Cellocity

Release 1.0.2

Jens Eriksson

Aug 13, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION TO CELLOCITY

1.1 A 30 second pitch

Cellocity is a bioimage analysis tool for quantifying confluent cell layer dynamics. The main advantages of Cellocity is its ability to work on unlabeled Brightfield time lapse microscopy data, and to both quantify and visualize abstract optical flow analyses to the user.

1.2 Installing Cellocity

Cellocity is available on the Python package index and the latest release can be installed using pip:

```
pip install cellocity
```

You can also clone the Github repository if you are interested in getting the current development version of Cellocity:

```
git clone https://github.com/Oftatkofta/cellocity.git cellocity
cd cellocity
pip install -e .
```

Cellocity requires Python (>3.7), tifffile (2020.5.5), python-OpenCV (4.2.0.34), OpenPIV (0.21.3), Numpy (1.18.4), Pandas (1.0.3) to function correctly. Additionally, you need Matplotlib (3.2.1) and Seaborn (0.10.1) in order to visualize the validation output. If you perform a pip install from PyPi, all dependencies will be installed automatically.

1.3 Citing Cellocity

If you have found Cellocity useful in your project and want to cite it, you can use our JOSS publication [![DOI](https://joss.theoj.org/papers/10.21105/joss.02818/status.svg){}](https://doi.org/10.21105/joss.02818)

1.4 Cellocity development history

Cellocity has been developed over multiple years and several projects. The nucleus was developed in Stig Ove Bøe's research group at Oslo University Hospital and at the Nanoscopy Gaustad imaging core facility at the University of Oslo. Many of Cellocity's core algorithm implementations and methods, such as the 5-sigma correlation length analysis, were presented in a Nature Communications publication in 2018 [3].

The framework is currently being used and further developed as the analysis backbone for studies of microbial interactions with the gut epithelium in the Sellin Laboratory at Uppsala University.

1.5 Cellocity backbone

Cellocity is built on top of Christoph Gohlke's Tifffile library and uses the `Tifffile` object to read input and write output files. Cellocity also relies heavily on OpenCV [1] and OpenPIV [5] for optical flow analysis and output visualizations. NumPy [6] is used internally for image data manipulation in the form of `numpy.ndarrays`, and matplotlib is used to generate output plots [2].

1.6 Cellocity architecture

The core element in Cellocity is the `Channel` object, which represents one Z-plane of one time lapse image channel. `Channel` objects also handle image pre-processing, such as temporal or spatial median filtering. `Channel` objects are given as input to `Analyzer` objects, which perform specific analyses on the data. `Analyzer` objects can then, in turn, be given to `Analysis` objects, which take care of performing further analyses, such as calculating the alignment index, instantaneous order parameter (ψ), and correlation length.

1.7 Exam of al- go- rithms and vec- tor field quan- tifi- ca- tions im- ple- mented

Instantaneous Order

ψ = 1 cor- re- sponds to a per- fectly uni- form ve- loc- ity field, where all the cells move in the same direction and with the same speed, while $\psi \approx 0$ is expected for a randomly oriented velocity field. See [4] for details.

Alignment Index

The Align- ment In- dex de- scribes

how well each vector in a vector field aligns with the average velocity vector. See [4] for further details.

5-σ correlation lengt

5-σ correlation length is a way to measure the correlation length in large vector fields. It finds the average distance at which the direction of velocities are no longer significantly different at a level of 5 standard deviations (σ). The algorithm was originally presented and utilized in [3]. A more detailed description can be found in the *Developer Information*.

1.8 Exam

Simple file loading example:

```
from
↪cellocity.
↪channel
↪import
↪Channel
from
↪tifffile
↪import
↪Tiffile
```

(continues on next page)

(continued from previous page)

```
tif␣
↪=␣
↪Tifffile(m
channel_
↪1␣
↪=␣
↪Channel(0,
↪␣
↪tif,
↪␣
↪
↪"channel␣
↪name
↪")␣
↪
↪#0-
↪indexed␣
↪channels,
↪␣
↪meaning␣
↪ch1␣
↪in ImageJ
```

Simple pre-processing ex-am-ple:

```
from␣
↪cellocity.
↪channel␣
↪import␣
↪MedianChar


↪#Trim␣
↪Channel␣
↪to␣
↪frame␣
↪2-
↪40
channel_
↪1.
↪trim(2,
↪␣
↪41)

↪#3-
↪frame␣
↪gliding␣
↪temporal␣
↪median␣
↪projectio
↪by␣
↪default
```

(continues on next page)

(continued from previous page)

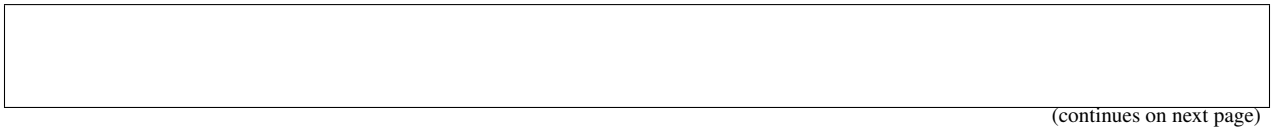
```
channel_
↪1_
↪median␣
↪=␣
↪MedianChar
↪1)
```

Simple optical flow calculation example:

```
from␣
↪cellocity.
↪analysis␣
↪import␣
↪FarenbackA

flow_
↪Ch1␣
↪=␣
↪FarenbackA
↪1_
↪median,
↪␣
↪
↪"um/
↪min
↪")
flow_
↪Ch1.
↪doFarenbac
```

Simple analysis data readout example:



```
from␣
↪cellocity.
↪analysis␣
↪import␣
↪FlowSpeedA
```

(continues on next page)

(continued from previous page)

```
 ␣
 ↪␣
 ↪␣
 ↪␣
 ↪speed_
 ↪analysis_
 ↪Ch1␣
 ↪=␣
 ↪FlowSpeedA
 ↪Ch1)

 ␣
 ↪␣
 ↪␣
 ↪␣
 ↪speed_
 ↪analysis_
 ↪Ch1.
 ↪calculateA

 ␣
 ↪␣
 ↪␣
 ↪␣
 ↪speed_
 ↪analysis_
 ↪Ch1.
 ↪saveCVS(
 ↪"/
 ↪path/
 ↪to/
 ↪savefolder
 ↪")
```

For more detailed examples please check out the tutorial section.

1.9 Supp

If something is unclear or if you are in need of support, please contact the developer by creating a new support issue.

1.10 Refe

CHAPTER

TWO

CELLOCITY TUTORIAL

2.1 Step-by-step guide

This tutorial will show you how to:

1. Load a file and create a *cellocity.channel.Channel* object.
2. Preprocess the `Channel` object.
3. Prepare for analysis by creating an *cellocity.analysis.Analyzer* object from the `Channel` object.
4. Extract data by creating an *cellocity.analysis.Analysis* object.

2.1.1 Load a file and create a `Channel` object

```
from cellocity.channel import Channel
import tifffile

my_filename = "2_channel_micromanager_timelapse.ome.tif"
chToAnalyze = 0  # 0-based indexing of channels

#safely load file
with tifffile.TiffFile(my_filename, multifile=False) as tif:

    #strips ome.tif from filename
    label = my_filename.split(".")[0]
    channelName = label + "_Ch" + str(chToAnalyze + 1)
    channel_0 = Channel(chToAnalyze, tif, name=channelName)
```

Warning: Cellocity assumes that it can hold all Channel data in RAM.

A `Tifffile` does not load all its image data into RAM when created, however upon accessing data during `Channel` creation some of it will be cached, thus increasing its size somewhat. `Channel` objects store all image data in RAM and can get quite hefty for long time lapses.

2.1.2 Preprocess `Channel` object

First, we will check if the frame interval stated in the metadata is in agreement with the time stamps of the individual frames in the channel (within 1%). This is done with the `Channel.doFrameIntervalSanityCheck(maxDiff=0.01)` method. If there is an discrepancy between the actual frame intervals and the intended, if can be fixed by calling the `Channel.fixFrameInterval()` method, which overwrites the intended frame interval with the actual average frame interval.

```
if not channel_0.doFrameIntervalSanityCheck():
        channel_0.fixFrameInterval()
```

Note: Checking and fixing the frame interval is currently only possible on MicroManager ome.tif files. Individual frame timestamps are lost when saving .tif files in ImageJ.

`Channel` objects have convenient preprocessing methods, such as trimming frames and temporal median filtering. Let's start by trimming our newly created channel to frames 10-60, meaning we discard frames 0-9 and from frame 60 onward to the end.

```
#Trim channel to include frame 10-59
channel_0.trim(10,60)
```

Now let's employ a temporal median filter, meaning we do a median filtering over time. This will have the effect of filtering out fast moving free-floating debris, thus greatly reducing the noise in the final analysis. This is done by creating a child *`cellocity.channel.MedianChannel`* object. Median filtering can be done with a gliding window (default), or by binning the frames. `MedianChannel` takes care of properly recalculating frame intervals in either case. The default frame sampling interval is 3.

```
from cellocity.channel import MedianChannel

gliding_median_channel_0 = MedianChannel(channel_0)

binned_4frame_median_channel_0 = MedianChannel(channel_0,
                                               doGlidingProjection=False,
                                               frameSamplingInterval=4)
```

`MedianChannel` objects can also be created by calling the `.getTemporalMedianChannel()` method on a `Channel`. The following code gives identical results to the above example:

```
arguments ={doGlidingProjection = True,
                frameSamplingInterval=3,
                startFrame=0,
                stopFrame=None
                }
gliding_median_channel_0 = channel_0.getTemporalMedianChannel(arguments)

arguments = {doGlidingProjection = False,
                frameSamplingInterval=4,
                startFrame=0,
                stopFrame=None}
binned_4frame_median_channel_0 = channel_0.getTemporalMedianChannel(arguments)
```

2.1.3 Prepare for Analysis by creating an `Analyzer` object

Now let's perform an optical flow analysis of our preprocessed `Channel`. This is done by instantiating an `Analyzer` object with a `Channel` as argument. In this case we will perform an optical flow analysis using the Farenback flow analysis from OpenCV. This is handled by a `FarenbackAnalyzer`, which is a specific subtype `FlowAnalyzer` of `Analyzer`.

