Flowtracks Documentation

Release 1.0

Yosef Meller

February 23, 2017

Contents

Flowtracks is a Python package for manipulating a trajectory database obtained from 3D Particle Tracking Velocimetry (3D-PTV) analysis. It understands the most common output formats for 3D-PTV trajectories used in the 3D-PTV community, and adds its own HDF5-based format, which allows faster and more flexible access to the trajectory database.

Contents:

Contents

CHAPTER 1

Modules Containing Flowtracks Basic Data Structures

The data structures of `ParticleSet` and its children, `ParticleSnapshot` (for frames) and `Trajectory` (for trajectories) are found in the module `flowtracks.trajectory`. Also in the modules are some functions to create and manipulate these structures. This page provides a reference for the content of this module.

class `flowtracks.trajectory.`**`Frame`**
This is basically a structure with no fancy behaviour. When it is returned from a Flowtracks function, it has two attributes, `particles` and `tracers` - each pointing to a *`ParticleSnapshot`* object holding data for particles of the respective type.

class `flowtracks.trajectory.`**`ParticleSet`**(*pos*, *velocity*, ***kwds*)
A base class for manipulting particle data. Knows how many particles it has, and holds a varying number of particle properties, each given for the entire set. Properties may be created at construction time or later.

When a property is created, it gets a setter method of the same name and a getter method prefixed with `set_`. This applies also for mandatory properties.

Arguments

- `pos`: a (t,3) array, the position of one particle in t time-points, [m].
- `velocity`: (t,3) array, corresponding velocity, [m/s].
- `kwds`: keyword arguments should be arrays whose first dimension == t. these are treated as extra attributes to be sliced when creating segments.

`__len__`()
Return the number of particles in the set.

`as_dict`()
Returns a dictionary with the "business" properties only, without all the Python bookkeeping and other stuff in the __dict__.

`create_property`(*propname*, *init_val*)
Add a property of the set, expected to be an array whose shape[0] == len(self).

Creates the method <propname>(self, selector=None). If selector is given, it will return only the selected time-points. Also creates set_<propname>(self, value, selector=None) which sets either the value over the entire trajectory or just for the selected time points (this requires the property to already exist for the full trajectory).

Arguments

- `propname`: a string, should be a valid Python identifier.
- `init_val`: the initial value for the property.

ext_schema()
Extended schema. Like *schema()* but the values of the returned dictionary are a tuple (type, shape). The shape is scalar, so it only supports 1D or 0D items.

has_property(*propname*)
Checks whether the looked-after property `propname` exists for this particle set.

schema()
Creates a dictionary keyed by property name whose values are the shape of one particle's value for that property. Example: {'pos': (3,), 'velocity': (3,)}

class `flowtracks.trajectory.`**ParticleSnapshot**(*pos*, *velocity*, *time*, *trajid*, ***kwds*)
This is one of the two main classes used for iteration over a scene. It inherits from *ParticleSet* with the added demand for a scalar time and a `trajid` property for trajectory ID (an integer unique among the scene's trajectories).

Arguments

- `pos`: a (p,3) array, the position of one particle of p, [m].
- `velocity`: (p,3) array, corresponding velocity, [m/s].
- `trajid`: (p,3) array, for each particle in the snapshot, the unique identifier of the trajectory it belongs to.
- `time`: scalar, the identifier of the frame from which this snapshot is taken.
- `kwds`: keyword arguments should be arrays whose first dimension == p. these are treated as extra attributes to be sliced when creating segments.

class `flowtracks.trajectory.`**Trajectory**(*pos*, *velocity*, *time*, *trajid*, ***kwds*)
This is one of the two main classes used for iteration over a scene. It inherits from *ParticleSet* with the added demand that a scalar trajectory ID (an integer unique amond the scene's trajectories) and a `time` property.

Arguments

- `pos`: a (t,3) array, the position of one particle in t time-points, [m].
- `velocity`: (t,3) array, corresponding velocity, [m/s].
- `time`: (t,) array, the clock ticks. No specific units needed.
- `trajid`: the unique identifier of this trajectory in the set of trajectories that belong to the same sequence.
- `kwds`: keyword arguments should be arrays whose first dimension == t. these are treated as extra attributes to be sliced when creating segments.

__getitem__(*selector*)
Gets the data for time points selected as a table of shape (n,8), concatenating position, velocity, time, broadcasted trajid.

Arguments

- `selector`: any 1d indexing expression known to numpy.

smoothed(*err_bound*, *order*)
Creates a trajectory generated from this trajectory using cubic B-spline interpolation.

Arguments

- `err_bound`: amount of deviation a particle is expeted to have around its observed place. Determines strength of smoothing.
- `order`: of the spline (odd, up to 5).

Returns

a new *Trajectory* object with the interpolated positions and velocities. If the length of the trajectory < 4, returns self.

flowtracks.trajectory.**mark_unique_rows**(*all_rows*)

Filter out rows whose position columns represent a particle that already appears, so that each particle position appears only once.

Arguments

- all_rows: an array with n rows and at least 3 columns for position.

Returns

an array with the indices of rows to take from the input such that in the result, the first 3 columns form a unique combination.

flowtracks.trajectory.**take_snapshot**(*trajects*, *frame*, *schema*)

Goes over a list of trajectories and extracts the particle data at a given time point. If the trajectory list is empty, creates an empty snapshot.

Arguments

- trajects: a list of :class:Trajectory objects to query.
- frame: the frame number to which snapshot data belongs.
- schema: a dict, {propname: shape tuple}, as given by the trajectory's *schema()*. This is only needed for consistency in the case of an empty trajectory list resulting in an empty snapshot.

Returns

a *ParticleSnapshot* object with all the particles in the given frame.

flowtracks.trajectory.**trajectories_in_frame**(*trajects*, *frame_num*, *start_times=None*, *end_times=None*, *segs=False*)

Notes the indices of trajectories participating in the frame for later extraction.

Arguments

- trajects: a list of :class:Trajectory objects to filter.
- frame_num: the time value (as found in trajectory.time()) at which the trajectory should be active.
- start_times, end_times: each a len(trajects) array containing the corresponding start/end frame number of each trajectory, respectively.
- segs: true if the trajectory should be active also in the following frame.

Returns

traj_nums = the indices of active trajectories in trajects.

CHAPTER 2

Input and Output Routines

The main entry points for using the module are the *trajectories()* function (or its counterpart :func‘iter_trajectories‘) for reading the data for a scene; and either *save_trajectories()* or *save_particles_table()* for saving scene data in, respectively, an obsolete format based on a directory of NPZ files, or in the newer, recommended, HDF5 format.

