napari-{{cookiecutter.plugin_name}} Documentation

{{cookiecutter.full_name}}

Jan 10, 2023

CONTENTS

CHAPTER

ONE

WHAT'S INCLUDED ?

Palmari is a Napari plugin providing tools to process movies of PALM experiments (photo-activated localization microscopy). It provides a **customizable pipeline scheme** with the following features :

- **Get your trajectories in a few clicks** with built-in, ready-to-use processing steps : localizer, tracker, drift corrector, ...
- **Direct visualization of results** to intuitively adjust a pipeline's parameters : visualize the effect of one or the other parameter. Want to test your results' robustness to different processing pipelines ? Palmari stores the localizations and trajectories output from each pipeline so that they can be easily compared afterwards.
- Designed from the start to **process series of PALM acquisitions** using a same pipeline : don't loose time writing scripts to go through all files in a folder. Once your pipeline is ready, process batches of experiments in a few clicks, direclty from Napari.
- Want to take advantage of HPC infrastructure ? Palmari has a Python interface. Its `Experiment` class allows you to keep track of your entire series of acquisitions and stores your processing results. Processing steps rely on Dask, so as to take advantage of multithreading when several cores are available.
- Easily **include your favorite processing steps** in your Palmari pipeline, see *here* for more details on implementations. If it's worth sharing, consider a merge request !

Important: This package is under development, if you wish to contribute, report a bug or suggest an addition, raise an issue or send an email to hverdier@pasteur.fr. Contributions enriching the set of built-in image processing steps, among others, are very welcome !

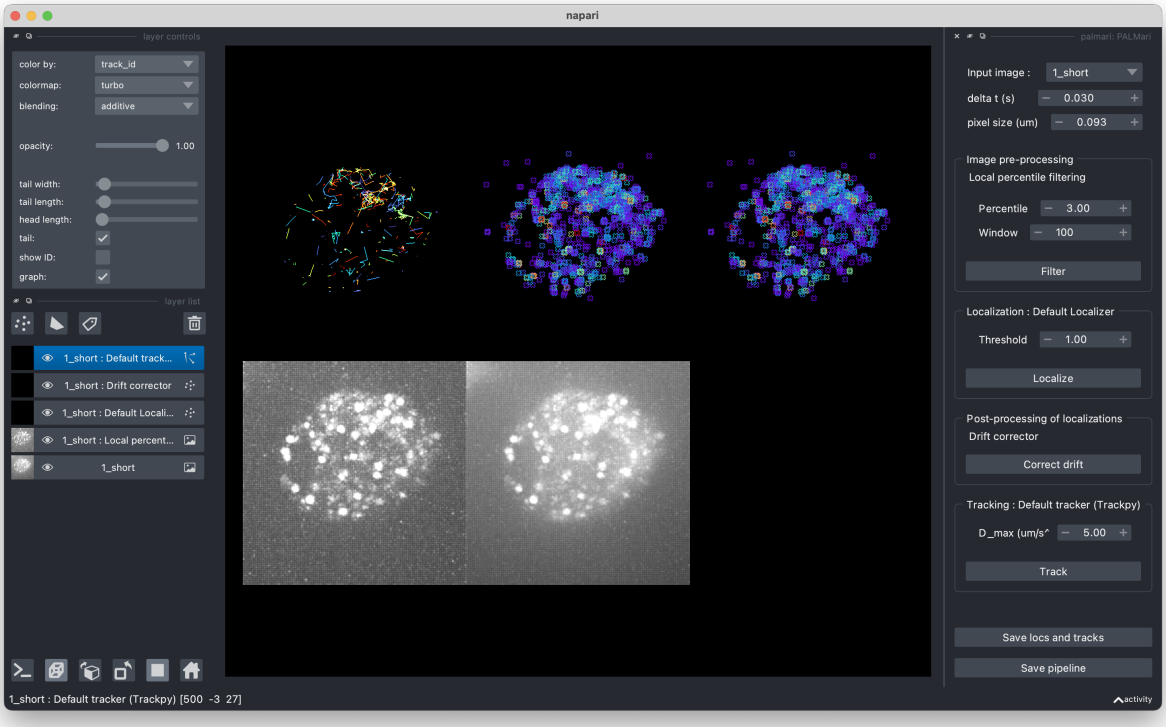


Fig. 1: Visualize processing steps and tweak their parameters with Palmari's interface in Napari.

CHAPTER

TWO

INSTALLATION

Install Palmari either via the Napari plugin manager, or using `pip`:

```
pip install palmari
```

CHAPTER

THREE

CONTENTS

3.1 Napari plugin

3.1.1 Processing a single experiment

To process an microscope recording using Palmari, open Napari and click on `Pulgins > palmari > Run PALMari on file...`. The Napari panel appears (by default, on the right of your screen).

Do not forget, in the "Input" section, to enter the pixel size and exposition time of the recording !

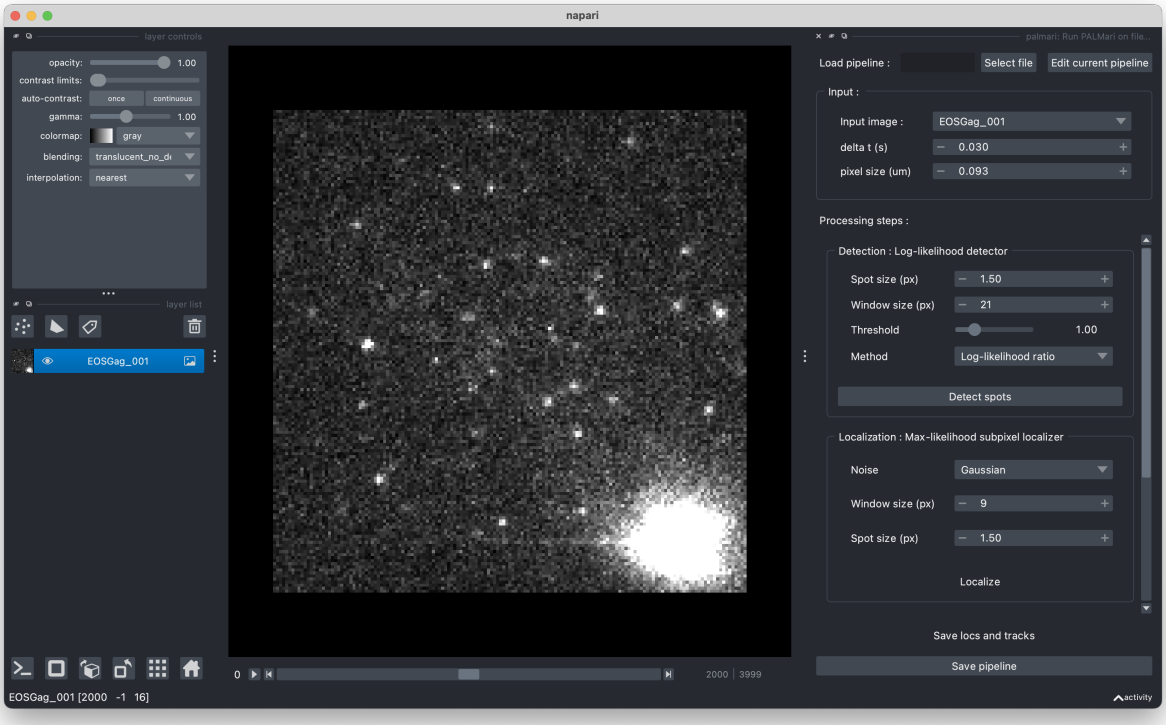


Fig. 1: Palmari panel, with default pipeline.

A standard pipeline (i.e. succession of processing steps) is loaded, with the following steps:

- detection
- sub-pixel localization

- tracking

You can change the parameters of these steps and run them successively.

After processing, you can export the obtained localizations and tracks by clicking on the “save locs and tracks” button !

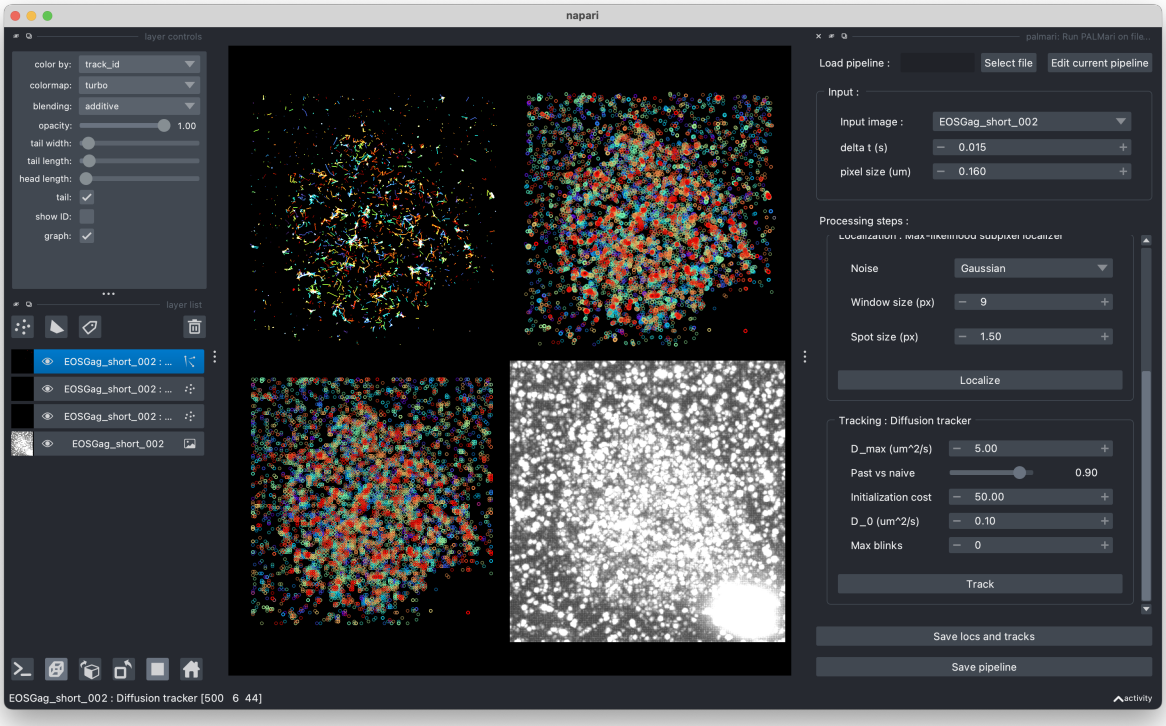


Fig. 2: Look at the intermediate steps using Napari’s grid view.

3.1.2 Customizing the pipeline

Do you want to add a drift correction step, or to use a different tracking algorithm ? Click on “Edit current pipeline” at the top of the panel, to change the steps. See *here* for more information about the available processing steps.

3.1.3 Export and load pipelines

You’re satisfied with your pipeline and want to save it for later ? Export it by clicking on the “Save pipeline” button. It will create a `.yaml` file at a location of your choice. To load en existing pipeline, click on the “Load pipeline / select file” button at the top of the Palmari panel.

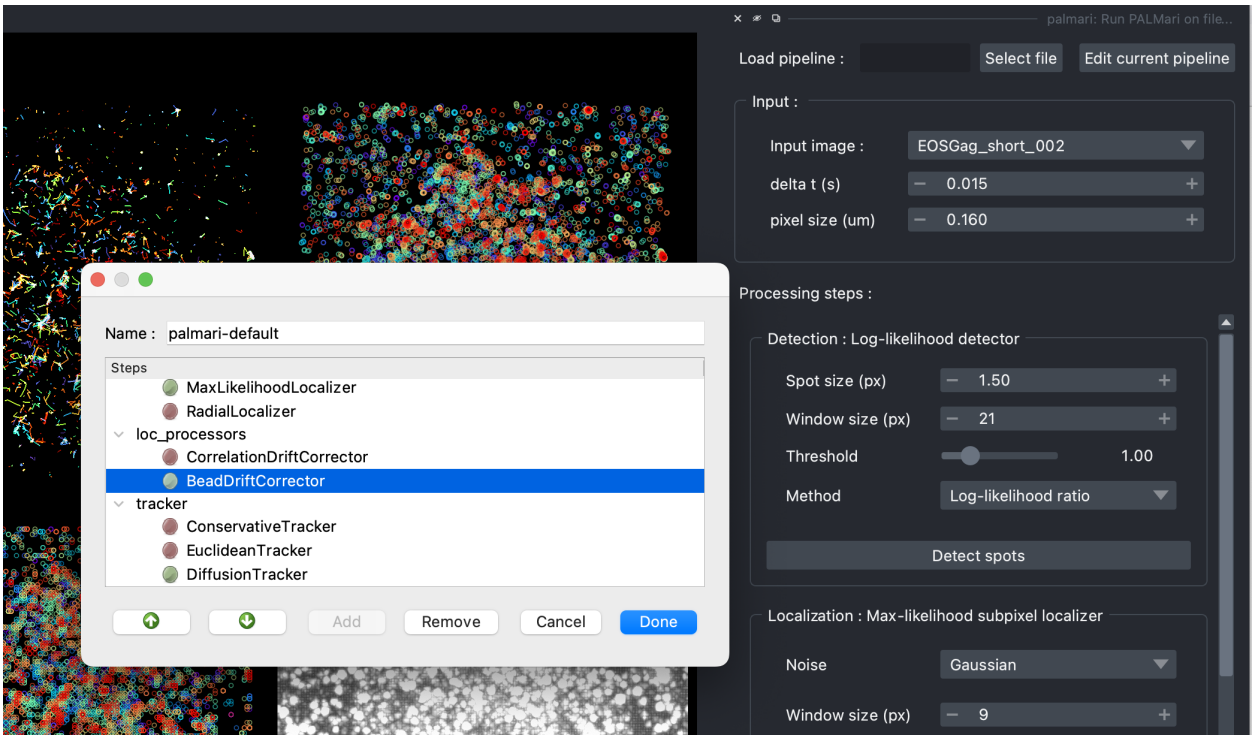


Fig. 3: The pipeline edition window.

3.1.4 Process Batches of experiments

If you have a series of experiments to process, click on `Plugins > palmari > run palmari on folder...`

A panel opens on the right. First, select the desired pipeline's `.yaml` file. Then, choose the location of your experiments and the folder where to export them.

Click on "Process files" and you're all set ! The progress bar will keep you informed of the processing, which might last for several hours, depending on the size of your files.

3.1.5 Loading pipelines with Python

You can also load pipelines programatically using Palmari's Python interface, with `TifPipeline.from_yaml()`.

Conversely, if you've defined your `TifPipeline` in a Python script or notebook, you can visualize its effect on an image using `TifPipeline.open_in_napari()`.

3.2 TIF processing pipeline

3.2.1 The `TifPipeline` class

Palmari provides a customizable pipeline structure for extracting localizations and tracks from PALM movies. These steps are handled by the `TifPipeline` class.

It is very easy to instanciate a default pipeline, containing only the minimal localization and tracking steps with default parameters :

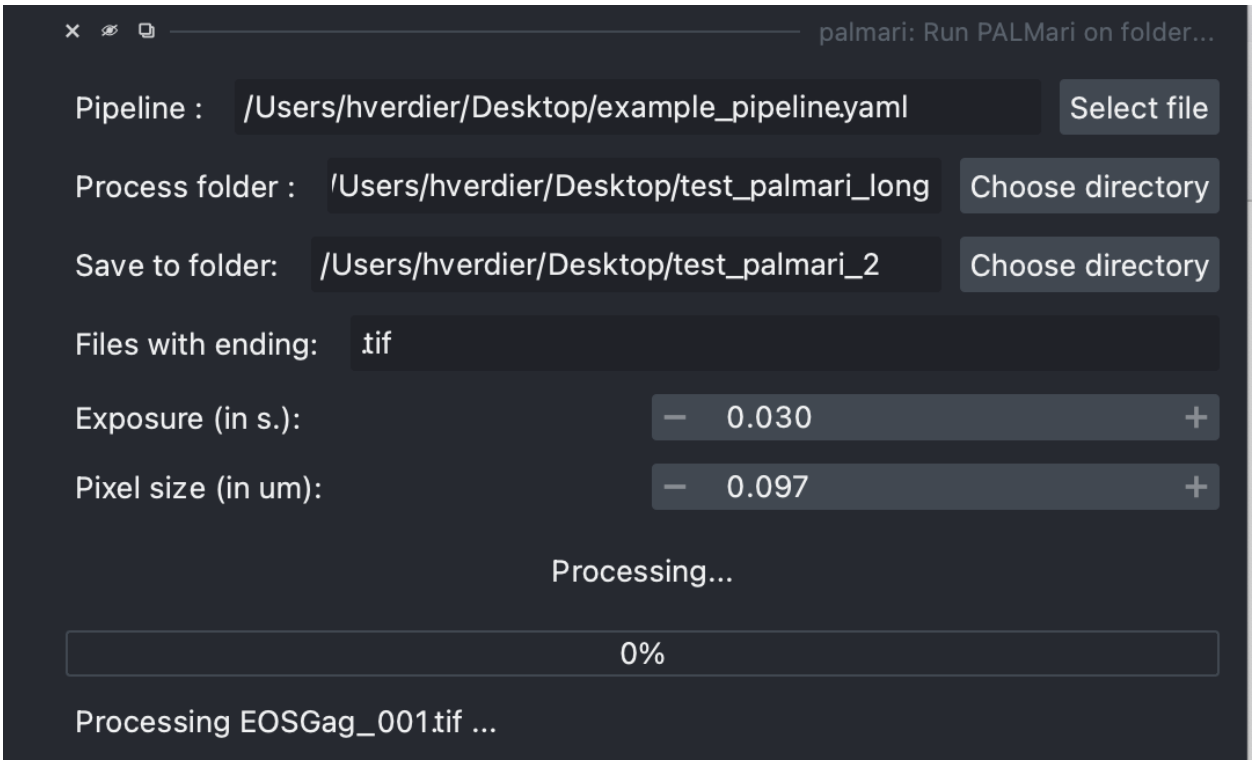


Fig. 4: Process all `.tif` in a folder, directly from Napari.

```
from palmari import TifPipeline

tp = TifPipeline.default_with_name("my_pipeline")
```

A pipeline's name is used when exporting the localizations ot the movies it processes (hey are placed in a folder named after the pipeline). Indeed, you might want to store localizations obtained using various pipelines applied on a same movie, in order to investigate the influence of the image processing step on your findings.

