PEPT Documentation

Release 0.5.2

PEPT maintainers

Apr 24, 2023

DOCUMENTATION

1 Positron Emission Particle Tracking 3

2 Tutorials and Documentation 5

3 Performance 7

4 Copyright 9

5 Indices and tables 11
- 5.1 Getting Started 11
- 5.2 Tutorials 11
- 5.3 Manual 29
- 5.4 Contributing 236
- 5.5 Citing 237

Bibliography 239

Python Module Index 241

Index 243

A Python library that unifies Positron Emission Particle Tracking (PEPT) research, including tracking, simulation, data analysis and visualisation tools.

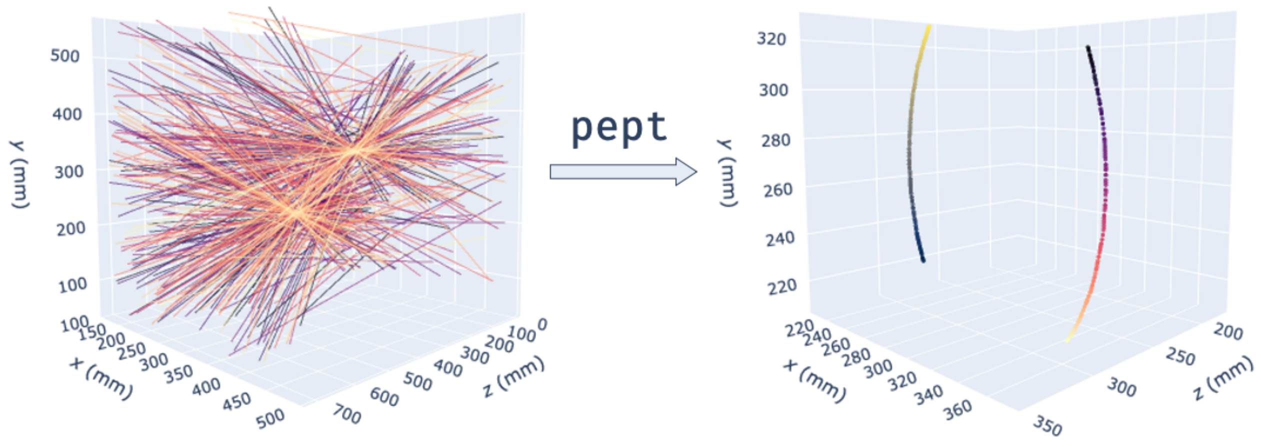
CHAPTER

ONE

POSITRON EMISSION PARTICLE TRACKING

PEPT is a technique developed at the University of Birmingham which allows the non-invasive, three-dimensional tracking of one or more ‘tracer’ particles through particulate, fluid or multiphase systems. The technique allows particle or fluid motion to be tracked with sub-millimetre accuracy and sub-millisecond temporal resolution and, due to its use of highly-penetrating 511keV gamma rays, can be used to probe the internal dynamics of even large, dense, optically opaque systems - making it ideal for industrial as well as scientific applications.

PEPT is performed by radioactively labelling a particle with a positron- emitting radioisotope such as fluorine-18 (18F) or gallium-68 (68Ga), and using the back-to-back gamma rays produced by electron-positron annihilation events in and around the tracer to triangulate its spatial position. Each detected gamma ray represents a line of response (LoR).



Transforming gamma rays, or lines of response (left) into individual tracer trajectories (right) using the *pept* library. Depicted is experimental data of two tracers rotating at 42 RPM, imaged using the University of Birmingham Positron Imaging Centre’s parallel screens PEPT camera.

CHAPTER

TWO

TUTORIALS AND DOCUMENTATION

A very fast-paced introduction to Python is available here (Google Colab tutorial link); it is aimed at engineers whose background might be a few lines written MATLAB, as well as moderate C/C++ programmers.

A beginner-friendly tutorial for using the *pept* package is available here (Google Colab link).

The links above point to Google Colaboratory, a Jupyter notebook-hosting website that lets you combine text with Python code, executing it on Google servers. Pretty neat, isn't it?

CHAPTER THREE

PERFORMANCE

Significant effort has been put into making the algorithms in this package as fast as possible. Most computationally intensive code has been implemented in *Cython*, *C* or *C++* and allows policy-based parallel execution, either on shared-memory machines using *joblib* / *ThreadPoolExecutor*, or on distributed computing clusters using *mpi4py.futures.MPIPoolExecutor*.

CHAPTER FOUR

COPYRIGHT

CHAPTER

FIVE

INDICES AND TABLES

5.1 Getting Started

These instructions will help you get started with PEPT data analysis.

5.1.1 Prerequisites

This package supports Python 3.6 and above - it is built and tested for Python 3.6, 3.7 and 3.8 on Windows, Linux and macOS (thanks to conda-forge, which is awesome!).

You can install it using the batteries-included Anaconda distribution or the bare-bones Python interpreter. You can also check out our Python and *pept* tutorials.

5.1.2 Installation

The easiest and quickest installation, if you are using Anaconda:

```
conda install -c conda-forge pept
```

You can also install the latest release version of *pept* from PyPI:

```
pip install --upgrade pept
```

Or you can install the development version from the GitHub repository:

```
pip install -U git+https://github.com/uob-positron-imaging-centre/pept
```

5.2 Tutorials

The main purpose of the PEPT library is to provide a common, consistent foundation for PEPT-related algorithms, including tracer tracking, visualisation and post-processing tools - such that they can be used interchangeably, mixed and matched for any PEPT camera and system. Virtually all PEPT processing routine follows these steps:

1. Convert raw gamma camera / scanner data into 3D lines (i.e. the captured gamma rays, or lines of response - LoRs).
2. Take a sample of lines, locate tracer locations, then repeat for the next samples.
3. Separate out individual tracer trajectories.

4. Visualise and post-process trajectories.

For these algorithm-agnostic steps, PEPT provides five base data structures upon which the rest of the library is built:

1. `pept.LineData`: general 3D line samples, formatted as *[time, x1, y1, z1, x2, y2, z2, extra…]*.
2. `pept.PointData`: general 3D point samples, formatted as *[time, x, y, z, extra…]*.
3. `pept.Pixels`: single 2D pixellised space with physical dimensions, including fast line traversal.
4. `pept.Voxels`: single 3D voxellised space with physical dimensions, including fast line traversal.

For example, once you convert your PEPT data - from any scanner - into `pept.LineData`, all the algorithms in this library can be used.

All the data structures above are built on top of NumPy and integrate natively with the rest of the Python / SciPy ecosystem. The rest of the PEPT library is organised into submodules:

1. `pept.scanners`: converters between native scanner data and the base data structures.
2. `pept.tracking`: radioactive tracer tracking algorithms, e.g. the Birmingham method, PEPT-ML, FPI.
3. `pept.plots`: PEPT data visualisation subroutines.
4. `pept.utilities`: general-purpose helpers, e.g. `read_csv`, `traverse3d`.
5. `pept.processing`: PEPT-oriented post-processing algorithms, e.g. `VectorField3D`.

If you are new to the PEPT library, we recommend going through this interactive online notebook, which introduces all the fundamental concepts of the library:

> https://colab.research.google.com/drive/1G8XHP9zWMMDVu23PXzANLCOKNP_RjBEO?usp=sharing

Once you get the idea of `LineData` samples, `Pipeline` and `PlotlyGrapher`, you can use these copy-pastable tutorials to build PEPT data analysis pipelines tailored to your specific systems.

5.2.1 Absolute Basics

The main purpose of the `pept` library is to provide a common, consistent foundation for PEPT-related algorithms, including tracer tracking, visualisation and post-processing tools - such that they can be used interchangeably, mixed and matched for different systems. Virtually *any* PEPT processing routine follows these steps:

1. Convert raw gamma camera / scanner data into *3D lines* (i.e. the captured gamma rays, or lines of response - LoRs).
2. Take a *sample* of lines, locate tracer locations, then repeat for the next samples.
3. Separate out individual tracer trajectories.
4. Visualise and post-process trajectories.

For these algorithm-agnostic steps, `pept` provides five base data structures upon which the rest of the library is built:

1. `pept.LineData`: general 3D line samples, formatted as *[time, x1, y1, z1, x2, y2, z2, extra…]*.
2. `pept.PointData`: general 3D point samples, formatted as *[time, x, y, z, extra…]*.
3. `pept.Pixels`: single 2D pixellised space with physical dimensions, including fast line traversal.
4. `pept.Voxels`: single 3D voxellised space with physical dimensions, including fast line traversal.

All the data structures above are built on top of NumPy and integrate natively with the rest of the Python / SciPy ecosystem. The rest of the `pept` library is organised into submodules:

- pept.scanners: converters between native scanner data and the base classes.
- pept.tracking: radioactive tracer tracking algorithms, e.g. the Birmingham method, PEPT-ML, FPI.
- pept.plots: PEPT data visualisation subroutines.
- pept.utilities: general-purpose helpers, e.g. read_csv, traverse3d.
- pept.processing: PEPT-oriented post-processing algorithms, e.g. occupancy2d.

pept.LineData

Generally, PEPT Lines of Response (LoRs) are lines in 3D space, each defined by two points, regardless of the geometry of the scanner used. This class is used to wrap LoRs (or any lines!), efficiently yielding samples of lines of an adaptive sample_size and overlap.

It is an abstraction over PET / PEPT scanner geometries and data formats, as once the raw LoRs (be they stored as binary, ASCII, etc.) are transformed into the common LineData format, any tracking, analysis or visualisation algorithm in the pept package can be used interchangeably. Moreover, it provides a stable, user-friendly interface for iterating over LoRs in *samples* - this is useful for tracking algorithms, as they generally take a few LoRs (a *sample*), produce a tracer position, then move to the next sample of LoRs, repeating the procedure. Using overlapping samples is also useful for improving the tracking rate of the algorithms.

Here are some basic examples of creating and using LineData samples - you're very much invited to copy and run them!

Initialise a LineData instance containing 10 lines with a sample_size of 3.

```
>>> import pept
>>> import numpy as np
>>> lines_raw = np.arange(70).reshape(10, 7)
>>> print(lines_raw)
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]
 [35 36 37 38 39 40 41]
 [42 43 44 45 46 47 48]
 [49 50 51 52 53 54 55]
 [56 57 58 59 60 61 62]
 [63 64 65 66 67 68 69]]

>>> line_data = pept.LineData(lines_raw, sample_size = 3)
>>> line_data
pept.LineData (samples: 3)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 10, columns: 7)
  [[ 0.  1. ...  5.  6.]
   [ 7.  8. ... 12. 13.]
   ...
   [56. 57. ... 61. 62.]
   [63. 64. ... 68. 69.]]
```

(continues on next page)

(continued from previous page)

```
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

Access samples using subscript notation. Notice how the samples are consecutive, as `overlap` is 0 by default.

```
>>> line_data[0]
pept.LineData (samples: 1)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 3, columns: 7)
  [[ 0.  1. ...  5.  6.]
   [ 7.  8. ... 12. 13.]
   [14. 15. ... 19. 20.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}

>>> line_data[1]
pept.LineData (samples: 1)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 3, columns: 7)
  [[21. 22. ... 26. 27.]
   [28. 29. ... 33. 34.]
   [35. 36. ... 40. 41.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

Now set an overlap of 2; notice how the number of samples changes:

```
>>> len(line_data)     # Number of samples
3

>>> line_data.overlap = 2
>>> len(line_data)
8
```

5.2.2 Saving / Loading Data

All PEPT objects can be saved in an efficient binary format using `pept.save` and `pept.load`:

```
import pept
import numpy as np

# Create some dummy data
lines_raw = np.arange(70).reshape((10, 7)
lines = pept.LineData(lines_raw)
```

(continues on next page)

(continued from previous page)

```
# Save data
pept.save("data.pickle", lines)

# Load data
lines_loaded = pept.load("data.pickle")
```

The binary approach has the advantage of preserving all your metadata saved in the object instances - e.g. `columns`, `sample_size` - allowing the full state to be reloaded.

Matrix-like data like `pept.LineData` and `pept.PointData` can also be saved in a slower, but human-readable CSV format using their class methods `.to_csv`; such tabular data can then be reinitialised using `pept.read_csv`:

```
# Save data in CSV format
lines.to_csv("data.csv")

# Load data back - *this will be a simple NumPy array!*
lines_raw = pept.read_csv("data.csv")

# Need to put the array back into a `pept.LineData`
lines = pept.LineData(lines_raw)
```

5.2.3 Plotting

Interactive 3D Plots

The easiest method of plotting 3D PEPT-like data is using the `pept.plots.PlotlyGrapher` interactive grapher:

```
# Plotting some example 3D lines
import pept
from pept.plots import PlotlyGrapher
import numpy as np

lines_raw = np.arange(70).reshape((10, 7))
lines = pept.LineData(lines_raw)

PlotlyGrapher().add_lines(lines).show()
```

```
# Plotting some example 3D points
import pept
from pept.plots import PlotlyGrapher
import numpy as np

points_raw = np.arange(40).reshape((10, 4))
points = pept.PointData(points_raw)

PlotlyGrapher().add_points(points).show()
```

The `PlotlyGrapher` object allows straightforward subplots creation:

```
# Plot the example 3D lines and points on separate subplots
grapher = PlotlyGrapher(cols = 2)
```

(continues on next page)

(continued from previous page)

```
grapher.add_lines(lines)                        # col = 1 by default
grapher.add_points(points, col = 2)

grapher.show()
```

```
# Plot the example 3D lines and points on separate subplots
grapher = PlotlyGrapher(rows = 2, cols = 2)

grapher.add_lines(lines, col = 2)               # row = 1 by default
grapher.add_points(points, row = 2, col = 2)

grapher.show()
```

Adding Colourbars

By default, the last column of a dataset is used to colour-code the resulting points:

```
from pept.plots import PlotlyGrapher
PlotlyGrapher().add_points(point_data).show()   # Colour-codes by the last column
```

You can change the column used to colour-code points using a numeric index (e.g. first column `colorbar_col = 0`, second to last column `colorbar_col = -2`) or named column (e.g. `colorbar_col = "error"`):

```
PlotlyGrapher().add_points(point_data, colorbar_col = -2).show()
PlotlyGrapher().add_points(point_data, colorbar_col = "label").show()   # Coloured by␣
↪trajectory
PlotlyGrapher().add_points(point_data, colorbar_col = "v").show()       # Coloured by␣
↪velocity
```

As a `PlotlyGrapher` will often manage multiple subplots, one shouldn't include explicit colourbars on the sides *for each dataset plotted*. Therefore, colourbars are hidden by default; add a colourbar by setting its title:

```
PlotlyGrapher().add_points(points, colorbar_title = "Velocity").show()
```

Histogram of Tracking Errors

The `Centroids(error = True)` filter appends a column "error" representing the relative error in the tracked position. You can select a named column via indexing, e.g. `trajectories["error"]`; you can then plot a histogram of the relative errors with:

```
import plotly.express as px
px.histogram(trajectories["error"]).show()          # Large values are noise
px.histogram(trajectories["cluster_size"]).show()   # Small values are noise
```

It is often useful to remove points with an error higher than a certain value, e.g. 20 mm:

```
trajectories = Condition("error < 20").fit(trajectories)

# Or simply append the `Condition` to the `pept.Pipeline`
```

(continues on next page)

(continued from previous page)

```
pipeline = pept.Pipeline([
    ...
    Condition("cluster_size > 30, error < 20"),
    ...
])
```

Exporting Plotly Graphs as Images

The standard output of the Plotly grapher is an interactive HTML webpage; however, this can lead to large file sizes or memory overflows. Plotly allows for graphs to be exported as images to alleviate some of these issues.

Ensure you have imported:

```
import plotly.express as px
import kaleido
import plotly.io as pio
```

There are two main ways of exporting as images:

```
# Save the inner plotly.Figure attribute of a `grapher`
# Format can be changed to other image formats
# Width and height can be adjusted to give the desired image size
grapher.fig.write_image("figure.png", width=2560, height=1440)
```

Modifying the Underlying Figure

You can access the Plotly figure wrapped and managed by a PlotlyGrapher using the `.fig` attribute:

```
grapher.fig.update_layout(xaxis_title = "Pipe Length (mm)")
```

5.2.4 Initialising PEPT Scanner Data

The `pept.scanners` submodule contains converters between scanner specific data formats (e.g. parallel screens / ASCII, modular camera / binary) and the `pept` base classes, allowing simple initialisation of `pept.LineData` from different sources.

ADAC Forte

The parallel screens detector used at Birmingham can output binary *list-mode* data, which can be converted using `pept.scanners.adac_forte(binary_file)`:

```
import pept

lines = pept.scanners.adac_forte("binary_file.da01")
```

If you have multiple files from the same experiment, e.g. “data.da01”, “data.da02”, etc., you can stitch them all together using a *glob*, “data.da*”:

```
import pept

# Multiple files starting with `binary_file.da`
lines = pept.scanners.adac_forte("binary_file.da*")
```

Parallel Screens

If you have your data as a CSV containing 5 columns *[t, x1, y1, x2, y2]* representing the coordinates of the two points defining an LoR on two parallel screens, you can use `pept.scanners.parallel_screens` to insert the missing coordinates and get the LoRs into the general `LineData` format *[t, x1, y1, z1, x2, y2, z2]*:

```
import pept

screen_separation = 500
lines = pept.scanners.parallel_screens(csv_or_array, screen_separation)
```

Modular Camera

Your modular camera data can be initialised using `pept.scanners.modular_camera`:

```
import pept

lines = pept.scanners.modular_camera(filepath)
```

5.2.5 Adaptive Sampling

Perhaps the most important decision a PEPT user must make is how the LoRs are divided into samples. The two most common approaches are:

Fixed sample size: a constant number of elements per sample, with potential overlap between samples.

- Advantages: effectively adapts spatio-temporal resolution, with higher accuracy in more active PEPT scanner regions.
- Disadvantages: when a tracer exits the field of view, the last LoRs will be joined with the first LoRs when the tracer re-enters the scanner in the same samples.

Fixed time window: a constant time interval in which LoRs are aggregated, with potential overlap.

- Advantages: robust to tracers moving out of the field of view.
- Disadvantages: non-adaptive temporal resolution.

The two approaches can be combined into a single `pept.AdaptiveWindow`, which works as a fixed time window, except when more LoRs are encountered than a given limit, in which case the time window is shrunk - hence adapting the time window depending on how many LoRs are intercepted in a given window.

```
import pept

# A time window of 5 ms shrinking when encountering more than 200 LoRs
lors = pept.LineData(..., sample_size = pept.AdaptiveWindow(5.0, 200))

# A time window of 12 ms with the number of LoRs capped at 400 LoRs and an overlap of 6␣
```

(continues on next page)

(continued from previous page)

```
↪ms
lors = pept.scanners.adac_forte(
    ...,
    sample_size = pept.AdaptiveWindow(12., 200),
    overlap = pept.AdaptiveWindow(6.),
)
```

Moreover, if an ideal number of LoRs is selected, there exists an optimum time window for which most samples will have roughly this ideal number of LoRs, except when the tracer is out of the field of view, or it's static. This can be automatically selected using `pept.tracking.OptimizeWindow`:

```
import pept
import pept.tracking as pt

# Find an adaptive time window that is ideal for about 200 LoRs per sample
lors = pept.LineData(...)
lors = pt.OptimizeWindow(ideal_elems = 200).fit(lors)
```

OptimizeWindow can be used at the start of a pipeline; an optional *overlap* parameter can be used to define an overlap as a ratio to the ideal time window found. For example, if the ideal time window found is 100 ms, an overlap of 0.5 will result in an overlapping time interval of 50 ms:

```
import pept
from pept.tracking import *

pipeline = pept.Pipeline([
    OptimizeWindow(200),
    BirminghamMethod(fopt = 0.5),
    Stack(),
])

locations = pipeline.fit(lors)
```

5.2.6 The Birmingham Method

The Birmingham Method is an efficient, analytical technique for tracking tracers using the LoRs from PEPT data.

If you are using it in your research, you are kindly asked to cite the following paper:

> *Parker DJ, Broadbent CJ, Fowles P, Hawkesworth MR, McNeil P. Positron emission particle tracking-a technique for studying flow within engineering equipment. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 1993 Mar 10;326(3):592-607.*

Birmingham Method recipe

```
import pept
from pept.tracking import *

pipeline = pept.Pipeline([
    BirminghamMethod(fopt = 0.5),
    Stack(),
])

locations = pipeline.fit(lors)
```

Recipe with Trajectory Separation

```
import pept
from pept.tracking import *

pipeline = pept.Pipeline([
    BirminghamMethod(fopt = 0.5),
    Segregate(window = 20, cut_distance = 10),
    Stack(),
])

locations = pipeline.fit(lors)
```

5.2.7 PEPT-ML

PEPT using Machine Learning is a modern clustering-based tracking method that was developed specifically for noisy, fast applications.

If you are using PEPT-ML in your research, you are kindly asked to cite the following paper:

> *Nicuşan AL, Windows-Yule CR. Positron emission particle tracking using machine learning. Review of Scientific Instruments. 2020 Jan 1;91(1):013329.*

PEPT-ML one pass of clustering recipe

The LoRs are first converted into `Cutpoints`, which are then assigned cluster labels using `HDBSCAN`; the cutpoints are then grouped into clusters using `SplitLabels` and the clusters' `Centroids` are taken as the particle locations. Finally, stack all centroids into a single `PointData`.

```
import pept
from pept.tracking import *

max_tracers = 1

pipeline = pept.Pipeline([
    Cutpoints(max_distance = 0.5),
    HDBSCAN(true_fraction = 0.15, max_tracers = max_tracers),
    SplitLabels() + Centroids(error = True),
```

(continues on next page)

(continued from previous page)

```
    Stack(),
])

locations = pipeline.fit(lors)
```

PEPT-ML second pass of clustering recipe

The particle locations will always have a bit of *scatter* to them; we can *tighten* those points into accurate, dense trajectories using a *second pass of clustering*.

Set a very small sample size and maximum overlap to minimise temporal smoothing effects, then recluster the tracer locations, split according to cluster label, compute centroids, and stack into a final `PointData`.

```
import pept
from pept.tracking import *

max_tracers = 1

pipeline = pept.Pipeline([
    Stack(sample_size = 30 * max_tracers, overlap = 30 * max_tracers - 1),
    HDBSCAN(true_fraction = 0.6, max_tracers = max_tracers),
    SplitLabels() + Centroids(error = True),
    Stack(),
])

locations2 = pipeline.fit(lors)
```

PEPT-ML complete recipe

Including two passes of clustering and trajectory separation: Including an example ADAC Forte data initisalisation, two passes of clustering, trajectory separation, plotting and saving trajectories as CSV.

```
# Import what we need from the `pept` library
import pept
from pept.tracking import *
from pept.plots import PlotlyGrapher, PlotlyGrapher2D


# Open interactive plots in the web browser
import plotly
plotly.io.renderers.default = "browser"


# Initialise data from file and set sample size and overlap
filepath = "DS1.da01"
max_tracers = 1

lors = pept.scanners.adac_forte(
    filepath,
    sample_size = 200 * max_tracers,
```

(continues on next page)

(continued from previous page)

```
    overlap = 150 * max_tracers,
)


# Select only the first 1000 samples of LoRs for testing; comment out for all
lors = lors[:1000]


# Create PEPT-ML processing pipeline
pipeline = pept.Pipeline([

    # First pass of clustering
    Cutpoints(max_distance = 0.2),
    HDBSCAN(true_fraction = 0.15, max_tracers = max_tracers),
    SplitLabels() + Centroids(error = True),

    # Second pass of clustering
    Stack(sample_size = 30 * max_tracers, overlap = 30 * max_tracers - 1),
    HDBSCAN(true_fraction = 0.6, max_tracers = max_tracers),
    SplitLabels() + Centroids(),

    # Trajectory separation
    Segregate(window = 20 * max_tracers, cut_distance = 10),
    Stack(),
])


# Process all samples in `lors` in parallel, using `max_workers` threads
trajectories = pipeline.fit(lors)


# Save trajectories as CSV
trajectories.to_csv(filepath + ".csv")

# Save as a fast binary; you can load them back with `pept.load("path")`
trajectories.save(filepath + ".pickle")


# Plot trajectories - first a 2D timeseries, then all 3D positions
PlotlyGrapher2D().add_timeseries(trajectories).show()
PlotlyGrapher().add_points(trajectories).show()
```

Example of a Complex Processing Pipeline

This is an example of "production code" used for tracking tracers in pipe flow imaging, where particles enter and leave the field of view regularly. This pipeline automatically:

- Sets an optimum adaptive time window.
- Runs a first pass of clustering, keeping track of the number of LoRs around the tracers (`cluster_size`) and relative location error (`error`).
- Removes locations with too few LoRs or large errors.
- Sets a new optimum adaptive time window for a second pass of clustering.
- Removes spurious points while the tracer is out of the field of view.
- Separates out different tracer trajectories, removes the ones with too few points and groups them by trajectory.
- Computes the tracer velocity at each location on each trajectory.
- Removes locations at the edges of the detectors.

Each individual step could be an entire program on its own; with the PEPT `Pipeline` architecture, they can be chained in 17 lines of Python code, automatically using all processors available on parallelisable sections.

```
# Create PEPT-ML processing pipeline
pipeline = pept.Pipeline([
    OptimizeWindow(200, overlap = 0.5) + Debug(1),

    # First pass of clustering
    Cutpoints(max_distance = 0.2),
    HDBSCAN(true_fraction = 0.15),
    SplitLabels() + Centroids(cluster_size = True, error = True),

    # Remove erroneous points
    Condition("cluster_size > 30, error < 20"),

    # Second pass of clustering
    OptimizeWindow(30, overlap = 0.95) + Debug(1),
    HDBSCAN(true_fraction = 0.6),
    SplitLabels() + Centroids(),

    # Remove sparse points in time
    OutOfViewFilter(200.),

    # Trajectory separation
    Segregate(window = 20, cut_distance = 20, min_trajectory_size = 20),
    Condition("label >= 0"),
    GroupBy("label"),

    # Velocity computation
    Velocity(11),
    Velocity(11, absolute = True),

    # Cutoff points outside this region
    Condition("y > 100, y < 500"),
```

(continues on next page)

(continued from previous page)

```
    Stack(),
])
```

5.2.8 Feature Point Identification

FPI is a modern voxel-based tracer-location algorithm that can reliably work with unknown numbers of tracers in fast and noisy environments.

It was successfully used to track fast-moving radioactive tracers in pipe flows at the Virginia Commonwealth University. If you use this algorithm in your work, please cite the following paper:

> *Wiggins C, Santos R, Ruggles A. A feature point identification method for positron emission particle tracking with multiple tracers. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2017 Jan 21; 843:22-8.*

FPI Recipe

As FPI works on voxelized representations of the LoRs, the `Voxelize` filter is first used before `FPI` itself:

```
import pept
from pept.tracking import *

resolution = (100, 100, 100)

pipeline = pept.Pipeline([
    Voxelize(resolution),
    FPI(w = 3, r = 0.4),
    Stack(),
])

locations = pipeline.fit(lors)
```

5.2.9 Tracking Errors

When processing more difficult datasets - scattering environments, low tracer activities, etc. - it is often useful to use some tracer statistics to remove erroneous locations.

Most PEPT algorithms will include some measure of the tracer location errors, for example:

- The `Centroids(error = True)` filter appends a column "error" representing the standard deviation of the distances from the computed centroid to the constituent points. For a 500 mm scanner, a spread in a tracer location of 100 mm is clearly an erroneous point.
- The `Centroids(cluster_size = True)` filter appends a column "cluster_size" representing the number of points used to compute the centroid. If a sample of 200 LoRs yields a tracer location computed from 5 points, it is clearly noise.
- The `BirminghamMethod` filter includes a column "error" representing the standard deviation of the distances from the tracer position to the constituent LoRs.

Histogram of Tracking Errors

You can select a named column via string indexing, e.g. `trajectories["error"]`; you can then plot a histogram of the relative errors with:

```
import plotly.express as px
px.histogram(trajectories["error"]).show()          # Large values are noise
px.histogram(trajectories["cluster_size"]).show()   # Small values are noise
```

It is often useful to remove points with an error higher than a certain value, e.g. 20 mm:

```
trajectories = Condition("error < 20").fit(trajectories)

# Or simply append the `Condition` to the `pept.Pipeline`
pipeline = pept.Pipeline([
    ...
    Condition("cluster_size > 30, error < 20"),
    ...
])
```

5.2.10 Trajectory Separation

Segregate Points

We can separate out trajectory segments / points that are spatio-temporally far away to:

1. Remove spurious, noisy points.
2. Separate out continuous trajectory segments.

The *spatio-temporal metric* differentiates between points that may be in the same location at different times. This is achieved by allowing points to be connected in a sliding window approach.

The `pept.tracking.Segregate` algorithm works by creating a *Minimum Spanning Tree* (MST, or minimum distance path) connecting all points in a dataset, then *cutting* all paths longer than a `cut_distance`. All distinct segments are assigned a trajectory `'label'` (integer starting from 0); trajectories with fewer than `min_trajectory_size` points are considered noise (label *-1*).

```
from pept.tracking import *

trajectories = Segregate(window = 20, cut_distance = 10.).fit(trajectories)
```

Consider all trajectories with fewer than 50 points to be noise:

```
segr = Segregate(
    window = 20,
    cut_distance = 10.,
    min_trajectory_size = 50,
)

trajectories = segr.fit(trajectories)
```

This step adds a new column “label”. We can group each individual trajectory into a list with `GroupBy`:

```
traj_list = GroupBy("label").fit(trajectories)
traj_list[0]     # First trajectory
```

[New in pept-0.5.2] Only connect points within a time interval; in other words, disconnect into different trajectories points whose timestamps are further apart than `max_time_interval`:

```
 segr = Segregate(
    window = 20,
    cut_distance = 10.,
    min_trajectory_size = 50,
    max_time_interval = 2000,       # Disconnect tracer with >2s gap
)

trajectories = segr.fit(trajectories)
```

5.2.11 Filtering Data

There are many filters in `pept.tracking`, you can check out the Manual at the top of the page for a complete list. Here are examples with the most important ones.

Remove

Simply remove a column:

```
from pept.tracking import *

trajectories = Remove("label").fit(trajectories)
```

Or multiple columns:

```
trajectories = Remove("label", "error").fit(trajectories)
```

Condition

One of the most important filters, selecting only data that satisfies a condition:

```
from pept.tracking import *

trajectories = Condition("error < 15").fit(trajectories)
```

Or multiple ones:

```
trajectories = Condition("error < 15, label >= 0").fit(trajectories)
```

In the simplest case, you just use the column name **as the first argument** followed by a comparison. If the column name is not the first argument, you must use single quotes:

```
trajectories = Condition("0 <= 'label'").fit(trajectories)
```

You can also use filtering functions from NumPy in the condition string (i.e. anything returning a boolean mask):

```
# Remove all NaNs and Infs from the 'x' column
trajectories = Condition("np.isfinite('x')")
```

Finally, you can supply your own function receiving a NumPy array of the data and returning a boolean mask:

```
def last_column_filter(data):
    return data[:, -1] > 10

trajectories = Condition(last_column_filter).fit(trajectories)
```

Or using inline functions (i.e. `lambda`):

```
# Select points within a vertical cylinder with radius 10
trajectories = Condition(lambda x: x[:, 1]**2 + x[:, 3]**2 < 10**2).fit(trajectories)
```

SamplesCondition

While `Condition` is applied on individual points, we could filter entire samples - for example, select only trajectories with more than 30 points:

```
import pept.tracking as pt

long_trajectories_filter = pept.Pipeline([
    # Segregate points - appends "label" column
    pt.Segregate(window = 20, cut_distance = 10),

    # Group points into samples; e.g. sample 1 contains all points with label 1
    pt.GroupBy("label"),

    # Now each sample is an entire trajectory which we can filter
    pt.SamplesCondition("sample_size > 30"),

    # And stack all remaining samples back into a single PointData
    pt.Stack(),
])

long_trajectories = long_trajectories_filter.fit(trajectories)
```

The condition can be based on the sample itself, e.g. keep only samples that lie completely beyond x=0:

```
# Keep only samples for which all points' X coordinates are bigger than 0
Condition("np.all(sample['x'] > 0)")
```

GroupBy

Stack all samples (i.e. LineData or PointData) and split them into a list according to a named / numeric column index:

```
from pept.tracking import *

group_list = GroupBy("label").fit(trajectories)
```

RemoveStatic

Remove tracer locations when it spends more than *time_window* without moving more than *max_distance*:

```
from pept.tracking import *

# Remove positions that spent more than 2 seconds without moving more than 20 mm
nonstatic = RemoveStatic(time_window = 2000, max_distance = 20).fit(trajectories)
```

5.2.12 Extracting Velocities

When extracting post-processed data from tracer trajectories for e.g. probability distributions, it is often important to **sample data at fixed timesteps**. As PEPT is natively a Lagrangian technique where tracers can be tracked more often in more sensitive areas of the gamma scanners, we have to convert those "randomly-sampled" positions into regular timesteps using Interpolate.

First, Segregate points into individual, continuous trajectory segments, GroupBy according to each trajectory's label, then Interpolate into regular timesteps, then compute each point's Velocity (dimension-wise or absolute) and finally Stack them back into a PointData:

```
from pept.tracking import *

pipe_vel = pept.Pipeline([
    Segregate(window = 20, cut_distance = 10.),
    GroupBy("label"),
    Interpolate(timestep = 5.),
    Velocity(window = 7),
    Stack(),
])

trajectories = pipe_vel.fit(trajectories)
```

The Velocity step appends columns ["vx", "vy", "vz"] (default) or ["v"] (if absolute = True). You can add both if you wish:

```
from pept.tracking import *

pept.Pipeline([
    Segregate(window = 20, cut_distance = 10.),
    GroupBy("label"),
    Interpolate(timestep = 5.),
    Velocity(window = 7),                       # Appends vx, vy, vz
    Velocity(window = 7, absolute = True),      # Appends v
```

(continues on next page)

(continued from previous page)

```
    Stack(),
])
```

5.2.13 Interpolating Timesteps

When extracting post-processed data from tracer trajectories for e.g. probability distributions, it is often important to **sample data at fixed timesteps**. As PEPT is natively a Lagrangian technique where tracers can be tracked more often in more sensitive areas of the gamma scanners, we have to convert those "randomly-sampled" positions into regular timesteps using `Interpolate`.

First, `Segregate` points into individual, continuous trajectory segments, `GroupBy` according to each trajectory's label, then `Interpolate` into regular timesteps and finally `Stack` them back into a `PointData`:

```
from pept.tracking import *

pipe = pept.Pipeline([
    Segregate(window = 20, cut_distance = 10.),
    GroupBy("label"),
    Interpolate(timestep = 5.),
    Stack(),
])

trajectories = pipe.fit(trajectories)
```

5.3 Manual

All public `pept` subroutines are fully documented here, along with copy-pastable examples. The *base* functionality is summarised below; the rest of the library is organised into submodules, which you can access on the left. You can also use the *Search* bar in the top left to go directly to what you need.

We really appreciate all help with writing useful documentation; if you feel something can be improved, or would like to share some example code, by all means get in contact with us - or be a superhero and click *Edit this page* on the right and submit your changes to the GitHub repository directly!

5.3.1 Base Functions

pept.read_csv(filepath_or_buffer[, ...])	Read a given number of lines from a file and return a numpy array of the values.
pept.load(filepath)	Load a binary saved / pickled object from *filepath*.
pept.save(filepath, obj)	Save an object *obj* instance as a binary file at *filepath*.

pept.read_csv

`pept.read_csv`(*filepath_or_buffer*, *skiprows=None*, *nrows=None*, *dtype=<class 'float'>*, *sep='\\s+'*, *header=None*, *engine='c'*, *na_filter=False*, *quoting=3*, *memory_map=True*, ***kwargs*)

Read a given number of lines from a file and return a numpy array of the values.

This is a convenience function that's simply a proxy to *pandas.read_csv*, configured with default parameters for fast reading and parsing of usual PEPT data.

Most importantly, it reads from a **space-separated values** file at *filepath_or_buffer*, optionally skipping *skiprows* lines and reading in *nrows* lines. It returns a *numpy.ndarray* with *float* values.

The parameters below are sent to *pandas.read_csv* with no further parsing. The descriptions below are taken from the *pandas* documentation.

Parameters

filepath_or_buffer
[`str`, `path object` or file-like `object`] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv. If you want to pass in a path object, pandas accepts any *os.PathLike*. By file-like object, we refer to objects with a *read()* method, such as a file handler (e.g. via builtin *open* function) or *StringIO*.

skiprows
[list-like, `int` or `callable()`, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

nrows
[`int`, optional] Number of rows of file to read. Useful for reading pieces of large files.

dtype
[`Type name`, `default` *float*] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'}.

sep
[`str`, `default` *"s+"*] Delimiter to use. Separators longer than 1 character and different from 's+' will be interpreted as regular expressions and will also force the use of the Python parsing engine.

header
[`int`, `list of int`, "infer", optional] Row number(s) to use as the column names, and the start of the data. By default assume there is no header present (i.e. *header = None*).

engine
[{'c', 'python'}, `default` "c"] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

na_filter
[bool, `default` *True*] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

quoting
[`int` or csv.QUOTE_* `instance`, `default` *csv.QUOTE_NONE*] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

memory_map
[bool, `default True`] If a filepath is provided for filepath_or_buffer, map the file object

directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

****kwargs**
[optional] Extra keyword arguments that will be passed to *pandas.read_csv*.

pept.load

`pept.load`(*filepath*)

Load a binary saved / pickled object from *filepath*.

Most often the full object state was saved using the *pept.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`object`
The loaded Python object instance (e.g. *pept.LineData*).

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> pept.save("lines.pickle", lines)
```

```
>>> lines_reloaded = pept.load("lines.pickle")
```

pept.save

`pept.save`(*filepath*, *obj*)

Save an object *obj* instance as a binary file at *filepath*.

Saves the full object state, including e.g. the inner *.lines* NumPy array, *sample_size*, etc. in a fast, portable binary format. Load back the object using the *pept.load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

obj
[`object`] Any - tipically PEPT-oriented - object to be saved in the binary *pickle* format.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> pept.save("lines.pickle", lines)
```

```
>>> lines_reloaded = pept.load("lines.pickle")
```

5.3.2 Base Classes

`pept.LineData`(lines[, sample_size, overlap, ...])	A class for PEPT LoR data iteration, manipulation and visualisation.
`pept.PointData`(points[, sample_size, ...])	A class for general PEPT point-like data iteration, manipulation and visualisation.
`pept.Pixels`(pixels_array, xlim, ylim, **kwargs)	A class managing a 2D pixel space with physical dimensions, including tools for pixel manipulation and visualisation.
`pept.Voxels`(voxels_array, xlim, ylim, zlim, ...)	A class managing a 3D voxel space with physical dimensions, including tools for voxel manipulation and visualisation.
`pept.Pipeline`(transformers)	A PEPT processing pipeline, chaining multiple *Filter* and *Reducer* for efficient, parallel execution.

pept.LineData

class pept.**LineData**(*lines*, *sample_size=None*, *overlap=None*, *columns=['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']*, ***kwargs*)

Bases: `IterableSamples`

A class for PEPT LoR data iteration, manipulation and visualisation.

Generally, PEPT Lines of Response (LoRs) are lines in 3D space, each defined by two points, regardless of the geometry of the scanner used. This class is used for the encapsulation of LoRs (or any lines!), efficiently yielding samples of *lines* of an adaptive *sample_size* and *overlap*.

It is an abstraction over PET / PEPT scanner geometries and data formats, as once the raw LoRs (be they stored as binary, ASCII, etc.) are transformed into the common *LineData* format, any tracking, analysis or visualisation algorithm in the *pept* package can be used interchangeably. Moreover, it provides a stable, user-friendly interface for iterating over LoRs in *samples* - this is useful for tracking algorithms, as they generally take a few LoRs (a *sample*), produce a tracer position, then move to the next sample of LoRs, repeating the procedure. Using overlapping samples is also useful for improving the tracking rate of the algorithms.

This is the base class for LoR data; the subroutines for transforming other data formats into *LineData* can be found in *pept.scanners*. If you'd like to integrate another scanner geometry or raw data format into this package, you can check out the *pept.scanners.parallel_screens* module as an example. This usually only involves writing a single function by hand; then all attributes and methods from *LineData* will be available to your new data format. If you'd like to use *LineData* as the base for other algorithms, you can check out the *pept.tracking.peptml.cutpoints* module as an example; the *Cutpoints* class iterates the samples of LoRs in any *LineData* **in parallel**, using *concurrent.futures.ThreadPoolExecutor*.

See also:

pept.PointData
: Encapsulate points for ease of iteration and plotting.

pept.read_csv
: Fast CSV file reading into numpy arrays.

PlotlyGrapher
: Easy, publication-ready plotting of PEPT-oriented data.

pept.tracking.Cutpoints
: Compute cutpoints from *pept.LineData*.

Notes

The class saves *lines* as a **C-contiguous** numpy array for efficient access in C / Cython functions. The inner data can be mutated, but do not change the number of rows or columns after instantiating the class.

