particle-tracker-one-d

Release 0.7.0

Jan 06, 2021

Contents:

This software was developed to track particles in Kymographs, eg. intensity graphs. It is based on the article Sbalzarini, Ivo F., and Petros Koumoutsakos., Journal of structural biology 151.2 (2005): 182-195 to track feature points but with the exception, that in this implementation the only noise filtering is the boxcar averaging and the frames are one dimensional. The feature points that are found are linked together by minimising a cost function and results in one or more trajectories.

From the trajectories one can easily plot the velocity auto correlation function and calculate the diffusion coefficient from either the mean squared displacement function or by a covariance based estimator.

CHAPTER 1

Installation

To install the particle tracker simply use pip

```
pip install particle-tracker-one-d
```

or download the repo from git

```
git clone https://gitlab.com/langhammerlab/1d-particle-tracking.git
```

CHAPTER 2

Quick start

2.1 Particle tracking

```
import matplotlib.pyplot as plt
import numpy as np
from particle_tracker_one_d import ParticleTracker

# Import the frames and the time data
frames = np.load('examples/frames.npy')
time = np.load('examples/time.npy')

# Normalise the intensity
frames_normalised = ParticleTracker.normalise_intensity(frames)

# Create a particle tracker instance
pt = ParticleTracker(frames=frames_normalised, time=time)

# Set the properies of the particle tracker
pt.boxcar_width = 10
pt.change_cost_coefficients(1,1,0,6)
pt.integration_radius_of_intensity_peaks = 10
pt.particle_detection_threshold = 0.6
pt.maximum_number_of_frames_a_particle_can_disappear_and_still_be_linked_to_other_
↪particles = 5
pt.maximum_distance_a_particle_can_travel_between_frames = 40


# Create a figure
plt.figure(figsize=(8,20))
ax=plt.axes()

# Plot the kymograph
pt.plot_all_frames(ax=ax, aspect='auto')
```

(continues on next page)

(continued from previous page)

```
# Plot all the trajectories
for t in pt.trajectories:
    t.plot_trajectory(x='position',y='frame_index',ax=ax, marker='o')
```

2.2 Trajectory analysis

```
# Select one of the trajectories
trajectory = pt.trajectories[0]

# Set the pixel width
trajectory.pixel_width = 5e-4

# Create a figure
plt.figure(figsize=(8,8))
ax=plt.axes()

# Plot the velocity auto correlation function to make sure particle steps are␣
↪uncorrelated
trajectory.plot_velocity_auto_correlation(ax=ax)

# Calculate the diffusion coefficient using a covariance based estimator
print(trajectory.calculate_diffusion_coefficient_using_covariance_based_estimator())
```

2.3 Shortest path finder

```
import matplotlib.pyplot as plt
import numpy as np
from particle_tracker_one_d import ShortestPathFinder

# Import the frames and the time data
frames = np.load('examples/frames.npy')
time = np.load('examples/time.npy')

# Normalise the intensity
frames_normalised = ParticleTracker.normalise_intensity(frames)

# Create a shortest path finder instance
spf = ShortestPathFinder(frames=frames_normalised, time=time)

# Set the properies of the path finder
spf.boxcar_width = 5
spf.integration_radius_of_intensity_peaks = 20

# Set start point
spf.start_point = (0,90)

# Set end point
spf.end_point = (232,2)

fig = plt.figure(figsize=(5,15))
ax = plt.axes()
```

(continues on next page)

(continued from previous page)

```
# Plot the frames and the trajectory
spf.plot_all_frames(ax=ax, aspect='auto')
spf.trajectory.plot_trajectory(x='position', y='frame_index',ax=ax, marker='o')

ax.set_ylim([232, 90])
fig.tight_layout()
```

CHAPTER 3

How does it work?

I will here give a basic explanation of how the algorithms work. For deeper understanding of the particle tracking algorithm, I refer you to

1. Sbalzarini, Ivo F., and Petros Koumoutsakos., Journal of structural biology 151.2 (2005): 182-195.

and for the shortest path finder I refer you to 2. Dijkstra's algorithm

3.1 Definitions

A particle observation $p = [t, \hat{x}_p]$ where t is the frame index and $\hat{x}_p$ is the estimated position comes with associated intensity moments defined as the 0th order moment

$$m_0(p) = \sum_{i^2 \leq w^2} I^t(\hat{x}_p + i)$$

and the 2nd order moment

$$m_2(p) = \sum_{i^2 \leq w^2} i^2 I^t(\hat{x}_p + i)$$

where I^t is the intensity of frame t and w is the integration radius.