The trajectory reader, unless otherwise noted, will try to infer the format from the file name (see *infer_format()*).

The rest of the content of this module is composed of readers and writers for the various formats. They are documented here alongside the main entry points, so that users may access them directly if needed.

`flowtracks.io.`**`infer_format`**(*fname*)

Try to guess the format of a particles data file by its name.

Arguments

- `fname`: the file name from which to guess the format.

Returns

A string marking the format. Currently one of ‘acc’, ‘mat’, ‘xuap’, ‘npz’, ‘hdf’ or ‘ptvis’.

`flowtracks.io.`**`iter_trajectories_ptvis`**(*fname*, *first=None*, *last=None*, *frate=1.0*, *xuap=False*, *traj_min_len=None*)

Extract all trajectories in a directory of ptv_is/xuap files, as generated by programs in the 3d-ptv/pyptv family.

Arguments

- `fname`: a template file name representing all ptv_is/xuap files in the directory, with exactly one ‘%d’ representing the frame number. If no ‘%d’ is found, the input is assumed to be in the Ron Shnapp format- single file of concatenated ptv_is files, each stripped of the particle count line (first line) and separated from the next by an empty line.
- `first, last`: inclusive range of frames to read, rel. filename numbering.
- `frate`: frame rate, used for calculating velocities by backward derivative.
- `xuap`: The format is extended with colums for velocity and acceleration.
- `traj_min_len`: do not include trajectories shorter than this many frames.

Yields

each of the trajectories in the ptv_is data in order, as a *Trajectory* instance with velocity and acceleration.

`flowtracks.io.`**`load_trajectories`**(*res_dir*, *first=None*, *last=None*)

Load a series of trajectories and associated data from a directory containing npz trajectory files, as created by save_trajectories().

Arguments

- `res_dir`: path to the directory holding the trajectory files.

Returns

- `trajects`: a list of Trajectory objects created from the files is res_dir
- `per_traject_adds`: a dictionary of named added date. Each value is a dictionary keyed by trajid.

`flowtracks.io.`**`read_frame_data`**(*conf_fname*)

Read a configuration file in INI format, which specifies the locations where particle positions and velocities should be read from, and directly stores some scalar frame values, like particle densidy etc.

Arguments

- `conf_fname`: name of the config file

Returns

- `particle`: a Particle object holding particle properties.
- `frate`: the frame rate at which the scene was shot.
- `frame, next_frame`: Frame objects holding the tracers and particles data for the time points indicated in config, and the one immediately following it.

`flowtracks.io.`**`save_particles_table`**(*filename*, *trajects*, *trim=None*)

Save trajectory data as a table of particles, with added columns for time (frame number) and trajid - the last one may be indexed. Note that no extra (per-trajectory or meta) data is allowed here, unlike the npz save format.

Arguments

- `filename`: name of output PyTables HDF5 file to create. The 'h5' extension is recommended so that infer_format() knows what to do with it.
- `trajects`: a list of Trajectory objects to save.
- `trim`: if None, remove this many time points from each end of each trajectory before saving.

`flowtracks.io.`**`save_trajectories`**(*output_dir*, *trajects*, *per_traject_adds*, ***kwds*)

Save for each trajectory the data for this trajectory, as well as additional data attached to each trajectory, such as trajectory reconstructions. Creates in the output directory one npz file per trajectory, containing the arrays of the trajectory as well as the added arrays.

Arguments

- `output_dir`: name of the directory where output should be placed. Will be created if it does not exist.
- `trajects`: a list of Trajectory objects.
- `per_traject_adds`: a dictionary, whose keys are the array names to use when saving, and vaslues are trajid-keyed dictionaries with the actual arrays to save for each trajectory.
- `kwds`: free arrays to save in the output dir

`flowtracks.io.`**`trajectories`**(*fname*, *first*, *last*, *frate*, *fmt=None*, *traj_min_len=None*, *iter_allowed=False*)

Extract all trajectories in a given target location. The location format is interpreted based on the format of the data files, in the respective trajectories_* functions.

Trajectories of one frame are filtered out.

Arguments

- `fname`: a template file name, as needed by the appropriate suboridinate function.
- `first, last`: inclusive range of frames to read, rel. filename numbering.

- `frate`: frame rate under which the film was shot - needed for ptvis trajectories.
- `traj_min_len`: on some formats, (currently ptv_is and xuap) it is possible to filter trajectories with less frames than this, saving memory.
- `iter_allowed`: may return an iterator instead of a list.

Returns

a list (or iterator) of Trajectory objects.

`flowtracks.io.`**`trajectories_acc`**(*fname*, *first=None*, *last=None*)
Extract all trajectories in a directory of trajAcc files.

Arguments

- `fname`: a template file name representing all trajAcc files in the directory, with exactly one '%d' representing the frame number.
- `first, last`: inclusive range of frames to read, rel. filename numbering.

Returns

- `trajects`: a list of *`Trajectory`* objects, one for each trajectory contained in the mat file.

`flowtracks.io.`**`trajectories_mat`**(*fname*)
Extracts all trajectories from a Matlab file. the file is formated as a list of trajectory record arrays, containing attributes 'xf', 'yf', 'zf' for position, 'uf', 'vf', 'wf' for velocity, and 'axf', 'ayf', 'azf' for acceleration.

Arguments

- `fname`: path to the Matlab file.

Returns

- `trajects`: a list of *`Trajectory`* objects, one for each trajectory contained in the mat file.

`flowtracks.io.`**`trajectories_ptvis`**(*fname*, *first=None*, *last=None*, *frate=1.0*, *xuap=False*, *traj_min_len=None*)
Extract all trajectories in a directory of ptv_is files, as generated by programs in the 3d-ptv/pyptv family. supports xuap files as well.

Arguments

- `fname`: a template file name representing all ptv_is/xuap files in the directory, with exactly one '%d' representing the frame number. If no '%d' is found, the input is assumed to be in the Ron format - single file of concatenated ptv_is files, each stripped of the particle count line (first line) and separated from the next by an empty line.
- `first, last`: inclusive range of frames to read, rel. filename numbering.
- `frate`: frame rate, used for calculating velocities by backward derivative.
- `xuap`: The format is extended with colums for velocity and acceleration.
- `traj_min_len`: do not include trajectories shorter than this many frames.

Returns

each of the trajectories in the ptv_is/xuap data in order, as a *`Trajectory`* instance with velocity and acceleration.

`flowtracks.io.`**`trajectories_table`**(*fname*, *first=None*, *last=None*)
Reads trajectories from a PyTables HDF5 file, as saved by save_particles_table().

Arguments

- `fname`: path to file to read.

- `first, last`: inclusive range of frames to read.

