# saspt

***Release 1.0***

**Alec Heckert**

**Oct 01, 2022**

# CONTENTS

`saspt` is a Python tool for analyzing single particle tracking (SPT) experiments. It uses *state arrays*, a kind of variational Bayesian framework that learns intelligible models given raw trajectories from an SPT experiment.

There are a lot of great SPT analysis tools out there. We found it useful to write `saspt` because:

- it is simple and flexible enough to accommodate a wide variety of underlying stochastic models;
- its outputs are familiar `numpy` and `pandas` objects;
- it imposes no prior beliefs on the number of dynamic states;
- it uses tried-and-true Bayesian methods (marginalization over nuisance parameters) to deal with measurement error in a natural way.

I originally wrote `saspt` to deal with live cell protein tracking experiments. In the complex intracellular environment, a protein can occupy a large and unknown number of molecular states with distinct dynamics. `saspt` provides a simple way to measure the number, characteristics, and fractional occupations of these states. It is also "smart" enough to deal with situations where there may *not* be discrete states.

If you want to jump right into working with `saspt`, see *Quickstart*. If you want a more detailed explanation of why `saspt` exists, see *Background*. If you want to dig into the guts of the actual model and inference algorithm, see *The state array model*.

(`saspt` stands for "state arrays for single particle tracking".)

CHAPTER

ONE

# WHAT DOES SASPT DO?

`saspt` takes a set of trajectories from a tracking experiment, and identifies a mixture model to explain them. It is designed to work natively with `numpy` and `pandas` objects.

CHAPTER

# TWO

# WHAT DOESN'T SASPT DO?

1. `saspt` doesn't do tracking; it takes trajectories as input. (See: *Q. Does saspt provide a way to do tracking?*)
2. `saspt` doesn't model *transitions between states*. For that purpose, we recommend the excellent vbSPT package.
3. `saspt` doesn't check the quality of the input data.
4. `saspt` expects you to know the parameters for your imaging experiment, including pixel size, frame rate, and focal depth.

Currently `saspt` only supports a small range of physical models. That may change as the package grows.

## 2.1 Install

`saspt` has only been tested with Python 3.

### 2.1.1 Option 1: install with pip

```
pip install saspt
```

### 2.1.2 Option 2: install from source

1. Clone the `saspt` repo:

```
git clone https://github.com/alecheckert/saspt.git
cd saspt
```

2. If you are using the `conda` package manager, you build and switch to the `saspt_env` conda environment with all the necessary dependencies:

```
conda env create -f example_env.yaml
conda activate saspt_env
```

3. Install the `saspt` package:

```
pip install .
```

We recommend running the testing suite after installing:

```
pytest tests
```

### 2.1.3 Dependencies

- `numpy`
- `scipy`
- `dask`
- `pandas`
- `matplotlib`
- `tqdm` (pip)
- `seaborn`

All dependencies are available via `conda` using either the `defaults` or `conda-forge` channels (example environment spec).

## 2.2 Quickstart

**Note:** Before running, see *Install* for installation instructions.

**Note:** Want more code and less prose? Check out examples.py, a simple executable that demos most of stuff in this Quickstart.

This is a quick guide to getting started with `saspt`. It assumes you're familiar with single particle tracking (SPT), have seen mixture models before, and have installed `saspt`. For a more detailed explanation of `saspt`'s purpose, model, and range of applicability, see *Background*.

### 2.2.1 Run state arrays on a single SPT experiment

We'll use the sample set of trajectories that comes with `saspt`:

```
>>> import pandas as pd, numpy as np, matplotlib.pyplot as plt
>>> from saspt import sample_detections, StateArray, RBME
>>> detections = sample_detections()
```

The expected format for input trajectories is described under *Note on input format*. **Importantly, the units for the XY coordinates are in pixels, not microns.**

Next, we'll set the parameters for a state array that uses mixtures of regular Brownian motions with localization error (RBMEs):

```
>>> settings = dict(
...     likelihood_type = RBME,
...     pixel_size_um = 0.122,
...     frame_interval = 0.01,
...     focal_depth = 0.7,
...     progress_bar = True,
... )
```

Only three parameters are actually required:

- `likelihood_type`: the type of physical model to use
- `pixel_size_um`: size of camera pixels after magnification in microns
- `frame_interval`: time between frames in seconds

Additionally, we've set the `focal_depth`, which accounts for the finite focal depth of high-NA objectives used for SPT experiments, and `progress_bar`, which shows progress during inference. A full list of parameters and their meaning is described at *StateArrayParameters*.

Construct a `StateArray` with these settings:

```
>>> SA = StateArray.from_detections(detections, **settings)
```

If we run `print(SA)`, we get

```
StateArray:
  likelihood_type   : rbme
  n_tracks          : 1130
  n_jumps           : 3933
  parameter_names   : ('diff_coef', 'loc_error')
  shape             : (100, 36)
```

This means that this state array infers occupations on a 2D parameter grid of diffusion coefficient (`diff_coef`) and localization error (`loc_error`), using 1130 trajectories. The shape of the parameter grid is `(100, 36)`, meaning that the grid uses 100 distinct diffusion coefficients and 36 distinct localization errors (the default). These define the range of physical models that can be described with this state array. We can get the values of these parameters using the `StateArray.parameter_values` attribute:

```
>>> diff_coefs, loc_errors = SA.parameter_values
>>> print(diff_coefs.shape)
(100,)
>>> print(loc_errors.shape)
(36,)
```

The `StateArray` object provides two estimates of the state occupations at each point on this parameter grid:

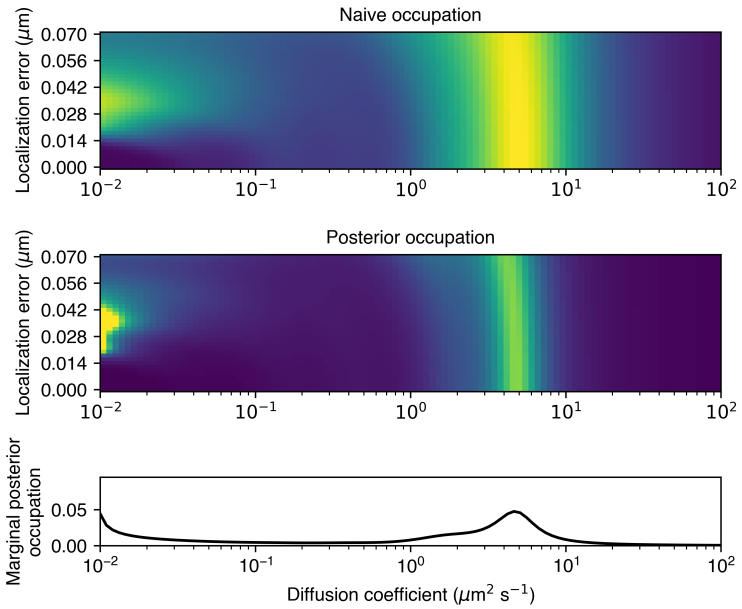
- The "naive" estimate, a quick and dirty estimate from the raw likelihood function
- The "posterior" estimate, which uses the full state array model

The posterior estimate is more precise than the naive estimate, but also requires more trajectories and time. The more trajectories are present in the input, the more precise the posterior estimate becomes.

The `StateArray` object provides a built-in plot to compare the naive and posterior estimates:

```
>>> SA.plot_occupations("rbme_occupations.png")
```

The plot will look something like this:




The bottom row shows the posterior occupations marginalized on diffusion coefficient. This is a simple and powerful mechanism to account for the influence of localization error.

In this case, the state array identified a dominant diffusive state with a diffusion coefficient of about 5 $\mu$m$^2$/sec. We can also see a less-populated state between about 1 and 3 $\mu$m$^2$/sec, and some very slow particles with diffusion coefficients in the range 0.01 to 0.1 $\mu$m$^2$/sec.

We can retrieve the raw arrays used in this plot via the `naive_occs` and `posterior_occs` attributes. Both are arrays defined on the same grid of diffusion coefficient vs. localization error:

```
>>> naive_occs = SA.naive_occs
>>> posterior_occs = SA.posterior_occs
>>> print(naive_occs.shape)
(100, 36)
>>> print(posterior_occs.shape)
(100, 36)
```

Along with the state occupations, the `StateArray` object also infers the probabilities of each *trajectory-state assignment*. As with the state occupations, the trajectory-state assignment probabilities have both "naive" and "posterior" versions that we can compare:

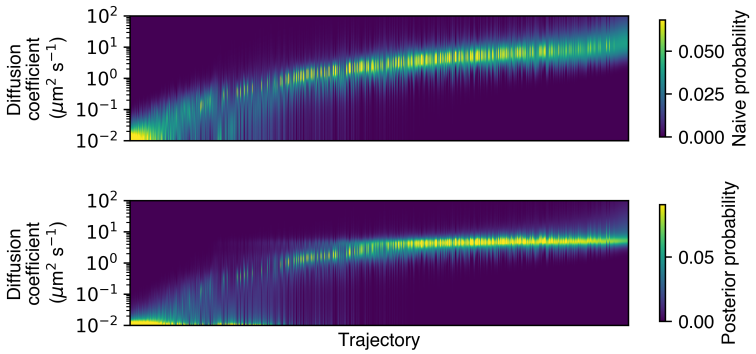
```
>>> naive_assignment_probabilities = SA.naive_assignment_probabilities
>>> posterior_assignment_probabilities = SA.posterior_assignment_probabilities
>>> print(naive_assignment_probabilities.shape)
(100, 36, 1130)
>>> print(posterior_assignment_probabilities.shape)
(100, 36, 1130)
```

Notice that these arrays have one element per point in our 100-by-36 parameter grid and per trajectory. For example, the marginal probability that trajectory 100 has each of the 100 diffusion coefficients is:

```
>>> posterior_assignment_probabilities[:,:,100].sum(axis=1)
```

`StateArray` provides a plot to compare the naive and posterior assignment probabilities across all trajectories:

```
>>> SA.plot_assignment_probabilities('rbme_assignment_probabilities.png')
```




Each column in this plot represents a single trajectory, and the rows represent the probability of the trajectories having a particular diffusion coefficient. (The trajectories are sorted by their posterior mean diffusion coefficient.)

**There are also a couple of related plots (not illustrated here):**

- `saspt.StateArray.plot_temporal_assignment_probabilities()`: shows the assignment probabilities as a function of the frame(s) in which the respective trajectories were found
- `saspt.StateArray.plot_spatial_assignment_probabilities()`: shows the assignment probabilities as a function of the spatial location of the component detections

Finally, `StateArray` provides the naive and posterior state occupations as a `pandas.DataFrame`:

```
>>> occupations = SA.occupations_dataframe
>>> print(occupations)
      diff_coef  loc_error  naive_occupation  mean_posterior_occupation
0          0.01      0.000          0.000017                   0.000009
1          0.01      0.002          0.000017                   0.000008
2          0.01      0.004          0.000016                   0.000008
...         ...        ...               ...                        ...
3597     100.00      0.066          0.000042                   0.000014
3598     100.00      0.068          0.000041                   0.000014
3599     100.00      0.070          0.000041                   0.000014

[3600 rows x 4 columns]
```

Each row corresponds to a single point on the parameter grid. For instance, if we wanted to get the probability that a particle has a diffusion coefficient less than 0.1 $\mu m^2$/sec, we could do:

```
>>> selected = occupations['diff_coef'] < 0.1
>>> naive_estimate = occupations.loc[selected, 'naive_occupation'].sum()
>>> posterior_estimate = occupations.loc[selected, 'mean_posterior_occupation'].sum()
>>> print(naive_estimate)
0.24171198737935867
>>> print(posterior_estimate)
0.2779671727562628
```

In this case, the naive and posterior estimates are quite similar.

### 2.2.2 Run state arrays on a SPT dataset

Often we want to run state arrays on more than one SPT experiment and compare the output between experimental conditions. The `StateArrayDataset` object is intended to be a simple solution that provides:

- methods to parallelize state array inference across multiple SPT experiments
- outputs and visualizations to help compare between experimental conditions

In this example, we'll use an example from the saspt repo. You can follow along by cloning the `saspt` repo and navigating to the `examples` subdirectory:

```
$ git clone https://github.com/alecheckert/saspt.git
$ cd saspt/examples
$ ls -1
examples.py
experiment_conditions.csv
u2os_ht_nls_7.48ms
u2os_rara_ht_7.48ms
```

**The `examples` subdirectory contains a small SPT dataset where two proteins have been tracked:**

- `HT-NLS`: HaloTag (HT) fused to a nuclear localization signal (NLS), labeled with the photoactivatable fluorescent dye PA-JFX549
- `RARA-HT`: retinoic acid receptor $\alpha$ (RARA) fused to HaloTag (HT), labeled with the photoactivatable fluorescent dye PA-JFX549

Each protein has 11 SPT experiments, stored as CSV files in the `examples/u2os_ht_nls_7.48ms` and `examples/u2os_rara_ht_7.48ms` subdirectories. We also have a registry file (`experiment_conditions.csv`) that contains the assignment of each file to an experimental condition:

```
>>> paths = pd.read_csv('experiment_conditions.csv')
```

In this case, we have two columns: `filepath` encodes the path to the CSV corresponding to each SPT experiment, while `condition` encodes the experimental condition. (It doesn't actually matter what these are named as long as they match the `path_col` and `condition_col` parameters below.)

```
>>> print(paths)
                                    filepath     condition
0    u2os_ht_nls_7.48ms/region_0_7ms_trajs.csv   HaloTag-NLS
1   u2os_ht_nls_7.48ms/region_10_7ms_trajs.csv   HaloTag-NLS
2    u2os_ht_nls_7.48ms/region_1_7ms_trajs.csv   HaloTag-NLS
..                                         ...           ...
19  u2os_rara_ht_7.48ms/region_7_7ms_trajs.csv  RARA-HaloTag
20  u2os_rara_ht_7.48ms/region_8_7ms_trajs.csv  RARA-HaloTag
21  u2os_rara_ht_7.48ms/region_9_7ms_trajs.csv  RARA-HaloTag

[22 rows x 2 columns]
```

Specify some parameters related to this analysis:

```
>>> settings = dict(
...     likelihood_type = RBME,
...     pixel_size_um = 0.16,
...     frame_interval = 0.00748,
...     focal_depth = 0.7,
```

(continues on next page)

(continued from previous page)

```
...     path_col = 'filepath',
...     condition_col = 'condition',
...     progress_bar = True,
...     num_workers = 6,
... )
```

**Warning:** The `num_workers` attribute specifies the number of parallel processes to use when running inference. Don't set this higher than the number of CPUs on your computer, or you're likely to suffer performance hits.

Create a `StateArrayDataset` with these settings:

```
>>> from saspt import StateArrayDataset
>>> SAD = StateArrayDataset.from_kwargs(paths, **settings)
```

If you do `print(SAD)`, you'll get some basic info on this dataset:

```
>>> print(SAD)
StateArrayDataset:
  likelihood_type    : rbme
  shape              : (100, 36)
  n_files            : 22
  path_col           : filepath
  condition_col      : condition
  conditions         : ['HaloTag-NLS' 'RARA-HaloTag']
```

We can get more detailed information on these experiments (such as the detection density, mean trajectory length, etc.) by accessing the `raw_track_statistics` attribute:

```
>>> stats = SAD.raw_track_statistics
>>> print(stats)
    n_tracks  n_jumps  ...                                  filepath     condition
0       2387     1520  ...   u2os_ht_nls_7.48ms/region_0_7ms_trajs.csv   HaloTag-NLS
1       4966     5341  ...  u2os_ht_nls_7.48ms/region_10_7ms_trajs.csv   HaloTag-NLS
2       3294     2584  ...   u2os_ht_nls_7.48ms/region_1_7ms_trajs.csv   HaloTag-NLS
..       ...      ...  ...                                        ...          ...
19      5418    13129  ...  u2os_rara_ht_7.48ms/region_7_7ms_trajs.csv  RARA-HaloTag
20      9814    26323  ...  u2os_rara_ht_7.48ms/region_8_7ms_trajs.csv  RARA-HaloTag
21      7530    18978  ...  u2os_rara_ht_7.48ms/region_9_7ms_trajs.csv  RARA-HaloTag

[22 rows x 13 columns]
>>> print(stats.columns)
Index(['n_tracks', 'n_jumps', 'n_detections', 'mean_track_length',
       'max_track_length', 'fraction_singlets', 'fraction_unassigned',
       'mean_jumps_per_track', 'mean_detections_per_frame',
       'max_detections_per_frame', 'fraction_of_frames_with_detections',
       'filepath', 'condition'],
      dtype='object')
```

To get the naive and posterior state occupations for each file in this dataset:

```
>>> marginal_naive_occs = SAD.marginal_naive_occs
>>> marginal_posterior_occs = SAD.marginal_posterior_occs
>>> print(marginal_naive_occs.shape)
>>> print(marginal_posterior_occs.shape)
```

**Note:** It can take a few minutes to compute the posterior occupations for a dataset of this size. If you need a quick estimate for a test, try reducing the `max_iter` or `sample_size` parameters.

These occupations are "marginal" in the sense that they've been marginalized onto the parameter of interest in most SPT experiments: the diffusion coefficient. (You can get the original, unmarginalized occupations via the `StateArrayDataset.posterior_occs` and `StateArrayDataset.naive_occs` attributes.)