`FarenbackAnalyzer` takes two arguments, one `Channel` and one **unit**. **unit** is a string indicating the unit that we want the output to be in. Currently only "um/s", "um/min", and "um/h" are implemented. Cellocity handles all unit conversions automatically in the background.

```
from cellocity.analysis import FarenbackAnalyzer

fb_analyzer_ch0 = FarenbackAnalyzer(channel = gliding_median_channel_0, unit = "um/h")
fb_analyzer_ch0.doFarenbackFlow()
```

Note: Quite a lot of effort has gone in to selecting sensible default parameters that work well for microscopy data for the `FlowAnalyzer` objects `FarenbackAnalyzer` and `OpenPivAnalyzer`, as is demonstrated in the *Validation of the Cellocity Software* section.

2.1.4 Extract data by creating an `Analysis` object.

Great, now we have calculated the optical flow of channel_0 with the default parameters. Now its time to extract data. This is done by creating `Analysis` objects. In our case we want to analyse the flow speeds of our channel. To do this we can utilize the `FlowSpeedAnalysis` class, which works on `FlowAnalyzer` objects.

```
from cellocity.analysis import FlowSpeedAnalysis

speed_analysis_ch0 = FlowSpeedAnalysis(fb_analyzer_ch0)
speed_analysis_ch0.calculateSpeeds()
speed_analysis_ch0.calculateAverageSpeeds()
```

When speeds have been calculated the results can be stored either as a 32-bit tif, where pixel values represent flow speeds in the location of the pixel, or the average speed of each frame can be saved as a .csv file for further processing.

```
from pathlib import Path

savepath = Path("path/to/save/folder")

speed_analysis_ch0.saveArrayAsTif(outdir=savepath):
speed_analysis_ch0.saveCSV(outdir=savepath, fname="mySpeeds.csv", tunit="s")
```

That's it! If you want more detailed information, please check the *The Cellocity API Reference* , the *Validation of the Cellocity Software* contains more examples of different `Analysis` objects in use, and the *Developer Information* contains information on how to submit a bug report.

CHAPTER

THREE

VALIDATION OF THE CELLOCITY SOFTWARE

3.1 Validation dataset

In order to validate the underlying analyzers in Cellocity we have generated a "ground truth", real-world microscopy dataset. The dataset was generated by translating and imaging, with DIC contrast, a fixed monolayer of primary gut epithelium on a high precision linear microscope stage, using a wide selection of magnifications. 10 images were acquired with the stage translated 1 μm in either the X, Y or both directions simultaneously between frames. Images were acquired on a Nikon Eclipse Ti-2 microscope, equipped with a Photometrics Prime 95B camera (1608x1608, 11 um pixel size). The general structure of the dataset is outlined in the table below.

Objective	Tube lens	Total magnification	Pixel Size (um)	X translation (um)	Y translation (um)	Filename
Nikon 10X/0.45 Air Pln.Apo.Lmbd	1X	10X	1.1235	0	1	fixed_monolayer_DIC_10X_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 10X/0.45 Air Pln.Apo.Lmbd	1X	10X	1.1235	1	0	fixed_monolayer_DIC_10X_dX-1um_dY-0um_1_MMStack.ome.tif
Nikon 10X/0.45 Air Pln.Apo.Lmbd	1X	10X	1.1235	1	1	fixed_monolayer_DIC_10X_dX-1um_dY-1um_1_MMStack.ome.tif
Nikon 10X/0.45 Air Pln.Apo.Lmbd	1.5X	15X	0.749	0	1	fixed_monolayer_DIC_15X_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 10X/0.45 Air Pln.Apo.Lmbd	1.5X	15X	0.749	1	0	fixed_monolayer_DIC_15X_dX-1um_dY-0um_1_MMStack.ome.tif
Nikon 10X/0.45 Air Pln.Apo.Lmbd	1.5X	15X	0.749	1	1	fixed_monolayer_DIC_15X_dX-1um_dY-1um_1_MMStack.ome.tif
Nikon 40X/0.6 Air S.Pln.Fl.	1X	40X	0.286	0	1	fixed_monolayer_DIC_40X_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 40X/0.6 Air S.Pln.Fl.	1X	40X	0.286	1	0	fixed_monolayer_DIC_40X_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 40X/0.6 Air S.Pln.Fl.	1X	40X	0.286	1	1	fixed_monolayer_DIC_40X_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 40X/0.6 Air S.Pln.Fl.	1.5X	60X	0.191	0	1	fixed_monolayer_DIC_60Xopt_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 40X/0.6 Air S.Pln.Fl.	1.5X	60X	0.191	1	0	fixed_monolayer_DIC_60Xopt_dX-1um_dY-0um_1_MMStack.ome.tif
Nikon 40X/0.6 Air S.Pln.Fl.	1.5X	60X	0.191	1	1	fixed_monolayer_DIC_60Xopt_dX-1um_dY-1um_1_MMStack.ome.tif
Nikon 60X/0.7 Air S.Pln.Fl.	1X	60X	0.125	0	1	fixed_monolayer_DIC_60X_dX-0um_dY-1um_1_MMStack.ome.tif
Nikon 60X/0.7 Air S.Pln.Fl.	1X	60X	0.125	1	0	fixed_monolayer_DIC_60X_dX-1um_dY-0um_1_MMStack.ome.tif
Nikon 60X/0.7 Air S.Pln.Fl.	1X	60X	0.125	1	1	fixed_monolayer_DIC_60X_dX-1um_dY-1um_1_MMStack.ome.tif

This dataset allowed us compare the "golden standard" of cell layer dynamics analysis, Particle Image Velocimetry (PIV) analysis, with the less frequently used Optical Flow analysis. Our conclusion mirror what was found in[1], which

[1] Dhruv K. Vig, Alex E. Hamby and Charles W. Wolgemuth. On the Quantification of Cellular Velocity Fields. *Biophysical Journal*, 110:1469-

is that Optical Flow analysis is indeed superior to PIV analysis, both with respect to accuracy and efficiency. The following section will substantiate this finding. All analyses were run on a early 2020 Dell XPS 15 7590 laptop, running Windows 10.

3.2 Downloading the validation dataset

The dataset has been deposited into the BioStudies database with the accession number S-BSST461 and can be downloaded from there.

3.3 Performing the validation on your local installation

All the validation figures can be re-generated on your local install by running the following code:

```
from cellocity import validation
from pathlib import Path

inpath = Path("path/to/S-BSST641/")
outpath = Path("path/to/output/folder")

validation.run_base_validation(inpath, outpath)
```

Alternatively the validation code can be run as a script found in /tests:

```
>>python run_base_validation.py -i "path/to/S-BSST641/" -o "path/to/output/folder"
```

After some time you should have generated the 3 figures below in this chapter in your chosen output folder. To test the 5-sgigma analysis run the following code:

```
from cellocity import validation
from pathlib import Path

inpath = Path("path/to/S-BSST641/")
outpath = Path("path/to/output/folder")

validation.run_5sigma_validation(inpath, outpath)
```

Alternatively the validation code can be run as a script found in /tests:

```
>>python run_5sigma_validation.py -i "path/to/S-BSST641/" -o "path/to/output/folder"
```

First we'll start by looking at the base validation.

1475, 2016. doi:10.1016/j.bpj.2016.02.032.