3.2 Particle Tracker

The particle tracker finds arbitrary number of trajectories in a kymograph. This is done by linking particle detections together by minimising a cost describing the chance of detections being the same particle in two different frames.

3.2.1 Initialisation

To create an instance of the particle tracker one has to provide the frames and the corresponding times.

```
from particle_tracker_one_d import ParticleTracker

# Import frames and time data
frames = np.load('examples/frames.npy')
time = np.load('examples/time.npy')

# Create a particle tracker instance
pt = ParticleTracker(frames=frames, time=times)
```

Preferably the frames should be normalised according to $I_{normalised} = (I - I_{min})/(I_{max} - I_{min})$ but is not a requirement.

3.2.2 Finding particle positions

To detect possible particles an intensity detection threshold is set by the attribute `pt.particle_detection_threshold`. Local maximas over this threshold are considered as initial possible particle detections. If however, two maximas are found within a distance of `2 * pt.integration_radius_of_intensity_peaks` pixels, the lowest maxima of the two points is discarded. Each position is then refined using a centroid estimation around the local maxima, using the same integration radius.

3.2.3 Linking procedure

The linking procedure is then performed as described in[1] where two frames are analysed at a time and a cost for linking a particle in frame t with either a particle or a dummy particle in frame $t + r$, is calculated for all combination of particles in both frames. A link matrix is then optimized to find the set of links between particles yielding the lowest total cost, still respecting the topography requirement, that each particle in frame t can only be linked to one particle in frame $t + r$. The maximum r allowed is set by the attribute `pt.maximum_number_of_frames_a_particle_can_disappear_and_still_be_linked_to_other_particles`. The cost function used for the associations is

$$\phi(p_1, p_2) = a \cdot (\hat{x}_{p_1} - \hat{x}_{p_2})^2 + b \cdot (m_0(p_1) - m_0(p_2))^2 + c \cdot (m_2(p_1) - m_2(p_2))^2 + d \cdot (r - 1)^2$$

where the $d \cdot (r-1)^2$ term is added to promote the linking of particle detections closer in time. To change the coefficients a, b, c, d one can call the function `pt.change_cost_coefficients(a=1,b=1,c=1,d=1)`. In case of many particles there is a limit to the maximum number of pixels a particle can move between two consecutive frames set by the instance attribute `pt.maximum_distance_a_particle_can_travel_between_frames`. If this limit is exceeded the cost is set to infinity and any linking is effectively blocked. When all link matrices are optimised the trajectories are built. If a particle is linked to the dummy particle, the lowest cost association to a non-dummy particle in the following frames is linked. If no such association does exist, the trajectory is ended.

3.3 Shortest Path Finder

The intention of the Shortest Path Finder is to refine trajectories that are missing positions in several frames. This is done by finding a path connecting a start, an end and intermediate static positions. It is based on the well known Dijkstra's algorithm.

3.3.1 Initialisation

To create an instance of the shortest path finder one has to provide the frames and the corresponding times.

```
from particle_tracker_one_d import ShortestPathFinder

# Import frames and time data
frames = np.load('examples/frames.npy')
time = np.load('examples/time.npy')

# Create a particle tracker instance
spf = ShortestPathFinder(frames=frames, time=times)
```

Preferably the frames should be normalised according to $I_{normalised} = (I - I_{min})/(I_{max} - I_{min})$ but is not a requirement.

3.3.2 Set the static points

The shortest path finder needs a start and an en point. These are set by

```
spf.start_point = (start_frame, start_position)
spf.start_point = (end_frame, end_position)
```

both the values should be integers and represent the indices of the start and end point in the frames. There is then a possibility to add more points that the trajectory is forced to go through. This is done by the attribute `static_points`

```
spf.static_points =[(frame_1, position_1), (frame_2, position_2), ..., (frame_n,␣
↪position_n)]
```

3.3.3 Finding particle positions

Possible particles is then found by looking for intensity maximas over the intensity detection threshold that is set by the attribute `spf.particle_detection_threshold` in the frames between the start and end point, skipping the frames with the static points. If however, two maximas are found within a double distance of the attribute `spf.integration_radius_of_intensity_peaks`, the lowest maxima of the two points is discarded. Each position is then refined using a centroid estimation around the local maxima, using the same integration radius. This also includes the start, the end and the static points.

