postpic Documentation

Release v0.3.1+298.g833fe56.dirty

the postpic developers

Mar 13, 2018

Contents

CHAPTER 1

Introduction

POSTPIC

The open source particle-in-cell post processor.

Particle-in-cell simulations are a valuable tool for the simulation of non-equelibrium systems in plasma- or astrophysics. Such simulations usually produce a large amount of data consisting of electric and magnetic field data as well as particle positions and momenta. While there are various PIC codes freely available, the task of post-processing – essentially condensing the large amounts of data into small units suitable for plotting routines – is typically left to each user individually. As post-processing may be a time consuming and error-prone process, this python package has been developed.

Postpic can handle two different types of data:

Field data which is data sampled on a predefined grid, such as electic and magnetic fields, particle- or charge densities, currents, etc. Fields are usually the data, which can be plotted directly. See `postpic.Field`.

Particle data which is data of multiple particles and for each particle positions (x, y, z) and momenta (px, py, pz) are known. Particles usually also have *weight*, *charge*, *time* and a unique *id*. Postpic can transform particle data to field data using the same algorithm and particle shapes, which are used in most PIC Simulations. The particle-to-grid routines are written in C for maximum performance. See `postpic.MultiSpecies`.

1.1 What is postpic?

Postpic is an open-source package aiming to ease the postprocessing of particle-in-cell simulation data. Particle-in-cell simulations are often used to simulate the behaviour of plasmas in non-equilibrium states. The Datareader package contains methods and interfaces to read data from any Simulation.

The basic concept consits of two different types of readers:

1.1.1 The Dumpreader

This has to be subclassed from Dumpreader_ifc and allows to read a single dump created by the simulation. To identify which dump should be read its initialized with a dumpidentifier. This dumpidentifier can be almost anything, but in the easiest case this is the filepath pointing to a single file containing every information about this simulation dump. With this information the dumpreader must be able to read all data regarding this dump (which is a lot: X, Y, Z, Px, Py, Py, weight, mass, charge, ID,.. for all particle species, electric and magnetic fields on grid, the grid itself, mabe particle ids,. . .)

1.1.2 The Simulationreader

This has to be subclassed from Simulationreader_ifc and allows to read a full list of simulation dumps. Thus an alternate Name for this class could be "Dumpsequence". This allows the code to track particles from different times of the simulation or create plots with a time axis.

Stephan Kuschel 2014

postpic.datareader.**chooseCode**(*code*)

Chooses appropriate reader for the given simulation code. After choosing a preset of the correct reader, the functions *postpic.readDump()* and *postpic.readSim()* are setup for this preset.

Parameters **code** (*string*) –

Possible options are:

- "DUMMY": dummy class creating fake data.
- "EPOCH": .sdf files written by EPOCH1D, EPOCH2D or EPOCH3D.
- "openPMD": .h5 files written in openPMD Standard
- "piconGPU": same as "openPMD"
- "VSIM": .hdf5 files written by VSim.

postpic.datareader.**readDump**(*dumpidentifier*, ***kwargs*)

After using the fucntion *postpic.chooseCode()*, this function should be the main function for reading a dump into postpic.

Parameters

- **dumpidentifier** (*str*) – Identifies the dump. For most dumpreaders this is a string poining to the file or folder. See what the specific reader of your format expects.
- ****kwargs** – will be forwarded to the dumpreader.

Returns the dumpreader for this specific data dump.

Return type Dumpreader

postpic.datareader.**readSim**(*simidentifier*, ***kwargs*)

After using the function *postpic.chooseCode()*, this function should be the main function for reading a simulation into postpic. A simulation is equivalent to a series of dumps in a specific order (not neccessarily time order).

Parameters

- **simidentifier** (*str*) – Identifies the simulation. For EPOCH this should be a string pointing to a *.visit* file. Specifics depend on the current simreader class, as set by *chooseCode*.
- ****kwargs** – will be forwarded to the simreader.

Returns the Simulationreader

postpic.datareader.**setdumpreadercls**(*dumpreadercls*)
Sets the class that is used for reading dumps on later calls of *postpic.readDump()*.

Parameters **dumpreadercls** (Dumpreader_ifc) –

Note: This should only be used, for testing. A set of presets is provided by *postpic.chooseCode()*.

postpic.datareader.**setsimreadercls**(*simreadercls*)
Sets the class that is used for reading a simulation on later calls of *postpic.readSim()*.

Parameters **simreadercls** (Simulationreader_ifc) –

Note: This should only be used, for testing. A set of presets is provided by *postpic.chooseCode()*.

class postpic.datareader.**Dumpreader_ifc**(*dumpidentifier*, *name=None*)
Interface class for reading a single dump. A dump contains informations about the simulation at a single timestep (Usually E- and B-Fields on grid + particles).

Any Dumpreader_ifc implementation will always be initialized using a dumpidentifier. This dumpidentifier can be anything, that points to the data of a dump. In the easiest case this is just the filename of the dump holding all data of that timestep (for example .sdf file for EPOCH, .hdf5 for some other code).

The dumpreader should provide all necessary informations in a unified interface but at the same time it should not restrict the user to these properties of there dump only. The recommended implementation is shown here (EPOCH and VSim reader work like this): All (!) data, that is saved in a single dump should be accessible via the self.__getitem__(key) method. Together with the self.keys() method, this will ensure, that every dumpreader works as a dictionary and every dump attribute is accessible via this dictionary.

- Level 0:

__getitem__ and keys(self) are level 0 methods, meaning it must be possible to access everthing with those methods.

- Level 1:

provide direct data access by forwarding the requests to the corresponding Level 0 or Level 1 methods.

- Level 2:

provide user access to the data by forwarding the request to Level 1 or Level 2 methods, but NOT to Level 0 methods.

If some attribute wasnt dumped a KeyError must be thrown. This allows classes which are using the reader to just exit if a needed property wasnt dumped or to catch the KeyError and proceed by actively ignoring it.

It is highly recommended to also override the functions __str__ and gridpoints.

Parameters **dumpidentifier** – variable type whatever identifies the dump. It is recommended to use a String here pointing to a file.

data(*key*)
access to every raw data. needs to return numpy arrays corresponding to the “key”.

getSpecies(*species*, *attrib*)
This function gives access to any of the particle properties in ..helper.attribidentify This method can behave in the following ways: 1) Return a list of scalar properties for each particle of this species 2) Return a single float (i.e. *1.2*, NOT *[1.2]*) to show that

every particle of this species has the same scalar value to thisdimmax property assigned. This might be quite often used for charge or mass that are defined per species.

3. Raise a KeyError if the requested property or species wasn dumped.

gridnode(*key*, *axis*)
The grid nodes along "axis". Grid nodes include the beginning and the end of the grid. Example: If the grid has 20 grid points, it has 21 grid nodes or grid edges.

gridoffset(*key*, *axis*)
offset of the beginning of the first cell of the grid.

gridpoints(*key*, *axis*)
Number of grid points along "axis". It is highly recommended to override this method due to performance reasons.

gridspacing(*key*, *axis*)
size of one grid cell in the direction "axis".

keys()

Returns a list of keys, that can be used in __getitem__ to read any information from this dump.

simdimensions()
the number of spatial dimensions the simulations was using. Must be 1, 2 or 3.

simextent(*axis*)
returns the extent of the actual simulation box. Override in your own reader class for better performance implementation.

class postpic.datareader.**Simulationreader_ifc**(*simidentifier*, *name=None*)
Interface for reading the data of a full Simulation.

Any Simulationreader_ifc implementation will always be initialized using a simidentifier. This simidentifier can be anything, that points to the data of multiple dumps. In the easiest case this can be the .visit file.

The Simulationreader_ifc is subclass of collections.Sequence and will thus behave as a Sequence. The Objects in the Sequence are supposed to be subclassed from Dumpreader_ifc.

It is highly recommended to also override the __str__ function.

Parameters **simidentifier** – variable type something identifiying a series of dumps.

postpic.**readDump**(*dumpidentifier*, ***kwargs*)
After using the fucntion *postpic.chooseCode()*, this function should be the main function for reading a dump into postpic.

Parameters

- **dumpidentifier** (*str*) – Identifies the dump. For most dumpreaders this is a string poining to the file or folder. See what the specific reader of your format expects.
- ****kwargs** – will be forwarded to the dumpreader.

Returns the dumpreader for this specific data dump.

Return type Dumpreader

postpic.**readSim**(*simidentifier*, ***kwargs*)
After using the function *postpic.chooseCode()*, this function should be the main function for reading a simulation into postpic. A simulation is equivalent to a series of dumps in a specific order (not neccessarily time order).

Parameters

- **simidentifier** (*str*) – Identifies the simulation. For EPOCH this should be a string pointing to a *.visit* file. Specifics depend on the current simreader class, as set by *chooseCode*.
- ****kwargs** – will be forwarded to the simreader.

Returns the Simulationreader

CHAPTER 2

Getting started

The following script should just show an example how to get started using the postpic postprocessor. This script uses the dummy reader (thus auto generated random data). Thats why there is no input file needed to read the simulation data from.

```
#!/usr/bin/env python
#
# This file is part of postpic.
#
# postpic is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# postpic is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with postpic. If not, see <http://www.gnu.org/licenses/>.
#
# Copyright Stephan Kuschel 2015
#

def main():
    import numpy as np
    import postpic as pp

    # postpic will use matplotlib for plotting. Changing matplotlibs backend
    # to "Agg" makes it possible to save plots without a display attached.
    # This is necessary to run this example within the "run-tests" script
    # on travis-ci.
    import matplotlib; matplotlib.use('Agg')
```

```
    # choose the dummy reader. This reader will create fake data for testing.
    pp.chooseCode('dummy')

    dr = pp.readDump(3e5)  # Dummyreader takes a float as argument, not a string.
    # set and create directory for pictures.
    savedir = '_examplepictures/'
    import os
    if not os.path.exists(savedir):
        os.mkdir(savedir)

    # initialze the plotter object.
    # project name will be prepended to all output names
    plotter = pp.plotting.plottercls(dr, outdir=savedir, autosave=True, project=
↪'simpleexample')

    # we will need a refrence to the MultiSpecies quite often
    from postpic.particles import MultiSpecies

    # create MultiSpecies Object for every particle species that exists.
    pas = [MultiSpecies(dr, s) for s in dr.listSpecies()]

    if True:
        # Plot Data from the FieldAnalyzer fa. This is very simple: every line␣
↪creates one plot
        plotter.plotField(dr.Ex())  # plot 0
        plotter.plotField(dr.Ey())  # plot 1
        plotter.plotField(dr.Ez())  # plot 2
        plotter.plotField(dr.energydensityEM())  # plot 3

        # Using the MultiSpecies requires an additional step:
        # 1) The MultiSpecies.createField method will be used to create a Field object
        # with choosen particle scalars on every axis
        # 2) Plot the Field object
        optargsh={'bins': [300,300]}
        for pa in pas:
            # create a Field object nd holding the number density
            nd = pa.createField('x', 'y', optargsh=optargsh, simextent=True)
            # plot the Field object nd
            plotter.plotField(nd, name='NumberDensity')    # plot 4
            # if you like to keep working with the just created number density
            # yourself, it will convert to an numpy array whenever needed:
            arr = np.asarray(nd)
            print('Shape of number density: {}'.format(arr.shape))

            # more advanced: create a field holding the total kinetic energy on grid
            ekin = pa.createField('x', 'y', weights='Ekin_MeV', optargsh=optargsh,␣
↪simextent=True)
            # The Field objectes can be used for calculations. Here we use this to
            # calculate the average kinetic energy on grid and plot
            plotter.plotField(ekin / nd, name='Avg Kin Energy (MeV)')  # plot 5
            # use optargsh to force lower resolution
            # plot number density
            plotter.plotField(pa.createField('x', 'y', optargsh=optargsh),␣
↪lineoutx=True, lineouty=True)  # plot 6
            # plot phase space
            plotter.plotField(pa.createField('x', 'p', optargsh=optargsh))  # plot 7
            plotter.plotField(pa.createField('x', 'gamma', optargsh=optargsh))  #␣
↪plot 8
```

```
            plotter.plotField(pa.createField('x', 'beta', optargsh=optargsh))  # plot␣
↪9

            # same with high resolution
            plotter.plotField(pa.createField('x', 'y', optargsh={'bins': [1000,1000]}
↪))  # plot 10
            plotter.plotField(pa.createField('x', 'p', optargsh={'bins': [1000,1000]}
↪))  # plot 11

            # advanced: postpic has already defined a lot of particle scalars as Px,␣
↪Py, Pz, P, X, Y, Z, gamma, beta, Ekin, Ekin_MeV, Ekin_MeV_amu, ... but if needed␣
↪you can also define your own particle scalar on the fly.
            # In case its regularly used it should be added to postpic. If you dont␣
↪know how, just let us know about your own useful particle scalar by email or adding␣
↪an issue at
            # https://github.com/skuschel/postpic/issues

            # define your own particle scalar: p_r = sqrt(px**2 + py**2)/p
            plotter.plotField(pa.createField('sqrt(px**2 + py**2)/p', 'sqrt(x**2 +␣
↪y**2)', optargsh={'bins':[400,400]}))  # plot 12

            # however, since its unknown to the program, what quantities were␣
↪calculated the axis of plot 12 will only say "unknown"
            # this can be avoided in two ways:
            # 1st: define your own ScalarProperty(name, expr, unit):
            p_perp = pp.particles.ScalarProperty('sqrt(px**2 + py**2)/p', name='p_perp
↪', unit='kg*m/s')
            r_xy = pp.particles.ScalarProperty('sqrt(x**2 + y**2)', name='r_xy', unit=
↪'m')
            # this will create an identical plot, but correcly labled
            plotter.plotField(pa.createField(p_perp, r_xy, optargsh={'bins':[400,400]}
↪))  # plot 13
            # if those quantities are reused often, teach postip to recognize them␣
↪within the string expression:
            pp.particles.particle_scalars.add(p_perp)
            #pp.particles.scalars.add(r_xy)  # we cannot execute this line, because r_
↪xy is already predefinded
            plotter.plotField(pa.createField('p_perp', 'r_xy', optargsh={'bins':[400,
↪400]}))  # plot 14



            # choose particles by their properies
            # this has been the old interface, which would still work
            # def cf(ms):
            #     return ms('x') > 0.0  # only use particles with x > 0.0
            # cf.name = 'x>0.0'
            # pa.compress(cf)
            # nicer is the new filter function, which does exactly the same:
            pf = pa.filter('x>0')
            # plot 15, compare with plot 10
            plotter.plotField(pf.createField('x', 'y', optargsh={'bins': [1000,1000]}
↪))
            # plot 16, compare with plot 12
            plotter.plotField(pf.createField('p_perp', 'r_xy', optargsh={'bins':[400,
↪400]}))

            plotter.plotField(dr.divE())  # plot 13
```

```
if __name__=='__main__':
    main()
```

CHAPTER 3

Changelog of postpic

3.1 current master

Incompatible adjustments to previous version

- `postpic.Field` method `exporttocsv` is removed. Use `export` instead.
- `postpic.Field` method `transform` is renamed to `map_coordinates`, matching the underlying scipy-function.
- `postpic.Field` method `mean` has now an interface matching `ndarray.mean`. This means that, if the `axis` argument is not given, it averages across all axes instead the last axis.
- `postpic.Field.map_coordinates` applies now the Jacobian determinant of the transformation, in order to preserve the definite integral. In your code you will need to turn calls to `Field.transform` into calls to `Field.map_coordinates` and set the keyword argument `preserve_integral=False` to get the old behaviour.
- The functions `MultiSpecies.compress`, `MultiSpecies.filter`, `MultiSpecies.uncompress` and `ParticleHistory.skip` return a new object now. Before this release, they modified the current object. Assuming `ms` is a `MultiSpecies` object, the corresponding adjustemens read: old: `ms.filter('gamma > 2')` new: `ms = ms.filter('gamma > 2')`

Other improvements and new features

- `postpic.Field` has methods `.loadfrom`, `.saveto` and `.export`. `.saveto` saves the complete Field object as a `.npz` file. Use `.loadfrom` to load a Field object from file. `.export` is able to write `.csv` files and `.vtk` files in addition.
- `postpic` has a new function `time_profile_at_plane` that 'measures' the temporal profile of a pulse while passing through a plane
- `postpic` has a new function `unstagger_fields` that will take a set of staggered fields and returns the fields after removing the stagger
- `postpic` has a new function `export_vector_vtk` that takes up to three fields and exports them as a vector field in the `.vtk` format

- `postpic` has a new function `export_scalars_vtk` that takes up to four fields and exports them as multiple scalar fields on the same grid in the `.vtk` format
- `postpic.Field` works now with all numpy ufuncs, also with `ufunc.reduce`, `ufunc.outer`, `ufunc.accumulate` and `ufunc.at`
- `postpic.Field` now supports broadcasting like numpy arrays, for binary operators as well as binary ufunc operations
- `postpic.Field` has methods `.swapaxes`, `.transpose` and property `.T` compatible to numpy.ndarray
- `postpic.Field` has methods `all`, `any`, `max`, `min`, `prod`, `sum`, `ptp`, `std`, `var`, `mean`, `clip` compatible to numpy.ndarray
- `postpic.Field` has a new method `map_axis_grid` for transforming the coordinates only along one axis which is simpler than `map_coordinates`, but also takes care of the Jacobian
- `postpic.Field` has a new method `autocutout` used to slice away close-to-zero regions from the borders
- `postpic.Field` has a new method `fft_autopad` used to pad a small number of grid points to each axis such that the dimensions of the Field are favourable to FFTW
- `postpic.Field` has a new method `adjust_stagger_to` to adjust the grid origin to match the grid origin of another field
- `postpic.Field.topolar` has new defaults for extent and shape
- `postpic.Field.integrate` now uses the simpson method by default
- New module `postpic.experimental` to contain experimental algorithms for your reference. These algorithms are not meant to be useable as-is, but may serve as recipes to write your own algorithms.
- k-space reconstruction from EPOCH dumps has greatly improved accuracy due to a new algorithm correctly incorporating the frequency response of the implicit linear interpolation performed by EPOCH's half-steps

3.2 v0.3.1

2017-10-03

Only internal changes. Versioning is handled by versioneer.