Examples

Initialise a *LineData* instance containing 10 lines with a *sample_size* of 3.

```
>>> import pept
>>> import numpy as np
>>> lines_raw = np.arange(70).reshape(10, 7)
>>> print(lines_raw)
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]
 [35 36 37 38 39 40 41]
 [42 43 44 45 46 47 48]
 [49 50 51 52 53 54 55]
 [56 57 58 59 60 61 62]
 [63 64 65 66 67 68 69]]
```

```
>>> line_data = pept.LineData(lines_raw, sample_size = 3)
>>> line_data
pept.LineData (samples: 3)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 10, columns: 7)
  [[ 0.  1. ...  5.  6.]
   [ 7.  8. ... 12. 13.]
   ...
   [56. 57. ... 61. 62.]
   [63. 64. ... 68. 69.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

Access samples using subscript notation. Notice how the samples are consecutive, as *overlap* is 0 by default.

```
>>> line_data[0]
pept.LineData (samples: 1)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 3, columns: 7)
  [[ 0.  1. ...  5.  6.]
   [ 7.  8. ... 12. 13.]
   [14. 15. ... 19. 20.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

```
>>> line_data[1]
pept.LineData (samples: 1)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 3, columns: 7)
  [[21. 22. ... 26. 27.]
   [28. 29. ... 33. 34.]
   [35. 36. ... 40. 41.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

Now set an overlap of 2; notice how the number of samples changes:

```
>>> len(line_data)     # Number of samples
3
```

```
>>> line_data.overlap = 2
>>> len(line_data)
8
```

Notice how rows are repeated from one sample to the next when accessing them, because *overlap* is now 2:

```
>>> line_data[0]
pept.LineData (samples: 1)
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 3, columns: 7)
  [[ 0.  1. ...  5.  6.]
   [ 7.  8. ... 12. 13.]
   [14. 15. ... 19. 20.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

```
>>> line_data[1]
pept.LineData (samples: 1)
```

(continues on next page)

(continued from previous page)

```
--------------------------
sample_size = 3
overlap = 0
lines =
  (rows: 3, columns: 7)
  [[ 7.  8. ... 12. 13.]
   [14. 15. ... 19. 20.]
   [21. 22. ... 26. 27.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

Now change *sample_size* to 5 and notice again how the number of samples changes:

```
>>> len(line_data)
8
```

```
>>> line_data.sample_size = 5
>>> len(line_data)
2
```

```
>>> line_data[0]
pept.LineData (samples: 1)
--------------------------
sample_size = 5
overlap = 0
lines =
  (rows: 5, columns: 7)
  [[ 0.  1. ...  5.  6.]
   [ 7.  8. ... 12. 13.]
   ...
   [21. 22. ... 26. 27.]
   [28. 29. ... 33. 34.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

```
>>> line_data[1]
pept.LineData (samples: 1)
--------------------------
sample_size = 5
overlap = 0
lines =
  (rows: 5, columns: 7)
  [[21. 22. ... 26. 27.]
   [28. 29. ... 33. 34.]
   ...
   [42. 43. ... 47. 48.]
   [49. 50. ... 54. 55.]]
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
attrs = {}
```

Notice how the samples do not cover the whole input *lines_raw* array, as the last lines are omitted - think of the *sample_size* and *overlap*. They are still inside the inner *lines* attribute of *line_data* though:

```
>>> line_data.lines
array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.],
       [ 7.,  8.,  9., 10., 11., 12., 13.],
       [14., 15., 16., 17., 18., 19., 20.],
       [21., 22., 23., 24., 25., 26., 27.],
       [28., 29., 30., 31., 32., 33., 34.],
       [35., 36., 37., 38., 39., 40., 41.],
       [42., 43., 44., 45., 46., 47., 48.],
       [49., 50., 51., 52., 53., 54., 55.],
       [56., 57., 58., 59., 60., 61., 62.],
       [63., 64., 65., 66., 67., 68., 69.]])
```

Attributes

lines
[(N, M>=7) `numpy.ndarray`] An (N, M>=7) numpy array that stores the PEPT LoRs as time and cartesian (3D) coordinates of two points defining a line, followed by any additional data. The data columns are then *[time, x1, y1, z1, x2, y2, z2, etc.]*.

sample_size
[`int`, `list[int]`, `pept.TimeWindow` or `None`] Defining the number of LoRs in a sample; if it is an integer, a constant number of LoRs are returned per sample. If it is a list of integers, sample *i* will have length *sample_size[i]*. If it is a *pept.TimeWindow* instance, each sample will span a fixed time window. If *None*, custom sample sizes are returned as per the *samples_indices* attribute.

overlap
[`int`, `pept.TimeWindow` or `None`] Defining the overlapping LoRs between consecutive samples. If *int*, constant numbers of LoRs are used. If *pept.TimeWindow*, the overlap will be a constant time window across the data timestamps (first column). If *None*, custom sample sizes are defined as per the *samples_indices* attribute.

samples_indices
[(S, 2) `numpy.ndarray`] A 2D NumPy array of integers, where row *i* defines the i-th sample's start and end row indices, i.e. *sample[i] == data[samples_indices[i, 0]:samples_indices[i, 1]]*. The *sample_size* and *overlap* are simply friendly interfaces to setting the *samples_indices*.

columns
[(M,) `list[str]`] A list of strings with the same number of columns as *lines* containing each column's name.

attrs
[`dict[str, Any]`] A dictionary of other attributes saved on this class. Attribute names starting with an underscore are considered "hidden".

`__init__`(*lines*, *sample_size=None*, *overlap=None*, *columns=['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']*, ***kwargs*)

LineData class constructor.

Parameters

lines
[(N, M>=7) `numpy.ndarray`] An (N, M>=7) numpy array that stores the PEPT LoRs (or any generic set of lines) as time and cartesian (3D) coordinates of two points defining each line, followed by any additional data. The data columns are then *[time, x1, y1, z1, x2, y2, z2, etc.]*.

sample_size
[`int`, `default` 0] An *int* that defines the number of lines that should be returned when iterating over *lines*. A *sample_size* of 0 yields all the data as one single sample.

overlap
[`int`, `default` 0] An *int* that defines the overlap between two consecutive samples that are returned when iterating over *lines*. An overlap of 0 means consecutive samples, while an overlap of (*sample_size* - 1) means incrementing the samples by one. A negative overlap means skipping values between samples. An error is raised if *overlap* is larger than or equal to *sample_size*.

columns
[`List[str]`, `default` ["t", "x1", "y1", "z1", "x2", "y2", "z2"]] A list of strings corresponding to the column labels in *points*.

****kwargs**
[`extra keyword arguments`] Any extra attributes to set in *.attrs*.

Raises

`ValueError`
If *lines* has fewer than 7 columns.

`ValueError`
If *overlap* >= *sample_size* unless *sample_size* is 0. Overlap has to be smaller than *sample_size*. Note that it can also be negative.

Methods

`__init__`(lines[, sample_size, overlap, columns])	*LineData* class constructor.
`copy`([deep, data, extra, hidden])	Construct a similar object, optionally with different *data*.
`extra_attrs`()	
`hidden_attrs`()	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`plot`([sample_indices, ax, alt_axes, ...])	Plot lines from selected samples using matplotlib.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.
`to_csv`(filepath[, delimiter])	Write *lines* to a CSV file.

Attributes

attrs
columns
data
lines
overlap
sample_size
samples_indices

property lines

to_csv(*filepath*, *delimiter=' '*)

Write *lines* to a CSV file.

Write all LoRs stored in the class to a CSV file.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

delimiter
[str, default " "] The delimiter used to separate the values in the CSV file.

plot(*sample_indices=Ellipsis*, *ax=None*, *alt_axes=False*, *colorbar_col=0*)

Plot lines from selected samples using matplotlib.

Returns matplotlib figure and axes objects containing all lines included in the samples selected by *sample_indices*. *sample_indices* may be a single sample index (e.g. 0), an iterable of indices (e.g. [1,5,6]), or an Ellipsis (. . .) for all samples.

Parameters

sample_indices
[int or iterable or Ellipsis, default Ellipsis] The index or indices of the samples of lines. An *int* signifies the sample index, an iterable (list-like) signifies multiple sample indices, while an Ellipsis (. . .) signifies all samples. The default is . . . (all lines).

ax
[mpl_toolkits.mplot3D.Axes3D object, optional] The 3D matplotlib-based axis for plotting. If undefined, new Matplotlib figure and axis objects are created.

alt_axes
[bool, default False] If *True*, plot using the alternative PEPT-style axes convention: z is horizontal, y points upwards. Because Matplotlib cannot swap axes, this is achieved by swapping the parameters in the plotting call (i.e. *plt.plot(x, y, z)* -> *plt.plot(z, x, y)*).

colorbar_col
[int, default -1] The column in the data samples that will be used to color the lines. The default is -1 (the last column).

Returns

fig, ax
[matplotlib figure and axes objects]

Notes

Plotting all lines is very computationally-expensive for matplotlib. It is recommended to only plot a couple of samples at a time, or use the faster *pept.plots.PlotlyGrapher*.

Examples

Plot the lines from sample 1 in a *LineData* instance:

```
>>> lors = pept.LineData(...)
>>> fig, ax = lors.plot(1)
>>> fig.show()
```

Plot the lines from samples 0, 1 and 2:

```
>>> fig, ax = lors.plot([0, 1, 2])
>>> fig.show()
```

property attrs

property columns

copy(*deep=True*, *data=None*, *extra=True*, *hidden=True*, ***attrs*)
Construct a similar object, optionally with different *data*. If *extra*, extra attributes are propagated; same for *hidden*.

property data

extra_attrs()

hidden_attrs()

static load(*filepath*)
Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

property overlap

property sample_size

property samples_indices

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.PointData

class pept.**PointData**(*points*, *sample_size=None*, *overlap=None*, *columns=['t', 'x', 'y', 'z']*, ***kwargs*)

Bases: *`IterableSamples`*

A class for general PEPT point-like data iteration, manipulation and visualisation.

In the context of positron-based particle tracking, points are defined by a timestamp, 3D coordinates and any other extra information (such as trajectory label or some tracer signature). This class is used for the encapsulation of 3D points - be they tracer locations, cutpoints, etc. -, efficiently yielding samples of *points* of an adaptive *sample_size* and *overlap*.

Much like a complement to *LineData*, *PointData* is an abstraction over point-like data that may be encountered in the context of PEPT (e.g. pre-tracked tracer locations), as once the raw points are transformed into the common *PointData* format, any tracking, analysis or visualisation algorithm in the *pept* package can be used interchangeably. Moreover, it provides a stable, user-friendly interface for iterating over points in *samples* - this can be useful for tracking algorithms, as some take a few points (a *sample*), produce an accurate tracer location, then move to the next sample of points, repeating the procedure. Using overlapping samples is also useful for improving the time resolution of the algorithms.

This is the base class for point-like data; subroutines that accept and/or return *PointData* instances (or subclasses thereof) can be found throughout the *pept* package. If you'd like to create new algorithms based on them, you can check out the *pept.tracking.peptml.cutpoints* module as an example; the *Cutpoints* class receives a *LineData* instance, transforms the samples of LoRs into cutpoints, then initialises itself as a *PointData* subclass - thereby inheriting all its methods and attributes.

Raises

`ValueError`
If *overlap* >= *sample_size*. Overlap is required to be smaller than *sample_size*, unless *sample_size* is 0. Note that it can also be negative.

See also:

`pept.LineData`
Encapsulate LoRs for ease of iteration and plotting.

`pept.read_csv`
Fast CSV file reading into numpy arrays.

`pept.plots.PlotlyGrapher`
Easy, publication-ready plotting of PEPT-oriented data.

`pept.tracking.Cutpoints`
Compute cutpoints from *pept.LineData*.

Notes

This class saves *points* as a **C-contiguous** numpy array for efficient access in C / Cython functions. The inner data can be mutated, but do not change the number of rows or columns after instantiating the class.

Examples

Initialise a *PointData* instance containing 10 points with a *sample_size* of 3.

```
>>> import numpy as np
>>> import pept
>>> points_raw = np.arange(40).reshape(10, 4)
>>> print(points_raw)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]
 [24 25 26 27]
 [28 29 30 31]
 [32 33 34 35]
 [36 37 38 39]]
```

```
>>> point_data = pept.PointData(points_raw, sample_size = 3)
>>> point_data
pept.PointData (samples: 3)
---------------------------
sample_size = 3
```

(continues on next page)

(continued from previous page)

```
overlap = 0
points =
  (rows: 10, columns: 4)
  [[ 0.   1.   2.   3.]
   [ 4.   5.   6.   7.]
   ...
   [32.  33.  34.  35.]
   [36.  37.  38.  39.]]
columns = ['t', 'x', 'y', 'z']
attrs = {}
```

Access samples using subscript notation. Notice how the samples are consecutive, as *overlap* is 0 by default.

```
>>> point_data[0]
pept.PointData (samples: 1)
---------------------------
sample_size = 3
overlap = 0
points =
  (rows: 3, columns: 4)
  [[ 0.   1.   2.   3.]
   [ 4.   5.   6.   7.]
   [ 8.   9.  10.  11.]]
columns = ['t', 'x', 'y', 'z']
attrs = {}
```

```
>>> point_data[1]
pept.PointData (samples: 1)
---------------------------
sample_size = 3
overlap = 0
points =
  (rows: 3, columns: 4)
  [[12.  13.  14.  15.]
   [16.  17.  18.  19.]
   [20.  21.  22.  23.]]
columns = ['t', 'x', 'y', 'z']
attrs = {}
```

Now set an overlap of 2; notice how the number of samples changes:

```
>>> len(point_data)         # Number of samples
3
```

```
>>> point_data.overlap = 2
>>> len(point_data)
8
```

Notice how rows are repeated from one sample to the next when accessing them, because *overlap* is now 2:

```
>>> point_data[0]
array([[ 0.,   1.,   2.,   3.],
```

(continues on next page)

(continued from previous page)

```
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
>>> point_data[1]
array([[ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```

Now change *sample_size* to 5 and notice again how the number of samples changes:

```
>>> len(point_data)
8
```

```
>>> point_data.sample_size = 5
>>> len(point_data)
2
```

```
>>> point_data[0]
pept.PointData (samples: 1)
---------------------------
sample_size = 3
overlap = 0
points =
  (rows: 3, columns: 4)
  [[ 0.  1.  2.  3.]
   [ 4.  5.  6.  7.]
   [ 8.  9. 10. 11.]]
columns = ['t', 'x', 'y', 'z']
attrs = {}
```

```
>>> point_data[1]
pept.PointData (samples: 1)
---------------------------
sample_size = 3
overlap = 0
points =
  (rows: 3, columns: 4)
  [[ 4.  5.  6.  7.]
   [ 8.  9. 10. 11.]
   [12. 13. 14. 15.]]
columns = ['t', 'x', 'y', 'z']
attrs = {}
```

Notice how the samples do not cover the whole input *points_raw* array, as the last lines are omitted - think of the *sample_size* and *overlap*. They are still inside the inner *points* attribute of *point_data* though:

```
>>> point_data.points
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
```

(continues on next page)

(continued from previous page)

```
        [16., 17., 18., 19.],
        [20., 21., 22., 23.],
        [24., 25., 26., 27.],
        [28., 29., 30., 31.],
        [32., 33., 34., 35.],
        [36., 37., 38., 39.]])
```

Attributes

points
[(N, M) numpy.ndarray] An (N, M >= 4) numpy array that stores the points as time, followed by cartesian (3D) coordinates of the point, followed by any extra information. The data columns are then *[time, x, y, z, etc]*.

sample_size
[int, list[int], *pept.TimeWindow* or None] Defining the number of points in a sample; if it is an integer, a constant number of points are returned per sample. If it is a list of integers, sample *i* will have length *sample_size[i]*. If it is a *pept.TimeWindow* instance, each sample will span a fixed time window. If *None*, custom sample sizes are returned as per the *samples_indices* attribute.

overlap
[int, *pept.TimeWindow* or None] Defining the overlapping points between consecutive samples. If *int*, constant numbers of points are used. If *pept.TimeWindow*, the overlap will be a constant time window across the data timestamps (first column). If *None*, custom sample sizes are defined as per the *samples_indices* attribute.

samples_indices
[(S, 2) numpy.ndarray] A 2D NumPy array of integers, where row *i* defines the i-th sample's start and end row indices, i.e. *sample[i] == data[samples_indices[i, 0]:samples_indices[i, 1]]*. The *sample_size* and *overlap* are simply friendly interfaces to setting the *samples_indices*.

columns
[(M,) list[str]] A list of strings with the same number of columns as *points* containing each column's name.

attrs
[dict[str, Any]] A dictionary of other attributes saved on this class. Attribute names starting with an underscore are considered "hidden".

`__init__`(*points*, *sample_size=None*, *overlap=None*, *columns=['t', 'x', 'y', 'z']*, ***kwargs*)
PointData class constructor.

Parameters

points
[(N, M) numpy.ndarray] An (N, M >= 4) numpy array that stores points (or any generic 2D set of data). It expects that the first column is time, followed by cartesian (3D) coordinates of points, followed by any extra information the user needs. The data columns are then *[time, x, y, z, etc]*.

sample_size
[int, default 0] An *int`* that defines the number of points that should be returned when iterating over *points*. A *sample_size* of 0 yields all the data as one single sample.

overlap
[`int`, `default` 0] An *int* that defines the overlap between two consecutive samples that are returned when iterating over *points*. An overlap of 0 means consecutive samples, while an overlap of (*sample_size* - 1) implies incrementing the samples by one. A negative overlap means skipping values between samples. An error is raised if *overlap* is larger than or equal to *sample_size*.

columns
[`List[str]`, `default` ["t", "x", "y", "z"]] A list of strings corresponding to the column labels in *points*.

****kwargs**
[`extra keyword arguments`] Any extra attributes to set on the class instance.

Raises

`ValueError`
If *line_data* does not have (N, M) shape, where M >= 4.

Methods

__init__(points[, sample_size, overlap, columns])	*PointData* class constructor.
copy([deep, data, extra, hidden])	Construct a similar object, optionally with different *data*.
extra_attrs()	
hidden_attrs()	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
plot([sample_indices, ax, alt_axes, ...])	Plot points from selected samples using matplotlib.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.
to_csv(filepath[, delimiter])	Write the inner *points* to a CSV file.

Attributes

attrs	
columns	
data	
overlap	
points	
sample_size	
samples_indices	

property points

`to_csv`(*filepath*, *delimiter=' '*)

Write the inner *points* to a CSV file.

Write all points stored in the class to a CSV file.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

delimiter
[`str`, `default` " "] The delimiter used to separate the values in the CSV file.

`plot`(*sample_indices=Ellipsis*, *ax=None*, *alt_axes=False*, *colorbar_col=-1*)

Plot points from selected samples using matplotlib.

Returns matplotlib figure and axes objects containing all points included in the samples selected by *sample_indices*. *sample_indices* may be a single sample index (e.g. 0), an iterable of indices (e.g. [1,5,6]), or an Ellipsis (. . .) for all samples.

Parameters

sample_indices
[`int` or iterable or `Ellipsis`, `default` `Ellipsis`] The index or indices of the samples of points. An *int* signifies the sample index, an iterable (list-like) signifies multiple sample indices, while an Ellipsis (. . .) signifies all samples. The default is . . . (all points).

ax
[`mpl_toolkits.mplot3D.Axes3D` `object`, optional] The 3D matplotlib-based axis for plotting. If undefined, new Matplotlib figure and axis objects are created.

alt_axes
[bool, `default` `False`] If *True*, plot using the alternative PEPT-style axes convention: z is horizontal, y points upwards. Because Matplotlib cannot swap axes, this is achieved by swapping the parameters in the plotting call (i.e. *plt.plot(x, y, z)* -> *plt.plot(z, x, y)*).

colorbar_col
[`int`, `default` -1] The column in the data samples that will be used to color the points. The default is -1 (the last column).

Returns

fig, ax
[`matplotlib figure and axes objects`]

Notes

Plotting all points is very computationally-expensive for matplotlib. It is recommended to only plot a couple of samples at a time, or use the faster *pept.plots.PlotlyGrapher*.

Examples

Plot the points from sample 1 in a *PointData* instance:

```
>>> point_data = pept.PointData(...)
>>> fig, ax = point_data.plot(1)
>>> fig.show()
```

Plot the points from samples 0, 1 and 2:

```
>>> fig, ax = point_data.plot([0, 1, 2])
>>> fig.show()
```

property attrs

property columns

copy(*deep=True*, *data=None*, *extra=True*, *hidden=True*, ***attrs*)

Construct a similar object, optionally with different *data*. If *extra*, extra attributes are propagated; same for *hidden*.

property data

extra_attrs()

hidden_attrs()

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

property overlap

property sample_size

property samples_indices

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.Pixels

class pept.**Pixels**(*pixels_array*, *xlim*, *ylim*, ***kwargs*)

Bases: `object`

A class managing a 2D pixel space with physical dimensions, including tools for pixel manipulation and visualisation.

The *.pixels* attribute is simply a *numpy.ndarray[ndim=2, dtype=float64]*. If you think of *Pixels* as an image, the origin is the top left corner, the X-dimension is the left edge and the Y-dimension is the top edge, so that it can be indexed as *.pixels[ix, iy]*.

The *.attrs* dictionary can be used to store extra information.

See also:

`konigcell.Voxels`
A class managing a physical 3D voxel space.

`konigcell.dynamic2d`
Rasterize moving particles' trajectories.

`konigcell.static2d`
Rasterize static particles' positions.

`konigcell.dynamic_prob2d`
2D probability distribution of a quantity.

Notes

The class saves *pixels* as a **contiguous** numpy array for efficient access in C / Cython functions. The inner data can be mutated, but do not change the shape of the array after instantiating the class.

Examples

Create a zeroed 4x4 Pixels grid:

```
>>> import konigcell as kc
>>> pixels = kc.Pixels.zeros((4, 4), xlim = [0, 10], ylim = [0, 5])
>>> pixels
Pixels
------
xlim = [ 0. 10.]
ylim = [0. 5.]
pixels =
  (shape: (4, 4))
  [[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]
attrs = {}
```

Or create a Pixels instance from another array (e.g. an image or matrix):

```
>>> import numpy as np
>>> matrix = np.ones((3, 3))
>>> pixels = kc.Pixels(matrix, xlim = [0, 10], ylim = [-5, 5])
>>> pixels
Pixels
------
xlim = [ 0. 10.]
ylim = [-5.  5.]
pixels =
  (shape: (3, 3))
  [[1. 1. 1.]
   [1. 1. 1.]
   [1. 1. 1.]]
attrs = {}
```

Access pixels' properties directly:

```
>>> pixels.xlim             # ndarray[xmin, xmax]
>>> pixels.ylim             # ndarray[ymin, ymax]
>>> pixels.pixel_size       # ndarray[xsize, ysize]
>>> pixels.pixels.shape     # pixels resolution - tuple[nx, ny]
```

You can save extra attributes about the pixels instance in the *attrs* dictionary:

```
>>> pixels.attrs["dpi"] = 300
>>> pixels
Pixels
```

(continues on next page)

(continued from previous page)

```
------
xlim = [ 0. 10.]
ylim = [-5.  5.]
pixels =
  (shape: (3, 3))
  [[1. 1. 1.]
   [1. 1. 1.]
   [1. 1. 1.]]
attrs = {
  'dpi': 300
}
```

The lower left and upper right corners of the pixel grid, in physical coordinates (the ones given by xlim and ylim):

```
>>> pixels.lower
array([ 0., -5.])
```

```
>>> pixels.upper
array([10.,  5.])
```

You can access the underlying NumPy array directly:

```
>>> pixels.pixels
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Indexing is forwarded to the NumPy array:

```
>>> pixels[:, :]
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Transform physical units into pixel indices:

```
>>> pixels.from_physical([5, 0])                    # pixel centres
array([1., 1.])
```

```
>>> pixels.from_physical([5, 0], corner = True)     # lower left corners
array([1.5, 1.5])
```

Transform pixel indices into physical units:

```
>>> pixels.to_physical([0, 0])                      # pixels centres
array([ 1.66666667, -3.33333333])
```

```
>>> pixels.to_physical([0, 0], corner = True)       # lower left corners
array([ 0., -5.])
```

Save Pixels instance to disk, as a binary archive:

```
>>> pixels.save("pixels.pickle")
>>> pixels = kc.load("pixels.pickle")
```

Create deep copy of a Pixels instance:

```
>>> Pixels.copy()
```

Matplotlib plotting (optional, if Matplotlib is installed):

```
>>> fig, ax = pixels.plot()
>>> fig.show()
```

Plotly trace (optional, if Plotly is installed):

```
>>> import plotly.graph_objs as go
>>> fig = go.Figure()
>>> fig.add_trace(pixels.heatmap_trace())
>>> fig.show()
```

Attributes

pixels
[(`M`, `N`) `np.ndarray`[ndim=2, dtype=float64]] The 2D numpy array containing the pixel values. This class assumes a uniform grid of pixels - that is, the pixel size in each dimension is constant, but can vary from one dimension to another.

xlim
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lower and upper boundaries of the pixellised volume in the x-dimension, formatted as [x_min, x_max].

ylim
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lower and upper boundaries of the pixellised volume in the y-dimension, formatted as [y_min, y_max].

pixel_size
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lengths of a pixel in the x- and y-dimensions, respectively.

pixel_grids
[`list`[(M+1,) `np.ndarray`, (N+1,) `np.ndarray`]] A list containing the pixel gridlines in the x- and y-dimensions. Each dimension's gridlines are stored as a numpy of the pixel delimitations, such that it has length (M + 1), where M is the number of pixels in a given dimension.

lower
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lower left corner of the pixel rectangle; corresponds to [xlim[0], ylim[0]].

upper
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The upper right corner of the pixel rectangle; corresponds to [xlim[1], ylim[1]].

attrs
[`dict`[`Any`, `Any`]] A dictionary storing any other user-defined information.

`__init__`(*pixels_array*, *xlim*, *ylim*, ***kwargs*)
Pixels class constructor.

Parameters

pixels_array
[3D numpy.ndarray] A 3D numpy array, corresponding to a pre-defined pixel space.

xlim
[(2,) numpy.ndarray] The lower and upper boundaries of the pixellised volume in the x-dimension, formatted as [x_min, x_max].

ylim
[(2,) numpy.ndarray] The lower and upper boundaries of the pixellised volume in the y-dimension, formatted as [y_min, y_max].

****kwargs**
[extra keyword arguments] Extra user-defined attributes to be saved in *.attrs*.

Raises

ValueError
If *pixels_array* does not have exactly 2 dimensions or if *xlim* or *ylim* do not have exactly 2 values each.

Notes

No copies are made if *pixels_array*, *xlim* and *ylim* are contiguous NumPy arrays with dtype=float64.

Methods

`__init__`(pixels_array, xlim, ylim, **kwargs)	*Pixels* class constructor.
`add_lines`(lines[, verbose])	Pixellise a sample of lines, adding 1 to each pixel traversed, for each line in the sample.
`copy`([pixels_array, xlim, ylim])	Create a copy of the current *Pixels* instance, optionally with new *pixels_array*, *xlim* and / or *ylim*.
`from_lines`(lines, number_of_pixels[, xlim, ...])	Create a pixel space and traverse / pixellise a given sample of *lines*.
`from_physical`(locations[, corner])	Transform *locations* from physical dimensions to pixel indices.
`heatmap_trace`([colorscale, transpose, xgap, ...])	Create and return a Plotly *Heatmap* trace of the pixels.
`load`(filepath)	Load a saved / pickled *Pixels* object from *filepath*.
`plot`([ax])	Plot pixels as a heatmap using Matplotlib.
`save`(filepath)	Save a *Pixels* instance as a binary *pickle* object.
`to_physical`(indices[, corner])	Transform *indices* from pixel indices to physical dimensions.
`zeros`(shape, xlim, ylim, **kwargs)	

Attributes

attrs
lower
pixel_grids
pixel_size
pixels
upper
xlim
ylim

property pixels

property xlim

property ylim

property attrs

property pixel_size

property pixel_grids

property lower

property upper

static zeros(*shape*, *xlim*, *ylim*, ***kwargs*)

save(*filepath*)

Save a *Pixels* instance as a binary *pickle* object.

Saves the full object state, including the inner *.pixels* NumPy array, *xlim*, etc. in a fast, portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *Pixels* instance, then load it back:

```
>>> import numpy as np
>>> import konigcell as kc
>>>
>>> grid = np.zeros((640, 480))
>>> pixels = kc.Pixels(grid, [0, 20], [0, 10])
>>> pixels.save("pixels.pickle")
```

```
>>> pixels_reloaded = kc.Pixels.load("pixels.pickle")
```

static load(*filepath*)

Load a saved / pickled *Pixels* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.Pixels
The loaded *pept.Pixels* instance.

Examples

Save a *Pixels* instance, then load it back:

```
>>> import numpy as np
>>> import konigcell as kc
>>>
>>> grid = np.zeros((640, 480))
>>> pixels = kc.Pixels(grid, [0, 20], [0, 10])
>>> pixels.save("pixels.pickle")
```

```
>>> pixels_reloaded = kc.Pixels.load("pixels.pickle")
```

copy(*pixels_array=None*, *xlim=None*, *ylim=None*, ***kwargs*)

Create a copy of the current *Pixels* instance, optionally with new *pixels_array*, *xlim* and / or *ylim*.

The extra attributes in *.attrs* are propagated too. Pass new attributes as extra keyword arguments.

from_physical(*locations*, *corner=False*)

Transform *locations* from physical dimensions to pixel indices. If *corner* = *True*, return the index of the bottom left corner of each pixel; otherwise, use the pixel centres.

Examples

Create a simple *konigcell.Pixels* grid, spanning [-5, 5] mm in the X-dimension and [10, 20] mm in the Y-dimension:

```
>>> import konigcell as kc
>>> pixels = kc.Pixels.zeros((5, 5), xlim=[-5, 5], ylim=[10, 20])
>>> pixels
Pixels
------
xlim = [-5.  5.]
ylim = [10. 20.]
pixels =
  (shape: (5, 5))
  [[0. 0. ... 0. 0.]
   [0. 0. ... 0. 0.]
   ...
   [0. 0. ... 0. 0.]
   [0. 0. ... 0. 0.]]
attrs = {}
```

```
>>> pixels.pixel_size
array([2., 2.])
```

Transform physical coordinates to pixel coordinates:

```
>>> pixels.from_physical([-5, 10], corner = True)
array([0., 0.])
```

```
>>> pixels.from_physical([-5, 10])
array([-0.5, -0.5])
```

The pixel coordinates are returned exactly, as real numbers. For pixel indices, round them into values:

```
>>> pixels.from_physical([0, 15]).astype(int)
array([2, 2])
```

Multiple coordinates can be given as a 2D array / list of lists:

```
>>> pixels.from_physical([[0, 15], [5, 20]])
array([[2. , 2. ],
       [4.5, 4.5]])
```

to_physical(*indices*, *corner=False*)

Transform *indices* from pixel indices to physical dimensions. If *corner* = *True*, return the coordinates of the bottom left corner of each pixel; otherwise, use the pixel centres.

Examples

Create a simple *konigcell.Pixels* grid, spanning [-5, 5] mm in the X-dimension and [10, 20] mm in the Y-dimension:

```
>>> import konigcell as kc
>>> pixels = kc.Pixels.zeros((5, 5), xlim=[-5, 5], ylim=[10, 20])
>>> pixels
Pixels
------
xlim = [-5.  5.]
ylim = [10. 20.]
pixels =
  (shape: (5, 5))
  [[0. 0. ... 0. 0.]
   [0. 0. ... 0. 0.]
   ...
   [0. 0. ... 0. 0.]
   [0. 0. ... 0. 0.]]
attrs = {}
```

```
>>> pixels.pixel_size
array([2., 2.])
```

Transform physical coordinates to pixel coordinates:

```
>>> pixels.to_physical([0, 0], corner = True)
array([-5., 10.])
```

```
>>> pixels.to_physical([0, 0])
array([-4., 11.])
```

Multiple coordinates can be given as a 2D array / list of lists:

```
>>> pixels.to_physical([[0, 0], [4, 3]])
array([[-4., 11.],
       [ 4., 17.]])
```

heatmap_trace(*colorscale='Magma'*, *transpose=True*, *xgap=0.0*, *ygap=0.0*)

Create and return a Plotly *Heatmap* trace of the pixels.

Parameters

colorscale
[str, default "Magma"] The Plotly scheme for color-coding the pixel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

transpose
[bool, default True] Transpose the heatmap (i.e. flip it across its diagonal).

Examples

Create a Pixels array and plot it as a heatmap using Plotly:

```
>>> import konigcell as kc
>>> import numpy as np
>>> import plotly.graph_objs as go
```

```
>>> pixels_raw = np.arange(150).reshape(10, 15)
>>> pixels = kc.Pixels(pixels_raw, [-5, 5], [-5, 10])
```

```
>>> fig = go.Figure()
>>> fig.add_trace(pixels.heatmap_trace())
>>> fig.show()
```

`plot`(*ax=None*)

Plot pixels as a heatmap using Matplotlib.

Returns matplotlib figure and axes objects containing the pixel values colour-coded in a Matplotlib image (i.e. heatmap).

Parameters

ax
[`mpl_toolkits.mplot3D.Axes3D` `object`, optional] The 3D matplotlib-based axis for plotting. If undefined, new Matplotlib figure and axis objects are created.

Returns

`fig`, `ax`
Matplotlib figure and axes objects.

Examples

Pixellise an array of lines and plot them with Matplotlib:

```
>>> lines = np.array(...)                    # shape (N, M >= 7)
>>> lines2d = lines[:, [0, 1, 2, 4, 5]]      # select x, y of lines
>>> number_of_pixels = [10, 10]
>>> pixels = pept.Pixels.from_lines(lines2d, number_of_pixels)
```

```
>>> fig, ax = pixels.plot()
>>> fig.show()
```

`add_lines`(*lines*, *verbose=False*)

Pixellise a sample of lines, adding 1 to each pixel traversed, for each line in the sample.

Parameters

lines
[(`M`, `N` >= 5) `numpy.ndarray`] The sample of 2D lines to pixellise. Each line is defined as a timestamp followed by two 2D points, such that the data columns are *[time, x1, y1, x2, y2, . . .]*. Note that there can be extra data columns which will be ignored.

verbose
[bool, `default` `False`] Time the pixel traversal and print it to the terminal.

Raises

`ValueError`
If *lines* has fewer than 5 columns.

`static from_lines`(*lines*, *number_of_pixels*, *xlim=None*, *ylim=None*, *verbose=True*)

Create a pixel space and traverse / pixellise a given sample of *lines*.

The *number_of_pixels* in each dimension must be defined. If the pixel space boundaries *xlim* or *ylim* are not defined, they are inferred as the boundaries of the *lines*.

Parameters

lines
[(M, N>=5) `numpy.ndarray`] The lines that will be pixellised, each defined by a timestamp and two 2D points, so that the data columns are [time, x1, y1, x2, y2]. Note that extra columns are ignored.

number_of_pixels
[(2,) `list[int]`] The number of pixels in the x- and y-dimensions, respectively.

xlim
[(2,) `list[float]`, optional] The lower and upper boundaries of the pixellised volume in the x-dimension, formatted as [x_min, x_max]. If undefined, it is inferred from the boundaries of *lines*.

ylim
[(2,) `list[float]`, optional] The lower and upper boundaries of the pixellised volume in the y-dimension, formatted as [y_min, y_max]. If undefined, it is inferred from the boundaries of *lines*.

Returns

`pept.Pixels`
A new *Pixels* object with the pixels through which the lines were traversed.

Raises

`ValueError`
If the input *lines* does not have the shape (M, N>=5). If the *number_of_pixels* is not a 1D list with exactly 2 elements, or any dimension has fewer than 2 pixels.

pept.Voxels

`class pept.Voxels`(*voxels_array*, *xlim*, *ylim*, *zlim*, ***kwargs*)

Bases: `object`

A class managing a 3D voxel space with physical dimensions, including tools for voxel manipulation and visualisation.

The *.voxels* attribute is simply a *numpy.ndarray[ndim=3, dtype=float64]*. The *.attrs* dictionary can be used to store extra information.

See also:

`konigcell.Pixels`
A class managing a physical 2D pixel space.

`konigcell.dynamic3d`
Rasterize moving particles' trajectories.

`konigcell.static3d`
Rasterize static particles' positions.

`konigcell.dynamic_prob3d`
3D probability distribution of a quantity.

Attributes

voxels
[(`M`, `N`, `P`) `np.ndarray`[ndim=3, dtype=float64]] The 3D numpy array containing the voxel values. This class assumes a uniform grid of voxels - that is, the voxel size in each dimension is constant, but can vary from one dimension to another.

xlim
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lower and upper boundaries of the voxellised volume in the x-dimension, formatted as [x_min, x_max].

ylim
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lower and upper boundaries of the voxellised volume in the y-dimension, formatted as [y_min, y_max].

zlim
[(2,) `np.ndarray`[ndim=1, dtype=float64]] The lower and upper boundaries of the voxellised volume in the z-dimension, formatted as [z_min, z_max].

voxel_size
[(3,) `np.ndarray`[ndim=1, dtype=float64]] The lengths of a voxel in the x-, y- and z-dimensions, respectively.

voxel_grids
[(3,) `list`[`np.ndarray`[ndim=1, dtype=float64]]] A list containing the voxel gridlines in the x-, y-, and z-dimensions. Each dimension's gridlines are stored as a numpy of the voxel delimitations, such that it has length (M + 1), where M is the number of voxels in given dimension.

lower
[(3,) `np.ndarray`[ndim=1, dtype=float64]] The lower left corner of the voxel box; corresponds to [xlim[0], ylim[0], zlim[0]].

upper
[(3,) `np.ndarray`[ndim=1, dtype=float64]] The upper right corner of the voxel box; corresponds to [xlim[1], ylim[1], zlim[1]].

attrs
[`dict`[`Any`, `Any`]] A dictionary storing any other user-defined information.

`__init__`(*voxels_array*, *xlim*, *ylim*, *zlim*, ***kwargs*)
Voxels class constructor.

Parameters

voxels_array
[3D `numpy.ndarray`] A 3D numpy array, corresponding to a pre-defined voxel space.

xlim
[(2,) `numpy.ndarray`] The lower and upper boundaries of the voxellised volume in the x-dimension, formatted as [x_min, x_max].

ylim
[(2,) `numpy.ndarray`] The lower and upper boundaries of the voxellised volume in the y-dimension, formatted as [y_min, y_max].

zlim
[(2,) `numpy.ndarray`] The lower and upper boundaries of the voxellised volume in the z-dimension, formatted as [z_min, z_max].

****kwargs**
[`extra keyword arguments`] Extra user-defined attributes to be saved in *.attrs*.

Raises

`ValueError`
If *voxels_array* does not have exactly 3 dimensions or if *xlim*, *ylim* or *zlim* do not have exactly 2 values each.

Methods

`__init__`(voxels_array, xlim, ylim, zlim, ...)	*Voxels* class constructor.
`add_lines`(lines[, verbose])	Voxellise a sample of lines, adding 1 to each voxel traversed, for each line in the sample.
`copy`([voxels_array, xlim, ylim, zlim])	Create a copy of the current *Voxels* instance, optionally with new *voxels_array*, *xlim* and / or *ylim*.
`cube_trace`(index[, color, opacity, ...])	Get the Plotly *Mesh3d* trace for a single voxel at *index*.
`cubes_traces`([condition, color, opacity, ...])	Get a list of Plotly *Mesh3d* traces for all voxels selected by the *condition* filtering function.
`from_lines`(lines, number_of_voxels[, xlim, ...])	Create a voxel space and traverse / voxellise a given sample of *lines*.
`from_physical`(locations[, corner])	Transform *locations* from physical dimensions to voxel indices.
`heatmap_trace`([ix, iy, iz, width, ...])	Create and return a Plotly *Heatmap* trace of a 2D slice through the voxels.
`load`(filepath)	Load a saved / pickled *Voxels* object from *filepath*.
`plot`([condition, ax, alt_axes])	Plot the voxels in this class using Matplotlib.
`plot_volumetric`([condition, mode, colorscale])	Create a volumetric PyVista plot - check the *mode* argument for the available types.
`save`(filepath)	Save a *Voxels* instance as a binary *pickle* object.
`scatter_trace`([condition, size, color, ...])	Create and return a trace for all the voxels in this class, with possible filtering.
`to_physical`(indices[, corner])	Transform *indices* from voxel indices to physical dimensions.
`vtk`([condition])	Return a PyVista VTK object, exposing all VTK functionality.
`zeros`(shape, xlim, ylim, zlim, **kwargs)	Create a Voxels object filled with zeros.

Attributes

attrs
lower
upper
voxel_grids
voxel_size
voxels
xlim
ylim
zlim

property voxels

property xlim

property ylim

property zlim

property voxel_size

property voxel_grids

property lower

property upper

property attrs

save(*filepath*)

Save a *Voxels* instance as a binary *pickle* object.

Saves the full object state, including the inner *.voxels* NumPy array, *xlim*, etc. in a fast, portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *Voxels* instance, then load it back:

```
>>> import numpy as np
>>> import konigcell as kc
>>>
>>> grid = np.zeros((64, 48, 32))
>>> voxels = kc.Voxels(grid, [0, 20], [0, 10])
>>> voxels.save("voxels.pickle")
```