It takes just one line to run a pipeline on a movie or a batch of movies. The `TifPipeline` class uses Dask to take advantage of multithreading when possible and limit the memory footprint of the processing. By default, it will not re-process movies on which it has already been run.

```
tp.process(acq) # for a single acquisition
# or
tp.process(exp) # for all acquisitions of an experiment
```

3.2.2 Processing steps

Each step of the pipeline is an instance of a `ProcessingStep` subclass. Steps are divided in four main categories :

1. Optionally, a few **movie pre-processors** (subclasses of `MoviePreprocessor`). Background removal or denoising steps fall into this category. These steps take as input a movie and output a modified movie.
2. One **detector** (subclass of `Detector`). This step (maximum one per pipeline) takes as input the pre-processed movie and outputs a dataframe containing detected spots.
3. One **sub-pixel localizer** (subclass of `SubpixelLocalizer`). This step refines the precision of the detected spots' localizations
4. Optionally, one might have **localization** processors (subclasses of `LocsProcessor`). These step modify localizations, add columns to the localizations table or discard localizations. They take as input the localizations dataframe as well as the original movie, and output a modified localizations dataframe. This is useful for drift correction, for instance.
5. One **tracker** (subclass of `Tracker`). This step links consecutive localizations of one same particle, by adding an *n* column, correspondoing to the particle's ID, to the localizations dataframe. It takes as input the localizations dataframe.

Order	Type of step	Mandatory	Multiple sub-steps ?	Included
1	Image processing	N	Y	`WindowPercentileFilter`
2	Detector	Y	N	`BaseDetector`
3	Sub-pixel localizer	Y	N	`MaxLikelihoodLocalizer`, `RadialLocalizer`
4	Localizations processing	N	Y	`CorrelationDriftCorrector`, `BeadDriftCorrector`
5	Tracker	Y	Y	`ConservativeTracker`, `EuclideanTracker`, `DiffusionTracker`

In the table, "Mandatory" means that a pipeline must have one such step. On the contrary, non-mandatory steps can be omitted. If a pipeline does not mention any particular class/setting to use for a mandatory step, the default class for this step will be used, with default parameters.

3.2.3 Use built-in steps

Provided processing steps

Palmari comes with a few built-in processing steps, which you can use to compose yout processing pipeline.

- `WindowPercentileFilter` shifts pixel values by removing, for each pixel, the i-th hpercentile taken in a window of surrounding frames. If the pixel's value is lower than the percentile, it is set to 0. This is meant to remove background fluorescence. Parameters are :

  `window_size` : the size of the considered window, in number of frames

  `percentile` : the threshold percentage.

- `Detector` detects spots with a pixel-level precision. Two methods are available:

  `llr` : Perform a **log-likelihood ratio** test for the presence of spots. This is the ratio of likelihood of a Gaussian spot in the center of the subwindow, relative to the likelihood of flat background with Gaussian noise.

  `log` : Detect spots by Laplacian-of-Gaussian filtering, followed by a thresholding step.

These methods, whose implementations were adapted from Quot, require three parameters:

`w` : the size of the window in which the center of the spot will be assumed

`t` : the thresholding level above which a spot is detected.

`sigma` : the diameter of the spots (in pixels)

- `SubpixelLocalizer` refines the localization, using a maximum-likelihood approach. For details about the implementation, see here There are several parameters:

  `method` : a string, equal to `ls_int_gaussian` or `poisson_int_gaussian`, indicating the assumed distribution of noise.

  `window_size` : the size of the region, around the detected spot, on which the fit happens

  `sigma` : the sigma parameter of the fitted Gaussian PSF

- `RadialLocalizer` refines the localization, using a faster but less precise approach based on radial symmetry. It has one parameter:

  `window_size` : the size of the square (in pixels) used to estimate the center of radial symmetry.

- `CorrelationDriftCorrector` corrects drift using time correlation between densities computed on time-wise binned localizations. Densities are simply estimated using 2D histograms. One drift vector is estimated per time bin, and the level of drift applied to each point is determined by interpolation. Parameters are :

  `max_n_bins` : maximum number of time bins.

  `min_n_locs_per_bin` : minimum number of localizations to form a time bin.

- `BeadDriftCorrector` corrects drift using a bead's position. The bead is detected in the image (brightest spot) and its position over time is smoothed using a Gaussian filter. It only requires one parameter:

  `sigma` : the diameter of the bead (in pixels).

- `ConservativeTracker` tracks localizations using the Trackpy package. No missing localization is allowed (trajectories are cut if one point is missing). If there are two candidate localizations inside the search radius, the trajectory is cut as well. It takes one argument :

  `max_diffusivity` : estimation of the maximum diffusion coefficient, which defines the maximum distance between two successive localizations (search radius) : sqrt{4 D Delta t}

- `DiffusionTracker` builds tracks from successive localizations using the an MTT (multi-target tracking) algorithm whose implementation was adapted from Quot, itself an adaptation of Sergé et. al. "Dynamic multiple-target tracing to probe spatiotemporal cartography of cell membranes" Nature Methods 5, 687-694 (2008). Linking options are weighted according to their likelihood, estimated *via* an underlying model of Brownian diffusion with diffusivity coefficient *D*. A linear assignment problem is then solved in order to find the optimal matching. There are several parameters:

  `max_diffusivity` : estimation of the maximum diffusion coefficient, which defines the maximum distance between two successive localizations (search radius) : sqrt{4 D Delta t}

  `max_blinks` : the maximum number of frame during which a particle is allowed to disappear

  `d_bound_naive` : the naive estimate of the diffusion coefficient

  `init_cost` : the cost of starting a new trajectory

  `y_diff` : the relative importance of the trajectory's past (vs. the naive guess) in the estimation of its diffusion coefficient.

- `DiffusionTracker` same as above but with a weighting of options simply based on the Euclidean distance. Its parameters are:

`max_diffusivity` : estimation of the maximum diffusion coefficient, which defines the maximum distance between two successive localizations (search radius) : sqrt{4 D Delta t}

`max_blinks` : the maximum number of frame during which a particle is allowed to disappear

`init_cost` : the cost of starting a new trajectory

Configure your pipeline

We recommend using the `from_dict()` class method to instanciate your pipelines, specifying the desired classes and parameters in a Python dictionnary. Steps must be grouped by categories using the `movie_preprocessors`, `localizer`, `locs_processors` and `tracker` keys. If no localizer or tracker is found, the default classes with default parameters are used. If a class has no parameters, simply use an empty dictionnary as a value : `{"MyStepWithoutArgs":{}}`.

```
tp = TifPipeline.from_dict({
    "name":"default_with_percentile_filtering",
    "movie_preprocessors":[{"WindowPercentileFilter":{"percentile":10,"window_size":300}}
↪]
})

tp = TifPipeline.from_dict({
    "name":"stricter_than_default",
    "localizer":{"Detector":{"t":1.5}},
)
```

Export your pipeline's configuration

Pipelines can be exported and loaded from YAML files, so that they can easily be shared and re-used.

```
tp.to_yaml("myproject/mypipeline.yaml") # Export
tp = TifPipeline.from_yaml("myproject/mypipeline.yaml") # Load
```

The YAML file for the `tp2` pipeline is

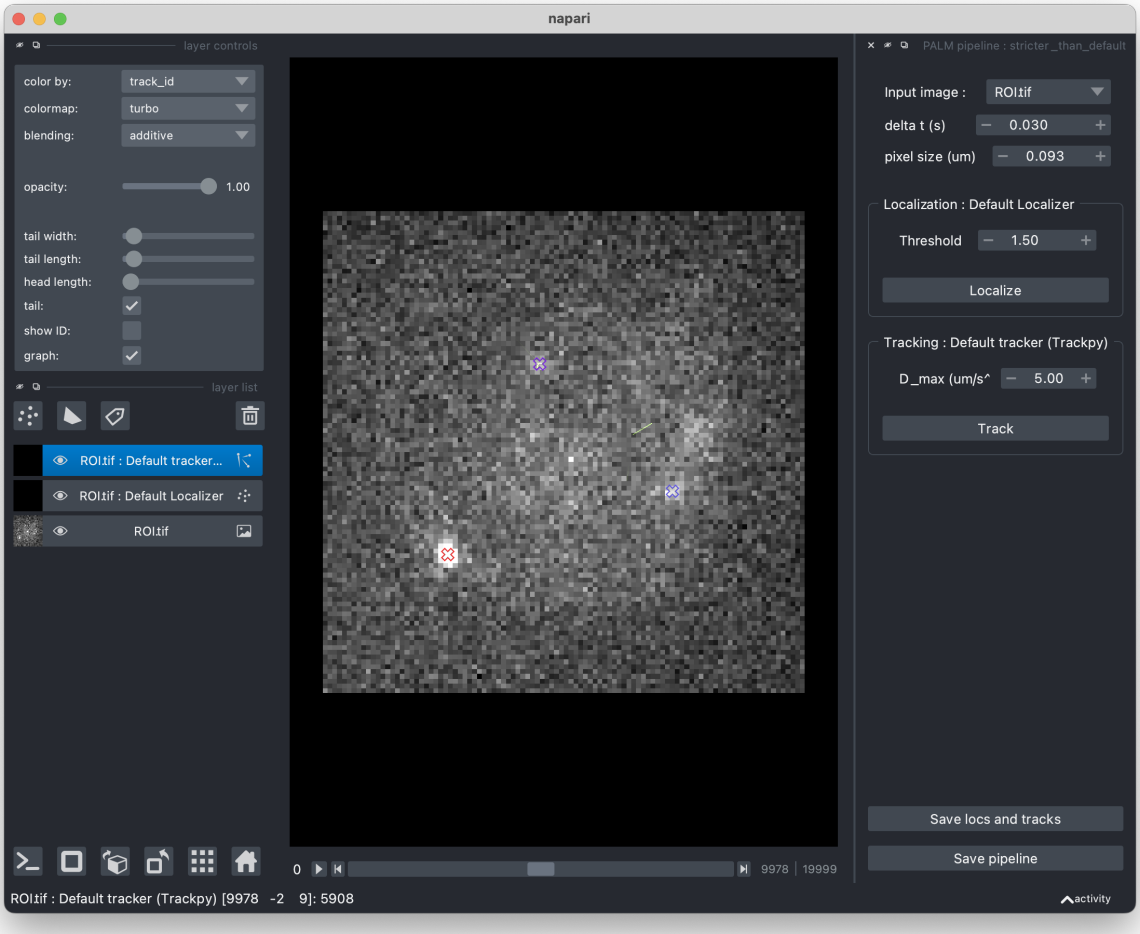
Listing 1: myproject/mypipeline.yaml

```yaml
name: stricter_than_default
localizer:
    Detector:
        t: 1.5
tracker:
    ConservativeTracker:
        max_diffusivity: 5.0
```

3.2.4 Tune your pipeline with the Napari viewer

If you would like to adjust your pipeline's parameters on one of your movies, you can use the `TifPipelineWidget.view_pipeline()` function. This will open a Napari viewer allowing you to see the effect of each step's parameters on the processing of your movie. When you're satisfied, save the pipeline to a file by clicking the "Export pipeline" button ! You'll then be able to load it in a script or notebook using `TifPipeline.from_yaml()`.

```
TifPipelineWidget.view_pipeline(acq=acq)
# or
TifPipelineWidget.view_pipeline(tif_file="ROI.tif")
```



3.2.5 Make your own processing steps !

Do you want to remove some artifact proper to your optical setup ? To use the new state-of-the-art localizer instead of the rudimentary one provided by PALM-tools (inspired from ThunderSTORM's one) ?

Good news : the `TifPipeline` class is actually quite customizable and open to add-ons ! If you want to use your own steps, subclass the corresponding abstract base class : for a localizer, `Localizer`, for a movie pre-processor, `MoviePreprocessor`, etc...

One method must be overriden in your subclass, whose name depends on the type of step (see the code for details).

Important: Stick to the argument and output types provided in the abstract base classes for things to run smoothly. Note that movie pre-processors' `preprocess()` functions expect Dask arrays while detectors' `detect_slice()` expect numpy arrays : in this last case, Dask arrays are sliced by blocks of successive frames by the pipeline.

As an example, here is the code of the `ConservativeTracker` class, based on Trackpy. The source code of `BaseDetector` and other built-in steps might guide you when implementing your own processing steps.

```
class ConservativeTracker(Tracker):

    def __init__(self, max_diffusivity: float = 5.0):
        # Attributes will automatically be detected as parameters of the step and stored/
↪loaded.
        # Parameters must have default values
        self.max_diffusivity = max_diffusivity

    def track(self, locs: pd.DataFrame):
        # This is where the actual tracking happen.
        import trackpy as tp

        delta_t = self.estimate_delta_t(locs)  # This is a Tracker's method.
        dim = 2
        max_radius = np.sqrt(2 * dim * self.max_diffusivity * delta_t)
        logging.info("Max radius is %.2f" % max_radius)
        tracks = tp.link(locs, search_range=max_radius, link_strategy="drop")
        locs["n"] = tracks["particle"]
        return locs

    @property
    def name(self):
        # This is for printing
        return "Default tracker (Trackpy)"

    # The following dicts are used when setting the parameters through a graphic␣
↪interface, using open_in_napari()
    widget_types = {
        "max_diffusivity": "FloatSpinBox",
        "delta_t": "FloatSpinBox",
    }
    # For details about widget types, see https://napari.org/magicgui/
    widget_options = {
        "delta_t": {
            "step": 0.01,
            "tooltip": "time interval between frames (in seconds)",
            "min": 0.0,
            "label": "Time delta (s)",
        },
        "max_diffusivity": {
            "step": 1.0,
            "tooltip": "Assumed maximum diffusivity (in microns per square second).\
↪nThis is used in conjunction with the Time delta to set the maximal distance between␣
↪consecutive localizations",
            "label": "D_max (um^2/s)",
            "min": 0.0,
        },
    }
```

3.3 Data classes

3.3.1 `Experiment`

An `Experiment` is **set of movies** which are meant to be analyzed together. This class scans all `.tif` files in `data_folder` and stores localizations and tracks generated by *pipelines* in the `export_folder`.

```
exp = Experiment(data_folder="myproject/data/raw",export_folder="myproject/data/processed
↪")
```

Important: Pixel size and camera exposure are assumed constant across all movies of an experiment. The first time you create an Experiment with a given `export_folder`, you will be prompted to enter the corresponding values. They will then be stored in the export folder and you will not have to enter them again.

Process batches of movies

Once you have set up your TifPipeline, you might want to run it on all your movies. To do so, simply run the following command. This will create in your experiment's directory a folder named after yout pipeline's name. The processing outputs will be written in this folder and loaded when needed. The pipeline's parameters will be saved in your Experiment's `export_folder`.

```
tp.process(exp)
```

The index table

An experiment stores movie-specific information in an index table (`index_df`) in order to be accessible for later analysis : which type of cell did you observe, which protein did you look at, how were the cells treated, etc… Below is an example of what such a file might look like.

file	cell_type	observation_time
ROI1.tif	neuron	T+1h
ROI2.tif	astrocyte	T+1h
ROI3.tif	neuron	T+2h

Populate columns from file names

This information can be entered manually, by editing the corresponding `.csv` file or by overriding the `custom_fields` property of a subclass of `Experiment` to automatically populate the columns from the file names. It is a dictionnary whose keys are the desired column names and whose values indicate how to populate the rows.