Returns

- `trajects`: a list of Trajectory objects, each trimmed to the frame range.

CHAPTER 3

Particle class reference

This class is needed for modeling the dynamics of a particle in a flow scene.

class `flowtracks.particle.`**`Particle`**(*diameter*, *density*)

A class to hold particle properties.

Arguments

- `diameter`: particle diameter, [m]
- `density`: particle density, [kg/m^3]

CHAPTER 4

Interpolation module reference

Interpolation routines.

References

Documentation

class flowtracks.interpolation.**GeneralInterpolant**(*method*, *num_neighbs=None*, *radius=None*, *param=None*)

Holds all parameters necessary for performing an interpolation. Use is as a callable object after initialization, see *__call__()*.

Arguments

- `method`: interpolation method. Either 'inv' for inverse-distance weighting, 'rbf' for gaussian-kernel Radial Basis Function method, or 'corrfun' for using a correlation function.
- `radius`: of the search area for neighbours, [m]. If None, select closest `neighbs`.
- `neighbs`: number of closest neighbours to interpolate from. If None. uses 4 neighbours for 'inv' method, and 7 for 'rbf', unless `radius` is not None, then `neighbs` is ignored.
- `param`: the parameter adjusting the interpolation method. For IDW it is the inverse power (default 1), for rbf it is epsilon (default 1e5).

__call__(*tracer_pos*, *interp_points*, *data*, *companionship=None*)

Sets up the necessary parameters, and performs the interpolation. Does not change the scene set by set_scene if any, so may be used for any off-scene interpolation.

Arguments

- `tracer_pos`: (n,3) array, the x,y,z coordinates of one tracer per row, in [m]
- `interp_points`: (m,3) array, coordinates of points where interpolation will be done.
- `data`: (n,d) array, the for the d-dimensional data for tracer n. For example, in velocity interpolation this would be (n,3), each tracer having 3 components of velocity.
- `companionship`: an optional array denoting for each interpolation point the index of a tracer that should be excluded from it ("companion tracer"), useful esp. for analysing a simulated particle that started from a true tracer.

Returns

- `vel_interp`: an (m,3) array with the interpolated value at the position of each particle, [m/s].

current_relative_positions()
Returns an (m,k,3) array, the distance between interpolation point m and each of k nearest neighbours on each axis.

eulerian_jacobian(*local_interp=None*, *eps=0.0001*)
A general way to calculate the velocity derivatives. It could be enhanced in the future by specific analytical derivatives of the different interpolation methods. The Jacobian is calculated for the current scene, as recorded with `set_scene()`

Arguments

- `local_interp`: results of interpolation already performed at the position where derivatives are wanted. If not given, an interpolation of recorded scene data is automatically performed.
- `eps`: the dx in each direction.

Returns

interpolate(*subset=None*)
Performs an interpolation over the recorded scene.

Arguments

- `subset`: a neighbours selection array, such as returned from `which_neighbours()`, to replace the recorded selection. Default value (None) uses the recorded selection. The recorded selection is not changed, so `subset` is forgotten after the call.

Returns

an (m,3) array with the interpolated value at the position of each of m particles.

neighb_dists(*tracer_pos*, *interp_points*, *companionship=None*)
The distance from each interpolation point to each data point of those used for interpolation. Assumes, for now, a constant number of neighbours.

Arguments

- `tracer_pos`: (n,3) array, the x,y,z coordinates of one tracer per row, in [m]
- `interp_points`: (m,3) array, coordinates of points where interpolation will be done.
- `companionship`: an optional array denoting for each interpolation point the index of a tracer that should be excluded from it ("companion tracer"), useful esp. for analysing a simulated particle that started from a true tracer.

Returns

- `ndists`: an (m,c) array, for c closest neighbours as defined during object construction.

save_config(*cfg*)
Adds the keys necessary for recreating this interpolant into a configuration object. It is the caller's responsibility to do a writeback to file.

Arguments

- `cfg`: a ConfigParser object.

set_data_on_current_field(*data*)
Change the data on the existing interpolation points. This enables redoing an interpolation on a scene without recalculating weights, when weights are only dependent on position, as they are for most interpolation methods.

Arguments

- `data`: (n,d) array, the for the d-dimensional data for n tracers. For example, in velocity interpolation this would be (n,3), each tracer having 3 components of velocity.

set_field_positions(*positions*)

Sets the positions of points where there is data for interpolation. This sets up the structures for efficiently finding distances etc.

Arguments

- `positions`: (n,3) array, for position of n points in 3D space.

set_interp_points(*points*, *companionship=None*)

Sets the points into which interpolation will be done. It is possible to set this once and then do interpolation of several datasets into these points.

Arguments

- `positions`: (m,3) array, for position of m target points in 3D space.
- `companionship`: an optional array denoting for each interpolation point the index of a tracer that should be excluded from it ("companion tracer"), useful esp. for analysing a simulated particle that started from a true tracer.

set_scene(*tracer_pos*, *interp_points*, *data=None*, *companionship=None*)

Records scene data for future interpolation using the same scene.

Arguments

- `tracer_pos`: (n,3) array, the x,y,z coordinates of one tracer per row, in [m]
- `interp_points`: (m,3) array, coordinates of points where interpolation will be done.
- `data`: (n,d) array, the for the d-dimensional data for tracer n. For example, in velocity interpolation this would be (n,3), each tracer having 3 components of velocity.
- `companionship`: an optional array denoting for each interpolation point the index of a tracer that should be excluded from it ("companion tracer"), useful esp. for analysing a simulated particle that started from a true tracer.

trim_points(*which*)

Remove interpolation points from the scene.

Arguments

- `which`: a boolean array, length is number of current particle list (as given in set_scene), True to trim a point, False to keep.

`flowtracks.interpolation.`**Interpolant**(*method*, *num_neighbs=None*, *radius=None*, *param=None*)

Factory function. Returns an object of the interpolant class that matches the given method. All classes are subclassed from GeneralInterpolant.

Arguments

- `method`: interpolation method. Either 'inv' for inverse-distance weighting, 'rbf' for gaussian-kernel Radial Basis Function method, or 'corrfun' for using a correlation function.
- `radius`: of the search area for neighbours, [m]. If None, select closest `neighbs`.
- `neighbs`: number of closest neighbours to interpolate from. If None. uses 4 neighbours for 'inv' method, and 7 for 'rbf', unless `radius` is not None, then `neighbs` is ignored.
- `param`: the parameter adjusting the interpolation method. For IDW it is the inverse power (default 1), for rbf it is epsilon (default 1e5).

class flowtracks.interpolation.**InverseDistanceWeighter**(*num_neighbs=None*, *radius=None*, *param=None*)

Holds all parameters necessary for performing an inverse-distance interpolation [1]. Use is either as a callable object after initialization, see *__call__()*, or by setting a scene for repeated interpolation, see *set_scene()* and interpolate()

Arguments

- num_neighbs: number of closest neighbours to interpolate from. If None uses 4 neighbours, unless radius is not None, then neighbs is ignored.
- radius: of the search area for neighbours, [m]. If None, select closest neighbs.
- param: the inverse power of distance to use (default 1).

