ColiCoords Documentation

Release 0.1.5

Jochem Smit

Nov 27, 2019

Contents:

CHAPTER 1

Installation

ColiCoords is available on PyPi and Conda Forge. Currently, python >= 3.6 is required.

Installation by Conda.:

```
conda install -c conda-forge colicoords
```

For installation via PyPi a C++ compiler is required for installing the dependency mahotas. Alternatively, mahotas can be installed separately from Conda.

To install ColiCoords from pypi:

```
pip install colicoords
```

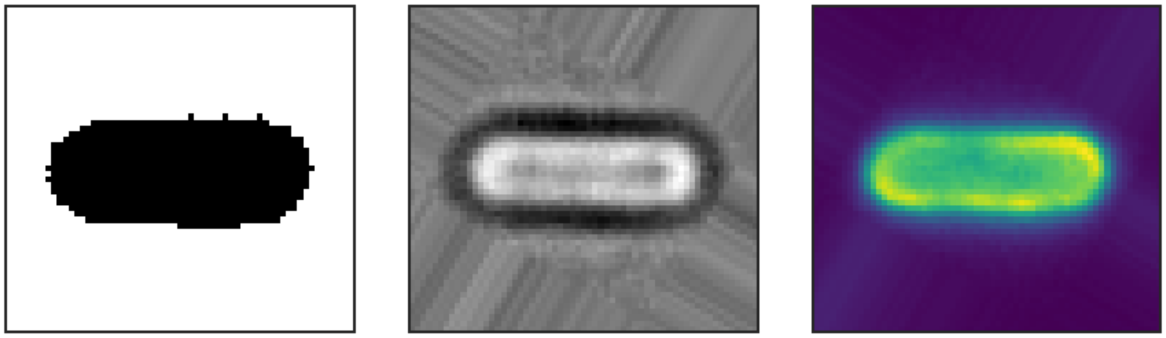
CHAPTER 2

Introduction

ColiCoords is a python project aimed to make analysis of fluorescence microscopy data from rodlike cells more streamlined and intuitive. These goals are achieved by describing the shape of the cell by a 2nd degree polynomial, and this simple mathematical description together with a data structure to organize cell data on a single cell basis allows for straightforward and detailed analysis.

2.1 Using ColiCoords

The basic princlples of *ColiCoords* will first be demonstrated with a simple example. We start out with a binary, brightfield and fluorescence image of a horizontally oriented cell. To turn this data into a `Cell` object we need to first create a `Data` class and use the `add_data()` method to add the images as an `ndarray`, as well as indicate the data class (binary, brightfield, fluorescence or storm). This data interally within the `Cell` object as well as to initialize it.



Binary (left), brightfield (middle) and fluorescence (right) input images.

```
import tifffile
from colicoords import Data, Cell

binary_img = tifffile.imread('data/01_binary.tif')
brightfield_img = tifffile.imread('data/01_brightfield.tif')
fluorescence_img = tifffile.imread('data/01_fluorescence.tif')

data = Data()
data.add_data(binary_img, 'binary')
data.add_data(brightfield_img, 'brightfield')
data.add_data(fluorescence_img, 'fluorescence', name='flu_514')

cell = Cell(data)
```

The *`Cell`* object has two main attributes: `data` and `coords`. The `data_dict` attribute is the instance of *`Data`* used to initialize the `Cell` object and holds all images as `ndarray` subclasses in the attribute `data_dict`. The `coords` attribute is an instance of *`Coordinates`* and is used to optimize the cell's coordinate system and perform related calculations. The coordinate system described by a polynomial of second degree and together with left and right bounds and the radius of the cell they parameterize the coordinate system. These values are first initial guesses based on the binary image but can be optimized iteratively:

```
cell.optimize()
```

More details on optimization of the coordinate system can be found in the section *Coordinate Optimization*. The cells coordinate system allows for the conversion of carthesian input coordinates to be transformed to cell coordinates. The details of the coordinate system and applications are described in section coords.

2.2 Plotting radial distributions

In this section we will go over an example of plotting the radial distribution of the input fluorescence image. `ColiCoords` can plot distribution of signals from both image or localization-based (see also *Processing of SMLM data*) data along both the longitudinal or radial axis.

```
from colicoords.plot import CellPlot
import matplotlib.pyplot as plt

cp = CellPlot(cell)

plt.figure()
cp.imshow('flu_514', cmap='viridis', interpolation='nearest')
cp.plot_outline()
cp.plot_midline()
plt.show()
```

This shows the brightfield image together with the cell outline and midline optimized from the binary image, which was derived from the brightfield image. As can be seen the coordinate system does not completely match the fluorescence image of the cell. This is because the binary image is only a crude estimation of the actual cell position and shape. The coordinate system can be optimize based on the brightfield image to refine the coordinate system. Other input data channels (fluorescence, storm) can be used as described in the section *Coordinate Optimization*.

```
cell.optimize('brightfield')
cell.measure_r('brightfield', mode='min')
```

Here, the first line optimizes the coordinate system to match the shape of the cell as its measured in the brightfield image through an iterative bootstrapping process. Then, in the second line, the radius of the cell is determined from

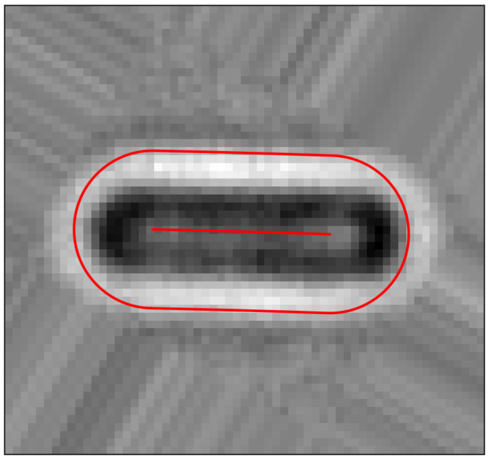


Fig. 1: Brightfield image with cell midline and outline.

the brightfield image. The keyword argument *mode='min'* indicates that in this case the radius of the cell is defined as where the pixel values on the radial distribution are minimum. Note that this value for the radius is not used in transforming coordinates from carthesian to cell coordinates but only in geometrical properties such as cell area or volume. This gives the following result:

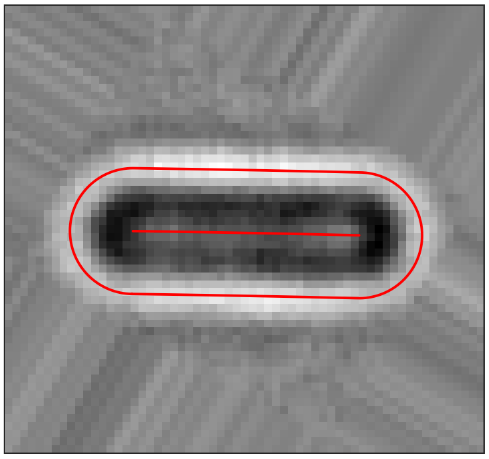


Fig. 2: Brightfield image with optimized coordinate system.

To plot the radial distribution of the `flu_514` fluorescence channel:

```
f, (ax1, ax2) = plt.subplots(1, 2)
cp.plot_r_dist(ax=ax1)
cp.plot_r_dist(ax=ax2, norm_x=True, norm_y=True)
plt.tight_layout()
```

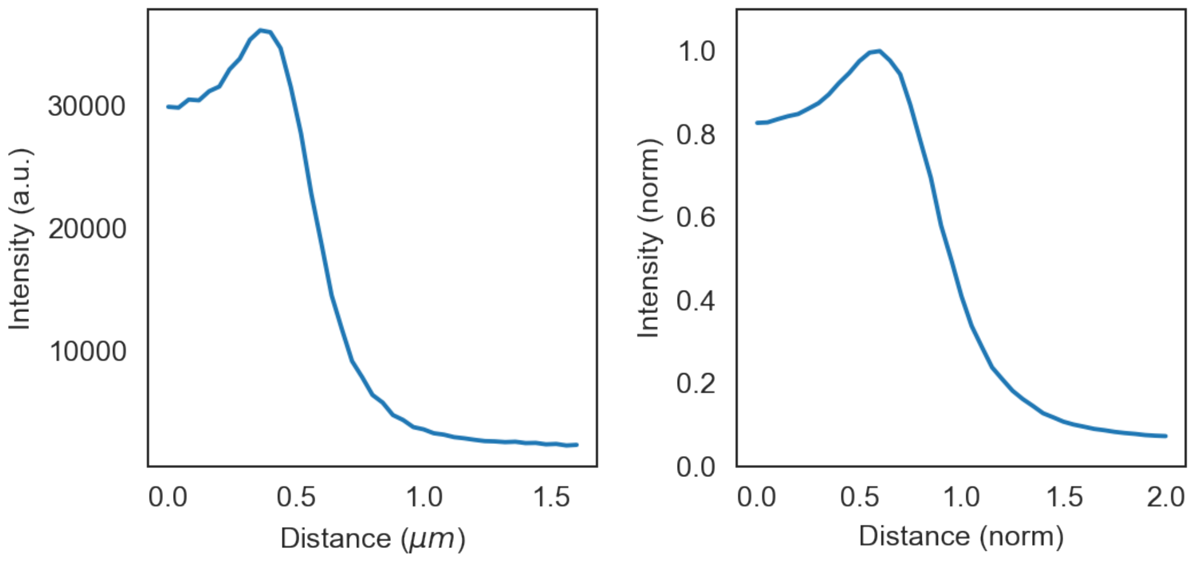


Fig. 3: Radial distribution curve of fluorescence as measured (left) and normalized (right).

The displayed curve is constructed by calcuating the radial distance for every the (x, y) coordinates pair for each pixels. The final curve is calculated from all datapoints by convolution with a gaussian kernel.

The data can be accessed directly from the `Cell` object by calling `r_dist`. The radial distribution curves can be normalized in both `x` and `y` directions. When normalized in the `x` direction the radius obtained from the brightfield image is set to one, thereby eliminating cell-to-cell variations in width.

CHAPTER 3

Batch Processing

`ColiCoords` provides utility functions to easily convert stacks of images of cells optionally together with single-particle localization data (STORM). Binary images of the cells are required to identify the cells positions as well as to determine their orientation to horizontally align the cells. All preprocessing on input data such as background correction, flattening, alignment or drift correction needs to be done before the data is input into `ColiCoords`.

The segmentation process to generate binary images does not need to be done with high accuracy since the binary is only used to identify the position and calcuate initial guesses for the coordinate system. Optimization of the coordinate system can be done in a second step based on other input images (e.g. brightfield or STORM/PAINT membrane marker).

The segmentation can be done via classical methods such as thresholding and watershedding or though cell-analysis software such as CellProfiler or Ilastik. In the *CNN* module of *ColiCoords* provides the user with a Keras/Tensorflow implementation the the U-Net convolutional neural network architecture [RFB15]. The networks are fast an robost and allow for high-troughput segmentation (>10 images/second) on consumer level graphics cards. Example notebooks can be found in the examples directory with implementations of training and applying these neural networks.

3.1 Making a Data object

After preprocessing and segmentation, the data can be input into *ColiCoords*. This is again handled by the *`Data`* object. Images are added as `ndarray` whose shape should be identical. STORM data is input as Structured Arrays (see also *Processing of SMLM data*).

