traja Documentation

Release 22.0.0

Justin Shenk

Oct 11, 2022

GETTING STARTED

Trajectory Analysis in Python

Traja allows analyzing trajectory datasets using a wide range of tools, including pandas and R. Traja extends the capability of pandas `DataFrame` specific for animal or object trajectory analysis in 2D, and provides convenient interfaces to other geometric analysis packages (eg, shapely).

CHAPTER

ONE

DESCRIPTION

The Traja Python package is a toolkit for the numerical characterization and analysis of the trajectories of moving animals. Trajectory analysis is applicable in fields as diverse as optimal foraging theory, migration, and behavioural mimicry (e.g. for verifying similarities in locomotion). A trajectory is simply a record of the path followed by a moving object. Traja operates on trajectories in the form of a series of locations (as x, y coordinates) with times. Trajectories may be obtained by any method which provides this information, including manual tracking, radio telemetry, GPS tracking, and motion tracking from videos.

The goal of this package (and this document) is to aid biological researchers, who may not have extensive experience with Python, to analyse trajectories without being restricted by a limited knowledge of Python or programming. However, a basic understanding of Python is useful.

If you use Traja in your publications, please cite our paper in Journal of Open Source Software:

```
@article{Shenk2021,
doi = {10.21105/joss.03202},
url = {https://doi.org/10.21105/joss.03202},
year = {2021},
publisher = {The Open Journal},
volume = {6},
number = {63},
pages = {3202},
author = {Justin Shenk and Wolf Byttner and Saranraj Nambusubramaniyan and Alexander␣
↪Zoeller},
title = {Traja: A Python toolbox for animal trajectory analysis},
journal = {Journal of Open Source Software}
}
```

1.1 Installation

1.1.1 Installing traja

traja requires Python 3.6+ to be installed. For installing on Windows, it is recommend to download and install via conda.

To install via conda:

```
conda install -c conda-forge traja
```

To install via pip:

```
pip install traja
```

To install the latest development version, clone the *GitHub* repository and use the setup script:

```
git clone https://github.com/traja-team/traja.git
cd traja
pip install .
```

1.1.2 Dependencies

Installation with pip should also include all dependencies, but a complete list is

- numpy
- matplotlib
- scipy
- pandas

To install all optional dependencies run:

```
pip install 'traja[all]'
```

1.2 Gallery

A gallery of examples

1.2.1 Animate trajectories

traja allows animating trajectories.

```
import traja

df = traja.generate(1000, seed=0)
```

Plot a animation of trajectory

An animation is generated using *animate()*.

```
anim = traja.plotting.animate(df) # save=True saves to 'trajectory.mp4'
```

Total running time of the script: (0 minutes 0.000 seconds)

1.2.2 3D Plotting with traja

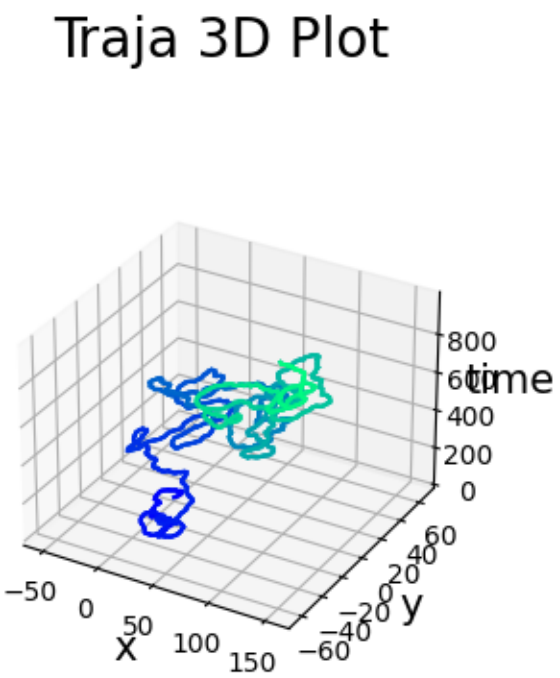
Plot trajectories with time in the vertical axis. Note: Adjust matplotlib args `dist`, `labelpad`, `aspect` and `adjustable` as needed.

```
import traja

df = traja.TrajaDataFrame({"x": [0, 1, 2, 3, 4], "y": [1, 3, 2, 4, 5]})
trj = traja.generate(seed=0)
```

Plot a trajectory in 3D

A 3D plot is generated using *plot_3d()*.

```
trj.traja.plot_3d(dist=15, labelpad=32, title="Traja 3D Plot")
```



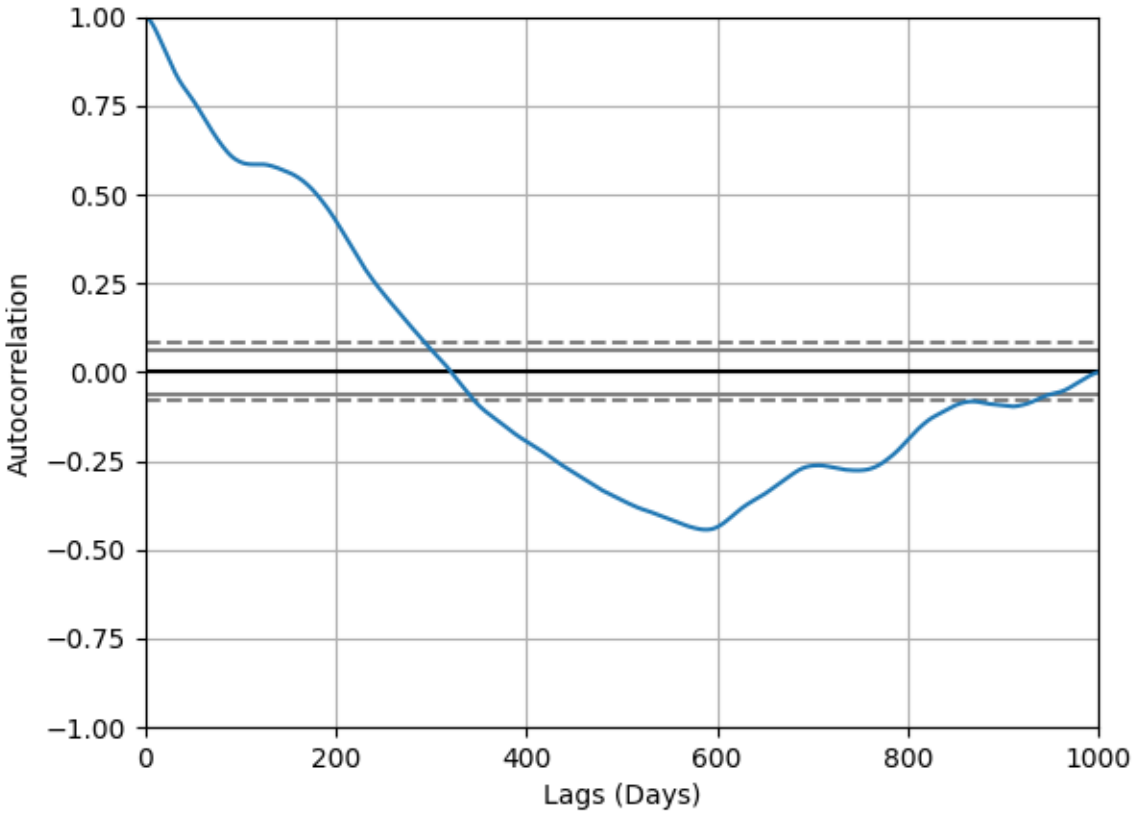

```
<Axes3DSubplot:title={'center':'Traja 3D Plot'}, xlabel='x', ylabel='y'>
```

Total running time of the script: (0 minutes 1.686 seconds)

1.2.3 Autocorrelation plotting with traja

Plot autocorrelation of a trajectory with *traja.plotting.plot_autocorrelation()*

Wrapper for pandas `pandas.plotting.autocorrelation_plot()`.




```
<Figure size 640x480 with 1 Axes>
```

```
import traja

trj = traja.generate(seed=0)
trj.traja.plot_autocorrelation('x')
```

Total running time of the script: (0 minutes 0.210 seconds)

1.2.4 Average direction for each grid cell

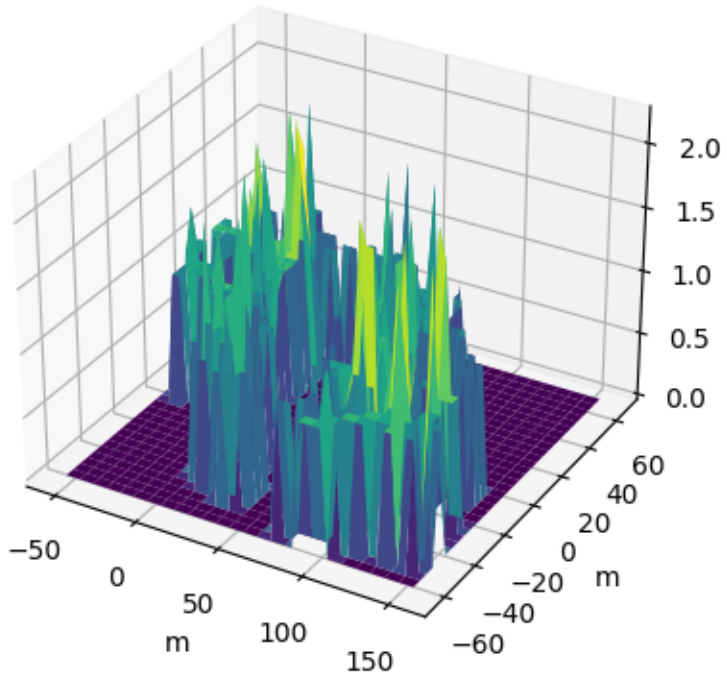
See the flow between grid cells.

```
import traja

df = traja.generate(seed=0)
```

Average Flow (3D)

Flow can be plotted by specifying the *kind* parameter of *traja.plotting.plot_flow()* or by calling the respective functions.

```
import traja

traja.plotting.plot_surface(df, bins=32)
```

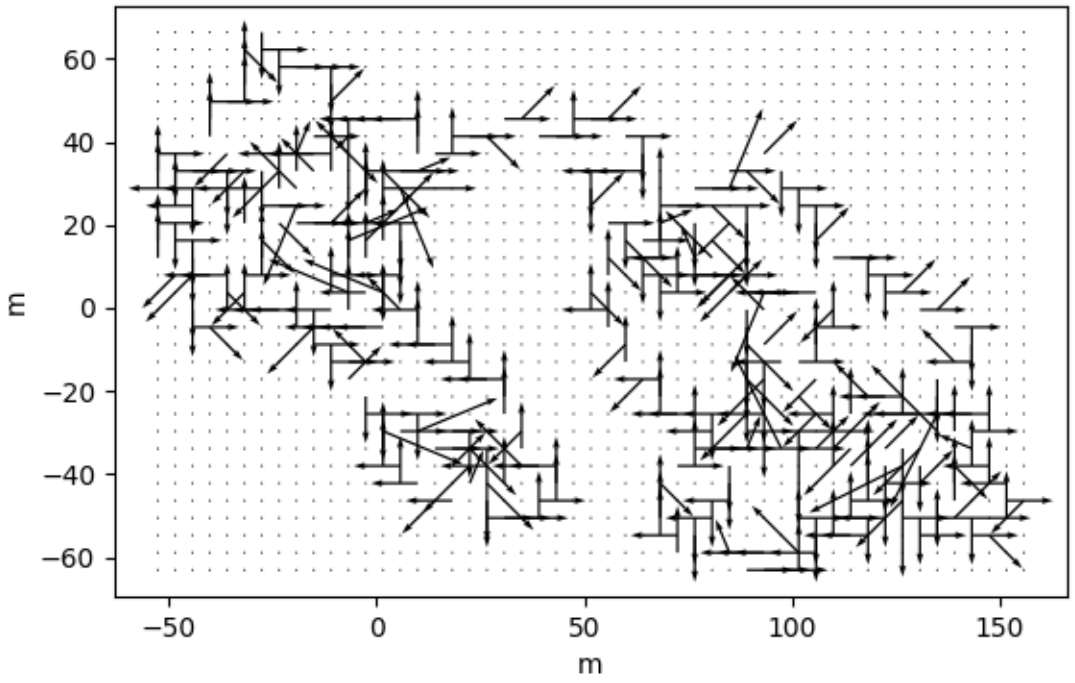



```
<Axes3DSubplot:xlabel='m', ylabel='m'>
```

Quiver

Quiver plot Additional arguments can be specified as a dictionary to *quiverplot_kws*.

```
traja.plotting.plot_quiver(df, bins=32)
```

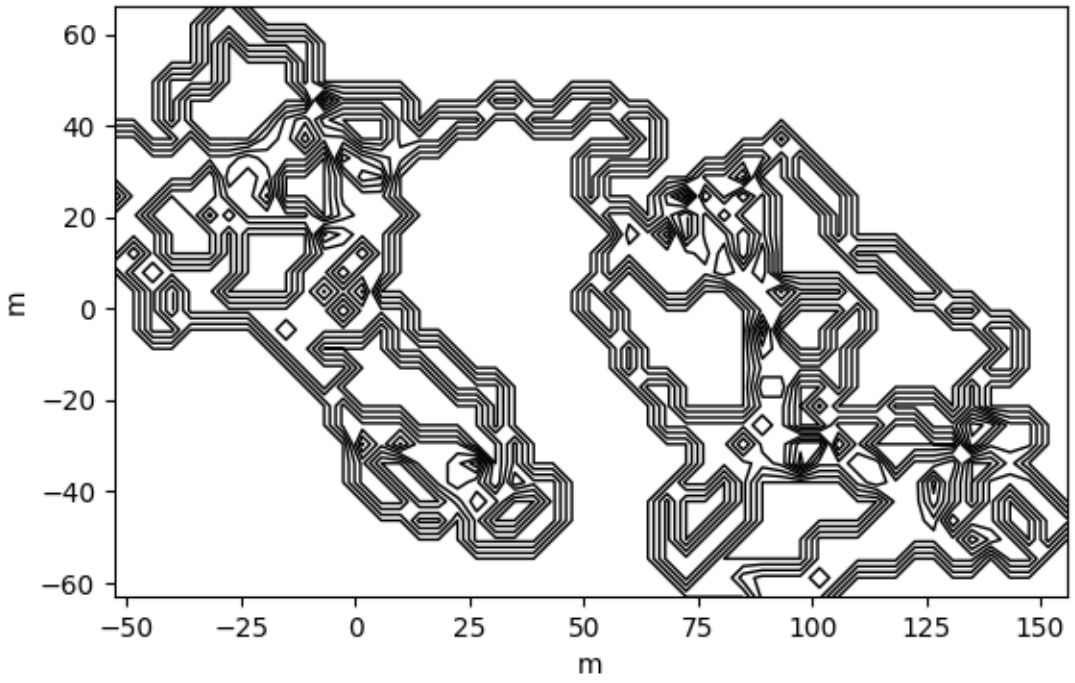



```
<AxesSubplot:xlabel='m', ylabel='m'>
```

Contour

Parameters *filled* and *quiver* are both enabled by default and can be disabled. Additional arguments can be specified as a dictionary to *contourplot_kws*.

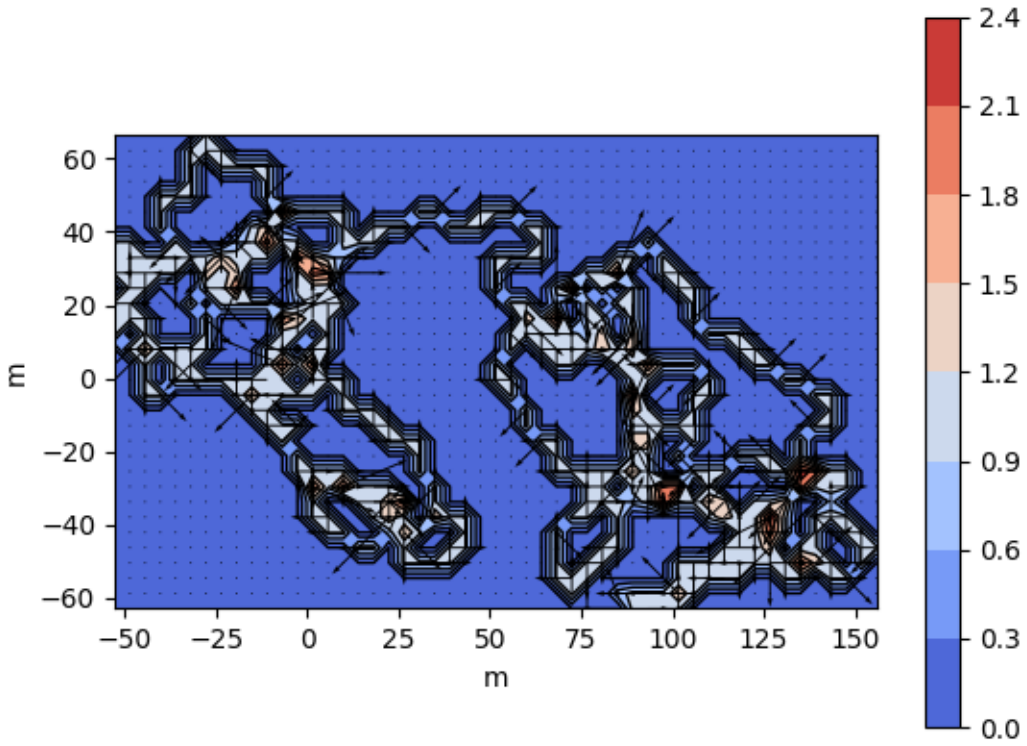
```
traja.plotting.plot_contour(df, filled=False, quiver=False, bins=32)
```




```
<AxesSubplot:xlabel='m', ylabel='m'>
```

Contour (Filled)

```
traja.plotting.plot_contour(df, bins=32, contourfplot_kws={"cmap": "coolwarm"})
```

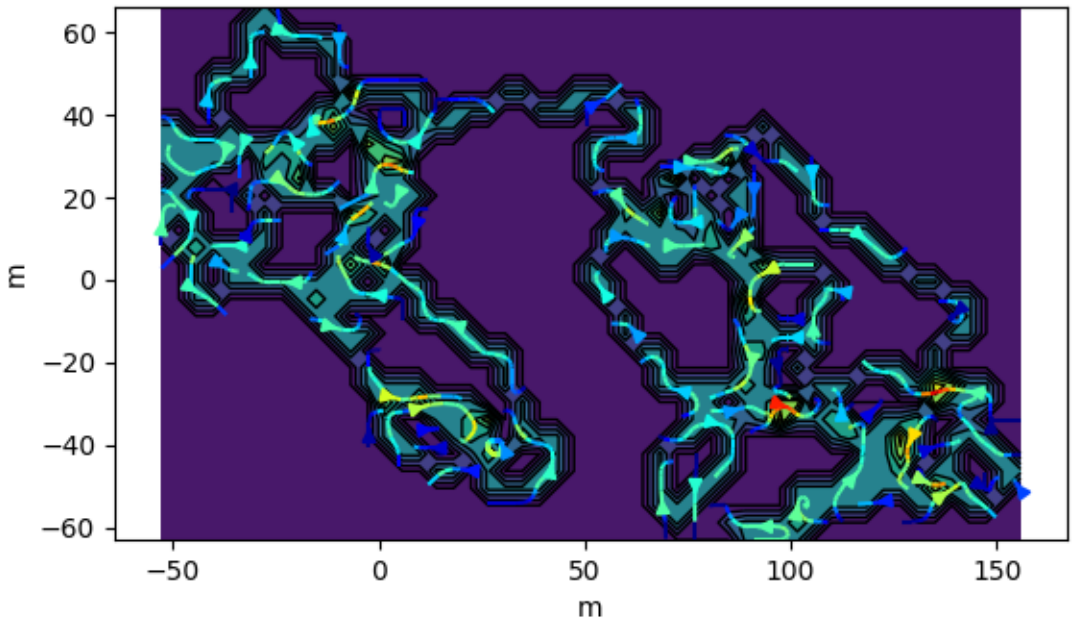



```
<AxesSubplot:xlabel='m', ylabel='m'>
```

Stream

'cmap' can be specified, eg, 'coolwarm', 'viridis', etc. Additional arguments can be specified as a dictionary to 'stream-plot_kws'.