3.3.4 Finding the shortest path

The algorithm then finds the shortest path defined by the cost/distance between particles

$$\phi(p_1, p_2) = a \cdot (\hat{x}_{p_1} - \hat{x}_{p_2})^2 + b \cdot (m_0(p_1) - m_0(p_2))^2 + c \cdot (m_2(p_1) - m_2(p_2))^2$$

where the coefficients a, b, c can be changed using the function `spf.change_cost_coefficients(a=1, b=1, c=1)`. The algorithm then works as follows:

1. Store all positions in arrays $\{P^t\}_{t=t_0}^{t_n}$. Where t_0 and t_n is the start and end indices of the frames.
2. Start at $t = t_0$ and calculate the cost between the position at t_0 and the positions at t_1. Store these costs in a matrix $C^1 = c_{ij} = \phi(p_i, p_j)$. These now describe the cost to go to each position in frame t_1.
3. Continue calculate for all n the cost between particles in frame t_n to particles in frame t_{n+1} and add the lowest cost from the i:th column in the previous cost matrix
$$C^n = c_{ij} = \phi(p_i, p_j) + min_{i'(C_{i'in-1})}$$
4. Find the lowest value in C^n. Wich is the lowest possible cost path from the first position to the final, passing through all the positions in the initial sparse trajectory.

5. Build the trajectory by going backwards in the cost matrices following the lowest cost path.

3.4 Trajectory

The trajecory class is made for analysing the trajectories found by the particle tracker or the shoertest path finder. It has some methods attached to it to make this easier and faster. It is possible to instanciate the trajectory class but the intentional way is that it is delivered to the user as already instanciated objects under the attribute `pt.trajectories` and `spf.trajectory`. If you want to make your own instance, it is preferably done like this

```
t = Trajectory()
t.particle_positions = positions
```

positions should be a `numpy` structured array with field names 'time', 'frame_index' and 'position'.

3.4.1 Velocity auto correlation

A common way to check that a trajectory represents free diffusion is to plot the velocity auto correlation and check if velocitites are correlated. This can be done with the method `t.plot_velocity_auto_correlation()`.

3.4.2 Calculate diffusion coefficients

The software comes with two methods of determining the diffusion coefficient, either by fitting a straight line to the mean squared displacement function `t.calculate_diffusion_coefficient_from_mean_square_displacement_function()` or by a covariance based estimator `t.calculate_diffusion_coefficient_using_covariance_based_estimator()`. For more information about determining diffusion coefficients, I refer you to

Optimal estimation of diffusion coefficients from single-particle trajectories.

CHAPTER 4

Particle Tracker

class particle_tracker_one_d.**ParticleTracker**(*frames*, *time*, *automatic_update=True*)

Dynamic Particle tracker object which finds trajectories in the frames. Trajectories are automatically updated when properties are changed.

Parameters

frames: np.array The frames in which trajectories are to be found. The shape of the np.array should be (nFrames,xPixels). The intensity of the frames should be normalised according to $I_n = (I - I_{min})/(I_{max} - I_{min})$, where I is the intensity of the frames, I_{min}, I_{max} are the global intensity minima and maxima of the frames.

time: np.array The corresponding time of each frame.

Attributes

frames np.array:

time np.array:

boxcar_width int:

integration_radius_of_intensity_peaks int:

particle_detection_threshold float:

maximum_number_of_frames_a_particle_can_disappear_and_still_be_linked_to_other_p int:

maximum_distance_a_particle_can_travel_between_frames int:

particle_positions list:

Methods

change_cost_coefficients([a, b, c, d])	Change the coefficients of the cost function $c(p_1, p_2) = a \cdot (x_{p_1} - x_{p_2})^2 + b \cdot (m_0(p_1) - m_0(p_2))^2 + b \cdot (m_2(p_1) - m_2(p_2))^2) + d \cdot (t_{p_1} - t_{p_2})^2$
get_frame_at_time(time)	time: float
normalise_intensity(frames)	frames: np.array
plot_all_frames([ax])	ax: matplotlib axes instance
plot_all_particles([ax])	ax: matplotlib axes instance
plot_frame(frame_index[, ax])	frame_index: index
plot_frame_at_time(time[, ax])	time: float
plot_moments([ax])	ax: matplotlib axes instance

boxcar_width

int: Number of values used in the boxcar averaging of the frames.

change_cost_coefficients (*a=1*, *b=1*, *c=1*, *d=1*)

Change the coefficients of the cost function $c(p_1, p_2) = a \cdot (x_{p_1} - x_{p_2})^2 + b \cdot (m_0(p_1) - m_0(p_2))^2 + b \cdot (m_2(p_1) - m_2(p_2))^2) + d \cdot (t_{p_1} - t_{p_2})^2$

a: float

b: float

c: float

d: float

frames

np.array: The frames which the particle tracker tries to find trajectories in. If the property boxcar_width!=0 it will return the smoothed frames.

get_frame_at_time (*time*)

time: float Time of the frame which you want to get.