A *`Data`* object can be prepares as follows:

```
import tifffile
from colicoords import Data, data_to_cells

binary_stack = tifffile.imread('data/02/binary_stack.tif')
flu_stack = tifffile.imread('data/02/brightfield_stack.tif')
brightfield_stack = tifffile.imread('data/02/fluorescence_stack.tif')

data = Data()
```

(continues on next page)

(continued from previous page)

```
data.add_data(binary_stack, 'binary')
data.add_data(flu_stack, 'fluorescence')
data.add_data(brightfield_stack, 'brightfield')
```

The *Data* class supports iteration and Numpy-like indexing. This indexing capability is used by the helper function `data_to_cells()` to cut individual cells out of the data across all data channels. Every cell is then oriented horizontally based on the image moments in the binary image (as default). A name attribute is assigned based on the image the cell originates from and the label in the binary image, ie A *Cell* object is initialized together with its own coordinate system and placed in an instance of *CellList*. This object is a container for *Cell* objects and supports Numpy-like indexing and allows for batch operations to be done on all cells in the container.

To generate *Cell* objects from the data and to subsequently optimize all cells' coordinate systems:

3.2 Cell objects and optimization

```
cell_list = data_to_cells(data)
cell_list.optimize('brightfield')
cell_list.measure_r('brightfield', mode='mid')
```

High-performance computing is supported for timely optimizing many cell object though calling *optimize_mp()* (see *Coordinate Optimization*).

The returned *CellList* object is basically an `ndarray` of *colicoords.cell.Cell* objects. Many of the single-cell attributes can be accessed which are returned in the form of a list or array for the whole set of cells.

3.3 Plotting

CellListPlot can be used to easily plot fluorescence distribution of the set of cells or histogram certain properties.

```
from colicoords import CellListPlot

clp = CellListPlot(cell_list)
fig, axes = plt.subplots(2, 2)
clp.hist_property(ax=axes[0,0], tgt='radius')
clp.hist_property(ax=axes[0,1], tgt='length')
clp.hist_property(ax=axes[1,0], tgt='area')
clp.hist_property(ax=axes[1,1], tgt='volume')
plt.tight_layout()
```

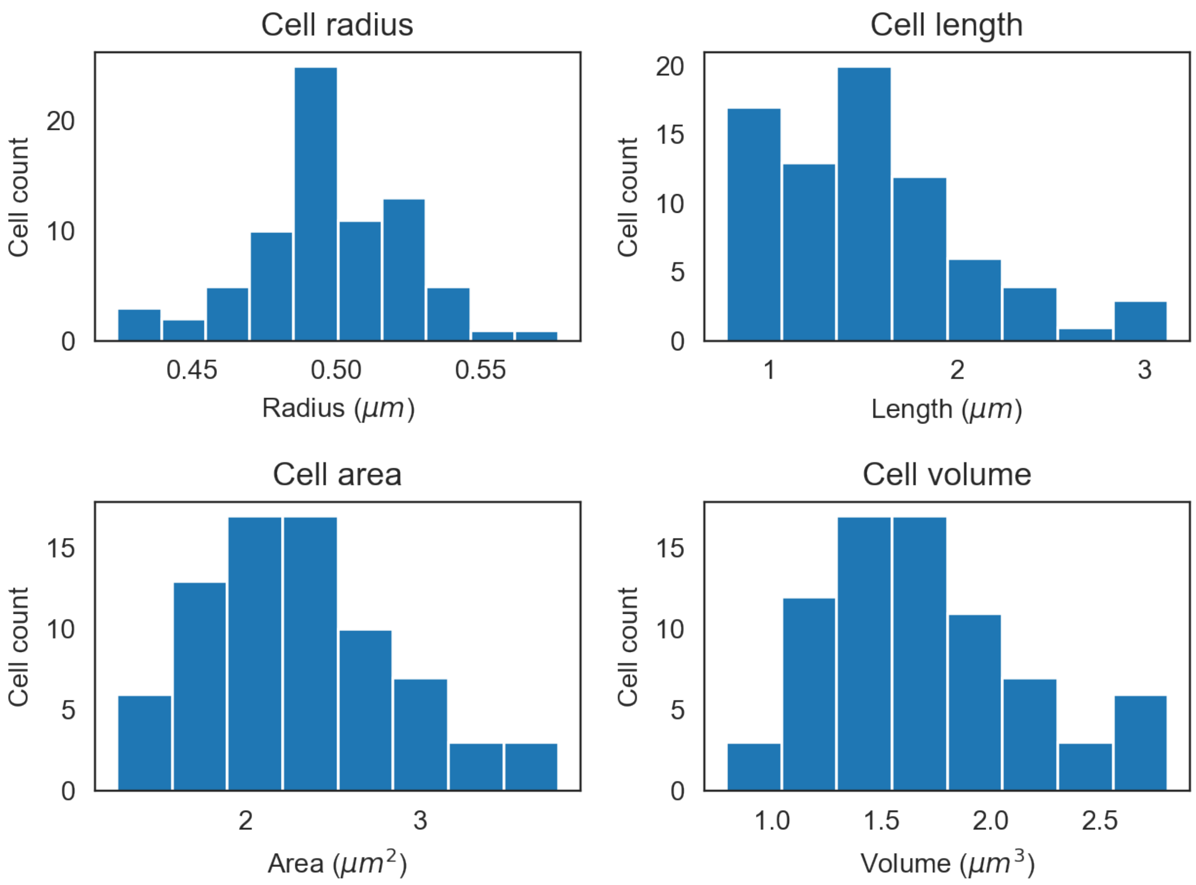
When using *CellList* the function *r_dist()* returns the radial distributions of all cells in the list.

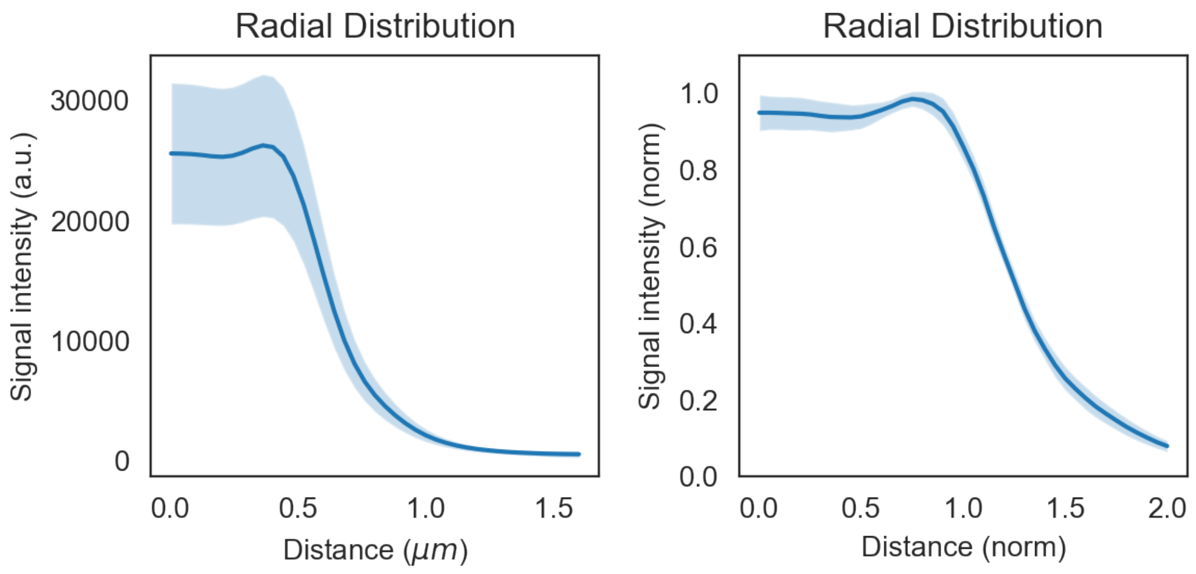
```
x, y = cell_list.r_dist(20, 1)
```

Here, the arguments given are the *stop* and *step* parameters for the x-axis, respectively. The returned *y* is an array where each row holds the radial distribution for a given cell.

To plot the radial distributions via *CellListPlot*:

```
f, axes = plt.subplots(1, 2)
clp.plot_r_dist(ax=axes[0])
axes[0].set_ylim(0, 35000)
clp.plot_r_dist(ax=axes[1], norm_y=True, norm_x=True)
plt.tight_layout()
```


Cell radius
Cell count
Radius (μm)
Cell length
Length (μm)
Cell area
Area (μm^2)
Cell volume
Volume (μm^3)


Radial Distribution
Signal intensity (a.u.)
Distance (μm)
Radial Distribution
Signal intensity (norm)
Distance (norm)

The band around the line shows the sample's standard deviation. By normalizing each curve on the y-axis variation in absolute intensity is eliminated and the curve shows only the shape and its standard deviation. Normalization on the x-axis sets the radius measured by the brightfield in the previous step to one, thereby eleminating cell width variations.

CHAPTER 4

Coordinate Optimization

Upon initialization of a *Cell* object, the parameters of the coordinate system are initialized with on initial guesses based on the binary image of the cell. This is only a rough estimation aimed to provide a starting point for further optimization. In `ColiCoords`, this fitting and optimization is handled by `symfit`, a python package that provides a symbolic and intuitive API for using minimizers from `scipy` or other minimization solutions.

The shorthand approach for optimizing the coordinate system is:

```
cell.optimize()
```

By calling `optimize()` the coordinate system is optimized for the current `Cell` object by using the default settings. This means the optimization is performed on the binary image using the `Powell` minimizer algoritm. Although the optimization is fast, different data sources and minimizers can yield more accurate results.

4.1 Input data classes and cell functions

`ColiCoords` can optimize the coordinate system of the cell based on all compatible data classes, either binary, brightfield, fluorescence, or STORM data. All optimization methods are implemented by calculating some property by applying the current coordinate system on the respective data element, and then comparing this property to the measured data by calculating the chi-squared.

For example, optimimzation based on the brightfield image can be done as follows:

```
cell.optimize('brightfield')
```

Where it is assumed that the brightfield data element is named *'brightfield'*. The appropriate function that is used for the optimization is chosen automatically based on the data class and can be supplied optionally by the *cell_function* keyword argument. Note that optimizaion by brightfield cannot be used to determine the value for the cell's radius parameter, for this the function *measure_r()* has to be used.

In the case of brightfield optimization, a `NumericalCellModel` is generated which is used by `symfit` to perform the minimization. When the default *cell_function* (*CellImageFunction*) is called it first calculates the radial distribution of the brightfield image, and this radial distribution is then used to reconstruct the brightfield image. The resulting image is an estimation of the measured brightfield image and by iterative bootstrapping of this process the optimal parameters can be found. This particular optimization strategy can be use for any roughly isotropic image - ie a cell image that looks identical in all directions radially outward - and is thus independent of brightfield image type and focal plane and can also be applied to selected fluorescence images.

The most accurate approach of optimization is by optimizing on a STORM dataset of a membrane marker. Here, the default *cell_function* used (`CellSTORMMembraneFunction`) calculates for every localization the distance *r* to the midline of the cell. This is compared to the current radius parameter of the coordinate system to give the chi-squared. This fitting is a special case since the dependent data (*y*-data) also depends on the optimization parameter r. To allow a variable dependent data for fitting, the class `RadialData` is used, which mimics a `ndarray`, however whose value depends on the current value of the *r* parameter.

4.2 Minimizers and bounds

Optimization can be done by any `symfit` compatible minimizer. All minimizers are imported via `colicoords.minimizers`. More details on the different minimizers can be found in the `symfit` or `scipy` docs.

The default minimizer, `Powell` is fast but does not always converge to the global minimum. To increase the probability to find the global minimum, the minimizer `DifferentialEvolution` is used. This minimizer searches the parameter space defined by bounds on `Parameter` objects defined in the model scan for candidate solutions.

4.3 Multiprocessing and high-performance computing

The optimization process can take up to tens of seconds per cell, especially if a global minimizer is used. Although the process only needs to take place once, the optimization process of several thousands of cells can take too much time to be conveniently executed on normal desktop PCs. `ColiCoords` therefore supports multiprocessing so that the user can take advantage of parallel high-performance computing. To perform optimization in parallel:

```
cells.optimize_mp()
```

Where *cells* is a `CellList` object. The cells to be divided is equally distributed among the spawned processes, which is by default equal to the number of physical cores present on the host machine.