__call__(*tracer_pos*, *interp_points*, *data*, *companionship=None*)

Sets up the necessary parameters, and performs the interpolation. Does not change the scene set by set_scene if any, so may be used for any off-scene interpolation.

Arguments

- tracer_pos: (n,3) array, the x,y,z coordinates of one tracer per row, in [m]
- interp_points: (m,3) array, coordinates of points where interpolation will be done.
- data: (n,d) array, the for the d-dimensional data for tracer n. For example, in velocity interpolation this would be (n,3), each tracer having 3 components of velocity.
- companionship: an optional array denoting for each interpolation point the index of a tracer that should be excluded from it ("companion tracer"), useful esp. for analysing a simulated particle that started from a true tracer.

Returns

- vel_interp: an (m,3) array with the interpolated value at the position of each particle, [m/s].

eulerian_jacobian(*local_interp=None*, *eps=None*)

Velocity derivatives. The Jacobian is calculated for the current scene, as recorded with set_scene()

Arguments

- local_interp: results of interpolation already performed at the position where derivatives are wanted. If not given, an interpolation of recorded scene data is automatically performed.
- eps: unused, here for compatibility with base class.

Returns

(m,d,3) array, for m interpolation points and d interpolation dimentions. For each point, [i,j] = du_i/dx_j

set_scene(*tracer_pos*, *interp_points*, *data=None*, *companionship=None*)

Adds to the base class only a precalculation of weights.

trim_points(*which*)

Remove interpolation points from the scene.

Arguments

- which: a boolean array, length is number of current particle list (as given in set_scene), True to trim a point, False to keep.

weights(*dists*, *use_parts*)

Calculate the respective weight of each tracer j=1..n in the interpolation point i=1..m. The actual weight is normalized to the sum of weights in the interpolation, not here.

[1] http://en.wikipedia.org/wiki/Inverse_distance_weighting

Arguments

- `dists`: (m,n) array, the distance of interpolation_point i=1...m from tracer j=1...n, for (row,col) (i,j) [m] where n is the number of nearest neighbours.
- `use_parts`: (m,n) boolean array, whether tracer j is a neighbour of particle i, same indexing as `dists`.

Returns

- `weights`: an (m,n) array.

`flowtracks.interpolation.`**`corrfun_interp`**(*dists*, *use_parts*, *data*, *corrs_hist*, *corrs_bins*)

For each of n particle, generate the velocity interpolated to its position from all neighbours as selected by caller. The weighting of neighbours is by the correlation function, e.g. if the distance at neighbor i is r_i, then it adds $\rho(r_i) * v_i$ to the interpolated velocity. This is done for each component separately.

Arguemnts

- `dists`: (m,n) array, the distance of interpolation_point $i = 1...m$ from tracer $j = 1...n$, for (row,col) (i,j) [m]
- `use_parts`: (m,n) boolean array, whether tracer j is a neighbour of particle i, same indexing as `dists`.
- `data`: (n,d) array, the d components of the data that is interpolated from, for each of n tracers.
- `corrs_hist`: the correlation function histogram, an array of b bins.
- `corrs_bins`: same size array, the bin start point for each bin.

Returns

- `vel_avg`: an (m,3) array with the interpolated velocity at each interpolation point, [units of `data`].

`flowtracks.interpolation.`**`interpolant`**(*method*, *num_neighbs=None*, *radius=None*, *param=None*)

Factory function. Returns an object of the interpolant class that matches the given method. All classes are subclassed from GeneralInterpolant.

Arguments

- `method`: interpolation method. Either 'inv' for inverse-distance weighting, 'rbf' for gaussian-kernel Radial Basis Function method, or 'corrfun' for using a correlation function.
- `radius`: of the search area for neighbours, [m]. If None, select closest `neighbs`.
- `neighbs`: number of closest neighbours to interpolate from. If None. uses 4 neighbours for 'inv' method, and 7 for 'rbf', unless `radius` is not None, then `neighbs` is ignored.
- `param`: the parameter adjusting the interpolation method. For IDW it is the inverse power (default 1), for rbf it is epsilon (default 1e5).

`flowtracks.interpolation.`**`rbf_interp`**(*tracer_dists*, *dists*, *use_parts*, *data*, *epsilon=0.01*)

Radial-basis interpolation [3] for each particle, from all neighbours selected by caller. The difference from inv_dist_interp is that the weights are independent of interpolation point, among other differences.

Arguments

- `tracer_dists`: (n,n) array, the distance of tracer $i = 1...n$ from tracer $j = 1...n$, for (row,col) (i,j) [m]
- `dists`: (m,n) array, the distance from interpolation point $i = 1...m$ to tracer j. [m]
- `use_parts`: (m,n) boolean array, True where tracer $j = 1...n$ is a neighbour of interpolation point $i = 1...m$.
- `data`: (n,d) array, the d components of the data for each of n tracers.

Returns

- `vel_interp`: an (m,3) array with the interpolated velocity at the position of each particle, [m/s].

`flowtracks.interpolation.`**`read_interpolant`**(*conf_fname*)

Builds an Interpolant object based on values in an INI-formatted file.

Arguments

- `conf_fname`: path to configuration file.

Returns

an Interpolant object constructed from values in the configuration file.

`flowtracks.interpolation.`**`select_neighbs`**(*tracer_pos*, *interp_points*, *radius=None*, *num_neighbs=None*, *companionship=None*)

For each of m interpolation points, find its distance to all tracers. Use result to decide which tracers are the neighbours of each interpolation point, based on either a fixed radius or the closest num_neighbs.

Arguments

- `tracer_pos`: (n,3) array, the x,y,z coordinates of one tracer per row, [m]
- `interp_points`: (m,3) array, coordinates of points where interpolation will be done.
- `radius`: of the search area for neighbours, [m]. If None, select closest num_neighbs.
- `num_neighbs`: number of closest neighbours to interpolate from. If None. uses all neighbours in a given radius. `radius` has precedence.
- `companionship`: an optional array denoting for each interpolation point the index of a tracer that should be excluded from it ("companion tracer"), useful esp. for interpolating tracers unto themselves and for analysing a simulated particle that started from a true tracer.

