molyso Documentation

Release 1.0.6

Christian C. Sachs

Jan 12, 2021

Contents

See *molyso Readme* for information.

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

Contents:

molyso *mo*ther machine ana*ly*sis *so*ftware

1.1 molyso Readme

1.1.1 Publication

When using *molyso* for scientific applications, cite our publication:

> Sachs CC, Grünberger A, Helfrich S, Probst C, Wiechert W, Kohlheyer D, Nöh K (2016) Image-Based Single Cell Profiling: High-Throughput Processing of Mother Machine Experiments. PLoS ONE 11(9): e0163453. doi: 10.1371/journal.pone.0163453

It is available on the PLoS ONE homepage at DOI: 10.1371/journal.pone.0163453

1.1.2 Example Datasets

You can find example datasets (as used in the publication) deposited at zenodo DOI: 10.5281/zenodo.53764.

1.1.3 Documentation

Documentation can be built using sphinx, but is available online as well at Read the Docs.

1.1.4 License

molyso is free/libre open source software under the 2-clause BSD License. See *License*

1.1.5 Prerequisites

molyso needs Python 3, if you don't have a Python installation or are not familiar with installing packages from source, it is suggested that you use the Anaconda Python distribution, available for Windows, Linux and macOS.

1.1.6 Ways to install molyso

There are different ways to install molyso, for ease of use it is suggested to use the Anaconda Python distribution and the conda package manager. Alternatively, you can use molyso inside a Docker container, see the Docker section near the end.

1.1.6.1 With Anaconda

```
> conda config --add channels conda-forge
> conda config --add channels modsim

> conda install -y molyso
```

1.1.6.2 Alternatively, manually from github

```
> git clone https://github.com/modsim/molyso
> cd molyso
> python setup.py install --user
```

1.1.7 First Steps

molyso is packaged as a Python module, to run it, just use:

```
> python -m molyso
```

And you will be greeted by the help screen of molyso:

```
    \   /\  /\  /                          -------------------------
     | | |O| | |    molyso                  Developed  2013 - 2021 by
     | | | | |O|                            Christian   C.  Sachs  at
     |O| |O| |O|    MOther    machine       ModSim / Microscale Group
     \_/ \_/ \_/    anaLYsis SOftware       Research  Center  Juelich
```

(continues on next page)

(continued from previous page)

```
    ----------------------------------------------------------------------
    If you use this software in a publication, cite our paper:


    Sachs CC, Grünberger A, Helfrich S, Probst C, Wiechert W, Kohlheyer D, Nöh K␣
↪(2016)
    Image-Based Single Cell Profiling:
    High-Throughput Processing of Mother Machine Experiments.
    PLoS ONE 11(9): e0163453. doi: 10.1371/journal.pone.0163453

    ----------------------------------------------------------------------

usage: __main__.py [-h] [-m MODULES] [-p] [-gt GROUND_TRUTH] [-ct CACHE_TOKEN]
                   [-tp TIMEPOINTS] [-mp MULTIPOINTS] [-o TABLE_OUTPUT]
                   [-ot TRACKING_OUTPUT] [-nb] [-cpu MP] [-debug] [-do] [-nci]
                   [-cfi] [-ccb CHANNEL_BITS] [-cfb CHANNEL_FLUORESCENCE_BITS]
                   [-q] [-nc [IGNORECACHE]] [-nt] [-t TUNABLES]
                   [-s TUNABLE_LIST TUNABLE_LIST] [-pt] [-rt READ_TUNABLES]
                   [-wt WRITE_TUNABLES]
                   input

molyso: MOther machine anaLYsis SOftware

positional arguments:
  input                 input file

optional arguments:
  -h, --help            show this help message and exit
  -m MODULES, --module MODULES
  -p, --process
  -gt GROUND_TRUTH, --ground-truth GROUND_TRUTH
  -ct CACHE_TOKEN, --cache-token CACHE_TOKEN
  -tp TIMEPOINTS, --timepoints TIMEPOINTS
  -mp MULTIPOINTS, --multipoints MULTIPOINTS
  -o TABLE_OUTPUT, --table-output TABLE_OUTPUT
  -ot TRACKING_OUTPUT, --output-tracking TRACKING_OUTPUT
  -nb, --no-banner
  -cpu MP, --cpus MP
  -debug, --debug
  -do, --detect-once
  -nci, --no-channel-images
  -cfi, --channel-fluorescence-images
  -ccb CHANNEL_BITS, --channel-image-channel-bits CHANNEL_BITS
  -cfb CHANNEL_FLUORESCENCE_BITS, --channel-image-fluorescence-bits CHANNEL_
↪FLUORESCENCE_BITS
  -q, --quiet
  -nc [IGNORECACHE], --no-cache [IGNORECACHE]
  -nt, --no-tracking
  -t TUNABLES, --tunables TUNABLES
  -s TUNABLE_LIST TUNABLE_LIST, --set-tunable TUNABLE_LIST TUNABLE_LIST
  -pt, --print-tunables
  -rt READ_TUNABLES, --read-tunables READ_TUNABLES
  -wt WRITE_TUNABLES, --write-tunables WRITE_TUNABLES

error: the following arguments are required: input
```

There are three modes of operation, batch processing, interactive viewer, and ground truth generation. The most important part for routine use is batch processing, which will process a whole file or selected time/multi points and produce tabular output and/or tracking visualizations. The interactive viewer can be used to show channel and cell detection on the given dataset, as a first step to check if the settings are applicable. The ground truth viewer is more of a tool for verification of results, the kymograph of a preanalyzed dataset can be visualized *without*

tracking, and individual cell generations can be marked manually, yielding a growth rate which can be compared to the automatic analysis.

To start the interactive viewer, just call molyso without any other parameters:

```
> python -m molyso dataset.ome.tiff
```

To start batch processing, run molyso with the `-p` option. Give an output file for tabular output with `-o` and/or an output directory for individual tracked kymographs with `-ot`.

Note: While OME-TIFF file contain calibration of time and voxel size, simple `.tif` files may not, you can tell molyso manually about the calibration by adding comma-delimited parameters after the file name (followed by a question mark): Example:

```
> python -m molyso "filename.tif?interval=300,calibration=0.08"
```

Supported are among others: the acquisition `interval` (seconds), and the pixel size `calibration` in um per pixel. Some older files may have incorrectly labeled axes, since molyso expects the time axis to be correctly labeled, it might be necessary to reorder the axes, this can be done on the fly, by passing e.g. `?swap_axes=Z.T`. Don't forget to escape/quote the ? in the command line.

```
> python -m molyso dataset.ome.tiff -p -o results.txt -ot dataset_tracking
```

molyso writes cache files in the current directory which contain temporary analysis results. If you want to re-generate tabular output *e.g.*, those files will be read in and already performed analysis steps will be skipped. They are used as well, to show the kymograph for ground truth data mode. They can be kept if you plan any of the mentioned steps, if you are finished with an analysis, they can be deleted as well.

Once *molyso* has run, you will need to post-process the data to extract the information you're interested in. Take a look at the Jupyter/IPython Notebooks.

1.1.8 Docker

Docker is a containerization platform allowing for applications to be run with bundled dependencies without explicit installation steps.

You can use the following commands to run molyso in lieu of the aforementioned calls, e.g. for analysis:

```
> docker run --tty --interactive --rm --volume "`pwd`:/data" --user `id -u` modsim/
↪molyso -p <parameters ...>
```

And to run interactive mode (display on local X11, under Linux):

```
> docker run --tty --interactive --rm --volume "`pwd`:/data" --user `id -u` --env␣
↪DISPLAY=$DISPLAY --volume /tmp/.X11-unix:/tmp/.X11-unix modsim/molyso
↪<parameters ...>
```

Docker usage has just been tested with Linux host systems.