4.4 Models and advanced usage

The default model used is `NumericalCellModel`. Contrary to typical `symfit` workflow, the `Parameter` objects are defined and initialized by the model itself, and then used to make up the model. To adjust parameter values and bound manually, the user must directly interact with a `CellFit` object instead of calling `optimize()`.

```
from colicoords import CellFit
fit = CellFit(cell)
print(fit.model.params) # [a0, a1, a2, r, xl, xr]
# Set the minimum bound of the `a0` parameter to 5.
fit.model.params[0].min = 5
# Se the value of the `r`parameter to 8.
fit.model.params[3].value = 8
```

The fitting can then be executed by calling `fit.execute()` as usual.

4.5 Custom minimization functions

The minimization function *cell_function* is a subclass of `CellMinimizeFunctionBase` by default. This when this object is used it is initialized by `CellFit` with the instance of the cell object and the name of the target data element. These attributes are then accessible in the custom `__call__` method of the function object.

The `__call__` function must take the coordinate parameters with their values as keyword arguments and should return the calculated data which is compared to the target data element to calculate the chi-squared. Alternatively, the *target_data* property can be used, as is done for `CellSTORMMembraneFunction` to specify a different target.

Alternatively, any custom callable can be given as *cell_function*, as long as it complies with the above requirements.

CHAPTER 5

Processing of SMLM data

SMLM-type datasets can be processed via ColiCoords in a similar fashion as image-based data. The data flow of this type data element is exactly the same as other data classes. The input data format is a numpy structured array, where the required entires are *x*, *y* coordinates as well as a *frame* entry which associated the given localization with an image frame. Optionally, the structured array can be supplemented with additional information such as number of number of photons (intensity), parameters of fitted gaussian or chi-squared value.

5.1 ThunderSTORM

A helper function is provided to load the .csv output from the ThunderSTORM super-resolution analysis package into a numpy structured table which can be used in ColiCoords.

The output from `load_thunderstorm()` is a numpy structured array with at least the fields *x*, *y* and *frame*. The *frame* entry is used for slicing a `Data` object in the z or t dimension; axis 0 for a 3D `colicoords.data_models.Data` object.

```
import tifffile
from colicoords import Data, data_to_cells, load_thunderstorm

storm_table = load_thunderstorm('storm_table.csv')

binary_stack = tifffile.imread('data/02_binary_stack.tif')
flu_stack = tifffile.imread('data/02_brightfield_stack.tif')
brightfield_stack = tifffile.imread('data/02_fluorescence_stack.tif')

data = Data()
data.add_data(binary_stack, 'binary')
data.add_data(flu_stack, 'fluorescence')
data.add_data(brightfield_stack, 'brightfield')
```

5.2 3D-DAOSTORM

A helper function is provided to load the HDF5 output from the 3D-DAOSTORM super-resolution analysis package into a numpy structured table which can be used in ColiCoords.

The output from `load_daostorm()` is a numpy structured array with the fields *x*, *y* and *frame*. The *frame* entry is used for slicing a `Data` object in the z or t dimension; axis 0 for a 3D `colicoords.data_models.Data` object.

```
import tifffile
from colicoords import Data, data_to_cells
from colicoords.daostormIO import load_daostorm

storm_table = load_daostorm('storm_table.hdf5')
binary = tifffile.


binary_stack = tifffile.imread('data/02_binary_stack.tif')
flu_stack = tifffile.imread('data/02_brightfield_stack.tif')
brightfield_stack = tifffile.imread('data/02_fluorescence_stack.tif')

data = Data()
data.add_data(binary_stack, 'binary')
data.add_data(flu_stack, 'fluorescence')
data.add_data(brightfield_stack, 'brightfield')
```

Other super-resolution software will be supported in the future, at the moment users should load the data themselves and parse to a numpy structured array using standard python functions.

CHAPTER 6

Indexing

(section under construction)

```
data_slice = data[5:10, 0:100, 0:100]
print(data.shape)
print(data_slice.shape)
>>> (20, 512, 512)
>>> (20, 100, 100)
```

This particular slicing operation selects images 5 through 10 and takes the upper left 100x100 square. STORM data is automatically sliced accordingly if its present in the data class. This slicing functionality is used by the `data_to_cells` method to obtain single-cell objects.

CHAPTER 7

Aligning Cells

The relative coordinates and associated data of every Cell object can be transformed in order to spatially align and overlap a set of measured cells. To do so, a cell with desired dimensions is generated first:

This model cell is used as a reference for alignment of a set of cells.

```
from colicoords import load, align_cells
cells = load('data/03_synthetic_cells.hdf5')
aligned = align_cells(model_cell, cells, r_norm=True, sigma=1)
```

The return value is a copy of model_cell where all aligned data elements from cells has been added. When the boolean r_norm keyword argument is set to True, the cells radial coordinate are normalized by using the current value of r in their coordinate system.

To align image-based data elements (fluorescence, brightfield) all pixel coordinates of all images are transformed to the model cell's coordinate system. The resulting point cloud is then convoluted with a gaussian kernel where the size of the kernel is determined by the sigma parameter.

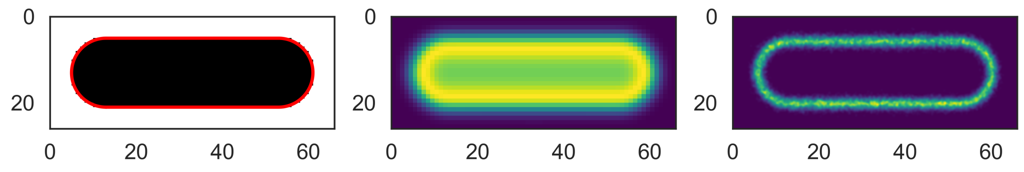
Localization-based data elements (storm) are aligned by transforming all localization parameters to the model cell's coordinate system and then combining all localizations in one data element.

Data elements can be aligned individually by using the function align_data_element.

The final result of the alignment of this dataset can be visualized by:

```
from colicoords import CellPlot
cp = CellPlot(aligned_cell)
fig, axes = plt.subplots(1, 3, figsize=(8, 1.5))
cp.imshow('binary', ax=axes[0])
cp.plot_outline(ax=axes[0])
cp.imshow('fluorescence', ax=axes[1])
cp.plot_storm(method='gauss', ax=axes[2], upscale=10)
```

The above block of code takes about 10 minutes to render the STORM image on a typical CPU due to the large amount of localizations (>35k) and pixel upscale factor used.


0
20
0
20
40
60
0
20
0
20
40
60
0
20
0
20
40
60

CHAPTER 8

Configuration

The `config` module can be used to alter and `ColiCoords`' default configuration values. These are mostly default values in relation to the generation of graphs via the `plot` module and they do not affect coordinate transformations. These default values for plot generation can be overruled by giving explicit keyword arguments to plot functions.

Name	Type	Default value	Units	Description
IMG_PIXELSIZE	`float`	80	Nanometers	Pixel size of the acquired images.
ENDCAP_RANGE	`float`	20.0	Pixels	Default bounds for positions of cell's poles used in bounded optimization.
R_DIST_STOP	`float`	20.0	Pixels	Upper limit for generation of radial distribution curves.
R_DIST_STEP	`float`	0.5	Pixels	Step size between datapoints for generation of radial distribution curves
R_DIST_SIGMA	`float`	0.3	Pixels	Size of the sigma parameter of gaussian used for convolution to generate radial distribution curves.
L_DIST_NBINS	`int`	100	-	Number of bins to generate the longitudinal distribution curves.
L_DIST_SIGMA	`float`	0.5	Pixels	Size of the sigma parameter of gaussian used for convolution to generate longitudinal distribution curves.
PHI_DIST_STEP	`float`	1.0	Degrees	Step size between datapoints for generation of angular distribution curves.
PHI_DIST_SIGMA	`float`	5.0	Degrees	Size of the sigma parameter of gaussian used for convolution to generate longitudinal distribution curves.
CACHE_DIR	`str`		-	Path to the chache dir directory.
DEBUG	`bool`	False	-	Set to `True` to print `numpy` division warnings.

CHAPTER 9

Module Documentation

This page contains the full API docs of `ColiCoords`

9.1 Cell

class `colicoords.cell.`**`Cell`**(*data_object*, *name=None*, *init_coords=True*, ***kwargs*)

ColiCoords' main single-cell object.

This class organizes all single-cell associated data together with an internal coordinate system.

Parameters

data_object [*`Data`*] Holds all data describing this single cell.

coords [*`Coordinates`*] Calculates transformations from/to cartesian and cellular coordinates.

name [`str`] Name identifying the cell (optional).

****kwargs:** Additional kwargs passed to *`Coordinates`*.

Attributes

data [*`Data`*] Holds all data describing this single cell.

coords [*`Coordinates`*] Calculates and optimizes the cell's coordinate system.

name [`str`] Name identifying the cell (optional).

Methods

`copy`(self)	Make a copy of the cell object and all its associated data elements.
`get_intensity`(self[, mask, data_name, func])	Returns the mean fluorescence intensity.

Continued on next page

Table 1 – continued from previous page

`l_classify`(self[, data_name])	Classifies foci in STORM-type data by they x-position along the long axis.
`l_dist`(self, nbins[, start, stop, . . .])	Calculates the longitudinal distribution of signal for a given data element.
`measure_r`(self[, data_name, mode, in_place])	Measure the radius of the cell.
`optimize`(self[, data_name, cell_function, . . .])	Optimize the cell's coordinate system.
`phi_dist`(self, step[, data_name, r_max, . . .])	Calculates the angular distribution of signal for a given data element.
`r_dist`(self, stop, step[, data_name, . . .])	Calculates the radial distribution of a given data element.
`reconstruct_image`(self, data_name[, norm_x, . . .])	Reconstruct the image from a given data element and the cell's current coordinate system.

area
`float`: Area (2d) of the cell in square pixels.

circumference
`float`: Circumference of the cell in pixels.

copy (*self*)
Make a copy of the cell object and all its associated data elements.

This is a deep copy meaning that all numpy data arrays are copied in memory and therefore modifying the copied cell object does not modify the original cell object.

Returns

cell [`Cell`] Copied cell object.

get_intensity (*self*, *mask='binary'*, *data_name=''*, *func=<function mean at 0x7fb3544dd048>*)
Returns the mean fluorescence intensity.

Mean fluorescence intensity either in the region masked by the binary image or reconstructed binary image derived from the cell's coordinate system.

Parameters

mask [`str`] Either 'binary' or 'coords' to specify the source of the mask used. 'binary' uses the binary image as mask, 'coords' uses reconstructed binary from coordinate system.

data_name [`str`:] The name of the image data element to get the intensity values from.

func [`callable`] This function is applied to the data elements pixels selected by the masking operation. The default is *np.mean()*.

Returns

value [`float`] Mean fluorescence pixel value.

l_classify (*self*, *data_name=''*)
Classifies foci in STORM-type data by they x-position along the long axis.

The spots are classified into 3 categories: 'poles', 'between' and 'mid'. The pole category are spots who are to the left and right of xl and xr, respectively. The class 'mid' is a section in the middle of the cell with a total length of half the cell's length, the class 'between' is the remaining two quarters between 'mid' and 'poles'.

Parameters

data_name [`str`] Name of the STORM-type data element to classify. When its not specified the first STORM data element is used.

Returns

l_classes [`tuple`] Tuple with number of spots in poles, between and mid classes, respectively.

`l_dist`(*self*, *nbins*, *start=None*, *stop=None*, *data_name=''*, *norm_x=False*, *l_mean=None*, *r_max=None*, *storm_weight=False*, *method='gauss'*, *sigma=0.5*)

Calculates the longitudinal distribution of signal for a given data element.

