Plonk Documentation

Release 0.7.4

Daniel Mentiplay

Oct 13, 2021

CONTENTS

1 About 3

2 Contents 5

3 Citation 105

4 Change log 107

5 Contributors 109

6 License 111

7 Other packages 113

8 Index 115

Python Module Index 117

Index 119

Smoothed particle hydrodynamics analysis and visualization with Python.

- Docs: https://plonk.readthedocs.io/
- Repo: https://www.github.com/dmentipl/plonk

CHAPTER

ONE

ABOUT

Plonk is a Python tool for analysis and visualization of smoothed particle hydrodynamics data with a focus on astrophysical fluid dynamics.

With Plonk we aim to integrate the high quality SPH visualisation of Splash into the modern Python astronomer workflow, and to provide a framework for analysis of smoothed particle hydrodynamics simulation data.

Note: Plonk is being used for scientific publications. However, no software is bug free. Please use Plonk and if you have any issues or feature requests please leave feedback by raising a GitHub issue.

CHAPTER

TWO

CONTENTS

2.1 Getting started

A gentle guide to getting started with Plonk.

2.1.1 Installation

Conda

You can install Plonk via the package manager Conda from the conda-forge channel.

```
conda install plonk --channel conda-forge
```

This will install the required dependencies.

Note: You can simply use `conda install plonk` if you add the `conda-forge` channel with `conda config --add channels conda-forge`. I also recommend strictly using conda-forge which you can do with `conda config --set channel_priority true`. Both of these commands modify the Conda configuration file `~/.condarc`.

pip

You can also install Plonk from PyPI via pip.

```
python -m pip install plonk
```

This should install the required dependencies.

Source

You can install Plonk from source as follows.

```
# clone via HTTPS
$ git clone https://github.com/dmentipl/plonk.git

# or clone via SSH
$ git clone git@github.com:dmentipl/plonk
```

(continues on next page)

(continued from previous page)

```
$ cd plonk
$ python -m pip install -e .
```

This assumes you have already installed the dependencies. One way to do this is by setting up a conda environment. The environment.yml file provided sets up a conda environment "plonk" for using or developing Plonk.

```
conda env create --file environment.yml
conda activate plonk
```

Plonk has Python runtime requirements listed in setup.cfg in the `install_requires` variable.

2.1.2 Overview

This document gives an overview of using Plonk for analysis and visualization of smoothed particle hydrodynamics data. For a further guide see *Usage*.

Working with SPH data

First import the Plonk package.

```
>>> import plonk
```

We also import Matplotlib and NumPy, for later.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

Snapshots

SPH snapshot files are represented by the *Snap* class. This object contains a properties dictionary, particle arrays, which are lazily loaded from file. Here we demonstrate instantiating a *Snap* object, and accessing some properties and particle arrays.

First, we load the snapshot with the *load_snap()* function. You can pass a string or `pathlib.Path` object to point to the location of the snapshot in the file system.

```
>>> filename = 'disc_00030.h5'
>>> snap = plonk.load_snap(filename)
```

You can access arrays by their name passed in as a string.

```
>>> snap['position']
array([[-3.69505001e+12,  7.42032967e+12, -7.45096980e+11],
       [-1.63052677e+13,  1.16308971e+13,  1.92879212e+12],
       [-7.66283930e+12,  1.62532232e+13,  2.34302988e+11],
       ...,
       [ 1.39571712e+13, -1.16179990e+13,  8.09090354e+11],
       [ 9.53716176e+12,  9.98500386e+12,  4.93933367e+11],
       [ 1.21421196e+12,  2.08618956e+13,  1.12998892e+12]]) <Unit('meter')>
```

There may be a small delay as the data is read from file. After the array is read from file it is cached in memory, so that subsequent calls are faster.

To see what arrays are loaded into memory you can use the *loaded_arrays()* method.

```
>>> snap.loaded_arrays()
['position']
```

Use *available_arrays()* to see what arrays are available. Some of these arrays are stored on file, while others are computed as required.

```
>>> snap.available_arrays()
['angular_momentum',
 'angular_velocity',
 'azimuthal_angle',
 'density',
 'dust_to_gas_ratio',
 'id',
 'kinetic_energy',
 'mass',
 'momentum',
 'polar_angle',
 'position',
 'pressure',
 'radius_cylindrical',
 'radius_spherical',
 'smoothing_length',
 'sound_speed',
 'specific_angular_momentum',
 'specific_kinetic_energy',
 'stopping_time',
 'sub_type',
 'temperature',
 'timestep',
 'type',
 'velocity',
 'velocity_divergence',
 'velocity_radial_cylindrical',
 'velocity_radial_spherical']
```

You can also define your own alias to access arrays. For example, if you prefer to use the name `'coordinate'` rather than `'position'`, use the *add_alias()* method to add an alias.

```
>>> snap.add_alias(name='position', alias='coordinate')
>>> snap['coordinate']
array([[-3.69505001e+12,  7.42032967e+12, -7.45096980e+11],
       [-1.63052677e+13,  1.16308971e+13,  1.92879212e+12],
       [-7.66283930e+12,  1.62532232e+13,  2.34302988e+11],
       ...,
       [ 1.39571712e+13, -1.16179990e+13,  8.09090354e+11],
       [ 9.53716176e+12,  9.98500386e+12,  4.93933367e+11],
       [ 1.21421196e+12,  2.08618956e+13,  1.12998892e+12]]) <Unit('meter')>
```

The *Snap* object has a *properties* attribute which is a dictionary of metadata, i.e. non-array data, on the snapshot.

```
>>> snap.properties['time']
61485663602.558136 <Unit('second')>

>>> list(snap.properties)
['adiabatic_index',
 'dust_method',
 'equation_of_state',
 'grain_density',
 'grain_size',
 'smoothing_length_factor',
 'time']
```

Units are available. We make use of the Python units library Pint. The code units of the data are available as *code_units*.

```
>>> snap.code_units['length']
149600000000.0 <Unit('meter')>
```

You can set default units as follows.

```
>>> snap.set_units(position='au', density='g/cm^3', velocity='km/s')
<plonk.Snap "disc_00030.h5">

>>> snap['position']
array([[ -24.6998837 ,   49.60184016,   -4.98066567],
       [-108.99398271,   77.74774493,   12.89317897],
       [ -51.22291689,  108.64608658,    1.56621873],
       ...,
       [  93.29792694,  -77.66152625,    5.40843496],
       [  63.75198868,   66.74562821,    3.30174062],
       [   8.11650561,  139.453159  ,    7.55350939]]) <Unit('astronomical_unit')>
```

Sink particles are handled separately from the fluid, e.g. gas or dust, particles. They are available as an attribute *sinks*.

```
>>> snap.sinks
<plonk.snap sinks>

>>> sinks = snap.sinks

>>> sinks.available_arrays()
['accretion_radius',
 'last_injection_time',
 'mass',
 'mass_accreted',
 'position',
 'softening_radius',
 'spin',
 'velocity']

>>> sinks['spin']
array([[ 3.56866999e+36, -1.17910663e+37,  2.44598074e+40],
       [ 4.14083556e+36,  1.19118555e+36,  2.62569386e+39]]) <Unit('kilogram * meter **␣
↪2 / second')>
```

Simulation

SPH simulation data is usually spread over multiple files of, possibly, different types, even though, logically, a simulation is a singular "object". Plonk has the *Simulation* class to represent the complete data set. *Simulation* is an aggregation of the *Snap* and pandas DataFrames to represent time series data (see below), plus metadata, such as the directory on the file system.

Use the *load_simulation()* function to instantiate a *Simulation* object.

```
>>> prefix = 'disc'
>>> sim = plonk.load_simulation(prefix=prefix)
```

Each of the snapshots are available via *snaps* as a list. We can get the first five snapshots with the following.

```
>>> sim.snaps[:5]
[<plonk.Snap "disc_00000.h5">,
 <plonk.Snap "disc_00001.h5">,
 <plonk.Snap "disc_00002.h5">,
 <plonk.Snap "disc_00003.h5">,
 <plonk.Snap "disc_00004.h5">]
```

The *Simulation* class has an attribute *time_series* which contains time series data as a pandas DataFrame discussed in the next section.

Time series

SPH simulation datasets often include auxiliary files containing globally-averaged time series data output more frequently than snapshot files. For example, Phantom writes text files with the file extension ".ev". These files are output every time step rather than at the frequency of the snapshot files.

We store this data in a pandas DataFrame. Use *load_time_series()* to instantiate.

```
>>> ts = plonk.load_time_series('disc01.ev')
```

The data may be split over several files, for example, if the simulation was run with multiple jobs on a computation cluster. In that case, pass in a list of files in chronological order to *load_time_series()*, and Plonk will concatenate the data removing any duplicated time steps.

The underlying data is stored as a pandas[1] DataFrame. This allows for the use of typical pandas operations with which users in the scientific Python community may be familiar with.

```
>>> ts
             time  energy_kinetic  energy_thermal  ...  gas_density_average  dust_
↪density_max  dust_density_average
0        0.000000        0.000013        0.001186  ...         8.231917e-10      1.
↪720023e-10          8.015937e-12
1        1.593943        0.000013        0.001186  ...         8.229311e-10      1.
↪714059e-10          8.015771e-12
2        6.375774        0.000013        0.001186  ...         8.193811e-10      1.
↪696885e-10          8.018406e-12
3       25.503096        0.000013        0.001186  ...         7.799164e-10      1.
↪636469e-10          8.061417e-12
4       51.006191        0.000013        0.001186  ...         7.249247e-10      1.
↪580470e-10          8.210622e-12
```

(continues on next page)

[1] See https://pandas.pydata.org/ for more on pandas.

(continued from previous page)

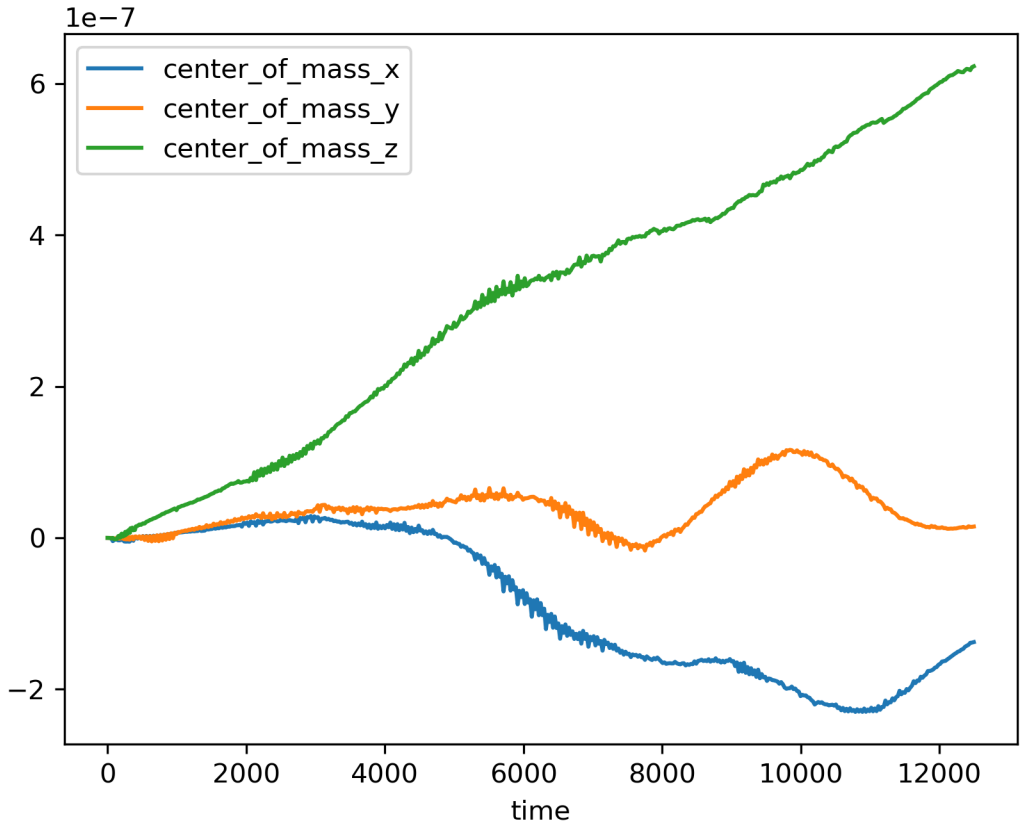
```
..           ...            ...            ...  ...                 ...            ␣
↪ ...                    ...
548  12394.504462       0.000013       0.001186  ...       6.191121e-10      1.
↪481833e-09          2.482929e-11
549  12420.007557       0.000013       0.001186  ...       6.189791e-10      1.
↪020596e-09          2.483358e-11
550  12445.510653       0.000013       0.001186  ...       6.188052e-10      8.
↪494835e-10          2.488946e-11
551  12471.013748       0.000013       0.001186  ...       6.186160e-10      6.
↪517475e-10          2.497029e-11
552  12496.516844       0.000013       0.001186  ...       6.184558e-10      5.
↪205011e-10          2.506445e-11

[553 rows x 21 columns]
```

You can plot columns with the pandas plotting interface.

```
>>> ts.plot('time', ['center_of_mass_x', 'center_of_mass_y', 'center_of_mass_z'])
```

The previous code produces the following figure.



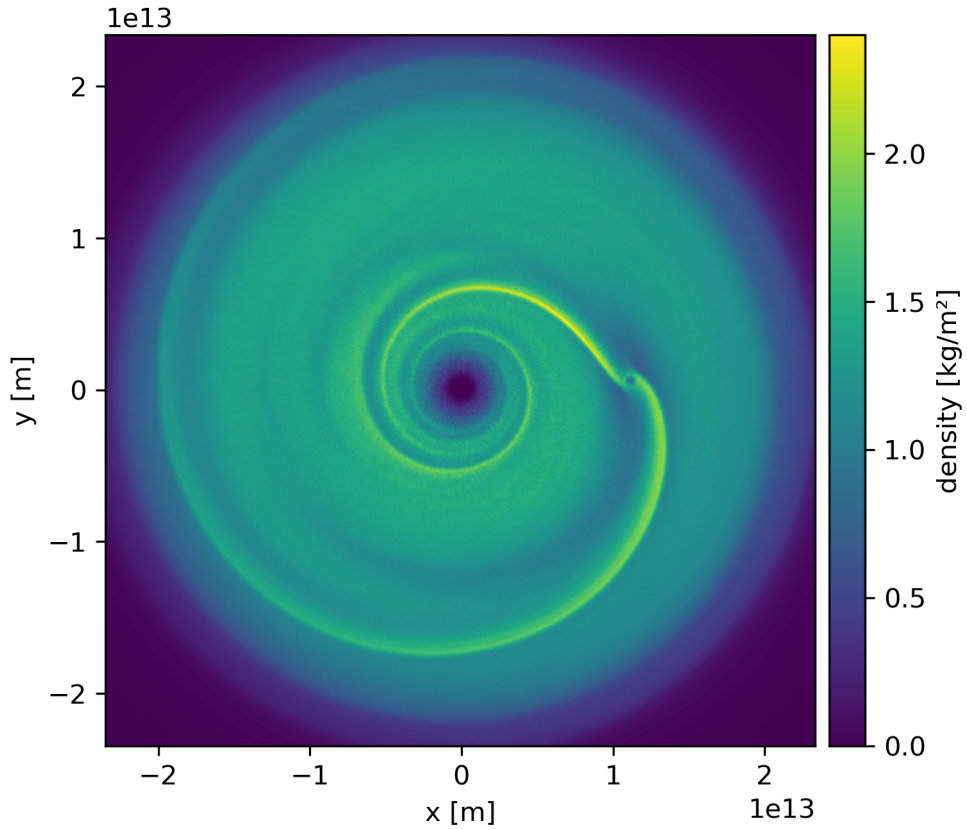
Visualization of SPH data

SPH particle data is not gridded like the data produced by, for example, finite difference or finite volume hydrodynamical codes. One visualization method is to plot the particles as a scatter plot, and possibly color the particles with the magnitude of a quantity of interest. An alternative is to interpolate any quantity on the particles to a pixel grid with weighted kernel density estimation. This is what Splash does. For the technical details, see Price (2007), PASA, 24, 3, 159. We use the same numerical method as Splash, with the Python function compiled with Numba so it has the same performance as the Fortran code.

You can use the *image()* method to interpolate a quantity to a pixel grid to show as an image. For example, in the following we produce a plot of column density, i.e. a projection plot.

```
>>> filename = 'disc_00030.h5'
>>> snap = plonk.load_snap(filename)

>>> snap.image(quantity='density')
```



This produces an image via Matplotlib. The function returns a Matplotlib `Axes` object.

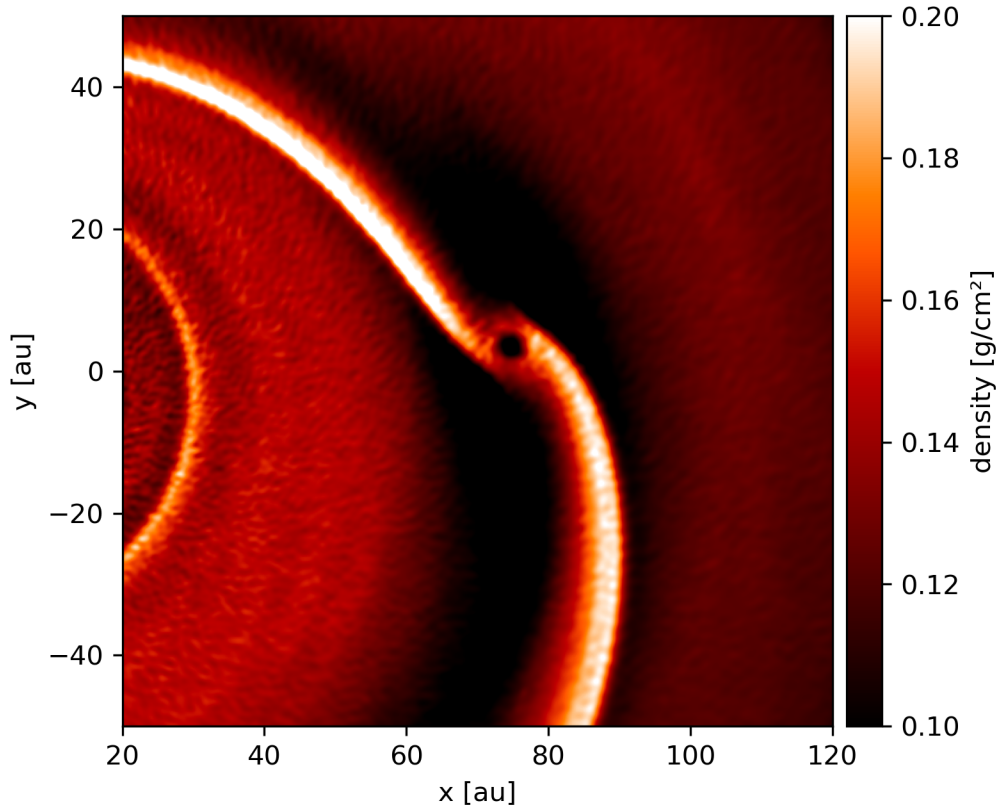
Alternatively, you can pass keyword arguments to the matplotlib functions. For example, we set the units, the colormap to 'gist_heat' and set the colorbar minimum and maxiumum. In addition, we set the extent, i.e. the x- and y-limits.

```
>>> snap.set_units(position='au', density='g/cm^3', projection='cm')
>>> snap.image(
...     quantity='density',
...     extent=(20, 120, -50, 50),
...     cmap='gist_heat',
```

(continues on next page)

(continued from previous page)

```
...     vmin=0.1,
...     vmax=0.2,
... )
```



More fine-grained control can be achieved by using the full details of *image()*. See the API for more details.

Analysis of SPH data

Subsnaps

When analyzing SPH data it can be useful to look at a subset of particles. For example, the simulation we have been working with has dust and gas. So far we have been plotting the total density. We may want to visualize the dust and gas separately.

To do this we make a *SubSnap* object. We can access these quantities using the *family()* method. Given that there may be sub-types of dust, using 'dust' returns a list by default. In this simulation there is only one dust species. We can squeeze all the dust sub-types together using the `squeeze` argument.

```
>>> gas = snap.family('gas')
>>> dust = snap.family('dust', squeeze=True)
```

You can access arrays on the *SubSnap* objects as for any *Snap* object.

```
>>> gas['mass'].sum().to('solar_mass')
0.0010000000000000005 <Unit('solar_mass')>
```

(continues on next page)

(continued from previous page)

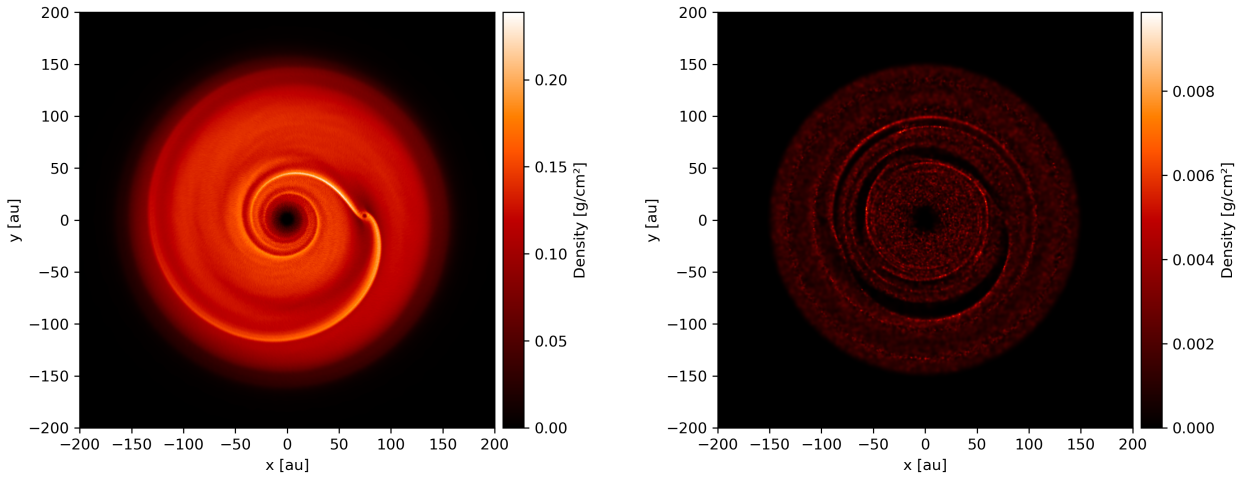
```
>>> dust['mass'].sum().to('earth_mass')
3.326810503428664 <Unit('earth_mass')>
```

Let's plot the gas and dust side-by-side.

```
>>> subsnaps = [gas, dust]
>>> extent = (-200, 200, -200, 200)

>>> fig, axs = plt.subplots(ncols=2, figsize=(14, 5))

>>> for subsnap, ax in zip(subsnaps, axs):
...     subsnap.image(quantity='density', extent=extent, cmap='gist_heat', ax=ax)
```



Derived arrays

Sometimes you need new arrays on the particles that are not available in the snapshot files. Many are available in Plonk already. The following code block lists the available raw Phantom arrays on the file.

```
>>> list(snap._file_pointer['particles'])
['divv', 'dt', 'dustfrac', 'h', 'itype', 'tstop', 'vxyz', 'xyz']
```

To see all available arrays on the *Snap* object:

```
>>> snap.available_arrays()
['angular_momentum',
 'angular_velocity',
 'azimuthal_angle',
 'density',
 'dust_to_gas_ratio',
 'id',
 'kinetic_energy',
 'mass',
 'momentum',
 'polar_angle',
```

(continues on next page)

(continued from previous page)

```
 'position',
 'pressure',
 'radius_cylindrical',
 'radius_spherical',
 'smoothing_length',
 'sound_speed',
 'specific_angular_momentum',
 'specific_kinetic_energy',
 'stopping_time',
 'sub_type',
 'temperature',
 'timestep',
 'type',
 'velocity',
 'velocity_divergence',
 'velocity_radial_cylindrical',
 'velocity_radial_spherical']
```

This is a disc simulation. You can add quantities appropriate for discs with the *add_quantities()* method.

```
>>> previous_arrays = snap.available_arrays()

>>> snap.add_quantities('disc')

# Additional available arrays
>>> set(snap.available_arrays()) - set(previous_arrays)
{'eccentricity',
 'inclination',
 'keplerian_frequency',
 'semi_major_axis',
 'stokes_number'}
```

You can create a new, derived array on the particles as follows.

```
>>> snap['rad'] = np.sqrt(snap['x'] ** 2 + snap['y'] ** 2)

>>> snap['rad']
array([8.28943225e+12, 2.00284678e+13, 1.79690392e+13, ...,
       1.81598604e+13, 1.38078875e+13, 2.08972008e+13]) <Unit('meter')>
```

Where, here, we have used the fact that Plonk knows that 'x' and 'y' refer to the x- and y-components of the position array.

Alternatively, you can define a function for a derived array. This makes use of the decorator *add_array()*.

```
>>> @snap.add_array()
... def radius(snap):
...     radius = np.hypot(snap['x'], snap['y'])
...     return radius

>>> snap['radius']
array([8.28943225e+12, 2.00284678e+13, 1.79690392e+13, ...,
       1.81598604e+13, 1.38078875e+13, 2.08972008e+13]) <Unit('meter')>
```

Units

Plonk uses Pint to set arrays to physical units.

```
>>> snap = plonk.load_snap(filename)

>>> snap['x']
array([-3.69505001e+12, -1.63052677e+13, -7.66283930e+12, ...,
        1.39571712e+13,  9.53716176e+12,  1.21421196e+12]) <Unit('meter')>
```

It is easy to convert quantities to different units as required.

```
>>> snap['x'].to('au')
array([ -24.6998837 , -108.99398271,  -51.22291689, ...,   93.29792694,
         63.75198868,    8.11650561]) <Unit('astronomical_unit')>
```

Profiles

Generating a profile is a convenient method to reduce the dimensionality of the full data set. For example, we may want to see how the surface density and aspect ratio of the disc vary with radius.