- If the value is an integer `i`, the column's rows will take the value of the `i`-th part of the file path (note that it can be negative, if you'd like to start from the end).
- If it is a string, the rows will take the value `True` if the file path contains this string, and `False` otherwise.
- Finally, if it is a callable taking one argument, then rows' values will be `callable(file_path)`.

```
from palmari import Experiment

def myfunc(s:str):
    return s.split("_")[4].upper()

class MyExperiment(Experiment):

    @property
    def custom_fields(self) -> dict:

        return {
            "field_1":"sometext", # True if the file's name contains "sometext"
            "field_2":3, # returns the third part of the file name (delimited by the␣
↪filesystem's separator)
            "field_3":myfunc # Returns the output of myfunc
        }
```

Tip: Access the file names of an experiment's movies by treating the experiment as a list.

```
for file_path in exp:
    # do stuff
# or
my_file = exp[0]
```

3.3.2 Acquisition

Each **movie** in an `Experiment` is an instance of the `Acquisition` class. It must be instanciated with a reference to a `TifPipeline` responsible for generating or loading its localizations.

```
acq = Acquisition(tif_file=exp[0], experiment=exp, tif_pipeline=tp)
# To process a single acquisition with TifPipeline tp, use the following function
tp.process(acq)
```

If the movie has been processed by the given pipeline, then you can access its localizations table using `.locs`. Here is what this table looks like after an acquisition has been processed by a standard pipeline. The `n` column identifies the trajectory to which the localization belongs. Time is expressed in seconds and coordinates are in micrometers.

	frame	x	y	ratio	sigma	total_intensity	t	n
0	0	4.634299	2.957245	1.140336	4.373642	3572.0	0.00	0
1	0	5.133801	5.043820	1.003860	4.088229	3611.0	0.00	1
2	2	4.424841	5.378470	1.337645	4.451456	3790.0	0.06	2
3	2	5.673679	2.878053	1.114035	4.624364	4374.0	0.06	3
4	3	4.039363	4.773757	1.458809	3.972434	5811.0	0.09	2

After the movie has been localized and tracked, you can visualize the localizations superimposed to the camera recording in Napari using the `.view()` function.

```
acq.view() # View the .tif movie with superimposed localizations and tracks, using␣
↪Napari.
```

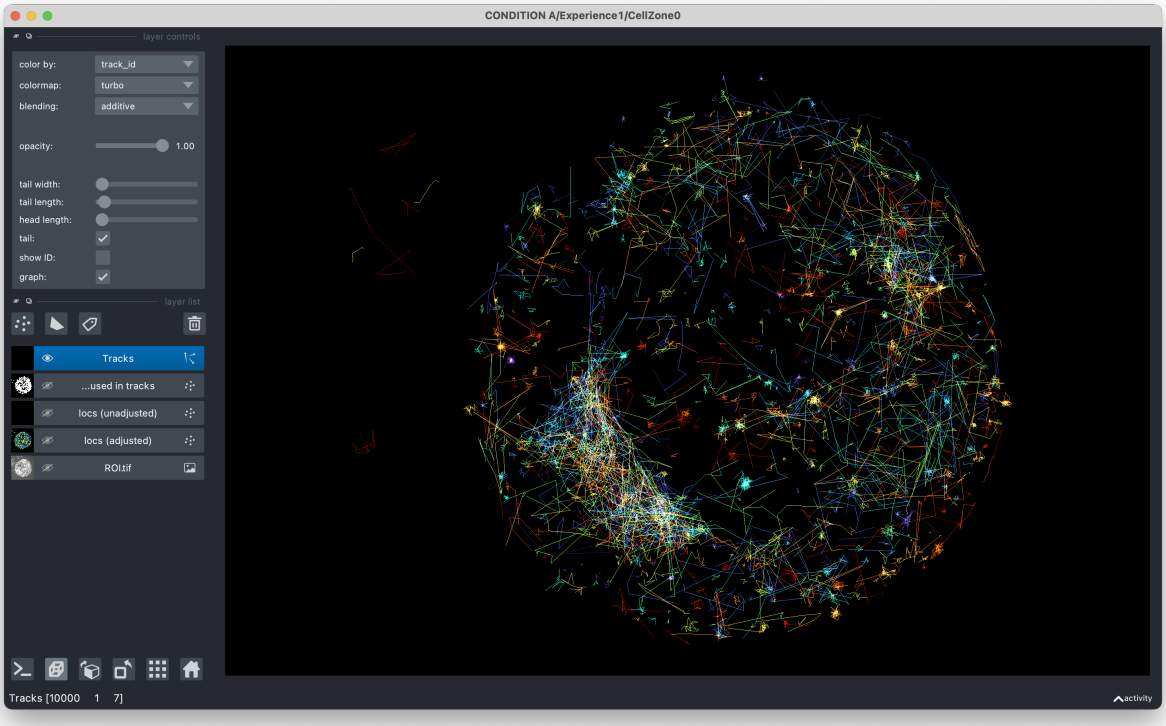


Fig. 5: There you are !

3.4 Examples

- **Your first Palmari pipeline** :

3.5 Authors

3.5.1 Main developer

Hippolyte Verdier <hverdier@pasteur.fr>

3.5.2 Team

Decision and Bayesian computation lab, Institut Pasteur, Paris

3.6 palmari

3.6.1 palmari package

Subpackages

palmari.data_structure package

Submodules

palmari.data_structure.acquisition module

class palmari.data_structure.acquisition.**Acquisition**(*tif_file*, *experiment:* Experiment, *tif_pipeline:* TifPipeline)

Bases: `object`

An acquisition corresponds to a PALM movie. It is part of an `Experiment`, and bound to a `TifPipeline` with which it is processed.

property ID: str

add_columns_to_loc(*columns: pandas.core.frame.DataFrame*, *save: bool = False*)

add_traj_cols_to_locs(*traj_columns: pandas.core.frame.DataFrame*)

traj_columns is a dataframe whose index corresponds to the 'n' column (traj ID) We merge it with the locs dataframe

basic_stats()

compute_intensity()

compute_tubeness()

correct_drift()

property drift_is_corrected: bool

get_property(*col: str*) → Any

Access an acquisition's property, read from its experiment's index table.

Parameters **col** (*str*) – name of the index table column to look into

Returns value of the corresponding row x column in the experiment's index table

Return type Any

get_traj(*n: int*)

property image: dask.array.core.Array

The actual movie, loaded with Dask.

Returns the movie.

Return type da.Array

property intensity: pandas.core.frame.DataFrame

property intensity_is_computed: bool

property intensity_path: str

property is_localized: bool

property is_processed: bool

property is_tracked

localize()

property locs: pandas.core.frame.DataFrame

property locs_path: str

property polygon_files: List

property polygon_folder: str

property raw_locs: pandas.core.frame.DataFrame

property raw_locs_path: str

track()

trajectories_list(*min_length: int = 7*, *return_indices: bool = True*, *filter: Optional[Callable] = None*)

Returns a list of trajectories whose length is above a given threshold, possibly filtered according to the locs they're based on

Parameters

- **min_length** (*int, optional*) – Defaults to 7.
- **return_indices** (*bool, optional*) – Whether to return indices ('n' column of the locs DataFrame) along with coordinates. Defaults to True.
- **filter** (*Callable, optional*) – Callable, which takes as input the locs DataFrame and returns a boolean Series with the same index. Defaults to None.

Returns either a list of trajectories, or a tuple containing this same list and the list of indices

Return type _type_

property tubeness: numpy.array

property tubeness_is_computed: bool

property tubeness_path: str

view(*min_traj_length=1*, *polygon_ID_col: Optional[str] = None*, *short_for_tests: bool = False*, *contrast_limits: tuple = (100, 500)*)

view_points(*viewer=None*, *polygon_ID_col: Optional[str] = None*, *subsample: Optional[int] = None*)

view_polygons(*viewer=None*)

view_tracks(*viewer=None*, *min_length=1*, *polygon_ID_col: Optional[str] = None*)

palmari.data_structure.experiment module

class palmari.data_structure.experiment.**Experiment**(*data_folder: str*, *export_folder: str*, *DT: Optional[float] = None*, *pixel_size: Optional[float] = None*, *file_pattern: Optional[str] = None*)

Bases: object

add_new_roi_to_index(*f: str*)

property all_files: List[str]

Return all files indexed in self.index_df

Returns all files indexed in self.index_df

Return type List[*Acquisition*]

check_export_folder_and_load_info()

property custom_fields: dict

Override this in a subclass of Experience to meet your needs keys of the dict are column names values are used to fill the columns, using the TIF file name of each acquisition values can be :

- string : True if the file name contains that string
- int : the i-th part of the file name, when split using the filesystem separator
- callable : callable(filename)

for instance

```
{"condition":get_condition_from_name}
```

classmethod from_single_tif(*tif_file: str*, *export_folder: str*)

get_ID_of_acq(*acquisition:* palmari.data_structure.acquisition.Acquisition)

property index_df: pandas.core.frame.DataFrame

look_for_new_columns(*overwrite=False*)

Computes custom columns

Parameters **overwrite** (*bool, optional*) – Whether to overwrite pre-existing values. Defaults to False.

look_for_updates()

Look if the index dataframe matches the reality of present/absent files And computes custom columns if needed

parameter_influence_on_stats(*param_name: str*, *stats: list = ['n_locs']*, *mode: str = 'hist'*)

Plots the influence of a processing parameter on some statistics.

Parameters

- **param_name** (*str*) – The parameter whose influence is studied. it should be one returned by acquisition.basic_stats
- **stats** (*list, optional*) – Statistics to compare. Defaults to ["n_locs"].
- **mode** (*str, optional*) – Plotting mode. supported : "bars" and "hist". Defaults to "hist".

remove_old_roi_from_index(*f: str*)

property runs_stats: pandas.core.frame.DataFrame

save_index()

Just saves index_df

scan_folder() → list

Module contents

palmari.processing package

Subpackages

palmari.processing.steps package

Submodules

palmari.processing.steps.base module

class palmari.processing.steps.base.**Detector**

Bases: *palmari.processing.steps.base.ProcessingStep*

property action_name

cols_dtype = {'frame': <class 'int'>, 'x': <class 'float'>, 'y': <class 'float'>}

detect(*img: numpy.array*, *frame_start: int = 0*) → pandas.core.frame.DataFrame

abstract detect_frame(*img: numpy.array*) → pandas.core.frame.DataFrame

Detect spots from a temporal slice of an image. Override in subclasses

Parameters **img** (*np.array*) – 3D array [T, X, Y]

Returns localizations table with the following columns : x, y, frame

Return type pd.DataFrame

property is_detector

property is_localizer

movie_detection(*mov: dask.array.core.Array*)

property name

process(**args*)

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

class palmari.processing.steps.base.**LocProcessor**

Bases: *palmari.processing.steps.base.ProcessingStep*

property name

process(**args*)

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

class palmari.processing.steps.base.**MoviePreProcessor**

Bases: *palmari.processing.steps.base.ProcessingStep*

property name

abstract preprocess(*mov: dask.array.core.Array*) → pandas.core.frame.DataFrame

process(**args*)

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

class palmari.processing.steps.base.**ProcessingStep**

Bases: `abc.ABC`

property is_detector

property is_localizer

abstract property name

abstract process(**args*)

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

to_dict() → dict

update_param(*param*, **args*)

widget_options = {}

widget_types = {}

class palmari.processing.steps.base.**SubpixelLocalizer**

Bases: *palmari.processing.steps.base.ProcessingStep*

property action_name

cols_dtype = {'frame': <class 'int'>, 'x': <class 'float'>, 'y': <class 'float'>}

property is_localizer

localize(*img: numpy.array*, *detections: pandas.core.frame.DataFrame*, *frame_start: int = 0*) → pandas.core.frame.DataFrame

abstract localize_frame(*img: numpy.array*, *detections: numpy.array*) → pandas.core.frame.DataFrame

Extract localizations from a slice of a .tif file. Override in subclasses

Parameters

- **img** (*np.array*) – 2D array [X, Y]
- **detections** (*np.array*) – 2D array [X, Y] center of spots (pixel indices)

Returns localizations table with the following columns : x, y (in pixel units)

Return type pd.DataFrame

movie_localization(*mov: dask.array.core.Array*, *detections: pandas.core.frame.DataFrame*)

property name

process(**args*)

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

class palmari.processing.steps.base.**Tracker**

Bases: *palmari.processing.steps.base.ProcessingStep*

property action_name

estimate_delta_t(*locs*)

property name

process(**args*)

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

abstract track(*locs: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

palmari.processing.steps.drift_corrector module

class palmari.processing.steps.drift_corrector.**BeadDriftCorrector**(*sigma: int = 3*)

Bases: *palmari.processing.steps.base.LocProcessor*

property action_name

property name

process(*mov: dask.array.core.Array*, *locs: pandas.core.frame.DataFrame*, *pixel_size: float*) → pandas.core.frame.DataFrame

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

widget_options = {'sigma': {'label': 'Bead radius (in pixels)', 'max': 50, 'min': 2}}

widget_types = {'sigma': 'FloatSpinBox'}

class palmari.processing.steps.drift_corrector.**CorrelationDriftCorrector**(*min_n_locs_per_bin: int = 10000*, *max_n_bins: int = 20*)

Bases: *palmari.processing.steps.base.LocProcessor*

property action_name

property name

process(*mov: dask.array.core.Array*, *locs: pandas.core.frame.DataFrame*, *pixel_size: float*) → pandas.core.frame.DataFrame

Depending on their types, subclasses might have different effector functions. e.g. Trackers' effector function is named `.track()` This function is just meant to be a link towards the effector function

widget_options = {'max_n_bins': {'label': 'Bins', 'max': 50, 'min': 2, 'tooltip': 'Maximum number of bins. One shift per bin will be computed.'}, 'min_n_locs_per_bin': {'label': 'Locs / bin', 'max': 10000000, 'min': 100, 'step': 1000, 'tooltip': 'Minimum number of localizations per bin'}}

widget_type = {'max_n_bins': 'Spinner', 'min_n_locs_per_bin': 'Spinner'}

palmari.processing.steps.quot_localizer module

class palmari.processing.steps.quot_localizer.**BaseDetector**(*k: float = 1.5*, *w: int = 21*, *t: float = 1.0*, *method: str = 'llr'*)

Bases: *palmari.processing.steps.base.Detector*

detect_frame(*img: numpy.array*) → pandas.core.frame.DataFrame

Detect spots from a temporal slice of an image. Override in subclasses

Parameters **img** (*np.array*) – 3D array [T, X, Y]

Returns localizations table with the following columns : x, y, frame

Return type pd.DataFrame

property name

widget_options = {'k': {'label': 'Spot size (px)', 'min': 0.0, 'step': 0.1}, 'method': {'choices': [('Log-likelihood ratio', 'llr'), ('Laplacian of Gaussians', 'log')], 'label': 'Method'}, 't': {'label': 'Threshold', 'max': 3.0, 'min': 0.5, 'readout': True, 'step': 0.1}, 'w': {'label': 'Window size (px)', 'min': 5, 'step': 2, 'tooltip': 'Size of the square window used for testing.'}}

widget_types = {'method': 'ComboBox', 't': 'FloatSlider'}

class palmari.processing.steps.quot_localizer.**MaxLikelihoodLocalizer**(*method: str = 'ls_int_gaussian'*, *window_size: int = 9*, *sigma: float = 1.5*)

Bases: *palmari.processing.steps.base.SubpixelLocalizer*

cols_dtype = {'H_det': <class 'float'>, 'I0': <class 'float'>, 'I0_err': <class 'float'>, 'bg': <class 'float'>, 'bg_err': <class 'float'>, 'error_flag': <class 'int'>, 'frame': <class 'int'>, 'rmse': <class 'float'>, 'snr': <class 'float'>, 'x': <class 'float'>, 'x_err': <class 'float'>, 'y': <class 'float'>, 'y_err': <class 'float'>}

localize_frame(*img: numpy.array*, *detections: numpy.array*) → pandas.core.frame.DataFrame

Extract localizations from a slice of a .tif file. Override in subclasses

Parameters

- **img** (*np.array*) – 2D array [X, Y]
- **detections** (*np.array*) – 2D array [X, Y] center of spots (pixel indices)

Returns localizations table with the following columns : x, y (in pixel units)

Return type pd.DataFrame

property name

widget_options = {'method': {'choices': [('Gaussian', 'ls_int_gaussian'), ('Poisson', 'poisson_int_gaussian')], 'label': 'Noise', 'tooltip': 'Assumed type of background noise', 'value': 'ls_int_gaussian'}, 'sigma': {'label': 'Spot size (px)', 'step': 0.1}, 'window_size': {'label': 'Window size (px)', 'step': 1}}

widget_types = {'method': 'ComboBox'}

class palmari.processing.steps.quot_localizer.**RadialLocalizer**(*window_size: int = 9*)

Bases: *palmari.processing.steps.base.SubpixelLocalizer*

cols_dtype = {'I0': <class 'float'>, 'bg': <class 'float'>, 'error_flag': <class 'bool'>, 'snr': <class 'float'>, 'x': <class 'float'>, 'y': <class 'float'>}

localize_frame(*img: numpy.array*, *detections: numpy.array*) → pandas.core.frame.DataFrame

Extract localizations from a slice of a .tif file. Override in subclasses

Parameters

- **img** (*np.array*) – 2D array [X, Y]
- **detections** (*np.array*) – 2D array [X, Y] center of spots (pixel indices)

Returns localizations table with the following columns : x, y (in pixel units)

Return type pd.DataFrame

property name

widget_options = {'window_size': {'label': 'Window size (px)', 'step': 1}}

palmari.processing.steps.quot_tracker module

class palmari.processing.steps.quot_tracker.**DiffusionTracker**(*max_diffusivity: float = 5.0*, *max_blinks: int = 0*, *d_bound_naive: float = 0.1*, *init_cost: float = 50.0*, *y_diff: float = 0.9*)

Bases: *palmari.processing.steps.base.Tracker*

property name

track(*locs: pandas.core.frame.DataFrame*)

widget_options = {'d_bound_naive': {'label': 'D_0 (um^2/s)', 'min': 0.0, 'step': 0.1, 'tooltip': "First guess of a trajectory's diffusivity"}, 'init_cost': {'label': 'Initialization cost', 'min': 1, 'step': 10, 'tooltip': 'Cost of initializing a new trajectory when a reconnection is available in the search radius'}, 'max_blinks': {'label': 'Max blinks', 'max': 2, 'min': 0, 'step': 1, 'tooltip': "Maximum number of tolerated blinks (i.e. number of frames during which a particle can 'disappear')."}, 'max_diffusivity': {'label': 'D_max (um^2/s)', 'min': 0.0, 'step': 1.0, 'tooltip': 'Assumed maximum diffusivity (in microns per square second).\nThis is used in conjunction with the Time delta to set the maximal distance between consecutive localizations'}, 'y_diff': {'label': 'Past vs naive', 'max': 1.0, 'min': 0.0, 'step': 0.01, 'tooltip': "Relative weight of a trajectory's past in the estimation of its diffusivity"}}

widget_types = {'d_bound_naive': 'FloatSpinBox', 'init_cost': 'FloatSpinBox', 'max_blinks': 'SpinBox', 'max_diffusivity': 'FloatSpinBox', 'y_diff': 'FloatSlider'}

class palmari.processing.steps.quot_tracker.**EuclideanTracker**(*max_diffusivity: float = 5.0*, *max_blinks: int = 0*, *scale: float = 1.0*, *init_cost: float = 50.0*)