Parameters

nbins [`int`] Number of bins between *start* and *stop*.

start [`float`] Distance from *xl* as starting point for the distribution, units are either pixels or normalized units if *norm_x=True*.

stop [`float`] Distance from *xr* as end point for the distribution, units are are either pixels or normalized units if *norm_x=True*.

data_name [`str`] Name of the data element to use.

norm_x [`bool`] If *True* the output distribution will be normalized.

l_mean [`float`, optional] When *norm_x* is *True*, all length coordinates are divided by the length of the cell to normalize it. If *l_mean* is given, the length coordinates at the poles are divided by *l_mean* instead to allow equal scaling of all pole regions.

r_max [`float`, optional] Datapoints within r_max from the cell midline will be included. If *None* the value from the cell's coordinate system will be used.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

method [`str`] Method of averaging datapoints to calculate the final distribution curve.

sigma [`float`] Applies only when *method* is set to 'gauss'. *sigma* gives the width of the gaussian used for convoluting datapoints.

Returns

xvals [`ndarray`] Array of distances along the cell midline, values are the middle of the bins/kernel.

yvals [`ndarray`] Array of bin heights.

`length`

`float`: Length of the cell in pixels.

`measure_r`(*self*, *data_name='brightfield'*, *mode='max'*, *in_place=True*, ***kwargs*)

Measure the radius of the cell.

The radius is found by the intensity-mid/min/max-point of the radial distribution derived from brightfield (default) or another data element.

Parameters

data_name [`str`] Name of the data element to use.

mode [`str`] Mode to find the radius. Can be either 'min', 'mid' or 'max' to use the minimum, middle or maximum value of the radial distribution, respectively.

in_place [`bool`] If *True* the found value of *r* is directly substituted in the cell's coordinate system, otherwise the value is returned.

Returns

radius [`float`] The measured radius *r* if *in_place* is *False*, otherwise *None*.

optimize(*self*, *data_name='binary'*, *cell_function=None*, *minimizer=<class 'symfit.core.minimizers.Powell'>*, ***kwargs*)

Optimize the cell's coordinate system.

The optimization is performed on the data element given by `data_name` using the function *cell_function*. A default function depending on the data class is used of objective is omitted.

Parameters

data_name [`str`, optional] Name of the data element to perform optimization on.

cell_function Optional subclass of *`CellMinimizeFunctionBase`* to use as objective function.

minimizer [Subclass of `symfit.core.minimizers.BaseMinimizer` or `Sequence`] Minimizer to use for the optimization. Default is the `Powell` minimizer.

****kwargs :** Additional kwargs are passed to `execute()`.

Returns

result [`FitResults`] `symfit` fit results object.

phi_dist(*self*, *step*, *data_name=''*, *r_max=None*, *r_min=0*, *storm_weight=False*, *method='gauss'*, *sigma=5*)

Calculates the angular distribution of signal for a given data element.

Parameters

step [`float`] Step size between datapoints.

data_name [`str`] Name of the data element to use.

r_max [`float`, optional] Datapoints within r_max from the cell midline will be included. If *None* the value from the cell's coordinate system will be used.

r_min [`float`, optional] Datapoints outside of r_min from the cell midline will be included.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

method [`str`] Method of averaging datapoints to calculate the final distribution curve.

sigma [`float`] Applies only when *method* is set to 'gauss'. *sigma* gives the width of the gaussian used for convoluting datapoints.

Returns

xvals [`ndarray`] Array of angles along the cell pole, values are the middle of the bins/kernel.

yvals_l [`ndarray`] Array of bin heights for the left pole.

yvals_r [`ndarray`] Array of bin heights for the right pole.

r_dist(*self*, *stop*, *step*, *data_name=''*, *norm_x=False*, *limit_l=None*, *storm_weight=False*, *method='gauss'*, *sigma=0.3*)

Calculates the radial distribution of a given data element.

Parameters

stop [`float`] Until how far from the cell spine the radial distribution should be calculated.

step [`float`] The binsize of the returned radial distribution.

data_name [`str`] The name of the data element on which to calculate the radial distribution.

norm_x [`bool`] If *True* the returned distribution will be normalized with the cell's radius set to 1.

limit_l [`str`] If *None*, all datapoints are used. This can be limited by providing the value *full* (omit poles only), 'poles' (include only poles), or a float value between 0 and 1 which will limit the data points by longitudinal coordinate around the midpoint of the cell.

storm_weight [`bool`] Only applicable for analyzing STORM-type data elements. If *True* the returned histogram is weighted with the values in the 'Intensity' field.

method [`str`, either 'gauss' or 'box'] Method of averaging datapoints to calculate the final distribution curve.

sigma [`float`] Applies only when *method* is set to 'gauss'. *sigma* gives the width of the gaussian used for convoluting datapoints.

Returns

xvals [`ndarray`] Array of distances from the cell midline, values are the middle of the bins.

yvals [`ndarray`] Array of in bin heights.

radius

`float`: Radius of the cell in pixels.

reconstruct_image(*self*, *data_name*, *norm_x=False*, *r_scale=1*, ***kwargs*)

Reconstruct the image from a given data element and the cell's current coordinate system.

Parameters

data_name [`str`] Name of the data element to use.

norm_x [`bool`] Boolean indicating whether or not to normalize to r=1.

r_scale [`float`] Stretch or compress the image in the radial direction by this factor. Values > 1 will compress the image.

****kwargs** Optional keyword arguments are 'stop' and 'step' which are passed to *r_dist*.

Returns

img [`ndarray`] Image of the reconstructed cell.

surface

`float`: Total surface area (3d) of the cell in square pixels.

volume

`float`: Volume of the cell in cubic pixels.

class `colicoords.cell.`**CellList**(*cell_list*)

List equivalent of the *`Cell`* object.

This Object holding a list of cell objects exposing several methods to either apply functions to all cells or to extract values from all cell objects. It supports iteration over Cell objects and Numpy-style array indexing.

Parameters

cell_list [`list` or `numpy.ndarray`] List of array of *`Cell`* objects.

Attributes

cell_list [`ndarray`] Numpy array of *Cell* objects

data [CellListData] Object with common attributes for all cells

Methods

append(self, cell_obj)	Append Cell object *cell_obj* to the list of cells.
copy(self)	Make a copy of the *CellList* object and all its associated data elements.
execute(self, worker)	Apply worker function *worker* to all cell objects and returns the results.
execute_mp(self, worker[, processes])	Apply worker function *worker* to all cell objects and returns the results.
get_intensity(self[, mask, data_name, func])	Returns the fluorescence intensity for each cell.
l_classify(self[, data_name])	Classifies foci in STORM-type data by they x-position along the long axis.
l_dist(self, nbins[, start, stop, . . .])	Calculates the longitudinal distribution of signal for a given data element for all cells.
measure_r(self[, data_name, mode, in_place])	Measure the radius of the cells.
optimize(self[, data_name, cell_function, . . .])	Optimize the cell's coordinate system.
optimize_mp(self[, data_name, . . .])	Optimize all cell's coordinate systems using *optimize* through parallel computing.
phi_dist(self, step[, data_name, . . .])	Calculates the angular distribution of signal for a given data element for all cells.
r_dist(self, stop, step[, data_name, . . .])	Calculates the radial distribution for all cells of a given data element.

append (*self*, *cell_obj*)
Append Cell object *cell_obj* to the list of cells.

Parameters

cell_obj [Cell] Cell object to append to current cell list.

area
ndarray: Array of cell's area in square pixels

circumference
ndarray: Array of cell's circumference in pixels

copy (*self*)
Make a copy of the *CellList* object and all its associated data elements.

This is a deep copy meaning that all numpy data arrays are copied in memory and therefore modifying the copied cell objects does not modify the original cell objects.

Returns

cell_list [CellList] Copied *CellList* object

execute (*self*, *worker*)
Apply worker function *worker* to all cell objects and returns the results.

Parameters

worker [callable] Worker function to be executed on all cell objects.

Returns

res [list] List of resuls returned from *worker*

execute_mp(*self*, *worker*, *processes=None*)
Apply worker function *worker* to all cell objects and returns the results.

Parameters

worker [`callable`] Worker function to be executed on all cell objects.

processes [`int`] Number of parallel processes to spawn. Default is the number of logical processors on the host machine.

Returns

res [`list`] List of results returned from `worker`.

get_intensity(*self*, *mask='binary'*, *data_name="*, *func=<function mean at 0x7fb3544dd048>*)
Returns the fluorescence intensity for each cell.

Mean fluorescence intensity either in the region masked by the binary image or reconstructed binary image derived from the cell's coordinate system. The default return value is the mean fluorescence intensity. Integrated intensity can be calculated by using *func=np.sum*.

Parameters

mask [`str`] Either 'binary' or 'coords' to specify the source of the mask used. 'binary' uses the binary image as mask, 'coords' uses reconstructed binary from coordinate system

data_name [`str`] The name of the image data element to get the intensity values from.

func [`callable`] This function is applied to the data elements pixels selected by the masking operation. The default is *np.mean()*.

Returns

value [`float`] Mean fluorescence pixel value.

l_classify(*self*, *data_name="*)
Classifies foci in STORM-type data by they x-position along the long axis.

The spots are classified into 3 categories: 'poles', 'between' and 'mid'. The pole category are spots who are to the left and right of xl and xr, respectively. The class 'mid' is a section in the middle of the cell with a total length of half the cell's length, the class 'between' is the remaining two quarters between 'mid' and 'poles'

Parameters

data_name [`str`] Name of the STORM-type data element to classify. When its not specified the first STORM data element is used.

Returns

array [`ndarray`] Array of tuples with number of spots in poles, between and mid classes, respectively.

l_dist(*self*, *nbins*, *start=None*, *stop=None*, *data_name="*, *norm_x=True*, *method='gauss'*, *r_max=None*, *storm_weight=False*, *sigma=None*)
Calculates the longitudinal distribution of signal for a given data element for all cells.

Normalization by cell length is enabled by default to remove cell-to-cell variations in length.

Parameters

nbins [`int`] Number of bins between *start* and *stop*.

start [`float`] Distance from *xl* as starting point for the distribution, units are either pixels or normalized units if *norm_x=True*.

stop [`float`] Distance from *xr* as end point for the distribution, units are are either pixels or normalized units if *norm_x=True*.

data_name [`str`] Name of the data element to use.

norm_x [`bool`] If *True* the output distribution will be normalized.

r_max [`float`, optional] Datapoints within r_max from the cell midline will be included. If *None* the value from the cell's coordinate system will be used.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

method [`str`] Method of averaging datapoints to calculate the final distribution curve.

sigma [`float` or array_like] Applies only when *method* is set to 'gauss'. *sigma* gives the width of the gaussian used for convoluting datapoints. To use a different sigma for each cell *sigma* can be given as a list or array.

Returns

xvals [`ndarray`] Array of distances along the cell midline, values are the middle of the bins/kernel

yvals [`ndarray`] 2D array where every row is the bin heights per cell.

`length`
`ndarray` Array of cell's lengths in pixels

`measure_r`(*self*, *data_name='brightfield'*, *mode='max'*, *in_place=True*, ***kwargs*)
Measure the radius of the cells.

The radius is found by the intensity-mid/min/max-point of the radial distribution derived from brightfield (default) or another data element.

Parameters

data_name [`str`] Name of the data element to use.

mode [`str`] Mode to find the radius. Can be either 'min', 'mid' or 'max' to use the minimum, middle or maximum value of the radial distribution, respectively.

in_place [`bool`] If *True* the found value of *r* is directly substituted in the cell's coordinate system, otherwise the value is returned.

Returns

radius [`np.ndarray`] The measured radius *r* values if *in_place* is *False*, otherwise *None*.