Returns

- `dists`: (m,n) array, the distance from each interpolation point to each tracer.
- `use_parts`: (m,n) boolean array, True where tracer $j = 1...n$ is a neighbour of interpolation point $i = 1...m$.

CHAPTER 5

Reference for flowtracks.graphics

Various specialized graphing routines. The Probability Density Function graphing is best accessed by calling *pdf_graph()* on the raw data, but you can generate the PDF from the data separately (e.g. using *pdf_bins()*) and calling *generalized_histogram_disp()* on the result.

The other facility here is a function to plot a time-dependent 3D vector as 3 component subplots, which is another customary presentation in fluid dynamics circles. See *plot_vectors()*.

flowtracks.graphics.**generalized_histogram_disp**(*hist*, *bin_edges*, *log_bins=False*, *log_density=False*, *marker='o'*)

Draws a given histogram according to the visual custom of the fluid dynamics community.

Arguments

- `hist`: an array containing the number of values (or density) for each bin.
- `bin_edges`: the start value of each bin, same length as `hist`.
- `log_bins`: indicates that the bin edges are log-spaced.
- `log_densify`: Show the log of the probability density value. May cause problems if `log_bins` is True.
- `marker`: marker style for matplotlib.

Returns

the list of lines drawn, Matplotlib objects.

flowtracks.graphics.**pdf_bins**(*data*, *num_bins*, *log_bins=False*)

Generate a PDF of the given data possibly with logarithmic bins, ready for using in a histogram plot.

Arguments

- `data`: the samples to histogram.
- `bins`: the number of bins in the histogram.
- `log_bins`: if True, the bin edges are equally spaced on the log scale, otherwise they are linearly spaced (a normal histogram). If True, `data` should not contain zeros.

Returns

- `hist`: num_bins-lenght array of density values for each bin.
- `bin_edges`: array of size num_bins + 1 with the edges of the bins including the ending limit of the bins.

flowtracks.graphics.**pdf_graph**(*data*, *num_bins*, *log=False*, *log_density=False*, *marker='o'*)

Draw a PDF of the given data, according to the visual custom of the fluid dynamics community, and possibly with logarithmic bins.

Arguments

- `data`: the samples to histogram.
- `bins`: the number of bins in the histogram.
- `log`: if True, the bin edges are equally spaced on the log scale, otherwise they are linearly spaced (a normal histogram). If True, `data` should not contain zeros.
- `log_density`: Show the log of the probability density value. Only if log is False.
- `marker`: override the circle marker with any string acceptable to matplotlib.

`flowtracks.graphics.`**`plot_vectors`**(*vecs*, *indep*, *xlabel*, *fig=None*, *marker='-'*, *ytick_dens=None*, *yticks_format=None*, *unit_str=''*, *common_scale=None*, *arrows=None*, *arrow_color=None*)

Plot 3D vectors as 3 subplots sharing the same independent axis.

Arguments

- `vecs`: an (n,3) array, with n vectors to plot against the independent variable.
- `indep`: the corresponding n values of the independent variable.
- `xlabel`: label for the independent axis.
- `fig`: an optional figure object to use. If None, one will be created.
- `ytick_dens`: if not None, place this many yticks on each subplot, instead of the automatic tick marks.
- `yticks_format`: a pyplot formatter object.
- `unit_str`: a string to add to the Y labels representing the vector's units.
- `arrows`: an (n,3) array of values to represent as vertical arrows attached to each trajectory point.
- `arrow_color`: a matplotlib color spec for the arrow bodies.

Returns

- `fig`: the figure object used for plotting.

CHAPTER 6

Reference for the smoothing module

Trajectory smoothing routines. These are routines that are out of the Trajectory object because they precompute values that are dependent only on the smoothing method, and not on the trajectory itself, so they may be shared for processing a whole list of trajectories.

flowtracks.smoothing.**savitzky_golay**(*trajs*, *fps*, *window_size*, *order*)

Smooth (and optionally differentiate) data with a Savitzky-Golay filter. The Savitzky-Golay filter removes high frequency noise from data. It has the advantage of preserving the original shape and features of the signal better than other types of filtering approaches, such as moving averages techniques.

Parameters

- `trajs`: a list of Trajectory objects
- `window_size`: int, the length of the window. Must be an odd integer number.
- `fps`: frames per second, used for calculating velocity and acceleration.
- `order`: int, the order of the polynomial used in the filtering. Must be less then *window_size* - 1.

Returns

- `new_trajs`: a list of Trajectory objects representing the smoothed trajectories. Trajectories shorter than the window size are discarded.

Notes

The Savitzky-Golay is a type of low-pass filter, particularly suited for smoothing noisy data. The main idea behind this approach is to make for each point a least-square fit with a polynomial of high order over a odd-sized window centered at the point.

References

Data by Simplified Least Squares Procedures. Analytical Chemistry, 1964, 36 (8), pp 1627-1639.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery Cambridge University Press ISBN-13: 9780521880688

CHAPTER 7

Reference for flowtracks.pairs

Pair particles to closest tracers.

`flowtracks.pairs.`**`particle_pairs`**(*primary_trajects*, *secondary_trajects*, *trajids*, *time_points*)

For each of a set of select particles in the primary trajectories, find the closest particle in the secondary set.

Arguments

- `primary_trajects`: a list of Trajectory objects, some of which contain the source points.
- `secondary_trajects`: a list of Trajectory objects, in which to look for the pair points.
- `trajid, time_points`: each an n-length array for n pairs to produce, holding correspondingly the trajectory id and index into the trajectory of the points in the primary set to which a pair is sought.

Returns

- `pair_trid, pair_time`: coordinates of the found pairs, element i describes the pair of particle i in (trajid, time_points). Format is the same as that of `trajid, time_points`. For particles without a match, returns -1 as the pair_time value.

CHAPTER 8

Scene and Dual Scene Manipulation

A module for manipulating PTV analyses saved as HDF5 files in the flowtracks format. Allows reading the data by iterating over frames or over trajectories.

Main design goals:

1. Keep as little as possible in memory.
2. Minimize separate file accesses by allowing reading by frames instead of only by trajectories as in the old code.

class `flowtracks.scene.`**DualScene**(*tracers_path*, *particles_path*, *frate*, *particle*, *frame_range=None*)

Holds a scene orresponding to the dual-PTV systems, which shoot separate but coordinated streams for the tracers data and inertial particles data.