The same information is also provided as a `pandas.DataFrame`:

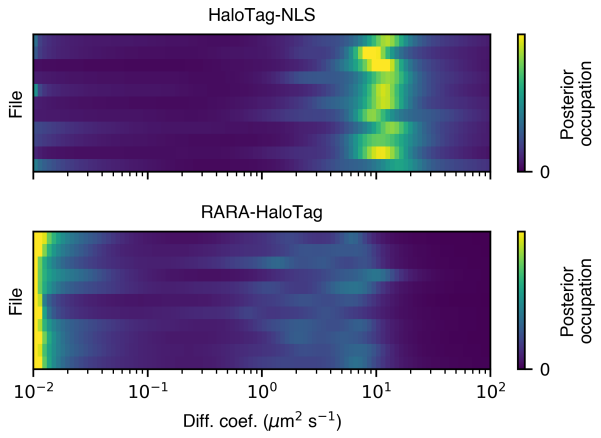
```
>>> occupations = SAD.marginal_posterior_occs_dataframe
```

For example, imagine we want to calculate the posterior probability that a particle had a diffusion coefficient less than 0.5 $\mu$m$^2$/sec for each file. We could do this by taking

```
>>> print(occupations.loc[occupations['diff_coef'] < 0.5].groupby(
...   'filepath')['mean_posterior_occupation'].sum())
filepath
u2os_ht_nls_7.48ms/region_0_7ms_trajs.csv      0.188782
u2os_ht_nls_7.48ms/region_10_7ms_trajs.csv     0.103510
u2os_ht_nls_7.48ms/region_1_7ms_trajs.csv      0.091148
...
u2os_rara_ht_7.48ms/region_7_7ms_trajs.csv     0.579444
u2os_rara_ht_7.48ms/region_8_7ms_trajs.csv     0.553111
u2os_rara_ht_7.48ms/region_9_7ms_trajs.csv     0.650187
Name: posterior_occupation, dtype: float64
```

The `StateArrayDataset` provides a few plots to visualize these occupations:

```
>>> SAD.posterior_heat_map('posterior_heat_map.png')
```

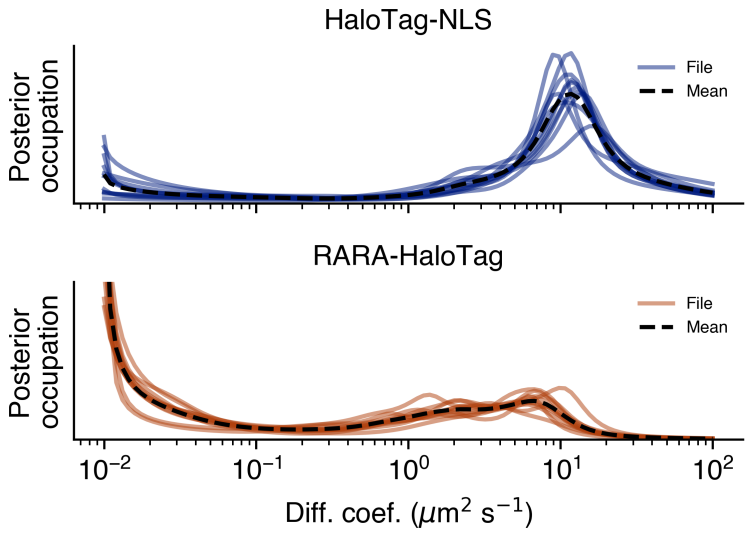



Notice that the two kinds of proteins have different diffusive profiles: HaloTag-NLS occupies a narrow range of diffusion coefficients centered around 10 $\mu$m$^2$/sec, while RARA-HaloTag has a much broader range of free diffusion coefficients

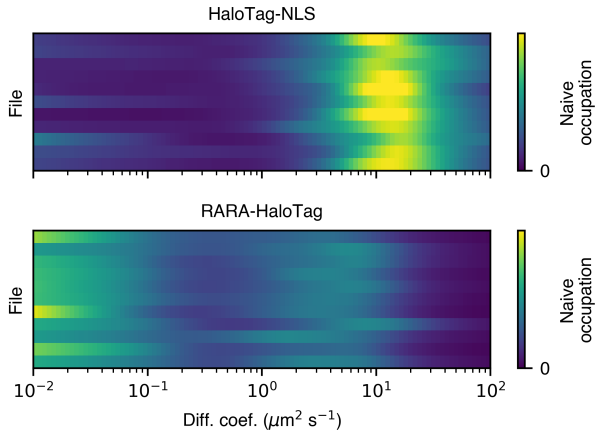
with a substantial immobile fraction (showing up at the lower end of the diffusion coefficient range).

The heat map plot is useful to judge how consistent the result is across SPT experiments in the same condition. We can also compare the variability using an alternative line plot representation:

```
>>> SAD.posterior_line_plot('posterior_line_plot.png')
```

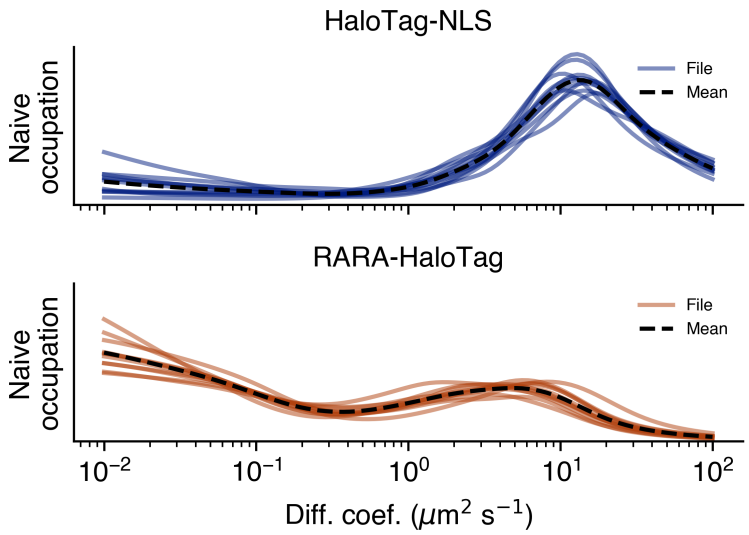



```
>>> SAD.naive_heat_map('naive_heat_map.png')
```




Notice that the information provided by the naive occupations is qualitatively similar but less precise than the posterior occupations.

```
>>> SAD.naive_line_plot('naive_line_plot.png')
```

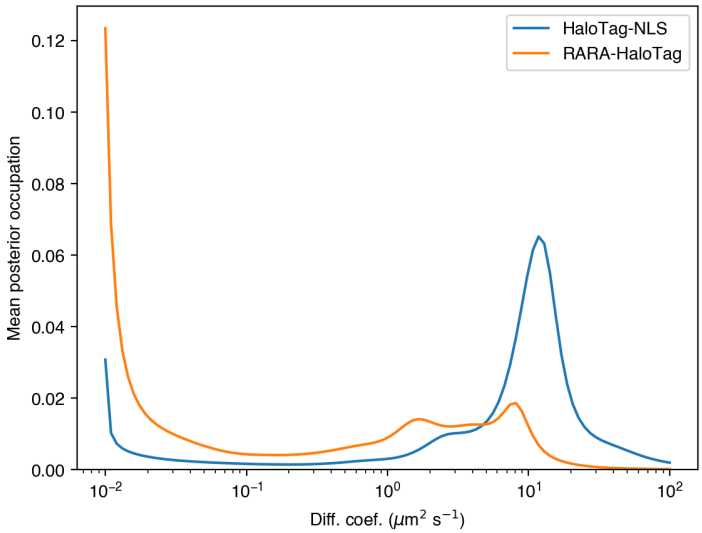



Additionally, rather than performing state array inference on each file individually, we can aggregate trajectories across all files matching a particular condition:

```
>>> posterior_occs, condition_names = SAD.infer_posterior_by_condition('condition')
>>> print(posterior_occs.shape)
(2, 100)
>>> print(condition_names)
['HaloTag-NLS', 'RARA-HaloTag']
```

The results are unnormalized (they reflect the total number of jumps in each condition). We can normalize and plot the results by doing:

```
>>> from saspt import normalize_2d
>>> posterior_occs = normalize_2d(posterior_occs, axis=1)
>>> diff_coefs = SAD.likelihood.diff_coefs
>>> for c in range(posterior_occs.shape[0]):
...     plt.plot(diff_coefs, posterior_occs[c,:], label=condition_names[c])
>>> plt.xscale('log')
>>> plt.xlabel('Diff. coef. ($\mu$m$^{2}$ s$^{-1}$)')
>>> plt.ylabel('Mean posterior occupation')
>>> plt.ylim((0, plt.ylim()[1]))
>>> plt.legend()
>>> plt.show()
```




The more trajectories we aggregate, the better our state occupation estimates become. `saspt` performs best when using large datasets with tens of thousands of trajectories per condition.

## 2.3 Background

We developed `saspt` to analyze a kind of high-speed microscopy called *live cell protein tracking*. These experiments rely on fast, sensitive cameras to recover the paths of fluorescently labeled protein molecules inside living cells.

The movie below is an example of a kind of protein tracking called SPT-PALM (Manley et al. 2013). The microscope in this experiment is focused on a cell, but nearly all of the cell is invisible. The only part that we can see are bright little dots, each of which is a dye molecule attached to a kind of protein called NPM1-HaloTag. Only a few of these dye molecules are active at any given time. This keeps their density low enough to track their paths through the cell.

Fig. 1: A short segment taken from an SPT-PALM movie in live U2OS osteosarcoma cells. Each bright dot is a single dye molecule covalently linked to an NPM1-HaloTag protein. The dye in this experiment is PA-JFX549, generously provided by the lab of Luke Lavis.

A quick glance reveals that not all molecules behave the same way. Some are nearly stationary; others wander rapidly around the cell. In `saspt`, we refer to these categories of behavior as *states*. In protein tracking, a major goal when applying tracking to a new protein target is to figure out

1. the number of distinct states the protein can occupy;
2. the characteristics of each state (*state parameters*);
3. the fraction of molecules in each state (*state occupations*).

Together, we refer to this information as a *model*. The goal of `saspt` is to recover a model given some observed trajectories. This specific kind of model, where we have a collection of observations from different states, is called a *mixture model*.

It's important to keep in mind that similar states can arise in distinct ways. For example, a slow-moving state might indicate that the protein is interacting with an immobile scaffold (like the cell membrane or DNA), or that it's simply too big to diffuse quickly in the crowded environment of the cell. Good SPT studies use biochemical perturbations - such as mutations or domain deletions - to tease apart the molecular origins of each state.

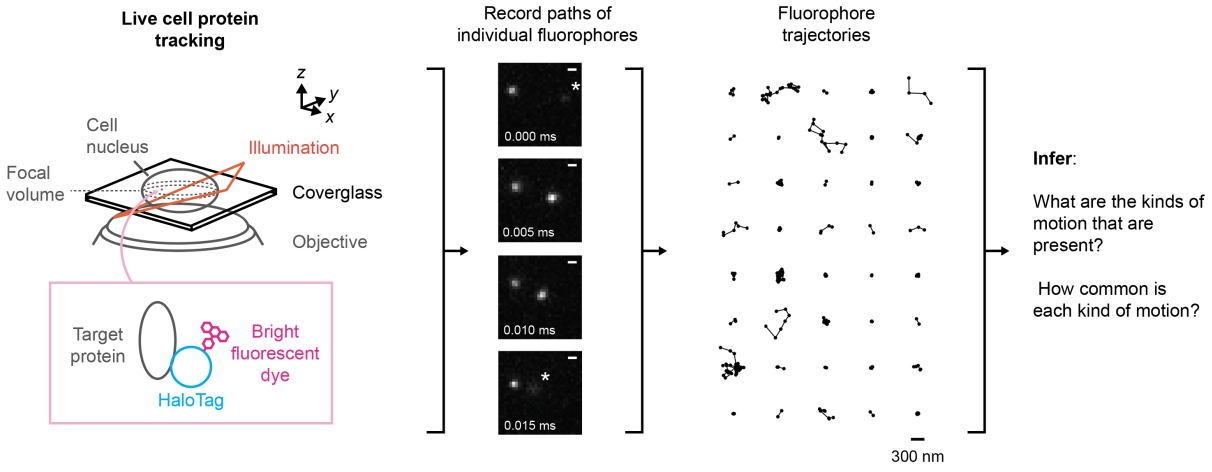



Fig. 2: Schematic of the workflow for a live cell protein tracking experiment. A protein target is conjugated to a bright fluorescent label (in this case, a fluorescent dye molecule via a HaloTag domain). This label is then used to track the protein around inside the cell (*center*), producing a large set of short trajectories (*right*). A common goal in postprocessing is to use the set of trajectories to learn about the dynamics, or modes of mobility, exhibited by the protein target.

### 2.3.1 Model fitting vs. model selection

Approaches to analyze protein tracking data fall into one of two categories:

1. *Model fitting*: we start with a known physical model. The goal is to fit the coefficients of that model given the observed data. Example: given a model with two Brownian states, recover the occupations and diffusion coefficients of each state.
2. *Model selection*: we start with some uncertainty in the physical model (often, but not always, because we don't know the number of states). The goal of inference is to recover both the model and its coefficients.

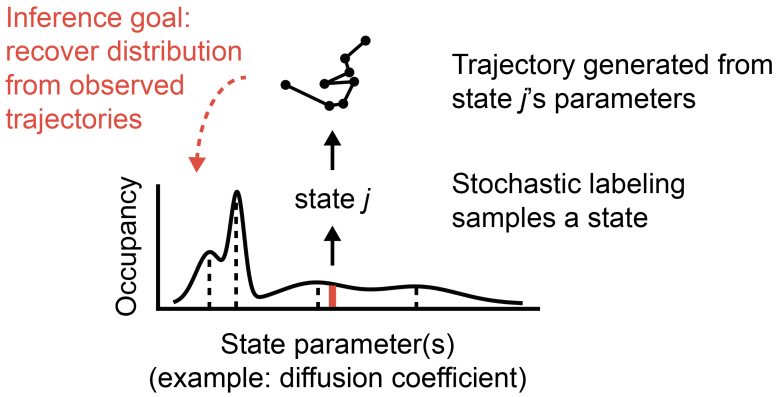
Model selection is harder than model fitting. Some approaches, such as maximum likelihood inference, tend to exploit every degree of freedom we afford them in the model. As a result, when we increase model complexity (by adding new states, for instance), we get better (higher likelihood) models. But this is mostly because they explain *noise* better, rather than intrinsic properties of the system. Such models generalize poorly to new data.

That's not very useful. Ideally we'd like to find the simplest model required to explain the observed data. Simple models often yield more intelligible insights into underlying behavior and generalize more cleanly to new data.

A key insight of early research into Bayesian methods was that such methods "pruned away" superfluous complexity in a model, providing a natural mechanism for model selection. For instance, when provided with a mixture model with a large number of states, Bayesian inference tends to drive the most state occupations to zero. In the context of machine learning, this property is sometimes referred to as *automatic relevance determination* (ARD). A more familiar analogy may be Occam's razor: when two possible models can explain the data equally well, we favor the simpler one.

### 2.3.2 State arrays

The *state array* is the model that underlies the `saspt` package. It takes trajectories from a protein tracking experiment and identifies a generative model for those trajectories, including the number of distinct states, their characteristics, and their occupations.




To do so, it relies on a variational Bayesian inference routine that "prunes away" superfluous states on a large grid of possibilities, leading to minimal models that describe observed data. It is particularly designed to deal with two limitations of protein tracking datasets:

1. Trajectories are often very short, due to bleaching and defocalization. (See: *Q. What is defocalization?*)
2. Apparent motion in tracking experiments can come from *localization error*, imprecision in the subpixel estimate of each emitter's position.

## 2.4 The state array model

This section describes the underlying state array model used by `saspt`. State arrays are just Bayesian mixture models with large number of mixture components ("states") that are situated on a fixed grid of parameters. They rely on a variational Bayesian inference routine that prunes away superfluous states, selecting minimal models to describe observed SPT datasets. What makes them work is similar to what makes nonparametric Bayesian methods work (namely, automatic relevance determination). But their structure leads to a much more efficient and scalable inference routine.

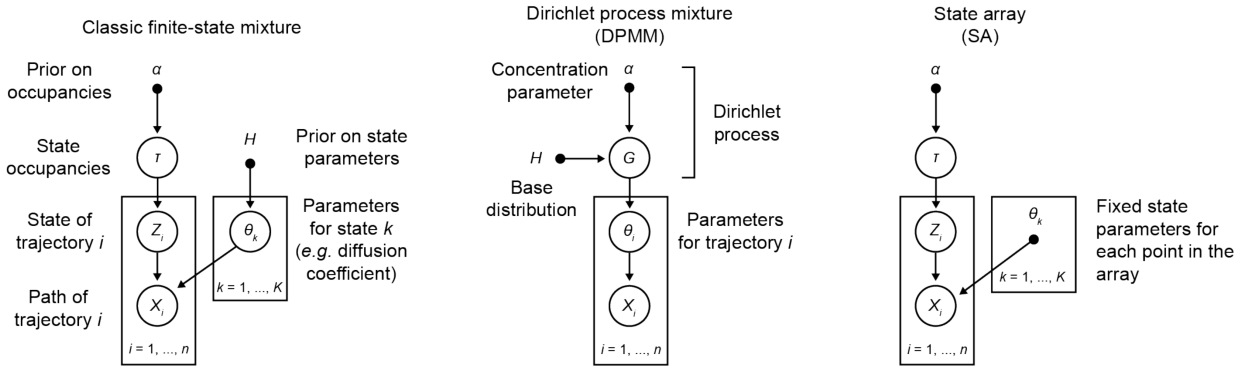



Fig. 3: Graphical model comparison of finite state mixtures, Dirichlet process mixtures, and state arrays. Open circles indicate unobserved variables, closed circles/dots indicate observed variables.

