SWIFTsimIO

Release 7.0.0

Josh Borrow

Nov 09, 2023

CONTENTS

`swiftsimio` is a toolkit for reading SWIFT data, an astrophysics simulation code. It is used to ensure that everything that you read has a symbolic unit attached, and can be used for visualisation. The final key feature that it enables is the use of the cell metadata in `SWIFT` snapshots to enable partial reading.

CHAPTER

ONE

GETTING STARTED

The SWIFT astrophysical simulation code (http://swift.dur.ac.uk) is used widely. There exists many ways of reading the data from SWIFT, which outputs HDF5 files. These range from reading directly using `h5py` to using a complex system such as `yt`; however these either are unsatisfactory (e.g. a lack of unit information in reading HDF5), or too complex for most use-cases. This (thin) wrapper provides an object-oriented API to read (dynamically) data from SWIFT.

Getting set up with *`swiftsimio`* is easy; it (by design) has very few requirements. There are a number of optional packages that you can install to make the experience better and these are recommended. All requirements are detailed below.

1.1 Requirements

This requires `python` `v3.8.0` or higher. Unfortunately it is not possible to support *`swiftsimio`* on versions of python lower than this. It is important that you upgrade if you are still a `python2` user.

1.1.1 Python packages

- `numpy`, required for the core numerical routines.
- `h5py`, required to read data from the SWIFT HDF5 output files.
- `unyt`, required for symbolic unit calculations (depends on `sympy`).

1.1.2 Optional packages

- `numba`, highly recommended should you wish to use the in-built visualisation tools.
- `scipy`, required if you wish to generate smoothing lengths for particle types that do not store this variable in the snapshots (e.g. dark matter)
- `tqdm`, required for progress bars for some long-running tasks. If not installed no progress bar will be shown.

1.2 Installing

swiftsimio can be installed using the python packaging manager, `pip`, or any other packaging manager that you wish to use:

```
pip install swiftsimio
```

Note that this will install any required packages for you.

To set up the code for development, first clone the latest master from GitHub:

```
git clone https://github.com/SWIFTSIM/swiftsimio.git
```

and install with `pip` using the `-e` flag,

```
cd swiftsimio
pip install -e .
```

1.3 Usage

There are many examples of using *swiftsimio* available in the swiftsimio_examples repository, which also includes examples for reading older (e.g. EAGLE) datasets.

Example usage is shown below, which plots a density-temperature phase diagram, with density and temperature given in CGS units:

```
import swiftsimio as sw

# This loads all metadata but explicitly does _not_ read any particle data yet
data = sw.load("/path/to/swift/output")

import matplotlib.pyplot as plt

data.gas.densities.convert_to_cgs()
data.gas.temperatures.convert_to_cgs()

plt.loglog()

plt.scatter(
   data.gas.densities,
   data.gas.temperatures,
   s=1
)

plt.xlabel(fr"Gas density $\left[{data.gas.densities.units.latex_repr}\right]$")
plt.ylabel(fr"Gas temperature $\left[{data.gas.temperatures.units.latex_repr}\right]$")

plt.tight_layout()

plt.savefig("test_plot.png", dpi=300)
```

Don't worry too much about this for now if you can't understand it, we will get into this much more heavily in the next section.

In the above it's important to note the following:

- All metadata is read in when the *swiftsimio.load()* function is called.
- Only the density and temperatures (corresponding to the `PartType0/Densities` and `PartType0/Temperatures`) datasets are read in.
- That data is only read in once the `swiftsimio.objects.cosmo_array.convert_to_cgs()` method is called.
- `swiftsimio.objects.cosmo_array.convert_to_cgs()` converts data in-place; i.e. it returns *None*.
- The data is cached and not re-read in when `plt.scatter` is called.

CHAPTER

TWO

LOADING DATA

The main purpose of *swiftsimio* is to load data. This section will tell you all about four main objects:

- *swiftsimio.reader.SWIFTUnits*, responsible for creating a correspondence between the SWIFT units and `unyt` objects.
- *swiftsimio.reader.SWIFTMetadata*, responsible for loading any required information from the SWIFT headers into python-readable data.
- *swiftsimio.reader.SWIFTDataset*, responsible for holding all particle data, and keeping track of the above two objects.
- *swiftsimio.reader.SWIFTParticleTypeMetadata*, responsible for cataloguing metadata just about individual particle types, like gas, including what particle fields are present.

To get started, first locate any SWIFT data that you wish to analyse. If you don't have any handy, you can always download our test cosmological volume at:

`http://virgodb.cosma.dur.ac.uk/swift-webstorage/IOExamples/cosmo_volume_example.hdf5`

with associated halo catalogue at

`http://virgodb.cosma.dur.ac.uk/swift-webstorage/IOExamples/cosmo_volume_example.properties`

which is needed should you wish to use the `velociraptor` integration library in the visualisation examples.

To create your first instance of *swiftsimio.reader.SWIFTDataset*, you can use the helper function *swiftsimio.load* as follows:

```
from swiftsimio import load

# Of course, replace this path with your own snapshot should you be using
# custom data.
data = load("cosmo_volume_example.hdf5")
```

The type of `data` is now *swiftsimio.reader.SWIFTDataset*. Have a quick look around this dataset in an `iPython` shell, or a `jupyter` notebook, and you will see that it contains several sub-objects:

- `data.gas`, which contains all information about gas particles in the simulation.
- `data.dark_matter`, likewise containing information about the dark matter particles in the simulation.
- `data.metadata`, an instance of *swiftsimio.reader.SWIFTMetadata*
- `data.units`, an instance of *swiftsimio.reader.SWIFTUnits*

2.1 Using metadata

Let's begin by reading some useful metadata straight out of our `data.metadata` object. For instance, we may want to know the box-size of our simulation:

```
meta = data.metadata
boxsize = meta.boxsize

print(boxsize)
```

This will output `[142.24751067 142.24751067 142.24751067] Mpc` - note the units that are attached. These units being attached to everything is one of the key advantages of using *swiftsimio*. It is really easy to convert between units; for instance if we want that box-size in kiloparsecs,

```
boxsize.convert_to_units("kpc")

print(boxsize)
```

Now outputting `[142247.5106242 142247.5106242 142247.5106242] kpc`. Neat! This is all thanks to our tight integration with `unyt`. If you have more complex units, it is often useful to specify them in terms of `unyt` objects as follows:

```
import unyt

new_units = unyt.cm * unyt.Mpc / unyt.kpc
new_units.simplify()

boxsize.convert_to_units(new_units)
```

In general, we suggest using `unyt` unit objects rather than strings. You can find more information about `unyt` on the unyt documentation website.

There is lots of metadata available through this object; the best way to see this is by exploring the object using `dir()` in an interactive shell, but as a summary:

- All metadata from the snapshot is available through many variables, for example the `meta.hydro_scheme` property.
- The numbers of particles of different types are available through `meta.n_{gas,dark_matter,stars,black_holes}`.
- Several pre-LaTeXed strings are available describing the configuration state of the code, such as `meta.hydro_info`, `meta.compiler_info`.
- Several snapshot-wide parameters, such as `meta.a` (current scale factor), `meta.t` (current time), `meta.z` (current redshift), `meta.run_name` (the name of this run, specified in the SWIFT parameter file), and `meta.snapshot_date` (a `datetime` object describing when the snapshot was written to disk).
- If you have `astropy` installed, you can also use the `metadata.cosmology` object, which is an `astropy.cosmology.w0waCDM` instance.

2.2 Reading particle data

To find out what particle properties are present in our snapshot, we can use the instance of *swiftsimio.reader.SWIFTMetadata*, `data.metadata`, which contains several instances of *swiftsimio.reader.SWIFTParticleTypeMetadata* describing what kinds of fields are present in gas or dark matter:

```
# Note that gas_properties is an instance of SWIFTParticleTypeMetadata
print(data.metadata.gas_properties.field_names)
```

This will print a large list, like

```
['coordinates',
'densities',
...
'temperatures',
'velocities']
```

These individual attributes can be accessed through the object-oriented interface, for instance,

```
x_gas = data.gas.coordinates
rho_gas = data.gas.densities
x_dm = data.dark_matter.coordinates
```

Only at this point is any information actually read from the snapshot, so far we have only read three arrays into memory - in this case corresponding to `/PartType0/Coordinates`, `/PartType1/Coordinates`, and `/PartType0/Densities`.

This allows you to be quite lazy when writing scripts; you do not have to write, for instance, a block at the start of the file with a `with h5py.File(...) as handle:` and read all of the data at once, you can simply access data whenever you need it through this predictable interface.

Just like the boxsize, these carry symbolic `unyt` units,

```
print(x_gas.units)
```

will output `Mpc`. We can again convert these to whatever units we like. For instance, should we wish to convert our gas densities to solar masses per cubic megaparsec,

```
new_density_units = unyt.Solar_Mass / unyt.Mpc**3

rho_gas.convert_to_units(new_density_units)

print(rho_gas.units.latex_repr)
```

which will output `'\\frac{M_\\odot}{\\rm{Mpc}^{3}}'`. This is a LaTeX representation of those symbolic units that we just converted our data to - this is very useful for making plots as it can ensure that your data and axes labels always have consistent units.

2.3 Named columns

SWIFT can output custom metadata in `SubgridScheme/NamedColumns` for multi dimensional tables containing columns that carry individual data. One common example of this is the element mass fractions of gas and stellar particles. These are then placed in an object hierarchy, as follows:

```
print(data.gas.element_mass_fractions)
```

This will output: Named columns instance with ['hydrogen', 'helium', 'carbon', 'nitrogen', 'oxygen', 'neon', 'magnesium', 'silicon', 'iron'] available for "Fractions of the particles' masses that are in the given element"

Then, to access individual columns (in this case element abundances):

```
# Access the silicon abundance
data.gas.element_mass_fractions.silicon
```

2.4 Non-unyt properties

Each data array has some custom properties that are not present within the base `unyt.unyt_array` class. We create our own version of this in *swiftsimio.objects.cosmo_array*, which allows each dataset to contain its own cosmology and name properties.

For instance, should you ever need to know what a dataset represents, you can ask for a description:

```
print(rho_gas.name)
```

which will output `Co-moving mass densities of the particles`. They include scale-factor information, too, through the `cosmo_factor` object,

```
# Conversion factor to make the densities a physical quantity
print(rho_gas.cosmo_factor.a_factor)
physical_rho_gas = rho_gas.cosmo_factor.a_factor * rho_gas

# Symbolic scale-factor expression
print(rho_gas.cosmo_factor.expr)
```

which will output `132651.002785671` and `a**(-3.0)`. This is an easy way to convert your co-moving values to physical ones.

An even easier way to convert your properties to physical is to use the built-in `to_physical` and `convert_to_physical` methods, as follows:

```
physical_rho_gas = rho_gas.to_physical()

# Convert in-place
rho_gas.convert_to_physical()
```

2.5 User-defined particle types

It is now possible to add user-defined particle types that are not already present in the *swiftsimio* metadata. All you need to do is specify the three names (see below) and then the particle datasets that you have provided in SWIFT will be automatically read.

```
import swiftsimio as sw
import swiftsimio.metadata.particle as swp
from swiftsimio.objects import cosmo_factor, a

swp.particle_name_underscores[6] = "extratype"
swp.particle_name_class[6] = "Extratype"
swp.particle_name_text[6] = "Extratype"

data = sw.load(
    "extra_test.hdf5",
)
```

CHAPTER

THREE

MASKING

swiftsimio provides unique functionality (when compared to other software packages that read HDF5 data) through its masking facility.

SWIFT snapshots contain cell metadata that allow us to spatially mask the data ahead of time. *swiftsimio* provides a number of objects that help with this. This functionality is provided through the *swiftsimio.masks* sub-module but is available easily through the *swiftsimio.mask()* top-level function.

This functionality is used heavily in our VELOCIraptor integration library for only reading data that is near bound objects.

There are two types of mask, with the default only allowing spatial masking. Full masks require significantly more memory overhead and are generally much slower than the spatial only mask.

3.1 Spatial-only masking

Spatial only masking is approximate and allows you to only load particles within a given region. It is precise to the top-level cells that are defined within SWIFT. It will always load all of the particles that you request, but for simplicity it may also load some particles that are slightly outside of the region of interest. This is because it works as follows:

1. Load the top-level cell metadata.
2. Find the overlap between the specified region and these cells.
3. Load all cells within that overlap.

As you can see, the edges of regions may load in extra information as we always load the whole top-level cell.

3.1.1 Example

In this example we will use the *swiftsimio.masks.SWIFTMask* object to load the bottom left 'half' corner of the box.

```
import swiftsimio as sw

filename = "cosmo_volume_example.hdf5"

mask = sw.mask(filename)
# The full metadata object is available from within the mask
boxsize = mask.metadata.boxsize
# load_region is a 3x2 list [[left, right], [bottom, top], [front, back]]
load_region = [[0.0 * b, 0.5 * b] for b in boxsize]
```

(continues on next page)

(continued from previous page)

```
# Constrain the mask
mask.constrain_spatial(load_region)

# Now load the snapshot with this mask
data = sw.load(filename, mask=mask)
```

`data` is now a regular *swiftsimio.reader.SWIFTDataset* object, but it only ever loads particles that are (approximately) inside the `load_region` region.

Importantly, this method has a tiny memory overhead, and should also have a relatively small overhead when reading the data. This allows you to use snapshots that are much larger than the available memory on your machine and process them with ease.

It is also possible to build up a region with a more complicated geometry by making repeated calls to `constrain_spatial()` and setting the optional argument *intersect=True*. By default any existing selection of cells would be overwritten; this option adds any additional cells that need to be selected for the new region to the existing selection instead. For instance, to add the diagonally opposed octant to the selection made above (and so obtain a region shaped like two cubes with a single corner touching):

```
additional_region = [[0.5 * b, 1.0 * b] for b in boxsize]
mask.constrain_spatial(additional_region, intersect=True)
```

In the first call to `constrain_spatial()` the *intersect* argument can be set to *True* or left *False* (the default): since no mask yet exists both give the same result.

3.2 Full mask

The below example shows the use of a full masking object, used to constrain densities of particles and only load particles within that density window.

```
import swiftsimio as sw

# This creates and sets up the masking object.
mask = sw.mask("cosmological_volume.hdf5", spatial_only=False)

# This ahead-of-time creates a spatial mask based on the cell metadata.
mask.constrain_spatial([
    [0.2 * mask.metadata.boxsize[0], 0.7 * mask.metadata.boxsize[0]],
    None,
    None]
)

# Now, just for fun, we also constrain the density between
# 0.4 g/cm^3 and 0.8. This reads in the relevant data in the region,
# and tests it element-by-element. Note that using masks of this type
# is significantly slower than using the spatial-only masking.
density_units = mask.units.mass / mask.units.length**3
mask.constrain_mask("gas", "density", 0.4 * density_units, 0.8 * density_units)

# Now we can grab the actual data object. This includes the mask as a parameter.
data = sw.load("cosmo_volume_example.hdf5", mask=mask)
```

When the attributes of this data object are accessed, *only* the ones that belong to the masked region (in both density and spatial) are read. I.e. if I ask for the temperature of particles, it will recieve an array containing temperatures of particles that lie in the region [0.2, 0.7] and have a density between 0.4 and 0.8 g/cm^3.

3.3 Writing subset of snapshot

In some cases it may be useful to write a subset of an existing snapshot to its own hdf5 file. This could be used, for example, to extract a galaxy halo that is of interest from a snapshot so that the file is easier to work with and transport.

To do this the `write_subset` function is provided. It can be used, for example, as follows

```
import swiftsimio as sw
import unyt

mask = sw.mask("eagle_snapshot.hdf5")
mask.constrain_spatial([
    [unyt.unyt_quantity(100, unyt.kpc), unyt.unyt_quantity(1000, unyt.kpc)],
    None,
    None])

sw.subset_writer.write_subset("test_subset.hdf5", mask)
```

This will write a snapshot which contains the particles from the specified snapshot whose x-coordinate is within the range [100, 1000] kpc. This function uses the cell mask which encompases the specified spatial domain to successively read portions of datasets from the input file and writes them to a new snapshot.

Due to the coarse grained nature of the cell mask, particles from outside this range may also be included if they are within the same top level cells as particles that fall within the given range.

Please note that it is important to run `constrain_spatial` as this generates and stores the cell mask needed to write the snapshot subset.

CHAPTER

FOUR

VISUALISATION

swiftsimio provides visualisation routines accelerated with the `numba` module. They work without this module, but we strongly recommend installing it for the best performance (1000x+ speedups). These are provided in the `swiftismio.visualisation` sub-modules.

The three built-in rendering types (described below) have the following common interface:

```
{render_func_name}_gas(
    data=data, # SWIFTsimIO dataset
    resolution=1024, # Resolution along one axis of the output image
    project="masses", # Variable to project, e.g. masses, temperatures, etc.
    parallel=False, # Construct the image in (thread) parallel?
    region=None, # None, or a list telling which region to render_func_name
    periodic=True, # Whether or not to apply periodic boundary conditions
)
```

The output of these functions comes with associated units and has the correct dimensions. There are lower-level APIs (also documented here) that provide additional functionality.

4.1 Projection

The *swiftsimio.visualisation.projection* sub-module provides an interface to render SWIFT data projected to a grid. This takes your 3D data and projects it down to 2D, such that if you request masses to be smoothed then these functions return a surface density.

This effectively solves the equation:

$\tilde{A}_i = \sum_j A_j W_{ij,2D}$

with $\tilde{A}_i$ the smoothed quantity in pixel i, and j all particles in the simulation, with W the 2D kernel. Here we use the Wendland-C2 kernel.