```
>>> voxels_reloaded = kc.Voxels.load("voxels.pickle")
```

static load(*filepath*)

Load a saved / pickled *Voxels* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.Voxels
The loaded *pept.Voxels* instance.

Examples

Save a *Voxels* instance, then load it back:

```
>>> import numpy as np
>>> import konigcell as kc
>>>
>>> grid = np.zeros((64, 48, 32))
>>> voxels = kc.Voxels(grid, [0, 20], [0, 10])
>>> voxels.save("voxels.pickle")
```

```
>>> voxels_reloaded = kc.Voxels.load("voxels.pickle")
```

copy(*voxels_array=None*, *xlim=None*, *ylim=None*, *zlim=None*, ***kwargs*)

Create a copy of the current *Voxels* instance, optionally with new *voxels_array*, *xlim* and / or *ylim*.

The extra attributes in *.attrs* are propagated too. Pass new attributes as extra keyword arguments.

static zeros(*shape*, *xlim*, *ylim*, *zlim*, ***kwargs*)

Create a Voxels object filled with zeros.

from_physical(*locations*, *corner=False*)

Transform *locations* from physical dimensions to voxel indices. If *corner = True*, return the index of the bottom left corner of each voxel; otherwise, use the voxel centres.

Examples

Create a simple *konigcell.Voxels* grid, spanning [-5, 5] mm in the X-dimension, [10, 20] mm in the Y-dimension and [0, 10] in Z:

```
>>> import konigcell as kc
>>> voxels = kc.Voxels.zeros((5, 5, 5), xlim=[-5, 5], ylim=[10, 20],
                             zlim=[0, 10])
>>> voxels
Voxels
------
xlim = [-5.  5.]
ylim = [10. 20.]
zlim = [10. 20.]
voxels =
  (shape: (5, 5, 5))
  [[[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]
   [[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]
   ...
   [[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]
   [[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]]
attrs = {}
```

```
>>> voxels.voxel_size
array([2., 2., 2.])
```

Transform physical coordinates to voxel coordinates:

```
>>> voxels.from_physical([-5, 10, 0], corner = True)
array([0., 0., 0.])
```

```
>>> voxels.from_physical([-5, 10, 0])
array([-0.5, -0.5, -0.5])
```

The voxel coordinates are returned exactly, as real numbers. For voxel indices, round them into values:

```
>>> voxels.from_physical([0, 15, 0]).astype(int)
array([2, 2, 0])
```

Multiple coordinates can be given as a 2D array / list of lists:

```
>>> voxels.from_physical([[0, 15, 0], [5, 20, 10]])
array([[ 2. ,  2. , -0.5],
       [ 4.5,  4.5,  4.5]])
```

to_physical(*indices*, *corner=False*)

Transform *indices* from voxel indices to physical dimensions. If *corner* = *True*, return the coordinates of the bottom left corner of each voxel; otherwise, use the voxel centres.

Examples

Create a simple *konigcell.Voxels* grid, spanning [-5, 5] mm in the X-dimension, [10, 20] mm in the Y-dimension and [0, 10] in Z:

```
>>> import konigcell as kc
>>> voxels = kc.Voxels.zeros((5, 5, 5), xlim=[-5, 5], ylim=[10, 20],
                             zlim=[0, 10])
>>> voxels
Voxels
------
xlim = [-5.  5.]
ylim = [10. 20.]
zlim = [10. 20.]
voxels =
  (shape: (5, 5, 5))
  [[[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]
   [[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]
   ...
   [[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]
   [[0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]
    ...
    [0. 0. ... 0. 0.]
    [0. 0. ... 0. 0.]]]
attrs = {}
```

```
>>> voxels.voxel_size
array([2., 2., 2.])
```

Transform physical coordinates to voxel coordinates:

```
>>> voxels.to_physical([0, 0, 0], corner = True)
array([-5., 10., 0.])
```

```
>>> voxels.to_physical([0, 0, 0])
array([-4., 11., 1.])
```

Multiple coordinates can be given as a 2D array / list of lists:

```
>>> voxels.to_physical([[0, 0, 0], [4, 4, 3]])
array([[-4., 11.,  1.],
       [ 4., 19.,  7.]])
```

`plot`(*condition=<function Voxels.<lambda>>*, *ax=None*, *alt_axes=False*)

Plot the voxels in this class using Matplotlib.

This plots the centres of all voxels encapsulated in a *pept.Voxels* instance, colour-coding the voxel value.

The *condition* parameter is a filtering function that should return a boolean mask (i.e. it is the result of a condition evaluation). For example *lambda x: x > 0* selects all voxels that have a value larger than 0.

Parameters

condition
[`function`, `default` *lambda voxel_data: voxel_data > 0*] The filtering function applied to the voxel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all voxels that should be plotted. The default, *lambda x: x > 0* selects all voxels which have a value larger than 0.

ax
[`mpl_toolkits.mplot3D.Axes3D` `object`, optional] The 3D matplotlib-based axis for plotting. If undefined, new Matplotlib figure and axis objects are created.

alt_axes
[bool, `default` `False`] If *True*, plot using the alternative PEPT-style axes convention: z is horizontal, y points upwards. Because Matplotlib cannot swap axes, this is achieved by swapping the parameters in the plotting call (i.e. *plt.plot(x, y, z)* -> *plt.plot(z, x, y)*).

Returns

fig, ax
Matplotlib figure and axes objects.

Notes

Plotting all points is very computationally-expensive for matplotlib. It is recommended to only plot a couple of samples at a time, or use Plotly, which is faster.

Examples

Voxellise an array of lines and add them to a *PlotlyGrapher* instance:

```
>>> import konigcell as kc
>>>
>>> lines = np.array(...)           # shape (N, M >= 7)
>>> number_of_voxels = [10, 10, 10]
>>> voxels = kc.Voxels(lines, number_of_voxels)
```

```
>>> fig, ax = voxels.plot()
>>> fig.show()
```

`plot_volumetric`(*condition=<function Voxels.<lambda>>*, *mode='box'*, *colorscale='magma'*)

Create a volumetric PyVista plot - check the *mode* argument for the available types.

Parameters

condition
[`function`, `default` *lambda voxel_data: voxel_data > 0*] The filtering function applied to the voxel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all voxels that should be plotted. The default, *lambda x: x > 0* selects all voxels which have a value larger than 0.

mode
["box", "plane", "slice"] Use a VTK clip box, clip plane or clip slice.

colorscale
[`str`, `default` "magma"] The PyVista colorscale to use.

Returns

`pyvista.Plotter`
A PyVista Figure object that can be `.show()`.

`vtk`(*condition=<function Voxels.<lambda>>*)

Return a PyVista VTK object, exposing all VTK functionality.

Parameters

condition
[`function`, `default` *lambda voxel_data: voxel_data > 0*] The filtering function applied to the voxel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all voxels that should be plotted. The default, *lambda x: x > 0* selects all voxels which have a value larger than 0.

Returns

`pyvista.UniformGrid`
A VTK UniformGrid object.

`cube_trace`(*index*, *color=None*, *opacity=0.4*, *colorbar=True*, *colorscale='magma'*)

Get the Plotly *Mesh3d* trace for a single voxel at *index*.

This renders the voxel as a cube. While visually accurate, this method is *very* computationally intensive - only use it for fewer than 100 cubes. For more voxels, use the *voxels_trace* method.

Parameters

index
[(3,) `tuple`] The voxel indices, given as a 3-tuple.

color
[`str` or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[`float`, `default` 0.4] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[bool, `default` `True`] If set to True, will color-code the voxel values. Is overridden if *color* is set.

colorscale
[`str`, `default` "Magma"] The Plotly scheme for color-coding the voxel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

Raises

`ValueError`
If *index* does not contain exactly three values.

Notes

If you want to render a small number of voxels as cubes using Plotly, use the *cubes_traces* method, which creates a list of individual cubes for all voxels, using this function.

`cubes_traces`(*condition=<function Voxels.<lambda>>*, *color=None*, *opacity=0.4*, *colorbar=True*, *colorscale='magma'*)

Get a list of Plotly *Mesh3d* traces for all voxels selected by the *condition* filtering function.

The *condition* parameter is a filtering function that should return a boolean mask (i.e. it is the result of a condition evaluation). For example *lambda x: x > 0* selects all voxels that have a value larger than 0.

This renders each voxel as individual cubes. While visually accurate, this method is *very* computationally intensive - only use it for fewer than 100 cubes. For more voxels, use the *voxels_trace* method.

Parameters

condition
[`function`, `default` *lambda voxels: voxels > 0*] The filtering function applied to the voxel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all voxels that should be plotted. The default, *lambda x: x > 0* selects all voxels which have a value larger than 0.

color
[`str` or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[`float`, `default` 0.4] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[bool, `default` `True`] If set to True, will color-code the voxel values. Is overridden if *color* is set.

colorscale
[str, default "magma"] The Plotly scheme for color-coding the voxel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

Examples

Plot a *konigcell.Voxels* on a *plotly.graph_objs.Figure*.

```
>>> import konigcell as kc
>>> voxels = ...
```

```
>>> import plotly.graph_objs as go
>>>
>>> fig = go.Figure()
>>> fig.add_traces(voxels.cubes_traces())  # small number of voxels
>>> fig.show()
```

scatter_trace(*condition=<function Voxels.<lambda>>*, *size=4*, *color=None*, *opacity=0.4*, *colorbar=True*, *colorscale='Magma'*, *colorbar_title=None*)

Create and return a trace for all the voxels in this class, with possible filtering.

Creates a *plotly.graph_objects.Scatter3d* object for the centres of all voxels encapsulated in a *pept.Voxels* instance, colour-coding the voxel value.

The *condition* parameter is a filtering function that should return a boolean mask (i.e. it is the result of a condition evaluation). For example *lambda x: x > 0* selects all voxels that have a value larger than 0.

Parameters

condition
[function, default *lambda voxel_data: voxel_data > 0*] The filtering function applied to the voxel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all voxels that should be plotted. The default, *lambda x: x > 0* selects all voxels which have a value larger than 0.

size
[float, default 4] The size of the plotted voxel points. Note that due to the large number of voxels in typical applications, the *voxel centres* are plotted as square points, which provides an easy to understand image that is also fast and responsive.

color
[str or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[float, default 0.4] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[bool, default True] If set to True, will color-code the voxel values. Is overridden if *color* is set.

colorscale
[str, default "Magma"] The Plotly scheme for color-coding the voxel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at

plotly.com/python/builtin-colorscales/. Only has an effect if *colorbar = True* and *color* is not set.

colorbar_title
[`str`, optional] If set, the colorbar will have this title above it.

Examples

Voxellise an array of lines and add them to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher()
>>> lines = np.array(...)           # shape (N, M >= 7)
>>> number_of_voxels = [10, 10, 10]
>>> voxels = pept.Voxels.from_lines(lines, number_of_voxels)
>>> grapher.add_lines(lines)
>>> grapher.add_trace(voxels.voxels_trace())
>>> grapher.show()
```

`heatmap_trace`(*ix=None*, *iy=None*, *iz=None*, *width=0*, *colorscale='Magma'*, *transpose=True*)

Create and return a Plotly *Heatmap* trace of a 2D slice through the voxels.

The orientation of the slice is defined by the input *ix* (for the YZ plane), *iy* (XZ), *iz* (XY) parameters - which correspond to the voxel index in the x-, y-, and z-dimension. Importantly, at least one of them must be defined.

Parameters

ix
[`int`, optional] The index along the x-axis of the voxels at which a YZ slice is to be taken. One of *ix*, *iy* or *iz* must be defined.

iy
[`int`, optional] The index along the y-axis of the voxels at which a XZ slice is to be taken. One of *ix*, *iy* or *iz* must be defined.

iz
[`int`, optional] The index along the z-axis of the voxels at which a XY slice is to be taken. One of *ix*, *iy* or *iz* must be defined.

width
[`int`, `default` 0] The number of voxel layers around the given slice index to collapse (i.e. accumulate) onto the heatmap.

colorscale
[`str`, `default` "Magma"] The Plotly scheme for color-coding the voxel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

transpose
[bool, `default True`] Transpose the heatmap (i.e. flip it across its diagonal).

Raises

`ValueError`
If neither of *ix*, *iy* or *iz* was defined.

Examples

Voxellise an array of lines and add them to a *PlotlyGrapher* instance:

```
>>> lines = np.array(...)           # shape (N, M >= 7)
>>> number_of_voxels = [10, 10, 10]
>>> voxels = pept.Voxels(lines, number_of_voxels)
```

```
>>> import plotly.graph_objs as go
>>> fig = go.Figure()
>>> fig.add_trace(voxels.heatmap_trace())
>>> fig.show()
```

`add_lines`(*lines*, *verbose=False*)

Voxellise a sample of lines, adding 1 to each voxel traversed, for each line in the sample.

Parameters

lines
[(`M`, `N` >= 7) `numpy.ndarray`] The sample of 3D lines to voxellise. Each line is defined as a timestamp followed by two 3D points, such that the data columns are *[time, x1, y1, z1, x2, y2, z2, . . .]*. Note that there can be extra data columns which will be ignored.

verbose
[`bool`, `default False`] Time the voxel traversal and print it to the terminal.

Raises

`ValueError`
If *lines* has fewer than 7 columns.

`static from_lines`(*lines*, *number_of_voxels*, *xlim=None*, *ylim=None*, *zlim=None*, *verbose=True*)

Create a voxel space and traverse / voxellise a given sample of *lines*. The *number_of_voxels* in each dimension must be defined. If the voxel space boundaries *xlim*, *ylim* or *zlim* are not defined, they are inferred as the boundaries of the *lines*.

Parameters

lines
[(`M`, N>=7) `numpy.ndarray` or *`pept.LineData`*] The lines that will be voxellised, each defined by a timestamp and two 3D points, so that the data columns are [time, x1, y1, z1, x2, y2, z2, . . .]. Note that extra columns are ignored.

number_of_voxels
[(3,) `list[int]`] The number of voxels in the x-, y-, and z-dimensions, respectively.

xlim
[(2,) `list[float]`, optional] The lower and upper boundaries of the voxellised volume in the x-dimension, formatted as [x_min, x_max]. If undefined, it is inferred from the boundaries of *lines*.

ylim
[(2,) `list[float]`, optional] The lower and upper boundaries of the voxellised volume in the y-dimension, formatted as [y_min, y_max]. If undefined, it is inferred from the boundaries of *lines*.

zlim
[(2,) `list[float]`, optional] The lower and upper boundaries of the voxellised volume in the z-dimension, formatted as [z_min, z_max]. If undefined, it is inferred from the boundaries of *lines*.

Returns

pept.Voxels
A new *Voxels* object with the voxels through which the lines were traversed.

Raises

`ValueError`
If the input *lines* does not have the shape (M, N>=7). If the *number_of_voxels* is not a 1D list with exactly 3 elements, or any dimension has fewer than 2 voxels.

pept.Pipeline

class pept.**Pipeline**(*transformers*)

Bases: *PEPTObject*

A PEPT processing pipeline, chaining multiple *Filter* and *Reducer* for efficient, parallel execution.

After a pipeline is constructed, the *fit(samples)* method can be called, which will apply the chain of filters and reducers on the samples of data.

A filter is simply a transformation applied to a sample (e.g. *Voxelliser* on a single sample of *LineData*). A reducer is a transformation applied to a list of *all* samples (e.g. *Stack* on all samples of *PointData*).

Note that only filters can be applied in parallel, but the great advantage of a *Pipeline* is that it significantly reduces the amount of data copying and intermediate results' storage. Reducers will require collecting all results.

There are three execution policies at the moment: "sequential" is single-threaded (slower, but easy to debug), "joblib" (very fast on medium datasets due to joblib's caching) and any *concurrent.futures.Executor* subclass (e.g. MPIPoolExecutor for parallel processing on distributed clusters).

Examples

A pipeline can be created in two ways: either by adding (+) multiple transformers together, or explicitly constructing the *Pipeline* class.

The first method is the most straightforward:

```
>>> import pept
```

```
>>> filter1 = pept.tracking.Cutpoints(max_distance = 0.5)
>>> filter2 = pept.tracking.HDBSCAN(true_fraction = 0.1)
>>> reducer = pept.tracking.Stack()
>>> pipeline = filter1 + filter2 + reducer
```

```
>>> print(pipeline)
Pipeline
--------
transformers = [
    Cutpoints(append_indices = False, cutoffs = None, max_distance = 0.5)
    HDBSCAN(clusterer = HDBSCAN(), max_tracers = 1, true_fraction = 0.1)
    Stack(overlap = None, sample_size = None)
]
```

```
>>> lors = pept.LineData(...)         # Some samples of lines
>>> points = pipeline.fit(lors)
```

The chain of filters can also be applied to a single sample:

```
>>> point = pipeline.fit_sample(lors[0])
```

The pipeline's *fit* method allows specifying an execution policy:

```
>>> points = pipeline.fit(lors, executor = "sequential")
>>> points = pipeline.fit(lors, executor = "joblib")
```

```
>>> from mpi4py.futures import MPIPoolExecutor
>>> points = pipeline.fit(lors, executor = MPIPoolExecutor)
```

The *pept.Pipeline* constructor can also be called directly, which allows the enumeration of filters:

```
>>> pipeline = pept.Pipeline([filter1, filter2, reducer])
```

Adding new filters is very easy:

```
>>> pipeline_extra = pipeline + filter2
```

Attributes

`transformers`
: [`list`[`pept.base.Filter` or `pept.base.Reducer`]] The list of *Transformer* to be applied; this includes both *Filter* and *Reducer* instances.

`__init__`(*transformers*)

Construct the class from an iterable of `Filter`, `Reducer` and/or other `Pipeline` instances (which will be flattened).

Methods

`__init__`(transformers)	Construct the class from an iterable of `Filter`, `Reducer` and/or other `Pipeline` instances (which will be flattened).
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples[, executor, max_workers, verbose])	Apply all transformers defined to all *samples*.
`fit_sample`(sample)	Apply all transformers - consecutively - to a single sample of data.
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`optimise`(lines[, max_evals, executor, ...])	
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.
`steps`()	Return the order of processing steps to apply as a list where all consecutive sequences of filters are collapsed into tuples.

Attributes

filters	Only the *Filter* instances from the *transformers*.
reducers	Only the *Reducer* instances from the *transformers*.
transformers	The list of *Transformer* to be applied; this includes both *Filter* and *Reducer* instances.

property filters

Only the *Filter* instances from the *transformers*. They can be applied in parallel.

property reducers

Only the *Reducer* instances from the *transformers*. They require collecting all parallel results.

property transformers

The list of *Transformer* to be applied; this includes both *Filter* and *Reducer* instances.

fit_sample(*sample*)

Apply all transformers - consecutively - to a single sample of data. The output type is simply what the transformers return.

fit(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply all transformers defined to all *samples*. Filters are applied according to the *executor* policy (e.g. parallel via "joblib"), while reducers are applied on a single thread.

Parameters

samples
[`Iterable`] An iterable (e.g. list, tuple, LineData, list[PointData]), whose elements will be passed through the pipeline.

executor
["sequential", "joblib", `or` *concurrent.futures.Executor* `subclass`, `default` "joblib"] The execution policy controlling how the chain of filters are applied to each sample in *samples*; "sequential" is single threaded (slow, but easy to debug), "joblib" is multi-threaded (very fast due to joblib's caching). Alternatively, a *concurrent.futures.Executor* subclass can be used (e.g. *MPIPoolExecutor* for distributed computing on clusters).

max_workers
[`int`, optional] The maximum number of workers to use for parallel executors. If *None* (default), the maximum number of CPUs are used.

verbose
[bool, `default True`] If True, show extra information during processing, e.g. loading bars.

steps()

Return the order of processing steps to apply as a list where all consecutive sequences of filters are collapsed into tuples.

E.g. [F, F, R, F, R, R, F, F, F] -> [(F, F), R, (F), R, R, (F, F, F)].

optimise(*lines*, *max_evals=200*, *executor='joblib'*, *max_workers=None*, *verbose=True*, ***free_parameters*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

5.3.3 Auxilliaries

pept.TimeWindow(window)	Define a *sample_size* as a fixed time window / slice.
pept.AdaptiveWindow(window[, max_elems])	Define a *sample_size* as a time window with a maximum limit of elements.

pept.TimeWindow

class pept.**TimeWindow**(*window: float*)

Bases: object

Define a *sample_size* as a fixed time window / slice. You can use this as a direct replacement of the *sample_size* and *overlap*.

```
points = pept.PointData(sample_size = pept.TimeWindow(5.5))
```

__init__(*window: float*) → None

Methods

__init__(window)	

Attributes

window	

window: float

pept.AdaptiveWindow

class pept.**AdaptiveWindow**(*window: float*, *max_elems: int = 9223372036854775807*)

Bases: object

Define a *sample_size* as a time window with a maximum limit of elements. All samples with more than *max_elems* elements will be shortened.

You can use this as a direct replacement of the *sample_size* and *overlap*.

```
points = pept.PointData(sample_size = pept.AdaptiveWindow(5.5, 200))
points.overlap = AdaptiveWindow(2.)
```

The adaptive time window approach combines the advantages of fixed sample sizes and time windowing:

- Time windows are robust to tracers moving in and out of the field of view, as they simply ignore the time slices where almost no LoRs are recorded.
- Fixed sample sizes effectively adapt their spatio-temporal resolution, allowing for higher accuracy when tracers are passing through more active scanner regions.

All samples with more than *ideal_elems* are shortened, such that time windows are shrinked when the tracer activity permits. There exists an ideal time window such that most samples will have roughly *ideal_elems*, with a few higher activity ones that are shortened; `OptimizeWindow` finds this ideal time window for `pept.AdaptiveWindow`.

New in pept-0.5.1

`__init__`(*window: float*, *max_elems: int = 9223372036854775807*)

Methods

`__init__`(window[, max_elems])	

Base / Abstract Classes (`pept.base`)

`pept.base.PEPTObject`()	Base class for all PEPT-oriented objects.
`pept.base.IterableSamples`(data[, ...])	An class for iterating through an array (or array-like) in samples with potential overlap.
`pept.base.Transformer`()	Base class for PEPT filters (transforming a sample into another) and reducers (transforming a list of samples).
`pept.base.Filter`()	Abstract class from which PEPT filters inherit.
`pept.base.Reducer`()	Abstract class from which PEPT reducers inherit.
`pept.base.PointDataFilter`()	An abstract class that defines a filter for samples of *pept.PointData*.
`pept.base.LineDataFilter`()	An abstract class that defines a filter for samples of *pept.LineData*.
`pept.base.VoxelsFilter`()	An abstract class that defines a filter for samples of *pept.Voxels*.

pept.base.PEPTObject

class `pept.base.`**`PEPTObject`**

Bases: `object`

Base class for all PEPT-oriented objects.

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.IterableSamples

class `pept.base.`**`IterableSamples`**(*data*, *sample_size=None*, *overlap=None*, *columns=[]*, ***kwargs*)

Bases: *`PEPTObject`*, `Collection`

An class for iterating through an array (or array-like) in samples with potential overlap.

This class can be used to access samples of data of an adaptive `sample_size` and `overlap` without requiring additional storage.

The samples from the underlying data can be accessed using both indexing (`samples[0]`) and iteration (`for sample in samples:  ...`).

Particular cases:

1. If sample_size == 0, all data_samples is returned as one single sample.
2. If overlap >= sample_size, an error is raised.
3. If overlap < 0, lines are skipped between samples.

Raises

`ValueError`
If *overlap* >= *sample_size* unless *sample_size* is 0. Overlap must be smaller than *sample_size*. Note that it can also be negative.

See also:

`pept.LineData`
Encapsulate LoRs for ease of iteration and plotting.

`pept.PointData`
Encapsulate points for ease of iteration and plotting.

Attributes

data
[iterable `that supports slicing`] An iterable (e.g. numpy array) that supports slicing syntax (data[5:7]) storing the data that will be iterated over in samples.

sample_size
[`int`] The number of rows in *data* to be returned in a single sample. A *sample_size* of 0 yields all the data as a single sample.

overlap
[`int`] The number of overlapping rows from *data* between two consecutive samples. An overlap of 0 implies consecutive samples, while an overlap of (*sample_size* - 1) means incrementing the samples by one. A negative overlap implies skipping values between samples.

`__init__`(*data*, *sample_size=None*, *overlap=None*, *columns=[]*, ***kwargs*)

IterableSamples class constructor.

Parameters

data
[iterable] The data that will be iterated over in samples; most commonly a NumPy array.

sample_size
[`int` or `Iterable[Int]`, optional] The number of rows in *data* to be returned in a single sample. A *sample_size* of 0 yields all the data as a single sample.

overlap
[int, optional] The number of overlapping rows from *data* between two consecutive samples. An overlap of 0 implies consecutive samples, while an overlap of (*sample_size* - 1) means incrementing the samples by one. A negative overlap implies skipping values between samples.

Methods

`__init__`(data[, sample_size, overlap, columns])	*IterableSamples* class constructor.
`copy`([deep, data, extra, hidden])	Construct a similar object, optionally with different *data*.
`extra_attrs`()	
`hidden_attrs`()	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

`attrs`
`columns`
`data`
`overlap`
`sample_size`
`samples_indices`

property data

property columns

property attrs

extra_attrs()

hidden_attrs()

property samples_indices

property sample_size

property overlap

copy(*deep=True*, *data=None*, *extra=True*, *hidden=True*, ***attrs*)

Construct a similar object, optionally with different *data*. If *extra*, extra attributes are propagated; same for *hidden*.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.Transformer

class pept.base.**Transformer**

Bases: `ABC`, *PEPTObject*

Base class for PEPT filters (transforming a sample into another) and reducers (transforming a list of samples).

You should only need to subclass *Filter* and *Reducer* (or even, better, their more specialised subclasses, e.g. *LineDataFilter*).

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.Filter

class pept.base.**Filter**

Bases: *Transformer*

Abstract class from which PEPT filters inherit. You only need to define a method *def fit_sample(self, sample)*, which processes a *single* sample.

If you define a filter on *LineData*, you should subclass *LineDataFilter*. Same goes for *PointData* with *PointDataFilter*.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(sample)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

abstract fit_sample(*sample*)

fit(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.Reducer

class pept.base.**Reducer**

Bases: *Transformer*

Abstract class from which PEPT reducers inherit. You only need to define a method *def fit(self, samples)*, which processes an *iterable* of samples (most commonly a *LineData* or *PointData*).

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

abstract fit(*samples*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` `subclass` `instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.PointDataFilter

`class pept.base.`**`PointDataFilter`**

Bases: *Filter*

An abstract class that defines a filter for samples of *pept.PointData*.

An implementor must define the method *def fit_sample(self, sample)*.

A default *fit* method is provided for convenience, calling *fit_sample* on each sample from an iterable according to a given execution policy (e.g. "sequential", "joblib", or *concurrent.futures.Executor* subclasses, such as *ProcessPoolExecutor* or *MPIPoolExecutor*).

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(point_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(sample)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit(*point_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

abstract fit_sample(*sample*)

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.LineDataFilter

class `pept.base.`**`LineDataFilter`**

Bases: `Filter`

An abstract class that defines a filter for samples of *pept.LineData*.

An implementor must define the method *def fit_sample(self, sample)*.

A default *fit* method is provided for convenience, calling *fit_sample* on each sample from an iterable according to a given execution policy (e.g. "sequential", "joblib", or *concurrent.futures.Executor* subclasses, such as *ProcessPoolExecutor* or *MPIPoolExecutor*).

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

abstract `fit_sample`(*sample*)

static `load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.base.VoxelsFilter

class pept.base.**VoxelsFilter**

Bases: *Filter*

An abstract class that defines a filter for samples of *pept.Voxels*.

An implementor must define the method *def fit_sample(self, sample)*.

A default *fit* method is provided for convenience, calling *fit_sample* on each sample from an iterable according to a given execution policy (e.g. “sequential”, “joblib”, or *concurrent.futures.Executor* subclasses, such as *ProcessPoolExecutor* or *MPIPoolExecutor*).

__init__(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(voxels[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit(*voxels*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

abstract **fit_sample**(*sample*)

static **load**(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Initialising Scanner Data (`pept.scanners`)

Convert data from different PET / PEPT scanner geometries and data formats into the common base classes.

The PEPT base classes *PointData*, *LineData*, and *VoxelData* are abstractions over the type of data that may be encountered in the context of PEPT (e.g. LoRs are *LineData*, trajectory points are *PointData*). Once the raw data is transformed into the common formats, any tracking, analysis or visualisation algorithm in the *pept* package can be used interchangeably.

The *pept.scanners* subpackage provides modules for transforming the raw data from different PET / PEPT scanner geometries (parallel screens, modular cameras, etc.) and data formats (binary, ASCII, etc.) into the common base classes.

If you'd like to integrate another scanner geometry or raw data format into this package, you can check out the *pept.scanners.parallel_screens* function as an example. This usually only involves writing a single function by hand; then all functionality from *LineData* will be available to your new data format, for free.

`pept.scanners.adac_forte`(filepath[, ...])	Initialise PEPT lines of response (LoRs) from a binary file outputted by the ADAC Forte parallel screen detector list mode (common file extension ".da01").
`pept.scanners.parallel_screens`(...[, ...])	Initialise PEPT LoRs for parallel screens PET/PEPT detectors from an input CSV file or array.
`pept.scanners.ADACGeometricEfficiency`(separation)	Compute the geometric efficiency of a parallel screens PEPT detector at different 3D coordinates using Antonio Guida's formula [1].
`pept.scanners.modular_camera`(data_file[, ...])	Initialise PEPT LoRs from the modular camera DAQ.

pept.scanners.adac_forte

`pept.scanners.`**`adac_forte`**(*filepath*, *sample_size=None*, *overlap=None*, *verbose=True*)

Initialise PEPT lines of response (LoRs) from a binary file outputted by the ADAC Forte parallel screen detector list mode (common file extension ".da01").

Parameters

filepath
[`str`] The path to a ADAC Forte-generated binary file from which the LoRs will be read into the *LineData* format. If you have multiple files, use a wildcard (*) after their common substring to concatenate them, e.g. "DS1.da*" will add ["DS1.da01", "DS1.da02", "DS1.da02_02"].

sample_size
[int, `default` 0] An *int* that defines the number of lines that should be returned when iterating over *lines*. A *sample_size* of 0 yields all the data as one single sample. A good starting value would be 200 times the maximum number of tracers that would be tracked.

overlap
[int, `default` 0] An *int* that defines the overlap between two consecutive samples that are returned when iterating over *lines*. An overlap of 0 implies consecutive samples, while an overlap of (*sample_size* - 1) means incrementing the samples by one. A negative overlap means skipping values between samples. An error is raised if *overlap* is larger than or equal to *sample_size*.

verbose
[bool, `default` `True`] An option that enables printing the time taken for the initialisation of an instance of the class. Useful when reading large files (10gb files for PEPT data is not unheard of).

Returns

`LineData`
The initialised LoRs.

Raises

`FileNotFoundError`
If the input *filepath* does not exist.

`ValueError`
If *overlap* >= *sample_size*. Overlap has to be smaller than *sample_size*. Note that it can also be negative.

See also:

`pept.LineData`
Encapsulate LoRs for ease of iteration and plotting.

`pept.PointData`
Encapsulate points for ease of iteration and plotting.

`pept.read_csv`
Fast CSV file reading into numpy arrays.

`PlotlyGrapher`
Easy, publication-ready plotting of PEPT-oriented data.

Examples

Initialise a *ParallelScreens* array for three LoRs on a parallel screens PEPT scanner (i.e. each line is defined by **two** points each) with a head separation of 500 mm:

```
>>> lors = pept.scanners.adac_forte("binary_data_adac.da01")
Initialised the PEPT data in 0.011 s.
```

```
>>> lors
LineData
--------
sample_size = 0
overlap =     0
```

(continues on next page)

(continued from previous page)

```
samples =     1
lines =
  [[0.00000000e+00 1.62250000e+02 3.60490000e+02 ... 4.14770000e+02
    3.77010000e+02 3.10000000e+02]
   [4.19512195e-01 2.05910000e+02 2.68450000e+02 ... 3.51640000e+02
    2.95000000e+02 3.10000000e+02]
   [8.39024390e-01 3.16830000e+02 1.26260000e+02 ... 2.74350000e+02
    3.95300000e+02 3.10000000e+02]
   ...
   [1.98255892e+04 2.64320000e+02 2.43080000e+02 ... 2.25970000e+02
    4.01200000e+02 3.10000000e+02]
   [1.98263928e+04 3.19780000e+02 3.38660000e+02 ... 2.75530000e+02
    5.19200000e+02 3.10000000e+02]
   [1.98271964e+04 2.41310000e+02 4.15360000e+02 ... 2.91460000e+02
    4.63150000e+02 3.10000000e+02]]
lines.shape = (32526, 7)
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
```

pept.scanners.parallel_screens

`pept.scanners.parallel_screens`(*filepath_or_array*, *screen_separation*, *sample_size=None*, *overlap=None*, *verbose=True*, ***kwargs*)

Initialise PEPT LoRs for parallel screens PET/PEPT detectors from an input CSV file or array.

The expected data columns in the file are `[time, x1, y1, x2, y2]`. This is automatically transformed into the standard *Lines* format with columns being *[time, x1, y1, z1, x2, y2, z2]*, where *z1 = 0* and *z2 = screen_separation*.

ParallelScreens can be initialised with a predefined numpy array of LoRs or read data from a *.csv*.

Parameters

filepath_or_array
[[`str`, `pathlib.Path`, `IO`] `or` `numpy.ndarray` (`N`, 5)] A path to a file to be read from or an array for initialisation. A path is a string with the (absolute or relative) path to the data file or a URL from which the PEPT data will be read. It should include the full file name, along with its extension (.csv, .a01, etc.).

screen_separation
[`float`] The separation (in *mm*) between the two PEPT screens corresponding to the *z* coordinate of the second point defining each line. The attribute *lines*, with columns *[time, x1, y1, z1, x2, y2, z2]*, will have *z1 = 0* and *z2 = screen_separation*.

sample_size
[`int`, `default` 0] An *int* that defines the number of lines that should be returned when iterating over *lines*. A *sample_size* of 0 yields all the data as one single sample. A good starting value would be 200 times the maximum number of tracers that would be tracked.

overlap
[`int`, `default` 0] An *int* that defines the overlap between two consecutive samples that are returned when iterating over *lines*. An overlap of 0 implies consecutive samples, while an overlap of (*sample_size* - 1) means incrementing the samples by one. A negative overlap means skipping values between samples. An error is raised if *overlap* is larger than or equal to *sample_size*.

verbose
[bool, `default True`] An option that enables printing the time taken for the initialisation of an instance of the class. Useful when reading large files (10gb files for PEPT data is not unheard of).

****kwargs**
[`other keyword arguments`] Other keyword arguments to be passed to *pept.read_csv*, e.g. "skiprows" or "max_rows". See the *pept.read_csv* documentation for other arguments.

Returns

`LineData`
The initialised LoRs.

Raises

`ValueError`
If *overlap* >= *sample_size*. Overlap has to be smaller than *sample_size*. Note that it can also be negative.

`ValueError`
If the data file does not have the (N, M >= 5) shape.

See also:

`pept.LineData`
Encapsulate LoRs for ease of iteration and plotting.

`pept.PointData`
Encapsulate points for ease of iteration and plotting.

`pept.read_csv`
Fast CSV file reading into numpy arrays.

`PlotlyGrapher`
Easy, publication-ready plotting of PEPT-oriented data.

Examples

Initialise a *LineData* array for three LoRs on a parallel screens PEPT scanner (i.e. each line is defined by **two** points each) with a head separation of 500 mm:

```
>>> lors_raw = np.array([
>>>     [2, 100, 150, 200, 250],
>>>     [4, 350, 250, 100, 150],
>>>     [6, 450, 350, 250, 200]
>>> ])
```

```
>>> screen_separation = 500
>>> lors = pept.scanners.parallel_screens(lors_raw, screen_separation)
Initialised PEPT data in 0.001 s.
```

```
>>> lors
LineData
--------
sample_size = 0
overlap =     0
```

(continues on next page)

(continued from previous page)

```
samples =     1
lines =
  [[  2. 100. 150.   0. 200. 250. 500.]
   [  4. 350. 250.   0. 100. 150. 500.]
   [  6. 450. 350.   0. 250. 200. 500.]]
lines.shape = (3, 7)
columns = ['t', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2']
```

pept.scanners.ADACGeometricEfficiency

class pept.scanners.**ADACGeometricEfficiency**(*separation*, *xlim=[111.78, 491.78]*, *ylim=[46.78, 556.78]*)

Bases: *PEPTObject*

Compute the geometric efficiency of a parallel screens PEPT detector at different 3D coordinates using Antonio Guida's formula [1].

The default *xlim* and *ylim* values represent the active detector area of the ADAC Forte scanner used at the University of Birmingham, but can be changed to any parallel screens detector active area range.

This class assumes PEPT coordinates, with the Y and Z axes being swapped, such that Y points upwards and Z is perpendicular to the two detectors.

References

[1]

Examples

Simply instantiate the class with the head separation, then 'call' it with the (x, y, z) coordinates of the point at which to evaluate the geometric efficiency:

```
>>> import pept
>>> separation = 500
>>> geom = pept.scanners.ADACGeometricEfficiency(separation)
>>> eg = geom(250, 250, 250)
```

Alternatively, the separation may be specified using the both the starting and ending limits:

```
>>> separation = [-10, 510]
>>> geom = pept.scanners.ADACGeometricEfficiency(separation)
>>> eg = geom(250, 250, 250)
```

You can evaluate multiple points by using a list / array of values:

```
>>> geom([250, 260], 250, 250)
array([0.18669302, 0.19730517])
```

Compute the variation in geometric efficiency in the XY plane:

```
>>> separation = 500
>>> geom = pept.scanners.ADACGeometricEfficiency(separation)
>>> # Range of x, y values to evaluate the geometric efficiency at
>>> import numpy as np
>>> x = np.linspace(120, 480, 100)
>>> y = np.linspace(50, 550, 100)
>>> z = 250
```

```
>>> # Evaluate EG on a 2D grid of values at all combinations of x, y
>>> xx, yy = np.meshgrid(x, y)
>>> eg = geom(xx, yy, z)
```

The geometric efficiencies can be visualised using a Plotly heatmap or contour plot:

```
>>> import plotly.graph_objs as go
>>> fig = go.Figure()
>>> fig.add_trace(go.Contour(x = x, y = y, z = eg))
>>> fig.show()
```

For an interactive 3D volumetric / voxel plot, you can use PyVista:

```
>>> # Import necessary libraries; you may need to install PyVista
>>> import numpy as np
>>> import pept
>>> import pyvista as pv
```

```
>>> # Instantiate the ADACGeometricEfficiency class
>>> geom = pept.scanners.ADACGeometricEfficiency(500)
```

```
>>> # Lower and upper corners of the grid over which to compute the GE
>>> lower = np.array([115, 50, 5])
>>> upper = np.array([490, 550, 495])
```

```
>>> # Create 3D meshgrid of values and evaluate the GE at each point
>>> n = 40
>>> x = np.linspace(lower[0], upper[0], n)
>>> y = np.linspace(lower[1], upper[1], n)
>>> z = np.linspace(lower[2], upper[2], n)
>>> xx, yy, zz = np.meshgrid(x, y, z)
>>> eg = geom(xx, yy, zz)
```

```
>>> # Create PyVista grid of values
>>> grid = pv.UniformGrid()
>>> grid.dimensions = np.array(eg.shape) + 1
>>> grid.origin = lower
>>> grid.spacing = (upper - lower) / n
>>> grid.cell_arrays["values"] = eg.flatten(order="F")
```

```
>>> # Create PyVista volumetric / voxel plot with an interactive clipper
>>> p = pv.Plotter()
>>> p.add_mesh_clip_plane(grid)
>>> p.show()
```

Attributes

xlim
[(2,) `np.ndarray`, `default` [111.78, 491.78]] The limits of the active detector area in the *x*-dimension.

ylim
[(2,) `np.ndarray`, `default` [46.78, 556.78]] The limits of the active detector area in the *y*-dimension.

zlim
[(2,) `np.ndarray`] The limits of the active detector area in the *z*-dimension.

`__init__`(*separation*, *xlim=[111.78, 491.78]*, *ylim=[46.78, 556.78]*)

Methods

`__init__`(separation[, xlim, ylim])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`eg`(x, y, z)	Return the geometric efficiency evaluated at a single point (x, y, z) *in PEPT coordinates*, i.e. Y points upwards.
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`eg`(*x*, *y*, *z*)

Return the geometric efficiency evaluated at a single point (x, y, z) *in PEPT coordinates*, i.e. Y points upwards.

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.scanners.modular_camera

`pept.scanners.`**`modular_camera`**(*data_file*, *sample_size=None*, *overlap=None*, *verbose=True*)

Initialise PEPT LoRs from the modular camera DAQ.

Can read data from a *.da_1* file or equivalent. The file must contain the standard datawords from the modular camera output. This will then be automatically transformed into the standard *LineData* format with every row being *[time, x1, y1, z1, x2, y2, z2]*, where the geometry is derived from the C-extension. The current useable geometry is a square layout with 4 stacks for 4 modules, separated by 250 mm.