3.3 v0.3

2017-09-28

Many improvements in terms of speed and features. Unfortunately some changes are not backwards-compatible to v0.2.3, so you may have to adapt your code to the new interface. For details, see the corresponding section below.

Highlights

- kspace reconstruction and propagation of EM waves.
- `postpic.Field` properly handles operator overloading and slicing. Slicing can be index based (integers) or referring the actual physical extent on the axis of a Field object (using floats).
- Expression based interface to particle properties (see below)

Incompatible adjustments to previous version

- New dependency: Postpic requires the `numexpr` package to be installed now.

- Expression based interface of for particles: If `ms` is a `postpic.MultiSpecies` object, then the call `ms.X()` has been deprecated. Use `ms('x')` instead. This new particle interface can handle expressions that the `numexpr` package understands. Also `ms('sqrt(x**2 + gamma - id)')` is valid. This interface is easier to use, has better functionality and is faster due to `numexpr`. The list of known per particle scalars and their definitions is available at `postpic.particle_scalars`. In addition all constants of `scipy.constants.*` can be used. In case you find particle scalar that you use regularly which is not in the list, please open an issue and let us know!
- The `postpic.Field` class now behaves more like an `numpy.ndarray` which means that almost all functions return a new field object instead of modifying the current. This change affects the following functions: `half_resolution`, `autoreduce`, `cutout`, `mean`.

Other improvements and new features

- `postpic.helper.kspace` can reconstruct the correct k-space from three EM fields provided to distinguish between forward and backward propagating waves (thanks to @Ablinne)
- `postpic.helper.kspace_propagate` will turn the phases in k-space to propagate the EM-wave.
- List of new functions in `postpic` from `postpic.helper` (thanks to @Ablinne): `kspace_epoch_like`, `kspace`, `kspace_propagate`.
- `Field.fft` function for fft optimized with pyfftw (thanks to @Ablinne).
- `Field.__getitem__` to slice a Field object. If integers are provided, it will interpret them as gridpoints. If float are provided they are interpreted as the physical region of the data and slice along the corresponding axis positions (thanks to @Ablinne).
- `Field` class has been massively impoved (thanks to @Ablinne): The operator overloading is now properly implemented and thanks to `__array__` method, it can be interpreted by numpy as an ndarray whenever necessary.
- List of new functions of the `Field` class (thanks to @Ablinne): `meshgrid`, `conj`, `replace_data`, `pad`, `transform`, `squeeze`, `integrate`, `fft`, `shift_grid_by`, `__getitem__`, `__setitem__`.
- List of new properties of the `Field` class (thanks to @Ablinne): `matrix`, `real`, `imag`, `angle`.
- Many performance optimizations using pyfftw library (optional) or numexpr (now required by postpic) or by avoiding in memory data copying.
- Lots of fixes

3.4 v0.2.3

2017-02-17

This release brings some bugfixes and various new features.

Bugfixes

- Particle property Bz.
- plotting of contourlevels.

Improvements and new features

- openPMD support (thanks to @ax3l).
- ParticleHistory class to collect particle information over the entire simulation.
- added particle properties v{x,y,z} and beta{x,y,z}.
- Lots of performance improvemts: particle data will be much less copied in memory now.

3.5 v0.2.2 and earlier

There hasnt been any changelog. Dont use those versions anymore.

CHAPTER 4

Contributing to the postpic code base

Any help contributing to the postpic project ist greatly appreciated! Feel free to contact any of the developers or ask for help using the Issues Page.

4.1 Why me?

because you are using it!

4.2 How to contribute?

Reporting bugs or asking questions works with a GitHub account simply on the Issues page.

For any coding you need to be familiar with git. Its a distributed version control system created by Linus Torvalds (and more importantly: he is also using it for maintaining the linux kernel). There is a nice introduction to git at try.github.io/, but in general you can follow the bootcamp section at https://help.github.com/ for your first steps.

One of the most comprehensive guides is probably this book. Just start reading from the beginning. It is worth it!

4.3 The Workflow

Adding a feature is often triggered by the personal demand for it. Thats why production ready features should propagte to master as fast as possible. Everything on master is considered to be production ready. We follow the github-flow describing this very nicely.

In short:

1. Fork the PostPic repo to your own GitHub account.
2. Clone from your fork to your local computer.

3. Create a branch whose name tells what you do. Something like `codexy-reader` or `fixwhatever`,... is a good choice. Do NOT call it `issue42`. Git history should be clearly readable without external information. If its somehow unspecific in the worst case call it `dev` or even commit onto your `master` branch.
4. Implement a new feature/bugfix/documentation/whatever commit to your local repository. It is highly recommended that the new features will have test cases.
5. KEEP YOUR FORK UP TO DATE! Your fork is yours, only. So you have to update it to whatever happens in the main repository. To do so add the main repository as a second remote with

```
git remote add upstream git@github.com:skuschel/postpic.git
```

and pull from it regularly with

```
git pull --rebase upstream master
```

1. Make sure all tests are running smoothly (the `run-tests.py` script also involves pep8 style verification!) Run `run-tests.py` before EVERY commit!
2. push to your fork and create a pull request EARLY! Even if your feature or fix is not yet finished, create the pull request and start it with `WIP:` or `[WIP]` (work-in-progress) to show its not yet ready to merge in. But the pull request will * trigger travis.ci to run the tests whenever you push * show other people what you work on * ensure early feedback on your work

4.4 Coding and general remaks

- Make sure, that the `run-tests.py` script exits without error on EVERY commit. To do so, it is HIGHLY RECOMMENDED to add the `pre-commit` script as the git pre-commit hook. For instructions see pre-commit.
- The Coding style is according to slightly simplified pep8 rules. This is included in the `run-tests.py` script. If that script runs without error, you should be good to go commit.
- Add the GPLv3+ licence notice on top of every new file. If you add a new file you are free to add your name as a author. This will let other people know that you are in charge if there is any trouble with the code. This is only useful if the file you provide adds functionality like a new datareader. Thats why the `__init__.py` files typically do not have a name written. In doubt, the git revision history will always show who added which line.

4.5 What to contribute?

Here is a list for your inspiration:

- Add Documentation and usage examples.
- Report bugs at the Issues page.
- Fix bugs from the Issues page.
- Add python docstrings to the codebase.
- Add new features.
- Add new datareader for additional file formats.
- Add test cases
- ...

CHAPTER 5

Postpic API Documentation

5.1 postpic

5.1.1 postpic package

POSTPIC

The open source particle-in-cell post processor.

Particle-in-cell simulations are a valuable tool for the simulation of non-equelibrium systems in plasma- or astrophysics. Such simulations usually produce a large amount of data consisting of electric and magnetic field data as well as particle positions and momenta. While there are various PIC codes freely available, the task of post-processing – essentially condensing the large amounts of data into small units suitable for plotting routines – is typically left to each user individually. As post-processing may be a time consuming and error-prone process, this python package has been developed.

Postpic can handle two different types of data:

Field data which is data sampled on a predefined grid, such as electic and magnetic fields, particle- or charge densities, currents, etc. Fields are usually the data, which can be plotted directly. See `postpic.Field`.

Particle data which is data of multiple particles and for each particle positions (*x*, *y*, *z*) and momenta (*px*, *py*, *pz*) are known. Particles usually also have *weight*, *charge*, *time* and a unique *id*. Postpic can transform particle data to field data using the same algorithm and particle shapes, which are used in most PIC Simulations. The particle-to-grid routines are written in C for maximum performance. See `postpic.MultiSpecies`.

class postpic.**Field**(*matrix*, *name=''*, *unit=''*, ***kwargs*)

Bases: postpic._compat.mixins.NDArrayOperatorsMixin

The Field Object carries data in form of an *numpy.ndarray* together with as many Axis objects as the data's dimensions. Additionaly the Field object provides any information that is necessary to plot _and_ annotate the plot.

Create a Field object from scratch. The only required argument is *matrix* which contains the actual data.

A *name* and a *unit* may be supplied.

The axis may be specified in different ways:

- by passing a list of Axis object as *axes*
- by passing arrays with the grid_nodes as *xedges*, *yedges* and *zedges*. This is intended to work with *np.histogram*.
- by not passing anything, which will create default axes from 0 to 1.

T
Return the Field with the order of axes reversed. In 2D this is the usual matrix transpose operation.

adjust_stagger_to(*other*)

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

Refer to *numpy.all* for full documentation.

See also:

numpy.all() equivalent function

angle

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

atleast_nd(*n*)
Make sure the field has at least 'n' dimensions

autocutout(*axes=None*, *fractions=(0.001*, *0.002)*)
Automatically cuts out the main feature of the field by removing border regions that only contain small numbers.

This is done axis by axis. For each axis, the mean across all other axes is taken. The maximum *max* of the remaining 1d-*array* is taken and searched for the outermost boundaries a, d such that all values out of array[a:d] are smaller then fractions[0]*max. A second set of boundaries b, c is searched such that all values out of array[b:c] are smaller then fractions[1]*max. Because fractions[1] should be larger than fractions[0], array[b:c] should be contained completely in array[a:d].

A padding length x is chosen such that array[b-x:c+x] is entirely within array[a:d].

Then the corresponding axis of the field is sliced to [b-x:c+x] and multiplied with a tukey-window such that the region [b:c] is left untouched and the field in the padding region smoothly vanishes on the outer border.

This process is repeated for all axes in *axes* or for all axes if *axes* is None.

autoreduce(*maxlen=4000*)
Reduces the Grid to a maximum length of maxlen per dimension by just executing half_resolution as often as necessary.

clip(*a_min*, *a_max*, *out=None*)

conj()

cutout (*newextent*)
only keeps that part of the data, that belongs to newextent.

dimensions
returns only present dimensions. [] and [[]] are interpreted as -1 np.array(2) is interpreted as 0 np.array([1,2,3]) is interpreted as 1 and so on...

ensure_frequency_domain()

ensure_spatial_domain()

ensure_transform_state (*transform_states*)
Makes sure that the field has the given transform_states. *transform_states* may be a single boolean, indicating the same desired transform_state for all axes. It may be a list of the desired transform states for all the axes or a dictionary indicating the desired transform states of specific axes.

export (*filename*, ***kwargs*)

Uses *postpic.export_field* to export this field to a file. All ‘‘**kwargs‘ will be forwarded to this function. Format is recognized by the extension of the filename.

export Field object as a file. Format depends on the extention of the filename. Currently supported are: .npz:

uses *numpy.savez*.

.csv: uses *numpy.savetxt*.

.vtk: vtk export to paraview

extent
returns the extents in a linearized form, as required by “matplotlib.pyplot.imshow”.

fft (*axes=None*, *exponential_signs='spatial'*, ***kwargs*)
Performs Fourier transform on any number of axes.

The argument axis is either an integer indicating the axis to be transformed or a tuple giving the axes that should be transformed. Automatically determines forward/inverse transform. Transform is only applied if all mentioned axes are in the same transform state. If an axis is transformed twice, the origin of the axis is restored.

Parameters

- **exponential_signs** – configures the sign convention of the exponential.
  - exponential_signs == ‘spatial’: fft using exp(-ikx), ifft using exp(ikx)
  - exponential_signs == ‘temporal’: fft using exp(iwt), ifft using exp(-iwt)
- ****kwargs** – keyword-arguments are passed to the underlying fft implementation.

fft_autopad (*axes=None*, *fft_padsize=<postpic.helper.FFTW_Pad object>*)
Automatically pad the array to a size such that computing its FFT using FFTW will be fast.

Parameters **fft_padsize** (*callable*) – The default for keyword argument *fft_padsize* is a callable, that is used to calculate the padded size for a given size.

By default, this uses *fft_padsize=helper.fftw_padsize* which finds the next larger “good” grid size according to what the FFTW documentation says.

However, the FFTW documentation also says: “(...) Transforms whose sizes are powers of 2 are especially fast.”

If you don't worry about the extra padding, you can pass *fft_padsize=helper.fft_padsize_power2* and this method will pad to the next power of 2.

grid

grid_nodes

half_resolution(*axis*)
Halfs the resolution along the given axis by removing every second *grid_node* and averaging every second data point into one.

If there is an odd number of grid points, the last point will be ignored (that means, the extent will change by the size of the last grid cell).

Returns the modified *Field*.

Return type *Field*

imag

integrate(*axes=None*, *method=<function simps>*)
Calculates the definite integral along the given axes.

Parameters **method** (*callable*) – Choose the method to use. Available options:

- 'constant'
- any function with the same signature as scipy.integrate.simps (default).

islinear()

label

classmethod loadfrom(*filename*)
construct a new field object from file. currently, the following file formats are supported: *.npz

map_axis_grid(*axis*, *transform*, *preserve_integral=True*, *jacobian_func=None*)
Transform the Field to new coordinates along one axis.

This function transforms the coordinates of one axis according to the function *transform* and applies the jacobian to the data.

Please note that no interpolation is applied to the data, instead a non-linear axis grid is produced. If you want to interpolate the data to a new (linear) grid, use the method *map_coordinates()* instead.

In contrast to *map_coordinates()*, the function transform is not used to pull the new data points from the old grid, but is directly applied to the axis. This reverses the direction of the transform. Therfore, in order to preserve the integral, it is necessary to divide by the Jacobian.

Parameters

- **axis** (*int*) – the index or name of the axis you want to apply transform to.
- **transform** (*callable*) – the transformation function which takes the old coordinates as an input and returns the new grid
- **preserve_integral** (*bool*) – Divide by the jacobian of transform, in order to preserve the integral.
- **jacobian_func** (*callable*) – If given, this is expected to return the derivative of transform. If not given, the derivative is numerically approximated.

map_coordinates(*newaxes*, *transform=None*, *complex_mode='polar'*, *preserve_integral=True*, *jacobian_func=None*, *jacobian_determinant_func=None*, ***kwargs*)
Transform the Field to new coordinates.

