tracklib developer's guide

This document is intended to explain and document style and implementation choices made in the library. Before deciding that stupid design choices were made, consult the corresponding section here. (Though there might of course be explicitly made decisions that nevertheless turn out to be stupid).

This document is organized like the library, with individual modules/submodules/functions each having their own section. For completeness, everything in the library should have a section here, even if that section is empty. Since ideally the reasoning behind most things would be clear from documentation/code, this document serves merely as a "last resort" of kinds. For all those things that the user doesn't get to have an opinion on, but where it might be relevant for devs to know the reasoning.

Side note: workflow

While python of course does not have to be compiled, there is a Makefile for this library. This can be used to run the tests and check for code coverage, compile this Developer's guide into a pdf, and build the documentation. Recommended use is to either run `make all` whenever you change anything, or, if you need the output some other place, adapt the `DUMPPATH` in the Makefile and run `make myall`.

tracklib

Docstring notation: use double backticks (``) around code expressions (actual examples go into doctest blocks!), single backticks (`) around any single identifier (module names, functions, attributes, ...), and the construction `!var` for identifiers that Sphinx should not attempt to generate links for (such as np.ndarray, keyword arguments, etc.)

trajectory

Trajectory

The main purpose of this class is to provide a common type for trajectories of all kinds, with differing spatial dimension or number of loci. The main functionality of the base class are the modifier functions (abs, diff, relative, ...) that can be used to perform standard processing tasks.

Any specific trajectory will be of a subclass of Trajectory, such that functionality that is specific to say two-locus trajectories in three dimensions can be implemented. It is unlikely that this will find use, but the specializations depending on either number of loci or spatial dimensions prove to be useful.

The plotting functions might undergo review at some point.

We chose to base this library on constant framerate trajectories, because some analysis methods (e.g. MSD) do not work as well with freely spaced trajectories.

> Currently the paradigm concerning the actual data is to try and avoid ever having to directly access it from outside the class. This leads to complications (see []-operator), so maybe we should move away from it? It is mostly a style question...

EDIT: Resorted to accepting some direct access to the data. The new paradigm: for everything that requires the (N, T, d) shape, you can just as well access Trajectory.data directly. Most of the time, the []-operator will probably be preferred anyways.

fromArray()

N, T, d

___len___()

___getitem___()

When providing element access, do we return a three dimensional array, or do we squeeze it (remove single entry dimensions)? There are arguments for both: on the one hand, keeping dimensionality definite means that we know exactly what to expect from the []-operator. On the other hand, it is annoying to have a bunch of single entry dimensions around; consider accessing one time point of a single locus trajectory: non-squeezed this would give a (1, 1, d) array, so we'd have to write something like `traj[t][0, 0, :]`, which is ugly.

After some deliberation, the best solution seems to be to squeeze the N and T dimensions, but leave d as it is. When processing trajectories we usually will know (or check) N, and what happens to T is determined by the user-provided slice. We do for the most part want to write analysis methods that are agnostic to d, so polymorphism seems useful here. The only problem with this solution is that for some "naturally 1d" trajectories (such as absolute distances between loci) it might be annoying to carry that extra dimension. I deem the polymorphism argument to be stronger though.

Modifiers: abs, diff, relative, dims

Note that fromArray() already copies the array passed to it, so there's no need to do that explicity. Otherwise, the key for these functions is that they're chainable:

```
traj_processd = traj.relative().abs().diff()
```

(or something like that).

#yield_dims()

More of a sketch of an idea. Would it be useful to have something like this?

plot_vstime()

plot_spatial()

What exactly to do here depends on N and d, but independently. This is thus implemented in Trajectory_?d, calling the 'raw' plotting function in Trajectory_?N.

N12Error

This special exception might be useful if there were more use cases for it, which might happen as the library grows. Right now it's a bit pointless.

taggedset

TaggedSet

The idea here is a many-to-many dict: have a bunch of data that can belong to one or more subsets of the whole data set. Of course one subset will usually also contain more than one datum, thus many-to-many. It is very natural to then select some of these subsets for processing. For practically all purposes the class will then behave as if it contained only those data in the current selection. The idea for usage is thus: load all data whatsoever into one TaggedSet object, then work with subsets of this.