Parameters

data_file
[`str`] A string with the (absolute or relative) path to the data file from which the PEPT data will be read. It should include the full file name, along with the extension (.da_1)

sample_size
[`int`, optional] An *int* ` that defines the number of lines that should be returned when iterating over *_lines*. A *sample_size* of 0 yields all the data as one single sample. (Default is 200)

overlap
[`int`, optional] An *int* that defines the overlap between two consecutive samples that are returned when iterating over *_lines*. An overlap of 0 means consecutive samples, while an overlap of (*sample_size* - 1) means incrementing the samples by one. A negative overlap

means skipping values between samples. An error is raised if *overlap* is larger than or equal to *sample_size*. (Default is 0)

verbose
[bool, optional] An option that enables printing the time taken for the initialisation of an instance of the class. Useful when reading large files (10gb files for PEPT data is not unheard of). (Default is True)

Returns

`LineData`
The initialised LoRs.

Raises

`ValueError`
If *overlap* >= *sample_size*. Overlap has to be smaller than *sample_size*. Note that it can also be negative.

`ValueError`
If the data file does not have (N, 7) shape.

Tracking Algorithms (`pept.tracking`)

Tracer location, identification and tracking algorithms.

The *pept.tracking* subpackage hosts different tracking algorithms, working with both the base classes, as well as with generic NumPy arrays.

All algorithms here are either `pept.base.Filter` or `pept.base.Reducer` subclasses, implementing the *.fit* and *.fit_sample* methods; here is an example using PEPT-ML:

```
>>> from pept.tracking import *
>>>
>>> cutpoints = Cutpoints(0.5).fit(lines)
>>> clustered = HDBSCAN(0.15).fit(cutpoints)
>>> centres = (SplitLabels() + Centroids() + Stack()).fit(clustered)
```

Once the processing steps have been tuned (see the *Tutorials*), you can chain all filters into a *pept.Pipeline* for efficient, parallel execution:

```
>>> pipeline = (
>>>     Cutpoints(0.5) +
>>>     HDBSCAN(0.15) +
>>>     SplitLabels() + Centroids() + Stack()
>>> )
>>> centres = pipeline.fit(lines)
```

If you would like to implement a PEPT algorithm, all you need to do is to subclass a `pept.base.Filter` and define the method `.fit_sample(sample)` - and you get parallel execution and pipeline chaining for free!

```
>>> import pept
>>>
>>> class NewAlgorithm(pept.base.LineDataFilter):
>>>     def __init__(self, setting1, setting2 = None):
>>>         self.setting1 = setting1
>>>         self.setting2 = setting2
```

(continues on next page)

(continued from previous page)

```
>>>
>>>     def fit_sample(self, sample: pept.LineData):
>>>         processed_points = ...
>>>         return pept.PointData(processed_points)
```

Tracking Optimisation

pept.tracking.Debug([verbose, max_samples])	Print types and statistics about the objects being processed in a `pept.Pipeline`.
pept.tracking.OptimizeWindow(ideal_elems[, ...])	Automatically determine optimum adaptive time window to have an ideal number of elements per sample.

pept.tracking.Debug

class `pept.tracking.`**`Debug`**(*verbose=5*, *max_samples=10*)

Bases: *Reducer*

Print types and statistics about the objects being processed in a `pept.Pipeline`.

Reducer signature:

```
      PointData -> Debug.fit -> PointData
       LineData -> Debug.fit -> LineData
list[PointData] -> Debug.fit -> list[PointData]
 list[LineData] -> Debug.fit -> list[LineData]
     np.ndarray -> Debug.fit -> np.ndarray
            Any -> Debug.fit -> Any
```

This is a reducer, so it will collect all samples processed up to the point of use, print them, and return them unchanged.

New in pept-0.5.1

Examples

A `Debug` is normally added in a `Pipeline`:

```
>>> import pept
>>> import pept.tracking as pt
>>>
>>> pept.Pipeline([
>>>     # First pass of clustering
>>>     pt.Cutpoints(max_distance = 0.2),
>>>     pt.HDBSCAN(true_fraction = 0.15),
>>>     pt.SplitLabels() + pt.Centroids(cluster_size = True, error = True),
>>>
>>>     pt.Debug(),
>>>     pt.Stack(),
>>> ])
```

__init__(*verbose=5*, *max_samples=10*)

Methods

`__init__`([verbose, max_samples])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

fit(*samples*)

pept.tracking.OptimizeWindow

class pept.tracking.**OptimizeWindow**(*ideal_elems*, *overlap=0.0*, *low=0.3*, *high=3*)

Bases: *Reducer*

Automatically determine optimum adaptive time window to have an ideal number of elements per sample.

Reducer signature:

```
      LineData -> OptimizeWindow.fit -> LineData
 list[LineData] -> OptimizeWindow.fit -> LineData
     PointData -> OptimizeWindow.fit -> PointData
list[PointData] -> OptimizeWindow.fit -> PointData
  numpy.ndarray -> OptimizeWindow.fit -> PointData
```

The adaptive time window approach combines the advantages of fixed sample sizes and time windowing:

- Time windows are robust to tracers moving in and out of the field of view, as they simply ignore the time slices where almost no LoRs are recorded.
- Fixed sample sizes effectively adapt their spatio-temporal resolution, allowing for higher accuracy when tracers are passing through more active scanner regions.

All samples with more than *ideal_elems* are shortened, such that time windows are shrinked when the tracer activity permits. There exists an ideal time window such that most samples will have roughly *ideal_elems*, with a few higher activity ones that are shortened; `OptimizeWindow` finds this ideal time window for `pept.AdaptiveWindow`.

New in pept-0.5.1

Examples

Find an adaptive time window that is ideal for about 200 LoRs per sample:

```
>>> import pept
>>> import pept.tracking as pt
>>> lors = pept.LineData(...)
>>> lors = pt.OptimizeWindow(ideal_elems = 200).fit(lors)
```

OptimizeWindow can be used at the start of a pipeline; an optional *overlap* parameter can be used to define an overlap as a ratio to the ideal time window found. For example, if the ideal time window found is 100 ms, an overlap of 0.5 will result in an overlapping time interval of 50 ms:

```
>>> pipeline = pept.Pipeline([
>>>     pt.OptimizeWindow(200, overlap = 0.5),
>>>     pt.BirminghamMethod(0.5),
>>>     pt.Stack(),
>>> ])
```

__init__(*ideal_elems*, *overlap=0.0*, *low=0.3*, *high=3*)

Methods

`__init__`(ideal_elems[, overlap, low, high])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`evaluate`(window)	
`fit`(data)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*data*)

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

evaluate(*window*)

General-Purpose Transformers

pept.tracking.Stack([sample_size, overlap])	Stack iterables - for example a `list[pept.LineData]` into a single `pept.LineData`, a `list[list]` into a flattened `list`.
pept.tracking.SplitLabels([remove_labels, ...])	Split a sample of data into unique `label` values, optionally removing noise and extracting *_lines* attributes.
pept.tracking.SplitAll	alias of *GroupBy*
pept.tracking.GroupBy(column)	Stack all samples and split them into a list according to a named / numeric column index.
pept.tracking.Centroids([error, ...])	Compute the geometric centroids of a list of samples of points.
pept.tracking.LinesCentroids([remove, ...])	Compute the minimum distance point of some `pept.LineData` while iteratively removing a fraction of the furthest lines.
pept.tracking.Condition(*conditions)	Select only data satisfying multiple conditions, given as a string, a function or list thereof; e.g.
pept.tracking.SamplesCondition(*conditions)	Select only *samples* satisfying multiple conditions, given as a string, a function or list thereof; e.g.
pept.tracking.Remove(*columns)	Remove columns (either column names or indices) from *pept.LineData* or *pept.PointData*.
pept.tracking.Swap(*swaps[, inplace])	Swap two columns in a LineData or PointData.

pept.tracking.Stack

class pept.tracking.Stack(*sample_size=None*, *overlap=None*)

Bases: `Reducer`

Stack iterables - for example a `list[pept.LineData]` into a single `pept.LineData`, a `list[list]` into a flattened `list`.

Reducer signature:

```
     list[LineData] -> Stack.fit -> LineData
    list[PointData] -> Stack.fit -> PointData

    list[list[Any]] -> Stack.fit -> list[Any]
list[numpy.ndarray] -> Stack.fit -> numpy.ndarray

              other -> Stack.fit -> other
```

Can optionally set a given *sample_size* and *overlap*. This is useful when collecting a list of processed samples back into a single object.

__init__(*sample_size=None*, *overlap=None*)

Methods

`__init__`([sample_size, overlap])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit(*samples*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.SplitLabels

`class pept.tracking.`**`SplitLabels`**(*remove_labels=True*, *extract_lines=False*, *noise=False*)

Bases: *Filter*

Split a sample of data into unique `label` values, optionally removing noise and extracting *_lines* attributes.

Filter signature:

```
# `extract_lines` = False (default)
 LineData -> SplitLabels.fit_sample -> list[LineData]
PointData -> SplitLabels.fit_sample -> list[PointData]

# `extract_lines` = True and PointData.attrs["_lines"] exists
PointData -> SplitLabels.fit_sample -> list[LineData]
```

The sample of data must have a column named exactly "label". If `remove_label = True` (default), the "label" column is removed.

`__init__`(*remove_labels=True*, *extract_lines=False*, *noise=False*)

Methods

__init__([remove_labels, extract_lines, noise])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(sample)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit_sample(*sample: IterableSamples*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.SplitAll

`pept.tracking.`**`SplitAll`**
alias of *GroupBy*

pept.tracking.GroupBy

class `pept.tracking.`**`GroupBy`**(*column*)
Bases: *Reducer*

Stack all samples and split them into a list according to a named / numeric column index.

Reducer signature:

```
          LineData -> SplitAll.fit -> list[LineData]
    list[LineData] -> SplitAll.fit -> list[LineData]

         PointData -> SplitAll.fit -> list[PointData]
   list[PointData] -> SplitAll.fit -> list[PointData]

     numpy.ndarray -> SplitAll.fit -> list[numpy.ndarray]
list[numpy.ndarray] -> SplitAll.fit -> list[numpy.ndarray]
```

If using a LineData / PointData, you can use a columns name as a string, e.g. `SplitAll("label")` or a number `SplitAll(4)`. If using a NumPy array, only numeric indices are accepted.

`__init__`(*column*)

Methods

`__init__`(column)	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Centroids

class pept.tracking.**Centroids**(*error=False*, *cluster_size=False*, *weight=True*)

Bases: *Filter*

Compute the geometric centroids of a list of samples of points.

Filter signature:

```
      PointData -> Centroids.fit_sample -> PointData
list[PointData] -> Centroids.fit_sample -> PointData
  numpy.ndarray -> Centroids.fit_sample -> PointData
```

This filter can be used right after `pept.tracking.SplitLabels`, e.g.:

```
>>> (SplitLabels() + Centroids()).fit(points)
```

If *error* = *True*, append a measure of error on the computed centroid as the standard deviation in distances from centroid to all points. It is saved in an extra column "error".

If *cluster_size* = *True*, append the number of points used for each centroid in an extra column "cluster_size" - unless *weight* = *True*, in which case it is the sum of weights.

If *weight* = *True* and there is a column "weight" in the PointData, compute weighted centroids and standard deviations (if *error* = *True*) and the sum of weights (if *cluster_size* = *True*). The "weight" column is removed in the output centroid.

__init__(*error=False*, *cluster_size=False*, *weight=True*)

Methods

__init__([error, cluster_size, weight])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(points)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit_sample`(*points*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` `subclass` `instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.LinesCentroids

class pept.tracking.**LinesCentroids**(*remove=0.1*, *iterations=6*)

Bases: *Filter*

Compute the minimum distance point of some `pept.LineData` while iteratively removing a fraction of the furthest lines.

Filter signature:

```
list[LineData] -> LinesCentroids.fit_sample -> PointData
      LineData -> LinesCentroids.fit_sample -> PointData
 numpy.ndarray -> LinesCentroids.fit_sample -> PointData
```

The code below is adapted from the PEPT-EM algorithm developed by Antoine Renaud and Sam Manger.

__init__(*remove=0.1*, *iterations=6*)

Methods

__init__([remove, iterations])	
centroid(lors)	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
distance_matrix(x, lors)	
fit(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(lines)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
predict(lines)	
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

static centroid(*lors*)

static distance_matrix(*x*, *lors*)

predict(*lines*)

fit_sample(*lines*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Condition

class pept.tracking.**Condition**(**conditions*)

Bases: `Filter`

Select only data satisfying multiple conditions, given as a string, a function or list thereof; e.g. `Condition("error < 15")` selects all points whose "error" column value is smaller than 15.

Filter signature:

```
PointData -> Condition.fit_sample -> PointData
 LineData -> Condition.fit_sample -> LineData
```

In the simplest case, a column name is specified, plus a comparison, e.g. `Condition("error < 15, y > 100")`; multiple conditions may be concatenated using a comma.

More complex conditions - where the column name is not the first operand - can be constructed using single quotes, e.g. using NumPy functions in `Condition("np.isfinite('x')")` to filter out NaNs and Infs. Quotes can be used to index columns too: `Condition("'0' < 150")` selects all rows whose first column is smaller than 150.

Generally, you can use any function returning a boolean mask, either as a string of code `Condition("np.isclose('x', 3)")` or a user-defined function receiving a NumPy array `Condition(lambda x: x[:, 0] < 10)`.

Finally, multiple such conditions may be supplied separately: `Condition(lambda x: x[:, -1] > 10, "'t' < 50")`.

__init__(**conditions*)

Methods

`__init__`(*conditions)	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

`conditions`	

property conditions

fit_sample(*sample: IterableSamples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.SamplesCondition

class pept.tracking.**SamplesCondition**(**conditions*)

Bases: *Reducer*

Select only *samples* satisfying multiple conditions, given as a string, a function or list thereof; e.g. `Condition("sample_size > 30")` selects all samples with a sample size larger than 30.

Filter signature:

```
PointData -> SamplesCondition.fit_sample -> PointData
 LineData -> SamplesCondition.fit_sample -> LineData
```

This is different to a *Condition*, which selects individual points; for *SamplesCondition*, each sample will be passed through the conditions.

Conditions can be defined as Python code using the following variables:

- *sample* - this is the full PointData or LineData, e.g. only keep samples with more than 30 points with "len(sample.points) > 30".
- *data* - this is the raw NumPy array of data wrapped by a PointData or LineData, e.g. only keep samples which have all X coordinates beyond 100 with *SamplesCondition("np.all(data[:, 1] > 100)")*.
- *sample_size* - this is a shorthand for the number of data points, e.g. only keep samples with more than 30 points with "sample_size > 30".

Conditions can also be Python functions:

```
>>> def high_velocity_filter(sample):
>>>     return np.all(sample["v"] > 5)
```

```
>>> from pept.tracking import SamplesCondition
>>> filtered = SamplesCondition(high_velocity_filter).fit(point_data)
```

__init__(**conditions*)

Methods

__init__(*conditions)	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

conditions

property conditions

fit(*samples*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Remove

class pept.tracking.**Remove**(**columns*)

Bases: *Filter*

Remove columns (either column names or indices) from *pept.LineData* or *pept.PointData*.

Filter signature:

```
 pept.LineData -> Remove.fit_sample -> pept.LineData
pept.PointData -> Remove.fit_sample -> pept.PointData
```

Examples

To remove a single column named “line_index”:

```
>>> import pept
>>> from pept.tracking import *
>>> points = pept.PointData(...)     # Some dummy data
```

```
>>> rem = Remove("line_index")
>>> points_without = rem.fit_sample(points)
```

Remove all columns starting with “line_index” using a glob operator (*):

```
>>> points_without = Remove("line_index*").fit_sample(points)
```

Remove the first column based on its index:

```
>>> points_without = Remove(0).fit_sample(points)
```

Finally, multiple removals may be chained into a list:

```
>>> points_without = Remove(["line_index*", -1]).fit_sample(points)
```

__init__(**columns*)

Methods

`__init__`(*columns)	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

`columns`

`property columns`

`fit_sample`(*sample:* IterableSamples)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Swap

class pept.tracking.**Swap**(**swaps*, *inplace=True*)

Bases: *Filter*

Swap two columns in a LineData or PointData.

Filter signature:

```
 LineData -> Swap.fit_sample -> LineData
PointData -> Swap.fit_sample -> PointData
```

For example, swap the Y and Z axes: `Swap("y, z").fit_sample(points)`. Add multiple swaps as separate arguments: `Swap("y, z", "label, x")`.

You can also swap columns at numerical indices by single-quoting them: `Swap("'0', '1'")`.

New in pept-0.4.3

__init__(**swaps*, *inplace=True*)

Methods

`__init__`(*swaps[, inplace])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

`swaps`	

property swaps

fit_sample(*sample:* IterableSamples)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file` handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Space Transformers

pept.tracking.Voxelize(number_of_voxels[, ...])	Asynchronously voxelize samples of lines from a *pept.LineData*.
pept.tracking.Interpolate(timestep[, ...])	Interpolate between data points at a fixed sampling rate; useful for Eulerian fields computation.
pept.tracking.Reorient([dimensions, basis, ...])	Rotate a dataset such that it is oriented according to its principal axes.
pept.tracking.OutOfViewFilter([max_time, k])	Remove tracer locations that are sparse *in time* - ie the k-th nearest detection is later than `max_time`.
pept.tracking.RemoveStatic(time_window, ...)	Remove parts of a *PointData* where the tracer remains static.

pept.tracking.Voxelize

class pept.tracking.**Voxelize**(*number_of_voxels*, *xlim=None*, *ylim=None*, *zlim=None*, *set_lims=None*)

Bases: `LineDataFilter`

Asynchronously voxelize samples of lines from a *pept.LineData*.

Filter signature:

```
LineData -> Voxelize.fit_sample -> PointData
```

This filter is much more memory-efficient than voxelizing all samples of LoRs at once - which often overflows the available memory. Most often this is used alongside voxel-based tracking algorithms, e.g. `pept.tracking.FPI`:

```
>>> from pept.tracking import *
>>> pipeline = pept.Pipeline([
>>>     Voxelize((50, 50, 50)),
>>>     FPI(3, 0.4),
>>>     Stack(),
>>> ])
```

Parameters

number_of_voxels
[3-tuple] A tuple-like containing exactly three integers specifying the number of voxels to be used in each dimension.

xlim
[(2,) `list[float]`, optional] The lower and upper boundaries of the voxellised volume in the x-dimension, formatted as [x_min, x_max]. If undefined, it is inferred from the bounding box of each sample of lines.

ylim
[(2,) `list[float]`, optional] The lower and upper boundaries of the voxellised volume in the y-dimension, formatted as [y_min, y_max]. If undefined, it is inferred from the bounding box of each sample of lines.

zlim
[(2,) `list[float]`, optional] The lower and upper boundaries of the voxellised volume in the z-dimension, formatted as [z_min, z_max]. If undefined, it is inferred from the bounding box of each sample of lines.

set_lims
[(N, 7) `numpy.ndarray` or `pept.LineData`, optional] If defined, set the system limits upon creating the class to the bounding box of the lines in *set_lims*.

__init__(*number_of_voxels*, *xlim=None*, *ylim=None*, *zlim=None*, *set_lims=None*)

Methods

`__init__`(number_of_voxels[, xlim, ylim, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample_lines)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.
`set_lims`(lines[, set_xlim, set_ylim, set_zlim])	

Attributes

`number_of_voxels`
`xlim`
`ylim`
`zlim`

set_lims(*lines*, *set_xlim=True*, *set_ylim=True*, *set_zlim=True*)

property number_of_voxels

property xlim

property ylim

property zlim

fit_sample(*sample_lines*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)
Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Interpolate

class `pept.tracking.`**`Interpolate`**(*timestep*, *interpolator=<class 'scipy.interpolate.interpolate.interp1d'>*, ***kwargs*)

Bases: `PointDataFilter`

Interpolate between data points at a fixed sampling rate; useful for Eulerian fields computation.

Filter signature:

```
PointData -> Interpolate.fit_sample -> PointData
```

By default, the linear interpolator *scipy.interpolate.interp1d* is used. You can specify a different interpolator and keyword arguments to send it. E.g. nearest-neighbour interpolation: `Interpolate(1., kind = "nearest")` or cubic interpolation: `Interpolate(1., kind = "cubic")`.

All data columns except timestamps are interpolated.

`__init__`(*timestep*, *interpolator=<class 'scipy.interpolate.interpolate.interp1d'>*, ***kwargs*)

Methods

`__init__`(timestep[, interpolator])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(point_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit_sample`(*sample*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*point_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Reorient

class pept.tracking.**Reorient**(*dimensions='xyz'*, *basis=None*, *origin=None*)

Bases: *Reducer*

Rotate a dataset such that it is oriented according to its principal axes.

Reducer signature:

```
      PointData -> Reorient.fit -> PointData
list[PointData] -> Reorient.fit -> PointData
     np.ndarray -> Reorient.fit -> PointData
```

By default, this reducer reorients the points such that the axis along which it is most spread out (e.g. lengthwise in a pipe) becomes the X-axis. The input argument *dimensions* sets this - the default *"xyz"* can be changed to e.g. *"zyx"* so that the longest data axis becomes the Z-axis.

The reducer also sets three attributes on the returned *PointData*: - *origin*: the origin relative to which the initial data was rotated. - *basis*: the principal components - or change of basis 3x3 matrix. - *eigenvalues*: how spread out the data is in each initial dimension.

If you'd like to reorient a second dataset to the same basis as a first one, set the *basis* and *origin* arguments.

New in pept-0.5.0

Examples

Reorient a dataset by aligning the longest principal component (e.g. lengthwise in a pipe) to the X-axis:

```
>>> import pept.tracking as pt
>>> data = PointData(...)
>>> reoriented = pt.Reorient().fit(data)
```

Reorient it such that the longest principal component (e.g. vertical in a mixer) becomes the Z-axis:

```
>>> reoriented = pt.Reorient("zyx").fit(data)
```

Reorient a second dataset to the same orientation basis as the first one:

```
>>> reoriented2 = pt.Reorient(
>>>     basis = reoriented.attrs["basis"],
>>>     origin = reoriented.attrs["origin"],
>>> ).fit(other_data)
```

`__init__`(*dimensions='xyz'*, *basis=None*, *origin=None*)

Methods

`__init__`([dimensions, basis, origin])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.OutOfViewFilter

class pept.tracking.**OutOfViewFilter**(*max_time=200.0*, *k=5*)

Bases: *Reducer*

Remove tracer locations that are sparse *in time* - ie the `k`-th nearest detection is later than `max_time`.

Reducer signature:

```
      PointData -> OutOfViewFilter.fit -> PointData
list[PointData] -> OutOfViewFilter.fit -> PointData
  numpy.ndarray -> OutOfViewFilter.fit -> PointData
```

This reducer (i.e. stacks all data samples, then processes it) is useful when the tracer goes out of the PEPT scanners and there are a few sparse noisy detections to remove.

New in pept-0.5.1

Examples

Select only tracer locations whose next detection is within 200 ms.

```
>>> import pept
>>> import pept.tracking as pt
>>> trajectories = pept.PointData(...)
>>> # Only keep points whose next detection is within 200 ms
>>> inview = pt.OutOfViewFilter(max_time = 200.).fit(trajectories)
```

`__init__`(*max_time=200.0*, *k=5*)

Methods

`__init__`([max_time, k])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.RemoveStatic

class `pept.tracking.`**`RemoveStatic`**(*time_window*, *max_distance*, *quantile=0.9*)

Bases: *Reducer*

Remove parts of a *PointData* where the tracer remains static.

Reducer signature:

```
      PointData -> OutOfViewFilter.fit -> PointData
list[PointData] -> OutOfViewFilter.fit -> PointData
  numpy.ndarray -> OutOfViewFilter.fit -> PointData
```

If there is a *time_window* in which the tracer does not move more than *max_distance*, it is removed.

The distances moved are computed relative to the average position within each time window; to make the reducer more robust to noise, the given distance *quantile* is compared to *max_distance*.

New in pept-0.5.2

Examples

Given some trajectories from e.g. a long experiment where the particle may have got stuck at some points, we can remove the static windows with:

```python
import pept
import pept.tracking as pt

trajectories = ...

# Remove positions that spent more than 2 seconds without moving more
# than 20 mm
trajectories_nonstatic = RemoveStatic(
    time_window = 2000,
    max_distance = 20,
).fit(trajectories)
```

This reducer, like the rest in *pept.tracking*, can be chained into a pipeline, for example:

```python
import pept
import pept.tracking as pt

pipeline = pept.Pipeline([
    # Remove positions with high errors
    pt.Condition("error < 20"),

    # Remove tracers that got stuck
    pt.RemoveStatic(time_window = 2000, max_distance = 20),

    # Trajectory separation
    pt.Segregate(window = 20, cut_distance = 15),

    # Group each trajectory into its own sample, then stack them
    pt.GroupBy("label"),
    pt.Stack(),
])
```

__init__(*time_window*, *max_distance*, *quantile=0.9*)

Methods

__init__(time_window, max_distance[, quantile])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples*)

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Tracer Locating Algorithms

pept.tracking.BirminghamMethod([fopt, get_used])	The Birmingham Method is an efficient, analytical technique for tracking tracers using the LoRs from PEPT data.
pept.tracking.Cutpoints(max_distance[, ...])	Transform LoRs (a *pept.LineData* instance) into *cutpoints* (a *pept.PointData* instance) for clustering, in parallel.
pept.tracking.Minpoints(num_lines, max_distance)	Transform LoRs (a *pept.LineData* instance) into *minpoints* (a *pept.PointData* instance) for clustering, in parallel.
pept.tracking.HDBSCAN(true_fraction[, ...])	Use HDBSCAN to cluster some `pept.PointData` and append a cluster label to each point.
pept.tracking.FPI([w, r, lld_counts, verbose])	FPI is a modern voxel-based tracer-location algorithm that can reliably work with unknown numbers of tracers in fast and noisy environments.

pept.tracking.BirminghamMethod

class `pept.tracking.`**`BirminghamMethod`**(*fopt=0.5*, *get_used=False*)

Bases: *LineDataFilter*

The Birmingham Method is an efficient, analytical technique for tracking tracers using the LoRs from PEPT data.

Two main methods are provided: *fit_sample* for tracking a single numpy array of LoRs (i.e. a single sample) and *fit* which tracks all the samples encapsulated in a *pept.LineData* class *in parallel*.

For the given *sample* of LoRs (a numpy.ndarray), this function minimises the distance between all of the LoRs, rejecting a fraction of lines that lie furthest away from the calculated distance. The process is repeated iteratively until a specified fraction (*fopt*) of the original subset of LORs remains.

This class is a wrapper around the *birmingham_method* subroutine (implemented in C), providing tools for asynchronously tracking samples of LoRs. It can return *PointData* classes which can be easily manipulated and visualised.

See also:

pept.LineData
Encapsulate LoRs for ease of iteration and plotting.

pept.PointData
Encapsulate points for ease of iteration and plotting.

pept.utilities.read_csv
Fast CSV file reading into numpy arrays.

`PlotlyGrapher`
Easy, publication-ready plotting of PEPT-oriented data.

`pept.scanners.ParallelScreens`
Initialise a *pept.LineData* instance from parallel screens PEPT detectors.

Examples

A typical workflow would involve reading LoRs from a file, instantiating a *BirminghamMethod* class, tracking the tracer locations from the LoRs, and plotting them.

```
>>> import pept
>>> from pept.tracking.birmingham_method import BirminghamMethod
```

```
>>> lors = pept.LineData(...)   # set sample_size and overlap appropriately
>>> bham = BirminghamMethod()
>>> locations = bham.fit(lors)  # this is a `pept.PointData` instance
```

```
>>> grapher = PlotlyGrapher()
>>> grapher.add_points(locations)
>>> grapher.show()
```

Attributes

fopt
[`float`] Floating-point number between 0 and 1, representing the target fraction of LoRs in a sample used to locate a tracer.

get_used
[bool, `default False`] If True, attach an attribute `._lines` to the output PointData containing the sample of LoRs used (+ a column *used*).

`__init__`(*fopt=0.5*, *get_used=False*)

BirminghamMethod class constructor.

fopt
[float, default 0.5] Float number between 0 and 1, representing the fraction of remaining LORs in a sample used to locate the particle.

verbose
[bool, default False] Print extra information when initialising this class.

Methods

`__init__`([fopt, get_used])	*BirminghamMethod* class constructor.
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	Use the Birmingham method to track a tracer location from a numpy array (i.e.
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit_sample`(*sample*)

Use the Birmingham method to track a tracer location from a numpy array (i.e. one sample) of LoRs.

For the given *sample* of LoRs (a numpy.ndarray), this function minimises the distance between all of the LoRs, rejecting a fraction of lines that lie furthest away from the calculated distance. The process is repeated iteratively until a specified fraction (*fopt*) of the original subset of LORs remains.

Parameters

sample
[(N, M>=7) `numpy.ndarray`] The sample of LORs that will be clustered. Each LoR is expressed as a timestamps and a line defined by two points; the data columns are then *[time, x1, y1, z1, x2, y2, z2, extra. . .]*.

Returns

locations
[`numpy.ndarray` or `pept.PointData`] The tracked locations found.

used
[`numpy.ndarray`, optional] If *get_used* is true, then also return a boolean mask of the LoRs used to compute the tracer location - that is, a vector of the same length as *sample*, containing 1 for the rows that were used, and 0 otherwise. [Used for multi-particle tracking, not implemented yet].

Raises

`ValueError`
If *sample* is not a numpy array of shape (N, M), where M >= 7.

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Cutpoints

class pept.tracking.**Cutpoints**(*max_distance*, *cutoffs=None*, *append_indices=False*)

Bases: `LineDataFilter`

Transform LoRs (a *pept.LineData* instance) into *cutpoints* (a *pept.PointData* instance) for clustering, in parallel.

Under typical usage, the *Cutpoints* class is initialised with a *pept.LineData* instance, automatically calculating the cutpoints from the samples of lines. The *Cutpoints* class inherits from *pept.PointData*, such that once the cutpoints have been computed, all the methods from the parent class *pept.PointData* can be used on them (such as visualisation functionality).

For more control over the operations, *pept.tracking.peptml.find_cutpoints* can be used - it receives a generic numpy array of LoRs (one 'sample') and returns a numpy array of cutpoints.

See also:

pept.LineData
Encapsulate LoRs for ease of iteration and plotting.

pept.tracking.HDBSCAN
Efficient, parallel HDBSCAN-based clustering of (cut)points.

pept.read_csv
Fast CSV file reading into numpy arrays.

Examples

Compute the cutpoints for a *LineData* instance between lines that are less than 0.1 apart:

```
>>> line_data = pept.LineData(example_data)
>>> cutpts = peptml.Cutpoints(0.1).fit(line_data)
```

Compute the cutpoints for a single sample:

```
>>> sample = line_data[0]
>>> cutpts_sample = peptml.Cutpoints(0.1).fit_sample(sample)
```

Attributes

max_distance
[`float`] The maximum distance between any two lines for their cutpoint to be considered. A good starting value would be 0.1 mm for small tracers and/or clean data, or 0.2 mm for larger tracers and/or noisy data.

cutoffs
[list-like `of length` 6] A list (or equivalent) of the cutoff distances for every axis, formatted as *[x_min, x_max, y_min, y_max, z_min, z_max]*. Only the cutpoints which fall within these cutoff distances are considered. The default is None, in which case they are automatically computed using *pept.tracking.peptml.get_cutoffs*.

`__init__`(*max_distance*, *cutoffs=None*, *append_indices=False*)
Cutpoints class constructor.

Parameters

max_distance
[`float`] The maximum distance between any two lines for their cutpoint to be considered. A good starting value would be 0.1 mm for small tracers and/or clean data, or 0.5 mm for larger tracers and/or noisy data.

cutoffs
[list-like `of length` 6, optional] A list (or equivalent) of the cutoff distances for every axis, formatted as *[x_min, x_max, y_min, y_max, z_min, z_max]*. Only the cutpoints which fall within these cutoff distances are considered. The default is None, in which case they are automatically computed using *pept.tracking.peptml.get_cutoffs*.

append_indices
[bool, `default False`] If set to *True*, the indices of the individual LoRs that were used to compute each cutpoint are also appended to the returned array.

Raises

`ValueError`
If *cutoffs* is not a one-dimensional array with values formatted as *[min_x, max_x, min_y, max_y, min_z, max_z]*.

Methods

`__init__`(max_distance[, cutoffs, append_indices])	Cutpoints class constructor.
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample_lines)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

`append_indices`
`cutoffs`
`max_distance`

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static **load**(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

property max_distance

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

property cutoffs

property append_indices

fit_sample(*sample_lines*)

pept.tracking.Minpoints

class pept.tracking.**Minpoints**(*num_lines*, *max_distance*, *cutoffs=None*, *append_indices=False*)

Bases: `LineDataFilter`

Transform LoRs (a *pept.LineData* instance) into *minpoints* (a *pept.PointData* instance) for clustering, in parallel.

Given a sample of lines, the minpoints are the minimum distance points (MDPs) for every possible combination of *num_lines* lines that satisfy the following conditions:

1. Are within the *cutoffs*.
2. Are closer to all the constituent LoRs than *max_distance*.

Under typical usage, the *Minpoints* class is initialised with a *pept.LineData* instance, automatically calculating the minpoints from the samples of lines. The *Minpoints* class inherits from *pept.PointData*, such that once the cutpoints have been computed, all the methods from the parent class *pept.PointData* can be used on them (such as visualisation functionality).

For more control over the operations, *pept.tracking.peptml.find_minpoints* can be used - it receives a generic numpy array of LoRs (one 'sample') and returns a numpy array of cutpoints.

See also:

pept.LineData
Encapsulate LoRs for ease of iteration and plotting.

pept.tracking.peptml.HDBSCANClusterer
Efficient, parallel HDBSCAN-based clustering of cutpoints.

pept.scanners.ParallelScreens
Read in and initialise a *pept.LineData* instance from parallel screens PET/PEPT detectors.

pept.utilities.read_csv
Fast CSV file reading into numpy arrays.

Examples

Compute the minpoints for a *LineData* instance for all triplets of lines that are less than 0.1 from those lines:

```
>>> line_data = pept.LineData(example_data)
>>> minpts = peptml.Minpoints(line_data, 3, 0.1)
```

Compute the minpoints for a single sample:

```
>>> sample = line_data[0]
>>> cutpts_sample = peptml.find_minpoints(sample, 3, 0.1)
```

Attributes

line_data
[instance of *pept.LineData*] The LoRs for which the cutpoints will be computed. It must be an instance of *pept.LineData*.

num_lines
[int] The number of lines in each combination of LoRs used to compute the MDP. This function considers every combination of *num_lines* from the input *sample_lines*. It must be smaller or equal to the number of input lines *sample_lines*.

max_distance
[float] The maximum allowed distance between an MDP and its constituent lines. If any distance from the MDP to one of its lines is larger than *max_distance*, the MDP is thrown away. A good starting value would be 0.1 mm for small tracers and/or clean data, or 0.2 mm for larger tracers and/or noisy data.

cutoffs
[list-like of length 6] A list (or equivalent) of the cutoff distances for every axis, formatted as *[x_min, x_max, y_min, y_max, z_min, z_max]*. Only the minpoints which fall within these cutoff distances are considered. The default is None, in which case they are automatically computed using *pept.tracking.peptml.get_cutoffs*.

__init__(*num_lines*, *max_distance*, *cutoffs=None*, *append_indices=False*)
Minpoints class constructor.

Parameters

num_lines
[int] The number of lines in each combination of LoRs used to compute the MDP. This function considers every combination of *num_lines* from the input *sample_lines*. It must be smaller or equal to the number of input lines *sample_lines*.

max_distance
[float] The maximum allowed distance between an MDP and its constituent lines. If any distance from the MDP to one of its lines is larger than *max_distance*, the MDP is thrown away. A good starting value would be 0.1 mm for small tracers and/or clean data, or 0.2 mm for larger tracers and/or noisy data.

cutoffs
[list-like `of length` 6, optional] A list (or equivalent) of the cutoff distances for every axis, formatted as *[x_min, x_max, y_min, y_max, z_min, z_max]*. Only the minpoints which fall within these cutoff distances are considered. The default is None, in which case they are automatically computed using *pept.tracking.peptml.get_cutoffs*.

append_indices
[bool, `default False`] If set to *True*, the indices of the individual LoRs that were used to compute each minpoint are also appended to the returned array.

Raises

`TypeError`
If *line_data* is not an instance of *pept.LineData*.

`ValueError`
If 2 <= num_lines <= len(sample_lines) is not satisfied.

`ValueError`
If *cutoffs* is not a one-dimensional array with values formatted as *[min_x, max_x, min_y, max_y, min_z, max_z]*.

Methods

`__init__`(num_lines, max_distance[, cutoffs, ...])	Minpoints class constructor.
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample_lines)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

append_indices
cutoffs
max_distance
num_lines

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` `subclass` `instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

property num_lines

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

property max_distance

property cutoffs

property append_indices

fit_sample(*sample_lines*)

pept.tracking.HDBSCAN

class pept.tracking.**HDBSCAN**(*true_fraction*, *max_tracers=1*)

Bases: *PointDataFilter*

Use HDBSCAN to cluster some `pept.PointData` and append a cluster label to each point.

Filter signature:

```
PointData -> HDBSCAN.fit_sample -> PointData
```

The only free parameter to select is the `true_fraction`, a relative measure of the ratio of inliers to outliers. A noisy sample - e.g. first pass of clustering of cutpoints - may need a value of *0.15*. A cleaned up dataset - e.g. a second pass of clustering - can work with *0.6*.

You can also set the maximum number of tracers visible at any one time in the system in `max_tracers` (default 1). This is simply an inverse scaling factor, but the `true_fraction` is quite robust with varying numbers of tracers.

__init__(*true_fraction*, *max_tracers=1*)

Methods

__init__(true_fraction[, max_tracers])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(point_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(sample_points)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*point_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename` or `file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
`[filename` or `file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

fit_sample(*sample_points*)

pept.tracking.FPI

class pept.tracking.**FPI**(*w=3.0*, *r=0.4*, *lld_counts=0.0*, *verbose=False*)

Bases: `VoxelsFilter`

FPI is a modern voxel-based tracer-location algorithm that can reliably work with unknown numbers of tracers in fast and noisy environments.

It was successfully used to track fast-moving radioactive tracers in pipe flows at the Virginia Commonwealth University. If you use this algorithm in your work, please cite the following paper:

> Wiggins C, Santos R, Ruggles A. A feature point identification method for positron emission particle tracking with multiple tracers. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2017 Jan 21; 843:22-8.

Permission was granted by Dr. Cody Wiggins in March 2021 to publish his code in the *pept* library under the GNU v3.0 license.

Two main methods are provided: *fit_sample* for tracking a single voxel space (i.e. a single *pept.Voxels*) and *fit* which tracks all the samples encapsulated in a *pept.VoxelData* class *in parallel*.

See also:

pept.LineData
Encapsulate LoRs for ease of iteration and plotting.

pept.PointData
Encapsulate points for ease of iteration and plotting.

pept.utilities.read_csv
Fast CSV file reading into numpy arrays.

PlotlyGrapher
Easy, publication-ready plotting of PEPT-oriented data.

Examples

A typical workflow would involve reading LoRs from a file, creating a lazy *VoxelData* voxellised representation, instantiating an *FPI* class, tracking the tracer locations from the LoRs, and plotting them.

```
>>> import pept
>>>
>>> lors = pept.LineData(...)   # set sample_size and overlap appropriately
>>> voxels = pept.tracking.Voxelize((50, 50, 50)).fit(lors)
>>>
>>> fpi = pept.tracking.FPI(w = 3, r = 0.4)
>>> positions = fpi.fit(voxels) # this is a `pept.PointData` instance
```

A much more efficient approach would be to create a *pept.Pipeline* containing a voxelization step and then FPI:

```
>>> from pept.tracking import *
>>>
>>> pipeline = Voxelize((50, 50, 50)) + FPI() + Stack()
>>> positions = pipeline.fit(lors)
```

Finally, plotting results:

```
>>> from pept.plots import PlotlyGrapher
>>>
>>> grapher = PlotlyGrapher()
>>> grapher.add_points(positions)
>>> grapher.show()
```

```
>>> from pept.plots import PlotlyGrapher2D
>>> PlotlyGrapher2D().add_timeseries(positions).show()
```

Attributes

w
[`double`] Search range to be used in local maxima calculation. Typical values for w are 2 - 5 (lower number for more particles or smaller particle separation).

r
[`double`] Fraction of peak value used as threshold. Typical values for r are usually between 0.3 and 0.6 (lower for more particles, higher for greater background noise)

lld_counts
[`double`, `default` 0] A secondary lld to prevent assigning local maxima to voxels with very low values. The parameter lld_counts is not used much in practice - for most cases, it can be set to zero.

`__init__`(*w=3.0*, *r=0.4*, *lld_counts=0.0*, *verbose=False*)

FPI class constructor.

Parameters

w
[`double`] Search range to be used in local maxima calculation. Typical values for w are 2 - 5 (lower number for more particles or smaller particle separation).

r
[`double`] Fraction of peak value used as threshold. Typical values for r are usually between 0.3 and 0.6 (lower for more particles, higher for greater background noise)

lld_counts
[`double`, `default` 0] A secondary lld to prevent assigning local maxima to voxels with very low values. The parameter *lld_counts* is not used much in practice - for most cases, it can be set to zero.

verbose
[bool, `default` `False`] Show extra information on class instantiation.

