multi_locus_analysis Documentation

Release 0.0.19+1.g5dd6d52.dirty

Bruno Beltran

Aug 28, 2019

Contents

MLA is a collection of utilities for analyzing multi-locus particle-tracking data.

Please see one of our various tutorials to get started analyzing diffusing particle trajectories, first passage times, MSDs, correlations, etc. and comparing these to established analytical diffusion theory.

To compute MSDs, velocity correlations, and other time-dependent moments of the trajectory, see *Trajectory Statistics*. For computing waiting times—given a trajectory that can take multiple different states (such as "paired" and "unpaired")—see *Waiting Times*. For examples of comparing the analytical theory of (polymer) diffusions to the data, see *Analytical Theory*. Custom plotting code used in the *theory* section is documented in our *plotting guide*, but for actual example plots, see the other tutorials.

CHAPTER 1

Trajectory Statistics

All utilities for calculating moments of a trajectory (i.e. statistics of particle motion), are contained within the *multi_locus_analysis.stats* module.

This statement is made more precise in that module's docstring:

1.1 `multi_locus_analysis.stats`

For computing statistics of particle motions

When dealing with trajectory data, all the common readouts can be thought of as moments of the trajectory with different time lags or discrete derivatives applied.

More precisely, suppose you have trajectory data $X_i(t_k)$, then we define the discrete velocity (the first discrete derivative of the particle as $V_i^\delta(t_k) = X_i(t_k + \delta) - X_i(t_k)$ and the discrete relative velocities $V_{ij}^\delta(t_k) = X_{ij}(t_k + \delta) - X_{ij}(t_k)$ where X_{ij} is the distance between the points i and j. This module simply helps quickly calculate moments of the $V^\delta(t)$.

For example, the ensemble MSD is simply $E_i[(V_i^\delta(0))^2]$ as a function of δ, where $E_\chi[\cdot]$ indicates the average is taken over the variable χ. The time-averaged MSD for a single particle on the other hand is given by $E_t[(V_i^\delta(t))^2]$.

This module provides a simple set of functions for easily extracting the moment of any product of any derivatives with any combination of time lags of a set of trajectories, $X_i(t_k)$.

1.2 Tutorial

Tip: Like the rest of `multi_locus_analysis`, these functions are optimized for use with long-form `pandas` DataFrames. The following tutorial assumes that your data set is small enough for approximately n^3 data points to fit in memory. In order to take up only linear space, simply compose the desired functions instead of applying them one-by-one (which keeps around unnecessary intermediates).

1.2.1 Example data

You can load in the data we will use in this tutorial as follows:

```
>>> import multi_locus_analysis as mla
>>> from multi_locus_analysis.examples import burgess
>>> df = mla.examples.burgess.df
>>> df.head()
                                                    X    Y    Z  foci    t
locus genotype exp.rep meiosis cell  frame spot
HET5  WT       2       t0      1     1     1      1.6  2.4  3.1   unp    0
                                     2     1      1.9  1.5  3.1   unp   30
                                     3     1      2.0  1.8  3.0   unp   60
                                     4     1      2.1  1.9  3.0   unp   90
                                     5     1      2.2  1.8  3.0   unp  120
```

As is typical in this kind of data, there are some number of columns holding the actual measurements, and some number which are simply record-keeping columns. In the Burgess data a uniquely-identifying-subset of the record-keepign columns have been pre-emptively made into an index.

We will be using named collections of columns throughout this tutorial to group together parts of the DataFrame as needed (for example, to get each continuous trajectory we use `traj_cols`). Please see `multi_locus_analysis.examples.burgess` for detailed exaplanations of the information in each column. For now, we need only point out that each unique combination of `['locus', 'genotype', 'exp.rep', 'meiosis', 'cell']` defines a movie, which contains two spots diffusing relative to each other: these are homologous loci at different times in meiosis. To save some typing, we extract the groupings we will use later:

```
>>> cell_cols = burgess.cell_cols
>>> traj_cols = burgess.traj_cols
>>> frame_cols = burgess.frame_cols
```

Different tasks will be easier or harder with the index set to identify a single spot at a single time, or a single time point with different columns for each spot. We can use the `multi_locus_analysis.pivot_loci()` helper to move from one representation to the other

```
>>> mla.pivot_loci(df, pivot_cols=['X', 'Y', 'Z']).head()
                                             foci    t   X1   Y1   Z1   X2   Y2   Z2
locus genotype exp.rep meiosis cell  frame
HET5  WT       2       t0      1     1        unp    0  1.6  2.4  3.1  2.1  1.5  3.1
                                     2        unp   30  1.9  1.5  3.1  1.9  2.5  3.1
                                     3        unp   60  2.0  1.8  3.0  1.5  2.5  3.4
                                     4        unp   90  2.1  1.9  3.0  1.4  2.2  3.4
                                     5        unp  120  2.2  1.8  3.0  1.5  2.4  3.4
```

1.2.2 Velocities to MS(C)Ds

With almost any set of trajectories, we first extract all the velocities that we are interested in. If we have drift-free, individual trajectories, this can be done simply as

```
>>> # subsampling makes computation easy for demo purposes
>>> # group by remaining columns that uniquely id a trajectory
>>> vels = df.loc['URA3', 'WT', 9].groupby(traj_cols[3:]).apply(
>>>     mla.stats.pos_to_all_vel, xcol='X', ycol='Y', zcol='Z', framecol='t')
>>> vels.head()
                                tf   vx   vy   vz
meiosis cell spot ti delta
t0      2    1    0  0           0  0.0  0.0  0.0
                     30         30  0.0 -0.1  0.3
                     60         60  0.0  0.0  0.3
                     90         90  0.1 -0.1  0.2
                     120       120  0.2 -0.2  0.3
```

If there are sufficient trajectories at each time point, then sometimes it is possible to first subtract out the drift first

```
>>> raise NotImplementedError('Pandas shape error in following code')
>>> for i in ['X', 'Y', 'Z']:
>>>     df['dc'+i] = df[i] - df.groupby(frame_cols)[i].mean()
>>> # groupby remaining cols that uniquely id a trajectory
>>> dc_vels = df.loc['HET5', 'WT', 2].groupby(traj_cols[3:]).apply(
>>>     mla.stats.pos_to_all_vel, xcol='dcX', ycol='dcY', zcol='dcZ',
>>>     framecol='t')
>>> dc_vels.head()
```

However, in our example data, we have two trajectories, so it makes more sense to simply work directly with the distance between them as our drift-free measurement.

```
>>> df = mla.pivot_loci(df, pivot_cols=['X', 'Y', 'Z'])
>>> for i in ['X', 'Y', 'Z']:
>>>     df[i+'12'] = df[i+'2'] - df[i+'1']
>>> df.head()
```

We can then get all the $V^\delta_{12}(t_k)$ by simply using these new $X_{12}(t_k)$ columns:

```
>>> d_vels = df.groupby(cell_cols).apply(mla.stats.pos_to_all_vel, xcol='dX', ycol='dY
↪', zcol='dZ', framecol='t')
```

By default, *pos_to_all_vel()* computes all velocities. If all non-overlapping velocities are required instead, such as when computing standard error in a time-averaged MS(c)D measurement, pass the `force_independence` kwarg.

```
>>> # first subsample so that this runs quickly on a single machine
>>> df = df.loc['URA3', 'WT', 9].copy() # one experiment
>>> remaining_index = cell_cols[3:]
>>> d_vels = df.groupby(remaining_index).apply(mla.stats.pos_to_all_vel, \
>>>     xcol='X12', ycol='Y12', zcol='Z12', framecol='t', \
>>>     force_independence=True)
>>> d_vels['v'] = np.sqrt(d_vels['vx']**2 + d_vels['vy']**2 + d_vels['vz']**2)
>>> mscds_per_stage = d_vels.groupby(['meiosis', 'delta'])['v'].agg(['mean', 'std',
↪'count'])
```

The MSD of the distance between two homolog pairs is often called the MSCD in the meiosis literature (standing for Mean Squared Change in Displacement). These MSCDs can be simply plotted via

```
>>> plt.figure(figsize=[4, 3])
>>> for label, data in mscds_per_stage.groupby('meiosis'):
>>>     data = data.reset_index()
```

(continues on next page)

(continued from previous page)

```
>>>     data = data[data['delta'] > 0] # for log-log scale prettiness
>>>     # SEM bars
>>>     plt.errorbar(data['delta']/60, data['mean'], \
>>>                  data['std']/np.sqrt(data['count']), \
>>>                  label=label)
>>> plt.yscale('log')
>>> plt.xscale('log')
>>> plt.ylim([0.01, 1])
>>> plt.xlabel('time lag, \delta (min)')
>>> plt.ylabel('MSCD, $<V_{12}^\delta(t)>$ ($\mu{}m^2/s$)')
>>> plt.legend()
```

1.2.3 Velocity correlations

In order to calculate the velocity autocorrelation, we include the function *all_vel_to_corr()*, which can be used to get numbers that can be groupby'd and averaged to get the velocity autocorrelation.

However, doing this naively on any reasonably-sized dataset will result in way too large of a DataFrame to fit in RAM. This is because this function is simply calculating $V^\delta(t + t_k) \cdot V^\delta(t_k)$ for every single δ and t_k. While this works well for small datasets (like single movies), even clever use of Pandas will often make applying this function to a large dataset spicy. Therefore, we provide convenience functions *vels_to_cvvs_by_hand()* and *vvc_stats_by_hand()* to first write out all these products and then take the appropriate averages (respectively).

If we wanted to use them on the above velocities, we would simply run:

Warning: The following creates a temporary file of approximately 2.1GB.

```
>>> mla.stats.vels_to_cvvs_by_hand(d_vels, ['meiosis'], 'vvs.csv',
>>>                                max_t_over_delta=4)
>>> cvv_stats = mla.stats.vvc_stats_by_hand('vvs.csv', ['meiosis'])
>>> cvv_stats = mla.stats.cvv_by_hand_make_usable(cvv_stats, ['meiosis'])
>>> cvv_stats.set_index(['meiosis', 'delta', 't'], inplace=True)
                        sqs       sum      ...      cvv_normed  ste_normed
meiosis delta t                            ...
t0      30    0    8787.4250  1427.42      ...        1.000000    0.065668
              30   6635.5702  -570.96      ...       -0.421770    0.060173
              60   6503.1084   -88.80      ...       -0.067135    0.060968
              90   6000.1380   -76.52      ...       -0.058923    0.059647
              120  5884.4852   191.24      ...        0.150432    0.060341
```

We can then plot the velocity correlation curves (for example at meiotic stage `t0`) by simply running

```
>>> from multi_locus_analysis import plotting
>>> plotting.cvv_plot_sized(cvv.loc['t0'].reset_index(), data_deltas=30*np.arange(1,␣
↪5))
```

Note: There should be a plot here but it has not been inserted yet pending come organizational decisions.

where we ignore larger values of `delta` because the amount of data available for averaging decreases to a single point as `delta` approaches the length of the observation.