Interfaces: this class actually does implement the Sequence interface, implicitly. We do not implement the Set/MutableSet interface though, because dealing with copy operations would be tricky: we want `__iter__` to return just the data, no tags, but the Set functions assume that iterating through the Set gives full information. Apart from that the functionality added by Set/MutableSet

(comparisons and set operations) is not particularly relevant to this class, so we resort to implementing just the &= operator by hand, because it's useful. Note that `mergein()` has slightly more functionality though.

makeTagsSet()

add()

___iter___(), ___call___()

___len___()

___getitem___()

makeSelection(), refineSelection()

The default value for `logic` is debatable. Depending on how the class is used, `any` or `all` can make more sense.

saveSelection(), restoreSelection()

Need the copying to prevent accidentally giving away access to `self._selection`.

copySelection()

Should be mostly unnecessary (see the paradigm about using just one data set above), but who knows.

mergein(), ___iand___()

addTags(), tagset()

filter(), process()

map_unique()

clean

split_trajectory_at_big_steps()

It might make sense to expand the capabilities of this function to N=2 (TODO)

split_dataset_at_big_steps()

load

csv

evalSPT()

Basically a legacy function, since now this job is done by `csv()`. However, it might still be nice to have this module organized by file type, let's see how this develops as we add more supported file types.

util.mcmc

Sampler

propose_update(), logL(), callback_logging()

Note that `self.stepsize` will only be set in `run()`. That means that if it is used in these overridden methods, they won't run outside of the sampler. This seems to be reasonable, so is not considered a bug.

configure()

We choose this implementation over e.g. passing a dict to run(), because it is clearer / easier to document. While this might sound like a non-reason, it does imply that this way usage will be clearer. For example it makes it a bit harder to reuse this config dict somewhere else (because we do not allow additional entries), such that we avoid giant, undocumentable config messes.

run()

Returns logL at all steps (not just after burn in), because we mostly need the likelihood history to check convergence, which ideally happens during burn in.

util.util

Currently just a graveyard, might be deprecated.

models.rouse

Likelihood is implemented as an individual function instead of a method mainly for a conceptual reason: the likelihood gives a "score" for a combination of (trace, looptrace, model). There is no reason why one of these should be preferred, so it makes more sense to have the likelihood separately, rather than integrating it into Model.

Model

Note the two different modes for dynamics: propagate() propagates an ensemble (mean + covariance/sem), while evolve() evolves an explicit conformation.

___eq___, ___repr___

give_matrices()

setup_dynamics()

check_setup_called()

_propagate_ode()

Careful with integrators: the three Runge-Kutta integrators 'RK45', 'RK23', and 'DOP853' were seen to give covariance matrices with negative eigenvalues. This was not observed for 'Radau', 'BDF', 'LSODA', of which 'LSODA' seemed to be the fastest.

_propagate_exp()

propagate()

conf_ss()

evolve()

conformations_from_looptrace()

likelihood

It is a bit unclear, where the parameters for the measurement process should be stored. For now, we decided to shove the measurement vector into the model,

while the localization error stays separate / goes into Trajectory.meta. There might be better solutions.

__likelihood__filter()

The `if noise == 0` block is nonsense. I haven't found a good way of dealing with zero noise in this method yet.

__likelihood__direct()

models.statgauss

sampleMSD

subtractMean: is it possible to incorporate finite-size effects here? (for a finite trajectory, the mean actually shouldn't be zero exactly, but follow some distribution that is tightly peaked around zero; is this independent of the displacements?)

dataset

control

Should this be merged with dataset somehow? They perform very similar tasks.

analysis.chi2

chi2vsMSD()

summary__plot()

analysis.kld

perezcruz()

Need to update: incorporate parity (due to library restructuring)

analysis.kli

traj_likelihood()

LoopSequence

toLooptrace(), fromLooptrace()

numLoops()

plottable()

LoopSequenceMCMC

The trick with copying attributes from the parent class is supposed to make Sphinx recognize these and print the docstring. Should be removed if it screws up anything important.

In the other direction, we do not want propopse_update() and logL() to show up in the documentation, because the user should not interact with them. Thus we simply set the docstring to the empty string.

setup()

propose_update()

logL()

LoopTraceMCMC

propose_update

logL

analysis.msd

MSDtraj

MSDdataset

MSD

Does this aggregate make sense?

scaling

analysis.plots

length_distribution()

msd_overview()

trajectories_spatial()

distance_distribution()

tests

Not strictly part of the library, but here are notes on the tests/test.py script:

For analysis methods it is difficult to check quantitative correctness (will maybe come at some point, with example data). For now, we mostly just check that all the code runs and passes some basic consistency checks.