3.4 Process time

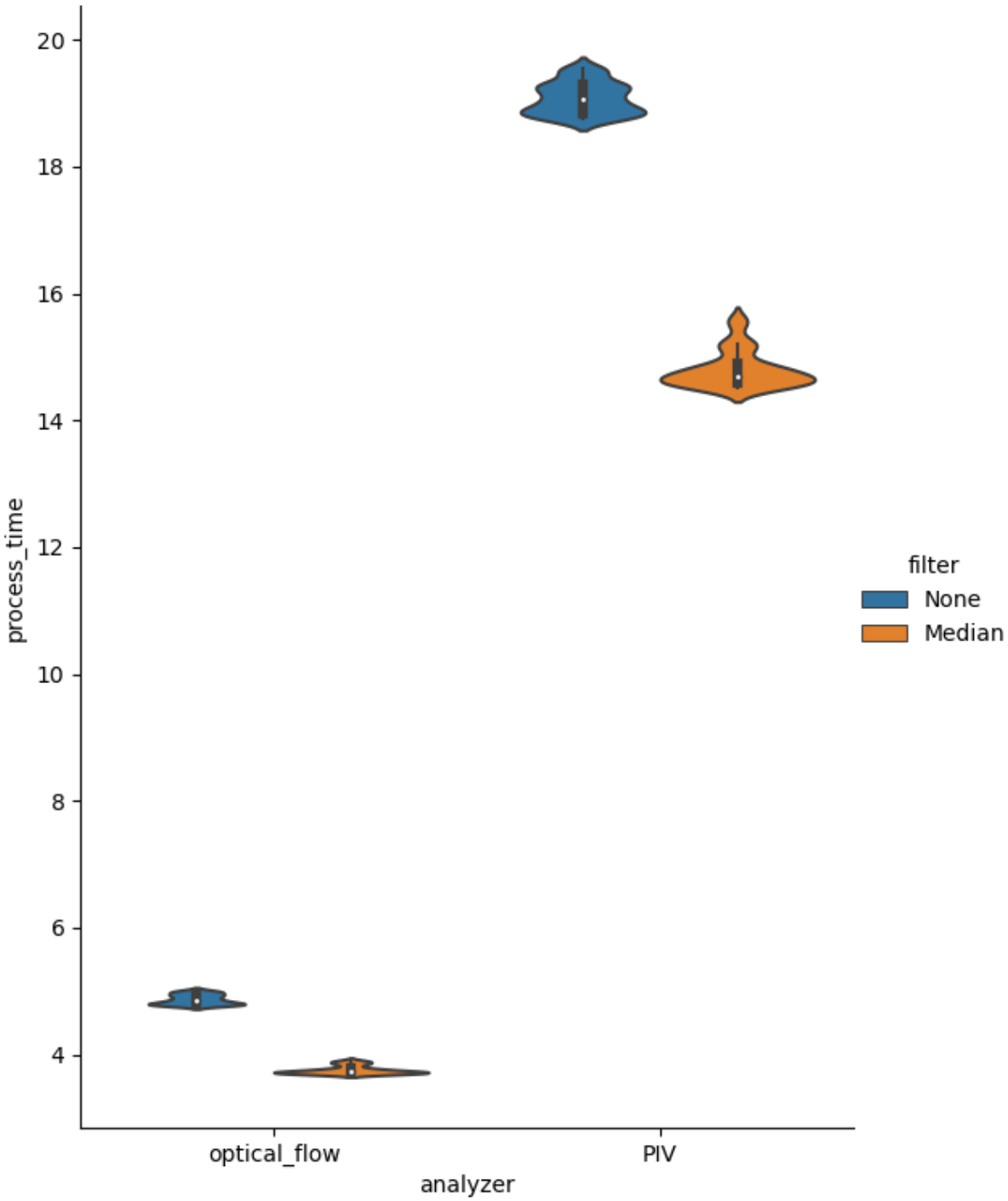


Fig. 1: Figure showing violin plots of processing times for individual files in the test dataset. Process time is in seconds and denotes time to run either the `OpenPivAnalyzer` or the `FarenbackAnalyzer` on both a `Channel` and a `MedianChannel` object created from each file in the dataset. Each file is a 10x1608x1608 16-bit array.

Optical Flow is clearly faster to process all files by a factor of ~3-4X. Now, let's compare overall accuracy. Since the dataset was created by translating a high precision stage on a well calibrated microscope,

know that the speed of the apparent flow is dependent on the translation distance. In our case we translated the stage 1 μm between images, and if we set the frame interval to 1 second, then the speed should be 1 $\mu m/s$ for the X and Y translation and $\sqrt{2} = 1.42$ $\mu m/s$ for the X+Y translation.

3.5 Analysis of flow speeds

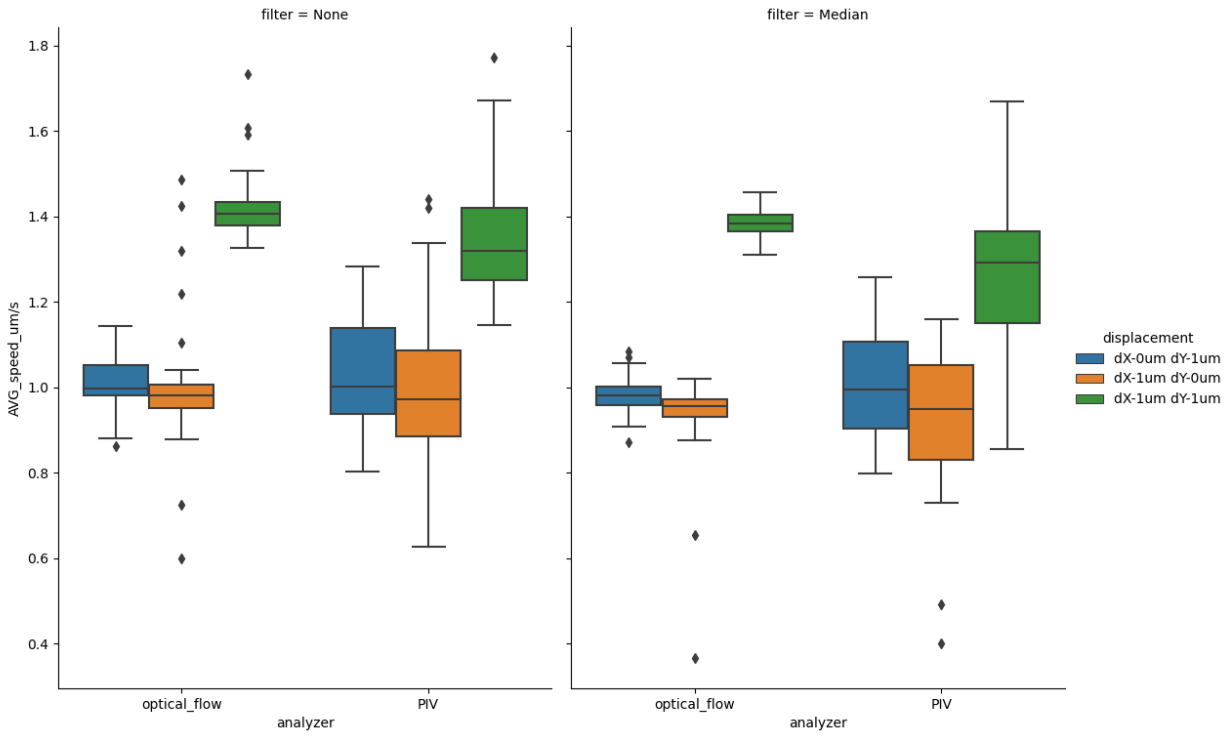


Fig. 2: Figure showing box plots of average speeds for each frame for each file in the test dataset. y-axis denotes the speed in $\mu m/s$, as read out by the `.calculateAverageSpeeds()` method of `FlowSpeedAnalyser`.

Both `Analyzers` produce results close to the expected, but the `OpenPivAnal` has a tendency to underestimate the speed and has greater variance.

Cell monolayers grown on loose hydrogel support, as those used in our validation dataset here, are seldom completely planar and portions are often out of focus during imaging. This phenomenon has also been captured in the analysis. If we draw a visualization of the flow generated superimposed on the background `Channel`, we can study this phenomenon in more detail.

3.6 Qualitative vector field comparison

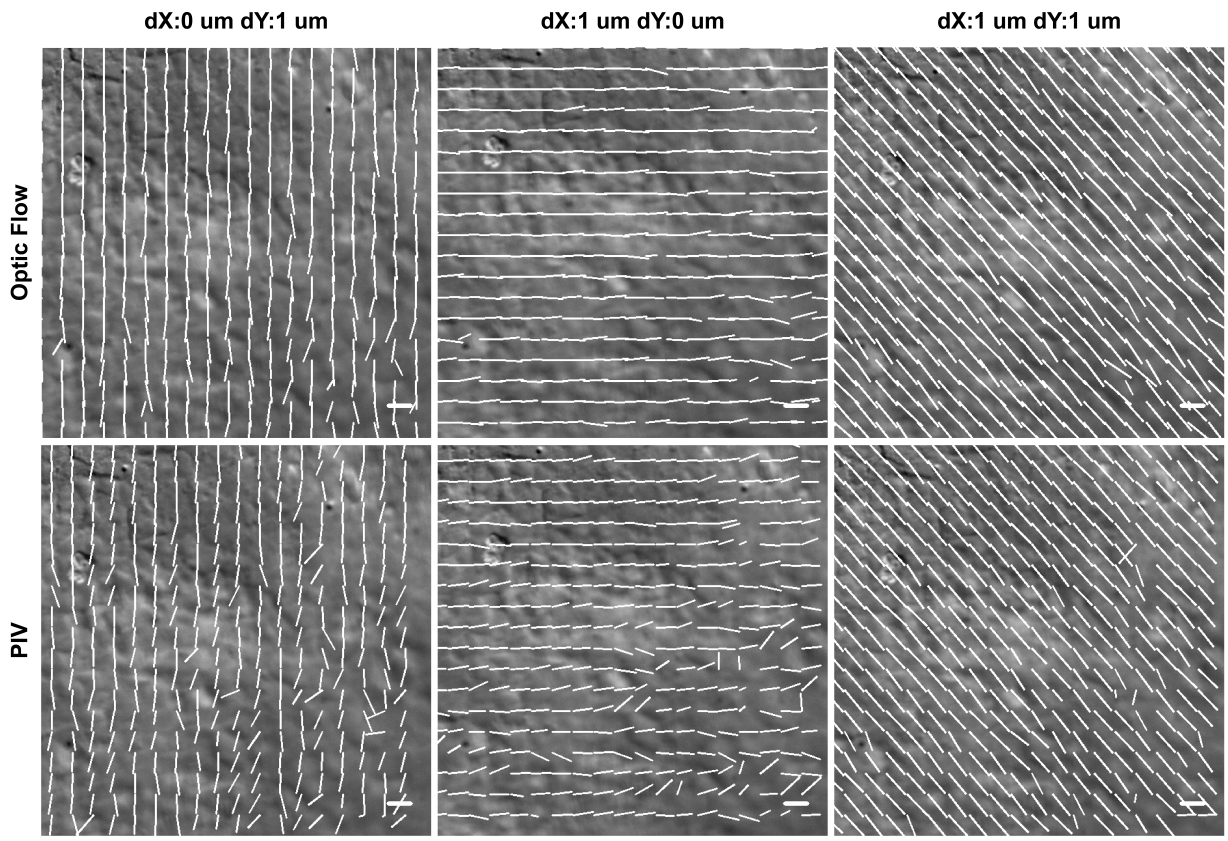


Fig. 3: Figure showing flow vector visualization of a 600x600 crop from the bottom right corner of the final frame from the 40X magnification files in the dataset. Images were generated using the `.draw_all_flow_frames_superimposed()` method common to all `FlowAnalysis` objects. Horizontal scale bar denotes a flow of 1 $\mu m/s$.