The primary function here is *swiftsimio.visualisation.projection.project_gas()*, which allows you to create a gas projection of any field. See the example below.

4.1.1 Example

```
from swiftsimio import load
from swiftsimio.visualisation.projection import project_gas

data = load("cosmo_volume_example.hdf5")

# This creates a grid that has units msun / Mpc^2, and can be transformed like
# any other unyt quantity
mass_map = project_gas(
    data,
    resolution=1024,
    project="masses",
    parallel=True,
    periodic=True,
)

# Let's say we wish to save it as msun / kpc^2,
from unyt import msun, kpc
mass_map.convert_to_units(msun / kpc**2)

from matplotlib.pyplot import imsave
from matplotlib.colors import LogNorm

# Normalize and save
imsave("gas_surface_dens_map.png", LogNorm()(mass_map.value), cmap="viridis")
```

This basic demonstration creates a mass surface density map.

To create, for example, a projected temperature map, we need to remove the surface density dependence (i.e. `project_gas()` returns a surface temperature in units of K / kpc^2 and we just want K) by dividing out by this:

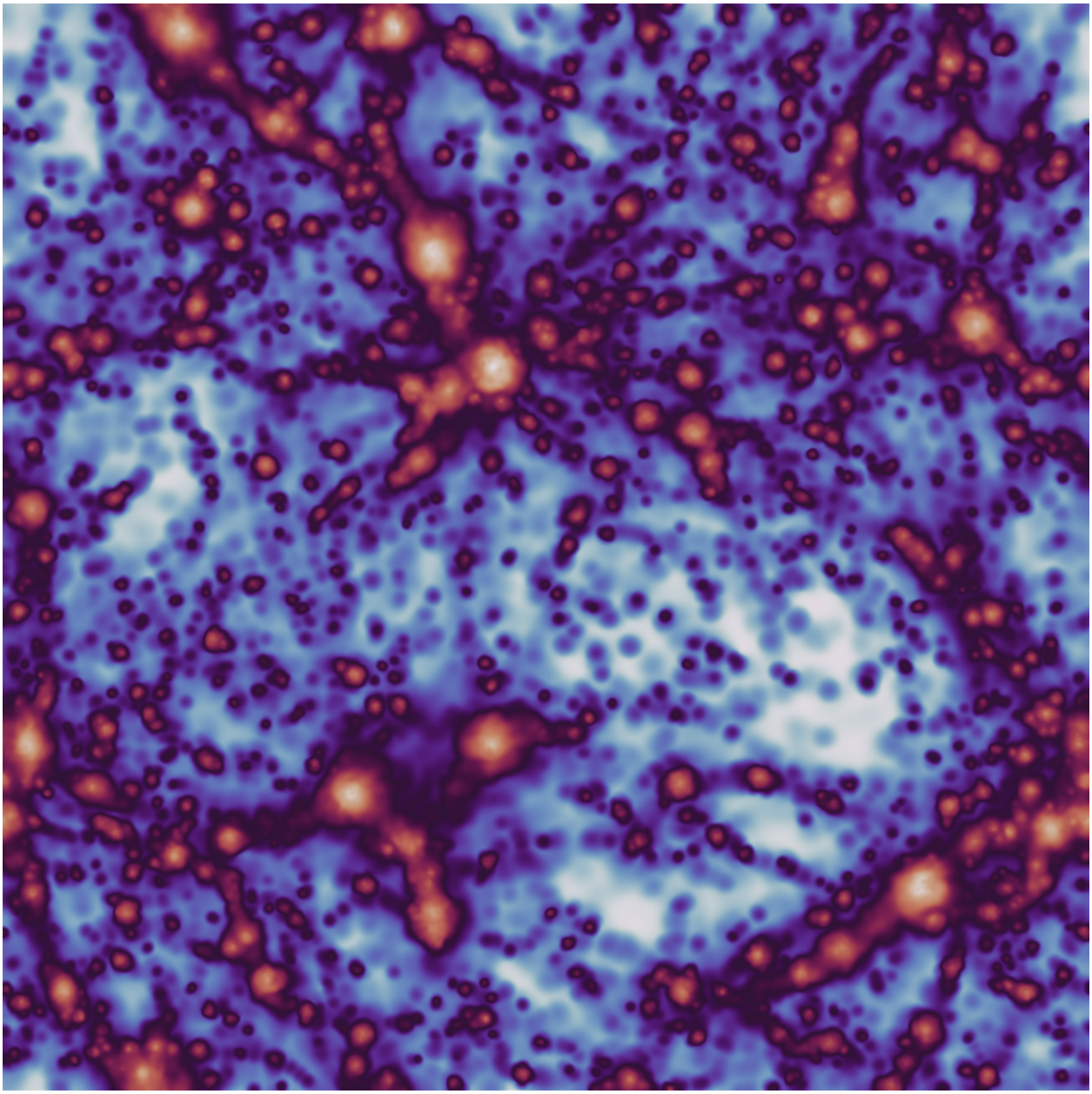
```
from swiftsimio import load
from swiftsimio.visualisation.projection import project_gas

data = load("cosmo_volume_example.hdf5")

# First create a mass-weighted temperature dataset
data.gas.mass_weighted_temps = data.gas.masses * data.gas.temperatures

# Map in msun / mpc^2
mass_map = project_gas(
    data,
    resolution=1024,
    project="masses",
    parallel=True,
    periodic=True,
)
# Map in msun * K / mpc^2
mass_weighted_temp_map = project_gas(
    data,
    resolution=1024,
    project="mass_weighted_temps",
    parallel=True,
```

(continues on next page)

(continued from previous page)

```
    periodic=True,
)

temp_map = mass_weighted_temp_map / mass_map

from unyt import K
temp_map.convert_to_units(K)

from matplotlib.pyplot import imsave
from matplotlib.colors import LogNorm

# Normalize and save
imsave("temp_map.png", LogNorm()(temp_map.value), cmap="twilight")
```

The output from this example, when used with the example data provided in the loading data section should look something like:



4.1.2 Backends

In certain cases, rather than just using this facility for visualisation, you will wish that the values that are returned to be as well converged as possible. For this, we provide several different backends. These are passed as `backend="str"` to all of the projection visualisation functions, and are available in the module `swiftsimio.visualisation.projection.projection_backends`. The available backends are as follows:

- `fast`: The default backend - this is extremely fast, and provides very basic smoothing, with a return type of single precision floating point numbers.
- `histogram`: This backend provides zero smoothing, and acts in a similar way to the `np.hist2d` function but with the same arguments as `scatter`.
- `reference`: The same backend as `fast` but with two distinguishing features; all calculations are performed in

double precision, and it will return early with a warning message if there are not enough pixels to fully resolve each kernel. Regular users should not use this mode.

- `renormalised`: The same as `fast`, but each kernel is evaluated twice and renormalised to ensure mass conservation within floating point precision. Returns single precision arrays.
- `subsampled`: This is the recommended mode for users who wish to have converged results even at low resolution. Each kernel is evaluated at least 32 times, with overlaps between pixels considered for every single particle. Returns in double precision.
- `subsampled_extreme`: The same as `subsampled`, but provides 64 kernel evaluations.
- `gpu`: The same as `fast` but uses CUDA for faster computation on supported GPUs. The parallel implementation is the same function as the non-parallel.

Example:

```
from swiftsimio import load
from swiftsimio.visualisation.projection import project_gas

data = load("cosmo_volume_example.hdf5")

subsampled_array = project_gas(
   data,
   resolution=1024,
   project="entropies",
   parallel=True,
   backend="subsampled",
   periodic=True,
)
```

This will likely look very similar to the image that you make with the default `backend="fast"`, but will have a well-converged distribution at any resolution level.

4.1.3 Periodic boundaries

Cosmological simulations and many other simulations use periodic boundary conditions. This has implications for the particles at the edge of the simulation box: they can contribute to pixels on multiple sides of the image. If this effect is not taken into account, then the pixels close to the edge will have values that are too low because of missing contributions.

All visualisation functions by default assume a periodic box. Rather than simply projecting each individual particle once, four additional periodic copies of each particle are also projected. Most copies will project outside the valid pixel range, but the copies that do not ensure that pixels close to the edge receive all necessary contributions. Thanks to Numba optimisations, the overhead of these additional copies is relatively small.

There are some caveats with this approach. If you try to visualise a subset of the particles in the box (e.g. using a mask), then only periodic copies of particles in this subset will be used. If the subset does not include particles on the other side of the periodic boundary, then these will still be missing from the projection. The same is true if you visualise a region of the box. The periodic boundary wrapping is also not compatible with rotations (see below) and should therefore not be used together with a rotation.

4.1.4 Rotations

Sometimes you will need to visualise a galaxy from a different perspective. The *swiftsimio.visualisation.rotation* sub-module provides routines to generate rotation matrices corresponding to vectors, which can then be provided to the `rotation_matrix` argument of `project_gas()` (and `project_gas_pixel_grid()`). You will also need to supply the `rotation_center` argument, as the rotation takes place around this given point. The example code below loads a snapshot, and a halo catalogue, and creates an edge-on and face-on projection using the integration in `velociraptor`. More information on possible integrations with this library is shown in the `velociraptor` section.

```
from swiftsimio import load, mask
from velociraptor import load as load_catalogue
from swiftsimio.visualisation.rotation import rotation_matrix_from_vector
from swiftsimio.visualisation.projection import project_gas_pixel_grid

import unyt
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

# Radius around which to load data, we will visualise half of this
size = 1000 * unyt.kpc

snapshot_filename = "cosmo_volume_example.hdf5"
catalogue_filename = "cosmo_volume_example.properties"

catalogue = load_catalogue(catalogue_filename)

# Which halo should we visualise?
halo = 0

x = catalogue.positions.xcmbp[halo]
y = catalogue.positions.ycmbp[halo]
z = catalogue.positions.zcmbp[halo]

lx = catalogue.angular_momentum.lx[halo]
ly = catalogue.angular_momentum.ly[halo]
lz = catalogue.angular_momentum.lz[halo]

# The angular momentum vector will point perpendicular to the galaxy disk.
# If your simulation contains stars, use lx_star
angular_momentum_vector = np.array([lx.value, ly.value, lz.value])
angular_momentum_vector /= np.linalg.norm(angular_momentum_vector)

face_on_rotation_matrix = rotation_matrix_from_vector(
   angular_momentum_vector
)
edge_on_rotation_matrix = rotation_matrix_from_vector(
   angular_momentum_vector,
   axis="y"
)

region = [
   [x - size, x + size],
```

(continues on next page)

(continued from previous page)

```
    [y - size, y + size],
    [z - size, z + size],
]

visualise_region = [
    x - 0.5 * size, x + 0.5 * size,
    y - 0.5 * size, y + 0.5 * size,
]

data_mask = mask(snapshot_filename)
data_mask.constrain_spatial(region)
data = load(snapshot_filename, mask=data_mask)

# Use project_gas_pixel_grid to generate projected images

common_arguments = dict(
    data=data,
    resolution=512,
    parallel=True,
    region=visualise_region,
    periodic=False, # disable periodic boundaries when using rotations
)

un_rotated = project_gas_pixel_grid(**common_arguments)

face_on = project_gas_pixel_grid(
    **common_arguments,
    rotation_center=unyt.unyt_array([x, y, z]),
    rotation_matrix=face_on_rotation_matrix,
)

edge_on = project_gas_pixel_grid(
    **common_arguments,
    rotation_center=unyt.unyt_array([x, y, z]),
    rotation_matrix=edge_on_rotation_matrix,
)
```

Using this with the provided example data will just show blobs due to its low resolution nature. Using one of the EAGLE volumes (`examples/EAGLE_ICs`) will produce much nicer galaxies, but that data is too large to provide as an example in this tutorial.

You can also provide an extra two values, the z min and max, as part of the `region` parameter. This may have some slight performance impact, so it is generally advised that you do this on sub-loaded volumes only.

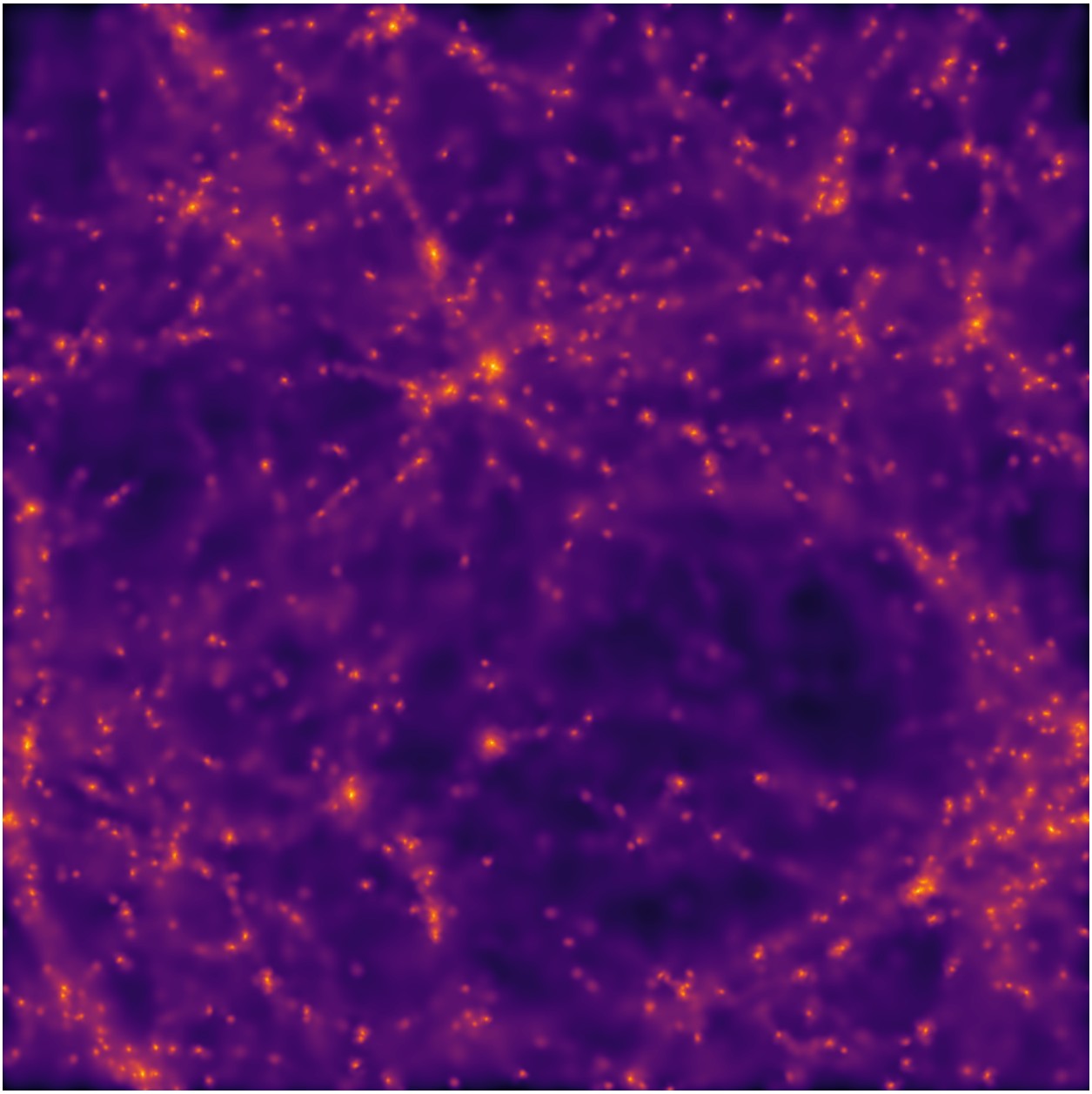
4.1.5 Other particle types

Other particle types are able to be visualised through the use of the *swiftsimio.visualisation.projection.project_pixel_grid()* function. This does not attach correct symbolic units, so you will have to work those out yourself, but it does perform the smoothing. We aim to introduce the feature of correctly applied units to these projections soon.

To use this feature for particle types that do not have smoothing lengths, you will need to generate them, as in the example below where we create a mass density map for dark matter. We provide a utility to do this through *swiftsimio.visualisation.smoothing_length_generation.generate_smoothing_lengths()*.

```
from swiftsimio import load
from swiftsimio.visualisation.projection import project_pixel_grid
from swiftsimio.visualisation.smoothing_length_generation import generate_smoothing_
↪lengths

data = load("cosmo_volume_example.hdf5")

# Generate smoothing lengths for the dark matter
data.dark_matter.smoothing_length = generate_smoothing_lengths(
    data.dark_matter.coordinates,
    data.metadata.boxsize,
    kernel_gamma=1.8,
    neighbours=57,
    speedup_fac=2,
    dimension=3,
)

# Project the dark matter mass
dm_mass = project_pixel_grid(
    # Note here that we pass in the dark matter dataset not the whole
    # data object, to specify what particle type we wish to visualise
    data=data.dark_matter,
    boxsize=data.metadata.boxsize,
    resolution=1024,
    project="masses",
    parallel=True,
    region=None,
    periodic=True,
)

from matplotlib.pyplot import imsave
from matplotlib.colors import LogNorm

# Everyone knows that dark matter is purple
imsave("dm_mass_map.png", LogNorm()(dm_mass), cmap="inferno")
```

The output from this example, when used with the example data provided in the loading data section should look something like:



4.1.6 Lower-level API

The lower-level API for projections allows for any general positions, smoothing lengths, and smoothed quantities, to generate a pixel grid that represents the smoothed version of the data.