Parameters

- **newaxes** (*list*) – The new axes of the new coordinates.
- **transform** (*callable*) – a callable that takes the new coordinates as input and returns the old coordinates from where to sample the Field. It is basically the inverse of the transformation that you want to perform. If transform is not given, the identity will be used. This is suitable for simple interpolation to a new extent/shape. Example for cartesian -> polar:

```
>>> def T(r, theta):
>>>     x = r * np.cos(theta)
>>>     y = r * np.sin(theta)
>>>     return x, y
```

  Note that this function actually computes the cartesian coordinates from the polar coordinates, but stands for transforming a field in cartesian coordinates into a field in polar coordinates.

  However, in order to preserve the definite integral of the field, it is necessary to multiply with the Jacobian determinant of T.

$$\tilde{U}(r,\theta) = U(T(r,\theta)) \cdot \det \frac{\partial(x,y)}{\partial(r,\theta)}$$

  such that

$$\int_V \mathrm{d}x\,\mathrm{d}y\,U(x,y) = \int_{T^{-1}(V)} \mathrm{d}r\,\mathrm{d}\theta\,\tilde{U}(r,\theta)\,.$$

- **complex_mode** – The complex_mode specifies how to proceed with complex data.
  - complex_mode = 'cartesian' - interpolate real/imag part (fastest)
  - complex_mode = 'polar' - interpolate abs/phase If skimage.restoration is available, the phase will be unwrapped first (default)
  - complex_mode = 'polar-no-unwrap' - interpolate abs/phase Skip unwrapping the phase, even if skimage.restoration is available
- **preserve_integral** (*bool*) – If True (the default), the data will be multiplied with the Jacobian determinant of the coordinate transformation such that the integral over the data will be preserved.

  In general, you will want to do this, because the physical unit of the new Field will correspond to the new axis of the Fields. Please note that Postpic, currently, does not automatically change the unit members of the Axis and Field objects, this you will have to do manually.

  There are, however, exceptions to this rule. Most prominently, if you are converting to polar coordinates, it depends on what you are going to do with the transformed Field. If you intend to do a Cartesian r-theta plot or are interested in a lineout for a single value of theta, you do want to apply the Jacobian determinant. If you had a density in e.g. J/m^2 than, in polar coordinates, you want to have a density in J/m/rad. If you intend, on the other hand, to do a polar plot, you do not want to apply the Jacobian. In a polar plot, the data points are plotted with variable density which visually takes care of the Jacobian automatically. A polar plot of the polar data should look like a Cartesian plot of the original data with just a peculiar coordinate grid drawn over it.
- **jacobian_determinant_func** (*callable*) – A callable that returns the jacobian determinant of the transform. If given, this takes precedence over the following option.

- **jacobian_func** (*callable*) – a callable that returns the jacobian of the transform. If this is not given, the jacobian is numerically approximated.
- ****kwargs** – Additional keyword arguments are passed to *scipy.ndimage.map_coordinates*, see the documentation of that function.

matrix

max(*axis=None*, *out=None*)
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*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See also:

numpy.mean() equivalent function

meshgrid(*sparse=True*)

min(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

See also:

numpy.amin() equivalent function

pad(*pad_width*, *mode='constant'*, ***kwargs*)
Pads the data using *np.pad* and takes care of the axes. See documentation of *numpy.pad*.

In contrast to *np.pad*, *pad_width* may be given as integers, which will be interpreted as pixels, or as floats, which will be interpreted as distance along the appropriate axis.

All other parameters are passed to *np.pad* unchanged.

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

Refer to *numpy.ptp* for full documentation.

See also:

numpy.ptp() equivalent function

real

replace_data(*other*)

saveto(*filename*)
Save a Field object as a file. Use *loadfrom()* to load Field objects.

setaxisobj(*axis*, *axisobj*)
replaces the current axisobject for axis axis by the new axisobj axisobj.

shape

shift_grid_by(*dx*, *interpolation='fourier'*)
Translate the Grid by *dx*. This is useful to remove the grid stagger of field components.

If all axis will be shifted, *dx* may be a list. Otherwise dx should be a mapping from axis to translation distance.

The keyword-argument interpolation indicates the method to be used and may be one of *['linear', 'fourier']*. In case of interpolation = 'fourier' all axes must have same transform_state.

spacing
returns the grid spacings for all axis.

squeeze()
removes axes that have length 1, reducing self.dimensions.

Note, that axis with length 0 will not be removed! *numpy.squeeze* also does not remove length=0 directions.

Same as *numpy.squeeze*.

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See also:

numpy.std() equivalent function

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
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*)
Swaps the axes *axis1* and *axis2*, equivalent to *numpy.swapaxes*.

topolar(*extent=None*, *shape=None*, *angleoffset=0*, ***kwargs*)
Transform the Field to polar coordinates.

This is a convenience wrapper for `map_coordinates()` which will let you easily define the desired grid in polar coordinates.

Parameters

- **extent** – should be of the form *extent=(phimin, phimax, rmin, rmax)* or *extent=(phimin, phimax)*
- **shape** – should be of the form *shape=(N_phi, N_r)*,

- **angleoffset** – can be any real number and will rotate the zero-point of the angular axis.
- **complex_mode** – The complex_mode specifies how to proceed with complex data.
  - complex_mode = 'cartesian' - interpolate real/imag part (fastest)
  - complex_mode = 'polar' - interpolate abs/phase If skimage.restoration is available, the phase will be unwrapped first (default)
  - complex_mode = 'polar-no-unwrap' - interpolate abs/phase Skip unwrapping the phase, even if skimage.restoration is available
- **preserve_integral** (*bool*) – If True (the default), the data will be multiplied with the Jacobian determinant of the coordinate transformation such that the integral over the data will be preserved.

  In general, you will want to do this, because the physical unit of the new Field will correspond to the new axis of the Fields. Please note that Postpic, currently, does not automatically change the unit members of the Axis and Field objects, this you will have to do manually.

  There are, however, exceptions to this rule. Most prominently, if you are converting to polar coordinates, it depends on what you are going to do with the transformed Field. If you intend to do a Cartesian r-theta plot or are interested in a lineout for a single value of theta, you do want to apply the Jacobian determinant. If you had a density in e.g. J/m^2 than, in polar coordinates, you want to have a density in J/m/rad. If you intend, on the other hand, to do a polar plot, you do not want to apply the Jacobian. In a polar plot, the data points are plotted with variable density which visually takes care of the Jacobian automatically. A polar plot of the polar data should look like a Cartesian plot of the original data with just a peculiar coordinate grid drawn over it.
- **jacobian_determinant_func** (*callable*) – A callable that returns the jacobian determinant of the transform. If given, this takes precedence over the following option.
- **jacobian_func** (*callable*) – a callable that returns the jacobian of the transform. If this is not given, the jacobian is numerically approximated.
- ****kwargs** – Additional keyword arguments are passed to *scipy.ndimage.map_coordinates*, see the documentation of that function.

transpose(**axes*)
transpose method equivalent to *numpy.ndarray.transpose*. If *axes* is empty, the order of the axes will be reversed. Otherwise axes[i] == j means that the i'th axis of the returned Field will be the j'th axis of the input Field.

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

Refer to *numpy.var* for full documentation.

See also:

numpy.var() equivalent function

class postpic.**Axis**(*name=''*, *unit=''*, ***kwargs*)
Bases: object

Axis handling for a single Axis.

Create an Axis object from scratch.

The least required arguments are any of:

- grid
- grid_node
- extent _and_ n

The remaining fields will be deduced from the givens.

More arguments may be supplied, as long as they are compatible.

extent

grid

grid_node

half_resolution()
removes every second grid_node.

islinear(*force=False*)
Checks if the axis has a linear grid.

label

physical_length

spacing

value_to_index(*value*)

class postpic.**PhysicalConstants**
Bases: object

gives you some constants.

c = 299792458.0

epsilon0 = 8.854187817620389e-12

mass_u = 1.67266490646e-27

me = 9.109383e-31

mu0 = 1.2566370614359173e-06

static ncrit(*laslambda*)
Critical plasma density in particles per m^3 for a given wavelength laslambda in m.

static ncrit_um(*lambda_um*)
Critical plasma density in particles per m^3 for a given wavelength lambda_um in microns.

qe = 1.602176565e-19

postpic.**unstagger_fields**(**fields*, ***kwargs*)
Unstagger a collection of fields.

This functions shifts the origins of the grids of the given fields such that they coincide. Since the choice of the common origin is somewhat arbitrary, it might be overriden by a keyword-argument *origin*, as may be the interpolation *method*. See *Field.shift_grid_by* for available methods.

postpic.**kspace_epoch_like**(*component*, *fields*, *dt*, *extent=None*, *omega_func=<function omega_free>*, *align_to='B'*)
Reconstruct the physical kspace of one polarization component See documentation of kspace

This function will use special care to make sure, that the implicit linear interpolation introduced by Epochs half-steps will not impede the accuracy of the reconstructed k-space. The frequency response of the linear interpolation is modelled and removed from the interpolated fields.

dt: time-step of the simulation, this is used to calculate the frequency response due to the linear interpolated half-steps

For the current version of EPOCH, v4.9, use the following: align_to == 'B' for intermediate dumps, align_to == "E" for final dumps

postpic.**kspace**(*component*, *fields*, *extent=None*, *interpolation=None*, *omega_func=<function omega_free>*)

Reconstruct the physical kspace of one polarization component This function basically computes one component of

E = 0.5*(E - omega/k^2 * Cross[k, E])

or B = 0.5*(B + 1/omega * Cross[k, B]).

component must be one of ["Ex", "Ey", "Ez", "Bx", "By", "Bz"].

The necessary fields must be given in the dict fields with keys chosen from ["Ex", "Ey", "Ez", "Bx", "By", "Bz"]. Which are needed depends on the chosen component and the dimensionality of the fields. In 3D the following fields are necessary:

Ex, By, Bz -> Ex Ey, Bx, Bz -> Ey Ez, Bx, By -> Ez

Bx, Ey, Ez -> Bx By, Ex, Ez -> By Bz, Ex, Ey -> Bz

In 2D, components which have "k_z" in front of them (see cross-product in equations above) are not needed. In 1D, components which have "k_y" or "k_z" in front of them (see cross-product in equations above) are not needed.

The keyword-argument extent may be a list of values [xmin, xmax, ymin, ymax, ...] which denote a region of the Fields on which to execute the kspace reconstruction.

The keyword-argument interpolation indicates whether interpolation should be used to remove the grid stagger. If interpolation is None, this function works only for non-staggered grids. Other choices for interpolation are "linear" and "fourier".

The keyword-argument omega_func may be used to pass a function that will calculate the dispersion relation of the simulation may be given. The function will receive one argument that contains the k mesh.

postpic.**kspace_propagate**(*kspace*, *dt*, *nsteps=1*, ***kwargs*)

Evolve time on a field. This function checks the transform_state of the field and transforms first from spatial domain to frequency domain if necessary. In this case the inverse transform will also be applied to the result before returning it. This works, however, only correctly with fields that are the inverse transforms of a k-space reconstruction, i.e. with complex fields.

dt: time in seconds

This function will return an infinite generator that will do arbitrary many time steps.

If yield_zeroth_step is True, then the kspace will also be yielded after removing the antipropagating waves, but before the first actual step is done.

If a vector moving_window_vect is passed to this function, which is ideally identical to the mean propagation direction of the field in forward time direction, an additional linear phase is applied in order to keep the pulse inside of the box. This effectively enables propagation in a moving window. If dt is negative, the window will actually move the opposite direction of moving_window_vect. Additionally, all modes which propagate in the opposite direction of the moving window, i.e. all modes for which dot(moving_window_vect, k)<0, will be deleted.

The motion of the window can be inhibited by specifying move_window=False. If move_window is None, the moving window is automatically enabled if moving_window_vect is given.

The deletion of the antipropagating modes can be inhibited by specifying remove_antipropagating_waves=False. If remove_antipropagating_waves is None, the deletion of the antipropagating modes is automatically enabled if moving_window_vect is given.

nsteps: number of steps to take

If nsteps == 1, this function will just return the result. If nsteps > 1, this function will return a generator that will generate the results. If you want a list, just put list(...) around the return value.

postpic.**time_profile_at_plane**(*kspace_or_complex_field*, *axis='x'*, *value=None*, *dir=1*, ***kwargs*)

'Measure' the time-profile of the propagating *complex_field* while passing through a plane.

The arguments *axis*, *value* and *dir* specify the plane and main propagation direction.

axis specifies the axis perpendicular to the measurement plane.

dir=1 specifies propagation towards positive *axis*, *dir=-1* specifies the opposite direction of propagation.

value specifies the position of the plane along *axis*. If *value=None,* a default is chosen, depending on *dir*.

If *dir=-1*, the starting point of the axis is used, which lies at the 0-component of the inverse transform.

If *dir=1*, the end point of the axis + one axis spacing is used, which, via periodic boundary conditions of the fft, also lies at the 0-component of the inverse transform.

If the given *value* differs from these defaults, an initial propagation with moving window will be performed, such that the desired plane lies in the default position.

For example *axis='x'* and *value=0.0* specifies the 'x=0.0' plane while *dir=1* specifies propagation towards positive 'x' values. The 'x' axis starts at 2e-5 and ends at 6e-5 with a grid spacing of 1e-6. The default value for the measurement plane would have been 6.1e-5 so an initial backward propagation with dt = -6.1e-5/c is performed to move the pulse in front of the'x=0.0 plane.

Additional *kwargs* are passed to kspace_propagate if they are not overridden by this function.

class postpic.**ScalarProperty**(*expr*, *name=None*, *unit=None*, *symbol=None*)

Bases: object

evaluate(*vars*)

vars must be a dictionary containing variables used within the expression "expr".

expr

input_names

The list of variables used within this expression.

name

symbol

unit

class postpic.**MultiSpecies**(*dumpreader*, **speciess*, ***kwargs*)

Bases: object

The MultiSpecies class. Different MultiSpecies can be added together to create a combined collection.

ignore_missing_species = False set to true to ignore missing species.

The MultiSpecies class will return a list of values for every particle property.

Ekin()

Deprecated since version unknown: The function Ekin is deprecated. Use self("Ekin") instead.

Ekin_MeV()
Deprecated since version unknown: The function Ekin_MeV is deprecated. Use self("Ekin_MeV") instead.

Ekin_MeV_amu()
Deprecated since version unknown: The function Ekin_MeV_amu is deprecated. Use self("Ekin_MeV_amu") instead.

Ekin_MeV_qm()
Deprecated since version unknown: The function Ekin_MeV_qm is deprecated. Use self("Ekin_MeV_qm") instead.

Ekin_keV()
Deprecated since version unknown: The function Ekin_keV is deprecated. Use self("Ekin_keV") instead.

Ekin_keV_amu()
Deprecated since version unknown: The function Ekin_keV_amu is deprecated. Use self("Ekin_keV_amu") instead.

Ekin_keV_qm()
Deprecated since version unknown: The function Ekin_keV_qm is deprecated. Use self("Ekin_keV_qm") instead.

Eruhe()
Deprecated since version unknown: The function Eruhe is deprecated. Use self("Eruhe") instead.

ID()
Deprecated since version unknown: The function ID is deprecated. Use self("id") instead.

P()
Deprecated since version unknown: The function P is deprecated. Use self("p") instead.

Px()
Deprecated since version unknown: The function Px is deprecated. Use self("px") instead.

Py()
Deprecated since version unknown: The function Py is deprecated. Use self("py") instead.

Pz()
Deprecated since version unknown: The function Pz is deprecated. Use self("pz") instead.

V()
Deprecated since version unknown: The function V is deprecated. Use self("v") instead.

Vx()
Deprecated since version unknown: The function Vx is deprecated. Use self("vx") instead.

Vy()
Deprecated since version unknown: The function Vy is deprecated. Use self("vy") instead.

Vz()
Deprecated since version unknown: The function Vz is deprecated. Use self("vz") instead.

X()
Deprecated since version unknown: The function X is deprecated. Use self("x") instead.

X_um()
Deprecated since version unknown: The function X_um is deprecated. Use self("x_um") instead.

Y()
Deprecated since version unknown: The function Y is deprecated. Use self("y") instead.