Studying the above figure allows us to get a deeper understanding of why optical flow and PIV differ. Note that the area in the bottom right corner is not properly focused. This causes the PIV algorithm problems in accurately determining the flow, as illustrated by the inhomogeneities in the vector field. This error can be quantified by calculating the alignment index, a measurement on how well each component vector aligns with the average flow. In our test dataset the flow should be close to completely uniform, giving an expected alignment index of 1.0.

3.7 Quantitative vector field comparison

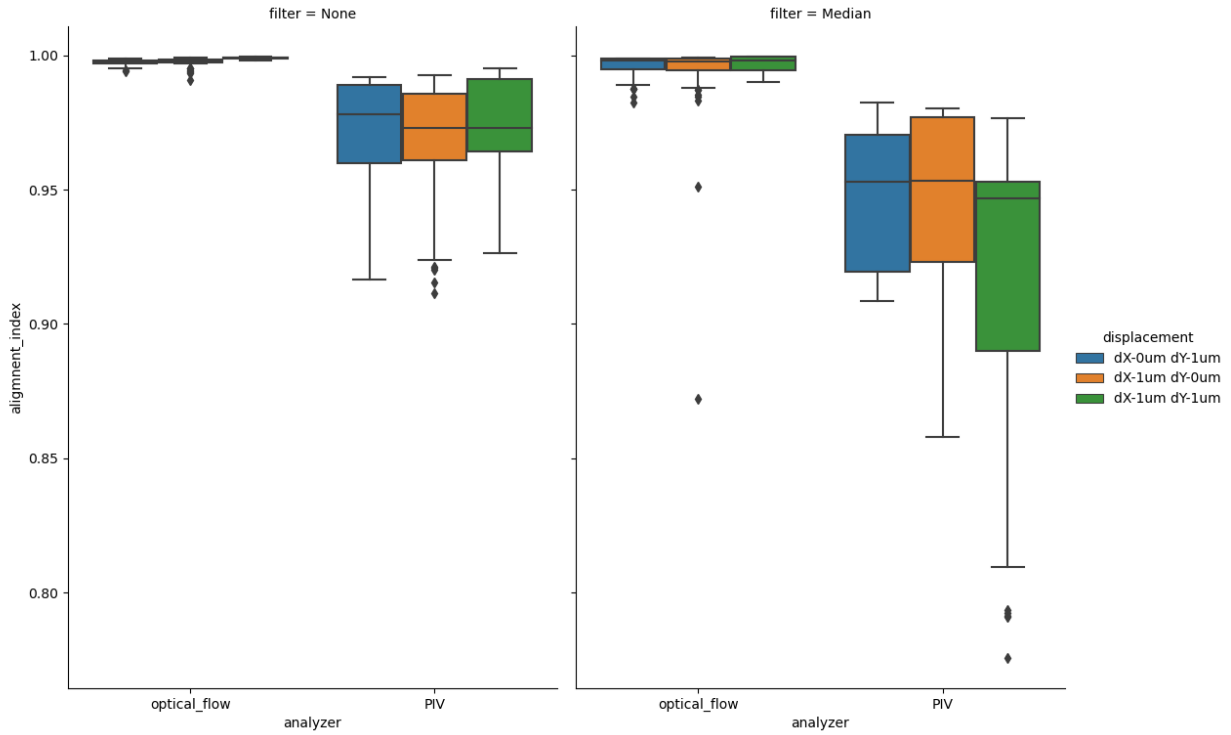
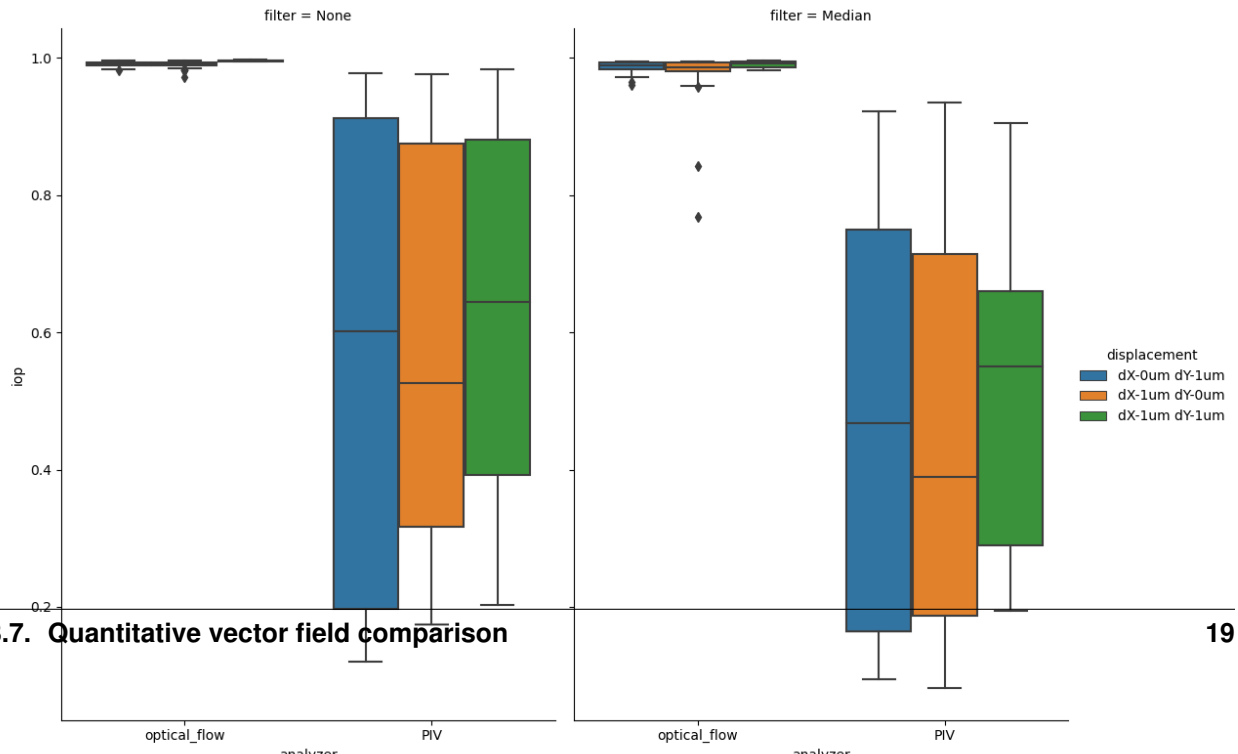


Fig. 4: Figure showing box plots of average alignment indexes for each frame for each file in the test dataset. y-axis denotes the Alignment Index (dimensionless), as read out by the `.getAvgAlignIdxs()` method of `AlignmentIndexAnalysis`.

Quantifying how well the vector field is aligned allows us to confirm our initial observation that PIV analysis does produce more variability in the direction of the flow vectors. Optical Flow generates alignment indexes very close to the expected value of 1.0, even after temporal median filtering.



The instantaneous order parameter is a measure of how ordered

a vector field is, 0 represents a completely random field and 1 represents a comletely homogenous field, where all vectors have the same direction and magnitude. The expected value for the test data set is 1.

Lastly, let's have a look at the 5-σ correlation length analysis. 5-σ correlation length is a way to measure the correlation length in large vector fields. The algorithm finds the average distance at which the direction of velocities are no longer significantly different at a level of 5 standard deviations (σ). The algorithm was originally presented and utilized in[2]. A more detailed description can be found in the *Developer Information*.

Once again we see an advantage in utilizing optical flow when compared to PIV. In the figure above the calculated correlation lengths for each file and frame are divided by the size of the field of view (FOV), giving us a metric to compare across magnifications. Optical flow captures almost all of the "true" correlation length, while PIV is only able to capture ~80-85% of the "true" correlation length. Besides being less accurate on this type of data, PIV analysis also downsamples the images, which gives the `FiveSigmaAnalysis` fewer vectors to use as a basis for correlation length calculations. This is also evident in the average frame processing time (below).