Notice how the correlation decays as $delta$ increases. For details on why this means that the correlation is actually increasing as delta increases, please see the paper in preparation (Newman, Beltran, and Calhoon et al).

1.2.4 Notes

For a listing of all functions exposed by the `stats` module, see the *API reference* section.

CHAPTER 2

Finite window correction

Measurements of “waiting times” or “survival times” taken within a finite time interval require special statistical treatment to account for the bias towards short measurements introduced by the measurement itself. For mathematical details, see the manuscript in preparation (Beltran et. al., *in preparation*).

Otherwise, the process for these corrections in explained in the module’s doctring

2.1 `multi_locus_analysis.finite_window`

For correcting interior- and exterior-censored data

This module is designed to facilitate working with Markov processes which have been observed over finite windows of time.

Suppose you wish to observe a process that switches between states A and B, and observe the process over a window of time $[0, T]$.

There will be two types of “observed” times, which we we call “interior times” (those when the entire time spent in the state is observed) and “exterior times” (the lengths spent in the state that we observed at the start/end point of our window).

For example, suppose you are observing two fluorescent loci that are either in contact (A) or not (B). If you measure for 5s, and the loci begin in contact, come apart at time t=2s, and rejoin at time t=4s,

```
    A A A A A A B B B B B B A A A A
    | - - | - - | - - | - - | - - | -- | -- | ...
    |                             |
t = 0s    1s    2s    3s    4s    5s
```

we would say that T = 5s, and you measured one “interior” time of length 2s, and two exterior times, one of length 2s and one of length 1s (at the start and end of the window, respectively).

When histogramming these times, we must apply a statistical correction to the final histogram weights due to the effects of the finite observation window.

The distribution of the exterior times $f_E(t)$ is exactly equal to the survival function of the actual distribution $S(t) = 1 - \int_0^t f(s)ds$ normalized to equal one over the observation interval. No functions are currently included that leverage this, since extracting information from the survival function is likely only worth it if a large fraction of your observations are exterior times.

On the other hand, the interior times distribution is given by $f_I(t) = f(t)(T - t)$ for $t \in [0, T]$. In order to plot the actual shape of $f(t)$ with this biasing removed, we provide the cdf_exact_given_windows function below.

A typical workflow is, given an array of interior times, `t`, and an array of the window sizes each time was observed within, `w`, is to extract the CDF exactly, then optionally convert that to a PDF to display a regular histogram

```
>>> x, cdf = cdf_exact_given_windows(t, w, pad_left_at_x=0)
>>> xp, pdf = bars_given_cdf(x, cdf)
>>> confint = simultaneous_confint_from_cdf(0.05, len(t), x, cdf)
>>> xc, confs = bars_given_confint(x, confint)
>>> plt.plot(xp, pdf, 'k', xc, confs, 'r-.')
```

The remainder of the functions herein are related to extracting confidence intervals for these distributional estimates.

For details of the derivation, see the Biophysical Journal paper in preparation by Beltran and Spakowitz.

2.2 Tutorial

2.2.1 Generating AB Trajectories

We use the following functions to generate example data (for the case when the means of the distribution are not known or hard to calculate, see the function `multi_locus_analysis.finite_window.ab_window()`):

`multi_locus_analysis.finite_window.`**`ab_window_fast`**`(`*rands, means, window_size, num_replicates=1, states=[0, 1], seed=None, random_state=None*`)`

Simulate a two-state system switching between states A and B.

In addition to functions that can generate random waiting times for each state, this "fast" version of the code requires the average waiting times are are means[0], means[1], respectively.

Warning: apparently, `bruno_util.random.strong_default_seed()` is broken (or this function is) because passing a seed does not make the output reproducible.

Parameters

- **`rands`** (*`(2,) List[scipy.stats.rv_continuous]`*) – One of the random variables defined in `scipy.stats`. Alternatively, any callable that takes *random_state* and *size* kwargs. *random_state* should accept a `np.random.RandomState` seed. *size* will be a tuple specifying output shape of random number array requested.
- **`means`** (*`(2,) array_like`*) – average waiting times for each of the states
- **`window_size`** (*`float`*) – the width of the window over which the observation takes place
- **`num_replicates`** (*`int`*) – number of times to run the simulation, default to 1
- **`states`** (*`(2,) array_like`*) – the "names" of each state, default to [0,1]

- **seed** (*np.random.RandomState*) – state to start the simulation with

Returns df – The start/end times of each waiting time simulated. This data frame has *columns=['replicate', 'state', 'start_time', 'end_time', 'window_start', 'window_end']*.

Return type pd.DataFrame

Notes

Consider the waiting time intersecting the left boundary of the observation window. The left boundary will be a uniform fraction of the way through this wait time. This can easily be seen in the case of finite-variance wait times using CLT and starting the switching process arbitrarily far left of the window of observation, or in general be imposed by requiring time-homogeneity of the experiment.

We use this fact here to speed up correct simulation of time-homogenous windows by directly simulating only the waiting times within the windows instead of also simulating a long run of "pre-equilibrating" waiting times some offset before the window, as in `ab_window()`.

Here's an example of a so-called time-homogeneous process that switches between states "exp(4,4)" and "exp(1,6)". We observe 1000 copies of the process for 20 time units.

```
>>> import scipy.stats
>>> from multi_locus_analysis import finite_window as fw
>>> e44 = scipy.stats.expon(scale=4, loc=4)
>>> e16 = scipy.stats.expon(scale=6, loc=1)
>>> trajs = fw.ab_window_fast([e44.rvs, e16.rvs], [e44.mean(), e16.mean()],
>>>     window_size=20, num_replicates=10000, states=['exp(4,4)', 'exp(1,6)'])
>>> trajs.head()
   replicate     state  start_time   end_time  window_start  window_end
0          0  exp(1,6)   -3.327818   0.788432             0          20
1          0  exp(4,4)    0.788432   6.865933             0          20
2          0  exp(1,6)    6.865933  11.175979             0          20
3          0  exp(4,4)   11.175979  17.382677             0          20
4          0  exp(1,6)   17.382677  18.787130             0          20
```

From each trajectory, we can extract the wait times:

```
>>> waits = trajs.groupby('replicate').apply(fw.traj_to_waits)
>>> waits.head()
            replicate     state  start_time   end_time  window_start  window_end ␣
↪wait_time  window_size      wait_type
rank_order
0                   0  exp(1,6)    0.000000   0.788432             0          20   0.
↪788432           20  left exterior
1                   0  exp(4,4)    0.788432   6.865933             0          20   6.
↪077500           20       interior
2                   0  exp(1,6)    6.865933  11.175979             0          20   4.
↪310046           20       interior
3                   0  exp(4,4)   11.175979  17.382677             0          20   6.
↪206698           20       interior
4                   0  exp(1,6)   17.382677  18.787130             0          20   1.
↪404453           20       interior
```

2.2.2 Rescaling Exact Waiting Times

Below, we plot three curves alongside each other. In black, we show the actual distribution of "waiting time"s in state "exp(4,4)" from the simulation above. In green is the empirical distribution function of the "interior times". If you didn't have specific experience with lifetimes data, it might seem like one could simply histogram the interior times in order to reproduce the actual distribution (since you observed the whole waiting time). However, as you can see, even if we only take "interior" times, the empirical CDF of the data does not match the distribution that we put in (e.g. *expon(loc=4, scale=4)* in this case, from above), even if we are extra generous and rescale the CDFs to match at "t=window_size".

```
>>> # plot kaplan-meier estimate first, library creates new figure
>>> import lifelines
>>> waits44 = waits[waits['state'] == 'exp(4,4)']
>>> kmf = lifelines.KaplanMeierFitter().fit(
>>>     waits44['wait_time'].values,
>>>     event_observed=(waits44['wait_type'] == 'interior').values,
>>>     label='Meier-Kaplan Estimator, $\pm$95% conf int'
>>> )
>>> kmf.plot_cumulative_density()
>>>
>>> # plot actual distribution
>>> t = np.linspace(0, 20, 100)
>>> plt.plot(t, e44.cdf(t), 'k-.', label='Actual CDF, exp(4,4)')
>>>
>>> # now compute the empirical distribution of the "interior" times
>>> interior_44 = waits.loc[
>>>     (waits['state'] == 'exp(4,4)') & (waits['wait_type'] == 'interior'),
>>>     'wait_time'
>>> ].values
>>> x44, cdf44 = fw.ecdf(interior_44, pad_left_at_x=0)
>>> plt.plot(x44, cdf44*e44.cdf(x44[-1]), label='Empirical CDF, exp(4,4)')
>>> # prettify the plot
>>> plt.xlabel('t')
>>> plt.ylabel(r'$P(\mathrm{wait} <= t)$')
>>> plt.legend()
```

The classical solution to this problem is to use the "Kaplan-Meier" correction. This is the blue line in the figure above, along with 95% (pointwise) confidence intervals. However, if the interval of observation is large enough that multiple state changes can be observed in one trajectory, then the Kaplan-Meier correction will under-correct, because there is negative autocorrelation between "successful" observation of times within a given trajectory (i.e. if we observe a long time, then we will be near to the end of our observation window, so the next observation *must* be small). The Meier-Kaplan correction was designed to work with right-censored data, but only with right-censored data where every single observed wait time is perfectly independent.

