pept Documentation

Release 0.4.1

pept maintainers

Sep 09, 2021

DOCUMENTATION

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

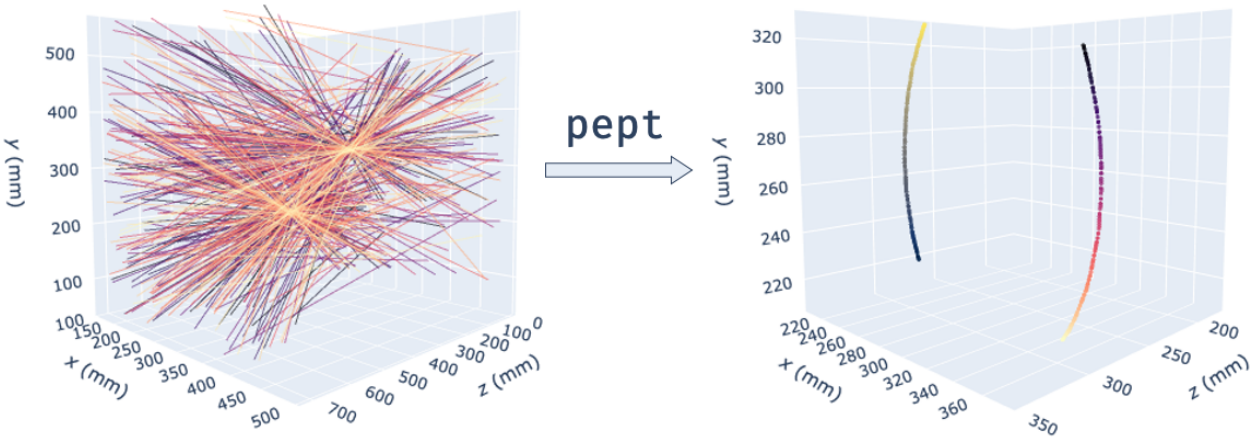
CHAPTER

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

This part contains copy-pastable tutorials for using the `pept` library to initialise, track, post-process and visualise your PEPT data.

5.2.1 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)

# 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)
```

5.2.2 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
```

(continues on next page)

(continued from previous page)

```
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)

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()
```

5.2.3 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")
```

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)
```

5.2.4 PEPT-ML

PEPT-ML one pass of clustering recipe

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

locations = pipeline.fit(lors)
```

PEPT-ML second pass of clustering recipe

```
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(),
    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,
    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 = 0.5),
    HDBSCAN(true_fraction = 0.15, max_tracers = max_tracers),
    SplitLabels() + Centroids(),

    # 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),
])


# Process all samples in `lors` in parallel, using `max_workers` threads
trajectories = pipeline.fit(lors, max_workers = 16)
```

(continues on next page)

(continued from previous page)

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


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

5.2.5 The Birmingham Method

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),
])

locations = pipeline.fit(lors)
```

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)	A class that manages a 2D pixel space, including tools for pixel traversal of lines, manipulation and visualisation.
pept.Voxels(voxels_array, xlim, ylim, zlim)	A class that manages a single 3D voxel space, including tools for voxel traversal of lines, 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: *pept.base.iterable_samples.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.]]
```

(continues on next page)

(continued from previous page)

```
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)
--------------------------
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: *pept.base.iterable_samples.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
```

(continues on next page)

(continued from previous page)

```
pept.PointData (samples: 3)
---------------------------
sample_size = 3
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.],
       [ 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.],
```

(continues on next page)

(continued from previous page)

```
       [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.]])
```

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*)

Bases: numpy.ndarray, *pept.base.iterable_samples.PEPTObject*

A class that manages a 2D pixel space, including tools for pixel traversal of lines, manipulation and visualisation.

This class can be instantiated in a couple of ways:

1. The constructor receives a pre-defined pixel space (i.e. a 2D numpy array), along with the space boundaries *xlim* and *ylim*.
2. The *from_lines* method receives a sample of 2D lines (i.e. a 2D numpy array), each defined by two points, creating a pixel space and traversing / pixellising the lines.
3. The *empty* method creates a pixel space filled with zeros.

This subclasses the *numpy.ndarray* class, so any *Pixels* object acts exactly like a 2D numpy array. All numpy methods and operations are valid on *Pixels* (e.g. add 1 to all pixels with *pixels += 1*).

It is possible to add multiple samples of lines to the same pixel space using the *add_lines* method.

See also:

pept.LineData Encapsulate lines 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.

Notes

The traversed lines do not need to be fully bounded by the pixel space. Their intersection is automatically computed.

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

This class is most often instantiated from a sample of lines to pixellise:

```
>>> import pept
>>> import numpy as np
```

```
>>> lines = np.arange(70).reshape(10, 7)
```

```
>>> number_of_pixels = [3, 4]
>>> pixels = pept.Pixels.from_lines(lines, number_of_pixels)
>>> Initialised Pixels class in 0.0006861686706542969 s.
```

```
>>> print(pixels)
>>> pixels:
>>> [[[2. 1. 0. 0. 0.]
>>>   [0. 2. 0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
>>>   [0. 0. 0. 0. 0.]
>>>   [0. 0. 0. 0. 0.]]
```

```
>>>  [[0. 0. 0. 0. 0.]
>>>   [0. 1. 1. 0. 0.]
>>>   [0. 0. 1. 1. 0.]
>>>   [0. 0. 0. 0. 0.]]
```

```
>>>  [[0. 0. 0. 0. 0.]
>>>   [0. 0. 0. 0. 0.]
>>>   [0. 0. 0. 2. 0.]
>>>   [0. 0. 0. 1. 2.]]]
```

```
>>> number_of_pixels =    (3, 4, 5)
>>> pixel_size =          [22.  16.5 13.2]
```

```
>>> xlim =                [ 1. 67.]
>>> ylim =                [ 2. 68.]
>>> zlim =                [ 3. 69.]
```