1.1.9 Third Party Licenses

Note that this software contains the following portions from other authors, under the following licenses (all BSD-flavoured):

molyso/generic/otsu.py:

> **functions threshold_otsu and histogram by the scikit-image team, licensed BSD (see file head).**
> Copyright (C) 2011, the scikit-image team

1.2 License

1.2.1 The 2-clause BSD License

1.3 Tunables

1.3.1 Introduction

Tunables are configuration parameters for the analysis. You can either let *molyso* read or write all tunables from/to a file, or you can set individual tunables per command line. Note that tunables are defined where they are used, and to *collect* all tunables, a typical run has to be performed (use -cpu 0 to disable parallelism!).

Tunables are are read/written in JSON. JSON is used as well to set tunables on the command line, e.g.:

```
> python -m molyso -t '{"cells.empty_channel.skipping":true}'
```

See below for a table of tunables. Note that for most data sets it is not necessary to modify tunables, and their particular action is best understood by looking up their usage in the source code ...

Various tunables will as well affect processing speed.

1.3.2 Table of Tunables

Name	Default	Type	Description
cells.empty_channel.skipping	False	bool	For empty channel detection, whether it is enabled.
cells.empty_channel.skipping.outlier_times_sigma	2.0	float	For empty channel detection, maximum sigma used for thresholding the profile.
cells.extrema.order	15	int	For cell detection, window width of the local extrema detector.
cells.filtering.maximum_brightness	0.5	float	For cell detection, maximum brightness a cell may have.
cells.filtering.minimum_prominence	10.0	float	For cell detection, minimum prominence a cell must have.
cells.minimal_length.in_mu	1.0	float	The minimal allowed cell size (Smaller cells will be filtered out).
cells.otsu_bias	1.0	float	Bias factor for the cell detection Otsu image.
cells.smoothing.length	10	int	Length of smoothing Hamming window for cell detection.
channels.horizontal.fft_oversampling	8	int	For channel detection, FFT oversampling factor.
channels.horizontal.noise_suppression_factor.lower	0.1	float	For channel detection, lower profile, noise reduction, reduction factor.
channels.horizontal.noise_suppression_factor.upper	0.1	float	For channel detection, upper profile, noise reduction, reduction factor.
channels.horizontal.noise_suppression_range.lower	0.5	float	For channel detection, lower profile, noise reduction, reduction range.
channels.horizontal.noise_suppression_range.upper	0.5	float	For channel detection, upper profile, noise reduction, reduction range.
channels.horizontal.profile_smoothing_width.lower	5	int	For channel detection, lower profile, smoothing window width.
channels.horizontal.profile_smoothing_width.upper	5	int	For channel detection, upper profile, smoothing window width.
channels.horizontal.threshold_factor	0.2	float	For channel detection, threshold factor for l/r border determination.
channels.vertical.alternate.delta	5	int	For channel detection (alternate, vertical), acceptable delta.
channels.vertical.alternate.fft_smoothing_width	3	int	For channel detection (alternate, vertical), spectrum smoothing width.
channels.vertical.alternate.split_factor	60	int	For channel detection (alternate, vertical), split factor.
channels.vertical.method	alternate	str	For channel detection, vertical method to use (either alternate or recursive).
colors.cell	#005b82	str	For debug output, cell color.
colors.channel	#e7af12	str	For debug output, channel color.
colors.visualization.track.alpha	0.3	float	Track alpha for visualization.
colors.visualization.track.color	#005B82	str	Track color for visualization.
colors.visualization.track.random	1	int	Randomize tracking color palette?
colors.visualization.track.random.seed	314159265	int	Random seed for tracking visualization.
orientation-detection.strips	10	int	Number of strips for orientation correction.
tracking.empty_channel_filtering.minimum_mean_cells	2.0	float	For empty channel removal, minimum of cell mean per channel.

1.4 molyso2vizardous

You can convert *molyso*'s tabular output format to PhyloXML/MetaXML lineage trees for viewing and analysis with *Vizardous* see:

> Helfrich, S. *et al.*, **2015**. "Vizardous: Interactive Analysis of Microbial Populations with Single Cell Resolution" *Bioinformatics* (Oxford, England). DOI: 10.1093/bioinformatics/btv468

You can download *Vizardous* at https://github.com/modsim/vizardous .

The appropriate tool is embedded in molyso in the molyso.util.molyso2vizardous package.

```
> python3 -m molyso.util.molyso2vizardous

usage: __main__.py [-h] [-o OUTPUT] [-d MINIMUM_DEPTH] [-q] input

molyso2vizardous molyso-tabular data format to Vizardous metaXML/phyloXML
converter

positional arguments:
  input                 input file

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
  -d MINIMUM_DEPTH, --minimum-depth MINIMUM_DEPTH
  -q, --quiet

error: the following arguments are required: input
```

```
> python3 -m molyso.util.molyso2vizardous results.txt
```

The tool will then generate many individual files for each found track (you can filter out too short tracks by using the -d MINIMUM_DEPTH option). Note that the internal XML representation is very memory consuming.

1.5 Example Jupyter Notebooks

In the folder *examples*, there are currently two Jupyter/IPython notebooks stored:

- *Example Analysis.ipynb* which contains some possible downstream processing, turning the tabular data from *molyso* into graphs.
- *Example molyso Embedding.ipynb* which contains a little peek into using *molyso* as a library.

Contents

The notebooks are embedded into the documentation in a read-only manner. You can see them below.

1.5.1 Example Analysis

This Jupyter/IPython notebook explains how to post-process the tabular tab-separated value format produced by *molyso* to generate various graphs.

If you are unfamiliar with IPython and the NumPy/SciPy/matplotlib/pandas... Stack, we'd suggest you first read a bit about it. We think it's definitely worth giving it a try, as it is a greatly versatile tool for scientific programming endeavours.

This notebook expects a prepared data file in the current working directory.

An example file is shipped with this notebook, it was created by calling molyso:

```
python3 -m molyso MM_Cglutamicum_D_aceE.ome.tiff -p -o MM_Cglutamicum_D_aceE_
↪results.tsv
```

The file format is a tab separated table format, which can easily be opened with Pandas.