In order to correctly reproduce the actual underlying distribution, we can apply a more general correction using `multi_locus_analysis.finite_window.ecdf_windowed()`.

```
>>> plt.figure(figsize=[4,3])
>>> x, cdf = fw.ecdf_windowed(interior_44, 20)
>>> plt.plot(x, cdf, label='Empirical CDF, state 1')
>>> plt.plot(t, e44.cdf(t), 'k-.', label='Actual CDF')
>>> plt.xlabel('t')
>>> plt.ylabel(r'$P(\mathrm{wait} <= t)$')
```

As you can see, this still does not perfectly reproduce the distribution, but the bias is effectively zero for quite a large amount of time. We plot the probability distribution functions (using a gaussian kernal density estimator) below, and compare it to the result of using the Meier-Kaplan corrector. Notice that the Meier-Kaplan corrector mis-estimates the "slope" of the PDF in semilog space (which would lead to a mis-estimate of the parameter of the distribution) while our estimator has the correct slope for quite some time, and so a fit to the PDF (correctly cut off where the variance in the estimator starts to increase) would correctly estimate the parameters of the actual distribution.

```
>>> plt.figure(figsize=[4,3])
>>> kernel = fw.smooth_pdf(x, cdf)
>>> plt.plot(t, kernel(t), label='Gaussian KDE, exp(4,4)')
>>> plt.plot(t, e44.pdf(t), 'k-.', label='Actual PDF')
>>> plt.yscale('log')
>>> plt.ylim([e44.pdf(20), 0.4])
>>> plt.xlabel('t')
>>> plt.ylabel(r'$P(\mathrm{wait} <= t)$')
```

```
>>> plt.figure(figsize=[4,3])
>>> kcdf = kmf.cumulative_density_at_times(x)
>>> kernel = fw.smooth_pdf(x, kcdf)
>>> plt.plot(t, kernel(t), label='Meier-Kaplan + "Gaussian KDE"')
>>> plt.plot(t, e44.pdf(t), 'k-.', label='Actual PDF')
>>> plt.yscale('log')
>>> plt.ylim([e44.pdf(20), 0.4])
>>> plt.xlabel('t')
>>> plt.ylabel(r'$P(\mathrm{wait} <= t)$')
```

2.2.3 A Caveat for Discrete Movies

While we would ideally have an exact measurement of when transitions between *A* and *B* states happen, it is more often the case that we have a "movie" of sorts: where we measure the state of the system at a fixed set of times.

This only provides us with upper and lower bounds for the actual waiting time that we're trying to observe. For example, consider the trajectory depicted below.

```
    A A A A A B B B B B B B A A A A B
    | - - - | - - - | - - - | - - - |  ...
    |                               |
t = 0s      1s      2s      3s      4s
```

This trajectory, when measured at the discrete times shown, would look like

```
>>> pd.Series({0: 'A', 1: 'A', 2: 'B', 3: 'A', 4: 'B'}).head()
    0    A
    1    A
    2    B
    3    A
    4    B
    dtype: object
```

Naively, if you only had this movie in front of you with no knowledge of the actual underlying state change times, it might seem to suggest that there was an exterior-censored "A" of length 2, one each interior censored times of length 1, and one exterior-censored "B" time of length 1. However, by looking at the true trajctory above, we see that the first "A" wait was much shorter than 2s, and the first "B" wait was much longer than 1s, whereas the last "A" wait just happened to match up with our prediction of 1s.

Because our normalization factor depends non-linearly on the observed waiting time, one might guess that simply using the "naive" times might cause bias. We will show that this is the case by generating some artificial movies ourselves.

2.2.4 Generating Discrete Trajectories (Movies)

multi_locus_analysis.finite_window includes a convenience method for generating "movies" from the output of the *AB_window** functions.

`multi_locus_analysis.finite_window.`**`state_changes_to_trajectory`**(*traj*, *times*, *state_col='state'*, *start_times_col='start_time'*, *end_times_col=None*)

Converts a series of state change times into a Series containing observations at the times requested. The times become the index.

Parameters

- **`times`** (*`(N,) array_like`*) – times at which to "measure" what state we're in to make the new trajectories.
- **`traj`** (*`pd.DataFrame`*) – should have *state_col* and *start_times_col* columns. the values of *state_col* will be copied over verbatim.
- **`state_col`** (*`string`*) – name of column containing the state being transitioned out of for each measurement in *traj*.
- **`start_times_col`** (*`string`*) – name of column containing times at which *traj* changed state
- **`end_times_col`** (*`(optional) string`*) – by default, the function assumes that times after the last start time are in the same state. if passed, this column is used to determine at what time the last state "finished". times after this will be labeled as NaN.

Returns **movie** – Series defining the "movie" with frames taken at *times* that simply measures what state *traj* is in at each frame. index is *times*, *state_col* is used to name the Series.

Return type pd.Series

Notes

A start time means that if we observe at that time, the state transition will have already happened (right-continuity). This is confusing in words, but simple to see in an example (see the example below).

Examples

For the DataFrame

```
>>> df = pd.DataFrame([['A',  -1, 0.1], ['B', 0.1, 1.0]],
>>>     columns=['state', 'start_time', 'end_time'])
```

the discretization into tenths of seconds would give

```
>>> state_changes_to_trajectory(df, times=np.linspace(0, 1, 11),
>>>     end_times_col='end_time')
t
0.0      A
0.1      B
0.2      B
0.3      B
0.4      B
0.5      B
0.6      B
0.7      B
0.8      B
```

(continues on next page)

(continued from previous page)

```
0.9      B
1.0    NaN
Name: state, dtype: object
```

Notice in particular how at 0.1, the state is already 'B'. Similarly at time 1.0 the state is already "unknown". This is what is meant by the Notes section above.

If the *end_times_col* argument is omitted, then the last observed state is assumed to continue for all *times* requested from then on:

```
>>> state_changes_to_trajectory(df, times=np.linspace(0, 1, 11))
t
0.0    A
0.1    B
0.2    B
0.3    B
0.4    B
0.5    B
0.6    B
0.7    B
0.8    B
0.9    B
1.0    B
Name: state, dtype: object
```

This function has an alias for convenience (`multi_locus_analysis.finite_window.traj_to_movie()`).

```
>>> movies = trajs.groupby('replicate').apply(traj_to_movie,
times=np.linspace(0, 1, 10))
>>> movies.head()
t          0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
replicate
0            0    0    1    1    1    1    0    0    0    1    1
1            0    1    1    1    1    1    1    1    1    1    1
2            1    1    1    1    0    1    1    1    0    1    1
3            0    0    1    1    1    1    1    1    1    1    0
4            0    0    0    1    1    1    1    1    1    1    0
```

We can get a long-form DataFrame by simply doing

```
>>> movies.T.unstack()
>>> movies.name = 'state' # name resulting column from unstack
>>> movies.head()
replicate  t
0          0.0    0
           0.5    0
           1.0    1
           1.5    1
           2.0    1
Name: state, dtype: int64
```

As is clear from the following plot, the data being effectively discretized creates a bias in the tail of the distribution, even when the times are corrected with our method.

2.3 Theoretical Details

The following section contains a complete derivation of the framework used to generate the corrections used in this module.

2.3.1 Motivating (A)synchronicity

We first motivate our definition of "(a)synchronicity", the critical property that allows us to correct for the effects of observing in a finite window.

Suppose a process starts at $-t_{\text{inf}}$ (WLOG, assume it starts in state A). For times after $-t_{\text{inf}} \lll 0$, the process switches between states A and B. The distribution of times spent in each state before switching are IID, and distributed like $f_A(t)$ and $f_B(t)$, respectively. We then are able to observe the process during the interval of time $[0, T]$.

This can be thought of as a renewal(-reward) process that started far in the past. As long as the starting point, $-t_{\text{inf}}$, is sufficiently far in the past, and the distributions $f_*(t)$ have finite variance, various convenient properties hold true for the observed state switching times between 0 and T. We use the same example as in the "tutorial" section in what follows.

The first convenient property is that the switching times are uniformly distributed within the observation interval (as $-t_{\text{inf}} \to -\infty$). Intuitively, this just means that $-t_{\text{inf}}$ is far enough in the past that, independently of the distribution, we are not biased towards the switching times being early or late in our observation interval (i.e. we have lost all memory of the "real" start time).

Now let's label the observed state switches as $t_0, \ldots, t_{n-1}$, with t_0 and t_n corresponding to the "actual" (unobserved) state switch times flanking the observation interval. The next useful property is that the start of the observation interval ($t = 0$) is uniformly distributed within $[t_0, t_1]$ (similarly, the end of the observation interval, $t = T$, is uniformly distributed in $[t_{n-1}, t_n]$.

For the interior times, we can simply use the first fact to derive our interior time correction. Since we know the starting times of each state are uniformly distributed, we immediately can tell that if a waiting time of length τ has a start time within the interval, then the the fraction of times that this waiting time will end up being an interior time is just $(T - \tau)/T$. More precisely, we have that

$$P(t_{i+1} \leq T | t_{i+1} - t_i = \tau, t_i \in [0, T]) = \int_0^T 1_{t_i + \tau \leq T} f_{\text{Unif}[0,T]}(t_i) dt_i$$

which is just equal to $(T - \tau)/T$.

This correction factor can be visualized easily as simply counting what fraction of "start" times of a given length lead to "end" times still inside the interval. Namely, it's the green part of the interval in the following diagram:

On the other hand, we have to be careful about the distribution of exterior times, even if we do somehow magically have the values of t_0 and the state at $t = 0$. You can't simply assume that $t_1 - t_0$ is distributed like $f_A(t)$ or $f_B(t)$. After all, in fact it is distributed like $t f_*(t)$. This is because (loosely speaking) if you fill the real line with a bunch of intervals whose lengths are distributed like $f(t)$, then you choose a point on the real line at random, you are more likely to land in an interval of size t the longer that t is.

2.3.2 (A)synchronicity

While the explicit framework presented above is a useful tool, it is ill-defined for heavy-tailed processes, in which we are primarily concerned when making these types of corrections. In order to retain the useful properties of the system that made it possible to derive the interior and exterior times distributions, we simply notice that the *real* property that we want to be true when measuring these systems is *asynchronicity*, or what a physicist might call "symmetry under time translations" or "time homogeneity". In short, we want to impose the constraint that we are only interested in scientific measurements where changing the interval of observation $[0, T]$ to $[0 + \tau, T + \tau]$ for any τ will not change any properties of the measurement.

Note: We leave as an exercise to the reader to show that:

1. the renewal process of the previous section is a special case of an asynchronous process
2. this definition of asynchronicity produces all three properties we demonstrated for our renewal process above

On the other extreme from asynchronicity is the situation in which the Meier-Kaplan correction was originally designed to be used. Namely, we could imagine that a perfectly *synchronous* process is one where t_0 is fixed to be at time $t = 0$, meaning that $t_1 - t_0$ is distributed as just $f_*(t)$.

While in principle anything between asynchrony and synchrony is possible, it is true in general that almost all scientific measurements area already done using either purely synchronous or asynchronous systems, since it is intuitively clear that a lack of understanding of the synchronicity of one's system can lead to uninterpretable results.

2.3.3 Laplace Formalism

CHAPTER 3

Analytical theory

Here, we show some best practices for comparing MSDs, displacement distributions, and velocity correlations to the theory of diffusing molecules.

The literature contains many good reviews on these topics, so we cover only some often overlooked pitfalls.

First, we review the most-used *model* of viscoelastic diffusion (for both free molecules and large polymers).

The first common pitfall we address is the *effect of confinement* on MSDs, especially on extracting diffusivity and the subdiffusivity coefficient (i.e. Hurst index).

Then, we discuss the effects of heterogeneity between measurements on the *gaussianity of the displacement distribution* and what that looks like for real data.

Finally, we discuss how to use the velocity cross correlation of two particles to *extract information* about whether those two particles are connected (such as by being on the same polymer) using a drift-independent approach.