```
>>> pixel_grids:
>>> [array([ 1., 23., 45., 67.]),
>>>  array([ 2. , 18.5, 35. , 51.5, 68. ]),
>>>  array([ 3. , 16.2, 29.4, 42.6, 55.8, 69. ])]
```

Note that it is important to define the *number_of_pixels*.

Attributes

pixels: (M, N) numpy.ndarray The 2D numpy array containing the number of lines that pass through each pixel. They are stored as *float`s. 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. The number of pixels in each dimension is defined by `number_of_pixels*.

number_of_pixels: 2-tuple A 2-tuple corresponding to the shape of *pixels*.

pixel_size: (2,) numpy.ndarray The lengths of a pixel in the x- and y-dimensions, respectively.

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].

pixel_grids: list[numpy.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.

Methods

save(filepath)	Save a *Pixels* instance as a binary *pickle* object.
load(filepath)	Load a saved / pickled *Pixels* object from *filepath*.
from_lines(lines, number_of_pixels, xlim = None, ylim = None, verbose = True)	Create a pixel space and traverse / pixellise a given sample of *lines*.
empty(number_of_pixels, xlim, ylim, verbose = False)	Create an empty pixel space for the 2D rectangle bounded by *xlim* and *ylim*.
get_cutoff(p1, p2)	Return a numpy array containing the minimum and maximum value found across the two input arrays.
add_lines(lines, verbose = False)	Pixellise a sample of lines, adding 1 to each pixel traversed, for each line in the sample.
cube_trace(index, color = None, opacity = 0.4, colorbar = True, colorscale = "magma")	Get the Plotly *Mesh3d* trace for a single pixel at *index*.
cubes_traces(condition = lambda pixels: pixels > 0, color = None, opacity = 0.4, colorbar = True, colorscale = "magma")	Get a list of Plotly *Mesh3d* traces for all pixel selected by the *condition* filtering function.
pixels_trace(condition = lambda pixels: pixels > 0, size = 4, color = None, opacity = 0.4, colorbar = True, colorscale = "Magma", colorbar_title = None)	Create and return a trace for all the pixels in this class, with possible filtering.
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.

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

Methods

__init__(*args, **kwargs)	
add_lines(lines[, verbose])	Pixellise a sample of lines, adding 1 to each pixel traversed, for each line in the sample.
all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out])	Return indices of the maximum values along the given axis.
argmin([axis, out])	Return indices of the minimum values along the given axis.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap([inplace])	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.

continues on next page

Table 7 – continued from previous page

clip([min, max, out])	Return an array whose values are limited to [min, max].
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.
dot(b[, out])	Dot product of two arrays.
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
empty(number_of_pixels, xlim, ylim)	Create an empty pixel space for the 3D cube bounded by *xlim* and *ylim*.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
from_lines(lines, number_of_pixels[, xlim, ...])	Create a pixel space and traverse / pixellise a given sample of *lines*.
get_cutoff(p1, p2)	Return a numpy array containing the minimum and maximum value found across the two input arrays.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
heatmap_trace([colorscale, transpose, xgap, ...])	Create and return a Plotly *Heatmap* trace of the pixels.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
load(filepath)	Load a saved / pickled *Pixels* object from *filepath*.
max([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
pixels_trace([condition, opacity, colorscale])	Create and return a trace with all the pixels in this class, with possible filtering.
plot([ax])	Plot pixels as a heatmap using Matplotlib.
prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set a.flat[n] = values[n] for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.

continues on next page

Table 7 – continued from previous page

reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
save(filepath)	Save a *Pixels* instance as a binary *pickle* object.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
tostring([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
view([dtype][, type])	New view of array with the same data.

Attributes

T	The transposed array.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
itemsize	Length of one array element in bytes.

continues on next page

Table 8 – continued from previous page

nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
number_of_pixels	
pixel_grids	
pixel_size	
pixels	
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.
xlim	
ylim	

property pixels

property number_of_pixels

property xlim

property ylim

property pixel_size

property pixel_grids

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.

`static empty`(*number_of_pixels*, *xlim*, *ylim*)

Create an empty pixel space for the 3D cube bounded by *xlim* and *ylim*.

Parameters

number_of_pixels: (2,) numpy.ndarray A list-like containing the number of pixels to be created in the x- and y-dimension, respectively.

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]. Time the pixellisation step and print it to the terminal.

Raises

`ValueError` If *number_of_pixels* does not have exactly 2 values, or it has values smaller than 2. If *xlim* or *ylim* do not have exactly 2 values each.

`static get_cutoff`(*p1*, *p2*)

Return a numpy array containing the minimum and maximum value found across the two input arrays.

Parameters

p1 [(N,) `numpy.ndarray`] The first 1D numpy array.

p2 [(N,) `numpy.ndarray`] The second 1D numpy array.

Returns

(2,) `numpy.ndarray` The minimum and maximum value found across *p1* and *p2*.

Notes

The input parameters *must* be numpy arrays, otherwise an error will be raised.

`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:

```
>>> pixels = pept.Pixels.empty((640, 480), [0, 20], [0, 10])
>>> pixels.save("pixels.pickle")
```

```
>>> pixels_reloaded = pept.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:

```
>>> pixels = pept.Pixels.empty((640, 480), [0, 20], [0, 10])
>>> pixels.save("pixels.pickle")
```

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

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.

pixels_trace(*condition=<function Pixels.<lambda>>*, *opacity=0.9*, *colorscale='Magma'*)

Create and return 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

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()
```

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

Pixellise an array of lines and add them to a *PlotlyGrapher2D* instance:

```
>>> 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 = pept.visualisation.PlotlyGrapher2D()
>>> grapher.add_pixels(pixels)
>>> grapher.show()
```

Or add them directly to a raw *plotly.graph_objs* figure:

```
>>> import plotly.graph_objs as go
>>> 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()
```

T

The transposed array.

Same as `self.transpose()`.

See also:

transpose

Examples

```
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

all(*axis=None*, *out=None*, *keepdims=False*, ***, *where=True*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See also:

`numpy.all` equivalent function

any(*axis=None*, *out=None*, *keepdims=False*, ***, *where=True*)

Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

See also:

`numpy.any` equivalent function

argmax(*axis=None*, *out=None*)
Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

See also:

`numpy.argmax` equivalent function

argmin(*axis=None*, *out=None*)
Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

See also:

`numpy.argmin` equivalent function

argpartition(*kth*, *axis=- 1*, *kind='introselect'*, *order=None*)
Returns the indices that would partition this array.