Methods

`__init__`([w, r, lld_counts, verbose])	*FPI* class constructor.
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(voxels[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(voxels)	Use the FPI algorithm to locate a tracer from a single voxellised space (i.e.
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit_sample`(*voxels:* Voxels)

Use the FPI algorithm to locate a tracer from a single voxellised space (i.e. from one sample of LoRs).

A sample of LoRs can be voxellised using the *pept.Voxels.from_lines* method before calling this function.

Parameters

voxels
[`pept.Voxels`] A single voxellised space (i.e. from a single sample of LoRs) for which the tracers' locations will be found using the FPI method.

Returns

locations
[`numpy.ndarray` or `pept.PointData`] The tracked locations found; if *as_array* is True, they are returned as a NumPy array with columns [time, x, y, z, error_x, error_y, error_z]. If *as_array* is False, the points are returned in a *pept.PointData* for ease of visualisation.

Raises

`TypeError`
If *voxels* is not an instance of *pept.Voxels* (or subclass thereof).

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*voxels*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Trajectory Separation Algorithms

pept.tracking.Segregate(window, cut_distance)	Segregate the intertwined points from multiple trajectories into individual paths.
pept.tracking.Reconnect(tmax, dmax[, ...])	Best-fit trajectory segment reconstruction based on time, distance and arbitrary tracer signatures.

pept.tracking.Segregate

class pept.tracking.**Segregate**(*window*, *cut_distance*, *min_trajectory_size=5*, *max_time_interval=1.7976931348623157e+308*)

Bases: *Reducer*

Segregate the intertwined points from multiple trajectories into individual paths.

Reducer signature:

```
      pept.PointData -> Segregate.fit -> pept.PointData
list[pept.PointData] -> Segregate.fit -> pept.PointData
       numpy.ndarray -> Segregate.fit -> pept.PointData
```

The points in *point_data* (a numpy array or *pept.PointData*) are used to construct a minimum spanning tree in which every point can only be connected to *points_window* points around it - this "window" refers to the points in the initial data array, sorted based on the time column; therefore, only points within a certain time-frame can be connected. All edges (or "connections") in the minimum spanning tree that are larger than *trajectory_cut_distance* are removed (or "cut") and the remaining connected "clusters" are deemed individual trajectories if they contain more than *min_trajectory_size* points.

The trajectory indices (or labels) are appended to *point_data*. That is, for each data point (i.e. row) in *point_data*, a label will be appended starting from 0 for the corresponding trajectory; a label of -1 represents noise. If *point_data* is a numpy array, a new numpy array is returned; if it is a *pept.PointData* instance, a new instance is returned.

This function uses single linkage clustering with a custom metric for spatio-temporal data to segregate trajectory points. The single linkage clustering was optimised for this use-case: points are only connected if they are within a certain *points_window* in the time-sorted input array. Sparse matrices are also used for minimising the memory footprint.

See also:

`Reconnect`
Connect segregated trajectories based on tracer signatures.

`PlotlyGrapher`
Easy, publication-ready plotting of PEPT-oriented data.

Examples

A typical workflow would involve transforming LoRs into points using some tracking algorithm. These points include all tracers moving through the system, being intertwined (e.g. for two tracers A and B, the *point_data* array might have two entries for A, followed by three entries for B, then one entry for A, etc.). They can be segregated based on position alone using this function; take for example two tracers that go downwards (below, 'x' is the position, and in parens is the array index at which that point is found).

```
`points`, numpy.ndarray, shape (10, 4), columns [time, x, y, z]:
    x (1)                           x (2)
     x (3)                         x (4)
       x (5)                     x (7)
       x (6)                    x (9)
      x (8)                     x (10)
```

```
>>> import pept.tracking.trajectory_separation as tsp
>>> points_window = 10
>>> trajectory_cut_distance = 15    # mm
>>> segregated_trajectories = tsp.segregate_trajectories(
>>>     points, points_window, trajectory_cut_distance
>>> )
```

```
`segregated_trajectories`, numpy.ndarray, shape (10, 5),
columns [time, x, y, z, trajectory_label]:
    x (1, label = 0)            x (2, label = 1)
     x (3, label = 0)          x (4, label = 1)
       x (5, label = 0)      x (7, label = 1)
       x (6, label = 0)     x (9, label = 1)
      x (8, label = 0)      x (10, label = 1)
```

Attributes

window
[int] Two points are "reachable" (i.e. they can be connected) if and only if they are within *points_window* in the time-sorted input *point_data*. As the points from different trajectories are intertwined (e.g. for two tracers A and B, the *point_data* array might have two entries for A, followed by three entries for B, then one entry for A, etc.), this should optimally be the largest number of points in the input array between two consecutive points on the same trajectory. If *points_window* is too small, all points in the dataset will be unreachable. Naturally, a larger *time_window* correponds to more pairs needing to be checked (and the function will take a longer to complete).

cut_distance
[float] Once all the closest points are connected (i.e. the minimum spanning tree is constructed), separate all trajectories that are further apart than *trajectory_cut_distance*.

min_trajectory_size
[float, `default` 5] After the trajectories have been cut, declare all trajectories with fewer points than *min_trajectory_size* as noise.

max_time_interval
[float, `default` `np.finfo`(float):obj:*.max*] Only connect points if the time difference between their timestamps is smaller than *max_time_interval*. *Setting added in pept-0.5.2*.

`__init__`(*window*, *cut_distance*, *min_trajectory_size=5*, *max_time_interval=1.7976931348623157e+308*)

Methods

`__init__`(window, cut_distance[, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(points)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*points*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.Reconnect

class pept.tracking.**Reconnect**(*tmax*, *dmax*, *column='label'*, *num_points=10*, ***signatures*)

Bases: *Reducer*

Best-fit trajectory segment reconstruction based on time, distance and arbitrary tracer signatures.

Reducer signature:

```
      pept.PointData -> Segregate.fit -> pept.PointData
list[pept.PointData] -> Segregate.fit -> pept.PointData
       numpy.ndarray -> Segregate.fit -> pept.PointData
```

After a trajectory segregation step (e.g. using Segregate), you may be left with multiple smaller trajectory segments. Some trajectories can be reconstructed even when losing the tracers for a bit.

When a tracer is lost for less than *tmax* time and *dmax* distance, its trajectory segments are reconnected; if multiple condidates are possible, the best fit is used.

Multiple tracer signatures can be used to improve the reconnection step; supply them as data column names and difference thresholds, e.g. an extra keyword argument `v = 1` will join trajectories whose difference in velocity is smaller than 1 m/s.

The last *num_points* points on a segment are averaged before they are connected with the first *num_points* on another segment.

New in pept-0.4.2

Examples

Reconnect segments that are closer than 1 second in time and 0.1 m apart:

```
>>> from pept.tracking import *
>>> trajectories = Reconnect(tmax = 1000, dmax = 100).fit(segments)
```

You can use the *cluster_size* (set by the `Centroids` filter) as a tracer signature; allow segments to be reconnected if the difference in their cluster size is < 100:

```
>>> trajectories = Reconnect(1000, 100, cluster_size = 100).fit(segments)
```

And a velocity *v* difference < 0.1:

```
>>> Reconnect(1000, 100, cluster_size = 100, v = 0.1).fit(segments)
```

`__init__`(*tmax*, *dmax*, *column='label'*, *num_points=10*, ***signatures*)

Methods

`__init__`(tmax, dmax[, column, num_points])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(points)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*points*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename or file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Time Of Flight Algorithms

pept.tracking.TimeOfFlight([...])	Compute the positron annihilation locations of each LoR as given by the Time Of Flight (ToF) data of the two LoR timestamps.
pept.tracking.CutpointsToF([max_distance, ...])	Compute cutpoints from all pairs of lines whose Time Of Flight-predicted locations are closer than *max_distance*.
pept.tracking.GaussianDensity([sigma])	Append weights according to the Gaussian distribution that best fits the samples of points.

pept.tracking.TimeOfFlight

class pept.tracking.**TimeOfFlight**(*temporal_resolution=None*, *points=False*)

Bases: *LineDataFilter*

Compute the positron annihilation locations of each LoR as given by the Time Of Flight (ToF) data of the two LoR timestamps.

Filter signature:

```
LineData -> TimeOfFlight.fit_sample -> LineData  (points = False)
LineData -> TimeOfFlight.fit_sample -> PointData (points = True)
```

Importantly, the input LineData must have at least 8 columns, formatted as [t1, x1, y1, z1, x2, y2, z2, t2] - notice the different timestamps of the two LoR ends.

If *points = False* (default), the computed ToF points are saved as an extra attribute "tof" in the input LineData; otherwise they are returned directly.

The *temporal_resolution* should be set to the FWHM of the temporal resolution in the LoR timestamps, in self-consistent units of measure (e.g. m / s or mm / ms, but not mm / s). If it is set, the "temporal_resolution" and "spatial_resolution" extra attributes are set on the ToF points.

New in pept-0.4.2

Examples

Generate 10 random LoRs between (-100, 100) mm and ms with 8 columns for the ToF format.

```
>>> import numpy as np
>>> import pept
```

```
>>> rng = np.random.default_rng(0)
>>> lors = pept.LineData(
>>>     rng.uniform(-100, 100, (10, 8)),
>>>     columns = ["t1", "x1", "y1", "z1", "x2", "y2", "z2", "t2"],
>>> )
>>> lors
pept.LineData (samples: 1)
--------------------------
sample_size = 10
overlap = 0
lines =
  (rows: 10, columns: 8)
  [[ 57.4196615  -52.1261114  ...  -9.93212667  59.26485406]
   [-53.8715582  -89.59573979 ... -40.26077344  34.39897559]
   ...
   [ 51.59020047   2.55174465 ... -31.13800424 -13.94025361]
   [ 93.21241616  12.44636845 ... -75.08905883 -42.3338486 ]]
columns = ['t1', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2', 't2']
attrs = {}
```

Compute Time of Flight annihilation locations from the two timestamps in the data above. Assume a temporal resolution of 100 ps - be careful to use self-consistent units; in this case we are using mm and ms:

```
>>> from pept.tracking import *
```

```
>>> temporal_resolution = 100e-12 * 1000    # ms
>>> lors_tof = TimeOfFlight(temporal_resolution).fit_sample(lors)
>>> lors_tof
pept.LineData (samples: 1)
--------------------------
sample_size = 10
overlap = 0
lines =
  (rows: 10, columns: 8)
  [[ 57.4196615  -52.1261114  ...  -9.93212667  59.26485406]
   [-53.8715582  -89.59573979 ... -40.26077344  34.39897559]
   ...
   [ 51.59020047   2.55174465 ... -31.13800424 -13.94025361]
   [ 93.21241616  12.44636845 ... -75.08905883 -42.3338486 ]]
columns = ['t1', 'x1', 'y1', 'z1', 'x2', 'y2', 'z2', 't2']
attrs = {
  'tof': pept.PointData (samples: 1)
---------------------------
sample_...
}
```

```
>>> lors_tof.attrs["tof"]
pept.PointData (samples: 1)
---------------------------
sample_size = 10
overlap = 0
points =
  (rows: 10, columns: 4)
  [[ 5.64970655e+01 -3.22092074e+07  2.41767704e+08 -1.30428351e+08]
   [-9.80068250e+01 -2.48775932e+09 -1.12904720e+10 -6.43480969e+09]
   ...
   [ 1.88249731e+01  3.34819602e+09 -8.78848458e+09  2.83529405e+09]
   [ 2.54392837e+01  1.90343279e+10 -1.92717662e+09 -6.84078611e+09]]
columns = ['t', 'x', 'y', 'z']
attrs = {
  'temporal_resolution': 1.0000000000000001e-07
  'spatial_resolution': 29.9792458
}
```

Alternatively, you can extract only the ToF points directly:

```
>>> tof = TimeOfFlight(temporal_resolution, points = True).fit_sample(lors)
>>> tof
pept.PointData (samples: 1)
---------------------------
sample_size = 10
overlap = 0
points =
  (rows: 10, columns: 4)
  [[ 5.64970655e+01 -3.22092074e+07  2.41767704e+08 -1.30428351e+08]
```

(continues on next page)

(continued from previous page)

```
   [-9.80068250e+01 -2.48775932e+09 -1.12904720e+10 -6.43480969e+09]
   ...
   [ 1.88249731e+01  3.34819602e+09 -8.78848458e+09  2.83529405e+09]
   [ 2.54392837e+01  1.90343279e+10 -1.92717662e+09 -6.84078611e+09]]
columns = ['t', 'x', 'y', 'z']
attrs = {
  'temporal_resolution': 1.0000000000000001e-07
  'spatial_resolution': 29.9792458
}
```

`__init__`(*temporal_resolution=None*, *points=False*)

Methods

`__init__`([temporal_resolution, points])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit_sample`(*sample:* LineData)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.CutpointsToF

class `pept.tracking.`**`CutpointsToF`**(*max_distance=None*, *cutoffs=None*, *append_indices=False*, *cutpoints_only=False*)

Bases: `LineDataFilter`

Compute cutpoints from all pairs of lines whose Time Of Flight-predicted locations are closer than *max_distance*.

Filter signature:

```
LineData -> CutpointsToF.fit_sample -> PointData
```

If the `TimeOfFlight` filter was used and a temporal resolution was specified (as a FWHM), then *max_distance* is automatically inferred as the minimum between 2 * "spatial_resolution" and the dimension-wise standard deviation of the input points.

The *cutoffs* parameter may be set as [xmin, xmax, ymin, ymax, zmin, zmax] for a minimum bounding box outside of which cutpoints are discarded. Otherwise it is automatically set to the minimum bounding box containing all input LoRs.

If *append_indices* = *True*, two extra columns are appended to the result as "line_index1" and "line_index2" containing the indices of the LoRs that produced each cutpoint; an extra attribute "_lines" is also set to the input *LineData*.

If *cutpoints_only = False* (default), the Time Of Flight-predicted positron annihilation locations are also appended to the returned points.

New in pept-0.4.2

See also:

`pept.LineData`
: Encapsulate LoRs for ease of iteration and plotting.

`pept.tracking.HDBSCAN`
: Efficient, HDBSCAN-based clustering of (cut)points.

`pept.read_csv`
: Fast CSV file reading into numpy arrays.

Examples

Make sure to use the `TimeOfFlight` filter to compute to ToF annihilation locations; if you specify a temporal resolution, the *max_distance* parameter is automatically computed:

```
>>> import pept
>>> from pept.tracking import *
```

```
>>> line_data = pept.LineData(example_tof_data)
>>> line_data_tof = TimeOfFlight(100e-9).fit_sample(line_data)
>>> cutpoints_tof = CutpointsToF().fit_sample(line_data_tof)
```

Alternatively, set *max_distance* yourself:

```
>>> line_data = pept.LineData(example_tof_data)
>>> line_data_tof = TimeOfFlight().fit_sample(line_data)
>>> cutpoints_tof = CutpointsToF(5.0).fit_sample(line_data_tof)
```

`__init__`(*max_distance=None*, *cutoffs=None*, *append_indices=False*, *cutpoints_only=False*)

Methods

`__init__`([max_distance, cutoffs, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(line_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(sample_lines)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

Attributes

append_indices
cutoffs
max_distance

property max_distance

property cutoffs

property append_indices

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*line_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

fit_sample(*sample_lines:* LineData)

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.tracking.GaussianDensity

class pept.tracking.**GaussianDensity**(*sigma=None*)

Bases: *Filter*

Append weights according to the Gaussian distribution that best fits the samples of points.

Filter signature:

```
      PointData -> GaussianDensity.fit_sample -> PointData
  numpy.ndarray -> GaussianDensity.fit_sample -> PointData
list[PointData] -> GaussianDensity.fit_sample -> list[PointData]
```

This is treated as an optimisation problem: find the 3D location that maximises the sum of Probability Distributions (PDF) centered at each point.

```
Given N points p_1, p_2, ..., p_N:

          N
maximise sum( exp( -0.5 * |x - p_i|^2 / sigma^2 ) )
   x      i
```

Each point is then assigned a weight corresponding to its PDF - i.e. the exponential part - saved in the *weight* column.

Sigma controls the standard deviation of the Gaussian distribution centred at each point; this corresponds to the relative uncertainty in each point's location. For `TimeOfFlight` data, leave *sigma = None* and it will be computed from the "spatial_resolution" attribute.

You can use `Centroids` afterwards to compute the weighted centroid, i.e. where the tracer is. For multiple particle tracking (or just more robustness to noise) you can use *HDBSCAN* + *SplitLabels* beforehand.

New in pept-0.4.2

__init__(*sigma=None*)

Methods

__init__([sigma])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(samples[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
fit_sample(points)	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`fit`(*samples*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

fit_sample(*points*)

Post Processing Algorithms

pept.tracking.Velocity(window[, degree, ...])	Append the dimension-wise or absolute velocity to samples of points using a 2D fitted polynomial in a rolling window mode.

pept.tracking.Velocity

class pept.tracking.**Velocity**(*window*, *degree=2*, *absolute=False*)

Bases: *PointDataFilter*

Append the dimension-wise or absolute velocity to samples of points using a 2D fitted polynomial in a rolling window mode.

Filter signature:

```
PointData -> Velocity.fit_sample -> PointData
```

If Numba is installed, a fast, natively-compiled algorithm is used.

If *absolute* = *False*, the “vx”, “vy” and “vz” columns are appended. If *absolute* = *True*, then the “v” column is appended.

__init__(*window*, *degree=2*, *absolute=False*)

Methods

`__init__`(window[, degree, absolute])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(point_data[, executor, max_workers, verbose])	Apply self.fit_sample (implemented by subclasses) according to the execution policy.
`fit_sample`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit_sample(*samples*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*point_data*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

Apply self.fit_sample (implemented by subclasses) according to the execution policy. Simply return a list of processed samples. If you need a reduction step (e.g. stack all processed samples), apply it in the subclass.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Post Processing (pept.processing)

The PEPT-oriented post-processing suite, including occupancy grid, vector velocity fields, etc.

This module contains fast, robust functions that operate on PEPT-like data and integrate with the *pept* library's base classes.

Probability / Residence Distributions

pept.processing.DynamicProbability2D(...[, ...])	Compute the 2D probability distribution of some tracer quantity (eg velocity in each cell).
pept.processing.DynamicProbability3D(...[, ...])	Compute the 3D probability distribution of some tracer quantity (eg velocity in each cell).
pept.processing.ResidenceDistribution2D(diameter)	Compute the 2D residence distribution of some tracer quantity (eg time spent in each cell).
pept.processing.ResidenceDistribution3D(diameter)	Compute the 3D residence distribution of some tracer quantity (eg time spent in each cell).

pept.processing.DynamicProbability2D

class pept.processing.**DynamicProbability2D**(*diameter*, *column*, *dimensions='xy'*, *resolution=(512, 512)*, *xlim=None*, *ylim=None*, *max_workers=None*, *verbose=True*)

Bases: *Reducer*

Compute the 2D probability distribution of some tracer quantity (eg velocity in each cell).

Reducer signature:

```
      PointData -> DynamicProbability2D.fit -> Pixels
list[PointData] -> DynamicProbability2D.fit -> Pixels
  numpy.ndarray -> DynamicProbability2D.fit -> Pixels
```

This reducer calculates the average value of the tracer quantity in each cell of a 2D pixel grid; it uses the full projected tracer area for the pixelization step, so the distribution is accurate for arbitrarily fine resolutions.

Parameters

diameter
[float] The diameter of the imaged tracer.

column
[str or int] The *PointData* column used to compute the probability distribution, given as a name (*str*) or index (*int*).

dimensions
[str or list[int], default "xy"] The tracer coordinates used to rasterize its trajectory, given as a string (e.g. "xy" projects the points onto the XY plane) or a list with two column indices (e.g. [1, 3] for XZ).

resolution
[tuple[int, int], default (512, 512)] The number of pixels used for the rasterization grid in the X and Y dimensions.

xlim
[tuple[float, float], optional] The physical limits in the X dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

ylim
[tuple[float, float], optional] The physical limits in the y dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

max_workers
[int, optional] The maximum number of workers (threads, processes or ranks) to use by the parallel executor; if 1, it is sequential (and produces the clearest error messages should they happen). If unset, the `os.cpu_count()` is used.

verbose
[bool or str default True] If True, time the computation and print the state of the execution.

Examples

Compute the velocity probability distribution of a single tracer trajectory having a column named "v" corresponding to the tracer velocity:

```
>>> trajectories = pept.load(...)
>>> pixels_vel = DynamicProbability2D(1.2, "v", "xy").fit(trajectories)
```

Plot the pixel grid:

```
>>> from pept.plots import PlotlyGrapher2D
>>> PlotlyGrapher2D().add_pixels(pixels_vel).show()
```

For multiple tracer trajectories, you can use `Segregate` then `SplitAll('label')` before calling this reducer to rasterize each trajectory separately:

```
>>> vel_pipeline = pept.Pipeline([
>>>     Segregate(20, 10),
>>>     SplitAll("label"),
>>>     DynamicProbability2D(1.2, "v", "xy")
>>> ])
>>> pixels_vel = vel_pipeline.fit(trajectories)
```

__init__(*diameter*, *column*, *dimensions='xy'*, *resolution=(512, 512)*, *xlim=None*, *ylim=None*, *max_workers=None*, *verbose=True*)

Methods

`__init__`(diameter, column[, dimensions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit(*samples*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.DynamicProbability3D

class pept.processing.**DynamicProbability3D**(*diameter*, *column*, *dimensions='xyz'*, *resolution=(50, 50, 50)*, *xlim=None*, *ylim=None*, *zlim=None*, *max_workers=None*, *verbose=True*)

Bases: *Reducer*

Compute the 3D probability distribution of some tracer quantity (eg velocity in each cell).

Reducer signature:

```
      PointData -> DynamicProbability3D.fit -> Voxels
list[PointData] -> DynamicProbability3D.fit -> Voxels
  numpy.ndarray -> DynamicProbability3D.fit -> Voxels
```

This reducer calculates the average value of the tracer quantity in each cell of a 3D voxel grid; it uses the full projected tracer area for the voxelization step, so the distribution is accurate for arbitrarily fine resolutions.

Parameters

diameter
[float] The diameter of the imaged tracer.

column
[str or int] The *PointData* column used to compute the probability distribution, given as a name (*str*) or index (*int*).

dimensions
[str or list[int], default "xyz"] The tracer coordinates used to rasterize its trajectory, given as a string (e.g. "xyz" or "zyx") or a list with three column indices (e.g. [1, 2, 3] for XYZ).

resolution
[tuple[int, int, int], default (50, 50, 50)] The number of pixels used for the rasterization grid in the X, Y, Z dimensions.

xlim
[tuple[float, float], optional] The physical limits in the X dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

ylim
[tuple[float, float], optional] The physical limits in the y dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

zlim
[tuple[float, float], optional] The physical limits in the z dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

max_workers
[int, optional] The maximum number of workers (threads, processes or ranks) to use by the parallel executor; if 1, it is sequential (and produces the clearest error messages should they happen). If unset, the `os.cpu_count()` is used.

verbose
[bool or str default True] If True, time the computation and print the state of the execution.

Examples

Compute the velocity probability distribution of a single tracer trajectory having a column named "v" corresponding to the tracer velocity:

```
>>> trajectories = pept.load(...)
>>> voxels_vel = DynamicProbability3D(1.2, "v").fit(trajectories)
```

Plot the pixel grid:

```
>>> from pept.plots import PlotlyGrapher
>>> PlotlyGrapher().add_voxels(voxels_vel).show()
```

For multiple tracer trajectories, you can use `Segregate` then `SplitAll('label')` before calling this reducer to rasterize each trajectory separately:

```
>>> vel_pipeline = pept.Pipeline([
>>>     Segregate(20, 10),
>>>     SplitAll("label"),
>>>     DynamicProbability3D(1.2, "v")
>>> ])
>>> voxels_vel = vel_pipeline.fit(trajectories)
```

`__init__`(*diameter*, *column*, *dimensions='xyz'*, *resolution=(50, 50, 50)*, *xlim=None*, *ylim=None*, *zlim=None*, *max_workers=None*, *verbose=True*)

Methods

`__init__`(diameter, column[, dimensions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.ResidenceDistribution2D

class pept.processing.**ResidenceDistribution2D**(*diameter*, *column='t'*, *dimensions='xy'*, *resolution=(512, 512)*, *xlim=None*, *ylim=None*, *max_workers=None*, *verbose=True*)

Bases: *Reducer*

Compute the 2D residence distribution of some tracer quantity (eg time spent in each cell).

Reducer signature:

```
      PointData -> ResidenceDistribution2D.fit -> Pixels
list[PointData] -> ResidenceDistribution2D.fit -> Pixels
  numpy.ndarray -> ResidenceDistribution2D.fit -> Pixels
```

This reducer calculates the cumulative value of the tracer quantity in each cell of a 2D pixel grid; it uses the full projected tracer area for the pixelization step, so the distribution is accurate for arbitrarily fine resolutions.

Parameters

diameter
[float] The diameter of the imaged tracer.

column
[str or int, default "t"] The *PointData* column used to compute the residence distribution, given as a name (*str*) or index (*int*).

dimensions
[str or list[int], default "xy"] The tracer coordinates used to rasterize its trajectory, given as a string (e.g. "xy" projects the points onto the XY plane) or a list with two column indices (e.g. [1, 3] for XZ).

resolution
[tuple[int, int], default (512, 512)] The number of pixels used for the rasterization grid in the X and Y dimensions.

xlim
[tuple[float, float], optional] The physical limits in the X dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

ylim
[tuple[float, float], optional] The physical limits in the y dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

max_workers
[int, optional] The maximum number of workers (threads, processes or ranks) to use by the parallel executor; if 1, it is sequential (and produces the clearest error messages should they happen). If unset, the `os.cpu_count()` is used.

verbose
[bool or str default True] If True, time the computation and print the state of the execution.

Examples

Compute the residence time distribution of a single tracer trajectory:

```
>>> trajectories = pept.load(...)
>>> pixels_rtd = ResidenceDistribution2D(1.2, "t", "xy").fit(trajectories)
```

Plot the pixel grid:

```
>>> from pept.plots import PlotlyGrapher2D
>>> PlotlyGrapher2D().add_pixels(pixels_rtd).show()
```

For multiple tracer trajectories, you can use `Segregate` then `SplitAll('label')` before calling this reducer to rasterize each trajectory separately:

```
>>> rtd_pipeline = pept.Pipeline([
>>>     Segregate(20, 10),
>>>     SplitAll("label"),
>>>     ResidenceDistribution2D(1.2, "t", "xy")
>>> ])
>>> pixels_rtd = rtd_pipeline.fit(trajectories)
```

`__init__`(*diameter*, *column='t'*, *dimensions='xy'*, *resolution=(512, 512)*, *xlim=None*, *ylim=None*, *max_workers=None*, *verbose=True*)

Methods

`__init__`(diameter[, column, dimensions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.ResidenceDistribution3D

class pept.processing.**ResidenceDistribution3D**(*diameter*, *column='t'*, *dimensions='xyz'*, *resolution=(50, 50, 50)*, *xlim=None*, *ylim=None*, *zlim=None*, *max_workers=None*, *verbose=True*)

Bases: *Reducer*

Compute the 3D residence distribution of some tracer quantity (eg time spent in each cell).

Reducer signature:

```
      PointData -> ResidenceDistribution3D.fit -> Pixels
list[PointData] -> ResidenceDistribution3D.fit -> Pixels
  numpy.ndarray -> ResidenceDistribution3D.fit -> Pixels
```

This reducer calculates the cumulative value of the tracer quantity in each cell of a 3D voxel grid; it uses the full projected tracer area for the voxelization step, so the distribution is accurate for arbitrarily fine resolutions.

Parameters

diameter
[float] The diameter of the imaged tracer.

column
[str or int] The *PointData* column used to compute the probability distribution, given as a name (*str*) or index (*int*).

dimensions
[str or list[int], default "xyz"] The tracer coordinates used to rasterize its trajectory, given as a string (e.g. "xyz" or "zyx") or a list with three column indices (e.g. [1, 2, 3] for XYZ).

resolution
[tuple[int, int, int], default (50, 50, 50)] The number of pixels used for the rasterization grid in the X, Y, Z dimensions.

xlim
[tuple[float, float], optional] The physical limits in the X dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

ylim
[tuple[float, float], optional] The physical limits in the y dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

zlim
[tuple[float, float], optional] The physical limits in the z dimension of the pixel grid. If unset, it is automatically computed to contain all tracer positions (default).

max_workers
[int, optional] The maximum number of workers (threads, processes or ranks) to use by the parallel executor; if 1, it is sequential (and produces the clearest error messages should they happen). If unset, the `os.cpu_count()` is used.

verbose
[bool or str default True] If True, time the computation and print the state of the execution.

Examples

Compute the residence time distribution of a single tracer trajectory:

```
>>> trajectories = pept.load(...)
>>> voxels_rtd = ResidenceDistribution3D(1.2, "t").fit(trajectories)
```

Plot the pixel grid:

```
>>> from pept.plots import PlotlyGrapher
>>> PlotlyGrapher().add_voxels(voxels_rtd).show()
```

For multiple tracer trajectories, you can use `Segregate` then `SplitAll('label')` before calling this reducer to rasterize each trajectory separately:

```
>>> rtd_pipeline = pept.Pipeline([
>>>     Segregate(20, 10),
>>>     SplitAll("label"),
>>>     ResidenceDistribution3D(1.2, "t")
```

(continues on next page)

(continued from previous page)

```
>>> ])
>>> voxels_rtd = rtd_pipeline.fit(trajectories)
```

`__init__`(*diameter*, *column='t'*, *dimensions='xyz'*, *resolution=(50, 50, 50)*, *xlim=None*, *ylim=None*, *zlim=None*, *max_workers=None*, *verbose=True*)

Methods

`__init__`(diameter[, column, dimensions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
`[filename` or `file handle]` If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Vector Grids

pept.processing.VectorField2D(diameter[, ...])	Compute a 2D vector field - effectively two 2D grids computed from two columns, for example X and Y velocities.
pept.processing.VectorGrid2D(xpixels, ypixels)	Object produced by `VectorField2D` storing 2 grids of voxels *xpixels*, *ypixels*, for example velocity vector fields / quiver plots.
pept.processing.VectorField3D(diameter[, ...])	Compute a 3D vector field - effectively three 3D grids computed from three columns, for example X, Y and Z velocities.
pept.processing.VectorGrid3D(xvoxels, ...)	Object produced by `VectorField3D` storing 3 grids of voxels *xvoxels*, *yvoxels*, *zvoxels*, for example velocity vector fields / quiver plots.

pept.processing.VectorField2D

class pept.processing.**VectorField2D**(*diameter*, *columns=['vx', 'vy']*, *dimensions='xy'*, *resolution=(50, 50)*, *xlim=None*, *ylim=None*, *max_workers=None*, *verbose=True*)

Bases: *Reducer*

Compute a 2D vector field - effectively two 2D grids computed from two columns, for example X and Y velocities.

Reducer signature:

```
      PointData -> VectorField2D.fit -> VectorGrid2D
list[PointData] -> VectorField2D.fit -> VectorGrid2D
  numpy.ndarray -> VectorField2D.fit -> VectorGrid2D
```

Examples

Compute a velocity vector field in the Y and Z dimensions (velocities were first calculated using `pept.tracking.Velocity`):

```
>>> from pept.processing import *
>>> trajectories = pept.PointData(...)
>>> field = VectorField2D(0.6, ["vy", "vz"], "yz").fit(trajectories)
>>> field
VectorGrid2D(xpixels, ypixels)
```

Create a quiver plot using Plotly (may be a bit slow):

```
>>> scaling = 16
>>> fig = field.quiver(scaling)
>>> fig.show()
```

Create a 2D vector field (needs PyVista):

```
>>> scaling = 16
>>> fig = field.vectors(scaling)
>>> fig.plot(cmap = "magma")
```

`__init__`(*diameter*, *columns=['vx', 'vy']*, *dimensions='xy'*, *resolution=(50, 50)*, *xlim=None*, *ylim=None*, *max_workers=None*, *verbose=True*)

Methods

`__init__`(diameter[, columns, dimensions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

pept.PEPTObject subclass instance
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)
Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[filename or file handle] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.VectorGrid2D

class pept.processing.**VectorGrid2D**(*xpixels:* Pixels, *ypixels:* Pixels)
Bases: object

Object produced by VectorField2D storing 2 grids of voxels *xpixels*, *ypixels*, for example velocity vector fields / quiver plots.

Examples

Compute a velocity vector field in the Y and Z dimensions (velocities were first calculated using pept.tracking.Velocity):

```
>>> from pept.processing import *
>>> trajectories = pept.PointData(...)
>>> field = VectorField2D(0.6, ["vy", "vz"], "yz").fit(trajectories)
>>> field
VectorGrid2D(xpixels, ypixels)
```

Create a quiver plot using Plotly (may be a bit slow):

```
>>> scaling = 16
>>> fig = field.quiver(scaling)
>>> fig.show()
```

Create a 2D vector field (needs PyVista):

```
>>> scaling = 16
>>> fig = field.vectors(scaling)
>>> fig.plot(cmap = "magma")
```

`__init__`(*xpixels:* Pixels, *ypixels:* Pixels)

Methods

`__init__`(xpixels, ypixels)	
`quiver`([factor])	
`vectors`([factor])	

`vectors`(*factor=1*)

`quiver`(*factor=1*)

pept.processing.VectorField3D

class `pept.processing.`**`VectorField3D`**(*diameter*, *columns=['vx', 'vy', 'vz']*, *dimensions='xyz'*, *resolution=(50, 50, 50)*, *xlim=None*, *ylim=None*, *zlim=None*, *max_workers=None*, *verbose=True*)

Bases: `Reducer`

Compute a 3D vector field - effectively three 3D grids computed from three columns, for example X, Y and Z velocities.

Reducer signature:

```
      PointData -> VectorField3D.fit -> VectorGrid3D
list[PointData] -> VectorField3D.fit -> VectorGrid3D
  numpy.ndarray -> VectorField3D.fit -> VectorGrid3D
```

Examples

Compute a 3D velocity vector field (velocities were first calculated using `pept.tracking.Velocity`):

```
>>> from pept.processing import *
>>> trajectories = pept.PointData(...)
>>> field = VectorField3D(0.6).fit(trajectories)
>>> field
VectorGrid3D(xvoxels, yvoxels, zvoxels)
```

Create a 3D vector field (needs PyVista):

```
>>> scaling = 16
>>> fig = field.vectors(scaling)
>>> fig.plot(cmap = "magma")
```

`__init__`(*diameter*, *columns=['vx', 'vy', 'vz']*, *dimensions='xyz'*, *resolution=(50, 50, 50)*, *xlim=None*, *ylim=None*, *zlim=None*, *max_workers=None*, *verbose=True*)

Methods

`__init__`(diameter[, columns, dimensions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(samples)	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*samples*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.VectorGrid3D

class `pept.processing.`**`VectorGrid3D`**(*xvoxels: Voxels*, *yvoxels: Voxels*, *zvoxels: Voxels*)

Bases: `object`

Object produced by `VectorField3D` storing 3 grids of voxels *xvoxels*, *yvoxels*, *zvoxels*, for example velocity vector fields / quiver plots.

Examples

Compute a 3D velocity vector field (velocities were first calculated using `pept.tracking.Velocity`):

```
>>> from pept.processing import *
>>> trajectories = pept.PointData(...)
>>> field = VectorField3D(0.6).fit(trajectories)
>>> field
VectorGrid3D(xvoxels, yvoxels, zvoxels)
```

Create a 3D vector field (needs PyVista):

```
>>> scaling = 16
>>> fig = field.vectors(scaling)
>>> fig.plot(cmap = "magma")
```

__init__(*xvoxels: Voxels*, *yvoxels: Voxels*, *zvoxels: Voxels*)

Methods

`__init__`(xvoxels, yvoxels, zvoxels)	
`vectors`([factor])	

vectors(*factor=1*)

Mixing Quantification

`pept.processing.LaceyColors`(p1, p2[, ax1, ...])	Compute Lacey-like mixing image, with streamlines passing through plane 1 being split into Red and Blue tracers, then evaluated into RGB pixels at a later point in plane 2.
`pept.processing.LaceyColorsLinear`(directory, ...)	Apply the *LaceyColors* mixing algorithm at *num_divisions* equidistant points between *p1* and *p2*, saving images at each step in *directory*.
`pept.processing.RelativeDeviations`(p1, p2[, ...])	Compute a Lagrangian mixing measure - the changes in tracer distances to a point P1 as they pass through an "inlet" plane and another point P2 when reaching an "outlet" plane.
`pept.processing.RelativeDeviationsLinear`(...)	Apply the *RelativeDeviations* mixing algorithm at *num_divisions* equidistant points between *p1* and *p2*, saving histogram images at each step in *directory*.
`pept.processing.AutoCorrelation`([lag, ...])	Compute autocorrelation of multiple measures (eg YZ velocities) as a function of a lagging variable (eg time).
`pept.processing.SpatialProjections`(...[, ...])	Project multiple tracer passes onto a moving 2D plane along a given *direction* between *start* and *end* coordinates, saving each frame in *directory*.

pept.processing.LaceyColors

class pept.processing.**LaceyColors**(*p1*, *p2*, *ax1=None*, *ax2=None*, *basis1=None*, *basis2=None*, *xlim=None*, *ylim=None*, *max_distance=10*, *resolution=(8, 8)*)

Bases: `Reducer`

Compute Lacey-like mixing image, with streamlines passing through plane 1 being split into Red and Blue tracers, then evaluated into RGB pixels at a later point in plane 2.

Intuitively, red and blue pixels will contain unmixed streamlines, while purple pixels will indicate mixing.

Reducer signature:

```
      PointData -> LaceyColors.fit -> (height, width, 3) pept.Voxels
 list[PointData] -> LaceyColors.fit -> (height, width, 3) pept.Voxels
list[np.ndarray] -> LaceyColors.fit -> (height, width, 3) pept.Voxels
```

Each sample in the input `PointData` is treated as a separate streamline / tracer pass. You can group passes using `Segregate + GroupBy("label")`.

The first plane where tracers are split into Red and Blue streamlines is defined by a point *p1* and direction axis *ax1*. **The point `p1` should be the middle of the pipe**.

The second plane where mixing is evaluated is similarly defined by *p2* and *ax2*. **The point `p2` should be the middle of the volume / pipe**.

If the direction vectors *ax1* and *ax2* are undefined (*None*), the tracers are assumed to follow a straight line between *p1* and *p2*.

The *max_distance* parameter defines the maximum distance allowed between a point and a plane to be considered part of it. The *resolution* defines the number of pixels in the height and width of the resulting image.

New in pept-0.5.1

Examples

Consider a pipe-flow experiment, with tracers moving from side to side in multiple passes / streamlines. First locate the tracers, then split their trajectories into each individual pass:

```
>>> import pept
>>> from pept.tracking import *
>>>
>>> split_pipe = pept.Pipeline([
>>>     Segregate(window = 10, max_distance = 20),  # Appends label column
>>>     GroupBy("label"),                           # Splits into samples
>>>     Reorient(),                                 # Align with X axis
>>>     Center(),                                   # Center points at 0
>>>     Stack(),
>>> ])
>>> streamlines = split_pipe.fit(trajectories)
```

Now each sample in *streamlines* corresponds to a single tracer pass, e.g. *streamlines[0]* is the first pass, *streamlines[1]* is the second. The passes were reoriented and centred such that the pipe is aligned with the X axis.

Now the *LaceyColors* reducer can be used to create an image of the mixing between the pipe entrance and exit:

```
>>> from pept.processing import LaceyColors
>>> entrance = [-100, 0, 0]     # Pipe data was aligned with X and centred
>>> exit = [100, 0, 0]
>>> lacey_image = LaceyColors(entrance, exit).fit(streamlines)
>>> print(lacey_image.voxels)    # RGB channels of image
(8, 8, 3)
```

Now the image can be visualised e.g. with Plotly:

```
>>> from pept.plots import PlotlyGrapher2D
>>> PlotlyGrapher2D().add_image(lacey_image).show()
```

`__init__`(*p1*, *p2*, *ax1=None*, *ax2=None*, *basis1=None*, *basis2=None*, *xlim=None*, *ylim=None*, *max_distance=10*, *resolution=(8, 8)*)

Methods

`__init__`(p1, p2[, ax1, ax2, basis1, basis2, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(trajectories[, executor, max_workers, ...])	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

fit(*trajectories*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.LaceyColorsLinear

class pept.processing.**LaceyColorsLinear**(*directory*, *p1*, *p2*, *xlim=None*, *ylim=None*, *num_divisions=50*, *max_distance=10*, *resolution=(8, 8)*, *height=1000*, *width=1000*, *prefix='frame'*)

Bases: *Reducer*

Apply the *LaceyColors* mixing algorithm at *num_divisions* equidistant points between *p1* and *p2*, saving images at each step in *directory*.

Reducer signature:

```
      PointData -> LaceyColors.fit -> (height, width, 3) np.ndarray
 list[PointData] -> LaceyColors.fit -> (height, width, 3) np.ndarray
list[np.ndarray] -> LaceyColors.fit -> (height, width, 3) np.ndarray
```

For details about the mixing algorithm itself, check the *LaceyColors* documentation.

