.
# The first filter is based on the min/max ranges. Observe that these values are␣
↪defined in
# terms of minimum and maximum displacement in pixel/frames.
settings.min_max_u_disp = (-30, 30)
settings.min_max_v_disp = (-30, 30)
# The second filter is based on the global STD threshold
settings.std_threshold = 7  # threshold of the std validation
# The third filter is the median test (not normalized at the moment)
settings.median_threshold = 3  # threshold of the median validation
# On the last iteration, an additional validation can be done based on the S/N.
settings.median_size=1 #defines the size of the local median
'Validation based on the signal to noise ratio'
# Note: only available when extract_sig2noise==True and only for the last
# pass of the interrogation
# Enable the signal to noise ratio validation. Options: True or False
# settings.do_sig2noise_validation = False # This is time consuming
# minmum signal to noise ratio that is need for a valid vector
settings.sig2noise_threshold = 1.2
'Outlier replacement or Smoothing options'
# Replacment options for vectors which are masked as invalid by the validation
settings.replace_vectors = True # Enable the replacment. Chosse: True or False
settings.smoothn=True #Enables smoothing of the displacemenet field
settings.smoothn_p=0.5 # This is a smoothing parameter
# select a method to replace the outliers: 'localmean', 'disk', 'distance'
settings.filter_method = 'localmean'
# maximum iterations performed to replace the outliers
settings.max_filter_iteration = 4
settings.filter_kernel_size = 2  # kernel size for the localmean method
'Output options'
# Select if you want to save the plotted vectorfield: True or False
settings.save_plot = False
# Choose wether you want to see the vectorfield or not :True or False
settings.show_plot = True
settings.scale_plot = 200  # select a value to scale the quiver plot of the␣
↪vectorfield
```

(continues on next page)

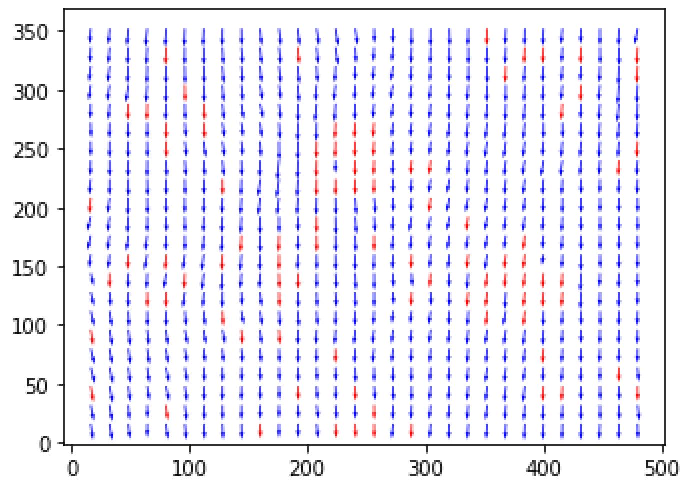
(continued from previous page)

```
# run the script with the given settings
```

### 1.4.2 Run the `windef.py` function, called `piv` with these settings

[9]:
```
windef.piv(settings)
```

```
/home/user/Documents/repos/openpiv-python/openpiv/data/test1
exp1_001_a.bmp
True
[PosixPath('/home/user/Documents/repos/openpiv-python/openpiv/data/test1/exp1_001_a.
↪bmp')]
```



```
Image Pair 1
exp1_001_a exp1_001_b
```

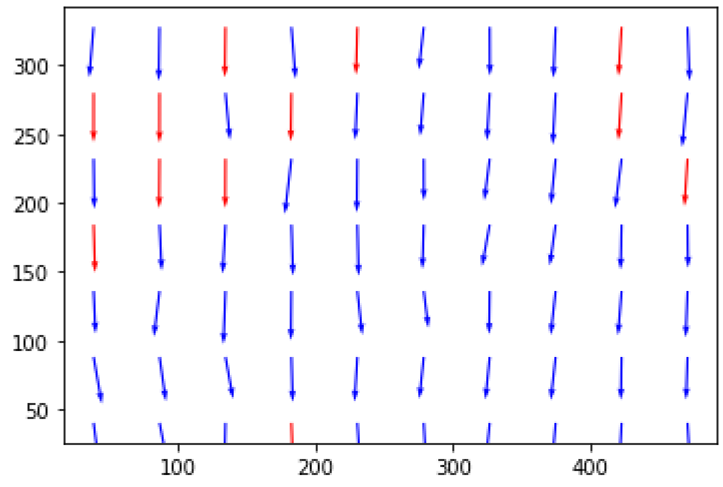
### 1.4.3 Run the extended search area PIV for comparison

[10]:
```

# we can run it from any folder
path = settings.filepath_images


frame_a  = tools.imread( path / settings.frame_pattern_a)
frame_b  = tools.imread(  path / settings.frame_pattern_b)

frame_a = (frame_a).astype(np.int32)
frame_b = (frame_b).astype(np.int32)

u, v, sig2noise = process.extended_search_area_piv( frame_a, frame_b, \
    window_size=32, overlap=16, dt=1, search_area_size=64, sig2noise_method='peak2peak
↪' )
x, y = process.get_coordinates( image_size=frame_a.shape,
                               search_area_size=64, overlap=16 )
mask_s2n = validation.sig2noise_val(sig2noise, threshold = 1.3 )
mask_g = validation.global_val( u, v, (-1000, 2000), (-1000, 1000) )
invalid = mask_s2n | mask_g
```

(continues on next page)

(continued from previous page)

```
u, v = filters.replace_outliers( u, v, invalid, method='localmean',
                                max_iter=10, kernel_size=2)
x, y, u, v = scaling.uniform(x, y, u, v, scaling_factor = 1)
x, y, u, v = tools.transform_coordinates(x, y, u, v)
tools.save(x, y, u, v, invalid, 'test1.vec')
tools.display_vector_field('test1.vec', scale=75, width=0.0035);
```



## 1.4.4 Options for creating lists of images

Options:

```
    settings.pattern_a = 'image_*_a.bmp'
    settings.pattern_b = 'image_*_b.bmp'

or
    settings.pattern_a = '000*.tif'
    settings.pattern_b = '(1+2),(2+3)'

will create PIV of these pairs: 0001.tif+0002.tif, 0002.tif+0003.tif ...

    settings.pattern_a = '000*.tif'
    settings.pattern_b = '(1+3),(2+4)'

will create PIV of these pairs: 0001.tif+0003.tif, 0002.tif+0004.tif ...
or
    settings.pattern_a = '000*.tif'
    settings.pattern_b = '(1+2),(3+4)'