`name`
`ndarray`: Array of cell's names

`optimize`(*self*, *data_name='binary'*, *cell_function=None*, *minimizer=<class 'symfit.core.minimizers.Powell'>*, ***kwargs*)
Optimize the cell's coordinate system.

The optimization is performed on the data element given by `data_name` using objective function *objective*. A default depending on the data class is used of objective is omitted.

Parameters

data_name [`str`, optional] Name of the data element to perform optimization on.

cell_function Optional subclass of *`CellMinimizeFunctionBase`* to use as objective function.

minimizer [Subclass of `symfit.core.minimizers.BaseMinimizer` or `Sequence`] Minimizer to use for the optimization. Default is the `Powell` minimizer.

****kwargs :** Additional kwargs are passed to `execute()`.

Returns

res_list [`list` of `FitResults`] List of *symfit* `FitResults` object.

`optimize_mp`(*self*, *data_name='binary'*, *cell_function=None*, *minimizer=<class 'symfit.core.minimizers.Powell'>*, *processes=None*, ***kwargs*)

Optimize all cell's coordinate systems using *optimize* through parallel computing.

A call to this method must be protected by if __name__ == '__main__' if its not executed in jupyter notebooks.

Parameters

data_name [`str`, optional] Name of the data element to perform optimization on.

cell_function Optional subclass of *`CellMinimizeFunctionBase`* to use as objective function.

minimizer [Subclass of `symfit.core.minimizers.BaseMinimizer` or `Sequence`] Minimizer to use for the optimization. Default is the `Powell` minimizer.

processes [`int`] Number of parallel processes to spawn. Default is the number of logical processors on the host machine.

****kwargs :** Additional kwargs are passed to `execute()`.

Returns

res_list [`list` of `FitResults`] List of *symfit* `FitResults` object.

`phi_dist`(*self*, *step*, *data_name=''*, *storm_weight=False*, *method='gauss'*, *sigma=5*, *r_max=None*, *r_min=0*)

Calculates the angular distribution of signal for a given data element for all cells.

Parameters

step [`float`] Step size between datapoints.

data_name [`str`] Name of the data element to use.

r_max [`float`, optional] Datapoints within r_max from the cell midline will be included. If *None* the value from the cell's coordinate system will be used.

r_min [`float`, optional] Datapoints outside of r_min from the cell midline will be included.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

method [`str`] Method of averaging datapoints to calculate the final distribution curve.

sigma [`float`] Applies only when *method* is set to 'gauss'. *sigma* gives the width of the gaussian used for convoluting datapoints.

Returns

xvals [`ndarray`] Array of distances along the cell midline, values are the middle of the bins/kernel.

yvals_l [`ndarray`] Array of bin heights for the left pole.

yvals_r [ndarray] Array of bin heights for the right pole.

r_dist(*self*, *stop*, *step*, *data_name=''*, *norm_x=False*, *limit_l=None*, *storm_weight=False*, *method='gauss'*, *sigma=0.3*)
Calculates the radial distribution for all cells of a given data element.

Parameters

stop [float] Until how far from the cell spine the radial distribution should be calculated

step [float] The binsize of the returned radial distribution

data_name [str] The name of the data element on which to calculate the radial distribution

norm_x [bool] If *True* the returned distribution will be normalized with the cell's radius set to 1.

limit_l [str] If *None*, all datapoints are used. This can be limited by providing the value *full* (omit poles only), 'poles' (include only poles), or a float value between 0 and 1 which will limit the data points by longitudinal coordinate around the midpoint of the cell.

storm_weight [bool] Only applicable for analyzing STORM-type data elements. If *True* the returned histogram is weighted with the values in the 'Intensity' field.

method [str, either 'gauss' or 'box'] Method of averaging datapoints to calculate the final distribution curve.

sigma [float] Applies only when *method* is set to 'gauss'. *sigma* gives the width of the gaussian used for convoluting datapoints

Returns

xvals [ndarray] Array of distances from the cell midline, values are the middle of the bins

yvals [ndarray] 2D Array where each row is the bin heights for each cell.

radius
ndarray Array of cell's radii in pixels

surface
ndarray: Array of cell's surface area (3d) in square pixels

volume
ndarray: Array of cell's volume in cubic pixels

class colicoords.cell.**Coordinates**(*data*, *initialize=True*, ***kwargs*)
Cell's coordinate system described by the polynomial p(x) and associated functions.

Parameters

data [*Data*] The *data* object defining the shape.

initialize [bool, optional] If *False* the coordinate system parameters are not initialized with initial guesses.

****kwargs** Can be used to manually supply parameter values if *initialize* is *False*.

Attributes

xl [float] Left cell pole x-coordinate.

xr [float] Right cell pole x-coordinate.

r [float] Cell radius.

coeff [ndarray] Coefficients [a0, a1, a2] of the polynomial a0 + a1*x + a2*x**2 which describes the cell's shape.

Methods

calc_lc(self, xp, yp)	Calculates distance of xc along the midline the cell corresponding to the points (xp, yp).
calc_perimeter(self, xp, yp)	Calculates how far along the perimeter of the cell the points (xp, yp) lay.
calc_phi(self, xp, yp)	Calculates the angle between the line perpendical to the cell midline and the line between (xp, yp) and (xc, p(xc)).
calc_rc(self, xp, yp)	Calculates the distance of (xp, yp) to (xc, p(xc)).
calc_xc(self, xp, yp)	Calculates the coordinate xc on p(x) closest to xp, yp.
calc_xc_mask(self, xp, yp)	Calculated whether point (xp, yp) is in either the left or right polar areas, or in between.
calc_xc_masked(self, xp, yp)	Calculates the coordinate xc on p(x) closest to (xp, yp), where xl < xc < xr.
full_transform(self, xp, yp)	Transforms image coordinates (xp, yp) to cell coordinates (xc, lc, rc, psi).
get_core_points(self[, xl, xr])	Returns the coordinates of the roughly estimated 'core' points of the cell.
get_idx_xc(self, xp, yp)	Finds the indices of the arrays xp an yp where they either belong to the left or right polar regions, as well as coordinates xc.
p(self, x_arr)	Calculate p(x).
p_dx(self, x_arr)	Calculate the derivative p'(x) evaluated at x.
q(self, x, xp)	array_like: Returns q(x) where q(x) is the line perpendicular to p(x) at xp
rev_calc_perimeter(self, par_values)	For a given distance along the perimeter calculate the *xp*, *yp* cartesian coordinates.
rev_transform(self, lc, rc, phi[, l_norm])	Reverse transform from cellular coordinates *lc*, *rc*, *phi* to cartesian coordinates *xp*, *yp*.
sub_par(self, par_dict)	Substitute the values in *par_dict* as the coordinate systems parameters.
transform(self, xp, yp)	Transforms image coordinates (xp, yp) to cell coordinates (lc, rc, psi)

a0

float: Polynomial p(x) 0th degree coefficient.

a1

float: Polynomial p(x) 1st degree coefficient.

a2

float: Polynomial p(x) 2nd degree coefficient.

calc_lc(*self*, *xp*, *yp*)

Calculates distance of xc along the midline the cell corresponding to the points (xp, yp).

The returned value is the distance from the points (xp, yp) to the midline of the cell.

Parameters

xp [float: or ndarray] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

lc [`float` or `ndarray`] Distance along the midline of the cell.

`calc_perimeter`(*self*, *xp*, *yp*)

Calculates how far along the perimeter of the cell the points (xp, yp) lay.

The perimeter of the cell is the current outline as described by the current coordinate system. The zero-point is the top-left point where the top membrane section starts (lc=0, phi=0) and increases along the perimeter clockwise.

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

per [`float` or `ndarray`] Length along the cell perimeter.

`calc_phi`(*self*, *xp*, *yp*)

Calculates the angle between the line perpendical to the cell midline and the line between (xp, yp) and (xc, p(xc)).

The returned values are in degrees. The angle is defined to be 0 degrees for values in the upper half of the image (yp < p(xp)), running from 180 to zero along the right polar region, 180 degrees in the lower half and running back to 0 degrees along the left polar region.

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

phi [`float` or `ndarray`] Angle phi for (xp, yp).

`calc_rc`(*self*, *xp*, *yp*)

Calculates the distance of (xp, yp) to (xc, p(xc)).

The returned value is the distance from the points (xp, yp) to the midline of the cell.

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

rc [`float` or `ndarray`] Distance to the midline of the cell.

`calc_xc`(*self*, *xp*, *yp*)

Calculates the coordinate xc on p(x) closest to xp, yp.

All coordinates are cartesian. Solutions are found by solving the cubic equation.

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

xc [`float` or `ndarray`] Cellular x-coordinate for point(s) xp, yp

`calc_xc_mask`(*self*, *xp*, *yp*)
Calculated whether point (xp, yp) is in either the left or right polar areas, or in between.

Returned values are 1 for left pole, 2 for middle, 3 for right pole.

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

xc_mask [`float`: or `ndarray`:] Array to mask different cellular regions.

`calc_xc_masked`(*self*, *xp*, *yp*)
Calculates the coordinate xc on p(x) closest to (xp, yp), where xl < xc < xr.

Parameters

xp [`float`: or `ndarray`:] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float`: or `ndarray`:] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

xc_mask [`float` or `ndarray`] Cellular x-coordinate for point(s) xp, yp, where xl < xc < xr.

`full_transform`(*self*, *xp*, *yp*)
Transforms image coordinates (xp, yp) to cell coordinates (xc, lc, rc, psi).

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp.

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp.

Returns

coordinates [`tuple`] Tuple of cellular coordinates xc, lc, rc, psi.

`get_core_points`(*self*, *xl=None*, *xr=None*)
Returns the coordinates of the roughly estimated 'core' points of the cell.

Used for determining the initial guesses for the coefficients of p(x).

Parameters

xl [float, optional] Starting point x of where to get the 'core' points.

xr [float, optional] End point x of where to get the 'core' points.

Returns

xvals [np.ndarray] Array of x coordinates of 'core' points.

yvals [np.ndarray] Array of y coordinates of 'core' points.

get_idx_xc (*self*, *xp*, *yp*)
Finds the indices of the arrays xp an yp where they either belong to the left or right polar regions, as well as coordinates xc.

Parameters

xp [ndarray] Input x-coordinates. Must be the same shape as yp.

yp [ndarray] Input y-coordinates. Must be the same shape as xp.

Returns

idx_left [ndarray] Index array of elements in the area of the cell's left pole.

idx_right [ndarray] Index array of elements in the area of cell's right pole.

xc [ndarray] Cellular coordinates *xc* corresponding to *xp*, *yp*, extending into the polar regions.

lc
ndarray: Matrix of shape m x n equal to cell with distance l along the cell mideline.

length
float: Length of the cell in pixels.

p (*self*, *x_arr*)
Calculate p(x).

The function p(x) describes the midline of the cell.

Parameters

x_arr [ndarray] Input x values.

Returns

p [ndarray] Evaluated polynomial p(x)

p_dx (*self*, *x_arr*)
Calculate the derivative p'(x) evaluated at x.

Parameters

x_arr :class:'~numpy.ndarray': Input x values.

Returns

p_dx [ndarray] Evaluated function p'(x).

phi
ndarray: Matrix of shape m x n equal to cell with angle psi relative to the cell midline.

q (*self*, *x*, *xp*)
array_like: Returns q(x) where q(x) is the line perpendicular to p(x) at xp

rc
ndarray: Matrix of shape m x n equal to cell with distance r to the cell midline.

rev_calc_perimeter (*self*, *par_values*)

For a given distance along the perimeter calculate the *xp*, *yp* cartesian coordinates.