Y_um()
Deprecated since version unknown: The function Y_um is deprecated. Use self("Y_mu") instead.

Z()
Deprecated since version unknown: The function Z is deprecated. Use self("z") instead.

Z_um()
Deprecated since version unknown: The function Z_um is deprecated. Use self("z_um") instead.

add(*dumpreader*, *species*, *ignore_missing_species=False*)
adds a species to this MultiSpecies. This function modifies the current Object and always returns None.

species can be a single species name
or a reserved name for collection of species, such as ions adds all available particles that are ions nonions adds all available particles that are not ions ejected noejected all

Optional arguments

----

ignore_missing_species = False
set to True to ignore if the species is missing.

angle_xaxis()
Deprecated since version unknown: The function angle_xaxis is deprecated. Use self("angle_xaxis") instead.

angle_xy()
Deprecated since version unknown: The function angle_xy is deprecated. Use self("angle_xy") instead.

angle_xz()
Deprecated since version unknown: The function angle_xz is deprecated. Use self("angle_xz") instead.

angle_yx()
Deprecated since version unknown: The function angle_yx is deprecated. Use self("angle_yx") instead.

angle_yz()
Deprecated since version unknown: The function angle_yz is deprecated. Use self("angle_yz") instead.

angle_zx()
Deprecated since version unknown: The function angle_zx is deprecated. Use self("angle_zx") instead.

angle_zy()
Deprecated since version unknown: The function angle_zy is deprecated. Use self("angle_zy") instead.

beta()
Deprecated since version unknown: The function beta is deprecated. Use self("beta") instead.

betax()
Deprecated since version unknown: The function betax is deprecated. Use self("betax") instead.

betay()
Deprecated since version unknown: The function betay is deprecated. Use self("betay") instead.

betaz()
Deprecated since version unknown: The function betaz is deprecated. Use self("betaz") instead.

charge()
Deprecated since version unknown: The function charge is deprecated. Use self("charge") instead.

charge_e()
Deprecated since version unknown: The function charge_e is deprecated. Use self("charge_e") instead.

compress(*condition*, *name='unknown condition'*)
works like numpy.compress. Returns a new MultiSpecies instance.

Additionaly you can specify a name, that gets saved in the compresslog.

condition has to be one out of: 1) condition = [True, False, True, True, ... , True, False] condition is a list of length N, specifing which particles to keep. Example: cfintospectrometer = lambda x: x.angle_offaxis() < 30e-3 cfintospectrometer.name = '< 30mrad offaxis' pa.compress(cfintospectrometer(pa), name=cfintospectrometer.name) 2) condition = [1, 2, 4, 5, 9, ... , 805, 809] condition can be a list of arbitraty length, so only the particles with the ids listed here are kept.

name – name the condition. This can later be reviewed by calling 'self.compresslog()'

compressfn(*conditionf*, *name='unknown condition'*)
like "compress", but accepts a function.

Returns a new MultiSpecies instance.

name – name the condition.

createField(**sps*, ***kwargs*)
Creates an n-d Histogram enclosed in a Field object.

Parameters

- ***sps** – list of scalarfunctions/strings/scalar-properties, that will be evaluated to data for each axis. the number of args given determins the dimensionality of the field returned by this function (maximum 3)
- **name** (*string, optional*) – addes a name. usually used for generating savenames. Defaults to "distfn".
- **title** (*string, options*) – overrides the title. Autocreated if title==None. Defaults to None.
- **rangex** (*list of two values, optional*) – the xrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.
- **rangey** (*list of two values, optional*) – the yrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.
- **rangez** (*list of two values, optional*) – the zrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.

dumpreader
returns the dumpreader if the dumpreader of **all** species are pointing to the same dump. This should be mostly the case.

Otherwise returns None.

filter(*condition*, *name=None*)
like compress, but takes a ScalarProperty or a str, which are required to evaluate to a boolean list to filter particles. This is the preferred method to filter particles by a value of their property.

Returns a new MultiSpecies instance.

gamma()
Deprecated since version unknown: The function gamma is deprecated. Use self("gamma") instead.

gamma_m1()
Deprecated since version unknown: The function gamma_m1 is deprecated. Use self("gamma_m1") instead.

getcompresslog()

initial_npart
Original number of particles (before the use of compression or filter).

mass()
Deprecated since version unknown: The function mass is deprecated. Use self("mass") instead.

mass_u()
Deprecated since version unknown: The function mass_u is deprecated. Use self("mass_u") instead.

mean(*expr*, *weights='1'*)
the mean of a value given by the function func. The particle weight of the individual particles will be included in the calculation. An additional weight can be given as well.

median(*expr*, *weights='1'*)
The median

name
an alias to self.species

npart
Number of Particles.

nspecies
Number of species.

quantile(*expr*, *q*, *weights='1'*)
The qth-quantile of the distribution.

r_xy()
Deprecated since version unknown: The function r_xy is deprecated. Use self("r_xy") instead.

r_xyz()
Deprecated since version unknown: The function r_xyz is deprecated. Use self("r_xyz") instead.

r_yz()
Deprecated since version unknown: The function r_yz is deprecated. Use self("r_yz") instead.

r_zx()
Deprecated since version unknown: The function r_zx is deprecated. Use self("r_zx") instead.

simextent(*axis*)
the combined simextent for all species and dumps included in this MultiSpecies object.

simgridpoints(*axis*)
this function is for convenience only and is likely to be removed in the future. Particlarly it is impossible to define the grid of the simulation if the MultiSpecies object consists of multiple dumps from different simulations.

species
returns an string name for the species involved. Basically only returns unique names from all species (used for plotting and labeling purposes – not for completeness). May be overwritten.

speciess
a complete list of all species involved.

time()
Deprecated since version unknown: The function time is deprecated. Use self("time") instead.

uncompress()
Returns a new MultiSpecies instance, with all previous calls of "compress" or "filter" undone.

var(*expr*, *weights='1'*)
variance

weight()
Deprecated since version unknown: The function weight is deprecated. Use self("weight") instead.

class postpic.**ParticleHistory**(*sr*, *speciess*, *ids=None*)
Bases: object

Represents a list of particles including their history that can be found in all the dumps defined by the simulation reader sr.

Parameters

- **sr** (*iterable of datareader*) – a collection of datareader to use. Usually a Simulationreader object
- **speciess** (*string or iterable of strings*) – a species name or a list of species names. Those particles can be included into the history.
- **ids** (*iterable of int*) – list of ids to use (default: None). If this is None all particles in speciess will be tracked. If a list of ids is given, these ids will be serached in speciess only.

collect(**scalarfs*)
Collects the given particle properties for all particles for all times.

*scalarfs: the scalarfunction(s) defining the particle property

numpy.ndarray holding the different particles in the same order as the list of *self.ids*, meaning the particle on position *particle_idx* has the ID *self.ids[particle_idx]*. every array element holds the history for a single particle. Indexorder of returned array: [particle_idx][scalarf_idx, collection_idx]

skip(*n*)
takes only everth (n+1)-th particle

postpic.**histogramdd**(*data*, ***kwargs_in*)
Creates a histogram of the data. This function has the similar signature and return values as *numpy.histogramdd*. In addition this function supports the *shape* keyword argument to choose the particle shape used. If used with *shape=0* the results of this function and the *numpy.histogramdd* are identical, however, this function is approx. factor 2 or 3 faster.

Parameters

- **data** (*sequence of ndarray or ndarray (1D or 2D)*) –

  The input (particle) data for the histogram.

  - A 1D numpy array (for 1D histogram).
  - A sequence providing the data for the different axis, i.e. *(datax, datay, dataz)* (preferred).
  - A (N, D)-array, i.e. *[[x1, y1, z1], [x2, y2, z2]]* – must be a numpy array!
- **bins** (*sequence or int*) – The number of bins to use for each dimension
- **range** (*sequence, optional*) – A sequence of lower and upper bin edges to be used if the edges are not given explicitly in bins. Defaults to the minimum and maximum values along each dimension.
- **weights** (*1D numpy array*) – The weights to be used for each data point
- **shape** (*int*) –

  possible choices are:

  - 0 - use nearest grid point (NGP)

- 1 - use tophat shape of width 1 bin
- 2 - triangular shape (default)
- 3 - spline 3 shape

Returns

- **H** (*ndarray*) – the final histogram
- **edges** (*list*) – A list of D arrays describing the edges for each dimension

class postpic.**SpeciesIdentifier**

Bases: *postpic.helper.PhysicalConstants*

This Class provides static methods for deriving particle properties from species Names. The only reason for this to be a class is that it can be used as a mixin.

classmethod identifyspecies(*species*)

Returns a dictionary containing particle informations deduced from the species name. The following keys in the dictionary will always be present: name species name string mass kg (SI) charge C (SI) tracer boolean ejected boolean

Valid Examples: Periodic Table symbol + charge state: c6, F2, H1, C6b ionm#c# defining mass and charge: ionm12c2, ionc20m110 advanced examples: ejected_tracer_ionc5m20b, ejected_tracer_electronx, ejected_c6b, tracer_proton, protonb

static isejected(*species*)

classmethod ision(*species*)

postpic.**chooseCode**(*code*)

Chooses appropriate reader for the given simulation code. After choosing a preset of the correct reader, the functions *postpic.readDump()* and *postpic.readSim()* are setup for this preset.

Parameters **code** (*string*) –

Possible options are:

- "DUMMY": dummy class creating fake data.
- "EPOCH": .sdf files written by EPOCH1D, EPOCH2D or EPOCH3D.
- "openPMD": .h5 files written in openPMD Standard
- "piconGPU": same as "openPMD"
- "VSIM": .hdf5 files written by VSim.

postpic.**readDump**(*dumpidentifier*, ***kwargs*)

After using the fucntion *postpic.chooseCode()*, this function should be the main function for reading a dump into postpic.

Parameters

- **dumpidentifier** (*str*) – Identifies the dump. For most dumpreaders this is a string poining to the file or folder. See what the specific reader of your format expects.
- ****kwargs** – will be forwarded to the dumpreader.

Returns the dumpreader for this specific data dump.

Return type Dumpreader

postpic.**readSim**(*simidentifier*, ***kwargs*)

After using the function *postpic.chooseCode()*, this function should be the main function for reading a

simulation into postpic. A simulation is equivalent to a series of dumps in a specific order (not neccessarily time order).

Parameters

- **simidentifier** (*str*) – Identifies the simulation. For EPOCH this should be a string pointing to a *.visit* file. Specifics depend on the current simreader class, as set by *chooseCode*.
- ****kwargs** – will be forwarded to the simreader.

Returns the Simulationreader

postpic.**export_field**(*filename*, *field*, ***kwargs*)

export Field object as a file. Format depends on the extention of the filename. Currently supported are: .npz:

uses *numpy.savez*.

.csv: uses *numpy.savetxt*.

.vtk: vtk export to paraview

postpic.**load_field**(*filename*)

construct a new field object from file. currently, the following file formats are supported: *.npz

postpic.**export_scalar_vtk**(*filename*, *scalarfield*, ***kwargs*)

exports one 2D or 3D scalar field object to a VTK file which is suitable for viewing in ParaView. It is assumed that all fields are defined on the same grid.

postpic.**export_scalars_vtk**(*filename*, **fields*, ***kwargs*)

exports a set of scalar fields to a VTK file suitable for viewing in ParaView. Up to four fields may be given

postpic.**export_vector_vtk**(*filename*, **fields*, ***kwargs*)

exports a vector field to a VTK file suitable for viewing in ParaView. Three 3D fields are expected, which will form the X, Y and Z component of the vector field. If less than tree fields are given, the missing components will be assumed to be zero.

5.1.1.1 Subpackages

postpic.datareader package

The Datareader package contains methods and interfaces to read data from any Simulation.

The basic concept consits of two different types of readers:

The Dumpreader

This has to be subclassed from Dumpreader_ifc and allows to read a single dump created by the simulation. To identify which dump should be read its initialized with a dumpidentifier. This dumpidentifier can be almost anything, but in the easiest case this is the filepath pointing to a single file containing every information about this simulation dump. With this information the dumpreader must be able to read all data regarding this dump (which is a lot: X, Y, Z, Px, Py, Py, weight, mass, charge, ID,.. for all particle species, electric and magnetic fields on grid, the grid itself, mabe particle ids,...)

The Simulationreader

This has to be subclassed from Simulationreader_ifc and allows to read a full list of simulation dumps. Thus an alternate Name for this class could be "Dumpsequence". This allows the code to track particles from different times of the simulation or create plots with a time axis.

Stephan Kuschel 2014

postpic.datareader.**chooseCode**(*code*)

Chooses appropriate reader for the given simulation code. After choosing a preset of the correct reader, the functions *postpic.readDump()* and *postpic.readSim()* are setup for this preset.

Parameters **code** (*string*) –

Possible options are:

- "DUMMY": dummy class creating fake data.
- "EPOCH": .sdf files written by EPOCH1D, EPOCH2D or EPOCH3D.
- "openPMD": .h5 files written in openPMD Standard
- "piconGPU": same as "openPMD"
- "VSIM": .hdf5 files written by VSim.

postpic.datareader.**readDump**(*dumpidentifier*, ***kwargs*)

After using the fucntion *postpic.chooseCode()*, this function should be the main function for reading a dump into postpic.

Parameters

- **dumpidentifier** (*str*) – Identifies the dump. For most dumpreaders this is a string poining to the file or folder. See what the specific reader of your format expects.
- ****kwargs** – will be forwarded to the dumpreader.

Returns the dumpreader for this specific data dump.

Return type Dumpreader

postpic.datareader.**readSim**(*simidentifier*, ***kwargs*)

After using the function *postpic.chooseCode()*, this function should be the main function for reading a simulation into postpic. A simulation is equivalent to a series of dumps in a specific order (not neccessarily time order).

Parameters

- **simidentifier** (*str*) – Identifies the simulation. For EPOCH this should be a string pointing to a *.visit* file. Specifics depend on the current simreader class, as set by *chooseCode*.
- ****kwargs** – will be forwarded to the simreader.

Returns the Simulationreader

postpic.datareader.**setdumpreadercls**(*dumpreadercls*)

Sets the class that is used for reading dumps on later calls of *postpic.readDump()*.

Parameters **dumpreadercls** (Dumpreader_ifc) –

Note: This should only be used, for testing. A set of presets is provided by *postpic.chooseCode()*.

postpic.datareader.**setsimreadercls**(*simreadercls*)
Sets the class that is used for reading a simulation on later calls of *postpic.readSim()*.

Parameters **simreadercls** (Simulationreader_ifc) –

Note: This should only be used, for testing. A set of presets is provided by *postpic.chooseCode()*.

class postpic.datareader.**Dumpreader_ifc**(*dumpidentifier*, *name=None*)
Bases: postpic._field_calc.FieldAnalyzer

Interface class for reading a single dump. A dump contains informations about the simulation at a single timestep (Usually E- and B-Fields on grid + particles).

Any Dumpreader_ifc implementation will always be initialized using a dumpidentifier. This dumpidentifier can be anything, that points to the data of a dump. In the easiest case this is just the filename of the dump holding all data of that timestep (for example .sdf file for EPOCH, .hdf5 for some other code).

The dumpreader should provide all necessary informations in a unified interface but at the same time it should not restrict the user to these properties of there dump only. The recommended implementation is shown here (EPOCH and VSim reader work like this): All (!) data, that is saved in a single dump should be accessible via the self.__getitem__(key) method. Together with the self.keys() method, this will ensure, that every dumpreader works as a dictionary and every dump attribute is accessible via this dictionary.

- Level 0:

__getitem__ and keys(self) are level 0 methods, meaning it must be possible to access everthing with those methods.

- Level 1:

provide direct data access by forwarding the requests to the corresponding Level 0 or Level 1 methods.

- Level 2:

provide user access to the data by forwarding the request to Level 1 or Level 2 methods, but NOT to Level 0 methods.

If some attribute wasnt dumped a KeyError must be thrown. This allows classes which are using the reader to just exit if a needed property wasnt dumped or to catch the KeyError and proceed by actively ignoring it.