Fig. 6: Figure showing flow vector visualization of the diagonal translation at 60X magnification. Images were generated using the `draw_all_flow_frames_superimposed()` method common to

[2] Emma Lång, Anna Połeć, Anna Lång, Marijke Valk, Pernille Blicher, Alexander D. Rowe, Kim A. Tønseth, Catherine J. Jackson, Tor P. Utheim, Liesbeth M. C. Janssen, Jens Eriksson and Stig Ove Bøe. Coordinated collective migration and asymmetric cell division in confluent human keratinocytes without wounding. *Nature communications*, 1:2041-1723, 2018. doi:10.1038/s41467-018-05578-7.

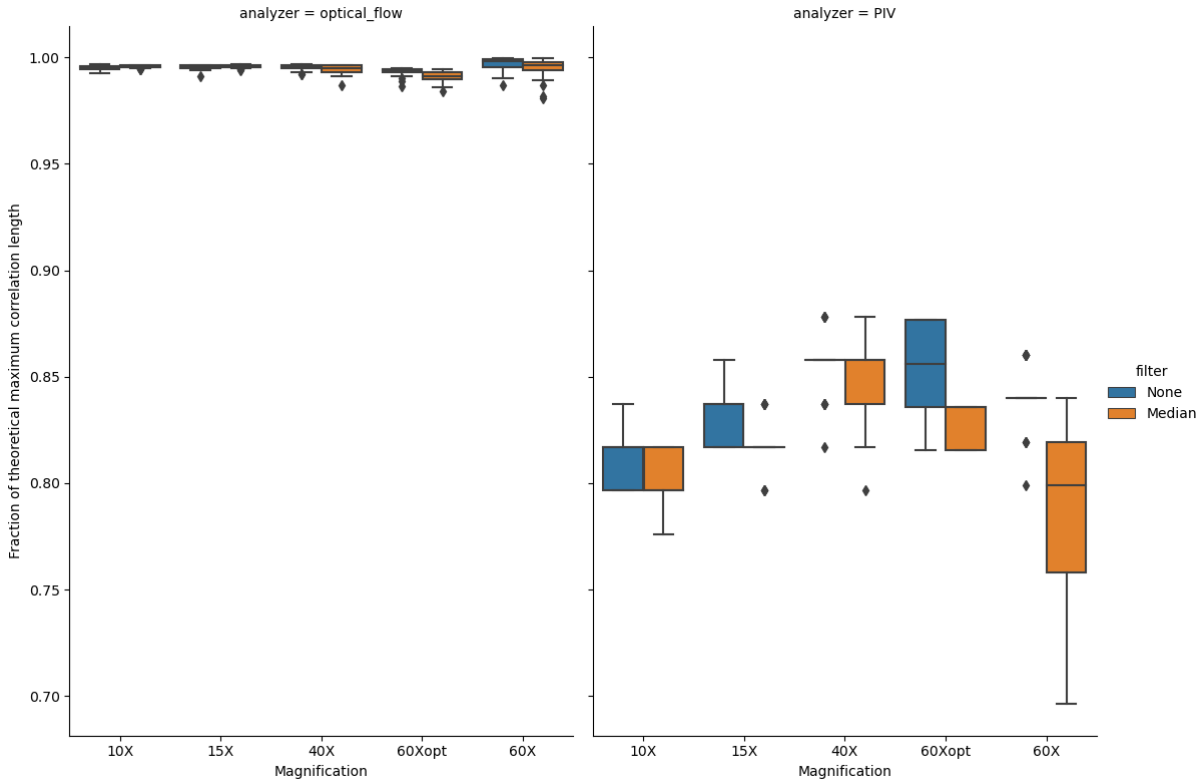
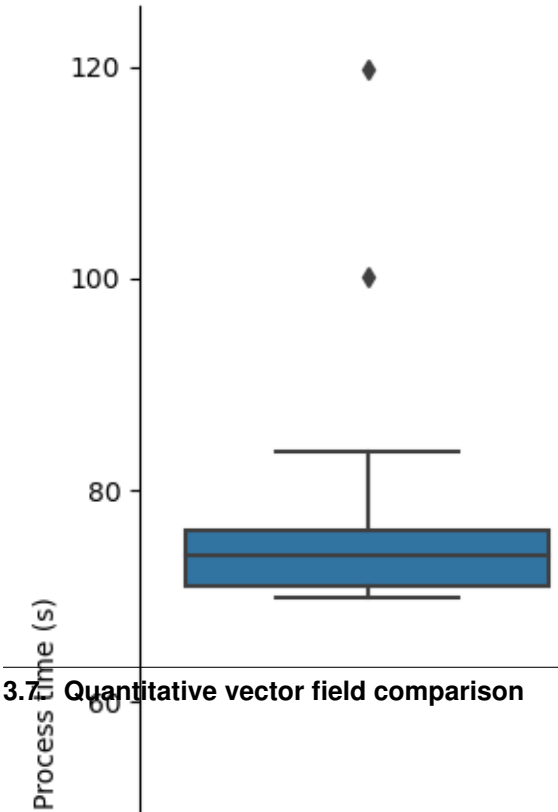


Fig. 7: Figure showing box plots of how well the calculated 5-σ correlation lengths agree with the theoretical maximum value. The value is expressed as a fraction of the calculated correlation length and the theoretical maximum value, which is given by the magnification. Figure generated by passing either `OpenPivAnalyzer` (right) or `FarenbackAnalyzer` (left) objects to `FiveSigmaAnalysis` and calling the `calculateCorrelationAllFrames()` metod with (orange) and without (blue) temporal median filtering of raw input data.



Here we see an advantage in using PIV because processing times are orders of magnitude lower for the PIV data. The complex-

ity of the 5-σ correlation length analysis algorithm is $O(n)$, meaning processing time grows liearly with input size. It is possible to reduce the processing time of optical flow data by tweaking the `r_step` parameter of the `_get_all_angles()` metod of `FiveSigmaAnalysis`. The default value is 1, which means that the comparison is growing outwards by 1 pixel per step, but a value of 2 would halve the number of comparisons calculated with little expected effect on the final result.

3.8 In conclusion

Optical Flow and PIV analysis of transmitted light microscopy time-lapse data is commonly used in studies of confluent cell layer dynamics phenomena, for example collective cell migration and wound healing. This is particularly relevant for studies of primary cells, due to the difficulty in reliably labelling these for cell tracking. To our knowledge, there has not been a systematic evaluation of different pre-processing modalities and optical flow analysis algorithms on actual real-world, non-simulated, microscopy data. We therefore anticipate that others will find this software package and the validation dataset described in this chapter useful.

3.9 References

CHAPTER

FOUR

DEVELOPER INFORMATION

This section contains information relevant for developing and extending Cellocity functionality. It also contains random tidbits of general information that I have uncovered during the development process of this framework. I present it here in the hope that someone may find it useful.

4.1 Contributing to Cellocity

Contributions are welcome and appreciated. Just fork the Github repository and create a pull request. Information on how to do so can be found here. Before you do so, please make sure that the documentation strings are written in reStructuredText so that Sphinx-autodoc can generate automatic API documentation. It would also be greatly appreciated if the general architecture of `Channel`, `Analyzer`, and `Analysis` objects is maintained.

4.2 Bug reports and feature requests

Bug reposts and feature requests can be submitted through Github.

4.3 A note on metadata and file formats

It goes without saying that you need to have a well calibrated microscope that writes correct metadata into your image files, in order to perform meaningful cell dynamics analysis. The minimal amount of information needed is data on the pixel size and the time resolution between frames. Image format specific metadata, such as Micromanager-metadata and IJmetadata contain this information and constitute the primary source used throughout Cellocity.

Micromanager saves its metadata in a private IFD tag (51123), which `Tifffile` reads in as a `dict`, accessible via `tif.micromanager_metadata`. The structure of the dictionary is, annoyingly, slightly different between the 1.4.23, 2.0-beta, and 2.0-gamma branches of Micromanager. In 1.4.23 and 2.0-gamma the frame interval is stored in `tif.micromanager_metadata['Summary']['Interval_ms']`, but in 2.0-beta it is stored in `tif.micromanager_metadata['Summary']['WaitInterval']`. The discrepancy is probably due to the fact that this value records the wait interval time between frames of the acquisition, not the *actual* frame interval. It is possible to setup an acquisition with a frame interval that the microscope physically cannot keep up with. Therefore, Cellocity performs an additional sanity check of the individual time stamps of the frames (see `Channel.doFrameIntervalSanityCheck()`), in order to make sure you do not run into this problem during analysis.

4.4 Pixel resolution in Micromanager vs ImageJ .tif files

Relevant standard tags in the TIFF specification are **XResolution**, **YResolution**, and **ResolutionUnit**. The resolution tags are rational numbers, meaning they are generated by dividing two 32-bit integer values. **ResolutionUnit** is specified as being either *None*, *Inch* or *Centimeter*. No other units are specified.

When Micromanager saves an ome.tif it writes a rounded off value into the XResolution and YResolution tif tags, and it sets the **ResolutionUnit** tag to CENTIMETER. This value carries less precision than the 'PixelSizeUm' entry in the custom TIFF-tag 'MicroManagerMetadata', but the TIFF is correctly readable with roughly intact size calibration data in any reader obeying the TIFF standard.

When ImageJ (v. 1.52p) saves a Hyperstack as tif, it writes the 'Pixel Width' and 'Pixel Height' values into the **XResolution** and **YResolution** tags with higher precision. However, it sets the **ResolutionUnit** tag to *None*, probably because microns, the standard micrograph unit, are not specified in the TIFF standard.