The generated images (saved in *directory* with *height* x *width* pixels) can be stitched into a video using *pept.plots.make_video*.

New in pept-0.5.1

Examples

Consider a pipe-flow experiment, with tracers moving from side to side in multiple passes / streamlines. First locate the tracers, then split their trajectories into each individual pass:

```
>>> import pept
>>> from pept.tracking import *
>>>
>>> split_pipe = pept.Pipeline([
>>>     Segregate(window = 10, max_distance = 20),  # Appends label column
>>>     GroupBy("label"),                           # Splits into samples
>>>     Reorient(),                                 # Align with X axis
>>>     Center(),                                   # Center points at 0
>>>     Stack(),
>>> ])
>>> streamlines = split_pipe.fit(trajectories)
```

Now each sample in *streamlines* corresponds to a single tracer pass, e.g. *streamlines[0]* is the first pass, *streamlines[1]* is the second. The passes were reoriented and centred such that the pipe is aligned with the X axis.

Now the *LaceyColorsLinear* reducer can be used to create images of the mixing between the pipe entrance and exit:

```
>>> from pept.processing import LaceyColorsLinear
>>> entrance = [-100, 0, 0]     # Pipe data was aligned with X and centred
>>> exit = [100, 0, 0]
>>> LaceyColorsLinear(
>>>     directory = "lacey",    # Creates directory and saves images there
>>>     p1 = entrance,
>>>     p2 = exit,
>>> ).fit(streamlines)
```

Now the directory “lacey” was created inside your current working folder, and all Lacey images saved there as “frame0000.png”, “frame0001.png”, etc. You can stitch all images together into a video using *pept.plots.make_video*:

```
>>> import pept
>>> pept.plots.make_video("lacey/frame*.png", output = "lacey/video.avi")
```

`__init__`(*directory*, *p1*, *p2*, *xlim=None*, *ylim=None*, *num_divisions=50*, *max_distance=10*, *resolution=(8, 8)*, *height=1000*, *width=1000*, *prefix='frame'*)

Methods

`__init__`(directory, p1, p2[, xlim, ylim, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(trajectories[, executor, max_workers, ...])	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*trajectories*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.RelativeDeviations

class pept.processing.**RelativeDeviations**(*p1*, *p2*, *ax1=None*, *ax2=None*, *max_distance=10*, *histogram=True*, ***kwargs*)

Bases: *Reducer*

Compute a Lagrangian mixing measure - the changes in tracer distances to a point P1 as they pass through an "inlet" plane and another point P2 when reaching an "outlet" plane.

A deviation is computed for each tracer trajectory, yielding a range of deviations that can e.g be histogrammed (default). Intuitively, mixing is stronger if this distribution of deviations is wider.

Reducer signature:

```
If ``histogram = True`` (default)
       PointData -> LaceyColors.fit -> plotly.graph_objs.Figure
 list[PointData] -> LaceyColors.fit -> plotly.graph_objs.Figure
list[np.ndarray] -> LaceyColors.fit -> plotly.graph_objs.Figure

If ``histogram = False`` (return deviations)
       PointData -> LaceyColors.fit -> (N,) np.ndarray
 list[PointData] -> LaceyColors.fit -> (N,) np.ndarray
list[np.ndarray] -> LaceyColors.fit -> (N,) np.ndarray
```

Each sample in the input `PointData` is treated as a separate streamline / tracer pass. You can group passes using `Segregate + GroupBy("label")`.

The first plane where the distances from tracers to a point *p1* is defined by the point *p1* and direction axis *ax1*. **The point `p1` should be the middle of the pipe**.

The second plane where relative distances are evaluated is similarly defined by *p2* and *ax2*. **The point `p2` should be the middle of the volume / pipe**.

If the direction vectors *ax1* and *ax2* are undefined (*None*), the tracers are assumed to follow a straight line between *p1* and *p2*.

The *max_distance* parameter defines the maximum distance allowed between a point and a plane to be considered part of it. The *resolution* defines the number of pixels in the height and width of the resulting image.

The following attributes are always set. A Plotly figure is only generated and returned if *histogram = True* (default).

The extra keyword arguments `**kwargs` are passed to the histogram creation routine *pept.plots.histogram*. You can e.g. set the YAxis limits by adding *ylim = [0, 20]*.

New in pept-0.5.1

Examples

Consider a pipe-flow experiment, with tracers moving from side to side in multiple passes / streamlines. First locate the tracers, then split their trajectories into each individual pass:

```
>>> import pept
>>> from pept.tracking import *
>>>
>>> split_pipe = pept.Pipeline([
>>>     Segregate(window = 10, max_distance = 20),  # Appends label column
>>>     GroupBy("label"),                           # Splits into samples
>>>     Reorient(),                                 # Align with X axis
>>>     Center(),                                   # Center points at 0
>>>     Stack(),
>>> ])
>>> streamlines = split_pipe.fit(trajectories)
```

Now each sample in *streamlines* corresponds to a single tracer pass, e.g. *streamlines[0]* is the first pass, *streamlines[1]* is the second. The passes were reoriented and centred such that the pipe is aligned with the X axis.

Now the *RelativeDeviations* reducer can be used to create a histogram of tracer deviations due to mixing:

```
>>> from pept.processing import RelativeDeviations
>>> entrance = [-100, 0, 0]     # Pipe data was aligned with X and centred
>>> exit = [100, 0, 0]
>>> fig = RelativeDeviations(entrance, exit).fit(streamlines)
>>> fig.show()
```

The deviations themselves can be extracted directly for further processing:

```
>>> mixing_algorithm = RelativeDeviations(entrance, exit, histogram=False)
>>> mixing_algorithm.fit(streamlines)
>>> deviations = mixing_algorithm.deviations
>>> inlet_points = mixing_algorithm.points1
>>> outlet_points = mixing_algorithm.points2
```

Attributes

points1
[*pept.PointData*] The tracer points selected at the inlet around *p1*.

points2
[*pept.PointData*] The tracer points selected at the outlet around *p2*.

deviations
[(N,) `np.ndarray`] The vector of tracer deviations for each tracer pass in *points1* and *points2*.

`__init__`(*p1*, *p2*, *ax1=None*, *ax2=None*, *max_distance=10*, *histogram=True*, ***kwargs*)

Methods

`__init__`(p1, p2[, ax1, ax2, max_distance, ...])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(trajectories[, executor, max_workers, ...])	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*trajectories*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.RelativeDeviationsLinear

class pept.processing.**RelativeDeviationsLinear**(*directory*, *p1*, *p2*, *num_divisions=50*, *max_distance=10*, *height=1000*, *width=2000*, *prefix='frame'*, ***kwargs*)

Bases: `Reducer`

Apply the *RelativeDeviations* mixing algorithm at *num_divisions* equidistant points between *p1* and *p2*, saving histogram images at each step in *directory*.

Reducer signature:

```
      PointData -> LaceyColors.fit -> plotly.graph_objs.Figure
 list[PointData] -> LaceyColors.fit -> plotly.graph_objs.Figure
list[np.ndarray] -> LaceyColors.fit -> plotly.graph_objs.Figure
```

For details about the mixing algorithm itself, check the *RelativeDeviations* documentation.

This algorithm saves a rich set of data:

- Individual histograms for each point along P1-P2 are saved in the given *directory*.
- A Plotly figure of computed statistics is returned, including the deviations' mean, standard deviation, skewness and kurtosis.
- The raw data is saved as object attributes (see below).

The generated images (saved in *directory* with *height* x *width* pixels) can be stitched into a video using *pept.plots.make_video*.

The extra keyword arguments `**kwargs` are passed to the histogram creation routine *pept.plots.histogram*. You can e.g. set the YAxis limits by adding *ylim = [0, 20]*.

New in pept-0.5.1

Examples

Consider a pipe-flow experiment, with tracers moving from side to side in multiple passes / streamlines. First locate the tracers, then split their trajectories into each individual pass:

```
>>> import pept
>>> from pept.tracking import *
>>>
>>> split_pipe = pept.Pipeline([
>>>     Segregate(window = 10, max_distance = 20),  # Appends label column
>>>     GroupBy("label"),                           # Splits into samples
>>>     Reorient(),                                 # Align with X axis
>>>     Center(),                                   # Center points at 0
>>>     Stack(),
>>> ])
>>> streamlines = split_pipe.fit(trajectories)
```

Now each sample in *streamlines* corresponds to a single tracer pass, e.g. *streamlines[0]* is the first pass, *streamlines[1]* is the second. The passes were reoriented and centred such that the pipe is aligned with the X axis.

Now the *RelativeDeviationsLinear* reducer can be used to create images of the mixing between the pipe entrance and exit:

```
>>> from pept.processing import RelativeDeviationsLinear
>>> entrance = [-100, 0, 0]     # Pipe data was aligned with X and centred
>>> exit = [100, 0, 0]
>>> summary_fig = RelativeDeviationsLinear(
>>>     directory = "deviations",   # Creates directory to save images
>>>     p1 = entrance,
>>>     p2 = exit,
>>> ).fit(streamlines)
>>> summary_fig.show()              # Summary statistics: mean, std, etc.
```

Now the directory "deviations" was created inside your current working folder, and all relative deviation histograms were saved there as "frame0000.png", "frame0001.png", etc. You can stitch all images together into a video using *pept.plots.make_video*:

```
>>> import pept
>>> pept.plots.make_video(
>>>     "deviations/frame*.png",
>>>     output = "deviations/video.avi"
>>> )
```

The raw deviations and statistics can also be extracted directly:

```
>>> mixing_algorithm = RelativeDeviationsLinear(
>>>     directory = "deviations",   # Creates directory to save images
```

(continues on next page)

(continued from previous page)

```
>>>     p1 = entrance,
>>>     p2 = exit,
>>> )
>>> fig = mixing_algorithm.fit(streamlines)
>>> fig.show()
```

```
>>> deviations = mixing_algorithm.deviations
>>> mean = mixing_algorithm.mean
>>> std = mixing_algorithm.std
>>> skew = mixing_algorithm.skew
>>> kurtosis = mixing_algorithm.kurtosis
```

Attributes

deviations
[`list`[(N,) `np.ndarray`]] A list of deviations computed by *RelativeDeviations* at each point between P1 and P2.

mean
[(N,) `np.ndarray`] A vector of mean tracer deviations at each point between P1 and P2.

std
[(N,) `np.ndarray`] A vector of the tracer deviations' standard deviation at each point between P1 and P2.

skew
[(N,) `np.ndarray`] A vector of the tracer deviations' adjusted skewness at each point between P1 and P2. A normal distribution has a value of 0; positive values indicate a longer right distribution tail; negative values indicate a heavier left tail.

kurtosis
[(N,) `np.ndarray`] A vector of the tracer deviations' Fisher kurtosis at each point between P1 and P2. A normal distribution has a value of 0; positive values indicate a "thin" distribution; negative values indicate a heavy, wide distribution.

`__init__`(*directory*, *p1*, *p2*, *num_divisions=50*, *max_distance=10*, *height=1000*, *width=2000*, *prefix='frame'*, ***kwargs*)

Methods

`__init__`(directory, p1, p2[, num_divisions, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(trajectories[, executor, max_workers, ...])	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*trajectories*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.AutoCorrelation

class pept.processing.**AutoCorrelation**(*lag='t'*, *signals=['vx', 'vy', 'vz']*, *span=None*, *num_divisions=500*, *max_distance=10*, *normalize=False*, *preprocess=True*, ***kwargs*)

Bases: `Reducer`

Compute autocorrelation of multiple measures (eg YZ velocities) as a function of a lagging variable (eg time).

Reducer signature:

```
      PointData -> AutoCorrelation.fit -> PlotlyGrapher2D
 list[PointData] -> AutoCorrelation.fit -> PlotlyGrapher2D
list[np.ndarray] -> AutoCorrelation.fit -> PlotlyGrapher2D
```

Each sample in the input `PointData` is treated as a separate streamline / tracer pass. You can group passes using `Segregate + GroupBy("label")`.

Autocorrelation and autocovariance each refer to about 3 different things in each field. The formula used here, inspired by the VACF in molecular dynamics and generalised for arbitrary measures, is:

$$C(L_i) = \frac{\sum_N V(L_0) \cdot V(L_i)}{N}$$

i.e. the autocorrelation C at a lag of Li is the average of the dot products of quantities V for all N tracers. For example, the velocity autocorrelation function with respect to time would be the average of *vx(0) vx(t) + vy(0) vy(t) + vz(0) vz(t)* at a given time *t*.

The input *lag* defines the column used as a lagging variable; it can be given as a named column string (e.g. *"t"*) or index (e.g. *0*).

The input *signals* define the quantities for which the autocorrelation is computed, given as a list of column names (e.g. *["vy", "vz"]*) or indices (e.g. *[5, 6]*).

The input *span*, if defined, is the minimum and maximum values for the *lag* (e.g. start and end times) for which the autocorrelation will be computed. By default it is automatically computed as the range of values.

The input *num_divisions* is the number of lag points between *span[0]* and *span[1]* for which the autocorrelation will be computed.

The *max_distance* parameter defines the maximum distance allowed between a lag value and the closest trajectory value for it to be considered.

If *normalize* is True, then the formula used becomes:

$$C(L_i) = \frac{\sum_N V(L_0) \cdot V(L_i) / V(L_0) \cdot V(L_0)}{N}$$

If *preprocess* is True, then the times of each tracer pass is taken relative to its start; only relevant if using time as the lagging variable.

The extra keyword arguments `**kwargs` are passed to *PlotlyGrapher2D.add_points*. You can e.g. set the YAxis limits by adding *ylim = [0, 20]*.

The extra keyword arguments `**kwargs` are passed to *plotly.graph_objs.Scatter*. You can e.g. set a different colorscheme with "marker_colorscheme = 'Viridis'".

New in pept-0.5.1

Examples

Consider a pipe-flow experiment, with tracers moving from side to side in multiple passes / streamlines. First locate the tracers, then split their trajectories into each individual pass:

```
>>> import pept
>>> from pept.tracking import *
>>>
>>> split_pipe = pept.Pipeline([
>>>     Segregate(window = 10, max_distance = 20),  # Appends label column
>>>     GroupBy("label"),                           # Splits into samples
>>>     Reorient(),                                 # Align with X axis
>>>     Center(),                                   # Center points at 0
>>>     Velocity(7),                                # Compute vx, vy, vz
>>>     Stack(),
>>> ])
>>> streamlines = split_pipe.fit(trajectories)
```

Now each sample in *streamlines* corresponds to a single tracer pass, e.g. *streamlines[0]* is the first pass, *streamlines[1]* is the second. The passes were reoriented and centred such that the pipe is aligned with the X axis.

Now the *AutoCorrelation* algorithm can be used to compute the VACF:

```
>>> from pept.processing import AutoCorrelation
>>> fig = AutoCorrelation("t", ["vx", "vy", "vz"]).fit(streamlines)
>>> fig.show()
```

The radial velocity autocorrelation can be computed as a function of the pipe length (X axis as it was reoriented):

```
>>> entrance = -100
>>> exit = 100
>>> ac = AutoCorrelation("x", ["vy", "vz"], span = [entrance, exit])
>>> ac.fit(streamlines).show()
```

The raw lags and autocorrelations plotted can be accessed directly:

```
>>> ac.lags
>>> ac.correlation
```

The radial location can be autocorrelated with time, then normalised to show periodic movements (e.g. due to a mixer):

```
>>> ac = AutoCorrelation("t", ["y", "z"], normalize = True)
>>> ac.fit(streamlines).show()
```

`__init__`(*lag='t'*, *signals=['vx', 'vy', 'vz']*, *span=None*, *num_divisions=500*, *max_distance=10*, *normalize=False*, *preprocess=True*, ***kwargs*)

Methods

__init__([lag, signals, span, ...])	
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.
fit(trajectories[, executor, max_workers, ...])	
load(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
save(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

`fit`(*trajectories*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

pept.processing.SpatialProjections

class pept.processing.**SpatialProjections**(*directory*, *start*, *end*, *dimension='x'*, *num_divisions=500*, *max_distance=10*, *colorbar_col=-1*, *height=1000*, *width=1000*, *prefix='frame'*, ***kwargs*)

Bases: `Reducer`

Project multiple tracer passes onto a moving 2D plane along a given *direction* between *start* and *end* coordinates, saving each frame in *directory*.

Reducer signature:

```
       PointData -> SpatialProjections.fit -> None
 list[PointData] -> SpatialProjections.fit -> None
list[np.ndarray] -> SpatialProjections.fit -> None
```

Each sample in the input `PointData` is treated as a separate streamline / tracer pass. You can group passes using `Segregate + GroupBy("label")`.

The generated images (saved in *directory* with *height* x *width* pixels) can be stitched into a video using *pept.plots.make_video*.

The extra keyword arguments `**kwargs` are passed to the histogram creation routine *pept.plots.histogram*. You can e.g. set the YAxis limits by adding *ylim = [0, 20]*.

New in pept-0.5.1

Examples

Consider a pipe-flow experiment, with tracers moving from side to side in multiple passes / streamlines. First locate the tracers, then split their trajectories into each individual pass:

```
>>> import pept
>>> from pept.tracking import *
>>>
>>> split_pipe = pept.Pipeline([
>>>     Segregate(window = 10, max_distance = 20),  # Appends label column
>>>     GroupBy("label"),                           # Splits into samples
>>>     Reorient(),                                 # Align with X axis
>>>     Center(),                                   # Center points at 0
>>>     Stack(),
>>> ])
>>> streamlines = split_pipe.fit(trajectories)
```

Now each sample in *streamlines* corresponds to a single tracer pass, e.g. *streamlines[0]* is the first pass, *streamlines[1]* is the second. The passes were reoriented and centred such that the pipe is aligned with the X axis.

Now the *RelativeDeviationsLinear* reducer can be used to create images of the mixing between the pipe entrance and exit:

```
>>> from pept.processing import SpatialProjections
>>> entrance_x = -100                   # Pipe data was aligned with X
>>> exit_x = 100
>>> SpatialProjections(
>>>     directory = "projections",      # Creates directory to save images
>>>     start = entrance_x,
>>>     end = exit_x,
>>> ).fit(streamlines)
```

Now the directory "projections" was created inside your current working folder, and eachc projected frame was saved there as "frame0000.png", "frame0001.png", etc. You can stitch all images together into a video using *pept.plots.make_video*:

```
>>> import pept
>>> pept.plots.make_video(
>>>     "projections/frame*.png",
>>>     output = "projections/video.avi"
>>> )
```

The raw projections can also be extracted directly:

```
>>> sp = SpatialProjections(
>>>     directory = "projections",    # Creates directory to save images
>>>     p1 = entrance_x,
>>>     p2 = exit_x,
>>> )
>>> sp.fit(streamlines)
>>> sp.projections
```

Attributes

projections
[`list`[(N, 5), `np.ndarray`]] A list of frames for each division between *start* and *end*, with each frame saving 5 columns [t, x, y, z, colorbar_col].

`__init__`(*directory*, *start*, *end*, *dimension='x'*, *num_divisions=500*, *max_distance=10*, *colorbar_col=-1*, *height=1000*, *width=1000*, *prefix='frame'*, ***kwargs*)

Methods

`__init__`(directory, start, end[, dimension, ...])	
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`fit`(trajectories[, executor, max_workers, ...])	
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.

copy(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

fit(*trajectories*, *executor='joblib'*, *max_workers=None*, *verbose=True*)

static load(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject subclass instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

save(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

Visualisation (`pept.plots`)

PEPT-oriented visulisation tools.

Visualisation functions and classes for PEPT data, transparently working with both *pept* base classes and raw NumPy arrays (e.g. *PlotlyGrapher.add_lines* handles both *pept.LineData* and (N, 7) NumPy arrays).

The *PlotlyGrapher* class creates interactive, publication-ready 3D figures with optional subplots which can also be exported to portable HTML files. The *PlotlyGrapher2D* class is its two-dimensional counterpart, handling e.g. *pept.Pixels*.

pept.plots.format_fig(fig[, size, font, ...])	Format a Plotly figure to a consistent theme for the Nature Computational Science journal.
pept.plots.histogram(data[, nbins, ...])	Create histogram of data with PEPT-relevant defaults for *plotly.express.histogram*.
pept.plots.make_video(frames[, output, fps, ...])	Stitch multiple images from *frames* into a video saved to *output*.
pept.plots.PlotlyGrapher([rows, cols, xlim, ...])	A class for PEPT data visualisation using Plotly-based 3D graphs.
pept.plots.PlotlyGrapher2D([rows, cols, ...])	A class for PEPT data visualisation using Plotly-based 2D graphs.

pept.plots.format_fig

pept.plots.**format_fig**(*fig*, *size=20*, *font='Computer Modern'*, *template='plotly_white'*)

Format a Plotly figure to a consistent theme for the Nature Computational Science journal.

pept.plots.histogram

pept.plots.**histogram**(*data*, *nbins=None*, *histnorm='percent'*, *marginal='box'*, *xlim=None*, *ylim=None*, *xaxis_title=None*, *yaxis_title=None*, ***kwargs*)

Create histogram of data with PEPT-relevant defaults for *plotly.express.histogram*.

You can check the official documentation for all available options: https://plotly.github.io/plotly.py-docs/generated/plotly.express.histogram.html.

Parameters

data
[(N,) numpy.ndarray-like] A 1D vector of values to histogram.

nbins
[`int`, optional] Positive integer. Sets the number of bins.

histnorm
[`str`, `default` "percent"] One of *'percent'*, *'probability'*, *'density'*, or *'probability density'* If *None*, the output of *histfunc* is used as is. If *'probability'*, the output of *histfunc* for a given bin is divided by the sum of the output of *histfunc* for all bins. If *'percent'*, the output of *histfunc* for a given bin is divided by the sum of the output of *histfunc* for all bins and multiplied by 100. If *'density'*, the output of *histfunc* for a given bin is divided by the size of the bin. If *'probability density'*, the output of *histfunc* for a given bin is normalized such that it corresponds to the probability that a random event whose distribution is described by the output of *histfunc* will fall into that bin.

marginal
[`str`, `default` "box"] One of *'rug'*, *'box'*, *'violin'*, or *'histogram'*. If set, a subplot is drawn alongside the main plot, visualizing the distribution.

xlim
[`list of two numbers`, optional] If provided, overrides auto-scaling on the x-axis in cartesian coordinates.

ylim
[`list of two numbers`, optional] If provided, overrides auto-scaling on the y-axis in cartesian coordinates.

xaxis_title
[`str`, optional] X-axis label.

yaxis_title
[`str`, optional] Y-axis label.

pept.plots.make_video

`pept.plots.make_video`(*frames*, *output='video.avi'*, *fps=10*, *verbose=True*)

Stitch multiple images from *frames* into a video saved to *output*.

Parameters

frames
[`str` or `list[str]`] Either a prefix for the frame names (e.g. "directory/frame*.png") or a list of paths to individual frames.

output
[`str`, `default` "video.avi"] Name of output video.

fps
[`int`, `default` 10] Number of frames per second.

Examples

Stitch all files matching a glob prefix: >>> from pept.plots import make_video >>> make_video("lacey/frame*.png", "lacey/video.avi")

Stitch individual files: >>> make_video(["frame0.png", "frame1.png", "frame2.png"])

pept.plots.PlotlyGrapher

class pept.plots.**PlotlyGrapher**(*rows=1*, *cols=1*, *xlim=None*, *ylim=None*, *zlim=None*, *subplot_titles=[' ']*)

Bases: *PEPTObject*

A class for PEPT data visualisation using Plotly-based 3D graphs.

The **PlotlyGrapher** class can create and automatically configure an arbitrary number of 3D subplots for PEPT data visualisation. They are by default set to use the *alternative PEPT 3D axes convention* - having the *y*-axis pointing upwards, such that the vertical screens of a PEPT scanner represent the *xy*-plane.

This class can be used to draw 3D scatter or line plots, with optional colour-coding using extra data columns (e.g. relative tracer activity or trajectory label).

It also provides easy access to the most common configuration parameters for the plots, such as axes limits, subplot titles, colorbar titles, etc. It can work with pre-computed Plotly traces (such as the ones from the *pept* base classes), as well as with numpy arrays.

Raises

ValueError
If *xlim*, *ylim* or *zlim* are not lists of length 2.

Examples

The figure is created when instantiating the class.

```
>>> grapher = PlotlyGrapher()
>>> lors = LineData(raw_lors...)        # Some example lines
>>> points = PointData(raw_points...)   # Some example points
```

Creating a trace based on a numpy array:

```
>>> sample_lors = lors[0]           # A numpy array of a single sample
>>> sample_points = points[0]
>>> grapher.add_lines(sample_lors)
>>> grapher.add_points(sample_points)
```

Showing the plot:

```
>>> grapher.show()
```

If you'd like to show the plot in your browser, you can set the default Plotly renderer:

```
>>> import plotly
>>> plotly.io.renderers.default = "browser"
```

Return pre-computed traces that you can add to other figures:

```
>>> PlotlyGrapher.lines_trace(lines)
>>> PlotlyGrapher.points_trace(points)
```

More examples are given in the docstrings of the *add_points*, *add_lines* methods.

Attributes

xlim
[`list` or `numpy.ndarray`] A list of length 2, formatted as *[x_min, x_max]*, where *x_min* is the lower limit of the x-axis of all the subplots and *x_max* is the upper limit of the x-axis of all the subplots.

ylim
[`list` or `numpy.ndarray`] A list of length 2, formatted as *[y_min, y_max]*, where *y_min* is the lower limit of the y-axis of all the subplots and *y_max* is the upper limit of the y-axis of all the subplots.

zlim
[`list` or `numpy.ndarray`] A list of length 2, formatted as *[z_min, z_max]*, where *z_min* is the lower limit of the z-axis of all the subplots and *z_max* is the upper limit of the z-axis of all the subplots.

fig
[`Plotly.Figure instance`] A Plotly.Figure instance, with any number of subplots (as defined by *rows* and *cols*) pre-configured for PEPT data.

`__init__`(*rows=1*, *cols=1*, *xlim=None*, *ylim=None*, *zlim=None*, *subplot_titles=[' ']*)

PlotlyGrapher class constructor.

Parameters

rows
[`int`, optional] The number of rows of subplots. The default is 1.

cols
[`int`, optional] The number of columns of subplots. The default is 1.

xlim
[`list` or `numpy.ndarray`, optional] A list of length 2, formatted as *[x_min, x_max]*, where *x_min* is the lower limit of the x-axis of all the subplots and *x_max* is the upper limit of the x-axis of all the subplots.

ylim
[`list` or `numpy.ndarray`, optional] A list of length 2, formatted as *[y_min, y_max]*, where *y_min* is the lower limit of the y-axis of all the subplots and *y_max* is the upper limit of the y-axis of all the subplots.

zlim
[`list` or `numpy.ndarray`, optional] A list of length 2, formatted as *[z_min, z_max]*, where *z_min* is the lower limit of the z-axis of all the subplots and *z_max* is the upper limit of the z-axis of all the subplots.

subplot_titles
[`list of str, default` [” “]] A list of the titles of the subplots - e.g. [“plot a)”, “plot b)”]. The default is a list of empty strings.

Raises

`ValueError`
If *rows* < 1 or *cols* < 1.

`ValueError`
If *xlim*, *ylim* or *zlim* are not lists of length 2.

Methods

`__init__`([rows, cols, xlim, ylim, zlim, ...])	*PlotlyGrapher* class constructor.
`add_lines`(lines[, row, col, width, color, ...])	Create and plot a trace for all the lines in a numpy array or *pept.LineData*, with possible color-coding.
`add_pixels`(pixels[, row, col, condition, ...])	Create and plot a trace with all the pixels in this class, with possible filtering.
`add_points`(points[, row, col, size, color, ...])	Create and plot a trace for all the points in a numpy array or *pept.PointData*, with possible color-coding.
`add_trace`(trace[, row, col])	Add a precomputed Plotly trace to a given subplot.
`add_traces`(traces[, row, col])	Add a list of precomputed Plotly traces to a given subplot.
`add_voxels`(voxels[, row, col, condition, ...])	Create and plot a trace for all the voxels in a *pept.Voxels* instance, with possible filtering.
`copy`([deep])	Create a deep copy of an instance of this class, including all inner attributes.
`create_figure`()	Create a Plotly figure, pre-configured for PEPT data.
`equalise_axes`()	Equalise the axes of all subplots by setting the system limits *xlim* and *ylim* to equal values, such that all data plotted is within the plotted bounds.
`lines_trace`(lines[, width, color, opacity, ...])	Static method for creating a Plotly trace of lines.
`load`(filepath)	Load a saved / pickled *PEPTObject* object from *filepath*.
`points_trace`(points[, size, color, opacity, ...])	Static method for creating a Plotly trace of points.
`save`(filepath)	Save a *PEPTObject* instance as a binary *pickle* object.
`show`([equal_axes])	Show the Plotly figure, optionally setting equal axes limits.
`to_html`(filepath[, equal_axes, include_plotlyjs])	Save the current Plotly figure as a self-contained HTML webpage.
`xlabel`(label[, row, col])	
`ylabel`(label[, row, col])	
`zlabel`(label[, row, col])	

Attributes

`fig`
`xlim`
`ylim`
`zlim`

create_figure()

Create a Plotly figure, pre-configured for PEPT data.

This function creates a Plotly figure with an arbitrary number of subplots, as given in the class instantiation call. It configures them to have the *y*-axis pointing upwards, as per the PEPT 3D axes convention. It also sets the axes limits and labels.

Returns

fig
[`Plotly Figure instance`] A Plotly Figure instance, with any number of subplots (as defined when instantiating the class) pre-configured for PEPT data.

`property xlim`

`property ylim`

`property zlim`

`property fig`

`xlabel`(*label*, *row=1*, *col=1*)

`ylabel`(*label*, *row=1*, *col=1*)

`zlabel`(*label*, *row=1*, *col=1*)

`static points_trace`(*points*, *size=2.0*, *color=None*, *opacity=0.8*, *colorbar=True*, *colorbar_col=-1*, *colorscale='Magma'*, *colorbar_title=None*, ***kwargs*)

Static method for creating a Plotly trace of points. See *PlotlyGrapher.add_points* for the full documentation.

`add_points`(*points*, *row=1*, *col=1*, *size=2.0*, *color=None*, *opacity=0.8*, *colorbar=True*, *colorbar_col=-1*, *colorscale='Magma'*, *colorbar_title=None*, ***kwargs*)

Create and plot a trace for all the points in a numpy array or *pept.PointData*, with possible color-coding.

Creates a *plotly.graph_objects.Scatter3d* object for all the points included in the numpy array or *pept.PointData* instance (or subclass thereof!) *points* and adds it to the subplot determined by *row* and *col*.

The expected data row is [time, x1, y1, z1, . . .].

Parameters

points
[(`M`, `N` >= 4) `numpy.ndarray` or *`pept.PointData`*] The expected data columns are: [time, x1, y1, z1, etc.]. If a *pept.PointData* instance (or subclass thereof) is received, the inner *points* will be used.

row
[`int`, `default` 1] The row of the subplot to add a trace to.

col
[`int`, `default` 1] The column of the subplot to add a trace to.

size
[`float`, `default` 2.0] The marker size of the points.

color
[`str` or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[`float`, `default` 0.8] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[bool, default True] If set to True, will color-code the data in the *points* column *colorbar_col*. Is overridden by *color* if set.

colorbar_col
[int, default -1] The column in *points* that will be used to color the points. Only has an effect if *colorbar* is set to True. The default is -1 (the last column).

colorscale
[str, default "Magma"] The Plotly scheme for color-coding the *colorbar_col* column in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

colorbar_title
[str, optional] If set, the colorbar will have this title above it.

Raises

`ValueError`
If *points* is not a numpy.ndarray with shape (M, N), where N >= 4.

Notes

If a colorbar is to be used (i.e. *colorbar = True* and *color = None*) and there are fewer than 10 unique values in the *colorbar_col* column in *points*, then the points for each unique label will be added as separate traces.

This is helpful for cases such as when plotting points with labelled trajectories, as when there are fewer than 10 trajectories, the distinct colours automatically used by Plotly when adding multiple traces allow the points to be better distinguished.

Examples

Add an array of points (data columns: [time, x, y, z]) to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher()
>>> points_raw = np.array(...)      # shape (N, M >= 4)
>>> grapher.add_points(points_raw)
>>> grapher.show()
```

Add all the points in a *PointData* instance:

```
>>> point_data = pept.PointData(...)    # Some example data
>>> grapher.add_points(point_data)
>>> grapher.show()
```

If you have an extremely large number of points in a numpy array, you can plot every 10th point using slices:

```
>>> pts = np.array(...)         # shape (N, M >= 4), N very large
>>> grapher.add_points(pts[::10])
```

static lines_trace(*lines*, *width=2.0*, *color=None*, *opacity=0.6*, *colorbar=True*, *colorbar_col=0*, *colorscale='Magma'*, *colorbar_title=None*)

Static method for creating a Plotly trace of lines. See *PlotlyGrapher.add_lines* for the full documentation.

`add_lines`(*lines*, *row=1*, *col=1*, *width=2.0*, *color=None*, *opacity=0.6*, *colorbar=True*, *colorbar_col=0*, *colorscale='Magma'*, *colorbar_title=None*)

Create and plot a trace for all the lines in a numpy array or *pept.LineData*, with possible color-coding.

Creates a *plotly.graph_objects.Scatter3d* object for all the lines included in the numpy array or *pept.LineData* instance (or subclass thereof!) *lines* and adds it to the subplot determined by *row* and *col*.

It expects LoR-like data, where each line is defined by two points. The expected data columns are [time, x1, y1, z1, x2, y2, z2, . . .].

Parameters

lines
[(`M`, `N` >= 7) `numpy.ndarray` or `pept.LineData`] The expected data columns: [time, x1, y1, z1, x2, y2, z2, etc.]. If a *pept.LineData* instance (or subclass thereof) is received, the inner *lines* will be used.

row
[`int`, `default` 1] The row of the subplot to add a trace to.

col
[`int`, `default` 1] The column of the subplot to add a trace to.

width
[`float`, `default` 2.0] The width of the lines.

color
[`str` or list-like, optional] Can be a single color (e.g. “black”, “rgb(122, 15, 241)”) or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[`float`, `default` 0.6] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[`bool`, `default` `True`] If set to True, will color-code the data in the *lines* column *colorbar_col*. Is overridden if *color* is set. The default is True, so that every line has a different color.

colorbar_col
[`int`, `default` 0] The column in the data samples that will be used to color the points. Only has an effect if *colorbar* is set to True. The default is 0 (the first column - time).

colorscale
[`str`, `default` “Magma”] The Plotly scheme for color-coding the *colorbar_col* column in the input data. Typical ones include “Cividis”, “Viridis” and “Magma”. A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

colorbar_title
[`str`, optional] If set, the colorbar will have this title above it.

Raises

`ValueError`
If *lines* is not a numpy.ndarray with shape (M, N), where N >= 7.

Examples

Add an array of lines (data columns: [t, x1, y1, z1, x2, y2, z2]) to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher()
>>> lines_raw = np.array(...)          # shape (N, M >= 7)
>>> grapher.add_lines(lines_raw)
>>> grapher.show()
```

Add all the lines in a *LineData* instance:

```
>>> line_data = pept.LineData(...)      # Some example data
>>> grapher.add_lines(line_data)
>>> grapher.show()
```

If you have a very large number of lines in a numpy array, you can plot every 10th point using slices:

```
>>> lines_raw = np.array(...)       # shape (N, M >= 7), N very large
>>> grapher.add_lines(lines_raw[::10])
```

add_pixels(*pixels*, *row=1*, *col=1*, *condition=<function PlotlyGrapher.<lambda>>*, *opacity=0.9*, *colorscale='Magma'*)

Create and plot a trace with all the pixels in this class, with possible filtering.

Creates a *plotly.graph_objects.Surface* object for the centres of all pixels encapsulated in a *pept.Pixels* instance, colour-coding the pixel value.

The *condition* parameter is a filtering function that should return a boolean mask (i.e. it is the result of a condition evaluation). For example *lambda x: x > 0* selects all pixels that have a value larger than 0.

Parameters

pixels
[`pept.Pixels`] The pixel space, encapsulated in a *pept.Pixels* instance (or subclass thereof). Only *pept.Pixels* are accepted as raw pixels on their own do not contain data about the spatial coordinates of the pixel box.

row
[`int`, `default` 1] The row of the subplot to add a trace to.

col
[`int`, `default` 1] The column of the subplot to add a trace to.

condition
[`function`, `default` *lambda pixels: pixels > 0*] The filtering function applied to the pixel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all pixels that should be plotted. The default, *lambda x: x > 0* selects all pixels which have a value larger than 0.

opacity
[`float`, `default` 0.4] The opacity of the surface, where 0 is transparent and 1 is fully opaque.

colorscale
[`str`, `default` "Magma"] The Plotly scheme for color-coding the voxel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

Examples

Pixellise an array of lines and add them to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher()
>>> lines = np.array(...)                    # shape (N, M >= 7)
>>> lines2d = lines[:, [0, 1, 2, 4, 5]]      # select x, y of lines
>>> number_of_pixels = [10, 10]
>>> pixels = pept.Pixels.from_lines(lines2d, number_of_pixels)
>>> grapher.add_lines(lines)
>>> grapher.add_trace(pixels.pixels_trace())
>>> grapher.show()
```

add_voxels(*voxels*, *row=1*, *col=1*, *condition=<function PlotlyGrapher.<lambda>>*, *size=4*, *color=None*, *opacity=0.4*, *colorbar=True*, *colorscale='Magma'*, *colorbar_title=None*)

Create and plot a trace for all the voxels in a *pept.Voxels* instance, with possible filtering.

Creates a *plotly.graph_objects.Scatter3d* object for the centres of all voxels encapsulated in a *pept.Voxels* instance, colour-coding the voxel value. The trace is added to the subplot determined by *row* and *col*.

The *condition* parameter is a filtering function that should return a boolean mask (i.e. it is the result of a condition evaluation). For example *lambda x: x > 0* selects all voxels that have a value larger than 0.

Parameters

voxels
[`pept.Voxels`] The voxel space, encapsulated in a *pept.Voxels* object.

row
[`int`, `default` 1] The row of the subplot to add a trace to.

col
[`int`, `default` 1] The column of the subplot to add a trace to.

condition
[`function`, `default` *lambda voxel_data: voxel_data > 0*] The filtering function applied to the voxel data before plotting it. It should return a boolean mask (a numpy array of the same shape, filled with True and False), selecting all voxels that should be plotted. The default, *lambda x: x > 0* selects all voxels which have a value larger than 0.

size
[`float`, `default` 4] The size of the plotted voxel points. Note that due to the large number of voxels in typical applications, the *voxel centres* are plotted as square points, which provides an easy to understand image that is also fast and responsive.

color
[`str` or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[`float`, `default` 0.4] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[`bool`, `default` `True`] If set to True, will color-code the voxel values. Is overridden if *color* is set.

colorscale
[`str`, `default` "Magma"] The Plotly scheme for color-coding the voxel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at

plotly.com/python/builtin-colorscales/. Only has an effect if *colorbar = True* and *color* is not set.

colorbar_title
[str, optional] If set, the colorbar will have this title above it.

Raises

TypeError
If *voxels* is not an instance of *pept.Voxels* or subclass thereof.

Examples

Voxellise an array of lines and add them to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher()
>>> lines = np.array(...)           # shape (N, M >= 7)
>>> number_of_voxels = [10, 10, 10]
>>> voxels = pept.Voxels(lines, number_of_voxels)
>>> grapher.add_lines(lines)
>>> grapher.add_voxels(voxels)
>>> grapher.show()
```

add_trace(*trace*, *row=1*, *col=1*)
Add a precomputed Plotly trace to a given subplot.

The equivalent of the Plotly figure.add_trace method.

Parameters

trace
[Plotly trace (Scatter3d)] A precomputed Plotly trace

row
[int, default 1] The row of the subplot to add a trace to.

col
[int, default 1] The column of the subplot to add a trace to.

add_traces(*traces*, *row=1*, *col=1*)
Add a list of precomputed Plotly traces to a given subplot.

The equivalent of the Plotly figure.add_traces method.

Parameters

traces
[list [Plotly trace (Scatter3d)]] A list of precomputed Plotly traces

row
[int, default 1] The row of the subplot to add the traces to.

col
[int, default 1] The column of the subplot to add the traces to.

equalise_axes()
Equalise the axes of all subplots by setting the system limits *xlim* and *ylim* to equal values, such that all data plotted is within the plotted bounds.

`copy`(*deep=True*)

Create a deep copy of an instance of this class, including all inner attributes.

`static load`(*filepath*)

Load a saved / pickled *PEPTObject* object from *filepath*.

Most often the full object state was saved using the *.save* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Returns

`pept.PEPTObject` `subclass` `instance`
The loaded object.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`save`(*filepath*)

Save a *PEPTObject* instance as a binary *pickle* object.

Saves the full object state, including inner attributes, in a portable binary format. Load back the object using the *load* method.

Parameters

filepath
[`filename` or `file handle`] If filepath is a path (rather than file handle), it is relative to where python is called.