To do this we use the *Profile* class in the `analysis` module.

```
>>> snap = plonk.load_snap(filename)

>>> snap.add_quantities('disc')

>>> prof = plonk.load_profile(snap, cmin='10 au', cmax='200 au')

>>> prof
<plonk.Profile "disc_00030.h5">
```

To see what profiles are loaded and what are available use the *loaded_profiles()* and *available_profiles()* methods.

```
>>> prof.loaded_profiles()
['number', 'radius', 'size']

>>> prof.available_profiles()
['alpha_shakura_sunyaev',
 'angular_momentum_mag',
 'angular_momentum_phi',
 'angular_momentum_theta',
 'angular_momentum_x',
 'angular_momentum_y',
 'angular_momentum_z',
 'angular_velocity',
 'aspect_ratio',
 'azimuthal_angle',
 'density',
 'disc_viscosity',
 'dust_to_gas_ratio_001',
 'eccentricity',
```

(continues on next page)

(continued from previous page)

```
 'epicyclic_frequency',
 'id',
 'inclination',
 'keplerian_frequency',
 'kinetic_energy',
 'mass',
 'midplane_stokes_number_001',
 'momentum_mag',
 'momentum_x',
 'momentum_y',
 'momentum_z',
 'number',
 'polar_angle',
 'position_mag',
 'position_x',
 'position_y',
 'position_z',
 'pressure',
 'radius',
 'radius_cylindrical',
 'radius_spherical',
 'scale_height',
 'semi_major_axis',
 'size',
 'smoothing_length',
 'sound_speed',
 'specific_angular_momentum_mag',
 'specific_angular_momentum_x',
 'specific_angular_momentum_y',
 'specific_angular_momentum_z',
 'specific_kinetic_energy',
 'stokes_number_001',
 'stopping_time_001',
 'sub_type',
 'surface_density',
 'temperature',
 'timestep',
 'toomre_q',
 'type',
 'velocity_divergence',
 'velocity_mag',
 'velocity_radial_cylindrical',
 'velocity_radial_spherical',
 'velocity_x',
 'velocity_y',
 'velocity_z']
```

To load a profile, simply call it.

```
>>> prof['scale_height']
array([7.97124951e+10, 9.84227660e+10, 1.18761140e+11, 1.37555034e+11,
       1.57492383e+11, 1.79386553e+11, 1.99666627e+11, 2.20898696e+11,
```

(continues on next page)

(continued from previous page)

```
       2.46311866e+11, 2.63718852e+11, 2.91092881e+11, 3.11944125e+11,
       3.35101091e+11, 3.58479038e+11, 3.86137691e+11, 4.09731915e+11,
       4.34677739e+11, 4.61230912e+11, 4.87471032e+11, 5.07589421e+11,
       5.38769961e+11, 5.64068787e+11, 5.88487300e+11, 6.15197082e+11,
       6.35591229e+11, 6.75146081e+11, 6.94786320e+11, 7.32840731e+11,
       7.65750662e+11, 7.96047221e+11, 8.26764128e+11, 8.35658042e+11,
       8.57126522e+11, 8.99037935e+11, 9.26761324e+11, 9.45798955e+11,
       9.65997944e+11, 1.01555592e+12, 1.05554201e+12, 1.07641999e+12,
       1.10797835e+12, 1.13976869e+12, 1.17181502e+12, 1.20083661e+12,
       1.23779947e+12, 1.24785058e+12, 1.28800980e+12, 1.31290341e+12,
       1.33602542e+12, 1.34618607e+12, 1.36220344e+12, 1.39412340e+12,
       1.41778445e+12, 1.46713623e+12, 1.51140364e+12, 1.54496807e+12,
       1.58327631e+12, 1.60316504e+12, 1.63374960e+12, 1.64446331e+12,
       1.66063803e+12, 1.67890856e+12, 1.68505873e+12, 1.68507230e+12,
       1.71353612e+12, 1.71314330e+12, 1.75704484e+12, 1.79183025e+12,
       1.83696336e+12, 1.88823477e+12, 1.93080810e+12, 1.98301979e+12,
       2.05279086e+12, 2.11912539e+12, 2.14224572e+12, 2.21647741e+12,
       2.27153917e+12, 2.36605186e+12, 2.38922067e+12, 2.53901104e+12,
       2.61297334e+12, 2.64782574e+12, 2.73832897e+12, 3.04654121e+12,
       3.13575612e+12, 3.37636281e+12, 3.51482502e+12, 3.69591185e+12,
       3.88308614e+12, 3.83982313e+12, 4.00147149e+12, 4.37288049e+12,
       4.35330982e+12, 4.44686164e+12, 4.47133547e+12, 4.83307604e+12,
       4.63783507e+12, 4.95119779e+12, 5.17961431e+12, 5.29308491e+12]) <Unit('meter')>
```

You can convert the data in the *Profile* object to a pandas DataFrame with the *to_dataframe()* method. This takes all loaded profiles and puts them into the DataFrame with units indicated in brackets.

```
>>> profiles = [
...     'radius',
...     'angular_momentum_phi',
...     'angular_momentum_theta',
...     'surface_density',
...     'scale_height',
... ]

>>> df = prof.to_dataframe(columns=profiles)

>>> df
      radius [m]  angular_momentum_phi [rad]  angular_momentum_theta [rad]  surface_
↪density [kg / m ** 2]  scale_height [m]
0   1.638097e+12                   -0.019731                      0.049709             ␣
↪        0.486007      7.971250e+10
1   1.922333e+12                    1.914841                      0.053297             ␣
↪        0.630953      9.842277e+10
2   2.206569e+12                    1.293811                      0.055986             ␣
↪        0.808415      1.187611e+11
3   2.490805e+12                   -2.958286                      0.057931             ␣
↪        0.956107      1.375550e+11
4   2.775041e+12                   -1.947547                      0.059679             ␣
↪        1.095939      1.574924e+11
..           ...                         ...                           ...             ␣
↪            ...               ...
```

(continues on next page)

(continued from previous page)

```
95  2.864051e+13                    3.045660                    0.168944
↪        0.013883      4.833076e+12
96  2.892475e+13                   -0.054956                    0.161673
↪        0.013246      4.637835e+12
97  2.920898e+13                   -0.217485                    0.169546
↪        0.011058      4.951198e+12
98  2.949322e+13                   -1.305261                    0.175302
↪        0.011367      5.179614e+12
99  2.977746e+13                    2.642077                    0.176867
↪        0.010660      5.293085e+12

[100 rows x 5 columns]
```

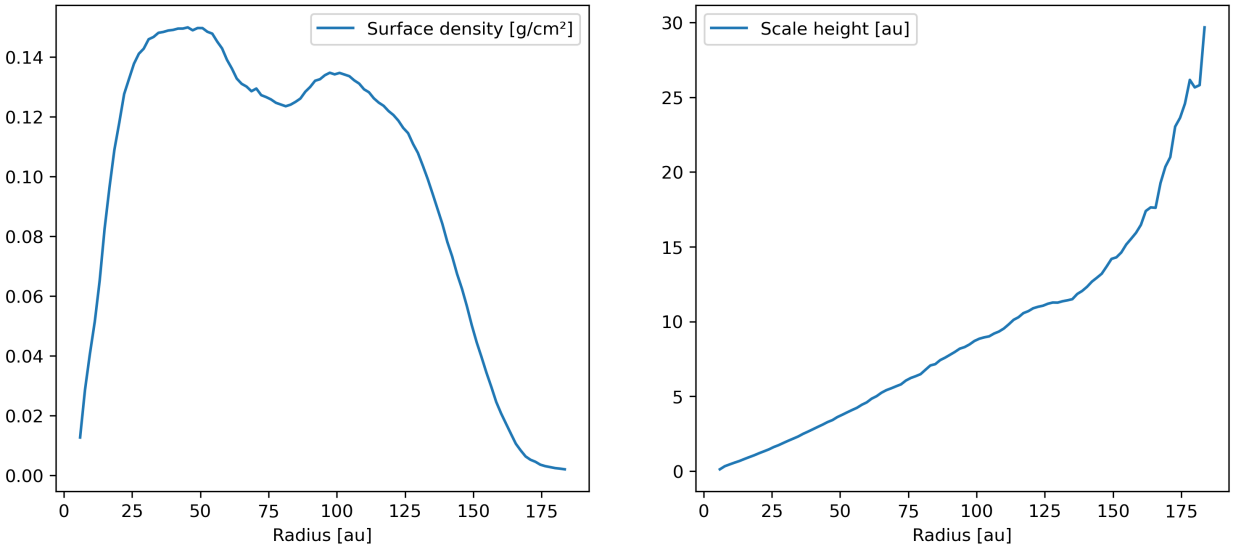
We can also plot the profiles.

```
>>> prof.set_units(position='au', scale_height='au', surface_density='g/cm^2')

>>> fig, axs = plt.subplots(ncols=2, figsize=(12, 5))

>>> prof.plot('radius', 'surface_density', ax=axs[0])

>>> prof.plot('radius', 'scale_height', ax=axs[1])
```



Important: To follow along, download the sample data `plonk_example_data.tar` from figshare. Then extract with `tar xvf plonk_example_data.tar` and change into the `plonk_example_data` directory. This data set is from a Phantom simulation of a dust and gas protoplanetary disc with an embedded protoplanet.

Data file formats

Plonk supports the following SPH file formats:

- Phantom output in HDF form (as opposed to the sphNG-based Fortran binary format).

Note: HDF5 output was added to Phantom as an option with git commit 9b22ded on the 14th of March 2019. See the Phantom documentation for instructions on how to compile with HDF5 output and to convert from the sphNG-based output.

2.1.3 Getting help

If you need help, try the following:

1. Check this documentation.
2. Check the built-in help, e.g. `help(plonk.load_snap)`.
3. File an issue, as a bug report or feature request, using the issue tracker on GitHub.

If you don't get an immediate response, please be patient. Plonk is maintained by one person, @dmentipl.

2.2 User guide

The user guide is in two parts: *Usage* containing short, self-contained code snippets demonstrating Plonk features with detaileed description, and *Examples* containing a collection of scripts with a science focus.

2.2.1 Usage

Usage guide for Plonk. This section contains short, self-contained code snippets demonstrating usage of Plonk features. See the *Examples* section for more complicated usage.

Data

Load snapshot

Load data from a single snapshot into *Snap* and see what arrays are available with *available_arrays()*.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap.available_arrays()
['angular_momentum',
 'angular_velocity',
 'azimuthal_angle',
 'density',
 'dust_to_gas_ratio',
 'id',
 'kinetic_energy',
```

(continues on next page)

(continued from previous page)

```
 'mass',
 'momentum',
 'polar_angle',
 'position',
 'pressure',
 'radius_cylindrical',
 'radius_spherical',
 'smoothing_length',
 'sound_speed',
 'specific_angular_momentum',
 'specific_kinetic_energy',
 'stopping_time',
 'sub_type',
 'temperature',
 'timestep',
 'type',
 'velocity',
 'velocity_divergence',
 'velocity_radial_cylindrical',
 'velocity_radial_spherical']
```

Load a single snapshot and access particle arrays and *properties*.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap['position']
array([[-3.69505001e+12,  7.42032967e+12, -7.45096980e+11],
       [-1.63052677e+13,  1.16308971e+13,  1.92879212e+12],
       [-7.66283930e+12,  1.62532232e+13,  2.34302988e+11],
       ...,
       [ 1.39571712e+13, -1.16179990e+13,  8.09090354e+11],
       [ 9.53716176e+12,  9.98500386e+12,  4.93933367e+11],
       [ 1.21421196e+12,  2.08618956e+13,  1.12998892e+12]]) <Unit('meter')>

>>> snap['position'].to('au')
array([[ -24.6998837 ,   49.60184016,   -4.98066567],
       [-108.99398271,   77.74774493,   12.89317897],
       [ -51.22291689,  108.64608658,    1.56621873],
       ...,
       [  93.29792694,  -77.66152625,    5.40843496],
       [  63.75198868,   66.74562821,    3.30174062],
       [   8.11650561,  139.453159  ,    7.55350939]]) <Unit('astronomical_unit')>

>>> snap.properties['time']
61485663602.558136 <Unit('second')>
```

Load a single snapshot and access sink arrays via *sinks* attribute.

```
>>> import plonk
```

(continues on next page)

(continued from previous page)

```
>>> snap = plonk.load_snap('disc_00030.h5')

>>> sinks = snap.sinks

>>> sinks.available_arrays()
['accretion_radius',
 'last_injection_time',
 'mass',
 'mass_accreted',
 'position',
 'softening_radius',
 'spin',
 'velocity']

>>> sinks['spin']
array([[ 3.56866999e+36, -1.17910663e+37,  2.44598074e+40],
       [ 4.14083556e+36,  1.19118555e+36,  2.62569386e+39]]) <Unit('kilogram * meter **␣
↪2 / second')>
```

Load time series data

Load a Phantom time series (`.ev`) file with *load_time_series()* and see what columns are available.

```
>>> import plonk

>>> ts = plonk.load_time_series('disc01.ev')

>>> ts.columns
Index(['time', 'energy_kinetic', 'energy_thermal', 'energy_magnetic',
       'energy_potential', 'energy_total', 'momentum', 'angular_momentum',
       'density_max', 'density_average', 'timestep', 'entropy',
       'mach_number_rms', 'velocity_rms', 'center_of_mass_x',
       'center_of_mass_y', 'center_of_mass_z', 'gas_density_max',
       'gas_density_average', 'dust_density_max', 'dust_density_average'],
      dtype='object')

>>> ts
             time  energy_kinetic  ...  dust_density_max  dust_density_average
0        0.000000        0.000013  ...      1.720023e-10          8.015937e-12
1        1.593943        0.000013  ...      1.714059e-10          8.015771e-12
2        6.375774        0.000013  ...      1.696885e-10          8.018406e-12
3       25.503096        0.000013  ...      1.636469e-10          8.061417e-12
4       51.006191        0.000013  ...      1.580470e-10          8.210622e-12
..            ...             ...  ...               ...                   ...
548  12394.504462        0.000013  ...      1.481833e-09          2.482929e-11
549  12420.007557        0.000013  ...      1.020596e-09          2.483358e-11
550  12445.510653        0.000013  ...      8.494835e-10          2.488946e-11
551  12471.013748        0.000013  ...      6.517475e-10          2.497029e-11
552  12496.516844        0.000013  ...      5.205011e-10          2.506445e-11

[553 rows x 21 columns]
```

Load simulation

Load a simulation with *load_simulation()* and access snapshots and time series data with *snaps* and *time_series* attributes.

```
>>> import plonk

>>> sim = plonk.load_simulation(prefix='disc')

>>> sim.snaps
[<plonk.Snap "disc_00000.h5">,
 <plonk.Snap "disc_00001.h5">,
 <plonk.Snap "disc_00002.h5">,
 <plonk.Snap "disc_00003.h5">,
 <plonk.Snap "disc_00004.h5">,
 <plonk.Snap "disc_00005.h5">,
 <plonk.Snap "disc_00006.h5">,
 <plonk.Snap "disc_00007.h5">,
 <plonk.Snap "disc_00008.h5">,
 <plonk.Snap "disc_00009.h5">,
 <plonk.Snap "disc_00010.h5">,
 <plonk.Snap "disc_00011.h5">,
 <plonk.Snap "disc_00012.h5">,
 <plonk.Snap "disc_00013.h5">,
 <plonk.Snap "disc_00014.h5">,
 <plonk.Snap "disc_00015.h5">,
 <plonk.Snap "disc_00016.h5">,
 <plonk.Snap "disc_00017.h5">,
 <plonk.Snap "disc_00018.h5">,
 <plonk.Snap "disc_00019.h5">,
 <plonk.Snap "disc_00020.h5">,
 <plonk.Snap "disc_00021.h5">,
 <plonk.Snap "disc_00022.h5">,
 <plonk.Snap "disc_00023.h5">,
 <plonk.Snap "disc_00024.h5">,
 <plonk.Snap "disc_00025.h5">,
 <plonk.Snap "disc_00026.h5">,
 <plonk.Snap "disc_00027.h5">,
 <plonk.Snap "disc_00028.h5">,
 <plonk.Snap "disc_00029.h5">,
 <plonk.Snap "disc_00030.h5">]

>>> sim.time_series['global']
         time [s]  ...  dust_density_average [kg / m ** 3]
0    0.000000e+00  ...                          4.762293e-15
1    8.005946e+06  ...                          4.762195e-15
2    3.202378e+07  ...                          4.763760e-15
3    1.280951e+08  ...                          4.789313e-15
4    2.561903e+08  ...                          4.877956e-15
..            ...  ...                                   ...
548  6.225423e+10  ...                          1.475116e-14
549  6.238233e+10  ...                          1.475371e-14
550  6.251042e+10  ...                          1.478691e-14
```

(continues on next page)

(continued from previous page)

```
551  6.263852e+10  ...                        1.483493e-14
552  6.276661e+10  ...                        1.489087e-14

[553 rows x 21 columns]

>>> sim.time_series['sinks']
[          time [s]  position_x [m]  ...  sink_sink_force_y [N]  sink_sink_force_z [N]
 0     4.002973e+06   -1.068586e+10  ...           2.455750e+18           2.020208e+14
 1     8.005946e+06   -1.068584e+10  ...           4.911471e+18           2.020271e+14
 2     1.601189e+07   -1.068574e+10  ...           9.822886e+18           2.020559e+14
 3     3.202378e+07   -1.068536e+10  ...           1.964551e+19           2.021794e+14
 4     6.404757e+07   -1.068380e+10  ...           3.928915e+19           1.407952e+14
 ...            ...             ...  ...                    ...                    ...
 1038  6.257447e+10   -9.976304e+09  ...           7.513519e+20           7.902581e+15
 1039  6.263852e+10   -9.895056e+09  ...           7.884134e+20           7.664701e+15
 1040  6.270257e+10   -9.809975e+09  ...           8.251714e+20           7.628553e+15
 1041  6.276661e+10   -9.721096e+09  ...           8.616085e+20           7.705555e+15
 1042  6.283066e+10   -9.628452e+09  ...           8.977201e+20           7.391809e+15

 [1043 rows x 18 columns],
          time [s]  position_x [m]  ...  sink_sink_force_y [N]  sink_sink_force_z [N]
 0     4.002973e+06    1.120931e+13  ...          -2.573360e+21          -2.116959e+17
 1     8.005946e+06    1.120928e+13  ...          -5.146689e+21          -2.117025e+17
 2     1.601189e+07    1.120918e+13  ...          -1.029332e+22          -2.117327e+17
 3     3.202378e+07    1.120877e+13  ...          -2.058637e+22          -2.118621e+17
 4     6.404757e+07    1.120715e+13  ...          -4.117069e+22          -1.475378e+17
 ...            ...             ...  ...                    ...                    ...
 1038  6.257447e+10    1.041137e+13  ...          -7.748986e+23          -8.150240e+18
 1039  6.263852e+10    1.032805e+13  ...          -8.131071e+23          -7.904764e+18
 1040  6.270257e+10    1.024073e+13  ...          -8.510027e+23          -7.867359e+18
 1041  6.276661e+10    1.014946e+13  ...          -8.885717e+23          -7.946693e+18
 1042  6.283066e+10    1.005427e+13  ...          -9.257998e+23          -7.623017e+18

 [1043 rows x 18 columns]]
```

Analysis

Sub-snaps

Access the gas and dust subsets of the particles as a *SubSnap*.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> gas = snap.family('gas')

# Dust can have multiple sub-species
# So we combine into one family with squeeze=True
>>> dust = snap.family('dust', squeeze=True)
```

(continues on next page)

(continued from previous page)

```
>>> gas['mass'].sum().to('solar_mass')
0.0010000000000000005 <Unit('solar_mass')>

>>> dust['mass'].sum().to('earth_mass')
3.326810503428664 <Unit('earth_mass')>
```

Generate a *SubSnap* with a boolean mask.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap['x'].to('au').min()
-598.1288172965254 <Unit('astronomical_unit')>

# Particles with positive x-coordinate.
>>> mask = snap['x'] > 0

>>> subsnap = snap[mask]

>>> subsnap['x'].to('au').min()
0.0002668455543031563 <Unit('astronomical_unit')>
```

Generate a *SubSnap* of particles from lists or slices of indices.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

# Generate SubSnap from slices
>>> subsnap = snap[:1000]

# Generate SubSnap from lists
>>> subsnap = snap[[0, 1, 2, 3, 4]]
```

Filters

Filters are available to generate *SubSnap* from geometric shapes.

```
>>> import plonk

>>> from plonk.analysis import filters

>>> au = plonk.units['au']

>>> snap = plonk.load_snap('disc_00030.h5')

>>> width = 50 * au
>>> subsnap = filters.box(snap=snap, xwidth=width, ywidth=width, zwidth=width)

>>> radius_min, radius_max, height = 50 * au,  100 * au, 10 * au
```

(continues on next page)

(continued from previous page)

```
>>> subsnap = filters.annulus(
...     snap=snap, radius_min=radius_min, radius_max=radius_max, height=height
... )
```

Quantities

Calculate extra quantities on particle arrays. Note that many of these are available by default.

```
>>> import plonk

>>> from plonk.analysis import particles

>>> snap = plonk.load_snap('disc_00030.h5')

# Calculate angular momentum of each particle
>>> particles.angular_momentum(snap=snap)
array([[-2.91051358e+36,  5.03265707e+36,  6.45532986e+37],
       [ 7.83210945e+36, -5.83981869e+36,  1.01424601e+38],
       [ 5.42707198e+35, -1.15387855e+36,  9.77918546e+37],
       ...,
       [-3.65688200e+35,  2.36337004e+35,  9.70192741e+36],
       [-8.47414806e+35,  3.91073248e+35,  8.45673620e+36],
       [-1.04934629e+36, -5.04112542e+35,  1.04345024e+37]]) <Unit('kilogram * meter **␣
↪2 / second')>
```

Calculate total (summed) quantities on a Snap.

```
>>> import plonk

>>> from plonk.analysis import total

>>> snap = plonk.load_snap('disc_00030.h5')

# Calculate the center of mass over all particles including sinks
>>> total.center_of_mass(snap=snap).to('au')
array([-1.52182463e-07,  1.25054338e-08,  6.14859166e-07]) <Unit('astronomical_unit')>

# Calculate the kinetic energy including sinks
>>> total.kinetic_energy(snap=snap).to('joule')
2.232949413087673e+34 <Unit('joule')>
```

Profiles

A *Profile* allows for creating a 1-dimensional profile through the 3-dimensional data. Here we create a (cylindrical) radial profile.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> prof = plonk.load_profile(snap)

>>> prof.available_profiles()
['alpha_shakura_sunyaev',
 'angular_momentum_mag',
 'angular_momentum_phi',
 'angular_momentum_theta',
 'angular_momentum_x',
 'angular_momentum_y',
 'angular_momentum_z',
 'angular_velocity',
 'aspect_ratio',
 'azimuthal_angle',
 'density',
 'disc_viscosity',
 'dust_to_gas_ratio_001',
 'epicyclic_frequency',
 'id',
 'kinetic_energy',
 'mass',
 'midplane_stokes_number_001',
 'momentum_mag',
 'momentum_x',
 'momentum_y',
 'momentum_z',
 'number',
 'polar_angle',
 'position_mag',
 'position_x',
 'position_y',
 'position_z',
 'pressure',
 'radius',
 'radius_cylindrical',
 'radius_spherical',
 'scale_height',
 'size',
 'smoothing_length',
 'sound_speed',
 'specific_angular_momentum_mag',
 'specific_angular_momentum_x',
 'specific_angular_momentum_y',
 'specific_angular_momentum_z',
 'specific_kinetic_energy',
```

(continues on next page)

(continued from previous page)

```
 'stopping_time_001',
 'sub_type',
 'surface_density',
 'temperature',
 'timestep',
 'toomre_q',
 'type',
 'velocity_divergence',
 'velocity_mag',
 'velocity_radial_cylindrical',
 'velocity_radial_spherical',
 'velocity_x',
 'velocity_y',
 'velocity_z']

>>> prof['surface_density']
array([0.12710392, 0.28658185, 0.40671266, 0.51493316, 0.65174709,
       0.82492413, 0.96377964, 1.08945358, 1.18049604, 1.27653871,
       1.32738967, 1.37771242, 1.41116016, 1.42827418, 1.45969001,
       1.46731756, 1.48121301, 1.48415196, 1.48896081, 1.49099377,
       1.49539866, 1.49549864, 1.49946459, 1.48970975, 1.49726806,
       1.49707047, 1.48474985, 1.47849345, 1.45204807, 1.42910354,
       1.39087639, 1.36186174, 1.32811369, 1.31057511, 1.30137812,
       1.28580834, 1.29475762, 1.27265139, 1.2662418 , 1.25830579,
       1.2470909 , 1.24128492, 1.23557015, 1.24083293, 1.25015857,
       1.26132853, 1.28408577, 1.30015172, 1.32080284, 1.325977  ,
       1.33936347, 1.34760897, 1.34222981, 1.34707782, 1.34162702,
       1.33612932, 1.32209663, 1.31135862, 1.29220491, 1.28232641,
       1.26204789, 1.24767264, 1.23697665, 1.21953283, 1.20616179,
       1.18754849, 1.16305682, 1.14546076, 1.10968249, 1.07937633,
       1.0369441 , 0.99232149, 0.94296769, 0.89226746, 0.84172944,
       0.78206348, 0.73299116, 0.67446142, 0.62486291, 0.56701135,
       0.5031995 , 0.44594058, 0.39603015, 0.34398414, 0.29642473,
       0.24606244, 0.20750469, 0.17334624, 0.13960351, 0.10626775,
       0.08377139, 0.06366415, 0.05257149, 0.04586044, 0.03616855,
       0.03122829, 0.02804837, 0.02473014, 0.02287971, 0.02059255]) <Unit('kilogram /␣
↪meter ** 2')>
```

Plot a radial profile.

```
>>> import matplotlib.pyplot as plt

>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> prof = plonk.load_profile(snap)

>>> prof.set_units(position='au', scale_height='au')

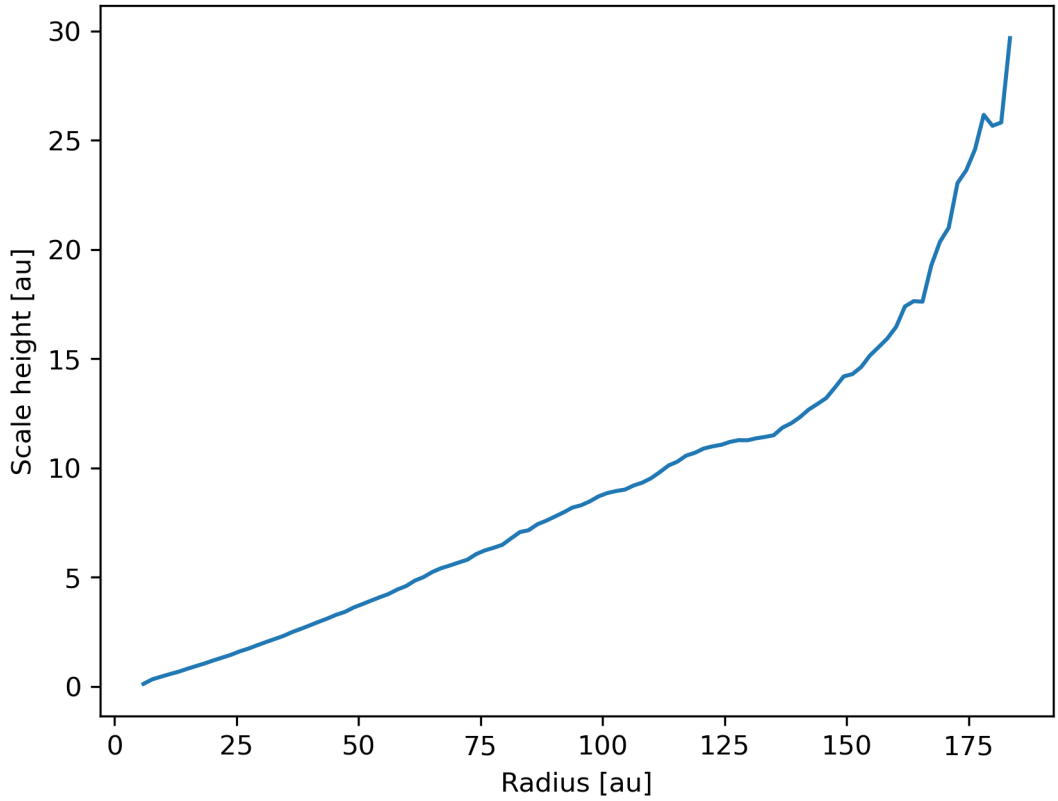
>>> ax = prof.plot('radius', 'scale_height')
```