will create PIV of these pairs: 0001.tif+0002.tif, 0003.tif+0004.tif ...
```

## 1.5 OpenPIV masking tutorial

In this tutorial we focus on the two ways you can use image masking in OpenPIV.

**Definitions**:

1. *static mask* - an image with regions that should not be processed are marked as 1 (white color in black and white image or True) and regions that are processed are unmasked (zeros, False)
2. *dynamic mask* - every pair of images (frame A, B) are processed to find out the region that has to be masked, e.g. a fish body around which we want to analyze PIV vectors. An average mask is then applied to both frames and PIV analysis

OpenPIV uses these masks in two ways:

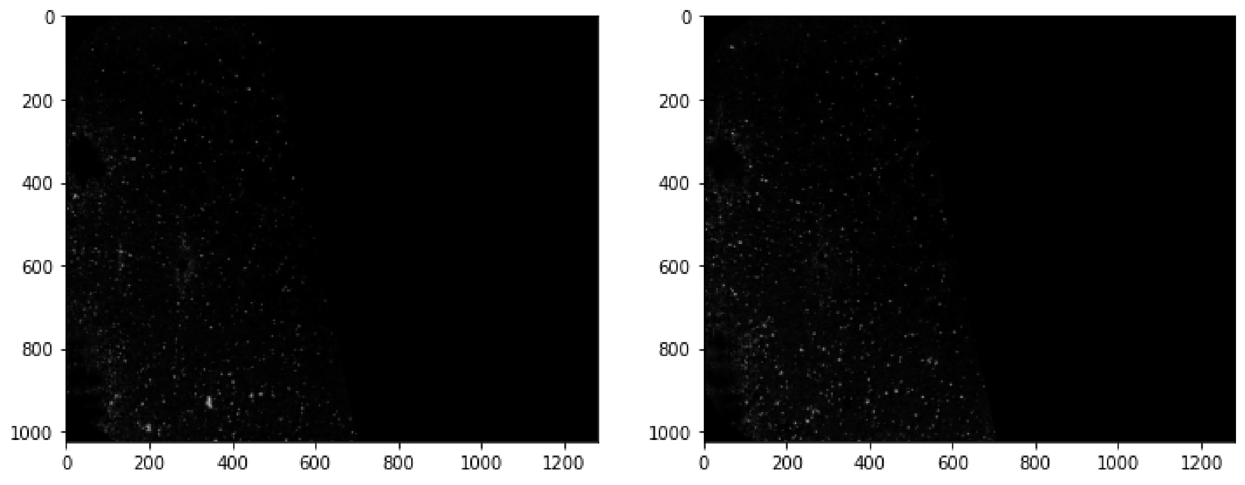
1. masked image regions are set to zero or completely black. `frame_a[image_mask] = 0`
2. PIV analysis in a completely black interrogation windows result in a zero peak and marked as invalid.
3. in addition, the image mask is converted in a set of `x,y` coordinates on a PIV grid that mark the masked region in the vector field. These `mask_coords` are propagating through the window deformation and stored with the `x,y,u,v,mask` in the ASCII result files. The vector fields `u,v` are `numpy.MaskedArray` so the masked regions are invalid and should not appear in the plot. They could be also replaced by zeros or `NaN` if needed.

[1]:
```
from openpiv import tools, pyprocess, validation, filters, scaling, preprocess
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import imageio
import importlib_resources
import pathlib
```

[2]:
```
path = importlib_resources.files('openpiv') # pathlib.Path type
```

[3]:
```
frame_a  = tools.imread( path / 'data' / 'test3' / 'pair_4_frame_0.jpg' )
frame_b  = tools.imread( path / 'data' / 'test3' / 'pair_4_frame_1.jpg' )
```

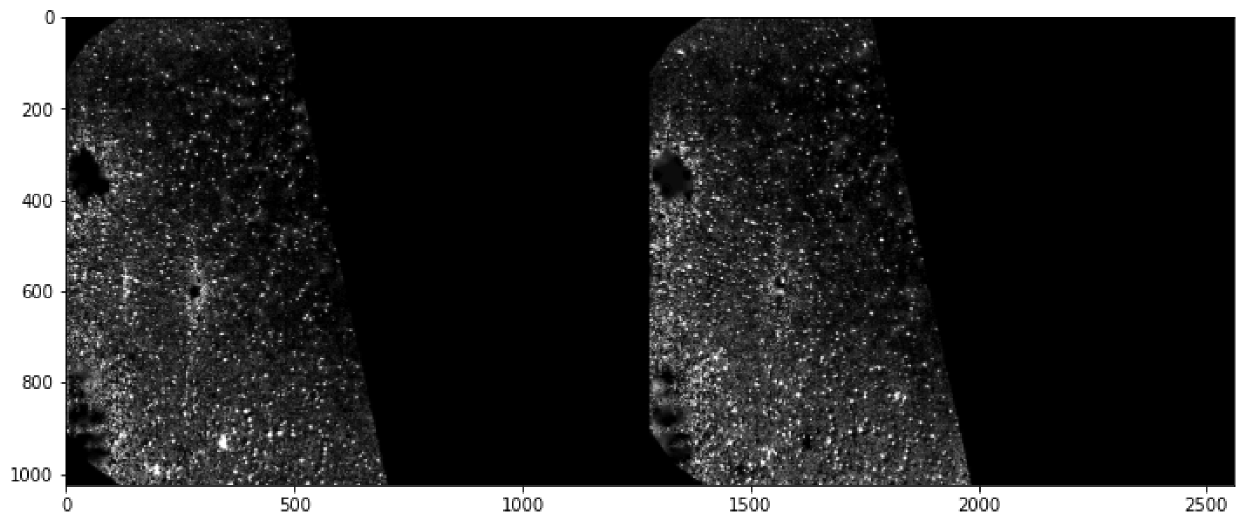
[4]:
```
fig,ax = plt.subplots(1,2,figsize=(12,10))
ax[0].imshow(frame_a,cmap='gray');
ax[1].imshow(frame_b,cmap='gray');
```



### 1.5.1 Rescale intensity or stretch histogram to get better contrast

```
[5]: frame_a = preprocess.contrast_stretch(frame_a)
     frame_b = preprocess.contrast_stretch(frame_b)
     plt.figure(figsize=(12,12))
     plt.imshow(np.c_[frame_a, frame_b], cmap='gray')
```

```
[5]: <matplotlib.image.AxesImage at 0x7fec92ebf970>
```



#### Processing

In this tutorial, we are going to use the `extended_search_area_piv` function, wich is a standard PIV cross-correlation algorithm.

This function allows the search area (`search_area_size`) in the second frame to be larger than the interrogation window in the first frame (`window_size`). Also, the search areas can overlap (`overlap`).

The `extended_search_area_piv` function will return three arrays. 1. The `u` component of the velocity vectors 2. The `v` component of the velocity vectors 3. The signal-to-noise ratio (S2N) of the cross-correlation map of each vector. The higher the S2N of a vector, the higher the probability that its magnitude and direction are correct.

```
[6]: winsize = 64 # pixels, interrogation window size in frame A
     searchsize = 64  # pixels, search area size in frame B
     overlap = 32 # pixels, 50% overlap
     dt = 1.0 # sec, time interval between the two frames

     u, v, sig2noise = pyprocess.extended_search_area_piv(
         frame_a.astype(np.int32),
         frame_b.astype(np.int32),
         window_size=winsize,
         overlap=overlap,
         dt=dt,
         search_area_size=searchsize,
         sig2noise_method='peak2peak',
     )
```

The function `get_coordinates` finds the center of each interrogation window. This will be useful later on when plotting the vector field.

[7]:
```
x, y = pyprocess.get_coordinates(
    image_size=frame_a.shape,
    search_area_size=searchsize,
    overlap=overlap,
)
```

### Post-processing

Strictly speaking, we are ready to plot the vector field. But before we do that, we can perform some convenient pos-processing.

To start, lets use the function `sig2noise_val` to get a mask indicating which vectors have a minimum amount of S2N. Vectors below a certain threshold are substituted by `NaN`. If you are not sure about which threshold value to use, try taking a look at the S2N histogram with:

`plt.hist(sig2noise.flatten())`

[8]:
```
flags = validation.sig2noise_val(
    sig2noise,
    threshold = 1.0
)
```

Another useful function is `replace_outliers`, which will find outlier vectors, and substitute them by an average of neighboring vectors. The larger the `kernel_size` the larger is the considered neighborhood. This function uses an iterative image inpainting algorithm. The amount of iterations can be chosen via `max_iter`.

[9]:
```
u, v = filters.replace_outliers(
    u, v,
    flags,
    method='localmean',
    max_iter=10,
    kernel_size=2,
)
```

Next, we are going to convert pixels to millimeters, and flip the coordinate system such that the origin becomes the bottom left corner of the image.

[10]:
```
# convert x,y to mm
# convert u,v to mm/sec

xs, ys, us, vs = scaling.uniform(
    x, y, u, v,
    scaling_factor = 96.52,  # 96.52 pixels/millimeter
)