Examples

Save a *LineData* instance, then load it back:

```
>>> lines = pept.LineData([[1, 2, 3, 4, 5, 6, 7]])
>>> lines.save("lines.pickle")
```

```
>>> lines_reloaded = pept.LineData.load("lines.pickle")
```

`show`(*equal_axes=True*)

Show the Plotly figure, optionally setting equal axes limits.

Note that the figure will be shown on the Plotly-configured renderer (e.g. browser, or PDF). The available renderers can be found by running the following code:

```
>>> import plotly.io as pio
>>> pio.renderers
```

If you want an interactive figure in the browser, run the following:

```
>>> pio.renderers.default = "browser"
```

Parameters

equal_axes
[`bool`, `default True`] Set *xlim*, *ylim*, *zlim* to equal ranges such that the axes limits are equalised. Only has an effect if *xlim*, *ylim* and *zlim* are all *None*. If *False*, the default Plotly behaviour is used (i.e. automatically use min, max for each dimension).

`to_html`(*filepath*, *equal_axes=True*, *include_plotlyjs=True*)
Save the current Plotly figure as a self-contained HTML webpage.

Parameters

filepath
[`str` or `writeable`] Path or open file descriptor to save the HTML file to.

equal_axes
[`bool`, `default True`] Set *xlim*, *ylim* to equal ranges such that the axes limits are equalised. Only has an effect if both *xlim* and *ylim* are *None*. If *False*, the default Plotly behaviour is used (i.e. automatically use min, max for each dimension).

include_plotlyjs
[`True` or "cdn", `default True`] If *True*, embed the Plotly.JS library in the HTML file, allowing the graph to be shown offline, but adding 3 MB. If "cdn", the Plotly.JS library will be downloaded dynamically.

Examples

Add 10 random points to a *PlotlyGrapher2D* instance and save the figure as an HTML webpage:

```
>>> fig = pept.visualisation.PlotlyGrapher2D()
>>> fig.add_points(np.random.random((10, 3)))
>>> fig.to_html("random_points.html")
```

pept.plots.PlotlyGrapher2D

class `pept.plots.`**`PlotlyGrapher2D`**(*rows=1*, *cols=1*, *xlim=None*, *ylim=None*, *subplot_titles=[' ']*, ***kwargs*)
Bases: `object`

A class for PEPT data visualisation using Plotly-based 2D graphs.

The **PlotlyGrapher** class can create and automatically configure an arbitrary number of 2D subplots for PEPT data visualisation.

This class can be used to draw 2D scatter or line plots, with optional colour-coding using extra data columns (e.g. relative tracer activity or trajectory label).

It also provides easy access to the most common configuration parameters for the plots, such as axes limits, subplot titles, colorbar titles, etc. It can work with pre-computed Plotly traces (such as the ones from the *pept* base classes), as well as with numpy arrays.

Examples

The figure is created when instantiating the class.

```
>>> import numpy as np
>>> from pept.visualisation import PlotlyGrapher2D
```

```
>>> grapher = PlotlyGrapher2D()
>>> lines = np.random.random((100, 5))      # columns [t, x1, y1, x2, y2]
>>> points = np.random.random((100, 3))     # columns [t, x, y]
```

Creating a trace based on a numpy array:

```
>>> grapher.add_lines(lines)
>>> grapher.add_points(points)
```

Showing the plot:

```
>>> grapher.show()
```

If you'd like to show the plot in your browser, you can set the default Plotly renderer:

```
>>> import plotly
>>> plotly.io.renderers.default = "browser"
```

Return pre-computed traces that you can add to other figures:

```
>>> PlotlyGrapher2D.lines_trace(lines)
>>> PlotlyGrapher2D.points_trace(points)
```

More examples are given in the docstrings of the *add_points*, *add_lines* methods.

Attributes

xlim
[`list` or `numpy.ndarray`] A list of length 2, formatted as *[x_min, x_max]*, where *x_min* is the lower limit of the x-axis of all the subplots and *x_max* is the upper limit of the x-axis of all the subplots.

ylim
[`list` or `numpy.ndarray`] A list of length 2, formatted as *[y_min, y_max]*, where *y_min* is the lower limit of the y-axis of all the subplots and *y_max* is the upper limit of the y-axis of all the subplots.

fig
[`Plotly.Figure instance`] A Plotly.Figure instance, with any number of subplots (as defined by *rows* and *cols*) pre-configured for PEPT data.

`__init__`(*rows=1*, *cols=1*, *xlim=None*, *ylim=None*, *subplot_titles=[' ']*, ***kwargs*)
PlotlyGrapher class constructor.

Parameters

rows
[`int`, optional] The number of rows of subplots. The default is 1.

cols
[`int`, optional] The number of columns of subplots. The default is 1.

xlim
[`list` or `numpy.ndarray`, optional] A list of length 2, formatted as *[x_min, x_max]*, where *x_min* is the lower limit of the x-axis of all the subplots and *x_max* is the upper limit of the x-axis of all the subplots.

ylim
[`list` or `numpy.ndarray`, optional] A list of length 2, formatted as *[y_min, y_max]*, where *y_min* is the lower limit of the y-axis of all the subplots and *y_max* is the upper limit of the y-axis of all the subplots.

subplot_titles
[`list of str`, `default` [" "]] A list of the titles of the subplots - e.g. ["plot a)", "plot b)"]. The default is a list of empty strings.

Raises

`ValueError`
If *rows* < 1 or *cols* < 1.

`ValueError`
If *xlim* or *ylim* are not lists of length 2.

Methods

__init__([rows, cols, xlim, ylim, ...])	*PlotlyGrapher* class constructor.
add_image(image, **kwargs)	Create and plot a *go.Image* trace.
add_lines(lines[, row, col, width, color, ...])	Create and plot a trace for all the lines in a numpy array, with possible color-coding.
add_pixels(pixels[, row, col, colorscale, ...])	Create and plot a trace with all the pixels in this class, with possible filtering.
add_points(points[, row, col, size, color, ...])	Create and plot a trace for all the points in a numpy array, with possible color-coding.
add_timeseries(points[, rows_cols, size, ...])	Add a timeseries plot for each dimension in *points* vs.
add_trace(trace[, row, col])	Add a precomputed Plotly trace to a given subplot.
add_traces(traces[, row, col])	Add a list of precomputed Plotly traces to a given subplot.
create_figure(**kwargs)	Create a Plotly figure, pre-configured for PEPT data.
equalise_axes()	Equalise the axes of all subplots by setting the system limits *xlim* and *ylim* to equal values, such that all data plotted is within the plotted bounds.
equalise_separate()	Equalise the axes of all subplots *individually* by setting the system limits in each dimension to equal values, such that all data plotted is within the plotted bounds.
lines_trace(lines[, width, color, opacity])	Static method for creating a Plotly trace of lines.
points_trace(points[, size, color, opacity, ...])	Static method for creating a Plotly trace of points.
show([equal_axes])	Show the Plotly figure, optionally setting equal axes limits.
timeseries_trace(points[, size, color, ...])	Static method for creating a list of 3 Plotly traces of timeseries.
to_html(filepath[, equal_axes, include_plotlyjs])	Save the current Plotly figure as a self-contained HTML webpage.
xlabel(label[, row, col])	
ylabel(label[, row, col])	

Attributes

fig
xlim
ylim

create_figure(***kwargs*)

Create a Plotly figure, pre-configured for PEPT data.

This function creates a Plotly figure with an arbitrary number of subplots, as given in the class instantiation call.

Returns

fig
[`Plotly Figure instance`] A Plotly Figure instance, with any number of subplots (as defined when instantiating the class) pre-configured for PEPT data.

`property xlim`

`property ylim`

`xlabel`(*label*, *row=1*, *col=1*)

`ylabel`(*label*, *row=1*, *col=1*)

`property fig`

`static timeseries_trace`(*points*, *size=6.0*, *color=None*, *opacity=0.8*, *colorbar=True*, *colorbar_col=-1*, *colorscale='Magma'*, *colorbar_title=None*, ***kwargs*)

Static method for creating a list of 3 Plotly traces of timeseries. See *PlotlyGrapher2D.add_timeseries* for the full documentation.

`add_timeseries`(*points*, *rows_cols=[(1, 1), (2, 1), (3, 1)]*, *size=6.0*, *color=None*, *opacity=0.8*, *colorbar=True*, *colorbar_col=-1*, *colorscale='Magma'*, *colorbar_title=None*, ***kwargs*)

Add a timeseries plot for each dimension in *points* vs. time.

If the current PlotlyGrapher2D figure does not have enough rows and columns to accommodate the three subplots (at coordinates *rows_cols*), the inner figure will be regenerated with enough rows and columns.

Parameters

points
[(`M`, `N` >= 4) `numpy.ndarray` or *`pept.PointData`*] The expected data columns are: [time, x1, y1, z1, etc.]. If a *pept.PointData* instance (or subclass thereof) is received, the inner *points* will be used.

rows_cols
[`list`[`tuple`[2]]] A list with 3 tuples, each tuple containing the subplot indices to plot the x, y, and z coordinates (indexed from 1).

size
[`float`, `default` 6.0] The marker size of the points.

color
[`str` or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[`float`, `default` 0.8] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[bool, `default` `True`] If set to True, will color-code the data in the *points* column *colorbar_col*. Is overridden by *color* if set.

colorbar_col
[`int`, `default` -1] The column in *points* that will be used to color the points. Only has an effect if *colorbar* is set to True. The default is -1 (the last column).

colorscale
[`str`, `default` "Magma"] The Plotly scheme for color-coding the *colorbar_col* column in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

colorbar_title
[str, optional] If set, the colorbar will have this title above it.

Raises

ValueError
If *points* is not a numpy.ndarray with shape (M, N), where N >= 4.

Notes

If a colorbar is to be used (i.e. *colorbar = True* and *color = None*) and there are fewer than 10 unique values in the *colorbar_col* column in *points*, then the points for each unique label will be added as separate traces.

This is helpful for cases such as when plotting points with labelled trajectories, as when there are fewer than 10 trajectories, the distinct colours automatically used by Plotly when adding multiple traces allow the points to be better distinguished.

Examples

Add an array of 3D points (data columns: [time, x, y, z]) to a *PlotlyGrapher2D* instance:

```
>>> grapher = PlotlyGrapher2D()
>>> points_raw = np.array(...)      # shape (N, M >= 4)
>>> grapher.add_timeseries(points_raw)
>>> grapher.show()
```

Add all the points in a *PointData* instance:

```
>>> point_data = pept.PointData(...)    # Some example data
>>> grapher.add_timeseries(point_data)
>>> grapher.show()
```

static points_trace(*points*, *size=2.0*, *color=None*, *opacity=0.8*, *colorbar=True*, *colorbar_col=-1*, *colorscale='Magma'*, *colorbar_title=None*, ***kwargs*)

Static method for creating a Plotly trace of points. See *PlotlyGrapher2D.add_points* for the full documentation.

add_points(*points*, *row=1*, *col=1*, *size=6.0*, *color=None*, *opacity=0.8*, *colorbar=True*, *colorbar_col=-1*, *colorscale='Magma'*, *colorbar_title=None*, ***kwargs*)

Create and plot a trace for all the points in a numpy array, with possible color-coding.

Creates a *plotly.graph_objects.Scatter* object for all the points included in the numpy array *points* and adds it to the subplot selected by *row* and *col*.

The expected data columns are [time, x1, y1, …].

Parameters

points
[(M, N >= 2) numpy.ndarray] Points to plot. The expected data columns are: [t, x1, y1, etc.].

row
[int, default 1] The row of the subplot to add a trace to.

col
[int, default 1] The column of the subplot to add a trace to.

size
[float, default 2.0] The marker size of the points.

color
[str or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)") or a colorbar list. Overrides *colorbar* if set. For more information, check the Plotly documentation. The default is None.

opacity
[float, default 0.8] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

colorbar
[bool, default True] If set to True, will color-code the data in the *points* column *colorbar_col*. Is overridden by *color* if set.

colorbar_col
[int, default -1] The column in *points* that will be used to color the points. Only has an effect if *colorbar* is set to True. The default is -1 (the last column).

colorscale
[str, default "Magma"] The Plotly scheme for color-coding the *colorbar_col* column in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

colorbar_title
[str, optional] If set, the colorbar will have this title above it.

Raises

`ValueError`
If *points* is not a numpy.ndarray with shape (M, N), where N >= 3.

Examples

Add an array of points (data columns: [time, x, y]) to a *PlotlyGrapher2D* instance:

```
>>> grapher = PlotlyGrapher2D()
>>> points_raw = np.random.random((10, 3))
>>> grapher.add_points(points_raw)
>>> grapher.show()
```

If you have an extremely large number of points in a numpy array, you can plot every 10th point using slices:

```
>>> pts = np.array(...)          # shape (N, M >= 3), N very large
>>> grapher.add_points(pts[::10])
```

static lines_trace(*lines*, *width=2.0*, *color=None*, *opacity=0.6*, ***kwargs*)
Static method for creating a Plotly trace of lines. See *PlotlyGrapher2D.add_lines* for the full documentation.

add_lines(*lines*, *row=1*, *col=1*, *width=2.0*, *color=None*, *opacity=0.6*, ***kwargs*)
Create and plot a trace for all the lines in a numpy array, with possible color-coding.

Creates a *plotly.graph_objects.Scatter* object for all the lines included in the numpy array *lines* and adds it to the subplot determined by *row* and *col*.

It expects LoR-like data, where each line is defined by two points. The expected data columns are [x1, y1, x2, y2, …].

Parameters

lines
[(M, N >= 5) numpy.ndarray] The expected data columns are: [time, x1, y1, x2, y2, etc.].

row
[int, default 1] The row of the subplot to add a trace to.

col
[int, default 1] The column of the subplot to add a trace to.

width
[float, default 2.0] The width of the lines.

color
[str or list-like, optional] Can be a single color (e.g. "black", "rgb(122, 15, 241)").

opacity
[float, default 0.6] The opacity of the lines, where 0 is transparent and 1 is fully opaque.

Raises

ValueError
If *lines* is not a numpy.ndarray with shape (M, N), where N >= 5.

Examples

Add an array of lines (data columns: [time, x1, y1, x2, y2]) to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher2D()
>>> lines_raw = np.random.random((100, 5))
>>> grapher.add_lines(lines_raw)
>>> grapher.show()
```

If you have a very large number of lines in a numpy array, you can plot every 10th point using slices:

```
>>> lines_raw = np.array(...)       # shape (N, M >= 5), N very large
>>> grapher.add_lines(lines_raw[::10])
```

add_pixels(*pixels*, *row=1*, *col=1*, *colorscale='Magma'*, *transpose=True*, *xgap=0.0*, *ygap=0.0*, ***kwargs*)
Create and plot a trace with all the pixels in this class, with possible filtering.

Creates a *plotly.graph_objects.Heatmap* object for the centres of all pixels encapsulated in a *pept.Pixels* instance, colour-coding the pixel value.

The *condition* parameter is a filtering function that should return a boolean mask (i.e. it is the result of a condition evaluation). For example *lambda x: x > 0* selects all pixels that have a value larger than 0.

Parameters

pixels
[*pept.Pixels*] The pixel space, encapsulated in a *pept.Pixels* instance (or subclass thereof). Only *pept.Pixels* are accepted as raw pixels on their own do not contain data about the spatial coordinates of the pixel box.

row
[int, default 1] The row of the subplot to add a trace to.

col
[int, default 1] The column of the subplot to add a trace to.

colorscale
[str, default "Magma"] The Plotly scheme for color-coding the pixel values in the input data. Typical ones include "Cividis", "Viridis" and "Magma". A full list is given at *plotly.com/python/builtin-colorscales/*. Only has an effect if *colorbar = True* and *color* is not set.

transpose
[bool, default True] Transpose the heatmap (i.e. flip it across its diagonal).

Examples

Pixellise an array of lines and add them to a *PlotlyGrapher* instance:

```
>>> grapher = PlotlyGrapher2D()
>>> lines = np.array(...)                   # shape (N, M >= 7)
>>> lines2d = lines[:, [0, 1, 2, 4, 5]]     # select x, y of lines
>>> number_of_pixels = [10, 10]
>>> pixels = pept.Pixels.from_lines(lines2d, number_of_pixels)
>>> grapher.add_lines(lines)
>>> grapher.add_pixels(pixels)
>>> grapher.show()
```

add_image(*image*, ***kwargs*)
Create and plot a *go.Image* trace.

Parameters

image
[(width, height, 3 or 4) np.ndarray] An image with 3 (RGB) or 4 (RGBA) channels.

****kwargs**
[keyword arguments] Other arguments to be passed to the plotly.graph_objs.Image constructor.

add_trace(*trace*, *row=1*, *col=1*)
Add a precomputed Plotly trace to a given subplot.

The equivalent of the Plotly figure.add_trace method.

Parameters

trace
[Plotly trace] A precomputed Plotly trace.

row
[int, default 1] The row of the subplot to add a trace to.

col
[int, default 1] The column of the subplot to add a trace to.

add_traces(*traces*, *row=1*, *col=1*)
Add a list of precomputed Plotly traces to a given subplot.

The equivalent of the Plotly figure.add_traces method.

Parameters

traces
[list [Plotly trace]] A list of precomputed Plotly traces

row
[int, default 1] The row of the subplot to add the traces to.

col
[int, default 1] The column of the subplot to add the traces to.

equalise_axes()

Equalise the axes of all subplots by setting the system limits *xlim* and *ylim* to equal values, such that all data plotted is within the plotted bounds.

equalise_separate()

Equalise the axes of all subplots *individually* by setting the system limits in each dimension to equal values, such that all data plotted is within the plotted bounds.

show(*equal_axes=True*)

Show the Plotly figure, optionally setting equal axes limits.

Note that the figure will be shown on the Plotly-configured renderer (e.g. browser, or PDF). The available renderers can be found by running the following code:

```
>>> import plotly.io as pio
>>> pio.renderers
```

If you want an interactive figure in the browser, run the following:

```
>>> pio.renderers.default = "browser"
```

Parameters

equal_axes
[bool, default True] Set *xlim*, *ylim* to equal ranges such that the axes limits are equalised. Only has an effect if both *xlim* and *ylim* are *None*. If *False*, the default Plotly behaviour is used (i.e. automatically use min, max for each dimension).

to_html(*filepath*, *equal_axes=True*, *include_plotlyjs=True*)

Save the current Plotly figure as a self-contained HTML webpage.

Parameters

filepath
[str or writeable] Path or open file descriptor to save the HTML file to.

equal_axes
[bool, default True] Set *xlim*, *ylim* to equal ranges such that the axes limits are equalised. Only has an effect if both *xlim* and *ylim* are *None*. If *False*, the default Plotly behaviour is used (i.e. automatically use min, max for each dimension).

include_plotlyjs
[True or "cdn", default True] If *True*, embed the Plotly.JS library in the HTML file, allowing the graph to be shown offline, but adding 3 MB. If "cdn", the Plotly.JS library will be downloaded dynamically.

Examples

Add 10 random points to a *PlotlyGrapher2D* instance and save the figure as an HTML webpage:

```
>>> fig = pept.visualisation.PlotlyGrapher2D()
>>> fig.add_points(np.random.random((10, 3)))
>>> fig.to_html("random_points.html")
```

pept.utilities

PEPT-oriented utility functions.

The utility functions include low-level optimised Cython functions (e.g. *find_cutpoints*) that are of common interest across the *pept* package, as well as I/O functions, parallel maps and pixel/voxel traversal algorithms.

Even though the functions are grouped in directories (subpackages) and files (modules), unlike the rest of the package, they are all imported into the *pept.utilities* root, so that their import paths are not too long.

pept.utilities.find_cutpoints(const double[, ...)	Compute the cutpoints from a given array of lines.
pept.utilities.find_minpoints(const double[, ...)	Compute the minimum distance points (MDPs) from all combinations of *num_lines* lines given in an array of lines *sample_lines*.
pept.utilities.group_by_column(data_array, ...)	Group the rows in a 2D *data_array* based on the unique values in a given *column_to_separate*, returning the groups as a list of numpy arrays.
pept.utilities.number_of_lines(...)	Return the number of lines (or rows) in a file.
pept.utilities.read_csv(filepath_or_buffer)	Read a given number of lines from a file and return a numpy array of the values.
pept.utilities.read_csv_chunks(...[, ...])	Read chunks of data from a file lazily, returning numpy arrays of the values.
pept.utilities.parallel_map_file(func, ...)	Utility for parallelising (read CSV chunk -> process chunk) workflows.
pept.utilities.traverse2d(double[, , ...)	Fast pixel traversal for 2D lines (or LoRs).
pept.utilities.traverse3d(double[, , , ...)	Fast voxel traversal for 3D lines (or LoRs).
pept.utilities.ChunkReader(...[, skiprows, ...])	Class for fast, on-demand reading / parsing and iteration over chunks of data from CSV files.

pept.utilities.find_cutpoints

pept.utilities.**find_cutpoints**(*const double[:, :] sample_lines*, *double max_distance*, *const double[:] cutoffs*, *bool append_indices=0*)

Compute the cutpoints from a given array of lines.

```
Function signature:
    find_cutpoints(
        double[:, :] sample_lines,  # LoRs in sample
        double max_distance,        # Max distance between two LoRs
        double[:] cutoffs,          # Spatial cutoff for cutpoints
        bint append_indices = False # Append LoR indices used
    )
```

This is a low-level Cython function that does not do any checks on the input data - it is meant to be used in other modules / libraries. For a normal user, the *pept.tracking.peptml* function *find_cutpoints* and class *Cutpoints* are recommended as higher-level APIs. They do check the input data and are easier to use (for example, they automatically compute the cutoffs).

A cutpoint is the point in 3D space that minimises the distance between any two lines. For any two non-parallel 3D lines, this point corresponds to the midpoint of the unique segment that is perpendicular to both lines.

This function considers every pair of lines in *sample_lines* and returns all the cutpoints that satisfy the following conditions:

1. The distance between the two lines is smaller than *max_distance*.
2. The cutpoints are within the *cutoffs*.

Parameters

sample_lines
[(`N`, `M` >= 7) `numpy.ndarray`] The sample of lines, where each row is [time, x1, y1, z1, x2, y2, z2], containing two points [x1, y1, z1] and [x2, y2, z2] defining an LoR.

max_distance
[`float`] The maximum distance between two LoRs for their cutpoint to be considered.

cutoffs
[(6,) `numpy.ndarray`] Only consider the cutpoints that fall within the cutoffs. *cutoffs* has the format [min_x, max_x, min_y, max_y, min_z, max_z].

append_indices
[bool, optional] If set to *True*, the indices of the individual LoRs that were used to compute each cutpoint is also appended to the returned array. Default is *False*.

Returns

cutpoints
[(`M`, 4) `or` (`M`, 6) `numpy.ndarray`] A numpy array of the calculated weighted cutpoints. If *append_indices* is *False*, then the columns are [time, x, y, z]. If *append_indices* is *True*, then the columns are [time, x, y, z, i, j], where *i* and *j* are the LoR indices from *sample_lines* that were used to compute the cutpoints. The time is the average between the timestamps of the two LoRs that were used to compute the cutpoint. The first column (for time) is sorted.

Examples

```
>>> import numpy as np
>>> from pept.utilities import find_cutpoints
>>>
>>> lines = np.random.random((500, 7)) * 500
>>> max_distance = 0.1
>>> cutoffs = np.array([0, 500, 0, 500, 0, 500], dtype = float)
>>>
>>> cutpoints = find_cutpoints(lines, max_distance, cutoffs)
```

pept.utilities.find_minpoints

pept.utilities.**find_minpoints**(*const double[:, :] sample_lines*, *Py_ssize_t num_lines*, *double max_distance*, *const double[:] cutoffs*, *bool append_indices=0*)

Compute the minimum distance points (MDPs) from all combinations of *num_lines* lines given in an array of lines *sample_lines*.

```
Function signature:
    find_minpoints(
        double[:, :] sample_lines,  # LoRs in sample
        Py_ssize_t num_lines,       # Number of LoRs in combinations
        double max_distance,        # Max distance from MDP to LoRs
        double[:] cutoffs,          # Spatial cutoff for minpoints
        bool append_indices = 0     # Append LoR indices used
    )
```

Given a sample of lines, this functions computes the minimum distance points (MDPs) for every possible combination of *num_lines* lines. The returned numpy array contains all MDPs that satisfy the following:

1. Are within the *cutoffs*.
2. Are closer to all the constituent LoRs than *max_distance*.

Parameters

sample_lines
[(M, N) numpy.ndarray] A 2D array of lines, where each line is defined by two points such that every row is formatted as *[t, x1, y1, z1, x2, y2, z2, etc.]*. It *must* have at least 2 lines and the combination size *num_lines must* be smaller or equal to the number of lines. Put differently: 2 <= num_lines <= len(sample_lines).

num_lines
[int] The number of lines in each combination of LoRs used to compute the MDP. This function considers every combination of *numlines* from the input *sample_lines*. It must be smaller or equal to the number of input lines *sample_lines*.

max_distance
[float] The maximum allowed distance between an MDP and its constituent lines. If any distance from the MDP to one of its lines is larger than *max_distance*, the MDP is thrown away.

cutoffs
[(6,) numpy.ndarray] An array of spatial cutoff coordinates with *exactly 6 elements* as [x_min, x_max, y_min, y_max, z_min, z_max]. If any MDP lies outside this region, it is thrown away.

append_indices
[bool] A boolean specifying whether to include the indices of the lines used to compute each MDP. If *False*, the output array will only contain the [time, x, y, z] of the MDPs. If *True*, the output array will have extra columns [time, x, y, z, line_idx(1), …, line_idx(n)] where n = *num_lines*.

Returns

minpoints
[(M, N) numpy.ndarray] A 2D array of *float`s containing the time and coordinates of the*

MDPs [time, x, y, z]. The time is computed as the average of the constituent lines. If `append_indices is *True*, then *num_lines* indices of the constituent lines are appended as extra columns: [time, x, y, z, line_idx1, line_idx2, ..].

Notes

There must be at least two lines in *sample_lines* and *num_lines* must be greater or equal to the number of lines (i.e. *len(sample_lines)*). Put another way: 2 <= num_lines <= len(sample_lines).

This is a low-level Cython function that does not do any checks on the input data - it is meant to be used in other modules / libraries. For a normal user, the *pept.tracking.peptml* function *find_minpoints* and class *Minpoints* are recommended as higher-level APIs. They do check the input data and are easier to use (for example, they automatically compute the cutoffs).

Examples

```
>>> import numpy as np
>>> from pept.utilities import find_minpoints
>>>
>>> lines = np.random.random((500, 7)) * 500
>>> num_lines = 3
>>> max_distance = 0.1
>>> cutoffs = np.array([0, 500, 0, 500, 0, 500], dtype = float)
>>>
>>> minpoints = find_minpoints(lines, num_lines, max_distance, cutoffs)
```

pept.utilities.group_by_column

`pept.utilities.`**`group_by_column`**(*data_array*, *column_to_separate*)

Group the rows in a 2D *data_array* based on the unique values in a given *column_to_separate*, returning the groups as a list of numpy arrays.

Parameters

data_array
[`(M, N) numpy.ndarray`] A generic 2D numpy array-like (will be converted using numpy.asarray).

column_to_separate
[`int`] The column index in *data_array* from which the unique values will be used for grouping.

Returns

groups
[`list` of `numpy.ndarray`] A list whose elements are 2D numpy arrays - these are sub-arrays from *data_array* for which the entries in the column *column_to_separate* are the same.

Raises

`ValueError`
If data_array does not have exactly 2 dimensions.

Examples

Separate a 6x3 numpy array based on the last column:

```
>>> x = np.array([
>>>     [1, 2, 1],
>>>     [5, 3, 1],
>>>     [1, 1, 2],
>>>     [5, 2, 1],
>>>     [2, 4, 2]
>>> ])
>>> x_sep = pept.utilities.group_by_column(x, -1)
>>> x_sep
>>> [array([[1, 2, 1],
>>>         [5, 3, 1],
>>>         [5, 2, 1]]),
>>>  array([[1, 1, 2],
>>>         [2, 4, 2]])]
```

pept.utilities.number_of_lines

`pept.utilities.`**`number_of_lines`**`(`*filepath_or_buffer*`)`

Return the number of lines (or rows) in a file.

Parameters

filepath_or_buffer
[`str`, `path object` or file-like `object`] Path to the file.

Returns

`int`
The number of lines in the file pointed at by *filepath_or_buffer*.

pept.utilities.read_csv

`pept.utilities.`**`read_csv`**`(`*filepath_or_buffer*, *skiprows=None*, *nrows=None*, *dtype=<class 'float'>*, *sep='\\s+'*, *header=None*, *engine='c'*, *na_filter=False*, *quoting=3*, *memory_map=True*, ***kwargs*`)`

Read a given number of lines from a file and return a numpy array of the values.

This is a convenience function that's simply a proxy to *pandas.read_csv*, configured with default parameters for fast reading and parsing of usual PEPT data.

Most importantly, it reads from a **space-separated values** file at *filepath_or_buffer*, optionally skipping *skiprows* lines and reading in *nrows* lines. It returns a *numpy.ndarray* with *float* values.

The parameters below are sent to *pandas.read_csv* with no further parsing. The descriptions below are taken from the *pandas* documentation.

Parameters

filepath_or_buffer
[`str`, `path object` or file-like `object`] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv. If you want to pass in a

path object, pandas accepts any *os.PathLike*. By file-like object, we refer to objects with a *read()* method, such as a file handler (e.g. via builtin *open* function) or *StringIO*.

skiprows
[list-like, `int` or `callable()`, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

nrows
[`int`, optional] Number of rows of file to read. Useful for reading pieces of large files.

dtype
[`Type name`, `default` *float*] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'}.

sep
[`str`, `default` *"s+"*] Delimiter to use. Separators longer than 1 character and different from 's+' will be interpreted as regular expressions and will also force the use of the Python parsing engine.

header
[`int`, `list of int`, "infer", optional] Row number(s) to use as the column names, and the start of the data. By default assume there is no header present (i.e. *header = None*).

engine
[{'c', 'python'}, `default` "c"] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

na_filter
[bool, `default` *True*] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

quoting
[`int` or csv.QUOTE_* `instance`, `default` *csv.QUOTE_NONE*] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

memory_map
[bool, `default True`] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

****kwargs**
[optional] Extra keyword arguments that will be passed to *pandas.read_csv*.

pept.utilities.read_csv_chunks

`pept.utilities.`**`read_csv_chunks`**(*filepath_or_buffer*, *chunksize*, *skiprows=None*, *nrows=None*, *dtype=<class 'float'>*, *sep='\\s+'*, *header=None*, *engine='c'*, *na_filter=False*, *quoting=3*, *memory_map=True*, ***kwargs*)

Read chunks of data from a file lazily, returning numpy arrays of the values.

This function returns a generator - an object that can be iterated over once, creating data on-demand. This means that chunks of data will be read only when being accessed, making it a more efficient alternative to *read_csv* for large files (> 1.000.000 lines).

A more convenient and feature-complete alternative is *pept.utilities.ChunkReader* which is more reusable and can access out-of-order chunks using subscript notation (i.e. data[0]).

This is a convenience function that's simply a proxy to *pandas.read_csv*, configured with default parameters for fast reading and parsing of usual PEPT data.

Most importantly, it lazily read chunks of size *chunksize* from a **space-separated values** file at *filepath_or_buffer*, optionally skipping *skiprows* lines and reading in *nrows* lines. It returns *numpy.ndarray`s with `float* values.

The parameters below are sent to *pandas.read_csv* with no further parsing. The descriptions below are taken from the *pandas* documentation.

Parameters

filepath_or_buffer
[`str`, `path object` or file-like `object`] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv. If you want to pass in a path object, pandas accepts any *os.PathLike*. By file-like object, we refer to objects with a *read()* method, such as a file handler (e.g. via builtin *open* function) or *StringIO*.

chunksize
[`int`] Number of lines read in a chunk of data. Return TextFileReader object for iteration.

skiprows
[list-like, `int` or `callable()`, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

nrows
[`int`, optional] Number of rows of file to read. Useful for reading pieces of large files.

dtype
[`Type name`, `default` *float*] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'}.

sep
[`str`, `default` *"s+"*] Delimiter to use. Separators longer than 1 character and different from 's+' will be interpreted as regular expressions and will also force the use of the Python parsing engine.

header
[`int`, `list of int`, "infer", optional] Row number(s) to use as the column names, and the start of the data. By default assume there is no header present (i.e. *header = None*).

engine
[{'c', 'python'}, `default` "c"] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

na_filter
[bool, `default` *True*] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

quoting
[`int` or csv.QUOTE_* `instance`, `default` *csv.QUOTE_NONE*] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

memory_map
[bool, `default True`] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

****kwargs**
[optional] Extra keyword arguments that will be passed to *pandas.read_csv*.

pept.utilities.parallel_map_file

`pept.utilities.`**`parallel_map_file`**(*func*, *fname*, *start*, *end*, *chunksize*, **args*, *dtype=<class 'float'>*, *processes=None*, *callback=<function <lambda>>*, *error_callback=<function <lambda>>*, ***kwargs*)

Utility for parallelising (read CSV chunk -> process chunk) workflows.

This function reads individual chunks of data from a CSV-formatted file, then asynchronously sends them as numpy arrays to an arbitrary function *func* for processing. In effect, it reads a file in one main thread and processes it in separate threads.

This is especially useful when dealing with very large files (like we do in PEPT…) that you'd like to process in batches, in parallel.

Parameters

func
[`callable()`] The function that will be called with each chunk of data, the chunk number, the other positional arguments **args* and keyword arguments ***kwargs*: *func(data_chunk, chunk_number, *args, **kwargs)*. *data_chunk* is a numpy array returned by *numpy.loadtxt* and *chunk_number* is an int. *func* must be picklable for sending to other threads.

fname
[`file`, `str`, or `pathlib.Path`] The file, filename, or generator that numpy.loadtxt will be supplied with.

start
[`int`] The starting line number that the chunks will be read from.

end
[`int`] The ending line number that the chunks will be read from. This is exclusive.

chunksize
[`int`] The number of lines that will be read for each chunk.

***args**
[`additional positional arguments`] Additional positional arguments that will be supplied to *func*.

dtype
[`type`] The data type of the numpy array that is returned by numpy.loadtxt. The default is *float*.

processes
[`int`] The maximum number of threads that will be used for calling *func*. If left to the default *None*, then the number returned by *os.cpu_count()* will be used.

callback
[`callable()`] When the result from a *func* call becomes ready callback is applied to it, that is unless the call failed, in which case the error_callback is applied instead.

error_callback
[`callable()`] If the target function *func* fails, then the error_callback is called with the exception instance.

****kwargs**
[`additional keybord arguments`] Additional keyword arguments that will be supplied to *func*.

Returns

`list`
A Python list of the *func* call returns. The results are not necessarily in order, though this can be verified by using the chunk number that is supplied to each call to *func*. If *func* does not return anything, it will simply be a list of *None*.

Notes

This function uses *numpy.loadtxt* to read chunks of data and *multiprocessing.Pool.apply_async* to call *func* asynchronously.

As the calls to *func* happen in different threads, all the usual parallel processing issues apply. For example, *func* should not save data to the same file, as it will overwrite results from different threads and may become corrupt - however, there is a workaround for this particular case: because the chunk numbers are guaranteed to be unique, any data can be saved to a file whose name includes this chunk number, making it unique.

Examples

For a random file-like CSV data object:

```
>>> import io
>>> flike = io.StringIO("1,2,3\n4,5,6\n7,8,9")
>>> def func(data, chunk_number):
>>>     return (data, chunk_number)
>>> results = parallel_map_file(func, flike, 0, 3, 1)
>>> print(results)
>>> [ ([1, 2, 3], 0), ([4, 5, 6], 1), ([7, 8, 9], 2) ]
```

pept.utilities.traverse2d

`pept.utilities.traverse2d`(*double[:, :] pixels*, *const double[:, :] lines*, *const double[:] grid_x*, *const double[:] grid_y*) → void

Fast pixel traversal for 2D lines (or LoRs).

```
Function Signature:
    traverse2d(
        double[:, :] pixels,        # Initialised to zero!
        double[:, :] lines,         # Has exactly 7 columns!
        double[:] grid_x,           # Has pixels.shape[0] + 1 elements!
        double[:] grid_y,           # Has pixels.shape[1] + 1 elements!
    )
```

This function computes the number of lines that passes through each pixel, saving the result in *pixels*. It does so in an efficient manner, in which for every line, only the pixels that it passes through are traversed.

As it is highly optimised, this function does not perform any checks on the validity of the input data. Please check the parameters before calling *traverse2d*, as it WILL segfault on wrong input data. Details are given below, along with an example call.

Parameters

pixels
[numpy.ndarray(dtype = numpy.float64, ndim = 2)] The *pixels* parameter is a numpy.ndarray of shape (X, Y) that has been initialised to zeros before the function call. The values will be modified in-place in the function to reflect the number of lines that pass through each pixel.

lines
[numpy.ndarray(dtype = numpy.float64, ndim = 2)] The *lines* parameter is a numpy.ndarray of shape(N, 5), where each row is formatted as [time, x1, y1, x2, y2]. Only indices 1:5 will be used as the two points P1 = [x1, y1] and P2 = [x2, y2] defining the line (or LoR).

grid_x
[numpy.ndarray(dtype = numpy.float64, ndim = 1)] The grid_x parameter is a one-dimensional grid that delimits the pixels in the x-dimension. It must be *sorted* in ascending order with *equally-spaced* numbers and length X + 1 (pixels.shape[0] + 1).

grid_y
[numpy.ndarray(dtype = numpy.float64, ndim = 1)] The grid_y parameter is a one-dimensional grid that delimits the pixels in the y-dimension. It must be *sorted* in ascending order with *equally-spaced* numbers and length Y + 1 (pixels.shape[1] + 1).

Notes

This function is an adaptation of a widely-used algorithm [1], optimised for PEPT LoRs traversal.

Examples

The input parameters can be easily generated using numpy before calling the function. For example, if a plane of 300 x 400 is split into 30 x 40 pixels, a possible code would be:

```
>>> import numpy as np
>>> from pept.utilities.traverse import traverse2d
>>>
>>> plane = [300, 400]
>>> number_of_pixels = [30, 40]
>>> pixels = np.zeros(number_of_pixels)
```

The grid has one extra element than the number of pixels. For example, 5 pixels between 0 and 5 would be delimited by the grid [0, 1, 2, 3, 4, 5] which has 6 elements (see off-by-one errors - story of my life).

```
>>> grid_x = np.linspace(0, plane[0], number_of_pixels[0] + 1)
>>> grid_y = np.linspace(0, plane[1], number_of_pixels[1] + 1)
>>>
>>> random_lines = np.random.random((100, 5)) * 100
```

Calling *traverse2d* will modify *pixels* in-place.

```
>>> traverse2d(pixels, random_lines, grid_x, grid_y)
```

pept.utilities.traverse3d

`pept.utilities.traverse3d`(*double[:, :, :] voxels*, *const double[:, :] lines*, *const double[:] grid_x*, *const double[:] grid_y*, *const double[:] grid_z*) → void

Fast voxel traversal for 3D lines (or LoRs).

```
Function Signature:
    traverse3d(
        long[:, :, :] voxels,       # Initialised!
        double[:, :] lines,         # Has exactly 7 columns!
        double[:] grid_x,           # Has voxels.shape[0] + 1 elements!
        double[:] grid_y,           # Has voxels.shape[1] + 1 elements!
        double[:] grid_z            # Has voxels.shape[2] + 1 elements!
    )
```

This function computes the number of lines that passes through each voxel, saving the result in *voxels*. It does so in an efficient manner, in which for every line, only the voxels that is passes through are traversed.

As it is highly optimised, this function does not perform any checks on the validity of the input data. Please check the parameters before calling *traverse3d*, as it WILL segfault on wrong input data. Details are given below, along with an example call.

Parameters

voxels
[`numpy.ndarray`(`dtype` = `numpy.float64`, `ndim` = 3)] The *voxels* parameter is a numpy.ndarray of shape (X, Y, Z) that has been initialised to zeros before the function call. The values will be modified in-place in the function to reflect the number of lines that pass through each voxel.

lines
[`numpy.ndarray`(`dtype` = `numpy.float64`, `ndim` = 2)] The *lines* parameter is a numpy.ndarray of shape(N, 7), where each row is formatted as [time, x1, y1, z1, x2, y2, z2]. Only indices 1:7 will be used as the two points P1 = [x1, y1, z2] and P2 = [x2, y2, z2] defining the line (or LoR).

grid_x
[`numpy.ndarray`(`dtype` = `numpy.float64`, `ndim` = 1)] The grid_x parameter is a one-dimensional grid that delimits the voxels in the x-dimension. It must be *sorted* in ascending order with *equally-spaced* numbers and length X + 1 (voxels.shape[0] + 1).

grid_y
[`numpy.ndarray`(`dtype` = `numpy.float64`, `ndim` = 1)] The grid_y parameter is a one-dimensional grid that delimits the voxels in the y-dimension. It must be *sorted* in ascending order with *equally-spaced* numbers and length Y + 1 (voxels.shape[1] + 1).

grid_z
[`numpy.ndarray`(`dtype` = `numpy.float64`, `ndim` = 1)] The grid_z parameter is a one-dimensional grid that delimits the voxels in the z-dimension. It must be *sorted* in ascending order with *equally-spaced* numbers and length Z + 1 (voxels.shape[2] + 1).

Notes

This function is an adaptation of a widely-used algorithm [1], optimised for PEPT LoRs traversal.

Examples

The input parameters can be easily generated using numpy before calling the function. For example, if a volume of 300 x 400 x 500 is split into 30 x 40 x 50 voxels, a possible code would be:

```
>>> import numpy as np
>>> from pept.utilities.traverse import traverse3d
>>>
>>> volume = [300, 400, 500]
>>> number_of_voxels = [30, 40, 50]
>>> voxels = np.zeros(number_of_voxels)
```

The grid has one extra element than the number of voxels. For example, 5 voxels between 0 and 5 would be delimited by the grid [0, 1, 2, 3, 4, 5] which has 6 elements (see off-by-one errors - story of my life).