Parameters

par_values [`float` or `ndarray`] Input parameter values. Must be between 0 and *:attr:~colicoords.Cell.circumference*

Returns

xp [`float` or `ndarray`] Cartesian x-coordinate corresponding to *lc*, *rc*, *phi*

yp [`float` or `ndarray`] Cartesian y-coordinate corresponding to *lc*, *rc*, *phi*

rev_transform (*self*, *lc*, *rc*, *phi*, *l_norm=True*)

Reverse transform from cellular coordinates *lc*, *rc*, *phi* to cartesian coordinates *xp*, *yp*.

Parameters

lc [`float` or `ndarray`] Input scalar or vector/matrix l-coordinate.

rc [`float` or `ndarray`] Input scalar or vector/matrix l-coordinate.

phi [`float` or `ndarray`] Input scalar or vector/matrix l-coordinate.

l_norm [`bool`, optional] If *True* (default), the lc coordinate has to be input as normalized.

Returns

xp [`float` or `ndarray`] Cartesian x-coordinate corresponding to *lc*, *rc*, *phi*

yp [`float` or `ndarray`] Cartesian y-coordinate corresponding to *lc*, *rc*, *phi*

sub_par (*self*, *par_dict*)

Substitute the values in *par_dict* as the coordinate systems parameters.

Parameters

par_dict [`dict`] Dictionary with parameters which values are set to the attributes.

transform (*self*, *xp*, *yp*)

Transforms image coordinates (xp, yp) to cell coordinates (lc, rc, psi)

Parameters

xp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as yp

yp [`float` or `ndarray`] Input scalar or vector/matrix x-coordinate. Must be the same shape as xp

Returns

coordinates [`tuple`] Tuple of cellular coordinates lc, rc, psi

x_coords

`ndarray``: Matrix of shape m x n equal to cell image with cartesian x-coordinates.

xc

`ndarray`: Matrix of shape m x n equal to cell image with x coordinates on p(x)

xc_mask

`ndarray`: Matrix of shape m x n equal to cell image where elements have values 1, 2, 3 for left pole, middle and right pole, respectively.

xc_masked

`ndarray`: Matrix of shape m x n equal to cell image with x coordinates on p(x) where xl < xc < xr.

y_coords
`ndarray`: Matrix of shape m x n equal to cell image with cartesian y-coordinates.

yc
`ndarray`: Matrix of shape m x n equal to cell image with y coordinates on p(x)

`colicoords.cell.`**`calc_lc`**(*xl*, *xr*, *coeff*)
Calculate *lc*.

The returned length is the arc length from *xl* to *xr* integrated along the polynomial p(x) described by *coeff*.

Parameters

xl [array_like] Left bound to calculate arc length from. Shape must be compatible with *xl*.

xr [array_like] Right bound to calculate arc length to. Shape must be compatible with *xr*.

coeff [array_like or `tuple`] Array or tuple with coordinate polynomial coefficients *a0*, *a1*, *a2*.

Returns

l [array_like] Calculated length *lc*.

`colicoords.cell.`**`optimize_worker`**(*cell*, ***kwargs*)
Worker object for optimize multiprocessing.

Parameters

cell [*`Cell`*] Cell object to optimize.

****kwargs** Additional keyword arguments passed to *`optimize()`*

Returns

result [`FitResults`]

`colicoords.cell.`**`solve_general`**(*a*, *b*, *c*, *d*)
Solve cubic polynomial in the form a*x^3 + b*x^2 + c*x + d.

Only works if polynomial discriminant < 0, then there is only one real root which is the one that is returned. [1]

Parameters

a [array_like] Third order polynomial coefficient.

b [array_like] Second order polynomial coefficient.

c [array_like] First order polynomial coefficient.

d [array_like] Zeroth order polynomial coefficient.

Returns

array [array_like] Real root solution.

`colicoords.cell.`**`solve_length`**(*xr*, *xl*, *coeff*, *length*)
Used to find *xc* in reverse coordinate transformation.

Function used to find cellular x coordinate *xr* where the arc length from *xl* to *xr* is equal to length given a coordinate system with *coeff* as coefficients.

Parameters

xr [`float`] Right boundary x coordinate of calculated arc length.

xl [`float`] Left boundary x coordinate of calculated arc length.

coeff [`list` or `ndarray`] Coefficients a0, a1, a2 describing the coordinate system.

length [`float`] Target length.

Returns

diff [`float`] Difference between calculated length and specified length.

`colicoords.cell.`**`solve_trig`**(*a*, *b*, *c*, *d*)

Solve cubic polynomial in the form a*x^3 + b*x^2 + c*x + d Only for polynomial discriminant > 0, the polynomial has three real roots [1]

Parameters

a [array_like] Third order polynomial coefficient.

b [array_like] Second order polynomial coefficient.

c [array_like] First order polynomial coefficient.

d [array_like] Zeroth order polynomial coefficient.

Returns

array [array_like] First real root solution.

9.2 Data

`class` `colicoords.data_models.`**`BinaryImage`**

Binary image data class.

Attributes

name [`str`] Name identifying the data element.

metadata [`dict`] Optional dict for metadata, load/save not implemented.

`orientation`

`float`: The main image axis orientation in degrees

`class` `colicoords.data_models.`**`BrightFieldImage`**

Brightfield image data class.

Attributes

name [`str`] Name identifying the data element.

metadata [`dict`] Optional dict for metadata, load/save not implemented.

`orientation`

`float`: The main image axis orientation in degrees

`class` `colicoords.data_models.`**`CellListData`**(*cell_list*)

Data class for CellList with common attributes for all cells. Individual data elements are accessed per cell.

Parameters

cell_list [`list` or `numpy.ndarray`] List of array of *`Cell`* objects.

Attributes

`dclasses` `list`: List of all data classes in the `Data` objects of the cells, if all are equal, else *None*.

`names` `list`: List of all data names in the `Data` objects of the cells, if all are equal, else *None*.

`shape` `tuple`: Tuple of cell's data element's shape if they are all equal, else *None*

dclasses
`list`: List of all data classes in the `Data` objects of the cells, if all are equal, else *None*.

names
`list`: List of all data names in the `Data` objects of the cells, if all are equal, else *None*.

shape
`tuple`: Tuple of cell's data element's shape if they are all equal, else *None*

class `colicoords.data_models.`**Data**
Main data class holding data from different input channels.

The data class is designed to combine and organize all different channels (brightfield, binary, fluorescence, storm) into one object. The class provides basic functionality such as rotation and slicing.

Data elements can be accessed from *data_dict* or by attribute '<class>_<name>', where class can be either 'flu', 'storm'. Binary and brightfield can bre accessed as properties.

Attributes

data_dict [`dict`] Dictionary with all data elements by their name.

flu_dict [`dict`] Subset of *data_dict* with all Fluorescence data elements.

storm_dict [`dict`] Subset of *data_dict* with all STORM data elements.

Methods

add_data(self, data, dclass[, name, metadata])	Add data to form a new data element.
copy(self)	Copy the data object.
prune(self, data_name)	Removes localizations from the STORM-dataset with name *data_name* which lie outside of the associated image.
rotate(self, theta)	Rotate all data elements and return a new `Data` object with rotated data elements.

next	

add_data (*self*, *data*, *dclass*, *name=None*, *metadata=None*)
Add data to form a new data element.

Parameters

data [array_like] Input data. Either np.ndarray with ndim 2 or 3 (images / movies) or numpy structured array for STORM data.

dclass [`str`] Data class. Must be either 'binary', 'brightfield', 'fluorescence' or 'storm'.

name [`str`, optional] The name to identify the data element. Default is equal to the data class.

metadata [`dict`] Associated metadata (load/save metadata currently not supported)

bf_img
`ndarray`: Returns the brightfield image if present, else `None`

binary_img
`ndarray`: Returns the binary image if present, else `None`

copy (*self*)
Copy the data object.

Returns

data [*Data*] Copied data object.

dclasses
`list`: List of all data classes in the `Data` object.

names
`list`: List of all data names in the `Data` object.

prune (*self*, *data_name*)
Removes localizations from the STORM-dataset with name *data_name* which lie outside of the associated image.

Parameters

data_name [`str`] Name of the data element to prune.

Returns

None

rotate (*self*, *theta*)
Rotate all data elements and return a new `Data` object with rotated data elements.

Parameters

theta [`float`] Rotation angle in degrees.

Returns

data [*colicoords.data_models.Data*] Rotated `Data`

class colicoords.data_models.**FluorescenceImage**
Fluorescence image data class.

Attributes

name [`str`] Name identifying the data element.

metadata [`dict`] Optional dict for metadata, load/save not implemented.

orientation
`float`: The main image axis orientation in degrees

class colicoords.data_models.**MetaData** *() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)*

class colicoords.data_models.**STORMTable**
STORM table data class.

Attributes

name [`str`] Name identifying the data element.

metadata [`dict`] Optional dict for metadata, load/save not implemented.

9.3 Plot

class colicoords.plot.**CellListPlot**(*cell_list*)

Object for plotting single-cell derived data

Parameters

cell_list [*CellList*] CellList object with Cell objects to plot.

Methods

figure()	Calls matplotlib.pyplot.figure()
hist_intensity(self[, mask, data_name, ax])	Histogram all cell's mean fluorescence intensity.
hist_l_storm(self[, data_name, ax])	Makes a histogram of the longitudinal distribution of localizations.
hist_phi_storm(self[, ax, data_name])	Makes a histogram of the angular distribution of localizations at the poles.
hist_property(self[, prop, ax])	Plot a histogram of a given geometrical property.
hist_r_storm(self[, data_name, ax, norm_x, . . .])	Makes a histogram of the radial distribution of localizations.
plot_kymograph(self[, mode, data_name, ax, . . .])	Plot a kymograph of a chosen axis distribution for a given data element.
plot_l_class(self[, data_name, ax, yerr])	Plots a bar chart of how many foci are in a given STORM data set in classes depending on x-position.
plot_l_dist(self[, ax, data_name, r_max, . . .])	Plots the longitudinal distribution of a given data element.
plot_phi_dist(self[, ax, data_name, r_max, . . .])	Plots the angular distribution of a given data element for all cells.
plot_r_dist(self[, ax, data_name, norm_x, . . .])	Plots the radial distribution of a given data element.
savefig(*args, **kwargs)	Calls matplotlib.pyplot.savefig(*args, **kwargs)
show()	Calls matplotlib.pyplot.show()

get_r_dist	

static figure()

Calls matplotlib.pyplot.figure()

hist_intensity(*self*, *mask='binary'*, *data_name=''*, *ax=None*, ***kwargs*)

Histogram all cell's mean fluorescence intensity. Intensities values are calculated by calling `Cell.get_intensity()`

Parameters

mask [str] Either 'binary' or 'coords' to specify the source of the mask used 'binary' uses the binary images as mask, 'coords' uses reconstructed binary from coordinate system.

data_name [str] The name of the image data element to get the intensity values from.

ax [matplotlib.axes.Axes, optinal] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to `ax.hist()`.

Returns

tuple [tuple] Return value is a tuple with *n*, *bins*, *patches* as returned by `hist()`.

hist_l_storm(*self*, *data_name=”*, *ax=None*, ***kwargs*)

Makes a histogram of the longitudinal distribution of localizations.

All cells are normalized by rescaling the longitudinal coordinates by the lenght of the cells. Polar regions are normalized by rescaling with the mean of the length of all cells to ensure uniform scaling of polar regions.

Parameters

data_name [`str`, optional] Name of the STORM data element to histogram. If omitted, the first STORM element is used.

ax [`matplotlib.axes.Axes`] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to `ax.hist()`

Returns

n [`ndarray`] The values of the histogram bins as produced by `hist()`

bins [`ndarray`] The edges of the bins.

patches [`list`] Silent list of individual patches used to create the histogram.

hist_phi_storm(*self*, *ax=None*, *data_name=”*, ***kwargs*)

Makes a histogram of the angular distribution of localizations at the poles.