# 0,0 shall be bottom left, positive rotation rate is counterclockwise
xs, ys, us, vs = tools.transform_coordinates(xs, ys, us, vs)
```

### Results

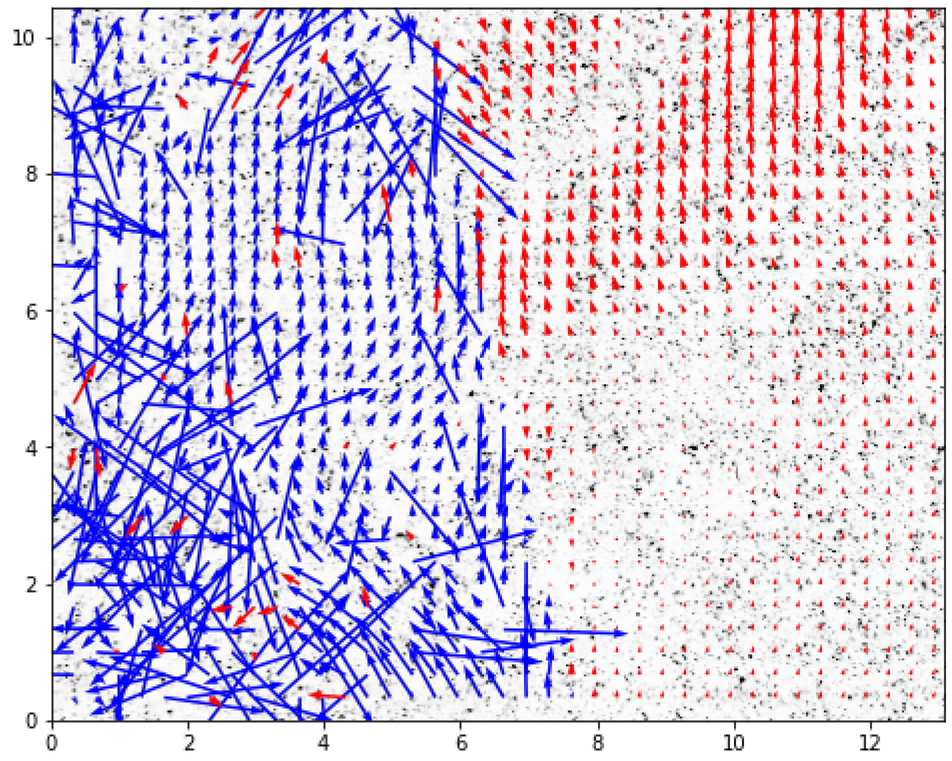
The function `save` is used to save the vector field to a ASCII tabular file. The coordinates and S2N mask are also saved.

[11]:
```
# tools.save(filename, x,y,u,v, image_grid_mask, invalid_flag)
tools.save('exp1_001.txt', xs, ys, us, vs, flags)
```

Finally, the vector field can be plotted with `display_vector_field`.

Vectors with S2N bellow the threshold are displayed in red.

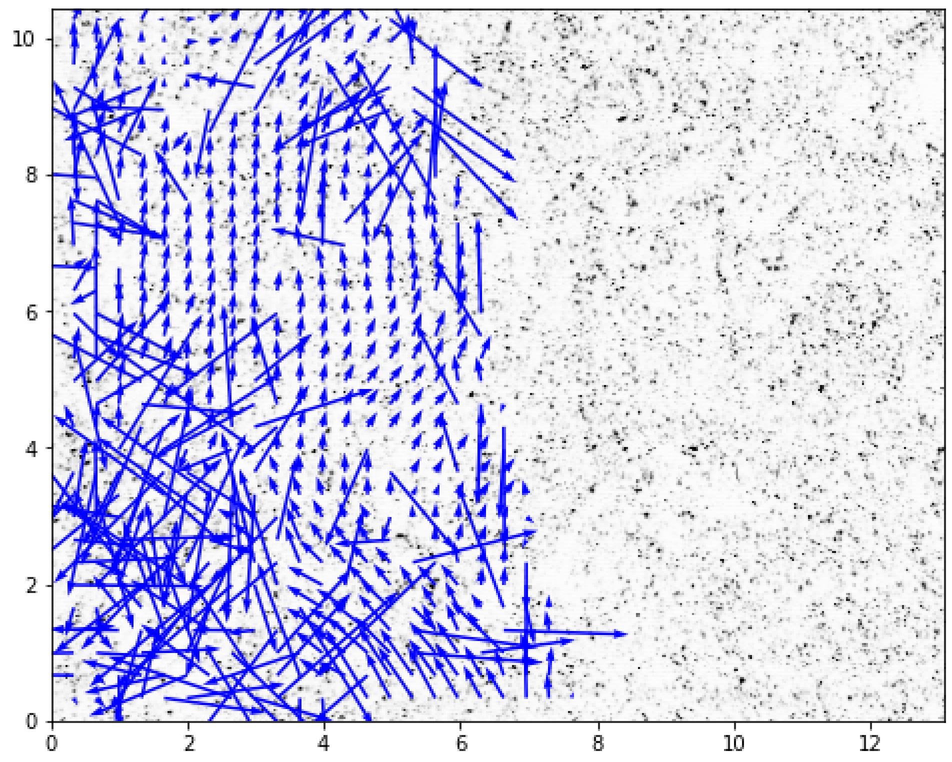
```
[12]: fig, ax = plt.subplots(figsize=(8,8))
      tools.display_vector_field(
          pathlib.Path('exp1_001.txt'),
          ax=ax, scaling_factor=96.52,
          scale=2, # scale defines here the arrow length
          width=0.0035, # width is the thickness of the arrow
          on_img=True, # overlay on the image
          image_name= str(path / 'data'/'test1'/'exp1_001_a.bmp'),
      );
```



### 1.5.2 If we do not want to show the invalid vectors

set `show_invalid = False`

```
[13]: fig, ax = plt.subplots(figsize=(8,8))
      tools.display_vector_field(
          pathlib.Path('exp1_001.txt'),
          ax=ax, scaling_factor=96.52,
          scale=2, # scale defines here the arrow length
          width=0.0035, # width is the thickness of the arrow
          on_img=True, # overlay on the image
          image_name= str(path / 'data'/'test1'/'exp1_001_a.bmp'),
          show_invalid=False,
      );
```



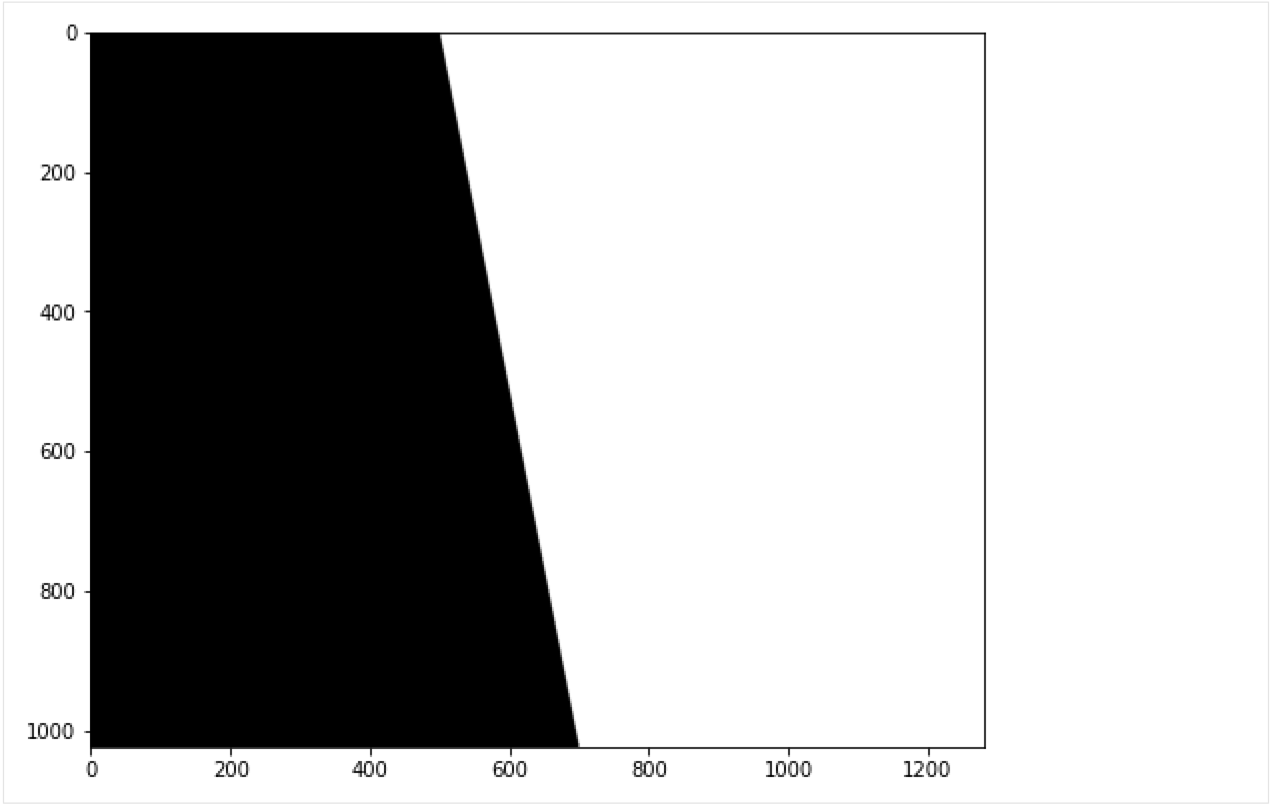
### 1.5.3 Tutorial on how to create a polygon mask

#### Start with drawing a polygon, use image coordinates

**Important** if the polygon touches borders, leave 5 pixels at least from the border

```
[14]: from skimage.draw import polygon

fig, ax = plt.subplots(figsize=(8,8))

img = 0*frame_a.copy()