```
>>> grid_x = np.linspace(0, volume[0], number_of_voxels[0] + 1)
>>> grid_y = np.linspace(0, volume[1], number_of_voxels[1] + 1)
>>> grid_z = np.linspace(0, volume[2], number_of_voxels[2] + 1)
>>>
>>> random_lines = np.random.random((100, 7)) * 300
```

Calling *traverse3d* will modify *voxels* in-place.

```
>>> traverse3d(voxels, random_lines, grid_x, grid_y, grid_z)
```

pept.utilities.ChunkReader

`class pept.utilities.ChunkReader`(*filepath_or_buffer*, *chunksize*, *skiprows=None*, *nrows=None*, *dtype=<class 'float'>*, *sep='\\s+'*, *header=None*, *engine='c'*, *na_filter=False*, *quoting=3*, *memory_map=True*, ***kwargs*)

Bases: `object`

Class for fast, on-demand reading / parsing and iteration over chunks of data from CSV files.

This is an abstraction above *pandas.read_csv* for easy and fast iteration over chunks of data from a CSV file. The chunks can be accessed using normal iteration (*for chunk in reader: …*) and subscripting (*reader[0]*).

The chunks are read lazily, only upon access. It is therefore a more efficient alternative to *read_csv* for large files (> 1.000.000 lines). For convenience, this class configures some default parameters for *pandas.read_csv* for fast reading and parsing of usual PEPT data.

Most importantly, it reads chunks containing *chunksize* lines from a **space-separated values** file at *filepath_or_buffer*, optionally skipping *skiprows* lines and reading in at most *nrows* lines. It returns *numpy.ndarray`s with `float* values.

Raises

`IndexError`
Upon access to a non-existent chunk using subscript notation (i.e. *data[100]* when there are 50 chunks).

See also:

pept.utilities.read_csv
Fast CSV file reading into numpy arrays.

pept.LineData
Encapsulate LoRs for ease of iteration and plotting.

pept.PointData
Encapsulate points for ease of iteration and plotting.

Examples

Say "data.csv" contains 1_000_000 lines of data. Read chunks of 10_000 lines as a time, skipping the first 100_000:

```
>>> from pept.utilities import ChunkReader
>>> chunks = ChunkReader("data.csv", 10_000, skiprows = 100_000)
>>> len(chunks)          # 90 chunks
>>> chunks.file_lines    # 1_000_000
```

Normal iteration:

```
>>> for chunk in chunks:
>>>     ... # neat operations
```

Access a single chunk using subscripting:

```
>>> chunks[0]   # First chunk
>>> chunks[-1]  # Last chunk
>>> chunks[100] # IndexError
```

Attributes

filepath_or_buffer
[str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be file://localhost/path/to/table.csv. If you want to pass in a path object, pandas accepts any *os.PathLike*. By file-like object, we refer to objects with a *read()* method, such as a file handler (e.g. via builtin *open* function) or *StringIO*.

number_of_chunks
[int] The number of chunks (also returned when using the *len* method), taking into account the lines skipped (*skiprows*), the number of lines in the file (*file_lines*) and the maximum number of lines to be read (*nrows*).

file_lines
[int] The number of lines in the file pointed at by *filepath_or_buffer*.

chunksize
[int] The number of lines in a chunk of data.

skiprows
[int] The number of lines to be skipped at the beginning of the file.

nrows
[int] The maximum number of lines to be read. Only has an effect if it is less than *file_lines* - *skiprows*. For example, if a file has 10 lines and *skiprows* = 5 and *chunksize* = 5, even if *nrows* were to be 20, the *number_of_chunks* should still be 1.

`__init__`(*filepath_or_buffer*, *chunksize*, *skiprows=None*, *nrows=None*, *dtype=<class 'float'>*, *sep='\\s+'*, *header=None*, *engine='c'*, *na_filter=False*, *quoting=3*, *memory_map=True*, ***kwargs*)

ChunkReader class constructor.

Parameters

filepath_or_buffer
[`str`, `path` `object` or file-like `object`] Any valid string path *to a local file* is acceptable. If you want to read in lines from an online location (i.e. using a URL), you should use *pept.utilities.read_csv*. If you want to pass in a path object, pandas accepts any *os.PathLike*. By file-like object, we refer to objects with a *read()* method, such as a file handler (e.g. via builtin *open* function) or *StringIO*.

chunksize
[`int`] Number of lines read in a chunk of data.

skiprows
[list-like, `int` or `callable()`, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

nrows
[`int`, optional] Number of rows of file to read. Useful for reading pieces of large files.

dtype
[`Type name`, `default` *float*] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'}.

sep
[`str`, `default` *"s+"*] Delimiter to use. Separators longer than 1 character and different from 's+' will be interpreted as regular expressions and will also force the use of the Python parsing engine.

header
[`int`, `list` `of` `int`, "infer", optional] Row number(s) to use as the column names, and the start of the data. By default assume there is no header present (i.e. *header = None*).

engine
[{'c', 'python'}, `default` "c"] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

na_filter
[bool, `default` *True*] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

quoting
[`int` or csv.QUOTE_* `instance`, `default` *csv.QUOTE_NONE*] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

memory_map
[bool, `default` `True`] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

****kwargs**
[optional] Extra keyword arguments that will be passed to *pandas.read_csv*.

Raises

EOFError
[End Of File Error] If *skiprows* >= *number_of_lines*.

Methods

__init__(filepath_or_buffer, chunksize[, ...])	ChunkReader class constructor.

Attributes

chunksize
file_lines
nrows
number_of_chunks
skiprows

property number_of_chunks

property file_lines

property chunksize

property skiprows

property nrows

pept.simulation

pept.simulation.Simulator(trajectory, ...[, ...])	Simulate PEPT data.

pept.simulation.Simulator

class pept.simulation.**Simulator**(*trajectory*, *sampling_times*, *shape_function*, *separation=712*, *decay_energy=0.6335*, *Zeff=7.22*, *Aeff=13*, *x_max=500*, *y_max=500*)

Bases: `object`

Simulate PEPT data.

__init__(*trajectory*, *sampling_times*, *shape_function*, *separation=712*, *decay_energy=0.6335*, *Zeff=7.22*, *Aeff=13*, *x_max=500*, *y_max=500*)

Simulator class constructor.

Methods

`__init__`(trajectory, sampling_times, ...[, ...])	Simulator class constructor.
`add_noise`(noise_ratio)	
`add_noise_and_spread`(noise_ratio[, ...])	
`add_spread`([max_spread, depth])	
`change_sampling_times`(new_sampling_times)	
`change_shape`(new_shape_function)	
`change_trajectory`(new_trajectory)	
`simulate`()	
`write_csv`(fname)	
`write_noise_csv`(fname)	

simulate()

add_noise(*noise_ratio*)

add_spread(*max_spread=4*, *depth=16*)

add_noise_and_spread(*noise_ratio*, *max_spread=4*, *depth=16*)

change_trajectory(*new_trajectory*)

change_sampling_times(*new_sampling_times*)

change_shape(*new_shape_function*)

write_csv(*fname*)

write_noise_csv(*fname*)

5.4 Contributing

The *pept* library is not a one-man project; it is being built, improved and extended continuously (directly or indirectly) by an international team of researchers of diverse backgrounds - including programmers, mathematicians and chemical / mechanical / nuclear engineers. Want to contribute and become a PEPTspert yourself? Great, join the team!

There are multiple ways to help:

- Open an issue mentioning any improvement you think *pept* could benefit from.
- Write a tutorial or share scripts you've developed that we can add to the *pept* documentation to help other people in the future.
- Share your PEPT-related algorithms - tracking, post-processing, visualisation, anything really! - so everybody can benefit from them.

Want to be a superhero and contribute code directly to the library itself? Grand - fork the project, add your code and submit a pull request (if that sounds like gibberish but you're an eager programmer, check this article). We are more than happy to work with you on integrating your code into the library and, if helpful, we can schedule a screen-to-screen meeting for a more in-depth discussion about the *pept* package architecture.

Naturally, anything you contribute to the library will respect your authorship - protected by the strong GPL v3.0 open-source license (see the "Licensing" section below). If you include published work, please add a pointer to your publication in the code documentation.

5.4.1 Licensing

The *pept* package is GPL v3.0 licensed. In non-lawyer terms, the key points of this license are:

- You can view, use, copy and modify this code **_freely_**.
- Your modifications must _also_ be licensed with GPL v3.0 or later.
- If you share your modifications with someone, you have to include the source code as well.

Essentially do whatever you want with the code, but don't try selling it saying it's yours :). This is a community-driven project building upon many other wonderful open-source projects (NumPy, Plotly, even Python itself!) without which *pept* simply would not have been possible. GPL v3.0 is indeed a very strong *copyleft* license; it was deliberately chosen to maintain the openness and transparency of great software and progress, and respect the researchers pushing PEPT forward. Frankly, open collaboration is way more efficient than closed, for-profit competition.

5.5 Citing

If you used this codebase or any software making use of it in a scientific publication, we ask you to cite the following paper:

> Nicuşan AL, Windows-Yule CR. Positron emission particle tracking using machine learning. Review of Scientific Instruments. 2020 Jan 1;91(1):013329. https://doi.org/10.1063/1.5129251

Because *pept* is a project bringing together the expertise of many people, it hosts multiple algorithms that were developed and published in other papers. Please check the documentation of the *pept* algorithms you are using in your research and cite the original papers mentioned accordingly.

5.5.1 References

Papers presenting PEPT algorithms included in this library:[1],[2],[3].

Pages

- genindex
- modindex
- search

[1] Parker DJ, Broadbent CJ, Fowles P, Hawkesworth MR, McNeil P. Positron emission particle tracking-a technique for studying flow within engineering equipment. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 1993 Mar 10;326(3):592-607.

[2] Nicuşan AL, Windows-Yule CR. Positron emission particle tracking using machine learning. Review of Scientific Instruments. 2020 Jan 1;91(1):013329.

[3] Wiggins C, Santos R, Ruggles A. A feature point identification method for positron emission particle tracking with multiple tracers. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2017 Jan 21;843:22-8.

BIBLIOGRAPHY

[1] Guida A. Positron emission particle tracking applied to solid-liquid mixing in mechanically agitated vessels (Doctoral dissertation, University of Birmingham).

[1] Amanatides J, Woo A. A fast voxel traversal algorithm for ray tracing. InEurographics 1987 Aug 24 (Vol. 87, No. 3, pp. 3-10).

[1] Amanatides J, Woo A. A fast voxel traversal algorithm for ray tracing. InEurographics 1987 Aug 24 (Vol. 87, No. 3, pp. 3-10)..

PYTHON MODULE INDEX

p

pept.plots, 199
pept.processing, 164
pept.scanners, 90
pept.simulation, 235
pept.tracking, 98
pept.utilities, 221

INDEX

Symbols

`__init__()` (*pept.AdaptiveWindow method*), 76
`__init__()` (*pept.LineData method*), 36
`__init__()` (*pept.Pipeline method*), 72
`__init__()` (*pept.Pixels method*), 51
`__init__()` (*pept.PointData method*), 44
`__init__()` (*pept.TimeWindow method*), 75
`__init__()` (*pept.Voxels method*), 59
`__init__()` (*pept.base.Filter method*), 82
`__init__()` (*pept.base.IterableSamples method*), 78
`__init__()` (*pept.base.LineDataFilter method*), 87
`__init__()` (*pept.base.PEPTObject method*), 76
`__init__()` (*pept.base.PointDataFilter method*), 85
`__init__()` (*pept.base.Reducer method*), 84
`__init__()` (*pept.base.Transformer method*), 81
`__init__()` (*pept.base.VoxelsFilter method*), 88
`__init__()` (*pept.plots.PlotlyGrapher method*), 202
`__init__()` (*pept.plots.PlotlyGrapher2D method*), 212
`__init__()` (*pept.processing.AutoCorrelation method*), 194
`__init__()` (*pept.processing.DynamicProbability2D method*), 165
`__init__()` (*pept.processing.DynamicProbability3D method*), 168
`__init__()` (*pept.processing.LaceyColors method*), 182
`__init__()` (*pept.processing.LaceyColorsLinear method*), 185
`__init__()` (*pept.processing.RelativeDeviations method*), 188
`__init__()` (*pept.processing.RelativeDeviationsLinear method*), 191
`__init__()` (*pept.processing.ResidenceDistribution2D method*), 171
`__init__()` (*pept.processing.ResidenceDistribution3D method*), 174
`__init__()` (*pept.processing.SpatialProjections method*), 197
`__init__()` (*pept.processing.VectorField2D method*), 176
`__init__()` (*pept.processing.VectorField3D method*), 179
`__init__()` (*pept.processing.VectorGrid2D method*), 178
`__init__()` (*pept.processing.VectorGrid3D method*), 180
`__init__()` (*pept.scanners.ADACGeometricEfficiency method*), 96
`__init__()` (*pept.simulation.Simulator method*), 235
`__init__()` (*pept.tracking.BirminghamMethod method*), 134
`__init__()` (*pept.tracking.Centroids method*), 109
`__init__()` (*pept.tracking.Condition method*), 113
`__init__()` (*pept.tracking.Cutpoints method*), 137
`__init__()` (*pept.tracking.CutpointsToF method*), 158
`__init__()` (*pept.tracking.Debug method*), 99
`__init__()` (*pept.tracking.FPI method*), 146
`__init__()` (*pept.tracking.GaussianDensity method*), 160
`__init__()` (*pept.tracking.GroupBy method*), 107
`__init__()` (*pept.tracking.HDBSCAN method*), 143
`__init__()` (*pept.tracking.Interpolate method*), 125
`__init__()` (*pept.tracking.LinesCentroids method*), 111
`__init__()` (*pept.tracking.Minpoints method*), 140
`__init__()` (*pept.tracking.OptimizeWindow method*), 102
`__init__()` (*pept.tracking.OutOfViewFilter method*), 129
`__init__()` (*pept.tracking.Reconnect method*), 152
`__init__()` (*pept.tracking.Remove method*), 117
`__init__()` (*pept.tracking.RemoveStatic method*), 131
`__init__()` (*pept.tracking.Reorient method*), 127
`__init__()` (*pept.tracking.SamplesCondition method*), 115
`__init__()` (*pept.tracking.Segregate method*), 150
`__init__()` (*pept.tracking.SplitLabels method*), 105
`__init__()` (*pept.tracking.Stack method*), 104
`__init__()` (*pept.tracking.Swap method*), 119
`__init__()` (*pept.tracking.TimeOfFlight method*), 156
`__init__()` (*pept.tracking.Velocity method*), 162
`__init__()` (*pept.tracking.Voxelize method*), 122
`__init__()` (*pept.utilities.ChunkReader method*), 233

A

`adac_forte()` (*in module pept.scanners*), 90

ADACGeometricEfficiency (*class in pept.scanners*), 94
AdaptiveWindow (*class in pept*), 75
add_image() (*pept.plots.PlotlyGrapher2D method*), 219
add_lines() (*pept.Pixels method*), 57
add_lines() (*pept.plots.PlotlyGrapher method*), 205
add_lines() (*pept.plots.PlotlyGrapher2D method*), 217
add_lines() (*pept.Voxels method*), 70
add_noise() (*pept.simulation.Simulator method*), 236
add_noise_and_spread() (*pept.simulation.Simulator method*), 236
add_pixels() (*pept.plots.PlotlyGrapher method*), 207
add_pixels() (*pept.plots.PlotlyGrapher2D method*), 218
add_points() (*pept.plots.PlotlyGrapher method*), 204
add_points() (*pept.plots.PlotlyGrapher2D method*), 216
add_spread() (*pept.simulation.Simulator method*), 236
add_timeseries() (*pept.plots.PlotlyGrapher2D method*), 215
add_trace() (*pept.plots.PlotlyGrapher method*), 209
add_trace() (*pept.plots.PlotlyGrapher2D method*), 219
add_traces() (*pept.plots.PlotlyGrapher method*), 209
add_traces() (*pept.plots.PlotlyGrapher2D method*), 219
add_voxels() (*pept.plots.PlotlyGrapher method*), 208
append_indices (*pept.tracking.Cutpoints property*), 139
append_indices (*pept.tracking.CutpointsToF property*), 159
append_indices (*pept.tracking.Minpoints property*), 143
attrs (*pept.base.IterableSamples property*), 79
attrs (*pept.LineData property*), 39
attrs (*pept.Pixels property*), 53
attrs (*pept.PointData property*), 47
attrs (*pept.Voxels property*), 61
AutoCorrelation (*class in pept.processing*), 193

B

BirminghamMethod (*class in pept.tracking*), 133

C

centroid() (*pept.tracking.LinesCentroids static method*), 111
Centroids (*class in pept.tracking*), 109
change_sampling_times() (*pept.simulation.Simulator method*), 236
change_shape() (*pept.simulation.Simulator method*), 236
change_trajectory() (*pept.simulation.Simulator method*), 236
ChunkReader (*class in pept.utilities*), 232
chunksize (*pept.utilities.ChunkReader property*), 235
columns (*pept.base.IterableSamples property*), 79
columns (*pept.LineData property*), 39
columns (*pept.PointData property*), 47
columns (*pept.tracking.Remove property*), 118
Condition (*class in pept.tracking*), 113
conditions (*pept.tracking.Condition property*), 113
conditions (*pept.tracking.SamplesCondition property*), 116
copy() (*pept.base.Filter method*), 82
copy() (*pept.base.IterableSamples method*), 79
copy() (*pept.base.LineDataFilter method*), 87
copy() (*pept.base.PEPTObject method*), 76
copy() (*pept.base.PointDataFilter method*), 86
copy() (*pept.base.Reducer method*), 84
copy() (*pept.base.Transformer method*), 81
copy() (*pept.base.VoxelsFilter method*), 89
copy() (*pept.LineData method*), 39
copy() (*pept.Pipeline method*), 73
copy() (*pept.Pixels method*), 54
copy() (*pept.plots.PlotlyGrapher method*), 209
copy() (*pept.PointData method*), 47
copy() (*pept.processing.AutoCorrelation method*), 195
copy() (*pept.processing.DynamicProbability2D method*), 166
copy() (*pept.processing.DynamicProbability3D method*), 168
copy() (*pept.processing.LaceyColors method*), 183
copy() (*pept.processing.LaceyColorsLinear method*), 185
copy() (*pept.processing.RelativeDeviations method*), 188
copy() (*pept.processing.RelativeDeviationsLinear method*), 191
copy() (*pept.processing.ResidenceDistribution2D method*), 171
copy() (*pept.processing.ResidenceDistribution3D method*), 174
copy() (*pept.processing.SpatialProjections method*), 198
copy() (*pept.processing.VectorField2D method*), 176
copy() (*pept.processing.VectorField3D method*), 179
copy() (*pept.scanners.ADACGeometricEfficiency method*), 96
copy() (*pept.tracking.BirminghamMethod method*), 135
copy() (*pept.tracking.Centroids method*), 110
copy() (*pept.tracking.Condition method*), 113
copy() (*pept.tracking.Cutpoints method*), 138
copy() (*pept.tracking.CutpointsToF method*), 159
copy() (*pept.tracking.Debug method*), 100
copy() (*pept.tracking.FPI method*), 147
copy() (*pept.tracking.GaussianDensity method*), 161
copy() (*pept.tracking.GroupBy method*), 108
copy() (*pept.tracking.HDBSCAN method*), 143
copy() (*pept.tracking.Interpolate method*), 125
copy() (*pept.tracking.LinesCentroids method*), 111

copy() (*pept.tracking.Minpoints method*), 142
copy() (*pept.tracking.OptimizeWindow method*), 102
copy() (*pept.tracking.OutOfViewFilter method*), 129
copy() (*pept.tracking.Reconnect method*), 152
copy() (*pept.tracking.Remove method*), 118
copy() (*pept.tracking.RemoveStatic method*), 131
copy() (*pept.tracking.Reorient method*), 127
copy() (*pept.tracking.SamplesCondition method*), 116
copy() (*pept.tracking.Segregate method*), 150
copy() (*pept.tracking.SplitLabels method*), 106
copy() (*pept.tracking.Stack method*), 104
copy() (*pept.tracking.Swap method*), 120
copy() (*pept.tracking.TimeOfFlight method*), 156
copy() (*pept.tracking.Velocity method*), 163
copy() (*pept.tracking.Voxelize method*), 123
copy() (*pept.Voxels method*), 62
create_figure() (*pept.plots.PlotlyGrapher method*), 203
create_figure() (*pept.plots.PlotlyGrapher2D method*), 214
cube_trace() (*pept.Voxels method*), 66
cubes_traces() (*pept.Voxels method*), 67
cutoffs (*pept.tracking.Cutpoints property*), 139
cutoffs (*pept.tracking.CutpointsToF property*), 159
cutoffs (*pept.tracking.Minpoints property*), 143
Cutpoints (*class in pept.tracking*), 136
CutpointsToF (*class in pept.tracking*), 157

D

data (*pept.base.IterableSamples property*), 79
data (*pept.LineData property*), 39
data (*pept.PointData property*), 47
Debug (*class in pept.tracking*), 99
distance_matrix() (*pept.tracking.LinesCentroids static method*), 111
DynamicProbability2D (*class in pept.processing*), 164
DynamicProbability3D (*class in pept.processing*), 167

E

eg() (*pept.scanners.ADACGeometricEfficiency method*), 96
equalise_axes() (*pept.plots.PlotlyGrapher method*), 209
equalise_axes() (*pept.plots.PlotlyGrapher2D method*), 220
equalise_separate() (*pept.plots.PlotlyGrapher2D method*), 220
evaluate() (*pept.tracking.OptimizeWindow method*), 103
extra_attrs() (*pept.base.IterableSamples method*), 79
extra_attrs() (*pept.LineData method*), 39
extra_attrs() (*pept.PointData method*), 47

F

fig (*pept.plots.PlotlyGrapher property*), 204
fig (*pept.plots.PlotlyGrapher2D property*), 215
file_lines (*pept.utilities.ChunkReader property*), 235
Filter (*class in pept.base*), 82
filters (*pept.Pipeline property*), 73
find_cutpoints() (*in module pept.utilities*), 221
find_minpoints() (*in module pept.utilities*), 223
fit() (*pept.base.Filter method*), 82
fit() (*pept.base.LineDataFilter method*), 87
fit() (*pept.base.PointDataFilter method*), 85
fit() (*pept.base.Reducer method*), 84
fit() (*pept.base.VoxelsFilter method*), 89
fit() (*pept.Pipeline method*), 73
fit() (*pept.processing.AutoCorrelation method*), 195
fit() (*pept.processing.DynamicProbability2D method*), 166
fit() (*pept.processing.DynamicProbability3D method*), 168
fit() (*pept.processing.LaceyColors method*), 183
fit() (*pept.processing.LaceyColorsLinear method*), 185
fit() (*pept.processing.RelativeDeviations method*), 188
fit() (*pept.processing.RelativeDeviationsLinear method*), 191
fit() (*pept.processing.ResidenceDistribution2D method*), 171
fit() (*pept.processing.ResidenceDistribution3D method*), 174
fit() (*pept.processing.SpatialProjections method*), 198
fit() (*pept.processing.VectorField2D method*), 176
fit() (*pept.processing.VectorField3D method*), 179
fit() (*pept.tracking.BirminghamMethod method*), 135
fit() (*pept.tracking.Centroids method*), 110
fit() (*pept.tracking.Condition method*), 114
fit() (*pept.tracking.Cutpoints method*), 138
fit() (*pept.tracking.CutpointsToF method*), 159
fit() (*pept.tracking.Debug method*), 101
fit() (*pept.tracking.FPI method*), 147
fit() (*pept.tracking.GaussianDensity method*), 161
fit() (*pept.tracking.GroupBy method*), 108
fit() (*pept.tracking.HDBSCAN method*), 144
fit() (*pept.tracking.Interpolate method*), 125
fit() (*pept.tracking.LinesCentroids method*), 111
fit() (*pept.tracking.Minpoints method*), 142
fit() (*pept.tracking.OptimizeWindow method*), 102
fit() (*pept.tracking.OutOfViewFilter method*), 129
fit() (*pept.tracking.Reconnect method*), 152
fit() (*pept.tracking.Remove method*), 118
fit() (*pept.tracking.RemoveStatic method*), 132
fit() (*pept.tracking.Reorient method*), 127
fit() (*pept.tracking.SamplesCondition method*), 116
fit() (*pept.tracking.Segregate method*), 150
fit() (*pept.tracking.SplitLabels method*), 106
fit() (*pept.tracking.Stack method*), 104

fit() (*pept.tracking.Swap method*), 120
fit() (*pept.tracking.TimeOfFlight method*), 156
fit() (*pept.tracking.Velocity method*), 163
fit() (*pept.tracking.Voxelize method*), 123
fit_sample() (*pept.base.Filter method*), 82
fit_sample() (*pept.base.LineDataFilter method*), 87
fit_sample() (*pept.base.PointDataFilter method*), 86
fit_sample() (*pept.base.VoxelsFilter method*), 89
fit_sample() (*pept.Pipeline method*), 73
fit_sample() (*pept.tracking.BirminghamMethod method*), 134
fit_sample() (*pept.tracking.Centroids method*), 109
fit_sample() (*pept.tracking.Condition method*), 113
fit_sample() (*pept.tracking.Cutpoints method*), 139
fit_sample() (*pept.tracking.CutpointsToF method*), 159
fit_sample() (*pept.tracking.FPI method*), 147
fit_sample() (*pept.tracking.GaussianDensity method*), 162
fit_sample() (*pept.tracking.HDBSCAN method*), 144
fit_sample() (*pept.tracking.Interpolate method*), 125
fit_sample() (*pept.tracking.LinesCentroids method*), 111
fit_sample() (*pept.tracking.Minpoints method*), 143
fit_sample() (*pept.tracking.Remove method*), 118
fit_sample() (*pept.tracking.SplitLabels method*), 106
fit_sample() (*pept.tracking.Swap method*), 120
fit_sample() (*pept.tracking.TimeOfFlight method*), 156
fit_sample() (*pept.tracking.Velocity method*), 163
fit_sample() (*pept.tracking.Voxelize method*), 123
format_fig() (*in module pept.plots*), 199
FPI (*class in pept.tracking*), 145
from_lines() (*pept.Pixels static method*), 58
from_lines() (*pept.Voxels static method*), 70
from_physical() (*pept.Pixels method*), 54
from_physical() (*pept.Voxels method*), 62

G

GaussianDensity (*class in pept.tracking*), 160
group_by_column() (*in module pept.utilities*), 224
GroupBy (*class in pept.tracking*), 107

H

HDBSCAN (*class in pept.tracking*), 143
heatmap_trace() (*pept.Pixels method*), 56
heatmap_trace() (*pept.Voxels method*), 69
hidden_attrs() (*pept.base.IterableSamples method*), 79
hidden_attrs() (*pept.LineData method*), 39
hidden_attrs() (*pept.PointData method*), 47
histogram() (*in module pept.plots*), 199

I

Interpolate (*class in pept.tracking*), 124
IterableSamples (*class in pept.base*), 78

L

LaceyColors (*class in pept.processing*), 181
LaceyColorsLinear (*class in pept.processing*), 184
LineData (*class in pept*), 32
LineDataFilter (*class in pept.base*), 87
lines (*pept.LineData property*), 38
lines_trace() (*pept.plots.PlotlyGrapher static method*), 205
lines_trace() (*pept.plots.PlotlyGrapher2D static method*), 217
LinesCentroids (*class in pept.tracking*), 111
load() (*in module pept*), 31
load() (*pept.base.Filter static method*), 82
load() (*pept.base.IterableSamples static method*), 80
load() (*pept.base.LineDataFilter static method*), 87
load() (*pept.base.PEPTObject static method*), 77
load() (*pept.base.PointDataFilter static method*), 86
load() (*pept.base.Reducer static method*), 84
load() (*pept.base.Transformer static method*), 81
load() (*pept.base.VoxelsFilter static method*), 89
load() (*pept.LineData static method*), 39
load() (*pept.Pipeline static method*), 73
load() (*pept.Pixels static method*), 54
load() (*pept.plots.PlotlyGrapher static method*), 210
load() (*pept.PointData static method*), 47
load() (*pept.processing.AutoCorrelation static method*), 195
load() (*pept.processing.DynamicProbability2D static method*), 166
load() (*pept.processing.DynamicProbability3D static method*), 168
load() (*pept.processing.LaceyColors static method*), 183
load() (*pept.processing.LaceyColorsLinear static method*), 185
load() (*pept.processing.RelativeDeviations static method*), 188
load() (*pept.processing.RelativeDeviationsLinear static method*), 192
load() (*pept.processing.ResidenceDistribution2D static method*), 171
load() (*pept.processing.ResidenceDistribution3D static method*), 174
load() (*pept.processing.SpatialProjections static method*), 198
load() (*pept.processing.VectorField2D static method*), 176
load() (*pept.processing.VectorField3D static method*), 179

load() (*pept.scanners.ADACGeometricEfficiency static method*), 96
load() (*pept.tracking.BirminghamMethod static method*), 135
load() (*pept.tracking.Centroids static method*), 110
load() (*pept.tracking.Condition static method*), 114
load() (*pept.tracking.Cutpoints static method*), 138
load() (*pept.tracking.CutpointsToF static method*), 159
load() (*pept.tracking.Debug static method*), 100
load() (*pept.tracking.FPI static method*), 147
load() (*pept.tracking.GaussianDensity static method*), 161
load() (*pept.tracking.GroupBy static method*), 108
load() (*pept.tracking.HDBSCAN static method*), 144
load() (*pept.tracking.Interpolate static method*), 125
load() (*pept.tracking.LinesCentroids static method*), 111
load() (*pept.tracking.Minpoints static method*), 142
load() (*pept.tracking.OptimizeWindow static method*), 102
load() (*pept.tracking.OutOfViewFilter static method*), 129
load() (*pept.tracking.Reconnect static method*), 152
load() (*pept.tracking.Remove static method*), 118
load() (*pept.tracking.RemoveStatic static method*), 132
load() (*pept.tracking.Reorient static method*), 127
load() (*pept.tracking.SamplesCondition static method*), 116
load() (*pept.tracking.Segregate static method*), 150
load() (*pept.tracking.SplitLabels static method*), 106
load() (*pept.tracking.Stack static method*), 104
load() (*pept.tracking.Swap static method*), 120
load() (*pept.tracking.TimeOfFlight static method*), 156
load() (*pept.tracking.Velocity static method*), 163
load() (*pept.tracking.Voxelize static method*), 123
load() (*pept.Voxels static method*), 62
lower (*pept.Pixels property*), 53
lower (*pept.Voxels property*), 61

M

make_video() (*in module pept.plots*), 200
max_distance (*pept.tracking.Cutpoints property*), 139
max_distance (*pept.tracking.CutpointsToF property*), 159
max_distance (*pept.tracking.Minpoints property*), 143
Minpoints (*class in pept.tracking*), 139
modular_camera() (*in module pept.scanners*), 97
module
    pept.plots, 199
    pept.processing, 164
    pept.scanners, 90
    pept.simulation, 235
    pept.tracking, 98
    pept.utilities, 221

N

nrows (*pept.utilities.ChunkReader property*), 235
num_lines (*pept.tracking.Minpoints property*), 142
number_of_chunks (*pept.utilities.ChunkReader property*), 235
number_of_lines() (*in module pept.utilities*), 225
number_of_voxels (*pept.tracking.Voxelize property*), 123

O

optimise() (*pept.Pipeline method*), 73
OptimizeWindow (*class in pept.tracking*), 101
OutOfViewFilter (*class in pept.tracking*), 128
overlap (*pept.base.IterableSamples property*), 79
overlap (*pept.LineData property*), 40
overlap (*pept.PointData property*), 47

P

parallel_map_file() (*in module pept.utilities*), 228
parallel_screens() (*in module pept.scanners*), 92
pept.plots
    module, 199
pept.processing
    module, 164
pept.scanners
    module, 90
pept.simulation
    module, 235
pept.tracking
    module, 98
pept.utilities
    module, 221
PEPTObject (*class in pept.base*), 76
Pipeline (*class in pept*), 71
pixel_grids (*pept.Pixels property*), 53
pixel_size (*pept.Pixels property*), 53
Pixels (*class in pept*), 48
pixels (*pept.Pixels property*), 53
plot() (*pept.LineData method*), 38
plot() (*pept.Pixels method*), 57
plot() (*pept.PointData method*), 46
plot() (*pept.Voxels method*), 65
plot_volumetric() (*pept.Voxels method*), 66
PlotlyGrapher (*class in pept.plots*), 201
PlotlyGrapher2D (*class in pept.plots*), 211
PointData (*class in pept*), 40
PointDataFilter (*class in pept.base*), 85
points (*pept.PointData property*), 45
points_trace() (*pept.plots.PlotlyGrapher static method*), 204
points_trace() (*pept.plots.PlotlyGrapher2D static method*), 216
predict() (*pept.tracking.LinesCentroids method*), 111

Q

quiver() (*pept.processing.VectorGrid2D method*), 178

R

read_csv() (*in module pept*), 30
read_csv() (*in module pept.utilities*), 225
read_csv_chunks() (*in module pept.utilities*), 226
Reconnect (*class in pept.tracking*), 151
Reducer (*class in pept.base*), 84
reducers (*pept.Pipeline property*), 73
RelativeDeviations (*class in pept.processing*), 186
RelativeDeviationsLinear (*class in pept.processing*), 189
Remove (*class in pept.tracking*), 117
RemoveStatic (*class in pept.tracking*), 130
Reorient (*class in pept.tracking*), 126
ResidenceDistribution2D (*class in pept.processing*), 170
ResidenceDistribution3D (*class in pept.processing*), 172

S

sample_size (*pept.base.IterableSamples property*), 79
sample_size (*pept.LineData property*), 40
sample_size (*pept.PointData property*), 47
samples_indices (*pept.base.IterableSamples property*), 79
samples_indices (*pept.LineData property*), 40
samples_indices (*pept.PointData property*), 47
SamplesCondition (*class in pept.tracking*), 115
save() (*in module pept*), 31
save() (*pept.base.Filter method*), 83
save() (*pept.base.IterableSamples method*), 80
save() (*pept.base.LineDataFilter method*), 88
save() (*pept.base.PEPTObject method*), 77
save() (*pept.base.PointDataFilter method*), 86
save() (*pept.base.Reducer method*), 84
save() (*pept.base.Transformer method*), 81
save() (*pept.base.VoxelsFilter method*), 89
save() (*pept.LineData method*), 40
save() (*pept.Pipeline method*), 74
save() (*pept.Pixels method*), 53
save() (*pept.plots.PlotlyGrapher method*), 210
save() (*pept.PointData method*), 47
save() (*pept.processing.AutoCorrelation method*), 195
save() (*pept.processing.DynamicProbability2D method*), 166
save() (*pept.processing.DynamicProbability3D method*), 169
save() (*pept.processing.LaceyColors method*), 183
save() (*pept.processing.LaceyColorsLinear method*), 186
save() (*pept.processing.RelativeDeviations method*), 189
save() (*pept.processing.RelativeDeviationsLinear method*), 192
save() (*pept.processing.ResidenceDistribution2D method*), 172
save() (*pept.processing.ResidenceDistribution3D method*), 174
save() (*pept.processing.SpatialProjections method*), 198
save() (*pept.processing.VectorField2D method*), 177
save() (*pept.processing.VectorField3D method*), 180
save() (*pept.scanners.ADACGeometricEfficiency method*), 97
save() (*pept.tracking.BirminghamMethod method*), 135
save() (*pept.tracking.Centroids method*), 110
save() (*pept.tracking.Condition method*), 114
save() (*pept.tracking.Cutpoints method*), 139
save() (*pept.tracking.CutpointsToF method*), 159
save() (*pept.tracking.Debug method*), 100
save() (*pept.tracking.FPI method*), 148
save() (*pept.tracking.GaussianDensity method*), 161
save() (*pept.tracking.GroupBy method*), 108
save() (*pept.tracking.HDBSCAN method*), 144
save() (*pept.tracking.Interpolate method*), 125
save() (*pept.tracking.LinesCentroids method*), 112
save() (*pept.tracking.Minpoints method*), 142
save() (*pept.tracking.OptimizeWindow method*), 102
save() (*pept.tracking.OutOfViewFilter method*), 130
save() (*pept.tracking.Reconnect method*), 153
save() (*pept.tracking.Remove method*), 119
save() (*pept.tracking.RemoveStatic method*), 132
save() (*pept.tracking.Reorient method*), 128
save() (*pept.tracking.SamplesCondition method*), 116
save() (*pept.tracking.Segregate method*), 151
save() (*pept.tracking.SplitLabels method*), 106
save() (*pept.tracking.Stack method*), 105
save() (*pept.tracking.Swap method*), 121
save() (*pept.tracking.TimeOfFlight method*), 157
save() (*pept.tracking.Velocity method*), 163
save() (*pept.tracking.Voxelize method*), 124
save() (*pept.Voxels method*), 61
scatter_trace() (*pept.Voxels method*), 68
Segregate (*class in pept.tracking*), 148
set_lims() (*pept.tracking.Voxelize method*), 123
show() (*pept.plots.PlotlyGrapher method*), 210
show() (*pept.plots.PlotlyGrapher2D method*), 220
simulate() (*pept.simulation.Simulator method*), 236
Simulator (*class in pept.simulation*), 235
skiprows (*pept.utilities.ChunkReader property*), 235
SpatialProjections (*class in pept.processing*), 196
SplitAll (*in module pept.tracking*), 107
SplitLabels (*class in pept.tracking*), 105
Stack (*class in pept.tracking*), 104
steps() (*pept.Pipeline method*), 73
Swap (*class in pept.tracking*), 119

swaps (*pept.tracking.Swap property*), 120

T

TimeOfFlight (*class in pept.tracking*), 154
timeseries_trace() (*pept.plots.PlotlyGrapher2D static method*), 215
TimeWindow (*class in pept*), 75
to_csv() (*pept.LineData method*), 38
to_csv() (*pept.PointData method*), 46
to_html() (*pept.plots.PlotlyGrapher method*), 211
to_html() (*pept.plots.PlotlyGrapher2D method*), 220
to_physical() (*pept.Pixels method*), 55
to_physical() (*pept.Voxels method*), 64
Transformer (*class in pept.base*), 81
transformers (*pept.Pipeline property*), 73
traverse2d() (*in module pept.utilities*), 229
traverse3d() (*in module pept.utilities*), 231

U

upper (*pept.Pixels property*), 53
upper (*pept.Voxels property*), 61

V

VectorField2D (*class in pept.processing*), 175
VectorField3D (*class in pept.processing*), 178
VectorGrid2D (*class in pept.processing*), 177
VectorGrid3D (*class in pept.processing*), 180
vectors() (*pept.processing.VectorGrid2D method*), 178
vectors() (*pept.processing.VectorGrid3D method*), 181
Velocity (*class in pept.tracking*), 162
voxel_grids (*pept.Voxels property*), 61
voxel_size (*pept.Voxels property*), 61
Voxelize (*class in pept.tracking*), 122
Voxels (*class in pept*), 58
voxels (*pept.Voxels property*), 61
VoxelsFilter (*class in pept.base*), 88
vtk() (*pept.Voxels method*), 66

W

window (*pept.TimeWindow attribute*), 75
write_csv() (*pept.simulation.Simulator method*), 236
write_noise_csv() (*pept.simulation.Simulator method*), 236

X

xlabel() (*pept.plots.PlotlyGrapher method*), 204
xlabel() (*pept.plots.PlotlyGrapher2D method*), 215
xlim (*pept.Pixels property*), 53
xlim (*pept.plots.PlotlyGrapher property*), 204
xlim (*pept.plots.PlotlyGrapher2D property*), 215
xlim (*pept.tracking.Voxelize property*), 123
xlim (*pept.Voxels property*), 61

Y

ylabel() (*pept.plots.PlotlyGrapher method*), 204
ylabel() (*pept.plots.PlotlyGrapher2D method*), 215
ylim (*pept.Pixels property*), 53
ylim (*pept.plots.PlotlyGrapher property*), 204
ylim (*pept.plots.PlotlyGrapher2D property*), 215
ylim (*pept.tracking.Voxelize property*), 123
ylim (*pept.Voxels property*), 61

Z

zeros() (*pept.Pixels static method*), 53
zeros() (*pept.Voxels static method*), 62
zlabel() (*pept.plots.PlotlyGrapher method*), 204
zlim (*pept.plots.PlotlyGrapher property*), 204
zlim (*pept.tracking.Voxelize property*), 123
zlim (*pept.Voxels property*), 61