3.1 Modeling viscoelastic diffusion

The crowded cellular environment is often subdiffusive, and across bacteria, in some yeast and some higher mammalian cells, it has been shown that this diffusion is well described by a fractional Brownian motion (for example, see (Weber et. al., PRL, 2010)).

A particle undergoing fractional Brownian motion can be modeled by a fractional Langevin equation (ignoring inertial terms)

$$\partial_t^\alpha X(t) = F_B(t)$$

where we use the usual Caputo definition of the fractional derivative ∂_t^α. This corresponds to a power-law memory kernel, $K(t - t_0) = \frac{(2-\alpha)(1-\alpha)}{|t-t_0|^\alpha}$, where

$$\xi \int_0^t K(t - t_0) \frac{dX(t_0)}{dt_0} dt_0 = \dots$$

replaces the constant viscosity of the regular Langevin equation

$$\xi \frac{dX(t)}{dt} = \ldots$$

Intuitively, this means that there is a viscoelastic force that tries to return the particle to it's original location (by opposing the historical velocity of the particle). The strength of this force falls off as $t^{-\alpha}$.

The right-hand side of this equation is called the "Brownian force" and is a gaussian process with mean zero and covariance determined by the fluctuation-dissipation theorem

$$\langle F_B(t) F_B(t') \rangle = \xi k_B T K(t - t') I.$$

The viscoelastic memory kernel is parameterized by the single parameter α, which describes how much the system wishes to return (elastically) to a previous configuration.

The simplest way to incorporate the effects of being on a polymer is to use the Rouse model for polymer dynamics. This model is a formal way to describe how a so-called "Gaussian chain" (i.e. the limit as the number of beads goes to infinity of a model where the polymer is composed of beads $X_i(t)$ connected by springs) behaves when undergoing thermal fluctuations.

$$\partial_t^\alpha X(n,t) = \frac{3k_B T}{b^2} \partial_n^2 X(n,t) + F_B(t)$$

where b is called the "Kuhn length" of the Gaussian chain, and describes (for a fixed time) how changing the length of the polymer changes the volume occupied by the polymer.

While the Gaussian chain (and by proxy, the Rouse model) is often unrealistic for isolated, short polymer chains, it can be shown mathematically (for example, see the section on Gaussian chains in Doi and Edwards) that every polymer has a Kuhn length, and behaves identically to a Gaussian chain on long enough length scales. Therefore, for the case of two loci on chromatin, separated by more than a couple of kilobases, we argue that the Rouse model is more than sufficient.

Note: Notice that this equation is very reminiscent of the Fokker-Planck-Kolmogorov equation for continuous time random walk (CTRW).

A the probability distribution for the position of a free particle undergoing a CTRW where the step sizes are Gaussian and the waiting times between steps are distributed like $\hat{\phi}(s) = 1/(1 + 1/\hat{\eta}(s))$ (where the hat denotes a Laplace transform), can be modeled via

$$\partial_t p(x,t) = \partial_t \int_0^t \eta(t - t') \partial_x^2 p(x, t') dt'$$

In the case where η is a power law (like $t^{1-\alpha}$), this reduces to our Langevin equation above, with the Brownian force removed.

These equations are fundamentally different however, because in our model the fluctuation dissipation theorem connects the Brownian force F_B to the viscosity. This means the velocity correlation for a fBm particle is given by $\langle V(t)V(0) \rangle \propto -t^\alpha$, whereas since the CTRW takes independent steps by construction, and so has $\langle V(t)V(0) \rangle = 0$ almost everywhere.

In what follows, we consider the diffusion to be well described by a simple fractional Brownian motion, with no random waiting times between collisions. The combination of these two models ((f)Bm and a CTRW) is discussed to some degree in the Supplemental Materials of (Beltran and Kannan, PRL 2019), and represents well a random walk with heavy-tailed "defects".

3.2 MSDs

The MSD of a particle diffusing according to the equations above can be written as

$$\frac{3k_BT}{\xi}\frac{\sin(\alpha\pi)}{\pi(1-\alpha/2)(1-\alpha)\alpha}t^\alpha$$

In other words, in log-log space, the MSD will be a line with slope α.

Note: While this is true even for arbitrarily short times in our fractional diffusion model (due to its fractal nature), for real particles diffusing in actual viscoelastic materials (like beads in a polymer gel), as we probe shorter and shorter times, there should be a time scale on which the elastic forces of the material are not felt. At these size scales in real data we will often see an MSD that looks like a regular diffusion (a line with slope $\alpha = 1$ in log-log space). At even shorter times (usually too short to measure), the particle will eventually exhibit ballistic behavior, as it travels processively between collisions between the molecules that surround it.

If we instead consider the diffusion of a polymer instead of a point, we get a new equation of motion.

TODO: add details on polymer MSD here.

This means that for a point on the polymer, the observed MSD in log-log space will be a line with slope $\alpha/2$ for short time scales. This persists until the terminal relaxation time $t_R = [N^2b^2\xi/(k_BT)]^{1/\alpha}$ of the polymer, which is the time scale at which the polymer as a whole can be said to be diffusing (in other words, the time scale at which the diffusion of the center of mass of the polymer begins to dominate over the internal fluctuations of the polymer). At times scales longer than the terminal relaxation time, the MSD will be a line with slope α, and the polymer as a whole will diffuse as if experiencing an effective viscosity of $N\xi$.

```
>>> import matplotlib as mpl
>>> from multi_locus_analysis import analytical
>>> alpha = 0.8; N = 100; b = 1
>>> t = np.logspace(-6, 6, 100)
>>> msd = analytical.rouse_mid_msd(t, alpha, b, N, num_modes=1000)
>>> plt.loglog(t[t>1], 3/N*np.sin(alpha*np.pi)/(
>>>     np.pi*(1-alpha/2)*(1-alpha)*alpha)*np.power(t[t>1], alpha),
>>>     '-.', label=r'$\propto t^\alpha$')
>>> plt.loglog(t, np.power(t, alpha/2), '-.', label=r'$t^{\alpha/2}$')
>>> plt.loglog(t, msd, 'k', label='MSD(t)')
>>> plt.xlabel('time (AU)')
>>> plt.ylabel(r'MSD ($\mu{}m^2$)')
>>> plt.legend()
```

Note that the drop-off on the left-hand side of the MSD plot is simply due to the finite number of Rouse modes included when numerically evaluating the MSD. The Rouse polymer is a “fractal” model, so the proper solution to the equations of motion presented above has an MSD whose short-time behavior scales as $t^{\alpha/2}$ indefinitely. The log of the number of modes included (optional arg *num_modes* in the call) corresponds exactly to the number of orders of magnitude that will have the correct scaling in the calculated MSD.

However, this drop-off is also instructive, because this fractal behavior is exactly the most unrealistic part of the Rouse model. In real systems, at short enough time scales, a monomer on the polymer will also transition back to the background slope of $\propto \alpha$. This means that if you measure an actual locus on a polymer, say a particular locus of a small plasmid in a mammalian cell (where $\alpha \approx 1$). Then we will see a curve that looks just like the black curve above. The long time crossover (from $t^{\alpha/2}$ to t^α) will measure the terminal relaxation time of the plasmid (the time scale after which diffusion of a locus on the plasmid is dominated by diffusion of the plasmid as a whole). The short crossover time (from t^α to $t^{\alpha/2}$) will correspond to the time scale at which thermal forces from the surrounding medium begin to be dominated by the elastic restoring force from the polymer chain.

Note: The fractional Brownian motion itself is also a fractal model, and as such has the similar shortcomings to the Rouse model. Namely, at long enough time scales, if confinement isn't a factor, it is likely that the MSD will transition eventually to a slope of $\alpha = 1$ for any realistic system. This may also be true for short enough times. In a sense, what one can say is that the fBm Rouse model is really telling us is that for the time scales described above, whatever the medium value of α is, being on a polymer simply halves it. Thus it is always important to compare polymer MSDs to the MSD of a free particle in the same medium, since many complex mediums will not have a single characteristic value of α, but instead have an MSD whose slope depends on the time or length scale (e.g. due to defects or traps of a particular size scale). If comparison to a free particle is not possible experimentally, there are also other ways to estimate the terminal relaxation time of the polymer if this is not possible (see velocity cross correlation discussion below).

So in experimental MSDs, where the chromosomal loci are confined to a given spatial region (whether that be a chromosome territory or the nucleus as a whole), we expect to see the MSD curve cut off at a different location depending on how the confinement time scale compares to the terminal relaxation time of the chromosome (or chromosomes, if they are linked together).

TODO: make plot showing three options (plateau before polymer regime, before terminal relaxation time, and after terminal relaxation time).

As can be seen from the plot above, even for this simplest possible model of Rouse polymer motion, simply fitting MSD curves with straight lines in log-log space can lead to extremely misleading results, due to the non-linear nature of the MSD curve itself. Therefore, it is important to establish the confinement diameter, and terminal relaxation time of the polymer at a minimum, before trying to fit a raw MSD to extract, for example, a value for α.

3.3 Displacement Distributions

TODO: document gaussianity vs laplace distributions of dispalcements here.

The paper of Lampo et al is extremely valuable for understanding how heterogeneity in diffusivities between different cells can lead to a population that appears to have a non-guassian displacements distribution even though each underlying cell may itself have gaussian displacements.

A more general overview of the combined effects of intra-cellular and inter-cellular heterogeneity can be found in the *later paper by Stylianidou and Lampo <https://journals.aps.org/pre/abstract/10.1103/PhysRevE.97.062410>_*.

This section will eventually summarize how these issues appear in practice, where the distribution of diffusivities is often not as clean as in the data that Lampo et al present.

3.4 Velocity Cross-Correlation

In order to measure whether (and how) stress is communicated between the loci being measured, we can look at their velocity cross correlation functions.

The velocity autocorrelation of a free particle undergoing fractional Brownian motion will serve as a useful comparison throughout what follows in order to understand limiting behavior of the more complicated, polymer case. Such a particle has a velocity autocorrelation function given by

$$\frac{C_v^{(\delta)}(t)}{C_v^{(\delta)}(0)} = \frac{|t-\delta|^\alpha - 2|t|^\alpha + |t+\delta|^\alpha}{2\delta^\alpha}$$

where we should notice that $C_v^{(\delta)}(0)$ is just the MSD as a function of δ. We plot this function below for a few values of α.

```
>>> td = np.linspace(0, 4, 401)
>>> betas = np.concatenate([np.linspace(0, 1, 6), np.array([1.5, 2])])
>>> betas[0] += 0.0001
>>> cmap = plt.get_cmap('viridis')
>>> cnorm = mpl.colors.Normalize(vmin=0, vmax=1.5)
>>> for beta in betas:
>>>     label = f'$\\beta = {beta:0.1}$' if beta < 1 else f'$\\beta = {beta}$'
>>>     plt.plot(td, analytical.frac_discrete_cv_normalized(td, 1, beta),
>>>              label=label, c=cmap(cnorm(beta)))
>>> plt.xlabel(r'$t/\delta$')
>>> plt.ylabel(r'$C_v^{(\delta)}(t)$')
>>> plt.legend(loc='upper right')
```

Notice that as α goes to 1, the velocity correlation goes to that of a regular Brownian motion, $\max\{1 - t/\delta, 0\}$. As α goes to 2, the velocity correlation becomes identically 1 (as expected for ballistic motion). As α goes to 0, the velocity correlation is identically zero, with unit spikes at zero and one in the positive and negative directions, respecitvely (as expected for totally arrested motion).