(continues on next page)

(continued from previous page)

```
>>> ax.set_ylabel('Scale height [au]')

>>> ax.legend().remove()
```



Generate and plot a *Profile* in the z-coordinate with a *SubSnap* of particles by radius.

```
>>> import matplotlib.pyplot as plt

>>> import plonk

>>> from plonk.analysis.filters import annulus

>>> snap = plonk.load_snap('disc_00030.h5')

>>> au = plonk.units('au')

>>> subsnap = annulus(snap=snap, radius_min=50*au, radius_max=55*au, height=100*au)

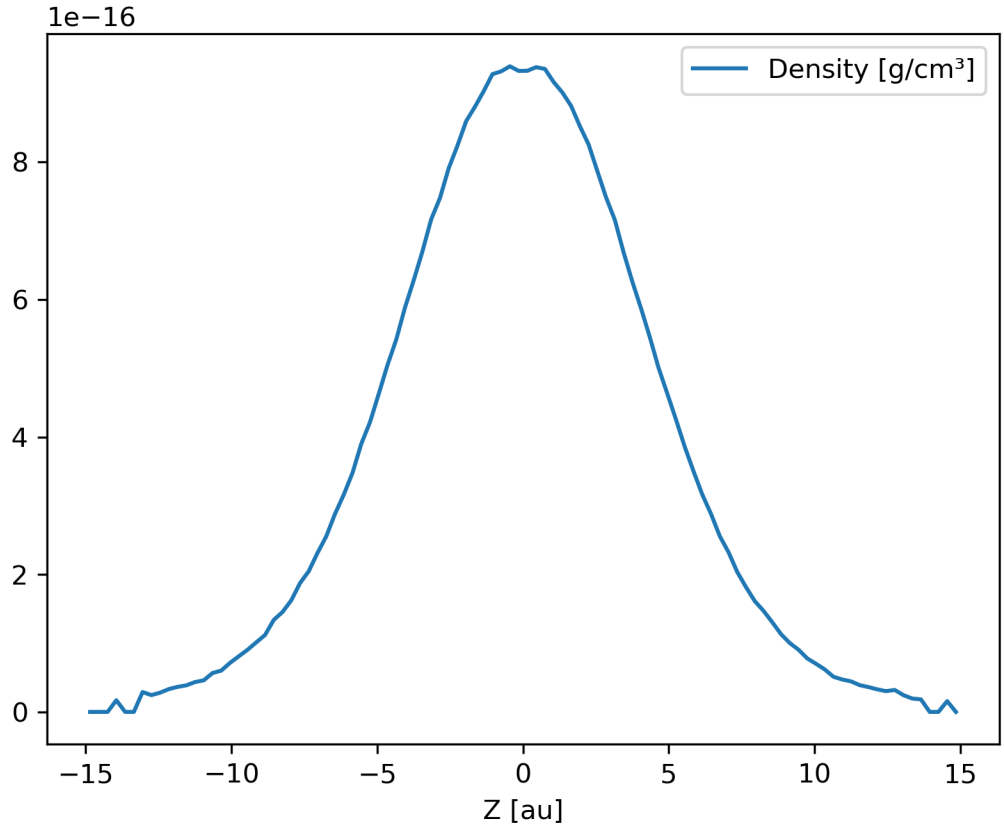
>>> prof = plonk.load_profile(
...     subsnap,
...     ndim=1,
...     coordinate='z',
...     cmin='-15 au',
...     cmax='15 au',
... )
```

(continues on next page)

(continued from previous page)

```
>>> prof.set_units(position='au', density='g/cm^3')

>>> ax = prof.plot('z', 'density')
```



Neighbours

Find particle neighbours using *neighbours()* via a k-d tree. The implementation uses the efficient scipy.spatial.cKDTree.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap.tree
<scipy.spatial.ckdtree.cKDTree at 0x7f93b2ebe5d0>

# Set the kernel
>>> snap.set_kernel('cubic')
<plonk.Snap "disc_00030.h5">

# Find the neighbours of the first three particles
>>> snap.neighbours([0, 1, 2])
array([list([0, 11577, 39358, 65206, 100541, 156172, 175668, 242733, 245164, 299982,␣
↪308097, 330616, 341793, 346033, 394169, 394989, 403486, 434384, 536961, 537992, 543304,
↪ 544019, 572776, 642232, 718644, 739509, 783943, 788523, 790235, 866558, 870590,␣
↪876347, 909455, 933386, 960933]),
```

(continues on next page)

(continued from previous page)

```
       list([1, 13443, 40675, 44855, 45625, 46153, 49470, 53913, 86793, 91142, 129970,␣
↪142137, 153870, 163901, 185811, 199424, 242146, 266164, 268662, 381989, 433794, 434044,
↪ 480663, 517156, 563684, 569709, 619541, 687099, 705301, 753942, 830461, 884950,␣
↪930245, 949838]),
       list([2, 7825, 22380, 30099, 36164, 65962, 67630, 70636, 82278, 88742, 127335,␣
↪145738, 158511, 171438, 248893, 274204, 274313, 282427, 388144, 436499, 444614, 534555,
↪ 561393, 599283, 712841, 790972, 813445, 824461, 853507, 912956, 982408, 986423,␣
↪1046879, 1092432])],
      dtype=object)
```

Get neighbours of one type only by first creating a *SubSnap*. Note that the returned indices are relative to type.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap.set_kernel('cubic')

>>> dust = snap.family('dust', squeeze=True)

# Find the neighbours of the first three dust particles
>>> dust.neighbours([0, 1, 2])
array([list([0, 10393, 14286, 19847, 20994, 25954, 33980, 52721, 59880, 66455, 70031,␣
↪75505, 75818, 82947, 93074, 93283, 95295]),
       list([1, 3663, 6676, 8101, 10992, 13358, 15606, 20680, 28851, 35049, 35791, 36077,
↪ 39682, 48589, 49955, 52152, 66884, 84440, 88573, 96170, 97200]),
       list([2, 6926, 9685, 12969, 15843, 31285, 34642, 45735, 47433, 53258, 54329,␣
↪56565, 58468, 63105, 63111, 63619, 67517, 70483, 73277, 74340, 78988, 81391, 83534,␣
↪83827, 85351, 86117, 94404, 97329])],
      dtype=object)
```

Visualization

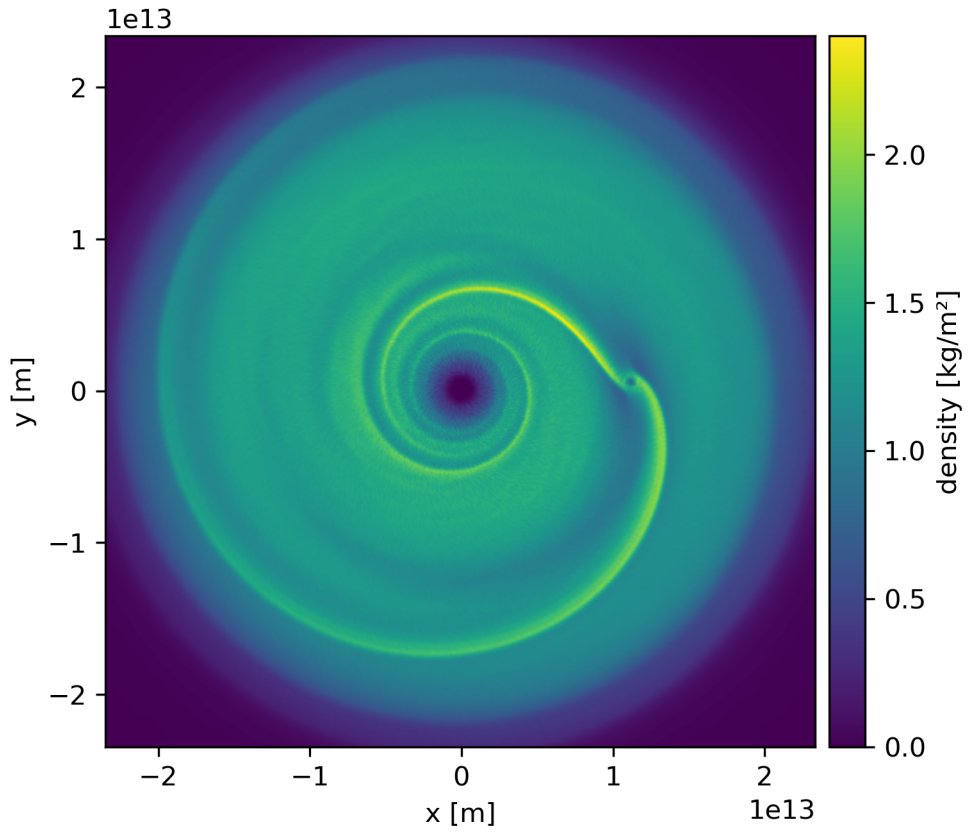
Projection plot

Produce a projection *image()* plot of density.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap.image(quantity='density')
```



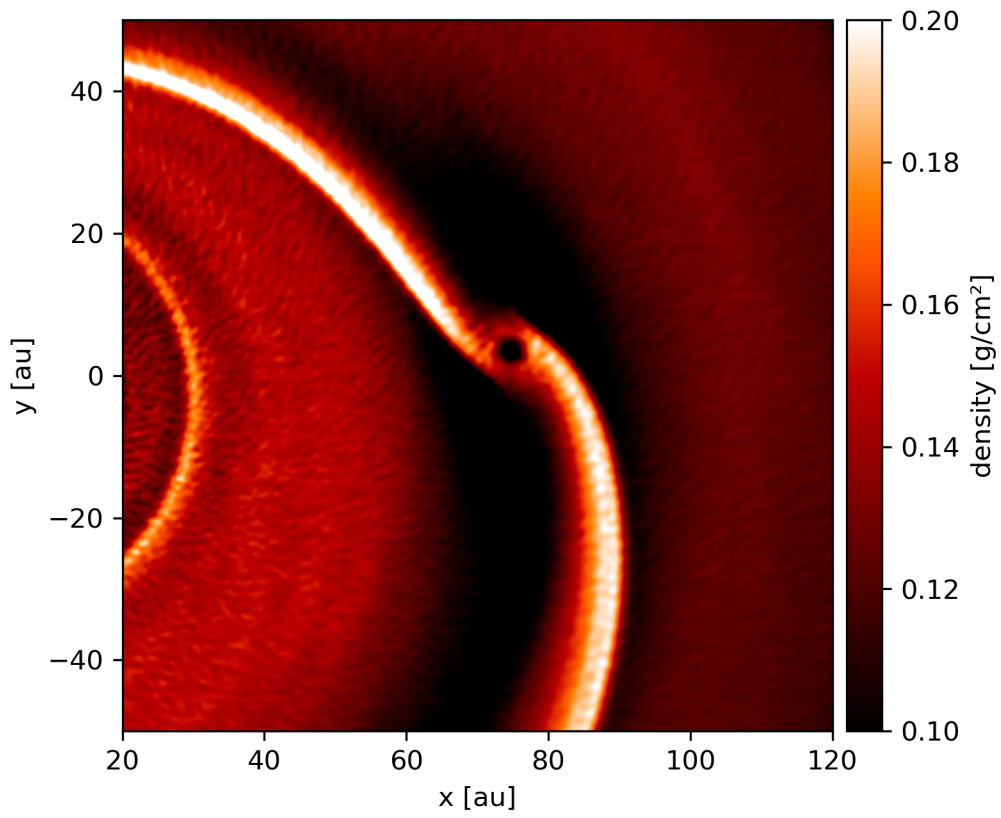
Set plot units, extent, colormap, and colorbar range.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> units = {'position': 'au', 'density': 'g/cm^3', 'projection': 'cm'}

>>> snap.image(
...     quantity='density',
...     extent=(20, 120, -50, 50),
...     units=units,
...     cmap='gist_heat',
...     vmin=0.1,
...     vmax=0.2,
... )
```



You can set the units on the *Snap* instead of passing into the *image()* method.

```
>>> snap.set_units(position='au', density='g/cm^3', projection='cm')

>>> snap.image(
...     quantity='density',
...     extent=(20, 120, -50, 50),
...     cmap='gist_heat',
...     vmin=0.1,
...     vmax=0.2,
... )
```

Cross-section plot

Produce a cross-section *image()* plot of density.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

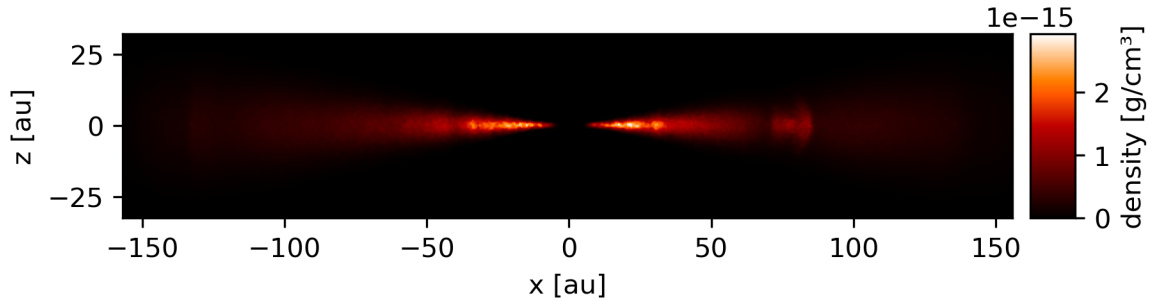
>>> snap.set_units(position='au', density='g/cm^3')

>>> snap.image(
...     quantity='density',
...     x='x',
```

(continues on next page)

(continued from previous page)

```
...     y='z',
...     interp='slice',
...     cmap='gist_heat',
... )
```



Particle plot

Produce a plot of the particles using *plot()* with z-coordinate on the x-axis and smoothing length on the y-axis.

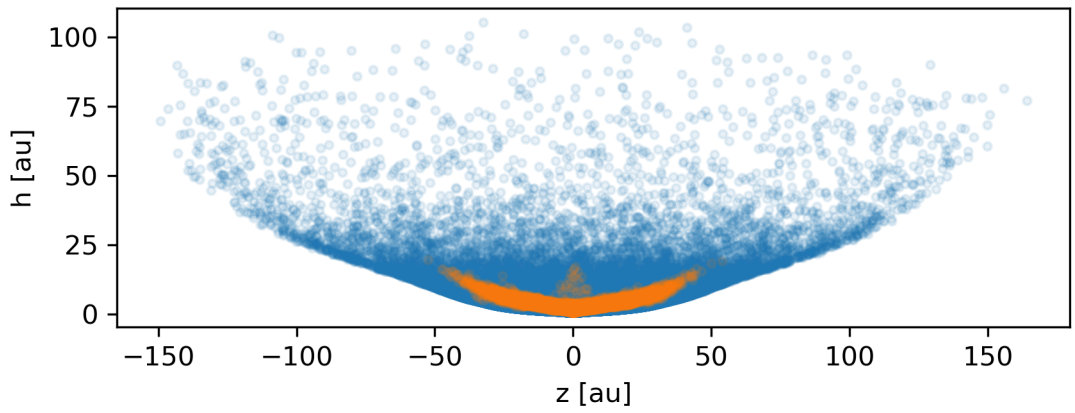
The different colours refer to different particle types.

```
>>> import plonk

>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap.set_units(position='au', smoothing_length='au')

>>> snap.plot(x='z', y='h', alpha=0.1)
```



Plot particles with color representing density.

```
>>> import plonk

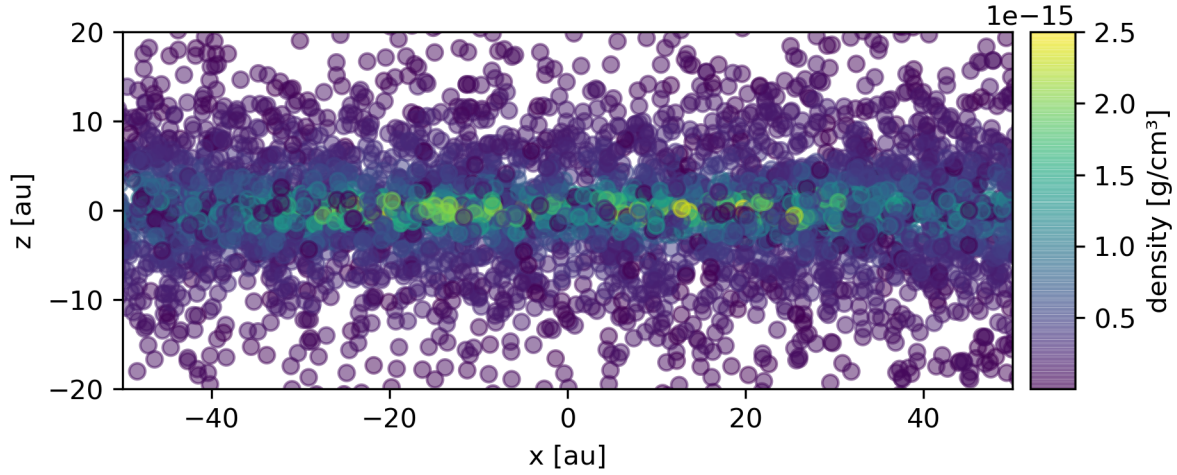
>>> snap = plonk.load_snap('disc_00030.h5')

>>> snap.set_units(position='au', density='g/cm^3')
```

(continues on next page)

(continued from previous page)

```
>>> ax = snap.plot(
...     x='x',
...     y='z',
...     c='density',
...     xlim=(-50, 50),
...     ylim=(-20, 20),
... )
```



Animations

Produce an animation of images using *animate()*.

```
>>> import plonk

>>> sim = plonk.load_simulation(prefix='disc')

>>> units={'position': 'au', 'density': 'g/cm^3', 'projection': 'cm'}

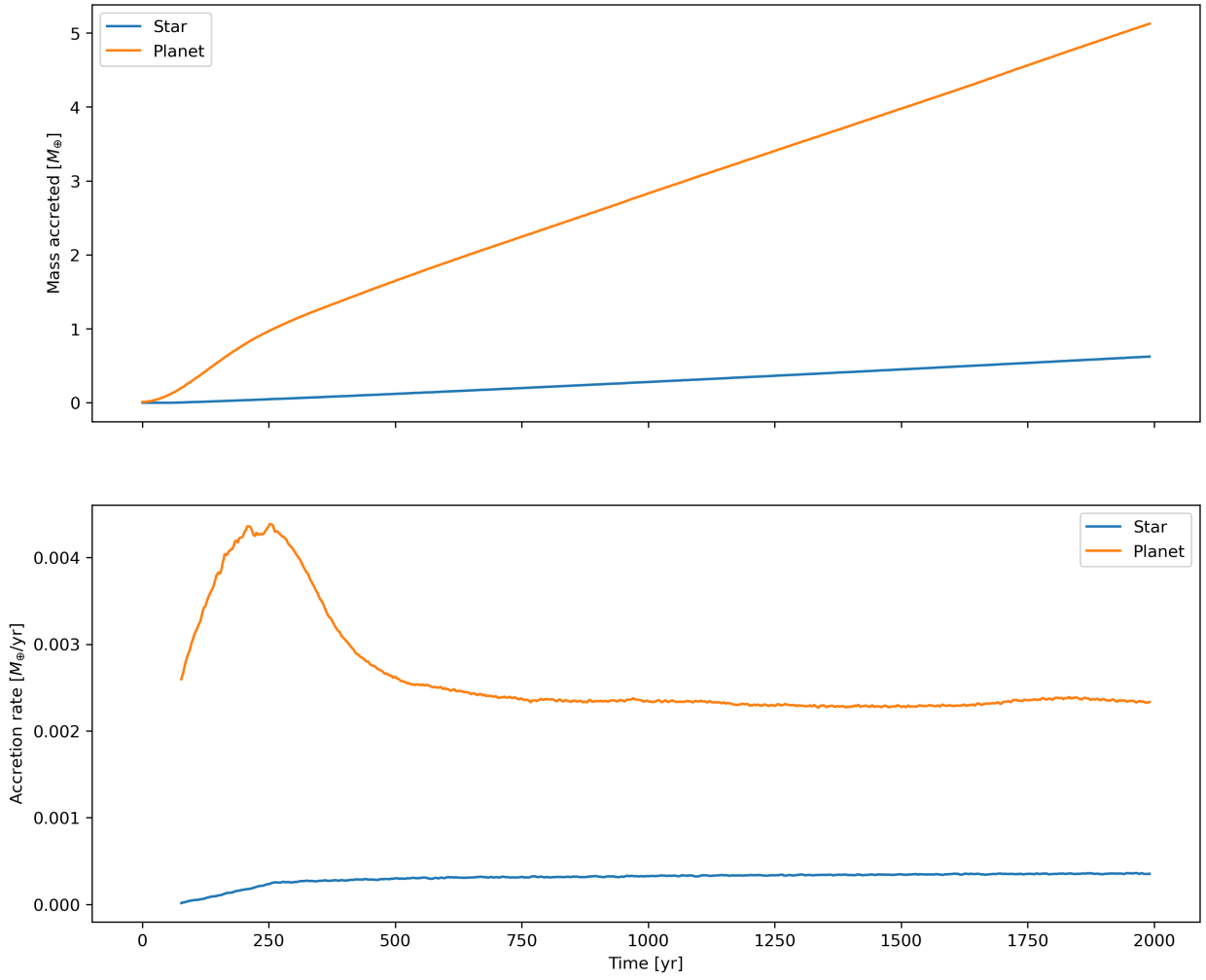
>>> plonk.animate(
...     filename='animation.mp4',
...     snaps=sim.snaps,
...     quantity='density',
...     extent=(-160, 160, -160, 160),
...     units=units,
...     adaptive_colorbar=False,
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

2.2.2 Examples

Here is a collection of examples using Plonk for analysis and visualization.

Accretion onto sinks

Plot mass accretion and accretion rate onto sink particles.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import matplotlib.pyplot as plt
import numpy as np
import plonk

# Load simulation
sim = plonk.load_simulation(prefix='disc')
sink_labels = ('Star', 'Planet')
```

(continues on next page)

(continued from previous page)

```
# Initialize figure
fig, ax = plt.subplots(ncols=1, nrows=2, sharex=True, figsize=(12, 10))

# Loop over sinks and plot
for idx, sink in enumerate(sim.time_series['sinks']):

    # Time in years
    sink['time [year]'] = (sink['time [s]'].to_numpy() * plonk.units('s')).to('year').m

    # Mass accreted in Earth mass
    sink['mass_accreted [earth_mass]'] = (
        (sink['mass_accreted [kg]'].to_numpy() * plonk.units('kg'))
        .to('earth_mass')
        .magnitude
    )

    # Calculate accretion rate
    sink['accretion_rate [kg / s]'] = np.gradient(
        sink['mass_accreted [kg]'], sink['time [s]']
    )

    # Take rolling average
    accretion_rate = (
        sink['accretion_rate [kg / s]'].rolling(window=100).mean().to_numpy()
    )

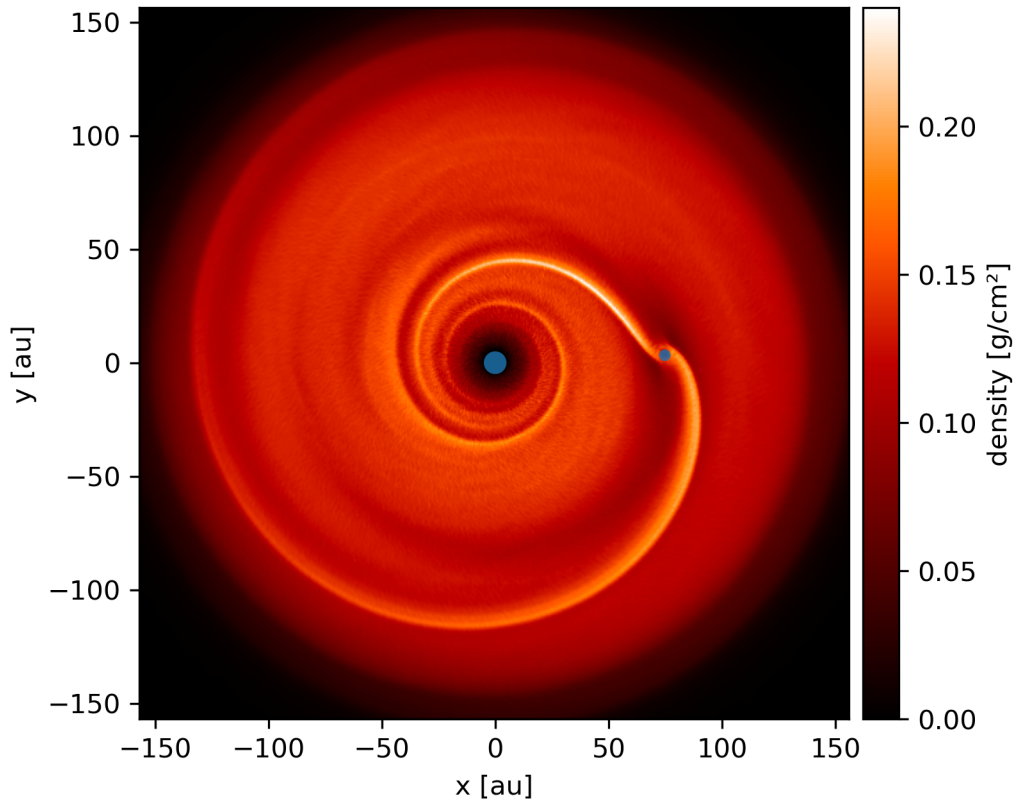
    # Convert to Earth mass / year
    sink['accretion_rate [earth_mass / year]'] = (
        (accretion_rate * plonk.units('kg/s')).to('earth_mass / year').magnitude
    )

    # Plot
    sink.plot(
        'time [year]',
        'mass_accreted [earth_mass]',
        ax=ax[0],
        label=f'{sink_labels[idx]}',
    )
    sink.plot(
        'time [year]',
        'accretion_rate [earth_mass / year]',
        ax=ax[1],
        label=f'{sink_labels[idx]}',
    )