See the following explanation of the table columns:

Column name	Contents
about_to_divide	Whether this cell is about to divide, *i.e.*, this is the last occurrence of this cell
cell_age	The cell age [h], relative to its "birth"
cellxposition	The horizontal position [µm] of the cell (*i.e.*, of the channel)
cellyposition	The vertical position [µm] of the cell within the channel
channel_average_cells	The average count of cells detected in this channel
channel_in_multipoint	The position number of the cell's channel within this multipoint position
channel_orientation	A heuristic result whether high or low y positions represent the open end
channel_width	The width of the cell's channel
division_age	The age [h] at which the cell will divide
elongation_rate	The elongation rate [µm·s^{1}] of the cell
fluorescence_*n*	The mean fluorescence of the cell (with background subtracted)
fluorescence_background_*n*	The background fluorescence of the cell's image
fluorescence_count	The count of fluorescences present in the dataset. The other fluorescence_*_*n* fields occurr dependent on this number.
fluorescence_raw_*n*	The mean raw fluorescence value of the cell
fluorescence_std_*n*	The standard deviation of the fluorescence of the cell
length	The cell length [µm]
multipoint	The multipoint number of the frame of the cell
timepoint	The timepoint [s] of the cell sighting
timepoint_num	The timepoint number (within the dataset) of the cell sighting
uid_cell	A unique id for this tracked cell
uid_parent	A unique id for the cell's parent cell
uid_thiscell	A unique id for this particular cell sighting
uid_track	A unique id for origin (the whole tracking from one start)

```
file_name = 'MM_Cglutamicum_D_aceE_results.tsv'
```

```
# Some general setup routines
%matplotlib inline
%config InlineBackend.figure_formats=['svg']
import pandas
import numpy
from matplotlib import pylab
pandas.options.display.max_columns = None
pylab.rcParams['figure.figsize'] = (10, 6)
pylab.rcParams['svg.fonttype'] = 'none'
pylab.rcParams['font.sans-serif'] = ['Arial']
pylab.rcParams['font.family'] = 'sans-serif'
try:
```

(continues on next page)

(continued from previous page)

```
    import seaborn
except ImportError:
    print("Optional dependency: seaborn to pretty up the graphs.")
```

```
# we first open the file via Pandas, it is formatted so that it can be read with␣
↪the read_table command.

results = pandas.read_table(file_name)

# let's take a sneak peek into the file:
results.head()
```

```
# Let's take a look at the growth rate.
# Therefore, we take a look at all division events:

division_events = results.query('about_to_divide == 1')

print("We found %d division events (out of %d overall cell sightings)" %␣
↪(len(division_events), len(results),))
```

```
We found 1165 division events (out of 23236 overall cell sightings)
```

```
pylab.title('Scatter plot of detected division events')
pylab.ylabel('Division time [h]')
pylab.xlabel('Experiment time [h]')
pylab.scatter(division_events.timepoint / (60.0*60.0), division_events.division_
↪age)
```

```
<matplotlib.collections.PathCollection at 0x7fc000aea240>
```

As you can see, the points are quite nicely crowded in a meaningful range, with some outliers. As a reminder, the dataset was acquired with a 15 min interval, which produces quite some error.

Let's look into a unified growth rate ...

```
division_events_on_first_day = results.query('about_to_divide == 1 and timepoint <␣
↪24*60*60')

doubling_times = numpy.array(division_events_on_first_day.division_age)

print("Unfiltered growth rate on first day μ=%f" % (numpy.log(2)/doubling_times.
↪mean(),))
```

```
Unfiltered growth rate on first day μ=0.483987
```

That way, the data contains quite some outliers, let's remove them by applying some biological prior knowledge:

```
mu_min = 0.01
mu_max = 1.00

filtered_doubling_times = doubling_times[
    ((numpy.log(2)/mu_min) > doubling_times) & (doubling_times > (numpy.log(2)/mu_
↪max))
]

print("Filtered growth rate on first day μ=%f" % (numpy.log(2)/filtered_doubling_
↪times.mean(),))
```

```
Filtered growth rate on first day μ=0.403989
```

Now, how do we generate an overall growth rate graph from the scattered points? We use the simple moving average to unify many points into a single points (over time). [And group the points by their timepoints to have a measure if enough division events occured within a certain time frame. There are multiple possible approaches here, and if precise μ is desired, the graph should be based on filtered data as well!]

```
#division_events = division_events.sort('timepoint')
division_events = division_events.query('timepoint < (50.0 * 60.0 * 60.0)')

grouped_division_events = division_events.groupby(by=('timepoint_num',))

window_width = 25

sma_division_age = pandas.rolling_mean(numpy.array(grouped_division_events.mean().
↪division_age), window_width)
sma_time = pandas.rolling_mean(numpy.array(grouped_division_events.mean().
↪timepoint), window_width)
sma_count = pandas.rolling_mean(numpy.array(grouped_division_events.count().
↪division_age), window_width)
sma_division_age[sma_count < 5] = float('nan')

t = sma_time / 60.0 / 60.0
mu = numpy.log(2)/sma_division_age

pylab.title('Growth graph')
pylab.xlabel('Experiment time [h]')
pylab.ylabel('Growth rate μ [h^1]')
pylab.plot(t, mu)
pylab.ylim(0, 0.6)
pylab.xlim(0, 60)
pylab.show()
```

```
fluor = results.query('fluorescence_0 == fluorescence_0')  # while the example␣
↪dataset does not contain nans, other data might
fluor = fluor.groupby(by=('timepoint_num'))

fluor_time = pandas.rolling_mean(numpy.array(fluor.timepoint.mean()), window_
↪width)  / (60.0*60.0)
fluor_value = pandas.rolling_mean(numpy.array(fluor.fluorescence_0.mean()), window_
↪width)

pylab.title('Growth and fluorescence graph')
pylab.xlabel('Experiment time [h]')
pylab.ylabel('Growth rate μ [h^1]')
pylab.ylim(0, 0.6)
pylab.plot(t, mu)
pylab.twinx()
pylab.ylabel('Fluorescence [a.u.]')
pylab.plot(0, 0, label='μ')  # to add a legend entry
pylab.plot(fluor_time, fluor_value, label='Fluorescence', color='yellow')
pylab.xlim(0, 60)
pylab.legend()
pylab.show()
```

Let's look into some single cell data, *e.g.*, cell length or fluorescence (note that different timepoints are used then in the paper):

```
# Dividing cells can be identified by the about_to_divide == 1 flag,
# cells, which resulted from a division have cell_age == 0

pre_division = results.query('about_to_divide==1')
post_division = results.query('cell_age==0')

pylab.subplot(2,2,1)
pylab.title('Cell lengths pre/post-division')
pylab.xlabel('Length [µm]')
pylab.ylabel('Occurrence [#]')
pre_division.length.hist(alpha=0.5, label='Pre-division')
post_division.length.hist(alpha=0.5, label='Post-division')
pylab.legend()
pylab.subplot(2,2,2)
pylab.title('Cell lengths boxplot')
pylab.ylabel('Length [µm]')
pylab.boxplot([pre_division.length, post_division.length], labels=['Pre-division',
↪'Post-division'])
pylab.show()
```

```
fluor = results.query('fluorescence_0 == fluorescence_0')  # while the example␣
↪dataset does not contain nans, other data might
pylab.title('Fluorescences pre/post-media change')
pylab.xlabel('Fluorescence [a.u.]')
pylab.ylabel('Occurrence [#]')
fluor.query('timepoint < 24*60*60').fluorescence_0.hist(alpha=0.5, label='Pre-
↪media change')
# 6h gap for the bacteria to start production
fluor.query('timepoint > 30*60*60').fluorescence_0.hist(alpha=0.5, label='Post-
↪media change')
pylab.legend()
pylab.show()
```

That's it so far. We hope this notebook gave you some ideas how to analyze your data.

1.5.2 Example molyso Embedding

This example should give a brief overview how *molyso* can easily be called and embedded. As of now, only the steps until after the cell detection are to be performed.