4.5 Creating your own image format reader

If you want to develop your own reader for your microscope raw data, I suggest you look up the Tiffile project. It already implements reading of many common tif-formats from multiple microscope vendors. It is a trivial addition to tweak the Channel object and create your own subclass version specific to your file format, since Channel objects are basically extensions of Tifffile objects.

Pragmatically, the easiest way to get your non-supported image data set into Cellocity is to open it in FIJI with the Bioformats importer and thereafter resave it as a hyperstack tif (making sure that the relevant image properties are set correctly). Then, you can use the ImageJ-reading capabilities built in to Cellocity and tiffile.

4.6 Detailed description of the 5-σ correlation length analysis algorithm

The 5-σ correlation length was defined as the largest distance, r, where the average angle between two velocity vectors r micrometers apart was < 90 with a statistical significance level of 5 σ ($p = 310^7$).

4.6.1 The algorithm steps:

1. Select each of the N vectors along the top left to bottom right diagonal of the `FlowAnalyzer` output velocity vector array as $\mathbf{v}_0$.
2. For each $\mathbf{v}_0$, expand linearly, one row/column position at a time, along the cardinal directions (up/down/left/right) and calculate the angle between $\mathbf{v}_0$ and each of the vectors $\mathbf{v}_r$, at each position. Do not include masked positions, or positions outside of the array. The angles θ for each $\mathbf{v}_0$ are calculated with the formula: $\cos\theta = \frac{\langle \mathbf{v}_0 * \mathbf{v}_r \rangle}{\langle |\mathbf{v}_0| * |\mathbf{v}_r| \rangle}$.
3. Record all the angles and distances between $\mathbf{v}_0$ and $\mathbf{v}_r$ for each N, and for each time point, t.
4. For each distance r, and time point t, average all the angles recorded at this distance: $\theta(r) = \frac{1}{N} * \sum_{i=1}^{N} \cos^{-1}\left(\frac{\langle \mathbf{v}_0 * \mathbf{v}_r \rangle}{\langle |\mathbf{v}_0| * |\mathbf{v}_r| \rangle}\right)$.
5. Compute the angular velocity correlation length at each time point. This is defined as the maximum distance where θ is < 90 with a statistical significance of 5 σ: $C_{vv}(t) = \max_{r \to \infty}(r)\{\mathrm{AVG}(\theta)(r) + 5 * \mathrm{SEM}(\theta(r)) < 90°\}$

CHAPTER

FIVE

THE CELLOCITY API REFERENCE

5.1 The channel module

class cellocity.channel.**Channel**(*chIndex*, *tiffFile*, *name*, *sliceIndex=0*)

Base Class to keep track of one channel (t,x,y) of microscopy data.

Channel Objects are created from tifffile.Tifffile and act as shallow copies of the TiffPage objects making up the channel, until a Numpy array is generated by 'getArray'. Then self.array is populated by a Numpy array from the raw image data, using the 'asarray' function in 'tiffile.Pages'. Only a single z-slice and channel are handled per Channel object. A reference to the base 'tifffile.Tifffile' is stored in self.tif.

There are currently two very similar subclasses of Channel, MM_Channel, and IJ_Channel to handle Micro-manager OME-TIFFs and ImageJ hyperstacks, respectively.

Parameters

- **chIndex** (*int*) – index of channel to create, 0-based.
- **tiffFile** (*:class:'tifffile'*) – TiffFile object to extract channel from
- **name** (*str*) – name of channel, used in Analysis output
- **sliceIndex** (*int*) – z-slice to extract, defaults to 0

doFrameIntervalSanityCheck(*maxDiff=0.01*)

Performs sanity check on frame intervals.

Checks if the intended frame interval from metadata matches the actual frame interval from individual frame time stamps. If the mean difference is more than maxDiff the function returns `False`. Defaults to allowing a 1% difference between mean actual frame interval and intended frame interval by default.

Parameters **maxDiff** (*float*) – Maximum allowed difference between actual frame intervals and the intended interval, expressed as a fraction.

Returns True if the fraction of actual and intended frame intervals is below maxDiff.

Return type bool

fixFrameInterval()

Replaces the intended frame interval with the actual.

Use this method in case the self.doFrameIntervalSanityCheck() method fails. The method overwrites the intended frame interval stored in self.finterval_ms with the actual, as calculated from the mean of all time stamp deltas.

Returns New frame interval

Return type float

getActualFrameIntevals_ms()
Returns the intervals between frames in ms as a list.

Note that the length of this list is 1 shorter than the number of frames because frame intervals are calculated. Returns None if only one frame exists in the channel.

Returns list of time intervals between frames, None if self.array contains fewer than 2 frames.

Return type list

getArray()
Returns channel image data as a numpy array.

Method populates the array from self.pages first time it is called.

Returns Channel image data as 3D-numpy array

Return type numpy.ndarray (type depends of original format)

getElapsedTimes_ms()
Returns a list of elapsed times in ms from the start of image acquisition.

Values are extracted from image timestamps. Note that this is only possible for MicroManager based Channels (and other timestamped formats). Since ImageJ does not store this information the frame interval value is trusted and used to calculate elapsed times.

Returns Timestamps of channel frames from the start of the acquisition.

Return type list

getIntendedFrameInterval_ms()
Returns the intended frame interval as recorded in image metadata.

Returns interval between successive frames in ms

Return type int

getPages()
Returns the TiffPages that make up the channel data

Returns a list of the TiffPages extracted from the Tifffile used to create the Channel

Return type list

getTemporalMedianChannel(***kwargs*)
Returns a new MedianChannel object where self.array has been replaced with temporal median filtered channel data

kwargs and defaults are: {doGlidingProjection = True, frameSamplingInterval=3, startFrame=0, stopFrame=None} Defaults to a gliding 3 frame temporal median of the whole channel if no kwargs are given.

Returns A MedianChannel object based on the current channel where self.array has been replaced by a numpy array of the type float32 representing the temporal median of Channel data.

Return type *MedianChannel*

getTiffFile()
Returns the *Tifffile* objedt that the *Channel* is based on.

Returns Tifffile-object used when Channel was created

Return type object tifffile.Tifffile

trim(*start*, *stop*)

Trims the channel from *start* frame to *stop* frame, removing pages and array pages outside the given range.

All relevant properties are also trimmed and the Channel name is appended with "_TRIM-`start`-*stop*"

Parameters

- **start** (*int*) – start frame of trim (0-indexed)
- **stop** (*int*) – stop frame of trim (not included)

Returns None, trims Channel in place

class cellocity.channel.**MedianChannel**(*channel*, *doGlidingProjection=True*, *frameSamplingInterval=3*, *startFrame=0*, *stopFrame=None*)

A subclass of `Channel` where the channel array has been temporal median filtered.

Temporal median filtering is very useful when performing optical flow based analysis of time lapse microscopy data, because it filters out fast moving free-floating debris from the dataset. Note that the median array will be shorter than the original array. In the default case, if a temporal median of 3 frames is applied, then the the output array will contain 3-1 = 2 frames less than the input if a gliding projection (default) is performed.

Parameters

- **channel** (*Channel object*) – Parent Channel object for the MedianChannel
- **doGlidingProjection** (*bool*) – Should a gliding projection be used? Defaults to `True`, if `False` a binned projection is performed, this will also recalculate the frame interval.
- **frameSamplingInterval** (*int*) – How many frames to use in temporal median projection, defaults to 3
- **startFrame** (*int*) – Start frame of median projection
- **stopFrame** (*int or None*) – Stop frame of median projection (non inclusive), defaults to `None` i.e. all frames

getTemporalMedianFilter(*doGlidingProjection*, *startFrame*, *stopFrame*, *frameSamplingInterval*)

Returns a temporal median filter of the parent Channel.

The function runs a gliding N-frame temporal median on every pixel to smooth out noise and to remove fast moving debris that is not migrating cells.

Parameters

- **doGlidingProjection** (*bool*) – Should a gliding (default) or binned projection be performed?
- **stopFrame** (*int*) – Last frame to analyze, defaults to analyzing all frames if `None`.
- **startFrame** (*int*) – First frame to analyze.
- **frameSamplingInterval** (*int*) – Do median projection every N frames.

Returns Numpy array

Return type numpt.ndarray

cellocity.channel.**normalization_to_8bit**(*image_stack*, *lowPcClip=0.175*, *highPcClip=0.175*)

Function to rescale 16/32/64 bit arrays to 8-bit for visualizing output

Defaults to saturate 0.35% of pixels, 0.175% in each end by default, which often produces nice results. This is the same as pressing 'Auto' in the ImageJ contrast manager. *numpy.interp()* linear interpolation is used for the mapping.

Parameters

- **image_stack** (*Numpy array*) – 3D Numpy array to be rescaled
- **lowPcClip** (*float*) – Fraction for black clipping bound
- **highPcClip** (*float*) – Fraction for white/saturated clipping bound

Returns 8-bit numpy array of the same shape as image_stack

Return type numpy.dtype('uint8')

cellocity.channel.**rehape3DArrayTo6D**(*array_3d*)
reshapes 3D (t, x, y) array to (t, 1, 1, x, y, 1).

Used when saving ImageJ compatible tifs using `Tifffile` where dimensions have to be in TZCYXS order.

Parameters **array_3d** – 3D numpy array

Returns None

cellocity.channel.**reshape6DArrayTo3D**(*array_6d*)
Undoes what reshape3DArrayTo6D does to the shape of the array.