Bases: *palmari.processing.steps.base.Tracker*

property name

track(*locs: pandas.core.frame.DataFrame*)

widget_options = {'init_cost': {'label': 'Initialization cost', 'min': 1, 'step': 10, 'tooltip': 'Cost of initializing a new trajectory when a reconnection is available in the search radius'}, 'max_blinks': {'label': 'Max blinks', 'max': 2, 'min': 0, 'step': 1, 'tooltip': "Maximum number of tolerated blinks (i.e. number of frames during which a particle can 'disappear')."}, 'max_diffusivity': {'label': 'D_max (um^2/s)', 'min': 0.0, 'step': 1.0, 'tooltip': 'Assumed maximum diffusivity (in microns per square second).\nThis is used in conjunction with the Time delta to set the maximal distance between consecutive localizations'}, 'scale': {'label': 'Weight scale', 'min': 0.1, 'step': 0.5, 'tooltip': 'Scaling factor between distance (in um) and coefficient of the corresponding assignment in the weight matrix'}}

widget_types = {'init_cost': 'FloatSpinBox', 'max_blinks': 'SpinBox', 'max_diffusivity': 'FloatSpinBox', 'scale': 'FloatSpinBox'}

palmari.processing.steps.trackpy_tracker module

class palmari.processing.steps.trackpy_tracker.**ConservativeTracker**(*max_diffusivity: float = 5.0*)

Bases: *palmari.processing.steps.base.Tracker*

property name

track(*locs: pandas.core.frame.DataFrame*)

widget_options = {'delta_t': {'label': 'Time delta (s)', 'min': 0.0, 'step': 0.01, 'tooltip': 'time interval between frames (in seconds)'}, 'max_diffusivity': {'label': 'D_max (um^2/s)', 'min': 0.0, 'step': 1.0, 'tooltip': 'Assumed maximum diffusivity (in microns per square second).\nThis is used in conjunction with the Time delta to set the maximal distance between consecutive localizations'}}

widget_types = {'delta_t': 'FloatSpinBox', 'max_diffusivity': 'FloatSpinBox'}

palmari.processing.steps.window_percentile module

class palmari.processing.steps.window_percentile.**WindowPercentileFilter**(*percentile: float = 3.0*, *window_size: int = 100*)

Bases: *palmari.processing.steps.base.MoviePreProcessor*

property action_name

property name

preprocess(*mov: dask.array.core.Array*) → dask.array.core.Array

widget_options = {'percentile': {'label': 'Percentile', 'min': 0.0, 'step': 1.0, 'tooltip': 'percentile of the pixel intensity values in the window which will be considered as the ground level.'}, 'window_size': {'label': 'Window', 'min': 100, 'step': 50, 'tooltip': 'Size of the windows along which quantiles are computed'}}

widget_types = {'percentile': 'FloatSpinBox', 'window_size': 'SpinBox'}

Module contents

Submodules

palmari.processing.edit_pipeline_window module

palmari.processing.tif_pipeline module

class palmari.processing.tif_pipeline.**TifPipeline**(*name: str*, *movie_preprocessors: List[*palmari.processing.steps.base.MoviePreProcessor*]*, *detector:* palmari.processing.steps.base.Detector, *localizer:* palmari.processing.steps.base.SubpixelLocalizer, *loc_processors: List[*palmari.processing.steps.base.LocProcessor*]*, *tracker:* palmari.processing.steps.base.Tracker)

Bases: `object`

property available_steps

can_be_removed(*step*)

contains_class(*step*)

classmethod default_with_name(*name: str*)

exp_params_path(*acq:* palmari.data_structure.acquisition.Acquisition) → str

exp_run_df_path(*acq:* palmari.data_structure.acquisition.Acquisition) → str

classmethod from_dict(*p: Dict*)

Instantiate from a dictionnary. Here's an example Dictionnary which could be passed as an argument :

```
{
    "name":"my_pipeline",
    "movie_preprocessors":[
        {
            "MyPreProcessingClass":{"param1":value,"param2":other_value}
        },
        {
            "WindowPercentileFilter":{}
            # If the parameter's dict is empty, default parameters will be used
        }
    }],
    "localizer":{
        "DefaultLocalizer":{"threshold_factor":1.5}
        },
    "tracker":{
        "UnknownClass":{"bla":bla}
        # If the class is not found, this will raise an exception
        # Similarly, if the class provided does not inherit Tracker, an␣
↪exception will be raised
    }
    # If some step is not mentioned (e.g. here, there's nothing about␣
↪localization processing), then
    # if it's movie_preprocessors, then no movie_preprocessors will be used␣
↪(same for loc_processors)
    # it it's localizer or tracker, then the default classes will be used.
}
```

classmethod from_yaml(*file*)

has_alternatives_to(*step*)

index_of(*step*)

is_already_localized(*acq:* palmari.data_structure.acquisition.Acquisition)

is_already_tracked(*acq:* palmari.data_structure.acquisition.Acquisition)

is_mandatory(*step*)

loc_processing(*mov: dask.array.core.Array*, *locs: pandas.core.frame.DataFrame*, *pixel_size: float = 1.0*) → pandas.core.frame.DataFrame

mark_as_localized(*acq:* palmari.data_structure.acquisition.Acquisition)

mark_as_tracked(*acq:* palmari.data_structure.acquisition.Acquisition)

movie_localization(*mov: dask.array.core.Array*, *DT: float*, *pixel_size: float*) → pandas.core.frame.DataFrame

movie_preprocessing(*mov: dask.array.core.Array*) → dask.array.core.Array

process(*to_process: Union[*palmari.data_structure.acquisition.Acquisition, palmari.data_structure.experiment.Experiment*]*, *force_reprocess: bool = False*)

step_class_of(*step*)

step_type_of(*step*)

to_dict() → dict

to_yaml(*fileName*)

tracking(*locs: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

palmari.processing.tif_pipeline_widget module

palmari.processing.utils module

palmari.processing.utils.**get_values_as_in_dict**(*dict_to_copy: dict*, *summary_string: str*) → dict

Module contents

palmari.quot package

Submodules

palmari.quot.chunkFilter module

quot.filters – background subtraction and related filters to clean up dirty data for detection and localization

class palmari.quot.chunkFilter.**ChunkFilter**(*path*, *start=None*, *stop=None*, *method='identity'*, *chunk_size=40*, *method_static=True*, *init_load=True*, ***method_kwargs*)

Bases: *palmari.quot.read.ImageReader*

A chunkwise file reader that also performs common filtering operations, like subtracting background from each frame.

It is convenient to integrate file reading and filtering, since filtering often involves computing some kind of function on a chunk of frames. For instance, we can perform a rudimentary kind of background subtraction by subtracting the minimum value of each pixel in a 100-frame block from the corresponding pixel in each individual frame.

When iterating over this object, the ChunkFilter does the following:

1. Load a new chunk, cached at self.chunk.
2. **Perform some precomputations on the chunk - for instance,** find the through-time minimum of each pixel in frame.
3. **Return individual filtered frames using the calculated** precomputations.
4. Load new chunks as necessary.

init path : str, path to an image file start : int, the start frame when iterating stop : int, the stop frame when iterating method : str, the filtering method to use chunk_size : int, the block size to use for filtering method_static : bool, the method is unlikely to change

during the existence of this ChunkFilter. This is not true, for instance, when using the ChunkFilter in a GUI setting where the user can rapidly change between different filtering settings. If False, the ChunkFilter caches information about the chunk much more aggressively. This makes switching between filtering methods much faster at the cost of memory.

init_load [bool, load the first chunk during init.] Generally keep this True unless you have a very good reason not to.

**method_kwargs : to the filtering method

attributes self.cache self.block_starts (1D ndarray, int) self.block_start (int), the first frame for the current block

filter_frame(*frame_index*)

Return a single filtered frame from the movie.

args frame_index : int

returns 2D ndarray (YX)

get_chunk_start(*frame_index*)

Return the starting frame for the corresponding chunk.

load_chunk(*start*, *recache=True*)

Load a new image chunk starting at the frame *start*. Saves the chunk at self.chunk.

args start : int recache : bool, recompute cached factors for the new chunk

set_chunk_size(*chunk_size*)

Change the chunk size for this ChunkFilter.

Parameters **chunk_size** (*int*) –

set_method_kwargs(*method=None*, *recache_filtered_image=False*, ***kwargs*)

Set the kwargs passed to the underlying filtering method. This includes factors that are precomputed for each chunk.

If *method* is None, then use the current method.

Sets the self.method_kwargs variable.

Parameters

- **method** (*str, the name of the method to use*) –
- **recache** (*bool, force the ChunkFilter to recache*) – the Gaussian filtered image, if it exists
- **kwargs** (*to the method*) –

set_subregion(***kwargs*)

Set a subregion for this ChunkFilter. All filtering methods will return only this subregion.

args y0, y1, x0, x1 (int), the limits of the subregion

palmari.quot.chunkFilter.**identity**(*img*, ***kwargs*)

Do not filter the image.

Parameters **img** (*2D ndarray (YX)*) –

Return type 2D ndarray (YX)

palmari.quot.chunkFilter.**simple_sub**(*img*, *sub_img*, *scale=1.0*)

Ffrom each pixel in *img*, subtract the corresponding pixel in *sub_img* multiplied by *scale*. Set all negative values to zero.

Parameters

- **img** (*2D ndarray (YX)*) –
- **sub_img** (*2D ndarray (YX)*) –
- **scale** (*float*) –

Return type 2D ndarray (YX)

palmari.quot.chunkFilter.**sub_gauss_filt_mean**(*img*, *gauss_filt_mean_img*, *k=5.0*, *scale=1.0*)

Subtract a Gaussian-filtered mean image from this frame.

Parameters

- **img** (*2D ndarray (YX)*) –
- **mean_img** (*2D ndarray (YX)*) –
- **k** (*float, kernel sigma*) –
- **scale** (*float*) –

Return type 2D ndarray

palmari.quot.chunkFilter.**sub_gauss_filt_median**(*img*, *gauss_filt_median_img*, *k=5.0*, *scale=1.0*)

Subtract a Gaussian-filtered median image from this frame.

Parameters

- **img** (*2D ndarray (YX)*) –
- **median_img** (*2D ndarray (YX)*) –
- **k** (*float, kernel sigma*) –
- **scale** (*float*) –

Return type 2D ndarray

palmari.quot.chunkFilter.**sub_gauss_filt_min**(*img*, *gauss_filt_min_img*, *k=5.0*, *scale=1.0*)

Subtract a Gaussian-filtered minimum image from this frame.

Parameters

- **img** (*2D ndarray (YX)*) –
- **min_img** (*2D ndarray (YX)*) –
- **k** (*float, kernel sigma*) –
- **scale** (*float*) –

Return type 2D ndarray

palmari.quot.chunkFilter.**sub_mean**(*img*, *mean_img*, *scale=1.0*)

Wrapper for simple_sub() that uses the pixelwise mean.

Parameters

- **img** (*2D ndarray (YX)*) –

- **mean_img** (*2D ndarray (YX)*) –
- **scale** (*float*) –

Return type 2D ndarray (YX)

palmari.quot.chunkFilter.**sub_median**(*img*, *median_img*, *scale=1.0*)

Wrapper for simple_sub() that uses the pixelwise median.

Parameters

- **img** (*2D ndarray (YX)*) –
- **median_img** (*2D ndarray (YX)*) –
- **scale** (*float*) –

Return type 2D ndarray (YX)

palmari.quot.chunkFilter.**sub_min**(*img*, *min_img*, *scale=1.0*)

Wrapper for simple_sub() that uses the pixelwise minimum.

Parameters

- **img** (*2D ndarray (YX)*) –
- **min_img** (*2D ndarray (YX)*) –
- **scale** (*float*) –

Return type 2D ndarray (YX)

palmari.quot.core module

core.py – high-level user functions for running filtering, detection, subpixel localization, and tracking sequentially on the same datasets

palmari.quot.core.**localize_file**(*path*, *out_csv=None*, *progress_bar=True*, ***kwargs*)

Run filtering, detection, and subpixel localization on a single image movie. This does NOT perform tracking.

Parameters

- **path** (*str, path to the image file*) –
- **out_csv** (*str, path to save file, if*) – desired
- **progress_bar** (*bool, show a progress bar*) –
- **kwargs** (*configuration*) –

Return type pandas.DataFrame, the localizations

palmari.quot.core.**retrack_file**(*path*, *out_csv=None*, ***kwargs*)

Given an existing set of localizations or trajectories, (re)run tracking to reconstruct trajectories.

Parameters

- **path** (str, path to a *trajs.csv file) –
- **out_csv** (*str, path to save the resulting trajectories, if*) – desired
- **kwargs** (*tracking configuration*) –

Return type pandas.DataFrame, the reconnected localizations

palmari.quot.core.**retrack_files**(*paths*, *out_suffix=None*, *num_workers=1*, ***kwargs*)

Given a set of localizations, run retracking on each file and save to a CSV.

If *out_suffix* is not specified, then the trajectories are saved to the original set of localization files (overwriting them).

Parameters

- **paths** (*list of str, a set of CSV files encoding trajectories*) –
- **out_suffix** (*str, the suffix to use when generating the output*) – paths. If *None*, then the output trajectories are saved to the original file path.
- **num_workers** (*int, the number of threads to use*) –
- **kwargs** (*tracking configuration*) –

palmari.quot.core.**track_directory**(*path*, *ext='.nd2'*, *num_workers=4*, *save=True*, *contains=None*, *out_dir=None*, ***kwargs*)

Find all image files in a directory and run localization and tracking.

Parameters

- **path** (*str, path to directory*) –
- **ext** (*str, image file extension*) –
- **num_workers** (*int, the number of threads*) – to use
- **save** (*bool, save the results to CSV*) – files
- **contains** (*str, a substring that all image files*) – are required to contain
- **out_dir** (*str, directory for output CSV files*) –
- **kwargs** (*configuration*) –

Returns with extension "_trajs.csv" in the same directory.

Return type None. Output of trackng is saved to files

palmari.quot.core.**track_file**(*path*, *out_csv=None*, *progress_bar=True*, ***kwargs*)

Run filtering, detection, subpixel localization, and tracking on a single target movie.

Parameters

- **path** (*str, path to the image file*) –
- **out_csv** (*str, path to save file, if*) – desired
- **progress_bar** (*bool, show a progress bar*) –
- **kwargs** (*tracking configuration*) –

Return type pandas.DataFrame, the reconnected localizations

palmari.quot.core.**track_files**(*paths*, *num_workers=4*, *save=True*, *out_dir=None*, ***kwargs*)

Run tracking on several files using parallelization.

Parameters

- **paths** (*list of str, paths to image files to track*) –
- **num_workers** (*int, the number of threads to use*) –
- **save** (*bool, save the output to CSVs files. The names*) – for these CSVs are generated from the names of the corresponding image files.

- **out_dir** (*str, output directory*) –
- **kwargs** (*tracking configuration, as read with*) – quot.io.read_config

Returns file

Return type list of pandas.DataFrame, the tracking results for each

palmari.quot.findSpots module

findSpots.py – detect spots in 2D images

palmari.quot.findSpots.**centered_gauss**(*I*, *k=1.0*, *w=9*, *t=200.0*, *return_filt=False*)

Convolve the image with a mean-subtracted Gaussian kernel, then apply a static threshold.

Parameters

- **I** (*2D ndarray (YX)*) –
- **k** (*float, kernel sigma*) –
- **w** (*int, kernel window size*) –
- **t** (*float, threshold*) –
- **return_filt** (*bool, also return the filtered image*) – and boolean image

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**detect**(*I*, *method=None*, ***kwargs*)

Run spot detection on a single image according to a detection method.

Parameters

- **I** (*2D ndarray (YX), image frame*) –
- **method** (*str, a method name in METHODS*) –
- **kwargs** (*special argument to the method*) –

Returns X coordinates of identified spots

Return type 2D ndarray of shape (n_spots, 2), the Y and

palmari.quot.findSpots.**dog**(*I*, *k0=1.0*, *k1=3.0*, *w=9*, *t=200.0*, *return_filt=False*)

Convolve the image with a difference-of-Gaussians kernel, then apply a static threshold.

Parameters

- **I** (*2D ndarray*) –
- **k0** (*float, positive kernel sigma*) –
- **k1** (*float, negative kernel sigma*) –

- **w** (*int, kernel size*) –
- **t** (*float, threshold*) –
- **return_filt** (*bool, also return the filtered image*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**dou**(*I*, *w0=3*, *w1=11*, *t=200.0*, *return_filt=False*)

Uniform-filter an image with two different kernel sizes and subtract them. This is like a poor-man's version of DoG filtering.

Parameters

- **I** (*2D ndarray*) –
- **w0** (*int, positive kernel size*) –
- **w1** (*int, negative kernel size*) –
- **t** (*float, threshold*) –
- **return_filt** (*bool, also return the filtered image*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**gauss**(*I*, *k=1.0*, *w=9*, *t=200.0*, *return_filt=False*)

Convolve the image with a simple Gaussian kernel, then apply a static threshold

Parameters

- **I** (*2D ndarray (YX)*) –
- **k** (*float, kernel sigma*) –
- **w** (*int, kernel window size*) –
- **t** (*float, threshold*) –
- **return_filt** (*bool, also return the filtered image*) – and boolean image

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**gauss_filt_min_max**(*I*, *k=1.0*, *w=9*, *t=200.0*, *mode='constant'*, *return_filt=False*, ***kwargs*)

Similar to min_max_filter(), but perform a Gaussian convolution prior to looking for spots by min/max filtering.

Parameters

- **I** (*2D ndarray*) –
- **k** (*float, Gaussian kernel sigma*) –
- **w** (*int, window size for test*) –
- **t** (*float, threshold for spot detection*) –
- **mode** (*str, behavior at boundaries (see scipy.ndimage*) – documentation)
- **kwargs** (*to ndimage.maximum_filter/minimum_filter*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**hess_det**(*I*, *k=1.0*, *t=200.0*, *return_filt=False*)

Use the local Hessian determinant of the image as the criterion for detection. The Hessian determinant is related to the "spot-ness" of an image and is generally a better criterion for detection than the Laplacian alone (as in LoG filtering).