As shown in (Lampo & Spakowitz, Biophysical Journal, 2016), the velocity cross-correlation of two loci on a Rouse polymer can be computed exactly.

$$C_{vv}^{(\delta)}(t, n_1, n_2) = \left\langle \vec{V}^{(\delta)}(n_1, t) \cdot \vec{V}^{(\delta)}(n_2, 0) \right\rangle$$

after a cosine transform in n, and expanding in the eigenbasis of the fractional derivative, $E_{\alpha,1}$ (the Mittag-Leffler functions), we get

$$C_{vv}^{(\delta)}(t, n_1, n_2) = \frac{3k_BT}{N\xi\Gamma(3-\alpha)\Gamma(1+\alpha)} \frac{2C_v^{(\delta)}(t)}{C_v^{(\delta)}(0)}$$

$$+ \sum_{p=1}^{\infty} \frac{3k_BT}{k_p} \phi_p(n_1)\phi_p(n_2) \left(-E_{\alpha,1}(-\frac{k_p|t+\delta|^\alpha}{N\xi\Gamma(3-\alpha)}) \right.$$

$$\left. +2E_{\alpha,1}(-\frac{k_p|t|^\alpha}{N\xi\Gamma(3-\alpha)}) - E_{\alpha,1}(-\frac{k_p|t-\delta|^\alpha}{N\xi\Gamma(3-\alpha)}) \right)$$

The first term captures the motion of the center of mass of the polymer, while the terms in the sum capture the motion due to different "modes" of the polymer.

The coefficients $k_p = \frac{3\pi^2 k_B T}{N b^2} p^2$ then determine how strongly to weight each mode as a function of t and δ.

Because $E_{\alpha,1}(x) \to 0$ as $x \to -\infty$, and since $k_p \propto p^2$, the summation converges fairly quickly. However, using the numerical values for $\phi_p(n_i)$ tends to cause numerical instabilities. Instead, we consider several limiting cases where properties of the cosine function allow us to simplify the expression.

Firstly, the polymer locus's autocorrelation function, i.e. $n_1 = n_2$ is just

$$TODO$$

The unnormalized version of this equation can be found in Eq. 33 of (Weber & Spakowitz Physical Review E, 2010).

3.4.1 Infinite Polymer Limit

In the simple, but typical case where $\alpha = 1$, the infinite polymer case becomes significantly simpler than the general case of a finite polymer.

We can express the velocity correlation of two loci on an infinite polymer in a regular viscous medium in terms of the Meijer G-functions, which are defined as the Mellin-Barnes integrals

$$G_{p,q}^{m,n}\left(\begin{matrix} a_1,\dots,a_n; a_{n+1}\dots a_p \\ b_1,\dots,b_m; b_{m+1}\dots b_q \end{matrix} \;\middle|\; z;r\right) = \frac{1}{2\pi i}\int_L \frac{\prod_{j=1}^m \Gamma(b_j+s)\prod_{j=1}^n \Gamma(1-a_j-s)}{\prod_{j=n+1}^p \Gamma(a_j+s)\prod_{j=m+1}^q \Gamma(1-b_j-s)} z^{-s/r} ds$$

For two loci separated by a distance Δn, the correlation function is simply given by

$$\begin{aligned} C_{vv}^{(\delta)}(t,\Delta n) &= \frac{3k_B T}{\delta^2\sqrt{\xi k}} \\ &\times \left\{ |t-\delta|^{1/2} G_{1,2}^{2,0}\left[\frac{|\Delta n|^2 \xi}{4k}|t-\delta|^{-1}\Big|_{0,1/2}^{3/2}\right] \right. \\ &+ |t+\delta|^{1/2} G_{1,2}^{2,0}\left[\frac{|\Delta n|^2 \xi}{4k}|t+\delta|^{-1}|_{0,1/2}^{3/2}\right] \\ &\left. -2|t|^{1/2} G_{1,2}^{2,0}\left[\frac{|\Delta n|^2 \xi}{4k}|t|^{-1}|_{0,1/2}^{3/2}\right] \right\}. \end{aligned}$$

This function can be evaluated exactly using any numerical library that has routines for evaluating Meijer G-functions. In particular, in Python, this functionality is implemented in the well-tested package *mpmath*.

3.4.2 Velocity Cross-Correlation "Trick"

Let us consider the velocity autocorrelation of the distance between two monomers $C_{\Delta V}^{(\delta)}$, where $\Delta V^{(\delta)}(t) = \Delta X(t+\delta) - \Delta X(t)$ and *Delta X(t) = X_2(t) - X_1(t)*.

This quantity has the useful property of being drift-free. That is, an arbitrary rotation and translation can be applied to the sample at each time without affecting the values of $\Delta V^{(\delta)}(t)$.

It is simple enough to notice that this is simply equal to

$$C_{\Delta V}^{(\delta)}(t,\Delta n) = C_{vv}^{(\delta)}(t,\Delta n) - 2 * C_v^{(\delta)}(t)$$

where the monomer velocity autocorrelation $C_v^{(\delta)}(t)$ and the velocity cross correlation between two monomers $C_{vv}^{(\delta)}$ is as above.

CHAPTER 4

Plotting utilities

Helper functions used to generate the examples in the *Analytical Theory* tutorial are contained in the *`multi_locus_analysis.plotting`* module. Please see the *API docs* for details.

Historically, this package's author has been able to make most plots related to multi locus analysis with a couple of short lines of `matplotlib` code.

Histograms in Python are not the most elegant, but *`multi_locus_analysis.finite_window`* has several methods for representing histograms fairly as smooth (or step) functions that can simply be plotted.

Documentation on how to get around several common issues encountered when plotting (in particular, using colors, colorbars, normalization and facets, can be found in the docs for the external module `bruno_util.plotting`.

All generally-applicable plotting routines developed within this codebase have been moved to `bruno_util`, along with their documentation.

4.1 Displacement Distributions

The displacement distribution, $f_{V_i^\delta}(x)$ describes how far (x) a particle has moved after a given time δ.

This is a distribution for every value of δ. Plotting one curve for every value of δ is unwieldy:

Note: There will be a plot here, but not made yet.

We provide *`multi_locus_analysis.plotting.make_all_disps_hist()`* for quickly plotting subsets of the histograms, comparing to analytical theory, etc. See *Displacement Distributions* for mathematical details.

Note: There will be a plot here, but not made yet.

4.1.1 Velocity Correlations

Similarly to the displacements distribution, the velocity correlation function $< V_{ij}^{\delta}(t) \cdot V_{ij}^{\delta}(t) >$ is a function for each value of δ, and so is cumbersome to plot.

We provide `multi_locus_analysis.plotting.cvv_plot_sized()` to aid in subsampling, weighting the plotted curves visually based on their statistical certainty, and other useful tricks, like comparing to the analytical theory of polymer diffusion as described in our *analytical theory tutorial*.

CHAPTER 5

API reference

Click any module for detailed descriptions of available functions, usage, etc.

5.1 Trajectory statistics

`stats`	For computing statistics of particle motions

5.1.1 multi_locus_analysis.stats

For computing statistics of particle motions

When dealing with trajectory data, all the common readouts can be thought of as moments of the trajectory with different time lags or discrete derivatives applied.

More precisely, suppose you have trajectory data $X_i(t_k)$, then we define the discrete velocity (the first discrete derivative of the particle as $V_i^\delta(t_k) = X_i(t_k+\delta) - X_i(t_k)$ and the discrete relative velocities $V_{ij}^\delta(t_k) = X_{ij}(t_k+\delta) - X_{ij}(t_k)$ where X_{ij} is the distance between the points i and j. This module simply helps quickly calculate moments of the $V^\delta(t)$.

For example, the ensemble MSD is simply $E_i[(V_i^\delta(0))^2]$ as a function of δ, where $E_\chi[\cdot]$ indicates the average is taken over the variable χ. The time-averaged MSD for a single particle on the other hand is given by $E_t[(V_i^\delta(t))^2]$.

This module provides a simple set of functions for easily extracting the moment of any product of any derivatives with any combination of time lags of a set of trajectories, $X_i(t_k)$.

`multi_locus_analysis.stats.`**`all_vel_to_corr`**(*vel*, *dxcol='vx'*, *dycol='vy'*, *dzcol=None*, *framecol='tf'*, *max_dt=None*, *max_t_over_delta=4*, *allowed_dts=None*)

```
>>> all_vel.reset_index(level=pandas_util.multiindex_col_ix(all_vel, 'ti'),␣
↪inplace=True)
```

`multi_locus_analysis.stats.`**`combine_moments`**(*data*)

Takes a column from output of df.groupby([. . .]).apply(moments) and combines the values to get overall moments for full dataset.

Example:

```
grouped_stats = df.groupby(['experiment']).apply(lp.moments)
overall_stats = grouped_stats.groupby(level=0, axis=1).apply(lp.combine_moments).
↪unstack()
```

Example:

```
def grouper(g):
    return g.groupby(level=0, axis=1).apply(lp.combine_moments).unstack()
grouped_stats[new_col] = make_interesting_value(grouped_stats)
overall_stats = df.groupby(new_col).apply(grouper)
```

`multi_locus_analysis.stats.`**`convex_hull`**(*df, xcol='x', ycol='y', zcol=None, tcol='t', allowed_t=None, max_t=None, volume=False, area=False*)

Compute the convex hull of a trajectory

For a DataFrame containing a trajectory with (X,[Y,[Z]]) values in *xcol*, *ycol*, and *zcol* (respectively), use scipy.spatial.ConvexHull (uses "QHull" under the hood) to calculate the convex hull (including area/volume), optionally only looking at certain times along the trajectory.

2D by default (xcol='x', ycol='y').

`multi_locus_analysis.stats.`**`corr_prod`**(*df, xcol, tcol=None, max_dt=None, max_t_over_delta=4, allowed_dts=None, t_step=None*)

Calculates the time averaged autocorraltion correlation over a column, with an optional time column.