# Set plot labels
ax[0].set_ylabel('Mass accreted [$M_{\oplus}$]')
ax[1].set_xlabel('Time [yr]')
ax[1].set_ylabel('Accretion rate [$M_{\oplus}$/yr]')
```

Accretion radius

Plot the accretion radius on the sink particles.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import plonk
from plonk.utils.visualize import plot_smoothing_length

snap = plonk.load_snap('disc_00030.h5')

# Here "..." means take all sinks
indices = ...

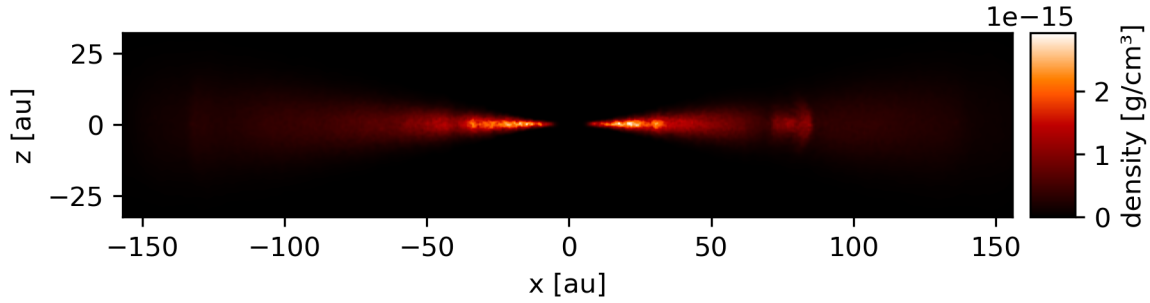
snap.set_units(position='au', density='g/cm^3', projection='cm')

ax = snap.image('density', cmap='gist_heat')

plot_smoothing_length(snap=snap.sinks, indices=indices, alpha=0.8, ax=ax)
```

Cross section

Plot cross section at z=0.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import plonk

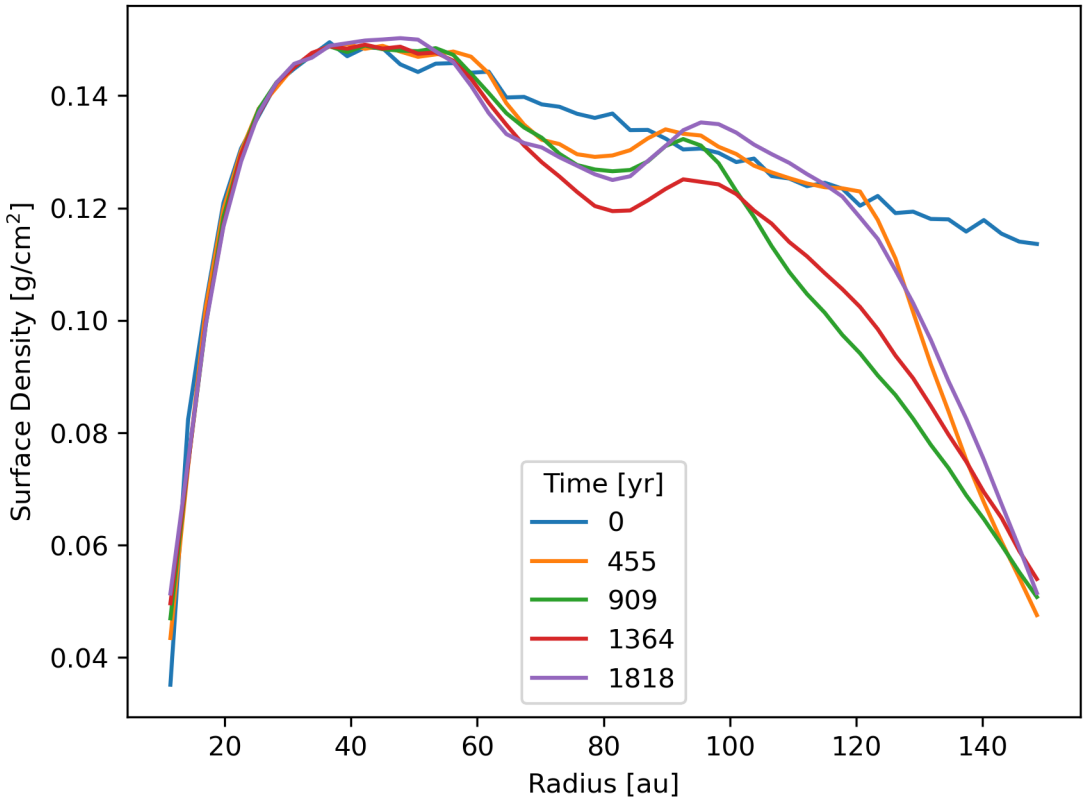
snap = plonk.load_snap('disc_00030.h5')

snap.set_units(position='au', density='g/cm^3', projection='cm')

snap.image(
    quantity='density',
    x='x',
    y='z',
    interp='slice',
    cmap='gist_heat',
)
```

Density profiles

Plot a density profile for multiple snapshots.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import matplotlib.pyplot as plt
import numpy as np
import plonk

# Load simulation
sim = plonk.load_simulation(prefix='disc')

# Generate density profiles for every 7th snap
stride = 7
times = sim.properties['time'].to('year')[::stride]
profiles = list()
for snap in sim.snaps[::stride]:
    profile = plonk.load_profile(snap, cmin='10 au', cmax='150 au', n_bins=50)
    profiles.append(profile)

# Plot profiles
fig, ax = plt.subplots()
units = {'position': 'au', 'surface_density': 'g/cm^2'}
for time, profile in zip(times, profiles):
    label = f'{time.m:.0f}'
    profile.plot(
        'radius', 'surface_density', units=units, label=label, ax=ax
```

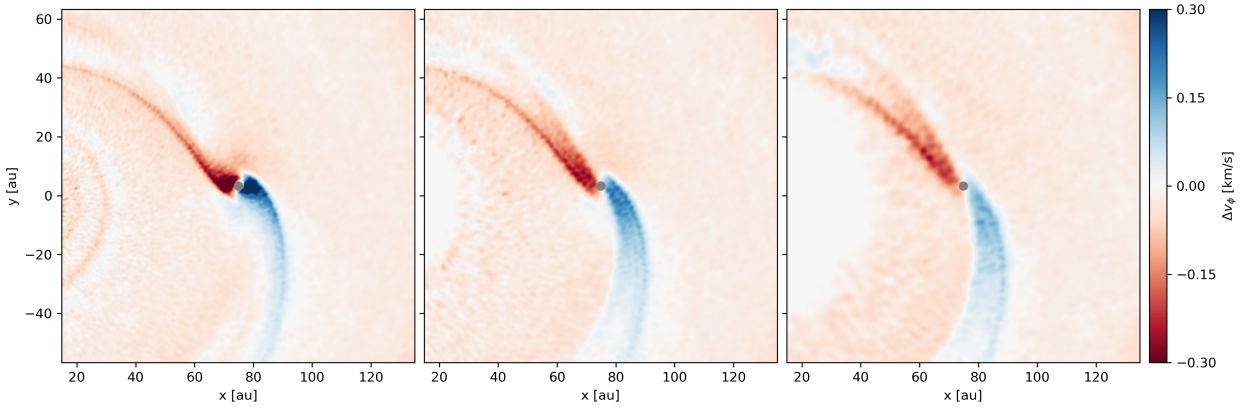
(continues on next page)

(continued from previous page)

```
    )
ax.set_ylabel('Surface Density [g/cm${}^2$]')
ax.legend(title='Time [yr]', loc='best')
```

Deviation from Keplerian

Plot deviation from Keplerian velocity around planet.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import matplotlib.pyplot as plt
import numpy as np
import plonk
from mpl_toolkits.axes_grid1 import AxesGrid

au = plonk.units('au')
km_s = plonk.units('km/s')

# Index of sink particles
star_index = 0
planet_index = 1

# Altitudes for slices
z_slices = (0.0, 5.0, 10.0) * au

# Maximum velocity (km/s)
velocity_max = 0.3 * km_s

# Width for plot
window_width = 120 * au

# Load data
snap = plonk.load_snap('disc_00030.h5')

# Set default units
```

(continues on next page)

(continued from previous page)

```
snap.set_units(position='au', velocity='km/s')

# Star and planet
star = snap.sinks[star_index]
planet = snap.sinks[planet_index]

# Set window around planet for plot
extent = (
    planet['x'] - window_width / 2,
    planet['x'] + window_width / 2,
    planet['y'] - window_width / 2,
    planet['y'] + window_width / 2,
)


@snap.add_array()
def delta_keplerian(snap):
    """Deviation from Keplerian velocity."""
    G = plonk.units.gravitational_constant
    M_star = star['mass']
    v_k = np.sqrt(G * M_star / snap['R'] ** 3)
    return ((snap['v_phi'] - v_k) * snap['R']).to_base_units()


# Set units for plot
snap.set_units(position='au', delta_keplerian='km/s')

# Generate figure and grid
fig = plt.figure(figsize=(15, 5))
grid = AxesGrid(
    fig,
    111,
    nrows_ncols=(1, 3),
    axes_pad=0.1,
    cbar_mode='single',
    cbar_location='right',
    cbar_pad=0.1,
)

# Focus on deviation from Keplerian of gas
gas = snap['gas']

# Loop over z-slices
for slice_offset, ax in zip(z_slices, grid):
    gas.image(
        quantity='delta_keplerian',
        extent=extent,
        interp='slice',
        slice_offset=slice_offset,
        vmin=-velocity_max,
        vmax=velocity_max,
        cmap='RdBu',
```

(continues on next page)

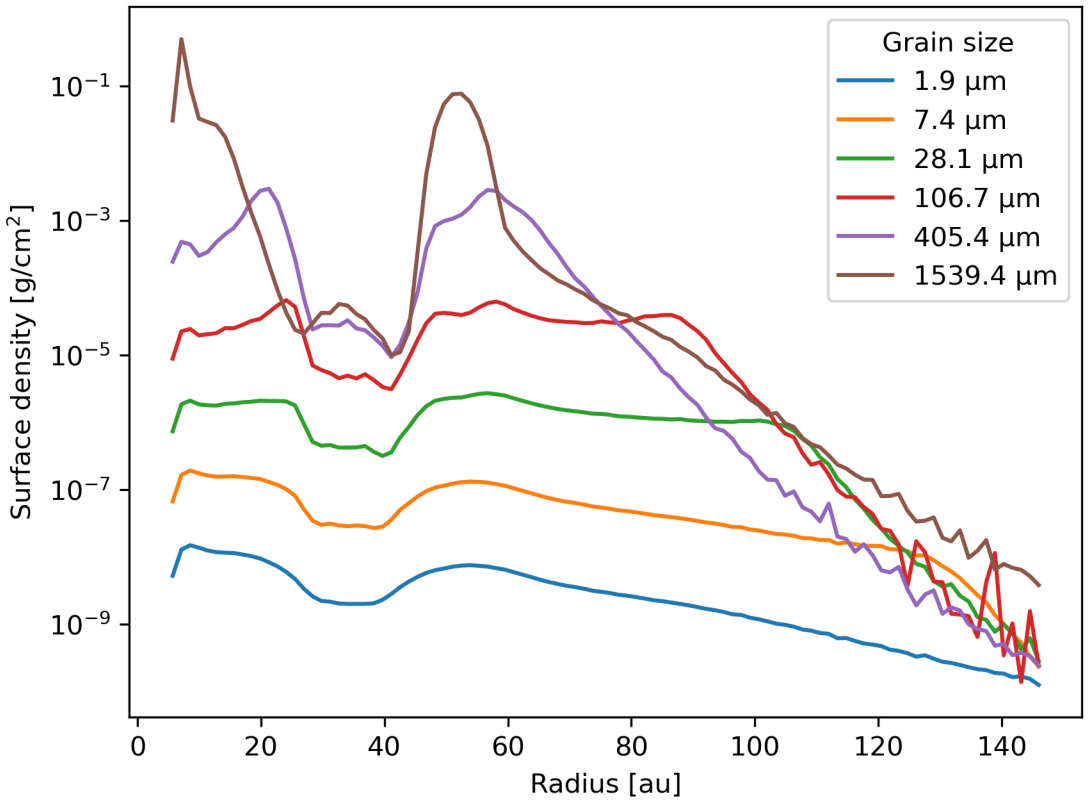
(continued from previous page)

```
        show_colorbar=False,
        ax=ax,
    )
    # Plot planet marker
    ax.plot(planet['x'].to('au').m, planet['y'].to('au').m, 'o', color='gray')

# Add colorbar
cbar = grid.cbar_axes[0].colorbar(ax.images[0])
cbar.set_label_text(r'$\Delta v_{\phi}$ [km/s]')
```

Dust profiles

Plot the surface density profile of each dust species.



Note: The data is from a Phantom simulation with multiple dust species using the mixture (or "1-fluid") method with an embedded planet available from figshare.

```
import plonk

snap = plonk.load_snap('dstau2mj_00130.h5')

prof = plonk.load_profile(snap)
```

(continues on next page)

(continued from previous page)

```
# Set the profile units for plotting
prof.set_units(position='au', dust_surface_density='g/cm^2')

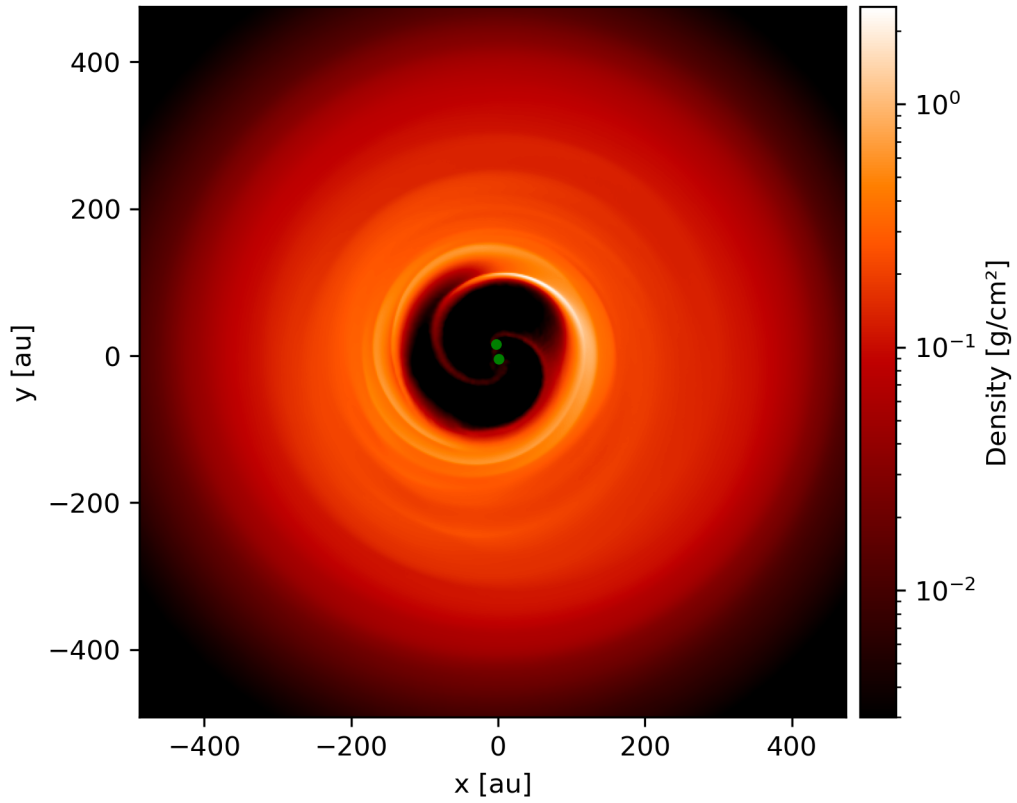
# Get the grain size per species as labels for the plot
labels = [f'{size:.1f~P}' for size in snap.properties['grain_size'].to('micrometer')]

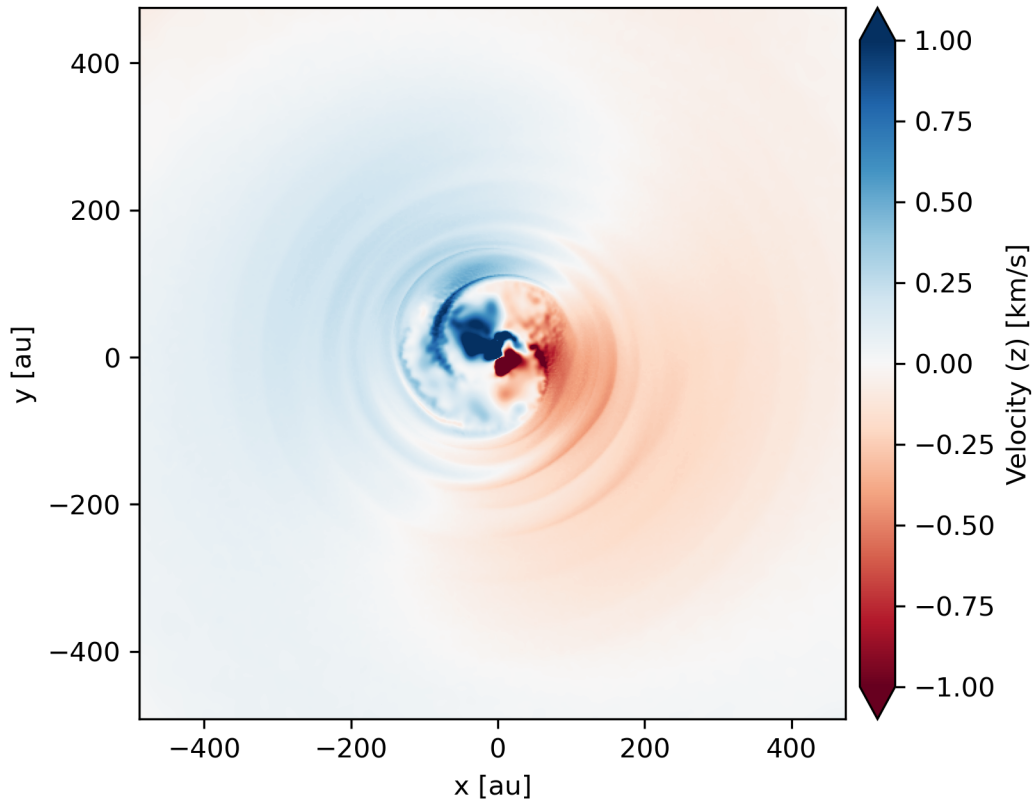
# Set the y-label and set the y-scale to logarithmic
ax_kwargs = {'ylabel': r'Surface density [g/cm${}^2$]', 'yscale': 'log'}

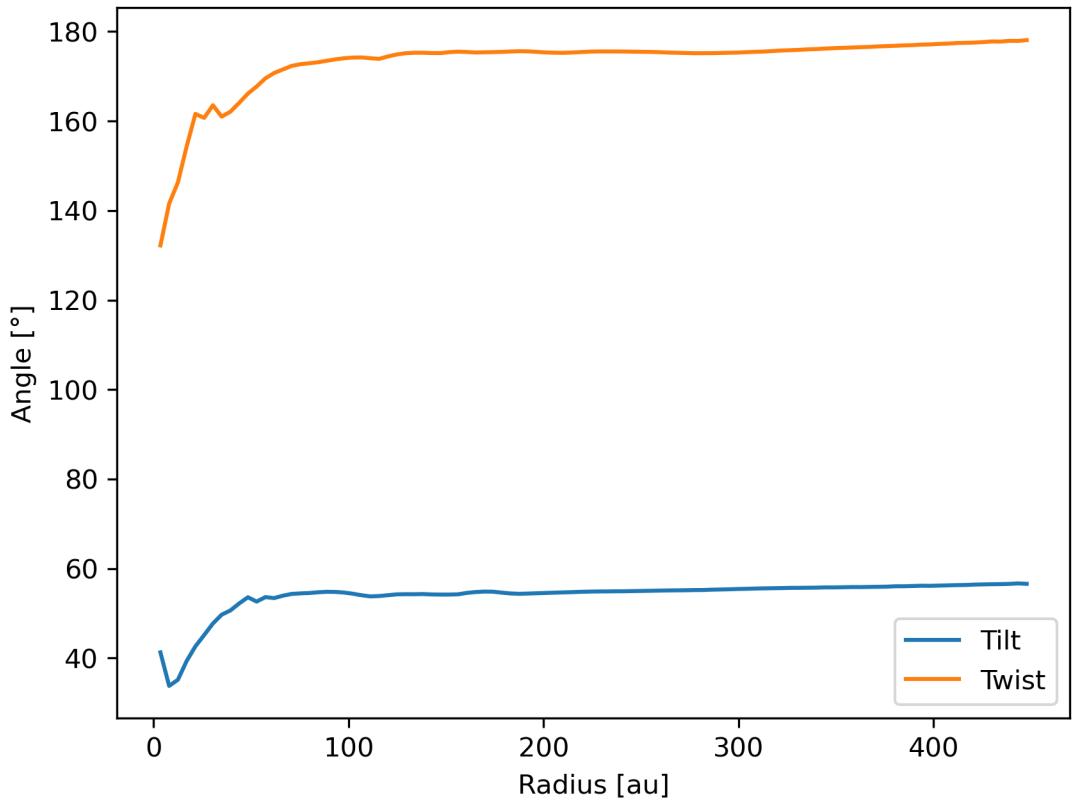
# Make the plot
ax = prof.plot(x='radius', y='dust_surface_density', label=labels, ax_kwargs=ax_kwargs)
ax.legend(title='Grain size')
```

Misaligned binary disc

Plot the column density, kinematics, and tilt and twist of a disc around a binary.




400
200
0
−200
−400
y [au]
−400
−200
0
200
400
x [au]
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Velocity (z) [km/s]


180
160
140
120
100
80
60
40
Angle [°]
0
100
200
300
400
Radius [au]
Tilt
Twist

Note: The data is from a Phantom gas simulation of a disc around a misaligned binary available from figshare.

```
import plonk
from plonk import analysis

# Load snap
snap = plonk.load_snap('disc_00060.h5')
snap.set_units(density='g/cm^3', position='au', projection='cm', velocity='km/s')

# Plot column density with sinks
ax = snap.image(quantity='density', cmap='gist_heat', norm='log', vmin=3e-3)
snap.sinks.plot(color='green', ax=ax)

# Plot kinematics, i.e. z-velocity
ax = snap.image(
    quantity='velocity_z',
    interp='slice',
    cmap='RdBu',
    vmin=-1,
    vmax=1,
    colorbar_kwargs={'extend': 'both'},
)

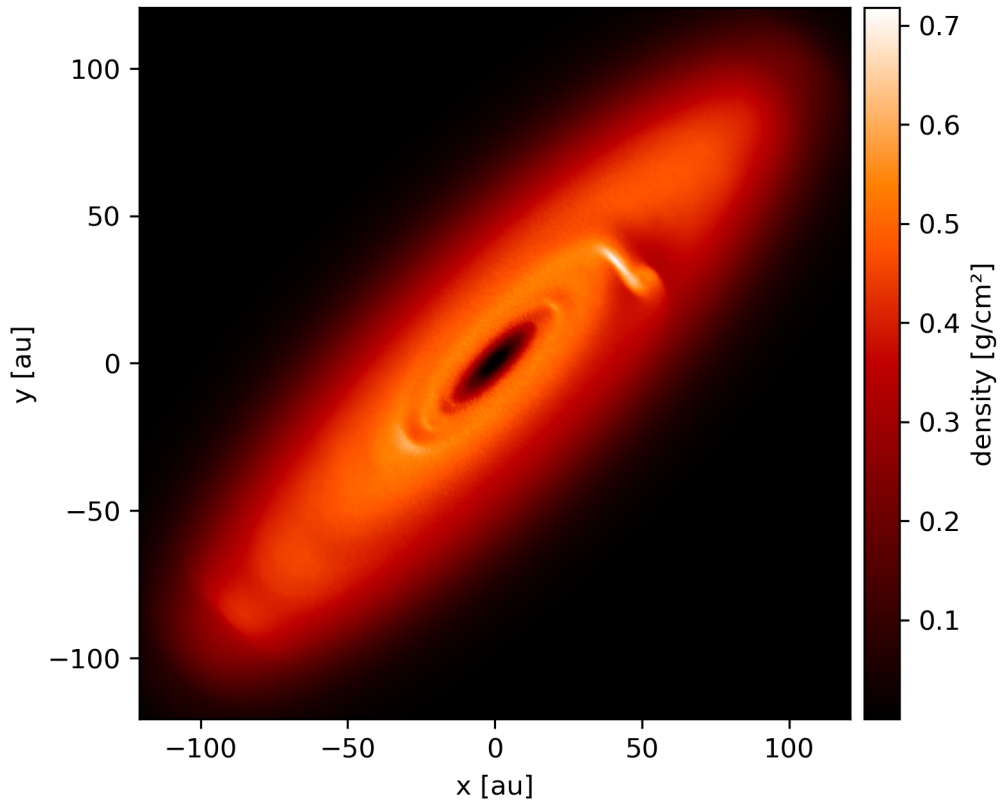
# Rotate to face-on w.r.t. total angular momentum of system
analysis.discs.rotate_face_on(snap=snap, sinks=True)

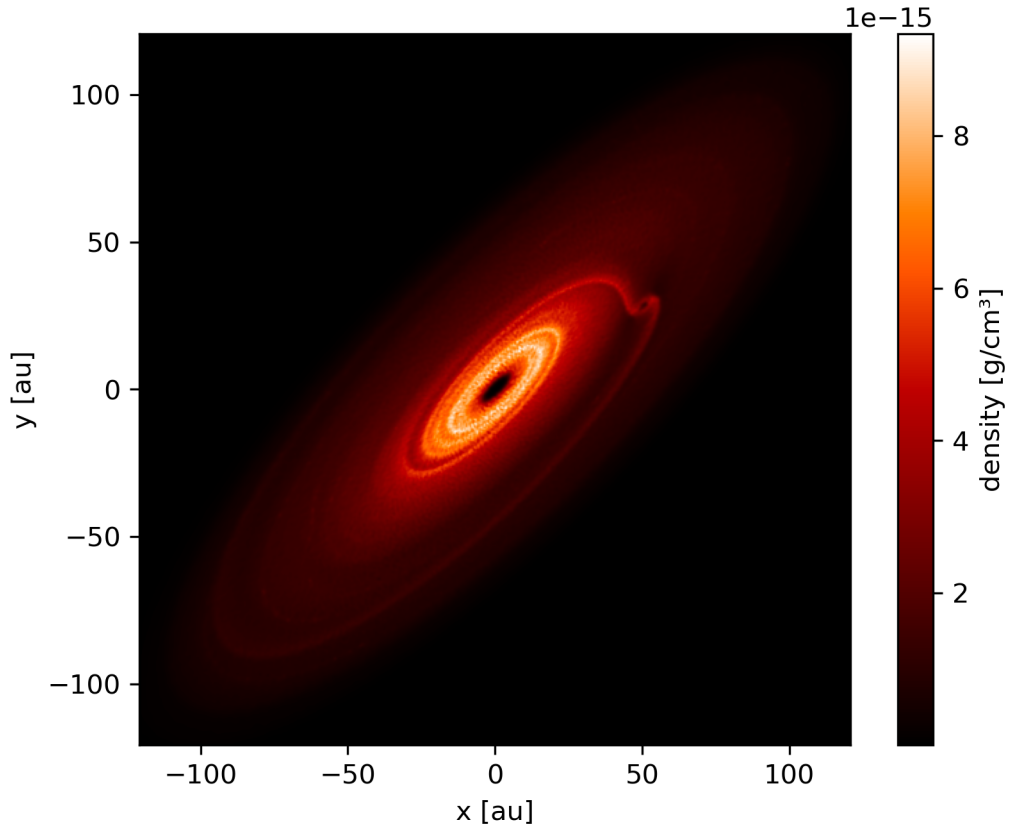
# Generate radial profile in plane perpendicular to total system angular momentum
prof = plonk.load_profile(snap, cmax='450 au')
prof.set_units(
    position='au', angular_momentum_theta='degrees', angular_momentum_phi='degrees'
)