This API is available through `swiftsimio.visualisation.projection.scatter()` and `swiftsimio.visualisation.projection.scatter_parallel()` for the parallel version. The parallel version uses significantly more memory as it allocates a thread-local image array for each thread, summing them in the end. Here we will only describe the `scatter` variant, but they behave in the exact same way.

By default this uses the "fast" backend. To use the others, you can select them manually from the module, or by using the `backends` and `backends_parallel` dictionaries in *swiftsimio.visualisation.projection*.

To use this function, you will need:

- x-positions of all of your particles, `x`.
- y-positions of all of your particles, `y`.
- A quantity which you wish to smooth for all particles, such as their mass, `m`.
- Smoothing lengths for all particles, `h`.
- The resolution you wish to make your square image at, `res`.

Optionally, you will also need: + the size of the simulation box in x and y, `box_x` and `box_y`.

The key here is that only particles in the domain [0, 1] in x, and [0, 1] in y will be visible in the image. You may have particles outside of this range; they will not crash the code, and may even contribute to the image if their smoothing lengths overlap with [0, 1]. You will need to re-scale your data such that it lives within this range. Then you may use the function as follows:

```
from swiftsimio.visualisation.projection import scatter

# Using the variable names from above
out = scatter(x=x, y=y, h=h, m=m, res=res)
```

`out` will be a 2D `numpy` grid of shape `[res, res]`. You will need to re-scale this back to your original dimensions to get it in the correct units, and do not forget that it now represents the smoothed quantity per surface area.

If the optional arguments `box_x` and `box_y` are provided, they should contain the simulation box size in the same re-scaled coordinates as `x` and `y`. The projection backend will then correctly apply periodic boundary wrapping. If `box_x` and `box_y` are not provided or set to 0, no periodic boundaries are applied.

4.2 Slices

The *swiftsimio.visualisation.slice* sub-module provides an interface to render SWIFT data onto a slice. This takes your 3D data and finds the 3D density at fixed z-position, slicing through the box.

This effectively solves the equation:

$\tilde{A}_i = \sum_j A_j W_{ij,3D}$

with $\tilde{A}_i$ the smoothed quantity in pixel i, and j all particles in the simulation, with W the 3D kernel. Here we use the Wendland-C2 kernel. Note that here we take the kernel at a fixed z-position.

The primary function here is *swiftsimio.visualisation.slice.slice_gas()*, which allows you to create a gas slice of any field. See the example below.

4.2.1 Example

```
from swiftsimio import load
from swiftsimio.visualisation.slice import slice_gas

data = load("cosmo_volume_example.hdf5")

# This creates a grid that has units msun / Mpc^3, and can be transformed like
# any other unyt quantity. The position of the slice along the z axis is
# provided in the z_slice argument.
mass_map = slice_gas(
    data,
```

(continues on next page)

(continued from previous page)

```
    z_slice=0.5 * data.metadata.boxsize[2],
    resolution=1024,
    project="masses",
    parallel=True,
    periodic=True,
)

# Let's say we wish to save it as g / cm^2,
from unyt import g, cm
mass_map.convert_to_units(g / cm**3)

from matplotlib.pyplot import imsave
from matplotlib.colors import LogNorm

# Normalize and save
imsave("gas_slice_map.png", LogNorm()(mass_map.value), cmap="viridis")
```

This basic demonstration creates a mass density map.

To create, for example, a projected temperature map, we need to remove the density dependence (i.e. `slice_gas()` returns a volumetric temperature in units of K / kpc^3 and we just want K) by dividing out by this:

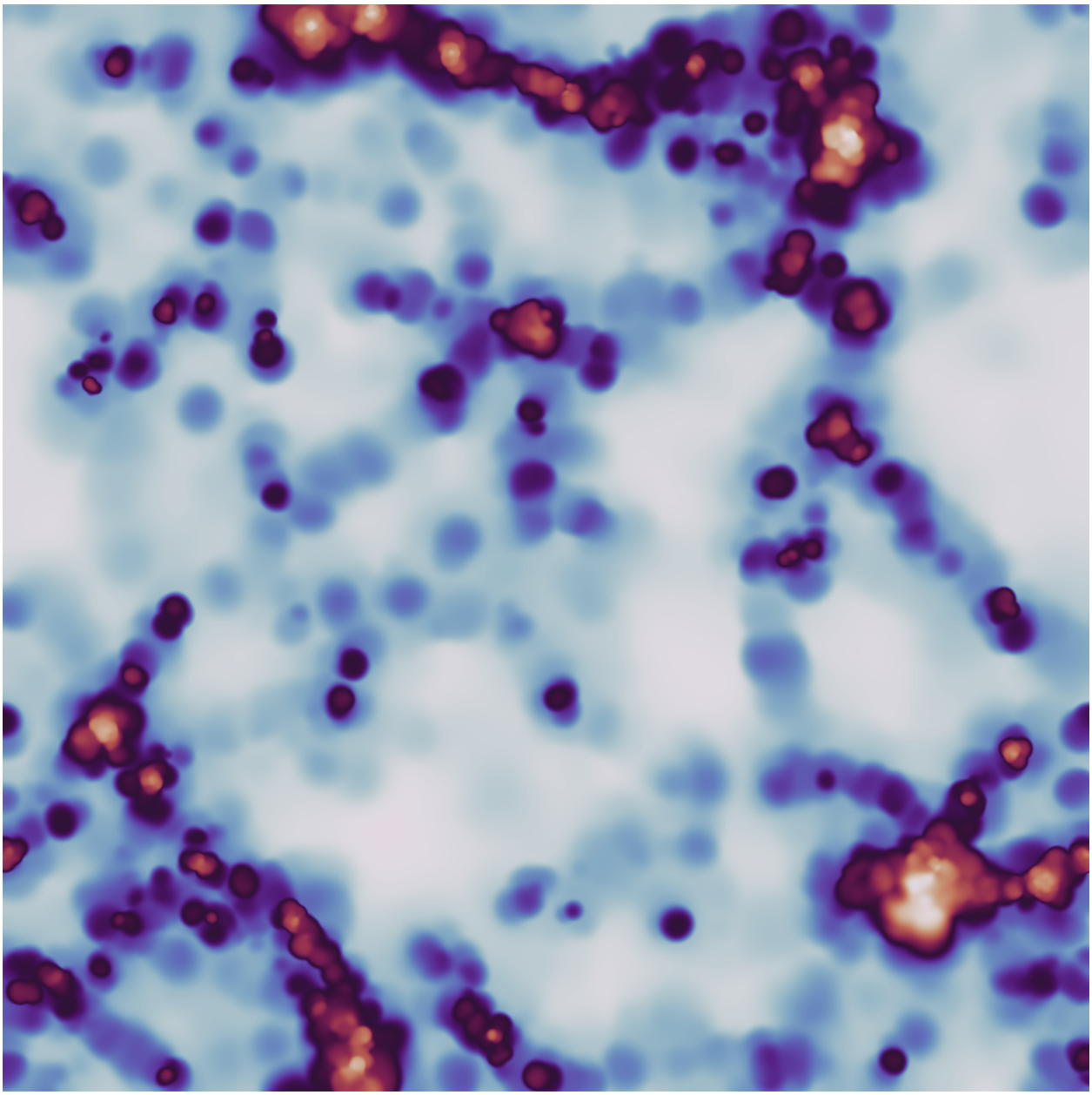
```
from swiftsimio import load
from swiftsimio.visualisation.slice import slice_gas

data = load("cosmo_volume_example.hdf5")

# First create a mass-weighted temperature dataset
data.gas.mass_weighted_temps = data.gas.masses * data.gas.temperatures

# Map in msun / mpc^3
mass_map = slice_gas(
    data,
    z_slice=0.5 * data.metadata.boxsize[2],
    resolution=1024,
    project="masses",
    parallel=True,
    periodic=True,
)

# Map in msun * K / mpc^3
mass_weighted_temp_map = slice_gas(
    data,
    z_slice=0.5 * data.metadata.boxsize[2],
    resolution=1024,
    project="mass_weighted_temps",
    parallel=True,
    periodic=True,
)

temp_map = mass_weighted_temp_map / mass_map

from unyt import K
```

(continues on next page)

(continued from previous page)

```
temp_map.convert_to_units(K)

from matplotlib.pyplot import imsave
from matplotlib.colors import LogNorm

# Normalize and save
imsave("temp_map.png", LogNorm()(temp_map.value), cmap="twilight")
```

The output from this example, when used with the example data provided in the loading data section should look something like:



4.2.2 Periodic boundaries

Cosmological simulations and many other simulations use periodic boundary conditions. This has implications for the particles at the edge of the simulation box: they can contribute to pixels on multiple sides of the image. If this effect is not taken into account, then the pixels close to the edge will have values that are too low because of missing contributions.

All visualisation functions by default assume a periodic box. Rather than simply summing each individual particle once, eight additional periodic copies of each particle are also accounted for. Most copies will contribute outside the valid pixel range, but the copies that do not ensure that pixels close to the edge receive all necessary contributions. Thanks to Numba optimisations, the overhead of these additional copies is relatively small.

There are some caveats with this approach. If you try to visualise a subset of the particles in the box (e.g. using a mask), then only periodic copies of particles in this subset will be used. If the subset does not include particles on the other side of the periodic boundary, then these will still be missing from the slice. The same is true if you visualise a region of the box. The periodic boundary wrapping is also not compatible with rotations (see below) and should therefore not be used together with a rotation.

4.2.3 Rotations

Rotations of the box prior to slicing are provided in a similar fashion to the *swiftsimio.visualisation.projection* sub-module, by using the *swiftsimio.visualisation.rotation* sub-module. To rotate the perspective prior to slicing a `rotation_center` argument in `slice_gas()` needs to be provided, specifying the point around which the rotation takes place. The angle of rotation is specified with a matrix, supplied by `rotation_matrix` in `slice_gas()`. The rotation matrix may be computed with `rotation_matrix_from_vector()`. This will result in the perspective being rotated to be along the provided vector. This approach to rotations applied to the above example is shown below.

```
from swiftsimio import load
from swiftsimio.visualisation.slice import slice_gas
from swiftsimio.visualisation.rotation import rotation_matrix_from_vector

data = load("cosmo_volume_example.hdf5")

# First create a mass-weighted temperature dataset
data.gas.mass_weighted_temps = data.gas.masses * data.gas.temperatures

# Specify the rotation parameters
center = 0.5 * data.metadata.boxsize
rotate_vec = [0.5,0.5,1]
matrix = rotation_matrix_from_vector(rotate_vec, axis='z')

# Map in msun / mpc^3
# If a rotation center is provided, z_slice is taken relative to this
# center, resulting in a slice perpendicular to the rotated z axis
mass_map = slice_gas(
    data,
    z_slice=0. * data.metadata.boxsize[2],
    resolution=1024,
    project="masses",
    rotation_matrix=matrix,
    rotation_center=center,
    parallel=True,
```

(continues on next page)

(continued from previous page)

```
    periodic=False, # disable periodic boundaries when using rotations
)

# Map in msun * K / mpc^3
mass_weighted_temp_map = slice_gas(
    data,
    z_slice=0. * data.metadata.boxsize[2],
    resolution=1024,
    project="mass_weighted_temps",
    rotation_matrix=matrix,
    rotation_center=center,
    parallel=True,
    periodic=False,
)

temp_map = mass_weighted_temp_map / mass_map

from unyt import K
temp_map.convert_to_units(K)

from matplotlib.pyplot import imsave
from matplotlib.colors import LogNorm

# Normalize and save
imsave("temp_map.png", LogNorm()(temp_map.value), cmap="twilight")
```

4.2.4 Lower-level API

The lower-level API for slices allows for any general positions, smoothing lengths, and smoothed quantities, to generate a pixel grid that represents the smoothed, sliced, version of the data.

This API is available through *swiftsimio.visualisation.slice.slice_scatter()* and *swiftsimio.visualisation.slice.slice_scatter_parallel()* for the parallel version. The parallel version uses significantly more memory as it allocates a thread-local image array for each thread, summing them in the end. Here we will only describe the `scatter` variant, but they behave in the exact same way.

To use this function, you will need:

- x-positions of all of your particles, `x`.
- y-positions of all of your particles, `y`.
- z-positions of all of your particles, `z`.
- Where in the range you wish to slice, `z_slice`.
- A quantity which you wish to smooth for all particles, such as their mass, `m`.
- Smoothing lengths for all particles, `h`.
- The resolution you wish to make your square image at, `res`.

Optionally, you will also need: + the size of the simulation box in x, y and z, `box_x`, `box_y` and `box_z`.

The key here is that only particles in the domain [0, 1] in x and y will be visible in the image. You may have particles outside of this range; they will not crash the code, and may even contribute to the image if their smoothing lengths overlap with [0, 1]. You will need to re-scale your data such that it lives within this range. Smoothing lengths and z

coordinates need to be re-scaled in the same way (using the same scaling factor), but z coordinates do not need to lie in the domain [0, 1]. Then you may use the function as follows:

```
from swiftsimio.visualisation.slice import slice_scatter

# Using the variable names from above
out = slice_scatter(x=x, y=y, z=z, h=h, m=m, z_slice=z_slice, res=res)
```

`out` will be a 2D `numpy` grid of shape `[res, res]`. You will need to re-scale this back to your original dimensions to get it in the correct units, and do not forget that it now represents the smoothed quantity per volume.

If the optional arguments `box_x`, `box_y` and `box_z` are provided, they should contain the simulation box size in the same re-scaled coordinates as `x`, `y` and `z`. The slicing function will then correctly apply periodic boundary wrapping. If `box_x`, `box_y` and `box_z` are not provided or set to 0, no periodic boundaries are applied.

4.3 Volume Rendering

The *swiftsimio.visualisation.volume_render* sub-module provides an interface to render SWIFT data onto a fixed grid. This takes your 3D data and finds the 3D density at fixed positions, allowing it to be used in codes that require fixed grids such as radiative transfer programs.

This effectively solves the equation:

$\tilde{A}_i = \sum_j A_j W_{ij,3D}$

with $\tilde{A}_i$ the smoothed quantity in pixel i, and j all particles in the simulation, with W the 3D kernel. Here we use the Wendland-C2 kernel.

The primary function here is *swiftsimio.visualisation.volume_render.render_gas()*, which allows you to create a gas density grid of any field, see the example below.

4.3.1 Example

```
from swiftsimio import load
from swiftsimio.visualisation.volume_render import render_gas

data = load("cosmo_volume_example.hdf5")

# This creates a grid that has units msun / Mpc^3, and can be transformed like
# any other unyt quantity.
mass_grid = render_gas(
    data,
    resolution=256,
    project="masses",
    parallel=True,
    periodic=True,
)
```

This basic demonstration creates a mass density cube.

To create, for example, a projected temperature cube, we need to remove the density dependence (i.e. `render_gas()` returns a volumetric temperature in units of K / kpc^3 and we just want K) by dividing out by this:

```
from swiftsimio import load
from swiftsimio.visualisation.volume_render import render_gas

data = load("cosmo_volume_example.hdf5")

# First create a mass-weighted temperature dataset
data.gas.mass_weighted_temps = data.gas.masses * data.gas.temperatures

# Map in msun / mpc^3
mass_cube = render_gas(
    data,
    resolution=256,
    project="masses",
    parallel=True,
    periodic=True,
)

# Map in msun * K / mpc^3
mass_weighted_temp_cube = render_gas(
    data,
    resolution=256,
    project="mass_weighted_temps",
    parallel=True,
    periodic=True,
)

# A 256 x 256 x 256 cube with dimensions of temperature
temp_cube = mass_weighted_temp_cube / mass_cube
```

4.3.2 Periodic boundaries

Cosmological simulations and many other simulations use periodic boundary conditions. This has implications for the particles at the edge of the simulation box: they can contribute to voxels on multiple sides of the image. If this effect is not taken into account, then the voxels close to the edge will have values that are too low because of missing contributions.

All visualisation functions by default assume a periodic box. Rather than simply summing each individual particle once, eight additional periodic copies of each particle are also taken into account. Most copies will contribute outside the valid voxel range, but the copies that do not ensure that voxels close to the edge receive all necessary contributions. Thanks to Numba optimisations, the overhead of these additional copies is relatively small.

There are some caveats with this approach. If you try to visualise a subset of the particles in the box (e.g. using a mask), then only periodic copies of particles in this subset will be used. If the subset does not include particles on the other side of the periodic boundary, then these will still be missing from the voxel cube. The same is true if you visualise a region of the box. The periodic boundary wrapping is also not compatible with rotations (see below) and should therefore not be used together with a rotation.