# ax.imshow(img, cmap='gray')

rr, cc = polygon(
    [0,frame_a.shape[0]-1,frame_a.shape[0]-1,0],
    [500,700,frame_a.shape[1]-1,frame_a.shape[1]-1]
 )
img[rr, cc] = 1

ax.imshow(img, cmap='gray')
```

```
[14]: <matplotlib.image.AxesImage at 0x7fec92cbd2b0>
```



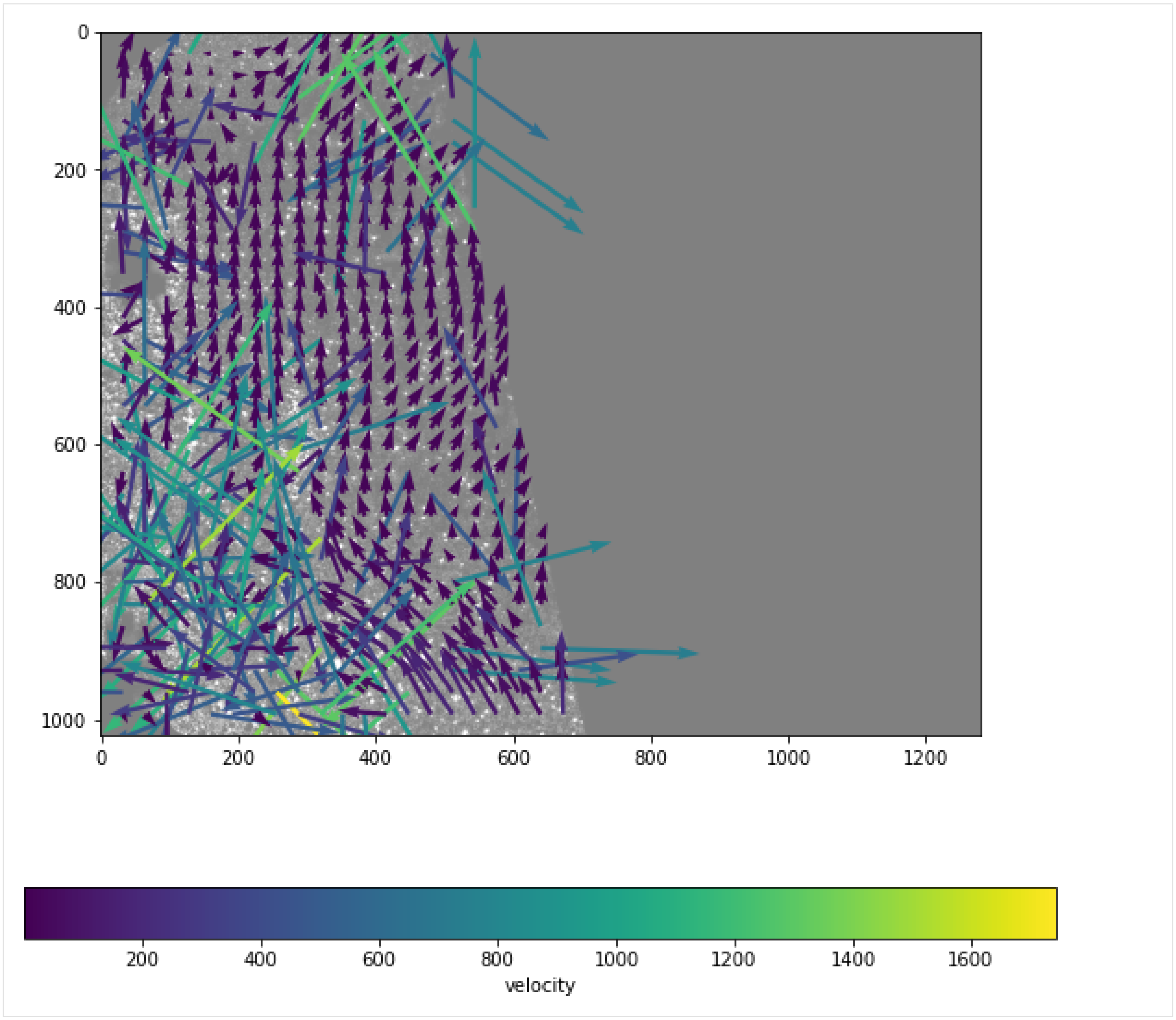
```
[15]: # convert img to a boolean mask
      img = np.where(img, True, False)
```

```
[16]: # Note that mask_coordins for polygon are in image coordinates
      # and here we need the grid coordinates, so we exchange x,y

      # grid_mask = preprocess.prepare_mask_from_polygon(x, y, np.array(mask_coords)[:,::-
      ↪1])

      grid_mask = preprocess.prepare_mask_on_grid(x,y,img)
```

Now use the grid mask to create masked arrays, like in `windef.py`

```
[17]: masked_u = np.ma.masked_array(u, mask=grid_mask)
      masked_v = np.ma.masked_array(v, mask=grid_mask)
```

```
[28]: fig, ax = plt.subplots(figsize=(10,10))
      ax.imshow(frame_a, alpha=.5,cmap='gray',origin='lower')
      Q = ax.quiver( x, y, masked_u, -masked_v, masked_u**2+masked_v**2, scale=150, width=.
      ↪005,)
      ax.invert_yaxis()
      cb = fig.colorbar(Q,orientation='horizontal')
      cb.set_label('velocity')
```