# Plot tilt and twist as radial profile
prof.plot(
    x='radius',
    y=['angular_momentum_theta', 'angular_momentum_phi'],
    label=['Tilt', 'Twist'],
    ax_kwargs={'ylabel': 'Angle [°]'},
)
```

Rotate snapshot

Rotate a snapshot and plot column density and density cross-section in the disc plane.





Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import numpy as np
import plonk
from plonk import analysis

# Load the snapshot
snap = plonk.load_snap('disc_00030.h5')

# Define a rotation axis and angle
angle = np.pi / 2.5
axis = [1, 1, 0]

# Apply the rotation to the snapshot
snap.rotate(axis=axis, angle=angle)

# Plot units
snap.set_units(position='au', density='g/cm^3', projection='cm')

# Plot projection
snap.image(quantity='density', cmap='gist_heat')

# Plot cross-section in the disc plane
```

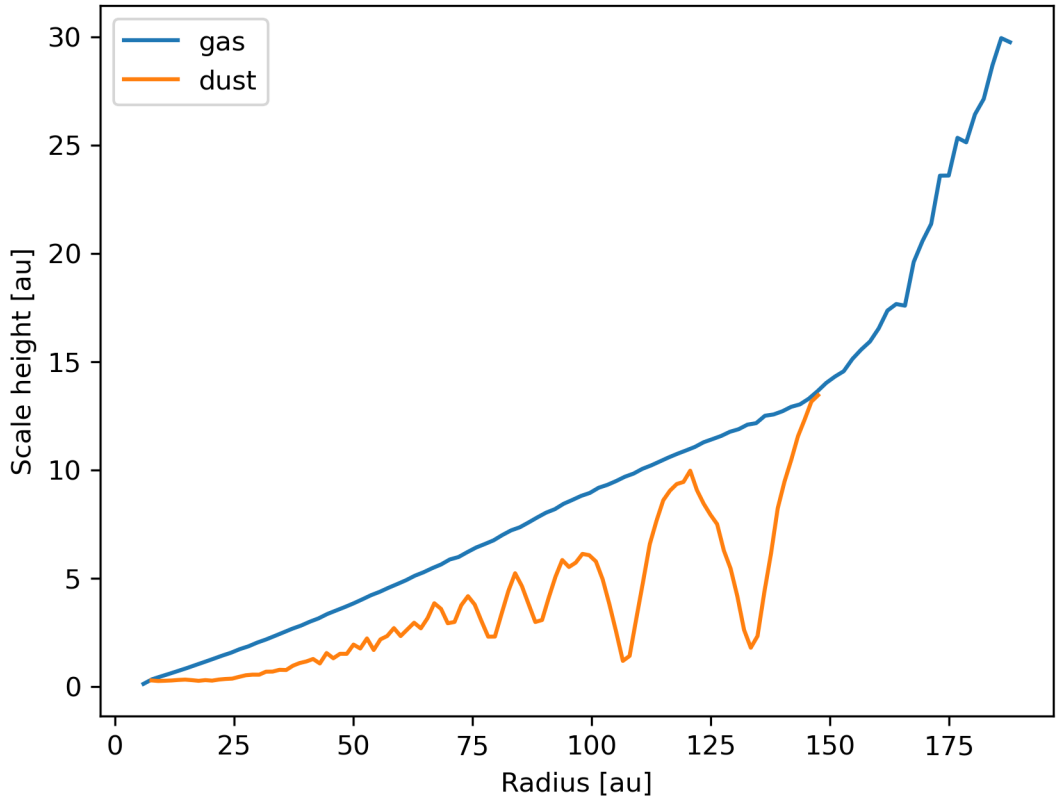
(continues on next page)

(continued from previous page)

```
slice_normal = analysis.discs.unit_normal(snap=snap)
snap.image(
    quantity='density', interp='slice', slice_normal=slice_normal, cmap='gist_heat'
)
```

Scale height of dust and gas

Plot the dust and gas scale heights.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import matplotlib.pyplot as plt
import plonk

snap = plonk.load_snap('disc_00030.h5')

subsnaps = snap.subsnaps_as_dict(squeeze=True)

profs = {family: plonk.load_profile(subsnap) for family, subsnap in subsnaps.items()}

fig, ax = plt.subplots()
```

(continues on next page)

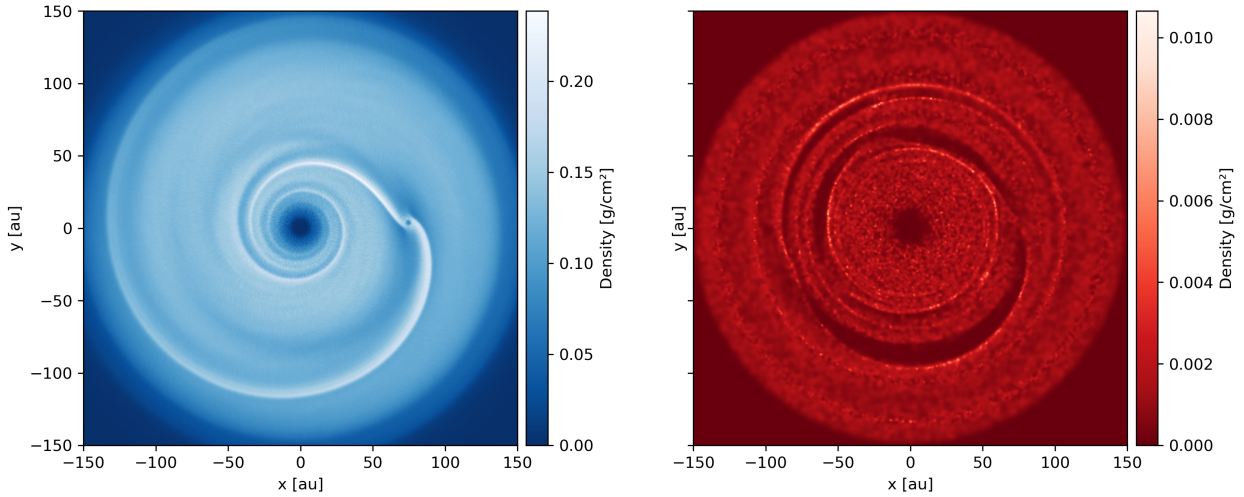
(continued from previous page)

```
for label, prof in profs.items():
    prof.set_units(position='au', scale_height='au')
    prof.plot('radius', 'scale_height', label=label, ax=ax)

ax.set_ylabel('Scale height [au]')
```

Side by side

Plot dust and gas side-by-side.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import matplotlib.pyplot as plt
import plonk

# Load the snapshot
snap = plonk.load_snap('disc_00030.h5')

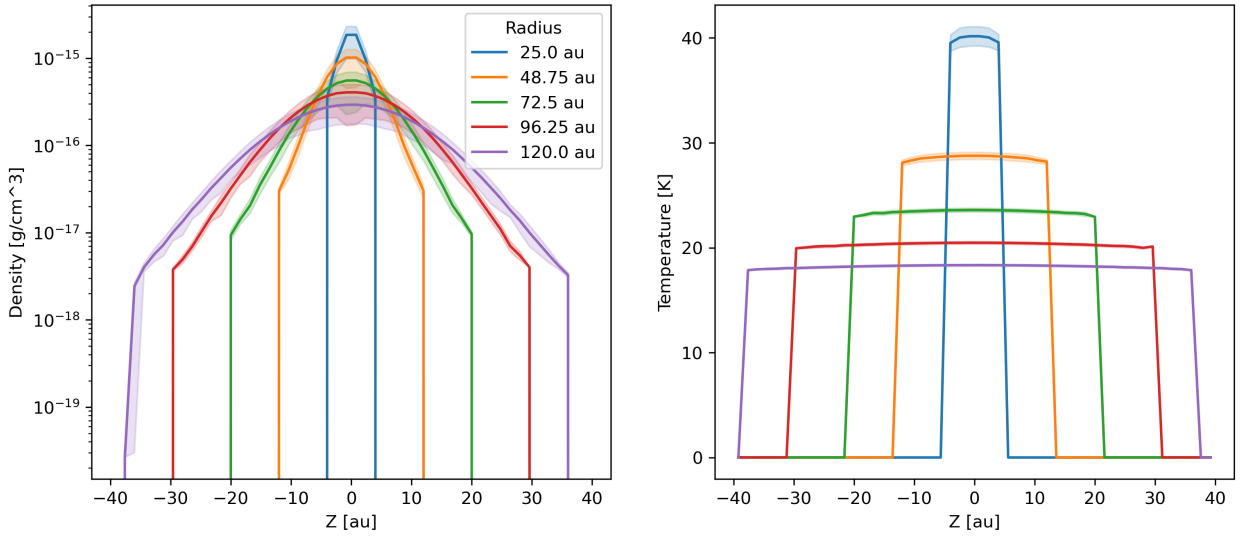
# Set units for plot
snap.set_units(position='au', density='g/cm^3', projection='cm')

# Specify dust and gas subsnaps
gas = snap.family('gas')
dust = snap.family('dust', squeeze=True)
extent = (-150, 150, -150, 150) * plonk.units('au')

# Make plot
fig, axs = plt.subplots(ncols=2, sharey=True, figsize=(13, 5))
gas.image(quantity='density', extent=extent, cmap='Blues_r', ax=axs[0])
dust.image(quantity='density', extent=extent, cmap='Reds_r', ax=axs[1])
```

Vertical profile in a disc

Calculate and plot the density and temperature vertical profiles at multiple radii in a disc.



Note: The data is from the example dataset of a Phantom simulation with a single dust species using the separate particles (or "2-fluid") method with an embedded planet.

```
import matplotlib.pyplot as plt
import numpy as np
import plonk
from plonk import analysis

au = plonk.units('au')

# Load snapshot
snap = plonk.load_snap('disc_00030.h5')

# Set molecular weight for temperature
snap.set_molecular_weight(2.381)

# Choose radii at which to calculate z-profiles and thickness
radius = np.linspace(25, 120, 5) * au
dR = 2.0 * au

# Vertical height
vertical_height = 40 * au

# Plot units
snap.set_units(position='au', density='g/cm^3', temperature='K')

# Make figure and axes
fig, axs = plt.subplots(ncols=2, figsize=(12, 5))

# Loop over radius
```

(continues on next page)

(continued from previous page)

```
for R in radius:

    # Generate an annulus SubSnap
    subsnap = analysis.filters.annulus(
        snap=snap,
        radius_min=R - dR,
        radius_max=R + dR,
        height=1000 * au,
    )

    # Create vertical profile
    prof = plonk.load_profile(
        subsnap,
        ndim=1,
        coordinate='z',
        cmin=-vertical_height,
        cmax=vertical_height,
        n_bins=50,
    )

    # Plot density
    ax_kwargs = {
        'xlabel': 'Altitude [au]',
        'ylabel': r'Density [g/cm${}^3$]',
        'yscale': 'log'
    }
    prof.plot(
        x='z',
        y='density',
        std='shading',
        label=f'{R:~P}',
        ax_kwargs=ax_kwargs,
        ax=axs[0],
    )

    # Plot temperature
    ax_kwargs = {
        'xlabel': 'Altitude [au]',
        'ylabel': 'Temperature [K]',
        'yscale': 'linear'
    }
    prof.plot(
        x='z',
        y='temperature',
        std='shading',
        label=f'{R:~P}',
        ax_kwargs=ax_kwargs,
        ax=axs[1],
    )

axs[0].legend(title='Radius', loc='upper right')
axs[1].legend().remove()
```

2.3 API reference

Here is the documentation for the Plonk API. It provides documentation of the available functions and classes in Plonk.

The documentation is generated from the function/class docstrings, and so is also available at the Python/IPython REPL or in a Jupyter notebook, e.g. `help(plonk.load_snap)` and `plonk.load_snap?`.

2.3.1 Data

SPH snapshot files are represented by the *Snap* class. This object contains a properties dictionary, particle arrays, which are lazily loaded from file, sink arrays, and additional metadata. There are methods for accessing arrays, sub-sets of particles, plotting, finding particle neighbours, etc.

Simulation is an aggregation of the *Snap* and pandas DataFrames to encapsulate all data within a single SPH simulation. In addition, you can load auxilliary SPH simulation data, such as globally-averaged time series data as pandas DataFrames.

Snap

class plonk.**Snap**

Smoothed particle hydrodynamics Snap object.

Snapshot files contain the state of the simulation at a point in time. Typical minimum data from a smoothed particle hydrodynamics simulation include the particle positions and smoothing length, from which the density field can be reconstructed, as well as the particle type. In addition, the particle velocities are required to restart the simulation.

Other data stored in the snapshot file include equation of state, dust, and magnetic field information, as well as numerical quantities related to time-stepping.

Examples

Use load_snap to generate a Snap object.

```
>>> snap = plonk.load_snap('file_name.h5')
```

To access arrays on the particles.

```
>>> snap['position']
>>> snap['density']
```

To access sink arrays.

```
>>> snap.sinks['position']
>>> snap.sinks['spin']
```

To access a subset of particles as a SubSnap.

```
>>> subsnap = snap[:100]
>>> subsnap = snap[snap['x'] > 0]
>>> subsnap = snap['gas']
```

To set a new array directly.

```
>>> snap['my_r'] = np.sqrt(snap['x'] ** 2 + snap['y'] ** 2)
```

Alternatively, define a function.

```
>>> @snap.add_array()
... def my_radius(snap):
...     return np.hypot(snap['x'], snap['y'])
```

Or, use an existing one function.

```
>>> @snap.add_array()
... def my_R(snap):
...     return plonk.analysis.particles.radial_distance(snap)
```

add_alias(*name*, *alias*)
Add alias to array.

Parameters

- **name** (*str*) – The name of the array.
- **alias** (*str*) – The alias to reference the array.

Return type None

add_array(*vector=False*, *dust=False*)
Decorate function to add array to Snap.

This function decorates a function that returns an array. The name of the function is the string with which to reference the array.

The function being decorated should return a Pint Quantity array, not a unitless numpy array.

Parameters

- **vector** (*bool*) – A bool to represent if the array is a vector in space that should have rotations applied to it. If True the rotation should be applied. If False the rotation cannot be applied. Default is False.
- **dust** (*bool*) – A bool to represent if the array is a dust array, in that it has one column per dust species. Default is False.

Returns The decorator which returns the array.

Return type Callable

add_quantities(*category='common'*)
Make extra quantities available on Snap.

Parameters

- **snap** – The Snap object to add extra quantities to.
- **category** (*str*) – The category from which to add extra quantities. Options include:
  - 'common': adds common quantities appropriate for most simulations, such as angular momentum.
  - 'disc': adds accretion disc quantities, such as eccentricity or Keplerian frequency.

  Default is 'common'.

add_unit(*name*, *unit*)
Add missing code unit to array.

Add code units to an array from file that has not had units set automatically.

Parameters

- **name** (*str*) – The name of the array
- **unit** (*str*) – A unit string representing the array code unit, e.g. '1.234 kg * m / s'.

Return type *plonk.snap.snap.Snap*

array(*name*, *sinks=False*)
Get a particle (or sink) array.

Parameters

- **name** (*str*) – A string representing the name of the particle array.
- **sinks** (*bool*) – Whether or not to reference particle or sinks arrays.

Returns

Return type Quantity

array_code_unit(*arr*)
Get array code units.

Parameters **arr** (*str*) – The string representing the quantity.

Returns The Pint unit quantity, or the float 1.0 if no unit found.

Return type Quantity

array_in_code_units(*name*)
Get array in code units.

Parameters

- **snap** – The Snap or SubSnap.
- **name** (*str*) – The array name.

Returns The array on the particles in code units.

Return type ndarray

available_arrays(*verbose=False*, *aliases=False*)
Return a list of available particle arrays.

Parameters

- **verbose** (*bool*) – Also display suffixed arrays, e.g. 'position_x', 'position_y', etc. Default is False
- **aliases** (*bool*) – If True, return array aliases. Default is False.

Returns A list of names of available arrays.

Return type List

base_array_name(*name*)
Get the base array name from a string.

For example, 'velocity_x' returns 'velocity', 'density' returns 'density', 'dust_fraction_001' returns 'dust_fraction', 'x' returns 'position'.

Parameters **name** (*str*) – The name as a string

Returns The base array name.

Return type str

bulk_load(*arrays=None*)
Load arrays into memory in bulk.

Parameters **arrays** (*Optional[List[str]]*) – A list of arrays to load as strings. If None, then load all available arrays.

Return type *plonk.snap.snap.Snap*

bulk_unload(*arrays=None*)
Un-load arrays from memory in bulk.

Parameters **arrays** (*Optional[List[str]]*) – A list of arrays to load as strings. If None, then unload all loaded arrays.

Return type *plonk.snap.snap.Snap*

property cache_arrays: bool
Cache arrays in memory for faster access.

close_file()
Close access to underlying file.

property code_units: Dict[str, Any]
Snap code units.

context(*cache=True*)
Context manager for caching particle arrays on a Snap.

Caching arrays in memory improves performance by saving slow reads from disc. Caching arrays is turned on by default on a Snap. However, it can be useful to turn it off for low memory requirements, such as reading from many snapshots. In this case it may be useful to use this context manager to improve performance inside the context without then leaving those arrays in memory afterwards.

Parameters

- **cache** (*bool*) – Turn caching arrays on or off during a context.
- **snap** (Snap) –

Examples

Temporarily turn on caching arrays.

Caching off >>> snap.cache_arrays False

No loaded arrays >>> snap.loaded_arrays() ()

Produce an image which loads arrays >>> with snap.context(cache=True): ... ax = snap.image('density')

No loaded arrays >>> snap.loaded_arrays() ()

property default_units: Dict[str, Any]
Snap default units.

family(*name*, *squeeze=False*)
Get a SubSnap of a particle family by name.

Parameters

- **name** (*str*) – A string representing the name of the family of particles, e.g. 'gas'.

- **squeeze** (*bool*) – Squeeze sub-types. If False and the particle family has sub-types then return a list of SubSnaps of each sub-type. Otherwise return a SubSnap with all particle of that type.

Returns

Return type *SubSnap* or List[*SubSnap*]

image(*quantity*, *, *x='x'*, *y='y'*, *interp='projection'*, *weighted=False*, *slice_normal=None*, *slice_offset=None*, *extent=None*, *units=None*, *ax=None*, *ax_kwargs={}*, *colorbar_kwargs={}*, ***kwargs*)

Visualize scalar SPH data as an image.

Visualize scalar smoothed particle hydrodynamics data by interpolation to a pixel grid.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **quantity** (*str*) – The quantity to visualize. Must be a string to pass to Snap.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **interp** (*str*) – The interpolation type. Default is 'projection'.
  - 'projection' : 2d interpolation via projection to xy-plane
  - 'slice' : 3d interpolation via cross-section slice.
- **weighted** (*bool*) – Whether to density weight the interpolation or not. Default is False.
- **slice_normal** (*Tuple[float, float, float]*) – The normal vector to the plane in which to take the cross-section slice as a tuple (x, y, z). Default is (0, 0, 1).
- **slice_offset** (*Union[Quantity, float]*) – The offset of the cross-section slice. Default is 0.0.
- **extent** (*Quantity*) – The range in the x and y-coord as (xmin, xmax, ymin, ymax) where xmin, etc. can be floats or quantities with units of length. The default is to set the extent to a box of size such that 99% of particles are contained within.
- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density. There is a special key 'projection' that corresponds to the length unit in the direction of projection for projected interpolation plots.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- **colorbar_kwargs** – Keyword arguments to pass to matplotlib Colorbar.
- ****kwargs** – Additional keyword arguments to pass to interpolation and matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Notes

Additional parameters passed as keyword arguments will be passed to lower level functions as required. E.g. Plonk uses matplotlib's imshow for a image plot, so additional arguments to imshow can be passed this way.

See below for additional parameters for interpolation, colorbars, etc. All other keyword arguments are passed to the appropriate matplotlib function.

Parameters

- **num_pixels** (*tuple*) – The number of pixels to interpolate particle quantities to as a tuple (nx, ny). Default is (512, 512).
- **show_colorbar** (*bool*) – Whether or not to display a colorbar. Default is True.
- **snap** (*SnapLike*) –
- **quantity** (*str*) –
- **x** (*str*) –
- **y** (*str*) –
- **interp** (*str*) –
- **weighted** (*bool*) –
- **slice_normal** (*Tuple[float, float, float]*) –
- **slice_offset** (*Union[Quantity, float]*) –
- **extent** (*Quantity*) –
- **units** (*Dict[str, str]*) –
- **ax** (*Any*) –

Return type Any

Examples

Show an image of the surface density in xy-plane.

```
>>> plonk.image(snap=snap, quantity='density')
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.image(quantity='density')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'density': 'g/cm^3', 'projection': 'cm'}
>>> snap.image(quantity='density', units=units)
```

Show a slice image of the density in xy-plane at z=0.

```
>>> snap.image(quantity='density', interp='slice')
```

load_snap(*filename*, *data_source*, *config=None*)
Load snapshot from file.

Parameters

- **filename** (*Union[str, pathlib.Path]*) – The path to the file.
- **data_source** (*optional*) – The SPH software that produced the data. Default is 'phantom'.
- **config** (*optional*) – The path to a Plonk config.toml file.

loaded_arrays()
Return a list of loaded arrays.

Returns A list of names of loaded particle arrays.

Return type List

neighbours(*indices*)
Get neighbours of particles.

Parameters **indices** (*Union[numpy.ndarray, List[int]]*) – The particle indices.

Returns An array of neighbours lists.

Return type ndarray of list

property num_dust_species: int
Return number of dust species.

property num_particles: int
Return number of particles.

property num_particles_of_type: Dict[str, Any]
Return number of particles per type.

property num_sinks: int
Return number of sinks.

particle_indices(*particle_type*, *squeeze=False*)
Particle indices of a particular type.

Parameters

- **particle_type** (*str*) – The particle type as a string.
- **squeeze** (*bool*) – If True return all subtypes in a single array. Default is False.

Returns If particle has no subtypes then returns an array of indices of that type. Otherwise return a list of arrays of indices, one for each subtype. However, if squeeze is True, return a single array.

Return type ndarray or list of ndarray

plot(*, *x='x'*, *y='y'*, *c=None*, *s=None*, *units=None*, *xlim=None*, *ylim=None*, *ax=None*, *ax_kwargs={}*, *colorbar_kwargs={}*, ***kwargs*)
Visualize SPH data as a particle plot.

Visualize SPH data by plotting the particles, or a subset of the particles, possibly with marker colors and different sizes.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.

- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **c** (*str*) – The quantity to color the particles. Must be a string to pass to Snap.
- **s** (*str*) – The quantity to set the particle size. Must be a string to pass to Snap.
- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density.
- **xlim** (*Quantity*) – The range in the x-coord as (xmin, xmax) where xmin/xmax can be floats or quantities with units of length.
- **ylim** (*Quantity*) – The range in the y-coord as (ymin, ymax) where ymin/ymax can be floats or quantities with units of length.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- **colorbar_kwargs** – Keyword arguments to pass to matplotlib Colorbar.
- ****kwargs** – Additional keyword arguments to pass to matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Examples

Show the particles in xy-plane.

```
>>> plonk.plot(snap=snap)
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.plot()
```

Plot density against x.

```
>>> snap.plot(x='x', y='density')
```

Color particles by density.

```
>>> snap.plot(x='x', y='y', c='density')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'density': 'g/cm^3'}
>>> snap.plot(x='x', y='y', c='density', units=units)
```

property properties: Dict[str, Any]
Snap properties.

read_extra_arrays(*filename=None*)
Read extra arrays from file.

Parameters **filename** (*optional*) – A filename to read from.

Returns The Snap.

Return type *Snap*

reopen_file()
Re-open access to the underlying file.

reset(*arrays=False*, *rotation=True*, *translation=True*)
Reset Snap.

Reset rotation and translations transformations on the Snap to initial (on-file) values. In addition, unload cached arrays.

Parameters

- **arrays** (*bool*) – Set to True to unload arrays from memory. Default is False.
- **rotation** (*bool*) – Set to True to reset rotation. Default is True.
- **translation** (*bool*) – Set to True to reset translation. Default is True.

Returns The reset Snap. Note that the reset operation is in-place.

Return type *Snap*

rotate(*rotation=None*, *axis=None*, *angle=None*)
Rotate snapshot.

The rotation can be defined by a scipy Rotation object, or a combination of a vector, specifying a rotation axis, and an angle, specifying the rotation in radians.

Parameters

- **rotation** (*Optional[scipy.spatial.transform.rotation.Rotation]*) – The rotation as a scipy.spatial.transform.Rotation object.
- **axis** (*Optional[Union[numpy.ndarray, List[float], Tuple[float, float, float]]]*) – An array specifying a rotation axis, like (x, y, z).
- **angle** (*Optional[float]*) – A float specifying the rotation in radians.

Returns The rotated Snap. Note that the rotation operation is in-place.

Return type *Snap*

Examples

Rotate a Snap by /3 around [1, 1, 0].

```
>>> axis = (1, 1, 0)
>>> angle = np.pi / 3
>>> snap.rotate(axis=axis, angle=angle)
```

set_central_body(*sink_idx*)
Set the central body for orbital dynamics.

The central body can be a sink particle or multiple sink particles given by a sink index, or list of sink indices. This method sets snap.properties['central_body'] with the central body mass, barycenter position and velocity. This is required to calculate orbital quantities on the particles such as eccentricity.

Parameters **sink_idx** (*Union[int, List[int]]*) – The sink index or list of indices.

Returns The Snap.

Return type *Snap*

set_kernel(*kernel*)
Set kernel.

Parameters **kernel** (*str*) – The kernel name as a string.

Returns The Snap.

Return type *Snap*

set_molecular_weight(*molecular_weight*)
Set molecular weight.

Set the molecular weight of the gas in gram / mole in snap.properties['molecular_weight']. This is required to calculate temperature.

Parameters **molecular_weight** (*float*) – The molecular weight in units of gram / mole. E.g. Phantom uses 2.381 for molecular hydrogen with solar metallicity.

Returns The Snap.

Return type *Snap*

set_units(***kwargs*)
Set default unit for arrays.

Parameters **kwargs** – Keyword arguments with keys as the array name, e.g. 'pressure', and with values as the unit as a string, e.g. 'pascal'.

Return type *plonk.snap.snap.Snap*

Examples

Set multiple default units.

```
>>> snap.set_units(pressure='pascal', density='g/cm^3')
```

property sinks: *plonk.snap.snap.Sinks*
Sink particle arrays.

subsnaps_as_dict(*squeeze=False*)
Return particle-type subsnaps as a dict of SubSnaps.