### 2.4.1 Likelihood functions

A physical model for the motion of particles in an SPT experiment is usually expressed in the form of a probability distribution $p(X|\boldsymbol{\theta})$, where $X$ is the observed path of the trajectory and $\boldsymbol{\theta}$ represents the *parameters* of the motion. Intuitively, this function tells us how likely we are to see a specific trajectory given some kind of physical model. We refer to the function $p(X|\boldsymbol{\theta})$ as a *likelihood function*.

As an example, take the case of regular Brownian motion (RBM), a model with a single parameter that characterizes each trajectory (the diffusion coefficient $D$). (For those familiar, this is equivalent to a scaled Wiener process.) Its likelihood function is

$$p(X|D) = \prod_{k=1}^{n} \frac{\exp\left(-\Delta r_k^2 / 4D\Delta t\right)}{(4\pi D\Delta t)^{\frac{d}{2}}}$$

where $\Delta r_k$ is the radial displacement of the $k^{\text{th}}$ jump in the trajectory, $\Delta t$ is the measurement interval, $d$ is the spatial dimension, and $n$ is the total number of jumps in the trajectory.

A common approach to recover the parameters of the motion is simply to find the parameters that maximize $p(X|\boldsymbol{\theta})$, holding $X$ constant at its observed value. In the case of RBM, this maximum likelihood estimate has a closed form - the mean squared displacement, or "MSD":

$$\hat{D}_{\text{mle}} = \frac{\sum_{k=1}^{n} \Delta r_k^2}{2dn\Delta t}$$

While this works well enough for one pure Brownian trajectory, this approach has several shortcomings when we try to generalize it:

1. Closed-form maximum likelihood solutions only exist for the simplest physical models, like RBM. Even introducing measurement error, a ubiquitous feature of SPT-PALM experiments, is sufficient to eliminate any closed-form solution.
2. Maximum likelihood does not provide any measure of confidence in the result. This becomes problematic for complex models with multiple parameters, where a large range of parameter vectors may give near-equivalent results. This means that even when our maximum likelihood estimators work perfectly, they are highly instable from one experiment to the next.

An alternative to maximum likelihood inference is to treat both $\mathbf{X}$ and $\boldsymbol{\theta}$ as random variables and evaluate the conditional probability $p(\boldsymbol{\theta}|\mathbf{X})$. For instance, we can estimate $\boldsymbol{\theta}$ by taking the mean of $p(\boldsymbol{\theta}|\mathbf{X})$. This is the Bayesian approach (and the one we use in `saspt`).

### 2.4.2 Mixture models

Suppose we observe $N$ trajectories in an SPT experiment, which we represent as a vector $\mathbf{X} = (X_1, ..., X_N)$. If all of the trajectories can be described by the same physical model, then the probability of seeing a set of trajectories $\mathbf{X}$ is just the product of the distributions over each $X_i$:

$$p(\mathbf{X}|\boldsymbol{\theta}) = \prod_{i=1}^{N} p(X_i|\boldsymbol{\theta})$$

In reality, this only describes the simplest situations because it assumes that the *same physical model governs all of the trajectories*. Most of the time we cannot assume that all trajectories originate from particles in the same physical state. Indeed, heterogeneity in a particle's dynamical states is often one of the things we hope to learn from an SPT experiment.

To deal with this complexity, we construct *mixture models*, which are exactly what they sound like: mixtures of particles in different *states*. Each state is governed by a different physical model. Parameters of interest include the model parameters characterizing each state, as well as the fraction of particles in each state (the state's *occupation*).

We formalize mixture models in the following way. Suppose we have a mixture of $K$ states. Instead of a single vector of state parameters, we'll have one vector for each state: $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_K)$. And in addition to the state parameters, we'll specify a set of *occupations* $\boldsymbol{\tau} = (\tau_1, ..., \tau_K)$ that describe the fraction of particles in each state. (These are also called *mixing probabilities*.)

With this formalism, the probability of seeing a single trajectory in state $j$ is $\tau_j$. The probability of seeing two trajectories in that state is $\tau_j^2$. And the probability of seeing $n_1$ trajectories in the first state, $n_2$ in the second state, and so on is

$$\tau_1^{n_1} \tau_2^{n_2} \cdots \tau_K^{n_K}$$

Of course, usually we don't *know* which state a given trajectory comes from. The more states we have, the more uncertainty there is.

The way to handle this in a Bayesian framework is to incorporate the uncertainty explicitly into the model by introducing a new random variable $\mathbf{Z}$ that we'll refer to as the *assignment matrix*. $\mathbf{Z}$ is a $N \times K$ matrix composed solely of 0s and 1s such that

$$Z_{ij} = \begin{cases} 1 & \text{if trajectory } i \text{ comes from a particle in state } j \\ 0 & \text{otherwise} \end{cases}$$

Notice that each row of $\mathbf{Z}$ contains a single 1 and the rest of the elements are 0. As an example, imagine we have three states and two trajectories, with the first trajectory assigned to state 1 and the second assigned to state 3. Then the assignment matrix would be

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Given a particular state occupation vector $\boldsymbol{\tau}$, the probability of seeing a particular set of assignments $\mathbf{Z}$ is

$$p(\mathbf{Z}|\boldsymbol{\tau}) = \prod_{j=1}^{K} \prod_{i=1}^{N} \tau_j^{Z_{ij}}$$

Notice that the probability and expected value for any given $Z_{ij}$ are the same:

$$p(Z_{ij}|\boldsymbol{\tau}) = \mathbb{E}\left[Z_{ij}|\boldsymbol{\tau}\right] = \tau_j$$

To review, we have four parameters that describe the mixture model:

- The state occupations $\boldsymbol{\tau}$, which describe the fraction of particles in each state;
- The state parameters $\boldsymbol{\theta}$, which describe the type of motion produced by particles in each state;
- The assignment matrix $\mathbf{Z}$, which describes the underlying state for each observed trajectory;
- The observed trajectories $\mathbf{X}$

### Bayesian mixture models

Of these four parameters, we only observe the trajectories $\mathbf{X}$ in an SPT experiment. The Bayesian approach is to infer the conditional distribution

$$p(\mathbf{Z}, \boldsymbol{\tau}, \boldsymbol{\theta}|\mathbf{X})$$

Using Bayes' theorem, we can rewrite this as

$$p(\mathbf{Z}, \boldsymbol{\tau}, \boldsymbol{\theta}|\mathbf{X}) \propto p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\tau}, \boldsymbol{\theta}) p(\mathbf{Z}, \boldsymbol{\tau}, \boldsymbol{\theta})$$

In order to proceed with this approach, it is necessary to specify the form of the last term, the *prior distribution*. Actually, since $\mathbf{Z}$ only depends on $\boldsymbol{\tau}$ and not $\boldsymbol{\theta}$, we can factor the prior as

$$p(\mathbf{Z}, \boldsymbol{\tau}, \boldsymbol{\theta}) = p(\mathbf{Z}|\boldsymbol{\tau})p(\boldsymbol{\tau})p(\boldsymbol{\theta})$$

We already saw the form of $p(\mathbf{Z}|\boldsymbol{\tau})$ earlier. $p(\boldsymbol{\theta})$ is usually chosen so that it is conjugate to the likelihood function (and, as we will see, it is irrelevant for state arrays). For the prior $p(\boldsymbol{\tau})$, we choose a Dirichlet distribution with parameter $\boldsymbol{\alpha}_0 = (\alpha_0, ..., \alpha_0) \in \mathbb{R}^K$:

$$\boldsymbol{\tau} \sim \text{Dirichlet}\,(\boldsymbol{\alpha}_0) = p(\boldsymbol{\tau}) = \frac{1}{B(\boldsymbol{\alpha}_0)} \prod_{j=1}^{K} \tau_j^{\alpha_0 - 1}$$

Each draw from this distribution is a possible set of state occupations $\boldsymbol{\tau}$, with the *mean* of these draws being a uniform distribution $(\frac{1}{K}, ..., \frac{1}{K})$. The variability of these draws about their mean is governed by $\alpha_0$, with high values of $\alpha_0$ producing distributions that are closer to a uniform distribution. ($\alpha_0$ is known as the *concentration parameter*.)

### 2.4.3 Infinite mixture models and ARD

There are many approaches to estimate the posterior distribution $p(\mathbf{Z}, \boldsymbol{\tau}, \boldsymbol{\theta}|\mathbf{Z})$, both numerical (Markov chain Monte Carlo) and approximative (variational Bayes with a factorable candidate posterior).

However, a fundamental problem is the choice of $K$, the number of states. Nearly everything depends on it.

Nonparametric Bayesian methods developed in the 1970s through 1990s proceeded on the realization that, as $K \to \infty$, the number of states with nonzero occupation in the posterior distribution approached a finite number. In effect, the these models "pruned" away superfluous features, leaving only the minimal models required to explain observed data. (In the context of machine learning, this property of Bayesian inference is called *automatic relevance determination* (ARD).)

These models replaced the separate priors $p(\boldsymbol{\tau})$ and $p(\boldsymbol{\theta})$ with a single prior $H(\boldsymbol{\theta})$ defined on the space of all possible parameters $\boldsymbol{\Theta}$. The models are known as *Dirichlet process mixture models* (DPMMs) because the priors are a kind of probability distribution called Dirichlet processes (essentially the infinite-dimensional version of a regular Dirichlet distribution).

In practice, however, such models are unwieldy. As MCMC methods, they are extremely computationally costly. This is particularly true for high-dimensional parameter vectors $\boldsymbol{\theta}$, for which inference on any kind of practical timescale is basically impossible. So while they solve the problem of choosing $K$, they introduce the equally dire problem of impractical runtimes.

### 2.4.4 State arrays

State arrays are a finite-state approximation of DPMMs. Instead of an infinite set of states, we choose a high but finite $K$ with state parameters $\theta_j$ that are situated on a fixed "parameter grid". Then, we rely mostly on the automatic relevance determination property of variational Bayesian inference to prune away the superfluous states. This leaves minimal models to describe observed trajectories. Because the states are chosen with fixed parameters, they only require that we evaluate the likelihood function *once*, at the beginning of inference. This shaves off an enormous amount of computational time relative to DPMMs.

In this section, we describe state arrays, landing at the actual algorithm for posterior inference used in `saspt`.

We choose a large set of $K$ different states with *fixed* state parameters $\boldsymbol{\theta}_j$ that are situated on a grid. Because the state parameters are fixed, the values of the likelihood function are constant and can be represented as a $N \times K$ matrix, $\mathbf{R}$:

$$R_{ij} = f(X_i|Z_{ij} = 1, \boldsymbol{\theta}_j)$$

The total probability function for the mixture model is then

$$p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau}) = p(\mathbf{X}|\mathbf{Z})p(\mathbf{Z}|\boldsymbol{\tau})p(\boldsymbol{\tau})$$

where

$$p(\mathbf{X}|\mathbf{Z}) = \prod_{i=1}^{N}\prod_{j=1}^{K} R_{ij}^{Z_{ij}}$$
$$p(\mathbf{Z}|\boldsymbol{\tau}) = \prod_{i=1}^{N}\prod_{j=1}^{K} \tau_j^{Z_{ij}}$$
$$p(\boldsymbol{\tau}) = \text{Dirichlet}(\alpha_0, ..., \alpha_0)$$

Following a variational approach, we seek an approximation to the posterior $q(\mathbf{Z}, \boldsymbol{\tau}) \approx p(\mathbf{Z}, \boldsymbol{\tau}|\mathbf{X})$ that maximizes the variational lower bound

$$L[q] = \sum_{\mathbf{Z}} \int_{\boldsymbol{\tau}} q(\mathbf{Z}, \boldsymbol{\tau}) \log\left[\frac{p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau})}{q(\mathbf{Z}, \boldsymbol{\tau})}\right] d\boldsymbol{\tau}$$

Under the assumption that $q$ factors as $q(\mathbf{Z}, \boldsymbol{\tau}) = q(\mathbf{Z})q(\boldsymbol{\tau})$, this criterion can be achieved via an expectation-maximization routine: alternately evaluating the two equations

$$\log q(\mathbf{Z}) = \mathbb{E}_{\boldsymbol{\tau}\sim q(\boldsymbol{\tau})}\left[\log p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau})\right] + \text{constant}$$
$$\log q(\boldsymbol{\tau}) = \mathbb{E}_{\mathbf{Z}\sim q(\mathbf{Z})}\left[\log p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau})\right] + \text{constant}$$

The constants are chosen so that the respective factors $q(\mathbf{Z})$ or $q(\boldsymbol{\tau})$ are normalized. These expectations are just shorthand for

$$\mathbb{E}_{\boldsymbol{\tau}\sim q(\boldsymbol{\tau})}\left[\log p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau})\right] = \int \log p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau}) q(\boldsymbol{\tau})\, d\boldsymbol{\tau}$$
$$\mathbb{E}_{\mathbf{Z}\sim q(\mathbf{Z})}\left[\log p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau})\right] = \sum_{\mathbf{Z}} \log p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\tau}) q(\mathbf{Z})$$

Evaluating the first of these factors (and ignoring terms that don't directly depend on $\boldsymbol{\tau}$), we have

$$\log q(\boldsymbol{\tau}) = \sum_{j=1}^{K}\left(\alpha_0 - 1 + \sum_{i=1}^{N}\mathbb{E}\left[Z_{ij}\right]\right)\log \tau_j + \text{constant}$$

From this, we can see that $q(\boldsymbol{\tau})$ is a Dirichlet distribution:

$$q(\boldsymbol{\tau}) = \text{Dirichlet}\left(\alpha_0 + \sum_{i=1}^{N}\mathbb{E}\left[Z_{i,0}\right], ..., \alpha_0 + \sum_{i=1}^{N}\mathbb{E}\left[Z_{i,K}\right]\right)$$

The distribution "counts" in terms of trajectories: each trajectory contributes one count (in the form of $Z_i$) to the posterior. This is not ideal: because SPT-PALM microscopes normally have a short focal depth due to their high numerical aperture, fast-moving particles contribute many short trajectories to the posterior while slow-moving particles contribute a few long trajectories. As a result, if we count by trajectories, we introduce strong *state biases* into the posterior. (This is exactly the reason why the popular MSD histogram method, which also "counts by trajectories", affords such inaccurate measurements of state occupations in realistic simulations of SPT-PALM experiments.)

A better way is to count the contributions to each state by *jumps* rather than trajectories. Because fast-moving and slow-moving states with equal occupation contribute the same number of *detections* within the focal volume, they contribute close to the same number of jumps (modulo the increased fraction of jumps from the fast-moving particle that "land" outside the focal volume).

Modifying this factor to count by jumps rather than trajectories, we have

$$q(\boldsymbol{\tau}) = \text{Dirichlet}\left(\alpha_0 + \alpha_1, ..., \alpha_0 + \alpha_K\right)$$
$$\alpha_j = \sum_{i=1}^{N} n_i \mathbb{E}\left[Z_{ij}\right]$$

where $n_i$ is the number of jumps observed for trajectory $i$.

Next, we evaluate $q(\mathbf{Z})$:

$$\log q(\mathbf{Z}) = \sum_{j=1}^{K}\sum_{i=1}^{N} \left(\log R_{ij} + \psi(\alpha_0 + \alpha_j)\right) Z_{ij} + \text{const}$$

where we have used the result that if $\boldsymbol{\tau} \sim \text{Dirichlet}\left(\boldsymbol{a}\right)$, then $\mathbb{E}\left[\tau_j\right] = \psi(a_j) - \psi(a_1 + ... + a_K)$, where $\psi$ is the digamma function.

Normalizing over each trajectory $i$, we have

$$q(\mathbf{Z}) = \prod_{i=1}^{N}\prod_{j=1}^{K} r_{ij}^{Z_{ij}}$$
$$r_{ij} = \frac{R_{ij} e^{\psi(\tau_j)}}{\sum\limits_{k=1}^{K} R_{ik} e^{\psi(\tau_k)}}$$

Under this distribution, we have

$$\mathbb{E}_{\mathbf{Z}\sim q(\mathbf{Z})}\left[Z_{ij}\right] = r_{ij}$$

To summarize, the joint posterior over $\mathbf{Z}$ and $\boldsymbol{\tau}$ is

$$q(\mathbf{Z}) = \prod_{i=1}^{N}\prod_{j=1}^{K} r_{ij}^{Z_{ij}}$$
$$q(\boldsymbol{\tau}) = \text{Dirichlet}\left(\alpha_0 + \alpha_1, ..., \alpha_0 + \alpha_K\right)$$
$$r_{ij} = \frac{R_{ij} e^{\psi(\tau_j)}}{\sum\limits_{k=1}^{K} R_{ik} e^{\psi(\tau_k)}}$$
$$\alpha_j = \sum_{i=1}^{N} n_i r_{ij}$$

The two factors of $q$ are completely specified by the factors $\mathbf{r}$ and $\boldsymbol{\tau}$. The algorithm for refining these factors is:

- Evaluate the likelihood function for each trajectory-state pairing: $R_{ij} = f(X_i|\boldsymbol{\theta}_j)$.
- Initialize $\boldsymbol{\alpha}$ and $\mathbf{r}$ such that

$$\alpha_j^{(0)} = \alpha_0$$
$$r_{ij}^{(0)} = \frac{R_{ij}}{\sum\limits_{k=1}^{K} R_{ik}}$$

- At each iteration $t = 1, 2, ...$:

1. For each $j = 1, ..., K$, set $\alpha_j = \alpha_0 + \sum_{i=1}^{N} n_i r_{ij}^{(t-1)}$.
2. For each $i = 1, ..., N$ and $j = 1, ..., K$, set $r_{ij}^{(*)} = R_{ij} e^{\psi(\alpha_j^{(t)})}$.
3. Normalize $\mathbf{r}$ over all states for each trajectory $r_{ij}^{(t)} = \frac{r_{ij}^*}{\sum_{k=1}^{K} r_{ik}^*}$

This is the state array algorithm implemented in `saspt`. After inference, we can summarize the posterior using its mean:

$$\mathbb{E}_{q(\boldsymbol{\tau})}[\tau_j] = \frac{\alpha_j + \alpha_0}{\sum_{k=1}^{K} \alpha_k + \alpha_0}$$
$$\mathbb{E}_{q(\mathbf{Z})}[Z_{ij}] = r_{ij}$$

These are the values reported to the user as `StateArray.posterior_occs` and `StateArray.posterior_assignment_probabilities`.