It is highly recommended to also override the functions __str__ and gridpoints.

Parameters **dumpidentifier** – variable type whatever identifies the dump. It is recommended to use a String here pointing to a file.

data(*key*)
access to every raw data. needs to return numpy arrays corresponding to the “key”.

dataB(*component*, ***kwargs*)

dataE(*component*, ***kwargs*)

getSpecies(*species*, *attrib*)
This function gives access to any of the particle properties in ..helper.attribidentify This method can behave in the following ways: 1) Return a list of scalar properties for each particle of this species 2) Return a single float (i.e. *1.2*, NOT *[1.2]*) to show that

every particle of this species has the same scalar value to thisdimmax property assigned. This might be quite often used for charge or mass that are defined per species.

3. Raise a KeyError if the requested property or species wasn dumped.

gridkeyB(*component*, ***kwargs*)

gridkeyE(*component*, ***kwargs*)

gridnode(*key*, *axis*)
The grid nodes along "axis". Grid nodes include the beginning and the end of the grid. Example: If the grid has 20 grid points, it has 21 grid nodes or grid edges.

gridoffset(*key*, *axis*)
offset of the beginning of the first cell of the grid.

gridpoints(*key*, *axis*)
Number of grid points along "axis". It is highly recommended to override this method due to performance reasons.

gridspacing(*key*, *axis*)
size of one grid cell in the direction "axis".

keys()

Returns a list of keys, that can be used in __getitem__ to read any information from this dump.

listSpecies()

name

simdimensions()
the number of spatial dimensions the simulations was using. Must be 1, 2 or 3.

simextent(*axis*)
returns the extent of the actual simulation box. Override in your own reader class for better performance implementation.

simgridpoints(*axis*)

simgridspacing(*axis*)

time()

timestep()

class postpic.datareader.**Simulationreader_ifc**(*simidentifier*, *name=None*)
Bases: collections.abc.Sequence

Interface for reading the data of a full Simulation.

Any Simulationreader_ifc implementation will always be initialized using a simidentifier. This simidentifier can be anything, that points to the data of multiple dumps. In the easiest case this can be the .visit file.

The Simulationreader_ifc is subclass of collections.Sequence and will thus behave as a Sequence. The Objects in the Sequence are supposed to be subclassed from Dumpreader_ifc.

It is highly recommended to also override the __str__ function.

Parameters **simidentifier** – variable type something identifiying a series of dumps.

name

times()

Submodules

postpic.datareader.datareader module

class postpic.datareader.datareader.**Dumpreader_ifc**(*dumpidentifier*, *name=None*)
Bases: postpic._field_calc.FieldAnalyzer

Interface class for reading a single dump. A dump contains informations about the simulation at a single timestep (Usually E- and B-Fields on grid + particles).

Any Dumpreader_ifc implementation will always be initialized using a dumpidentifier. This dumpidentifier can be anything, that points to the data of a dump. In the easiest case this is just the filename of the dump holding all data of that timestep (for example .sdf file for EPOCH, .hdf5 for some other code).

The dumpreader should provide all necessary informations in a unified interface but at the same time it should not restrict the user to these properties of there dump only. The recommended implementation is shown here (EPOCH and VSim reader work like this): All (!) data, that is saved in a single dump should be accessible via the self.__getitem__(key) method. Together with the self.keys() method, this will ensure, that every dumpreader works as a dictionary and every dump attribute is accessible via this dictionary.

- Level 0:

__getitem__ and keys(self) are level 0 methods, meaning it must be possible to access everthing with those methods.

- Level 1:

provide direct data access by forwarding the requests to the corresponding Level 0 or Level 1 methods.

- Level 2:

provide user access to the data by forwarding the request to Level 1 or Level 2 methods, but NOT to Level 0 methods.

If some attribute wasnt dumped a KeyError must be thrown. This allows classes which are using the reader to just exit if a needed property wasnt dumped or to catch the KeyError and proceed by actively ignoring it.

It is highly recommended to also override the functions __str__ and gridpoints.

Parameters **dumpidentifier** – variable type whatever identifies the dump. It is recommended to use a String here pointing to a file.

data(*key*)
access to every raw data. needs to return numpy arrays corresponding to the "key".

dataB(*component*, ***kwargs*)

dataE(*component*, ***kwargs*)

getSpecies(*species*, *attrib*)
This function gives access to any of the particle properties in ..helper.attribidentify This method can behave in the following ways: 1) Return a list of scalar properties for each particle of this species 2) Return a single float (i.e. *1.2*, NOT *[1.2]*) to show that

every particle of this species has the same scalar value to thisdimmax property assigned. This might be quite often used for charge or mass that are defined per species.

3. Raise a KeyError if the requested property or species wasn dumped.

gridkeyB(*component*, ***kwargs*)

gridkeyE(*component*, ***kwargs*)

gridnode(*key*, *axis*)
The grid nodes along "axis". Grid nodes include the beginning and the end of the grid. Example: If the grid has 20 grid points, it has 21 grid nodes or grid edges.

gridoffset(*key*, *axis*)
offset of the beginning of the first cell of the grid.

gridpoints(*key*, *axis*)
Number of grid points along "axis". It is highly recommended to override this method due to performance reasons.

gridspacing(*key*, *axis*)
size of one grid cell in the direction "axis".

keys()

Returns a list of keys, that can be used in __getitem__ to read any information from this dump.

listSpecies()

name

simdimensions()
the number of spatial dimensions the simulations was using. Must be 1, 2 or 3.

simextent(*axis*)
returns the extent of the actual simulation box. Override in your own reader class for better performance implementation.

simgridpoints(*axis*)

simgridspacing(*axis*)

time()

timestep()

class postpic.datareader.datareader.**Simulationreader_ifc**(*simidentifier*, *name=None*)

Bases: collections.abc.Sequence

Interface for reading the data of a full Simulation.

Any Simulationreader_ifc implementation will always be initialized using a simidentifier. This simidentifier can be anything, that points to the data of multiple dumps. In the easiest case this can be the .visit file.

The Simulationreader_ifc is subclass of collections.Sequence and will thus behave as a Sequence. The Objects in the Sequence are supposed to be subclassed from Dumpreader_ifc.

It is highly recommended to also override the __str__ function.

Parameters **simidentifier** – variable type something identifiying a series of dumps.

name

times()

postpic.datareader.dummy module

Dummy reader for creating fake simulation Data for testing purposes.

Stephan Kuschel 2014

class postpic.datareader.dummy.**Dummyreader**(*dumpid*, *dimensions=2*, *randfunc=<built-in method normal of mtrand.RandomState object>*, *seed=0*, ***kwargs*)

Bases: *postpic.datareader.datareader.Dumpreader_ifc*

Dummyreader creates fake Data for testing purposes.

Parameters **dumpid** – int the dumpidentifier is the dumpid in this case. It is a float variable, that will also change the dummyreaders output (for example it will pretend to have dumpid many particles).

data(*axis*)

getSpecies(*species*, *attrib*)

grid(*key*, *axis*)

Parameters **axis** – string or int the axisidentifier

Returns: list of grid points of the axis specified.

Thus only regular grids are supported currently.

gridnode(*key*, *axis*)

Parameters **axis** – string or int the axisidentifier

Returns: list of grid points of the axis specified.

Thus only regular grids are supported currently.

gridoffset(*key*, *axis*)

gridspacing(*key*, *axis*)

keys()

listSpecies()

simdimensions()

simextent(*axis*)

simgridpoints(*axis*)

time()

timestep()

class postpic.datareader.dummy.**Dummysim**(*simidentifier*, *dimensions=2*, ***kwargs*)

Bases: *postpic.datareader.datareader.Simulationreader_ifc*

postpic.datareader.epochsdf module

Reader for SDF File format written by the EPOCH Code.

Dependecies:

- sdf: The actual python reader for the .sdf file format written in C. It is part of the EPOCH code base and needs to be compiled and installed from there.

Written by Stephan Kuschel 2014, 2015

class postpic.datareader.epochsdf.**Sdfreader**(*sdffile*, ***kwargs*)

Bases: *postpic.datareader.datareader.Dumpreader_ifc*

The Reader implementation for Data written by the EPOCH Code in .sdf format. Written for SDF v2.2.0 or higher. SDF can be obtained without EPOCH from SDF.

Parameters **sdffile** – String A String containing the relative Path to the .sdf file.

data(*key*)

getSpecies(*species*, *attrib*)
Returns one of the attributes out of (x,y,z,px,py,pz,weight,ID,mass,charge) of this particle species. raises KeyError if the requested species or property wasnt dumped.

getderived()
Returns all Keys starting with "Derived/".

gridoffset(*key*, *axis*)

gridpoints(*key*, *axis*)

gridspacing(*key*, *axis*)

keys()

listSpecies()

simdimensions()

simextent(*axis*)
Returns the extent of the actual simulation box.

simgridpoints(*axis*)
Returns the number of grid points of the actual simulation.

time()

timestep()

class postpic.datareader.epochsdf.**Visitreader**(*visitfile*, *dumpreadercls=<class 'postpic.datareader.epochsdf.Sdfreader'>*, ***kwargs*)
Bases: *postpic.datareader.datareader.Simulationreader_ifc*

Reads a series of dumps specified in a .visit file. This is specifically written for .visit files from the EPOCH code, but should also work for any other code using these files.

postpic.datareader.openPMDh5 module

Support for hdf5 files following the openPMD Standard.

Dependecies:

- h5py: read hdf5 files with python

Written by Stephan Kuschel 2016

class postpic.datareader.openPMDh5.**OpenPMDreader**(*h5file*, ***kwargs*)
Bases: *postpic.datareader.datareader.Dumpreader_ifc*

The Reader implementation for Data written in the hdf5 file format following openPMD naming conventions.

Parameters **h5file** – String A String containing the relative Path to the .h5 file.

data(*key*)
should work with any key, that contains data, thus on every hdf5.Dataset, but not on hdf5.Group. Will extract the data, convert it to SI and return it as a numpy array. Constant records will be detected and converted to a numpy array containing a single value only.

getSpecies(*species*, *attrib*)
Returns one of the attributes out of (x,y,z,px,py,pz,weight,ID,mass,charge) of this particle species.

getderived()
return all other fields dumped, except E and B.

gridoffset(*key*, *axis*)

gridpoints(*key*, *axis*)

gridspacing(*key*, *axis*)

keys()

listSpecies()

simdimensions()
the number of spatial dimensions the simulation was using.

time()

timestep()

class postpic.datareader.openPMDh5.**FileSeries**(*simidentifier*, *dumpreadercls=<class 'postpic.datareader.openPMDh5.OpenPMDreader'>*, ***kwargs*)
Bases: *postpic.datareader.datareader.Simulationreader_ifc*

Reads a time series of dumps from a given directory. The simidentifier is expanded using glob in order to find matching files.

postpic.datareader.vsimhdf5 module

Reader for HDF5 File format written by the VSim Code: http://www.txcorp.com/support/vsim-support-menu/vsim-documentation Dependecies: h5py The Python actual reader for hdf5 file format. Georg Wittig, Stephan Kuschel 2014

class postpic.datareader.vsimhdf5.**Hdf5reader**(*h5file*, ***kwargs*)
Bases: *postpic.datareader.datareader.Dumpreader_ifc*

The Reader implementation for HDF5 Data written by the VSim Code. as argument h5file can be any *.h5 file of the dump of consideration.

dataB(*axis*, ***kwargs*)

dataE(*axis*, ***kwargs*)

getSpecies(*species*, *attrib*)
Returns one of the attributes out of (x,y,z,px,py,pz,weight,ID) of this particle species. Valid Scalar attributes are (mass, charge).

getderived()
Returns all Keys starting with "Derived/".

grid(*axis*)
returns the array of the positions of all cells on axis = axis.

keys()

listSpecies()
returns all h5 dumps that have a attribute "mass"

simdimensions()

time()

timestep()

class postpic.datareader.vsimhdf5.**VSimReader**(*path*, ***kwargs*)
Bases: *postpic.datareader.datareader.Simulationreader_ifc*

Represents a full Simulation (= Series of Dumps with equal output). The VSimReader must be initialized with the path to a folder containing all the dumps. It will then walk through all available .h5 files in this directory to identify the available timesteps of the simulation.

getDumpreader(*index*)

postpic.io package

The postpic.io module provides free functions for importing and exporting data.

postpic.io.**export_field**(*filename*, *field*, ***kwargs*)
export Field object as a file. Format depends on the extention of the filename. Currently supported are: .npz:

uses *numpy.savez*.

.csv: uses *numpy.savetxt*.

.vtk: vtk export to paraview

postpic.io.**load_field**(*filename*)
construct a new field object from file. currently, the following file formats are supported: *.npz

postpic.io.**export_scalar_vtk**(*filename*, *scalarfield*, ***kwargs*)
exports one 2D or 3D scalar field object to a VTK file which is suitable for viewing in ParaView. It is assumed that all fields are defined on the same grid.

postpic.io.**export_scalars_vtk**(*filename*, **fields*, ***kwargs*)
exports a set of scalar fields to a VTK file suitable for viewing in ParaView. Up to four fields may be given

postpic.io.**export_vector_vtk**(*filename*, **fields*, ***kwargs*)
exports a vector field to a VTK file suitable for viewing in ParaView. Three 3D fields are expected, which will form the X, Y and Z component of the vector field. If less than tree fields are given, the missing components will be assumed to be zero.

Submodules

postpic.io.common module

The postpic.io module provides free functions for importing and exporting data.

postpic.io.csv module

The postpic.io module provides free functions for importing and exporting data.

postpic.io.npy module

The postpic.io module provides free functions for importing and exporting data.

postpic.io.vtk module

The postpic.io.vtk module provides classes and functions to export fields to the vtk 2.0 legacy file format.

The files are written in binary form and may have single or double precision.

The vtk exporter is built up from multiple classes, representing the different parts of data that are put into a vtk file:

VtkFile: represents the actual file to be written. Works as a context-manager that opens and initializes the file on __enter__ and closes the file on __exit__. Next to the actual file object *vtkfile.file*, it carries the attributes *vtkfile.type*, vtkfile.dtype‘ and *vtkfile.mode*, which are later used to make sure the file is written in the intended format.

VtkData: represents the data that should be written to a file. Has a “tofile” method that will create a VtkFile and pass it to the “tofile” methods of the other objects that carry the actual data. VtkData stores one object that is an instance of *DataSet* that defines the grid that the data lives on and one or more objects that are instances of *Data* that contain the data to be exported.

DataSet: Superclass for different types of grid. Subclasses are *StructuredPoints* and *RectilinearGrid*. They have classmethods *.from_field* which allow the creation of objects from a given *Field*.

Data: Superclass for representing either *PointData* or *CellData*. So far only *PointData is used. This will be used to contain a subclass of ‘ArrayData*.

ArrayData: Superclass for representing a collection of *Scalars* or *Vectors* that are stored in an array that will be created from one or more ‘Field‘s.