The time average can be written .. math:

```
C_X(t) = E_\tau[X_{\tau + t}X_\tau]
```

`multi_locus_analysis.stats.`**`cvv_by_hand_make_usable`**(*cvv_stats, group_cols*)

add columns to vvc_stats_by_hand output that we typically want. (cvv, std, ste, cvv_normed, ste_normed). requires the group-by columns used to do the normalization via msds using groupby

`multi_locus_analysis.stats.`**`groupby_apply_efficient`**(*df, group_cols, apply_fun, n_between_gc=50, print_progress=False, apply_to_cols=None*)

An attempt to make groupby not leak memory. Kinda worked at one point (late 2016).

`multi_locus_analysis.stats.`**`moments`**(*df, cols=None, ns=[0, 1, 2]*)

Take a DataFrame and optionally a list of columns of interest and for each n in ns, calculate the nth moment of each column, where the 0th moment in defined for convenience to be the count.

Returns a series with multi-index with two levels. Level 0 contains the requested columns and level 1 contains the requested moment numbers (ns).

`multi_locus_analysis.stats.`**`pos_to_all_vel`**(*trace, xcol='x', ycol='y', zcol=None, framecol='i', delta=None, delta_max=None, deltas=None, absolute_time=None, force_independence=False*)

For getting displacement distributions for all possible deltas simultaneously.

```
>>> all_vel = df.groupby(bp.track_columns).apply(lambda df:
            mla.pos_to_all_vel(df, xcol='dX', ycol='dY', framecol='tp.n'
    ))
```

`multi_locus_analysis.stats.`**`scale_and_test_normality`**(*vx*)

Should be applied to a vector of velocities, vx

`multi_locus_analysis.stats.`**`traj_to_msds`**(*traj*, *xcol='x'*, *ycol='y'*, *framecol='frame id'*)

Data should be groupby'd trajectories. Takes x,y coordinate columns and the "time" column, expects integer index for time column.

`multi_locus_analysis.stats.`**`vels_to_cvvs_by_hand`**(*vels*, *groups*, *file_name*, *framecol='ti'*, *dxcol='vx'*, *dycol='vy'*, *dzcol='vz'*, *max_t_over_delta=None*, *max_dt=None*, *allowed_dts=None*, *deltas_of_interest=None*, *include_group=True*)

should be passed a velocities dataframe (as from pos_to_all_vel.

groups vels by groups, optionally ignores group labels to make a flat file with all vvc information with repeats of t,delta pairs as appropriate.

`multi_locus_analysis.stats.`**`vvc_stats_by_hand`**(*file_name*, *groups*, *print_interval=None*, *skip_lines=None*)

Calculates moments using the output of vels_to_cvvs_by_hand.

5.2 Finite window correction

finite_window	For correcting interior- and exterior-censored data

5.2.1 multi_locus_analysis.finite_window

For correcting interior- and exterior-censored data

This module is designed to facilitate working with Markov processes which have been observed over finite windows of time.

Suppose you wish to observe a process that switches between states A and B, and observe the process over a window of time $[0, T]$.

There will be two types of "observed" times, which we we call "interior times" (those when the entire time spent in the state is observed) and "exterior times" (the lengths spent in the state that we observed at the start/end point of our window).

For example, suppose you are observing two fluorescent loci that are either in contact (A) or not (B). If you measure for 5s, and the loci begin in contact, come apart at time t=2s, and rejoin at time t=4s,

```
        A  A  A  A  A  A  B  B  B  B  B  B  A  A  A  A
        | -  - | -  - | -  - | -  - | -  - | -- | -- | ...
        |                                  |
t  =  0s     1s     2s     3s     4s     5s
```

we would say that T = 5s, and you measured one "interior" time of length 2s, and two exterior times, one of length 2s and one of length 1s (at the start and end of the window, respectively).

When histogramming these times, we must apply a statistical correction to the final histogram weights due to the effects of the finite observation window.

The distribution of the exterior times $f_E(t)$ is exactly equal to the survival function of the actual distribution $S(t) = 1 - \int_0^t f(s)ds$ normalized to equal one over the observation interval. No functions are currently included that leverage this, since extracting information from the survival function is likely only worth it if a large fraction of your observations are exterior times.

On the other hand, the interior times distribution is given by $f_I(t) = f(t)(T - t)$ for $t \in [0, T]$. In order to plot the actual shape of $f(t)$ with this biasing removed, we provide the cdf_exact_given_windows function below.

A typical workflow is, given an array of interior times, `t`, and an array of the window sizes each time was observed within, `w`, is to extract the CDF exactly, then optionally convert that to a PDF to display a regular histogram

```
>>> x, cdf = cdf_exact_given_windows(t, w, pad_left_at_x=0)
>>> xp, pdf = bars_given_cdf(x, cdf)
>>> confint = simultaneous_confint_from_cdf(0.05, len(t), x, cdf)
>>> xc, confs = bars_given_confint(x, confint)
>>> plt.plot(xp, pdf, 'k', xc, confs, 'r-.')
```

The remainder of the functions herein are related to extracting confidence intervals for these distributional estimates.

For details of the derivation, see the Biophysical Journal paper in preparation by Beltran and Spakowitz.

`multi_locus_analysis.finite_window.`**`ab_window`**(*rands, window_size, offset, num_replicates=1, states=[0, 1], seed=None, random_state=None*)

Simulate an asynchronous two-state system from time 0 to *window_size*.

Similar to *`multi_locus_analysis.finite_window.ab_window_fast()`*, but designed to work when the means of the distributions being used are hard to calculate.

Simulate asynchronicity by starting the simulation in a uniformly random state at a time $-t_\infty$ (a large negative number).

Note: This number must be specified in the *offset* parameter and if it is not much larger than the means of the waiting times being used, the asynchronicity approximation will be very poor.

The simulation only records between times 0 and window_size.

Parameters

- **`rands`** (*`(2,) List[scipy.stats.rv_continuous]`*) – Callable that takes "random_state" and "size" kwargs that accept a np.random.RandomState seed and a tuple specifying array sizes, resp.
- **`window_size`** (*`float`*) – The width of the window over which the observation takes place
- **`offset`** (*`float`*) – The (negative) time at which to start (in state 0) in order to equilibrate teh simulation state by time t=0.
- **`states`** (*`(2,) array_like`*) – the "names" of each state, default to [0,1]
- **`num_replicates`** (*`int`*) – Number of times to run the simulation, default 1.
- **`seed`** (*`Optional[int]`*) – Random seed to start the simulation with
- **`random_state`** (*`np.random.RandomState`*) – Random state to start the simulation with. Preempts the seed argument.

Returns df – The start/end times of each waiting time simulated. This data frame has columns=['replicate', 'state', 'start_time', 'end_time', 'window_start', 'window_end'].

Return type pd.DataFrame

`multi_locus_analysis.finite_window.`**`ab_window_fast`**(*rands, means, window_size, num_replicates=1, states=[0, 1], seed=None, random_state=None*)

Simulate a two-state system switching between states A and B.

In addition to functions that can generate random waiting times for each state, this "fast" version of the code requires the average waiting times are are means[0], means[1], respectively.

Warning: apparently, `bruno_util.random.strong_default_seed()` is broken (or this function is) because passing a seed does not make the output reproducible.

Parameters

- **`rands`** (*`(2,) List[scipy.stats.rv_continuous]`*) – One of the random variables defined in `scipy.stats`. Alternatively, any callable that takes *random_state* and *size* kwargs. *random_state* should accept a `np.random.RandomState` seed. *size* will be a tuple specifying output shape of random number array requested.
- **`means`** (*`(2,) array_like`*) – average waiting times for each of the states
- **`window_size`** (*`float`*) – the width of the window over which the observation takes place
- **`num_replicates`** (*`int`*) – number of times to run the simulation, default to 1
- **`states`** (*`(2,) array_like`*) – the "names" of each state, default to [0,1]
- **`seed`** (*`np.random.RandomState`*) – state to start the simulation with

Returns df – The start/end times of each waiting time simulated. This data frame has *columns=['replicate', 'state', 'start_time', 'end_time', 'window_start', 'window_end'].*

Return type pd.DataFrame

Notes

Consider the waiting time intersecting the left boundary of the observation window. The left boundary will be a uniform fraction of the way through this wait time. This can easily be seen in the case of finite-variance wait times using CLT and starting the switching process arbitrarily far left of the window of observation, or in general be imposed by requiring time-homogeneity of the experiment.

We use this fact here to speed up correct simulation of time-homogenous windows by directly simulating only the waiting times within the windows instead of also simulating a long run of "pre-equilibrating" waiting times some offset before the window, as in *`ab_window()`*.

`multi_locus_analysis.finite_window.`**`bars_given_cdf`**(*x, cdf*)

takes x, cdf from cdf_exact* functions and makes a plottable histogram by tracing out the PDF. Works well for CDFs that come from observations on a fixed grid, and not well for continuous observations. (i.e. discrete_trajectory_to_wait_times output will work well, but not state_changes_to_wait_times).

`multi_locus_analysis.finite_window.`**`bars_given_confint`**(*x, confint*)

takes x, confint from cdf_exact*, binom.multinomial_proportions_confint (respectively), and rescales confint to correctly fit around the output of bars_given_cdf in an aesthetic way.

`multi_locus_analysis.finite_window.`**`bootstrapped_pmf_confint`**(*n_samples*, *alpha*, *x*, *cdf*, *num_bootstraps=1000*, *bonferroni=True*)

Given an empirical cdf (x, cdf), this function generates bootstrapped error bars that represent, pointwise, the area that a second observation of n_samples (need not equal the number of samples used to generate (x, cdf)) would lie between with probability 1-alpha if it had a true CDF given by the (continuous, linear interpolation of the) empircal CDF.

Parameters

- **`n_samples`** (*`int`*) – How many samples the secondary measurement has. This is the number of data points drawn in each bootstrap iteration.
- **`alpha`** (*`float in [0,1]`*) – 1 - confidence level desired
- **`x`** (*`(N,) array_like`*) – Values at which the empirical CDF was measured
- **`cdf`** (*`(N,) array_like`*) – Values of the empirical CDF
- **`num_bootstraps`** (*`(optional) int`*) – Number of bootstrapping interations to perform. WARNING: scales the memory required for now.
- **`bonferroni`** (*`(optional) bool`*) – Whether to scale alpha based on the number of bins so that the plot can be used to visually assert pointwise statistical significance at the requested alpha.

Returns **confint** – Upper and lower bounds of the confidence interval calculated.

Return type (2,N-1) array_like

`multi_locus_analysis.finite_window.`**`bootstrapped_pmf_confint_bars`**(*n_samples*, *x*, *cdf*, *num_bootstraps=1000*)

same as non-bars version, but returns the actual samples as pmf bars, ready to plot.