### 2.4.5 Accounting for defocalization

## 2.5 API

`saspt` is an object-oriented Python 3 library. The classes perform discrete roles:

- `TrajectoryGroup` represents a set of trajectories to be analyzed by state arrays
- `Likelihood` represents a likelihood function, or physical model for the type of motion. Some examples of likelihood functions include Brownian motion with localization error (RBME), various approximations of RBME, and fractional Brownian motion (FBM). The `Likelihood` class also defines the grid of *state parameters* on which the state array is constructed.
- `StateArrayParameters` is a struct with the settings for state array inference, including the pixel size, frame rate, and other imaging parameters.
- `StateArray` implements the actual state array inference routines.
- `StateArrayDataset` parallelizes state arrays across multiple target files, and provides some tools and visualizations for comparing between experimental conditions.

### 2.5.1 Note on input format

saspt expects input trajectories as a large table of spatiotemporal coordinates. Each coordinate represents the detection of a single fluorescent emitter, and is associated with a *trajectory index* that has been assigned by a tracking algorithm. An example is shown in the figure below.

The detection table (usually as a CSV) is the format expected by `saspt`. This format was chosen for simplicity.

`saspt` comes with an example of the kind of input it expects:

```
>>> from saspt import sample_detections
>>> detections = sample_detections()
>>> print(detections)
             y           x  frame  trajectory
0    575.730202   84.828673      0       13319
```

(continues on next page)

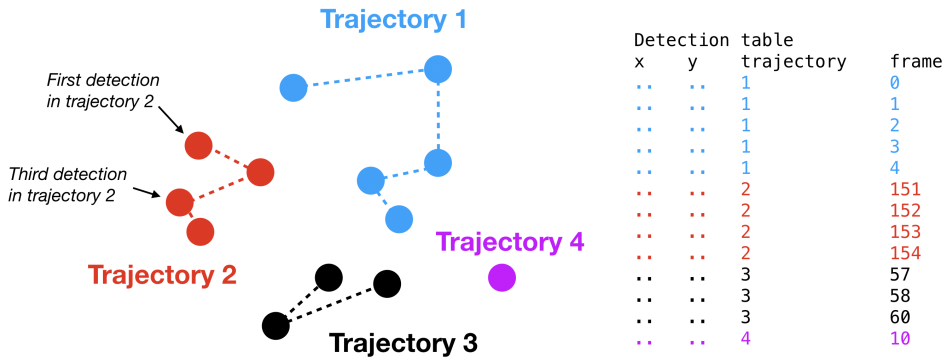



Fig. 4: Example of a table of detections. Each dot represents a detection and the dotted lines represent connections ("jumps") between detections in the same trajectory. Notice that the trajectories may contain "gaps", or missing frame indices, as in the case of trajectory 3.

(continued from previous page)

```
1    538.416604  485.924667      0        1562
2    107.647631   61.892363      0         363
..          ...         ...    ...         ...
493  366.475688   70.559735    297       14458
494  363.350134   67.585339    298       14458
495  360.006572   70.511980    299       14458

[496 rows x 4 columns]
```

The XY coordinates in **pixels**.

These can then be used to construct the objects that `saspt` expects (see the class hierarchy at *API*):

```
>>> from saspt import TrajectoryGroup, StateArrayParameters, StateArray, make_likelihood,
↪ RBME
>>> settings = dict(likelihood_type = RBME, pixel_size_um = 0.16, frame_interval = 0.
↪00748)
>>> params = StateArrayParameters(**settings)
>>> trajectories = TrajectoryGroup(detections, **settings)
>>> likelihood = make_likelihood(**settings)
>>> SA = StateArray(trajectories, likelihood, params)
>>> print(params)
StateArrayParameters:
  pixel_size_um:    0.16
  frame_interval:   0.00748
  focal_depth:  inf
  splitsize:    10
  sample_size:  10000
  start_frame:  0
  max_iter: 200
  conc_param:   1.0

>>> print(trajectories)
TrajectoryGroup:
```

(continues on next page)

(continued from previous page)

```
n_detections:   434
n_jumps:    370
n_tracks:   64

>>> print(SA)
StateArray:
  likelihood_type   : rbme
  n_tracks          : 64
  n_jumps           : 370
  parameter_names   : ('diff_coef', 'loc_error')
  shape             : (101, 36)
```

Although this approach is explicit, it is usually easier to use one of the alternative constructors that produce a `StateArray` directly from a set of detections:

```
>>> SA = StateArray.from_detections(detections, **settings)
```

This executes exactly the same steps implicitly:

```
>>> print(SA)
StateArray:
  likelihood_type   : rbme
  n_tracks          : 64
  n_jumps           : 370
  parameter_names   : ('diff_coef', 'loc_error')
  shape             : (101, 36)
```

### 2.5.2 TrajectoryGroup

**class** saspt.**TrajectoryGroup**(*detections: pandas.DataFrame*, *pixel_size_um: float*, *frame_interval: float*)

A set of trajectories to be analyzed with state arrays. TrajectoryGroup takes a raw set of trajectories produced by a tracking algorithm, such as quot, and performs some preprocessing steps to facilitate downstream calculations with state arrays. These include:

- remove all singlets (trajectories with a single detection), unassigned detections, and detections before an arbitrary start frame
- split long trajectories into smaller pieces, to minimize the effects of state transitions and tracking errors
- reindex trajectories so that the trajectory indices are contiguous between 0 and *n_tracks-1*

The TrajectoryGroup object also provides some methods to get some general information about the set of trajectories via the `raw_track_statistics` and `processed_track_statistics` attributes. An example:

```
>>> import numpy as np, pandas as pd
>>> from saspt import TrajectoryGroup

# Simple set of three detections belonging to two trajectories
>>> detections = pd.DataFrame({
...     'frame': [0, 0, 1],
...     'trajectory': [0, 1, 0],
...     'y': [1.1, 2.2, 3.3],
...     'x': [3.3, 2.2., 1.1]
```

(continues on next page)

(continued from previous page)

```
... })

# Imaging parameters
>>> kwargs = dict(pixel_size_um = 0.16, frame_interval = 0.00748)

# Make a TrajectoryGroup with these detections
>>> with TrajectoryGroup(detections, **kwargs) as TG:
...     print(TG)
TrajectoryGroup:
n_detections:   2
n_jumps:    1
n_tracks:   1

# Show some information about the raw trajectories
...     print(TG.raw_track_statistics)
{'n_tracks': 2, 'n_jumps': 1, 'n_detections': 3, 'mean_track_length': 1.5,
'max_track_length': 2, 'fraction_singlets': 0.5, 'fraction_unassigned': 0.0,
'mean_jumps_per_track': 0.5, 'mean_detections_per_frame': 1.5,
'max_detections_per_frame': 2, 'fraction_of_frames_with_detections': 1.0}

# Show some information about the trajectories after preprocessing
...    print(TG.processed_track_statistics)
{'n_tracks': 1, 'n_jumps': 1, 'n_detections': 2, 'mean_track_length': 2.0,
'max_track_length': 2, 'fraction_singlets': 0.0, 'fraction_unassigned': 0.0,
'mean_jumps_per_track': 1.0, 'mean_detections_per_frame': 1.0,
'max_detections_per_frame': 1, 'fraction_of_frames_with_detections': 1.0}
```

In this example, notice that trajectory 1 only has a single detection. As a result, it is filtered out by the preprocessing step, since it contributes no dynamic information to the result.

**__init__**(*self*, *detections: pandas.DataFrame*, *pixel_size_um: float*, *frame_interval: float*, *splitsize: int = DEFAULT_SPLITSIZE*, *start_frame: int = DEFAULT_START_FRAME*)

Default constructor for the TrajectoryGroup object.

**Parameters**

- **detections** (*pandas.DataFrame*) – raw detections/trajectories produced by a tracking algorithm. Each row of the DataFrame represents a single detections. Must contain at minimum the following four columns:
  1. `y`: the y-coordinate of the detection in pixels
  2. `x`: the x-coordinate of the detection in pixels
  3. `frame`: the frame index of the detection
  4. `trajectory`: the index of the trajectory to which the detection has been assigned by the tracking algorithm
- **pixel_size_um** (*float*) – size of camera pixels after magnification in microns
- **frame_interval** (*float*) – time between frames in seconds
- **splitsize** (*int*) – maximum trajectory length (in # of jumps) to consider. Trajectories longer than *splitsize* are broken into smaller pieces.
- **start_frame** (*int*) – disregard detections before this frame. Useful for restrict analysis to the later, lower-density parts of an SPT movie.

**Returns**
**new_instance** (*TrajectoryGroup*)

**property n_detections: int**

Total number of detections after preprocessing.

**property n_jumps: int**

Total number of *jumps* (particle-particle links) after preprocessing.

**property n_tracks: int**

Total number of trajectories (sequences of detections connected by links) in this dataset.

**property jumps: pandas.DataFrame**

The set of all *jumps* (particle-particle links) in these trajectories. Each row corresponds to a single jump. Contains the following columns:

- **frame** (`saspt.constants.FRAME`): frame index of the first detection participating in this jump
- **dframes** (`saspt.constants.DFRAMES`): difference in frames between the second and first detection in this jump. For instance, if `dframes == 1`, then the jump is a link between detections in subsequent frames.
- **trajectory** (`saspt.constants.TRACK`): index of the trajectory to which the detections in this jump have been assigned by the tracking algorithm.
- **dy** (`saspt.constants.DY`): jump distance in the *y*-dimension (in microns)
- **dx** (`saspt.constants.DX`): jump distance in the *x*-dimension (in microns)
- **dr2** (`saspt.constants.DR2`): mean squared jump distance in the *xy* plane. Equivalent to `(dy**2 + dx**2) / dframes`.
- **jumps_per_track** (`saspt.constants.JUMPS_PER_TRACK`): total number of jumps in the trajectory to which this jump belongs

Example:

```
# Simple set of detections belonging to 3 trajectories
>>> detections = pd.DataFrame({
...     'trajectory': [0, 0, 0, 1, 1, 1, 2, 2],
...     'frame':      [0, 1, 2, 0, 1, 2, 0, 1],
...     'y': [0., 1., 2., 0., 0., 0., 0., 3.],
...     'x': [0., 0., 0., 0., 2., 4., 0., 0.],
... })

# Imaging parameters
>>> kwargs = dict(pixel_size_um = 1.0, frame_interval = 0.00748)

# Make a TrajectoryGroup
>>> with TrajectoryGroup(detections, **kwargs) as TG:
...     print(TG.jumps)

   frame  dframes  trajectory   dy   dx  dr2  jumps_per_track
0      0        1           0  1.0  0.0  1.0                2
1      1        1           0  1.0  0.0  1.0                2
2      0        1           1  0.0  2.0  4.0                2
3      1        1           1  0.0  2.0  4.0                2
4      0        1           2  3.0  0.0  9.0                1
```

`property jumps_per_track: numpy.ndarray, shape (n_tracks,)`

Number of jumps per trajectory

`property raw_track_statistics: dict`

Summary statistics on the *raw* set of trajectories (*i.e.* the set of trajectories passed when constructing this TrajectoryGroup object).

These include:

- **n_tracks**: total number of trajectories
- **n_jumps**: total number of jumps
- **n_detections**: total number of detections
- **mean_track_length**: mean trajectory length in frames
- **max_track_length**: length of the longest trajectory in frames
- **fraction_singlets**: fraction of trajectories that have length 1 (in other words, they're just single detections)
- **fraction_unassigned**: fraction of detections that are not assigned to any trajectory (have trajectory index <0). May not be relevant for all tracking algorithms.
- **mean_jumps_per_track**: mean number of jumps per trajectory
- **mean_detections_per_frame**: mean number of detections per frame
- **max_detections_per_frame**: maximum number of detections per frame
- **fraction_of_frames_with_detections**: fraction of all frames between the minimum and maximum frame indices that had detections. If 1.0, then all frames contained at least one detected spot.

`property processed_track_statistics: dict`

Summary statistics on the *processed* set of trajectories (*i.e.* the set of trajectories after calling `TrajectoryGroup.preprocess` on the raw set of trajectories).

These are exactly the same as the metrics in `raw_track_statistics`.

`class property statistic_names: List[str]`

Names of each track summary statistic in `TrajectoryGroup.raw_track_statistics` and `TrajectoryGroup.processed_track_statistics`.

`get_track_vectors`(*self*, *n: int*) → Tuple[numpy.ndarray]

Return the jumps of every trajectory with *n* jumps as a `numpy.ndarray`.

**Parameters**
**n** (*int*) – number of jumps per trajectory

**Returns**
**V** (*numpy.ndarray*), **track_indices** (*numpy.ndarray*)

**V** is a 3D `numpy.ndarray` with shape `(n_tracks, 2, n)`. `V[:,0,:]` are the jumps along the *y*-axis, while `V[:,1,:]` are the jumps along the *x*-axis.

**track_indices** is a 1D `numpy.ndarray` with shape `(n_tracks,)` and gives the index of the trajectory corresponding to the first axis of `V`.

Using the example from above:

```
# Simple set of detections belonging to 3 trajectories
>>> detections = pd.DataFrame({
...     'trajectory': [0, 0, 0, 1, 1, 1, 2, 2],
...     'frame':      [0, 1, 2, 0, 1, 2, 0, 1],
...     'y': [0., 1., 2., 0., 0., 0., 0., 3.],
...     'x': [0., 0., 0., 0., 2., 4., 0., 0.],
})

# Make a TrajectoryGroup
>>> TG = TrajectoryGroup(detections, pixel_size_um=1.0,
...      frame_interval=0.00748)
>>> print(TG)
TrajectoryGroup:
n_detections:   8
n_jumps:    5
n_tracks:   3

# Get the jump vectors for all trajectories with 2 jumps
>>> V, track_indices = TG.get_jump_vectors(2)
>>> print(V)
[[[1. 1.]
  [0. 0.]]

 [[0. 0.]
  [2. 2.]]]

>>> print(track_indices)
[0 1]

# Get the jump vectors for all trajectories with 1 jump
>>> V, track_indices = TG.get_jump_vectors(1)
>>> print(V)
[[[3.]
  [0.]]]

>>> print(track_indices)
[2]
```

**subsample**(*self*, *size: int*) → *TrajectoryGroup*

Randomly subsample some number of trajectories from this TrajectoryGroup object to produce a new, smaller TrajectoryGroup object.

**Parameters**
**size** (*int*) – number of trajectories to subsample

**Returns**
**new_instance** (*TrajectoryGroup*)

Example:

```
# A TrajectoryGroup with 3 trajectories
>>> print(TG)
TrajectoryGroup:
n_detections:   8
```

(continues on next page)

(continued from previous page)

```
n_jumps:    5
n_tracks:   3

# Randomly subsample 2 of these trajectories
>>> TG2 = TG.subsample(2)
>>> print(TG2)
TrajectoryGroup:
n_detections:   5
n_jumps:    3
n_tracks:   2
```

**classmethod from_params**(*cls*, *detections: pandas.DataFrame*, *params:* StateArrayParameters) → *TrajectoryGroup*

Alternative constructor that uses a *StateArrayParameters* object rather than a set of keyword arguments.

**Parameters**

- **detections** (*pandas.DataFrame*) – the set of detections to use
- **params** (StateArrayParameters) – imaging and state array settings

**Returns**

**new_instance** (*TrajectoryGroup*)

Example usage:

```
>>> from saspt import StateArrayParameters, TrajectoryGroup
>>> params = StateArrayParameters(
...     pixel_size_um = 0.16,
...     frame_interval = 0.00748
... )
>>> TG = TrajectoryGroup.from_params(some_detections, params)
```

**classmethod from_files**(*cls*, *filelist: List[str]*, ***kwargs*) → *TrajectoryGroup*

Alternative constructor. Create a *TrajectoryGroup* by loading and concatenating detections directly from one or more files. The files must be readable by *saspt.io.load_detections*.

**Parameters**

- **filelist** (*List[str]*) – a list of paths to files containing detections
- **kwargs** – options to *TrajectoryGroup.__init__*

**Returns**

**new_instance** (*TrajectoryGroup*)

**classmethod preprocess**(*cls*, *detections: pandas.DataFrame*, *splitsize: int = DEFAULT_SPLITSIZE*, *start_frame: int = DEFAULT_START_FRAME*) → pandas.DataFrame

Preprocess some raw trajectories for state arrays. This involves:

- remove all singlets (trajectories of length 1), unassigned detections, and detections before *start_frame*
- break large trajectories into smaller pieces that have at most *splitsize* jumps
- reindex trajectories so that the set of all trajectory indices is contiguous between 0 and *n_tracks-1*

For most applications *preprocess* should not be called directly, and instead you should instantiate a *TrajectoryGroup* using one of the constructors.

**Parameters**

- **detections** (*pandas.DataFrame*) – indexed by detection. Must be recognize by *saspt.io.is_detections*.
- **splitsize** (*int*) – maximum trajectory length in *jumps*
- **start_frame** (*int*) – disregard detections recorded before this frame. Useful to restrict attention to later frames with lower density.