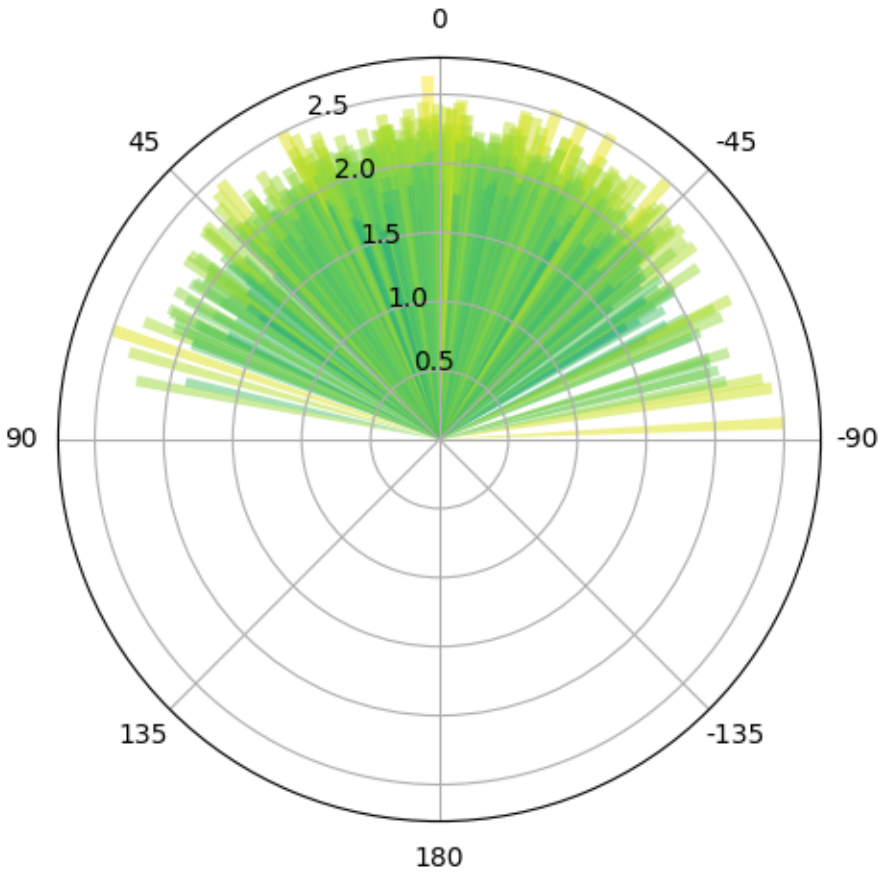
```
traja.plotting.plot_stream(df, cmap="jet", bins=32)
```




```
<AxesSubplot:xlabel='m', ylabel='m'>
```

Polar bar

```
traja.plotting.polar_bar(df)
```




```
/home/docs/checkouts/readthedocs.org/user_builds/traja/envs/stable/lib/python3.7/site-
↪packages/pandas/core/indexing.py:1681: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  self.obj[key] = empty_value
/home/docs/checkouts/readthedocs.org/user_builds/traja/envs/stable/lib/python3.7/site-
↪packages/pandas/core/indexing.py:1773: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_column(ilocs[0], value, pi)
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/traja/plotting.
↪py:1201: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
```

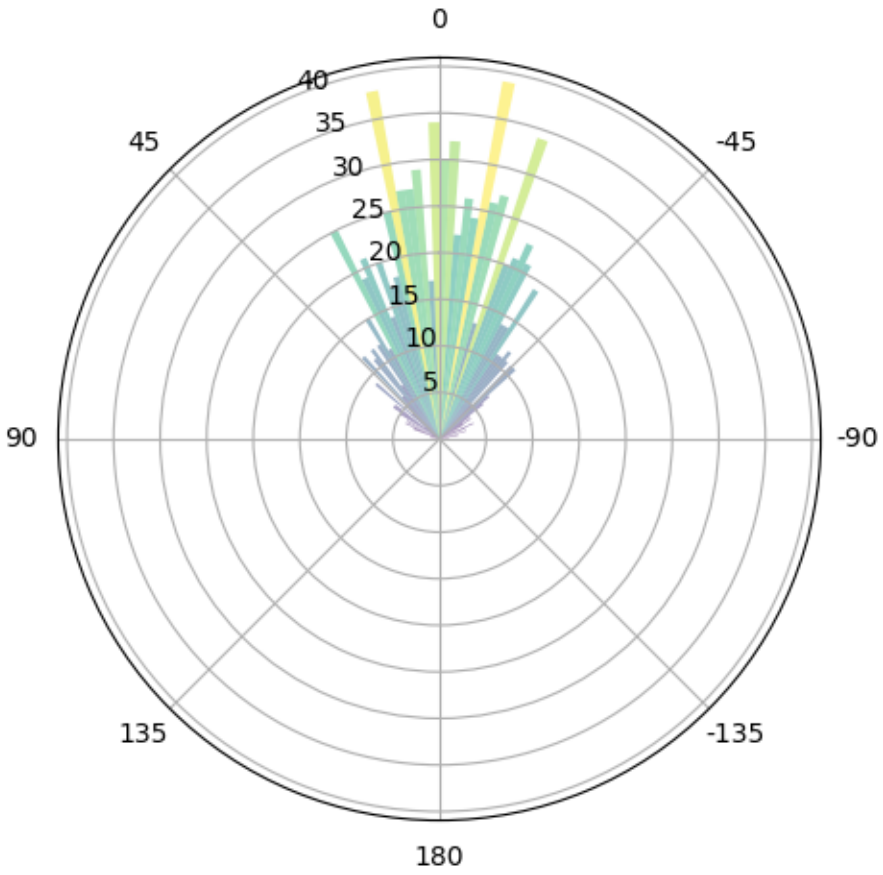
(continues on next page)

(continued from previous page)

```
  trj["turn_angle"] = feature_series
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/traja/frame.
↪py:162: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  self[name] = value
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/traja/plotting.
↪py:1164: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(["0", "45", "90", "135", "180", "-135", "-90", "-45"])

<PolarAxesSubplot:>
```

Polar bar (histogram)

```
traja.plotting.polar_bar(df, overlap=False)
```




```
/home/docs/checkouts/readthedocs.org/user_builds/traja/envs/stable/lib/python3.7/site-
↪packages/pandas/core/indexing.py:1681: SettingWithCopyWarning:
```

(continues on next page)

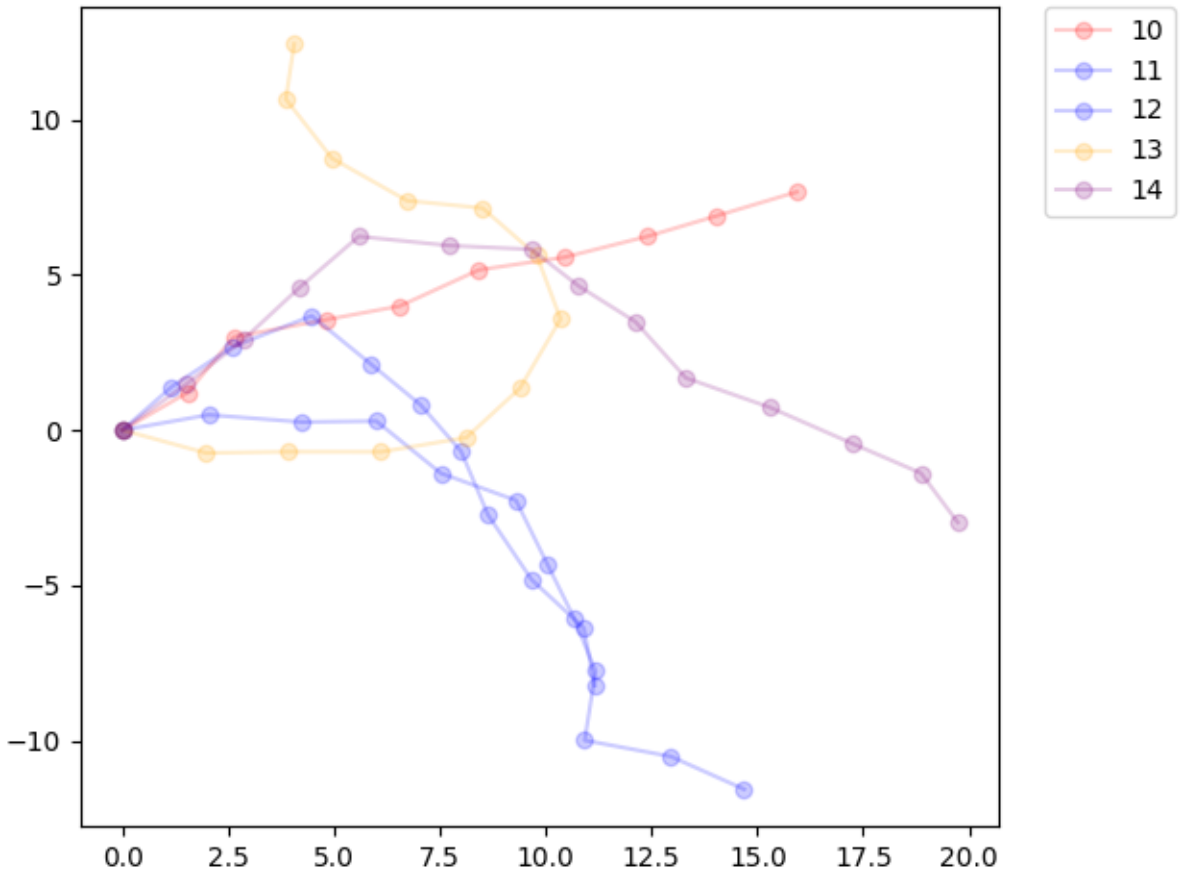
(continued from previous page)

```
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  self.obj[key] = empty_value
/home/docs/checkouts/readthedocs.org/user_builds/traja/envs/stable/lib/python3.7/site-
↪packages/pandas/core/indexing.py:1773: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_column(ilocs[0], value, pi)
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/traja/plotting.
↪py:1201: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  trj["turn_angle"] = feature_series
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/traja/frame.
↪py:162: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  self[name] = value
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/traja/plotting.
↪py:1164: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(["0", "45", "90", "135", "180", "-135", "-90", "-45"])

<PolarAxesSubplot:>
```

Total running time of the script: (0 minutes 13.922 seconds)

1.2.5 Plotting Multiple Trajectories

Plotting multiple trajectories is easy with *plot()*.



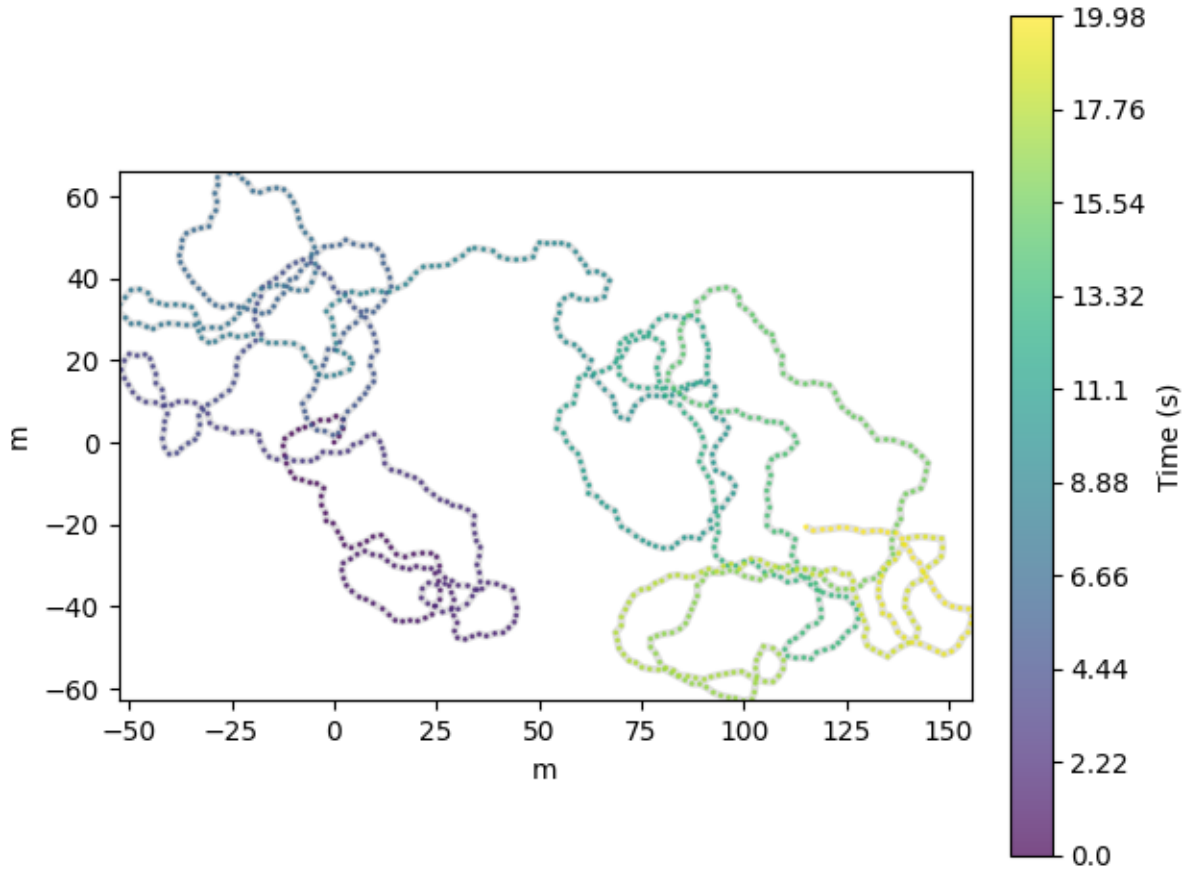

```
import traja
from traja import TrajaCollection

# Create a dictionary of DataFrames, with 'id' as key.
dfs = {idx: traja.generate(idx, seed=idx) for idx in range(10, 15)}

# Create a TrajaCollection.
trjs = TrajaCollection(dfs)

# Note: A TrajaCollection can also be instantiated with a DataFrame, containing and id␣
↪column,
# eg, TrajaCollection(df, id_col="id")

# 'colors' also allows substring matching, eg, {"car":"red", "person":"blue"}
lines = trjs.plot(
    colors={10: "red", 11: "blue", 12: "blue", 13: "orange", 14: "purple"}
)
```

Total running time of the script: (0 minutes 0.173 seconds)

1.2.6 Comparing

traja allows comparing trajectories using various methods.

```
import traja

df = traja.generate(seed=0)
df.traja.plot()
```

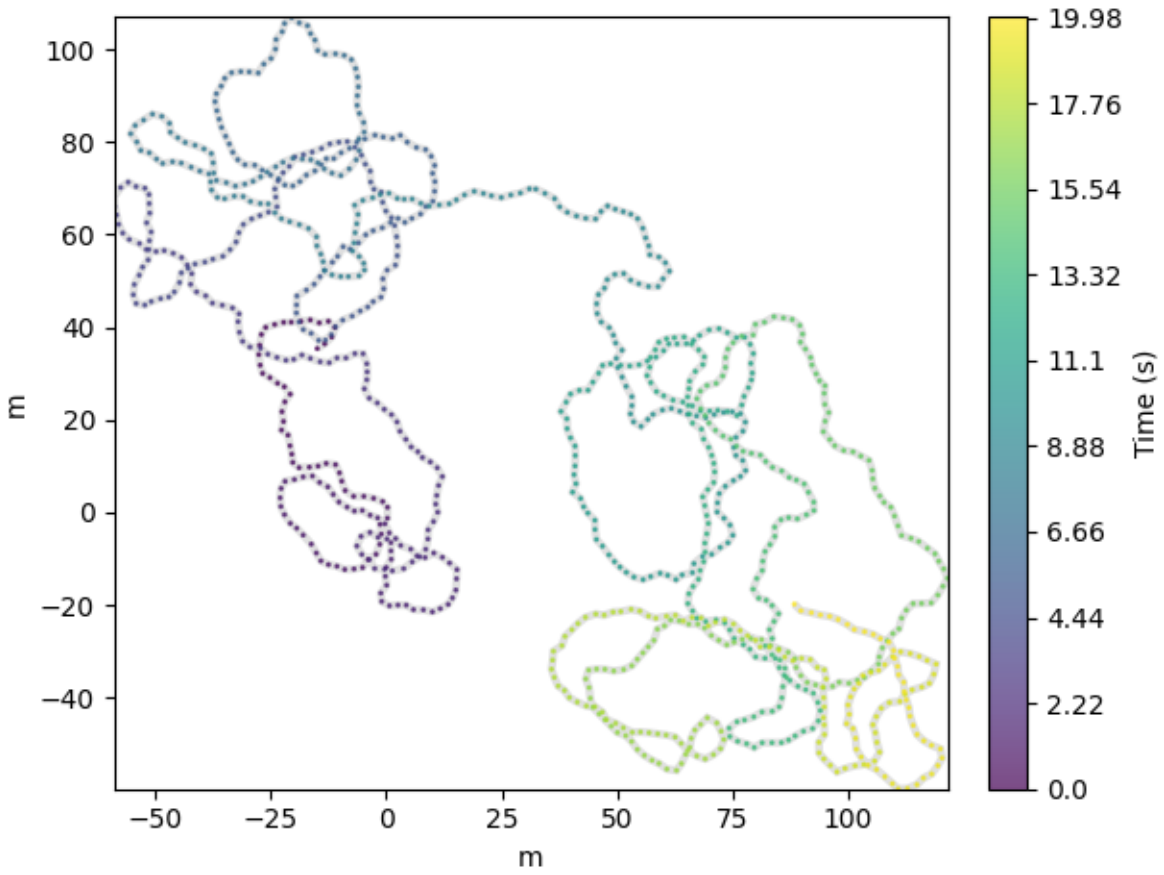



```
<matplotlib.collections.PathCollection object at 0x7f92fd0116d0>
```

Fast Dynamic Time Warping of Trajectories

Fast dynamic time warping can be performed using `fastdtw`. Source article: link.

```
import numpy as np

rotated = traja.rotate(df, angle=np.pi / 10)
rotated.traja.plot()
```

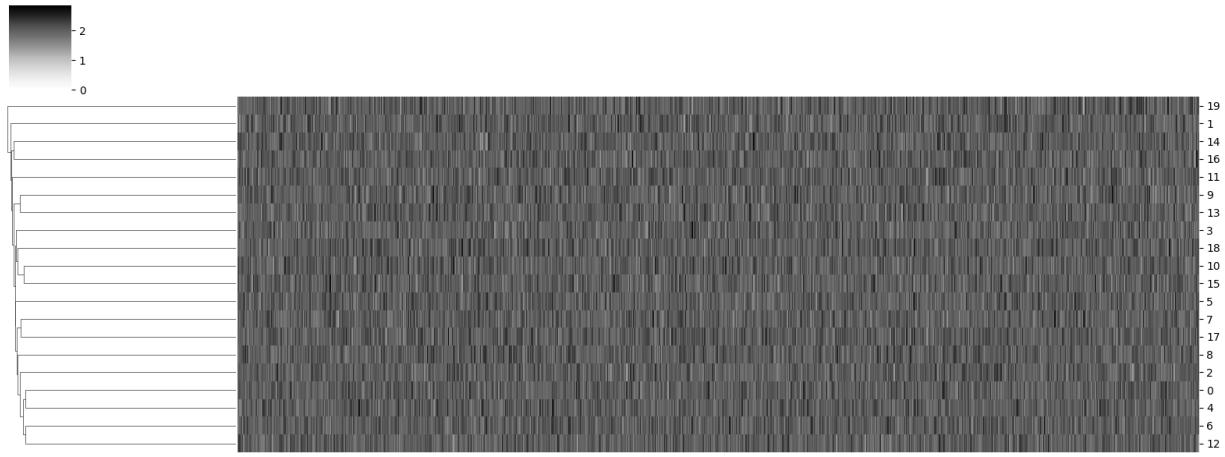



```
<matplotlib.collections.PathCollection object at 0x7f93155e3950>
```

Compare trajectories hierarchically

Hierarchical agglomerative clustering allows comparing trajectories as actograms and finding nearest neighbors. This is useful for comparing circadian rhythms, for example.

```
# Generate random trajectories
trjs = [traja.generate(seed=i) for i in range(20)]

# Calculate displacement
displacements = [trj.traja.calc_displacement() for trj in trjs]

traja.plot_clustermap(displacements)
```




```
/home/docs/checkouts/readthedocs.org/user_builds/traja/envs/stable/lib/python3.7/site-
↪packages/seaborn/matrix.py:657: UserWarning: Clustering large matrix with scipy.␣
↪Installing `fastcluster` may give better performance.
  warnings.warn(msg)

<seaborn.matrix.ClusterGrid object at 0x7f92fcd1b550>
```

Compare trajectories point-wise

```
dist = traja.distance_between(df.traja.xy, rotated.traja.xy)

print(f"Distance between the two trajectories is {dist}")
```

```
Distance between the two trajectories is 27254.161638114736
```

Total running time of the script: (0 minutes 1.407 seconds)

1.2.7 Plotting trajectories on a grid

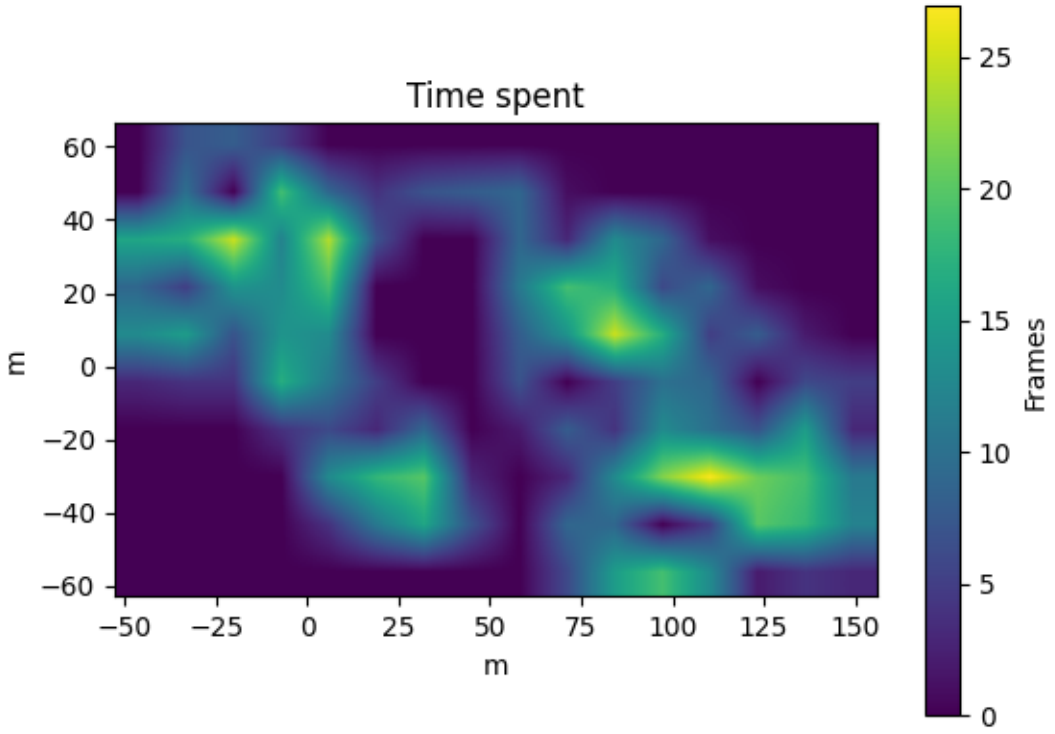
traja allows comparing trajectories using various methods.

```
import traja

df = traja.generate(seed=0)
```