Refer to *numpy.argpartition* for full documentation.

New in version 1.8.0.

See also:

`numpy.argpartition` equivalent function

argsort(*axis=- 1*, *kind=None*, *order=None*)
Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

See also:

`numpy.argsort` equivalent function

astype(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

Parameters

dtype [`str` or *`dtype`*] Typecode or data-type to which the array is cast.

order [{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok [bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy [bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t [ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Raises

ComplexWarning When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

base
Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

byteswap(*inplace=False*)
Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace [bool, optional] If `True`, swap bytes in-place, default is `False`.

Returns

out [ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

`A.newbyteorder().byteswap()` produces an array with the same values but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See also:

`numpy.choose` equivalent function

clip(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to *numpy.clip* for full documentation.

See also:

numpy.clip equivalent function

compress(*condition*, *axis=None*, *out=None*)
Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

See also:

numpy.compress equivalent function

conj()
Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

See also:

numpy.conjugate equivalent function

conjugate()
Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See also:

numpy.conjugate equivalent function

copy(*order='C'*)
Return a copy of the array.

Parameters

order [{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and numpy.copy() are very similar but have different default values for their order= arguments, and this function always passes sub-classes through.)

See also:

numpy.copy Similar function with different default behavior

numpy.copyto

Notes

This function is the preferred method for creating an array copy. The function numpy.copy() is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

ctypes

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters

None

Returns

c [Python `object`] Possessing attributes data, shape, strides, etc.

See also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.data

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

_ctypes.**shape**
(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be *ctypes.c_int*, *ctypes.c_long*, or *ctypes.c_longlong* depending on the platform. The c_intp type is defined accordingly in *numpy.ctypeslib*. The ctypes array contains the shape of the underlying array.

_ctypes.**strides**
(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

_ctypes.**data_as**(*obj*)
Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

_ctypes.**shape_as**(*obj*)
Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

_ctypes.**strides_as**(*obj*)
Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

cumprod(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

See also:

`numpy.cumprod` equivalent function

cumsum(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

See also:

`numpy.cumsum` equivalent function

data
Python buffer object pointing to the start of the array's data.

diagonal(*offset=0*, *axis1=0*, *axis2=1*)
Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See also:

`numpy.diagonal` equivalent function

dot(*b*, *out=None*)
Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

See also:

`numpy.dot` equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

dtype
Data-type of the array's elements.

Parameters

None

Returns

d [numpy *dtype* object]

See also:

`numpy.dtype`

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

dump(*file*)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file [`str` or `Path`] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

fill(*value*)
Fill the array with a scalar value.

Parameters

value [scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

flags
Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- WRITEBACKIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C) The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F) The data is in a single, Fortran-style contiguous segment.

OWNDATA (O) The array owns the memory it uses or borrows it from another object.

WRITEABLE (W) The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A) The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X) This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

UPDATEIFCOPY (U) (Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

FNC F_CONTIGUOUS and not C_CONTIGUOUS.

FORC F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B) ALIGNED and WRITEABLE.

CARRAY (CA) BEHAVED and C_CONTIGUOUS.

FARRAY (FA) BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

flatten(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order [{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y [ndarray] A copy of the input array, flattened to one dimension.

See also:

ravel Return a flattened array.

flat A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype [str or *dtype*] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset [int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

imag

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

item(**args*)

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** [Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z [Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset(**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args** [Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

itemsize
Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

max(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See also:

numpy.amax equivalent function

mean(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, ***, *where=True*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See also:

numpy.mean equivalent function

min(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

See also:

`numpy.amin` equivalent function

nbytes
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim
Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

newbyteorder(*new_order='S'*, */*)
Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

new_order [`str`, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian

- '=' - native order, equivalent to *sys.byteorder*
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr [array] New array object with the dtype reflecting given change to the byte order.

nonzero()

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See also:

numpy.nonzero equivalent function

partition(*kth*, *axis=- 1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth [int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind [{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.partition Return a parititioned copy of an array.

argpartition Indirect partition.

sort Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)
Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

See also:

numpy.prod equivalent function

ptp(*axis=None*, *out=None*, *keepdims=False*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

See also:

numpy.ptp equivalent function

put(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See also:

numpy.put equivalent function

ravel([*order*])
Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See also:

numpy.ravel equivalent function

ndarray.flat a flat iterator on the array.

real
The real part of the array.

See also:

`numpy.real` equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

repeat(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See also:

`numpy.repeat` equivalent function

reshape(*shape*, *order='C'*)
Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See also:

`numpy.reshape` equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

Parameters

new_shape [`tuple of ints, or n ints`] Shape of resized array.

refcheck [bool, optional] If False, reference count will not be checked. Default is True.

Returns

`None`

Raises

`ValueError` If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

`SystemError` If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See also:

`resize` Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See also:

`numpy.around` equivalent function

searchsorted(*v*, *side='left'*, *sorter=None*)
Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *numpy.searchsorted*

See also:

`numpy.searchsorted` equivalent function

setfield(*val*, *dtype*, *offset=0*)
Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val [`object`] Value to be placed in field.

dtype [*dtype* `object`] Data-type of the field in which to place *val*.

offset [`int`, optional] The number of bytes into the field at which to place *val*.