```
[ ]:
```

## 1.6 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.

### 1.6.1 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/openpiv/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.

### 1.6.2 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!

### 1.6.3 Things OpenPIV currently needs, (in order of importance)

- Good documentation (in progress)
- The implementation of advanced processing algorithms (in progress)
- Flow field filtering and validation functions
- Cython wrappers for C/C++ codes.
- A good graphical user interface (in progress)

### 1.6.4 How to test all the notebooks::

conda create -n openpiv conda activate openpiv conda install -c conda-forge openpiv conda install ipykernel python -m ipykernel install –user –name openpiv –display-name="openpiv" jupyter nbconvert –to html –ExecutePreprocessor.kernel_name=openpiv –execute *.ipynb

Then open the *openpiv/examples/notebooks* and check the HTML files. If one of those will fail, the error message will be in the command shell

### 1.6.5 If you need to install cv2::

conda install -c conda-forge opencv

## 1.7 API reference

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

### 1.7.1 The `openpiv.preprocess` module

This module contains image processing routines that improve images prior to PIV processing.

`openpiv.preprocess.`**`contrast_stretch`**(*img*, *lower_limit=2*, *upper_limit=98*)

Simple percentile-based contrast stretching

**Parameters**

- **`img`** (*`image`*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **`lower_limit`** (*`int`*) – lower percentile limit
- **`upper_limit`** (*`int`*) – upper percentile limit

**Returns** **img** – a filtered two dimensional array of the input image

**Return type** image

`openpiv.preprocess.`**`dynamic_masking`**(*image*, *method='edges'*, *filter_size=7*, *threshold=0.005*)

Dynamically masks out the objects in the PIV images

**Parameters**

- **`image`** (*`image`*) – a two dimensional array of uint16, uint8 or similar type
- **`method`** (*`string`*) – 'edges' or 'intensity': 'edges' method is used for relatively dark and sharp objects, with visible edges, on dark backgrounds, i.e. low contrast 'intensity' method is useful for smooth bright objects or dark objects or vice versa, i.e. images with high contrast between the object and the background
- **`filter_size`** (*`integer`*) – a scalar that defines the size of the Gaussian filter
- **`threshold`** (*`float`*) – a value of the threshold to segment the background from the object default value: None, replaced by sckimage.filter.threshold_otsu value

**Returns**

- **image** (*array of the same datatype as the incoming image with the*)
- *object masked out*
- *as a completely black region(s) of zeros (integers or floats).*

#### Example

frame_a = openpiv.tools.imread( 'Camera1-001.tif' ) imshow(frame_a) # original

frame_a = dynamic_masking(frame_a,method='edges',filter_size=7, threshold=0.005) imshow(frame_a) # masked

`openpiv.preprocess.`**`gen_lowpass_background`**(*img_list*, *sigma=3*, *resize=None*)

Generate a background by averaging a low pass of all images in an image list. Apply by subtracting generated background image.

**Parameters**

- **img_list** (*list*) – list of image directories
- **sigma** (*float*) – sigma of the gaussian filter
- **resize** (*int or float*) – disabled by default, normalize array and set value to user selected max pixel intensity

**Returns** **img** – a mean of all low-passed images

**Return type** image

`openpiv.preprocess.`**`gen_min_background`**(*img_list*, *resize=255*)

Generate a background by averaging the minimum intensity of all images in an image list. Apply by subtracting generated background image.

**Parameters**

- **img_list** (*list*) – list of image directories
- **resize** (*int or float*) – disabled by default, normalize array and set value to user selected max pixel intensity

**Returns** **img** – a mean of all images

**Return type** image

`openpiv.preprocess.`**`high_pass`**(*img*, *sigma=5*, *clip=False*)

Simple high pass filter

**Parameters**

- **img** (*image*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **sigma** (*float*) – sigma value of the gaussian filter

**Returns** **img** – a filtered two dimensional array of the input image

**Return type** image

`openpiv.preprocess.`**`instensity_cap`**(*img*, *std_mult=2*)

Simple intensity capping.

**Parameters**

- **img** (*image*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **std_mult** (*int*) – how strong the intensity capping is. Lower values yields a lower threshold

**Returns** **img** – a filtered two dimensional array of the input image

**Return type** image

`openpiv.preprocess.`**`intensity_clip`**(*img*, *min_val=0*, *max_val=None*, *flag='clip'*)

Simple intensity clipping

**Parameters**

- **img** (*image*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **min_val** (*int or float*) – min allowed pixel intensity
- **max_val** (*int or float*) – min allowed pixel intensity
- **flag** (*str*) – one of two methods to set invalid pixels intensities

**Returns** **img** – a filtered two dimensional array of the input image

**Return type** image

`openpiv.preprocess.`**`local_variance_normalization`**(*img*, *sigma_1=2*, *sigma_2=1*, *clip=True*)

Local variance normalization by two gaussian filters. This method is used by common commercial softwares

**Parameters**

- **`img`** (*`image`*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **`sigma_1`** (*`float`*) – sigma value of the first gaussian low pass filter
- **`sigma_2`** (*`float`*) – sigma value of the second gaussian low pass filter
- **`clip`** (*`bool`*) – set negative pixels to zero

**Returns** **img** – a filtered two dimensional array of the input image

**Return type** image

`openpiv.preprocess.`**`mask_coordinates`**(*image_mask*, *tolerance=1.5*, *min_length=10*, *plot=False*)

**Creates set of coordinates of polygons from the image mask** Inputs: mask : binary image of a mask.

[tolerance] : float - tolerance for approximate_polygons, default = 1.5

[min_length] : int - minimum length of the polygon, filters out the small polygons like noisy regions, default = 10

**Outputs:** mask_coord : list of mask coordinates in pixels

## Example

# if masks of image A and B are slightly different: image_mask = np.logical_and(image_mask_a, image_mask_b) mask_coords = mask_coordinates(image_mask)

`openpiv.preprocess.`**`normalize_array`**(*array*, *axis=None*)

Min/max normalization to [0,1].

**Parameters**

- **`array`** (*`np.ndarray`*) – array to normalize
- **`axis`** (*`int, tuple`*) – axis to find values for normalization

**Returns** **array** – normalized array

**Return type** np.ndarray

`openpiv.preprocess.`**`offset_image`**(*img*, *offset_x*, *offset_y*, *pad='zero'*)

Offset an image by padding.

**Parameters**

- **`img`** (*`image`*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **`offset_x`** (*`int`*) – offset an image by integer values. Positive values shifts the image to the right and negative values shift to the left
- **`offset_y`** (*`int`*) – offset an image by integer values. Positive values shifts the image to the top and negative values shift to the bottom

- **pad** (*str*) – pad the shift with zeros or a reflection of the shift

**Returns** **img** – a transformed two dimensional array of the input image

**Return type** image

openpiv.preprocess.**prepare_mask_from_polygon**(*x*, *y*, *mask_coords*)
Converts mask coordinates of the image mask to the grid of 1/0 on the x,y grid Inputs:

x,y : grid of x,y points mask_coords : array of coordinates in pixels of the image_mask

**Outputs:** grid of points of the mask, of the shape of x

openpiv.preprocess.**prepare_mask_on_grid**(*x: numpy.ndarray*, *y: numpy.ndarray*, *image_mask: numpy.ndarray*) → numpy.array
_summary_

**Parameters**

- **x** (*np.ndarray*) – x coordinates of vectors in pixels
- **y** (*np.ndarray*) – y coordinates of vectors in pixels
- **image_mask** (*np.ndarray*) – image of the mask, 1 or True is to be masked

**Returns** boolean array of the size of x,y with 1 where the values are masked

**Return type** np.ndarray

openpiv.preprocess.**standardize_array**(*array*, *axis=None*)
Standardize an array.

**Parameters**

- **array** (*np.ndarray*) – array to normalize
- **axis** (*int, tuple*) – axis to find values for standardization

**Returns** **array** – normalized array

**Return type** np.ndarray

openpiv.preprocess.**stretch_image**(*img*, *x_axis=0*, *y_axis=0*)
Stretch an image by interplation.

**Parameters**

- **img** (*image*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **x_axis** (*float*) – stretch the x-axis of an image where 0 == no stretching
- **y_axis** (*float*) – stretch the y-axis of an image where 0 == no stretching

**Returns** **img** – a transformed two dimensional array of the input image

**Return type** image

openpiv.preprocess.**threshold_binarize**(*img*, *threshold*, *max_val=255*)
Simple binarizing threshold

**Parameters**

- **img** (*image*) – a two dimensional array of float32 or float64, but can be uint16, uint8 or similar type
- **threshold** (*int or float*) – boundary where pixels set lower than the threshold are set to zero and values higher than the threshold are set to the maximum user selected value

- **max_val** (*int or float*) – maximum pixel value of the image

**Returns** **img** – a filtered two dimensional array of the input image

**Return type** image

## 1.7.2 The `openpiv.tools` module

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

`openpiv.tools.`**`convert_16bits_tif`**(*filename*, *save_name*)
convert 16 bits TIFF to an openpiv readable image

**Parameters**

- **filename** (*_type_*) – filename of a 16 bit TIFF
- **save_name** (*_type_*) – new image filename

`openpiv.tools.`**`display`**(*message*)
Display a message to standard output.

**Parameters** **message** (*string*) – a message to be printed

`openpiv.tools.`**`display_vector_field`**(*filename: Union[pathlib.Path, str], on_img: Optional[bool] = False, image_name: Union[pathlib.Path, str, None] = None, window_size: Optional[int] = 32, scaling_factor: Optional[float] = 1.0, ax: Optional[Any] = None, width: Optional[float] = 0.0025, show_invalid: Optional[bool] = True, **kw*)
Displays quiver plot of the data stored in the file

**Parameters**

- **filename** (*string*) – the absolute path of the text file
- **on_img** (*Bool, optional*) – if True, display the vector field on top of the image provided by image_name
- **image_name** (*string, optional*) – path to the image to plot the vector field onto when on_img is True
- **window_size** (*int, optional*) – when on_img is True, provide the interrogation window size to fit the background image to the vector field
- **scaling_factor** (*float, optional*) – when on_img is True, provide the scaling factor to scale the background image to the vector field
- **show_invalid** (*bool, show or not the invalid vectors, default is True*) –

**Key arguments** [(additional parameters, optional)] *scale*: [None | float] *width*: [None | float]

matplotlib.pyplot.quiver

### Examples

— only vector field >>> openpiv.tools.display_vector_field('./exp1_0000.txt',scale=100,

width=0.0025)

— vector field on top of image >>> openpiv.tools.display_vector_field(Path('./exp1_0000.txt'), on_img=True, image_name=Path('exp1_001_a.bmp'), window_size=32, scaling_factor=70, scale=100, width=0.0025)

`openpiv.tools.`**`display_windows_sampling`**(*x*, *y*, *window_size*, *skip=0*, *method='standard'*)

Displays a map of the interrogation points and windows

**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.
- **`window_size`** (*the interrogation window size, in pixels*) –
- **`skip`** (*the number of windows to skip on a row during display.*) – Recommended value is 0 or 1 for standard method, can be more for random method -1 to not show any window
- **`method`** (*can be only <standard> (uniform sampling and constant window size)*) – <random> (pick randomly some windows)

## Examples

```
>>> openpiv.tools.display_windows_sampling(x, y, window_size=32, skip=0, method=
↪'standard')
```

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

Read an image file into a numpy array using imageio 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** – a numpy array with grey levels

**Return type** np.ndarray

## Examples

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

`openpiv.tools.`**`imsave`**(*filename*, *arr*)

Write an image file from a numpy array using imageio imread

**Parameters**

- **`filename`** (*string*) – the absolute path of the image file that will be created
- **`arr`** (*2d np.ndarray*) – a 2d numpy array with grey levels

### Example