Parameters

- **frame** (*2D ndarray*) –
- **k** (*float, Gaussian filtering kernel size*) –
- **t** (*float, threshold for detection*) –
- **return_filt** (*bool*) –

Returns

- *If *return_filt** –

  (2D ndarray, filtered image; 2D ndarray, binary image; pandas.DataFrame, the detections

)

- *else* – pandas.DataFrame, the detections

palmari.quot.findSpots.**hess_det_broad_var**(*I*, *k=1.0*, *t=200.0*, *w0=15*, *w1=9*, *return_filt=False*)

Use the local Hessian determinant of the image as the criterion for detection. This method uses a broader definition of the Laplacian kernel than hess_det() or hess_det_var().

This method also normalizes the Hessian determinant against its local variance to give it the property of intensity invariance. Otherwise, the broader Laplacian kernel tends to produce quite different threshold values for different cameras.

Parameters

- **frame** (*2D ndarray*) –
- **k** (*float, Gaussian filtering kernel size*) –
- **t** (*float, threshold for detection*) –
- **w0** (*int, width of the box around each point*) – in which to calculate the variance
- **w1** (*int, width of the hollow subregion inside*) – each box to exclude from variance calculations
- **return_filt** (*bool*) –

Returns

- *If *return_filt** –

  (2D ndarray, filtered image; 2D ndarray, binary image; pandas.DataFrame, the detections

)

- *else* – pandas.DataFrame, the detections

palmari.quot.findSpots.**hess_det_var**(*I*, *k=1.0*, *t=200.0*, *w0=15*, *w1=9*, *return_filt=False*)

Similar to hess_det, this uses the local Hessian determinant of the image as the criterion for spot detection. However, it normalizes the Hessian by its local variance in a hollow ring around each point (see quot.helper.hollow_box_var). This endows it with a kind of invariance with respect to the absolute intensity of the image, resulting in more consistent threshold arguments.

Parameters

- **frame** (*2D ndarray*) –
- **k** (*float, Gaussian filtering kernel size*) –
- **t** (*float, threshold for detection*) –
- **w0** (*int, width of the box around each point*) – in which to calculate the variance
- **w1** (*int, width of the hollow subregion inside*) – each box to exclude from variance calculations
- **return_filt** (*bool*) –

Returns

- *If *return_filt** –

  (2D ndarray, filtered image; 2D ndarray, binary image; pandas.DataFrame, the detections

)

- *else* – pandas.DataFrame, the detections

palmari.quot.findSpots.**llr**(*I*, *k=1.0*, *w=9*, *t=20.0*, *return_filt=False*)

Perform a log-likelihood ratio test for the presence of spots. This is the ratio of likelihood of a Gaussian spot in the center of the subwindow, relative to the likelihood of flat background with Gaussian noise.

Parameters

- **I** (*2D ndarray*) –
- **k** (*float, Gaussian kernel sigma*) –
- **w** (*int, window size for test*) –
- **t** (*float, threshold for spot detection*) –
- **return_filt** (*bool, also return filtered image*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**llr_rect**(*I*, *w0=3*, *w1=11*, *t=20.0*, *return_filt=False*)

Perform a log-likelihood ratio test for the presence of square spots of size *w0* in an image using a test equivalent to llr(). While squares only roughly approximate real spots, the test can be performed extremely fast due to the fact that only uniform filtering is required.

Parameters

- **I** (*2D ndarray*) –
- **w0** (*int, spot kernel size*) –
- **w1** (*int, background kernel size*) –
- **t** (*float, threshold for spot detection*) –
- **return_filt** (*bool, also return filtered image*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**log**(*I*, *k=1.0*, *w=11*, *t=200.0*, *return_filt=False*)

Detect spots by Laplacian-of-Gaussian filtering.

Parameters

- **I** (*2D ndarray*) –
- **k** (*float, kernel sigma*) –
- **w** (*int, kernel size*) –
- **t** (*float, threshold*) –
- **return_filt** (*bool, also return the filtered image*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**min_max**(*I*, *w=9*, *t=200.0*, *mode='constant'*, *return_filt=False*, ***kwargs*)

Use the difference between the local maximum and local minimum in square subwindows to identify spots in an image.

Parameters

- **I** (*2D ndarray*) –
- **w** (*int, window size for test*) –
- **t** (*float, threshold for spot detection*) –
- **mode** (*str, behavior at boundaries (see scipy.ndimage*) – documentation)
- **kwargs** (*to ndimage.maximum_filter/minimum_filter*) –

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.findSpots.**mle_amp**(*I*, *k=1.0*, *w=9*, *t=200.0*, *return_filt=False*)

Convolve the image with a mean-subtracted Gaussian kernel, square the result and normalize by the then apply a static threshold. This is equivalent to

Parameters

- **I** (*2D ndarray (YX)*) –
- **k** (*float, kernel sigma*) –
- **w** (*int, kernel window size*) –
- **t** (*float, threshold*) –

- **return_filt** (*bool, also return the filtered image*) – and boolean image

Returns

- *if return_filt*
- *(* – 2D ndarray, the post-convolution image; 2D ndarray, the thresholded binary image; 2D ndarray, shape (n_spots, 2), the y and x

  coordinates of each spot
- *)*
- *else* –

  2D ndarray, shape (n_spots, 2), the y and x coordinates of each spot

palmari.quot.helper module

quot.helper.py – low-level utilities

palmari.quot.helper.**I0_is_crazy**(*I0*, *max_I0=10000*)

Determine whether the intensity estimate in subpixel localization looks crazy. For many methods, this is often the first thing to diverge.

Parameters

- **I0** (*float, the intensity estimate for*) – subpixel fitting
- **max_I0** (*float, the maximum tolerated intensity*) –

Return type bool, True if crazy

palmari.quot.helper.**amp_from_I**(*I0*, *sigma=1.0*)

Given a 2D Gaussian PSF, return the PSF peak amplitude given the intensity *I0*. *I0* is equal to the PSF integrated above background, while *amp* is equal to the PSF evaluated at its maximum.

Parameters

- **I0** (*float, intensity estimate*) –
- **sigma** (*float, width of Gaussian*) –

Return type float, amplitude estimate

palmari.quot.helper.**assign_methods**(*methods*)

Decorator that designates a wrapper function for other methods. Adds a new kwarg to the decorated function ("method") that looks up the corresponding method in the *methods* dict.

Example

def add(x, y): return x + y

def mult(x, y): return x * y

METHODS = {'add': add, 'mult': mult}

do_operation = assign_methods(METHODS)(lambda a, b: None)

This would be called: do_operation(2, 3, method='add') # is 5 do_operation(2, 3, method='mult') # is 6

Parameters **methods** (*dict that keys strings to methods*) –

Return type function, wrapper for the methods

palmari.quot.helper.**check_2d_gauss_fit**(*img_shape*, *pars*, *max_I0=10000*)

Check whether the fit parameters for a 2D symmetric Gaussian with static sigma are sane.

This includes:

- **Are the y and x coordinates inside the PSF** window?
- Are there negative intensities?
- Are there crazily high intensities?

Parameters

- **img_shape** (*(int, int), the PSF image shape*) –
- **pars** (*1D ndarray, (y, x, I0, bg), the*) – fit parameters
- **max_I0** (*float, maximum tolerated intensity*) –

Return type bool, True if the fit passes the checks

palmari.quot.helper.**concat_tracks**(**tracks*)

Join some trajectory dataframes together into a larger dataframe, while preserving uniqe trajectory indices.

Parameters **tracks** (*pandas.DataFrame with the "trajectory" column*) –

Return type pandas.DataFrame, the concatenated trajectories

palmari.quot.helper.**connected_components**(*semigraph*)

Find independent connected subgraphs in a bipartite graph (aka semigraph) by a floodfill procedure.

The semigraph is a set of edges betwen two sets of nodes- one set corresponding to each y-index, and the other corresponding to each x-index. Edges can only exist between a y-node and an x-node, so the semigraph is conveniently represented as a binary (0/1 values) 2D matrix.

The goal of this function is to identify independent subgraphs in the original semigraph. If I start on a y-node or x-node for one of these independent subgraphs and walk along edges, I can reach any point in that subgraph but no points in other subgraphs.

Each subgraph can itself be represented by a 2D ndarray along with a reference to the specific y-nodes and x-dnoes in the original semigraph corresponding to that subgraph.

For the purposes of tracking, it is also convenient to separate y-nodes and x-nodes that are NOT connected by any edges to other nodes. These are returned as separate ndarrays: y_without_x and x_without_y.

Parameters **semigraph** (*2D ndarray, with only 0 and 1 values*) –

Returns

- *(* – subgraphs : list of 2D ndarray subgraph_y_indices : list of 1D ndarray, the set of

  y-nodes corresponding to each subgraph

  subgraph_x_indices [list of 1D ndarray, the set of] x-nodes corresponding to each subgraph

  y_without_x [1D ndarray, the y-nodes that are] not connected to any x-nodes

  x_without_y [1D ndarray, the x-nodes that are] not connected to any y-nodes

- *)*

palmari.quot.helper.**estimate_I0**(*I*, *y*, *x*, *bg*, *sigma=1.0*)

Estimate the integrated intensity of a 2D integrated Gaussian PSF. This model has four parameters - y, x, I0, and bg - so with y, x, and bg constrained, we can solve for I0 directly.

In this method, we use the brightest pixel in the image to solve for I0 given the PSF model. This is easy but often an overestimate in the presence of Poisson noise.

Parameters

- **I** (*2D ndarray, the PSF image*) –
- **y** (*float, center of PSF in y and x*) –
- **x** (*float, center of PSF in y and x*) –
- **bg** (*float, estimated background intensity*) – per pixel (AU)
- **sigma** (*float, 2D integrated Gaussian PSF*) – radius

Return type float, estimate for I0

palmari.quot.helper.**estimate_I0_multiple_points**(*I*, *y*, *x*, *bg*, *sigma=1.0*)

Estimate the integrated intensity of a 2D integrated Gaussian PSF. This model has four parameters - y, x, I0, and bg - so with y, x, and bg constrained, we can solve for I0 directly.

In this method, we use the intensities at the 3x3 central points in the image window to obtain nine estimates for I0, then average them for the final estimate. This is more accurate than estimate_I0() but is slower. Probably overkill for a first guess.

Parameters

- **I** (*2D ndarray, the PSF image*) –
- **y** (*float, center of PSF in y and x*) –
- **x** (*float, center of PSF in y and x*) –
- **bg** (*float, estimated background intensity*) – per pixel (AU)
- **sigma** (*float, 2D integrated Gaussian PSF*) – radius

Return type float, estimate for I0

palmari.quot.helper.**estimate_snr**(*I*, *I0*)

Estimate the signal-to-noise ratio of a PSF, given an estimate *I0* for its intensity.

Parameters

- **I** (*2D ndarray, the PSF image*) –
- **I0** (*float, intensity estimate*) –

Return type float, SNR estimate

palmari.quot.helper.**fit_ls_int_gaussian**(*img*, *guess*, *sigma=1.0*, *ridge=0.0001*, *max_iter=20*, *damp=0.3*, *convergence=0.0001*, *divergence=1.0*)

Given an observed spot, estimate the maximum likelihood parameters for a 2D integrated Gaussian PSF sampled with normally-distributed noise. Here, we use a Levenberg- Marquardt procedure to find the ML parameters.

The model parameters are (y, x, I0, bg), where y, x is the spot center, I0 is the integrated intensity, and bg is the average background per pixel.

The function also returns several parameters that are useful for quality control on the fits.

Parameters

- **img** (*2D ndarray, the observed PSF*) –
- **guess** (*1D ndarray of shape (4), the initial*) – guess for y, x, I0, and bg
- **sigma** (*float, Gaussian PSF width*) –
- **ridge** (*float, initial regularization term*) – for inversion of the Hessian
- **max_iter** (*int, the maximum number of iterations*) – tolerated
- **damp** (*float, damping factor for update vector.*) – Larger means faster convergence but also more unstable.
- **convergence** (*float, the criterion for fit*) – convergence. Only y and x are tested for convergence.
- **divergence** (*float, the criterion for fit*) – divergence. Fitting is terminated when this is reached. CURRENTLY NOT IMPLEMENTED.

Returns

- *(* – 1D ndarray, the final parameter estimate;

  1D ndarray, the estimated error in each parameter (square root of the inverse Hessian diagonal);

  float, the Hessian determinant;

  float, the root mean squared deviation of the final PSF model from the observed PSF;

  int, the number of iterations
- *)*

palmari.quot.helper.**fit_ls_point_gaussian**(*img*, *guess*, *sigma=1.0*, *ridge=0.0001*, *max_iter=20*, *damp=0.3*, *convergence=0.0001*, *divergence=1.0*)

Given an observed spot, estimate the maximum likelihood parameters for a 2D pointwise-evaluated Gaussian PSF sampled with normally-distributed noise. This function uses a Levenberg-Marquardt procedure to find the ML parameters.

The underlying model for the pointwise-evaluated and integrated Gaussian PSFs is identical - a symmetric 2D Gaussian. The distinction is how they handle sampling on discrete pixels:

- **the point Gaussian takes the value on each pixel** to be equal to the Gaussian evaluated at the center of the pixel
- **the integrated Gaussian takes the value on each** pixel to be equal to the Gaussian integrated across the whole area of that pixel

Integrated Gaussians are more accurate and less prone to edge biases, but the math is slightly more complicated and fitting is slower as a result.

The model parameters are (y, x, I0, bg), where y, x is the spot center, I0 is the integrated intensity, and bg is the average background per pixel.

The function also returns several parameters that are useful for quality control on the fits.

Parameters

- **img** (*2D ndarray, the observed PSF*) –
- **guess** (*1D ndarray of shape (4), the initial*) – guess for y, x, I0, and bg
- **sigma** (*float, Gaussian PSF width*) –
- **ridge** (*float, initial regularization term*) – for inversion of the Hessian

- **max_iter** (*int, the maximum number of iterations*) – tolerated
- **damp** (*float, damping factor for update vector.*) – Larger means faster convergence but also more unstable.
- **convergence** (*float, the criterion for fit*) – convergence. Only y and x are tested for convergence.
- **divergence** (*float, the criterion for fit*) – divergence. Fitting is terminated when this is reached. CURRENTLY NOT IMPLEMENTED.

Returns

- *(* – 1D ndarray, the final parameter estimate;

  1D ndarray, the estimated error in each parameter (square root of the inverse Hessian diagonal);

  float, the Hessian determinant;

  float, the root mean squared deviation of the final PSF model from the observed PSF;

  int, the number of iterations

- *)*

palmari.quot.helper.**fit_poisson_int_gaussian**(*img*, *guess*, *sigma=1.0*, *ridge=0.0001*, *max_iter=20*, *damp=0.3*, *convergence=0.0001*, *divergence=1.0*)

Given an observed spot, estimate the maximum likelihood parameters for a 2D integrated Gaussian PSF model sampled with Poisson noise, using a Levenberg-Marquardt procedure.

While LM with Poisson noise is a little slower than LS routines, it is the most accurate model for the noise on EMCCD cameras.

The model parameters are (y, x, I0, bg), where y, x is the spot center, I0 is the integrated intensity, and bg is the average background per pixel.

The function also returns several parameters that are useful for quality control on the fits.

Parameters

- **img** (*2D ndarray, the observed PSF*) –
- **guess** (*1D ndarray of shape (4), the initial*) – guess for y, x, I0, and bg
- **sigma** (*float, Gaussian PSF width*) –
- **ridge** (*float, initial regularization term*) – for inversion of the Hessian
- **max_iter** (*int, the maximum number of iterations*) – tolerated
- **damp** (*float, damping factor for update vector.*) – Larger means faster convergence but also more unstable.
- **convergence** (*float, the criterion for fit*) – convergence. Only y and x are tested for convergence.
- **divergence** (*float, the criterion for fit*) – divergence. Fitting is terminated when this is reached. CURRENTLY NOT IMPLEMENTED.

Returns

- *(* – 1D ndarray, the final parameter estimate;

  1D ndarray, the estimated error in each parameter (square root of the inverse Hessian diagonal);

float, the Hessian determinant;

float, the root mean squared deviation of the final PSF model from the observed PSF;

int, the number of iterations

- *)*

palmari.quot.helper.**get_edges**(*bin_img*)

Given a binary image that is False outside of an object and True inside of it, return another binary image that is True for points in the original image that border a False pixel, and False otherwise.

Parameters **bin_img** (*2D ndarray, dtype bool*) –

Return type 2D ndarray, dtype bool, shape shape as bin_img

palmari.quot.helper.**get_ordered_mask_points**(*mask*, *max_points=100*)

Given the edges of a two-dimensional binary mask, construct a line around the mask.

Parameters

- **mask** (*2D ndarray, dtype bool, mask edges as*) – returned by get_edges
- **max_points** (*int, the maximum number of points tolerated*) – in the final mask. If the number of points exceeds this, the points are repeatedly downsampled until there are fewer than max_points.

Returns to this ROI

Return type 2D ndarray of shape (n_points, 2), the points belonging

palmari.quot.helper.**hollow_box_var**(*I*, *w0=15*, *w1=9*)

Calculate the local variance of every point in a 2D image, returning another 2D image composed of the variances.

Variances are calculated in a "hollow box" region around each point. The box is a square kernel of width *w0*; the hollow central region is width *w1*.

For instance, if w0 = 5 and w1 = 3, the kernel would be

1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1

Parameters

- **I** (*2D ndarray, image*) –
- **w0** (*int, width of the kernel*) –
- **w1** (*int, width of central region*) –

Return type 2D ndarray

palmari.quot.helper.**indices**(*size_y*, *size_x*)

Convenience wrapper for np.indices.