Plot a heat map of the trajectory

A heat map can be generated using `trip_grid()`.

```
df.traja.trip_grid()
```

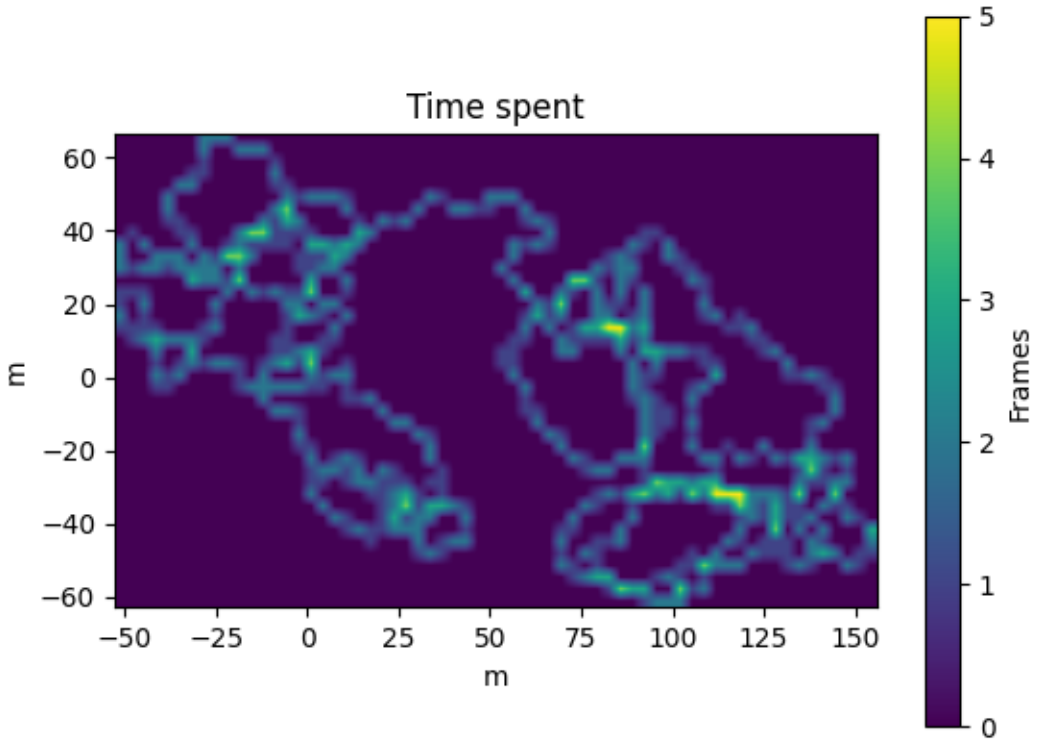



```
(array([[ 0.,  7.,  8.,  5.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.],
       [ 0., 10.,  0., 19.,  9.,  4.,  7.,  8.,  9.,  1.,  0.,  0.,  0.,
         0.,  0.,  0.],
       [16., 17., 25., 12., 24.,  7.,  0.,  0.,  9.,  3., 13.,  9.,  1.,
         0.,  0.,  0.],
       [ 9.,  5., 14., 13., 19.,  0.,  0.,  0., 11., 19., 17.,  6.,  9.,
         1.,  0.,  0.],
       [13., 15.,  7., 14., 13.,  0.,  0.,  0.,  8., 14., 25., 17.,  5.,
         8.,  2.,  0.],
       [ 3.,  4.,  4., 17., 11.,  5.,  0.,  0.,  7.,  0.,  5., 10.,  9.,
         0.,  6.,  5.],
       [ 0.,  0.,  0.,  4.,  6.,  3.,  9.,  0.,  2.,  8.,  4., 13., 10.,
         7., 15.,  3.],
       [ 0.,  0.,  0.,  0., 13., 18., 20.,  3.,  0.,  3., 13., 22., 27.,
        21., 19., 11.],
       [ 0.,  0.,  0.,  0.,  4., 11., 16.,  7.,  0.,  9.,  9.,  0.,  5.,
        20., 18., 12.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  6., 15., 19., 13.,
         2.,  4.,  3.]]), <matplotlib.image.AxesImage object at 0x7f931643ad90>)
```

Increase the grid resolution

Number of bins can be specified with the `bins` parameter.

```
df.traja.trip_grid(bins=40)
```




```
(array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]]), <matplotlib.image.AxesImage object at␣
↪0x7f9314441390>)
```

Convert coordinates to grid indices

Number of x and y bins can be specified with the `bins` parameter.

```
from traja.trajectory import grid_coordinates

grid_coords = grid_coordinates(df, bins=32)
print(grid_coords.head())
```

```
   xbin  ybin
0    12    15
1    13    15
2    13    16
3    13    16
4    13    17
```

Transitions as Markov first-order Markov model

Probability of transitioning between cells is computed using *traja.trajectory.transitions()*.

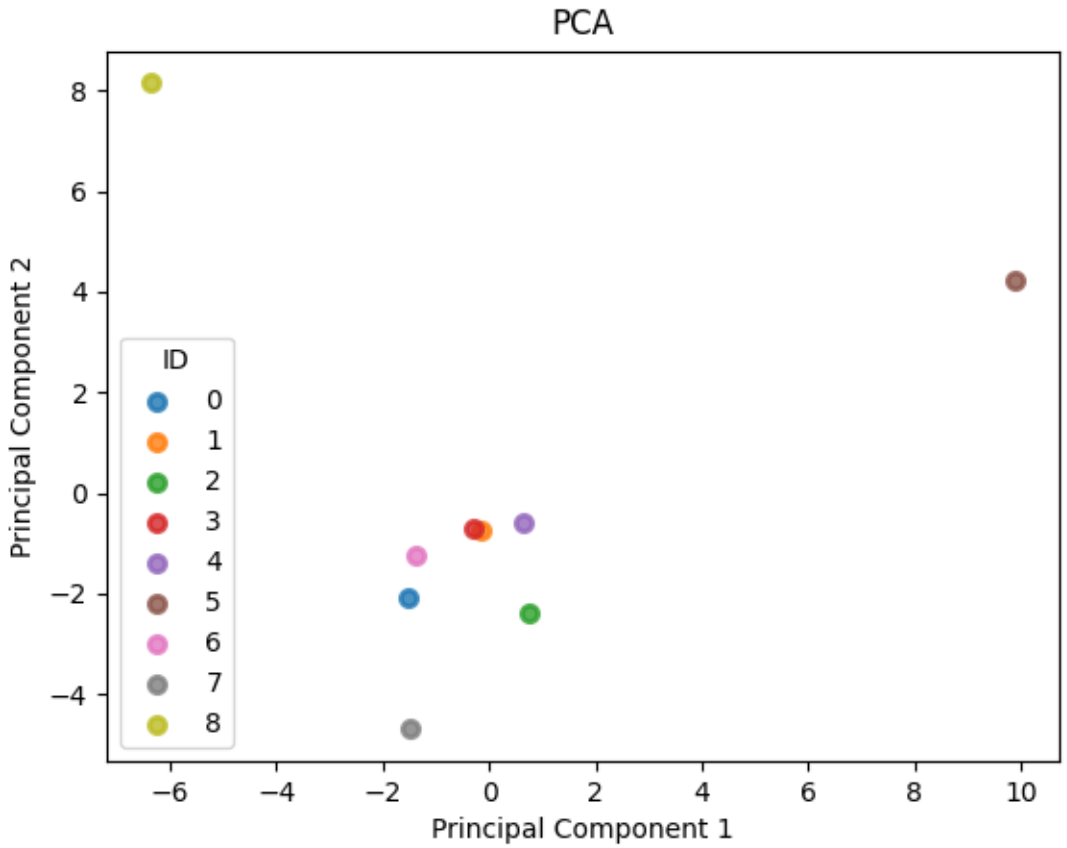
```
transitions_matrix = traja.trajectory.transitions(df, bins=32)
print(transitions_matrix[:10])
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Total running time of the script: (0 minutes 0.642 seconds)

1.2.8 Plot PCA with traja

Plot PCA of a trip grid with *traja.plotting.plot_pca()*




```
/home/docs/checkouts/readthedocs.org/user_builds/traja/checkouts/stable/docs/examples/
↪plot_pca.py:10: FutureWarning: The error_bad_lines argument has been deprecated and␣
↪will be removed in a future version.


  df = traja.dataset.example.jaguar()

<Figure size 640x480 with 1 Axes>
```

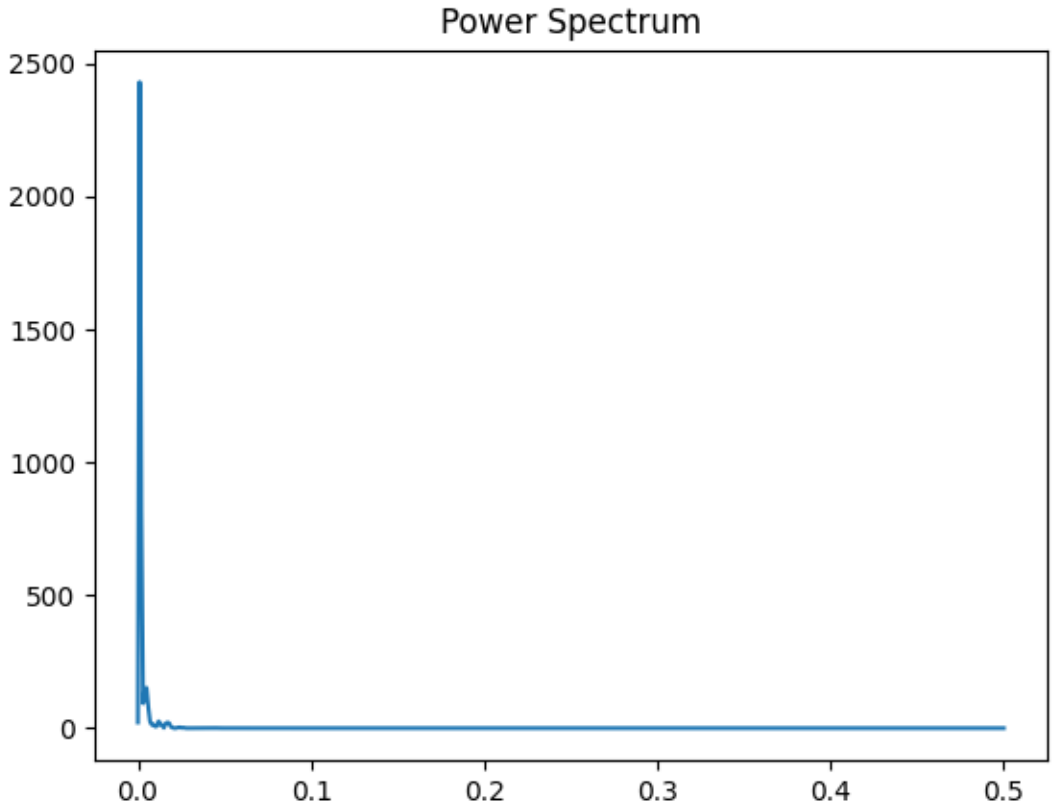
```
import traja

# Load sample jaguar dataset with trajectories for 9 animals
df = traja.dataset.example.jaguar()

# Bin trajectory into a trip grid then perform PCA
traja.plotting.plot_pca(df, id_col="ID", bins=(8,8))
```

Total running time of the script: (0 minutes 0.355 seconds)

1.2.9 Periodogram plot with traja

Plot periodogram or power spectrum with *traja.plotting.plot_periodogram()*.

Wrapper for pandas `scipy.signal.periodogram()`.




```
<Figure size 640x480 with 1 Axes>
```

```
import traja

trj = traja.generate(seed=0)
trj.traja.plot_periodogram('x')
```

Total running time of the script: (0 minutes 0.107 seconds)

1.3 Reading and Writing Files

1.3.1 Reading trajectory data

traja allows reading files via *traja.parsers.read_file()*. For example a CSV file `trajectory.csv` with the following contents:

```
x,y
1,1
1,2
1,3
```

Could be read in like:

```
import traja

df = traja.read_file('trajectory.csv')
```

`read_file` returns a *TrajaDataFrame* with access to all pandas and traja methods.

Any keyword arguments passed to *read_file* will be passed to `pandas.read_csv()`.

Data frames can also be read with pandas `pandas.read_csv()` and then converted to TrajaDataFrames with:

```
import traja
import pandas as pd

df = pd.read_csv('data.csv')

# If x and y columns are named different than "x" and "y", rename them, eg:
df = df.rename(columns={"x_col": "x", "y_col": "y"}) # original column names x_col, y_col

# If the time column doesn't include "time" in the name, similarly rename it to "time"

trj = traja.TrajaDataFrame(df)
```

1.3.2 Writing trajectory data

Files can be saved using the built-in pandas `pandas.to_csv()`.

```
df.to_csv('trajectory.csv')
```

1.4 Pandas Indexing and Resampling

Traja is built on top of pandas `DataFrame`, giving access to low-level pandas indexing functions.

This allows indexing, resampling, etc., just as in pandas:

```
from traja import generate, plot
import pandas as pd

```

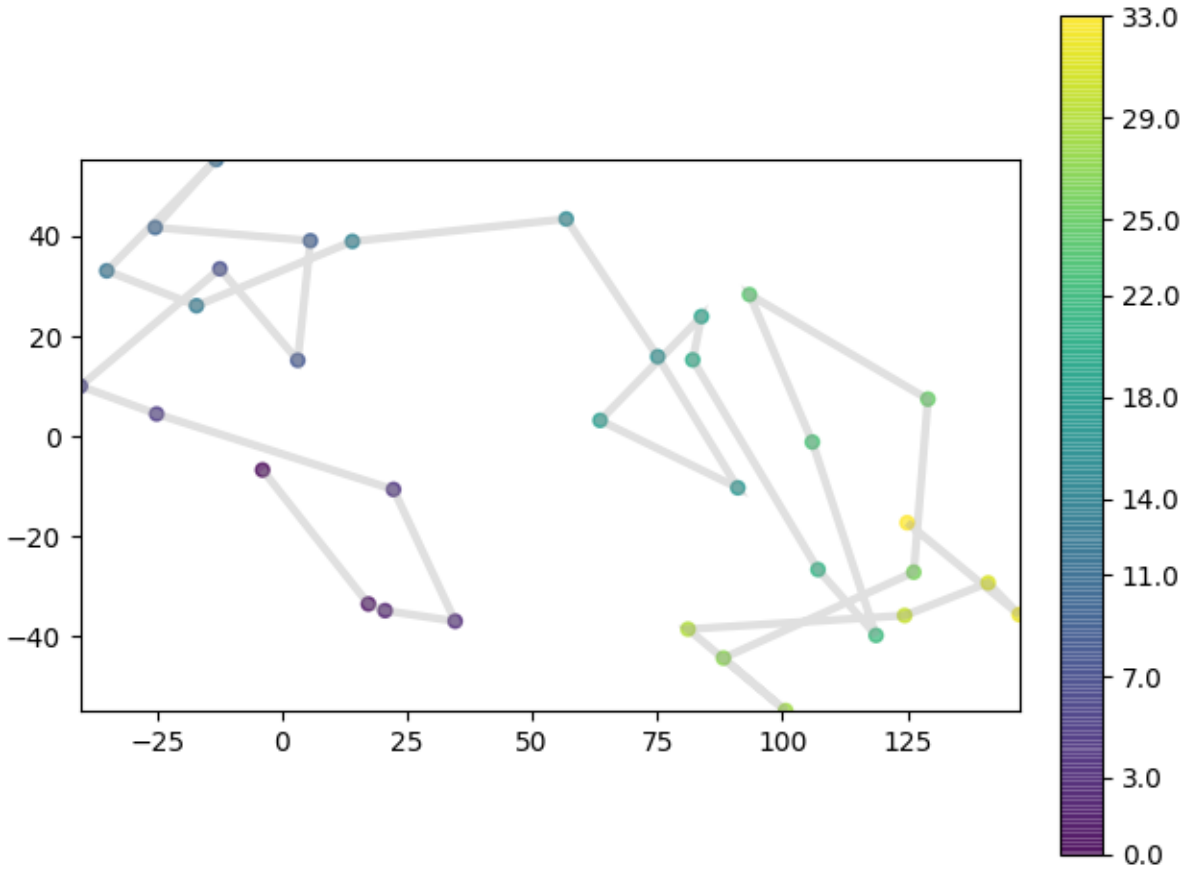
(continues on next page)

(continued from previous page)

```
# Generate random walk
df = generate(n=1000, fps=30)

# Select every second row
df[::2]

Output:
          x         y      time
0  0.000000  0.000000  0.000000
2  2.364589  3.553398  0.066667
4  0.543251  6.347378  0.133333
6 -3.307575  5.404562  0.200000
8 -6.697132  3.819403  0.266667
```

You can also do resampling to select average coordinate every second, for example:

```
# Convert 'time' column to timedelta
df.time = pd.to_timedelta(df.time, unit='s')
df = df.set_index('time')

# Resample with average for every second
resampled = df.resample('S').mean()
plot(resampled)
```



1.5 Generate Random Walk

Random walks can be generated using *generate()*.

```
In [1]: import traja

# Generate random walk
In [2]: df = traja.generate(1000)
```

traja.trajectory.**generate**(*n: int = 1000, random: bool = True, step_length: int = 2, angular_error_sd: float = 0.5, angular_error_dist: Optional[Callable] = None, linear_error_sd: float = 0.2, linear_error_dist: Optional[Callable] = None, fps: float = 50, spatial_units: str = 'm', seed: Optional[int] = None, convex_hull: bool = False, **kwargs*)

Generates a trajectory.

If `random` is `True`, the trajectory will be a correlated random walk/idiothetic directed walk (Kareiva & Shigesada, 1983), corresponding to an animal navigating without a compass (Cheung, Zhang, Stricker, & Srinivasan, 2008). If `random` is `False`, it will be(np.ndarray) a directed walk/allothetic directed walk/oriented path, corresponding to an animal navigating with a compass (Cheung, Zhang, Stricker, & Srinivasan, 2007, 2008).

By default, for both random and directed walks, errors are normally distributed, unbiased, and independent of each other, so are **simple directed walks** in the terminology of Cheung, Zhang, Stricker, & Srinivasan, (2008). This behaviour may be modified by specifying alternative values for the `angular_error_dist` and/or `linear_error_dist` parameters.

The initial angle (for a random walk) or the intended direction (for a directed walk) is `0` radians. The starting position is `(0, 0)`.

Parameters

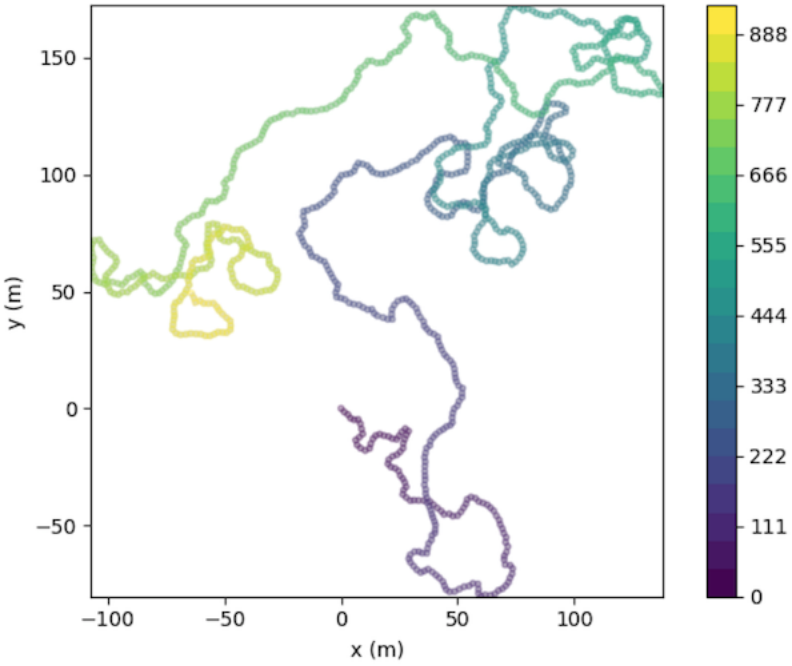
- **n** (*int*) – (Default value = 1000)
- **random** (*bool*) – (Default value = True)
- **step_length** – (Default value = 2)
- **angular_error_sd** (*float*) – (Default value = 0.5)
- **angular_error_dist** (*Callable*) – (Default value = None)
- **linear_error_sd** (*float*) – (Default value = 0.2)
- **linear_error_dist** (*Callable*) – (Default value = None)
- **fps** (*float*) – (Default value = 50)
- **convex_hull** (*bool*) – (Default value = False)
- **spatial_units** – (Default value = 'm')
- ****kwargs** – Additional arguments

Returns
Trajectory

Return type
trj (`traja.frame.TrajaDataFrame`)

Note: Based on Jim McLean's trajr, ported to Python.

Reference: McLean, D. J., & Skowron Volponi, M. A. (2018). trajr: An R package for characterisation of animal trajectories. Ethology, 124(6), 440-448. https://doi.org/10.1111/eth.12739.



1.6 Smoothing and Analysis

1.6.1 Smoothing

Smoothing can be performed using *smooth_sg()*.

traja.trajectory.**smooth_sg**(*trj: TrajaDataFrame*, *w: Optional[int] = None*, *p: int = 3*)

Returns DataFrame of trajectory after Savitzky-Golay filtering.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **w** (*int*) – window size (Default value = None)
- **p** (*int*) – polynomial order (Default value = 3)

Returns

Trajectory

Return type

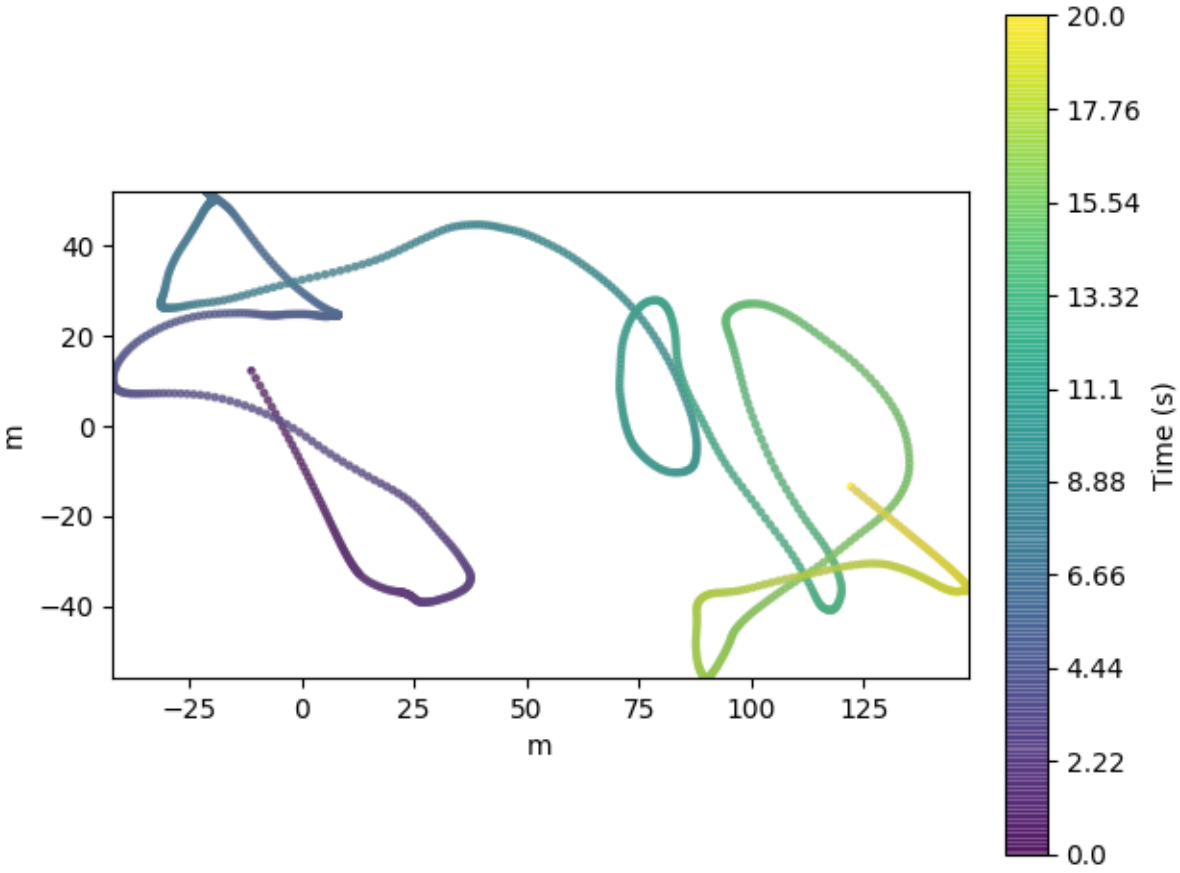
trj (TrajaDataFrame)