```
>>> image = openpiv.tools.imread( 'image.bmp' )
>>> image2 = openpiv.tools.negative(image)
>>> imsave( 'negative-image.tif', image2)
```

openpiv.tools.**mark_background**(*threshold: float*, *list_img: list*, *filename: str*) → numpy.ndarray
marks background

**Parameters**

- **threshold** (*float*) – threshold
- **list_img** (*list of images*) – _description_
- **filename** (*str*) – image filename to save the mask

**Returns** _description_

**Return type** _type_

openpiv.tools.**natural_sort**(*file_list: List[pathlib.Path]*) → List[pathlib.Path]
Creates naturally sorted list

openpiv.tools.**negative**(*image*)
Return the negative of an image

image : 2d np.ndarray of grey levels

**Returns** **(255-image)**

**Return type** 2d np.ndarray of grey levels

openpiv.tools.**rgb2gray**(*rgb: numpy.ndarray*) → numpy.ndarray
converts rgb image to gray

**Parameters** **rgb** (*_type_*) – numpy.ndarray, image size, three channels

**Returns** numpy.ndarray of the same shape, one channel

**Return type** gray

openpiv.tools.**save**(*filename: Union[pathlib.Path, str]*, *x: numpy.ndarray*, *y: numpy.ndarray*, *u: numpy.ndarray*, *v: numpy.ndarray*, *flags: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *fmt: str = '%.4e'*, *delimiter: str = '\t'*) → None
Save flow field to an ascii file.

**Parameters**

- **filename** (*string*) – the path of the file where to save the flow field
- **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.
- **flags** (*2d np.ndarray*) – a two dimensional integers array where elements corresponding to vectors: 0 - valid, 1 - invalid (, 2 - interpolated) default: None, will create all valid 0

- **mask** (*2d np.ndarray boolean, marks the image masked regions (dynamic and/or static)*) – default: None - will be all False

**fmt** [string] a format string. See documentation of numpy.savetxt for more details.

**delimiter** [string] character separating columns

### Examples

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

`openpiv.tools.`**`sorted_unique`**(*array: numpy.ndarray*) → numpy.ndarray
Creates sorted unique array

`openpiv.tools.`**`transform_coordinates`**(*x*, *y*, *u*, *v*)
Converts coordinate systems from/to the image based / physical based

Input/Output: x,y,u,v

image based is 0,0 top left, x = columns to the right, y = rows downwards and so u,v

physical or right hand one is that leads to the positive vorticity with the 0,0 origin at bottom left to be counterclockwise

## 1.7.3 The `openpiv.pyprocess` module

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

`openpiv.pyprocess.`**`correlate_windows`**(*window_a*, *window_b*, *correlation_method='fft'*, *convolve2d=<function convolve2d>*, *rfft2=<function rfft2>*, *irfft2=<function irfft2>*)
Compute correlation function between two interrogation windows. The correlation function can be computed by using the correlation theorem to speed up the computation. :param window_a: a two dimensions array for the first interrogation window :type window_a: 2d np.ndarray :param window_b: a two dimensions array for the second interrogation window :type window_b: 2d np.ndarray :param correlation_method: 'circular' - FFT based without zero-padding

'linear' - FFT based with zero-padding 'direct' - linear convolution based Default is 'fft', which is much faster.

**Parameters**

- **convolve2d** (*function*) – function used for 2d convolutions
- **rfft2** (*function*) – function used for rfft2
- **irfft2** (*function*) – function used for irfft2

**Returns**

- **corr** (*2d np.ndarray*) – a two dimensions array for the correlation function.
- *Note that due to the wish to use 2^N windows for faster FFT*
- *we use a slightly different convention for the size of the*
- *correlation map. The theory says it is M+N-1, and the*
- *'direct' method gets this size out*

- *the FFT-based method returns M+N size out, where M is the window_size*
- *and N is the search_area_size*
- *It leads to inconsistency of the output*

`openpiv.pyprocess.`**`correlation_to_displacement`**(*corr*, *n_rows*, *n_cols*, *subpixel_method='gaussian'*)

Correlation maps are converted to displacement for each interrogation window using the convention that the size of the correlation map is 2N -1 where N is the size of the largest interrogation window (in frame B) that is called search_area_size Inputs:

**corr** [3D nd.array] contains output of the fft_correlate_images

**n_rows, n_cols** [number of interrogation windows, output of the] get_field_shape

`openpiv.pyprocess.`**`extended_search_area_piv`**(*frame_a: numpy.ndarray, frame_b: numpy.ndarray, window_size: Union[int, Tuple[int, int]], overlap: Union[int, Tuple[int, int]] = (0, 0), dt: float = 1.0, search_area_size: Union[int, Tuple[int, int], None] = None, correlation_method: str = 'circular', subpixel_method: str = 'gaussian', sig2noise_method: Optional[str] = 'peak2mean', width: int = 2, normalized_correlation: bool = False, use_vectorized: bool = False*)

Standard PIV cross-correlation algorithm, with an option for extended area search that increased dynamic range. The search region in the second frame is larger than the interrogation window size in the first frame. For Cython implementation see openpiv.process.extended_search_area_piv

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].
- **`correlation_method`** (*`string`*) – one of the two methods implemented: 'circular' or 'linear', default: 'circular', it's faster, without zero-padding 'linear' requires also normalized_correlation = True (see below)
- **`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`.

- **search_area_size** (*int*) – the size of the interrogation window in the second frame, default is the same interrogation window size and it is a fallback to the simplest FFT based PIV
- **normalized_correlation** (*bool*) – if True, then the image intensity will be modified by removing the mean, dividing by the standard deviation and the correlation map will be normalized. It's slower but could be more robust

**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 != None )*) – a two dimensional array the signal to noise ratio for each window pair.

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

originally implemented in process.pyx in Cython and converted to a NumPy vectorized solution in pyprocess.py

openpiv.pyprocess.**fft_correlate_images**(*image_a: numpy.ndarray*, *image_b: numpy.ndarray*, *correlation_method: str = 'circular'*, *normalized_correlation: bool = True*, *conj: Callable = <ufunc 'conjugate'>*, *rfft2=<function rfft2>*, *irfft2=<function irfft2>*, *fftshift=<function fftshift>*) → numpy.ndarray

FFT based cross correlation of two images with multiple views of np.stride_tricks() The 2D FFT should be applied to the last two axes (-2,-1) and the zero axis is the number of the interrogation window This should also work out of the box for rectangular windows. :param image_a: and two last dimensions are interrogation windows of the first image :type image_a: 3d np.ndarray, first dimension is the number of windows, :param image_b: :type image_b: similar :param correlation_method: one of the three methods implemented: 'circular' or 'linear'

[default: 'circular].

**Parameters**

- **normalized_correlation** (*string*) – decides wetehr normalized correlation is done or not: True or False [default: True].
- **conj** (*function*) – function used for complex conjugate
- **rfft2** (*function*) – function used for rfft2
- **irfft2** (*function*) – function used for irfft2
- **fftshift** (*function*) – function used for fftshift

openpiv.pyprocess.**fft_correlate_windows**(*window_a*, *window_b*, *rfft2=<function rfft2>*, *irfft2=<function irfft2>*)

FFT based cross correlation it is a so-called linear convolution based, since we increase the size of the FFT to reduce the edge effects. This should also work out of the box for rectangular windows.

**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
- **rfft2** (*function*) – function used for rfft2
- **irfft2** (*function*) – function used for irfft2
- **from Stackoverflow** (*#*) –
- **scipy import linalg** (*from*) –
- **numpy as np** (*import*) –
- **works for rectangular windows as well** (*#*) –
- **= [[1 , 0 , 0 , 0] , [0 , -1 , 0 , 0] , [0 , 0 , 3 , 0] ,** (*x*) – [0 , 0 , 0 , 1], [0 , 0 , 0 , 1]]
- **= np.array(x,dtype=np.float)** (*x*) –
- **= [[4 , 5] , [3 , 4]]** (*y*) –
- **= np.array(y)** (*y*) –
- **("conv** (*print*) –
- **= np.array(x.shape)** (*s1*) –