4.3.3 Rotations

Rotations of the box prior to volume rendering are provided in a similar fashion to the *swiftsimio.visualisation.projection* sub-module, by using the *swiftsimio.visualisation.rotation* sub-module. To rotate the perspective prior to slicing a `rotation_center` argument in `render_gas()` needs to be provided, specifying the point around which the rotation takes place. The angle of rotation is specified with a matrix, supplied by `rotation_matrix` in `render_gas()`. The rotation matrix may be computed with `rotation_matrix_from_vector()`. This will result in the perspective being rotated to be along the provided vector. This approach to rotations applied to the above example is shown below.

```
from swiftsimio import load
from swiftsimio.visualisation.volume_render import render_gas
from swiftsimio.visualisation.rotation import rotation_matrix_from_vector

data = load("cosmo_volume_example.hdf5")

# First create a mass-weighted temperature dataset
data.gas.mass_weighted_temps = data.gas.masses * data.gas.temperatures

# Specify the rotation parameters
center = 0.5 * data.metadata.boxsize
rotate_vec = [0.5,0.5,1]
matrix = rotation_matrix_from_vector(rotate_vec, axis='z')

# Map in msun / mpc^3
mass_cube = render_gas(
    data,
    resolution=256,
    project="masses",
    rotation_matrix=matrix,
    rotation_center=center,
    parallel=True,
    periodic=False, # disable periodic boundaries for rotations
)

# Map in msun * K / mpc^3
mass_weighted_temp_cube = render_gas(
    data,
    resolution=256,
    project="mass_weighted_temps",
    rotation_matrix=matrix,
    rotation_center=center,
    parallel=True,
    periodic=False,
)

# A 256 x 256 x 256 cube with dimensions of temperature
temp_cube = mass_weighted_temp_cube / mass_cube
```

4.3.4 Lower-level API

The lower-level API for volume rendering allows for any general positions, smoothing lengths, and smoothed quantities, to generate a pixel grid that represents the smoothed, volume rendered, version of the data.

This API is available through *swiftsimio.visualisation.volume_render.scatter()* and *swiftsimio.visualisation.volume_render.scatter_parallel()* for the parallel version. The parallel version uses significantly more memory as it allocates a thread-local image array for each thread, summing them in the end. Here we will only describe the `scatter` variant, but they behave in the exact same way.

To use this function, you will need:

- x-positions of all of your particles, `x`.
- y-positions of all of your particles, `y`.
- z-positions of all of your particles, `z`.
- A quantity which you wish to smooth for all particles, such as their mass, `m`.
- Smoothing lengths for all particles, `h`.
- The resolution you wish to make your cube at, `res`.

Optionally, you will also need: + the size of the simulation box in x, y and z, `box_x`, `box_y` and `box_z`.

The key here is that only particles in the domain [0, 1] in x, [0, 1] in y, and [0, 1] in z. will be visible in the cube. You may have particles outside of this range; they will not crash the code, and may even contribute to the image if their smoothing lengths overlap with [0, 1]. You will need to re-scale your data such that it lives within this range. Then you may use the function as follows:

```
from swiftsimio.visualisation.volume_render import scatter

# Using the variable names from above
out = scatter(x=x, y=y, z=z, h=h, m=m, res=res)
```

`out` will be a 3D `numpy` grid of shape `[res, res, res]`. You will need to re-scale this back to your original dimensions to get it in the correct units, and do not forget that it now represents the smoothed quantity per volume.

If the optional arguments `box_x`, `box_y` and `box_z` are provided, they should contain the simulation box size in the same re-scaled coordinates as `x`, `y` and `z`. The rendering function will then correctly apply periodic boundary wrapping. If `box_x`, `box_y` and `box_z` are not provided or set to 0, no periodic boundaries are applied

4.4 Tools

swiftsimio includes a few tools to help you make your visualisations 'prettier'. Below we describe these tools and their use.

4.4.1 2D Color Maps

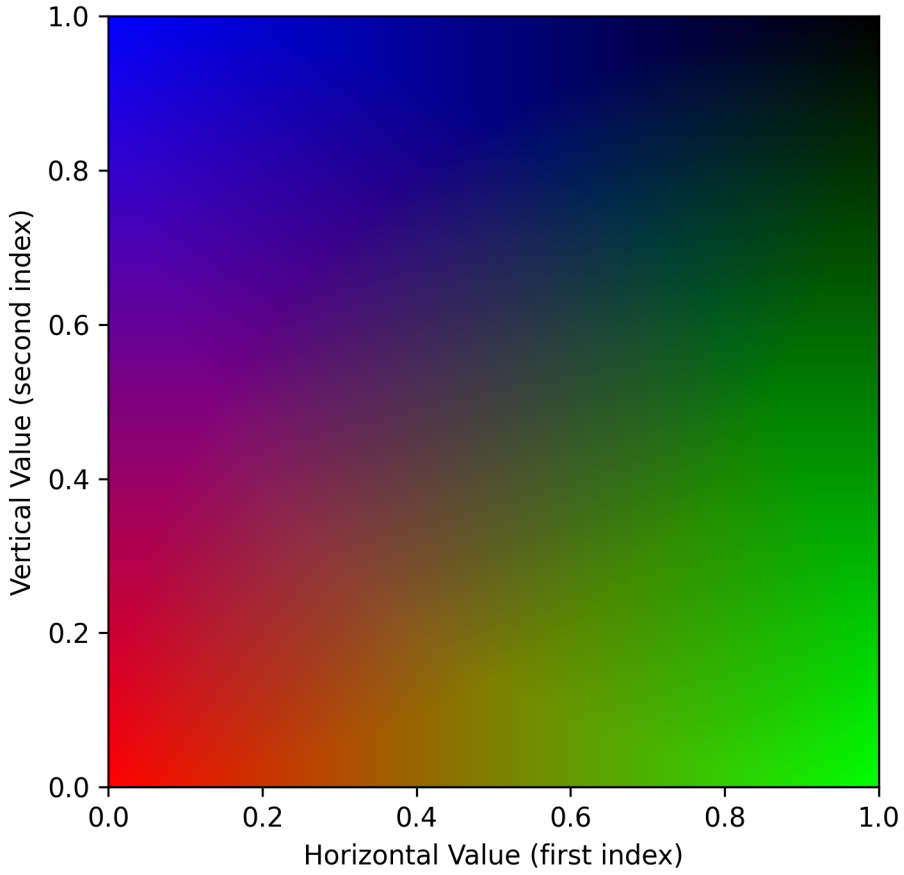
The *swiftsimio.visualisation.tools.cmaps* module includes three objects that can be used to deploy two dimensional colour maps. The first, *swiftsimio.visualisation.tools.cmaps.LinearSegmentedCmap2D*, and second *swiftsimio.visualisation.tools.cmaps.LinearSegmentedCmap2DHSV*, allow you to generate new color maps from sets of colors and coordinates.

```
bower = LinearSegmentedCmap2D(
        colors=[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0]],
        coordinates=[[0.0, 0.0], [1.0, 0.0], [0.0, 1.0], [1.0, 1.0]],
        name="bower"
)
```

This generates a color map that is a quasi-linear interpolation between all of the points. The map can be displayed using the `plot` method,

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

bower.plot(ax)
```

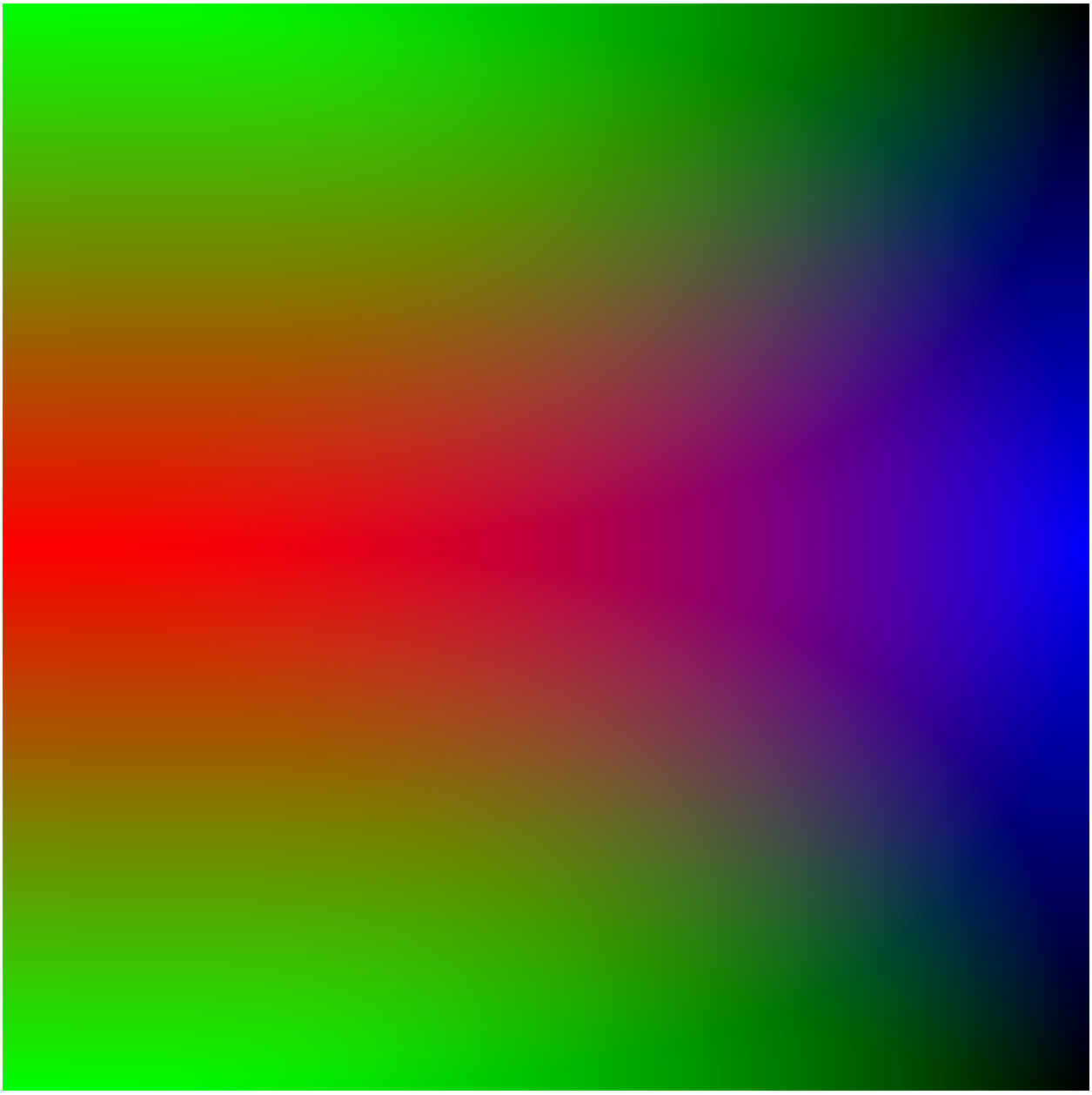
Which generates:



Finally, the color map can be applied to data by calling it:

```
def vertical_func(x):
        return abs(1.0 - 2.0 * x)

def horizontal_func(y):
        return y ** 2

raster_at = np.linspace(0, 1, 1024)

x, y = np.meshgrid(horizontal_func(raster_at), vertical_func(raster_at))

imaged = bower(x, y)

plt.imsave("test_2d_cmap_output.png", imaged)
```

Where here `imaged` is an RGBA array. This outputs:

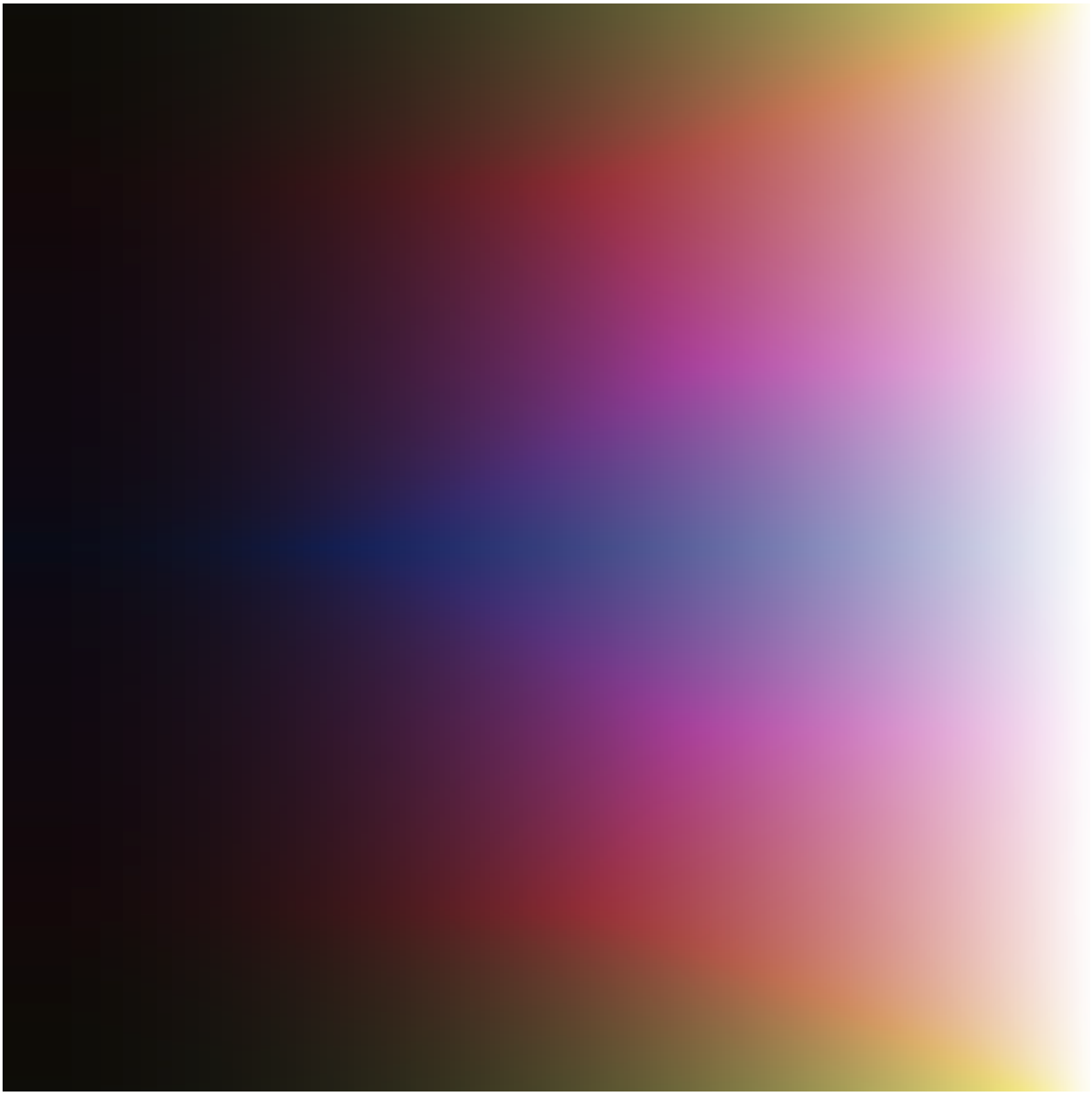


The final type of 2D color map is loaded from an image, such as the one displayed below which is similar to the famous color map used for the Millenium simulation.

This can be loaded using the *swiftsimio.visualisation.tools.cmaps.ImageCmap2D* class, as follows:

```
mill = ImageCmap2D(filename="millenium_cmap.png")
```

and can be used similarly to the other color maps. For the example above, this outputs the following:



This is the recommended way to use two dimensional color maps, as their generation can be quite complex and best left to image-generation programs such as GIMP or Photoshop.

CHAPTER

FIVE

VELOCIRAPTOR INTEGRATION

swiftsimio can be used with the `velociraptor` library to extract the particles contained within a given halo and its surrounding region.

The `velociraptor` library has documentation also available on ReadTheDocs here. It can be installed from PyPI using `pip install velociraptor`.

The overarching workflow for this integration is as follows:

- Load the halo catalogue and groups file using the `velociraptor` module.
- Get two objects, corresponding to the bound and unbound particles, for a halo.
- Use the *to_swiftsimio_dataset* to load the region around the halo with our ahead-of-time masking technique.
- Use the region around the halo directly, or use the mask provided for each particle type to only consider bound particles.

This workflow is explored below. You can use the example data available below if you do not have any SWIFT and VELOCIraptor data available.

`http://virgodb.cosma.dur.ac.uk/swift-webstorage/IOExamples/small_cosmo_volume.zip`

5.1 Example

First, we must load the VELOCIraptor catalogue as follows:

```
from velociraptor import load as load_catalogue
from velociraptor.particles import load_groups

catalogue_name = "velociraptor"
snapshot_name = "snapshot"

catalogue = load_catalogue(f"{catalogue_name}.properties")
groups = load_groups(f"{catalogue_name}.catalog_groups", catalogue=catalogue)
```

Then, to extract the largest halo in the volume

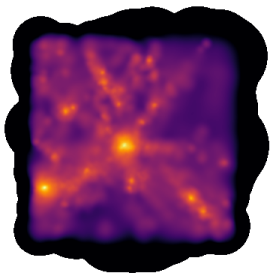
```
particles, unbound_particles = groups.extract_halo(halo_id=0)
```

To load the particles to a *swiftsimio* dataset,

```
from velociraptor.swift.swift import to_swiftsimio_dataset

data, mask = to_swiftsimio_dataset(
    particles,
    f"{snapshot_name}.hdf5",
    generate_extra_mask=True
)
```

with the `generate_extra_mask` providing the second return value which is a mask to extract only the bound particles in the system.

Making an image of the full box shows that only a small subsection of the volume has been loaded (those within twice the maximal usable radius within VELOCIraptor)



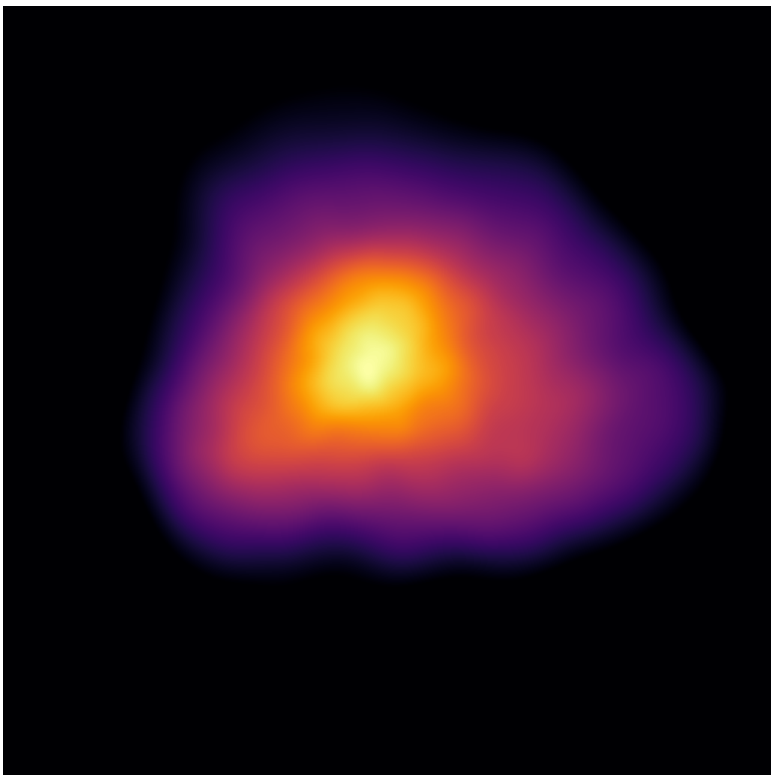
The code for making this image is as follows:

```
from swiftsimio.visualisation import project_gas_pixel_grid
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

grid = project_gas_pixel_grid(data=data, resolution=1024)

fig, ax = plt.subplots(figsize=(4, 4), dpi=1024 // 4)
fig.subplots_adjust(0, 0, 1, 1)
ax.axis("off")
ax.imshow(grid.T, origin="lower", cmap="inferno", norm=LogNorm(vmin=1e4, clip=True))
fig.savefig("load_halo_fullbox.png")
```

To make an image of just the central halo, we can access properties on the `particles` instance to get the position of the halo.