```
>> df = traja.generate()
>> traja.smooth_sg(df, w=101).head()
           x          y  time
0 -11.194803  12.312742  0.00
1 -10.236337  10.613720  0.02
2  -9.309282   8.954952  0.04
```

(continues on next page)

(continued from previous page)

```
3  -8.412910   7.335925  0.06
4  -7.546492   5.756128  0.08
```



1.6.2 Length

Length of trajectory can be calculated using *length()*.

`traja.trajectory.`**`length`**(*trj: TrajaDataFrame*) → float

Calculates the cumulative length of a trajectory.

Parameters

trj (TrajaDataFrame) – Trajectory

Returns

length (float)

```
>> df = traja.generate()
>> traja.length(df)
2001.142339606066
```

1.6.3 Distance

Net displacement of trajectory (start to end) can be calculated using *distance()*.

traja.trajectory.**distance**(*trj: TrajaDataFrame*) → float

Calculates the distance from start to end of trajectory, also called net distance, displacement, or bee-line from start to finish.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
distance (float)

```
>> df = traja.generate()
>> traja.distance(df)
117.01507823153617
```

1.6.4 Displacement

Displacement (distance travelled) can be calculated using *calc_displacement()*.

traja.trajectory.**calc_displacement**(*trj: TrajaDataFrame*, *lag=1*)

Returns a Series of float displacement between consecutive indices.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **lag** (*int*) – time steps between displacement calculation

Returns
Displacement series.

Return type
displacement (pandas.Series)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> traja.calc_displacement(df)
0         NaN
1    1.414214
2    1.414214
Name: displacement, dtype: float64
```

1.6.5 Derivatives

traja.trajectory.**get_derivatives**(*trj: TrajaDataFrame*)

Returns derivatives displacement, displacement_time, speed, speed_times, acceleration, acceleration_times as dictionary.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
Derivatives

Return type
derivs (DataFrame)

```
>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3],'time':[0.,0.2,0.4]})
>> df.traja.get_derivatives()
   displacement  displacement_time     speed  speed_times  acceleration ␣
↪acceleration_times
0           NaN                0.0       NaN          NaN           NaN ␣
↪    NaN
1      1.414214                0.2  7.071068          0.2           NaN ␣
↪    NaN
2      1.414214                0.4  7.071068          0.4           0.0 ␣
↪    0.4
```

1.6.6 Speed Intervals

traja.trajectory.**speed_intervals**(*trj: TrajaDataFrame*, *faster_than: Optional[float] = None*, *slower_than: Optional[float] = None*) → DataFrame

Calculate speed time intervals.

Returns a dictionary of time intervals where speed is slower and/or faster than specified values.

Parameters

- **faster_than** (*float, optional*) – Minimum speed threshold. (Default value = None)
- **slower_than** (*float or int, optional*) – Maximum speed threshold. (Default value = None)

Returns
result (DataFrame) – time intervals as dataframe

Note: Implementation ported to Python, heavily inspired by Jim McLean's trajr package.

```
>> df = traja.generate()
>> intervals = traja.speed_intervals(df, faster_than=100)
>> intervals.head()
   start_frame  start_time  stop_frame  stop_time  duration
0            1        0.02           3       0.06      0.04
1            4        0.08           8       0.16      0.08
2           10        0.20          11       0.22      0.02
3           12        0.24          15       0.30      0.06
4           17        0.34          18       0.36      0.02
```

1.7 Turns and Angular Analysis

1.7.1 Turns

Turns can be calculated using *calc_angle()*.

traja.trajectory.**calc_angle**(*trj: TrajaDataFrame*, *unit: str = 'degrees'*, *lag: int = 1*)

Returns a Series with angle between steps as a function of displacement with regard to x axis.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **unit** (*str*) – return angle in radians or degrees (Default value: 'degrees')
- **lag** (*int*) – time steps between angle calculation (Default value: 1)

Returns
Angle series.

Return type
angle (pandas.Series)

1.7.2 Heading

Heading can be calculated using *calc_heading()*.

traja.trajectory.**calc_heading**(*trj: TrajaDataFrame*)

Calculate trajectory heading.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
heading as a Series

Return type
heading (pandas.Series)

..doctest:

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> traja.calc_heading(df)
0     NaN
1    45.0
2    45.0
Name: heading, dtype: float64
```

1.7.3 Angles

Angles can be calculated using `angles()`.

1.8 Plotting Paths

Making plots of trajectories is easy using the `plot()` method.

See the gallery for more examples.

`traja.plotting.bar_plot`(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, ***kwargs*) → Axes

Plot trajectory for single animal over period.

Parameters

- **trj** (`traja.TrajaDataFrame`) – trajectory
- **bins** (*int or tuple*) – number of bins for x and y
- ****kwargs** – additional keyword arguments to `mpl_toolkits.mplot3d.Axed3D.plot()`

Returns
Axes of plot

Return type
ax (`PathCollection`)

`traja.plotting.plot`(*trj: TrajaDataFrame*, *n_coords: Optional[int] = None*, *show_time: bool = False*, *accessor: Optional[TrajaAccessor] = None*, *ax=None*, ***kwargs*) → PathCollection

Plot trajectory for single animal over period.

Parameters

- **trj** (`traja.TrajaDataFrame`) – trajectory
- **n_coords** (*int, optional*) – Number of coordinates to plot
- **show_time** (*bool*) – Show colormap as time
- **accessor** (`TrajaAccessor`, optional) – TrajaAccessor instance
- **ax** (`Axes`) – axes for plotting
- **interactive** (*bool*) – show plot immediately
- ****kwargs** – additional keyword arguments to `matplotlib.axes.Axes.scatter()`

Returns
collection that was plotted

Return type
collection (`PathCollection`)

`traja.plotting.plot_actogram`(*series: Series*, *dark=(19, 7)*, *ax: Optional[Axes] = None*, ***kwargs*)

Plot activity or displacement as an actogram.

Note: For published example see Eckel-Mahan K, Sassone-Corsi P. Phenotyping Circadian Rhythms in Mice. Curr Protoc Mouse Biol. 2015;5(3):271-281. Published 2015 Sep 1. doi:10.1002/9780470942390.mo140229

---

traja.plotting.**plot_contour**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *filled: bool = True*, *quiver: bool = True*, *contourplot_kws: dict = {}*, *contourfplot_kws: dict = {}*, *quiverplot_kws: dict = {}*, *ax: Optional[Axes] = None*, ***kwargs*) → Axes

Plot average flow from each grid cell to neighbor.

Parameters

- **trj** – Traja DataFrame
- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **filled** (*bool*) – Contours filled
- **quiver** (*bool*) – Quiver plot
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`
- **quiverplot_kws** – Additional keyword arguments for `quiver()`
- **ax** (*optional*) – Matplotlib Axes

Returns
Axes of quiver plot

Return type
ax (`Axes`)

traja.plotting.**plot_flow**(*trj: TrajaDataFrame*, *kind: str = 'quiver'*, **args*, *contourplot_kws: dict = {}*, *contourfplot_kws: dict = {}*, *streamplot_kws: dict = {}*, *quiverplot_kws: dict = {}*, *surfaceplot_kws: dict = {}*, ***kwargs*) → Figure

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **kind** (*str*) – Choice of 'quiver','contourf','stream','surface'. Default is 'quiver'.
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`
- **streamplot_kws** – Additional keyword arguments for `streamplot()`
- **quiverplot_kws** – Additional keyword arguments for `quiver()`
- **surfaceplot_kws** – Additional keyword arguments for `plot_surface()`

Returns
Axes of plot

Return type
ax (`Axes`)

traja.plotting.**plot_quiver**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *quiverplot_kws: dict = {}*, ***kwargs*) → Axes

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio

- **quiverplot_kws** – Additional keyword arguments for quiver()

Returns
Axes of quiver plot

Return type
ax (Axes)

traja.plotting.**plot_stream**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *cmap: str = 'viridis'*, *contourfplot_kws: dict = {}*, *contourplot_kws: dict = {}*, *streamplot_kws: dict = {}*, ***kwargs*) → Figure

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **contourplot_kws** – Additional keyword arguments for contour()
- **contourfplot_kws** – Additional keyword arguments for contourf()
- **streamplot_kws** – Additional keyword arguments for streamplot()

Returns
Axes of stream plot

Return type
ax (Axes)

traja.plotting.**plot_surface**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *cmap: str = 'viridis'*, ***surfaceplot_kws: dict*) → Figure

Plot surface of flow from each grid cell to neighbor in 3D.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **cmap** (*str*) – color map
- **surfaceplot_kws** – Additional keyword arguments for plot_surface()

Returns
Axes of quiver plot

Return type
ax (Axes)

traja.plotting.**polar_bar**(*trj: TrajaDataFrame*, *feature: str = 'turn_angle'*, *bin_size: int = 2*, *threshold: float = 0.001*, *overlap: bool = True*, *ax: Optional[Axes] = None*, ***plot_kws: str*) → Axes

Plot polar bar chart.

Parameters

- **trj** (traja.TrajaDataFrame) – trajectory
- **feature** (*str*) – Options: 'turn_angle', 'heading'
- **bin_size** (*int*) – width of bins
- **threshold** (*float*) – filter for step distance
- **overlap** (*bool*) – Overlapping shows all values, if set to false is a histogram

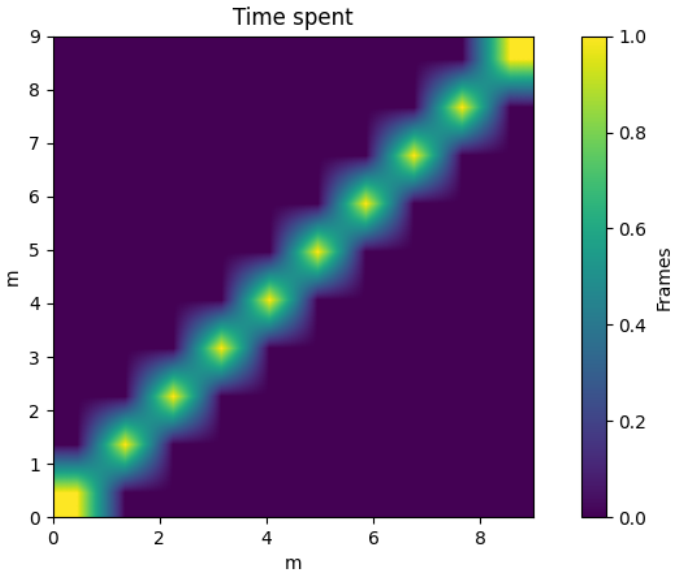
Returns
Axes of plot

Return type
ax (PathCollection)

1.8.1 Trip Grid

Trip grid can be plotted for `trip_grid`:

```
In [1]: import traja

In [2]: from traja import trip_grid

In [3]: df = traja.TrajaDataFrame({'x':range(10),'y':range(10)})

In [4]: hist, image = trip_grid(df);
```



If only the histogram is need for further computation, use the *hist_only* option:

```
In [5]: hist, _ = trip_grid(df, hist_only=True)

In [6]: print(hist[:5])
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
```

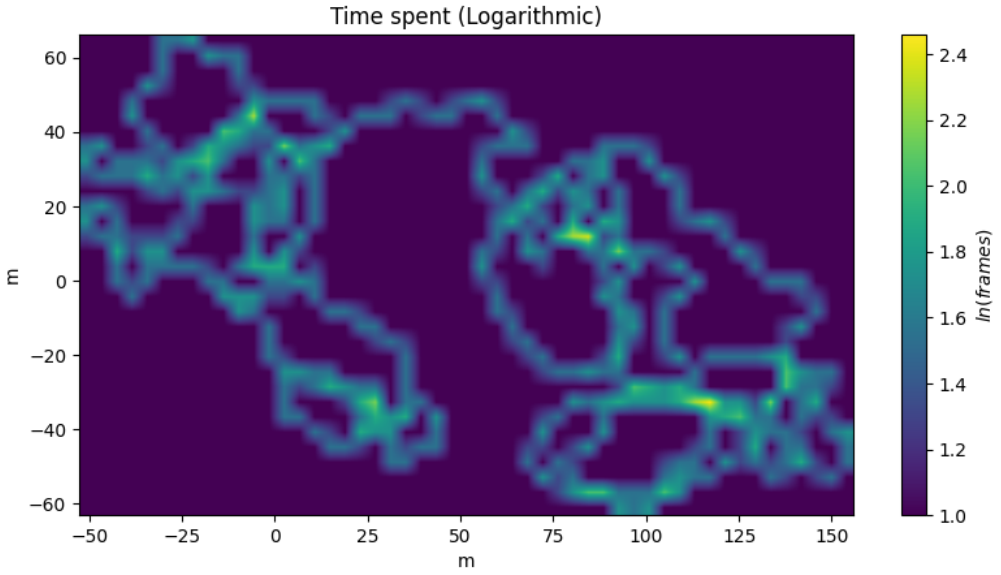
Highly dense plots be more easily visualized using the *bins* and *log* argument:

```
# Generate random walk
In [7]: df = traja.generate(1000)
```

(continues on next page)

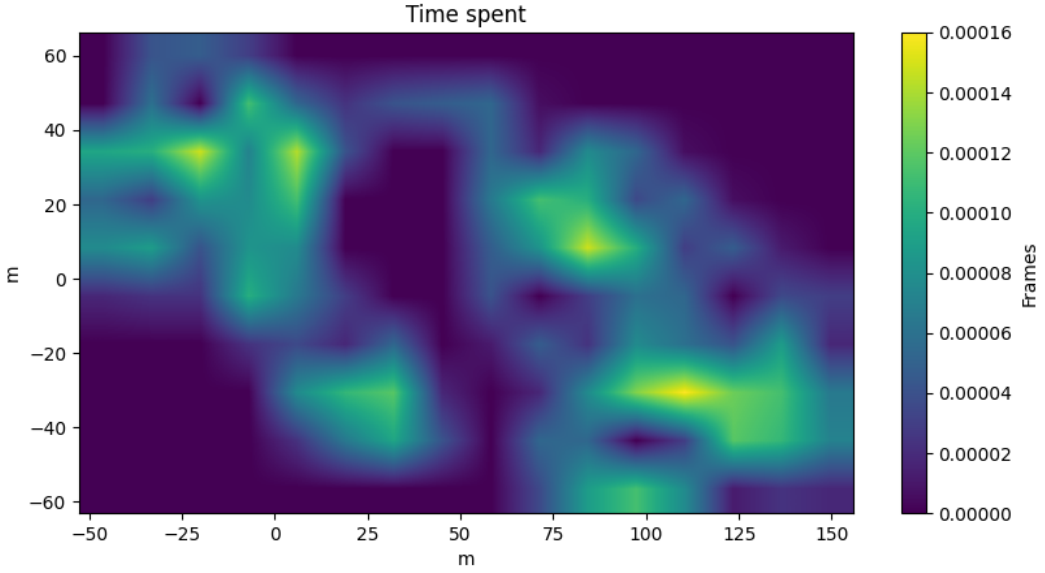
(continued from previous page)

```
In [8]: trip_grid(df, bins=32, log=True);
```



The plot can also be normalized into a density function with *normalize*:

```
In [9]: hist, _ = trip_grid(df, normalize=True);
```



1.8.2 Animate

traja.plotting.**animate**(*trj: TrajaDataFrame*, *polar: bool = True*, *save: bool = False*)

Animate trajectory.

Parameters

- **polar** (*bool*) – include polar bar chart with turn angle
- **save** (*bool*) – save video to `trajectory.mp4`

Returns
animation

Return type
anim (matplotlib.animation.FuncAnimation)

1.9 Periodicity

Several methods for analyzing periodicity are included.

1.9.1 Autocorrelation

Autocorrelation is plotted using `pandas.plotting.autocorrelation_plot()`.

traja.plotting.**plot_autocorrelation**(*trj: TrajaDataFrame*, *coord: str = 'y'*, *unit: str = 'Days'*, *xmax: int = 1000*, *interactive: bool = True*)

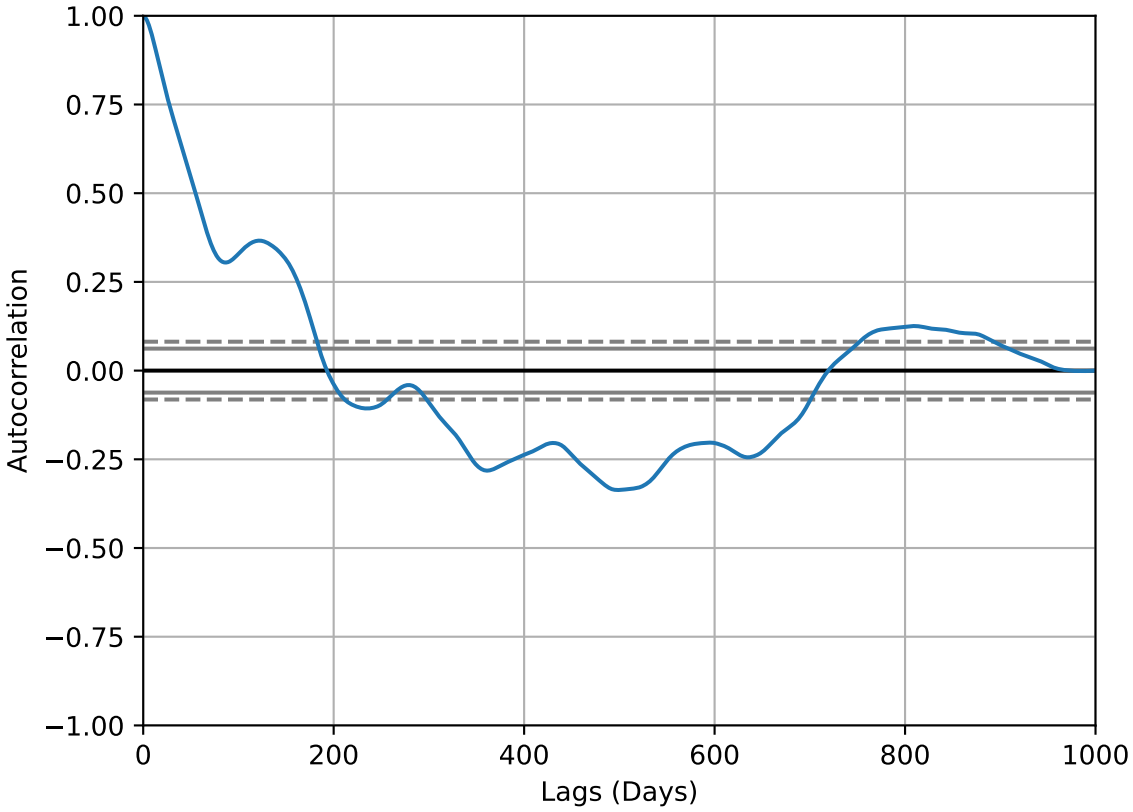
Plot autocorrelation of given coordinate.

Parameters

- **Trajectory** (*trj -*) –
- **'y'** (*coord - 'x' or*) –
- **string** (*unit -*) –
- **eg** –
- **'Days'** –
- **value** (*xmax - max xaxis*) –
- **immediately** (*interactive - Plot*) –

Returns
Matplotlib Figure

Note: Convenience wrapper for pandas `autocorrelation_plot()`.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Autocorrelation
0
200
400
600
800
1000
Lags (Days)

1.9.2 Periodogram (Power Spectum)

Convenience wrapper for `scipy.signal.periodogram()`.

`traja.plotting.plot_periodogram`(*trj*, *coord: str = 'y'*, *fs: int = 1*, *interactive: bool = True*)

Plot power spectral density of `coord` timeseries using a periodogram.

Parameters

- **Trajectory** (*trj -*) –
- **'y'** (*coord - choice of 'x' or*) –
- **frequency** (*fs - Sampling*) –
- **immediately** (*interactive - Plot*) –

Returns

Figure

Note: Convenience wrapper for `scipy.signal.periodogram()`.

1.10 Plotting Grid Cell Flow

Trajectories can be discretized into grid cells and the average flow from each grid cell to its neighbor can be plotted with *plot_flow()*, eg:

```
traja.plot_flow(df, kind='stream')
```

1.10.1 `plot_flow()` `kind` Arguments

- *surface* - 3D surface plot extending `mpl_toolkits.mplot3D.Axes3D.plot_surface`()
- *contourf* - Filled contour plot extending `matplotlib.axes.Axes.contourf()`
- *quiver* - Quiver plot extending `matplotlib.axes.Axes.quiver()`
- *stream* - Stream plot extending `matplotlib.axes.Axes.streamplot()`

See the gallery for more examples.

1.10.2 3D Surface Plot

`traja.plotting.plot_surface`(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *cmap: str = 'viridis'*, ***surfaceplot_kws: dict*) → Figure

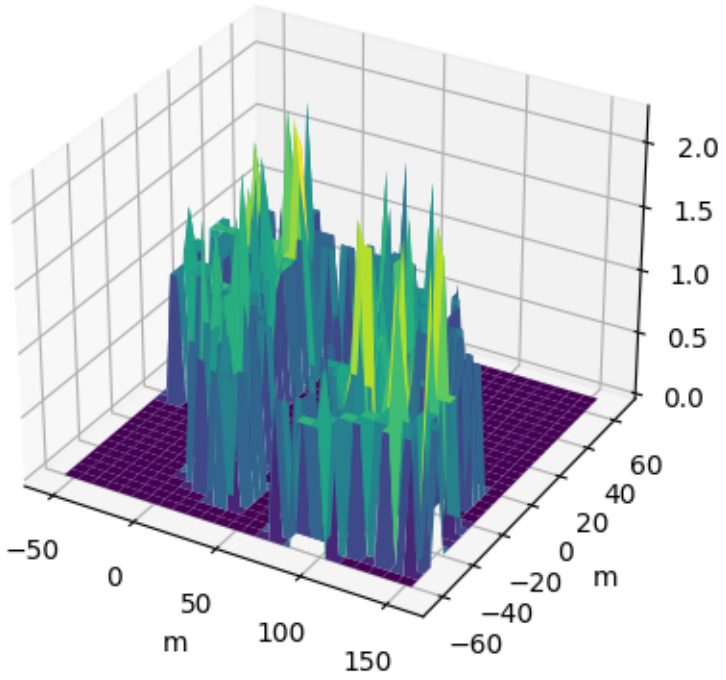
Plot surface of flow from each grid cell to neighbor in 3D.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **cmap** (*str*) – color map
- **surfaceplot_kws** – Additional keyword arguments for `plot_surface()`

Returns
Axes of quiver plot

Return type
ax (Axes)



1.10.3 Quiver Plot

traja.plotting.**plot_quiver**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *quiverplot_kws: dict = {}*, ***kwargs*) → Axes

Plot average flow from each grid cell to neighbor.

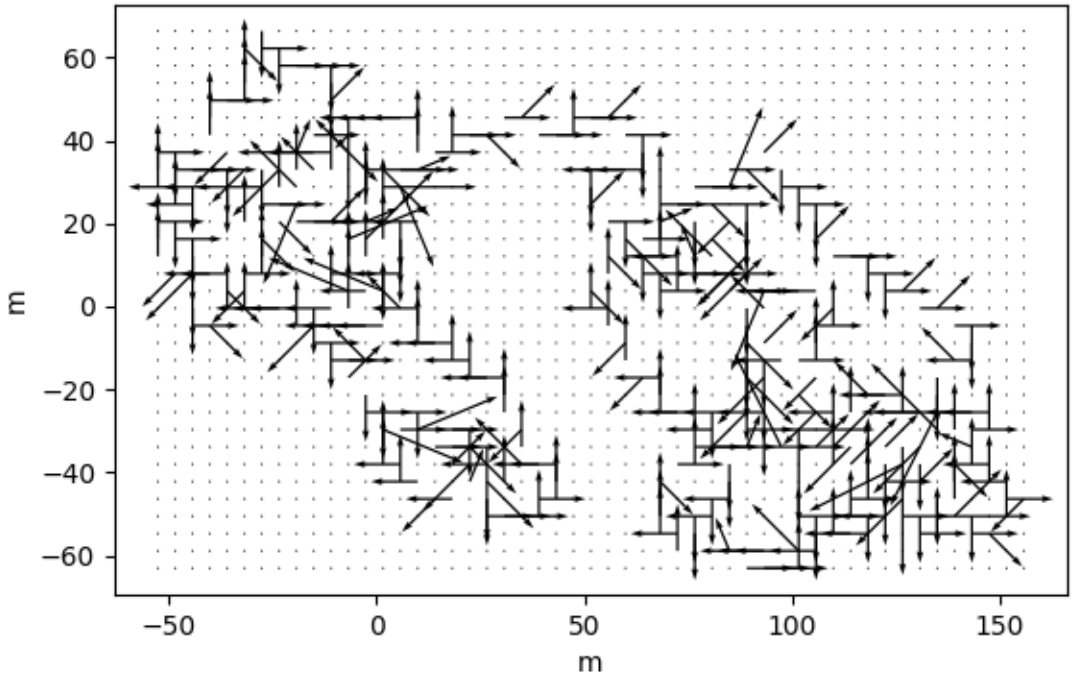
Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **quiverplot_kws** – Additional keyword arguments for quiver()

Returns
Axes of quiver plot

Return type
ax (Axes)

```
traja.plot_quiver(df, bins=32)
```



1.10.4 Contour Plot

traja.plotting.**plot_contour**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *filled: bool = True*, *quiver: bool = True*, *contourplot_kws: dict = {}*, *contourfplot_kws: dict = {}*, *quiverplot_kws: dict = {}*, *ax: Optional[Axes] = None*, ***kwargs*) → Axes

Plot average flow from each grid cell to neighbor.

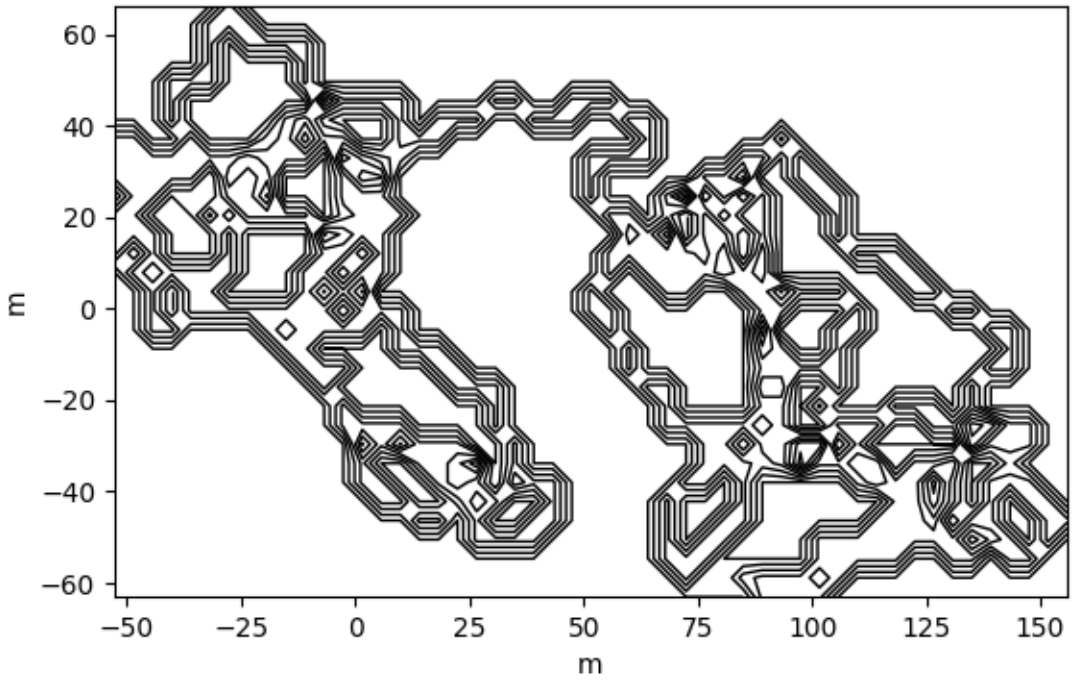
Parameters

- **trj** – Traja DataFrame
- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **filled** (*bool*) – Contours filled
- **quiver** (*bool*) – Quiver plot
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`
- **quiverplot_kws** – Additional keyword arguments for `quiver()`
- **ax** (*optional*) – Matplotlib Axes

Returns
Axes of quiver plot

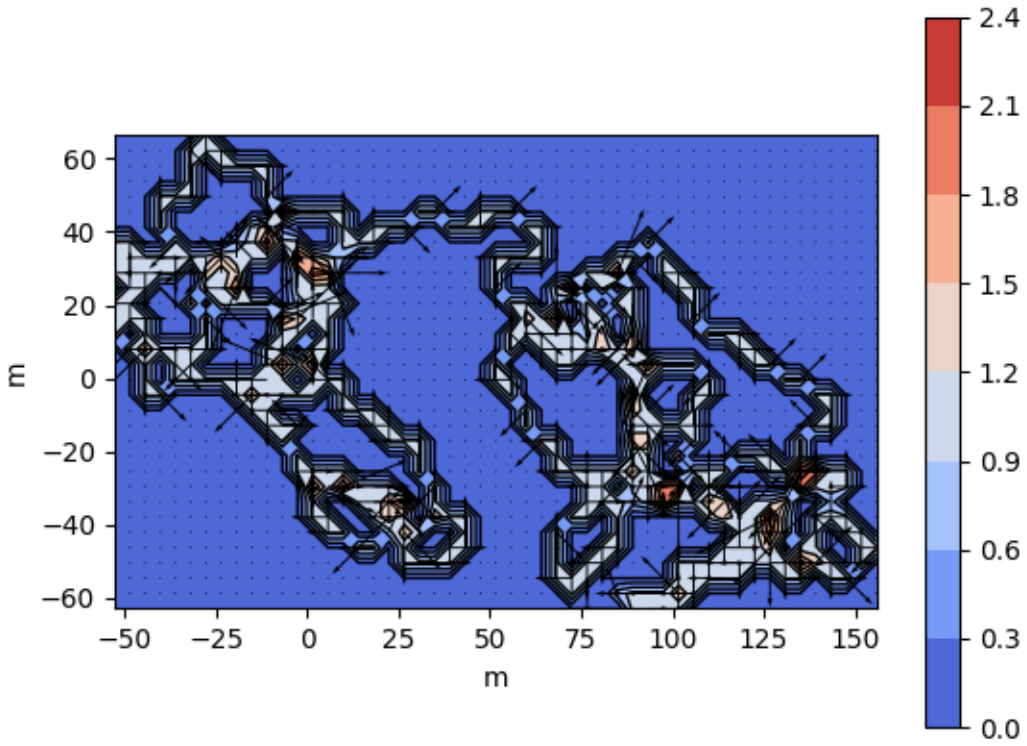
Return type
ax (Axes)

```
traja.plot_contour(df, filled=False, quiver=False, bins=32)
```



1.10.5 Contour Plot (Filled)

```
traja.plot_contour(df, filled=False, quiver=False, bins=32)
```



1.10.6 Stream Plot

traja.plotting.**plot_stream**(*trj: TrajaDataFrame*, *bins: Optional[Union[int, tuple]] = None*, *cmap: str = 'viridis'*, *contourfplot_kws: dict = {}*, *contourplot_kws: dict = {}*, *streamplot_kws: dict = {}*, ***kwargs*) → Figure

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`
- **streamplot_kws** – Additional keyword arguments for `streamplot()`

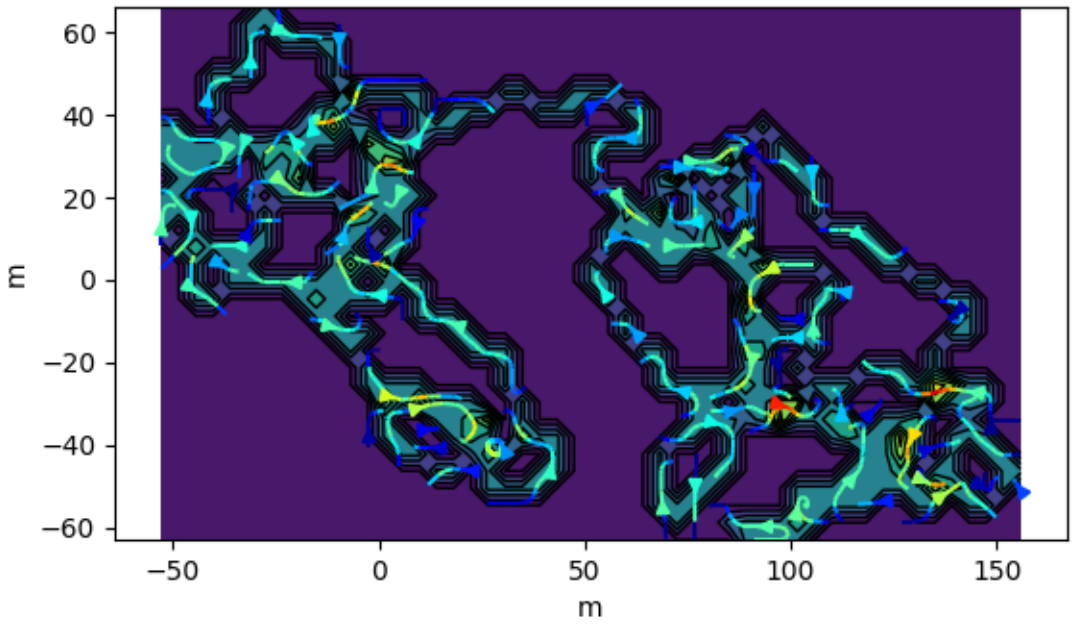
Returns

Axes of stream plot

Return type

ax (`Axes`)

```
traja.plot_contour(df, bins=32, contourfplot_kws={'cmap':'coolwarm'})
```



1.11 Resampling Trajectories

1.11.1 Rediscretize

Rediscretize the trajectory into consistent step lengths with `rediscretize()` where the *R* parameter is the new step length.

Note: Based on the appendix in Bovet and Benhamou, (1988) and Jim McLean's trajr implementation.

1.11.2 Resample time

resample_time() allows resampling trajectories by a `step_time`.

`traja.trajectory.`**`resample_time`**(*trj: TrajaDataFrame*, *step_time: str*, *new_fps: Optional[bool] = None*)

Returns a `TrajaDataFrame` resampled to consistent *step_time* intervals.

`step_time` should be expressed as a number-time unit combination, eg "2S" for 2 seconds and "2100L" for 2100 milliseconds.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **step_time** (*str*) – step time interval / offset string (eg, '2S' (seconds), '50L' (milliseconds), '50N' (nanoseconds))
- **new_fps** (*bool, optional*) – new fps

Results:

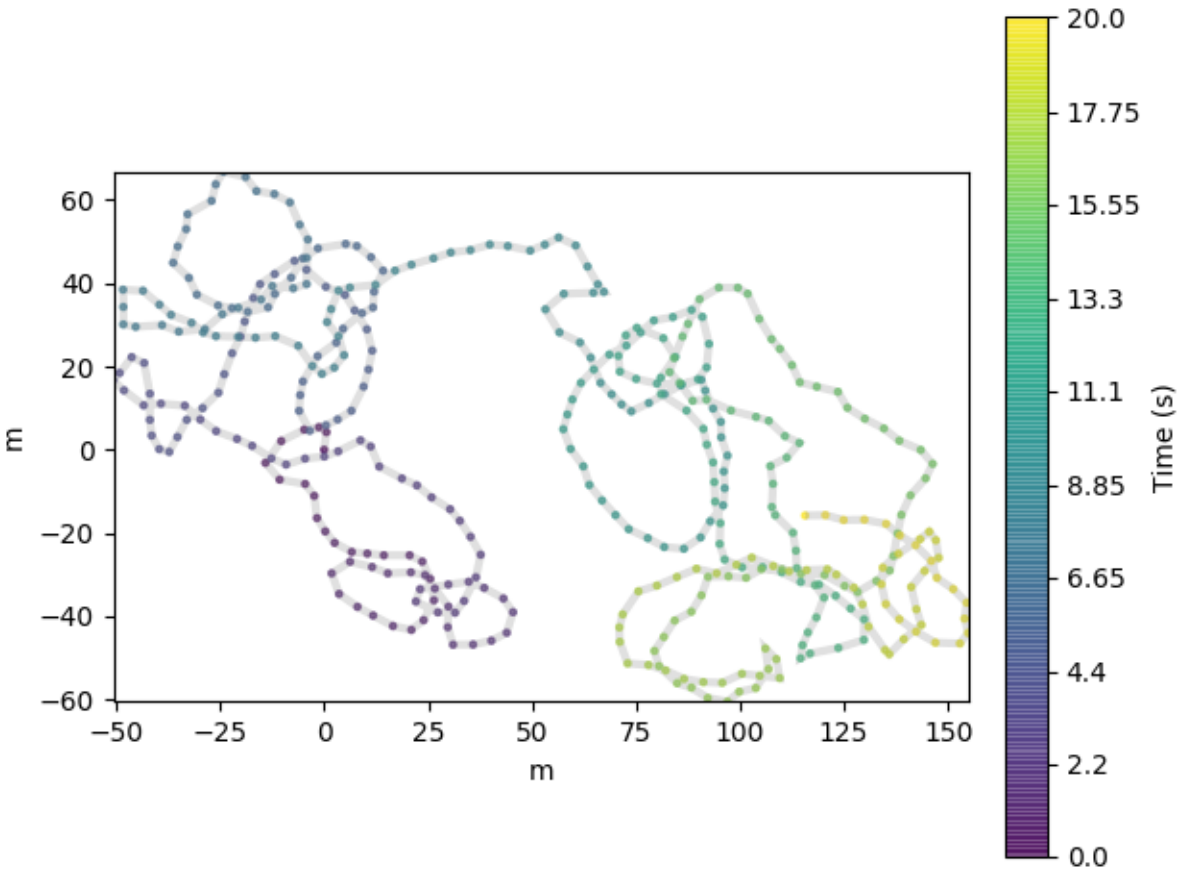
trj (TrajaDataFrame): Trajectory

```
>>> from traja import generate, resample_time
>>> df = generate()
>>> resampled = resample_time(df, '50L') # 50 milliseconds
>>> resampled.head()
                                  x         y
time
1970-01-01 00:00:00.000   0.000000  0.000000
1970-01-01 00:00:00.050   0.919113  4.022971
1970-01-01 00:00:00.100  -1.298510  5.423373
1970-01-01 00:00:00.150  -6.057524  4.708803
1970-01-01 00:00:00.200 -10.347759  2.108385
```

For example:

```
In [1]: import traja

# Generate a random walk
In [2]: df = traja.generate(n=1000) # Time is in 0.02-second intervals

In [3]: df.head()
Out[3]:
          x         y  time
0  0.000000  0.000000  0.00
1  1.225654  1.488762  0.02
2  2.216797  3.352835  0.04
3  2.215322  5.531329  0.06
4  0.490209  6.363956  0.08
```

```
In [4]: resampled = traja.resample_time(df, "50L") # 50 milliseconds

In [5]: resampled.head()
Out[5]:
                                  x         y
time
1970-01-01 00:00:00.000   0.000000  0.000000
1970-01-01 00:00:00.050   0.919113  4.022971
1970-01-01 00:00:00.100  -1.298510  5.423373
1970-01-01 00:00:00.150  -6.057524  4.708803
1970-01-01 00:00:00.200 -10.347759  2.108385

In [6]: fig = resampled.traja.plot()
```



1.11.3 Ramer–Douglas–Peucker algorithm

Note: Graciously yanked from Fabian Hirschmann's PyPI package `rdp`.

`rdp()` reduces the number of points in a line using the Ramer–Douglas–Peucker algorithm:

```
from traja.contrib import rdp

# Create dataframe of 1000 x, y coordinates
df = traja.generate(n=1000)

# Extract xy coordinates
xy = df.traja.xy

# Reduce points with epsilon between 0 and 1:
xy_ = rdp(xy, epsilon=0.8)


len(xy_)