- **= np.array(y.shape)** (*s2*) –
- **= s1 + s2 - 1** (*size*) –
- **= 2 ** np.ceil(np.log2(size))astype(int)** (*fsize*) –
- **= tuple([slice(0, int(sz)) for sz in size])** (*fslice*) –
- **= np.fft.fft2(x , fsize)** (*new_x*) –
- **= np.fft.fft2(y , fsize)** (*new_y*) –
- **= np.fft.ifft2(new_x*new_y)[fslice]copy()** (*result*) –
- **for my method** (*print ("fft*) –

openpiv.pyprocess.**find_all_first_peaks**(*corr*)

Find row and column indices of the first correlation peak.

**Parameters** **corr** (*np.ndarray*) – the correlation map fof the strided images (N,K,M) where N is the number of windows, KxM is the interrogation window size

**Returns**

- **index_list** (*integers, index of the peak position in (N,i,j)*)
- **peaks_max** (*amplitude of the peak*)

openpiv.pyprocess.**find_all_second_peaks**(*corr*, *width=2*)

Find row and column indices of the first correlation peak.

**Parameters**

- **`corr`** (*`np.ndarray`*) – the correlation map fof the strided images (N,K,M) where N is the number of windows, KxM is the interrogation window size
- **`width`** (*`int`*) – the half size of the region around the first correlation peak to ignore for finding the second peak

**Returns**

- **index_list** (*integers, index of the peak position in (N,i,j)*)
- **peaks_max** (*amplitude of the peak*)

`openpiv.pyprocess.`**`find_first_peak`**`(`*corr*`)`

Find row and column indices of the first correlation peak.

**Parameters** **`corr`** (*`np.ndarray`*) – the correlation map fof the strided images (N,K,M) where N is the number of windows, KxM is the interrogation window size

**Returns**

- **(i,j)** (*integers, index of the peak position*)
- **peak** (*amplitude of the peak*)

`openpiv.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`**`(`*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** – the fractional row and column indices for the sub-pixel approximation of the correlation peak. If the first peak is on the border of the correlation map or any other problem, the returned result is a tuple of NaNs.

**Return type** two elements tuple

`openpiv.pyprocess.`**`get_coordinates`**(*image_size: Tuple[int, int], search_area_size: int, overlap: int, center_on_field: bool = True*) → Tuple[numpy.ndarray, numpy.ndarray]

Compute the x, y coordinates of the centers of the interrogation windows. for the SQUARE windows only, see also get_rect_coordinates

the origin (0,0) is like in the image, top left corner positive x is an increasing column index from left to right positive y is increasing row index, from top to bottom

**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.
- **`search_area_size`** (*`int`*) – the size of the search area windows, sometimes it's equal to the interrogation window size in both frames A and B
- **`overlap`** (*`int = 0 (default is no overlap)`*) – 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.

  Coordinate system 0,0 is at the top left corner, positive x to the right, positive y from top downwards, i.e. image coordinate system

`openpiv.pyprocess.`**`get_field_shape`**(*image_size: Tuple[int, int], search_area_size: Tuple[int, int], overlap: Tuple[int, int]*) → Tuple[int, int]

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, easy to obtain using .shape
- **`search_area_size`** (*`tuple`*) – the size of the interrogation windows (if equal in frames A,B) or the search area (in frame B), the largest of the two
- **`overlap`** (*`tuple`*) – the number of pixel by which two adjacent interrogation windows overlap.

**Returns** **field_shape** – the shape of the resulting flow field

**Return type** 2-element tuple

`openpiv.pyprocess.`**`get_rect_coordinates`**(*image_size: Tuple[int, int], window_size: Union[int, Tuple[int, int]], overlap: Union[int, Tuple[int, int]], center_on_field: bool = False*)

Rectangular grid version of get_coordinates.

`openpiv.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.`**`nextpower2`**`(`*i*`)`

Find 2^n that is equal to or greater than.

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

**Normalize interrogation window or strided image of many windows,** by removing the mean intensity value per window and clipping the negative values to zero

**Parameters** **`window`** (*`2d np.ndarray`*) – the interrogation window array

**Returns** **window** – the interrogation window array, with mean value equal to zero and intensity normalized to -1 +1 and clipped if some pixels are extra low/high

**Return type** 2d np.ndarray

`openpiv.pyprocess.`**`sig2noise_ratio`**`(`*correlation: numpy.ndarray*, *sig2noise_method: str = 'peak2peak'*, *width: int = 2*`)` → numpy.ndarray

Computes the signal to noise ratio from the correlation map.

The signal to noise ratio is computed from the correlation map with one of two available method. It is a measure of the quality of the matching between to interrogation windows.

**Parameters**

- **`corr`** (*`3d np.ndarray`*) – the correlation maps of the image pair, concatenated along 0th axis
- **`sig2noise_method`** (*`string`*) – the method for evaluating the signal to noise ratio value from the correlation map. Can be *peak2peak*, *peak2mean* or None if no evaluation should be made.
- **`width`** (*`int, optional`*) – 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** **sig2noise** – the signal to noise ratios from the correlation maps.

**Return type** np.array

`openpiv.pyprocess.`**`sliding_window_array`**`(`*image: numpy.ndarray*, *window_size: Tuple[int, int] = (64, 64)*, *overlap: Tuple[int, int] = (32, 32)*`)` → numpy.ndarray

This version does not use numpy as_strided and is much more memory efficient. 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.`**`vectorized_correlation_to_displacements`**`(`*corr: numpy.ndarray*, *n_rows: Optional[int] = None*, *n_cols: Optional[int] = None*, *subpixel_method: str = 'gaussian'*, *eps: float = 1e-07*`)`

Correlation maps are converted to displacement for each interrogation window using the convention that the size of the correlation map is 2N -1 where N is the size of the largest interrogation window (in frame B) that is called search_area_size

**Parameters**

- **corr** (*3D nd.array*) – contains output of the fft_correlate_images
- **n_cols** (*n_rows,*) – number of interrogation windows, output of the get_field_shape
- **mask_width** (*int*) – distance, in pixels, from the interrogation window in which correlation peaks would be flagged as invalid

**Returns** **u, v** – 2d array of displacements in pixels/dt

**Return type** 2D nd.array

openpiv.pyprocess.**vectorized_sig2noise_ratio**(*correlation*, *sig2noise_method='peak2peak'*, *width=2*)

Computes the signal to noise ratio from the correlation map in a mostly vectorized approach, thus much faster.

The signal to noise ratio is computed from the correlation map with one of two available method. It is a measure of the quality of the matching between to interrogation windows.

**Parameters**

- **corr** (*3d np.ndarray*) – the correlation maps of the image pair, concatenated along 0th axis
- **sig2noise_method** (*string*) – the method for evaluating the signal to noise ratio value from the correlation map. Can be *peak2peak*, *peak2mean* or None if no evaluation should be made.
- **width** (*int, optional*) – 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** **sig2noise** – the signal to noise ratios from the correlation maps.

**Return type** np.array

### 1.7.4 The `openpiv.process` module

### 1.7.5 The `openpiv.lib` module

openpiv.lib.**replace_nans**(*array*, *max_iter*, *tol*, *kernel_size=2*, *method='disk'*)

**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 depend on the method type. See Methods below.
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.

Methods:

**localmean - A square kernel where all elements have the same value,** weights are equal to n/( (2*kernel_size+1)**2 -1 ), where n is the number of non-NaN elements.

**disk - A circular kernel where all elements have the same value,**

**kernel is calculated by::**

**if ((S-i)**2 + (S-j)**2)**0.5 <= S:** kernel[i,j] = 1.0

**else:** kernel[i,j] = 0.0

where S is the kernel radius.

**distance - A circular inverse distance kernel where elements are** weighted proportional to their distance away from the center of the kernel, elements farther away have less weight. Elements outside the specified radius are set to 0.0 as in 'disk', the remaining of the weights are calculated as:

```
maxDist = ((S)**2 + (S)**2)**0.5
kernel[i,j] = -1*(((S-i)**2 + (S-j)**2)**0.5 - maxDist)
```

where S is the kernel radius.

**Parameters**

- **array** (*2d or 3d np.ndarray*) – an array containing NaN elements that have to be replaced if array is a masked array (numpy.ma.MaskedArray), then the mask is reapplied after the replacement
- **max_iter** (*int*) – the number of iterations
- **tol** (*float*) – On each iteration check if the mean square difference between values of replaced elements is below a certain tolerance *tol*
- **kernel_size** (*int*) – the size of the kernel, default is 1
- **method** (*str*) – the method used to replace invalid values. Valid options are *localmean*, *disk*, and *distance*.

**Returns** **filled** – a copy of the input array, where NaN elements have been replaced.

**Return type** 2d or 3d np.ndarray

### 1.7.6 The `openpiv.filters` module

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

`openpiv.filters.`**`gaussian`**(*u: numpy.ndarray*, *v: numpy.ndarray*, *half_width: int = 1*) → Tuple[numpy.ndarray, numpy.ndarray]

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
- **half_width** (*int*) – the half width of the kernel. Kernel has shape 2*half_width+1, default = 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`**(*sigma: float*, *truncate: float = 4.0*) → numpy.ndarray

Return Gaussian that truncates at the given number of standard deviations.

`openpiv.filters.`**`replace_outliers`**(*u: numpy.ndarray*, *v: numpy.ndarray*, *flags: numpy.ndarray*, *w: Optional[numpy.ndarray] = None*, *method: str = 'localmean'*, *max_iter: int = 5*, *tol: float = 0.001*, *kernel_size: int = 1*) → Tuple[numpy.ndarray, ...]

**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 or 3d np.ndarray*) – the u velocity component field
- **v** (*2d or 3d np.ndarray*) – the v velocity component field
- **w** (*2d or 3d np.ndarray*) – the w velocity component field
- **flags** (*2d array of positions with invalid vectors*) –
- **grid_mask** (*2d array of positions masked by the user*) –
- **max_iter** (*int*) – the number of iterations
- **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 or 3d np.ndarray*) – the smoothed u velocity component field, where invalid vectors have been replaced
- **vf** (*2d or 3d np.ndarray*) – the smoothed v velocity component field, where invalid vectors have been replaced
- **wf** (*2d or 3d np.ndarray*) – the smoothed w velocity component field, where invalid vectors have been replaced

### 1.7.7 The `openpiv.validation` module

A module for spurious vector detection.

`openpiv.validation.`**`global_std`**(*u: numpy.ndarray*, *v: numpy.ndarray*, *std_threshold: int = 5*) → numpy.ndarray

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 masked np.ndarray*) – a two dimensional array containing the u velocity component.
- **v** (*2d masked 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** **flag** – a boolean array. True elements corresponds to outliers.

**Return type** boolean 2d np.ndarray

`openpiv.validation.`**`global_val`**(*u: numpy.ndarray, v: numpy.ndarray, u_thresholds: Tuple[int, int], v_thresholds: Tuple[int, int]*) → numpy.ndarray

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** **flag** – a boolean array. True elements corresponds to outliers.

**Return type** boolean 2d np.ndarray

`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.

**The image masked areas (obstacles, reflections) are marked as masked array:** u = np.ma.masked(u, flag = image_mask)

and it should not be replaced by the local median, but remain masked.

**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** **flag** – a boolean array. True elements corresponds to outliers.

**Return type** boolean 2d np.ndarray

`openpiv.validation.`**`sig2noise_val`**(*s2n: numpy.ndarray, threshold: float = 1.0*) → numpy.ndarray

Marks spurious vectors if signal to noise ratio is below a specified threshold.

**Parameters**

- **`u`** (*`2d or 3d np.ndarray`*) – a two or three dimensional array containing the u velocity component.

- **v** (*2d or 3d np.ndarray*) – a two or three dimensional array containing the v velocity component.
- **s2n** (*2d np.ndarray*) – a two or three dimensional array containing the value of the signal to noise ratio from cross-correlation function.
- **w** (*2d or 3d np.ndarray*) – a two or three dimensional array containing the w (in z-direction) velocity component.
- **threshold** (*float*) – the signal to noise ratio threshold value.

**Returns** **flag** – a boolean array. True elements corresponds to outliers.

**Return type** boolean 2d np.ndarray

### References

R. D. Keane and R. J. Adrian, Measurement Science & Technology, 1990,

1, 1202-1215.

openpiv.validation.**typical_validation**(*u: numpy.ndarray*, *v: numpy.ndarray*, *s2n: numpy.ndarray*, *settings: PIVSettings*) → numpy.ndarray

validation using gloabl limits and std and local median,

with a special option of 'no_std' for the case of completely uniform shift, e.g. in tests.

see windef.PIVSettings() for the parameters:

**MinMaxU** [two elements tuple] sets the limits of the u displacment component Used for validation.

**MinMaxV** [two elements tuple] sets the limits of the v displacment component Used for validation.

**std_threshold** [float] sets the threshold for the std validation

**median_threshold** [float] sets the threshold for the median validation

## 1.7.8 The `openpiv.scaling` module

Scaling utilities

openpiv.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

**Returns**

- **x** (*2d np.ndarray*)
- **y** (*2d np.ndarray*)
- **u** (*2d np.ndarray*)

- **v** (*2d np.ndarray*)

## 1.8 Frequently Asked Questions about PIV parameters

1. Can you please elaborate on the `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 sclt.

`sclt` is a shortcut for _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 **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: *scaling = sclt = 50 cm/pixels / 0.5 = 100 [cm/seconds/pixels]*

For example, if the vector is 10 pixels, then the result will be *100 * 10 = 1000 cm/s*

2. Whats the purpose of the local and global filtering?

**global filtering** supposingly removes the obvious **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.

**local filtering** is performed on small neighborhoods of vectors, e.g. 3 x 3 or 5 x 5, in order to find **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: **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?

*ffta=fft2(a2,Nfft,Nfft); fftb=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 **b2** is rotated before taking the correlation. *b2 = b2(end:-1:1,end:-1:1);*

Without rotation the result will be convolution, not correlation. The definition is **ifft(fft(a)*fft(conj(b)))**. 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

# Index

## C

contrast_stretch() (*in module openpiv.preprocess*), 30
convert_16bits_tif() (*in module openpiv.tools*), 34
correlate_windows() (*in module openpiv.pyprocess*), 37
correlation_to_displacement() (*in module openpiv.pyprocess*), 38

## D

display() (*in module openpiv.tools*), 34
display_vector_field() (*in module openpiv.tools*), 34
display_windows_sampling() (*in module openpiv.tools*), 35
dynamic_masking() (*in module openpiv.preprocess*), 30

## E

extended_search_area_piv() (*in module openpiv.pyprocess*), 38

## F

fft_correlate_images() (*in module openpiv.pyprocess*), 39
fft_correlate_windows() (*in module openpiv.pyprocess*), 39
find_all_first_peaks() (*in module openpiv.pyprocess*), 40
find_all_second_peaks() (*in module openpiv.pyprocess*), 40
find_first_peak() (*in module openpiv.pyprocess*), 41
find_second_peak() (*in module openpiv.pyprocess*), 41
find_subpixel_peak_position() (*in module openpiv.pyprocess*), 41

## G

gaussian() (*in module openpiv.filters*), 45
gaussian_kernel() (*in module openpiv.filters*), 45
gen_lowpass_background() (*in module openpiv.preprocess*), 30
gen_min_background() (*in module openpiv.preprocess*), 31
get_coordinates() (*in module openpiv.pyprocess*), 42
get_field_shape() (*in module openpiv.pyprocess*), 42
get_rect_coordinates() (*in module openpiv.pyprocess*), 42
global_std() (*in module openpiv.validation*), 46
global_val() (*in module openpiv.validation*), 47

## H

high_pass() (*in module openpiv.preprocess*), 31

## I

imread() (*in module openpiv.tools*), 35
imsave() (*in module openpiv.tools*), 35
instensity_cap() (*in module openpiv.preprocess*), 31
intensity_clip() (*in module openpiv.preprocess*), 31

## L

local_median_val() (*in module openpiv.validation*), 47
local_variance_normalization() (*in module openpiv.preprocess*), 32

## M

mark_background() (*in module openpiv.tools*), 36
mask_coordinates() (*in module openpiv.preprocess*), 32
moving_window_array() (*in module openpiv.pyprocess*), 42

## N

`natural_sort()` (*in module openpiv.tools*), 36

`negative()` (*in module openpiv.tools*), 36

`nextpower2()` (*in module openpiv.pyprocess*), 43

`normalize_array()` (*in module openpiv.preprocess*), 32

`normalize_intensity()` (*in module openpiv.pyprocess*), 43

## O

`offset_image()` (*in module openpiv.preprocess*), 32

`openpiv.filters` (*module*), 45

`openpiv.lib` (*module*), 44

`openpiv.preprocess` (*module*), 30

`openpiv.pyprocess` (*module*), 37

`openpiv.scaling` (*module*), 48

`openpiv.tools` (*module*), 34

`openpiv.validation` (*module*), 46

## P

`prepare_mask_from_polygon()` (*in module openpiv.preprocess*), 33

`prepare_mask_on_grid()` (*in module openpiv.preprocess*), 33

## R

`replace_nans()` (*in module openpiv.lib*), 44

`replace_outliers()` (*in module openpiv.filters*), 45

`rgb2gray()` (*in module openpiv.tools*), 36

## S

`save()` (*in module openpiv.tools*), 36

`sig2noise_ratio()` (*in module openpiv.pyprocess*), 43

`sig2noise_val()` (*in module openpiv.validation*), 47

`sliding_window_array()` (*in module openpiv.pyprocess*), 43

`sorted_unique()` (*in module openpiv.tools*), 37

`standardize_array()` (*in module openpiv.preprocess*), 33

`stretch_image()` (*in module openpiv.preprocess*), 33

## T

`threshold_binarize()` (*in module openpiv.preprocess*), 33

`transform_coordinates()` (*in module openpiv.tools*), 37

`typical_validation()` (*in module openpiv.validation*), 48

## U

`uniform()` (*in module openpiv.scaling*), 48

## V

`vectorized_correlation_to_displacements()` (*in module openpiv.pyprocess*), 43

`vectorized_sig2noise_ratio()` (*in module openpiv.pyprocess*), 44