Parameters **squeeze** (*optional*) – Squeeze sub-types. For each key, if False and the particle family has sub-types then return a list of SubSnaps of each sub-type. Otherwise return a SubSnap with all particle of that type.

Returns A dict of all SubSnaps.

Return type Dict

subsnaps_as_list(*squeeze=False*)
Return particle-type subsnaps as a list of SubSnaps.

Parameters **squeeze** (*optional*) – Squeeze sub-types. If True then each particle sub-type of the same type will be treated the same.

Returns A list of all SubSnaps.

Return type List[*SubSnap*]

to_dataframe(*columns=None*, *units=None*)
Convert Snap to DataFrame.

Parameters

- **columns** (*optional*) – A list of columns to add to the data frame. Default is None.
- **units** (*optional*) – A list of units corresponding to columns add to the data frame. Units must be strings, and must be base units. I.e. 'cm' not '10 cm'. If None, use default, i.e. cgs. Default is None.

Returns

Return type DataFrame

translate(*translation*, *unit=None*)
Translate snapshot.

I.e. shift the snapshot origin.

Parameters

- **translation** (*Union[pint.quantity.build_quantity_class.<locals>.Quantity, numpy.ndarray]*) – The translation as an array like (x, y, z), optionally with units. If no units are specified you must specify the units parameter below.
- **unit** (*Optional[str]*) – The length unit for the translation. E.g. 'au'.

Returns The translated Snap. Note that the translation operation is in-place.

Return type *Snap*

property tree: scipy.spatial.ckdtree.cKDTree
Particle neighbour kd-tree.

Trees are represented by scipy cKDTree objects.

vector(*quantity*, ***, *x='x'*, *y='y'*, *interp='projection'*, *weighted=False*, *slice_normal=None*, *slice_offset=None*, *extent=None*, *units=None*, *ax=None*, *ax_kwargs={}*, ***kwargs*)
Visualize vector SPH data as a vector plot.

Visualize scalar smoothed particle hydrodynamics data by interpolation to a pixel grid of arrows.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **quantity** (*str*) – The quantity to visualize. Must be a string to pass to Snap.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **interp** (*str*) – The interpolation type. Default is 'projection'.
  - 'projection' : 2d interpolation via projection to xy-plane
  - 'slice' : 3d interpolation via cross-section slice.
- **weighted** (*bool*) – Whether to density weight the interpolation or not. Default is False.
- **slice_normal** (*Tuple[float, float, float]*) – The normal vector to the plane in which to take the cross-section slice as a tuple (x, y, z). Default is (0, 0, 1).
- **slice_offset** (*Union[Quantity, float]*) – The offset of the cross-section slice. Default is 0.0.
- **extent** (*Quantity*) – The range in the x and y-coord as (xmin, xmax, ymin, ymax) where xmin, etc. can be floats or quantities with units of length. The default is to set the extent to a box of size such that 99% of particles are contained within.

- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density. There is a special key 'projection' that corresponds to the length unit in the direction of projection for projected interpolation plots.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- ****kwargs** – Additional keyword arguments to pass to interpolation and matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Notes

Additional parameters passed as keyword arguments will be passed to lower level functions as required.

See below for additional parameters for interpolation, vector properties, etc. All other keyword arguments are passed to the appropriate matplotlib function.

Parameters

- **num_pixels** (*tuple*) – The number of pixels to interpolate particle quantities to as a tuple (nx, ny). Default is (512, 512).
- **number_of_arrows** (*tuple*) – The number of arrows to display by sub-sampling the interpolated data. Default is (25, 25).
- **normalize_vectors** (*bool*) – Whether to normalize the arrows to all have the same length. Default is False.
- **snap** (*SnapLike*) –
- **quantity** (*str*) –
- **x** (*str*) –
- **y** (*str*) –
- **interp** (*str*) –
- **weighted** (*bool*) –
- **slice_normal** (*Tuple[float, float, float]*) –
- **slice_offset** (*Union[Quantity, float]*) –
- **extent** (*Quantity*) –
- **units** (*Dict[str, str]*) –
- **ax** (*Any*) –

Return type Any

Examples

Show a vector plot of velocity in xy-plane.

```
>>> plonk.vector(snap=snap, quantity='velocity')
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.vector(quantity='velocity')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'velocity': 'km/s', 'projection': 'km'}
>>> snap.vector(quantity='velocity', units=units)
```

Show a slice plot of the velocity in xy-plane at z=0.

```
>>> snap.vector(quantity='density', interp='slice')
```

write_extra_arrays(*arrays*, *filename=None*)

Write extra arrays to file.

Parameters

- **arrays** (*List[str]*) – A list of strings with array names.
- **filename** (*optional*) – A filename to write to.

Returns The Snap.

Return type *Snap*

class plonk.**SubSnap**(*base*, *indices*)

A Snap subset of particles.

A SubSnap can be generated from a Snap via an index array, a particle mask, or a string. SubSnaps can be used like a Snap, including: accessing arrays, plotting, finding neighbours, etc.

Parameters

- **base** (Snap) – The base Snap or SubSnap
- **indices** (*Union[ndarray, slice, List[int], int, tuple]*) – A (N,) array of particle indices to include in the SubSnap.

Examples

Generate a SubSnap directly.

```
>>> subsnap = SubSnap(base=snap, indices=[0, 1, 2, 3])
```

You can generate a SubSnap from a Snap object. For example, generate a SubSnap of the gas particles on a Snap.

```
>>> subsnap = snap['gas']
```

Generate a SubSnap of particles with a mask.

```
>>> subsnap = snap[snap['x'] > 0]
```

Generate a SubSnap of particles from indices.

```
>>> subsnap = snap[:100]
>>> subsnap = snap[[0, 9, 99]]
```

property indices: numpy.ndarray
Particle indices.

property sinks: *plonk.snap.snap.Sinks*
Sink particle arrays.

class plonk.**Sinks**(*base*, *indices=None*)
Sink particles in a Snap.

A Sinks object is generated from a Snap.

Parameters

- **base** (Snap) – The base Snap.
- **indices** (*optional*) – Indices to specify a subset of sink particles.

Examples

Generate a Sinks object directly.

```
>>> sinks = Sinks(base=snap)
```

Generate a Sinks object from a Snap object.

```
>>> sinks = snap.sinks
```

Choose a subset of sink particles.

```
>>> sinks = snap.sinks[[0, 1]]
>>> star = snap.sinks[0]
>>> planets = snap.sinks[1:4]
```

array(*name*)
Get an array.

Parameters **name** (*str*) – A string representing the name of the particle array.

Returns

Return type Quantity

available_arrays(*verbose=False*, *aliases=False*)
Return a list of available sink arrays.

Parameters

- **verbose** (*bool*) – Also display suffixed arrays, e.g. 'position_x', 'position_y', etc. Default is False
- **aliases** (*bool*) – If True, return array aliases. Default is False.

Returns A list of names of available arrays.

Return type List

property indices: numpy.ndarray
Sink particle indices.

loaded_arrays()
Return a list of loaded arrays.

Returns A list of names of loaded sink arrays.

Return type List

plot(*, *x='x'*, *y='y'*, *c=None*, *s=None*, *units=None*, *xlim=None*, *ylim=None*, *ax=None*, *ax_kwargs={}*, *colorbar_kwargs={}*, ***kwargs*)
Visualize SPH data as a particle plot.

Visualize SPH data by plotting the particles, or a subset of the particles, possibly with marker colors and different sizes.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **c** (*str*) – The quantity to color the particles. Must be a string to pass to Snap.
- **s** (*str*) – The quantity to set the particle size. Must be a string to pass to Snap.
- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density.
- **xlim** (*Quantity*) – The range in the x-coord as (xmin, xmax) where xmin/xmax can be floats or quantities with units of length.
- **ylim** (*Quantity*) – The range in the y-coord as (ymin, ymax) where ymin/ymax can be floats or quantities with units of length.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- **colorbar_kwargs** – Keyword arguments to pass to matplotlib Colorbar.
- ****kwargs** – Additional keyword arguments to pass to matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Examples

Show the particles in xy-plane.

```
>>> plonk.plot(snap=snap)
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.plot()
```

Plot density against x.

```
>>> snap.plot(x='x', y='density')
```

Color particles by density.

```
>>> snap.plot(x='x', y='y', c='density')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'density': 'g/cm^3'}
>>> snap.plot(x='x', y='y', c='density', units=units)
```

plonk.**load_snap**(*filename*, *data_source='phantom'*, *config=None*)

Load snapshot from file.

Parameters

- **filename** (*Union[str, pathlib.Path]*) – The path to the file.
- **data_source** (*optional*) – The SPH software that produced the data. Default is 'phantom'.
- **config** (*optional*) – The path to a Plonk config.toml file.

Simulation

class plonk.**Simulation**

Smoothed particle hydrodynamics simulation object.

This class aggregates snapshot files, global quantity and sink time series data. Snapshot files contain a complete snapshot of the simulation at a particular time. Other files contain time series of global quantities on particles such as energy and momentum, and time series data for sink particles.

Examples

Reading simulation data into a Simulation object.

```
>>> sim = plonk.load_simulation('prefix', path_to_directory)
```

Accessing the snapshots.

```
>>> sim.snaps
```

Accessing the properties.

```
>>> sim.properties
```

Accessing the global quantity and sink time series data.

```
>>> sim.time_series['global']
>>> sim.time_series['sinks']
```

property code_units: Dict[str, Any]

Units associated with the simulation.

load_simulation(*prefix*, *directory=None*, *data_source='Phantom'*)

Load Simulation.

Parameters

- **prefix** (*str*) – Simulation prefix, e.g. 'disc', if files are named like disc_00000.h5, disc01.ev, discSink0001N01.ev, etc.
- **directory** (*optional*) – Directory containing simulation snapshot files and auxiliary files. Default is None.
- **data_source** (*optional*) – The SPH code used to produce the simulation data. Default is 'Phantom'.

Return type *plonk.simulation.simulation.Simulation*

property properties: Dict[str, Any]
Properties associated with the simulation.

set_units_on_time_series(*config=None*)
Set physical units on time series data.

Parameters **config** (*optional*) – The path to a Plonk config.toml file.

property snaps: List[*Snap*]
List of Snap objects associated with the simulation.

property time_series: pandas.core.frame.DataFrame
Time series data.

to_array(*quantity*, *indices=None*)
Generate an array of a quantity over all snapshots.

Warning: this can be very memory intensive and slow.

Parameters

- **quantity** (*str*) – The quantity as a string, e.g. 'position'.
- **indices** (*Optional[List[int]]*) – You can select a subset of particles by indices corresponding to snap['id'].

Returns

Return type An array with units.

Examples

Get the position of every particle during the whole simulation.

```
>>> pos = sim.to_array(quantity='position')
>>> pos.shape
    (31, 1100000, 3)
```

unset_units_on_time_series(*config=None*)
Un-set physical units on time series data.

Parameters **config** (*optional*) – The path to a Plonk config.toml file.

visualize(*kind*, ***kwargs*)
Visualize a simulation.

Parameters

- **sim** (Simulation) – The Simulation object.
- **kind** (*str*) – The kind of plot: 'particle', 'image', 'vector'.

- ****kwargs** – Keyword arguments to pass to plotting methods such as plonk.image, plonk.plot, and plonk.vector.

Returns

Return type VisualizeSimulation

Examples

Visualize a simulation by density projection images.

```
>>> viz = visualize_sim(sim=sim, kind='image', quantity='density')
```

Alternatively.

```
>>> sim.visualize(kind='image', quantity='density')
```

Go forwards and backwards through snaps.

```
>>> viz.next()
>>> viz.prev()
```

Go to a particular snap, or skip ahead.

```
>>> viz.goto(10)
>>> viz.next(5)
```

plonk.**load_simulation**(*prefix*, *directory=None*, *data_source='Phantom'*)

Load Simulation.

Parameters

- **prefix** (*str*) – Simulation prefix, e.g. 'disc', if files are named like disc_00000.h5, disc01.ev, discSink0001N01.ev, etc.
- **directory** (*optional*) – Directory containing simulation snapshot files and auxiliary files. Default is None.
- **data_source** (*optional*) – The SPH code used to produce the simulation data. Default is 'Phantom'.

Return type *plonk.simulation.simulation.Simulation*

plonk.**load_time_series**(*filenames*, *data_source='Phantom'*, *config=None*)

Load time series data from file(s).

Time series files track global quantities, such as energy, momentum, and density, over time. The time increments in these files is smaller than the snapshot file output time. These files are typically stored as text files.

The data is stored as a pandas DataFrame.

Parameters

- **filename(s)** – Collection of paths to time series file(s) in chronological order. These should all contain the same columns.
- **data_source** (*optional*) – The code used to produce the data. Default is 'Phantom'.
- **config** (*optional*) – The path to a Plonk config.toml file.
- **filenames** (*Union[str, pathlib.Path, Tuple[str], Tuple[pathlib.Path], List[str], List[pathlib.Path]]*) –

Returns A pandas DataFrame with the time series data.

Return type dataframe

Examples

Reading a single time series file into a pandas DataFrame.

```
>>> file_name = 'simulation.ev'
>>> ts = plonk.load_time_series(file_name)
```

Reading a collection of time series files into a pandas DataFrame.

```
>>> file_names = ('sim01.ev', 'sim02.ev', 'sim03.ev')
>>> ts = plonk.load_time_series(file_names)
```

2.3.2 Analysis

Smoothed particle hydrodynamics analysis tools include taking radial (or linear) profiles, filtering particles, calculating derived quantities on particles, calculating global quantities, functions for sinks and discs, and functions for SPH summation and derivatives.

The *Profile* class provides methods for generating 1-dimensional profiles from the 3-dimensional particle data.

Profiles

class plonk.**Profile**(*snap*, *ndim=2*, *cmin=None*, *cmax=None*, *n_bins=100*, *aggregation='average'*, *spacing='linear'*, *coordinate='x'*, *ignore_accreted=True*)

Profiles.

A profile is a binning of particles in Cartesian slices, or cylindrical or spherical shells around the origin, i.e. (0, 0, 0). For cylindrical profiles the cylindrical cells are perpendicular to the xy-plane, i.e. the bins are averaged azimuthally and in the z-direction. Cartesian profiles can be in the x-, y-, and z-directions.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ndim** (*optional*) – The dimension of the profile. For ndim == 2, the radial binning is cylindrical in the xy-plane. For ndim == 3, the radial binning is spherical. For ndim == 1, the radial binning is Cartesian along the x-axis. Default is 2.
- **cmin** (*optional*) – The minimum coordinate for binning. Can be a string, e.g. '10 au', or a quantity with units, e.g. plonk.units('10 au'). Defaults to minimum on the particles.
- **cmax** (*optional*) – The maximum coordinate for binning. Can be a string, e.g. '10 au', or a quantity with units, e.g. plonk.units('10 au'). Defaults to the 99 percentile distance.
- **n_bins** (*optional*) – The number of radial bins. Default is 100.
- **aggregation** (*optional*) – The method to aggregate particle quantities in bins by. Options are 'average', 'mean', or 'median'. Here 'average' is a mass-weighted average. Default is 'average'.
- **spacing** (*optional*) – The spacing of radial bins. Can be 'linear' or 'log'. Default is 'linear'.

- **coordinate** (*optional*) – The coordinate ('x', 'y', or 'z') for Cartesian profiles only, i.e. when ndim==1. Default is 'x'. For cylindrical and spherical profiles the coordinate is 'radius'.
- **ignore_accreted** (*optional*) – Ignore particles accreted onto sinks. Default is True.

Examples

Generate profile from snapshot.

```
>>> prof = plonk.load_profile(snap=snap)
>>> prof = plonk.load_profile(snap=snap, n_bins=300)
>>> prof = plonk.load_profile(snap=snap, cmin='10 au', cmax='300 au')
>>> prof = plonk.load_profile(snap=snap, spacing='log')
```

To access a profile.

```
>>> prof['surface_density']
>>> prof['scale_height']
```

To set a new profile.

```
>>> prof['aspect_ratio'] = prof['scale_height'] / prof['radius']
```

Alternatively use the add_profile decorator.

```
>>> @prof.add_profile
... def mass(prof):
...     M = prof.snap['mass']
...     return prof.particles_to_binned_quantity('sum', M)
```

Plot one or many quantities on the profile.

```
>>> prof.plot('radius', 'density')
>>> prof.plot('radius', ['angular_momentum_x', 'angular_momentum_y'])
```

Plot a quantity on the profile with units.

```
>>> units = {'position': 'au', 'surface_density'='g/cm^2'}
>>> prof.plot('radius', 'surface_density', units=units)
```

add_alias(*name*, *alias*)

Add alias to array.

Parameters

- **name** (*str*) – The name of the array.
- **alias** (*str*) – The alias to reference the array.

Return type None

add_profile(*fn*)

Decorate function to add profile to Profile.

Parameters **fn** (*Callable*) – A function that returns the profile as an array. The name of the function is the string with which to reference the array.

Returns The function which returns the array.

Return type Callable

available_profiles()
Return a listing of available profiles.

Return type List[str]

base_profile_name(*name*)
Get the base profile name from a string.

For example, 'velocity_x' returns 'velocity', 'density' returns 'density', 'dust_fraction_001' returns 'dust_fraction', 'x' returns 'position'.

Parameters **name** (*str*) – The name as a string

Returns The base name.

Return type str

property default_units: Dict[str, Any]
Profile default units.

loaded_profiles()
Return a listing of loaded profiles.

Return type List[str]

particles_to_binned_quantity(*aggregation*, *array*)
Calculate binned quantities from particles.

This takes care of the bin indices and ignoring accreted particles (if requested in instantiating the profile).

Parameters

- **aggregation** (*str*) – The aggregation function that acts on particles in the radial bin.
- **array** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The particle array.

Return type pint.quantity.build_quantity_class.<locals>.Quantity

plot(*x*, *y*, *units=None*, *std=None*, *label=None*, *ax=None*, *ax_kwargs={}*, ***kwargs*)
Plot profile.

Parameters

- **x** (*str*) – The x axis to plot as a string.
- **y** (*Union[str, List[str]]*) – The y axis to plot. Can be string or a list of strings.
- **units** (*Optional[Dict[str, Union[str, List[str]]]]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density.
- **std** (*optional*) – Add standard deviation on profile. Can be 'shading' or 'errorbar'.
- **label** (*optional*) – A label for the plot. Can be a string or a list of strings, one per y.
- **ax** (*optional*) – A matplotlib Axes object to plot to.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- ****kwargs** – Keyword arguments to pass to Axes plot method.

Returns The matplotlib Axes object.

Return type ax

set_units(***kwargs*)
Set default unit for profiles.

Parameters **kwargs** – Keyword arguments with keys as the profile name, e.g. 'pressure', and with values as the unit as a string, e.g. 'pascal'.

Return type *plonk.analysis.profile.Profile*

Examples

Set multiple default units.

```
>>> profile.set_units(pressure='pascal', density='g/cm^3')
```

to_dataframe(*columns=None*, *units=None*)
Convert Profile to DataFrame.

Parameters

- **columns** (*optional*) – A list of columns to add to the data frame. If None, add all loaded columns. Default is None.
- **units** (*optional*) – A list of units corresponding to columns add to the data frame. Units must be strings, and must be base units. I.e. 'cm' not '10 cm'. If None, use default, i.e. cgs. Default is None.

Returns

Return type DataFrame

to_function(*profile*, ***kwargs*)
Create function via interpolation.

The function is of the coordinate of the profile, e.g. 'radius', and returns values of the selected profile, e.g. 'scale_height'. The function is generated from the scipy.interpolate function interp1d.

Parameters **profile** (*str*) – The profile function to create as a string, e.g. 'scale_height'.

Returns The function.

Return type Callable

Examples

Select all particles within a scale height in a disc.

```
>>> scale_height = prof.to_function('scale_height')
>>> subsnap = snap[np.abs(snap['z']) < scale_height(snap['R'])]
```

plonk.**load_profile**(*snap*, *ndim=2*, *cmin=None*, *cmax=None*, *n_bins=100*, *aggregation='average'*, *spacing='linear'*, *coordinate='x'*, *ignore_accreted=True*)
Profiles.

A profile is a binning of particles in Cartesian slices, or cylindrical or spherical shells around the origin, i.e. (0, 0, 0). For cylindrical profiles the cylindrical cells are perpendicular to the xy-plane, i.e. the bins are averaged azimuthally and in the z-direction. Cartesian profiles can be in the x-, y-, and z-directions.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ndim** (*optional*) – The dimension of the profile. For ndim == 2, the radial binning is cylindrical in the xy-plane. For ndim == 3, the radial binning is spherical. For ndim == 1, the radial binning is Cartesian along the x-axis. Default is 2.
- **cmin** (*optional*) – The minimum coordinate for binning. Can be a string, e.g. '10 au', or a quantity with units, e.g. plonk.units('10 au'). Defaults to minimum on the particles.
- **cmax** (*optional*) – The maximum coordinate for binning. Can be a string, e.g. '10 au', or a quantity with units, e.g. plonk.units('10 au'). Defaults to the 99 percentile distance.
- **n_bins** (*optional*) – The number of radial bins. Default is 100.
- **aggregation** (*optional*) – The method to aggregate particle quantities in bins by. Options are 'average', 'mean', or 'median'. Here 'average' is a mass-weighted average. Default is 'average'.
- **spacing** (*optional*) – The spacing of radial bins. Can be 'linear' or 'log'. Default is 'linear'.
- **coordinate** (*optional*) – The coordinate ('x', 'y', or 'z') for Cartesian profiles only, i.e. when ndim==1. Default is 'x'. For cylindrical and spherical profiles the coordinate is 'radius'.
- **ignore_accreted** (*optional*) – Ignore particles accreted onto sinks. Default is True.

Return type *Profile*

Examples

Generate profile from snapshot.

```
>>> prof = plonk.load_profile(snap=snap)
>>> prof = plonk.load_profile(snap=snap, n_bins=300)
>>> prof = plonk.load_profile(snap=snap, cmin='10 au', cmax='300 au')
>>> prof = plonk.load_profile(snap=snap, spacing='log')
```

To access a profile.

```
>>> prof['surface_density']
>>> prof['scale_height']
```

To set a new profile.

```
>>> prof['aspect_ratio'] = prof['scale_height'] / prof['radius']
```

Alternatively use the add_profile decorator.

```
>>> @prof.add_profile
... def mass(prof):
...     M = prof.snap['mass']
...     return prof.particles_to_binned_quantity('sum', M)
```

Plot one or many quantities on the profile.

```
>>> prof.plot('radius', 'density')
>>> prof.plot('radius', ['angular_momentum_x', 'angular_momentum_y'])
```

Plot a quantity on the profile with units.

```
>>> units = {'position': 'au', 'surface_density'='g/cm^2'}
>>> prof.plot('radius', 'surface_density', units=units)
```

Particle filters

Filter particles to produce SubSnaps.

plonk.analysis.filters.**annulus**(*snap, radius_min, radius_max, height, center=<Quantity([0 0 0], 'astronomical_unit')>*)

Particles within an annulus.

Parameters

- **snap** (*Union[plonk.snap.snap.Snap, plonk.snap.snap.SubSnap]*) – The Snap object.
- **radius_min** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The inner radius of the annulus.
- **radius_max** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The outer radius of the annulus.
- **height** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The height of the annulus.
- **center** (*optional*) – The center of the annulus as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The SubSnap with particles in the annulus.

Return type *SubSnap*

plonk.analysis.filters.**box**(*snap, xwidth, ywidth, zwidth, center=<Quantity([0 0 0], 'astronomical_unit')>*)

Particles within a box.

Parameters

- **snap** (*Union[plonk.snap.snap.Snap, plonk.snap.snap.SubSnap]*) – The Snap object.
- **xwidth** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The x-width of the box.
- **ywidth** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The y-width of the box.
- **zwidth** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The z-width of the box.
- **center** (*optional*) – The center of the box as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The SubSnap with particles in the box.

Return type *SubSnap*

plonk.analysis.filters.**cylinder**(*snap, radius, height, center=<Quantity([0 0 0], 'astronomical_unit')>*)

Particles within a cylinder.

Parameters

- **snap** (*Union[plonk.snap.snap.Snap, plonk.snap.snap.SubSnap]*) – The Snap object.
- **radius** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The radius of the cylinder.
- **height** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The height of the cylinder.
- **center** (*optional*) – The center of the cylinder as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The SubSnap with particles in the cylinder.

Return type *SubSnap*

plonk.analysis.filters.**shell**(*snap, radius_min, radius_max, center=<Quantity([0 0 0], 'astronomical_unit')>*)

Particles within a spherical shell.

Parameters

- **snap** (*Union[plonk.snap.snap.Snap, plonk.snap.snap.SubSnap]*) – The Snap object.
- **radius_min** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The inner radius of the shell.
- **radius_max** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The outer radius of the shell.
- **center** (*optional*) – The center of the shell as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The SubSnap with particles in the shell.

Return type *SubSnap*

plonk.analysis.filters.**sphere**(*snap, radius, center=<Quantity([0 0 0], 'astronomical_unit')>*)

Particles within a sphere.

Parameters

- **snap** (*Union[plonk.snap.snap.Snap, plonk.snap.snap.SubSnap]*) – The Snap object.
- **radius** (*pint.quantity.build_quantity_class.<locals>.Quantity*) – The radius of the sphere.
- **center** (*optional*) – The center of the sphere as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The SubSnap with particles in the sphere.

Return type *SubSnap*

Extra particle quantities

Calculate extra quantities on the particles.

plonk.analysis.particles.**angular_momentum**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the angular momentum.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the angular momentum as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The angular momentum on the particles.

Return type Quantity

plonk.analysis.particles.**angular_velocity**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the angular velocity.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the angular velocity as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The angular velocity on the particles.

Return type Quantity

plonk.analysis.particles.**azimuthal_angle**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the azimuthal angle.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the azimuthal angle as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The azimuthal angle on the particles.

Return type Quantity

plonk.analysis.particles.**dust_density**(*snap*, *ignore_accreted=False*)
Calculate the dust density per species.

For dust/gas mixtures this is from the dust fraction.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The dust density per species on the particles.

Return type Quantity

plonk.analysis.particles.**dust_mass**(*snap*, *ignore_accreted=False*)
Calculate the dust mass per species.

For dust/gas mixtures this is from the dust fraction.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The dust mass per species on the particles.

Return type Quantity

plonk.analysis.particles.**gas_density**(*snap*, *ignore_accreted=False*)
Calculate the gas density.

For dust/gas mixtures this is from the dust fraction.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The gas density on the particles.

Return type Quantity

plonk.analysis.particles.**gas_fraction**(*snap*, *ignore_accreted=False*)
Calculate the gas fraction.

For dust/gas mixtures this is from the dust fraction.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The gas fraction on the particles.

Return type Quantity

plonk.analysis.particles.**gas_mass**(*snap*, *ignore_accreted=False*)
Calculate the gas mass.

For dust/gas mixtures this is from the dust fraction.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The gas mass on the particles.