`multi_locus_analysis.finite_window.`**`bootstrapped_pmf_from_waits`**(*times*, *window_sizes*, *times_allowed*, *n_samples=None*, *alpha=0.05*, *num_bootstraps=1000*, *bonferroni=True*, *progress_bar=False*, ***kwargs*)

Takes n_samples, num_bootstraps (# iterations), and calculates the pmf of the data num_bootsraps times using n_samples-sized samples drawn with replacement from the wait_times/windows that were passed.

The internal call to cdf_exact_given_windows_quinn needs you to use the times_allowed argument, but all kwargs are forwarded to that function just in case.

`multi_locus_analysis.finite_window.`**`bootstrapped_pmf_from_waits_`**(*n_samples*, *num_bootstraps*, *times*, *window_sizes*, *times_allowed*, *progress_bar=False*, ***kwargs*)

Does shared work of creating non-bar pmfs, used by other bootstrapped_pmf_from_waits_* functions.

`multi_locus_analysis.finite_window.`**`bootstrapped_pmf_from_waits_bars`**(*times, window_sizes, times_allowed, n_samples=None, num_bootstraps=1000, **kwargs*)

Same as bootstrapped_pmf_from_waits, but returns bars_given_cdf, ready to plot results.

`multi_locus_analysis.finite_window.`**`discrete_trajectory_to_wait_times`**(*data, t_col='t', state_col='state'*)

Converts a discrete trajectory to a dataframe containing each wait time, with its start, end, rank order, and the state it's leaving.

Discrete here means that the state of the system was observed at finite time points (on a lattice in time), as opposed to a system where the exact times of transitions between states are known.

Because a discrete trajectory only bounds the wait times, and does not determine their exact lengths (as a continuous trajectory might), additional columns are included that explictly bound the wait times, in addition to returning the "natural" estimate.

Parameters

- **`data`** (*`pd.DataFrame`*) – should have at least states_column and time_column columns, and already be groupby'd so that there's only one "trajectory" within the DataFrame. One row should correspond to an observation at a particular time point.
- **`time_column`** (*`string`*) – the name of the column containing the time of each time point
- **`states_column`** (*`string`*) – the name of the column containing the state for each time point

Returns

wait_df – columns are ['wait_time', 'start_time', 'end_time', 'wait_state', 'wait_type', 'min_waits', 'max_waits'], where [wait,end,start]_time columns are self explanatory, wait_state is the value of the states_column during that waiting time, and wait_type is one of 'interior', 'left exterior', 'right exterior', 'full exterior', depending on what kind of waiting time was observed. See the *Notes* section below for detailed explanation of these categories. The 'min/max_waits' columns contain the minimum/maximum possible value of the wait time (resp.), given the observations.

The default index is named "rank_order", since it tracks the order (zero-indexed) in which the wait times occured.

Return type pd.DataFrame

Notes

the following types of wait times are of interest to us

1) *interior* censored times: whenever you are observing a switching process for a finite amount of time, any waiting time you observe the entirety of is called "interior" censored

2) *left exterior* censored times: whenever the waiting time you observe started before you began observation (it overlaps the "left" side of your interval of observation)

3) *right exterior* censored times: same as above, but overlappign the "right" side of your interval of observation.

4) *full exterior* censored times: whenever you observe the existence of a single, particular state throughout your entire window of observation.

`multi_locus_analysis.finite_window.`**`ecdf`**(*y*, *y_allowed=None*, *auto_pad_left=False*, *pad_left_at_x=None*)

Compute empirical cumulative distribution function (eCDF) from data.

Parameters

- **`y`** (*`(N,) array_like`*) – Values of the data.
- **`y_allowed`** (*`(M,) array_like`*) – Unique values that the data can take. Mostly useful for adding eCDF values at locations where data could or should have been observed but none was recorded.
- **`auto_pad_left`** (*`bool`*) – If left False, the data will not have a data value at the point where the eCDF equals zero. Use mean inter-data spacing to automatically generate an aesthetically reasonable such point.
- **`pad_left_at_x`** (*`bool`*) – Same as `auto_pad_left`, but specify the point at which to add the leftmost point.

Returns

- **x** (*(M,) array_like*) – The values at which the eCDF was computed. By default `np.sort(np.unique(y))`.
- **cdf** (*(M,) array_like*) – Values of the eCDF at each x.

Notes

If using `y_allowed`, the *pad_left* parameters are redundant.

`multi_locus_analysis.finite_window.`**`ecdf_windowed`**(*times*, *window_sizes*, *times_allowed=None*, *auto_pad_left=None*, *pad_left_at_x=None*, *window_func=None*)

Compute empirical cumulative distribution function (eCDF) from data taken within a finite observation interval.

Parameters

- **`times`** (*`(N,) array_like`*) – "Interior" waiting times.
- **`window_sizes`** (*`float or (N,) array_like`*) – The window size used. If a single value is passed, the window size is assumed to be constant.
- **`times_allowed`** (*`(M,) array_like`*) – Unique values that the data can take. Mostly useful for adding eCDF values at locations where data could or should have been observed but none was recorded (i.e. if a movie was taken with a given framerate but not all possible window lengths were observed.
- **`auto_pad_left`** (*`bool`*) – If left False, the data will not have a data value at the point where the eCDF equals zero. Use mean inter-data spacing to automatically generate an aesthetically reasonable such point.
- **`pad_left_at_x`** (*`bool`*) – Same as `auto_pad_left`, but specify the point at which to add the leftmost point.
- **`window_func`** (*`function`*) – The function used to reweight the observations. The default is equivalent to `lambda yi:  np.sum(np.heaviside(ymax - yi, 0)*(ymax - yi)))`.

Returns

- **x** (*(M,) array_like*) – The values at which the eCDF was computed. By default `np.sort(np.unique(y))`.
- **cdf** (*(M,) array_like*) – Values of the eCDF at each x.

Notes

If using `times_allowed`, the *pad_left* parameters are redundant.

`multi_locus_analysis.finite_window.`**`movie_to_waits`**`(`**args*, ***kwargs*`)`
Alias of *`multi_locus_analysis.finite_window.discrete_trajectory_to_wait_times()`*

`multi_locus_analysis.finite_window.`**`sample_from_cdf`**`(`*n*, *x*, *cdf*`)`
Takes a sample count, cdf in the form x,cdf, like from output of cdf_exact* functions [i.e. pairs of (x, P(X<=x))]. Samples from the empirical distribution function at the maximum "x" resolution allowed by x.

Returns fraction of the resampled data that fell into each bin. In other words, it returns pmf, as if one had done:

```
>>> pmf, x = np.histogram(samples, bins=x)
```

`multi_locus_analysis.finite_window.`**`smooth_pdf`**`(`*x*, *cdf*, *bw_method=None*`)`
Takes x, cdf from cdf_exact* and returns a kernel density estimator that can be evaluated at any X to get an estimate of pdf(X).

use bw_method to specify the way scipy.stats.gaussian_kde should determine the bandwidth of the gaussian.

`multi_locus_analysis.finite_window.`**`state_changes_to_trajectory`**`(`*traj*, *times*, *state_col='state'*, *start_times_col='start_time'*, *end_times_col=None*`)`
Converts a series of state change times into a Series containing observations at the times requested. The times become the index.

Parameters

- **`times`** (*`(N,) array_like`*) – times at which to "measure" what state we're in to make the new trajectories.
- **`traj`** (*`pd.DataFrame`*) – should have *state_col* and *start_times_col* columns. the values of *state_col* will be copied over verbatim.
- **`state_col`** (*`string`*) – name of column containing the state being transitioned out of for each measurement in *traj*.
- **`start_times_col`** (*`string`*) – name of column containing times at which *traj* changed state
- **`end_times_col`** (*`(optional) string`*) – by default, the function assumes that times after the last start time are in the same state. if passed, this column is used to determine at what time the last state "finished". times after this will be labeled as NaN.

Returns **movie** – Series defining the "movie" with frames taken at *times* that simply measures what state *traj* is in at each frame. index is *times*, *state_col* is used to name the Series.

Return type pd.Series

Notes

A start time means that if we observe at that time, the state transition will have already happened (right-continuity). This is confusing in words, but simple to see in an example (see the example below).

Examples

For the DataFrame

```
>>> df = pd.DataFrame([['A',  -1, 0.1], ['B', 0.1, 1.0]],
>>>     columns=['state', 'start_time', 'end_time'])
```

the discretization into tenths of seconds would give

```
>>> state_changes_to_trajectory(df, times=np.linspace(0, 1, 11),
>>>     end_times_col='end_time')
t
0.0       A
0.1       B
0.2       B
0.3       B
0.4       B
0.5       B
0.6       B
0.7       B
0.8       B
0.9       B
1.0     NaN
Name: state, dtype: object
```

Notice in particular how at 0.1, the state is already 'B'. Similarly at time 1.0 the state is already "unknown". This is what is meant by the Notes section above.

If the *end_times_col* argument is omitted, then the last observed state is assumed to continue for all *times* requested from then on:

```
>>> state_changes_to_trajectory(df, times=np.linspace(0, 1, 11))
t
0.0    A
0.1    B
0.2    B
0.3    B
0.4    B
0.5    B
0.6    B
0.7    B
0.8    B
0.9    B
1.0    B
Name: state, dtype: object
```

`multi_locus_analysis.finite_window.`**`state_changes_to_wait_times`**(*traj*)

Converts the output of `ab_window_fast()` into a `pd.DataFrame` containing each wait time, with its start, end, rank order, and the state it's leaving.

This function deals with "continuous" wait times, as in not measured at discrete time points (on a grid), so the wait times it returns are exact.

`multi_locus_analysis.finite_window.`**`traj_to_movie`**(**args*, ***kwargs*)

Alias of `multi_locus_analysis.finite_window.state_changes_to_trajectory()`

`multi_locus_analysis.finite_window.`**`traj_to_waits`**(**args*, ***kwargs*)

Alias of `multi_locus_analysis.finite_window.state_changes_to_wait_times()`

5.3 Plotting

`plotting`	For plotting multi_locus_analysis results

5.3.1 multi_locus_analysis.plotting

For plotting multi_locus_analysis results

`multi_locus_analysis.plotting.`**`cvv_plot_sized`**(*cvvs*, *analytical_deltas=[]*, *delta_col='delta'*, *t_col='t'*, *cvv_col='cvv_normed'*, *max_t_over_delta=4*, *data_deltas=None*, *A=1*, *beta=0.5*, *tDeltaN=None*, *cmap_name='viridis'*, *fig=None*, *alpha_map=None*, *size_map=None*, *theory_linewidth=2*, *include_errorbar=True*, *include_lines=False*, *include_points=False*, *data_line_alpha=1*, *data_linewidth=1*, *BASE_DOT_SIZE=10000*, ***kwargs*)

One pretty version of the Velocity-Velocity correlation plots for the experimental data, with some theory overlaid.