Using all of this, writing a vtk file with *Scalar* data attached to the points (*PointData*) of a *StructuredGrid* is as simple as:

```
VtkData(StructuredPoints.from_field(scalarfield),
        PointData(Scalars(scalarfield))
    ).tofile(filename)
```

class postpic.io.vtk.**ArrayData**(**fields*, ***kwargs*)
Bases: object

Superclass to represent different kinds of data that can be attributed to Points or Cells and are given as an iterable of Fields

tofile(*vtk*)

transform_data(*dtype*)

class postpic.io.vtk.**CellData**(*arraydata*)
Bases: *postpic.io.vtk.Data*

CellData associated with a *DataSet*

tofile(*vtk*)

class postpic.io.vtk.**Data**(*arraydata*)
Bases: object

Superclass to represent the attributed data associated with a *DataSet*.

tofile(*vtk*)

class postpic.io.vtk.**DataSet**
Bases: object

Superclass to represent different vtkDataSets

class postpic.io.vtk.**PointData**(*arraydata*)
Bases: *postpic.io.vtk.Data*

PointData associated with a *DataSet*

tofile(*vtk*)

class postpic.io.vtk.**RectilinearGrid**(*grid*)
Bases: *postpic.io.vtk.DataSet*

Class to represent a vtkRectilinearGrid

classmethod from_field(*field*)

tofile(*vtk*)

class postpic.io.vtk.**Scalars**(**fields*, ***kwargs*)
Bases: *postpic.io.vtk.ArrayData*

Class to represent a collection of Scalars

tofile(*vtk*)

class postpic.io.vtk.**StructuredPoints**(*dimensions*, *origin*, *spacing*)
Bases: *postpic.io.vtk.DataSet*

Class to represent a vtkStructuredPoints

classmethod from_field(*field*)

tofile(*vtk*)

class postpic.io.vtk.**Vectors**(**fields*, ***kwargs*)
Bases: *postpic.io.vtk.ArrayData*

Class to represent Vectors

tofile(*vtk*)

class postpic.io.vtk.**VtkData**(*dataset*, **data*)
Bases: object

Class to represent the data that should be written to a .vtk file. Uses *VtkFile*.

tofile(*fname*, *type='double'*, *mode='binary'*)

class postpic.io.vtk.**VtkFile**(*fname*, *type='double'*, *mode='binary'*)
Bases: object

Class used to write a .vtk file. Used by *VtkData*.

postpic.io.vtk.**export_scalar_vtk**(*filename*, *scalarfield*, ***kwargs*)
exports one 2D or 3D scalar field object to a VTK file which is suitable for viewing in ParaView. It is assumed that all fields are defined on the same grid.

postpic.io.vtk.**export_scalars_vtk**(*filename*, **fields*, ***kwargs*)
exports a set of scalar fields to a VTK file suitable for viewing in ParaView. Up to four fields may be given

postpic.io.vtk.**export_vector_vtk**(*filename*, **fields*, ***kwargs*)
exports a vector field to a VTK file suitable for viewing in ParaView. Three 3D fields are expected, which will form the X, Y and Z component of the vector field. If less than tree fields are given, the missing components will be assumed to be zero.

postpic.particles package

class postpic.particles.**ScalarProperty**(*expr*, *name=None*, *unit=None*, *symbol=None*)
Bases: object

evaluate(*vars*)
vars must be a dictionary containing variables used within the expression "expr".

expr

input_names
The list of variables used within this expression.

name

symbol

unit

class postpic.particles.**MultiSpecies**(*dumpreader*, **speciess*, ***kwargs*)
Bases: object

The MultiSpecies class. Different MultiSpecies can be added together to create a combined collection.

ignore_missing_species = False set to true to ignore missing species.

The MultiSpecies class will return a list of values for every particle property.

Ekin()
Deprecated since version unknown: The function Ekin is deprecated. Use self("Ekin") instead.

Ekin_MeV()
Deprecated since version unknown: The function Ekin_MeV is deprecated. Use self("Ekin_MeV") instead.

Ekin_MeV_amu()
Deprecated since version unknown: The function Ekin_MeV_amu is deprecated. Use self("Ekin_MeV_amu") instead.

Ekin_MeV_qm()
Deprecated since version unknown: The function Ekin_MeV_qm is deprecated. Use self("Ekin_MeV_qm") instead.

Ekin_keV()
Deprecated since version unknown: The function Ekin_keV is deprecated. Use self("Ekin_keV") instead.

Ekin_keV_amu()
Deprecated since version unknown: The function Ekin_keV_amu is deprecated. Use self("Ekin_keV_amu") instead.

Ekin_keV_qm()
Deprecated since version unknown: The function Ekin_keV_qm is deprecated. Use self("Ekin_keV_qm") instead.

Eruhe()
Deprecated since version unknown: The function Eruhe is deprecated. Use self("Eruhe") instead.

ID()
Deprecated since version unknown: The function ID is deprecated. Use self("id") instead.

P()
Deprecated since version unknown: The function P is deprecated. Use self("p") instead.

Px()
Deprecated since version unknown: The function Px is deprecated. Use self("px") instead.

Py()
Deprecated since version unknown: The function Py is deprecated. Use self("py") instead.

Pz()
Deprecated since version unknown: The function Pz is deprecated. Use self("pz") instead.

V()
Deprecated since version unknown: The function V is deprecated. Use self("v") instead.

Vx()
Deprecated since version unknown: The function Vx is deprecated. Use self("vx") instead.

Vy()
Deprecated since version unknown: The function Vy is deprecated. Use self("vy") instead.

Vz()
Deprecated since version unknown: The function Vz is deprecated. Use self("vz") instead.

X()
Deprecated since version unknown: The function X is deprecated. Use self("x") instead.

X_um()
Deprecated since version unknown: The function X_um is deprecated. Use self("x_um") instead.

Y()
Deprecated since version unknown: The function Y is deprecated. Use self("y") instead.

Y_um()
Deprecated since version unknown: The function Y_um is deprecated. Use self("Y_mu") instead.

Z()
Deprecated since version unknown: The function Z is deprecated. Use self("z") instead.

Z_um()
Deprecated since version unknown: The function Z_um is deprecated. Use self("z_um") instead.

add(*dumpreader*, *species*, *ignore_missing_species=False*)
adds a species to this MultiSpecies. This function modifies the current Object and always returns None.

species can be a single species name
or a reserved name for collection of species, such as ions adds all available particles that are ions nonions adds all available particles that are not ions ejected noejected all

Optional arguments

----

ignore_missing_species = False
set to True to ignore if the species is missing.

angle_xaxis()
Deprecated since version unknown: The function angle_xaxis is deprecated. Use self("angle_xaxis") instead.

angle_xy()
Deprecated since version unknown: The function angle_xy is deprecated. Use self("angle_xy") instead.

angle_xz()
Deprecated since version unknown: The function angle_xz is deprecated. Use self("angle_xz") instead.

angle_yx()
Deprecated since version unknown: The function angle_yx is deprecated. Use self("angle_yx") instead.

angle_yz()
Deprecated since version unknown: The function angle_yz is deprecated. Use self("angle_yz") instead.

angle_zx()
Deprecated since version unknown: The function angle_zx is deprecated. Use self("angle_zx") instead.

angle_zy()
Deprecated since version unknown: The function angle_zy is deprecated. Use self("angle_zy") instead.

beta()
Deprecated since version unknown: The function beta is deprecated. Use self("beta") instead.

betax()
Deprecated since version unknown: The function betax is deprecated. Use self("betax") instead.

betay()
Deprecated since version unknown: The function betay is deprecated. Use self("betay") instead.

betaz()
Deprecated since version unknown: The function betaz is deprecated. Use self("betaz") instead.

charge()
Deprecated since version unknown: The function charge is deprecated. Use self("charge") instead.

charge_e()
Deprecated since version unknown: The function charge_e is deprecated. Use self("charge_e") instead.

compress(*condition*, *name='unknown condition'*)
works like numpy.compress. Returns a new MultiSpecies instance.

Additionaly you can specify a name, that gets saved in the compresslog.

condition has to be one out of: 1) condition = [True, False, True, True, ... , True, False] condition is a list of length N, specifing which particles to keep. Example: cfintospectrometer = lambda x: x.angle_offaxis() < 30e-3 cfintospectrometer.name = '< 30mrad offaxis' pa.compress(cfintospectrometer(pa), name=cfintospectrometer.name) 2) condition = [1, 2, 4, 5, 9, ... , 805, 809] condition can be a list of arbitraty length, so only the particles with the ids listed here are kept.

name – name the condition. This can later be reviewed by calling 'self.compresslog()'

compressfn(*conditionf*, *name='unknown condition'*)
like "compress", but accepts a function.

Returns a new MultiSpecies instance.

name – name the condition.

createField(**sps*, ***kwargs*)
Creates an n-d Histogram enclosed in a Field object.

Parameters

- ***sps** – list of scalarfunctions/strings/scalar-properties, that will be evaluated to data for each axis. the number of args given determins the dimensionality of the field returned by this function (maximum 3)
- **name** (*string, optional*) – addes a name. usually used for generating savenames. Defaults to "distfn".
- **title** (*string, options*) – overrides the title. Autocreated if title==None. Defaults to None.

- **rangex** (*list of two values, optional*) – the xrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.
- **rangey** (*list of two values, optional*) – the yrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.
- **rangez** (*list of two values, optional*) – the zrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.

dumpreader
returns the dumpreader if the dumpreader of **all** species are pointing to the same dump. This should be mostly the case.

Otherwise returns None.

filter(*condition*, *name=None*)
like compress, but takes a ScalarProperty or a str, which are required to evaluate to a boolean list to filter particles. This is the preferred method to filter particles by a value of their property.

Returns a new MultiSpecies instance.

gamma()
Deprecated since version unknown: The function gamma is deprecated. Use self("gamma") instead.

gamma_m1()
Deprecated since version unknown: The function gamma_m1 is deprecated. Use self("gamma_m1") instead.

getcompresslog()

initial_npart
Original number of particles (before the use of compression or filter).

mass()
Deprecated since version unknown: The function mass is deprecated. Use self("mass") instead.

mass_u()
Deprecated since version unknown: The function mass_u is deprecated. Use self("mass_u") instead.

mean(*expr*, *weights='1'*)
the mean of a value given by the function func. The particle weight of the individual particles will be included in the calculation. An additional weight can be given as well.

median(*expr*, *weights='1'*)
The median

name
an alias to self.species

npart
Number of Particles.

nspecies
Number of species.

quantile(*expr*, *q*, *weights='1'*)
The qth-quantile of the distribution.

r_xy()
Deprecated since version unknown: The function r_xy is deprecated. Use self("r_xy") instead.

r_xyz()
Deprecated since version unknown: The function r_xyz is deprecated. Use self("r_xyz") instead.

r_yz()
Deprecated since version unknown: The function r_yz is deprecated. Use self("r_yz") instead.

r_zx()
Deprecated since version unknown: The function r_zx is deprecated. Use self("r_zx") instead.

simextent(*axis*)
the combined simextent for all species and dumps included in this MultiSpecies object.

simgridpoints(*axis*)
this function is for convenience only and is likely to be removed in the future. Particlarly it is impossible to define the grid of the simulation if the MultiSpecies object consists of multiple dumps from different simulations.

species
returns an string name for the species involved. Basically only returns unique names from all species (used for plotting and labeling purposes – not for completeness). May be overwritten.

speciess
a complete list of all species involved.

time()
Deprecated since version unknown: The function time is deprecated. Use self("time") instead.

uncompress()
Returns a new MultiSpecies instance, with all previous calls of "compress" or "filter" undone.

var(*expr*, *weights='1'*)
variance

weight()
Deprecated since version unknown: The function weight is deprecated. Use self("weight") instead.

class postpic.particles.**ParticleHistory**(*sr*, *speciess*, *ids=None*)
Bases: object

Represents a list of particles including their history that can be found in all the dumps defined by the simulation reader sr.

Parameters

- **sr** (*iterable of datareader*) – a collection of datareader to use. Usually a Simulationreader object
- **speciess** (*string or iterable of strings*) – a species name or a list of species names. Those particles can be included into the history.
- **ids** (*iterable of int*) – list of ids to use (default: None). If this is None all particles in speciess will be tracked. If a list of ids is given, these ids will be serached in speciess only.

collect(**scalarfs*)
Collects the given particle properties for all particles for all times.

*scalarfs: the scalarfunction(s) defining the particle property

numpy.ndarray holding the different particles in the same order as the list of *self.ids*, meaning the particle on position *particle_idx* has the ID *self.ids[particle_idx]*. every array element holds the history for a single particle. Indexorder of returned array: [particle_idx][scalarf_idx, collection_idx]

skip(*n*)
takes only everth (n+1)-th particle

postpic.particles.**histogramdd**(*data*, ***kwargs_in*)

Creates a histogram of the data. This function has the similar signature and return values as *numpy.histogramdd*. In addition this function supports the *shape* keyword argument to choose the particle shape used. If used with *shape=0* the results of this function and the *numpy.histogramdd* are identical, however, this function is approx. factor 2 or 3 faster.

Parameters

- **data** (*sequence of ndarray or ndarray (1D or 2D)*) –

  The input (particle) data for the histogram.

  - A 1D numpy array (for 1D histogram).
  - A sequence providing the data for the different axis, i.e. *(datax, datay, dataz)* (preferred).
  - A (N, D)-array, i.e. *[[x1, y1, z1], [x2, y2, z2]]* – must be a numpy array!

- **bins** (*sequence or int*) – The number of bins to use for each dimension
- **range** (*sequence, optional*) – A sequence of lower and upper bin edges to be used if the edges are not given explicitly in bins. Defaults to the minimum and maximum values along each dimension.
- **weights** (*1D numpy array*) – The weights to be used for each data point
- **shape** (*int*) –

  possible choices are:

  - 0 - use nearest grid point (NGP)
  - 1 - use tophat shape of width 1 bin
  - 2 - triangular shape (default)
  - 3 - spline 3 shape

Returns

- **H** (*ndarray*) – the final histogram
- **edges** (*list*) – A list of D arrays describing the edges for each dimension

class postpic.particles.**SpeciesIdentifier**

Bases: *postpic.helper.PhysicalConstants*

This Class provides static methods for deriving particle properties from species Names. The only reason for this to be a class is that it can be used as a mixin.

classmethod identifyspecies(*species*)

Returns a dictionary containing particle informations deduced from the species name. The following keys in the dictionary will always be present: name species name string mass kg (SI) charge C (SI) tracer boolean ejected boolean

Valid Examples: Periodic Table symbol + charge state: c6, F2, H1, C6b ionm#c# defining mass and charge: ionm12c2, ionc20m110 advanced examples: ejected_tracer_ionc5m20b, ejected_tracer_electronx, ejected_c6b, tracer_proton, protonb

static isejected(*species*)

classmethod ision(*species*)

Submodules

postpic.particles.particles module

Particle related routines.

class postpic.particles.particles.**MultiSpecies**(*dumpreader*, **speciess*, ***kwargs*)

Bases: object

The MultiSpecies class. Different MultiSpecies can be added together to create a combined collection.

ignore_missing_species = False set to true to ignore missing species.

The MultiSpecies class will return a list of values for every particle property.

Ekin()

Deprecated since version unknown: The function Ekin is deprecated. Use self("Ekin") instead.

Ekin_MeV()

Deprecated since version unknown: The function Ekin_MeV is deprecated. Use self("Ekin_MeV") instead.

Ekin_MeV_amu()

Deprecated since version unknown: The function Ekin_MeV_amu is deprecated. Use self("Ekin_MeV_amu") instead.

Ekin_MeV_qm()

Deprecated since version unknown: The function Ekin_MeV_qm is deprecated. Use self("Ekin_MeV_qm") instead.

Ekin_keV()

Deprecated since version unknown: The function Ekin_keV is deprecated. Use self("Ekin_keV") instead.

Ekin_keV_amu()

Deprecated since version unknown: The function Ekin_keV_amu is deprecated. Use self("Ekin_keV_amu") instead.

Ekin_keV_qm()

Deprecated since version unknown: The function Ekin_keV_qm is deprecated. Use self("Ekin_keV_qm") instead.

Eruhe()

Deprecated since version unknown: The function Eruhe is deprecated. Use self("Eruhe") instead.

ID()

Deprecated since version unknown: The function ID is deprecated. Use self("id") instead.

P()

Deprecated since version unknown: The function P is deprecated. Use self("p") instead.

Px()

Deprecated since version unknown: The function Px is deprecated. Use self("px") instead.

Py()

Deprecated since version unknown: The function Py is deprecated. Use self("py") instead.

Pz()

Deprecated since version unknown: The function Pz is deprecated. Use self("pz") instead.

V()

Deprecated since version unknown: The function V is deprecated. Use self("v") instead.

Vx()
Deprecated since version unknown: The function Vx is deprecated. Use self("vx") instead.

Vy()
Deprecated since version unknown: The function Vy is deprecated. Use self("vy") instead.

Vz()
Deprecated since version unknown: The function Vz is deprecated. Use self("vz") instead.

X()
Deprecated since version unknown: The function X is deprecated. Use self("x") instead.

X_um()
Deprecated since version unknown: The function X_um is deprecated. Use self("x_um") instead.

Y()
Deprecated since version unknown: The function Y is deprecated. Use self("y") instead.