Parameters

data_name [`str`, optional] Name of the STORM data element to histogram. If omitted, the first STORM element is used.

ax [`matplotlib.axes.Axes`] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to `ax.hist()`

Returns

n [`ndarray`] The values of the histogram bins as produced by `hist()`.

bins [`ndarray`] The edges of the bins.

patches [`list`] Silent list of individual patches used to create the histogram.

hist_property(*self*, *prop=’length’*, *ax=None*, ***kwargs*)

Plot a histogram of a given geometrical property.

Parameters

prop [`str`] Property to histogram. This can be one of ‘length’, radius, ‘circumference’, ‘area’, ‘surface’ or ‘volume’.

ax [`Axes`, optional] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to ax.hist().

Returns

tuple [`tuple`] Return value is a tuple with *n*, *bins*, *patches* as returned by `hist()`.

hist_r_storm(*self*, *data_name=”*, *ax=None*, *norm_x=True*, *limit_l=None*, ***kwargs*)

Makes a histogram of the radial distribution of localizations.

Parameters

data_name [`str`, optional] Name of the STORM data element to histogram. If omitted, the first STORM element is used.

ax [matplotlib.axes.Axes] Matplotlib axes to use for plotting.

norm_x [bool] If *True* all radial distances are normalized by dividing by the radius of the individual cells.

limit_l [str] If *None*, all datapoints are used. This can be limited by providing the value *full* (omit poles only), 'poles' (include only poles), or a float value between 0 and 1 which will limit the data points by longitudinal coordinate around the midpoint of the cell.

****kwargs** Additional kwargs passed to `ax.hist()`

Returns

n [ndarray] The values of the histogram bins as produced by `hist()`

bins [ndarray] The edges of the bins.

patches [list] Silent list of individual patches used to create the histogram.

plot_kymograph(*self*, *mode='r'*, *data_name="*, *ax=None*, *time_factor=1*, *time_unit='frames'*, *dist_kwargs=None*, *norm_y=True*, *aspect=1*, ***kwargs*)
Plot a kymograph of a chosen axis distribution for a given data element.

Each cell in the the `CellList` represents one point in time, where the first time point is the first cell in the list.

Parameters

ax [matplotlib.axes.Axes, optional] Matplotlib axes to use for plotting.

mode [str] Axis of distribution to plot. Options are 'r', 'l' or 'a'. Currently only 'r' is implemented.

data_name [str] Name of the data element to plot. Must be a 3D array

time_factor [float] Time factor per frame.

time_unit [str] Time unit.

dist_kwargs [dict] Additional kwargs passed to the function getting the distribution.

norm_y [bool] If *True* the output kymograph is normalized frame-wise.

aspect [float] Aspect ratio of output kymograph image.

****kwargs** Additional keyword arguments passed to ax.imshow()

Returns

image [matplotlib.image.AxesImage] Matplotlib image artist object

plot_l_class(*self*, *data_name="*, *ax=None*, *yerr='std'*, ***kwargs*)
Plots a bar chart of how many foci are in a given STORM data set in classes depending on x-position.

Parameters

data_name [str] Name of the data element to plot. Must have the data class 'storm'.

ax [matplotlib.axes.Axes] Matplotlib axes to use for plotting.

yerr [str] How to calculated error bars. Can be 'std' or 'sem' for standard deviation or standard error of the mean, respectively.

****kwargs** Optional kwargs passed to ax.bar().

Returns

container [BarContainer] Container with all the bars.

plot_l_dist(*self*, *ax=None*, *data_name="*, *r_max=None*, *norm_y=False*, *zero=False*, *storm_weight=False*, *band_func=<function std at 0x7fb3544dd1e0>*, *method='gauss'*, *dist_kwargs=None*, ***kwargs*)

Plots the longitudinal distribution of a given data element.

The data is normalized along the long axis to allow the combining of multiple cells with different lengths.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

data_name [`str`] Name of the data element to use.

r_max [`float`] Datapoints within *r_max* from the cell midline are included. If *None* the value from the cell's coordinate system will be used.

norm_y [`bool`] If *True* the output data will be normalized in the y (intensity).

zero [`bool`] If *True* the output data will be zeroed.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

band_func [`callable`] Callable to determine the fill area around the graph. Default is standard deviation.

method [`str`, either 'gauss' or 'box'] Method of averaging datapoints to calculate the final distribution curve.

dist_kwargs [`dict`] Additional kwargs to be passed to *`l_dist()`*

****kwargs** Optional kwargs passed to ax.plot()

Returns

line **`Line2D`** Matplotlib line artist object

plot_phi_dist(*self*, *ax=None*, *data_name="*, *r_max=None*, *r_min=0*, *storm_weight=False*, *band_func=<function std at 0x7fb3544dd1e0>*, *method='gauss'*, *dist_kwargs=None*, ***kwargs*)

Plots the angular distribution of a given data element for all cells.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

data_name [`str`] Name of the data element to use.

r_max: [`float`] Datapoints within r_max from the cell midline are included. If *None* the value from the cell's coordinate system will be used.

r_min: [`float`] Datapoints outside of r_min from the cell midline are included.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

band_func [`callable`] Callable to determine the fill area around the graph. Default is standard deviation.

method [`str`:] Method of averaging datapoints to calculate the final distribution curve.

dist_kwargs [`dict`] Additional kwargs to be passed to *`phi_dist()`*

Returns

lines [`tuple`] Tuple with Matplotlib line artist objects.

plot_r_dist(*self, ax=None, data_name='', norm_x=False, norm_y=False, zero=False, storm_weight=False, limit_l=None, method='gauss', band_func=<function std at 0x7fb3544dd1e0>, **kwargs*)

Plots the radial distribution of a given data element.

Parameters

ax [`matplotlib.axes.Axes`] Optional matplotlib axes to use for plotting.

data_name [`str`] Name of the data element to use.

norm_x: [`bool`] If *True* the output distribution will be normalized along the length axis.

norm_y: [`bool`] If *True* the output data will be normalized in the y (intensity).

zero [`bool`] If *True* the output data will be zeroed.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

limit_l [`str`] If *None*, all datapoints are taking into account. This can be limited by providing the value *full* (omit poles only), 'poles' (include only poles), or a float value which will limit the data points with around the midline where xmid - xlim < x < xmid + xlim.method : `str`, either 'gauss' or 'box'

method [`str`, either 'gauss' or 'box'] Method of averaging datapoints to calculate the final distribution curve.

band_func [`callable`] Callable to determine the fill area around the graph. Default is standard deviation.

****kwargs** Optional kwargs passed to ax.plot().

Returns

`Line2D` Matplotlib line artist object

static savefig(**args, **kwargs*)

Calls matplotlib.pyplot.savefig(*args, **kwargs)

static show()

Calls matplotlib.pyplot.show()

class colicoords.plot.**CellPlot**(*cell_obj*)

Object for plotting single-cell derived data.

Parameters

cell_obj [*`Cell`*] Single-cell object to plot.

Attributes

cell_obj [`Cell`] Single-cell object to plot.

Methods

`figure`(*args, **kwargs)	Calls `matplotlib.pyplot.figure()`
`hist_l_storm`(self[, data_name, ax, norm_x])	Makes a histogram of the longitudinal distribution of localizations.
`hist_phi_storm`(self[, ax, data_name])	Makes a histogram of the angular distribution of localizations at the poles.

Continued on next page

Table 6 – continued from previous page

hist_r_storm(self[, data_name, ax, norm_x, . . .])	Makes a histogram of the radial distribution of localizations.
imshow(self, img[, ax])	Call to matplotlib's imshow.
plot_bin_fit_comparison(self[, ax])	Plot the cell's binary image together with the calculated binary image from the coordinate system.
plot_binary_img(self[, ax])	Plot the cell's binary image.
plot_kymograph(self[, ax, mode, data_name, . . .])	Plot a kymograph of a chosen axis distribution for a given data element.
plot_l_class(self[, ax, data_name])	Plots a bar chart of how many foci are in a given STORM data set in classes depending on x-position.
plot_l_dist(self[, ax, data_name, r_max, . . .])	Plots the longitudinal distribution of a given data element.
plot_midline(self[, ax])	Plot the cell's coordinate system midline.
plot_outline(self[, ax])	Plot the outline of the cell based on the current coordinate system.
plot_phi_dist(self[, ax, data_name, r_max, . . .])	Plots the angular distribution of a given data element.
plot_r_dist(self[, ax, data_name, norm_x, . . .])	Plots the radial distribution of a given data element.
plot_simulated_binary(self[, ax])	Plot the cell's binary image calculated from the coordinate system.
plot_storm(self[, ax, data_name, method, . . .])	Graphically represent STORM data.
savefig(*args, **kwargs)	Calls `matplotlib.pyplot.savefig()`
show(*args, **kwargs)	Calls `matplotlib.pyplot.show()`

get_r_dist	

static figure(**args, **kwargs*)
Calls `matplotlib.pyplot.figure()`

hist_l_storm(*self, data_name=", ax=None, norm_x=True, **kwargs*)
Makes a histogram of the longitudinal distribution of localizations.

Parameters

data_name [`str`, optional] Name of the STORM data element to histogram. If omitted, the first STORM element is used.

ax [`matplotlib.axes.Axes`] Matplotlib axes to use for plotting.

norm_x [`bool`] Normalizes the longitudinal distribution by dividing by the length of the cell.

****kwargs** Additional kwargs passed to *ax.hist()*

Returns

n [`ndarray`] The values of the histogram bins as produced by `hist()`

bins [`ndarray`] The edges of the bins.

patches [`list`] Silent list of individual patches used to create the histogram.

hist_phi_storm(*self, ax=None, data_name=", **kwargs*)
Makes a histogram of the angular distribution of localizations at the poles.

Parameters

data_name [`str`, optional] Name of the STORM data element to histogram. If omitted, the first STORM element is used.

ax [`matplotlib.axes.Axes`] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to *ax.hist()*

Returns

n [`ndarray`] The values of the histogram bins as produced by `hist()`

bins [`ndarray`] The edges of the bins.

patches [`list`] Silent list of individual patches used to create the histogram.

`hist_r_storm`(*self*, *data_name=”*, *ax=None*, *norm_x=True*, *limit_l=None*, ***kwargs*)
Makes a histogram of the radial distribution of localizations.

Parameters

data_name [`str`, optional] Name of the STORM data element to histogram. If omitted, the first STORM element is used.

ax [`matplotlib.axes.Axes`] Matplotlib axes to use for plotting.

norm_x [`bool`] If *True* all radial distances are normalized by dividing by the radius of the individual cells.

limit_l [`str`] If *None*, all datapoints are taking into account. This can be limited by providing the value *full* (omit poles only), ‘poles’ (include only poles), or a float value which will limit the data points with around the midline where xmid - xlim < x < xmid + xlim.method : `str`, either ‘gauss’ or ‘box’

****kwargs** Additional kwargs passed to *ax.hist()*

Returns

n [`ndarray`] The values of the histogram bins as produced by `hist()`

bins [`ndarray`] The edges of the bins.

patches [`list`] Silent list of individual patches used to create the histogram.

`imshow`(*self*, *img*, *ax=None*, ***kwargs*)
Call to matplotlib’s imshow.

Default *extent* keyword arguments is provided to assure proper overlay of pixel and carthesian coordinates.

Parameters

img [`str` or `ndarray`] Image to show. It can be either a data name of the image-type data element to plot or a 2D numpy ndarray.

ax [`matplotlib.axes.Axes`] Optional matplotlib axes to use for plotting.

****kwargs:** Additional kwargs passed to ax.imshow().

Returns

image [`matplotlib.image.AxesImage`] Matplotlib image artist object.