Parameters

- **size_y** (*int, the number of indices*) – in the y direction
- **size_x** (*int, the number of indices*) – in the x direction

Returns

- *(* – 2D ndarray, the Y indices of each pixel; 2D ndarray, the X indices of each pixel
- *)*

palmari.quot.helper.**indices_1d**(*size_y*, *size_x*)

Cached convenience function to generate two sets of 1D indices.

Parameters

- **size_y** (*int, the number of indices*) – in the y direction
- **int** (*size_x ;*) – in the x direction
- **indices** (*the number of*) – in the x direction

Returns

- *(* – 1D ndarray, the Y indices of each pixel; 1D ndarray, the X indices of each pixel
- *)*

palmari.quot.helper.**int_gauss_psf_1d**(*Y*, *yc*, *sigma=1.0*)

Return a 2D integrated Gaussian PSF with unit intensity, projected onto one of the axes.

Parameters

- **Y** (*1D ndarray, coordinates along the axis*) –
- **yc** (*float, PSF center*) –
- **sigma** (*float, Gaussian sigma*) –

Returns each row of pixels

Return type 1D ndarray, the intensities projected along

palmari.quot.helper.**int_gauss_psf_2d**(*size_y*, *size_x*, *yc*, *xc*, *I0*, *sigma=1.0*)

Return a 2D integrated Gaussian PSF with intensity I0. Does not include a background term.

Parameters

- **size_y** (*int, the number of pixels in the y*) – direction
- **size_x** (*int, the number of pixels in the x*) – direction
- **yc** (*float, center of PSF in y*) –
- **xc** (*float, center of PSF in x*) –
- **sigma** (*float, PSF width*) –

Returns PSF

Return type 2D ndarray of shape (size_y, size_x), the

palmari.quot.helper.**int_gauss_psf_deriv_1d**(*Y*, *yc*, *sigma=1.0*)

Evaluate the derivative of an integrated Gaussian PSF model with unit intensity projected onto 1 axis with respect to its axis variable.

Parameters

- **Y** (*1D ndarray, the pixel indices at*) – which to evaluate the PSF
- **yc** (*float, the spot center*) –
- **sigma** (*float, Gaussian sigma*) –

Returns with respect to the axis variable at each pixel

Return type 1D ndarray, the derivative of the projection

palmari.quot.helper.**invert_hessian**(*H*, *ridge=0.0001*)

Invert a Hessian with ridge regularization to stabilize the inversion.

Parameters

- **H** (*2D ndarray, shape (n, n)*) –
- **ridge** (*float, regularization term*) –

Returns inverted Hessian

Return type 2D ndarray, shape (n, n), the

palmari.quot.helper.**label_spots**(*binary_img*, *intensity_img=None*, *mode='max'*)

Find continuous nonzero objects in a binary image, returning the coordinates of the spots.

If the objects are larger than a single pixel, then to find the central pixel do

1. use the center of mass (if mode == 'centroid')
2. use the brightest pixel (if mode == 'max')
3. **use the mean position of the binary spot** (if img_int is not specified)

Parameters

- **binary_img** (*2D ndarray (YX), dtype bool*) –
- **intensity_img** (*2D ndarray (YX)*) –
- **mode** (*str, 'max' or 'centroid'*) –

Returns the Y and X coordinate of each spot

Return type 2D ndarray (n_spots, 2), dtype int64,

palmari.quot.helper.**load_tracks_dir**(*dirname*, *suffix='trajs.csv'*, *start_frame=0*, *min_track_len=1*)

Load all of the trajectory files in a target directory into a single pandas.DataFrame.

Parameters

- **dirname** (*str, directory containing track CSVs*) –
- **suffix** (*str, extension for the track CSVs*) –
- **start_frame** (*int, exclude all trajectories before*) – this frame
- **min_track_len** (*int, the minimum trajectory length to*) – include

Return type pandas.DataFrame

palmari.quot.helper.**pad**(*I*, *H*, *W*, *mode='ceil'*)

Pad an array with zeroes around the edges, placing the original array in the center.

Parameters

- **I** (*2D ndarray, image to be padded*) –
- **H** (*int, height of final image*) –
- **W** (*int, width of final image*) –
- **mode** (*str, either 'ceil' or 'floor'. 'ceil'*) – yields convolution kernels that function similarly to scipy.ndimage.uniform_filter.

Return type 2D ndarray, shape (H, W)

palmari.quot.helper.**point_gauss_psf_1d**(*Y*, *yc*, *sigma=1.0*)

Evaluate a 1D point Gaussian with unit intensity at a set of points.

Parameters

- **Y** (*1D ndarray, coordinates along the axis*) –
- **yc** (*float, PSF center*) –
- **sigma** (*float, Gaussian sigma*) –

Returns each row of pixels

Return type 1D ndarray, the intensities projected along

palmari.quot.helper.**point_gauss_psf_2d**(*size_y*, *size_x*, *yc*, *xc*, *I0*, *sigma=1.0*)

Return a 2D pointwise-evaluated Gaussian PSF with intensity I0.

Parameters

- **size_y** (*int, size of the PSF subwindow*) – in y
- **size_x** (*int, size of the PSF subwindow*) – in x
- **yc** (*float, PSF center in y*) –
- **xc** (*float, PSF center in x*) –
- **I0** (*float, PSF intensity*) –
- **sigma** (*float, Gaussian sigma*) –

Return type 2D ndarray, the PSF model

palmari.quot.helper.**psf_int_proj_denom**(*sigma*)

Convenience function to produce the denominator for the Gaussian functions in int_gauss_psf_1d().

Parameters **sigma** (*float*) –

Return type float, np.sqrt(2 * (sigma**2))

palmari.quot.helper.**psf_point_proj_denom**(*sigma*)

Convenience function to produce the denominator for the Gaussian functions in psf_point_psf_1d().

Parameters **sigma** (*float, Gaussian sigma*) –

Return type float, float

palmari.quot.helper.**ring_mean**(*I*)

Take the mean of the outer ring of pixels in a 2D array.

Parameters **I** (*2D ndarray (YX)*) –

Return type float, mean

palmari.quot.helper.**ring_var**(*I*, *ddof=0*)

Take the variance of the outer ring of pixels in a 2D ndarray.

Parameters

- **I** (*2D ndarray, image*) –
- **ddof** (*int, delta degrees of freedom*) –

Return type float, variance estimate

palmari.quot.helper.**rs**(*psf_image*)

Localize the center of a PSF using the radial symmetry method.

Originally conceived by the criminally underrated Parasarathy R Nature Methods 9, pgs 724–726 (2012).

Parameters **psf_image** (*2D ndarray, PSF subwindow*) –

Return type float y estimate, float x estimate

palmari.quot.helper.**stable_divide_array**(*N*, *D*, *zero=0.001*)

Divide array N by array D, setting zeros in D to *zero*. Assumes nonzero D.

Parameters

- **N** (*ndarrays of the same shape*) –
- **D** (*ndarrays of the same shape*) –
- **zero** (*float*) –

Return type ndarray of the same shape as N and D

palmari.quot.helper.**stable_divide_float**(*N*, *D*, *inf=0.0*)

Divide float N by float D, returning *inf* if *D* is zero.

Parameters

- **N** (*float*) –
- **D** (*float*) –
- **inf** (*float*) –

Return type float

palmari.quot.helper.**threshold_image**(*I*, *t=200.0*, *return_filt=False*, *mode='max'*)

Take all spots in an image above a threshold *t* and return their approximate centers.

If *return_filt* is set, the function also returns the raw image and binary image. This is useful as a back door when writing GUIs.

Parameters

- **I** (*2D ndarray, image*) –
- **t** (*float, threshold*) –
- **return_filt** (*bool*) –
- **mode** (*str, either 'max' or 'centroid'*) –

Returns

- *If *return_filt* is set**
- *(* – 2D ndarray, same as I; 2D ndarray, binary image; 2D ndarray, shape (n_spots, 2), the spot coordinates
- *)*
- *else* –

  2D ndarray of shape (n_spots, 2), the spot coordinates

palmari.quot.helper.**time_f**(*f*)

Decorator. When executing the decorated function, also print the time to execute.

Parameters **f** (*function*) –

Return type output of f

palmari.quot.helper.**track_length**(*tracks*)

Generate a new column with the trajectory length in frames.

Parameters **tracks** (*pandas.DataFrame*) –

Returns column "track_length"

Return type pandas.DataFrame, the input dataframe with a new

palmari.quot.helper.**tracked_mat_to_csv**(*path*, *out_csv=None*, *pixel_size_um=0.16*)

Convert a file from *Tracked.mat format, a MATLAB-based format for trajectories to a DataFrame format.

Parameters

- **path** (str, path to a *Tracked.mat file) –
- **out_csv** (*str, file to save the result to*) –
- **pixel_size_um** (*float, the size of pixels in um*) –

Returns file as a DataFrame

Return type pandas.DataFrame, the contents of the *Tracked.mat

palmari.quot.mask module

mask.py – apply binary masks to an SPT movie

class palmari.quot.mask.**MaskInterpolator**(*mask_edges*, *mask_frames*, *n_vertices=101*, *interp_kind='linear'*, *plot=True*)

Bases: object

Given a set of 2D masks, interpolate the masks between frames to generate an approximation to the mask for intermediate frames.

In more detail:

We have a set of 2D masks defined in frames F0, F1, …, Fn. Masks are defined as a ringlike, connected set of points. We wish to use the masks defined in frames F(i) and F(i+1) to estimate the shape of the mask in any intermediate frame, assuming the mask varies smoothly/linearly between frame F(i) and F(i+1).

Instantiation of the LinearMaskInterpolator() accomplishes this, using either linear or spline interpolation. The resulting object can then be passed any frame index between F0 and Fn and will generate the corresponding 2D mask.

The resulting object can be passed a frame index, and will generate the corresponding 2D mask.

mask_edges [list of 2D ndarray of shape (n_points, 2),] the Y and X coordinates for each mask at each frame

mask_frames [list of int, the frame indices corresponding] to each mask edge

n_vertices [int, the number of vertices to use per] interpolated mask

interp [str, "linear" or "cubic", the type of] interpolation to use. Note that at least 4 masks are required for cubic spline interpolation.

plot [bool, show a plot of the vertex matching between] interpolated masks during initialization, for QC

__call__ : determine whether each of a set of points lies

inside or outside the mask

interpolate : generate an interpolated mask edge for an arbitrary

frame lying between the minimum and maximum frames for this object

interpolate(*frame*)

Interpolate the mask edges for a given frame index.

Parameters **frame** (*int, the frame index*) –

Returns coordinates for the points along the edge of the mask

Return type 2D ndarray of shape (self.n_vertices, 2), the YX

palmari.quot.mask.**circshift**(*points*, *shift*)

Circularly shift a set of points.

Parameters

- **points** (*ndarray of shape (n_points, D), the*) – D-dimensional coordinates of each point
- **shift** (*int, the index of the new starting point*) –

Returns but circularly shifted

Return type 2D ndarray of shape (n_points, D), the same points

Example

points_before = np.array([[1, 2], [3, 4], [5, 6]

])

points_after = circshift(points_before, 1)

points_after -> np.array([[3, 4], [5, 6], [1, 2]

])

palmari.quot.mask.**match_vertices**(*vertices_0*, *vertices_1*, *method='closest'*, *plot=False*)

Given two polygons with the same number of vertices, match each vertex in the first polygon with the "closest" vertex in the second polygon.

"closest" is in quotation marks here because, before matching, we align the two polygons by their mean position, so that the same match is returned regardless of whole-polygon shifts.

Parameters

- **vertices_0** (*2D ndarray, shape (n_points, 2), the*) – YX coordinates for the vertices of the first polygon
- **vertices_1** (*2D ndarray, shape (n_points, 2), the*) – YX coordinates for the vertices of the second polygon
- **method** (*str, the method to use to match vertices.*) – "closest": use the closest point between the two masks as the anchor point. "global": use the permutation that minimizes the total distance between the two sets of vertices.

- **plot** (*bool, show the result*) –

Returns second polygon circularly permuted to line them up with the matching vertex in the first polygon

Return type 2D ndarray, shape (n_points, 2), the vertices of the

palmari.quot.mask.**shoelace**(*points*)

Shoelace algorithm for computing the oriented area of a 2D polygon. This area is positive when the points that define the polygon are arranged counterclockwise, and negative otherwise.

Parameters **points** (*2D ndarray, shape (n_points, 2), the*) – vertices of the polygon

Returns *points*

Return type float, the oriented volume of the polygon defined by

palmari.quot.mask.**upsample_2d_path**(*points*, *kind='cubic'*, *n_vertices=101*)

Upsample a 2D path by interpolation.

Parameters

- **points** (*2D ndarray of shape (n_points, 2), the*) – Y and X coordinates of each point in the path, organized sequentially
- **kind** (*str, the kind of spline interpolation*) –
- **n_vertices** (*int, the number of points to use in the*) – upsampled path

Returns path

Return type 2D ndarray of shape (n_vertices, 2), the upsampled

palmari.quot.measureGain module

measureGain.py – measure camera gain and BG with a simple linear gain model

palmari.quot.measureGain.**measure_camera_gain**(**nd2_files*, *start_frame=100*, *plot=True*)

Measure the camera gain and offset from a set of background movies. These can be used as the "camera_gain" and "camera_bg" arguments for localization settings, allowing the user to retrieve the PSF intensities in terms of photons rather than grayvalues.

Each file in *nd2_files* should be a movie of an unlabeled, defocused, empty coverslip. Nothing should be movie, blinking, or exhibiting anything other than the ordinary camera noise. The movie should be acquired with exactly the same camera/stroboscopic settings as the SPT movies.

Ideally, each movie in *nd2_files* should use a different level of illumination, which facilitates more accurate measurement of the gain.

The method uses a pure Poisson noise model that is most applicable to EMCCD cameras.

Parameters

- **nd2_files** (*variable number of str, background movies*) –
- **start_frame** (*int, the first frame in each movie to*) – consider. This prevents photobleaching in the early frames from influencing the result.

Return type dict, floats keyed to "camera_gain" and "camera_bg"

palmari.quot.plot module

plot.py – simple plotting utilities for quot

palmari.quot.plot.**angular_dist**(*axes*, *tracks*, *min_disp=0.2*, *delta=1*, *n_bins=50*, *norm=True*, *pixel_size_um=0.16*, *bottom=0.02*, *angle_ticks=True*)

Plot the angular distribution of displacements for a set of trajectories.

Note that *axes* must be generated with projection = "polar". For instance,

fig = plt.figure() axes = fig.add_subplot(projection="polar")

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **tracks** (*pandas.DataFrame*) –
- **min_disp** (*float, the minimum displacement required*) – to consider a displacement for an angle in um
- **delta** (*int, the number of subsequent jumps over*) – which to calculate the angle. For example, if *delta* is 1, then angles are calculated between subsequent jumps in a trajectory. If *delta* is 2, then angles are calculated between jump n and jump n+2, etc.
- **n_bins** (*int, number of histogram bins*) –
- **norm** (*bool, normalize histogram*) –
- **pixel_size_um** (*float, size of pixels in um*) –
- **angle_ticks** (*bool, do tick labels for the angles*) –

Return type None, plots directly to *axes*

palmari.quot.plot.**angular_dist_files**(**csv_files*, *out_png=None*, *start_frame=0*, *filenames=False*, ***kwargs*)

For each of a set of trajectory CSVs, plot the angular distribution of displacements.

Parameters

- **csv_files** (*variadic str, paths to CSV files*) –
- **start_frame** (*int, the first frame in the file*) – consider
- **out_png** (*str, save path*) –
- **kwargs** (*to angular_dist(). These include:*) – min_disp (um), n_bins, norm, pixel_size_um

Return type None

palmari.quot.plot.**bond_angles**(*tracks*, *min_disp=0.2*, *delta=1*)

Return the set of angles between jumps originating from trajectories. Angles between pi and 2 * pi are reflected onto the interval 0, pi.

Parameters

- **tracks** (*pandas.DataFrame*) –
- **min_disp** (*float, pixels. Discard displacements less than*) – this displacement. This prevents us from being biased by localization error.

- **delta** (*int, the proximity of the two jumps with*) – respect to which the angle is calculated. For example, if *delta* is 1, then the angle between subsequent jumps is calculated. If *delta* is 2, then the angle between jump n and jump n+2 is calculated, and so on.

Returns angles in radians (from 0 to pi)

Return type 1D ndarray of shape (n_angles,), the observed

palmari.quot.plot.**coarsen_histogram**(*jump_length_histo*, *bin_edges*, *factor*)

Given a jump length histogram with many small bins, aggregate into a histogram with a small number of larger bins.

This is useful for visualization.

Parameters

- **jump_length_histo** (*2D ndarray, the jump length histograms*) – indexed by (frame interval, jump length bin)
- **bin_edges** (*1D ndarray, the edges of each jump length*) – bin in *jump_length_histo*
- **factor** (*int, the number of bins in the old histogram*) – to aggregate for each bin of the new histogram

Returns

- *(* – 2D ndarray, the aggregated histogram, 1D ndarray, the edges of each jump length bin the aggregated histogram
- *)*

palmari.quot.plot.**hex_cmap**(*cmap*, *n_colors*)

Generate a matplotlib colormap as a list of hex colors indices.

Parameters

- **cmap** (*str, name of a matplotlib cmap (for*) – instance, "viridis")
- **n_colors** (*int*) –

Return type list of str, hex color codes

palmari.quot.plot.**imlocdensities**(**csv_files*, *out_png=None*, *filenames=False*, ***kwargs*)

For each of a set of CSV files, make a KDE for localization density. This is a wrapper around imlocdensity().

If *out_png* is passed, this function saves the result to a file. Otherwise it plots to the screen.