Arguments

- `tracers_path, particles_path`: respectively the path to the tracers and particles HDF files.
- `frate`: frame rate at which the scene was shot, [1/s].
- `particle`: a Particle object describing the inertial particles' diameter and density.
- `frame_range`: a uniform frame range to set to both of them. The default is None, meaning to use all frames (assuming equal-length data streams)

get_particles()

Returns the *Scene* that manages inertial particles' data.

get_particles_path()

Returns the path to the HDF file holding inertial particle data

get_range()

Returns the frame renge set for the dual scene.

iter_frames(*frame_range=-1*)

Iterates over a scene represented by two HDF files (one for inertial particles, one for tracers), and returns a Frame object whose two attributes (.tracers, .particles) hold a corresponding ParticleSnapshot object.

Arguments

- `frame_range`: tuple (first, last) sets the frame range of both scenes to an identical frame range. Argument format as in Scene.set_frame_range(). Default is (-1) meaning to skip this. Then the object's initialization range is used, so initialize to a coordinated range if you use the default.

Yields

the Frame object for each frame in turn.

iter_segments(*frame_range=-1*)
Like iter_frames, but returns two consecutive frames, both having the same trajids set (in other words, both contain only particles from the first frame whose trajectory continues to the next frame).

Arguments

- `frame_range`: tuple (first, last) sets the frame range of both scenes to an identical frame range. Argument format as in Scene.set_frame_range(). Default is (-1) meaning to skip this. Then the object's initialization range is used, so initialize to a coordinated range if you use the default.

Yields

two Frame objects, representing the consecutive selective frames.

class `flowtracks.scene.`**Scene**(*file_name*, *frame_range=None*)
This class is the programmer's interface to an HDF files containing particle trajectory data. It manages access by frames or trajectories, as well as by segments.

Arguments

- `file_name`: path to the HDF file hilding the data.
- `frame_range`: use only frames in this range for iterating the data. the default is None, meaning to use all present frams.

collect(*keys*, *where=None*)
Get values of given keys, either all of them or the ones corresponding to a selection given by 'where'.

Arguments

- `keys`: a list of keys to take from the data
- `where`: a dictionary of particle property names, with a tuple (min,max,invert) as values. If `invert` is false, the search range is between min and max. Otherwise it is anywhere except that.

Returns

a list of arrays, in the order of `keys`.

iter_frames()
Iterator over frames. Generates a ParticleSnapshot object for each frame, in the file, ordered by frame number, and yields it.

iter_segments()
Iterates over frames, taking out only the particles whose trajectory continues in the next frame.

Yields

- `frame`: a ParticleSnapshot object representing the current frame with the particles that have continuing trajectories.
- `next_frame`: same object, for the same particles in the next frame (the time attribute is obviously +1 from `frame`).

iter_trajectories()
Iterator over trajectories. Generates a Trajectory object for each trajectory in the file (in no particular order, but the same order every time on the same PyTables version) and yields it.

keys()
Return all the possible trajectory properties that may be queried as a data series (i.e. not the scalar property trajid), as a list of strings.

set_frame_range(*frame_range*)
Prepare a query part that limits the frame numbers is needed.

Arguments

- `frame_range`: a tuple (first, last) frame number, with the usual pythonic convention that first <= i < last. Any element may be None, in which case no limit is generated for it, and for no limits at all, passing none instead of a tuple is acceptable.

shapes()
Return the number of components per item of each key in the order returned by `keys()`.

trajectory_by_id(*trid*)
Get trajectory data by trajectory ID.

Arguments

- `trid`: trajectory ID, a unique number assigned to each trajectory when the scene file was written.

Returns

a Trajectory object.

trajectory_ids()
Returns all trajectory IDs in the scene as an array.

`flowtracks.scene.`**gen_query_string**(*key*, *range_spec*)
A small utility to create query string suitable for PyTables' `read_where()` from a range specification.

Arguments

- `key`: name of search field.
- `range_spec`: a tuple (min, max, invert). If `invert` is false, the search range is between min and max. Otherwise it is anywhere except that. In regular ranges, the max boundary is excluded as usual in Python. In inverted range, consequentlt, it is the min boundary that's excluded.

Returns

A string representing all boolean conditions necessary for representing the given range.

Example

```
>>> gen_query_string('example', (-1, 1, False))
'((example >= -1) & (example < 1))'
```

```
>>> gen_query_string('example', (-1, 1, True))
'((example < -1) | (example >= 1))'
```

`flowtracks.scene.`**read_dual_scene**(*conf_fname*)
Read dual-scene parameters, such as unchanging particle properties and frame range. Values are stored in an INI-format file.

Arguments

- `conf_fname`: name of the config file

Returns

a DualScene object initialized with the configuration values found.

CHAPTER 9

The Basic Analysis Machinery

Infrastructure for running a frame-by-frame analysis on a DualScene object. The main point of interest here is *analysis()*, which performs a segment iteration over a *DualScene* and applies to each a user-selected list of analyzers. Analysers are instances of a *GeneralAnalyser* subclass which implements the necessary methods, as described in the base class documentation.

There is one base class supplied here, *FluidVelocitiesAnalyser*, which ties in the *flowtracks.interpolation* module for analysing the fluid velocity around a particle from its surrounding tracers.

class flowtracks.analysis.**FluidVelocitiesAnalyser**(*interp*)

Finds, for each particle in the particles set of a frame, the so-called *undisturbed* fluid velocity at the particle's position, by interpolating from nearby particles in the tracers set.

Arguments

- interp: the Interpolant object to use for finding velocities.

analyse(*frame*, *next_frame*)

Arguments

- frame, next_frame: the Frame object for the currently-analysed frame and the one after it, respectively.

Returns

a list of two arrays, each of shape (f,3) where f is the number of particles in the current frame. 1st array - fluid velocity. 2nd array - relative velocity.

descr()

Return a list of two tuples, each of the form (name, data type, row length), describing the arrays returned by analyse() for fluid velocity and relative velocity.

class flowtracks.analysis.**GeneralAnalyser**

This is the parent class for all analysers to be used by *analysis()*. It does not do anything but define and document the methods that must be implenmented by the child class (in other words, this class is abstract). Attempting to use its methods will result in a NotImplementedError.

analyse(*frame*, *next_frame*)

Arguments

- frame, next_frame: the Frame object for the currently-analysed frame and the one after it, respectively.

Returns

a list of arrays, each of shape (f,d) where f is the number of particles in the current frame, and d is the row length of the corresponding item returned by self.descr(). Each array's dtype also corresponds to the dtype given to it by self.descr().

descr()
Need to return a list of tuples, each of the form (name, data type, row length), e.g. ('trajid', int, 1)

flowtracks.analysis.**analysis**(*scene*, *analysis_file*, *conf_file*, *analysers*, *frame_range=-1*)
Generate the analysis table for a given scene with separate data for inertial particles and tracers.