Return type Quantity

plonk.analysis.particles.**kinetic_energy**(*snap*, *ignore_accreted=False*)
Calculate the kinetic energy.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The kinetic energy on the particles.

Return type Quantity

plonk.analysis.particles.**momentum**(*snap*, *ignore_accreted=False*)
Calculate the momentum.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The linear momentum on the particles.

Return type Quantity

plonk.analysis.particles.**polar_angle**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the polar angle.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the polar angle as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The azimuthal angle on the particles.

Return type Quantity

plonk.analysis.particles.**radius_cylindrical**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the cylindrical radial distance.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the cylindrical radius as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The radial distance on the particles.

Return type Quantity

plonk.analysis.particles.**radius_spherical**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the spherical radial distance.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the spherical radius as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The radial distance on the particles.

Return type Quantity

plonk.analysis.particles.**specific_angular_momentum**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the specific angular momentum.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.

- **origin** (*optional*) – The origin around which to compute the specific angular momentum as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The specific angular momentum on the particles.

Return type Quantity

plonk.analysis.particles.**specific_kinetic_energy**(*snap*, *ignore_accreted=False*)
Calculate the specific kinetic energy.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The specific kinetic energy on the particles.

Return type Quantity

plonk.analysis.particles.**temperature**(*snap*, *molecular_weight=None*, *ignore_accreted=False*)
Calculate the gas temperature.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **molecular_weight** (*float*) – The gas molecular weight in gram / mole. E.g. 2.381 for molecular hydrogen.
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The gas temperature on the particles.

Return type Quantity

plonk.analysis.particles.**velocity_radial_cylindrical**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the cylindrical radial velocity.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the cylindrical radial velocity as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The radial velocity on the particles.

Return type Quantity

plonk.analysis.particles.**velocity_radial_spherical**(*snap*, *origin=None*, *ignore_accreted=False*)
Calculate the spherical radial velocity.

Parameters

- **snap** (*Union[SnapLike, Sinks]*) – The Snap object.
- **origin** (*optional*) – The origin around which to compute the spherical radial velocity as a Quantity like (x, y, z) * au. Default is (0, 0, 0).
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The radial velocity on the particles.

Return type Quantity

Global quantities

Calculate global (total) quantities on the particles.

plonk.analysis.total.**accreted_mass**(*snap*)
Calculate the accreted mass.

Parameters **snap** (*SnapLike*) – The Snap object.

Returns The accreted mass.

Return type Quantity

plonk.analysis.total.**angular_momentum**(*snap*, *sinks=True*, *origin=None*)
Calculate the total angular momentum.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).
- **origin** (*optional*) – The origin around which to compute the angular momentum as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The total angular momentum like (lx, ly, lz).

Return type Quantity

plonk.analysis.total.**center_of_mass**(*snap*, *sinks=True*)
Calculate the center of mass.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The center of mass as a vector (cx, cy, cz).

Return type Quantity

plonk.analysis.total.**dust_mass**(*snap*, *squeeze=False*)
Calculate the total dust mass per species.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **squeeze** (*Union[bool, List[int]]*) – If True return all subtypes in a single array. Default is False.

Returns The total dust mass per species.

Return type Quantity

plonk.analysis.total.**gas_mass**(*snap*)
Calculate the total gas mass.

Parameters **snap** (*SnapLike*) – The Snap object.

Returns The total gas mass.

Return type Quantity

plonk.analysis.total.**kinetic_energy**(*snap*, *sinks=True*)
Calculate the total kinetic energy.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The total kinetic energy.

Return type Quantity

plonk.analysis.total.**mass**(*snap*, *sinks=True*)
Calculate the total mass.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The total mass.

Return type Quantity

plonk.analysis.total.**momentum**(*snap*, *sinks=True*)
Calculate the total momentum.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The total linear momentum like (px, py, pz).

Return type Quantity

plonk.analysis.total.**specific_angular_momentum**(*snap*, *sinks=True*, *origin=None*)
Calculate the total specific angular momentum.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).
- **origin** (*optional*) – The origin around which to compute the angular momentum as a Quantity like (x, y, z) * au. Default is (0, 0, 0).

Returns The total specific angular momentum on the particles like (hx, hy, hz).

Return type Quantity

plonk.analysis.total.**specific_kinetic_energy**(*snap*, *sinks=True*)
Calculate the total specific kinetic energy.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The total specific kinetic energy.

Return type Quantity

Sink quantities

Calculate extra quantities on the sinks.

plonk.analysis.sinks.**Hill_radius**(*primary*, *secondary*)
Calculate the Hill radius.

Parameters

- **primary** (Sinks) – The primary, i.e. heavy object, as a Sinks object. It must have one sink.
- **secondary** (Sinks) – The secondary, i.e. light objects, as a Sinks object. It can have more than one sink, e.g. when calculating the Hill radiius for multiple planets orbiting a star.

Returns

Return type Quantity

plonk.analysis.sinks.**Roche_sphere**(*sinks*)
Calculate an estimate of the Roche sphere for two bodies.

The Roche sphere radius is calculated around the first of the two sinks. Uses the formula from Eggleton (1983) ApJ 268, 368-369.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

plonk.analysis.sinks.**eccentricity**(*sinks*)
Calculate the eccentricity.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

plonk.analysis.sinks.**gravitational_potential_energy**(*sinks*)
Calculate the total gravitational potential energy.

Parameters **sinks** (Sinks) – The Sinks object.

Returns

Return type Quantity

plonk.analysis.sinks.**inclination**(*sinks*)
Calculate the inclination for two bodies.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

plonk.analysis.sinks.**kinetic_energy**(*sinks*)
Calculate the total kinetic energy.

Parameters **sinks** (Sinks) – The Sinks object.

Returns

Return type Quantity

plonk.analysis.sinks.**mean_motion**(*sinks*)
Calculate the mean motion for two bodies.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

plonk.analysis.sinks.**orbital_period**(*sinks*)
Calculate the orbital period for two bodies.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

plonk.analysis.sinks.**semi_major_axis**(*sinks*)
Calculate the semi-major axis for two bodies.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

plonk.analysis.sinks.**specific_angular_momentum**(*sinks*)
Calculate the specific orbital energy for two bodies.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns The specific angular momentum.

Return type Quantity

plonk.analysis.sinks.**specific_orbital_energy**(*sinks*)
Calculate the specific orbital energy for two bodies.

Parameters **sinks** (Sinks) – The Sinks object. Must have length 2.

Returns

Return type Quantity

Accretion discs

Analysis for accretion discs.

plonk.analysis.discs.**eccentricity**(*snap*, *central_body=None*, *ignore_accreted=False*)
Calculate the eccentricity.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **central_body** (*optional*) – A dictionary with the mass, position, and velocity (as Pint quantities) of the central body around which the particles are orbiting. If None, attempt to read from snap.properties['central_body'].
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The eccentricity on the particles.

Return type Quantity

plonk.analysis.discs.**inclination**(*snap*, *central_body=None*, *ignore_accreted=False*)
Calculate the inclination.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **central_body** (*optional*) – A dictionary with the mass, position, and velocity (as Pint quantities) of the central body around which the particles are orbiting. If None, attempt to read from snap.properties['central_body'].
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The inclination on the particles.

Return type Quantity

plonk.analysis.discs.**inclination_angle**(*snap*)
Calculate the disc inclination.

The inclination is calculated by taking the angle between the angular momentum vector and the z-axis, with the angular momentum calculated with respect to the center of mass.

Parameters **snap** (*SnapLike*) – The Snap object.

Returns The disc inclination.

Return type Quantity

plonk.analysis.discs.**keplerian_frequency**(*snap*, *central_body=None*, *ignore_accreted=False*)
Calculate the Keplerian orbital frequency.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **central_body** (*optional*) – A dictionary with the mass, position, and velocity (as Pint quantities) of the central body around which the particles are orbiting. If None, attempt to read from snap.properties['central_body'].
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The Keplerian frequency on the particles.

Return type Quantity

plonk.analysis.discs.**position_angle**(*snap*)
Calculate the disc position angle.

The position angle is taken from the x-axis in the xy-plane. It defines a unit vector around which the snap is inclined.

Parameters **snap** (*SnapLike*) – The Snap object.

Returns The disc position angle.

Return type Quantity

plonk.analysis.discs.**rotate_edge_on**(*snap*, *sinks=False*)
Rotate to edge-on with the angular momentum vector.

I.e. rotate such that the angular momentum points in the y-direction.

Parameters

- **snap** (*SnapLike*) – The Snap object.

- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The rotated Snap.

Return type *Snap*

plonk.analysis.discs.**rotate_face_on**(*snap*, *sinks=False*)
Rotate to face-on with the angular momentum vector.

I.e. rotate such that the angular momentum points in the z-direction.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns The rotated Snap.

Return type *Snap*

plonk.analysis.discs.**semi_major_axis**(*snap*, *central_body=None*, *ignore_accreted=False*)
Calculate the semi-major axis.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **central_body** (*optional*) – A dictionary with the mass, position, and velocity (as Pint quantities) of the central body around which the particles are orbiting. If None, attempt to read from snap.properties['central_body'].
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The semi-major axis on the particles.

Return type Quantity

plonk.analysis.discs.**stokes_number**(*snap*, *central_body=None*, *ignore_accreted=False*)
Calculate the Stokes number.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **central_body** (*optional*) – A dictionary with the mass, position, and velocity (as Pint quantities) of the central body around which the particles are orbiting. If None, attempt to read from snap.properties['central_body'].
- **ignore_accreted** (*optional*) – Ignore accreted particles. Default is False.

Returns The Stokes number on the particles.

Return type Quantity

plonk.analysis.discs.**unit_normal**(*snap*, *sinks=False*)
Calculate unit normal to plane of rotation.

I.e. calculate a unit angular momentum vector.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **sinks** (*optional*) – Include sink particles specified by a list of indices, or a bool indicating all sinks or no sinks. Default is True (all sinks).

Returns A unit angular momentum vector.

Return type ndarray

SPH summation

SPH sums over neighbours.

plonk.analysis.sph.**derivative**(*snap*, *derivative*, *quantity*, *kernel='cubic'*, *chunk_size=None*, *verbose=False*)

Calculate derivatives of quantities.

WARNING: This function is experimental.

Calculate derivatives such as grad, div, curl on any particle quantity using the SPH kernel gradient.

Parameters

- **snap** (Snap) – The Snap object.
- **derivative** (*str*) – Options are 'grad', 'div', 'curl'.
- **quantity** (*str*) – The quantity to take the derivative of. Must be a string to pass to Snap object which returns a scalar.
- **kernel** (*str*) – Kernel to compute density. E.g. 'cubic', 'quintic', or 'Wendland C4'.
- **chunk_size** (*optional*) – The size of chunks, in terms of particle number, for neighbour finding. If the chunk size is too large then the neighbour finding algorithm (scipy.spatial.cKDTree.query_ball_point) runs out of memory. Default is None.
- **verbose** (*optional*) – If True, print progress. Default is False.

Returns The derivative of the quantity.

Return type Quantity

plonk.analysis.sph.**summation**(*snap*, *result_shape*, *compute_function*, *compute_function_kwargs*, *kernel='cubic'*, *chunk_size=None*, *verbose=False*)

Calculate SPH sums.

WARNING: This function is experimental.

Parameters

- **snap** (Snap) – The Snap object.
- **result_shape** (*Tuple[int, ...]*) – The shape, as a tuple, of the returned sum.
- **compute_function** (*Callable*) – The function that computes the sums.
- **compute_function_kwargs** (*Dict[str, Any]*) – The keyword arguments to the function that computes the sums.
- **kernel** (*str*) – Kernel to compute density. E.g. 'cubic', 'quintic', or 'Wendland C4'.
- **chunk_size** (*optional*) – The size of chunks, in terms of particle number, for neighbour finding. If the chunk size is too large then the neighbour finding algorithm (scipy.spatial.cKDTree.query_ball_point) runs out of memory. Default is None.
- **verbose** (*optional*) – If True, print progress. Default is False.

Returns The result of the SPH summation.

Return type Quantity

2.3.3 Visualization

Smoothed particle hydrodynamics data as a *Snap* can be visualized by plotting the particles directly with or without color or size variation to represent a quantity, or via interpolation of scalar and vector quantities to a pixel grid, presented as an image, contour plot, vector plot, or stream plot.

Below are the functions for visualization, and interpolation, of particle data to a pixel grid, particle plots, and functions to produce animations of these visualizations and of *Profile* objects.

Visualize

plonk.**image**(*snap*, *quantity*, *, *x='x'*, *y='y'*, *interp='projection'*, *weighted=False*, *slice_normal=None*, *slice_offset=None*, *extent=None*, *units=None*, *ax=None*, *ax_kwargs={}*, *colorbar_kwargs={}*, ***kwargs*)

Visualize scalar SPH data as an image.

Visualize scalar smoothed particle hydrodynamics data by interpolation to a pixel grid.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **quantity** (*str*) – The quantity to visualize. Must be a string to pass to Snap.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **interp** (*str*) – The interpolation type. Default is 'projection'.
  - 'projection' : 2d interpolation via projection to xy-plane
  - 'slice' : 3d interpolation via cross-section slice.
- **weighted** (*bool*) – Whether to density weight the interpolation or not. Default is False.
- **slice_normal** (*Tuple[float, float, float]*) – The normal vector to the plane in which to take the cross-section slice as a tuple (x, y, z). Default is (0, 0, 1).
- **slice_offset** (*Union[Quantity, float]*) – The offset of the cross-section slice. Default is 0.0.
- **extent** (*Quantity*) – The range in the x and y-coord as (xmin, xmax, ymin, ymax) where xmin, etc. can be floats or quantities with units of length. The default is to set the extent to a box of size such that 99% of particles are contained within.
- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density. There is a special key 'projection' that corresponds to the length unit in the direction of projection for projected interpolation plots.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- **colorbar_kwargs** – Keyword arguments to pass to matplotlib Colorbar.
- ****kwargs** – Additional keyword arguments to pass to interpolation and matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Notes

Additional parameters passed as keyword arguments will be passed to lower level functions as required. E.g. Plonk uses matplotlib's imshow for a image plot, so additional arguments to imshow can be passed this way.

See below for additional parameters for interpolation, colorbars, etc. All other keyword arguments are passed to the appropriate matplotlib function.

Parameters

- **num_pixels** (*tuple*) – The number of pixels to interpolate particle quantities to as a tuple (nx, ny). Default is (512, 512).
- **show_colorbar** (*bool*) – Whether or not to display a colorbar. Default is True.
- **snap** (*SnapLike*) –
- **quantity** (*str*) –
- **x** (*str*) –
- **y** (*str*) –
- **interp** (*str*) –
- **weighted** (*bool*) –
- **slice_normal** (*Tuple[float, float, float]*) –
- **slice_offset** (*Union[Quantity, float]*) –
- **extent** (*Quantity*) –
- **units** (*Dict[str, str]*) –
- **ax** (*Any*) –

Return type Any

Examples

Show an image of the surface density in xy-plane.

```
>>> plonk.image(snap=snap, quantity='density')
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.image(quantity='density')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'density': 'g/cm^3', 'projection': 'cm'}
>>> snap.image(quantity='density', units=units)
```

Show a slice image of the density in xy-plane at z=0.

```
>>> snap.image(quantity='density', interp='slice')
```

plonk.**plot**(*snap*, ***, *x='x'*, *y='y'*, *c=None*, *s=None*, *units=None*, *xlim=None*, *ylim=None*, *ax=None*, *ax_kwargs={}*, *colorbar_kwargs={}*, ***kwargs*)

Visualize SPH data as a particle plot.

Visualize SPH data by plotting the particles, or a subset of the particles, possibly with marker colors and different sizes.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **c** (*str*) – The quantity to color the particles. Must be a string to pass to Snap.
- **s** (*str*) – The quantity to set the particle size. Must be a string to pass to Snap.
- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density.
- **xlim** (*Quantity*) – The range in the x-coord as (xmin, xmax) where xmin/xmax can be floats or quantities with units of length.
- **ylim** (*Quantity*) – The range in the y-coord as (ymin, ymax) where ymin/ymax can be floats or quantities with units of length.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- **colorbar_kwargs** – Keyword arguments to pass to matplotlib Colorbar.
- ****kwargs** – Additional keyword arguments to pass to matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Examples

Show the particles in xy-plane.

```
>>> plonk.plot(snap=snap)
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.plot()
```

Plot density against x.

```
>>> snap.plot(x='x', y='density')
```

Color particles by density.

```
>>> snap.plot(x='x', y='y', c='density')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'density': 'g/cm^3'}
>>> snap.plot(x='x', y='y', c='density', units=units)
```

plonk.**vector**(*snap*, *quantity*, *, *x='x'*, *y='y'*, *interp='projection'*, *weighted=False*, *slice_normal=None*, *slice_offset=None*, *extent=None*, *units=None*, *ax=None*, *ax_kwargs={}*, ***kwargs*)

Visualize vector SPH data as a vector plot.

Visualize scalar smoothed particle hydrodynamics data by interpolation to a pixel grid of arrows.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object to visualize.
- **quantity** (*str*) – The quantity to visualize. Must be a string to pass to Snap.
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **interp** (*str*) – The interpolation type. Default is 'projection'.
  - 'projection' : 2d interpolation via projection to xy-plane
  - 'slice' : 3d interpolation via cross-section slice.
- **weighted** (*bool*) – Whether to density weight the interpolation or not. Default is False.
- **slice_normal** (*Tuple[float, float, float]*) – The normal vector to the plane in which to take the cross-section slice as a tuple (x, y, z). Default is (0, 0, 1).
- **slice_offset** (*Union[Quantity, float]*) – The offset of the cross-section slice. Default is 0.0.
- **extent** (*Quantity*) – The range in the x and y-coord as (xmin, xmax, ymin, ymax) where xmin, etc. can be floats or quantities with units of length. The default is to set the extent to a box of size such that 99% of particles are contained within.
- **units** (*Dict[str, str]*) – The units of the plot as a dictionary. The keys correspond to quantities such as 'position', 'density', 'velocity', and so on. The values are strings representing units, e.g. 'g/cm^3' for density. There is a special key 'projection' that corresponds to the length unit in the direction of projection for projected interpolation plots.
- **ax** (*Any*) – A matplotlib Axes handle.
- **ax_kwargs** – Keyword arguments to pass to matplotlib Axes.
- ****kwargs** – Additional keyword arguments to pass to interpolation and matplotlib functions.

Returns The matplotlib Axes object.

Return type ax

Notes

Additional parameters passed as keyword arguments will be passed to lower level functions as required.

See below for additional parameters for interpolation, vector properties, etc. All other keyword arguments are passed to the appropriate matplotlib function.

Parameters

- **num_pixels** (*tuple*) – The number of pixels to interpolate particle quantities to as a tuple (nx, ny). Default is (512, 512).
- **number_of_arrows** (*tuple*) – The number of arrows to display by sub-sampling the interpolated data. Default is (25, 25).

- **normalize_vectors** (*bool*) – Whether to normalize the arrows to all have the same length. Default is False.
- **snap** (*SnapLike*) –
- **quantity** (*str*) –
- **x** (*str*) –
- **y** (*str*) –
- **interp** (*str*) –
- **weighted** (*bool*) –
- **slice_normal** (*Tuple[float, float, float]*) –
- **slice_offset** (*Union[Quantity, float]*) –
- **extent** (*Quantity*) –
- **units** (*Dict[str, str]*) –
- **ax** (*Any*) –

Return type Any

Examples

Show a vector plot of velocity in xy-plane.

```
>>> plonk.vector(snap=snap, quantity='velocity')
```

Alternatively, access the function as a method on the Snap object.

```
>>> snap.vector(quantity='velocity')
```

Set units for the plot.

```
>>> units = {'position': 'au', 'velocity': 'km/s', 'projection': 'km'}
>>> snap.vector(quantity='velocity', units=units)
```

Show a slice plot of the velocity in xy-plane at z=0.

```
>>> snap.vector(quantity='density', interp='slice')
```

plonk.**visualize_sim**(*sim*, *kind*, ***kwargs*)

Visualize a simulation.

Parameters

- **sim** (Simulation) – The Simulation object.
- **kind** (*str*) – The kind of plot: ‘particle’, ‘image’, ‘vector’.
- ****kwargs** – Keyword arguments to pass to plotting methods such as plonk.image, plonk.plot, and plonk.vector.

Returns

Return type VisualizeSimulation

Examples

Visualize a simulation by density projection images.

```
>>> viz = visualize_sim(sim=sim, kind='image', quantity='density')
```

Alternatively.

```
>>> sim.visualize(kind='image', quantity='density')
```

Go forwards and backwards through snaps.

```
>>> viz.next()
>>> viz.prev()
```

Go to a particular snap, or skip ahead.

```
>>> viz.goto(10)
>>> viz.next(5)
```

Animation

plonk.**animate**(*filename*, *, *snaps=None*, *profiles=None*, *quantity=None*, *x=None*, *y=None*, *fig=None*, *adaptive_colorbar=True*, *adaptive_limits=True*, *text=None*, *text_kwargs={}*, *func_animation_kwargs={}*, *save_kwargs={}*, ***kwargs*)

Animate a Snap or Profile visualization.

Pass in a list of Snap objects or Profile objects. If snaps are passed in and the quantity is None, then the animation will be of particle plots, otherwise it will be of images. If profiles are passed in the animation will be of profiles.

Parameters

- **filename** (*Union[str, Path]*) – The file name to save the animation to.
- **snaps** (*List[SnapLike]*) – A list of Snap objects to animate.
- **profiles** (*List[Profile]*) – A list of Profile objects to animate.
- **quantity** (*optional*) – The quantity to visualize. Must be a string to pass to Snap.
- **x** (*optional*) – The quantity for the x-axis. Must be a string to pass to Snap. For interpolated plots must be 'x', 'y', or 'z'.
- **y** (*optional*) – The quantity for the y-axis. Must be a string to pass to Snap. For interpolated plots must be 'x', 'y', or 'z'.
- **fig** (*optional*) – A matplotlib Figure object to animate. If None, generate a new Figure.
- **adaptive_colorbar** (*optional*) – If True, adapt colorbar range during animation. If False, the colorbar range is fixed by the initial image. Default is True.
- **adaptive_limits** (*optional*) – If True, adapt plot limits during animation. If False, the plot limits are fixed by the initial plot. Default is True.
- **text** (*optional*) – List of strings to display per snap.
- **text_kwargs** (*optional*) – Keyword arguments to pass to matplotlib text.
- **func_animation_kwargs** (*optional*) – Keyword arguments to pass to matplotlib FuncAnimation.

- **save_kwargs** (*optional*) – Keyword arguments to pass to matplotlib Animation.save.
- ****kwargs** – Arguments to pass to visualize.image.

Returns The matplotlib FuncAnimation object.

Return type anim

Examples

Make an image animation of projected density.

```
>>> units = {
...     'position': 'au',
...     'density': 'g/cm^3',
...     'projection': 'cm'
... }
>>> plonk.animate(
...     filename='animation.mp4',
...     snaps=snaps,
...     quantity='density',
...     units=units,
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

Make a particle animation of x vs density.

```
>>> units = {'position': 'au', 'density': 'g/cm^3'}
>>> plonk.animate(
...     filename='animation.mp4',
...     snaps=snaps,
...     x='x',
...     y='density',
...     units=units,
...     adaptive_limits=False,
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

Make a profile animation of radius vs surface density.

```
>>> units={'position': 'au', 'surface_density': 'g/cm^2'}
>>> plonk.animate(
...     filename='animation.mp4',
...     profiles=profiles,
...     x='radius',
...     y='surface_density',
...     units=units,
...     adaptive_limits=False,
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

plonk.visualize.**animation_images**(*, *filename*, *snaps*, *quantity*, *fig=None*, *adaptive_colorbar=True*, *text=None*, *text_kwargs={}*, *func_animation_kwargs={}*, *save_kwargs={}*, ***kwargs*)

Generate an animation of images.

Parameters

- **filename** (*Union[str, Path]*) – The file name to save the animation to.
- **snaps** (*List[SnapLike]*) – A list of Snap objects to animate.
- **quantity** (*str*) – The quantity to visualize. Must be a string to pass to Snap.
- **fig** (*optional*) – A matplotlib Figure object to animate. If None, generate a new Figure.
- **adaptive_colorbar** (*optional*) – If True, adapt colorbar range during animation. If False, the colorbar range is fixed by the initial image. Default is True.
- **text** (*optional*) – List of strings to display per snap.
- **text_kwargs** (*optional*) – Keyword arguments to pass to matplotlib text.
- **func_animation_kwargs** (*optional*) – Keyword arguments to pass to matplotlib FuncAnimation.
- **save_kwargs** (*optional*) – Keyword arguments to pass to matplotlib Animation.save.
- ****kwargs** – Arguments to pass to visualize.image.

Returns The matplotlib FuncAnimation object.

Return type anim

Examples

Make an animation of projected density.

```
>>> animation_images(
...     filename='animation.mp4',
...     snaps=snaps,
...     quantity='density',
...     units={'position': 'au', 'density': 'g/cm^3'},
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

plonk.visualize.**animation_particles**(*, *filename*, *snaps*, *fig=None*, *adaptive_limits=True*, *text=None*, *text_kwargs={}*, *func_animation_kwargs={}*, *save_kwargs={}*, ***kwargs*)

Generate an animation of particle plots.

Parameters

- **filename** (*Union[str, Path]*) – The file name to save the animation to.
- **snaps** (*List[SnapLike]*) – A list of Snap objects to animate.
- **fig** (*optional*) – A matplotlib Figure object to animate. If None, generate a new Figure.
- **adaptive_limits** (*optional*) – If True, adapt plot limits during animation. If False, the plot limits are fixed by the initial plot. Default is True.
- **text** (*optional*) – List of strings to display per snap.
- **text_kwargs** (*optional*) – Keyword arguments to pass to matplotlib text.
- **func_animation_kwargs** (*optional*) – Keyword arguments to pass to matplotlib FuncAnimation.
- **save_kwargs** (*optional*) – Keyword arguments to pass to matplotlib Animation.save.

- ****kwargs** – Arguments to pass to visualize.plot.

Returns The matplotlib FuncAnimation object.

Return type anim

Examples

Make an animation of x vs density on the particles.

```
>>> animation_particles(
...     filename='animation.mp4',
...     snaps=snaps,
...     x='x',
...     y='density',
...     adaptive_limits=False,
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

plonk.visualize.**animation_profiles**(*, *filename*, *profiles*, *x*, *y*, *fig=None*, *adaptive_limits=True*, *text=None*, *text_kwargs={}*, *func_animation_kwargs={}*, *save_kwargs={}*, ***kwargs*)

Generate an animation of a profile.

Parameters

- **filename** (*Union[str, Path]*) – The file name to save the animation to.
- **profiles** (*List[Profile]*) – A list of Profile objects to animate.
- **x** (*str*) – The quantity for the x-axis. Must be a string to pass to Snap.
- **y** (*Union[str, List[str]]*) – The quantity for the y-axis, or list of quantities. Must be a string (or list of strings) to pass to Snap.
- **fig** (*optional*) – A matplotlib Figure object to animate. If None, generate a new Figure.
- **adaptive_limits** (*optional*) – If True, adapt plot limits during animation. If False, the plot limits are fixed by the initial plot. Default is True.
- **text** (*optional*) – List of strings to display per profile plot.
- **text_kwargs** (*optional*) – Keyword arguments to pass to matplotlib text.
- **func_animation_kwargs** (*optional*) – Keyword arguments to pass to matplotlib FuncAnimation.
- **save_kwargs** (*optional*) – Keyword arguments to pass to matplotlib Animation.save.
- ****kwargs** – Arguments to pass to Profile.plot function.

Returns The matplotlib FuncAnimation object.

Return type anim

Examples

Make an animation of radius vs surface density.