`plot_bin_fit_comparison`(*self*, *ax=None*, ***kwargs*)
Plot the cell’s binary image together with the calculated binary image from the coordinate system.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to ax.plot().

Returns

image [`AxesImage`] Matplotlib image artist object.

`plot_binary_img`(*self*, *ax=None*, ***kwargs*)
Plot the cell's binary image.

Equivalent to `CellPlot.imshow('binary')`.

Parameters

ax [`matplotlib.axes.Axes`, optional] Optional matplotlib axes to use for plotting

****kwargs** Additional kwargs passed to ax.imshow().

Returns

image [`AxesImage`] Matplotlib image artist object

`plot_kymograph`(*self*, *ax=None*, *mode='r'*, *data_name=''*, *time_factor=1*, *time_unit='frames'*, *dist_kwargs=None*, *norm_y=True*, *aspect=1*, ***kwargs*)
Plot a kymograph of a chosen axis distribution for a given data element.

The data element must be a 3D array (t, y, x) where the first axis is the time dimension.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

mode [`str`] Axis of distribution to plot. Options are 'r', 'l' or 'a'. Currently only 'r' is implemented.

data_name [`str`] Name of the data element to plot. Must be a 3D array

time_factor [`float`] Time factor per frame.

time_unit [`str`] Time unit.

dist_kwargs [`dict`] Additional kwargs passed to the function getting the distribution.

norm_y [`bool`] If *True* the output kymograph is normalized frame-wise.

aspect [`float`] Aspect ratio of output kymograph image.

****kwargs** Additional keyword arguments passed to ax.imshow()

Returns

image [`matplotlib.image.AxesImage`] Matplotlib image artist object

`plot_l_class`(*self*, *ax=None*, *data_name=''*, ***kwargs*)
Plots a bar chart of how many foci are in a given STORM data set in classes depending on x-position.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

data_name [`str`] Name of the data element to plot. Must have the data class 'storm'.

****kwargs** Optional kwargs passed to ax.bar().

Returns

container [`BarContainer`] Container with all the bars.

`plot_l_dist`(*self*, *ax=None*, *data_name=''*, *r_max=None*, *norm_x=False*, *norm_y=False*, *zero=False*, *storm_weight=False*, *method='gauss'*, *dist_kwargs=None*, ***kwargs*)
Plots the longitudinal distribution of a given data element.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

data_name [`str`] Name of the data element to use.

r_max: [`float`] Datapoints within r_max from the cell midline are included. If *None* the value from the cell's coordinate system will be used.

norm_x [`bool`] If *True* the output distribution will be normalized along the length axis.

norm_y: [`bool`] If *True* the output data will be normalized in the y (intensity).

zero [`bool`] If *True* the output data will be zeroed.

storm_weight [`bool`] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

method [`str`:] Method of averaging datapoints to calculate the final distribution curve.

dist_kwargs [`dict`] Additional kwargs to be passed to *`l_dist()`*

Returns

line [`Line2D`] Matplotlib line artist object.

`plot_midline`(*self*, *ax=None*, ***kwargs*)
Plot the cell's coordinate system midline.

Parameters

ax [`Axes`, optional] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to ax.plot().

Returns

line [`Line2D`] Matplotlib line artist object

`plot_outline`(*self*, *ax=None*, ***kwargs*)
Plot the outline of the cell based on the current coordinate system.

The outline consists of two semicircles and two offset lines to the central parabola.[R114d31aa08a4-1]_[R114d31aa08a4-2]_

Parameters

ax [`Axes`, optional] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to ax.plot().

Returns

line [`Line2D`] Matplotlib line artist object.

`plot_phi_dist`(*self*, *ax=None*, *data_name="*, *r_max=None*, *r_min=0*, *storm_weight=False*, *method='gauss'*, *dist_kwargs=None*, ***kwargs*)
Plots the angular distribution of a given data element.

Parameters

ax [`matplotlib.axes.Axes`, optional] Matplotlib axes to use for plotting.

data_name [`str`] Name of the data element to use.

r_max: [`float`] Datapoints within r_max from the cell midline are included. If *None* the value from the cell's coordinate system will be used.

r_min: [`float`] Datapoints outside of r_min from the cell midline are included.

storm_weight [bool] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

method [str:] Method of averaging datapoints to calculate the final distribution curve.

dist_kwargs [dict] Additional kwargs to be passed to *phi_dist()*

Returns

lines [tuple] Tuple with Matplotlib line artist objects.

plot_r_dist(*self*, *ax=None*, *data_name="*, *norm_x=False*, *norm_y=False*, *zero=False*, *storm_weight=False*, *limit_l=None*, *method='gauss'*, *dist_kwargs=None*, ***kwargs*)

Plots the radial distribution of a given data element.

Parameters

ax [Axes, optional] Matplotlib axes to use for plotting.

data_name [str] Name of the data element to use.

norm_x [bool] If *True* the output distribution will be normalized along the length axis.

norm_y: [bool] If *True* the output data will be normalized in the y (intensity).

zero [bool] If *True* the output data will be zeroed.

storm_weight [bool] If *True* the datapoints of the specified STORM-type data will be weighted by their intensity.

limit_l [str] If *None*, all datapoints are used. This can be limited by providing the value *full* (omit poles only), 'poles' (include only poles), or a float value between 0 and 1 which will limit the data points by longitudinal coordinate around the midpoint of the cell.

method [str] Method of averaging datapoints to calculate the final distribution curve.

dist_kwargs [dict] Additional kwargs to be passed to *colicoords.cell.Cell.r_dist()*

****kwargs** Optional kwargs passed to `ax.plot()`.

Returns

line [Line2D] Matplotlib line artist object

plot_simulated_binary(*self*, *ax=None*, ***kwargs*)

Plot the cell's binary image calculated from the coordinate system.

Parameters

ax [matplotlib.axes.Axes, optional.] Matplotlib axes to use for plotting.

****kwargs** Additional kwargs passed to ax.imshow().

Returns

image [AxesImage] Matplotlib image artist object

plot_storm(*self*, *ax=None*, *data_name="*, *method='plot'*, *upscale=5*, *alpha_cutoff=None*, *storm_weight=False*, *sigma=0.25*, ***kwargs*)

Graphically represent STORM data.

Parameters

ax [Axes] Optional matplotlib axes to use for plotting.

data_name [str] Name of the data element to plot. Must be of data class 'storm'.

method [`str`] Method of visualization. Options are 'plot', 'hist', or 'gauss' just plotting points, histogram plot or gaussian kernel plot.

upscale [`int`] Upscale factor for the output image. Number of pixels is increased w.r.t. data.shape with a factor upscale**2

alpha_cutoff [`float`] Values (normalized) below *alpha_cutoff* are transparent, where the alpha is linearly scaled between 0 and *alpha_cutoff*

storm_weight [`bool`] If *True* the STORM data points are weighted by their intensity.

sigma [`float` or `string` or `ndarray`] Only applies for method 'gauss'. The value is the sigma which describes the gaussian kernel. If *sigma* is a scalar, the same sigma value is used for all data points. If *sigma* is a string it is interpreted as the name of the field in the STORM array to use. Otherwise, sigma can be an array with equal length to the number of datapoints.

****kwargs** Additional kwargs passed to ax.plot() or ax.imshow().

Returns

artist [`AxesImage` or `Line2D`] Matplotlib artist object.

static savefig(**args*, ***kwargs*)
Calls `matplotlib.pyplot.savefig()`

static show(**args*, ***kwargs*)
Calls `matplotlib.pyplot.show()`

9.4 Optimizers

class colicoords.fitting.**CellBinaryFunction**(*cell_obj*, *data_name*)
Binary data element objective function.

Calling this object with coordinate system parameters returns a binary image by thresholding the radial distance image with the radius of the cell.

Methods

`__call__`(self, **parameters)	Call self as a function.

class colicoords.fitting.**CellImageFunction**(*cell_obj*, *data_name*)
Image element objective function.

Calling this object with coordinate system parameters returns a reconstructed image of the target data element.

Methods

`__call__`(self, **parameters)	Call self as a function.

class colicoords.fitting.**CellMinimizeFunctionBase**(*cell_obj*, *data_name*)
Base class for Objective objects used by `CellFit` to optimize the coordinate system.

The base class takes a *`Cell`* object and the name of target data element to perform optimization on. Subclasses of `CellMinimizeFunctionBase` must implement the *__call__* builtin, which takes the coordinate system's parameters as keyword arguments.

Note that this is not an objective function to be minimized, but instead the return value is compared with the specified data element or specific target data to give the chi-squared.

Parameters

cell_obj [`Cell`] Cell object to optimize.

data_name [`str`] Target data element name.

class `colicoords.fitting.`**`CellSTORMMembraneFunction`**(**args*, ***kwargs*)
STORM membrane objective function.

Calling this object with coordinate system parameters returns a reconstructed image of the target data element.

Attributes

`target_data` Dependent (target) data for coordinate optimization based on STORM membrane markers

Methods

`__call__`(self, **parameters)	Call self as a function.

`target_data`
Dependent (target) data for coordinate optimization based on STORM membrane markers

class `colicoords.fitting.`**`DepCellFit`**(*cell_obj*, *data_name='binary'*, *objective=None*, *minimizer=<class 'symfit.core.minimizers.Powell'>*, ***kwargs*)

Attributes

data_elem

Methods

execute_stepwise	
fit_parameters	
renew_fit	

class `colicoords.fitting.`**`LinearModelFit`**(*model*, **args*, ***kwargs*)
Fitting of a model with linear parameters where the linear parameters are not fitted by `symfit` but instead solved as a system of linear equations.

Parameters

model [:]

Methods

`execute`(self, **kwargs)	Execute the fit.

execute (*self*, ***kwargs*)
Execute the fit.

Parameters **minimize_options** – keyword arguments to be passed to the specified minimizer.

Returns FitResults instance

class colicoords.fitting.**RadialData** (*cell_obj*, *length*)
Class mimicking a numpy ndarray used as dependent data for fitting STORM-membrane data.

The apparent value of this object is an array with length *length* and whoes values are all equal to the radius of *cell_obj*.

Parameters

cell_obj [*Cell*] Cell object whos radius gives this array's values.

length [int] Length of the array.

Attributes

shape

value

colicoords.fitting.**solve_linear_system** (*y_list*, *data*)
Solve system of linear eqns a1*y1 + a2*y2 == data but then also vector edition of that

9.5 FileIO

colicoords.fileIO.**load** (*file_path*)
Load Cell or CellList from disk.

Parameters

file_path [str] Source file path.

Returns

cell [*Cell* or *CellList*] Loaded Cell or CellList

colicoords.fileIO.**load_thunderstorm** (*file_path*, *pixelsize=None*)
Load a .csv file from THUNDERSTORM output.

Parameters

file_path [str] Target file path to THUNDERSTORM file.

pixelsize [float, optional] pixelsize in the THUNDERSTORM file to convert units to pixels. If not specified the default value in config is used.

colicoords.fileIO.**save** (*file_path*, *cell_obj*)
Save ColiCoords Cell objects to disk as hdf5-files.

Parameters

file_path [str] Target file path.

cell_obj [*Cell* or :class:'~colicoords.cell.CellList] Cell or CellList object to save to disk.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Bibliography

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: convolutional networks for biomedical image segmentation. 2015. arXiv:arXiv:1505.04597.

[1] https://en.wikipedia.org/wiki/Cubic_function#General_formula ..

[1] https://en.wikipedia.org/wiki/Cubic_function#Trigonometric_solution_for_three_real_roots ..

[1] T. W. Sederberg. “Computer Aided Geometric Design”. Computer Aided Geometric Design Course Notes. January 10, 2012

[2] Rida T.Faroukia, Thomas W. Sederberg, Analysis of the offset to a parabola, Computer Aided Geometric Design vol 12, issue 6, 1995

Python Module Index

Index