For the example, a test image shipped with *molyso* will be used.

```
# Some general setup routines
%matplotlib inline
%config InlineBackend.figure_formats=['svg']
import numpy
from matplotlib import pylab
pylab.rcParams.update({
    'figure.figsize': (10, 6),
    'svg.fonttype': 'none',
    'font.sans-serif': 'Arial',
    'font.family': 'sans-serif',
    'image.cmap': 'gray',
})
```

```
# the test image can be fetched by the `test_image` function
# it is included in molyso primarily to run testing routines
```

(continues on next page)

(continued from previous page)

```
from molyso.test import test_image
pylab.imshow(test_image())
```

```
<matplotlib.image.AxesImage at 0x7f09dbf04f98>
```

```
# the central starting point of the molyso highlevel interface
# is the Image class

from molyso.mm.image import Image
image = Image()
image.setup_image(test_image())

# as a first test, let's run the autorotate routine, which will
# automatically correct the rotation detected

image.autorotate()
print("Detected angle: %.4f°" % (image.angle,))
pylab.imshow(image.image)
```

```
Detected angle: -1.5074°
```

```
<matplotlib.image.AxesImage at 0x7f09c8443e48>
```

```
# the next two functions call the low-level steps
# therefore, while not much is to see here, ...
# the magic happens behind the curtains

image.find_channels()
image.find_cells_in_channels()

from molyso.debugging.debugplot import inject_poly_drawing_helper
inject_poly_drawing_helper(pylab)

# embedded debugging functionality can be used
# to produce an image with cells and channels drawn as overlay
image.debug_print_cells(pylab)
```

```
# let's look into an important part of the Image-class

# the channels member, which supports the iterator interface
# ... therefore, we call it as parameter to list()
# to get a list of channels

channels = list(image.channels)

# and print some info about it (the .cells member works analogously)
print("The first channel contains: %d cells." % (len(channels[0].cells),))
print("The third channel contains: %d cells." % (len(channels[2].cells),))
```

```
The first channel contains: 0 cells.
The third channel contains: 4 cells.
```

```
# connectivity to original image data remains,
# as long as it is not removed (due to memory/disk-space consumption reasons)

channel = channels[2]
pylab.imshow(channel.channel_image)
```

```
<matplotlib.image.AxesImage at 0x7f09c8369898>
```

```
# lets look into the channel's cells ...

print(list(channel.cells))
```

```
[<molyso.mm.cell_detection.Cell object at 0x7f09d9e46f48>, <molyso.mm.cell_
↪detection.Cell object at 0x7f09d9dd6248>, <molyso.mm.cell_detection.Cell object␣
↪at 0x7f09d9e35d88>, <molyso.mm.cell_detection.Cell object at 0x7f09d9e151c8>]
```

```
# the last call did not really advance our insights ...
# let's take a look at some properties of the cell:
# *local_*top and *local_*bottom ...
# which coincide witht the pixel positions within the channel image
# Note: there is a top and bottom member as well,
# but these values include the total offset of the channel within the image!

for n, cell in enumerate(channel.cells):
    print("Cell #%d from %d to %d" % (n, cell.local_top, cell.local_bottom,))
```

```
Cell #0 from 109 to 140
Cell #1 from 142 to 169
Cell #2 from 171 to 215
Cell #3 from 217 to 255
```

```
# again, connectivity to the image data remains ...
# lets show all individual cell images of that channel

# long line just to prettify the output ...
from functools import partial
next_subplot = partial(pylab.subplot, int(numpy.sqrt(len(channel.cells)))+1,␣
↪int(numpy.sqrt(len(channel.cells))))

for n, cell in enumerate(channel.cells):
    next_subplot(n+1)
    pylab.title('Cell #%d' % (n,))
    pylab.imshow(cell.cell_image)

pylab.tight_layout()
```

This was only a brief overview of some basic functionality. It might get expanded in the future. For now, if you'd like to get deeper insights on the working of *molyso*, I'd like to ask you to study the source files.

```
# PS: You can as well turn on Debug printing within IPython to get more insight on␣
↪the internals
from molyso.debugging import DebugPlot
DebugPlot.force_active = True
DebugPlot.post_figure = 'show'

image.find_channels()
image.find_cells_in_channels()
```

1.6 molyso Developer Documentation

See README/and documentation for end user information. See *License* for license information. (Or string `molyso.__license__`).

A short starting point to understanding *molyso*'s internal structure:

The *molyso.mm* module contains the Mother Machine specific code, as well as the main function (see high-level.py). Analysis code is basically split into two levels: an OOP representation of the data analyzed, as well as some core functions (functional) which perform individual processing step.

E.g., a *molyso.mm.channel_detection.Channel* class calls a *molyso.mm.cell_detection.find_cells_in_channel()* function, which returns mere numbers, of which the class constructs *molyso.mm.cell_detection.Cell* objects.

molyso.generic contains a mix of library functions necessary to perform these tasks, coarsely these can be separated into signal processing etc. functionality, and general utility functions.

molyso.debugging contains the DebugPlot class, a thin abstraction layer over matplotlib which allows for conditional, context manager based plot generation.

molyso.imageio contains the image reading functionality. The MultiImageStack is a base class and factory for opening, multi dimensional images. molyso contains reading code to open plain TIFF and OME-TIFF files, using tifffile.py. MultiImageStack acts as a implementation registry as well, if other formats should be supported, a reader subclass has to be generated, and registered at MultiImageStack. It then can automatically be used by molyso to open the format.

As mentioned earlier, the main function is within highlevel.py. It can divert program flow to two additional modes, which overtake if called: ground truth or interactive mode.

Processing itself is handled within highlevel.py.

Doctests can be called by calling the molyso.test module.

1.6.1 Submodules

1.6.1.1 molyso.debugging package

Debugging Module. Currently only contains DebugPlot, which is imported here for ease of use.

molyso.debugging.debugplot module

documentation

class molyso.debugging.debugplot.**DebugPlot**(**args*, ***kwargs*)
Bases: object

The DebugPlot class serves as an switchable abstraction layer to add plotting debug output facilities.

active = True

context = ''

default_config = {'figure.dpi': 150, 'figure.figsize': (12, 8), 'image.cmap': 'gr

diverted_outputs = {}

exit_handler_registered = False

exit_handlers = []

exp_plot_debugging = False

file_prefix = 'debug'

file_suffix = '.pdf'

files_to_merge = []

force_active = False

classmethod get_context()

Returns

classmethod get_file_for_merge()

Returns

individual_and_merge = False

individual_file_prefix = 'debug'

individual_files = False

classmethod new_pdf_output(*filename*, *collected*)

Parameters

- **filename** –
- **collected** –

classmethod pdfopener(*filename*)
Opens a new PdfPages output, ensuring it will be closed at exit.

Parameters **filename** – filename

Returns

post_figure = 'close'

pp = None

classmethod set_context(***kwargs*)

Parameters **kwargs** –

throw_on_anything = True

exception molyso.debugging.debugplot.**DebugPlotInterruptException**
Bases: Exception

Only for internal usage. Used to interrupt plot drawing early if it is disabled.

class molyso.debugging.debugplot.**DebugPlotInterruptThrower**
Bases: object

Dummy object which raises an exception on every call. To be used when debug mode is deactivated.

molyso.debugging.debugplot.**inject_poly_drawing_helper**(*p*)

Parameters **p** –

molyso.debugging.debugplot.**next_free_filename**(*prefix*, *suffix*)

Parameters

- **prefix** –
- **suffix** –

Returns

raise IOError

molyso.debugging.debugplot.**poly_drawing_helper**(*p*, *coordinates*, ***kwargs*)

Parameters

- **p** –
- **coordinates** –
- **kwargs** –

1.6.1.2 molyso.generic package

Generic module. Collection of various helper functionality, from library routines for signal processing, to os specific workarounds.

molyso.generic.etc module

etc.py contains various helper functions and classes, which are not directly related to data processing.

class molyso.generic.etc.**BaseCache**(*filename_to_be_hashed*, *ignore_cache='nothing'*, *cache_token=None*)