**Returns**

**processed_detections** (*pandas.DataFrame*)

### 2.5.3 Likelihood

**class Likelihood**

Abstract base class for likelihood functions, defining properties that all likelihood functions must implement.

Each Likelihood subclass evaluates on a set of trajectories at each of a grid of parameter values.

Instances of Likelihood subclasses should be generated with the *saspt.make_likelihood* function. For example:

```
>>> from saspt import make_likelihood, LIKELIHOOD_TYPES

# Imaging parameters
>>> kwargs = dict(
...     pixel_size_um = 0.16,
...     frame_interval = 0.00748,
...     focal_depth = 0.7
... )

# Make a likelihood function for each of the available likelihood types,
# and show the names of the parameters on which the likelihood function
# evaluates
>>> for likelihood_type in LIKELIHOOD_TYPES:
...     L = make_likelihood(likelihood_type, **kwargs)
...     print(f"Likelihood function '{L.name}' has parameters: {L.parameter_names}")
Likelihood function 'rbme' has parameters: ('diff_coef', 'loc_error')
Likelihood function 'rbme_marginal' has parameters: ('diff_coef',)
Likelihood function 'gamma' has parameters: ('diff_coef',)
Likelihood function 'fbme' has parameters: ('diff_coef', 'hurst_parameter')
```

**abstract property name: str**

The name of the likelihood function.

**abstract property shape: Tuple[int]**

The shape of the parameter grid on which the likelihood function evaluates.

**abstract property parameter_names: Tuple[str]**

Names of each parameter in the parameter grid.

**abstract property parameter_values: Tuple[numpy.ndarray]**

Values of each parameter in the parameter grid, given as a tuple of 1D `numpy.ndarray`. The parameter grid is the Cartesian product of these arrays.

**abstract property parameter_units: Tuple[str]**

Physical units for each parameter in the parameter grid.

**abstract __call__**(*self*, *trajectories:* TrajectoryGroup) → Tuple[numpy.ndarray]

Evaluate the log likelihood function on each of a set of trajectories at each point on the parameter grid.

**Parameters**

**trajectories** (TrajectoryGroup) – the trajectories on which to evaluate

**Returns**

**evaluated_log_likelihood** (*numpy.ndarray*), **jumps_per_track** (*numpy.ndarray*)

- **evaluated_log_likelihood** is the value of the log likelihood function evaluated on each of the trajectories at each of the parameter values. It is a `numpy.ndarray` with shape `(*self.shape, trajectories.n_tracks)`.
- **jumps_per_track** is the number of jumps in each trajectory. It is a 1D `numpy.ndarray` with shape `(trajectories.n_tracks,)`.

Example usage:

```
>>> from saspt import make_likelihood, RBME, TrajectoryGroup
>>> kwargs = dict(pixel_size_um = 0.16, frame_interval = 0.00748)

# Make an RBMELikelihood function
>>> likelihood = make_likelihood(RBME, **kwargs)

# Show the shape of the parameter grid on which this likelihood
# function is defined
>>> print(likelihood.shape)
(101, 36)

# Show the names of the parameters corresponding to each axis
# on the parameter grid
>>> print(likelihood.parameter_names)
('diff_coef', 'loc_error')

# Load some trajectories (available in saspt/tests/fixtures)
>>> tracks = TrajectoryGroup.from_files(["tests/fixtures/small_tracks_0.csv"],␣
↪**kwargs)

# Evaluate the log likelihood function on these trajectories
>>> log_L, jumps_per_track = likelihood(tracks)

# The log likelihood contains one element for each trajectory
# and each point in the parameter grid
>>> print(log_L.shape)
(101, 36, 39)

>>> print(jumps_per_track.shape)
(39,)
```

**abstract exp**(*self*, *log_L: numpy.ndarray*) → numpy.ndarray

Take the exponent of a log likelihood function produced by *Likelihood.__call__()* in a numerically stable way.

This function also normalizes the likelihood function so that the values of the likelihood sum to 1 across all states in each trajectories.

We can take the exponent of the log likelihood function from the example in *__call__*, above:

```
from saspt import RBME

# Make an RBME likelihood function
>>> likelihood = make_likelihood(RBME, **kwargs)

# Get the normalized likelihood
>>> normed_L = likelihood.exp(log_L)

# Likelihood is normalized across all states for each trajectory
>>> print(normed_L.sum(axis=(0,1)))
array([1., 1., 1., ..., 1., 1., 1.])
```

**Returns**

**L** (*numpy.ndarray*), shape *log_L.shape*, the normalized likelihood function for each trajectory-state assignment

**abstract correct_for_defocalization**(*self*, *occs: numpy.ndarray*, *normalize: bool*) → numpy..ndarray

Correct a set of state occupations on this parameter grid for the effect of defocalization.

**Parameters**

- **occs** (*numpy.ndarray*) – state occupations, with shape *self.shape*
- **normalize** (*bool*) – normalize the occupations after applying the correction

**Returns**

**corrected_occs** (*numpy.ndarray*, shape *self.shape*), corrected state occupations

**abstract marginalize_on_diff_coef**(*self*, *occs: numpy.ndarray*) → numpy.ndarray

Given a set of state occupations, marginalize over all parameters except the diffusion coefficient.

May raise `NotImplementedError` if the diffusion coefficient is not a parameter supported by this likelihood function. (Although this is the case for all Likelihood subclasses implemented to date!)

**Parameters**

**occs** (*numpy.ndarray*) – state occupations, shape *self.shape*

**Returns**

**marginal_occs** (*numpy.ndarray*), marginal state occupations. This will have a lower dimensionality than the input.

For example, suppose we are using the RBME likelihood with a parameter grid of shape (10, 6). Since the parameters for the RBME likelihood are `diff_coef` and `loc_error`, this means that the parameter grid has 10 distinct diffusion coefficient values and 6 distinct localization error values. After applying `marginalize_on_diff_coef`, the output has shape (10,) since the localization error is marginalized out.

In code, this situation is:

```
>>> import numpy as np
>>> from saspt import make_likelihood, RBME

# Define an RBME likelihood function on a grid of 10 diffusion
# coefficients and 6 localization errors
>>> likelihood = make_likelihood(
...     RBME,
...     pixel_size_um = 0.16,
...     frame_interval = 0.00748,
```

(continues on next page)

(continued from previous page)

```
...     focal_depth = 0.7,
...     diff_coefs = np.logspace(0.0, 1.0, 10),
...     loc_errors = np.linspace(0.0, 0.05, 6)
... )

>>> print(likelihood.shape)
(10, 6)

# Some random state occupations
>>> occs = np.random.dirichlet(np.ones(60)).reshape((10, 6))
>>> print(occs.shape)
(10, 6)

# Marginalize on diffusion coefficient
>>> marginal_occs = likelihood.marginalize_on_diff_coef(occs)
>>> print(marginal_occs.shape)
(10,)

# Plot marginal occupations as a function of diffusion coefficient
# (example)
>>> import matplotlib.pyplot as plt
>>> plt.plot(likelihood.diff_coefs, marginal_occs)
>>> plt.xlabel(f"Diff. coef. ({likelihood.parameter_units[0]})")
>>> plt.ylabel("Marginal occupation")
>>> plt.xscale('log'); plt.show(); plt.close()
```

### RBMELikelihood

**class RBMELikelihood**(*self*, *pixel_size_um: float*, *frame_interval: float*, *focal_depth: float = numpy.inf*, *diff_coefs: numpy.ndarray = DEFAULT_DIFF_COEFS*, *loc_errors: numpy.ndarray = DEFAULT_LOC_ERRORS*, ***kwargs*)

Subclass of *Likelihood* for the RBME (regular Brownian motion with localization error) likelihood function. Probably the most useful likelihood function in *saSPT*.

Suppose we image an RBME with diffusion coefficient $D$, localization error $\sigma$, and frame interval $\Delta t$. If there are $n$ jumps in the trajectory, and if $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are its jumps along $y$ and $x$ axes respectively, then the likelihood function is

$$f(\mathbf{x}, \mathbf{y} | D, \sigma) = \frac{\exp\left(-\frac{1}{2}\left(\mathbf{x}^T \Gamma^{-1} \mathbf{x} + \mathbf{y}^T \Gamma^{-1} \mathbf{y}\right)\right)}{(2\pi)^n \det(\Gamma)}$$

where $\Gamma \in \mathbb{R}^{n \times n}$ is the covariance matrix defined by

$$\Gamma_{ij} = \begin{cases} 2(D\Delta t + \sigma^2) & \text{if } i = j \\ -\sigma^2 & \text{if } |i - j| = 1 \\ 0 & \text{otherwise} \end{cases}$$

The parameter grid for *RBMELikelihood* is a two-dimensional array with the first axis corresponding to the diffusion coefficient and the second corresponding to localization error. By default, we use a set of logarithmically spaced diffusion coefficients between 0.01 and 100 $\mu$m$^2$ sec$^{-1}$ and linearly spaced localization errors between 0 and 0.08 $\mu$m.

In most situations, localization error is a nuisance parameter. When using the RBME likelihood function with a state array run, we usually marginalize over the localization error afterward. As a result, the RBME likelihood is

much more stable from day-to-day and microscope-to-microscope than likelihood functions that do not explicitly model the error, such as the *GammaLikelihood*.

See *Likelihood* for a description of the class properties and methods.

**Parameters**

- **pixel_size_um** (*float*) – camera pixel size after magnification in microns
- **frame_interval** (*float*) – time between frames in seconds
- **focal_depth** (*float*) – objective focal depth in microns. Used to calculate the effect of defocalization on apparent state occupations. If *numpy.inf*, no defocalization corrections are applied.
- **diff_coefs** (*numpy.ndarray*) – the set of diffusion coefficients to use for this likelihood function's parameter grid
- **loc_errors** (*numpy.ndarray*) – the set of localization errors to use for this likelihood function's parameter grid
- **kwargs** – ignored

### RBMEMarginalLikelihood

**class RBMEMarginalLikelihood**(*self*, *pixel_size_um: float*, *frame_interval: float*, *focal_depth: float = numpy.inf*, *diff_coefs: numpy.ndarray = DEFAULT_DIFF_COEFS*, *loc_errors: numpy.ndarray = DEFAULT_LOC_ERRORS*, ***kwargs*)

The underlying model is identical to *RBMELikelihood*. However, after evaluating the likelihood function on a 2D parameter grid of diffusion coefficient and localization error, we marginalize over the localization error to produce a 1D grid over the diffusion coefficient. State arrays then evaluate the posterior distribution over this 1D grid, rather than the 2D grid in *RBMELikelihood*.

In short, the order of state inference and marginalization is switched:

**RBMELikelihood:**

1. Evaluate 2D likelihood function over diffusion coefficient and localization error
2. Infer 2D posterior distribution over diffusion coefficient and localization error
3. Marginalize over localization error to get 1D distribution over diffusion coefficient

**RBMEMarginalLikelihood:**

1. Evaluate 2D likelihood function over diffusion coefficient and localization error
2. Marginalize over localization error to get 1D likelihood over diffusion coefficient
3. Infer 1D posterior distribution over diffusion coefficient

*RBMEMarginalLikelihood* generally is inferior to *RBMELikelihood* and is provided as a point of comparison.

**Parameters**

- **pixel_size_um** (*float*) – camera pixel size after magnification in microns
- **frame_interval** (*float*) – time between frames in seconds
- **focal_depth** (*float*) – objective focal depth in microns. Used to calculate the effect of defocalization on apparent state occupations. If *numpy.inf*, no defocalization corrections are applied.

- **diff_coefs** (*numpy.ndarray*) – the set of diffusion coefficients to use for this likelihood function's parameter grid
- **loc_errors** (*numpy.ndarray*) – the set of localization errors to marginalize over when evaluating the likelihood function
- **kwargs** – ignored

### GammaLikelihood

**class GammaLikelihood**(*self*, *pixel_size_um: float*, *frame_interval: float*, *focal_depth: float=numpy.inf*, *diff_coefs: numpy.ndarray=DEFAULT_DIFF_COEFS*, *loc_error: float=0.035*, *mode: str="point"*, ***kwargs*)

Subclass of *Likelihood* for the gamma approximation to the RBM (regular Brownian motion) likelihood function.

The gamma likelihood is obtained from the RBME likelihood by making two approximations:

- the localization error is treated as a constant
- we neglect the off-diagonal terms of the covariance matrix $\Gamma$

In this case, the likelihood function simplifies to a gamma distribution. Suppose that $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are the $x$- and $y$-jumps of a trajectory with $n$ total jumps, and let $S$ be the sum of its squared jumps:

$$S = \sum_{i=1}^{n} \left(x_i^2 + y_i^2\right)$$

Then the likelihood function can be expressed

$$f(S|D, \sigma^2) = \frac{S^{n-1} e^{-S/4D\Delta t}}{\Gamma(n)(4(D\Delta t + \sigma^2))^n}$$

Notice that in the term $4(D\delta t + \sigma^2)$, the contributions of diffusion ($D\Delta t$) and localization error ($\sigma^2$) cannot be distinguished without introducing the assumption that localization is constant. This is only approximately true, since the number of photons collectd per particle, the axial distance from the focus, and the motion of the particle will all influence localization error, creating variation within a single SPT movie. In particular, the gamma likelihood performs tolerably well when $D\Delta t \gg \sigma^2$, but is highly inaccurate when $D\Delta t \sim \sigma^2$.

The *GammaLikelihood* parameter grid is a simple 1D array of diffusion coefficients. By default, these are logarithmically spaced between 0.01 and 100 $\mu$m$^2$ sec$^{-1}$.

See *Likelihood* for a description of the class properties and methods.

**Parameters**

- **pixel_size_um** (*float*) – camera pixel size after magnification in microns
- **frame_interval** (*float*) – time between frames in seconds
- **focal_depth** (*float*) – objective focal depth in microns. Used to calculate the effect of defocalization on apparent state occupations. If *numpy.inf*, no defocalization corrections are applied.
- **diff_coefs** (*numpy.ndarray*) – the set of diffusion coefficients to use for this likelihood function's parameter grid
- **loc_error** (*float*) – the 1D localization error (assumed constant), expressed as the root variance in microns
- **mode** (*str*) – deprecated; ignored
- **kwargs** – ignored

### FBMELikelihood

**class FBMELikelihood**(*self*, *pixel_size_um: float*, *frame_interval: float*, *focal_depth: float*, *diff_coefs: numpy.ndarray = DEFAULT_DIFF_COEFS*, *hurst_pars: numpy.ndarray = DEFAULT_HURST_PARS*, *loc_error: float = 0.035*, ***kwargs*)

Likelihood function for fractional Brownian motion with localization error (FBME). This is similar to RBME but allows for temporal correlations (either positive or negative) between jumps in the same trajectory, depending on the value of the Hurst parameter:

- if $H < \frac{1}{2}$, jumps in the same trajectory are anticorrelated (the trajectory tends to return to where it came from)
- if $H = \frac{1}{2}$, the motion is Brownian (no correlation between the jumps)
- if $H > \frac{1}{2}$, jumps in the same trajectory are positively correlated (the trajectory tends to keep moving in the same direction)

In particular, if $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are the $x$- and $y$ components of the jumps of an FBME with $n$ jumps, diffusion coefficient $D$, Hurst parameter $H$, and localization error $\sigma$, then the likelihood function is defined

$$f(\mathbf{x}, \mathbf{y} | D, H, \sigma^2) = \frac{\exp\left(-\frac{1}{2}\left(\mathbf{x}^T \Gamma^{-1} \mathbf{x} + \mathbf{y}^T \Gamma^{-1} \mathbf{y}\right)\right)}{(2\pi)^n \det(\Gamma)}$$

where $\Gamma$ is the covariance matrix:

$$\Gamma_{ij} = \begin{cases} D\Delta t\left(|i-j+1|^{2H} + |i-j-1|^{2H} - 2|i-j|^{2H}\right) + 2\sigma^2 & \text{if } i = j \\ D\Delta t\left(|i-j+1|^{2H} + |i-j-1|^{2H} - 2|i-j|^{2H}\right) - \sigma^2 & \text{if } |i-j| = 1 \\ D\Delta t\left(|i-j+1|^{2H} + |i-j-1|^{2H} - 2|i-j|^{2H}\right) & \text{otherwise} \end{cases}$$

The parameter grid for *FBMELikelihood* is a 2D grid over diffusion coefficient and Hurst parameter, with localization error treated as a constant.