Returns

`None`

See also:

getfield

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write [bool, optional] Describes whether or not *a* can be written to.

align [bool, optional] Describes whether or not *a* is aligned properly for its type.

uic [bool, optional] Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
```

(continues on next page)

(continued from previous page)

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

See also:

`numpy.reshape` similar function

`ndarray.reshape` similar method

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

sort(*axis=- 1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind [{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort Return a sorted copy of an array.

numpy.argsort Indirect sort.

numpy.lexsort Indirect stable sort on multiple keys.

numpy.searchsorted Find elements in sorted array.

numpy.partition Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze(*axis=None*)

Remove axes of length one from *a*.

Refer to *numpy.squeeze* for full documentation.

See also:

numpy.squeeze equivalent function

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)

Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See also:

numpy.std equivalent function

strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

numpy.lib.stride_tricks.as_strided

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*)
Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

See also:

`numpy.sum` equivalent function

swapaxes(*axis1*, *axis2*)
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

See also:

`numpy.swapaxes` equivalent function

take(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

See also:

`numpy.take` equivalent function

tobytes(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order [{'C', 'F', 'A'}, optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s [`bytes`] Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

tofile(*fid*, *sep=''*, *format='%s'*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters

fid [`file` or `str` or `Path`] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep [`str`] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format [`str`] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

`tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *~numpy.ndarray.item* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

y [`object`, or `list` of `object`, or `list` of `list` of `object`, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
```

(continues on next page)

(continued from previous page)

```
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring(*order='C'*)

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *str*s.

Deprecated since version 1.19.0.

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See also:

`numpy.trace` equivalent function

transpose(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes [`None`, `tuple of ints`, `or n ints`]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out [`ndarray`] View of *a*, with axes suitably permuted.

See also:

`transpose` Equivalent function

`ndarray.T` Array property returning the array transposed.

ndarray.reshape Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)
Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

See also:

numpy.var equivalent function

view(*[dtype][, type]*)
New view of array with the same data.

Note: Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type [Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

pept.Voxels

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

Bases: *pept.base.iterable_samples.PEPTObject*, numpy.ndarray

A class that manages a single 3D voxel space, including tools for voxel traversal of lines, manipulation and visualisation.

This class can be instantiated in a couple of ways:

1. The constructor receives a pre-defined voxel space (i.e. a 3D numpy array), along with the space boundaries *xlim*, *ylim* and *zlim*.
2. The *from_lines* method receives a sample of 3D lines (i.e. a 2D numpy array), each defined by two points, creating a voxel space and traversing / voxellising the lines.
3. The *empty* method creates a voxel space filled with zeros.

This subclasses the *numpy.ndarray* class, so any *Voxels* object acts exactly like a 3D numpy array. All numpy methods and operations are valid on *Voxels* (e.g. add 1 to all voxels with *voxels += 1*).

It is possible to add multiple samples of lines to the same voxel space using the *add_lines* method.

If you want to voxellise multiple samples of lines, see the `pept.tracking.Voxelize` class.

See also:

pept.VoxelData Asynchronously manage multiple voxel spaces.

pept.LineData Encapsulate lines for ease of iteration and plotting.

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

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

Notes

The traversed lines do not need to be fully bounded by the voxel space. Their intersection is automatically computed.

The class saves *voxels* 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

This class is most often instantiated from a sample of lines to voxellise:

```
>>> import pept
>>> import numpy as np
```

```
>>> lines = np.arange(70).reshape(10, 7)
```

```
>>> number_of_voxels = [3, 4, 5]
>>> voxels = pept.Voxels.from_lines(lines, number_of_voxels)
>>> Initialised Voxels class in 0.0006861686706542969 s.
```

```
>>> print(voxels)
>>> voxels:
>>> [[[2. 1. 0. 0. 0.]
>>>   [0. 2. 0. 0. 0.]
>>>   [0. 0. 0. 0. 0.]
>>>   [0. 0. 0. 0. 0.]]
```

```
>>>  [[0. 0. 0. 0. 0.]
>>>   [0. 1. 1. 0. 0.]
>>>   [0. 0. 1. 1. 0.]
>>>   [0. 0. 0. 0. 0.]]
```

```
>>>  [[0. 0. 0. 0. 0.]
>>>   [0. 0. 0. 0. 0.]
>>>   [0. 0. 0. 2. 0.]
>>>   [0. 0. 0. 1. 2.]]]
```

```
>>> number_of_voxels =    (3, 4, 5)
>>> voxel_size =          [22.  16.5 13.2]
```

```
>>> xlim =                [ 1. 67.]
>>> ylim =                [ 2. 68.]
>>> zlim =                [ 3. 69.]
```

```
>>> voxel_grids:
>>> [array([ 1., 23., 45., 67.]),
>>>  array([ 2. , 18.5, 35. , 51.5, 68. ]),
>>>  array([ 3. , 16.2, 29.4, 42.6, 55.8, 69. ])]
```

Note that it is important to define the *number_of_voxels*.

Attributes

voxels: (M, N, P) numpy.ndarray The 3D numpy array containing the number of lines that pass through each voxel. They are stored as *float`s. 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. The number of voxels in each dimension is defined by `number_of_voxels*.

number_of_voxels: 3-tuple A 3-tuple corresponding to the shape of *voxels*.

voxel_size: (3,) numpy.ndarray The lengths of a voxel in the x-, y- and z-dimensions, respectively.

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].

voxel_grids: list[numpy.ndarray] 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.

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

Methods

__init__(*args, **kwargs)	
add_lines(lines[, verbose])	Voxellise a sample of lines, adding 1 to each voxel traversed, for each line in the sample.
all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out])	Return indices of the maximum values along the given axis.
argmin([axis, out])	Return indices of the minimum values along the given axis.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap([inplace])	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([deep])	Create a deep copy of an instance of this class, including all inner attributes.

continues on next page

Table 9 – continued from previous page

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.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.
dot(b[, out])	Dot product of two arrays.
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
empty(number_of_voxels, xlim, ylim, zlim)	Create an empty voxel space for the 3D cube bounded by *xlim*, *ylim* and *zlim*.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
from_lines(lines, number_of_voxels[, xlim, ...])	Create a voxel space and traverse / voxellise a given sample of *lines*.
get_cutoff(p1, p2)	Return a numpy array containing the minimum and maximum value found across the two input arrays.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
heatmap_trace([ix, iy, iz, width, ...])	Create and return a Plotly *Heatmap* trace of a 2D slice through the voxels.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
load(filepath)	Load a saved / pickled *Voxels* object from *filepath*.
max([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
plot([condition, ax, alt_axes])	Plot the voxels in this class using Matplotlib.
prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.

continues on next page

Table 9 – continued from previous page

round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
save(filepath)	Save a *Voxels* instance as a binary *pickle* object.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
tostring([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
view([dtype][, type])	New view of array with the same data.
voxels_trace([condition, size, color, ...])	Create and return a trace for all the voxels in this class, with possible filtering.

Attributes

T	The transposed array.
attrs	
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.

continues on next page

Table 10 – continued from previous page

itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
number_of_voxels	
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.
voxel_grids	
voxel_size	
voxels	
xlim	
ylim	
zlim	

property voxels

property number_of_voxels

property xlim

property ylim

property zlim

property voxel_size

property voxel_grids

property attrs

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.

static empty(*number_of_voxels*, *xlim*, *ylim*, *zlim*)

Create an empty voxel space for the 3D cube bounded by *xlim*, *ylim* and *zlim*.

Parameters

number_of_voxels: (3,) numpy.ndarray A list-like containing the number of voxels to be created in the x-, y- and z-dimension, respectively.

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].

Raises

`ValueError` If *number_of_voxels* does not have exactly 3 values, or it has values smaller than 2. If *xlim*, *ylim* or *zlim* do not have exactly 2 values each.

static get_cutoff(*p1*, *p2*)

Return a numpy array containing the minimum and maximum value found across the two input arrays.

Parameters

p1 [(N,) `numpy.ndarray`] The first 1D numpy array.

p2 [(N,) `numpy.ndarray`] The second 1D numpy array.

Returns

(2,) `numpy.ndarray` The minimum and maximum value found across *p1* and *p2*.

Notes

The input parameters *must* be numpy arrays, otherwise an error will be raised.

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:

```
>>> voxels = pept.Voxels.empty((64, 48, 32), [0, 20], [0, 10], [0, 5])
>>> voxels.save("voxels.pickle")
```

```
>>> voxels_reloaded = pept.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:

```
>>> voxels = pept.Voxels.empty((64, 48, 32), [0, 20], [0, 10], [0, 5])
>>> voxels.save("voxels.pickle")
```

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

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.

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 the faster *pept.plots.PlotlyGrapher*.

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)
```

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

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

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_traces(voxels.cubes_traces())  # small number of voxels
>>> grapher.show()
```

voxels_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()
```

T

The transposed array.

Same as `self.transpose()`.

See also:

transpose

Examples

```
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
```

(continues on next page)

(continued from previous page)

```
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

all(*axis=None*, *out=None*, *keepdims=False*, ***, *where=True*)
Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See also:

`numpy.all` equivalent function

any(*axis=None*, *out=None*, *keepdims=False*, ***, *where=True*)
Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

See also:

`numpy.any` equivalent function

argmax(*axis=None*, *out=None*)
Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

See also:

`numpy.argmax` equivalent function

argmin(*axis=None*, *out=None*)
Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

See also:

`numpy.argmin` equivalent function

argpartition(*kth*, *axis=- 1*, *kind='introselect'*, *order=None*)
Returns the indices that would partition this array.

Refer to *numpy.argpartition* for full documentation.

New in version 1.8.0.

See also:

`numpy.argpartition` equivalent function

argsort(*axis=- 1*, *kind=None*, *order=None*)
Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

See also:

`numpy.argsort` equivalent function

astype(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

Parameters

dtype [str or *dtype*] Typecode or data-type to which the array is cast.

order [{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok [bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy [bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t [ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Raises

ComplexWarning When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

base
Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

byteswap(*inplace=False*)
Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace [bool, optional] If `True`, swap bytes in-place, default is `False`.

Returns

out [ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

A.newbyteorder().byteswap() produces an array with the same values but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See also:

`numpy.choose` equivalent function

clip(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to *numpy.clip* for full documentation.

See also:

`numpy.clip` equivalent function

compress(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

See also:

`numpy.compress` equivalent function

conj()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

See also:

`numpy.conjugate` equivalent function

conjugate()

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See also:

`numpy.conjugate` equivalent function

copy(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

ctypes
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters

None

Returns

c [Python `object`] Possessing attributes data, shape, strides, etc.

See also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.data
A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

_ctypes.shape
(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be *ctypes.c_int*, *ctypes.c_long*, or *ctypes.c_longlong* depending on the platform. The c_intp type is defined accordingly in *numpy.ctypeslib*. The ctypes array contains the shape of the underlying array.

_ctypes.strides
(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

_ctypes.data_as(*obj*)
Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

_ctypes.**shape_as**(*obj*)
Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

_ctypes.**strides_as**(*obj*)
Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

cumprod(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

See also:

`numpy.cumprod` equivalent function

cumsum(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

See also:

`numpy.cumsum` equivalent function

data
Python buffer object pointing to the start of the array's data.

diagonal(*offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See also:

`numpy.diagonal` equivalent function

dot(*b*, *out=None*)

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

See also:

`numpy.dot` equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

dtype

Data-type of the array's elements.

Parameters

None

Returns

d [numpy *dtype* object]

See also:

`numpy.dtype`

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

dump(*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file [str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

fill(*value*)

Fill the array with a scalar value.

Parameters

value [scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

flags

Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- WRITEBACKIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C) The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F) The data is in a single, Fortran-style contiguous segment.

OWNDATA (O) The array owns the memory it uses or borrows it from another object.

WRITEABLE (W) The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A) The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X) This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

UPDATEIFCOPY (U) (Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

FNC F_CONTIGUOUS and not C_CONTIGUOUS.

FORC F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B) ALIGNED and WRITEABLE.

CARRAY (CA) BEHAVED and C_CONTIGUOUS.

FARRAY (FA) BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

flatten(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order [{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y [ndarray] A copy of the input array, flattened to one dimension.