`multi_locus_analysis.plotting.`**`make_all_disps_hist`**(*displacements*, *centering='mean, std'*, *cmap=None*, *reverse=False*, *alpha=1*, *cmap_log=False*, *factor_by=None*, *xlim=None*, *ylim=None*, *include_theory_data=False*, *include_theory_exact=True*, *vxcol='vx'*, *vycol='vy'*, *max_plots=inf*, *laplace=True*, *normal=True*, *axs=None*, *no_tick_labels=False*, *cbar=True*, *xbins=None*, *frames_to_sec=None*, *yscale='log'*, *omit_title=True*)

Make a manual factor plot of displacement histograms.

Parameters

- **`factor_by`** (*`List<str>`*) – list of level numbers into the displacements heirarchical index telling which set of levels to factor on.
- **`centering`** (*`str`*) – controls how to center the data so that it fits on a single plot. one of "none", "mean", "mean,std" for nothing, mean subtraction, and mean subtraction+dividing by std deviation (respectively).
- **`reverse`** (*`bool`*) – True plots smallest deltas on top, False the opposite

5.4 Diffusion theory

`analytical`	For computing analytical results relevant to diffusing loci

5.4.1 multi_locus_analysis.analytical

For computing analytical results relevant to diffusing loci

`multi_locus_analysis.analytical.`**`frac_cv`**(*t*, *alpha*, *kbT=1*, *xi=1*)
Velocity autocorrelation of a fractionally-diffusing particle. Weber, Phys Rev E, 2010 (Eq 32)

`multi_locus_analysis.analytical.`**`frac_discrete_cv`**(*t*, *delta*, *alpha*, *kbT=1*, *xi=1*)
Discrete velocity autocorrelation of a fractionally-diffusing particle. Weber, Phys Rev E, 2010 (Eq 33)

`multi_locus_analysis.analytical.`**`frac_discrete_cv_normalized`**(*t*, *delta*, *alpha*)
Normalized discrete velocity autocorrelation of a fractionally-diffusing particle. Should be equivalent to

frac_discrete_cv(t, delta, 1, 1)/frac_discrete_cv(0, delta, 1, 1)

Lampo, BPJ, 2016 (Eq 5)

`multi_locus_analysis.analytical.`**`frac_msd`**(*t*, *alpha*, *kbT=1*, *xi=1*)
MSD of fractionally diffusing free particle.

Weber, Phys Rev E, 2010 (Eq 10)

`multi_locus_analysis.analytical.`**`rouse_cv_mid`**(*t*, *alpha*, *b*, *N*, *kbT=1*, *xi=1*, *min_modes=1000*)
Velocity autocorrelation of midpoint of a rouse polymer.

Weber Phys Rev E 2010, Eq. 33.

`multi_locus_analysis.analytical.`**`rouse_cvv`**(*t*, *delta*, *n1*, *n2*, *alpha*, *b*, *N*, *kbT=1*, *xi=1*, *min_modes=500*, *rtol=1e-05*, *atol=1e-08*, *force_convergence=True*)
Velocity cross-correlation of two points on fractional Rouse polymer

rtol/atol specify when to stop adding rouse modes. a particle (t,delta) pair is considered to have converged when np.isclose returns true give rtol/atol and p is even (p odd contributes almost nothing)

Lampo, BPJ, 2016 Eq. 10.

`multi_locus_analysis.analytical.`**`rouse_cvv_ep`**(*t*, *delta*, *p*, *alpha*, *b*, *N*, *kbT=1*, *xi=1*)
Term in parenthesis in Lampo, BPJ, 2016 Eq. 10.

`multi_locus_analysis.analytical.`**`rouse_large_cvv_g`**(*t*, *delta*, *deltaN*, *b*, *kbT=1*, *xi=1*)
Cvv^delta(t) for infinite polymer.

Lampo, BPJ, 2016 Eq. 16.

`multi_locus_analysis.analytical.`**`rouse_mid_msd`**(*t*, *alpha*, *b*, *N*, *kbT=1*, *xi=1*, *num_modes=1000*)
Weber Phys Rev E 2010, Eq. 24.

`multi_locus_analysis.analytical.`**`rouse_mode`**
Eigenbasis for Rouse model.

Indexed by p, depends only on position n/N along the polymer of length N. N=1 by default.

Weber, Phys Rev E, 2010 (Eq 14)

`multi_locus_analysis.analytical.`**`rouse_mode_coef`**(*p*, *b*, *N*, *kbT=1*)
k_p: Weber Phys Rev E 2010, after Eq. 18.

`multi_locus_analysis.analytical.`**`rouse_mode_corr`**(*p*, *t*, *alpha*, *b*, *N*, *kbT=1*, *xi=1*)
Weber Phys Rev E 2010, Eq. 21.

`multi_locus_analysis.analytical.`**`rouse_nondim_`**(*t*, *delta*, *n1*, *n2*, *alpha*, *b*, *N*, *kbT=1*, *xi=1*)
uses parameters defined by Lampo et al, BPJ, 2016, eq 12

`multi_locus_analysis.analytical.`**`simple_rouse_mid_msd`**(*t*, *b*, *N*, *kbT=1*, *xi=1*, *num_modes=1000*)
modified from Weber Phys Rev E 2010, Eq. 24.

`multi_locus_analysis.analytical.`**`tDeltaN`**(*n1*, *n2*, *alpha*, *b*, *kbT*, *xi*)
Lampo et al, BPJ, 2016, eq 11

`multi_locus_analysis.analytical.`**`tR`**(*alpha*, *b*, *N*, *kbT=1*, *xi=1*)
Lampo et al, BPJ, 2016, eq 8

`multi_locus_analysis.analytical.`**`un_rouse_nondim`**(*tOverDelta*, *deltaOverTDeltaN*, *alpha*, *delN=0.001*)
Takes a requested choice of parameters to compare to Tom's calculated values and generates a full parameter list that satisfies those requirements.

uses approximation defined by Lampo et al, BPJ, 2016, eq 12

In Tom's paper, delN is non-dimensinoalized away into deltaOverTDeltaN, but that only works in the case where the chain is infinitely long. Otherwise, where exactly we choose n1,n2 will affect the velocity correlation because of the effects of the chain ends. While other choices are truly arbitrary (e.g. kbT, N, b, etc), delN can be used to specify the size of the segment (relative to the whole chain) used to compute the cross correlation.

`multi_locus_analysis.analytical.`**`vc`**(*t*, *delta*, *beta*)
velocity correlation of locus on rouse polymer. beta = alpha/2.

`multi_locus_analysis.analytical.`**`vvc_rescaled_theory`**(*t*, *delta*, *beta*, *A*, *tDeltaN*)
velocity cross correlation of two points on rouse polymer.

`multi_locus_analysis.analytical.`**`vvc_unscaled_theory`**(*t*, *delta*, *beta*, *A*, *tDeltaN*)
velocity cross correlation of two points on rouse polymer.

5.5 Fitting helpers

`fitting`	For fitting analytical theory to multi_locus_analysis results

5.5.1 multi_locus_analysis.fitting

For fitting analytical theory to multi_locus_analysis results

`multi_locus_analysis.fitting.`**`fit_full_fix_beta`**(*tdelta*, *A*, *tDeltaN*, *beta=0.5*)
Fit velocity cross-correlation to the case of two loci on a polymer, freely diffusing, but attached to each other (so beta = alpha/2 = 1/2).

multi_locus_analysis.fitting.**get_best_fit_fixed_beta**(*df, t_col='t', delta_col='delta', cvv_col='cvv_normed', ste_col='ste', p0=[1, 10], bounds=([0, 1.5], [10, 1000]), counts_col=None, beta=0.5, hack=False*)

beta = alpha/2

5.6 Example data

`examples`	Data Examples, Multi-locus Analysis
`examples.burgess`	

5.6.1 multi_locus_analysis.examples

Data Examples, Multi-locus Analysis

Current example data sets included are:

1. `multi_locus_analysis.examples.burgess` dataset. Fluorescent tracking of two homologous loci as a population of budding yeast cells progress through meiosis. (single color)

2. TODO… add a couple of test data sets for use in our testing code.

5.7 Utilities

`dataframes`	Utilities for massaging DataFrames

5.7.1 multi_locus_analysis.dataframes

Utilities for massaging DataFrames

multi_locus_analysis.dataframes.**array_from_numpy_string**(*s*)

For unserializing np.array's after DataFrame.to_csv

Notes

As of 2019-03-13, looks like instead of commas separating the numpy array elements, *two* spaces are printed. We just replace these spaces with commas after removing all other spaces and load the string as json.

multi_locus_analysis.dataframes.**pivot_loci**(*df, pivot_cols=['x', 'y', 'z'], spot_col='spot'*)

Move between "long" and "short" forms for the spot id column.

Simply put, we want to be able to transform between the following two dataframes:

```
Condensed form
                                          X1   Y1   Z1   X2   Y2   Z2     t  foci
locus genotype exp.rep meiosis cell frame
HET5  WT       2       t0      1    1     1.6  2.4  3.1  2.1  1.5  3.1    0   unp
                                    2     1.9  1.5  3.1  1.9  2.5  3.1   30   unp
                                    3     2.0  1.8  3.0  1.5  2.5  3.4   60   unp
```

(continues on next page)

(continued from previous page)

```
                                    4      2.1  1.9  3.0  1.4  2.2  3.4   90  unp
                                    5      2.2  1.8  3.0  1.5  2.4  3.4  120  unp
```

and

```
"Long" form
                                                X    Y    Z      t  foci
locus genotype exp.rep meiosis cell frame spot
HET5  WT       2       t0      1    1     1     1.6  2.4  3.1    0  unp
                                    2     1     1.9  1.5  3.1   30  unp
                                    3     1     2.0  1.8  3.0   60  unp
                                    ...
                                    1     2     2.1  1.5  3.1    0  unp
                                    2     2     1.9  2.5  3.1   30  unp
                                    3     2     1.5  2.5  3.4   60  unp
```

This function can infer which direction to pivot. Because of this, I have found using this function much more convenient (and a smaller cognitive load) than using a multiindex for the column names and using e.g. pd.unstack and friends.

Parameters

- **`pivot_cols`** (*List<str>*) – The names of the columns over which to pivot (without their numerical suffixes, these will be inferred).
- **`spot_col`** (*str*) – The name of the column that holds (or will hold) the spot id.

Returns **df** – The pivot-ed DataFrame.

Return type pd.DataFrame

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index