```
>>> animation_profiles(
...     filename='animation.mp4',
...     profiles=profiles,
...     x='radius',
...     y='surface_density',
...     units={'position': 'au', 'surface_density': 'g/cm^2'},
...     adaptive_limits=False,
...     save_kwargs={'fps': 10, 'dpi': 300},
... )
```

Interpolation

plonk.**interpolate**(*, *snap*, *quantity*, *x='x'*, *y='y'*, *interp*, *weighted=False*, *slice_normal=None*, *slice_offset=None*, *extent*, *num_pixels=None*)

Interpolate a quantity on the snapshot to a pixel grid.

Parameters

- **snap** (*SnapLike*) – The Snap (or SubSnap) object.
- **quantity** (*str*) – The quantity to visualize. Must be a string to pass to Snap,
- **x** (*str*) – The x-coordinate for the visualization. Must be a string to pass to Snap. Default is 'x'.
- **y** (*str*) – The y-coordinate for the visualization. Must be a string to pass to Snap. Default is 'y'.
- **interp** (*str*) – The interpolation type. Default is 'projection'.
  - 'projection' : 2d interpolation via projection to xy-plane
  - 'slice' : 3d interpolation via cross-section slice.
- **weighted** (*bool*) – Use density weighted interpolation. Default is False.
- **slice_normal** (*Tuple[float, float, float]*) – The normal vector to the plane in which to take the cross-section slice as an array (x, y, z).
- **slice_offset** (*Quantity*) – The offset of the cross-section slice. Default is 0.0.
- **extent** (*Quantity*) – The xy extent of the image as (xmin, xmax, ymin, ymax).
- **num_pixels** (*Tuple[float, float]*) – The pixel grid to interpolate the scalar quantity to, as (npixx, npixy). Default is (512, 512).

Returns The interpolated quantity on a pixel grid as a Pint Quantity. The shape for scalar data is (npixx, npixy), and for vector is (2, npixx, npixy).

Return type Quantity

Examples

Interpolate density to grid.

```
>>> grid_data = plonk.interpolate(
...     snap=snap,
...     quantity='density',
...     interp='projection',
...     extent=(-100, 100, -100, 100),
... )
```

plonk.visualize.interpolation.**scalar_interpolation**(*, *quantity*, *x_coordinate*, *y_coordinate*, *dist_from_slice=None*, *extent*, *smoothing_length*, *particle_mass*, *hfact*, *weighted=None*, *num_pixels=(512, 512)*)

Interpolate scalar quantity to a pixel grid.

Parameters

- **quantity** (*numpy.ndarray*) – A scalar quantity on the particles to interpolate.
- **x_coordinate** (*numpy.ndarray*) – Particle coordinate for x-axis in interpolation.
- **y_coordinate** (*numpy.ndarray*) – Particle coordinate for y-axis in interpolation.
- **dist_from_slice** (*Optional[numpy.ndarray]*) – The distance from the cross section slice. Only required for cross section interpolation.
- **extent** (*Tuple[float, float, float, float]*) – The range in the x- and y-direction as (xmin, xmax, ymin, ymax).
- **smoothing_length** (*numpy.ndarray*) – The smoothing length on each particle.
- **particle_mass** (*numpy.ndarray*) – The particle mass on each particle.
- **hfact** (*float*) – The smoothing length factor.
- **weighted** (*Optional[bool]*) – Use density weighted interpolation. Default is off.
- **num_pixels** (*Tuple[float, float]*) – The pixel grid to interpolate the scalar quantity to, as (npixx, npixy). Default is (512, 512).

Returns An array of scalar quantities interpolated to a pixel grid with shape (npixx, npixy).

Return type ndarray

plonk.visualize.interpolation.**vector_interpolation**(*, *quantity_x*, *quantity_y*, *x_coordinate*, *y_coordinate*, *dist_from_slice=None*, *extent*, *smoothing_length*, *particle_mass*, *hfact*, *weighted=None*, *num_pixels=(512, 512)*)

Interpolate scalar quantity to a pixel grid.

Parameters

- **quantity_x** (*numpy.ndarray*) – The x-component of a vector quantity to interpolate.
- **quantity_y** (*numpy.ndarray*) – The y-component of a vector quantity to interpolate.
- **x_coordinate** (*numpy.ndarray*) – Particle coordinate for x-axis in interpolation.
- **y_coordinate** (*numpy.ndarray*) – Particle coordinate for y-axis in interpolation.
- **dist_from_slice** (*Optional[numpy.ndarray]*) – The distance from the cross section slice. Only required for cross section interpolation.

- **extent** (*Tuple[float, float, float, float]*) – The range in the x- and y-direction as (xmin, xmax, ymin, ymax).
- **smoothing_length** (*numpy.ndarray*) – The smoothing length on each particle.
- **particle_mass** (*numpy.ndarray*) – The particle mass on each particle.
- **hfact** (*float*) – The smoothing length factor.
- **weighted** (*Optional[bool]*) – Use density weighted interpolation. Default is off.
- **num_pixels** (*Tuple[float, float]*) – The pixel grid to interpolate the scalar quantity to, as (npixx, npixy). Default is (512, 512).

Returns An array of vector quantities interpolated to a pixel grid with shape (2, npixx, npixy).

Return type ndarray

2.3.4 Utils

Here are some useful utility functions. There are SPH kernel functions, mathematical functions, utility functions relating to *Snap* objects, string functions, and visualization functions.

Kernels

SPH kernels.

plonk.utils.kernels.**kernel_cubic**(*q*)
Cubic kernel function.

The form of this function includes the "C_norm" factor. I.e. C_norm * f(q).

Parameters **q** – The particle separation in units of smoothing length, i.e. r/h.

Returns C_norm * f(q) for the cubic kernel.

Return type float

plonk.utils.kernels.**kernel_gradient_cubic**(*q*)
Cubic kernel gradient function.

The form of this function includes the "C_norm" factor. I.e. C_norm * f'(q).

Parameters **q** – The particle separation in units of smoothing length, i.e. r/h.

Returns C_norm * f'(q) for the cubic kernel.

Return type float

plonk.utils.kernels.**kernel_gradient_quintic**(*q*)
Quintic kernel gradient function.

The form of this function includes the "C_norm" factor. I.e. C_norm * f'(q).

Parameters **q** – The particle separation in units of smoothing length, i.e. r/h.

Returns C_norm * f'(q) for the quintic kernel.

Return type float

plonk.utils.kernels.**kernel_gradient_wendland_c4**(*q*)
Wendland C4 kernel gradient function.

The form of this function includes the "C_norm" factor. I.e. C_norm * f'(q).

Parameters **q** – The particle separation in units of smoothing length, i.e. r/h.

Returns C_norm * f'(q) for the Wendland C4 kernel.

Return type float

plonk.utils.kernels.**kernel_quintic**(*q*)
Quintic kernel function.

The form of this function includes the "C_norm" factor. I.e. C_norm * f(q).

Parameters **q** – The particle separation in units of smoothing length, i.e. r/h.

Returns C_norm * f(q) for the quintic kernel.

Return type float

plonk.utils.kernels.**kernel_wendland_c4**(*q*)
Wendland C4 kernel function.

The form of this function includes the "C_norm" factor. I.e. C_norm * f(q).

Parameters **q** – The particle separation in units of smoothing length, i.e. r/h.

Returns C_norm * f(q) for the Wendland C4 kernel.

Return type float

Math

Utils for math.

plonk.utils.math.**average**(*x*, *weights*, ***kwargs*)
Average.

Parameters

- **x** – The array (N,) to take the norm of. Can be ndarray or pint Quantity.
- **weights** – The weights for averaging.
- ****kwargs** – Keyword arguments to pass to np.average.

Returns The average of x.

Return type ndarray

plonk.utils.math.**cross**(*x*, *y*, ***kwargs*)
Cross product.

Parameters

- **x** – The two arrays (N, 3) to take the cross product of. Can be ndarray or pint Quantity.
- **y** – The two arrays (N, 3) to take the cross product of. Can be ndarray or pint Quantity.
- ****kwargs** – Keyword arguments to pass to np.cross.

Returns The cross product of x and y.

Return type ndarray

plonk.utils.math.**distance_from_plane**(*x*, *y*, *z*, *normal*, *height=0*)
Calculate distance from a plane.

Parameters

- **x** (*numpy.ndarray*) – The x-positions.

- **y** (*numpy.ndarray*) – The y-positions.
- **z** (*numpy.ndarray*) – The z-positions.
- **normal** (*numpy.ndarray*) – The normal vector describing the plane (x, y, z).
- **height** (*float*) – The height of the plane above the origin.

Returns

Return type The distance from the plane of each point.

plonk.utils.math.**norm**(*x*, ***kwargs*)
Norm of a vector.

Parameters

- **x** – The arrays (N, 3) to take the norm of. Can be ndarray or pint Quantity.
- ****kwargs** – Keyword arguments to pass to np.linalg.norm.

Returns The norm of x.

Return type ndarray

Snap

Utils for snaps.

plonk.utils.snap.**add_aliases**(*snap*, *filename=None*)
Add array aliases to a Snap.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **config** (*optional*) – The path to a Plonk config.toml file. If None, use the default file.
- **filename** (*Union[str, Path]*) –

plonk.utils.snap.**dust_array_names**(*name*, *num_dust_species*, *add_gas=False*)
List dust array names.

Parameters

- **name** (*str*) – The base array name, e.g. "dust_density" or "stopping_time".
- **num_dust_species** (*int*) – The number of dust species.
- **add_gas** (*bool*) – If True add the gas version of the dust name.

Returns A list of array names with appropriate suffixes.

Return type List

Examples

Get the dust density strings.

```
>>> dust_name_list('dust_density', 5)
['dust_density_001',
 'dust_density_002',
 'dust_density_003',
 'dust_density_004',
 'dust_density_005']
```

Get the dust density strings with gas.

```
>>> dust_name_list(name='dust_density', num_dust_species=5, add_gas=True)
['gas_density',
 'dust_density_001',
 'dust_density_002',
 'dust_density_003',
 'dust_density_004',
 'dust_density_005']
```

plonk.utils.snap.**gravitational_constant_in_code_units**(*snap*)

Gravitational constant in code units.

Parameters **snap** (*SnapLike*) – The Snap object.

Returns The gravitational constant in code units.

Return type float

plonk.utils.snap.**vector_array_names**(*name*, *add_mag=False*)

List vector array names.

Parameters

- **name** (*str*) – The base array name, e.g. “angular_momentum”.
- **add_mag** (*bool*) – If True add the magnitude of the array.

Returns A list of array names with appropriate suffixes.

Return type List

Examples

Get the angular momentum strings.

```
>>> vector_name_list('angular_momentum')
['angular_momentum_x',
 'angular_momentum_y',
 'angular_momentum_z']
```

Get the angular momentum strings with magnitude.

```
>>> vector_name_list(name='angular_momentum', add_mag=True)
['angular_momentum_x',
 'angular_momentum_y',
```

(continues on next page)

(continued from previous page)

```
'angular_momentum_z',
'angular_momentum_mag']
```

Strings

Utils for strings.

plonk.utils.strings.**is_documented_by**(*original*)
Wrap function to add docstring.

plonk.utils.strings.**pretty_array_name**(*s*)
Prettify an array name string.

Parameters **s** (*str*) – The string.

Returns

Return type str

plonk.utils.strings.**time_string**(*snap*, *unit*, *unit_str=None*, *float_format='.0f'*)
Generate time stamp string.

Parameters

- **snap** – The Snap object.
- **unit** (*str*) – The time unit as a string to pass to Pint. E.g. 'year'.
- **unit_str** (*Optional[str]*) – The unit string to print. If None, use the value from 'unit'. Default is None.
- **float_format** (*str*) – The format for the time float value. Default is '.0f'.

Return type str

Examples

Generate a list of strings of snapshot time, like ['0 yr', '10 yr', ...].

```
>>> text = [time_string(snap, 'year', 'yr') for snap in snaps]
```

Or, in terms of an orbital time, like '10 orbits'.

```
>>> plonk.units.define('binary_orbit = 100 years')
>>> time_string(snap, 'binary_orbit', 'orbits')
```

Visualize

Utils for visualize.

plonk.utils.visualize.**cartesian_to_polar**(*interpolated_data_cartesian*, *extent_cartesian*)
Convert interpolated Cartesian pixel grid to polar coordinates.

Parameters

- **interpolated_data_cartesian** (*numpy.ndarray*) – The interpolated data on a Cartesian grid.

- **extent_cartesian** (*Tuple[float, float, float, float]*) – The extent in Cartesian space as (xmin, xmax, ymin, ymax). It must be square.

Returns

- *interpolated_data_polar* – The interpolated data on a polar grid (R, phi).
- *extent_polar* – The extent on a polar grid (0, Rmax, 0, 2).

Return type Tuple[numpy.ndarray, Tuple[float, float, float, float]]

plonk.utils.visualize.**get_extent_from_percentile**(*snap*, *x*, *y*, *percentile=99*, *x_center_on=None*, *y_center_on=None*, *edge_factor=None*)

Get extent from percentile.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **x** (*str*) – The "x" coordinate.
- **y** (*str*) – The "y" coordinate.
- **percentile** (*optional*) – The percentile used in the calculation. Default is 99.
- **x_center_on** (*optional*) – Center on some x-value. Default is None.
- **y_center_on** (*optional*) – Center on some y-value. Default is None.
- **edge_factor** (*optional*) – Add extra spacing to extent. E.g. to add extra 5%, set this value to 0.05. Default is None.

Returns The extent of the box as (xmin, xmax, ymin, ymax).

Return type tuple

plonk.utils.visualize.**plot_smoothing_length**(*snap*, *indices*, *fac=1.0*, *units=None*, *x='x'*, *y='y'*, *ax=None*, ***kwargs*)

Plot smoothing length around particle.

Also works for plotting the accretion radius on a sink particle.

Parameters

- **snap** (*SnapLike*) – The Snap object.
- **indices** (*List[int]*) – The particle indices.
- **fac** (*float*) – Set the circle to be a multiple "fac" of the smoothing length.
- **units** (*Union[str, Quantity]*) – The distance units.
- **ax** (*Any*) – The matplotlib Axes to plot on.
- **x** (*str*) – The "x" coordinate.
- **y** (*str*) – The "y" coordinate.
- ****kwargs** – Keyword arguments to pass to matplotlib PatchCollection.

Returns A list of matplotlib circles.

Return type circles

CHAPTER

THREE

CITATION

If you use Plonk in a scientific publication, please cite the paper published in JOSS.

> Mentiplay, (2019). Plonk: Smoothed particle hydrodynamics analysis and visualization with Python. Journal of Open Source Software, 4(44), 1884, https://doi.org/10.21105/joss.01884.

For BibTeX entry, see CITATION.bib.

If you use the interpolation to pixel grid component of Plonk please cite the Splash paper.

CHAPTER

FOUR

CHANGE LOG

The change log is available at CHANGELOG.md.

CHAPTER

FIVE

CONTRIBUTORS

The main author is Daniel Mentiplay.

If you would like to contribute, see CONTRIBUTING.md.

CHAPTER

SIX

LICENSE

Copyright 2019-2021 Daniel Mentiplay and contributors.

Plonk is available under the MIT license. For details see LICENSE.

CHAPTER

SEVEN

OTHER PACKAGES

Here are some other, mature, Python analysis and visualization packages for smoothed particle hydrodynamics, and other scientific, data:

- pynbody — "an analysis package for astrophysical N-body and Smooth Particle Hydrodynamics simulations".
- Py-SPHViewer — "a parallel Python package to visualise and explore N-body + Hydrodynamics simulations using the Smoothed Particle Hydrodynamics (SPH) scheme".
- yt project — "yt is an open-source, permissively-licensed Python package for analyzing and visualizing volumetric data".

In addition, Splash is a mature, Unix command line, "free and open source visualisation tool for Smoothed Particle Hydrodynamics (SPH) simulations", written in Fortran.

CHAPTER

EIGHT

INDEX

- genindex

PYTHON MODULE INDEX

p

plonk.analysis.discs, 84
plonk.analysis.filters, 75
plonk.analysis.particles, 77
plonk.analysis.sinks, 83
plonk.analysis.sph, 87
plonk.analysis.total, 81
plonk.utils.kernels, 99
plonk.utils.math, 100
plonk.utils.snap, 101
plonk.utils.strings, 103
plonk.utils.visualize, 103

INDEX

A

accreted_mass() (*in module plonk.analysis.total*), 81
add_alias() (*plonk.Profile method*), 71
add_alias() (*plonk.Snap method*), 53
add_aliases() (*in module plonk.utils.snap*), 101
add_array() (*plonk.Snap method*), 53
add_profile() (*plonk.Profile method*), 71
add_quantities() (*plonk.Snap method*), 53
add_unit() (*plonk.Snap method*), 53
angular_momentum() (*in module plonk.analysis.particles*), 77
angular_momentum() (*in module plonk.analysis.total*), 81
angular_velocity() (*in module plonk.analysis.particles*), 77
animate() (*in module plonk*), 93
animation_images() (*in module plonk.visualize*), 94
animation_particles() (*in module plonk.visualize*), 95
animation_profiles() (*in module plonk.visualize*), 96
annulus() (*in module plonk.analysis.filters*), 75
array() (*plonk.Sinks method*), 65
array() (*plonk.Snap method*), 54
array_code_unit() (*plonk.Snap method*), 54
array_in_code_units() (*plonk.Snap method*), 54
available_arrays() (*plonk.Sinks method*), 65
available_arrays() (*plonk.Snap method*), 54
available_profiles() (*plonk.Profile method*), 72
average() (*in module plonk.utils.math*), 100
azimuthal_angle() (*in module plonk.analysis.particles*), 77

B

base_array_name() (*plonk.Snap method*), 54
base_profile_name() (*plonk.Profile method*), 72
box() (*in module plonk.analysis.filters*), 75
bulk_load() (*plonk.Snap method*), 55
bulk_unload() (*plonk.Snap method*), 55

C

cache_arrays (*plonk.Snap property*), 55
cartesian_to_polar() (*in module plonk.utils.visualize*), 103
center_of_mass() (*in module plonk.analysis.total*), 81
close_file() (*plonk.Snap method*), 55
code_units (*plonk.Simulation property*), 67
code_units (*plonk.Snap property*), 55
context() (*plonk.Snap method*), 55
cross() (*in module plonk.utils.math*), 100
cylinder() (*in module plonk.analysis.filters*), 75

D

default_units (*plonk.Profile property*), 72
default_units (*plonk.Snap property*), 55
derivative() (*in module plonk.analysis.sph*), 87
distance_from_plane() (*in module plonk.utils.math*), 100
dust_array_names() (*in module plonk.utils.snap*), 101
dust_density() (*in module plonk.analysis.particles*), 77
dust_mass() (*in module plonk.analysis.particles*), 77
dust_mass() (*in module plonk.analysis.total*), 81

E

eccentricity() (*in module plonk.analysis.discs*), 84
eccentricity() (*in module plonk.analysis.sinks*), 83

F

family() (*plonk.Snap method*), 55

G

gas_density() (*in module plonk.analysis.particles*), 78
gas_fraction() (*in module plonk.analysis.particles*), 78
gas_mass() (*in module plonk.analysis.particles*), 78
gas_mass() (*in module plonk.analysis.total*), 81
get_extent_from_percentile() (*in module plonk.utils.visualize*), 104
gravitational_constant_in_code_units() (*in module plonk.utils.snap*), 102
gravitational_potential_energy() (*in module plonk.analysis.sinks*), 83

H

Hill_radius() (*in module plonk.analysis.sinks*), 83

I

image() (*in module plonk*), 88
image() (*plonk.Snap method*), 56
inclination() (*in module plonk.analysis.discs*), 85
inclination() (*in module plonk.analysis.sinks*), 83
inclination_angle() (*in module plonk.analysis.discs*), 85
indices (*plonk.Sinks property*), 65
indices (*plonk.SubSnap property*), 65
interpolate() (*in module plonk*), 97
is_documented_by() (*in module plonk.utils.strings*), 103

K

keplerian_frequency() (*in module plonk.analysis.discs*), 85
kernel_cubic() (*in module plonk.utils.kernels*), 99
kernel_gradient_cubic() (*in module plonk.utils.kernels*), 99
kernel_gradient_quintic() (*in module plonk.utils.kernels*), 99
kernel_gradient_wendland_c4() (*in module plonk.utils.kernels*), 99
kernel_quintic() (*in module plonk.utils.kernels*), 100
kernel_wendland_c4() (*in module plonk.utils.kernels*), 100
kinetic_energy() (*in module plonk.analysis.particles*), 78
kinetic_energy() (*in module plonk.analysis.sinks*), 83
kinetic_energy() (*in module plonk.analysis.total*), 81

L

load_profile() (*in module plonk*), 73
load_simulation() (*in module plonk*), 69
load_simulation() (*plonk.Simulation method*), 67
load_snap() (*in module plonk*), 67
load_snap() (*plonk.Snap method*), 57
load_time_series() (*in module plonk*), 69
loaded_arrays() (*plonk.Sinks method*), 65
loaded_arrays() (*plonk.Snap method*), 58
loaded_profiles() (*plonk.Profile method*), 72

M

mass() (*in module plonk.analysis.total*), 82
mean_motion() (*in module plonk.analysis.sinks*), 84
module
    plonk.analysis.discs, 84
    plonk.analysis.filters, 75
    plonk.analysis.particles, 77
    plonk.analysis.sinks, 83
    plonk.analysis.sph, 87
    plonk.analysis.total, 81
    plonk.utils.kernels, 99
    plonk.utils.math, 100
    plonk.utils.snap, 101
    plonk.utils.strings, 103
    plonk.utils.visualize, 103
momentum() (*in module plonk.analysis.particles*), 79
momentum() (*in module plonk.analysis.total*), 82

N

neighbours() (*plonk.Snap method*), 58
norm() (*in module plonk.utils.math*), 101
num_dust_species (*plonk.Snap property*), 58
num_particles (*plonk.Snap property*), 58
num_particles_of_type (*plonk.Snap property*), 58
num_sinks (*plonk.Snap property*), 58

O

orbital_period() (*in module plonk.analysis.sinks*), 84

P

particle_indices() (*plonk.Snap method*), 58
particles_to_binned_quantity() (*plonk.Profile method*), 72
plonk.analysis.discs
    module, 84
plonk.analysis.filters
    module, 75
plonk.analysis.particles
    module, 77
plonk.analysis.sinks
    module, 83
plonk.analysis.sph
    module, 87
plonk.analysis.total
    module, 81
plonk.utils.kernels
    module, 99
plonk.utils.math
    module, 100
plonk.utils.snap
    module, 101
plonk.utils.strings
    module, 103
plonk.utils.visualize
    module, 103
plot() (*in module plonk*), 89
plot() (*plonk.Profile method*), 72
plot() (*plonk.Sinks method*), 66
plot() (*plonk.Snap method*), 58
plot_smoothing_length() (*in module plonk.utils.visualize*), 104

polar_angle() (*in module plonk.analysis.particles*), 79
position_angle() (*in module plonk.analysis.discs*), 85
pretty_array_name() (*in module plonk.utils.strings*), 103
Profile (*class in plonk*), 70
properties (*plonk.Simulation property*), 68
properties (*plonk.Snap property*), 59

R

radius_cylindrical() (*in module plonk.analysis.particles*), 79
radius_spherical() (*in module plonk.analysis.particles*), 79
read_extra_arrays() (*plonk.Snap method*), 59
reopen_file() (*plonk.Snap method*), 60
reset() (*plonk.Snap method*), 60
Roche_sphere() (*in module plonk.analysis.sinks*), 83
rotate() (*plonk.Snap method*), 60
rotate_edge_on() (*in module plonk.analysis.discs*), 85
rotate_face_on() (*in module plonk.analysis.discs*), 86

S

scalar_interpolation() (*in module plonk.visualize.interpolation*), 98
semi_major_axis() (*in module plonk.analysis.discs*), 86
semi_major_axis() (*in module plonk.analysis.sinks*), 84
set_central_body() (*plonk.Snap method*), 60
set_kernel() (*plonk.Snap method*), 60
set_molecular_weight() (*plonk.Snap method*), 61
set_units() (*plonk.Profile method*), 73
set_units() (*plonk.Snap method*), 61
set_units_on_time_series() (*plonk.Simulation method*), 68
shell() (*in module plonk.analysis.filters*), 76
Simulation (*class in plonk*), 67
Sinks (*class in plonk*), 65
sinks (*plonk.Snap property*), 61
sinks (*plonk.SubSnap property*), 65
Snap (*class in plonk*), 52
snaps (*plonk.Simulation property*), 68
specific_angular_momentum() (*in module plonk.analysis.particles*), 79
specific_angular_momentum() (*in module plonk.analysis.sinks*), 84
specific_angular_momentum() (*in module plonk.analysis.total*), 82
specific_kinetic_energy() (*in module plonk.analysis.particles*), 80
specific_kinetic_energy() (*in module plonk.analysis.total*), 82
specific_orbital_energy() (*in module plonk.analysis.sinks*), 84
sphere() (*in module plonk.analysis.filters*), 76
stokes_number() (*in module plonk.analysis.discs*), 86
SubSnap (*class in plonk*), 64
subsnaps_as_dict() (*plonk.Snap method*), 61
subsnaps_as_list() (*plonk.Snap method*), 61
summation() (*in module plonk.analysis.sph*), 87

T

temperature() (*in module plonk.analysis.particles*), 80
time_series (*plonk.Simulation property*), 68
time_string() (*in module plonk.utils.strings*), 103
to_array() (*plonk.Simulation method*), 68
to_dataframe() (*plonk.Profile method*), 73
to_dataframe() (*plonk.Snap method*), 61
to_function() (*plonk.Profile method*), 73
translate() (*plonk.Snap method*), 62
tree (*plonk.Snap property*), 62

U

unit_normal() (*in module plonk.analysis.discs*), 86
unset_units_on_time_series() (*plonk.Simulation method*), 68

V

vector() (*in module plonk*), 90
vector() (*plonk.Snap method*), 62
vector_array_names() (*in module plonk.utils.snap*), 102
vector_interpolation() (*in module plonk.visualize.interpolation*), 98
velocity_radial_cylindrical() (*in module plonk.analysis.particles*), 80
velocity_radial_spherical() (*in module plonk.analysis.particles*), 80
visualize() (*plonk.Simulation method*), 68
visualize_sim() (*in module plonk*), 92

W

write_extra_arrays() (*plonk.Snap method*), 64