Arguments

- scene: a DualScene object representing an experiment with coordinated particles and tracers data streams.
- analysis_file: path to the file where analysis should be saved. If the file exists, it will be cloberred.
- conf_file: name of config file used for creating the analysis.
- analysers: a list of GeneralAnalyser subclasses that do the actual analysis work and know all that is needed about output shape.
- frame_range: if -1 no adjustment is necessary, otherwise see *DualScene.iter_segments()*

flowtracks.analysis.**companion_indices**(*trids*, *companions*)
Return an array giving for each companion its respective index in the trajectory ID array, or a negative number if not found.

CHAPTER 10

Handling of Analysis Results

class flowtracks.an_scene.**AnalysedScene**(*analysis_file*)

A class for accessing data and analyses of a scene analysed and saved in the format used by flowtracks.analysis.analyse().

Initializes the objects according to config and data-source metadata saved in the analysis file.

Arguments

- analysis_file: path to the HDF file containing analysis results.

collect(*keys*, *where=None*)

Get values of a given key, either some of them or the ones corresponding to a selection given by 'where'

Arguments

- keys: a list of keys to take from the data
- where: a dictionary of derived-results keys, with a tuple (min,max,invert) as values. If invert is false, the search range is between min and max. Otherwise it is anywhere except that.

Returns

a list of arrays, in the order of keys.

iter_trajectories()

Iterator over inertial trajectories. Since the analysis is structured around the inertial particles of the internal DualScene, it is possible to iterate those trajectories, adding the corresponding fields of analysis to the same object. Generates a Trajectory object for each inertial particle trajectory in the particles file (in no particular order, but the same order every time on the same PyTables version) and yields it.

Note

each trajectory.

keys()

Return names that may be used to access data in any of the data sources available, whether analyses or inertial particles.

shapes()

Return the number of components per item of each key in the order returned by *keys()*.

trajectory_by_id(*trid*)

Retrieves an inertial trajectory with the respective analysis. See iter_trajectories for the full documentation.

Arguments

- trid: trajectory ID to fetch.

Returns

a Trajectory object with analysis keys added.

CHAPTER 11

Sequence Processing of non-HDF Formatted Particle Databases

class `flowtracks.sequence.`**Sequence**(*frange*, *frate*, *particle*, *part_tmpl*, *tracer_tmpl*, *smooth_tracers=False*, *traj_min_len=0.0*)

Tracks a dual particles database (for both inertial particles and tracers), allowing a number of underlying formats. Provides segment iteration and trajectory-mapping.

Arguments

- `frange`: tuple, (first frame #, after last frame #)
- `frate`: the frame rate at which the scene was shot.
- `particle`: a Particle object representing the suspended particles' properties.
- `part_tmpl, tracer_tmpl`: the filenames for particle- and tracer- databases respectively. Names must be as understood by :func:'flowtracks.io.trajectories'.
- `smooth_tracers`: if True, uses trajectory smoothing on the tracer trajectories when iterating over frames. Possibly out of date.
- `traj_min_len`: when reading trajectories (tracers and particles) discard trajectories shorter than this many frames.

__iter__()

Iterate over frames. For each frame return the data for the tracers and particles in it, as a tuple containing two *ParticleSnapshot* objects corresponding to the current frame data and the next frame's.

Returns

A Python iteraor.

iter_subrange(*first*, *last*)

The same as *__iter__()*, except it changes the frame range for the duration of the iteration.

Arguments

- `first, last`: frame numbers of the first and last frames in the acting range of frames from the sequence.

Returns

A Python iteraor.

map_trajectories(*func*, *subrange=None*, *history=False*, *args=()*)

Iterate over frames, for each frame call a function that generates a per-trajectory result and add the results up in a per-trajectory time-series.

Arguments

- `func`: the function to call. Returns a dictionary keyed by trajid. receives as arguments (self, particles, tracers) where particles, tracers are the sequence iteration results as given by __iter__.
- `subrange`: tuple (first, last). Iterate over a subrange of the sequence delimited by these frame numbers.
- `history`: true if the result of one frame depends on earlier results. If true, func receives a 4th argument, the accumulated results so far as a dictionary of time-series lists.
- `args`: a tuple of extra positional arguments to pass to the function after the usual arguments and the possible history argument.

Returns

a dictionary keyed by trajid, where for each trajectory a time series of results obtained during the trajectory's lifetime is the value.

part_fname()
Returns the file name used for reading inertial particles database.

part_format()
Returns the format inferred for the inertial particles database.

particle_trajectories()
Return (and possibly generate and cache) the list of *Trajectory* objects as selected by the particle selector.

range()
Returns the frame number range set for the object, as a tuple (first, last).

save_config(*cfg*)
Adds the keys necessary for recreating this sequence into a configuration object. It is the caller's responsibility to do a writeback to file.

Arguments

- `cfg`: a ConfigParser object.

set_particle_selector(*selector*)
Sets a filter on the particle trajectories used in sequencing.

Arguments

- `selector`: a function which receives a list of *Trajectory* objects and returns a sublist thereof.

set_tracer_selector(*selector*)
Sets a filter on the tracer trajectories used in sequencing.

Arguments

- `selector`: a function which receives a list of *Trajectory* objects and returns a sublist thereof.

subrange()
Returns the earliest and latest time points covered by the subset of trajectories that the particle selector selects, bounded by the range restricting the overall sequence.

tracer_trajectories()
Return (and possibly generate and cache) the list of *Trajectory* objects corresponding to tracers.

`flowtracks.sequence.`**read_sequence**(*conf_fname*, *smooth=None*, *traj_min_len=None*)
Read sequence-wide parameters, such as unchanging particle properties and frame range. Values are stored in an INI-format file.

Arguments

- `conf_fname`: name of the config file
- `smooth`: whether the sequence shoud use tracers trajectory-smoothing. Used to override the config value if present, and supply it if missing. If None and missing, default is False.
- `traj_min_len`: tells the sequence to ignore trajectories shorter than this many frames. Overrides file. If None and file has no value, default is 0.

Returns

a Sequence object initialized with the configuration values found.

CHAPTER 12

Getting Started

Obtaining the package and its dependencies

The most recent version of this package may be found under the auspices of the OpenPTV project, in its Github repository,

https://github.com/OpenPTV/postptv

Dependencies:

- The software depends on the SciPy package, obtainable from http://www.scipy.org/
- Some features depend on the Matplotlib package. Users which need those features may get Matplotlib at http://matplotlib.org/

Installation

To install this package, follow the standard procedure for installing Python modules. Using a terminal, change directory into the root directory of theis program's source code, then run

```
python setup.py install
```

Note that you may need administrative privileges on the machine you are using.