Output:
```

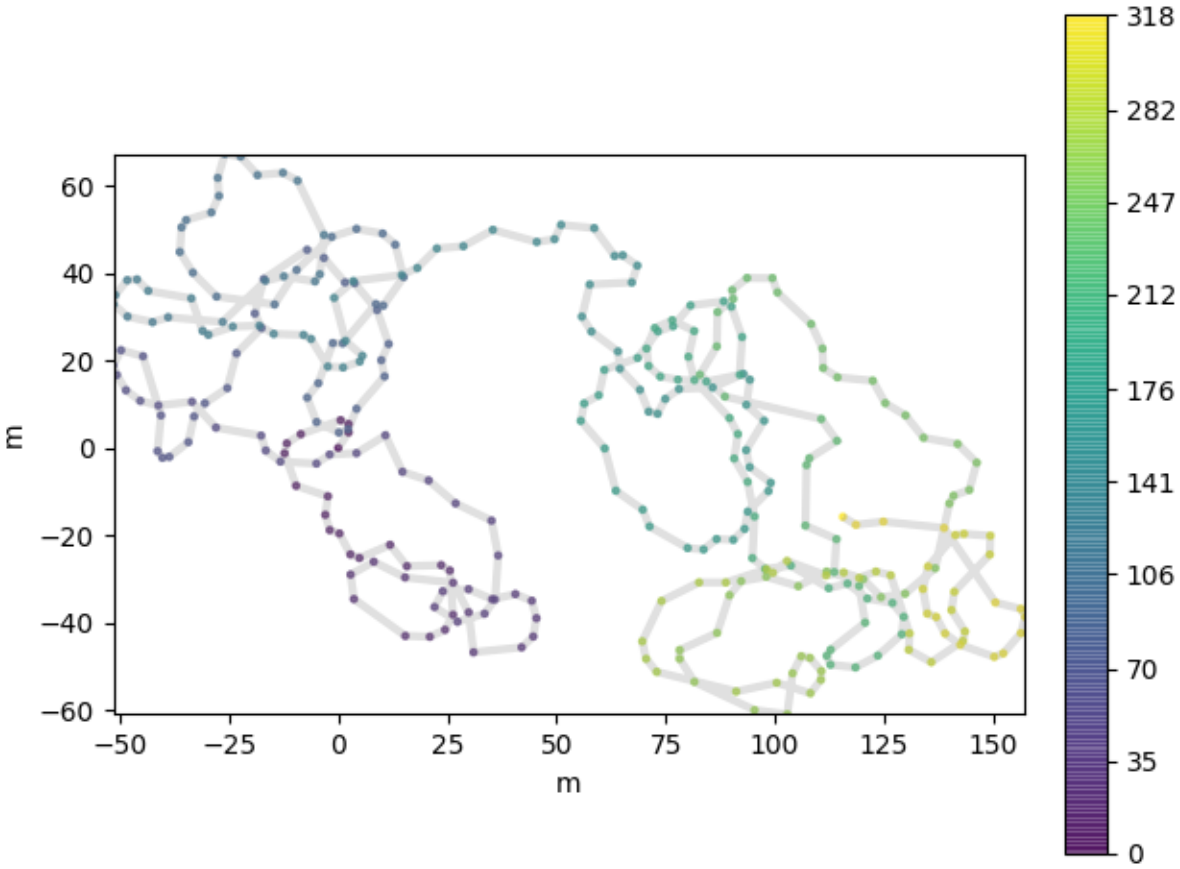
(continues on next page)

(continued from previous page)

```
317
```

Plotting, we can now see the many fewer points are needed to cover a similar area.:

```
df = traja.from_xy(xy_)
df.traja.plot()
```



1.12 Clustering and Dimensionality Reduction

1.12.1 Clustering Trajectories

Trajectories can be clustered using *traja.plotting.plot_clustermap()*.

Colors corresponding to each trajectory can be specified with the `colors` argument.

`traja.plotting.`**`plot_clustermap`**(*displacements: List[Series]*, *rule: Optional[str] = None*, *nr_steps=None*, *colors: Optional[List[Union[int, str]]] = None*, ***kwargs*)

Plot cluster map / dendrogram of trajectories with DatetimeIndex.

Parameters

- **`displacements`** – list of pd.Series, outputs of `traja.calc_displacement()`

- **rule** – how to resample series, eg '30s' for 30-seconds
- **nr_steps** – select first N samples for clustering
- **colors** – list of colors (eg, 'b','r') to map to each trajectory
- **kwargs** – keyword arguments for `seaborn.clustermap()`

Returns
a `seaborn.matrix.ClusterGrid()` instance

Return type
cg

Note: Requires seaborn to be installed. Install it with 'pip install seaborn'.

1.12.2 PCA

Prinicipal component analysis can be used to cluster trajectories based on grid cell occupancy. PCA is computed by converting the trajectory to a trip grid (see *traja.plotting.trip_grid()*) followed by PCA (`sklearn.decomposition.PCA`).

`traja.plotting.`**plot_pca**(*trj: TrajaDataFrame*, *id_col: str = 'id'*, *bins: tuple = (8, 8)*, *three_dims: bool = False*, *ax=None*)

Plot PCA comparing animals ids by trip grids.

Parameters

- **Trajectory** (*trj -*) –
- **IDs** (*id_col - column representing animal*) –
- **grid** (*bins - shape for binning trajectory into a trip*) –
- **Default** (*three_dims - 3D plot.*) – False (2D plot)
- **axes** (*ax - Matplotlib*) –

Returns
fig - Figure

1.13 Trajectory Collections

1.13.1 TrajaCollection

When handling multiple trajectories, Traja allows plotting and analysis simultaneously.

Initialize a *TrajaCollection()* with a dictionary or `DataFrame` and `id_col`.

Initializing with Dictionary

The keys of the dictionary can be used to identify types of objects in the scene, eg, “bus”, “car”, “person”:

```
dfs = {"car0":df0, "car1":df1, "bus0: df2, "person0": df3}
```

Or, arbitrary numbers can be used to initialize

class traja.frame.**TrajaCollection**(*trjs: Union[TrajaDataFrame, DataFrame, dict]*, *id_col: Optional[str] = None*, ***kwargs*)

Collection of trajectories.

Initializing with a DataFrame

A dataframe containing an id column can be passed directly to *TrajaCollection()*, as long as the `id_col` is specified:

```
trjs = TrajaCollection(df, id_col="id")
```

1.13.2 Grouped Operations

Operations can be applied to each trajectory with *apply_all()*.

TrajaCollection.**apply_all**(*method*, ***kwargs*)

Applies method to all trajectories

Parameters
method –

Returns

dataframe or series

```
>>> trjs = {ind: traja.generate(seed=ind) for ind in range(3)}
>>> coll = traja.TrajaCollection(trjs)
>>> angles = coll.apply_all(traja.calc_angle)
```

1.13.3 Plottting Multiple Trajectories

Plotting multiple trajectories can be achieved with *plot()*.

TrajaCollection.**plot**(*colors=None*, ***kwargs*)

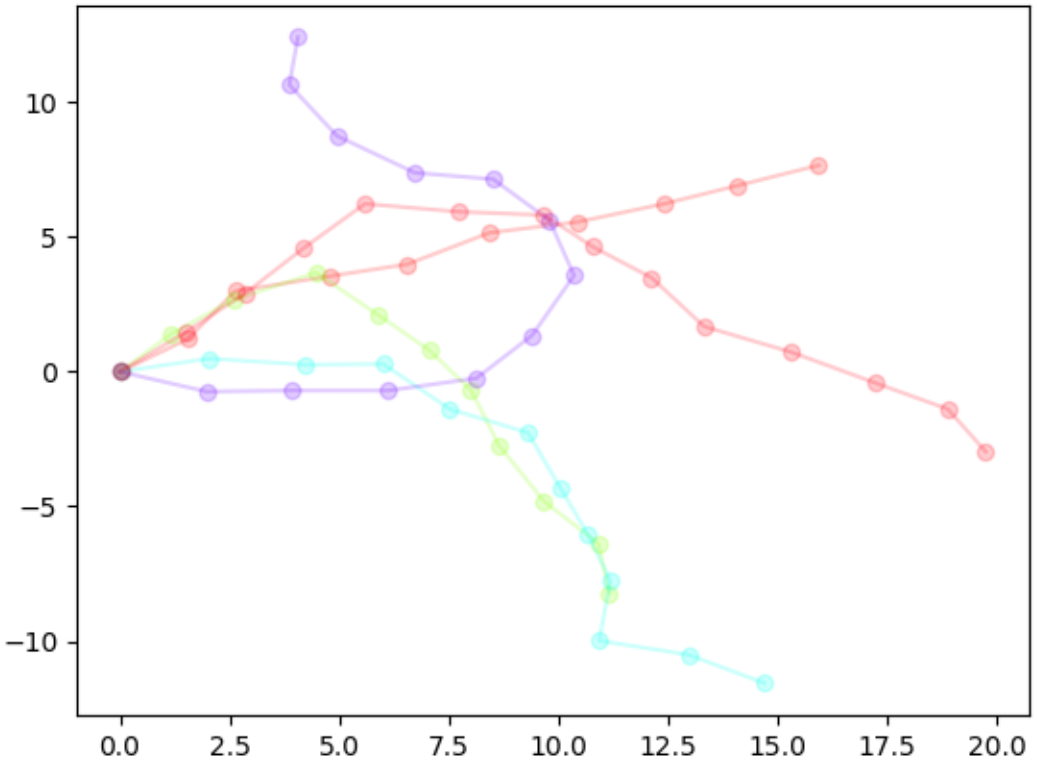
Plot collection of trajectories with colors assigned to each id.

```
>>> trjs = {ind: traja.generate(seed=ind) for ind in range(3)}
>>> coll = traja.TrajaCollection(trjs)
>>> coll.plot()
```

Colors can be specified for ids by supplying `colors` with a lookup dictionary:

or with a substring lookup:

colors = [“car”:”red”,
“bus”:”orange”, “12”:”red”, “13”:”orange”, “14”:”orange”]



1.14 Predicting Trajectories

Predicting trajectories with traja can be done with a recurrent neural network (RNN). Traja includes the Long Short Term Memory (LSTM), LSTM Autoencoder (LSTM AE) and LSTM Variational Autoencoder (LSTM VAE) RNNs. Traja also supports custom RNNs.

To model a trajectory using RNNs, one needs to fit the network to the model. Traja includes the MultiTaskRNNTrainer that can solve a prediction, classification and regression problem with traja DataFrames.

Traja also includes a DataLoader that handles traja dataframes.

Below is an example with a prediction LSTM via *traja.models.predictive_models.lstm.LSTM*.

```
import traja

df = traja.dataset.example.jaguar()
```

Note: LSTMs work better with data between -1 and 1. Therefore the data loader scales the data.

```
batch_size = 10 # How many sequences to train every step. Constrained by GPU memory.
num_past = 10 # How many time steps from which to learn the time series
```

(continues on next page)

(continued from previous page)

```
num_future = 10 # How many time steps to predict
split_by_id = False # Whether to split data into training, test and validation sets␣
↪based on
                    # the animal's ID or not. If True, an animal's entire trajectory will␣
↪only
                    # be used for training, or only for testing and so on.
                    # If your animals are territorial (like Jaguars) and you want to␣
↪forecast
                    # their trajectories, you want this to be false. If, however, you␣
↪want to
                    # classify the group membership of an animal, you want this to be␣
↪true,
                    # so that you can verify that previously unseen animals get assigned␣
↪to
                    # the correct class.
```

class traja.models.predictive_models.lstm.**LSTM**(*input_size: int*, *hidden_size: int*, *output_size: int*, *num_future: int = 8*, *batch_size: int = 8*, *num_layers: int = 1*, *reset_state: bool = True*, *bidirectional: bool = False*, *dropout: float = 0*, *batch_first: bool = True*)

Deep LSTM network. This implementation returns output_size outputs. :param input_size: The number of expected features in the input `x` :param hidden_size: The number of features in the hidden state `h` :param output_size: The number of output dimensions :param batch_size: Size of batch. Default is 8 :param sequence_length: The number of in each sample :param num_layers: Number of recurrent layers. E.g., setting `num_layers=2`

would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1

Parameters

- **reset_state** – If `True`, will reset the hidden and cell state for each batch of data
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`

dataloaders = traja.dataset.MultiModalDataLoader(df,
batch_size=batch_size, n_past=num_past, n_future=num_future, num_workers=0, split_by_id=split_by_id)

forward(*x*, *training=True*, *classify=False*, *regress=False*, *latent=False*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Note: The width of the hidden layers and depth of the network are the two main ways in which one tunes the performance of the network. More complex datasets require wider and deeper networks. Below are sensible defaults.

```
from traja.models.predictive_models.lstm import LSTM
input_size = 2 # Number of input dimensions (normally x, y)
output_size = 2 # Same as input_size when predicting
num_layers = 2 # Number of LSTM layers. Deeper learns more complex patterns but overfits.
hidden_size = 32 # Width of layers. Wider learns bigger patterns but overfits. Try 32,␣
↪64, 128, 256, 512
dropout = 0.1 # Ignore some network connections. Improves generalisation.

model = LSTM(input_size=input_size,
             hidden_size=hidden_size,
             num_layers=num_layers,
             output_size=output_size,
             dropout=dropout,
             batch_size=batch_size)
```

```
from traja.models.train import HybridTrainer

optimizer_type = 'Adam' # Nonlinear optimiser with momentum
loss_type = 'huber'

# Trainer
trainer = HybridTrainer(model=model,
                        optimizer_type=optimizer_type,
                        loss_type=loss_type)
# Train the model
trainer.fit(dataloaders, model_save_path='./model.pt', epochs=10, training_mode=
↪'forecasting')
```

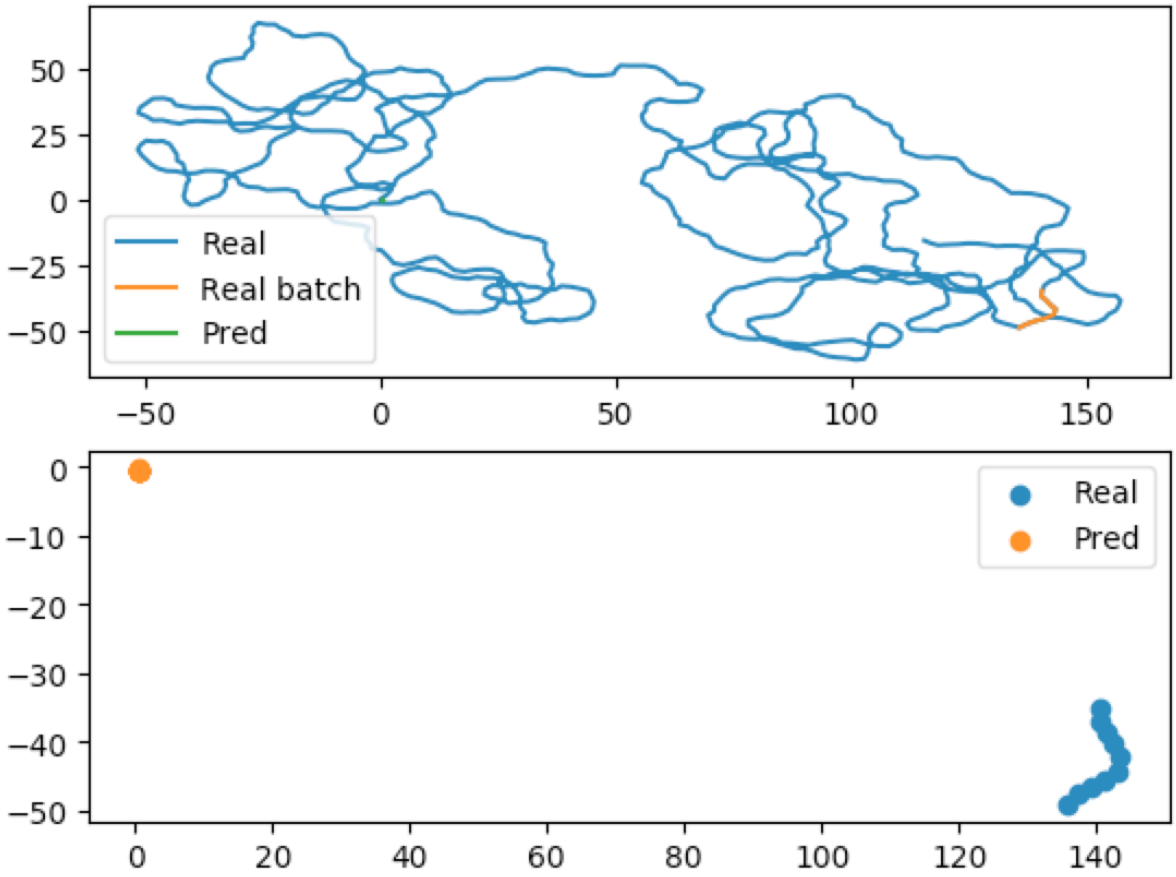
After training, you can determine the network's final performance with test data, if you want to pick the best model, or with validation data, if you want to determine the performance of your model.

The `dataloaders` dictionary contains the `sequential_test_loader` and `sequential_validation_loader`, that preserve the order of the original data. The dictionary also contains the 'test_loader' and `validation_loader` data loaders, where the order of the time series is randomised.

```
validation_loader = dataloaders['sequential_validation_loader']

trainer.validate(validation_loader)
```

Finally, you can display your training results using the built-in plotting libraries.

```
from traja.plotting import plot_prediction

batch_index = 0  # The batch you want to plot
plot_prediction(model, validation_loader, batch_index)
```



1.14.1 Parameter searching

When optimising neural networks, you often want to change the parameters. When training a forecaster, you have to reinitialise and retrain your model. However, when training a classifier or regressor, you can reset these on the fly, since they work directly on the latent space of your model. VAE models provide utility functions to make this easy.

```
from traja.models import MultiModelVAE
input_size = 2 # Number of input dimensions (normally x, y)
output_size = 2 # Same as input_size when predicting
num_layers = 2 # Number of LSTM layers. Deeper learns more complex patterns but overfits.
hidden_size = 32 # Width of layers. Wider learns bigger patterns but overfits. Try 32,␣
↪64, 128, 256, 512
dropout = 0.1 # Ignore some network connections. Improves generalisation.

# Classifier parameters
classifier_hidden_size = 32
num_classifier_layers = 4
num_classes = 42

# Regressor parameters
regressor_hidden_size = 18
```

(continues on next page)

(continued from previous page)

```
num_regressor_layers = 1
num_regressor_parameters = 3

model = MultiModelVAE(input_size=input_size,
                      hidden_size=hidden_size,
                      num_layers=num_layers,
                      output_size=output_size,
                      dropout=dropout,
                      batch_size=batch_size,
                      num_future=num_future,
                      classifier_hidden_size=classifier_hidden_size,
                      num_classifier_layers=num_classifier_layers,
                      num_classes=num_classes,
                      regressor_hidden_size=regressor_hidden_size,
                      num_regressor_layers=num_regressor_layers,
                      num_regressor_parameters=num_regressor_parameters)

new_classifier_hidden_size = 64
new_num_classifier_layers = 2

model.reset_classifier(classifier_hidden_size=new_classifier_hidden_size,
                       num_classifier_layers=new_num_classifier_layers)

new_regressor_hidden_size = 64
new_num_regressor_layers = 2
model.reset_regressor(regressor_hidden_size=new_regressor_hidden_size,
                      num_regressor_layers=new_num_regressor_layers)
```

1.15 Reference

1.15.1 Accessor Methods

The following methods are available via `traja.accessor.TrajaAccessor`:

class traja.accessor.**TrajaAccessor**(*pandas_obj*)

Accessor for pandas DataFrame with trajectory-specific numerical and analytical functions.

Access with *df.traja*.

property center

Return the center point of this trajectory.

property bounds

Return limits of x and y dimensions (`(xmin, xmax), (ymin, ymax)`).

night(*begin: str = '19:00'*, *end: str = '7:00'*)

Get nighttime dataset between *begin* and *end*.

Parameters

- **begin** (*str*) – (Default value = '19:00')
- **end** (*str*) – (Default value = '7:00')

Returns
Trajectory during night.

Return type
trj (TrajaDataFrame)

day(*begin: str = '7:00'*, *end: str = '19:00'*)

Get daytime dataset between *begin* and *end*.

Parameters

- **begin** (*str*) – (Default value = '7:00')
- **end** (*str*) – (Default value = '19:00')

Returns
Trajectory during day.

Return type
trj (TrajaDataFrame)

_get_time_col()

Returns time column in trajectory.

Args:

Returns
name of time column, 'index' or None

Return type
time_col (str or None)

between(*begin: str*, *end: str*)

Returns trajectory between *begin* and end` if *time* column is *datetime64*.

Parameters

- **begin** (*str*) – Beginning of time slice.
- **end** (*str*) – End of time slice.

Returns
Dataframe between values.

Return type
trj (TrajaDataFrame)

```
>>> s = pd.to_datetime(pd.Series(['Jun 30 2000 12:00:01', 'Jun 30 2000 12:00:02
↪', 'Jun 30 2000 12:00:03']))
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3],'time':s})
>>> df.traja.between('12:00:00','12:00:01')
                 time  x  y
0 2000-06-30 12:00:01  0  1
```

resample_time(*step_time: float*)

Returns trajectory resampled with `step_time`.

Parameters
step_time (*float*) – Step time

Returns
Dataframe resampled.

Return type
trj (TrajaDataFrame)

rediscretize_points(*R*, ***kwargs*)

Rediscretize points

trip_grid(*bins: Union[int, tuple] = 10*, *log: bool = False*, *spatial_units=None*, *normalize: bool = False*, *hist_only: bool = False*, *plot: bool = True*, ***kwargs*)

Returns a 2D histogram of trip.

Parameters

- **bins** (*int, optional*) – Number of bins (Default value = 16)
- **log** (*bool*) – log scale histogram (Default value = False)
- **spatial_units** (*str*) – units for plotting
- **normalize** (*bool*) – normalize histogram into density plot
- **hist_only** (*bool*) – return histogram without plotting

Returns
2D histogram as array image (matplotlib.collections.PathCollection: image of histogram

Return type
hist (numpy.ndarray)

plot(*n_coords: Optional[int] = None*, *show_time=False*, ***kwargs*)

Plot trajectory over period.

Parameters

- **n_coords** (*int*) – Number of coordinates to plot
- ****kwargs** – additional keyword arguments to matplotlib.axes.Axes.scatter()

Returns
Axes of plot

Return type
ax (Axes)

plot_3d(***kwargs*)

Plot 3D trajectory for single identity over period.

Args: trj (traja.TrajaDataFrame): trajectory n_coords (int, optional): Number of coordinates to plot **kwargs: additional keyword arguments to matplotlib.axes.Axes.scatter()

Returns
collection that was plotted

Return type
collection (PathCollection)

Note: Takes a while to plot large trajectories. Consider using first:

```
rt = trj.traja.rediscretize(R=1.) # Replace R with appropriate step length
rt.traja.plot_3d()
```

---

plot_flow(*kind='quiver'*, ***kwargs*)

Plot grid cell flow.

Parameters

- **kind** (*str*) – Kind of plot (eg, 'quiver','surface','contour','contourf','stream')
- ****kwargs** – additional keyword arguments to `matplotlib.axes.Axes.scatter()`

Returns
Axes of plot

Return type
ax (`Axes`)

plot_collection(*colors=None*, ***kwargs*)

apply_all(*method*, *id_col=None*, ***kwargs*)

Applies method to all trajectories and returns grouped dataframes or series

property xy

Returns a `numpy.ndarray` of x,y coordinates.

Parameters
split (*bool*) – Split into seaprate x and y `numpy.ndarrays`

Returns
xy (`numpy.ndarray`) – x,y coordinates (separate if *split* is *True*)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.xy
array([[0, 1],
       [1, 2],
       [2, 3]])
```

_check_has_time()

Check for presence of displacement time column.