**Parameters**

- **pixel_size_um** (*float*) – camera pixel size after magnification in microns
- **frame_interval** (*float*) – time between frames in seconds
- **focal_depth** (*float*) – objective focal depth in microns. Used to calculate the effect of defocalization on apparent state occupations. If *numpy.inf*, no defocalization corrections are applied.
- **diff_coefs** (*numpy.ndarray*) – the set of diffusion coefficients to use for this likelihood function's parameter grid
- **hurst_pars** (*numpy.ndarray*) – the set of Hurst parameters to use for this likelihood function's parameter grid
- **loc_error** (*float*) – the 1D localization error (assumed constant), expressed as a root variance in microns
- **kwargs** – ignored

### 2.5.4 StateArrayParameters

**class** saspt.**StateArrayParameters**(*pixel_size_um: float*, *frame_interval: float*)

**__init__**(*pixel_size_um: float*, *frame_interval: float*, *focal_depth: float = np.inf*, *splitsize: int = DEFAULT_SPLITSIZE*, *sample_size: int = DEFAULT_SAMPLE_SIZE*, *start_frame: int = DEFAULT_START_FRAME*, *max_iter: int = DEFAULT_MAX_ITER*, *conc_param: float = DEFAULT_CONC_PARAM*, *progress_bar: bool = False*, *num_workers: int = 1*)

**Parameters**

- **pixel_size_um** (*float*) – camera pixel size after magnification in microns
- **frame_interval** (*float*) – delay between frames in seconds
- **splitsize** (*int*) – maximum length of trajectories in frames. Trajectories longer than *splitsize* are split into smaller pieces.
- **sample_size** (*int*) – maximum number of trajectories to consider per state array. SPT experiments that exceed this number are subsampled.
- **start_frame** (*int*) – disregard detections before this frame. Useful to restrict analysis to later frames with lower detection density.
- **max_iter** (*int*) – maximum number of iterations of variational Bayesian inference to run when inferring the posterior distribution
- **conc_param** (*float*) – concentration parameter of the Dirichlet prior over state occupations. A *conc_param* of 1.0 is a naive prior; values less than 1.0 favor more states and values greater than 1.0 favor fewer states. Default value is 1.0.
- **progress_bar** (*bool*) – show progress and be a little verbose, where relevant
- **num_workers** (*int*) – number of parallel processes to use. Recommended not to set this higher than the number of CPUs.

**Returns**

new instance of StateArrayParameters

**property parameters: Tuple[str]**

Names of all parameters that directly impact the state array algorithm. Does not include parameters that determine implementation or display, such as *progress_bar* or *num_workers*

**property units: dict**

Physical units in which each parameter is defined

**__eq__**(*self*, *other:* StateArrayParameters) → bool

Check for equivalence of two StateArrayParameter objects

**__repr__**(*self*) → str

String representation of this StateArrayParameters object

### 2.5.5 StateArray

**class StateArray**(*self*, *trajectories: TrajectoryGroup*, *likelihood: :py:class:`Likelihood`*, *params: StateArrayParameters*)

Central class for running state array inference on one SPT experiment. Encapsulates routines to infer the occupation of each point on a parameter grid, given a set of trajectories.

Specifically, suppose that $\mathbf{X}$ is a set of $N$ trajectories (using whatever format is most convenient).

We select a grid of $K$ distinct *states* (represented, in this case, by the Likelihood object). Each state is associated with some *state parameters* that define its characteristics. As an example, the RBME likelihood uses two parameters for each state: a diffusion coefficient and a localization error. We use $\theta_j$ to indicate the tuple of all state parameters for state $j$.

Let $\mathbf{Z} \in \{0,1\}^{N\times K}$ be the trajectory-state assignment matrix, so that

$$Z_{ij} = \begin{cases} 1 & \text{if trajectory } i \text{ is assigned to state } j \\ 0 & \text{otherwise} \end{cases}$$

Further, let $\boldsymbol{\tau} \in \mathbb{R}^K$ be the vector of state occupations, so that $\sum_{j=1}^{K} \tau_j = 1$.

Notice that, given a particular state occupation vector $\boldsymbol{\tau}$, the probability to see the assignments $\mathbf{Z}$ is

$$p(\mathbf{Z}|\boldsymbol{\tau}) = \prod_{i=1}^{N}\prod_{j=1}^{K} \tau_j^{Z_{ij}}$$

Similarly, the probability to see trajectories $\mathbf{X}$ given the assignment matrix $\mathbf{Z}$ is

$$p(\mathbf{X}|\mathbf{Z}) = \prod_{i=1}^{N}\prod_{j=1}^{K} f(X_i|\theta_j)^{Z_{ij}}$$

where $f(X_i|\theta_j)$ is the likelihood function for state $j$ evaluated on trajectory $i$.

We seek the posterior distribution $p(\mathbf{Z}, \boldsymbol{\tau}|\mathbf{X})$. The StateArray class uses a variational Bayesian approach that approximates the posterior distribution as the product of two factors:

$$p(\mathbf{Z}, \boldsymbol{\tau}|\mathbf{X}) \approx q(\mathbf{Z})q(\boldsymbol{\tau})$$

The factor $q(\mathbf{Z})$ is given by the attribute *posterior_assignment_probabilities*, while the factor $q(\boldsymbol{\tau})$ is given by the attribute *posterior_occs*.

For the prior over the trajectory-state assignments, we take a uniform distribution over all states for each trajectory. For the prior over the state occupations, we take:

$$\boldsymbol{\tau} \sim \text{Dirichlet}\,(\alpha \cdot \mathbf{1})$$

Here, $\mathbf{1}$ is a $K$-vector of ones and $\alpha$ is the *concentration parameter*. Larger values of $\alpha$ require more data in order to depart from uniformity. The default value of $\alpha$ (`saspt.constants.DEFAULT_CONC_PARAM`) is 1.0. Reasonable values are between 0.5 and 1.0.

Additionally, the StateArray object implements an alternative ("naive") estimator for the state occupations. This is defined as

$$\tau_j \propto \eta_j^{-1} \sum_{i=1}^{N} n_i r_{ij}$$

$$r_{ij} = \frac{f(X_i|\theta_j)}{\sum_{k=1}^{K} f(X_i|\theta_k)}$$

where $n_i$ is the number of jumps in trajectory $i$ and $\eta_j$ is a correction for defocalization of state $j$. The naive estimator is considerably less precise than the posterior occupations, but has the virtue of speed and simplicity.

**Parameters**

- **trajectories** (TrajectoryGroup) – a set of trajectories to run this state array on
- **likelihood** (Likelihood) – the likelihood function to use
- **params** (StateArrayParameters) – parameters governing the state array algorithm, including the concentration parameter, maximum number of iterations, and so on

**likelihood:** *Likelihood*

The underlying likelihood function for this StateArray

**trajectories:** *TrajectoryGroup*

The underlying set of trajectories for this StateArray

**classmethod from_detections**(*cls*, *detections: pandas.DataFrame*, *likelihood_type: str*, ***kwargs*)

Alternative constructor; make a StateArray directly from a set of detections. This avoids the user needing to explicitly construct the *Likelihood* and *StateArrayParameters* objects.

**Parameters**

- **detections** (*pandas.DataFrame*) – input set of detections, with the columns **frame** (saspt.constants.FRAME), **trajectory** (saspt.constants.TRACK), **y** (saspt.constants.PY), and **x** (saspt.constants.PX)
- **likelihood_type** (*str*) – the type of likelihood function to use; an element of *saspt.constants.LIKELIHOOD_TYPES*
- **kwargs** – additional keyword arguments to the *StateArrayParameters* and *Likelihood* subclass. Must include *pixel_size_um* and *frame_interval*.

**Returns**

new instance of *StateArray*

**property n_tracks: int**

Number of trajectories in this SPT experiment after preprocessing. See *TrajectoryGroup*.

**property n_jumps: int**

Number of jumps (particle-particle links) in this SPT experiment after preprocessing. See *TrajectoryGroup*.

**property n_detections: int**

Number of detections in this SPT experiment after preprocessing. See *TrajectoryGroup*.

**property shape: Tuple[int]**

Shape of the parameter grid on which this state array is defined. Alias for *StateArray.likelihood.shape*.

**property likelihood_type: str**

Name of the likelihood function. Alias for *StateArray.likelihood.name*.

**property parameter_names: Tuple[str]**

Names of the parameters corresponding to each axis in the parameter grid. Alias for *StateArray.likelihood.parameter_names*

**property parameter_values: Tuple[numpy.ndarray]**

Values of the parameters corresponding to each axis in the parameter grid. Alias for *StateArray.likelihood.parameter_values*.

**property n_states: int**

Total number of states in the parameter grid; equivalent to the product of the dimensions of the parameter grid

**property jumps_per_track: numpy.ndarray**

1D *numpy.ndarray* of shape *(n_tracks,)*; number of jumps in each trajectory

**property naive_assignment_probabilities: numpy.ndarray**

*numpy.ndarray* of shape *(*self.shape, n_tracks)*; the "naive" probabilities for each trajectory-state assignment. These are just normalized likelihoods, and provide a useful counterpoint to the posterior trajectory-state assignments.

The naive probability to assign trajectory $i$ to state $j$ in a model with $K$ total states is

$$r_{ij} = \frac{f(X_i|\theta_j)}{\sum_{k=1}^{K} f(X_i|\theta_k)}$$

where $f(X_i|\theta_j)$ is the likelihood function evaluated on trajectory $X_i$ with state parameter(s) $\theta_j$.

Example:

```
>>> from saspt import sample_detections, StateArray, RBME

# Make a StateArray
>>> SA = StateArray.from_detections(
...     sample_detections(),
...     likelihood_type = RBME,
...     pixel_size_um = 0.16,
...     frame_interval = 0.00748
... )
>>> print(f"Shape of parameter grid: {SA.shape}")
Shape of parameter grid: (101, 36)

>>> print(f"Number of trajectories: {SA.n_tracks}")
Number of trajectories: 64

# Get the probabilities for each trajectory-state assignment
>>> naive_assign_probs = SA.naive_assignment_probabilities
>>> print(f"Shape of assignment probability matrix: {naive_assign_probs.shape}")
Shape of assignment probability matrix: (101, 36, 64)

# Example: probability to assign trajectory 10 to state (0, 24)
>>> p = naive_assign_probs[0, 24, 10]
>>> print(f"Naive probability to assign track 10 to state (0, 24): {p}")
Naive probability to assign track 10 to state (0, 24): 0.0018974905182505026

# Assignment probabilities are normalized over all states for each track
>>> print(naive_assign_probs.sum(axis=(0,1)))
[1. 1. 1. ... 1. 1. 1.]
```

**property posterior_assignment_probabilities: numpy.ndarray**

*numpy.ndarray* of shape *(*self.shape, n_tracks)*; the posterior probabilities for each trajectory-state assignment.

In math, if we have $N$ trajectories and $K$ states, then the posterior distribution over trajectory-state assignments is

$$p(\mathbf{Z}|\mathbf{r}) = \prod_{i=1}^{N}\prod_{j=1}^{K} r_{ij}^{Z_{ij}}$$

where $\mathbf{Z} \in \{0,1\}^{N\times K}$ is a matrix of trajectory-state assignments and $\mathbf{r} \in \mathbb{R}^{N\times K}$ is *posterior_assignment_probabilities*.

The distribution is normalized over all trajectories: $\sum_{j=1}^{K} r_{ij} = 1$ for any $i$.

**property prior_dirichlet_param: numpy.ndarray**

Shape *self.shape*; the parameter to the Dirichlet prior distribution over state occupations.

*saSPT* uses uniform priors by default.

In math:

$$\boldsymbol{\tau} \sim \text{Dirichlet}\left(\boldsymbol{\alpha}_0\right)$$

where $\boldsymbol{\tau}$ are the state occupations and $\boldsymbol{\alpha}_0$ is *prior_dirichlet_param*.

**property posterior_dirichlet_param: numpy.ndarray**

Shape *self.shape*; the parameter to the Dirichlet posterior distribution over state occupations.

In math:

$$\boldsymbol{\tau} \mid \mathbf{X} \sim \text{Dirichlet}\left(\boldsymbol{\alpha} + \boldsymbol{\alpha}_0\right)$$

where $\boldsymbol{\tau}$ are the state occupations, $\boldsymbol{\alpha}$ is *posterior_dirichlet_param*, and $\boldsymbol{\alpha}_0$ is *prior_dirichlet_param*.

**property prior_occs: numpy.ndarray**

Shape *self.shape*; mean occupations of each state in the parameter grid under the prior distribution. Since *saSPT* uses uniform priors, all values are equal to `1.0/self.n_states` $(1/K)$.

$$\boldsymbol{\tau}^{(\text{prior})} = \mathbb{E}\left[\boldsymbol{\tau}\right] = \int \boldsymbol{\tau}\, p(\boldsymbol{\tau})\, d\boldsymbol{\tau} = \frac{1}{K}$$

where $p(\boldsymbol{\tau})$ is the prior distribution over the state occupations and $K$ is the number of states.

**property naive_occs: numpy.ndarray**

Shape *self.shape*; naive estimate for the occupations of each state in the parameter grid.

These are obtained from the naive trajectory-state assignment probabilities by normalizing a weighted sum across all trajectories:

$$\tau_j^{(\text{naive})} \propto \eta_j^{-1} \sum_{i=1}^{N} n_i r_{ij}$$

where $n_i$ is the number of jumps in trajectory $i$, $r_{ij}$ is the naive probability to assign trajectory $i$ to state $j$, and $\eta_j$ is a potential correction factor for defocalization.

The naive state occupations are less precise than the posterior occupations, but also require fewer trajectories to estimate. As a result, they provide a useful "quick and dirty" estimate for state occupations, and also a sanity check when comparing against the posterior occupations.

**property posterior_occs: numpy.ndarray**

Shape *self.shape*; mean occupations of each state in the parameter grid under the posterior distribution:

$$\boldsymbol{\tau}^{(\text{posterior})} = \mathbb{E}\left[\boldsymbol{\tau}|\mathbf{X}\right] = \int \boldsymbol{\tau}\, p(\boldsymbol{\tau}|\mathbf{X})\, d\boldsymbol{\tau}$$

**property posterior_occs_dataframe: pandas.DataFrame**

Representation of *posterior_occs* as a *pandas.DataFrame*. Each row corresponds to a single state (element in the parameter grid), and the columns include the parameter values, naive occupation, and posterior occupation of that state.

Example:

```
>>> from saspt import sample_detections, StateArray, RBME

# Make a toy StateArray
>>> SA = StateArray.from_detections(sample_detections(),
...     likelihood_type=RBME, pixel_size_um = 0.16,
...     frame_interval = 0.00748, focal_depth = 0.7)

# Get the posterior distribution as a pandas.DataFrame
>>> posterior_df = SA.posterior_occs_dataframe
>>> print(posterior_df)

      diff_coef  loc_error  naive_occupation  mean_posterior_occupation
0          0.01      0.000          0.000002                   0.000002
1          0.01      0.002          0.000003                   0.000002
2          0.01      0.004          0.000004                   0.000003
3          0.01      0.006          0.000005                   0.000004
4          0.01      0.008          0.000009                   0.000007
...         ...        ...               ...                        ...
3631     100.00      0.062          0.000007                   0.000007
3632     100.00      0.064          0.000007                   0.000007
3633     100.00      0.066          0.000007                   0.000007
3634     100.00      0.068          0.000007                   0.000007
3635     100.00      0.070          0.000007                   0.000007
```

As an example calculation, we can estimate the fraction of particles with diffusion coefficients in the range 1.0 to 10.0 μm2/sec under the posterior distribution:

```
>>> diff_coefs_in_range = np.logical_and(
...     posterior_diff['diff_coef'] >= 1.0,
...     posterior_diff['diff_coef'] < 10.0)
>>> x = posterior_df.loc[diff_coefs_in_range, 'mean_posterior_occupation'].sum()
>>> print(f"Fraction of particles with diffusion coefficients between 0 and 10:
↪{x}")
0.15984985148415815
```

And just for fun, we can compare this with the estimate from the naive occupations:

```
>>> x = posterior_df.loc[diff_coefs_in_range, 'naive_occupation'].sum()
>>> print(f"Fraction of particles with diffusion coefficients between 0 and 10:
↪{x}")
0.15884454681112886
```

In this case, the naive and posterior estimates agree quite closely. We could get exactly the same result by doing

```
>>> in_range = np.logical_and(SA.diff_coefs>=1.0, SA.diff_coefs<10.0)

# Fraction of particles with diffusion coefficients in this range,
# under posterior mean occupations
>>> print(SA.posterior_occs[in_range,:].sum())
0.15984985148415815

# Fraction of particles with diffusion coefficients in this range,
# under the naive occupations
>>> print(SA.naive_occs[in_range,:].sum())
0.15884454681112886
```

**property diff_coefs: numpy.ndarray**

1D *numpy.ndarray*, the set of diffusion coefficients on which this state array is defined, corresponding to one of the axes in the parameter grid.

Not all likelihood functions may use diffusion coefficient as a parameter. In those cases, *diff_coefs* is an empty *numpy.ndarray*.

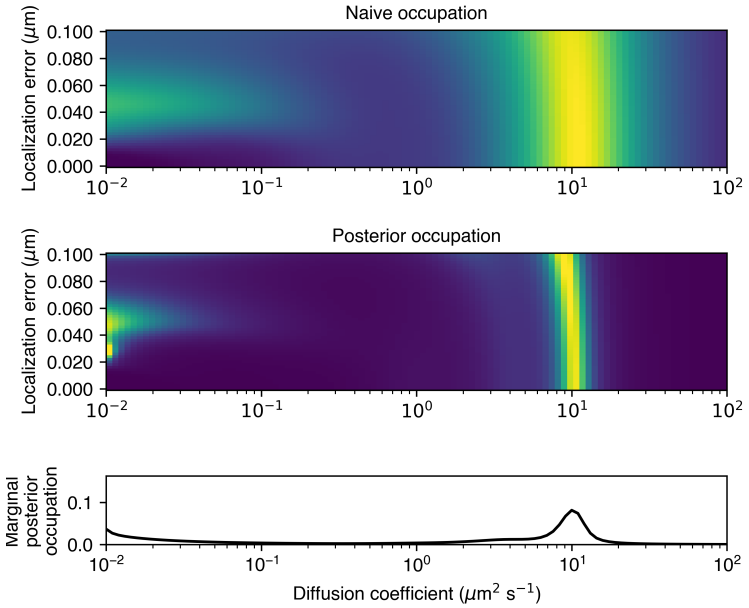
**marginalize_on_diff_coef:**

Alias for Likelihood.marginalize_on_diff_coef.