Y_um()
Deprecated since version unknown: The function Y_um is deprecated. Use self("Y_mu") instead.

Z()
Deprecated since version unknown: The function Z is deprecated. Use self("z") instead.

Z_um()
Deprecated since version unknown: The function Z_um is deprecated. Use self("z_um") instead.

add(*dumpreader*, *species*, *ignore_missing_species=False*)
adds a species to this MultiSpecies. This function modifies the current Object and always returns None.

species can be a single species name
or a reserved name for collection of species, such as ions adds all available particles that are ions nonions adds all available particles that are not ions ejected noejected all

Optional arguments

----

ignore_missing_species = False
set to True to ignore if the species is missing.

angle_xaxis()
Deprecated since version unknown: The function angle_xaxis is deprecated. Use self("angle_xaxis") instead.

angle_xy()
Deprecated since version unknown: The function angle_xy is deprecated. Use self("angle_xy") instead.

angle_xz()
Deprecated since version unknown: The function angle_xz is deprecated. Use self("angle_xz") instead.

angle_yx()
Deprecated since version unknown: The function angle_yx is deprecated. Use self("angle_yx") instead.

angle_yz()
Deprecated since version unknown: The function angle_yz is deprecated. Use self("angle_yz") instead.

angle_zx()
Deprecated since version unknown: The function angle_zx is deprecated. Use self("angle_zx") instead.

angle_zy()
Deprecated since version unknown: The function angle_zy is deprecated. Use self("angle_zy") instead.

beta()
Deprecated since version unknown: The function beta is deprecated. Use self("beta") instead.

betax()
Deprecated since version unknown: The function betax is deprecated. Use self("betax") instead.

betay()
Deprecated since version unknown: The function betay is deprecated. Use self("betay") instead.

betaz()
Deprecated since version unknown: The function betaz is deprecated. Use self("betaz") instead.

charge()
Deprecated since version unknown: The function charge is deprecated. Use self("charge") instead.

charge_e()
Deprecated since version unknown: The function charge_e is deprecated. Use self("charge_e") instead.

compress(*condition*, *name='unknown condition'*)
works like numpy.compress. Returns a new MultiSpecies instance.

Additionaly you can specify a name, that gets saved in the compresslog.

condition has to be one out of: 1) condition = [True, False, True, True, ... , True, False] condition is a list of length N, specifing which particles to keep. Example: cfintospectrometer = lambda x: x.angle_offaxis() < 30e-3 cfintospectrometer.name = '< 30mrad offaxis' pa.compress(cfintospectrometer(pa), name=cfintospectrometer.name) 2) condition = [1, 2, 4, 5, 9, ... , 805, 809] condition can be a list of arbitraty length, so only the particles with the ids listed here are kept.

name – name the condition. This can later be reviewed by calling 'self.compresslog()'

compressfn(*conditionf*, *name='unknown condition'*)
like "compress", but accepts a function.

Returns a new MultiSpecies instance.

name – name the condition.

createField(**sps*, ***kwargs*)
Creates an n-d Histogram enclosed in a Field object.

Parameters

- ***sps** – list of scalarfunctions/strings/scalar-properties, that will be evaluated to data for each axis. the number of args given determins the dimensionality of the field returned by this function (maximum 3)
- **name** (*string, optional*) – addes a name. usually used for generating savenames. Defaults to "distfn".
- **title** (*string, options*) – overrides the title. Autocreated if title==None. Defaults to None.
- **rangex** (*list of two values, optional*) – the xrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.
- **rangey** (*list of two values, optional*) – the yrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.
- **rangez** (*list of two values, optional*) – the zrange to include into the histogram. Defaults to None, determins the range by the range of scalars given.

dumpreader
returns the dumpreader if the dumpreader of **all** species are pointing to the same dump. This should be mostly the case.

Otherwise returns None.

filter(*condition*, *name=None*)
like compress, but takes a ScalarProperty or a str, which are required to evaluate to a boolean list to filter particles. This is the preferred method to filter particles by a value of their property.

Returns a new MultiSpecies instance.

gamma()
Deprecated since version unknown: The function gamma is deprecated. Use self("gamma") instead.

gamma_m1()
Deprecated since version unknown: The function gamma_m1 is deprecated. Use self("gamma_m1") instead.

getcompresslog()

initial_npart
Original number of particles (before the use of compression or filter).

mass()
Deprecated since version unknown: The function mass is deprecated. Use self("mass") instead.

mass_u()
Deprecated since version unknown: The function mass_u is deprecated. Use self("mass_u") instead.

mean(*expr*, *weights='1'*)
the mean of a value given by the function func. The particle weight of the individual particles will be included in the calculation. An additional weight can be given as well.

median(*expr*, *weights='1'*)
The median

name
an alias to self.species

npart
Number of Particles.

nspecies
Number of species.

quantile(*expr*, *q*, *weights='1'*)
The qth-quantile of the distribution.

r_xy()
Deprecated since version unknown: The function r_xy is deprecated. Use self("r_xy") instead.

r_xyz()
Deprecated since version unknown: The function r_xyz is deprecated. Use self("r_xyz") instead.

r_yz()
Deprecated since version unknown: The function r_yz is deprecated. Use self("r_yz") instead.

r_zx()
Deprecated since version unknown: The function r_zx is deprecated. Use self("r_zx") instead.

simextent(*axis*)
the combined simextent for all species and dumps included in this MultiSpecies object.

simgridpoints(*axis*)
this function is for convenience only and is likely to be removed in the future. Particlarly it is impossible to define the grid of the simulation if the MultiSpecies object consists of multiple dumps from different simulations.

species
returns an string name for the species involved. Basically only returns unique names from all species (used for plotting and labeling purposes – not for completeness). May be overwritten.

speciess
a complete list of all species involved.

time()
Deprecated since version unknown: The function time is deprecated. Use self("time") instead.

uncompress()
Returns a new MultiSpecies instance, with all previous calls of "compress" or "filter" undone.

var(*expr*, *weights='1'*)
variance

weight()
Deprecated since version unknown: The function weight is deprecated. Use self("weight") instead.

class postpic.particles.particles.**ParticleHistory**(*sr*, *speciess*, *ids=None*)
Bases: object

Represents a list of particles including their history that can be found in all the dumps defined by the simulation reader sr.

Parameters

- **sr** (*iterable of datareader*) – a collection of datareader to use. Usually a Simulationreader object
- **speciess** (*string or iterable of strings*) – a species name or a list of species names. Those particles can be included into the history.
- **ids** (*iterable of int*) – list of ids to use (default: None). If this is None all particles in speciess will be tracked. If a list of ids is given, these ids will be serached in speciess only.

collect(**scalarfs*)
Collects the given particle properties for all particles for all times.

*scalarfs: the scalarfunction(s) defining the particle property

numpy.ndarray holding the different particles in the same order as the list of *self.ids*, meaning the particle on position *particle_idx* has the ID *self.ids[particle_idx]*. every array element holds the history for a single particle. Indexorder of returned array: [particle_idx][scalarf_idx, collection_idx]

skip(*n*)
takes only everth (n+1)-th particle

postpic.particles.scalarproperties module

class postpic.particles.scalarproperties.**ScalarProperty**(*expr*, *name=None*, *unit=None*, *symbol=None*)
Bases: object

evaluate(*vars*)
vars must be a dictionary containing variables used within the expression "expr".

expr

input_names
The list of variables used within this expression.

name

symbol

unit

postpic.plotting package

The plot subpackage should provide an interface to various plot backends.

postpic.plotting.**use**(*plotcls*)

Submodules

postpic.plotting.plotter_matplotlib module

This package provides the MatplotlibPlotter Class.

This Class can be used to plot Field Objects using the matplotlib interface.

class postpic.plotting.plotter_matplotlib.**MatplotlibPlotter**(*reader*, *outdir='./'*, *autosave=False*, *project=None*, *ext='png'*, *size_inches=(9, 7)*, *dpi=160*, *facecolor=(1, 1, 1, 0.01)*, *transparent=False*)

Bases: object

Provides Methods to modify figures and axes objects for convenient plotting. It also autogenerates savenames and annotates the plot if a reader is given. A reader can be a dumpreader or a simulationreder.

class LinearSegmentedColormap(*name*, *segmentdata*, *N=256*, *gamma=1.0*)
Bases: matplotlib.colors.Colormap

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

static from_list(*name*, *colors*, *N=256*, *gamma=1.0*)
Make a linear segmented colormap with *name* from a sequence of *colors* which evenly transitions from colors[0] at val=0 to colors[-1] at val=1. *N* is the number of rgb quantization levels. Alternatively, a list of (value, color) tuples can be given to divide the range unevenly.

set_gamma(*gamma*)
Set a new gamma value and regenerate color map.

static addField1d(*ax*, *field*, *log10plot=True*, *xlim=None*, *ylim=None*, *scaletight=None*)

static addField2d(*figax*, *field*, *log10plot=True*, *interpolation='none'*, *contourlevels=array([], dtype=float64)*, *saveandclose=True*, *xlim=None*, *ylim=None*, *clim=None*, *savecsv=False*, *lineoutx=False*, *lineouty=False*, ***kwargs*)

static addFields1d(*ax*, **fields*, ***kwargs*)

static addaxislabels(*ax*, *field*)

static annotate(*figorax*, *title=None*, *time=None*, *step=None*, *project=None*, *dump=None*, *infostring=None*, *infos=None*)

static annotate_fromfield(*figorax*, *field*)

static annotate_fromreader(*figorax*, *reader*)

axesformatterx = <matplotlib.ticker.ScalarFormatter object>

axesformattery = <matplotlib.ticker.ScalarFormatter object>

efieldcdict = {'blue': ((0, 1, 1), (1, 0, 0)), 'green': ((0, 0, 0), (1, 0, 0)), 'red

lastsavename()
returns the last savenme. If there wasnt a last a new savename is created.

matplotlib = <module 'matplotlib' from '/usr/lib/python3/dist-packages/matplotlib/__in

plotField(*field*, *autoreduce=True*, *maxlen=6000*, *name=None*, ***kwargs*)
This is the main method, that should be used for plotting.

plotField2d(*field*, *name=None*, ***kwargs*)

plotFields(**fields*, ***kwargs*)

plotFields1d(**fields*, ***kwargs*)

plotallderived(*dumpreader*)
plots all fields dumped.

project

savefig(*fig*, *key*)

savename(*key*, *ext=None*)

static settext_ax(*ax*, *title=None*, *ur=None*, *ur2=None*, *ul=None*, *ul2=None*, *center=None*)

static settext_fig(*fig*, *title=None*, *ur=None*, *ur2=None*, *ul=None*, *ul2=None*, *center=None*)

symmap = <matplotlib.colors.LinearSegmentedColormap object>

static symmetricclim(*ax*)
symmetrize the clim around 0.

static symmetricclimaximage(*aximage*)
symmetrize the clim around 0.

5.1.1.2 Submodules

5.1.1.3 postpic.datahandling module

The Core module for final data handling.

This module provides classes for dealing with axes, grid as well as the Field class – the final output of the postpic postprocessor.

Terminology

A data field with N numeric points has N 'grid' points, but N+1 'grid_nodes' as depicted here:

```
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
  o   o   o   o   o     grid      (coordinates where data is sampled at)
o   o   o   o   o   o   grid_node (coordinates of grid cell boundaries)
|                   |   extent
```

class postpic.datahandling.**Field**(*matrix*, *name=''*, *unit=''*, ***kwargs*)

Bases: postpic._compat.mixins.NDArrayOperatorsMixin

The Field Object carries data in form of an *numpy.ndarray* together with as many Axis objects as the data's dimensions. Additionaly the Field object provides any information that is necessary to plot _and_ annotate the plot.

Create a Field object from scratch. The only required argument is *matrix* which contains the actual data.

A *name* and a *unit* may be supplied.

The axis may be specified in different ways:

- by passing a list of Axis object as *axes*
- by passing arrays with the grid_nodes as *xedges*, *yedges* and *zedges*. This is intended to work with *np.histogram*.
- by not passing anything, which will create default axes from 0 to 1.

T

Return the Field with the order of axes reversed. In 2D this is the usual matrix transpose operation.

adjust_stagger_to(*other*)

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

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See also:

numpy.all() equivalent function

angle

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

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

atleast_nd(*n*)

Make sure the field has at least 'n' dimensions

autocutout (*axes=None*, *fractions=(0.001*, *0.002)*)
Automatically cuts out the main feature of the field by removing border regions that only contain small numbers.

This is done axis by axis. For each axis, the mean across all other axes is taken. The maximum *max* of the remaining 1d-*array* is taken and searched for the outermost boundaries a, d such that all values out of array[a:d] are smaller then fractions[0]*max. A second set of boundaries b, c is searched such that all values out of array[b:c] are smaller then fractions[1]*max. Because fractions[1] should be larger than fractions[0], array[b:c] should be contained completely in array[a:d].

A padding length x is chosen such that array[b-x:c+x] is entirely within array[a:d].

Then the corresponding axis of the field is sliced to [b-x:c+x] and multiplied with a tukey-window such that the region [b:c] is left untouched and the field in the padding region smoothly vanishes on the outer border.

This process is repeated for all axes in *axes* or for all axes if *axes* is None.

autoreduce (*maxlen=4000*)
Reduces the Grid to a maximum length of maxlen per dimension by just executing half_resolution as often as necessary.

clip (*a_min*, *a_max*, *out=None*)

conj ()

cutout (*newextent*)
only keeps that part of the data, that belongs to newextent.

dimensions
returns only present dimensions. [] and [[]] are interpreted as -1 np.array(2) is interpreted as 0 np.array([1,2,3]) is interpreted as 1 and so on...

ensure_frequency_domain ()

ensure_spatial_domain ()

ensure_transform_state (*transform_states*)
Makes sure that the field has the given transform_states. *transform_states* may be a single boolean, indicating the same desired transform_state for all axes. It may be a list of the desired transform states for all the axes or a dictionary indicating the desired transform states of specific axes.

export (*filename*, ***kwargs*)

Uses *postpic.export_field* to export this field to a file. All ‘‘**kwargs‘ will be forwarded to this function. Format is recognized by the extension of the filename.

export Field object as a file. Format depends on the extention of the filename. Currently supported are: .npz:

uses *numpy.savez*.

.csv: uses *numpy.savetxt*.

.vtk: vtk export to paraview

extent
returns the extents in a linearized form, as required by “matplotlib.pyplot.imshow”.

fft (*axes=None*, *exponential_signs='spatial'*, ***kwargs*)
Performs Fourier transform on any number of axes.

The argument axis is either an integer indicating the axis to be transformed or a tuple giving the axes that should be transformed. Automatically determines forward/inverse transform. Transform is only applied if all mentioned axes are in the same transform state. If an axis is transformed twice, the origin of the axis is restored.

Parameters

- **exponential_signs** – configures the sign convention of the exponential.
  - exponential_signs == 'spatial': fft using exp(-ikx), ifft using exp(ikx)
  - exponential_signs == 'temporal': fft using exp(iwt), ifft using exp(-iwt)
- ****kwargs** – keyword-arguments are passed to the underlying fft implementation.

fft_autopad(*axes=None*, *fft_padsize=<postpic.helper.FFTW_Pad object>*)

Automatically pad the array to a size such that computing its FFT using FFTW will be fast.

Parameters **fft_padsize** (*callable*) – The default for keyword argument *fft_padsize* is a callable, that is used to calculate the padded size for a given size.

By default, this uses *fft_padsize=helper.fftw_padsize* which finds the next larger "good" grid size according to what the FFTW documentation says.

However, the FFTW documentation also says: "(...) Transforms whose sizes are powers of 2 are especially fast."

If you don't worry about the extra padding, you can pass *fft_padsize=helper.fft_padsize_power2* and this method will pad to the next power of 2.

grid

grid_nodes

half_resolution(*axis*)

Halfs the resolution along the given axis by removing every second *grid_node* and averaging every second data point into one.

If there is an odd number of grid points, the last point will be ignored (that means, the extent will change by the size of the last grid cell).

Returns the modified *Field*.