__getattr__(*name*)

Catch all method calls which are not defined and forward to modules.

transitions(**args*, ***kwargs*)

Calculate transition matrix

calc_derivatives(*assign: bool = False*)

Returns derivatives *displacement* and *displacement_time*.

Parameters
assign (*bool*) – Assign output to `TrajaDataFrame` (Default value = False)

Returns
Derivatives.

Return type
derivs (`OrderedDict`)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3],'time':[0., 0.2, 0.4]})
>>> df.traja.calc_derivatives()
   displacement  displacement_time
0           NaN                0.0
```

(continues on next page)

(continued from previous page)

```
1       1.414214                0.2
2       1.414214                0.4
```

get_derivatives() → DataFrame

Returns derivatives as DataFrame.

speed_intervals(*faster_than: Optional[Union[float, int]] = None*, *slower_than: Optional[Union[float, int]] = None*)

Returns `TrajaDataFrame` with speed time intervals.

Returns a dataframe of time intervals where speed is slower and/or faster than specified values.

Parameters

- **faster_than** (*float, optional*) – Minimum speed threshold. (Default value = None)
- **slower_than** (*float or int, optional*) – Maximum speed threshold. (Default value = None)

Returns

result (`DataFrame`) – time intervals as dataframe

Note: Implementation ported to Python, heavily inspired by Jim McLean's trajr package.

to_shapely()

Returns shapely object for area, bounds, etc. functions.

Args:

Returns

Shapely shape.

Return type

shape (shapely.geometry.linestring.LineString)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> shape = df.traja.to_shapely()
>>> shape.is_closed
False
```

calc_displacement(*assign: bool = True*) → Series

Returns `Series` of *float* with displacement between consecutive indices.

Parameters

assign (*bool, optional*) – Assign displacement to TrajaAccessor (Default value = True)

Returns

Displacement series.

Return type

displacement (`pandas.Series`)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.calc_displacement()
0         NaN
1    1.414214
```

(continues on next page)

(continued from previous page)

```
2    1.414214
Name: displacement, dtype: float64
```

calc_angle(*assign: bool = True*) → Series

Returns `Series` with angle between steps as a function of displacement with regard to x axis.

Parameters
assign (*bool, optional*) – Assign turn angle to TrajaAccessor (Default value = True)

Returns
Angle series.

Return type
angle (`pandas.Series`)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.calc_angle()
0     NaN
1    45.0
2    45.0
dtype: float64
```

scale(*scale: float*, *spatial_units: str = 'm'*)

Scale trajectory when converting, eg, from pixels to meters.

Parameters

- **scale** (*float*) – Scale to convert coordinates
- **spatial_units** (*str., optional*) – Spatial units (eg, 'm') (Default value = "m")

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.scale(0.1)
>>> df
     x    y
0  0.0  0.1
1  0.1  0.2
2  0.2  0.3
```

rediscretize(*R: float*)

Resample a trajectory to a constant step length. R is rediscretized step length.

Parameters
R (*float*) – Rediscretized step length (eg, 0.02)

Returns
rediscretized trajectory

Return type
rt (`traja.TrajaDataFrame`)

Note: Based on the appendix in Bovet and Benhamou, (1988) and Jim McLean's trajr implementation.

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.rediscretize(1.)
```

(continues on next page)

(continued from previous page)

```
          x         y
0  0.000000  1.000000
1  0.707107  1.707107
2  1.414214  2.414214
```

grid_coordinates(***kwargs*)

calc_heading(*assign: bool = True*)

Calculate trajectory heading.

Parameters
assign (*bool*) – (Default value = True)

Returns
heading as a Series

Return type
heading (pandas.Series)

..doctest:

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.calc_heading()
0     NaN
1    45.0
2    45.0
Name: heading, dtype: float64
```

calc_turn_angle(*assign: bool = True*)

Calculate turn angle.

Parameters
assign (*bool*) – (Default value = True)

Returns
Turn angle

Return type
turn_angle (Series)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> df.traja.calc_turn_angle()
0    NaN
1    NaN
2    0.0
Name: turn_angle, dtype: float64
```

1.15.2 Plotting functions

The following methods are available via *traja.plotting*:

plotting.**animate**(*polar: bool = True*, *save: bool = False*)

Animate trajectory.

Parameters

- **polar** (*bool*) – include polar bar chart with turn angle
- **save** (*bool*) – save video to `trajectory.mp4`

Returns
animation

Return type
anim (matplotlib.animation.FuncAnimation)

plotting.**bar_plot**(*bins: Optional[Union[int, tuple]] = None*, ***kwargs*) → Axes

Plot trajectory for single animal over period.

Parameters

- **trj** (`traja.TrajaDataFrame`) – trajectory
- **bins** (*int or tuple*) – number of bins for x and y
- ****kwargs** – additional keyword arguments to `mpl_toolkits.mplot3d.Axed3D.plot()`

Returns
Axes of plot

Return type
ax (`PathCollection`)

plotting.**color_dark**(*ax: Optional[Axes] = None*, *start: int = 19*, *end: int = 7*)

Color dark phase in plot. :param series: :type series: pd.Series :param ax (: class: *~matplotlib.axes.Axes*): axis to plot on (eg, *plt.gca()*) :param start: start of dark period/night :type start: int :param end: end of dark period/day :type end: hour

Returns
Axes of plot

Return type
ax (`AxesSubplot`)

plotting.**fill_ci**(*window: Union[int, str]*) → Figure

Fill confidence interval defined by SEM over mean of *window*. Window can be interval or offset, eg, '30s'.

plotting.**find_runs**() -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>)

Find runs of consecutive items in an array. From https://gist.github.com/alimanfoo/c5977e87111abe8127453b21204c1065.

plotting.**plot**(*n_coords: Optional[int] = None*, *show_time: bool = False*, *accessor: Optional[TrajaAccessor] = None*, *ax=None*, ***kwargs*) → PathCollection

Plot trajectory for single animal over period.

Parameters

- **trj** (`traja.TrajaDataFrame`) – trajectory
- **n_coords** (*int, optional*) – Number of coordinates to plot

- **show_time** (*bool*) – Show colormap as time
- **accessor** (TrajaAccessor, optional) – TrajaAccessor instance
- **ax** (Axes) – axes for plotting
- **interactive** (*bool*) – show plot immediately
- ****kwargs** – additional keyword arguments to matplotlib.axes.Axes.scatter()

Returns
collection that was plotted

Return type
collection (PathCollection)

plotting.**plot_3d**(***kwargs*) → PathCollection

Plot 3D trajectory for single identity over period.

Parameters

- **trj** (traja.TrajaDataFrame) – trajectory
- **n_coords** (*int, optional*) – Number of coordinates to plot
- ****kwargs** – additional keyword arguments to matplotlib.axes.Axes.scatter()

Returns
Axes of plot

Return type
ax (PathCollection)

Note: Takes a while to plot large trajectories. Consider using first:

```
rt = trj.traja.rediscretize(R=1.) # Replace R with appropriate step length
rt.traja.plot_3d()
```

---

plotting.**plot_actogram**(*dark=(19, 7)*, *ax: Optional[Axes] = None*, ***kwargs*)

Plot activity or displacement as an actogram.

Note: For published example see Eckel-Mahan K, Sassone-Corsi P. Phenotyping Circadian Rhythms in Mice. Curr Protoc Mouse Biol. 2015;5(3):271-281. Published 2015 Sep 1. doi:10.1002/9780470942390.mo140229

---

plotting.**plot_autocorrelation**(*coord: str = 'y'*, *unit: str = 'Days'*, *xmax: int = 1000*, *interactive: bool = True*)

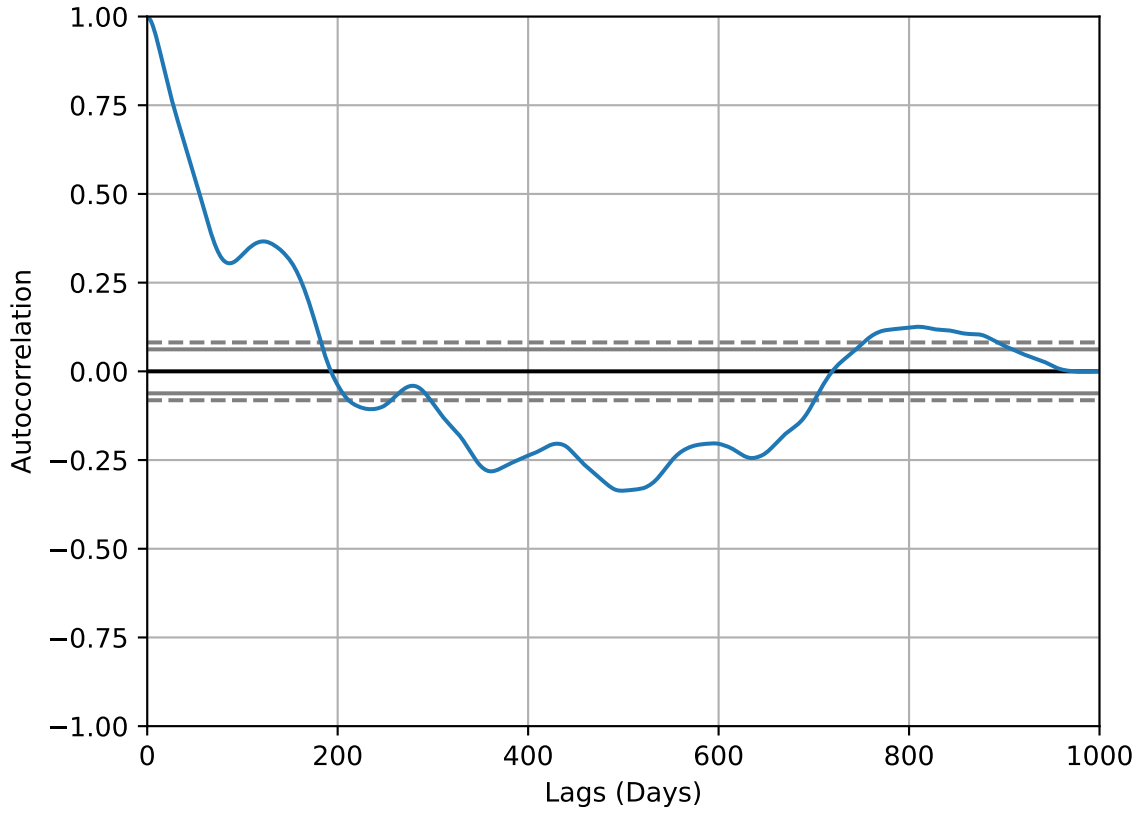
Plot autocorrelation of given coordinate.

Parameters

- **Trajectory** (*trj -*) –
- **'y'** (*coord - 'x' or*) –
- **string** (*unit -*) –
- **eg** –
- **'Days'** –
- **value** (*xmax - max xaxis*) –

- **immediately** (*interactive - Plot*) –

Returns
Matplotlib Figure



Note: Convenience wrapper for pandas `autocorrelation_plot()`.

`plotting.`**`plot_contour`**(*bins: Optional[Union[int, tuple]] = None*, *filled: bool = True*, *quiver: bool = True*, *contourplot_kws: dict = {}*, *contourfplot_kws: dict = {}*, *quiverplot_kws: dict = {}*, *ax: Optional[Axes] = None*, ***kwargs*) → Axes

Plot average flow from each grid cell to neighbor.

Parameters

- **trj** – Traja DataFrame
- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **filled** (*bool*) – Contours filled
- **quiver** (*bool*) – Quiver plot
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`

- **quiverplot_kws** – Additional keyword arguments for `quiver()`
- **ax** (*optional*) – Matplotlib Axes

Returns
Axes of quiver plot

Return type
ax (`Axes`)

plotting.**plot_clustermap**(*rule: Optional[str] = None*, *nr_steps=None*, *colors: Optional[List[Union[int, str]]] = None*, ***kwargs*)

Plot cluster map / dendrogram of trajectories with DatetimeIndex.

Parameters

- **displacements** – list of pd.Series, outputs of `traja.calc_displacement()`
- **rule** – how to resample series, eg '30s' for 30-seconds
- **nr_steps** – select first N samples for clustering
- **colors** – list of colors (eg, 'b','r') to map to each trajectory
- **kwargs** – keyword arguments for `seaborn.clustermap()`

Returns
a `seaborn.matrix.ClusterGrid()` instance

Return type
cg

Note: Requires seaborn to be installed. Install it with 'pip install seaborn'.

---

plotting.**plot_flow**(*kind: str = 'quiver'*, **args*, *contourplot_kws: dict = {}*, *contourfplot_kws: dict = {}*, *streamplot_kws: dict = {}*, *quiverplot_kws: dict = {}*, *surfaceplot_kws: dict = {}*, ***kwargs*) → Figure

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **kind** (*str*) – Choice of 'quiver','contourf','stream','surface'. Default is 'quiver'.
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`
- **streamplot_kws** – Additional keyword arguments for `streamplot()`
- **quiverplot_kws** – Additional keyword arguments for `quiver()`
- **surfaceplot_kws** – Additional keyword arguments for `plot_surface()`

Returns
Axes of plot

Return type
ax (`Axes`)

plotting.**plot_quiver**(*bins: Optional[Union[int, tuple]] = None*, *quiverplot_kws: dict = {}*, ***kwargs*) → Axes

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **quiverplot_kws** – Additional keyword arguments for `quiver()`

Returns
Axes of quiver plot

Return type
ax (`Axes`)

plotting.**plot_stream**(*bins: Optional[Union[int, tuple]] = None*, *cmap: str = 'viridis'*, *contourfplot_kws: dict = {}*, *contourplot_kws: dict = {}*, *streamplot_kws: dict = {}*, ***kwargs*) → Figure

Plot average flow from each grid cell to neighbor.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **contourplot_kws** – Additional keyword arguments for `contour()`
- **contourfplot_kws** – Additional keyword arguments for `contourf()`
- **streamplot_kws** – Additional keyword arguments for `streamplot()`

Returns
Axes of stream plot

Return type
ax (`Axes`)

plotting.**plot_surface**(*bins: Optional[Union[int, tuple]] = None*, *cmap: str = 'viridis'*, ***surfaceplot_kws: dict*) → Figure

Plot surface of flow from each grid cell to neighbor in 3D.

Parameters

- **bins** (*int or tuple*) – Tuple of x,y bin counts; if *bins* is int, bin count of x, with y inferred from aspect ratio
- **cmap** (*str*) – color map
- **surfaceplot_kws** – Additional keyword arguments for `plot_surface()`

Returns
Axes of quiver plot

Return type
ax (`Axes`)

plotting.**plot_transition_matrix**(*interactive=True*, ***kwargs*) → AxesImage

Plot transition matrix.

Parameters

- **data** (*trajectory or square transition matrix*) –
- **interactive** (*bool*) – show plot

- **kwargs** – kwargs to `traja.grid_coordinates()`

Returns
axesimage (matplotlib.image.AxesImage)

`plotting.`**plot_xy**(**args: Optional*, ***kwargs: Optional*)

Plot trajectory from xy values.

Parameters

- **xy** (*np.ndarray*) – xy values of dimensions N x 2
- ***args** – Plot args
- ****kwargs** – Plot kwargs

`plotting.`**polar_bar**(*feature: str = 'turn_angle'*, *bin_size: int = 2*, *threshold: float = 0.001*, *overlap: bool = True*, *ax: Optional[Axes] = None*, ***plot_kws: str*) → Axes

Plot polar bar chart.

Parameters

- **trj** (`traja.TrajaDataFrame`) – trajectory
- **feature** (*str*) – Options: 'turn_angle', 'heading'
- **bin_size** (*int*) – width of bins
- **threshold** (*float*) – filter for step distance
- **overlap** (*bool*) – Overlapping shows all values, if set to false is a histogram

Returns
Axes of plot

Return type
ax (`PathCollection`)

`plotting.`**plot_prediction**(*dataloader*, *index*, *scaler=None*)

`plotting.`**sans_serif**()

Convenience function for changing plot text to serif font.

`plotting.`**stylize_axes**()

Add top and right border to plot, set ticks.

`plotting.`**trip_grid**(*bins: Union[tuple, int] = 10*, *log: bool = False*, *spatial_units: Optional[str] = None*, *normalize: bool = False*, *hist_only: bool = False*, ***kwargs*) → Tuple[ndarray, PathCollection]

Generate a heatmap of time spent by point-to-cell gridding.

Parameters

- **bins** (*int, optional*) – Number of bins (Default value = 10)
- **log** (*bool*) – log scale histogram (Default value = False)
- **spatial_units** (*str*) – units for plotting
- **normalize** (*bool*) – normalize histogram into density plot
- **hist_only** (*bool*) – return histogram without plotting

Returns
2D histogram as array image (`matplotlib.collections.PathCollection`: image of histogram

Return type
hist (numpy.ndarray)

1.15.3 Analysis

The following methods are available via `traja.trajectory`:

`trajectory.calc_angle(unit: str = 'degrees', lag: int = 1)`

Returns a `Series` with angle between steps as a function of displacement with regard to x axis.

Parameters

- **trj** (`TrajaDataFrame`) – Trajectory
- **unit** (*str*) – return angle in radians or degrees (Default value: 'degrees')
- **lag** (*int*) – time steps between angle calculation (Default value: 1)

Returns
Angle series.

Return type
angle (pandas.Series)

`trajectory.calc_convex_hull() → array`

Identify containing polygonal convex hull for full Trajectory Interior points filtered with *traja.trajectory.inside()* method, takes quadrilateral using extrema points *(minx, maxx, miny, maxy)* - convex hull points MUST all be outside such a polygon. Returns an array with all points in the convex hull.

Implementation of Graham Scan *technique <https://en.wikipedia.org/wiki/Graham_scan>_*.

Returns
n x 2 (x,y) array

Return type
point_arr (ndarray)

```
>> #Quick visualizaation
>> import matplotlib.pyplot as plt
>> df = traja.generate(n=10000, convex_hull=True)
>> xs, ys = [*zip(*df.convex_hull)]
>> _ = plt.plot(df.x.values, df.y.values, 'o', 'blue')
>> _ = plt.plot(xs, ys, '-o', color='red')
>> _ = plt.show()
```

Note: Incorporates Akl-Toussaint method for filtering interior points.

Note: Performative loss beyond ~100,000-200,000 points, algorithm has O(nlogn) complexity.

`trajectory.calc_derivatives()`

Returns derivatives `displacement` and `displacement_time` as DataFrame.

Parameters
trj (`TrajaDataFrame`) – Trajectory

Returns
Derivatives.

Return type
derivs (DataFrame)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3],'time':[0., 0.2, 0.4]})
>>> traja.calc_derivatives(df)
   displacement  displacement_time
0           NaN                0.0
1      1.414214                0.2
2      1.414214                0.4
```

trajectory.**calc_displacement**(*lag=1*)

Returns a Series of float displacement between consecutive indices.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **lag** (*int*) – time steps between displacement calculation

Returns
Displacement series.

Return type
displacement (pandas.Series)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> traja.calc_displacement(df)
0         NaN
1    1.414214
2    1.414214
Name: displacement, dtype: float64
```

trajectory.**calc_heading**()

Calculate trajectory heading.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
heading as a Series

Return type
heading (pandas.Series)

..doctest:

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> traja.calc_heading(df)
0     NaN
1    45.0
2    45.0
Name: heading, dtype: float64
```

trajectory.**calc_turn_angle**()

Return a Series of floats with turn angles.

Parameters
trj (traja.frame.TrajaDataFrame) – Trajectory

Returns
Turn angle

Return type
turn_angle (Series)

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> traja.calc_turn_angle(df)
0    NaN
1    NaN
2    0.0
Name: turn_angle, dtype: float64
```

trajectory.**calc_flow_angles**()

Calculate average flow between grid indices.

trajectory.**cartesian_to_polar**() -> (<class 'float'>, <class 'float'>)

Convert numpy.ndarray **xy** to polar coordinates **r** and **theta**.

Parameters
xy (numpy.ndarray) – x,y coordinates

Returns
step-length and angle

Return type
r, theta (tuple of float)

trajectory.**coords_to_flow**(*bins: Optional[Union[int, tuple]] = None*)

Calculate grid cell flow from trajectory.

Parameters

- **trj** (*trajectory*) –
- **bins** (*int or tuple*) –

Returns
X coordinates of arrow locations Y (ndarray): Y coordinates of arrow locations U (ndarray): X component of vector dataset V (ndarray): Y component of vector dataset

Return type
X (ndarray)

trajectory.**determine_colinearity**(*p1: ndarray*, *p2: ndarray*)

Determine whether trio of points constitute a right turn, or whether they are left turns (or colinear/straight line).