Bases: object

A caching class

contains(*key*)

Parameters **key** –

Returns

static deserialize(*data*)

Parameters **data** –

Returns

get(*key*)

Parameters **key** –

Returns

static prepare_key(*key*)

Parameters **key** –

Returns

static serialize(*data*)

Parameters **data** –

Returns

set(*key*, *value*)

Parameters

- **key** –
- **value** –

Returns

molyso.generic.etc.**Cache**
alias of *molyso.generic.etc.FileCache*

class molyso.generic.etc.**FileCache**(*filename_to_be_hashed*, *ignore_cache='nothing'*, *cache_token=None*)
Bases: *molyso.generic.etc.BaseCache*

A caching class which stores the data in flat files.

build_cache_filename(*suffix*)

Parameters **suffix** –

Returns

contains(*key*)

Parameters **key** –

Returns

get(*key*)

Parameters **key** –

Returns

set(*key*, *value*)

Parameters

- **key** –
- **value** –

class molyso.generic.etc.**NotReallyATree**(*iterable*)
Bases: list

The class is a some-what duck-type compatible (it has a `query` method) dumb replacement for (c)KDTrees. It can be used to find the nearest matching point to a query point. (And does that by exhaustive search...)

query(*q*)
Finds the point which is nearest to `q`. Uses the Euclidean distance.

Parameters **q** – query point

Returns distance, index

Return type float, int

```
>>> t = NotReallyATree([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]])
>>> t.query([1.25, 1.25])
(0.3535533905932738, 0)
>>> t = NotReallyATree([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]])
>>> t.query([2.3535533905932737622, 2.3535533905932737622])
(0.5000000000000002, 1)
```

class molyso.generic.etc.**QuickTableDumper**(*recipient=None*)

Bases: object

Parameters **recipient** –

add(*row*)

Parameters **row** –

delimiter = '\t'

line_end = '\n'

precision = 8

stringify(*obj*)

Parameters **obj** –

Returns

write_list(*the_list*)

Parameters **the_list** –

class molyso.generic.etc.**Sqlite3Cache**(**args*, ***kwargs*)

Bases: *molyso.generic.etc.BaseCache*

A caching class which stores the data in a sqlite3 database.

contains(*key*)

Parameters **key** –

Returns

get(*key*)

Parameters **key** –

Returns

keys()

Returns

set(*key*, *value*)

Parameters

- **key** –
- **value** –

molyso.generic.etc.**bits_to_numpy_type**(*bits*)

Returns a numpy.dtype for one of the common image bit-depths: 8 for unsigned int, 16 for unsigned short, 32 for float

Parameters **bits** –

Returns

molyso.generic.etc.**correct_windows_signal_handlers**()

Dummy for non-windows os.

molyso.generic.etc.**debug_init**()
Initialized debug mode, as of now this means that DebugPlot is set to active (it will produce a debug.pdf)

molyso.generic.etc.**dummy_progress_indicator**()

Returns

molyso.generic.etc.**fancy_progress_bar**(*iterable*)

Parameters **iterable** –

Returns

molyso.generic.etc.**ignorant_next**(*iterable*)
Will try to iterate to the next value, or return None if none is available.

Parameters **iterable** –

Returns

molyso.generic.etc.**iter_time**(*iterable*)
Will print the total time elapsed during iteration of `iterable` afterwards.

Parameters **iterable** (*iterable*) – iterable

Return type iterable

Returns iterable

molyso.generic.etc.**parse_range**(*s*, *maximum=0*)

Parameters

- **s** –
- **maximum** –

Returns

molyso.generic.etc.**prettify_numpy_array**(*arr*, *space_or_prefix*)
Returns a properly indented string representation of a numpy array.

Parameters

- **arr** –
- **space_or_prefix** –

Returns

molyso.generic.etc.**silent_progress_bar**(*iterable*)
Dummy function, just returns an iterator.

Parameters **iterable** (*iterable*) – the iterable to turn into an iterable

Returns iterable

Return type iterable

```
>>> next(silent_progress_bar([1, 2, 3]))
1
```

molyso.generic.fft module

fft.py contains Fourier transform related helper functions mainly it abstracts the Fourier transform itself (currently just passing the calls through to the numpy functions)

molyso.generic.fft.**hires_power_spectrum**(*signal*, *oversampling=1*)
Return a high resolution power spectrum (compare *power_spectrum()*) Resolution is enhanced by feeding the FFT a n times larger, zero-padded signal, which will yield frequency values of higher precision.

Parameters

- **signal** (*numpy.array*) – input signal
- **oversampling** (*int*) – oversampling factor

Returns frequencies and fourier transformed values

Return type tuple(numpy.array, numpy.array)

molyso.generic.fft.**power_spectrum**(*signal*)
Return a power (absolute/real) spectrum (as opposed to the complex spectrum returned by *spectrum()* itself)

Parameters **signal** (*numpy.array*) – input signal

Returns frequencies and fourier transformed values

Return type tuple(numpy.array, numpy.array)

molyso.generic.fft.**spectrum**(*signal*)
Return a raw spectrum (values are complex). Use *power_spectrum()* to directly get real values.

Parameters **signal** (*numpy.array*) – input signal

Returns frequencies and fourier transformed values

Return type tuple(numpy.array, numpy.array)

molyso.generic.fft.**spectrum_bins**(*signal*)
Returns the bins associated with a Fourier transform of a signal of the same length of signal

Parameters **signal** (*numpy.array*) – input signal

Returns frequency bins

Return type numpy.array

molyso.generic.fft.**spectrum_bins_by_length**(*len_signal*)
Returns the bins associated with a Fourier transform of a signal of the length len_signal

Parameters **len_signal** (*int*) – length of the desired bin distribution

Returns frequency bins

Return type numpy.array

molyso.generic.fft.**spectrum_fourier**(*signal*)
Calls the Fourier transform, and returns only the first half of the transform results

Parameters **signal** (*numpy.array*) – input signal

Returns Fourier transformed data

Return type numpy.array

molyso.generic.otsu module

otsu.py contains an implementation of Otsu's thresholding method, taken verbatim from scikit-image.

molyso.generic.otsu.**histogram**(*image*, *nbins=256*)
Return histogram of image.

Unlike *numpy.histogram*, this function returns the centers of bins and does not rebin integer arrays. For integer arrays, each integer value has its own bin, which improves speed and intensity-resolution.

The histogram is computed on the flattened image: for color images, the function should be used separately on each channel to obtain a histogram for each color channel.

Parameters

- **image** (*numpy.ndarray*) – Input image.

- **nbins** (*int, optional*) – Number of bins used to calculate histogram. This value is ignored for integer arrays.

Returns The values of the histogram, the values at the center of the bins.

Return type tuple(numpy.ndarray, numpy.ndarray)

molyso.generic.otsu.**threshold_otsu**(*image*, *nbins=256*)
Return threshold value based on Otsu's method.

Parameters

- **image** (*numpy.ndarray*) – Input image
- **nbins** (*int, optional*) – Number of bins used to calculate histogram. This value is ignored for integer arrays.

Returns Upper threshold value. All pixels intensities that less or equal of this value assumed as foreground.