Parameters **array_6d** – 6D numpy array

Returns

5.2 The analysis module

class cellocity.analysis.**AlignmentIndexAnalysis**(*analyzer*)
Calculates the alignment index for the flow vectors in a FlowAnalyzer object.

Alignment index (AI) is defined as in Malinverno et. al 2017. For every frame the AI is the average of the dot products of the mean velocity vector with each individual vector, all divided by the product of their magnitudes.

The alignment index is 1 when the local velocity is parallel to the mean direction of migration (-1 if antiparallel).

Parameters **analyzer** – Analyzer object

calculateAlignIdxs()
Calculates the aligment index for each pixel in base FlowAnalyzer flow array and populates self.alignment_idxs

Returns numpy array with same size as analyzer flows, where every entry is the alignment index in that pixel

Return type numpy.ndarray

calculateAverage()
Calculates the average alignment index for each time point in self.alignment_idxs

Returns self.avg_alignment_idxs, 1D numpy.ndarray of the same length as self.alignment_idxs

Return type numpy.ndarray

getAvgAlignIdxAsDf()
Returns frame and average alignment index for the frame as a Pandas DataFrame.

Returns DataFrame with 1 column for average aligmnent index and index = frame number

Return type pandas.DataFrame

getAvgAlignIdxs()
Returns average alignment indexes for Analyzer

Returns

saveArrayAsTif(*outdir*, *fname=None*)
Saves the alignment index array as a 32-bit tif with imageJ metadata.

Pixel intensities encode alignment indexes.

Parameters

- **outdir** (*pathlib.Path*) – Directory to store file in
- **fname** – Filename, defaults to Analysis channel name with appended tags +_ai.tif if None

Returns None

saveCSV(*outdir*, *fname=None*, *tunit='s'*)
Saves a csv of average aligmnent indexes per frame in outdir.

Parameters

- **outdir** (*pathlib.Path*) – Directory where output is stored
- **fname** (*str*) – filename, defaults to channel name + ai.csv
- **tunit** (*str*) – Time unit in output one of: "s", "min", "h", "days"

Returns

class cellocity.analysis.**Analysis**(*analyzer*)
Base object for handling data output and analysis and of Analyzer classes.

Parameters **analyzer** – Analyzer object

getAnalyzer()
Returns the Analyzer that the Analysis is based on.

Returns the Analyzer that the Analyser is based on.

Return type *Analyzer*

getChannelName()
Returns the name of the channel that the base Analyzer, in turn, is based on.

Returns self.name of the Channel that the base Analyzer is based on.

Return type str

class cellocity.analysis.**Analyzer**(*channel*)
Base object for all Analysis object types, handles progress updates.

Parameters **channel** (class:*channel.Channel*) – A Channel object

getProgress()
Returns current progress in the interval 0-100.

Returns Percentage progress of analysis

Return type float

resetProgress()
Resets progressbar to 0

Returns

updateProgress(*increment*)
Updates self.progress by increment

Parameters **increment** –

Returns

class cellocity.analysis.**FarenbackAnalyzer**(*channel*, *unit*)
Performs OpenCV's Farenbäck optical flow anaysis.

Parameters

- **channel** – Channel object
- **unit** – (str) "um/s", "um/min", or "um/h"

doFarenbackFlow(*pyr_scale=0.5*, *levels=3*, *winsize=15*, *iterations=3*, *poly_n=5*, *poly_sigma=1.2*, *flags=0*)
Calculates Farenback flow for a single channel time lapse with validated default parameters.

returns numpy array of dtype int32 with flow in the unit px/frame Output values need to be multiplied by a scalar to be converted to speeds.

class cellocity.analysis.**FiveSigmaAnalysis**(*flowanalyzer*, *maxdist=None*)
Calculates the 5-sigma corrleation length for each frame of flow (see Lång et. al 2018 or the documentation for a more detailed explanation).

The 5-σ correlation length was defined as the largest distance, r, where the average angle between two velocity vectors r micrometers apart was < 90 with a statistical significance level of 5 σ $(p = 310^7)$.

Parameters

- **flowanalyzer** (analysis.FlowAnalyzer) – a FlowAnalyzer object
- **maxdist** (*int*) – Maximum distance (in pixels) to test if None defaults to max(flow width, height)

calculateCorrelationAllFrames(*n_sigma=5*)
Calculates correlation length for all flow frames

Parameters **n_sigma** –

Returns

calculateCorrelationOneFrame(*frame*, *n_sigma=5*)

Parameters

- **frame** – (flow) frame to calculate correlation length for
- **n_sigma** – Number of standard deviations to consider significant

Returns

getCorrelationLengths()
Returns correlation lengths as a dictionary frame:correlation_length_in_um

Returns

getCorrelationLengthsAsDf(*tunit='s'*)
Returns a Pandas DataFrame with index:time in "tunit" and Correlation length.

Parameters **tunit** – Time unit in output one of: "s", "min", "h", "days"

Returns pandas.Dataframe

saveCSV(*outdir*, *fname=None*, *tunit='s'*)
Saves a csv of correlation lengths per frame in outdir.

Parameters

- **outdir** (*pathlib.Path*) – Directory where output is stored
- **fname** (*str*) – filename, defaults to channel name + _Cvv.csv
- **tunit** (*str*) – Time unit in output one of: "s", "min", "h", "days"

Returns

class cellocity.analysis.**FlowAnalysis**(*analyzer*)

Base object for analysis of optical flow and PIV.

Works on FlowAnalyzer objects, such as FarenbackAnalyzer and OpenPIVAnalyzer. Needs a 4D (t, x, y, uv) numpy array representing a time lapse of a vector field to initialize.

Parameters **analyzer** – Analyzer object

draw_all_flow_frames(*scalebarFlag=False*, *scalebarLength=10*, ***kwargs*)

Draws flow on a black background as an 8-bit array.

Draws a subset of the flow as lines on top of a black background. Because the flow represents what happens between frames, the flow is not drawn on the last frame of the channel, which is discarded. Creates & populates self.drawnframes to store the drawn array. If the underlying channel object is 16-bit, it will converted to 8bit with the *channel.normailzation_to_8bit()* function.

Parameters

- **scalebarFlag** (*bool*) – Should a scale bar be drawn on the output?
- **scalebarLength** – What speed should the scale bar represent with its length the unit is set by the unit

given to the Analyzer :param kwargs: Additional arguments passed to self._draw_flow_frame() :type kwargs: dict

Returns 8bit numpy array

draw_all_flow_frames_superimposed(*scalebarFlag=False*, *scalebarLength=10*, ***kwargs*)

Draws flow superimposed on the background channel as an 8-bit array.

Draws a subset of the flow as lines on top of the background channel. Because the flow represents what happens between frames, the flow is not drawn on the last frame of the channel, which is discarded. Creates & populates self.drawnframes to store the drawn array. If the underlying channel object is 16-bit, it will converted to 8bit with the *channel.normailzation_to_8bit()* function.

Parameters

- **scalebarFlag** (*bool*) – Should a scale bar be drawn on the output?
- **scalebarLength** – What speed should the scalebar represent with its length the unit is set by the unit

given to the Analyzer :param kwargs: Additional arguments passed to self._draw_flow_frame() :type kwargs: dict

Returns 8bit numpy array

saveFlowAsTif(*outpath*)

Saves the drawn frames as an imageJ compatible tif with rudimentary metadata.

Parameters **outpath** (*Path object*) – Path to savefolder

Returns None

class cellocity.analysis.**FlowAnalyzer**(*channel*, *unit*)
Base object for all optical flow analysis object types.

Stores UV vector components in self.flows as a (t, x, y, uv) numpy array. Also calculates and stores a scaling factor that converts flow from pixels per frame to distance/time.

Parameters **unit** (*str*) – must be one of ["um/s", "um/min", "um/h"]

get_flow_shape()
Returns the shape of self.flows

Returns the shape of self.flows

Return type tuple

get_pixel_size()
Returns the pixel size in um of the Analyzer.

Some type of Analyzers, such as the OpenPivAnalyzer change the pixel size of the array by downsampling.

Returns pixel size in um of the Analyzer

Return type float

get_u_array(*frame*)
Returns the u-component array of self.flows at frame

Parameters **frame** (*int*) – frame to extract u-component matrix from

Returns u-component of velocity vectors as a 2D NumPy array

Return type numpy.ndarray

get_v_array(*frame*)
Returns the v-component array of self.flows

Parameters **frame** (*int*) – frame to extract v-component matrix from

Returns v-component of velocity vectors as a 2D NumPy array

Return type numpy.ndarray

class cellocity.analysis.**FlowSpeedAnalysis**(*analyzer*)
Handles all analysis and data output of speeds from FlowAnalyzers.

Calculates pixel-by-pixel speeds from flow vectors.

Parameters **analyzer** – Analyzer object

calculateAverageSpeeds()
Calculates the average speed for each time point in self.speeds

Returns self.avg_speeds

Return type 1D numpy.ndarray of the same length as self.speeds

calculateHistograms(*hist_range=None*, *nbins=100*, *density=True*)
Calculates a histogram for each frame in self.speeds

Parameters

- **hist_range** (*tuple*) – Range of histogram, defaults to 0-max
- **nbins** (*int*) – Number of bins in histogram, defaults to 100
- **density** (*bool*) – If `False`, the result will contain the number of samples in each bin. If `True` (default), the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1.