Returns

np.array Returns the frame which corresponds to the input time.

integration_radius_of_intensity_peaks

int: Number of pixels used when integrating the intensity peaks. No particles closer than twice this value will be found. If two peaks are found within twice this value, the one with highest intensity moment will be kept.

maximum_distance_a_particle_can_travel_between_frames

int: Max number of pixels a particle can travel between two consecutive frames.

maximum_number_of_frames_a_particle_can_disappear_and_still_be_linked_to_other_particl

int: Number of frames a particle can be invisible and still be linked in a trajectory.

static normalise_intensity (*frames*)

frames: np.array Normalises the intensity of the frames according to $I_n = (I - I_{min})/(I_{max} - I_{min})$, where I is the intensity of the frames, I_{min}, I_{max} are the global intensity minima and maxima of the frames.

Returns

np.array The normalised intensity.

particle_detection_threshold

float: Defines the threshold value for finding intensity peaks. Local maximas below this threshold will not be considered as particles. Should be a value between 0 and 1.

particle_positions

list: List with numpy arrays containing all particle positions.

plot_all_frames (*ax=None*, ***kwargs*)

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.imshow method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

plot_all_particles (*ax=None*, ***kwargs*)

ax: matplotlib axes instance The axes which you want the particle detections to be plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.scatter method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

plot_frame (*frame_index*, *ax=None*, ***kwargs*)

frame_index: index The index of the frame you want to plot.

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.plot method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

plot_frame_at_time (*time*, *ax=None*, ***kwargs*)

time: float The time of the frame you want to plot.

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.plot method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

plot_moments(*ax=None*, ***kwargs*)

ax: matplotlib axes instance The axes which you want the moments to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.scatter method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

time

np.array: The time for each frame.

trajectories

list: Returns a list with all found trajectories of type class: Trajectory.

CHAPTER 5

Shortest Path Finder

class particle_tracker_one_d.**ShortestPathFinder**(*frames*, *time*, *automatic_update=True*)

Class for finding shortest path between points in a set of frames.

Parameters

frames: np.array The frames in which trajectories are to be found. The shape of the np.array should be (nFrames,xPixels). The intensity of the frames should be normalised according to $I_n = (I - I_{min})/(I_{max} - I_{min})$, where I is the intensity of the frames, I_{min}, I_{max} are the global intensity minima and maxima of the frames.

time: np.array The corresponding time of each frame.

automatic_update: bool Choose if the class should update itself when changing properties.

Attributes

frames np.array:

time

boxcar_width int:

integration_radius_of_intensity_peaks int:

particle_detection_threshold float:

particle_positions np.array:

static_points

start_point tuple:

end_point tuple:

Methods

change_cost_coefficients([a, b, c])	Change the coefficients of the cost function $c(p_1, p_2) = a \cdot (x_{p_1} - x_{p_2})^2 + b \cdot (m_0(p_1) - m_0(p_2))^2 + b \cdot (m_2(p_1) - m_2(p_2))^2)$
plot_all_frames([ax])	ax: matplotlib axes instance
plot_moments([ax])	ax: matplotlib axes instance

boxcar_width

int: Number of values used in the boxcar averaging of the frames.

change_cost_coefficients (*a=1*, *b=1*, *c=1*)

Change the coefficients of the cost function $c(p_1, p_2) = a \cdot (x_{p_1} - x_{p_2})^2 + b \cdot (m_0(p_1) - m_0(p_2))^2 + b \cdot (m_2(p_1) - m_2(p_2))^2)$

a: float

b: float

c: float

end_point

tuple: (frame_index, position_index), The end point of the path you want to find.

frames

np.array: The frames which the particle tracker tries to find trajectories in. If the property boxcar_width!=0 it will return the smoothed frames.

integration_radius_of_intensity_peaks

int: Number of pixels used when integrating the intensity peaks. No particles closer than twice this value will be found. If two peaks are found within twice this value, the one with highest intensity moment will be kept.

particle_detection_threshold

float: Defines the threshold value for finding intensity peaks. Local maximas below this threshold will not be considered as particles.

particle_positions

np.array: Numpy array with all particle positions on the form *np.array((nParticles,), dtype=[('frame_index', np.int16), ('time', np.float32),('integer_position', np.int16), ('refined_position', np.float32)])*

plot_all_frames (*ax=None*, ***kwargs*)

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.imshow method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

plot_moments (*ax=None*, ***kwargs*)

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.scatter method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of an matplotlib axes object.