Return type float

molyso.generic.registration module

registration.py contains a simple 2D image registration function, which registers images by checking the individual horizontal and vertical shifts

molyso.generic.registration.**shift_image**(*image*, *shift*, *background='input'*)

Parameters

- **image** –
- **shift** –
- **background** –

Returns

raise ValueError

molyso.generic.registration.**translation_2x1d**(*image_a=None*, *image_b=None*, *ffts_a=()*, *ffts_b=()*, *return_a=False*, *return_b=False*)

Parameters

- **image_a** –
- **image_b** –
- **ffts_a** –
- **ffts_b** –
- **return_a** –
- **return_b** –

Returns

molyso.generic.rotation module

molyso.generic.signal module

signal processing helper routines

class molyso.generic.signal.**ExtremeAndProminence**

Bases: *molyso.generic.signal.ExtremeAndProminence*

Result of *find_extrema_and_prominence* call.

Variables

- **maxima** –
- **minima** –
- ***signal*** –
- **order** –
- **max_spline** –
- **min_spline** –
- **xpts** –
- **max_spline_points** –
- **min_spline_points** –
- **prominence** –

molyso.generic.signal.**each_image_slice**(*image*, *steps*, *direction='vertical'*)

Parameters

- **image** –
- **steps** –
- **direction** –

Returns

```
>>> list(each_image_slice(np.ones((4, 4,)), 2, direction='vertical'))
[(0, 2, array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]])), (1, 2, array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]]))]
>>> list(each_image_slice(np.ones((4, 4,)), 2, direction='horizontal'))
[(0, 2, array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])), (1, 2, array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]]))]
```

molyso.generic.signal.**find_extrema_and_prominence**(*signal*, *order=5*)

Generates various extra information / signals.

Parameters

- **signal** (*numpy.ndarray*) – input signal
- **order** (*int*) – relative minima/maxima order, see other functions

Returns an ExtremeAndProminence object with various information members

Return type *ExtremeAndProminence*

```
>>> result = find_extrema_and_prominence(np.array([1, 2, 3, 2, 1, 0, 1, 2, 15,␣
↪2, -15, 2, 1]), 2)
>>> result = result._replace(max_spline=None, min_spline=None)  # just for␣
↪doctests
>>> result
ExtremeAndProminence(maxima=array([2, 8]), minima=array([ 5, 10]),␣
↪signal=array([ 1,  2,  3,  2,  1,  0,  1,  2, 15,  2,␣
↪1]), order=2, max_spline=None, min_spline=None, xpts=array([ 0.,  1.,  2.,␣
↪3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]), max_spline_points=array([␣
↪3.    ,  2.4875,  3.    ,  4.3125,  6.2   ,  8.4375, 10.    ]),
```

(continues on next page)

(continued from previous page)

```
       13.0625, 15.    , 16.3875, 17.    , 16.6125, 15.    ]), min_spline_
↪points=array([-9.73055091e-16,  3.38571429e+00,  4.71428571e+00,  4.
↪35000000e+00,
        2.65714286e+00,  5.21804822e-15, -3.25714286e+00, -6.75000000e+00,
       -1.01142857e+01, -1.29857143e+01, -1.50000000e+01, -1.57928571e+01,
       -1.50000000e+01]), prominence=array([ 3.        , -0.89821429, -1.
↪71428571, -0.0375    ,  3.54285714,
        8.4375    , 14.05714286, 19.8125    , 25.11428571, 29.37321429,
       32.        , 32.40535714, 30.        ]))
```

molyso.generic.signal.**find_insides**(*signal*)

Parameters **signal** –

Returns

```
>>> find_insides(np.array([False, False, True, True, True, False, False, True,␣
↪True, False, False]))
array([[2, 5],
       [7, 9]])
```

molyso.generic.signal.**find_phase**(*signal_1=None*, *signal_2=None*, *fft_1=None*, *fft_2=None*, *return_1=False*, *return_2=False*)

Finds the phase (time shift) between two signals. Either signalX or fftX should be set; the FFTs can be returned in order to cache them locally...

Parameters

- **signal_1** (*numpy.ndarray or None*) – first input signal
- **signal_2** (*numpy.ndarray or None*) – second input signal
- **fft_1** (*numpy.ndarray or None*) – first input fft
- **fft_2** (*numpy.ndarray or None*) – second input fft
- **return_1** (*bool*) – whether fft1 should be returned
- **return_2** (*bool*) – whether fft2 should be returned

Returns (shift, (fft1 if return_1), (fft2 if return_2))

Return type tuple

```
>>> find_phase(np.array([0, 1, 0, 0, 0]), np.array([0, 0, 0, 1, 0]))
(2,)
```

molyso.generic.signal.**fit_to_type**(*image*, *new_dtype*)

Parameters

- **image** –
- **new_dtype** –

Returns

```
>>> fit_to_type(np.array([-7, 4, 18, 432]), np.uint8)
array([  0,   6,  14, 255], dtype=uint8)
>>> fit_to_type(np.array([-7, 4, 18, 432]), np.int8)
array([-128, -121, -113,  127], dtype=int8)
>>> fit_to_type(np.array([-7, 4, 18, 432]), np.bool)
array([False, False, False,  True])
>>> fit_to_type(np.array([-7, 4, 18, 432]), np.float32)
array([ -7.,   4.,  18., 432.], dtype=float32)
```

molyso.generic.signal.**horizontal_mean**(*image*)

Calculates the horizontal mean of an image. Note: Image is assumed HORIZONTAL x VERTICAL.

Parameters **image** (*numpy.ndarray*) – input image

Returns

```
>>> horizontal_mean(np.array([[ 1,  2,  3,  4],
...                           [ 5,  6,  7,  8],
...                           [ 9, 10, 11, 12],
...                           [13, 14, 15, 16]]))
array([ 7.,  8.,  9., 10.])
```

molyso.generic.signal.**normalize**(*data*)

normalizes the values in arr to 0 - 1

Parameters **data** – input array

Returns normalized array

```
>>> normalize(np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
>>> normalize(np.array([10, 15, 20]))
array([0. , 0.5, 1. ])
```

molyso.generic.signal.**one_every_n**(*length*, *n=1*, *shift=0*)

Parameters

- **length** –
- **n** –
- **shift** –

Returns

```
>>> one_every_n(10, n=2, shift=0)
array([1., 0., 1., 0., 1., 0., 1., 0., 1., 0.])
>>> one_every_n(10, n=2, shift=1)
array([0., 1., 0., 1., 0., 1., 0., 1., 0., 1.])
>>> one_every_n(10, n=1, shift=0)
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

molyso.generic.signal.**outliers**(*data*, *times_std=2.0*)

Parameters

- **data** –
- **times_std** –

Returns

```
>>> outliers(np.array([10, 9, 11, 40, 8, 12, 14, 7]), times_std=1.0)
array([False, False, False,  True, False, False, False, False])
```

molyso.generic.signal.**relative_maxima**(*signal*, *order=1*)

Parameters

- **signal** –
- **order** –

Returns

```
>>> relative_maxima(np.array([1, 2, 3, 2, 1, 0, 1, 2, 15, 2, -15, 2, 1]), 2)
array([2, 8])
```

molyso.generic.signal.**relative_minima**(*signal*, *order=1*)

Parameters

- **signal** –
- **order** –

Returns

```
>>> relative_minima(np.array([1, 2, 3, 2, 1, 0, 1, 2, 15, 2, -15, 2, 1]), 2)
array([ 5, 10])
```

molyso.generic.signal.**remove_outliers**(*data*, *times_std=2.0*)

Parameters

- **data** –
- **times_std** –

Returns