Parameters

- **p0** (ndarray) – First point [x,y] in line
- **p1** (ndarray) – Second point [x,y] in line
- **p2** (ndarray) – Third point [x,y] in line

Returns
(bool)

trajectory.**distance_between**(*B: TrajaDataFrame*, *method='dtw'*)

Returns distance between two trajectories.

Parameters

- **A** (TrajaDataFrame) – Trajectory 1
- **B** (TrajaDataFrame) – Trajectory 2
- **method** (*str*) – dtw for dynamic time warping, hausdorff for Hausdorff

Returns
Distance

Return type
distance (float)

trajectory.**distance**() → float

Calculates the distance from start to end of trajectory, also called net distance, displacement, or bee-line from start to finish.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
distance (float)

```
>> df = traja.generate()
>> traja.distance(df)
117.01507823153617
```

trajectory.**euclidean**(*v*, *w=None*)

Computes the Euclidean distance between two 1-D arrays.

The Euclidean distance between 1-D arrays *u* and *v*, is defined as

$$\frac{||u-v||_2}{\left(\sum{(w_i|(u_i-v_i)|^2)}\right)^{1/2}}$$

Parameters

- **u** (*(N,) array_like*) – Input array.
- **v** (*(N,) array_like*) – Input array.
- **w** (*(N,) array_like, optional*) – The weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0

Returns
euclidean – The Euclidean distance between vectors *u* and *v*.

Return type
double

Examples

```
>>> from scipy.spatial import distance
>>> distance.euclidean([1, 0, 0], [0, 1, 0])
1.4142135623730951
>>> distance.euclidean([1, 1, 0], [0, 1, 0])
1.0
```

trajectory.**expected_sq_displacement**(*n: int = 0*, *eqn1: bool = True*) → float

Expected displacement.

Note: This method is experimental and needs testing.

---

trajectory.**fill_in_traj**()

trajectory.**from_xy**()

Convenience function for initializing `TrajaDataFrame` with x,y coordinates.

Parameters
xy (`numpy.ndarray`) – x,y coordinates

Returns
Trajectory as dataframe

Return type
traj_df (`TrajaDataFrame`)

```
>>> import numpy as np
>>> xy = np.array([[0,1],[1,2],[2,3]])
>>> traja.from_xy(xy)
   x  y
0  0  1
1  1  2
2  2  3
```

trajectory.**generate**(*random: bool = True*, *step_length: int = 2*, *angular_error_sd: float = 0.5*, *angular_error_dist: Optional[Callable] = None*, *linear_error_sd: float = 0.2*, *linear_error_dist: Optional[Callable] = None*, *fps: float = 50*, *spatial_units: str = 'm'*, *seed: Optional[int] = None*, *convex_hull: bool = False*, ***kwargs*)

Generates a trajectory.

If `random` is `True`, the trajectory will be a correlated random walk/idiothetic directed walk (Kareiva & Shigesada, 1983), corresponding to an animal navigating without a compass (Cheung, Zhang, Stricker, & Srinivasan, 2008). If `random` is `False`, it will be(np.ndarray) a directed walk/allothetic directed walk/oriented path, corresponding to an animal navigating with a compass (Cheung, Zhang, Stricker, & Srinivasan, 2007, 2008).

By default, for both random and directed walks, errors are normally distributed, unbiased, and independent of each other, so are **simple directed walks** in the terminology of Cheung, Zhang, Stricker, & Srinivasan, (2008). This behaviour may be modified by specifying alternative values for the `angular_error_dist` and/or `linear_error_dist` parameters.

The initial angle (for a random walk) or the intended direction (for a directed walk) is `0` radians. The starting position is `(0, 0)`.

Parameters

- **n** (*int*) – (Default value = 1000)
- **random** (*bool*) – (Default value = True)
- **step_length** – (Default value = 2)
- **angular_error_sd** (*float*) – (Default value = 0.5)
- **angular_error_dist** (*Callable*) – (Default value = None)
- **linear_error_sd** (*float*) – (Default value = 0.2)
- **linear_error_dist** (*Callable*) – (Default value = None)
- **fps** (*float*) – (Default value = 50)
- **convex_hull** (*bool*) – (Default value = False)
- **spatial_units** – (Default value = 'm')
- ****kwargs** – Additional arguments

Returns
Trajectory

Return type
trj (`traja.frame.TrajaDataFrame`)

Note: Based on Jim McLean's trajr, ported to Python.

Reference: McLean, D. J., & Skowron Volponi, M. A. (2018). trajr: An R package for characterisation of animal trajectories. Ethology, 124(6), 440-448. https://doi.org/10.1111/eth.12739.

`trajectory.get_derivatives()`

Returns derivatives `displacement`, `displacement_time`, `speed`, `speed_times`, `acceleration`, `acceleration_times` as dictionary.

Parameters
trj (`TrajaDataFrame`) – Trajectory

Returns
Derivatives

Return type
derivs (`DataFrame`)

```
>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3],'time':[0.,0.2,0.4]})
>> df.traja.get_derivatives()
   displacement  displacement_time     speed  speed_times  acceleration ␣
↪acceleration_times
0           NaN                0.0       NaN          NaN           NaN          ␣
↪    NaN
1      1.414214                0.2  7.071068          0.2           NaN          ␣
↪    NaN
2      1.414214                0.4  7.071068          0.4           0.0          ␣
↪    0.4
```

`trajectory.grid_coordinates(bins: Optional[Union[int, tuple]] = None, xlim: Optional[tuple] = None, ylim: Optional[tuple] = None, assign: bool = False)`

Returns `DataFrame` of trajectory discretized into 2D lattice grid coordinates. :param trj: Trajectory :type trj:

~`traja.frame.TrajaDataFrame` :param bins: :type bins: tuple or int :param xlim: :type xlim: tuple :param ylim: :type ylim: tuple :param assign: Return updated original dataframe :type assign: bool

Returns
Trajectory is assign=True otherwise pd.DataFrame

Return type
trj (*TrajaDataFrame `*)

`trajectory.`**`inside`**(*bounds_xs: list*, *bounds_ys: list*, *minx: float*, *maxx: float*, *miny: float*, *maxy: float*)

Determine whether point lies inside or outside of polygon formed by "extrema" points - minx, maxx, miny, maxy. Optimized to be run as broadcast function in numpy along axis.

Parameters

- **pt** (ndarray) – Point to test whether inside or outside polygon
- **bounds_xs** (*list or tuple*) – x-coordinates of polygon vertices, in sequence
- **bounds_ys** (*list or tuple*) – y-coordinates of polygon vertices, same sequence
- **minx** (*float*) – minimum x coordinate value
- **maxx** (*float*) – maximum x coordinate value
- **miny** (*float*) – minimum y coordinate value
- **maxy** (*float*) – maximum y coordinate value

Returns
(bool)

Note: Ported to Python from C implementation by W. Randolph Franklin (WRF): <https://wrf.ecse.rpi.edu/Research/Short_Notes/pnpoly.html>

Boolean return "True" for OUTSIDE polygon, meaning it is within subset of possible convex hull coordinates.

`trajectory.`**`length`**`()` → float

Calculates the cumulative length of a trajectory.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
length (float)

```
>> df = traja.generate()
>> traja.length(df)
2001.142339606066
```

`trajectory.`**`polar_to_z`**(*theta: float*) → complex

Converts polar coordinates `r` and `theta` to complex number `z`.

Parameters

- **r** (*float*) – step size
- **theta** (*float*) – angle

Returns
complex number z

Return type
z (complex)

trajectory.**rediscretize_points**(*R: Union[float, int]*, *time_out=False*)

Returns a TrajaDataFrame rediscretized to a constant step length *R*.

Parameters

- **trj** (traja.frame.TrajaDataFrame) – Trajectory
- **R** (*float*) – Rediscretized step length (eg, 0.02)
- **time_out** (*bool*) – Include time corresponding to time intervals in output

Returns
rediscretized trajectory

Return type
rt (numpy.ndarray)

trajectory.**resample_time**(*step_time: str*, *new_fps: Optional[bool] = None*)

Returns a TrajaDataFrame resampled to consistent *step_time* intervals.

step_time should be expressed as a number-time unit combination, eg "2S" for 2 seconds and "2100L" for 2100 milliseconds.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **step_time** (*str*) – step time interval / offset string (eg, '2S' (seconds), '50L' (milliseconds), '50N' (nanoseconds))
- **new_fps** (*bool, optional*) – new fps

Results:
trj (TrajaDataFrame): Trajectory

```
>>> from traja import generate, resample_time
>>> df = generate()
>>> resampled = resample_time(df, '50L') # 50 milliseconds
>>> resampled.head()
                                  x         y
time
1970-01-01 00:00:00.000   0.000000  0.000000
1970-01-01 00:00:00.050   0.919113  4.022971
1970-01-01 00:00:00.100  -1.298510  5.423373
1970-01-01 00:00:00.150  -6.057524  4.708803
1970-01-01 00:00:00.200 -10.347759  2.108385
```

trajectory.**return_angle_to_point**(*p0: ndarray*)

Calculate angle of points as coordinates in relation to each other. Designed to be broadcast across all trajectory points for a single origin point p0.

Parameters

- **p1** (*np.ndarray*) – Test point [x,y]
- **p0** (*np.ndarray*) – Origin/source point [x,y]

Returns
r (float)

trajectory.**rotate**(*angle: Union[float, int] = 0*, *origin: Optional[tuple] = None*)

Returns a TrajaDataFrame Rotate a trajectory *angle* in radians.

Parameters

- **trj** (traja.frame.TrajaDataFrame) – Trajectory
- **angle** (*float*) – angle in radians
- **origin** (*tuple. optional*) – rotate around point (x,y)

Returns
Trajectory

Return type
trj (traja.frame.TrajaDataFrame)

Note: Based on Lyle Scott's implementation.

---

trajectory.**smooth_sg**(*w: Optional[int] = None*, *p: int = 3*)

Returns DataFrame of trajectory after Savitzky-Golay filtering.

Parameters

- **trj** (TrajaDataFrame) – Trajectory
- **w** (*int*) – window size (Default value = None)
- **p** (*int*) – polynomial order (Default value = 3)

Returns
Trajectory

Return type
trj (TrajaDataFrame)

```
>> df = traja.generate()
>> traja.smooth_sg(df, w=101).head()
           x          y  time
0 -11.194803  12.312742  0.00
1 -10.236337  10.613720  0.02
2  -9.309282   8.954952  0.04
3  -8.412910   7.335925  0.06
4  -7.546492   5.756128  0.08
```

trajectory.**speed_intervals**(*faster_than: Optional[float] = None*, *slower_than: Optional[float] = None*) → DataFrame

Calculate speed time intervals.

Returns a dictionary of time intervals where speed is slower and/or faster than specified values.

Parameters

- **faster_than** (*float, optional*) – Minimum speed threshold. (Default value = None)
- **slower_than** (*float or int, optional*) – Maximum speed threshold. (Default value = None)

Returns
result (DataFrame) – time intervals as dataframe

Note: Implementation ported to Python, heavily inspired by Jim McLean's trajr package.

```
>> df = traja.generate()
>> intervals = traja.speed_intervals(df, faster_than=100)
>> intervals.head()
   start_frame  start_time  stop_frame  stop_time  duration
0            1        0.02           3       0.06      0.04
1            4        0.08           8       0.16      0.08
2           10        0.20          11       0.22      0.02
3           12        0.24          15       0.30      0.06
4           17        0.34          18       0.36      0.02
```

trajectory.**step_lengths**()

Length of the steps of trj.

Parameters
trj (TrajaDataFrame) – Trajectory

trajectory.**to_shapely**()

Returns shapely object for area, bounds, etc. functions.

Parameters
trj (TrajaDataFrame) – Trajectory

Returns
shapely.geometry.linestring.LineString – Shapely shape.

```
>>> df = traja.TrajaDataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> shape = traja.to_shapely(df)
>>> shape.is_closed
False
```

trajectory.**traj_from_coords**(*x_col=1*, *y_col=2*, *time_col: Optional[str] = None*, *fps: Union[float, int] = 4*, *spatial_units: str = 'm'*, *time_units: str = 's'*) → TrajaDataFrame

Create TrajaDataFrame from coordinates.

Parameters

- **track** – N x 2 numpy array or pandas DataFrame with x and y columns
- **x_col** – column index or x column name
- **y_col** – column index or y column name
- **time_col** – name of time column
- **fps** – Frames per seconds
- **spatial_units** – default m, optional
- **time_units** – default s, optional

Returns
TrajaDataFrame

Return type
trj

```
>> xy = np.random.random((1000, 2))
>> trj = traja.traj_from_coord(xy)
>> assert trj.shape == (1000,4) # columns x, y, time, dt
```

trajectory.**transition_matrix**()

Returns np.ndarray of Markov transition probability matrix for grid cell transitions.

Parameters
grid_indices1D (np.ndarray) –

Returns
M (numpy.ndarray)

trajectory.**transitions**(***kwargs*)

Get first-order Markov model for transitions between grid cells.

Parameters

- **trj** (*trajectory*) –
- **kwargs** – kwargs to traja.grid_coordinates()

1.15.4 io functions

The following methods are available via traja.parsers:

parsers.**read_file**(*id: Optional[str] = None*, *xcol: Optional[str] = None*, *ycol: Optional[str] = None*, *parse_dates: Union[str, bool] = False*, *xlim: Optional[tuple] = None*, *ylim: Optional[tuple] = None*, *spatial_units: str = 'm'*, *fps: Optional[float] = None*, ***kwargs*)

Convenience method wrapping pandas *read_csv* and initializing metadata.

Parameters

- **filepath** (*str*) – path to csv file with *x*, *y* and *time* (optional) columns
- **id** (*str*) – id for trajectory
- **xcol** (*str*) – name of column containing x coordinates
- **ycol** (*str*) – name of column containing y coordinates
- **parse_dates** (*Union[list,bool]*) – The behavior is as follows: - boolean. if True -> try parsing the index. - list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- **xlim** (*tuple*) – x limits (min,max) for plotting
- **ylim** (*tuple*) – y limits (min,max) for plotting
- **spatial_units** (*str*) – for plotting (eg, 'cm')
- **fps** (*float*) – for time calculations
- ****kwargs** – Additional arguments for pandas.read_csv().

Returns
Trajectory

Return type
traj_df (TrajaDataFrame)

parsers.**from_df**(*xcol=None*, *ycol=None*, *time_col=None*, ***kwargs*)

Returns a traja.frame.TrajaDataFrame from a pandas DataFrame.

Parameters

- **df** (pandas.DataFrame) – Trajectory as pandas DataFrame
- **xcol** (*str*) –
- **ycol** (*str*) –
- **timecol** (*str*) –

Returns

Trajectory

Return type

traj_df (TrajaDataFrame)

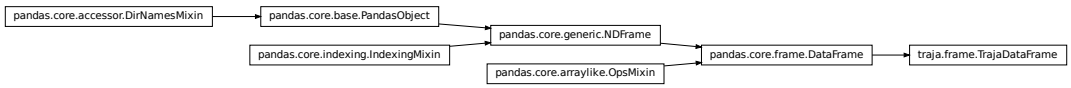
```
>>> df = pd.DataFrame({'x':[0,1,2],'y':[1,2,3]})
>>> traja.from_df(df)
   x  y
0  0  1
1  1  2
2  2  3
```

1.15.5 TrajaDataFrame

A TrajaDataFrame is a tabular data structure that contains x, y, and time columns.

All pandas DataFrame methods are also available, although they may not operate in a meaningful way on the x, y, and time columns.

Inheritance diagram:



1.15.6 TrajaCollection

A TrajaCollection holds multiple trajectories for analyzing and comparing trajectories. It has limited accessibility to lower-level methods.

class traja.frame.**TrajaCollection**(*trjs: Union[TrajaDataFrame, DataFrame, dict]*, *id_col: Optional[str] = None*, ***kwargs*)

Collection of trajectories.

TrajaCollection.**apply_all**(*method*, ***kwargs*)

Applies method to all trajectories

Parameters
method –

Returns

dataframe or series

```
>>> trjs = {ind: traja.generate(seed=ind) for ind in range(3)}
>>> coll = traja.TrajaCollection(trjs)
>>> angles = coll.apply_all(traja.calc_angle)
```

TrajaCollection.**plot**(*colors=None*, ***kwargs*)

Plot collection of trajectories with colors assigned to each id.

```
>>> trjs = {ind: traja.generate(seed=ind) for ind in range(3)}
>>> coll = traja.TrajaCollection(trjs)
>>> coll.plot()
```

1.15.7 API Pages

TrajaDataFrame(*args, **kwargs)	A TrajaDataFrame object is a subclass of pandas <~pandas.dataframe.DataFrame>.
TrajaCollection(trjs[, id_col])	Collection of trajectories.
read_file(filepath[, id, xcol, ycol, ...])	Convenience method wrapping pandas *read_csv* and initializing metadata.

traja.TrajaDataFrame

traja.TrajaCollection

traja.read_file

1.16 Support for Traja

1.16.1 Bugs

Bugs, issues and improvement requests can be logged in Github Issues.

1.16.2 Community

Community support is provided via Gitter. Just ask a question there.

1.17 Contributing to traja

(Contribution guidelines largely copied from geopandas)

1.17.1 Overview

Contributions to traja are very welcome. They are likely to be accepted more quickly if they follow these guidelines.

At this stage of traja development, the priorities are to define a simple, usable, and stable API and to have clean, maintainable, readable code. Performance matters, but not at the expense of those goals.

In general, traja follows the conventions of the pandas project where applicable.

In particular, when submitting a pull request:

- All existing tests should pass. Please make sure that the test suite passes, both locally and on Travis CI. Status on Travis will be visible on a pull request. If you want to enable Travis CI on your own fork, please read the pandas guidelines link above or the getting started docs.
- New functionality should include tests. Please write reasonable tests for your code and make sure that they pass on your pull request.
- Classes, methods, functions, etc. should have docstrings. The first line of a docstring should be a standalone summary. Parameters and return values should be ducumented explicitly.
- traja supports python 3 (3.6+). Use modern python idioms when possible.
- Follow PEP 8 when possible.
- Imports should be grouped with standard library imports first, 3rd-party libraries next, and traja imports third. Within each grouping, imports should be alphabetized. Always use absolute imports when possible, and explicit relative imports for local imports when necessary in tests.

Seven Steps for Contributing

There are seven basic steps to contributing to *traja*:

1) Fork the *traja* git repository
2) Create a development environment
3) Install *traja* dependencies
4) Make a `development` build of *traja*
5) Make changes to code and add tests
6) Update the documentation
7) Submit a Pull Request

Each of these 7 steps is detailed below.

1.17.2 1) Forking the *traja* repository using Git

To the new user, working with Git is one of the more daunting aspects of contributing to *traja**. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:

- Software Carpentry's Git Tutorial
- Atlassian
- the GitHub help pages.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the traja project page and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone git@github.com:your-user-name/traja.git traja-yourname
cd traja-yourname
git remote add upstream git://github.com/traja-team/traja.git
```

This creates the directory *traja-yourname* and connects your repository to the upstream (main project) *traja* repository.

The testing suite will run automatically on Travis-CI once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then Travis-CI needs to be hooked up to your GitHub repository. Instructions for doing so are here.

Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *traja*. You can have many shiny-new-features and switch in between them using the git checkout command.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the latest traja git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating.

1.17.3 2) Creating a development environment

A development environment is a virtual space where you can keep an independent installation of *traja*. This makes it easy to keep both a stable version of python in one place you use for work, and a development version (which you may break while playing with code) in another.

An easy way to create a *traja* development environment is as follows:

- Install either Anaconda or miniconda
- Make sure that you have *cloned the repository*
- `cd` to the *traja** source directory

Tell conda to create a new environment, named `traja_dev`, or any other name you would like for this environment, by running:

```
conda create -n traja_dev
```

For a python 3 environment:

```
conda create -n traja_dev python=3.8
```

This will create the new environment, and not touch any of your existing environments, nor any existing python installation.

To work in this environment, Windows users should `activate` it as follows:

```
activate traja_dev
```

Mac OSX and Linux users should use:

```
source activate traja_dev
```

You will then see a confirmation message to indicate you are in the new development environment.

To view your environments:

```
conda info -e
```

To return to you home root environment:

```
deactivate
```

See the full conda docs here.

At this point you can easily do a *development* install, as detailed in the next sections.

1.17.4 3) Installing Dependencies

To run *traja* in an development environment, you must first install *traja*'s dependencies. We suggest doing so using the following commands (executed after your development environment has been activated):

```
conda install -c conda-forge shapely
pip install -r requirements/dev.txt
```

This should install all necessary dependencies.

Next activate pre-commit hooks by running:

```
pre-commit install
```

1.17.5 4) Making a development build

Once dependencies are in place, make an in-place build by navigating to the git clone of the *traja* repository and running:

```
python setup.py develop
```

1.17.6 5) Making changes and writing tests

traja is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). This development process "relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test." So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to *traja*. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

traja uses the pytest testing system and the convenient extensions in numpy.testing.

Writing tests

All tests should go into the `tests` directory. This folder contains many current examples of tests, and we suggest looking to these for inspiration.

Running the test suite

The tests can then be run directly inside your Git clone (without having to install *traja*) by typing:

```
pytest
```

1.17.7 6) Updating the Documentation

traja documentation resides in the *doc* folder. Changes to the docs are make by modifying the appropriate file in the *source* folder within *doc*. *traja* docs us reStructuredText syntax, which is explained here and the docstrings follow the Numpy Docstring standard.

Once you have made your changes, you can build the docs by navigating to the *doc* folder and typing:

```
make html
```

The resulting html pages will be located in *doc/build/html*.

1.17.8 7) Submitting a Pull Request

Once you've made changes and pushed them to your forked repository, you then submit a pull request to have them integrated into the *traja* code base.

You can find a pull request (or PR) tutorial in the GitHub's Help Docs.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

R

S

T