**plot_occupations**(*self*, *out_png: str*, ***kwargs*)

Plot the naive and posterior occupations. The exact plot will depend on *likelihood_type*. For the RBME likelihood, three panels are shown:

- the upper panel shows the naive state occupations
- the middle panel shows the posterior state occupations
- the lower panel shows the posterior state occupations marginalized on diffusion coefficient

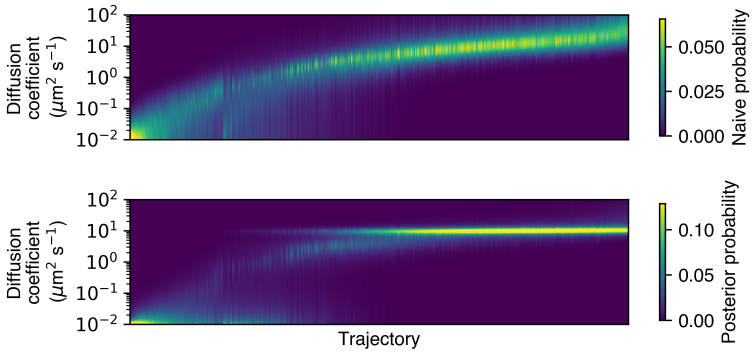


**Parameters**

- **out_png** (*str*) – save path for this plot
- **kwargs** – additional kwargs to the plotting function

**plot_assignment_probabilities**(*self*, *out_png: str*, ***kwargs*)

Plot the naive posterior trajectory-state assignments, marginalized on diffusion coefficient. Useful for judging heterogeneity between trajectories.



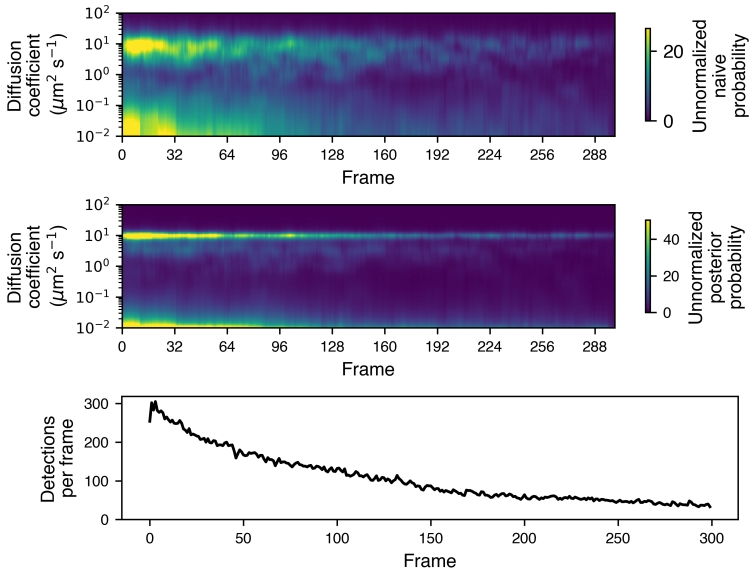

**Parameters**

- **out_png** (*str*) – save path for this plot
- **kwargs** – additional kwargs to the plotting function

**plot_temporal_assignment_probabilities**(*self*, *out_png: str*, *frame_block_size: int = None*, ***kwargs*)

Plot the posterior diffusion coefficient as a function of frame. Useful to judge whether the posterior distribution is stationary. This may not be the case if, for instance, there are lots of tracking errors in the earlier, denser part of the SPT movie.

The color map is proportional to the number of jumps in each frame block by default. To disable this, set the *normalize* parameter to *True*.



**Parameters**

- **out_png** (*str*) – save path for this plot
- **frame_block_size** (*int*) – number of frames per temporal bin. If None, attempts to find an appropriate block size for the SPT movie.
- **kwargs** – additional kwargs to the plotting function

**plot_spatial_assignment_probabilities**(*self*, *out_png: str*, ***kwargs*)

Plot the mean posterior diffusion coefficient as a function of space. Currently experimental and subject to change.

**Parameters**

- **out_png** (*str*) – save path for this plot
- **kwargs** – additional kwargs to the plotting function

### 2.5.6 StateArrayDataset

**class StateArrayDataset**(*self*, *paths: pandas.DataFrame*, *likelihood:* Likelihood, *params:* StateArrayParameters, *path_col: str*, *condition_col: str = None*, ***kwargs*)

Implements routines to run state arrays at the dataset level. Parallelizes inference across multiple files and provides visualization methods to compare different experimental conditions.

The structure of the SPT dataset is specified with the **paths** argument. This is a *pandas.DataFrame* that encodes the path and experimental condition for all files in the dataset. Only two columns in this DataFrame are recognized:

- **path_col** (*required*): encodes the path to each SPT trajectory file
- **condition_col** (*optional*): encodes the experimental condition to which that file belongs

If the DataFrame contains other columns, they are ignored.

An example is provided by the file experiment_conditions.csv under the **examples** folder in the saSPT repo. The **path_col** is *filepath* and **condition_col** is *condition*:

```
$ cat experiment_conditions.csv | head
filepath,condition
u2os_ht_nls_7.48ms/region_0_7ms_trajs.csv,HaloTag-NLS
u2os_ht_nls_7.48ms/region_10_7ms_trajs.csv,HaloTag-NLS
...
```

Running from the *saspt/examples* directory, we can construct a StateArrayDataset as follows:

```
import pandas as pd
from saspt import StateArrayDataset, RBME

# Load the paths DataFrame
paths = pd.read_csv("experiment_conditions.csv")

# Settings for state array inference
settings = dict(
    likelihood_type = RBME,    # type of likelihood function to use
    pixel_size_um = 0.16,      # camera pixel size in microns
    frame_interval = 0.00748, # frame interval in seconds
```

(continues on next page)

(continued from previous page)

```
    focal_depth = 0.7,          # objective focal depth in microns
    path_col = 'filepath',      # column in *paths* encoding file path
    condition_col='condition',# column in *paths* encoding
                                # experimental condition
    num_workers = 4,            # parallel processes to use
    progress_bar = True,        # show progress
)

# Make a StateArrayDataset with these settings
with StateArrayDataset.from_kwargs(paths, **settings) as SAD:
    print(SAD)
```

The output of this script is:

```
StateArrayDataset:
    likelihood_type    : rbme
    shape              : (101, 36)
    n_files            : 22
    path_col           : filepath
    condition_col      : condition
    conditions         : ['HaloTag-NLS' 'RARA-HaloTag']
```

StateArrayDataset implements several methods to get information about a dataset. Continuing with the example above,

```
# Make a StateArrayDataset with these settings
with StateArrayDataset.from_kwargs(paths, **settings) as SAD:

    # Save some statistics on each SPT file, including the number
    # of trajectories, trajectory length, etc.
    SAD.raw_track_statistics.to_csv("raw_track_statistics.csv", index=False)

    # Save the posterior state occupations to a file
    SAD.marginal_posterior_occs_dataframe.to_csv(
        "marginal_posterior_distribution.csv", index=False)

    # Make some plots comparing the naive state occupations across all
    # files in this dataset
    SAD.naive_heat_map("naive_heat_map.png")
    SAD.naive_line_plot("naive_line_plot.png")

    # Make some plots comparing the posterior state occupations across all
    # files in this dataset
    SAD.posterior_heat_map("posterior_heat_map.png")
    SAD.posterior_line_plot("posterior_line_plot.png")

    # Do some kind of specific calculation; for example, calculate the
    # fraction of particles with diffusion coefficients less than 1.0
    # in each file
    print(SAD.marginal_posterior_occs[:,SAD.diff_coefs<1.0].sum(axis=1))
```

**Warning:** An important parameter when constructing the StateArrayDataset is *num_workers*, the number of parallel processes. This should **not** be set higher than the number of CPUs you have access to. Otherwise you'll suffer performance drops.

**Parameters**

- **paths** (*pandas.DataFrame*) –
- **likelihood_type** (*str*) –
- **path_col** (*str*) –
- **condition_col** (*str*) –
- **pixel_size_um** (*float*) –
- **frame_interval** (*float*) –
- **focal_depth** (*float*) –
- **num_workers** (*int*) –
- **progress_bar** (*bool*) –

**classmethod from_kwargs**(*cls*, *paths: pandas.DataFrame*, *likelihood_type: str*, *path_col: str*, *condition_col: str = None*, ***kwargs*)

**Parameters**

- **paths** (*pandas.DataFrame*) –
- **likelihood_type** (*str*) –
- **path_col** (*str*) –
- **condition_col** (*str*) –
- **pixel_size_um** (*float*) –
- **frame_interval** (*float*) –
- **focal_depth** (*float*) –
- **num_workers** (*int*) –
- **progress_bar** (*bool*) –

**Returns**

new instance of *StateArrayDataset*

**likelihood:** *Likelihood*

The likelihood function used by all of the state arrays in this StateArrayDataset.

**property n_files:** **int**

Total number of files in this StateArrayDataset.

**property shape:** **Tuple[int]**

**property likelihood_type:** **str**

Name of the likelihood function; equivalent to *StateArrayDataset.likelihood.name*

**property n_diff_coefs: int**

Number of distinct diffusion coefficients in the parameter grid corresponding to this Likelihood function.

If *self.likelihood* does not use diffusion coefficient as a parameter, is 0.

**property jumps_per_file: numpy.ndarray**

Shape *(n_files,)*, the number of observed jumps in each file (after preprocessing).

**property raw_track_statistics: pandas.DataFrame**

Raw trajectory statistics for this dataset. Each row of the DataFrame corresponds to one file, and each column to an attribute of that file.

Continuing with the previous example,

```
>>> with StateArray(paths, **settings) as SAD:
...     track_stats = SAD.raw_track_statistics

>>> print(track_stats.columns)
Index(['n_tracks', 'n_jumps', 'n_detections', 'mean_track_length',
   'max_track_length', 'fraction_singlets', 'fraction_unassigned',
   'mean_jumps_per_track', 'mean_detections_per_frame',
   'max_detections_per_frame', 'fraction_of_frames_with_detections',
   'filepath', 'condition'],
  dtype='object')

>>> print(track_stats[['mean_track_length', 'fraction_singlets', 'condition']])
    mean_track_length  fraction_singlets     condition
0            1.636783           0.839129   HaloTag-NLS
1            2.075513           0.666734   HaloTag-NLS
2            1.784457           0.746812   HaloTag-NLS
3            1.986613           0.675709   HaloTag-NLS
4            2.004172           0.698274   HaloTag-NLS
..                ...                ...           ...
17           3.881071           0.571429  RARA-HaloTag
18           3.826364           0.557824  RARA-HaloTag
19           3.423219           0.591547  RARA-HaloTag
20           3.682189           0.536682  RARA-HaloTag
21           3.520319           0.595750  RARA-HaloTag

[22 rows x 3 columns]
```

Notice that the trajectories from the *HaloTag-NLS* conditions are shorter and more likely to be singlets than the trajectories in the *RARA-HaloTag* condition.

**property processed_track_statistics: pandas.DataFrame**

Trajectory statistics for this dataset after preprocessing. Each row of the DataFrame corresponds to one file, and each column to an attribute of that file.

This is identical in form to *raw_track_statistics*.

**property marginal_naive_occs: numpy.ndarray**

Shape *(n_files, n_diff_coefs)*, naive state occupations for each file marginalized on diffusion coefficient.

**property marginal_posterior_occs: numpy.ndarray**

Shape *(n_files, n_diff_coefs)*, posterior state occupations for each file marginalized on diffusion coefficient.

**property marginal_posterior_occs_dataframe: pandas.DataFrame**

*pandas.DataFrame* representation of *marginal_posterior_occs*. Each row corresponds to a single state in a single file, so that the total number of rows is equal to *n_files * n_diff_coefs*.

Continuing the example from above,

```
>>> SAD = StateArrayDataset.from_kwargs(paths, **settings)
>>> diff_coefs = SAD.diff_coefs
>>> df = SAD.marginal_posterior_occs_dataframe

# Calculate the estimated fraction of trajectories with diffusion
# coefficients below 0.1 μm2/sec for all files in this dataset
>>> print(df.loc[df['diff_coef'] < 0.1].groupby('filepath')['posterior_
↪occupation'].sum())
filepath
u2os_ht_nls_7.48ms/region_0_7ms_trajs.csv     0.173923
u2os_ht_nls_7.48ms/region_10_7ms_trajs.csv    0.067899
u2os_ht_nls_7.48ms/region_1_7ms_trajs.csv     0.165322
u2os_ht_nls_7.48ms/region_2_7ms_trajs.csv     0.020263
u2os_ht_nls_7.48ms/region_3_7ms_trajs.csv     0.101379
                                                 ...
u2os_rara_ht_7.48ms/region_5_7ms_trajs.csv    0.364910
u2os_rara_ht_7.48ms/region_6_7ms_trajs.csv    0.430909
u2os_rara_ht_7.48ms/region_7_7ms_trajs.csv    0.426619
u2os_rara_ht_7.48ms/region_8_7ms_trajs.csv    0.350441
u2os_rara_ht_7.48ms/region_9_7ms_trajs.csv    0.553296
Name: posterior_occupation, Length: 22, dtype: float64
```

**clear**(*self*)

Clear all cached attributes.

**apply_by**(*self*, *col: str*, *func: Callable*, *is_variadic: bool = False*, ***kwargs*)

Apply a function in parallel to groups of files identified by a common value in *self.paths[col]*. Essentially equivalent to a parallel version of *self.paths.groupby(col)[self.path_col].apply(func)*.

*func* should have the signature *func(paths: List[str], **kwargs)* if *is_variadic == False*, or *func(*paths: str, **kwargs)* if *is_variadic == True*.

**Parameters**

- **col** (*str*) – a column in *self.paths* to group files by
- **func** (*Callable*) – function to apply to each file group
- **is_variadic** (*bool*) – *func* is a variadic function
- **kwargs** – additional keyword arguments to *func*

**Returns**

**result** (*list*), **group_names** (*List[str]*)

**infer_posterior_by_condition**(*self*, *col: str*, *normalize: bool = False*)

Group files by common values of *self.paths[col]* and infer the marginal posterior occupations for each file group.

**Parameters**

- **col** (*str*) – column in *self.paths* to group by
- **normalize** (*bool*) – normalize the posterior occupations over all states for each file

**Returns**

**posterior_occs** (*numpy.ndarray*), **group_names** (*List[str]*). **posterior_occs** is a 2D array of shape *(n_groups, n_diff_coefs)* with the marginal posterior occupations for each file group, and **group_names** is the names of each group.

**calc_marginal_naive_occs**(*self*, **track_paths: str*) → numpy.ndarray:

Calculate the naive state occupations (marginalized on diffusion coefficient) for one or more files. If multiple files are passed, runs on the concatenation of the detections across files.

If you want to infer the marginal naive occupations for *all* of the files in this StateArrayDataset, use *StateArrayDataset.marginal_naive_occs* instead.

**Parameters**

**track_paths** (*str*) – full paths to one or more files with detections

**Returns**

**naive_occs** (*numpy.ndarray*), 1D array of shape *(n_diff_coefs,)*, the marginal naive state occupations

**calc_marginal_posterior_occs**(*self*, **track_paths: str*) → numpy.ndarray:

Calculate the posterior state occupations (marginalized on diffusion coefficient) for one or more files. If multiple files are passed, runs on the concatenation of the detections across files.

If you want to infer the marginal naive occupations for *all* of the files in this StateArrayDataset, use *StateArrayDataset.marginal_naive_occs* instead.

**Parameters**

**track_paths** (*str*) – full paths to one or more files with detections

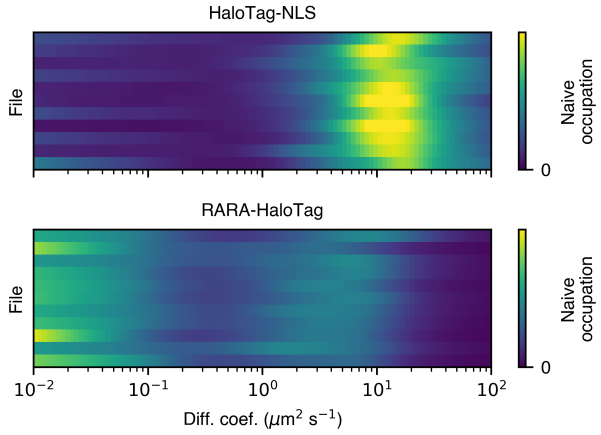
**Returns**

**naive_occs** (*numpy.ndarray*), 1D array of shape *(n_diff_coefs,)*, the marginal posterior state occupations

**naive_heat_map**(*self*, *out_png: str*, *normalize: bool = True*, *order_by_size: bool = True*, ***kwargs*)

Naive state occupations, marginalized on diffusion coefficient, shown as a heat map. Groups by condition.