Parameters

- **csv_files** (*variadic str, a set of CSV files*) –
- **out_png** (*str, save filename*) –
- **filenames** (*str, include the filenames as the*) – plot title
- **kwargs** (*keyword arguments to imlocdensity().*) – See that function's docstring for more info. These include: ymax, xmax, bin_size, kernel, vmax_perc, cmap, pixel_size_um, scalebar

palmari.quot.plot.**imlocdensity**(*axes*, *tracks*, *ymax=None*, *xmax=None*, *bin_size=0.1*, *kernel=3.0*, *vmax_perc=99*, *cmap='gray'*, *pixel_size_um=0.16*, *scalebar=False*)

Given a set of localizations from a single FOV, plot localization density.

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **tracks** (*pandas.DataFrame*) –
- **ymax** (*int, the height of the FOV in pixels*) –
- **xmax** (*int, the width of the FOV in pixels*) –
- **bin_size** (*float, the size of the histogram bins in*) – terms of pixels
- **kernel** (*float, the size of the Gaussian kernel*) – used for density estimation
- **vmax_perc** (*float, the percentile of the density*) – histogram used to define the upper contrast threshold
- **cmap** (*str*) –
- **pixel_size_um** (*float, size of pixels in um*) –
- **scalebar** (*bool, make a scalebar. Requires the*) – matplotlib_scalebar package.

Return type None; plots directly to *axes*

palmari.quot.plot.**imshow**(**imgs*, *vmax_mod=1.0*, *plot=True*)

Show a variable number of images side-by-side in a temporary window.

Parameters

- **imgs** (*2D ndarrays*) –
- **vmax_mod** (*float, modifier for the*) – white point as a fraction of max intensity

palmari.quot.plot.**jump_cdf_files**(**csv_files*, *out_png=None*, ***kwargs*)

For each of a set of trajectory CSVs, plot the jump length CDFs.

Parameters

- **csv_files** (*list of str, paths to CSV files*) –
- **out_png** (*str, output plot file*) –
- **kwargs** (*to plot_jump_cdfs(). These include:*) – n_frames, pixel_size_um, frame_interval, fontsize, cmap, start_frame, max_jumps_per_track, use_entire_track, linewidth

Returns on whether *out_png* is set

Return type None; either plots to screen or saves depending

palmari.quot.plot.**kill_ticks**(*axes*, *spines=True*)

Remove the y and x ticks from a plot.

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **spines** (*bool, also remove the spines*) –

Return type None

palmari.quot.plot.**locs_per_frame**(*axes*, *tracks*, *n_frames=None*, *kernel=5*, *fontsize=10*, *title=None*)

Plot the number of localizations per frame.

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **tracks** (*pandas.DataFrame*) –

- **n_frames** (*int, the number of frames in the*) – movie. If *None*, defaults to the maximum frame in *tracks*
- **kernel** (*float, size of uniform kernel used*) – for smoothing

Return type None

palmari.quot.plot.**locs_per_frame_files**(**csv_files*, *out_png=None*, ***kwargs*)

Given a set of trajectory CSVs, make a plot where each subpanel shows the number of localizations in that file as a function of time.

Parameters

- **csv_files** (*variadic str, paths to CSVs*) –
- **out_png** (*str, save file*) –
- **kwargs** (*to locs_per_frame()*) –

palmari.quot.plot.**max_int_proj**(*axes*, *nd2_path*, *vmax_perc=99*, *vmin=None*, *cmap='gray'*, *pixel_size_um=0.16*, *scalebar=False*)

Make a maximum intensity projection of a temporal image sequence in a Nikon ND2 file.

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **nd2_path** (*str, path to a target ND2 file*) –
- **vmax_perc** (*float, percentile of the intensity histogram*) – to use for the upper contrast bound
- **vmin** (*float*) –
- **cmap** (*str*) –
- **pixel_size_um** (*float, size of pixels in um*) –
- **scalebar** (*bool, make a scalebar*) –

Return type None; plots directly to *axes*

palmari.quot.plot.**plotRadialDisps**(*radial_disps*, *bin_edges*, *frame_interval=0.00548*, *plot=True*)

Plot a set of radial displacement histograms as a simple line plot.

Parameters

- **radial_disps** (*2D ndarray of shape (n_intervals, n_bins),*) – a set of radial displacement histograms for potentiall several frame intervals binned spatially
- **bin_edges** (*1D ndarray of shape (n_bins+1), the bin edge*) – definitions
- **frame_interval** (*float, seconds between frames*) –
- **plot** (*bool, immediately make a temporary plot to*) – show to the user. Otherwise return the Figure and Axes objects used to generate the image.

Returns

- *if plot* – None
- *else* –

  (matplotlib.figure.Figure, 1D array of matplotlib.axes.Axes

)

palmari.quot.plot.**plotRadialDispsBar**(*radial_disps*, *bin_edges*, *frame_interval=0.00548*, *model=None*, *plot=True*)

Plot a set of radial displacements as a bar graph. Also overlay a model as a line plot if desired.

Parameters

- **radial_disps** (*2D ndarray of shape (n_intervals,)* – n_bins), the radial displacements
- **bin_edges** (*1D ndarray of shape (n_bins+1), bin*) – edge definitions
- **frame_interval** (*float, in sec*) –
- **model** (*2D ndarray of shape (n_intervals,)* – n_bins), model at each point
- **plot** (*bool, show immediately*) –

Returns

- *if plot* – None
- *else* –

  (matplotlib.figure.Figure, array of matlotlib.axes.Axes

)

palmari.quot.plot.**plot_jump_cdfs**(*axes*, *tracks*, *n_frames=4*, *pixel_size_um=0.16*, *frame_interval=0.00748*, *fontsize=10*, *cmap='viridis'*, *start_frame=0*, *max_jumps_per_track=4*, *use_entire_track=False*, *linewidth=1*, *plot_max_r=None*, ***kwargs*)

Plot the empirical jump length probability cumulative distribution function.

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **tracks** (*pandas.DataFrame*) –
- **n_frames** (*int, the maximum number of frame*) – intervals to consider
- **pixel_size_um** (*float, size of pixels in um*) –
- **bin_size** (*float, size of the bins to use in*) – the plotted histogram
- **frame_interval** (*float, frame interval in seconds*) –
- **fontsize** (*int*) –
- **cmap** (*str*) –
- **start_frame** (*int, disregard jumps before this frame*) –
- **max_jumps_per_track** (*int, the maximum number of jumps*) – to consider from any one track
- **use_entire_track** (*bool, use all jumps from every track*) –
- **linewidth** (*int, width of the lines*) –
- **plot_max_r** (*float, the maximum jump length to show*) – in the plot
- **kwargs** (*to rad_disp_histogram_2d*) –

palmari.quot.plot.**plot_jump_pdfs**(*axes*, *tracks*, *n_frames=4*, *pixel_size_um=0.16*, *bin_size=0.02*, *norm=True*, *frame_interval=0.00748*, *fontsize=10*, *cmap='viridis'*, *start_frame=0*, *max_jumps_per_track=4*, *use_entire_track=False*, ***kwargs*)

Plot the empirical jump length probability density function.

Parameters

- **axes** (*matplotlib.axes.Axes*) –
- **tracks** (*pandas.DataFrame*) –
- **n_frames** (*int, the maximum number of frame*) – intervals to consider
- **pixel_size_um** (*float, size of pixels in um*) –
- **bin_size** (*float, size of the bins to use in*) – the plotted histogram
- **norm** (*bool, normalize to a PDF*) –
- **frame_interval** (*float, frame interval in seconds*) –
- **fontsize** (*int*) –
- **cmap** (*str*) –
- **start_frame** (*int, disregard jumps before this frame*) –
- **max_jumps_per_track** (*int, the maximum number of jumps*) – to consider from any one track
- **use_entire_track** (*bool, use all jumps from every track*) –
- **kwargs** (*to rad_disp_histogram_2d*) –

palmari.quot.plot.**plot_pixel_mean_variance**(*means*, *variances*, *origin_files=None*, *model_gain=None*, *model_bg=None*)

Plot pixel mean vs. variance for one or several movies, overlaying a linear gain model on top if desired.

Best called through quot.read.measure_camera_gain.

Parameters

- **means** (*list of 1D ndarray, the pixel*) – means for each movie to plot
- **variances** (*list of 1D ndarray, the pixel*) – variances for each movie to plot
- **origin_files** (*list of str, the labels for each*) – element in *means* and *variances*
- **model_gain** (*float, the camera gain*) –
- **model_bg** (*float, the camera BG*) –

palmari.quot.plot.**rad_disp_histogram_2d**(*tracks*, *n_frames=4*, *bin_size=0.001*, *max_jump=5.0*, *pixel_size_um=0.16*, *n_gaps=0*, *use_entire_track=False*, *max_jumps_per_track=10*)

Compile a histogram of radial displacements in the XY plane for a set of trajectories (“tracks”).

Identical with strobemodels.utils.rad_disp_histogram_2d.

Parameters

- **tracks** (*pandas.DataFrame*) –
- **n_frames** (*int, the number of frame delays to consider.*) – A separate histogram is compiled for each frame delay.

- **bin_size** (*float, the size of the bins in um. For typical*) – experiments, this should not be changed because some diffusion models (e.g. Levy flights) are contingent on the default binning parameters.
- **max_jump** (*float, the max radial displacement to consider in*) – um
- **pixel_size_um** (*float, the size of individual pixels in um*) –
- **n_gaps** (*int, the number of gaps allowed during tracking*) –
- **use_entire_track** (*bool, use every displacement in the dataset*) –
- **max_jumps_per_track** (*int, the maximum number of displacements*) – to consider per trajectory. Ignored if *use_entire_track* is *True*.

Returns

- *(* –

  2D ndarray of shape (n_frames, n_bins), the distribution of displacements at each time point;

  1D ndarray of shape (n_bins+1), the edges of each bin in um

- *)*

palmari.quot.plot.**track_length**(*tracks*)

Given a set of trajectories in DataFrame format, create a new columns ("track_length") with the length of the corresponding trajectory in frames.

Parameters **tracks** (*pandas.DataFrame*) –

Return type pandas.DataFrame, with the new column

palmari.quot.plot.**wireframe_overlay**(*img*, *model*, *plot=True*)

Make a overlay of two 2-dimensional functions.

Parameters

- **img** (*2D ndarray (YX), with the*) – same shape
- **model** (*2D ndarray (YX), with the*) – same shape

palmari.quot.plot.**wrapup**(*out_png*, *dpi=600*)

Save a figure to a PNG.

Parameters

- **out_png** (*str, save filename*) –
- **dpi** (*int, resolution*) –

palmari.quot.read module

quot/read.py – image file readers

class palmari.quot.read.**ImageReader**(*path*, *start=None*, *stop=None*, ***subregion*)

Bases: object

An image file reader that aims to provide a single API for several image types, relying on other packages to read individual file types. Focuses especially on methods to return temporal frames.

start and *stop* are the frame limits when iterating over the ImageReader. They do not prohibit the user from accessing frames outside this range with ImageReader.get_frame() or related methods.

init path : str start : int, limits of iteration. If not set
defaults to 0.

stop [int, limits of iteration. If not set] defaults to the last frame.

subregion [keys y0, y1, x0, x1 (all int), the] subregion to use for iteration if desired

`close()`

Close the filestream.

`property dtype`

Return the dtype for the arrays produced by self.get_frame() and related methods.

returns str

`get_frame(`*frame_index*, *c=0*`)`

Return a single frame from the file.

args frame_index : int c : int, channel index. Currently only
implemented for ND2 files.

returns 2D ndarray (YX)

`get_frame_range(`*start*, *stop*`)`

Get a contiguous range of frames as a 3D stack.

args start, stop : int

returns 3D ndarray (TYX)

`get_subregion(`*frame_index*, ***subregion*`)`

Get a subregion of a single frame.

args frame_index : int subregion : keys y0, y1, x0, x1,
the subregion limits

returns 3D ndarray (TYX)

`get_subregion_range(`***kwargs*`)`

Get a subregion of multiple frames.

kwargs

y0, y1, x0, x1, start, stop [int, the] subregion limits

returns 3D ndarray (TYX)

`imread()`

Read the entire stack into memory.

returns 3D ndarray (TYX)

`imsave(`*path*, ***kwargs*`)`

Save part of the movie to a TIF.

args path : str, out TIF kwargs : subregion kwargs

max_int_proj(***kwargs*)

Take a maximum intensity projection of the movie.

kwargs start, stop : int, the first and last frames to use

returns 2D ndarray (YX)

min_max(***kwargs*)

Get the minimum and maximum pixel intensities for a frame range.

kwargs start, stop : int, the first and last frames to use

returns (int min, int max)

property shape

Return the number of frames, height, and width of the image movie.

returns (int, int, int)

sum_proj(***kwargs*)

Take a sum projection of the movie.

kwargs start, stop : int, the first and last frames to use

returns 2D ndarray (YX), dtype float64

palmari.quot.read.**read_config**(*path*)

Read a config file.

args path : str

returns dict

palmari.quot.read.**save_config**(*path*, *config*)

Save data in a dict to a TOML file.

args path : str config : dict

palmari.quot.subpixel module

subpixel.py – localize PSFs to subpixel resolution

palmari.quot.subpixel.**centroid**(*I*, *sub_bg=False*)

Find the center of a spot by taking its center of mass.

Parameters

- **I** (*2D ndarray (YX), spot image*) –
- **sub_bg** (*bool, subtract background prior to*) – taking center of mass

Returns

- *dict {* – y : y centroid (pixels), x : x centroid (pixels), bg : estimated background intensity per

  pixel (AU),

  I0 [integrated spot intensity above] background (AU),

  error_flag [int, the error code. 0 indicates] no errors.

- *}*

palmari.quot.subpixel.**localize_frame**(*img*, *positions*, *method=None*, *window_size=9*, *camera_bg=0.0*, *camera_gain=1.0*, ***method_kwargs*)

Run localization on multiple spots in a large 2D image, returning the result as a pandas DataFrame.

Parameters

- **img** (*2D ndarray (YX), the image frame*) –
- **positions** (*2D ndarray of shape (n_spots, 2),*) – the y and x positions at which to localize spots
- **method** (*str, a method in METHODS*) –
- **window_size** (*int, the fitting window size*) –
- **camera_bg** (*float, the BG per subpixel in the camera*) –
- **camera_gain** (*float, the camera gain (grayvalues/photon)*) –
- **method_kwargs** (*to the localization method*) –

Returns plus all of the outputs from the localization function

Return type pandas.DataFrame with columns ['y_detect', 'x_detect']

palmari.quot.subpixel.**ls_int_gaussian**(*I*, *sigma=1.0*, *ridge=0.0001*, *max_iter=10*, *damp=0.3*, *convergence=0.0001*, *divergence=1.0*)

Estimate the maximum likelihood parameters for a symmetric 2D integrated Gaussian PSF model, given an observed spot *I* with normally-distributed noise.

This method uses radial symmetry for a first guess, followed by a Levenberg-Marquardt routine for refinement. The core calculation is performed in quot.helper.fit_ls_int_gaussian.

Parameters

- **I** (*2D ndarray (YX), the observed spot*) –
- **sigma** (*float, static Gaussian width*) –
- **ridge** (*float, initial regularization term*) – for inversion of the Hessian
- **max_iter** (*int, the maximum tolerated number*) – of iterations
- **damp** (*float, damping factor for the update*) – at each iteration. Larger means faster convergence, but less stable.
- **convergence** (*float, maximum magnitude of the update*) – vector at which to call convergence.
- **divergence** (*float, divergence criterion*) –

Returns related parameters about the fitting problem (see below). Some of these can be useful for QC.

Return type dict, the parameter estimate, error estimates, and

palmari.quot.subpixel.**ls_point_gaussian**(*I*, *sigma=1.0*, *ridge=0.0001*, *max_iter=10*, *damp=0.3*, *convergence=0.0001*, *divergence=1.0*)

Estimate the maximum likelihood parameters for a symmetric 2D pointwise-evaluated Gaussian PSF model, given an observed spot *I* with normally-distributed noise.

Both integrated and pointwise-evaluated models have the same underlying model: a symmetric, 2D Gaussian PSF. The distinction is how they handle sampling on discrete pixels:

- **the point Gaussian takes the value on each pixel to** be equal to the PSF function evaluated at the center of that pixel

- **the integrated Gaussian takes the value on each pixel** to be equal to the PSF integrated across the whole area of the pixel

Integrated Gaussian models are more accurate and less prone to edge biases, at the cost of increased complexity and perhaps lower speed.

This method uses radial symmetry for a first guess, followed by a Levenberg-Marquardt routine for refinement. The core calculation is performed in quot.helper.fit_ls_point_gaussian.

Parameters

- **I** (*2D ndarray (YX), the observed spot*) –
- **sigma** (*float, static Gaussian width*) –
- **ridge** (*float, initial regularization term*) – for inversion of the Hessian
- **max_iter** (*int, the maximum tolerated number*) – of iterations
- **damp** (*float, damping factor for the update*) – at each iteration. Larger means faster convergence, but less stable.
- **convergence** (*float, maximum magnitude of the update*) – vector at which to call convergence.
- **divergence** (*float, divergence criterion*) –

Returns related parameters about the fitting problem (see below). Some of these can be useful for QC.

Return type dict, the parameter estimate, error estimates, and

palmari.quot.subpixel.**poisson_int_gaussian**(*I*, *sigma=1.0*, *ridge=0.0001*, *max_iter=10*, *damp=0.3*, *convergence=0.0001*, *divergence=1.0*)

Estimate the maximum likelihood parameters for a symmetric 2D integrated Gaussian PSF model, given an observed spot *I* with Poisson-distributed noise.

This method uses radial symmetry for a first guess, followed by a Levenberg-Marquardt routine for refinement. The core calculation is performed in quot.helper.fit_poisson_int_gaussian.

Parameters

- **I** (*2D ndarray (YX), the observed spot*) –
- **sigma** (*float, static Gaussian width*) –
- **ridge** (*float, initial regularization term*) – for inversion of the Hessian
- **max_iter** (*int, the maximum tolerated number*) – of iterations
- **damp** (*float, damping factor for the update*) – at each iteration. Larger means faster convergence, but less stable.
- **convergence** (*float, maximum magnitude of the update*) – vector at which to call convergence.
- **divergence** (*float, divergence criterion*) –

Returns related parameters about the fitting problem (see below). Some of these can be useful for QC.