See also:

ravel Return a flattened array.

flat A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype [str or *dtype*] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset [int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

imag

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

item(**args*)

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** [Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z [Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset(**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args** [Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

itemsize
Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

max(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See also:

numpy.amax equivalent function

mean(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, ***, *where=True*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See also:

numpy.mean equivalent function

min(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

See also:

`numpy.amin` equivalent function

nbytes
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim
Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

newbyteorder(*new_order='S'*, */*)
Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

new_order [str, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian

- '=' - native order, equivalent to *sys.byteorder*
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr [array] New array object with the dtype reflecting given change to the byte order.

nonzero()

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See also:

numpy.nonzero equivalent function

partition(*kth*, *axis=- 1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth [int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind [{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.partition Return a parititioned copy of an array.

argpartition Indirect partition.

sort Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)
Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

See also:

numpy.prod equivalent function

ptp(*axis=None*, *out=None*, *keepdims=False*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

See also:

numpy.ptp equivalent function

put(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See also:

numpy.put equivalent function

ravel([*order*])
Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See also:

numpy.ravel equivalent function

ndarray.flat a flat iterator on the array.

real
The real part of the array.

See also:

`numpy.real` equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

repeat(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See also:

`numpy.repeat` equivalent function

reshape(*shape*, *order='C'*)
Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See also:

`numpy.reshape` equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

Parameters

new_shape [`tuple of ints, or n ints`] Shape of resized array.

refcheck [bool, optional] If False, reference count will not be checked. Default is True.

Returns

`None`

Raises

`ValueError` If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

`SystemError` If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See also:

resize Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See also:

`numpy.around` equivalent function

searchsorted(*v*, *side='left'*, *sorter=None*)
Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *numpy.searchsorted*

See also:

`numpy.searchsorted` equivalent function

setfield(*val*, *dtype*, *offset=0*)
Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val [`object`] Value to be placed in field.

dtype [*dtype* `object`] Data-type of the field in which to place *val*.

offset [`int`, optional] The number of bytes into the field at which to place *val*.

Returns

`None`

See also:

getfield

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write [bool, optional] Describes whether or not *a* can be written to.

align [bool, optional] Describes whether or not *a* is aligned properly for its type.

uic [bool, optional] Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
```

(continues on next page)

(continued from previous page)

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

See also:

`numpy.reshape` similar function

`ndarray.reshape` similar method

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

sort(*axis=- 1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind [{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort Return a sorted copy of an array.

numpy.argsort Indirect sort.

numpy.lexsort Indirect stable sort on multiple keys.

numpy.searchsorted Find elements in sorted array.

numpy.partition Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze(*axis=None*)

Remove axes of length one from *a*.

Refer to *numpy.squeeze* for full documentation.

See also:

`numpy.squeeze` equivalent function

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)

Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See also:

`numpy.std` equivalent function

strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*)
Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

See also:

`numpy.sum` equivalent function

swapaxes(*axis1*, *axis2*)
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

See also:

`numpy.swapaxes` equivalent function

take(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

See also:

`numpy.take` equivalent function

tobytes(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order [{'C', 'F', 'A'}, optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s [`bytes`] Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

tofile(*fid*, *sep=''*, *format='%s'*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters

fid [`file` or `str` or `Path`] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep [`str`] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format [`str`] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

`tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *~numpy.ndarray.item* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

y [`object`, or `list` of `object`, or `list` of `list` of `object`, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
```

(continues on next page)

(continued from previous page)

```
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring(*order='C'*)

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *str*s.

Deprecated since version 1.19.0.

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See also:

`numpy.trace` equivalent function

transpose(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes [`None`, `tuple of ints`, `or n ints`]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out [`ndarray`] View of *a*, with axes suitably permuted.

See also:

`transpose` Equivalent function

`ndarray.T` Array property returning the array transposed.

ndarray.reshape Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)
Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

See also:

numpy.var equivalent function

view(*[dtype][, type]*)
New view of array with the same data.

Note: Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type [Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

pept.Pipeline

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

Bases: *pept.base.iterable_samples.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)
```

(continues on next page)

(continued from previous page)

```
>>> 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*.
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: collections.abc.Iterable*, *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 [`IterableSamples`] Any subclass of *IterableSamples* (e.g. *pept.LineData*) that allows iterating through samples of data.

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)].

`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.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

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: *pept.base.iterable_samples.PEPTObject*, collections.abc.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.ABC, *pept.base.iterable_samples.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: *pept.base.pipelines.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: collections.abc.Iterable*, *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: *pept.base.pipelines.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: *pept.base.pipelines.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: collections.abc.Iterable[*pept.base.point_data.PointData*]*, *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: *pept.base.pipelines.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: collections.abc.Iterable[pept.base.line_data.LineData]*, *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: *pept.base.pipelines.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(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: collections.abc.Iterable[*pept.base.voxel_data.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
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: *pept.base.iterable_samples.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
>>>
>>>     def fit_sample(self, sample: pept.LineData):
>>>         processed_points = ...
>>>         return pept.PointData(processed_points)
```

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(column)	Stack all samples and split them into a list according to a named / numeric column index.
pept.tracking.Centroids([error, cluster_size])	Compute the geometric centroids of a list of samples of points.

continues on next page

Table 28 – continued from previous page

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.Remove(*columns)	Remove columns (either column names or indices) from *pept.LineData* or *pept.PointData*.

pept.tracking.Stack

class pept.tracking.**Stack**(*sample_size=None*, *overlap=None*)

Bases: pept.base.pipelines.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: collections.abc.Iterable*)

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: *pept.base.pipelines.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.lines exists
PointData -> SplitLabels.fit_sample -> list[LineData]
```

The sample of data must have a column named exactly "label". The filter normally removes the "label" column in the output (if `remove_label = True`).

__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:* pept.base.iterable_samples.IterableSamples)

copy(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples:* *collections.abc.Iterable*, *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

class pept.tracking.**SplitAll**(*column*)

Bases: *pept.base.pipelines.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.

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: collections.abc.Iterable*)

pept.tracking.Centroids

class pept.tracking.**Centroids**(*error=False*, *cluster_size=False*)

Bases: *pept.base.pipelines.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)
```

__init__(*error=False*, *cluster_size=False*)

Methods

__init__([error, cluster_size])	
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: collections.abc.Iterable*, *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: *pept.base.pipelines.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: collections.abc.Iterable*, *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: *pept.base.pipelines.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:* pept.base.iterable_samples.IterableSamples)

copy(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples: collections.abc.Iterable*, *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.Remove

class pept.tracking.**Remove**(**columns*)

Bases: *pept.base.pipelines.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:* pept.base.iterable_samples.IterableSamples)

copy(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

fit(*samples: collections.abc.Iterable*, *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.Voxelize

class pept.tracking.**Voxelize**(*number_of_voxels*, *xlim=None*, *ylim=None*, *zlim=None*, *set_lims=None*)

Bases: *pept.base.pipelines.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_lines [(`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_lines*.