```
>>> remove_outliers(np.array([10, 9, 11, 40, 8, 12, 14, 7]), times_std=1.0)
array([10,  9, 11,  8, 12, 14,  7])
```

molyso.generic.signal.**simple_baseline_correction**(*signal*, *window_width=None*)

Performs a simple baseline correction by subtracting a strongly smoothed version of the signal from itself.

Parameters

- **signal** – input signal
- **window_width** – smoothing window width

Returns

```
>>> simple_baseline_correction(np.array([10, 11, 12, 11, 10]))
array([-1.        ,  0.375     ,  1.        , -0.375     , -0.96428571])
```

molyso.generic.signal.**threshold_outliers**(*data*, *times_std=2.0*)

removes outliers

Parameters

- **data** –
- **times_std** –

Returns

```
>>> threshold_outliers(np.array([10, 9, 11, 40, 8, 12, 14, 7]), times_std=1.0)
array([10,  9, 11, 20,  8, 12, 14,  7])
```

molyso.generic.signal.**vertical_mean**(*image*)

Calculates the vertical mean of an image. Note: Image is assumed HORIZONTAL x VERTICAL.

Parameters **image** –

Returns

```
>>> vertical_mean(np.array([[ 1,  2,  3,  4],
...                         [ 5,  6,  7,  8],
...                         [ 9, 10, 11, 12],
```

(continues on next page)

(continued from previous page)

```
...                              [13, 14, 15, 16]]))
array([ 2.5,  6.5, 10.5, 14.5])
```

molyso.generic.smoothing module

smoothing.py contains the main smoothing function, which works by convolving a signal with a smoothing kernel, a signals function which acts as a cache for kernels, as well as the hamming_smooth function, which is the only one currently used by external files, providing a simplified interface for smoothing with hamming kernels.

molyso.generic.smoothing.**hamming_smooth**(*signal*, *window_width*, *no_cache=False*)

Smooths a signal by convolving with a hamming window of given width. Caches by the hamming windows by default.

Parameters

- **signal** (*numpy.ndarray*) – input signal to be smoothed
- **window_width** (*int*) – window width for the hamming kernel
- **no_cache** (*bool*) – default *False*, disables caching, *e.g.*, for non-standard window sizes

Returns the smoothed signal

Return type numpy.ndarray

```
>>> hamming_smooth(np.array([0, 0, 0, 0, 1, 0, 0, 0, 0]), 3)
array([0.        , 0.        , 0.        , 0.        , 0.06896552,
       0.86206897, 0.06896552, 0.        , 0.        , 0.        ])
```

molyso.generic.smoothing.**signals**(*function*, *parameters*)

Signal cache helper function. Either retrieves or creates and stores a signal which can be created by calling the given function with the given parameters.

Parameters

- **function** (*callable*) – Window function to be called
- **parameters** (**any*) – Parameters to be passed to the function

Returns function(*parameters)

Return type dependent on function

```
>>> signals(np.ones, 3)
array([1., 1., 1.])
```

molyso.generic.smoothing.**smooth**(*signal*, *kernel*)

Generic smoothing function, smooths by convolving one signal with another.

Parameters

- **signal** (*numpy.ndarray*) – input signal to be smoothed
- **kernel** (*numpy.ndarray*) – smoothing kernel to be used. will be normalized to $\sum = 1$

Returns The signal convolved with the kernel

Return type numpy.ndarray

```
>>> smooth(np.array([0, 0, 0, 0, 1, 0, 0, 0, 0]), np.ones(3))
array([0.        , 0.        , 0.        , 0.        , 0.33333333,
       0.33333333, 0.33333333, 0.        , 0.        , 0.        ])
```

molyso.generic.tunable module

tunable.py contains a tunable (settings to be changed depending on input data) management class

class molyso.generic.tunable.**TunableManager**
Bases: object

Static object handling tunables (that is, parameters which are user-changeable)

Tunables have default values, which must be set as a parameter by the function asking for the tunable. That way, default configuration is inlined, and does not need to be centrally managed. In order to collect all defaults, a typical run of the program has to be performed, and the collected default values to be dumped afterwards.

Variables

- ***defaults*** (*dict*) – Internal map of defaults
- ***current*** (*dict*) – Current tunables, with possible overrides.
- ***force_default*** (*bool*) – Whether to force usage of default values (default: *False*)

current = {}

defaults = {}

descriptions = {}

force_default = False

classmethod get_defaults()
Gets the defaults, which were collected during the calls asking for various tunables.

Returns either the overridden tunable or the default value

Return type dependent on default

```
>>> TunableManager.defaults = {}
>>> value = tunable('my.tunable', 3.1415)
>>> TunableManager.get_defaults()
{'my.tunable': 3.1415}
```

classmethod get_descriptions()
Gets descriptions.

Returns The descriptions.

Return type dict

classmethod get_table()

classmethod get_tunable(*what*, *default*)
Returns either an overridden tunable, or the default value. The result will be casted to the type of default.

Parameters

- **what** (*str*) – tunable to look up
- **default** – default value

Returns either the overridden tunable or the default value

```
>>> tunable('my.tunable', 3.1415)
3.1415
```

classmethod load_tunables(*data*)
Sets the tunables.

Parameters **data** (*dict*) – set of tunables to load

Return type None

```
>>> TunableManager.load_tunables({'foo': 'bar'})
>>> tunable('foo', 'not bar')
'bar'
```

logger = <Logger molyso.generic.tunable.TunableManager (WARNING)>

classmethod set_description(*what*, *description*)
Sets a description for a paremeter.

Parameters

- **what** – parameter to describe
- **description** – description

Returns

molyso.generic.tunable.**tunable**(*what*, *default*, *description=None*)
Syntactic sugar helper function, to quickly get a tunable. Calls: `TunableManager.get_tunable(what, default)`

Parameters

- **what** (*str or unicode*) – tunable to look up
- **default** – default value
- **description** – description

Returns either the overridden tunable or the default value

Return type type(default)