```
region = [
    particles.x_mbp - particles.r_200crit, particles.x_mbp + particles.r_200crit,
    particles.y_mbp - particles.r_200crit, particles.y_mbp + particles.r_200crit,
]

grid = project_gas_pixel_grid(data=data, resolution=1024, region=region)

fig, ax = plt.subplots(figsize=(4, 4), dpi=1024 // 4)
fig.subplots_adjust(0, 0, 1, 1)
ax.axis("off")
ax.imshow(grid.T, origin="lower", cmap="inferno", norm=LogNorm(vmin=1e4, clip=True))
fig.savefig("load_halo_selection.png")
```

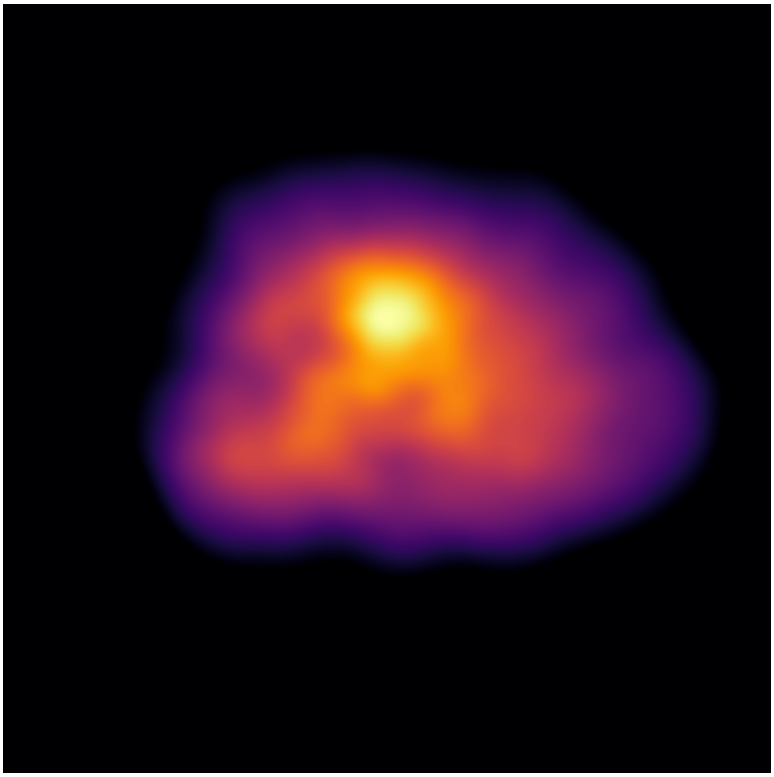
This produces the following image:



Then, finally, we can visualise only the bound particles, through the use of the `mask` object that was returned when we initially extracted the `swiftsimio` dataset:

```
grid = project_gas_pixel_grid(data=data, resolution=1024, region=region, mask=mask.gas)

fig, ax = plt.subplots(figsize=(4, 4), dpi=1024 // 4)
fig.subplots_adjust(0, 0, 1, 1)
ax.axis("off")
ax.imshow(grid.T, origin="lower", cmap="inferno", norm=LogNorm(vmin=1e4, clip=True))
fig.savefig("load_halo_bound_selection.png")
```

Producing the following image:



Hopefully, when you use this feature, you have more exciting data to use than the as-small-as-possible example that we show here!

CHAPTER

SIX

CREATING INITIAL CONDITIONS

Writing datasets that are valid for consumption for cosmological codes can be difficult, especially when considering how to best use units. SWIFT uses a different set of internal units (specified in your parameter file) that does not necessarily need to be the same set of units that initial conditions are specified in. Nevertheless, it is important to ensure that units in the initial conditions are all *consistent* with each other. To facilitate this, we use `unyt` arrays. The below example generates randomly placed gas particles with uniform densities.

The functionality to create initial conditions is available through the *swiftsimio.writer* sub-module, and the top-level `swiftsimio.Writer` object.

Note that the properties that *swiftsimio* requires in the initial conditions are the only ones that are actually read by SWIFT; other fields will be left un-read and as such should not be included in initial conditions files.

A current known issue is that due to inconsistencies with the initial conditions and simulation snapshots, *swiftsimio* is not actually able to read the inititial conditions that it produces. We are aiming to fix this in an upcoming release.

6.1 Example

```
from swiftsimio import Writer
from swiftsimio.units import cosmo_units

import unyt
import numpy as np

# Box is 100 Mpc
boxsize = 100 * unyt.Mpc

# Generate object. cosmo_units corresponds to default Gadget-oid units
# of 10^10 Msun, Mpc, and km/s
x = Writer(cosmo_units, boxsize)

# 32^3 particles.
n_p = 32**3

# Randomly spaced coordinates from 0, 100 Mpc in each direction
x.gas.coordinates = np.random.rand(n_p, 3) * (100 * unyt.Mpc)

# Random velocities from 0 to 1 km/s
x.gas.velocities = np.random.rand(n_p, 3) * (unyt.km / unyt.s)

# Generate uniform masses as 10^6 solar masses for each particle
```

(continues on next page)

(continued from previous page)

```
x.gas.masses = np.ones(n_p, dtype=float) * (1e6 * unyt.msun)

# Generate internal energy corresponding to 10^4 K
x.gas.internal_energy = (
    np.ones(n_p, dtype=float) * (1e4 * unyt.kb * unyt.K) / (1e6 * unyt.msun)
)

# Generate initial guess for smoothing lengths based on MIPS
x.gas.generate_smoothing_lengths(boxsize=boxsize, dimension=3)

# If IDs are not present, this automatically generates
x.write("test.hdf5")
```

Then, running `h5glance` on the resulting `test.hdf5` produces:

```
test.hdf5
├Header
  └5 attributes:
    ├BoxSize: 100.0
    ├Dimension: array [int64: 1]
    ├Flag_Entropy_ICs: 0
    ├NumPart_Total: array [int64: 6]
    └NumPart_Total_HighWord: array [int64: 6]
├PartType0
  ├Coordinates  [float64: 32768 × 3]
  ├InternalEnergy       [float64: 32768]
  ├Masses       [float64: 32768]
  ├ParticleIDs  [float64: 32768]
  ├SmoothingLength      [float64: 32768]
  └Velocities   [float64: 32768 × 3]
└Units
└5 attributes:
    ├Unit current in cgs (U_I): array [float64: 1]
    ├Unit length in cgs (U_L): array [float64: 1]
    ├Unit mass in cgs (U_M): array [float64: 1]
    ├Unit temperature in cgs (U_T): array [float64: 1]
    └Unit time in cgs (U_t): array [float64: 1]
```

Note you do need to be careful that your choice of unit system does *not* allow values over 2^31, i.e. you need to ensure that your provided values (with units) when *written* to the file are safe to be interpreted as (single-precision) floats. The only exception to this is coordinates which are stored in double precision.

CHAPTER

SEVEN

STATISTICS FILES

swiftsimio includes routines to load log files, such as the `SFR.txt` and `energy.txt`. This is available through the *swiftsimio.statistics.SWIFTStatisticsFile* object, or through the main `load_statistics` function.

7.1 Example

```
from swiftsimio import load_statistics

data = load_statistics("energy.txt")

print(data)

print(x.total_mass.name)
```

Will output:

```
Statistics file: energy.txt, containing fields: #, step, time, a, z, total_mass,
gas_mass, dm_mass, sink_mass, star_mass, bh_mass, gas_z_mass, star_z_mass,
bh_z_mass, kin_energy, int_energy, pot_energy, rad_energy, gas_entropy, com_x,
com_y, com_z, mom_x, mom_y, mom_z, ang_mom_x, ang_mom_y, ang_mom_z

'Total mass in the simulation'
```

CHAPTER

EIGHT

COMMAND-LINE UTILITIES

swiftsimio comes with some useful command-line utilities. Basic documentation for these is provided below, but you can always find up-to-date documentation by invoking these with `-h` or `--help`.

8.1 swiftsnap

The `swiftsnap` utility, introduced in *swiftsimio* version 3.1.2, allows you to preview the metadata inside a SWIFT snapshot file. Simply invoke it with the path to a snapshot, and it will show you a selection of useful metadata. See below for an example.

```
swiftsnap output_0103.hdf5
```

Produces the following output:

```
Untitled SWIFT simulation
Written at: 2020-06-01 08:44:51
Active policies: cosmological integration, hydro, keep, self gravity, steal
Output type: Snapshot, Output selection: Snapshot
LLVM/Clang (11.0.0)
Non-MPI version of SWIFT
SWIFT (io_selection_changes)
v0.8.5-725-g10d7d5b3-dirty
2020-05-29 18:00:58 +0100
Simulation state: z=0.8889, a=0.5294, t=6.421 Gyr
H_0=70.3 km/(Mpc*s), _crit=1.433e-05 cm**(-3)
_b=0.0455, _k=0, _lambda=0.724, _m=0.276, _r=0
=-1, _0=-1, _a=0
Gravity scheme: With per-particle softening
Hydrodynamics scheme: Gadget-2 version of SPH (Springel 2005)
Chemistry model: None
Cooling model: None
Entropy floor: None
Feedback model: None
Tracers: None
```

CHAPTER

NINE

API DOCUMENTATION

9.1 swiftsimio package

swiftsimio.**validate_file**(*filename*)

Checks that the provided file is a SWIFT dataset.

Parameters
filename (*str*) – name of file we want to check is a dataset

Returns
if *filename* is a SWIFT dataset return True, otherwise raise exception

Return type
bool

Raises
KeyError – Crash if the file is not a SWIFT data file

swiftsimio.**mask**(*filename*, *spatial_only=True*) → *SWIFTMask*

Sets up a masking object for you to use with the correct units and metadata available.

Parameters

- **filename** (*str*) – SWIFT data file to read from
- **spatial_only** (*bool, optional*) – Flag for only spatial masking, this is much faster but will not allow you to use masking on other variables (e.g. density). Defaults to True.

Returns
empty mask object set up with the correct units and metadata

Return type
SWIFTMask

Notes

If you are only planning on using this as a spatial mask, ensure that spatial_only remains true. If you require the use of the constrain_mask function, then you will need to use the (considerably more expensive, ~bytes per particle instead of ~bytes per cell spatial_only=False version).

swiftsimio.**load**(*filename*, *mask=None*) → *SWIFTDataset*

Loads the SWIFT dataset at filename.

Parameters

- **filename** (*str*) – SWIFT snapshot file to read

- **mask** (SWIFTMask, *optional*) – mask to apply when reading dataset

swiftsimio.**load_statistics**(*filename*) → *SWIFTStatisticsFile*

Loads a SWIFT statistics file (SFR.txt, energy.txt).

Parameters

filename (*str*) – SWIFT statistics file path

9.1.1 Subpackages

swiftsimio.initial_conditions package

Initial conditions generation.

Submodules

swiftsimio.initial_conditions.generate_particles module

Particle generation code.

TBD

swiftsimio.visualisation package

Visualisation sub-module for swiftismio.

Subpackages

swiftsimio.visualisation.projection_backends package

Backends for density projection.

These go in order (within the dictionary) from fastest to most accurate, with the "_reference" style being a developer-only indended feature.

Submodules

swiftsimio.visualisation.projection_backends.fast module

Fast backend.

This uses float32 precision and no special cases.

swiftsimio.visualisation.projection_backends.fast.**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Creates a weighted scatter plot

Computes contributions to from particles with positions (x,`y`) with smoothing lengths h weighted by quantities m. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.fast.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.gpu module

swiftsimio.visualisation.projection_backends.gpu.**kernel**(*r: float32*, *H: float32*)

Single precision kernel implementation for swiftsimio.

This is the Wendland-C2 kernel as shown in Denhen & Aly (2012)[1].

Parameters

- **r** (*float32*) – radius used in kernel computation
- **H** (*float32*) – kernel width (i.e. radius of compact support for the kernel)

Returns
Contribution to the density by the particle

Return type
float32

References

Notes

This is the cuda-compiled version of the kernel, designed for use within the gpu backend. It has no double precision cousin.

swiftsimio.visualisation.projection_backends.gpu.**scatter_gpu**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *box_x: float64*, *box_y: float64*, *img: float32*)

Creates a weighted scatter plot

Computes contributions to from particles with positions (x,`y`) with smoothing lengths h weighted by quantities m. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles

[1] Dehnen W., Aly H., 2012, MNRAS, 425, 1068

- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.
- **img** (*np.array[float32]*) – The output image.

Notes

Explicitly defining the types in this function allows for a performance improvement. This is the cuda version, and as such can only be ran on systems with a supported GPU. Do not call this where cuda is not available (checks can be performed using `swiftsimio.optional_packages.CUDA_AVAILABLE`)

`swiftsimio.visualisation.projection_backends.gpu.`**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows a performance improvement.

swiftsimio.visualisation.projection_backends.gpu.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows a performance improvement.

swiftsimio.visualisation.projection_backends.histogram module

Reference evaluation - returns a 2d histogram (i.e. no smoothing).

Uses double precision.

swiftsimio.visualisation.projection_backends.histogram.**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Creates a weighted scatter plot

Computes contributions to from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.histogram.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.kernels module

Projection kernels.

swiftsimio.visualisation.projection_backends.kernels.**kernel_single_precision**(*r: float32*, *H: float32*)

Single precision kernel implementation for swiftsimio.

This is the Wendland-C2 kernel as shown in Denhen & Aly (2012)[1].

Parameters

- **r** (*float32*) – radius used in kernel computation
- **H** (*float32*) – kernel width (i.e. radius of compact support for the kernel)

Returns
Contribution to the density by the particle

Return type
float32

See also:

kernel_double_precision

References

swiftsimio.visualisation.projection_backends.kernels.**kernel_double_precision**(*r: float64*, *H: float64*)

Single precision kernel implementation for swiftsimio.

This is the Wendland-C2 kernel as shown in Denhen & Aly (2012)[2].

Parameters

- **r** (*float32*) – radius used in kernel computation
- **H** (*float32*) – kernel width (i.e. radius of compact support for the kernel)

[1] Dehnen W., Aly H., 2012, MNRAS, 425, 1068
[2] Dehnen W., Aly H., 2012, MNRAS, 425, 1068

Returns
Contribution to the density by the particle

Return type
float32

See also:

kernel_single_precision

References

swiftsimio.visualisation.projection_backends.reference module

Reference evaluation - only returns a 'real' result if no particles lie below the resolution limit.

Uses double precision.

swiftsimio.visualisation.projection_backends.reference.**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Creates a weighted scatter plot

Computes contributions to from particles with positions (x,`y`) with smoothing lengths h weighted by quantities m. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.reference.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.renormalised module

Renormalised projection visualisation.

This version of the function is the same as *fast* but provides an explicit renormalisation of each kernel such that the mass is conserved up to floating point precision.

swiftsimio.visualisation.projection_backends.renormalised.**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Creates a weighted scatter plot

Computes contributions to from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.renormalised.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.subsampled module

Sub-sampled smoothing kernel with each kernel evaluated at least 32^2 times. This uses a dithered pre-calculated kernel for cell overlaps at small scales, and at large scales uses subsampling.

Uses double precision.

swiftsimio.visualisation.projection_backends.subsampled.**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Creates a weighted scatter plot

Computes contributions to from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles

- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.subsampled.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.projection_backends.subsampled_extreme module

Sub-sampled smoothing kernel with each kernel evaluated at least 64^2 times. This uses a dithered pre-calculated kernel for cell overlaps at small scales, and at large scales uses subsampling.

Uses double precision.

swiftsimio.visualisation.projection_backends.subsampled_extreme.**scatter**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Creates a weighted scatter plot

Computes contributions to from particles with positions (x,`y`) with smoothing lengths h weighted by quantities m. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

Uses 4x the number of sampling points as in scatter in subsampled.py

swiftsimio.visualisation.projection_backends.subsampled_extreme.**scatter_parallel**(*x: float64*, *y: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Creates a weighted scatter plot. Computes contributions from particles with positions (*x*,`y`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of pixels along one axis, i.e. this returns a square of res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x and y. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x and y. Used for periodic wrapping.

Returns
pixel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter
Creates 2D scatter plot from SWIFT data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

Uses 4x the number of sampling points as in scatter_parallel in subsampled.py

swiftsimio.visualisation.tools package

Submodules

swiftsimio.visualisation.tools.cmaps module

Two-dimensional colour map support, along with example colour maps.

swiftsimio.visualisation.tools.cmaps.**ensure_rgba**(*input_color: Iterable[float]*) → array

Ensures a colour is RGBA compliant.

Default alpha if missing: 1.0.

Parameters
input_color (*iterable*) – An iterable of maximum length 4, with RGBA values encoded as floating point 0.0 -> 1.0.

Returns
array_color – An array of length 4 as an RGBA color.

Return type
np.array

swiftsimio.visualisation.tools.cmaps.**apply_color_map**(*first_values*, *second_values*, *map_grid*)

Applies a 2D colour map by providing a 2D linear interpolation to the known fixed grid points. Not to be called on its own, as the map itself is provided by the `LinearSegmentedCmap2D`, but this is provided separately so it can be `numba`-accelerated.

Parameters

- **first_values** (*iterable[float]*) – Array or list to loop over, containing floats ranging from 0.0 to 1.0. Provides the normalisation for the horizontal component. Must be one-dimensional.
- **second_values** (*iterable[float]*) – Array or list to loop over, containing floats ranging from 0.0 to 1.0. Provides the normalisation for the vertical component. Must be one-dimensional.
- **map_grid** (*np.ndarray*) – 2D numpy array proided by `LinearSegmentedCmap2D`.

Returns
An N by 4 array (where N is the length of `first_value` and `second_value`) of RGBA components.

Return type
np.ndarray

class swiftsimio.visualisation.tools.cmaps.**Cmap2D**(*name: str | None = None*, *description: str | None = None*)

Bases: `object`

A generic two dimensional implementation of a colour map.

Developer use only.

colors: List[List[float]] = None

coordinates: List[List[float]] = None

generate_color_map_grid()

Generates the colour map grid and stores it in _color_map_grid. Imeplementation dependent.

property color_map_grid

Generates, or gets, the color map grid.

plot(*ax*, *include_points: bool = False*)

Plot the color map on axes.

Parameters

- **ax** (*matplotlib.Axis*) – Axis to be plotted on.
- **include_points** (*bool, optional*) – If true, plot the individual colours as points that make up the color map. Default: False.

class swiftsimio.visualisation.tools.cmaps.**LinearSegmentedCmap2D**(*colors: List[List[float]]*, *coordinates: List[List[float]]*, *name: str | None = None*, *description: str | None = None*)

Bases: *Cmap2D*

A two dimensional implementation of the linear segmented colour map.

generate_color_map_grid()

Generates the color map grid.

class swiftsimio.visualisation.tools.cmaps.**LinearSegmentedCmap2DHSV**(*colors: List[List[float]]*, *coordinates: List[List[float]]*, *name: str | None = None*, *description: str | None = None*)

Bases: *Cmap2D*

A two dimensional implementation of the linear segmented colour map, using the HSV space to combine the colours.

Parameters

- **colors** (*List[List[float]]*) – Individual colors (at coordinates below) that make up the color map.
- **coordinates** (*List[List[float]]*) – 2D coordinates in the plane to place the above colors at.
- **name** (*str, optional*) – Name of this color map (metadata)
- **description** (*str, optional*) – Optional metadata description of this colour map.

See also:

LinearSegmentedCmap2D, a cousin of this class that combines colours using the RGB space rather than HSV used here.

generate_color_map_grid()

Generates the color map grid.

class swiftsimio.visualisation.tools.cmaps.**ImageCmap2D**(*filename: str*, *name: str | None = None*, *description: str | None = None*)

Bases: *Cmap2D*

Creates a 2D color map from an image loaded from disk.

generate_color_map_grid()

Loads the image from file and stores it as the internal array.

Submodules

swiftsimio.visualisation.projection module

Calls functions from *projection_backends*.

swiftsimio.visualisation.projection.**project_pixel_grid**(*data: __SWIFTParticleDataset*, *boxsize: unyt_array*, *resolution: int*, *project: str | None = 'masses'*, *region: None | unyt_array = None*, *mask: None | array = None*, *rotation_matrix: None | array = None*, *rotation_center: None | unyt_array = None*, *parallel: bool = False*, *backend: str = 'fast'*, *periodic: bool = True*)

Creates a 2D projection of a SWIFT dataset, projected by the "project" variable (e.g. if project is Temperature, we return: bar{T} = sum_j T_j W_{ij}).

Default projection variable is mass. If it is None, then we don't weight with anything, providing a number density image.

Parameters

- **data** (*__SWIFTParticleDataset*) – The SWIFT dataset that you wish to visualise (get this from `load`)
- **boxsize** (*unyt_array*) – The box-size of the simulation.
- **resolution** (*int*) – The resolution of the image. All images returned are square, `res` by `res`, pixel grids.
- **project** (*str, optional*) – Variable to project to get the weighted density of. By default, this is mass. If you would like to mass-weight any other variable, you can always create it as `data.gas.my_variable = data.gas.other_variable * data.gas.masses`.
- **region** (*unyt_array, optional*) – Region, determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges) if it is not None. It should have a length of four or six, and take the form: `[x_min, x_max, y_min, y_max, {z_min, z_max}]`
- **mask** (*np.array, optional*) – Allows only a sub-set of the particles in data to be visualised. Useful in cases where you have read data out of a `velociraptor` catalogue, or if you only want to visualise e.g. star forming particles. This boolean mask is applied just before visualisation.
- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.

- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a projection along the z axis.
- **parallel** (*bool, optional*) – Defaults to `False`, whether or not to create the image in parallel. The parallel version of this function uses significantly more memory.
- **backend** (*str, optional*) – Backend to use. See documentation for details. Defaults to 'fast'.
- **periodic** (*bool, optional*) – Account for periodic boundary conditions for the simulation box? Defaults to `True`.

Returns
image – Projected image with units of project / length^2, of size `res` x `res`.

Return type
unyt_array

Notes

- Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- The returned array has x as the first component and y as the second component, which is the opposite to what `imshow` requires. You should transpose the array if you want it to be visualised the 'right way up'.

swiftsimio.visualisation.projection.**project_gas_pixel_grid**(*data: SWIFTDataset*, *resolution: int*, *project: str | None = 'masses'*, *region: None | unyt_array = None*, *mask: None | array = None*, *rotation_matrix: None | array = None*, *rotation_center: None | unyt_array = None*, *parallel: bool = False*, *backend: str = 'fast'*, *periodic: bool = True*)

Creates a 2D projection of a SWIFT dataset, projected by the "project" variable (e.g. if project is Temperature, we return: bar{T} = sum_j T_j W_{ij}).

This function is the same as `project_gas` but does not include units.

Default projection variable is mass. If it is None, then we don't weight with anything, providing a number density image.

Parameters

- **data** (SWIFTDataset) – The SWIFT dataset that you wish to visualise (get this from `load`)
- **resolution** (*int*) – The resolution of the image. All images returned are square, `res` by `res`, pixel grids.
- **project** (*str, optional*) – Variable to project to get the weighted density of. By default, this is mass. If you would like to mass-weight any other variable, you can always create it as `data.gas.my_variable = data.gas.other_variable * data.gas.masses`.
- **region** (*unyt_array, optional*) – Region, determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges) if it is not None. It should have a length of four or six, and take the form: `[x_min, x_max, y_min, y_max, {z_min, z_max}]`
- **mask** (*np.array, optional*) – Allows only a sub-set of the particles in data to be visualised. Useful in cases where you have read data out of a `velociraptor` catalogue, or if you

only want to visualise e.g. star forming particles. This boolean mask is applied just before visualisation.

- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.
- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a projection along the z axis.
- **parallel** (*bool, optional*) – Defaults to `False`, whether or not to create the image in parallel. The parallel version of this function uses significantly more memory.
- **backend** (*str, optional*) – Backend to use. See documentation for details. Defaults to 'fast'.
- **periodic** (*bool, optional*) – Account for periodic boundary conditions for the simulation box? Defaults to `True`.

Returns
image – Projected image with dimensions of project / length^2, of size `res` x `res`.

Return type
np.array

Notes

- Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- The returned array has x as the first component and y as the second component, which is the opposite to what `imshow` requires. You should transpose the array if you want it to be visualised the 'right way up'.

swiftsimio.visualisation.projection.**project_gas**(*data: SWIFTDataset*, *resolution: int*, *project: str | None = 'masses'*, *region: None | unyt_array = None*, *mask: None | array = None*, *rotation_center: None | unyt_array = None*, *rotation_matrix: None | array = None*, *parallel: bool = False*, *backend: str = 'fast'*, *periodic: bool = True*)

Creates a 2D projection of a SWIFT dataset, projected by the "project" variable (e.g. if project is Temperature, we return: bar{T} = sum_j T_j W_{ij}).

Default projection variable is mass. If it is None, then we don't weight with anything, providing a number density image.

Parameters

- **data** (SWIFTDataset) – The SWIFT dataset that you wish to visualise (get this from `load`)
- **resolution** (*int*) – The resolution of the image. All images returned are square, `res` by `res`, pixel grids.
- **project** (*str, optional*) – Variable to project to get the weighted density of. By default, this is mass. If you would like to mass-weight any other variable, you can always create it as `data.gas.my_variable = data.gas.other_variable * data.gas.masses`.
- **region** (*unyt_array, optional*) – Region, determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges) if it is not None. It should have a length of four or six, and take the form: `[x_min, x_max, y_min, y_max, {z_min, z_max}]`

- **mask** (*np.array, optional*) – Allows only a sub-set of the particles in data to be visualised. Useful in cases where you have read data out of a `velociraptor` catalogue, or if you only want to visualise e.g. star forming particles. This boolean mask is applied just before visualisation.
- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.
- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a projection along the z axis.
- **parallel** (*bool, optional*) – Defaults to `False`, whether or not to create the image in parallel. The parallel version of this function uses significantly more memory.
- **backend** (*str, optional*) – Backend to use. See documentation for details. Defaults to 'fast'.
- **periodic** (*bool, optional*) – Account for periodic boundary conditions for the simulation box? Defaults to `True`.

Returns
image – Projected image with units of project / length^2, of size `res` x `res`.

Return type
unyt_array

Notes

- Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- The returned array has x as the first component and y as the second component, which is the opposite to what `imshow` requires. You should transpose the array if you want it to be visualised the 'right way up'.

swiftsimio.visualisation.rotation module

Rotation matrix calculation routines.

swiftsimio.visualisation.rotation.**rotation_matrix_from_vector**(*vector: float64*, *axis: str = 'z'*) → array

Calculate a rotation matrix from a vector. The comparison vector is assumed to be along an axis, x, y, or z (by default this is z). The resulting rotation matrix gives a rotation matrix to align the co-ordinate axes to make the projection be top-down along this axis.

Parameters

- **vector** (*np.array[float64]*) – 3D vector describing the top-down direction that you wish to rotate to. For example, this could be the angular momentum vector for a galaxy if you wish to produce a top-down projection.
- **axis** (*str, optional*) – String describing the axis to project along. This should be one of x, y, or z. Defaults to z.

Returns
rotation_matrix – Rotation matrix (3x3).

Return type
np.array[float64]

swiftsimio.visualisation.slice module

Sub-module for slice plots in SWFITSIMio.

swiftsimio.visualisation.slice.**kernel**(*r: float | float32*, *H: float | float32*)

Kernel implementation for swiftsimio.

Parameters

- **r** (*float or float32*) – Distance from particle
- **H** (*float or float32*) – Kernel width (i.e. radius of compact support of kernel)

Returns
Contribution to density by particle at distance *r*

Return type
float

Notes

Swiftsimio uses the Wendland-C2 kernel as described in[1].

References

swiftsimio.visualisation.slice.**slice_scatter**(*x: float64*, *y: float64*, *z: float64*, *m: float32*, *h: float32*, *z_slice: float64*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*, *box_z: float64 = 0.0*) → ndarray

Creates a scatter plot of the given quantities for a particles in a data slice including periodic boundary effects.

Parameters

- **x** (*array of float64*) – x-positions of the particles. Must be bounded by [0, 1].
- **y** (*array of float64*) – y-positions of the particles. Must be bounded by [0, 1].
- **z** (*array of float64*) – z-positions of the particles. Must be bounded by [0, 1].
- **m** (*array of float32*) – masses (or otherwise weights) of the particles
- **h** (*array of float32*) – smoothing lengths of the particles
- **z_slice** (*float64*) – the position at which we wish to create the slice
- **res** (*int*) – the number of pixels.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **box_z** (*float64*) – box size in z, in the same rescaled length units as x, y and z. Used for periodic wrapping.

Returns
output array for scatterplot image

Return type
ndarray of float32

[1] Dehnen W., Aly H., 2012, MNRAS, 425, 1068

See also:

`scatter`
Create 3D scatter plot of SWIFT data

`scatter_parallel`
Create 3D scatter plot of SWIFT data in parallel

`slice_scatter_parallel`
Create scatter plot of a slice of data in parallel

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

`swiftsimio.visualisation.slice.`**`slice_scatter_parallel`**(*x: float64*, *y: float64*, *z: float64*, *m: float32*, *h: float32*, *z_slice: float64*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*, *box_z: float64 = 0.0*) → ndarray

Parallel implementation of slice_scatter

Creates a scatter plot of the given quantities for a particles in a data slice including periodic boundary effects.

Parameters

- **`x`** (*array of float64*) – x-positions of the particles. Must be bounded by [0, 1].
- **`y`** (*array of float64*) – y-positions of the particles. Must be bounded by [0, 1].
- **`z`** (*array of float64*) – z-positions of the particles. Must be bounded by [0, 1].
- **`m`** (*array of float32*) – masses (or otherwise weights) of the particles
- **`h`** (*array of float32*) – smoothing lengths of the particles
- **`z_slice`** (*float64*) – the position at which we wish to create the slice
- **`res`** (*int*) – the number of pixels.
- **`box_x`** (*float64*) – box size in x, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **`box_y`** (*float64*) – box size in y, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **`box_z`** (*float64*) – box size in z, in the same rescaled length units as x, y and z. Used for periodic wrapping.

Returns
output array for scatterplot image

Return type
ndarray of float32

See also:

`scatter`
Create 3D scatter plot of SWIFT data

`scatter_parallel`
Create 3D scatter plot of SWIFT data in parallel

slice_scatter
Create scatter plot of a slice of data

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.slice.**slice_gas_pixel_grid**(*data: SWIFTDataset*, *resolution: int*, *z_slice: unyt_quantity | None = None*, *project: str | None = 'masses'*, *parallel: bool = False*, *rotation_matrix: None | array = None*, *rotation_center: None | unyt_array = None*, *region: None | unyt_array = None*, *periodic: bool = True*)

Creates a 2D slice of a SWIFT dataset, weighted by data field, in the form of a pixel grid.

Parameters

- **data** (SWIFTDataset) – Dataset from which slice is extracted
- **resolution** (*int*) – Specifies size of return array
- **z_slice** (*unyt_quantity*) – Specifies the location along the z-axis where the slice is to be extracted, relative to the rotation center or the origin of the box if no rotation center is provided. If the perspective is rotated this value refers to the location along the rotated z-axis.
- **project** (*str, optional*) – Data field to be projected. Default is mass. If None then simply count number of particles
- **parallel** (*bool*) – used to determine if we will create the image in parallel. This defaults to False, but can speed up the creation of large images significantly at the cost of increased memory usage.
- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a slice perpendicular to the z axis.
- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.
- **region** (*unyt_array, optional*) – determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges) if it is not None. It should have a length of four, and take the form:

  [x_min, x_max, y_min, y_max]

  Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- **periodic** (*bool, optional*) – Account for periodic boundaries for the simulation box? Default is `True`.

Returns
Creates a *resolution* x *resolution* array and returns it, without appropriate units.

Return type
ndarray of float32

See also:

render_gas_voxel_grid
Creates a 3D voxel grid from a SWIFT dataset

swiftsimio.visualisation.slice.**slice_gas**(*data: SWIFTDataset*, *resolution: int*, *z_slice: unyt_quantity | None = None*, *project: str | None = 'masses'*, *parallel: bool = False*, *rotation_matrix: None | array = None*, *rotation_center: None | unyt_array = None*, *region: None | unyt_array = None*, *periodic: bool = True*)

Creates a 2D slice of a SWIFT dataset, weighted by data field

Parameters

- **data** (SWIFTDataset) – Dataset from which slice is extracted
- **resolution** (*int*) – Specifies size of return array
- **z_slice** (*unyt_quantity*) – Specifies the location along the z-axis where the slice is to be extracted, relative to the rotation center or the origin of the box if no rotation center is provided. If the perspective is rotated this value refers to the location along the rotated z-axis.
- **project** (*str, optional*) – Data field to be projected. Default is mass. If None then simply count number of particles
- **parallel** (*bool, optional*) – used to determine if we will create the image in parallel. This defaults to False, but can speed up the creation of large images significantly at the cost of increased memory usage.
- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a slice perpendicular to the z axis.
- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.
- **region** (*array, optional*) – determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges) if it is not None. It should have a length of four, and take the form:

  [x_min, x_max, y_min, y_max]

  Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- **periodic** (*bool, optional*) – Account for periodic boundaries for the simulation box? Default is `True`.

Returns
a *resolution* x *resolution* array of the contribution of the projected data field to the voxel grid from all of the particles

Return type
ndarray of float32

See also:

`slice_gas_pixel`

render_gas
Creates a 3D voxel grid of a SWIFT dataset with appropriate units

Notes

This is a wrapper function for slice_gas_pixel_grid ensuring that output units are appropriate

swiftsimio.visualisation.smoothing_length_generation module

Routines for generating (approximate) smoothing lengths for particles that do not usually carry a smoothing length field (e.g. dark matter).

swiftsimio.visualisation.smoothing_length_generation.**generate_smoothing_lengths**(*coordinates: unyt_array | cosmo_array*, *boxsize: unyt_array | cosmo_array*, *kernel_gamma: float32*, *neighbours=32*, *speedup_fac=2*, *dimension=3*)

Generates smoothing lengths that encompass a number of neighbours specified here.

Parameters

- **coordinates** (*unyt_array or cosmo_array*) – a cosmo_array that gives the coordinates of all particles
- **boxsize** (*unyt_array or cosmo_array*) – the size of the box (3D)
- **kernel_gamma** (*float32*) – the kernel gamma of the kernel being used
- **neighbours** (*int, optional*) – the number of neighbours to encompass
- **speedup_fac** (*int, optional*) – a parameter that neighbours is divided by to provide a speed-up by only searching for a lower number of neighbours. For example, if neighbours is 32, and speedup_fac is 2, we only search for 16 (32 / 2) neighbours, and extend the smoothing length out to (speedup)**(1/dimension) such that we encompass an approximately higher number of neighbours. A factor of 2 gives smoothing lengths the same as the full search within 10%, good enough for visualisation.
- **dimension** (*int, optional*) – the dimensionality of the problem (used for speedup_fac calculation).

Returns

smoothing lengths – an unyt array of smoothing lengths.

Return type

unyt_array

swiftsimio.visualisation.volume_render module

Basic volume render for SPH data. This takes the 3D positions of the particles and projects them onto a grid.

swiftsimio.visualisation.volume_render.**scatter**(*x: float64*, *y: float64*, *z: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*, *box_z: float64 = 0.0*) → ndarray

Creates a weighted voxel grid

Computes contributions to a voxel grid from particles with positions (*x*,`y`,`z`) with smoothing lengths *h* weighted by quantities *m*. This includes periodic boundary effects.

Parameters

- **x** (*np.array[float64]*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*np.array[float64]*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **z** (*np.array[float64]*) – array of z-positions of the particles. Must be bounded by [0, 1].
- **m** (*np.array[float32]*) – array of masses (or otherwise weights) of the particles
- **h** (*np.array[float32]*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of voxels along one axis, i.e. this returns a cube of res * res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **box_z** (*float64*) – box size in z, in the same rescaled length units as x, y and z. Used for periodic wrapping

Returns
voxel grid of quantity

Return type
np.array[float32, float32, float32]

See also:

scatter_parallel
Parallel implementation of this function

slice_scatter
Create scatter plot of a slice of data

slice_scatter_parallel
Create scatter plot of a slice of data in parallel

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.volume_render.**scatter_parallel**(*x: float64*, *y: float64*, *z: float64*, *m: float32*, *h: float32*, *res: int*, *box_x: float64 = 0.0*, *box_y: float64 = 0.0*, *box_z: float64 = 0.0*) → ndarray

Parallel implementation of scatter

Compute contributions to a voxel grid from particles with positions (x,`y`,`z`) with smoothing lengths h weighted by quantities m. This ignores boundary effects.

Parameters

- **x** (*array of float64*) – array of x-positions of the particles. Must be bounded by [0, 1].
- **y** (*array of float64*) – array of y-positions of the particles. Must be bounded by [0, 1].
- **z** (*array of float64*) – array of z-positions of the particles. Must be bounded by [0, 1].
- **m** (*array of float32*) – array of masses (or otherwise weights) of the particles
- **h** (*array of float32*) – array of smoothing lengths of the particles
- **res** (*int*) – the number of voxels along one axis, i.e. this returns a cube of res * res * res.
- **box_x** (*float64*) – box size in x, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **box_y** (*float64*) – box size in y, in the same rescaled length units as x, y and z. Used for periodic wrapping.
- **box_z** (*float64*) – box size in z, in the same rescaled length units as x, y and z. Used for periodic wrapping

Returns
voxel grid of quantity

Return type
ndarray of float32

See also:

scatter
Create voxel grid of quantity

slice_scatter
Create scatter plot of a slice of data

slice_scatter_parallel
Create scatter plot of a slice of data in parallel

Notes

Explicitly defining the types in this function allows for a 25-50% performance improvement. In our testing, using numpy floats and integers is also an improvement over using the numba ones.

swiftsimio.visualisation.volume_render.**render_gas_voxel_grid**(*data: SWIFTDataset*, *resolution: int*, *project: str | None = 'masses'*, *parallel: bool = False*, *rotation_matrix: None | array = None*, *rotation_center: None | unyt_array = None*, *region: None | unyt_array = None*, *periodic: bool = True*)

Creates a 3D render of a SWIFT dataset, weighted by data field, in the form of a voxel grid.

Parameters

- **data** (SWIFTDataset) – Dataset from which slice is extracted
- **resolution** (*int*) – Specifies size of return array
- **project** (*str, optional*) – Data field to be projected. Default is mass. If None then simply count number of particles
- **parallel** (*bool*) – used to determine if we will create the image in parallel. This defaults to False, but can speed up the creation of large images significantly at the cost of increased memory usage.
- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a volume render viewed along the z axis.
- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.
- **region** (*unyt_array, optional*) – determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges, and front and back edges) if it is not None. It should have a length of six, and take the form:

  [x_min, x_max, y_min, y_max, z_min, z_max]

  Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- **periodic** (*bool, optional*) – Account for periodic boundaries for the simulation box? Default is `True`.

Returns

Creates a *resolution* x *resolution* x *resolution* array and returns it, without appropriate units.

Return type

ndarray of float32

See also:

slice_gas_pixel_grid

Creates a 2D slice of a SWIFT dataset

swiftsimio.visualisation.volume_render.**render_gas**(*data: SWIFTDataset*, *resolution: int*, *project: str | None = 'masses'*, *parallel: bool = False*, *rotation_matrix: None | array = None*, *rotation_center: None | unyt_array = None*, *region: None | unyt_array = None*, *periodic: bool = True*)

Creates a 3D voxel grid of a SWIFT dataset, weighted by data field

Parameters

- **data** (SWIFTDataset) – Dataset from which slice is extracted
- **resolution** (*int*) – Specifies size of return array
- **project** (*str, optional*) – Data field to be projected. Default is mass. If None then simply count number of particles
- **parallel** (*bool*) – used to determine if we will create the image in parallel. This defaults to False, but can speed up the creation of large images significantly at the cost of increased memory usage.
- **rotation_matrix** (*np.array, optional*) – Rotation matrix (3x3) that describes the rotation of the box around `rotation_center`. In the default case, this provides a volume render viewed along the z axis.
- **rotation_center** (*np.array, optional*) – Center of the rotation. If you are trying to rotate around a galaxy, this should be the most bound particle.
- **region** (*unyt_array, optional*) – determines where the image will be created (this corresponds to the left and right-hand edges, and top and bottom edges, and front and back edges) if it is not None. It should have a length of six, and take the form: `[x_min, x_max, y_min, y_max, z_min, z_max]` Particles outside of this range are still considered if their smoothing lengths overlap with the range.
- **periodic** (*bool, optional*) – Account for periodic boundaries for the simulation box? Default is `True`.

Returns

a *resolution* x *resolution* x *resolution* array of the contribution of the projected data field to the voxel grid from all of the particles

Return type

ndarray of float32

See also:

slice_gas

Creates a 2D slice of a SWIFT dataset with appropriate units

render_gas_voxel_grid

Creates a 3D voxel grid of a SWIFT dataset

Notes

This is a wrapper function for slice_gas_pixel_grid ensuring that output units are appropriate

9.1.2 Submodules

swiftsimio.accelerated module

Functions that can be accelerated by numba. Numba does not use classes, unfortunately.

swiftsimio.accelerated.**ranges_from_array**(*array: array*) → ndarray

Finds contiguous ranges of IDs in sorted list of IDs

Parameters
array (*np.array of int*) – sorted list of IDs

Returns
list of length two arrays corresponding to contiguous ranges of IDs (inclusive) in the input array

Return type
np.ndarray

Examples

The array

[0, 1, 2, 3, 5, 6, 7, 9, 11, 12, 13]

would return

[[0, 4], [5, 8], [9, 10], [11, 14]]

swiftsimio.accelerated.**read_ranges_from_file_unchunked**(*handle: ~h5py._hl.dataset.Dataset*, *ranges: ~numpy.ndarray*, *output_shape: ~typing.Tuple*, *output_type: type = <class 'numpy.float64'>*, *columns: ~numpy.lib.index_tricks.IndexExpression = slice(None, None, None)*) → array

Takes a hdf5 dataset, and the set of ranges from ranges_from_array, and reads only those ranges from the file.

Unfortunately this functionality is not built into HDF5.

Parameters

- **handle** (*Dataset*) – HDF5 dataset to slice data from
- **ranges** (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)
- **output_shape** (*Tuple*) – Resultant shape of output.
- **output_type** (*type, optional*) – numpy type of output elements. If not supplied, we assume np.float64.
- **columns** (*np.lib.index_tricks.IndexExpression, optional*) – Selector for columns if using a multi-dimensional array. If the array is only a single dimension this is not used.

Returns
array – Result from reading only the relevant values from handle.

Return type
np.ndarray

swiftsimio.accelerated.**index_dataset**(*handle: Dataset*, *mask_array: array*) → array

Indexes the dataset using the mask array.

This is not currently a feature of h5py. (March 2019)

Parameters

- **handle** (*Dataset*) – data to be indexed
- **mask_array** (*np.array*) – mask used to index data

Returns
Subset of the data specified by the mask

Return type
np.array

swiftsimio.accelerated.**concatenate_ranges**(*ranges: ndarray*) → ndarray

Returns an array of ranges with consecutive ranges merged if there is no gap between them

Parameters
ranges (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)

Returns
two dimensional array of ranges

Return type
np.ndarray

Examples