The install script will install the Python package in the default place for your platform. Additionally, it will install example scripts in a subdirectory `flowtracks-examples/` under the default executable location, and a the documentation in the default package root. For information on where these directories are on your platform (and how to change them), refer to the Python documentation. Other standard features of the setup script are also described therein.

Documentation

This documentation is available in the source directory under the `docs/` subdirectory. It is maintained as a Sphinx project, so that you can build the documentation in one of several output formats, including HTML and PDF. To build, install Sphinx, then use

or replace `html` with any other builder supported by Sphinx.

Alternatively, the documentation is pre-built and available online on ReadTheDocs.

Examples

The `examples/` subdirectory in the source distribution contains two IPython notebooks, both available as HTML for direct viewing:

- a tutorial to the basic HDF5 analysis workflow (`HTML`).
- a demonstration of using the `flowtracks.interpolation` module (`HTML`).

Analysis Script

The script `analyse_fhdf.py` is installed by default. for instruction on its usage, run:

```
analyse_fhdf.py --help
```

As the help message printed informs, there are two mandatory command-line arguments. One is the data file for processing, the other is a config file with some rudimentary metadata. Examples for both are supplied in the `data/` subdirectory of this package. A config file accepted by the script looks something like this:

```
[Particle]
density = 1450
diameter = 500e-6

[Scene]
particles file = particles.h5
tracers file = tracers.h5
first frame = 10001
last frame = 10200
frame rate = 500
```

the file above may be used for producing an analysis from the files in the `data/` subdirectory, when it is the current directory. this has been done in both IPython examples mentioned above, where the usage is shown.

CHAPTER 13

General Facilities

Data structures

the most basic building blocks of any analysis are sets of particles representing a slice of the database. These are represented by a ParticleSet instance. ParticleSet is a flexible class. It holds a number of numpy arrays whose first dimension must have the same length; each is a column in a table of particle properties, whose each row represents one particle's data. It must contain particles' position and velocity data, but users may add more properties as relevant to their database. For details, ssee the ParticleSet documentation.

The two most common ways to slice a database are by frame (time point) and by trajectory (data fore the same particle over several frames). For this there are two classes provided by `flowtracks`, both derived from ParticleSet and thus behave in a similar way. They both expect the `time` and `trajid` (trajtectory ID) properties to exist for the particle data, but each class treats these properties differently.

ParticleSnapshot is a ParticleSet which assumes that all particles have the same `time`, so that this property is scalar. Similarly, the Trajectory class expects a same `trajid` across its data. A trajectory ID is simply an integer number unique to each trajectory. Users may select their numbering scheme when creating Trajectory objects from scratch, but in most cases the data is read from a file, in which case Flowtracks' input rutines handle the numbering automatically.

Refer to Modules Containing Flowtracks Basic Data Structures for the details of all these classes.

Input and Output

The module *`flowtracks.io`* provides several functions for reading and writing particle data. The currently-supported formats are:

- ptv_is - the format output by OpenPTV code as well as the older but still widely used 3DPTV. Composed of one file per frame, containing a particle's number, its number in the previous and next frame file, and current position.
- xuap - a similar format using one file per frame with columns for position, velocity, and acceleration for both the particle and the surrounding fluid. This file format represents an initial analysis of ptv_is raw data.
- acc - another frame-files format with each particle having, additionally to data provided in the xuap format, the time step relative to the trajectory start.
- mat - a Matlab file containing a list of trajectory structure-arrays with xuap-like data for each trajectory.
- hdf - Flowtracks' own optimised format, relying on the HDF5 file format and the PyTables package for fast handling thereof. It is highly recommended to use the other reading/writing functions in order to convert data in other formats to this format. This allows users a more flexible programmatic interface as well as the speed of PyTables.

Description of the relevant functions, as well as some other IO convenience facilities may be found in the module's reference documentation.

Basic Analysis and display

The package provides some facilities for analysing the database and extracting kinematic or dynamic information embedded in it. Dynamic analysis requires the particle size and diameter to be known (Flowtracks assumes a spherical particle for these analyses, but users may extend this behaviour). These properties may be stored in the Particle class provided by the package. *flowtracks.io* provides a way to read them from an INI file.

The *flowtracks.interpolation* module provides an object-oriented approach to interpolating the data. It offers some built-in interpolation methods, and is hoped to be extensible to other methods without much effort.

Some plotting support is provided by *flowtracks.graphics*. Functions therein allow users to generate probability distributions from data and to plot them using Matplotlib, and a function is provided that plots 3D vector data as 3 subplots of components.

Other facilities (*smoothing*, *nearest-neighbour searches*) are described in the respective module's documentation.

CHAPTER 14

HDF5-based fast databases

Above the layer of basic data structures, Flowtracks provides a generalized view of a scene, containing several trajectories across a number of frames. This view is iterable in several ways and provides general metadata access.

The `Scene` class is the most basic form of this view. It is tied to one HDF5 file exactly, which holds the database. This file may be iterated by trajectory, by frame, or by *segments*, a concept introduced by Flowtracks for easier time-derived analyses requiring the next time-point to be also known.

A segment, in the context of iterating a `Scene` is a tuple containing two `ParticleSnapshot()` objects, one for the current frame and one for the next. The next frame data is filtered to contain only particles that also appear in the current frame, unlike when iterating simply by frames.

The `DualScene` class extends this by tying itself into two HDF5 files, each representing a separate class of particles which coexist in the same experiment. This has been useful for measuring tracers and inertial particles simultaneously, but other users are of course possible. Iterating by frames is supported here, providing a `Frame` object on each iteration. Iterating by trajectories is ambiguous and not supported currently. Segments iteration, similarly to the frames iteration, returns two `Frame` objects.

The `flowtracks.analysis` module provides a function for applying analyser classes sequentially to segments iterated over, and generetes a properly sized HDF5 file in the format of the input file.

`AnalysedScene` objects track simultaneously the DualScene and an analysis file resulting from it. They contain the `collect()` facility. It allows finding of all (or selected) data belonging to a certain property, regardless of which of the files it is stored in.

CHAPTER 15

Manipulating text formats directly

Similarly to the *DualScene* class used with the HDF5 format, the *Sequence* class tracks two sets of particles and allows iterating by frame. Since this class relies on *Trajectory* lists as its underlying database, it does not provide a special facility for iterating over trajectories.

Though *Sequence* also accepts trajectory iterators, and *flowtracks.io* provides you with iterators if asked, the working memory used in actuality may still be large and the access times are much slower than the equivalent times achieved by the specialized HDF5 classes.

Corresponding to the *flowtracks.analysis* module, *Sequence* provides the *map_trajectories()* method for applying callback functions on an entire scene, frame by frame.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

f

Symbols

A

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

W