Return type dict, the parameter estimate, error estimates, and

palmari.quot.subpixel.**radial_symmetry**(*I*, *sigma=1.0*, ***kwargs*)

Estimate the center of a spot by the radial symmetry method, described in Parthasarathy et al. Nat Met 2012.

Also infer the intensity of the spots assuming an integrated Gaussian PSF. This is useful as a first guess for iterative localization techniques.

Parameters

- **I** (*2D ndarray, spot image*) –
- **sigma** (*float, static integrated 2D Gaussian*) – width

Returns

- *dict {* – y : estimated y center (pixels), x : estimated x center (pixels), I0 : estimated PSF intensity (AU), bg : estimated background intensity per

  pixel (AU),

  error_flag [int, the error flag. 0] if there are no errors,

- *}*

palmari.quot.track module

track.py – reconnect localizations into trajectories

class palmari.quot.track.**ScipyMunkres**

Bases: object

compute(*W*)

class palmari.quot.track.**Trajectory**(*start_idx*, *locs*, *subproblem_shape*, *max_blinks=0*)

Bases: object

Convenience class used internally by track_locs().

A Trajectory object specifies a set of indices to localizations in a large array that are to be reconnected into trajectories.

It also holds a reference to the original array, so that it can grab information about its localization when necessary.

When Trajectories are not reconnected in a given frame, their blink counter (self.n_blinks) is incremented. When this exceeds *max_blinks*, the Trajectories are marked for termination.

The Trajectory class is also convenient to hold associated information about the tracking problem, such as the number of competing trajectories and localizations, etc., that are returned at the end of tracking.

start_idx [int, the index of the first] localization in this Trajectory

locs : 2D ndarray, all localizations subproblem_shape: (int, int), the number of trajs

and locs in the subproblem that created this trajectory

max_blinks [int, the maximum tolerated number] of gaps in tracking

add_index(*idx*, *subproblem_shape*)

Extend this trajectory by one localization.

Parameters

- **idx** (*int, the index of the localization*) – in self.locs to add

- **subproblem_shape** (*(int, int), the size*) – of the tracking subproblem in which this localization was added

blink()

Skip a frame. If a Trajectory has been in blink for more than *self.n_blinks* frames, it marks itself for termination by setting self.active = False.

get_slice()

Return the slice of the localization array that corresponds to this trajectory.

Return type 2D ndarray of shape (traj_len, 5)

last_pos()

Return the last known position of this Trajectory.

Return type (float y, float x), in pixels

palmari.quot.track.**diffusion_weight_matrix**(*trajs*, *locs*, *frame_interval=0.00548*, *pixel_size_um=0.16*, *k_return_from_blink=1.0*, *d_max=5.0*, *y_diff=0.9*, *search_radius=2.5*, *d_bound_naive=0.1*, *init_cost=50.0*)

Generate the weight matrix for reconnection between a set of Trajectories and a set of localizations for the "diffusion" method.

In this method, the weight of reconnecting trajectory A to localization B is equal to the negative log likelihood of trajectory A diffusing to localization B in the relevant frame interval (equal to *frame_interval* if there are no gaps). The likelihood is evaluated with a 2D Brownian motion model.

A weighted combination of two such negative log-likelihoods is used. The first assumes a diffusion coefficient equal to the maximum likelihood diffusion coefficient for that trajectory (using the MSD method). The second assumes a diffusion coefficient equal to d_max. The relative weights of the two estimates are set by *y_diff*.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray, localizations to consider*) – for connection
- **frame_interval** (*float, seconds*) –
- **pixel_size_um** (*float, um*) –
- **k_return_from_blink** (*float, penalty to return a trajectory*) – from blinking status
- **d_max** (*float, the maximum expected diffusion*) – coefficient in um^2 s^-1
- **y_diff** (*float, the relative influence of the*) – particle's local history on its estimated diffusion coefficient
- **search_radius** (*float, um*) –
- **d_bound_naive** (*float, naive estimate for a particle's*) – local diffusion coefficient, um^2 s^-1
- **init_cost** (*float, static cost to initialize a new*) – trajectory when reconnections are available in the search radius

Returns for reconnection

Return type 2D ndarray of shape (n_trajs, n_locs), the weights

palmari.quot.track.**euclidean_weight_matrix**(*trajs*, *locs*, *pixel_size_um=0.16*, *scale=1.0*, *search_radius=2.5*, *init_cost=50.0*, ***kwargs*)

Generate the weight matrix for reconnection between Trajectories and localizations for the "euclidean" reconnection method.

Here, the weight to reconnect traj I with localization J is just the distance between the last known position of I and J, scaled by the constant *scale*.

If J is outside the search radius of I, the weight is infinite.

The weight to drop *I* or to start a new trajectory from *J* when other reconnections are available is *init_cost*.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray, localizations to consider*) – for connection
- **pixel_size_um** (*float, um*) –
- **scale** (*float, inflation factor for the distances*) –
- **search_radius** (*float, um*) –
- **init_cost** (*float, penalty for not performing*) – available reconnections
- **kwargs** (*discarded*) –

Returns weights

Return type 2D ndarray of shape (n_trajs, n_locs), the reconnection

palmari.quot.track.**has_match**(*trajs_0*, *trajs_1*)

Return True if a trajectory in trajs_0 exactly matches a trajectory in trajs_1.

palmari.quot.track.**is_duplicates**(*trajs*)

Return True if there are duplicates in a set of Trajectories.

palmari.quot.track.**reconnect_conservative**(*trajs*, *locs*, *locs_array*, *max_blinks=0*, *frame_interval=0.00548*, *pixel_size_um=0.16*, ***kwargs*)

Only reassign trajs to locs when the assignment is unambiguous (1 traj, 1 loc within the search radius).

For all other trajectories, terminate them. For all other locs, start new trajectories.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray with columns loc_idx, frame,*) – y, x, I0
- **max_blinks** (*int*) –
- **frame_interval** (*float*) –
- **pixel_size_um** (*float*) –
- **kwargs** (*ignored, possibly passed due to upstream*) – method disambiguation

Return type list of Trajectory

palmari.quot.track.**reconnect_diffusion**(*trajs*, *locs*, *locs_array*, *max_blinks=0*, *min_I0=0.0*, ***kwargs*)

Assign Trajectories to localizations on the basis of their expected probability of diffusion and their blinking status.

Each of the Trajectories is assumed to be a Brownian motion in 2D. Its diffusion coefficient is evaluated from its history by MSD if it is greater than length 1, or from d_bound_naive otherwise.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray, localizations to consider*) – for connection
- **locs_array** (*2D ndarray, all localizations in this*) – movie
- **max_blinks** (*int*) –
- **frame_interval** (*float, seconds*) –
- **pixel_size_um** (*float, um*) –
- **min_I0** (*float, AU*) –
- **k_return_from_blink** (*float, penalty to return a trajectory*) – from blinking status
- **d_max** (*float, the maximum expected diffusion*) – coefficient in um^2 s^-1
- **y_diff** (*float, the relative influence of the*) – particle's local history on its estimated diffusion coefficient
- **search_radius** (*float, um*) –
- **d_bound_naive** (*float, naive estimate for a particle's*) – local diffusion coefficient, um^2 s^-1

Return type list of Trajectory

palmari.quot.track.**reconnect_euclidean**(*trajs*, *locs*, *locs_array*, *max_blinks=0*, *min_I0=0.0*, ***kwargs*)

Assign Trajectories to localizations purely by minimizing the total Trajectory-localization distances.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray, localizations to consider*) – for connection
- **locs_array** (*2D ndarray, all localizations in this*) – movie
- **max_blinks** (*int*) –
- **min_I0** (*float, minimum intensity to start a*) – new trajectory
- **pixel_size_um** (*float, um*) –
- **scale** (*float, inflation factor for the distances*) –
- **search_radius** (*float, um*) –
- **init_cost** (*float, cost to start a new trajectory*) – if reconnections are available

Return type list of Trajectory

palmari.quot.track.**reconnect_hungarian**(*trajs*, *locs*, *locs_array*, *max_blinks=0*, *weight_method=None*, *min_I0=0.0*, ***kwargs*)

Assign Trajectories to localizations by assigning each possible reconnection a weight, then finding the assignment that minimizes the summed weights with the Hungarian algorithm.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray, localizations to consider*) – for connection

- **locs_array** (*2D ndarray, all localizations in this*) – movie
- **max_blinks** (*int*) –
- **weight_method** (*str, the method to use to generate*) – the weight matrix
- **min_I0** (*float, minimum intensity to start*) – a new Trajectory
- **kwargs** (*to weight_method*) –

Return type list of Trajectory

palmari.quot.track.**track**(*locs*, *method='diffusion'*, *search_radius=2.5*, *pixel_size_um=0.16*, *frame_interval=0.00548*, *min_I0=0.0*, *max_blinks=0*, *debug=False*, *max_spots_per_frame=None*, *reindex_unassigned=True*, ***kwargs*)

Given a dataframe with localizations, reconnect into trajectories.

Each frame-frame reconnection problem is considered separately and sequentially. For each problem:

1. **Figure out which localizations lie within the** the search radii of the current trajectories
2. **Identify disconnected "subproblems" in this** trajectory-localization adjacency map
3. **Solve all of the subproblems by a method** specified by the *method* kwarg
4. **Update the trajectories and proceed to the** next frame

The result is an assignment of each localization to a trajectory index. Localizations that were not reconnected into a trajectory for whatever reason are assigned a trajectory index of -1.

Parameters

- **locs** (*pandas.DataFrame, set of localizations*) –
- **method** (*str, the tracking method. Currently either*) – "diffusion", "euclidean", or "conservative"
- **search_radius** (*float, max jump length in um*) –
- **pixel_size_um** (*float, um per pixel*) –
- **frame_interval** (*float, seconds*) –
- **min_I0** (*float, only track spots with at least this*) – intensity
- **max_blinks** (*int, number of gaps allowed*) –
- **debug** (*bool*) –
- **max_spots_per_frame** –

  : int, don't track in frames with more than this number of spots
- ****kwargs** (*additional keyword arguments to the tracking*) – method

Return type pandas.Series, trajectory indices for each localization.

palmari.quot.track.**track_subset**(*locs*, *filters*, ***kwargs*)

Run tracking on a subset of trajectories - for instance, only on localizations in frames that meet a particular criterion.

filters should be a set of functions that take trajectory dataframes as argument and return a boolean pandas.Series indicating whether each localization passed or failed the criterion.

Example for the *filters* argument:

filters = [

lambda locs: filter_on_spots_per_frame(locs, max_spots_per_frame=3, filter_kernel=21

)

]

This would only track localizations belonging to frames with 3 or fewer total localizations in that frame. (After smoothing with a uniform kernel of width 21 frames, that is.)

Parameters

- **locs** (*pandas.DataFrame, localizations*) –
- **filters** (*list of functions, the filters*) –
- **kwargs** (*keyword arguments to the tracking method*) –

Return type pandas.DataFrame, trajectories

palmari.quot.track.**traj_loc_distance**(*trajs*, *locs*)

Return the distance between each trajectory and each localization.

Parameters

- **trajs** (*list of Trajectory*) –
- **locs** (*2D ndarray with columns loc_idx,*) – frame, y, x, I0

Returns Trajectory i and localization j

Return type 2D ndarray D, where D[i,j] is the distance between

palmari.quot.trajUtils module

trajUtils.py – functions to compute some common values on trajectories

palmari.quot.trajUtils.**filter_on_spots_per_frame**(*trajs*, *max_spots_per_frame=10*, *filter_kernel=21*)

Mask a set of localizations by the total number of localizations in the corresponding frame.

This function does the following:

1. Take a set of localizations.
2. Calculate the number of localizations in each frame.
3. **Smooth this signal with a uniform kernel of size** *filter_kernel*.
4. **Get the set of frames with total # of spots below** *max_spots_per_frame*.
5. **Mark each localization in these frames with *True*,** and otherwise with *False*.

Parameters

- **trajs** (*pandas.DataFrame, localizations*) –
- **max_spots_per_frame** (*int, the maximum number of spots*) – tolerated per frame
- **filter_kernel** (*int, the size of the smoothing*) – kernel. If *None*, no smoothing is performed.

Returns corresponding localization passed the filter.

Return type pandas.Series with index *trajs.index*. True if the

palmari.quot.trajUtils.**get_max_gap**(*trajs*)

Return the maximum gap present in a set of trajectories.

Parameters **trajs** (*pandas.DataFrame*) –

Returns that trajectories have no gaps (1 frame interval).

Return type int, the maximum gap present. For instance, 1 means

palmari.quot.trajUtils.**get_spots_per_frame**(*trajs*)

Return the number of spots per frame.

Parameters **trajs** (*pandas.DataFrame, localizations*) –

Returns and the max frame index in *trajs*. The column encoding the number of spots per frame is "n_spots_per_frame".

Return type pandas.DataFrame, indexed by frame between 0

palmari.quot.trajUtils.**radial_disp_histograms**(*trajs*, *n_intervals=1*, *pixel_size_um=0.16*, *first_only=False*, *n_gaps=0*, *bin_size=0.001*, *max_disp=5.0*)

Calculate radial displacement histograms for a set of trajectories.

Parameters

- **trajs** (*pandas.DataFrame*) –
- **n_intervals** (*int, the maximum number of frames*) – over which to compute displacements
- **pixel_size_um** (*float, size of pixels (um)*) –
- **first_only** (*bool, only take the displacements*) – relative to the first point of each trajectory
- **n_gaps** (*int, the number of gaps allowed*) – in tracking
- **bin_size** (*float, the spatial bin size in um*) –
- **max_disp** (*float, maximum displacement to*) – consider

Returns

- *(* –

  2D ndarray of shape (n_intervals, n_bins), the radial displacements in each bin for each interval;

  1D ndarray of shape (n_bins+1), the edges of the displacement bins in um

- *)*

palmari.quot.trajUtils.**radial_disps**(*trajs*, *pixel_size_um=0.16*, *first_only=False*)

Calculate the 2D radial displacement of each jump in every trajectory.

Parameters

- **trajs** (*pandas.DataFrame*) –
- **pixel_size_um** (*float, size of pixels*) –
- **first_only** (*float, only compute the first*) – displacement of each trajectory

Return type pandas.DataFrame, with the new column 'radial_disp_um'

palmari.quot.trajUtils.**traj_length**(*trajs*)

Compute the number of localizations corresponding to each trajectory.

Unassigned localizations (with a trajectory index of -1) are given traj_len = 0.

Parameters **trajs** (*pandas.DataFrame, localizations with the*) – 'trajectory' column

Returns column

Return type pandas.DataFrame, the same dataframe with "traj_len"

Module contents

__init__.py

ALL THE CODE IN THIS FOLDER WAS COPIED/ADAPTED FROM https://github.com/alecheckert/quot Which is under MIT LICENSE

palmari.tif_tools package

Submodules

palmari.tif_tools.cell_mask module

palmari.tif_tools.cell_mask.**get_cell_mask**(*data: numpy.ndarray*)

palmari.tif_tools.cell_mask.**tag_localizations_per_cell**(*pos: pandas.core.frame.DataFrame*, *labels: numpy.ndarray*, *scale: float*)

palmari.tif_tools.correct_drift module

palmari.tif_tools.correct_drift.**correct_drift**(*pos*, *L=0.2*, *step_size=0.03*, *prog_bar_position=None*, *min_n_locs_per_bin: int = 10000*, *max_n_bins: int = 20*)

palmari.tif_tools.correct_drift.**get_optimal_shift**(*pos1*, *pos2*, *L*, *step*)

palmari.tif_tools.density_filtering module

palmari.tif_tools.intensity module

palmari.tif_tools.intensity.**mean_intensity_center**(*data*)

palmari.tif_tools.intensity.**smooth_average**(*x*, *w*)

palmari.tif_tools.localization module

palmari.tif_tools.localization.**SMLM_filtering**(*data*, *filter_size*, *scale*)

palmari.tif_tools.localization.**SMLM_localization**(*data: numpy.ndarray*, *factor: float = 1.0*, *filter_size: int = 3*, *scale: float = 2.0*, *verbose: bool = False*, *return_all: bool = False*, *subpixel_mode: str = 'radial'*, *frame_start: int = 0*)

palmari.tif_tools.localization.**b_splines**(*x*, *scale*, *order*)

palmari.tif_tools.localization.**localize_movie**(*movie: Any*, *factor: float = 1.0*, *filter_size: int = 3*, *sliding_filter: bool = False*, *verbose: bool = False*, *subpixel_mode: str = 'radial'*, *progress_bar: bool = False*)

palmari.tif_tools.localization.**lsradialcenterfit**(*m*, *b*, *w*)

Adapted from Matlab code found in https://www.nature.com/articles/nmeth.2071 Least squares solution to determine the radial symmetry center. Inputs m, b, w are defined on a grid. w are the weights for each point.

palmari.tif_tools.localization.**make_filters**(*scale*, *order*, *L*)

palmari.tif_tools.localization.**phaser**(*ROI: numpy.array*)

Adapted from https://colab.research.google.com/drive/1Jir3HxTZ-au8L56ZrNHGxfBD0XlDkOMl

Parameters **ROI** (*np.array*) – 2D array on which to run the dubpixel localization.

Returns x, y, sigma. in pixels.

Return type tuple

palmari.tif_tools.localization.**plus_func**(*x*, *n*)

palmari.tif_tools.localization.**radialCenter**(*I*)

palmari.tif_tools.localization.**sliding_window_filter**(*data: dask.array.core.Array*, *percentile: float = 10*, *window_size: int = 100*)

Module contents

Module contents

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W