```
>>> tunable('my.tunable', 3.1415)
3.1415
```

1.6.1.3 molyso.imageio package

molyso.imageio.imagestack module

molyso.imageio.imagestack_ometiff module

1.6.1.4 molyso.mm package

MM Module, contains *Mother Machine* analysis specific functionality.

molyso.mm.cell_detection module

documentation

class molyso.mm.cell_detection.**Cell**(*top*, *bottom*, *channel*)
Bases: object

A Cell.

Parameters

- **top** – coordinate of the 'top' of the cell, in channel coordinates
- **bottom** – coordinate of the 'bottom' of the cell, in channel coordinates
- **channel** – Channel object the cell belongs to

bottom
Returns the absolute (on rotated image) coordinate of the cell bottom.

Returns

cell_image
The cell image, cropped out of the channel image.

Returns image

Return type numpy.ndarray

centroid
Returns the (absolute coordinate on rotated image) centroid (2D). :return: :rtype: list

centroid_1d
Returns the (one dimensional) (absolute coordinate on rotated image) centroid. :return: centroid :rtype: float

channel

crop_out_of_channel_image(*channel_image*)
Crops the clel out of a provided image. Used internally for *Cell.cell_image()*, and to crop cells out of fluorescence channel images.

Parameters **channel_image** (*numpy.ndarray*) –

Returns image

Return type numpy.ndarray

length
Returns the cell length.

Returns length

local_bottom

local_top

top
Returns the absolute (on rotated image) coordinate of the cell top.

Returns top

class molyso.mm.cell_detection.**Cells**(*channel*, *bootstrap=True*)
Bases: object

A Cells object, a collection of Cell objects.

cell_type
alias of *Cell*

cells_list

centroids
Returns the centroids of the cells.

Returns centroids

Return type list

channel

clean()
Performs clean-up.

nearest_tree

molyso.mm.cell_detection.**find_cells_in_channel**(*image*)

molyso.mm.cell_detection.**find_cells_in_channel_classic**(*image*)

Parameters **image** –

Returns

molyso.mm.channel_detection module

documentation

class molyso.mm.channel_detection.**Channel**(*image*, *left*, *right*, *top*, *bottom*)

Bases: object

Parameters

- **image** –
- **left** –
- **right** –
- **top** –
- **bottom** –

bottom

Returns

cells

cells_type

alias of *molyso.mm.cell_detection.Cells*

centroid

Returns

channel_image

clean()

Peforms clean up routines.

crop_out_of_image(*image*)

Parameters **image** –

Returns

detect_cells()

Performs Cell detection (by instantiating a Cells object).

get_coordinates()

Returns

image

left

putative_orientation

real_bottom

real_top

right

top

Returns

class molyso.mm.channel_detection.**Channels**(*image*, *bootstrap=True*)
Bases: object

docstring

align_with_and_return_indices(*other_channels*)

Parameters **other_channels** –

Returns

centroids

Returns

channel_type
alias of *Channel*

channels_list

clean()
Performs clean up routines.

find_nearest(*pos*)

Parameters **pos** –

Returns

image

nearest_tree

molyso.mm.channel_detection.**alternate_vertical_channel_region_detection**(*image*)

Parameters **image** –

Returns

molyso.mm.channel_detection.**find_channels**(*image*)
channel finder :param image: :return:

molyso.mm.channel_detection.**horizontal_channel_detection**(*image*)
@param image: @return:

molyso.mm.channel_detection.**vertical_channel_region_detection**(*image*)

Parameters **image** –

Returns

molyso.mm.fluorescence module

molyso.mm.highlevel module

molyso.mm.highlevel_interactive_ground_truth module

documentation

molyso.mm.highlevel_interactive_ground_truth.**interactive_ground_truth_main**(*args*, *tracked_results*)
Ground truth mode entry function.

Parameters

- **args** –
- **tracked_results** –

Returns

raise SystemExit

molyso.mm.highlevel_interactive_viewer module

documentation

molyso.mm.highlevel_interactive_viewer.**interactive_main**(*args*)

Parameters **args** –

Raises **SystemExit** –

molyso.mm.image module

molyso.mm.tracking module

documentation

class molyso.mm.tracking.**TrackedPosition**
Bases: object

A TrackedPosition object contains various CellTracker objects for each channel within a multipoint position, as well as other information associated with the position.

align_channels(*progress_indicator=<callable_iterator object>*)

Parameters **progress_indicator** –

find_first_valid_time()
Finds the first valid time point for the position.

get_tracking_work_size()

Returns

guess_channel_orientation()
Tries to guess the channel orientation. 1 if the closed end ('mother side') is the high coordinates, -1 if low ...

logger

perform_everything(*times*)

Parameters **times** –

Returns

perform_tracking(*progress_indicator=<callable_iterator object>*)

Parameters **progress_indicator** –

remove_empty_channels()
Removes empty channels from the data set.

remove_empty_channels_post_tracking()
Removes empty channels after tracking.

set_times(*times*)

Parameters **times** –

molyso.mm.tracking.**analyse_cell_fates**(*tracker*, *previous_cells*, *current_cells*)

Parameters

- **tracker** –
- **previous_cells** –

- **current_cells** –

Returns

molyso.mm.tracking.**each_k_tracking_tracker_channels_in_results**(*tracking*)

Parameters **tracking** –

molyso.mm.tracking.**each_pos_k_tracking_tracker_channels_in_results**(*inner_tracked_results*)

Parameters **inner_tracked_results** –

molyso.mm.tracking_infrastructure module

This module contains cell tracking infrastructure.

class molyso.mm.tracking_infrastructure.**CellCrossingCheckingGlobalDuoOptimizerQueue**
Bases: object

add_outcome(*cost*, *involved_a*, *involved_b*, *what*)

Parameters

- **cost** –
- **involved_a** –
- **involved_b** –
- **what** –

Returns

perform_optimal()

Returns

class molyso.mm.tracking_infrastructure.**CellTracker**
Bases: object

The CellTracker contains all tracks of a channel.

all_tracked_cells

average_cells
Returns the average count of cells present in this tracked channel.

Returns

get_cell_by_observation(*where*)
Returns the associated cell by its observation.

Parameters **where** –

Returns

is_tracked(*cell*)
Returns whether the cell is tracked.

Parameters **cell** –

Returns

new_cell()
Creates a new TrackedCell object associated with this tracker.

Returns

new_observed_cell(*where*)
Creates a new TrackedCell object, with added observation.

Parameters **where** –

Returns

new_observed_origin(*where*)
Creates a new TrackedCell object and adds it as an origin, with added observation. :param where: :return:

new_origin()
Creates a new TrackedCell object and adds it as an origin.

Returns

origins

tick()
Ticks the clock. Sets the internal timepoint counter forward by one.

timepoints

class molyso.mm.tracking_infrastructure.**TrackedCell**(*tracker*)
Bases: object

Parameters **tracker** –

add_child(*tcell*)

Parameters **tcell** –

Returns

add_children(**children*)

Parameters **children** –

add_observation(*cell*)

Parameters **cell** –

Returns

children

elongation_rates

Returns

parent

raw_elongation_rates

raw_trajectories

seen_as

tracker

trajectories

Returns

ultimate_parent

Returns

molyso.mm.tracking_infrastructure.**to_list**(*x*)

Parameters **x** –

Returns

molyso.mm.tracking_output module

documentation

molyso.mm.tracking_output.**analyze_tracking**(*cells*, *receptor*, *meta=None*)

Parameters

- **meta** –
- **cells** –
- **receptor** –

molyso.mm.tracking_output.**catch_attribute_error**(*what*, *otherwise*)
runs callable 'what' and catches AttributeError, returning 'otherwise' if one occurred :param what: callable :param otherwise: alternate result in case of IndexError :return: result of 'what' or 'otherwise' in case of IndexError

molyso.mm.tracking_output.**catch_index_error**(*what*, *otherwise*)
runs callable 'what' and catches IndexErrors, returning 'otherwise' if one occurred :param what: callable :param otherwise: alternate result in case of IndexError :return: result of 'what' or 'otherwise' in case of IndexError

molyso.mm.tracking_output.**get_object_unique_id**(*obj*)

Parameters **obj** –

Returns

molyso.mm.tracking_output.**iterate_over_cells**(*cells*)

Parameters **cells** –

Returns

molyso.mm.tracking_output.**plot_timeline**(*p*, *channels*, *cells*, *figure_presetup=None*, *figure_finished=None*, *show_images=True*, *show_overlay=True*, *leave_open=False*)

Parameters

- **p** –
- **channels** –
- **cells** –
- **figure_presetup** –
- **figure_finished** –
- **show_images** –
- **show_overlay** –
- **leave_open** –

molyso.mm.tracking_output.**s_to_h**(*s*)
converts seconds to hours :param s: seconds :return: hours

molyso.mm.tracking_output.**s_to_h_str**(*s*, **args*, ***kwargs*)
converts seconds to hours as a rounded string :param s: seconds :return: hours :param s: seconds :return: hours string

molyso.mm.tracking_output.**tracker_to_cell_list**(*tracker*)

Parameters **tracker** –

Returns

1.6.1.5 molyso.test package

Module contents

Python Module Index

m

Index

D

E

F

G

T

U

V

W