Return type *Field*

imag

integrate(*axes=None*, *method=<function simps>*)

Calculates the definite integral along the given axes.

Parameters **method** (*callable*) – Choose the method to use. Available options:

- 'constant'
- any function with the same signature as scipy.integrate.simps (default).

islinear()

label

classmethod loadfrom(*filename*)

construct a new field object from file. currently, the following file formats are supported: *.npz

map_axis_grid(*axis*, *transform*, *preserve_integral=True*, *jacobian_func=None*)
Transform the Field to new coordinates along one axis.

This function transforms the coordinates of one axis according to the function *transform* and applies the jacobian to the data.

Please note that no interpolation is applied to the data, instead a non-linear axis grid is produced. If you want to interpolate the data to a new (linear) grid, use the method *map_coordinates()* instead.

In contrast to *map_coordinates()*, the function transform is not used to pull the new data points from the old grid, but is directly applied to the axis. This reverses the direction of the transform. Therfore, in order to preserve the integral, it is necessary to divide by the Jacobian.

Parameters

- **axis** (*int*) – the index or name of the axis you want to apply transform to.
- **transform** (*callable*) – the transformation function which takes the old coordinates as an input and returns the new grid
- **preserve_integral** (*bool*) – Divide by the jacobian of transform, in order to preserve the integral.
- **jacobian_func** (*callable*) – If given, this is expected to return the derivative of transform. If not given, the derivative is numerically approximated.

map_coordinates(*newaxes*, *transform=None*, *complex_mode='polar'*, *preserve_integral=True*, *jacobian_func=None*, *jacobian_determinant_func=None*, ***kwargs*)
Transform the Field to new coordinates.

Parameters

- **newaxes** (*list*) – The new axes of the new coordinates.
- **transform** (*callable*) – a callable that takes the new coordinates as input and returns the old coordinates from where to sample the Field. It is basically the inverse of the transformation that you want to perform. If transform is not given, the identity will be used. This is suitable for simple interpolation to a new extent/shape. Example for cartesian -> polar:

```
>>> def T(r, theta):
>>>     x = r * np.cos(theta)
>>>     y = r * np.sin(theta)
>>>     return x, y
```

Note that this function actually computes the cartesian coordinates from the polar coordinates, but stands for transforming a field in cartesian coordinates into a field in polar coordinates.

However, in order to preserve the definite integral of the field, it is necessary to multiply with the Jacobian determinant of T.

$$\tilde{U}(r,\theta) = U(T(r,\theta)) \cdot \det \frac{\partial(x,y)}{\partial(r,\theta)}$$

such that

$$\int_V \mathrm{d}x\,\mathrm{d}y\, U(x,y) = \int_{T^{-1}(V)} \mathrm{d}r\,\mathrm{d}\theta\, \tilde{U}(r,\theta)\,.$$

- **complex_mode** – The complex_mode specifies how to proceed with complex data.
  - complex_mode = 'cartesian' - interpolate real/imag part (fastest)

  - complex_mode = 'polar' - interpolate abs/phase If skimage.restoration is available, the phase will be unwrapped first (default)
  - complex_mode = 'polar-no-unwrap' - interpolate abs/phase Skip unwrapping the phase, even if skimage.restoration is available

- **preserve_integral** (*bool*) – If True (the default), the data will be multiplied with the Jacobian determinant of the coordinate transformation such that the integral over the data will be preserved.

  In general, you will want to do this, because the physical unit of the new Field will correspond to the new axis of the Fields. Please note that Postpic, currently, does not automatically change the unit members of the Axis and Field objects, this you will have to do manually.

  There are, however, exceptions to this rule. Most prominently, if you are converting to polar coordinates, it depends on what you are going to do with the transformed Field. If you intend to do a Cartesian r-theta plot or are interested in a lineout for a single value of theta, you do want to apply the Jacobian determinant. If you had a density in e.g. J/m^2 than, in polar coordinates, you want to have a density in J/m/rad. If you intend, on the other hand, to do a polar plot, you do not want to apply the Jacobian. In a polar plot, the data points are plotted with variable density which visually takes care of the Jacobian automatically. A polar plot of the polar data should look like a Cartesian plot of the original data with just a peculiar coordinate grid drawn over it.

- **jacobian_determinant_func** (*callable*) – A callable that returns the jacobian determinant of the transform. If given, this takes precedence over the following option.
- **jacobian_func** (*callable*) – a callable that returns the jacobian of the transform. If this is not given, the jacobian is numerically approximated.
- ****kwargs** – Additional keyword arguments are passed to *scipy.ndimage.map_coordinates*, see the documentation of that function.

matrix

max(*axis=None*, *out=None*)
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*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See also:

numpy.mean() equivalent function

meshgrid(*sparse=True*)

min(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

See also:

numpy.amin() equivalent function

pad(*pad_width*, *mode='constant'*, ***kwargs*)
Pads the data using *np.pad* and takes care of the axes. See documentation of *numpy.pad*.

In contrast to *np.pad*, *pad_width* may be given as integers, which will be interpreted as pixels, or as floats, which will be interpreted as distance along the appropriate axis.

All other parameters are passed to *np.pad* unchanged.

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

Refer to *numpy.ptp* for full documentation.

See also:

numpy.ptp() equivalent function

real

replace_data(*other*)

saveto(*filename*)
Save a Field object as a file. Use *loadfrom()* to load Field objects.

setaxisobj(*axis*, *axisobj*)
replaces the current axisobject for axis axis by the new axisobj axisobj.

shape

shift_grid_by(*dx*, *interpolation='fourier'*)
Translate the Grid by *dx*. This is useful to remove the grid stagger of field components.

If all axis will be shifted, *dx* may be a list. Otherwise dx should be a mapping from axis to translation distance.

The keyword-argument interpolation indicates the method to be used and may be one of *['linear', 'fourier']*. In case of interpolation = 'fourier' all axes must have same transform_state.

spacing
returns the grid spacings for all axis.

squeeze()
removes axes that have length 1, reducing self.dimensions.

Note, that axis with length 0 will not be removed! *numpy.squeeze* also does not remove length=0 directions.

Same as *numpy.squeeze*.

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See also:

numpy.std() equivalent function

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
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*)
Swaps the axes *axis1* and *axis2*, equivalent to *numpy.swapaxes*.

topolar(*extent=None*, *shape=None*, *angleoffset=0*, ***kwargs*)
Transform the Field to polar coordinates.

This is a convenience wrapper for *map_coordinates()* which will let you easily define the desired grid in polar coordinates.

Parameters

- **extent** – should be of the form *extent=(phimin, phimax, rmin, rmax)* or *extent=(phimin, phimax)*
- **shape** – should be of the form *shape=(N_phi, N_r)*,
- **angleoffset** – can be any real number and will rotate the zero-point of the angular axis.
- **complex_mode** – The complex_mode specifies how to proceed with complex data.
  - complex_mode = 'cartesian' - interpolate real/imag part (fastest)
  - complex_mode = 'polar' - interpolate abs/phase If skimage.restoration is available, the phase will be unwrapped first (default)
  - complex_mode = 'polar-no-unwrap' - interpolate abs/phase Skip unwrapping the phase, even if skimage.restoration is available
- **preserve_integral** (*bool*) – If True (the default), the data will be multiplied with the Jacobian determinant of the coordinate transformation such that the integral over the data will be preserved.

  In general, you will want to do this, because the physical unit of the new Field will correspond to the new axis of the Fields. Please note that Postpic, currently, does not automatically change the unit members of the Axis and Field objects, this you will have to do manually.

  There are, however, exceptions to this rule. Most prominently, if you are converting to polar coordinates, it depends on what you are going to do with the transformed Field. If you intend to do a Cartesian r-theta plot or are interested in a lineout for a single value of theta, you do want to apply the Jacobian determinant. If you had a density in e.g. J/m^2 than, in polar coordinates, you want to have a density in J/m/rad. If you intend, on the other hand, to do a polar plot, you do not want to apply the Jacobian. In a polar plot, the data points are plotted with variable density which visually takes care of the Jacobian automatically. A polar plot of the polar data should look like a Cartesian plot of the original data with just a peculiar coordinate grid drawn over it.

- **jacobian_determinant_func** (*callable*) – A callable that returns the jacobian determinant of the transform. If given, this takes precedence over the following option.
- **jacobian_func** (*callable*) – a callable that returns the jacobian of the transform. If this is not given, the jacobian is numerically approximated.
- ****kwargs** – Additional keyword arguments are passed to *scipy.ndimage.map_coordinates*, see the documentation of that function.

transpose(**axes*)
transpose method equivalent to *numpy.ndarray.transpose*. If *axes* is empty, the order of the axes will be reversed. Otherwise axes[i] == j means that the i'th axis of the returned Field will be the j'th axis of the input Field.

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

Refer to *numpy.var* for full documentation.

See also:

numpy.var() equivalent function

class postpic.datahandling.**Axis**(*name=''*, *unit=''*, ***kwargs*)
Bases: object

Axis handling for a single Axis.

Create an Axis object from scratch.

The least required arguments are any of:

- grid
- grid_node
- extent _and_ n

The remaining fields will be deduced from the givens.

More arguments may be supplied, as long as they are compatible.

extent

grid

grid_node

half_resolution()
removes every second grid_node.

islinear(*force=False*)
Checks if the axis has a linear grid.

label

physical_length

spacing

value_to_index(*value*)

5.1.1.4 postpic.experimental module

Some experimental algorithms for your reference. Please note that these algorithms are not meant to be used as is and may need adjustment in order to be applicable to a wider range of cases.

postpic.experimental.**kspace_propagate_adaptive**(*field_in*, *axis=0*, *t_final=None*, ***kwargs*)

An adaptive method to use Fourier propagation (provided by the function *helper.kspace_propagate*) to get far field data. The field is padded, propagated and automatically sliced in repeating steps.

Note that this method is highly experimental and should not be trusted as is: It is merely meant as a recipe so you don't have to write your own function from scratch!

field_in: input field in either spatial or frequency domain *axis*: The direction in which to propagate. Currently only propagation parallel to the positive x, y or z direction is implemented. *yield_zeroth_step*: boolean that determines if the initial step is also output.

t_final: The time at which to stop the adaptive propagation.

5.1.1.5 postpic.helper module

Some global constants that are used in the code.

class postpic.helper.**PhysicalConstants**

Bases: object

gives you some constants.

c = 299792458.0

epsilon0 = 8.854187817620389e-12

mass_u = 1.67266490646e-27

me = 9.109383e-31

mu0 = 1.2566370614359173e-06

static ncrit(*laslambda*)

Critical plasma density in particles per m^3 for a given wavelength laslambda in m.

static ncrit_um(*lambda_um*)

Critical plasma density in particles per m^3 for a given wavelength lambda_um in microns.

qe = 1.602176565e-19

postpic.helper.**unstagger_fields**(**fields*, ***kwargs*)

Unstagger a collection of fields.

This functions shifts the origins of the grids of the given fields such that they coincide. Since the choice of the common origin is somewhat arbitrary, it might be overriden by a keyword-argument *origin*, as may be the interpolation *method*. See *Field.shift_grid_by* for available methods.

postpic.helper.**kspace_epoch_like**(*component*, *fields*, *dt*, *extent=None*, *omega_func=<function omega_free>*, *align_to='B'*)

Reconstruct the physical kspace of one polarization component See documentation of kspace

This function will use special care to make sure, that the implicit linear interpolation introduced by Epochs half-steps will not impede the accuracy of the reconstructed k-space. The frequency response of the linear interpolation is modelled and removed from the interpolated fields.

dt: time-step of the simulation, this is used to calculate the frequency response due to the linear interpolated half-steps

For the current version of EPOCH, v4.9, use the following: align_to == 'B' for intermediate dumps, align_to == "E" for final dumps

postpic.helper.**kspace**(*component*, *fields*, *extent=None*, *interpolation=None*, *omega_func=<function omega_free>*)

Reconstruct the physical kspace of one polarization component This function basically computes one component of

E = 0.5*(E - omega/k^2 * Cross[k, E])

or B = 0.5*(B + 1/omega * Cross[k, B]).

component must be one of ["Ex", "Ey", "Ez", "Bx", "By", "Bz"].

The necessary fields must be given in the dict fields with keys chosen from ["Ex", "Ey", "Ez", "Bx", "By", "Bz"]. Which are needed depends on the chosen component and the dimensionality of the fields. In 3D the following fields are necessary:

Ex, By, Bz -> Ex Ey, Bx, Bz -> Ey Ez, Bx, By -> Ez

Bx, Ey, Ez -> Bx By, Ex, Ez -> By Bz, Ex, Ey -> Bz

In 2D, components which have "k_z" in front of them (see cross-product in equations above) are not needed. In 1D, components which have "k_y" or "k_z" in front of them (see cross-product in equations above) are not needed.

The keyword-argument extent may be a list of values [xmin, xmax, ymin, ymax, ...] which denote a region of the Fields on which to execute the kspace reconstruction.

The keyword-argument interpolation indicates whether interpolation should be used to remove the grid stagger. If interpolation is None, this function works only for non-staggered grids. Other choices for interpolation are "linear" and "fourier".

The keyword-argument omega_func may be used to pass a function that will calculate the dispersion relation of the simulation may be given. The function will receive one argument that contains the k mesh.

postpic.helper.**kspace_propagate**(*kspace*, *dt*, *nsteps=1*, ***kwargs*)

Evolve time on a field. This function checks the transform_state of the field and transforms first from spatial domain to frequency domain if necessary. In this case the inverse transform will also be applied to the result before returning it. This works, however, only correctly with fields that are the inverse transforms of a k-space reconstruction, i.e. with complex fields.

dt: time in seconds

This function will return an infinite generator that will do arbitrary many time steps.

If yield_zeroth_step is True, then the kspace will also be yielded after removing the antipropagating waves, but before the first actual step is done.

If a vector moving_window_vect is passed to this function, which is ideally identical to the mean propagation direction of the field in forward time direction, an additional linear phase is applied in order to keep the pulse inside of the box. This effectively enables propagation in a moving window. If dt is negative, the window will actually move the opposite direction of moving_window_vect. Additionally, all modes which propagate in the opposite direction of the moving window, i.e. all modes for which dot(moving_window_vect, k)<0, will be deleted.

The motion of the window can be inhibited by specifying move_window=False. If move_window is None, the moving window is automatically enabled if moving_window_vect is given.

The deletion of the antipropagating modes can be inhibited by specifying remove_antipropagating_waves=False. If remove_antipropagating_waves is None, the deletion of the antipropagating modes is automatically enabled if moving_window_vect is given.

nsteps: number of steps to take

If nsteps == 1, this function will just return the result. If nsteps > 1, this function will return a generator that will generate the results. If you want a list, just put list(...) around the return value.

postpic.helper.**time_profile_at_plane**(*kspace_or_complex_field*, *axis='x'*, *value=None*, *dir=1*, ***kwargs*)

'Measure' the time-profile of the propagating *complex_field* while passing through a plane.

The arguments *axis*, *value* and *dir* specify the plane and main propagation direction.

axis specifies the axis perpendicular to the measurement plane.

dir=1 specifies propagation towards positive *axis*, *dir=-1* specifies the opposite direction of propagation.

value specifies the position of the plane along *axis*. If *value=None,* a default is chosen, depending on *dir*.

If *dir=-1*, the starting point of the axis is used, which lies at the 0-component of the inverse transform.

If *dir=1*, the end point of the axis + one axis spacing is used, which, via periodic boundary conditions of the fft, also lies at the 0-component of the inverse transform.

If the given *value* differs from these defaults, an initial propagation with moving window will be performed, such that the desired plane lies in the default position.

For example *axis='x'* and *value=0.0* specifies the 'x=0.0' plane while *dir=1* specifies propagation towards positive 'x' values. The 'x' axis starts at 2e-5 and ends at 6e-5 with a grid spacing of 1e-6. The default value for the measurement plane would have been 6.1e-5 so an initial backward propagation with dt = -6.1e-5/c is performed to move the pulse in front of the'x=0.0 plane.

Additional *kwargs* are passed to kspace_propagate if they are not overridden by this function.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z