__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: collections.abc.Iterable[*pept.base.line_data.LineData*]*, *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: *pept.base.pipelines.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: collections.abc.Iterable[pept.base.point_data.PointData], 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")
```

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: *pept.base.pipelines.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...]*.

get_used [bool, default False] If *True*, the function will 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.

as_array [bool, default True] If set to True, the tracked locations are returned as numpy arrays. If set to False, they are returned inside an instance of *pept.PointData* for ease of iteration and plotting.

verbose [bool, default False] Provide extra information when tracking a location: time the operation and show a progress bar.

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: collections.abc.Iterable[*pept.base.line_data.LineData*]*, *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: *pept.base.pipelines.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

line_data [instance of *pept.LineData*] The LoRs for which the cutpoints will be computed. It must be an instance of *pept.LineData*.

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: collections.abc.Iterable[*pept.base.line_data.LineData*]*, *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: *pept.base.pipelines.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.

Notes

Once instantiated with a *LineData*, the class computes the minpoints and *automatically sets the sample_size* to the average number of minpoints found per sample of LoRs.

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*.

sample_size, overlap, number_of_lines, etc. [inherited from *pept.PointData*] Additional attributes and methods are inherited from the base class *PointData*. Check its documentation for more information.

Methods

find_minpoints(line_data, num_lines, max_distance, cutoffs = None, append_indices = False, max_workers = None, verbose = True)	Compute the minpoints from the samples in a *LineData* instance.
sample, to_csv, plot, etc.	(inherited from pept.PointData) Additional attributes and methods are inherited from the base class *PointData*. Check its documentation for more information.

__init__(*num_lines*, *max_distance*, *cutoffs=None*, *append_indices=False*)
Cutpoints class constructor.

Parameters

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, 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.

max_workers [int, optional] The maximum number of threads that will be used for asynchronously computing the minpoints from the samples of LoRs in *line_data*.

verbose [bool, default True] Provide extra information when computing the cutpoints: time the operation and show a progress bar.

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, ...])	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
num_lines

copy(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

fit(*line_data: collections.abc.Iterable[*pept.base.line_data.LineData*]*, *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: *pept.base.pipelines.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: collections.abc.Iterable[pept.base.point_data.PointData]*, *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: *pept.base.pipelines.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(line_data[, 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:* pept.base.voxel_data.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.

timestamp: float, default 0. The timestamp to associate with the tracer positions found in this voxel space.

as_array: bool, default False If *True*, return the found tracers' locations as a NumPy array. Otherwise, return them in a *pept.PointData* instance.

verbose: bool, default False Show extra information on the sample processing step.

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(*line_data: collections.abc.Iterable[*pept.base.voxel_data.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.Segregate

class pept.tracking.**Segregate**(*window*, *cut_distance*, *min_trajectory_size=5*)

Bases: *pept.base.pipelines.Reducer*

Segregate the intertwined points from multiple trajectories into individual paths.

Reducer signature:

```
pept.PointData -> Segregate.fit_sample -> pept.PointData
list[pept.PointData] -> Segregate.fit_sample -> 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 timeframe 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:

connect_trajectories 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.

__init__(*window*, *cut_distance*, *min_trajectory_size=5*)

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: collections.abc.Iterable[pept.base.point_data.PointData]*)

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")
```

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: *pept.base.pipelines.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: collections.abc.Iterable[pept.base.point_data.PointData], 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.

pept.processing.circles2d(positions, ...[, ...])	Compute the 2D occupancy of circles of different radii.
pept.processing.occupancy2d(points, ...[, ...])	Compute the 2D occupancy / residence time distribution of a single circular particle moving along a trajectory.

pept.processing.circles2d

pept.processing.**circles2d**(*positions*, *number_of_pixels*, *radii=0.0*, *xlim=None*, *ylim=None*, *verbose=True*)

Compute the 2D occupancy of circles of different radii.

This corresponds to the pixellisation of circular particles, such that each pixel's value corresponds to the area covered by the particle.

You must specify the particles' *positions* (2D numpy array) and the *number_of_pixels* in each dimension (*[nx, ny, nz]*). The *radii* can be either:

1. Zero: the particles are considered to be points. Each pixel will have a value +1 for every particle.
2. Single positive value: all particles have the same radius.
3. List of values of same length as *positions*: specify each particle's radius.

The pixel area's bounds can be specified in *xlim* and *ylim*. If unset, they will be computed automatically based on the minimum and maximum values found in *positions*.

Parameters

positions: (P, 2) numpy.ndarray The particles' 2D positions, where each row is formatted as *[x_coordinate, y_coordinate]*.

number_of_pixels: (2,) list-like The number of pixels in the x-dimension and y-dimension. Each dimension must have at least 2 pixels.

radii: float or (P,) list-like The radius of each particle. If zero, every particle is considered as a discrete point. If a single *float*, all particles are considered to have the same radius. If it is a numpy array, it specifies each particle's radius, and must have the same length as *positions*.

xlim: (2,) list-like, optional The limits of the system over which the pixels span in the x-dimension, formatted as [xmin, xmax]. If unset, they will be computed automatically based on the minimum and maximum values found in *positions*.

ylim: (2,) list-like, optional The limits of the system over which the pixels span in the y-dimension, formatted as [ymin, ymax]. If unset, they will be computed automatically based on the minimum and maximum values found in *positions*.

verbose: bool, default True Time the pixellisation step and print it to the terminal.

Returns

pept.Pixels (numpy.ndarray subclass) The created pixels, each cell containing the area covered by particles. The *pept.Pixels* class inherits all properties and methods from *numpy.ndarray*, so you can use it exactly like you would a numpy array. It just contains extra attributes (e.g. *xlim*, *ylim*) and some PEPT-oriented methods (e.g. *pixels_trace*).

Raises

ValueError If *positions* is not a 2D array-like with exactly 2 columns, or *number_of_pixels* is not a 1D list-like with exactly 2 values or it contains a value smaller than 2. If *radii* is a list-like that does not have the same length as *positions*.

Examples

Create ten random particle positions between 0-100 and radii between 0.5-2.5:

```
>>> positions = np.random.random((10, 2)) * 100
>>> radii = 0.5 + np.random.random(len(positions)) * 2
```

Now pixellise those particles as circles over a grid of (20, 10) pixels:

```
>>> import pept.processing as pp
>>> num_pixels = (20, 10)
>>> pixels = pp.circles2d(positions, num_pixels, radii)
```

Alternatively, specify the system's bounds explicitly:

```
>>> pixels = pp.circles2d(
>>>     positions, (20, 10), radii, xlim = [10, 90], ylim = [-5, 105]
>>> )
```

You can plot those pixels in two ways - using *PlotlyGrapher* (this plots a 3D "heatmap", as a coloured surface):

```
>>> from pept.visualisation import PlotlyGrapher
>>> grapher = PlotlyGrapher()
>>> grapher.add_pixels(pixels)
>>> grapher.show()
```

Or using raw *Plotly* (this plots a "true" heatmap) - this is recommended:

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

pept.processing.occupancy2d

pept.processing.**occupancy2d**(*points*, *number_of_pixels*, *radius*, *xlim=None*, *ylim=None*, *omit_last=False*, *verbose=True*)

Compute the 2D occupancy / residence time distribution of a single circular particle moving along a trajectory.

This corresponds to the pixellisation of moving circular particles, such that for every two consecutive particle locations, a 2D cylinder (i.e. convex hull of two circles at the two particle positions), the fraction of its area that intersets a pixel is multiplied with the time between the two particle locations and saved in the input *pixels*.

You must specify the *points* (2D numpy array) recorded along a particle's trajectory, formatted as [time, x, y] of each location, along with the *number_of_pixels* in each dimension (*[nx, ny]*) and particle *radius*.

The pixel area's bounds can be specified in *xlim* and *ylim*. If unset, they will be computed automatically based on the minimum and maximum values found in *points*.

Parameters

points: (P, 3) numpy.ndarray The particles' 2D locations and corresponding timestamp, where each row is formatted as *[time, x_coordinate, y_coordinate]*. Must have at least two points.

number_of_pixels: (2,) list-like The number of pixels in the x-dimension and y-dimension. Each dimension must have at least 2 pixels.

radius: float The radius of the particle. It can be given in any system of units, as long as it is consistent with what is used for the particle locations.

xlim: (2,) list-like, optional The limits of the system over which the pixels span in the x-dimension, formatted as [xmin, xmax]. If unset, they will be computed automatically based on the minimum and maximum values found in *positions*.

ylim: (2,) list-like, optional The limits of the system over which the pixels span in the y-dimension, formatted as [ymin, ymax]. If unset, they will be computed automatically based on the minimum and maximum values found in *positions*.

omit_last: bool, default False If true, omit the last circle in the particle positions. Useful if rasterizing the same trajectory piece-wise; if you split the trajectory and call this function multiple times, set *omit_last = 0* to avoid considering the last particle location twice.

verbose: bool, default True Time the pixellisation step and print it to the terminal.

Returns

pept.Pixels (*numpy.ndarray* **subclass**) The created pixels, each cell containing the area covered by particles. The *pept.Pixels* class inherits all properties and methods from *numpy.ndarray*, so you can use it exactly like you would a numpy array. It just contains extra attributes (e.g. *xlim*, *ylim*) and some PEPT-oriented methods (e.g. *pixels_trace*).

Raises

`ValueError` If *positions* is not a 2D array-like with exactly 3 columns, or *number_of_pixels* is not a 1D list-like with exactly 2 values or it contains a value smaller than 2. If *xlim* or *ylim* have *max* < *min* or there are particle positions falling outside the system defined by *xlim* and *ylim*, including the area.

Examples

Create ten random particle positions between 0-100 and radius 0.2:

```
>>> positions = np.random.random((10, 2)) * 100
>>> radius = 0.2
```

Now pixellise this trajectory over a grid of (20, 10) pixels:

```
>>> import pept.processing as pp
>>> num_pixels = (20, 10)
>>> pixels = pp.occupancy2d(positions, num_pixels, radius)
```

Alternatively, specify the system's bounds explicitly:

```
>>> pixels = pp.occupancy2d(
>>>     positions, (20, 10), radius, xlim = [10, 90], ylim = [-5, 105]
>>> )
```

You can plot those pixels in two ways - using *PlotlyGrapher* (this plots a 3D "heatmap", as a coloured surface):

```
>>> from pept.visualisation import PlotlyGrapher
>>> grapher = PlotlyGrapher()
>>> grapher.add_pixels(pixels)
>>> grapher.show()
```

Or using raw *Plotly* (this plots a "true" heatmap) - this is recommended:

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

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.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.PlotlyGrapher

class pept.plots.**PlotlyGrapher**(*rows=1*, *cols=1*, *xlim=None*, *ylim=None*, *zlim=None*, *subplot_titles=[' ']*)

Bases: *pept.base.iterable_samples.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* or *pept.VoxelData* 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*)

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*)

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

voxels [*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* or *pept.VoxelData* 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* or `pept.VoxelData`] The voxel space, encapsulated in a *pept.Voxels* or *pept.VoxelData* instance (or subclass thereof). If a *VoxelData* is received, all the voxels will be accumulated / superimposed. Only these classes are accepted as raw voxels on their own do not contain data about the spatial coordinates of the voxel 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 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.

copy(*deep=True*)
Create a deep copy of an instance of this class, including all inner attributes.

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.

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_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.

continues on next page

Table 58 – continued from previous page

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*)

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*)

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*)

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*)

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*)

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*)

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*)
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_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]
```

(continues on next page)

(continued from previous page)

```
>>> 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)	

continues on next page

Table 64 – continued from previous page

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

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

X

Y

Z