Returns self.histograms

Return type tuple (numpy.ndarray, bins)

calculateSpeeds(*scaler=None*)
Calculates speeds from the flows in parent Analyzer

Turns a (t, x, y, uv) flow numpy array with u/v component vectors in to a (t, x, y) speed array. Populates self.speeds. Scales all the output by multiplying with scaler, defaults to using the self.scaler from the base FlowAnalyzer object if the scaler argument is `None`.

self.scaler is the scalar quantity that converts flow vectors from the general unit of pixels/frame in to the desired output unit, such as um/s.

:returns self.speeds :rtype: numpy.ndarray

getAvgSpeeds()
Returns average speed per frame as a 1D Numpy array.

Returns average speed per frame

Return type numpy.ndarray (1D)

getAvgSpeedsAsDf()
Returns frame and average speed for the frame as a Pandas DataFrame.

Returns DataFrame with 1 column for average speed and index = frame number

Return type pandas.DataFrame

getSpeeds()
Returns self.speeds.

Calculates self.speeds with default values if it has not already been calculated.

Returns self.speeds as a 3D Numpy array

Return type numpy.ndarray (3D)

saveArrayAsTif(*outdir*, *fname=None*)
Saves the speed array as a 32-bit tif with imageJ metadata.

Pixel intensities encode speeds in the chosen analysis unit

Parameters

- **outdir** (*pathlib.Path*) – Directory to store file in
- **fname** – Filename, defaults to Analysis channel name with appended tags +_speeds-SizeUnit-per-TimeUnit.tif if `None`

Returns None

saveCSV(*outdir*, *fname=None*, *tunit='s'*)
Saves a csv of average speeds per frame in outdir.

Parameters

- **outdir** (*pathlib.Path*) – Directory where output is stored
- **fname** (*str*) – filename, defaults to channel name + speeds.csv
- **tunit** (*str*) – Time unit in output one of: “s”, “min”, “h”, “days”

Returns

class cellocity.analysis.**IopAnalysis**(*flowanalyzer*)

Calculates the instantaneous order parameter (iop) for each frame of flow (see Malinverno et. al 2017 for a more detailed explanation).

The iop is a measure of how similar the vectors in a field are, which takes in to account both the direction and magnitudes of the vectors. iop is always between 0 and 1, with iop = 1 being a perfectly uniform field of identical vectors, and iop = 0 for a perfectly random field.

Parameters **flowanalyzer** (analysis.FlowAnalyzer) – a FlowAnalyzer object

calculateIops()

Calculates the IOP for each frame in base FlowAnalyzer flow array and populates self.iops

Returns list of the IOP from each frame

Return type list

getIops()

Returns the instantaneous order parameter for Analyzer

Returns list of instantaneous order parameters

Return type list

getIopsAsDf()

Returns frame and iop for the frame as a Pandas DataFrame.

Returns DataFrame with 1 column for iop and index = frame number

Return type pandas.DataFrame

saveCSV(*outdir*, *fname=None*, *tunit='s'*)

Saves a csv of the iop per frame in outdir.

Parameters

- **outdir** (*pathlib.Path*) – Directory where output is stored
- **fname** (*str*) – filename, defaults to channel name + iop.csv
- **tunit** (*str*) – Time unit in output one of: “s”, “min”, “h”, “days”

Returns

class cellocity.analysis.**OpenPivAnalyzer**(*channel*, *unit*)

Implements OpenPIV’s optical flow anaysis.

Parameters

- **channel** – Channel object
- **unit** – (str) “um/s”, “um/min”, or “um/h”

doOpenPIV(***piv_params*)

The function does PIV analysis between every frame in input Channel.

It populates self.flows with the u and v components of the velocity vectors as two (smaller) numpy arrays. An additional array, self.flow_coorinates, with the x and y coordinates corresponding to the centers of the search windows in the original input array is also also populated.

Parameters **piv_params** (*dict*) – parameters for the openPIV function extended_search_area_piv

Returns (u_component_array, v_component_array, original_x_coord_array, original_y_coord_array)

Return type tuple

5.3 The validation module

cellocity.validation.**combine_lcorr_and_process_time_to_df**(*lcorrdf*, *processtimedict*, *file_name*, *analyzer_name*)
Performs combination and mutation of correlation length dataframe to simplify visualization and plotting Used by run_5sigma_validation as a helper function.

cellocity.validation.**convertChannel**(*fname*, *finterval=1*)
Converts a mulitiposition MM file to a timelapse Channel with finterval second frame interval.

Parameters

- **fname** – Path to file
- **finterval** – desired frame interval in output `Channel`, defaults to 1 second

Returns Channel

Return type *channel.Channel*

cellocity.validation.**convertMedianChannel**(*fname*, *finterval=1*)
Converts a mulitiposition ome.tif MM file to a timelapse MedianChannel with finterval second frame interval.

Parameters

- **fname** – Path to file
- **finterval** – desired frame interval in output `Channel`, defaults to 1 second

Returns MedianChannel with default 3-frame gliding window

Return type *channel.MedianChannel*

cellocity.validation.**get_data_as_df**(*analyzer*, *analyzername*)
Creates FlowSpeedAnalysis(), AlignmentIndexAnalysis() and IopAnalysis() from a FlowAnalyzer.

Calculates average frame speeds and alignment indexes and returns a DataFrame with the results.

Parameters

- **analyzer** (`analysis.FlowAnalyzer`) – FlowAnalyzer
- **analyzername** (*str*) – Name of FlowAnalyzer

Returns pd.DataFrame containing results and information derived from channel.name

Return type pandas.DataFrame

cellocity.validation.**make_ai_plot**(*df*)
Generates a plot comparing average frame alignment indexes from dataframe

cellocity.validation.**make_channels**(*inpath*)
Creates a list of Channel objects from files in inPath.

Parameters **inpath** – Path

Returns list of Channels

Return type list

cellocity.validation.**make_fb_flow_analyzer**(*ch*)
Creates a FarenbackAnalyzer and performs optical flow calculations with default settings in um/s.

Parameters **ch** – channel.Channel

Returns anlysis.FarenbackAnalyzer

cellocity.validation.**make_iop_plot**(*df*)
Generates a plot comparing instantaneous order parameters

cellocity.validation.**make_lcorr_plot**(*lcorrdf*)
Generates a plot comparing correlation lengths between analyzers and magnifications

cellocity.validation.**make_lcorr_proces_time_plot**(*lcorrdf*)
Generates a bar plot comparing correlation length processing times for the two analyzers.

cellocity.validation.**make_piv_analyzer**(*ch*)
Creates an openPivAnalyzer and performs optical flow calculations with default settings in um/s.

Parameters **ch** – channel.Channel

Returns anlysis.OpenPivAnalyzer

cellocity.validation.**make_proces_time_plot**(*df*)
Generates a bar plot comparing processing times for the two analyzers.

cellocity.validation.**make_speed_plot**(*df*)
Generates a plot comparing average frame flow speeds from dataframe

cellocity.validation.**processAndMakeDf**(*ch_list*)
Creates analyzers from and runs test functions on a list of Channels.

Parameters **ch_list** (*list*) – List of Channel objects

Returns Pandas DataFrame with data from analysis

Return type pandas.DataFrame

cellocity.validation.**run_5sigma_validation**(*inpath*, *outpath*)
Runs the vaildation of the 5sigma analysis on files in inpath, saves figures and a csv in outpath.

Parameters

- **inpath** – input Path
- **outpath** – output Path

Returns None

cellocity.validation.**run_base_validation**(*inpath*, *outpath*)
Runs the basic validation on data in inpath, saves figures and csv files in outpath.

Parameters

- **inpath** – input Path
- **outpath** – output Path

Returns None

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Gary Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 25:120–125, 2000.

[2] John D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[3] Emma Lång, Anna Połeć, Anna Lång, Marijke Valk, Pernille Blicher, Alexander D. Rowe, Kim A. Tønseth, Catherine J. Jackson, Tor P. Utheim, Liesbeth M. C. Janssen, Jens Eriksson, and Stig Ove Bøe. Coordinated collective migration and asymmetric cell division in confluent human keratinocytes without wounding. *Nature communications*, 9(1):3665, 2018. doi:10.1038/s41467-018-05578-7.

[4] Chiara Malinverno, Salvatore Corallino, Fabio Giavazzi, Martin Bergert, Qingsen Li, Marco Leoni, Andrea Disanza, Emanuela Frittoli, Amanda Oldani, Emanuele Martini, Tobias Lendenmann, Gianluca Deflorian, Galina V. Beznoussenko, Dimos Poulikakos, Ong Kok Haur, Marina Uroz, Xavier Trepat, Dario Parazzoli, Paolo Maiuri, Weimiao Yu, Aldo Ferrari, Roberto Cerbino, and Giorgio Scita. Endocytic reawakening of motility in jammed epithelia. *Nature materials*, 16(5):587–596, 2017. doi:10.1038/nmat4848.

[5] Zachary J. Taylor, Roi Gurka, Gregory A. Kopp, and Alex Liberzon. Long-duration time-resolved piv to study unsteady aerodynamics. *IEEE Transactions on Instrumentation and Measurement*, 59(12):3262–3269, 2010. doi:10.1109/TIM.2010.2047149.

[6] Stefan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.

PYTHON MODULE INDEX

C

INDEX

I

M

N

O

P

R

S

T

U