With *normalize = True*:



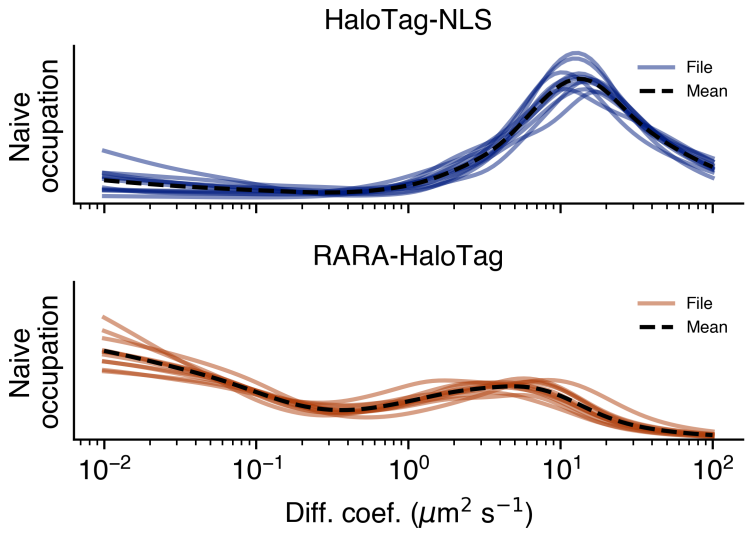

**Parameters**

- **out_png** (*str*) – save path for this plot

- **normalize** (*bool*) – normalize the state occupations for each file in the dataset. If False, the intensity for each file is proportional to the number of jumps observed in that SPT experiment.
- **order_by_size** (*bool*) – within each condition group, order the files by decreasing number of observed jumps.
- **kwargs** – additional kwargs to the plotting function

**naive_line_plot**(*self*, *out_png: str*, ***kwargs*)

Naive state occupations, marginalized on diffusion coefficient, shown as a line plot. Groups by condition.

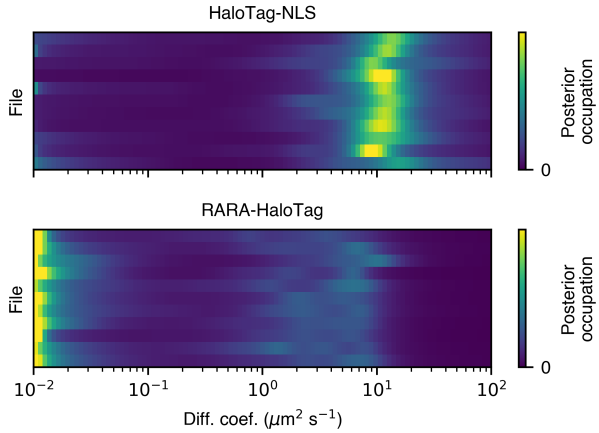



**Parameters**

- **out_png** (*str*) – save path for this plot
- **kwargs** – additional kwargs to the plotting function

**posterior_heat_map**(*self*, *out_png: str*, *normalize: bool = True*, *order_by_size: bool = True*, ***kwargs*)

Posterior mean state occupations, marginalized on diffusion coefficient, shown as a heat map. Groups by condition.

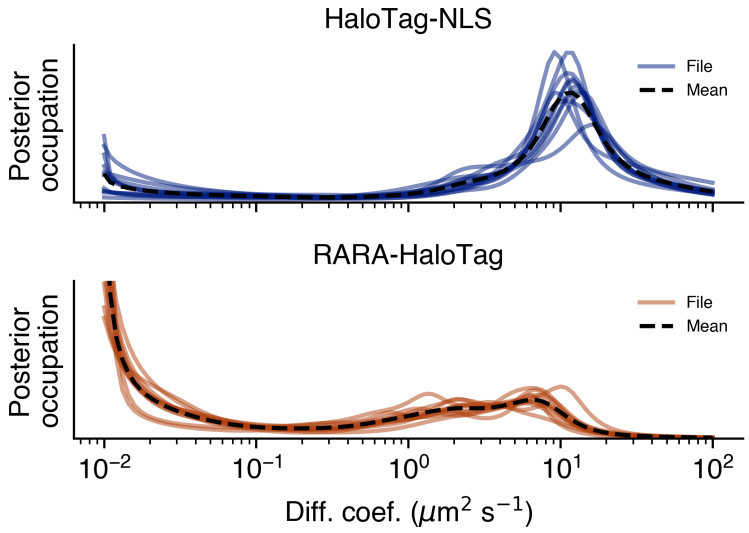


**Parameters**

- **out_png** (*str*) – save path for this plot
- **normalize** (*bool*) – normalize the state occupations for each file in the dataset. If False, the intensity for each file is proportional to the number of jumps observed in that SPT experiment.
- **order_by_size** (*bool*) – within each condition group, order the files by decreasing number of observed jumps.
- **kwargs** – additional kwargs to the plotting function

**posterior_line_plot**(*self*, *out_png: str*, ***kwargs*)

Posterior mean state occupations, marginalized on diffusion coefficient, shown as a line plot. Groups by condition.




**Parameters**

- **out_png** (*str*) – save path for this plot
- **kwargs** – additional kwargs to the plotting function

### 2.5.7 File reading

saspt.io.**is_detections**(*df: pandas.DataFrame*) → bool

Determine whether a *panda.DataFrame* is recognized as a viable set of SPT detections by the rest of *saSPT*. In particular, it must contain the following four columns:

- **frame** (`saspt.constants.FRAME`): frame index for this detection
- **trajectory** (`saspt.constants.TRACK`): trajectory index for this detection
- **x** (`saspt.constants.PX`): position of the detection in *pixels*
- **y** (`saspt.constants.PY`): position of the detection in *pixels*

Example:

```
>>> import pandas as pd
>>> from saspt.constants import FRAME, TRACK, PY, PX
>>> from saspt.io import is_detections
>>> print(is_detections(pd.DataFrame(index=[], columns=[FRAME, TRACK, PY, PX],␣
↪dtype=object)))
True
>>> print(is_detections(pd.DataFrame(index=[], columns=[FRAME, PY, PX],␣
↪dtype=object)))
False
```

**Parameters**
**df** (*pandas.DataFrame*) – each row corresponding to a detection

**Returns**
*bool*

saspt.io.**load_detections_from_file**(*filepath: str*) → pandas.DataFrame

Load detections from a file in one of the currently recognized formats.

At the moment, *saSPT* only recognizes a single file format for trajectories: a CSV where each row corresponds to a detection and the columns contain at minimum **frame**, **trajectory**, **x**, and **y**.

**Parameters**
**filepath** (*str*) – path to the file containing detections

**Returns**
**detections** (*pandas.DataFrame*), the set of detections

saspt.**load_detections**(**filepaths: str*) → pandas.DataFrame

Load detections from one or more files and concatenate into a single *pandas.DataFrame*. Increments trajectory indices, so that indices between detections from different files do not collide.

Example (using some files from the *saSPT* repo):

```
>>> from saspt.io import load_detections
>>> detections = load_detections(
...     'tests/fixtures/small_tracks_0.csv',
...     'tests/fixtures/small_tracks_1.csv'
... )
>>> print(detections)
     trajectory  ...  dataframe_idx
0             0  ...              0
1             0  ...              0
2             0  ...              0
3             1  ...              0
4             1  ...              0
..          ...  ...            ...
449         107  ...              1
450         107  ...              1
451         107  ...              1
452         107  ...              1
453         107  ...              1

[454 rows x 6 columns]
```

**Parameters**
**filepaths** (*str*) – one or more paths to files containing detections. Must be in a format recognized by *saspt.io.load_detections_from_file*.

**Returns**
**detections** (*pandas.DataFrame*), indexed by detection

saspt.io.**empty_detections**() → pandas.DataFrame

Return an empty set of detections. Useful mostly for tests.

**Returns**
**empty_detections** (*pandas.DataFrame*)

saspt.io.**sample_detections**() → pandas.DataFrame

Return a small, simple set of detections. Useful for illustrations and quick demos, especially in these docs.

```
>>> from saspt import sample_detections
>>> detections = sample_detections()
>>> print(detections)
              y           x  frame  trajectory
0    575.730202   84.828673      0       13319
1    538.416604  485.924667      0        1562
2    107.647631   61.892363      0         363
3    151.893969   63.246361      0         992
4    538.737277  485.856905      1        1562
..          ...         ...    ...         ...
491  365.801274   70.689108    296       14458
492  409.236744   10.312949    296       14375
493  366.475688   70.559735    297       14458
494  363.350134   67.585339    298       14458
495  360.006572   70.511980    299       14458

[496 rows x 4 columns]
```

**Returns**
*pandas.DataFrame* with columns *frame*, *trajectory*, *y*, and *x*

saspt.io.**concat_detections**(**detections: pandas.DataFrame*) → pandas.DataFrame

Concatenate multiple detection DataFrames while incrementing trajectory indices to prevent index collisions.

**Parameters**
**detections** (*pandas.DataFrame*) – one or more DataFrames containing detections

**Returns**
**concatenated_detections** (*pandas.DataFrame*)

### 2.5.8 Utilities

saspt.utils.**track_length**(*detections: pandas.DataFrame*) → pandas.DataFrame

Add a new column to a detection-level DataFrame with the length of each trajectory in frames.

Example:

```
>>> from saspt import sample_detections
>>> detections = sample_detections()
>>> from saspt.utils import track_length

# Calculate length of each trajectory in frames
>>> detections = track_length(detections)
>>> print(detections)
              y           x  frame  trajectory  track_length
0    575.730202   84.828673      0       13319           247
1    538.416604  485.924667      0        1562            13
2    107.647631   61.892363      0         363             3
3    151.893969   63.246361      0         992             8
4    538.737277  485.856905      1        1562            13
..          ...         ...    ...         ...           ...
491  365.801274   70.689108    296       14458             7
492  409.236744   10.312949    296       14375             1
493  366.475688   70.559735    297       14458             7
494  363.350134   67.585339    298       14458             7
495  360.006572   70.511980    299       14458             7

[496 rows x 5 columns]

# All detections in each trajectory have the same track length
>>> print(detections.groupby('trajectory')['track_length'].nunique())
trajectory
363      1
413      1
439      1
542      1
580      1
        ..
14174    1
14324    1
14360    1
14375    1
14458    1
Name: track_length, Length: 100, dtype: int64

# Mean trajectory length is ~5 frames
>>> print(detections.groupby('trajectory')['track_length'].first().mean())
4.96
```

**Parameters**

**detections** (*pandas.DataFrame*) –

**Returns**

*pandas.DataFrame*, the input DataFrame with a new column *track_length* (saspt.constants.

TRACK_LENGTH)

saspt.utils.**assign_index_in_track**(*detections: pandas.DataFrame*) → pandas.DataFrame

Given a set of detections, determine the index of each detection in its respective trajectory.

Sorts the input.

**Parameters**
**detections** (*pandas.DataFrame*) – input set of detections

**Returns**
*pandas.DataFrame*, input with the *index_in_track* column

saspt.utils.**cartesian_product**(**arrays: numpy.ndarray*) → numpy.ndarray

Take the Cartesian product of multiple 1D arrays.

**Parameters**
**arrays** (*numpy.ndarray*) – one or more 1D arrays

**Returns**
**product** (*numpy.ndarray*), 2D array. Each corresponds to a unique combination of the elements of *arrays*.

## 2.6 FAQS

### 2.6.1 Q. Does saspt provide a way to do tracking?

saspt only analyzes the *output* of a tracking algorithm; it doesn't produce the trajectories themselves. The general workflow is:

1. Acquire some raw SPT movies
2. Use a tracking algorithm to produce trajectories from the SPT movie
3. Feed the trajectories into saspt (or whatever your favorite analysis tool is)

There are lots of good tracking algorithms out there. In the sample data included in the saspt repo, we used an in-house tracking tool with a graphic user interface (quot). But - you know - we're biased.

Other popular options are TrackMate, the multiple-target tracing algorithm, or Sbalzerini's hill climbing algorithm. There are new tracking algorithms every day; use Google to see for yourself.

### 2.6.2 Q. Why doesn't saspt support input format *X*?

Because a table of detections is probably the simplest format that exists to describe trajectories, so we added it first. We're happy to expand support for additional formats (within reason) - let us know with a GitHub request.

### 2.6.3 Q. Why are the default diffusion coefficients log-spaced?

As the diffusion coefficient increases, our estimate of it becomes much more error-prone. This makes it difficult for humans to compare the occupations of states with widely varying diffusion coefficients. By plotting on a log scale, we minimize these perceptual differences so that humans can accurately compare states across the full range of biologically observed diffusion coefficients.

To demonstrate this effect, consider the likelihood function for the jumps of a 2D Brownian motion with no localization error, diffusion coefficient $D$, frame interval $\Delta t$, and $n$ total jumps. The maximum likelihood estimator for the diffusion coefficient is the mean-squared displacement (MSD):

$$\hat{D} = \frac{1}{4n\Delta t}\sum_{j=1}^{n}(\Delta x_j^2 + \Delta y_j^2)$$

where $(\Delta x_j, \Delta y_j)$ is the $j^{\text{th}}$ jump in the trajectory.

We can get the "best-case" error in this estimate using the Cramer-Rao lower bound (CRLB), which provides the minimum variance our estimator $\hat{D}$:

$$\text{Var}(\hat{D}) \geq \text{CRLB}(\hat{D}) = \frac{D^2}{n}$$

So the error in the estimate of $D$ actually increases as the *square* of $D$. If we throw in localization error (represented as the 1D spatial measurement variance, $\sigma^2$) and neglect the off-diagonal terms in the covariance matrix, we get the approximation

$$\text{Var}(\hat{D}) \geq \text{CRLB}(\hat{D}) \approx \frac{(D\Delta t + \sigma^2)^2}{n\Delta t^2}$$

Notice that this makes it even *harder* to estimate the diffusion coefficient, especially when $D\Delta t < \sigma^2$.

### 2.6.4 Q. How does `saspt` estimate the posterior occupations, given the posterior distribution?

`saspt` always uses the posterior mean. If $\boldsymbol{\alpha}$ is the parameter to the posterior Dirichlet distribution over state occupations, then the posterior mean $\boldsymbol{\tau}$ is simply the normalized Dirichlet parameter:

$$\boldsymbol{\tau} \sim \text{Dirichlet}\,(\boldsymbol{\alpha})$$

$$\mathbb{E}\left[\boldsymbol{\tau}|\boldsymbol{\alpha}\right] = \frac{1}{\sum_{j=1}^{K}\alpha_j}\begin{bmatrix}\alpha_1 \\ ... \\ \alpha_K\end{bmatrix}$$

We prefer the posterior mean to max *a posteriori* (MAP) or other estimators because it is very conservative and minimizes the occurrence of spurious features.

### 2.6.5 Q. I want to measure the fraction of particles in a particular state. How do I do that?

If you know the range of diffusion coefficients you're interested in, you can directly integrate the mean posterior occupations. Say we want the fraction of particles with diffusion coefficients between 1 and 10 $\mu$m$^2$/sec:

```
>>> occupations = SA.posterior_occs_dataframe
>>> in_range = (occupations['diff_coef'] >= 1.0) & (occupations['diff_coef'] < 10.0)
>>> print(occupations.loc[in_range, 'mean_posterior_occupation'].sum())
```

That being said, `saspt` does *not* provide any way to determine the endpoints for this range, and that is up to you or the methods you develop.

### 2.6.6 Q. What is defocalization?

Examining the movie in the section *Background*, you may notice that the particles are constantly wandering into and out of focus. The faster they move, the faster they escape the microscope's focus.

As it turns out, this behavior (termed "defocalization") has a dangerous side effect. If we want to know the *state occupations* - the fraction of proteins in a particular state - we may be tempted to report the *fraction of observed trajectories in that state*. The problem is that particles in fast states contribute many short trajectories, because they can wander in and out of focus multiple times before bleaching. By contrast, particles in slow states produce a few long trajectories; they don't move fast enough to reenter the focal volume before bleaching.

As a result, a mixture of equal parts fast and slow particles does *not* produce equal parts fast and slow trajectories.

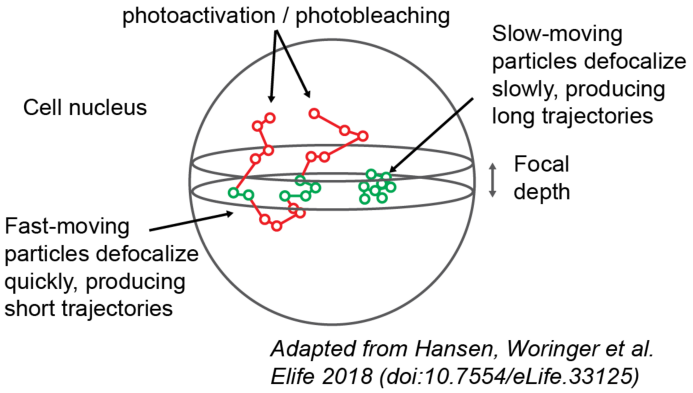



Fig. 5: Illustration of the defocalization problem. Particles inside the focus (green circles) are recorded by the microscope; particles outside the focus (red circles) are not recorded. Particles that traverse the focus multiple times are "fragmented" into multiple short trajectories.

Defocalization is the reason why the "MSD histogram" method - one of the most popular approaches to analyze protein tracking data - yields inaccurate results when applied to 2D imaging. A more detailed discussion can be found in the papers of Mazza and Hansen and Woringer.

`saspt` avoids the state estimation problem by computing state occupations in terms of *jumps* rather than trajectories. Additionally, an analytical correction factor (analogous to the empirical correction factor from Hansen and Woringer) can be applied to the data by passing the `focal_depth` parameter when constructing a `StateArray` or `StateArrayDataset` object.

## 2.7 References

The source code for `saspt` is publicly available under an MIT license.

If you use state arrays in your research, please cite the state array paper (currently in preprint):

Alec Heckert, Liza Dahal, Robert Tijan, Xavier Darzacq (2022) **Recovering mixtures of fast-diffusing states from short single-particle trajectories** eLife 11:e70169 (https://doi.org/10.7554/eLife.70169)

The trajectories used in the examples for the `saspt` repo were generated using the quot tool at Robert Tjian and Xavier Darzacq's laboratory at University of California, Berkeley. Dyes used in these experiments were generously provided by the laboratory of Luke Lavis at Janelia Research Campus.

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX

## Symbols



## A



## B



## C



## D



## E



## F



## G



## I



## J



## L



## M



## N

## P



## R



## S



## T



## U