`shortest_path`

dict: The shortest path between the start and end point, defined by the cost function. Cost, length and association matrix.

`start_point`

tuple: (frame_index, position_index), The start point of the path you want to find.

`trajectory`

trajectory: The shortest path between the start and end point represented as a trajectory.

CHAPTER 6

Trajectory

class particle_tracker_one_d.**Trajectory**(*pixel_width=1*)

Object that describes a trajectory. With functions for checking if the trajectory describes real diffusion, convenient plotting and calculations of diffusion coefficients.

Parameters

pixel_width: float Defines the length one pixel corresponds to. This value will be used when calculating diffusion coefficients. Default is 1.

Attributes

pixel_width

particle_positions: np.array Numpy array with all particle positions in the trajectory on the form *np.array((nParticles,), dtype=[('frame_index', np.int16), ('time', np.float32),('position', np.int16),('zeroth_order_moment', np.float32),('second_order_moment', np.float32)])*

Methods

calculate_diffusion_coefficient_from_fitting_to_mean_square_displacement([...])	Fits a straight line to the mean square displacement function and calculates the diffusion coefficient from the gradient of the line.
calculate_diffusion_coefficient_using_unbiased_estimator_of_the_diffusion_coefficient([R])	Unbiased estimator of the diffusion coefficient.
calculate_mean_square_displacement_function()	Calculate the average squared displacements for different time steps.
overlaps_with(trajectory)	Check if the trajectories overlaps
plot_trajectory([x, y, ax])	Plots the trajectory using the frame index and the particle position in pixels.
plot_velocity_auto_correlation([ax])	Plots the particle velocity auto correlation function which can be used for examining if the trajectory describes free diffusion.

Continued on next page

Table 1 – continued from previous page

split(trajectory)	If two trajectories overlaps, this function will split them into three or more non overlapping trajectories.

calculate_diffusion_coefficient_from_mean_square_displacement_function(*fit_range=None*)
Fits a straight line to the mean square displacement function and calculates the diffusion coefficient from the gradient of the line. The mean squared displacement of the particle position is proportional to $2Dt$ where D is the diffusion coefficient and t is the time.

fit_range: list, None (default) Define the range of the fit, the data for the fit will be *time[fit_range[0]:fit_range[1]‘* and *mean_squared_displacement[fit_range[0]:fit_range[1]]*.

Returns

diffusion_coefficient: float error: float

calculate_diffusion_coefficient_using_covariance_based_estimator(*R=None*)
Unbiased estimator of the diffusion coefficient. More info at *https://www.nature.com/articles/nmeth.2904*. If the motion blur coefficient is entered a variance estimate is also calculated.

R: float, motion blur coefficient

Returns

diffusion_coefficient: float variance_estimate: float

calculate_mean_square_displacement_function()
Calculate the average squared displacements for different time steps.

Returns

time: np.array

The time corresponding to the mean squared displacements.

msd: np.array The mean squared displacements of the trajectory.

density

float: How dense the trajectory is in time. Returns self.length/(self.particle_positions[‘frame_index’][-1]-self.particle_positions[‘frame_index’][0]).

length

int: The length of the trajectory. Returns self.particle_postions.shape[0]

overlaps_with(*trajectory*)
Check if the trajectories overlaps

trajectory: Trajectory to compare with. If both trajectories has any identical elements will return true otherwise false.

Returns

bool

plot_trajectory(*x='frame_index'*, *y='position'*, *ax=None*, ***kwargs*)
Plots the trajectory using the frame index and the particle position in pixels.

x: str ‘frame_index’, ‘time’, ‘position’ (default), ‘zeroth_order_moment’, ‘second_order_moment’ choose the x-axis value

y: str 'frame_index' (default), 'time', 'position', 'zeroth_order_moment', 'second_order_moment' choose the y-axis value

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.plot method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of a matplotlib axes object.

plot_velocity_auto_correlation(*ax=None*, ***kwargs*)

Plots the particle velocity auto correlation function which can be used for examining if the trajectory describes free diffusion.

ax: matplotlib axes instance The axes which you want the frames to plotted on. If none is provided a new instance will be created.

****kwargs:** Plot settings, any settings which can be used in matplotlib.pyplot.plot method.

Returns

matplotlib axes instance Returns the axes input argument or creates and returns a new instance of a matplotlib axes object.

split(*trajectory*)

If two trajectories overlaps, this function will split them into three or more non overlapping trajectories.

trajectory:

Returns

list Returns a list with the new trajectories

velocities

np.array: The velocities the particle moves at.

CHAPTER 7

Indices and tables

- search

Index

S

T

V