```
>>> concatenate_ranges([[1,5],[6,10],[12,15]])
np.ndarray([[1,10],[12,15]])
```

swiftsimio.accelerated.**get_chunk_ranges**(*ranges: ndarray*, *chunk_size: ndarray*, *array_length: int*) → ndarray

Return indices indicating which hdf5 chunk each range from *ranges* belongs to

Parameters

- **ranges** (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)
- **chunk_size** (*int*) – size of the hdf5 dataset chunks
- **array_length** (*int*) – size of the dataset

Returns
two dimensional array of bounds for the chunks that contain each range from *ranges*

Return type
np.ndarray

swiftsimio.accelerated.**expand_ranges**(*ranges: ndarray*) → array

Return an array of indices that are within the specified ranges

Parameters
ranges (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)

Returns
1D array of indices that fall within each range specified in *ranges*

Return type
np.array

swiftsimio.accelerated.**extract_ranges_from_chunks**(*array: ndarray*, *chunks: ndarray*, *ranges: ndarray*) → ndarray

Returns elements from array that are located within specified ranges

array is a portion of the dataset being read consisting of all the chunks that contain the ranges specified in *ranges*. The *chunks* array contains the indices of the upper and lower bounds of these chunks. To find the elements of the dataset that lie within the specified ranges we first create an array indexing which chunk each range belongs to. From this information we create an array of adjusted ranges that takes into account that the array is not the whole dataset. We then return the values in *array* that are within the adjusted ranges.

Parameters

- **array** (*np.ndarray*) – array containing data read in from snapshot
- **chunks** (*np.ndarray*) – two dimensional array of bounds for the chunks that contain each range from *ranges*
- **ranges** (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)

Returns
subset of *array* whose elements are within each range in *ranges*

Return type
np.ndarray

swiftsimio.accelerated.**read_ranges_from_file_chunked**(*handle: ~h5py._hl.dataset.Dataset*, *ranges: ~numpy.ndarray*, *output_shape: ~typing.Tuple*, *output_type: type = <class 'numpy.float64'>*, *columns: ~numpy.lib.index_tricks.IndexExpression = slice(None, None, None)*) → array

Takes a hdf5 dataset, and the set of ranges from ranges_from_array, and reads only those ranges from the file.

Unfortunately this functionality is not built into HDF5.

Parameters

- **handle** (*Dataset*) – HDF5 dataset to slice data from
- **ranges** (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)
- **output_shape** (*Tuple*) – Resultant shape of output.
- **output_type** (*type, optional*) – numpy type of output elements. If not supplied, we assume np.float64.
- **columns** (*np.lib.index_tricks.IndexExpression, optional*) – Selector for columns if using a multi-dimensional array. If the array is only a single dimension this is not used.

Returns
array – Result from reading only the relevant values from handle.

Return type
np.ndarray

swiftsimio.accelerated.**read_ranges_from_file**(*handle: ~h5py._hl.dataset.Dataset*, *ranges: ~numpy.ndarray*, *output_shape: ~typing.Tuple*, *output_type: type = <class 'numpy.float64'>*, *columns: ~numpy.lib.index_tricks.IndexExpression = slice(None, None, None)*) → array

Wrapper function to correctly select which version of read_ranges_from_file should be used

Parameters

- **handle** (*Dataset*) – HDF5 dataset to slice data from
- **ranges** (*np.ndarray*) – Array of ranges (see *ranges_from_array()*)
- **output_shape** (*Tuple*) – Resultant shape of output.
- **output_type** (*type, optional*) – numpy type of output elements. If not supplied, we assume np.float64.
- **columns** (*np.lib.index_tricks.IndexExpression, optional*) – Selector for columns if using a multi-dimensional array. If the array is only a single dimension this is not used.

Returns

array – Result from reading only the relevant values from handle.

Return type

np.ndarray

See also:

read_ranges_from_file_chunked

reads data within specified ranges for chunked hdf5

file, unchunked

swiftsimio.accelerated.**list_of_strings_to_arrays**(*lines: List[str]*) → array

Converts a list of space-delimited values to arrays.

Parameters

lines (*List[str]*) – List of strings containing numbers separated by a set of spaces.

Returns

arrays – List of numpy arrays, one per column.

Return type

List[np.array]

Notes

Currently not suitable for numba acceleration due to mixed datatype usage.

swiftsimio.conversions module

Includes conversions between SWIFT internal values and `astropy` ones for convenience.

`swiftsimio.conversions.`**`swift_cosmology_to_astropy`**(*cosmo: dict*, *units*) → dict

swiftsimio.masks module

Loading functions and objects that use masked information from the SWIFT snapshots.

class `swiftsimio.masks.`**`SWIFTMask`**(*metadata:* SWIFTMetadata, *spatial_only=True*)

Bases: `object`

Main masking object. This can have masks for any present particle field in it. Pass in the SWIFTMetadata.

`constrain_mask`(*ptype: str*, *quantity: str*, *lower: unyt_quantity*, *upper: unyt_quantity*)

Constrains the mask further for a given particle type, and bounds a quantity between lower and upper values.

We update the mask such that

lower < ptype.quantity <= upper

The quantities must have units attached.

Parameters

- **`ptype`** (*`str`*) – particle type
- **`quantity`** (*`str`*) – quantity being constrained
- **`lower`** (*`unyt.array.unyt_quantity`*) – constraint lower bound
- **`upper`** (*`unyt.array.unyt_quantity`*) – constraint upper bound

See also:

`constrain_spatial`
method to generate spatially constrained cell mask

`constrain_spatial`(*restrict*)

Uses the cell metadata to create a spatial mask.

This mask is necessarily approximate and is coarse-grained to the cell size.

Parameters

`restrict` (*`list`*) – length 3 list of length two arrays giving the lower and upper bounds for that axis, e.g.

restrict = [
[0.5, 0.7], [0.1, 0.9], [0.0, 0.1]

]

These values must have units associated with them. It is also acceptable to have a row as None to not restrict in this direction.

See also:

`constrain_mask`
method to further refine mask

convert_masks_to_ranges()

Converts the masks to range masks so that they take up less space.

This is non-reversible. It is also not required, but can help save space on highly constrained machines before you start reading in the data.

If you don't know what you are doing please don't use this.

get_masked_counts_offsets() -> *(typing.Dict[str, <built-in function array>], typing.Dict[str, <built-in function array>])*

Returns the particle counts and offsets in cells selected by the mask

Returns

Dictionaries containing the particle offets and counts for each particle type. For example, the particle counts dictionary would be of the form

```
{"gas": [g_0, g_1, ...],
 "dark matter": [bh_0, bh_1, ...], ...}
```

where the keys would be each of the particle types and values are arrays of the number of corresponding particles in each cell (in this case there would be g_0 gas particles in the first cell, g_1 in the second, etc.). The structure of the dictionaries is the same for the offsets, with the arrays now storing the offset of the first particle in the cell.

Return type
Dict[str, np.array], Dict[str, np.array]

swiftsimio.objects module

Contains global objects, e.g. the superclass version of the unyt_array that we use, called cosmo_array.

exception swiftsimio.objects.**InvalidScaleFactor**(*message=None*, **args*)

Bases: Exception

Raised when a scale factor is invalid, such as when adding two cosmo_factors with inconsistent scale factors.

class swiftsimio.objects.**cosmo_factor**(*expr*, *scale_factor*)

Bases: object

Cosmology factor class for storing and computing conversion between comoving and physical coordinates.

This takes the expected exponent of the array that can be parsed by sympy, and the current value of the cosmological scale factor a.

This should be given as the conversion from comoving to physical, i.e.

r = cosmo_factor * r' with r in physical and r' comoving

Examples

Typically this would make cosmo_factor = a for the conversion between comoving positions r' and physical co-ordinates r.

To do this, use the a imported from objects multiplied as you'd like:

```
density_cosmo_factor = cosmo_factor(a**3, scale_factor=0.97)
```

property a_factor

The a-factor for the unit.

e.g. for density this is 1 / a**3.

Returns
the a-factor for given unit

Return type
float

property redshift

Compute the redshift from the scale factor.

Returns
redshift from the given scale factor

Return type
float

Notes

Returns the redshift ..math:: z = frac{1}{a} - 1, where :math: *a* is the scale factor

class swiftsimio.objects.**cosmo_array**(*input_array*, *units=None*, *registry=None*, *dtype=None*, *bypass_validation=False*, *input_units=None*, *name=None*, *cosmo_factor=None*, *comoving=True*, *compression=None*)

Bases: unyt_array

Cosmology array class.

This inherits from the unyt.unyt_array, and adds three variables: compression, cosmo_factor, and comoving. Data is assumed to be comoving when passed to the object but you can override this by setting the latter flag to be False.

Parameters
unyt_array (*unyt.unyt_array*) – the inherited unyt_array

comoving

if True then the array is in comoving co-ordinates, and if False then it is in physical units.

Type
bool

cosmo_factor

Object to store conversion data between comoving and physical coordinates

Type
float

compression

String describing any compression that was applied to this array in the hdf5 file.

Type
string

astype(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

Parameters

- **dtype** (*str or dtype*) – Typecode or data-type to which the array is cast.
- **order** (*{'C', 'F', 'A', 'K'}, optional*) – Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.
- **casting** (*{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional*) – Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.
  - 'no' means the data types should not be cast at all.
  - 'equiv' means only byte-order changes are allowed.
  - 'safe' means only casts which can preserve values are allowed.
  - 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
  - 'unsafe' means any data conversions may be done.
- **subok** (*bool, optional*) – If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.
- **copy** (*bool, optional*) – By default, astype always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

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

Return type
ndarray

Notes

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

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

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

Examples

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

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

in_units(**args*, ***kwargs*)

Creates a copy of this array with the data converted to the supplied units, and returns it.

Optionally, an equivalence can be specified to convert to an equivalent quantity which is not in the same dimensions.

Parameters

- **units** (*Unit object or string*) – The units you want to get a new quantity in.
- **equivalence** (*string, optional*) – The equivalence you wish to use. To see which equivalencies are supported for this object, try the `list_equivalencies` method. Default: None
- **kwargs** (*optional*) – Any additional keyword arguments are supplied to the equivalence

Raises

- **If the provided unit does not have the same dimensions as the array** –
- **this will raise a UnitConversionError** –

Examples

```
>>> from unyt import c, gram
>>> m = 10*gram
>>> E = m*c**2
>>> print(E.in_units('erg'))
8.987551787368176e+21 erg
>>> print(E.in_units('J'))
898755178736817.6 J
```

byteswap(*inplace=False*)

Swap the bytes of the array elements

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

Parameters
inplace (*bool, optional*) – If `True`, swap bytes in-place, default is `False`.

Returns
out – The byteswapped array. If *inplace* is `True`, this is a view to self.

Return type
ndarray

Examples

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

Arrays of byte-strings are not swapped

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

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

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

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

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

See also:

numpy.compress
equivalent function

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

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

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

See also:

numpy.diagonal
equivalent function

flatten(*order='C'*)

Return a copy of the array collapsed into one dimension.

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

Returns
y – A copy of the input array, flattened to one dimension.

Return type
ndarray

See also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

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

newbyteorder(*new_order='S', /*)

Return the array with the same data viewed with a different byte order.

Equivalent to:

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

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

Parameters
new_order (*string, optional*) – Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian
- {'=', 'native'} - native order, equivalent to *sys.byteorder*
- {'|', 'I'} - ignore (no change to byte order)

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

Returns
new_arr – New array object with the dtype reflecting given change to the byte order.

Return type
array

ravel([*order*])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See also:

numpy.ravel
equivalent function

ndarray.flat
a flat iterator on the array.

repeat(*repeats*, *axis=None*)

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See also:

numpy.repeat
equivalent function

reshape(*shape*, *order='C'*)

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See also:

numpy.reshape
equivalent function

Notes

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

swapaxes(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

See also:

numpy.swapaxes
equivalent function

take(*indices*, *axis=None*, *out=None*, *mode='raise'*)

Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

See also:

numpy.take
equivalent function

transpose(**axes*)

Returns a view of the array with axes transposed.

Refer to *numpy.transpose* for full documentation.

Parameters

axes (None, tuple of ints, or *n* ints) –

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

Returns

p – View of the array with its axes suitably permuted.

Return type

ndarray

See also:

transpose

Equivalent function.

ndarray.T

Array property returning the array transposed.

ndarray.reshape

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

Examples

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

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

view(*[dtype][, type]*)

New view of array with the same data.

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

Parameters

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

Notes

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

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

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

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of `a` must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

Examples

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

Viewing array data using a different type and dtype:

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

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

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

(continues on next page)

(continued from previous page)

```
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

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

Using a view to convert an array to a recarray:

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

Views share data:

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

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

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

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[[ 256,  770],
        [3340, 3854]],

       [[1284, 1798],
        [4368, 4882]],

       [[2312, 2826],
        [5396, 5910]]], dtype=int16)
```

property T

View of the transposed array.

Same as `self.transpose()`.

Examples

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

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

See also:

transpose

property ua

Return an array filled with ones with the same units as this array

Example

```
>>> from unyt import km
>>> a = [4, 5, 6]*km
>>> a.unit_array
unyt_array([1, 1, 1], 'km')
>>> print(a + 7*a.unit_array)
[11 12 13] km
```

property unit_array

Return an array filled with ones with the same units as this array

Example

```
>>> from unyt import km
>>> a = [4, 5, 6]*km
>>> a.unit_array
unyt_array([1, 1, 1], 'km')
>>> print(a + 7*a.unit_array)
[11 12 13] km
```

convert_to_comoving() → None

Convert the internal data to be in comoving units.

convert_to_physical() → None

Convert the internal data to be in physical units.

to_physical()

Creates a copy of the data in physical units.

Returns
copy of cosmo_array in physical units

Return type
cosmo_array

to_comoving()

Creates a copy of the data in comoving units.

Returns
copy of cosmo_array in comoving units

Return type
cosmo_array

compatible_with_comoving()

Is this cosmo_array compatible with a comoving cosmo_array?

This is the case if the cosmo_array is comoving, or if the scale factor exponent is 0 (cosmo_factor.a_factor() == 1)

compatible_with_physical()

Is this cosmo_array compatible with a physical cosmo_array?

This is the case if the cosmo_array is physical, or if the scale factor exponent is 0 (cosmo_factor.a_factor == 1)

classmethod from_astropy(*arr*, *unit_registry=None*, *comoving=True*, *cosmo_factor=None*, *compression=None*)

Convert an AstroPy "Quantity" to a cosmo_array.

Parameters

- **arr** (*AstroPy Quantity*) – The Quantity to convert from.
- **unit_registry** (*yt UnitRegistry, optional*) – A yt unit registry to use in the conversion. If one is not supplied, the default one will be used.
- **comoving** (*bool*) – if True then the array is in comoving co-ordinates, and if False then it is in physical units.
- **cosmo_factor** (*float*) – Object to store conversion data between comoving and physical coordinates
- **compression** (*string*) – String describing any compression that was applied to this array in the hdf5 file.

Example

```
>>> from astropy.units import kpc
>>> cosmo_array.from_astropy([1, 2, 3] * kpc)
cosmo_array([1., 2., 3.], 'kpc')
```

classmethod from_pint(*arr*, *unit_registry=None*, *comoving=True*, *cosmo_factor=None*, *compression=None*)

Convert a Pint "Quantity" to a cosmo_array.

Parameters

- **arr** (*Pint Quantity*) – The Quantity to convert from.
- **unit_registry** (*yt UnitRegistry, optional*) – A yt unit registry to use in the conversion. If one is not supplied, the default one will be used.
- **comoving** (*bool*) – if True then the array is in comoving co-ordinates, and if False then it is in physical units.
- **cosmo_factor** (*float*) – Object to store conversion data between comoving and physical coordinates
- **compression** (*string*) – String describing any compression that was applied to this array in the hdf5 file.

Examples

```
>>> from pint import UnitRegistry
>>> import numpy as np
>>> ureg = UnitRegistry()
>>> a = np.arange(4)
>>> b = ureg.Quantity(a, "erg/cm**3")
>>> b
<Quantity([0 1 2 3], 'erg / centimeter ** 3')>
>>> c = cosmo_array.from_pint(b)
>>> c
cosmo_array([0, 1, 2, 3], 'erg/cm**3')
```

swiftsimio.optional_packages module

Imports of optional packages.

This includes:

- tqdm: progress bars
- scipy.spatial: KDTrees
- numba/cuda: visualisation

swiftsimio.optional_packages.**tqdm**(*x*, **args*, ***kwargs*)

swiftsimio.optional_packages.**cuda_jit**(**args*, ***kwargs*)

swiftsimio.reader module

This file contains four major objects:

- SWIFTUnits, which is a unit system that can be queried for units (and converts arrays to relevant unyt arrays when read from the HDF5 file)
- SWIFTMetadata, which contains all of the metadata from the file
- __SWIFTParticleDataset, which contains particle information but should never be directly accessed. Use generate_dataset to create one of these. The reasoning here is that properties can only be added to the class afterwards, and not directly in an _instance_ of the class.
- SWIFTDataset, a container class for all of the above.

class swiftsimio.reader.**MassTable**(*base_mass_table: array*, *mass_units: unyt_quantity*)

Bases: object

Extracts a mass table to local variables based on the particle type names.

class swiftsimio.reader.**MappingTable**(*data: ndarray*, *named_columns_x: List[str]*, *named_columns_y: List[str]*, *named_columns_x_name: str*, *named_columns_y_name: str*)

Bases: object

A mapping table from one named column instance to the other. Initially designed for the mapping between dust and elements.

class swiftsimio.reader.**SWIFTUnits**(*filename*)

Bases: object

Generates a unyt system that can then be used with the SWIFT data.

These give the unit mass, length, time, current, and temperature as unyt unit variables in simulation units. I.e. you can take any value that you get out of the code and multiply it by the appropriate values to get it 'unyt-ified' with the correct units.

mass

unit for mass used

Type
float

length

unit for length used

Type
float

time

unit for time used

Type
float

current

unit for current used

Type
float

temperature

unit for temperature used

Type
float

get_unit_dictionary()

Store unit data and metadata

Length 1 arrays are used to store the unit data. This dictionary also contains the metadata information that connects the unyt objects to the names that are stored in the SWIFT snapshots.

class swiftsimio.reader.**SWIFTMetadata**(*filename*, *units:* SWIFTUnits)

Bases: object

Loads all metadata (apart from Units, those are handled by SWIFTUnits) into dictionaries.

This also does some extra parsing on some well-used metadata.

header: dict

filename: str

units: *SWIFTUnits*

get_metadata()

Loads the metadata as specified in metadata.metadata_fields.

get_named_column_metadata()

Loads the custom named column metadata (if it exists) from SubgridScheme/NamedColumns.

get_mapping_metadata()

Gets the mappings based on the named columns (must have already been read), from the form:

SubgridScheme/{X}To{Y}Mapping.

Includes a hack of *Dust -> Grains* that will be deprecated.

postprocess_header()

Some minor postprocessing on the header to local variables.

load_particle_types()

Loads the particle types and metadata into objects:

metadata.<type>_properties

This contains six arrays,

metadata.<type>_properties.field_names metadata.<type>_properties.field_paths metadata.<type>_properties.field_units metadata.<type>_properties.field_cosmologies metadata.<type>_properties.field_descriptions metadata.<type>_properties.field_compressions

As well as some more information about the particle type.

extract_cosmology()

Creates an astropy.cosmology object from the internal cosmology system.

This will be saved as `self.cosmology`.

property present_particle_types

The particle types that are present in the file.

property present_particle_names

The particle _names_ that are present in the simulation.

property code_info

Gets a nicely printed set of code information with:

Name (Git Branch) Git Revision Git Date

property compiler_info

Gets information about the compiler and formats it as:

Compiler Name (Compiler Version) MPI library

property library_info

Gets information about the libraries used and formats it as:

FFTW vFFTW library version GSL vGSL library version HDF5 vHDF5 library version

property hydro_info

Gets information about the hydro scheme and formats it as:

Scheme Kernel function in DimensionD eta = Kernel eta (Kernel target N_ngb N_{ngb}) $C_{rm CFL}$ = CFL parameter

property viscosity_info

Gets information about the viscosity scheme and formats it as:

Viscosity Model $alpha_{V, 0}$ = Alpha viscosity, ell_V = Viscosity decay length [internal units], $beta_V$ = Beta viscosity Alpha viscosity (min) < $alpha_V$ < Alpha viscosity (max)

property diffusion_info

Gets information about the diffusion scheme and formats it as:

$lpha_{D, 0}$ = Diffusion alpha, eta_D = Diffusion beta Diffusion alpha (min) < $lpha_D$ < Diffusion alpha (max)

class swiftsimio.reader.**SWIFTParticleTypeMetadata**(*particle_type: int*, *particle_name: str*, *metadata:* SWIFTMetadata, *scale_factor: float*)

Bases: object

Object that contains the metadata for one particle type.

This, for instance, could be part type 0, or 'gas'. This will load in the names of all particle datasets, their units, possible named fields, and their cosmology, and present them for use in the actual i/o routines.

load_metadata(self):

Loads the required metadata.

load_field_names(self):

Loads in the field names.

load_field_units(self):

Loads in the units from each dataset.

load_field_descriptions(self):

Loads in descriptions of the fields for each dataset.

load_field_compressions(self):

Loads in compressions of the fields for each dataset.

load_cosmology(self):

Loads in the field cosmologies.

load_named_columns(self):

Loads the named column data for relevant fields.

load_metadata()

Loads the required metadata.

This includes loading the field names, units and descriptions, as well as the cosmology metadata and any custom named columns

load_field_names()

Loads in only the field names.

load_field_units()

Loads in the units from each dataset.

load_field_descriptions()

Loads in the text descriptions of the fields for each dataset.

load_field_compressions()

Loads in the string describing the compression filters of the fields for each dataset.

load_cosmology()

Loads in the field cosmologies.

load_named_columns()

Loads the named column data for relevant fields.

swiftsimio.reader.**generate_getter**(*filename*, *name: str*, *field: str*, *unit: unyt_quantity*, *mask: None | ndarray*, *mask_size: int*, *cosmo_factor:* cosmo_factor, *description: str*, *compression: str*, *columns: None | IndexExpression = None*)

Generates a function that:

a) If self._`name` exists, return it

b) If not, open *filename*

c) Reads filename[*field*]

d) Set self._`name`

e) Return self._`name`.

Parameters

- **filename** (*str*) – Filename of the HDF5 file that everything will be read from. Used to generate the HDF5 dataset.
- **name** (*str*) – Output name (snake_case) of the field.
- **field** (*str*) – Full path of field, including e.g. particle type. Examples include `/PartType0/Velocities`.
- **unit** (*unyt.unyt_quantity*) – Output unit of the resultant `cosmo_array`
- **mask** (*None or np.ndarray*) – Mask to be used with `accelerated.read_ranges_from_file`, i.e. an array of integers that describe ranges to be read from the file.
- **mask_size** (*int*) – Size of the mask if present.

- **cosmo_factor** (cosmo_factor) – Cosmology factor object corresponding to this array.
- **description** (*str*) – Description (read from HDF5 file) of the data.
- **compression** (*str*) – String describing the lossy compression filters that were applied to the data (read from the HDF5 file).
- **columns** (*np.lib.index_tricks.IndexExpression, optional*) – Index expression corresponding to which columns to read from the numpy array. If not provided, we read all columns and return an n-dimensional array.

Returns
getter – A callable object that gets the value of the array that has been saved to _name. This function takes only self from the :obj:__SWIFTParticleDataset class.

Return type
callable

Notes

The major use of this function is for its side effect of setting _name as a member of the class on first read. When the attribute is accessed, it will be dynamically read from the file, to keep initial memory usage as minimal as possible.

If the resultant array is modified, it will not be re-read from the file.

swiftsimio.reader.**generate_setter**(*name: str*)

Generates a function that sets self._name to the value that is passed to it.

Parameters
name (*str*) – the name of the attribute to set

Returns
setter – a callable object that sets the attribute specified by name to the value passed to it.

Return type
callable

swiftsimio.reader.**generate_deleter**(*name: str*)

Generates a function that destroys self._name (sets it back to None).

Parameters
name (*str*) – the name of the field to be destroyed

Returns
deleter – callable that destroys name field

Return type
callable

swiftsimio.reader.**generate_dataset**(*particle_metadata:* SWIFTParticleTypeMetadata, *mask*)

Generates a SWIFTParticleDataset _class_ that corresponds to the particle type given.

We _must_ do the following _outside_ of the class itself, as one can assign properties to a _class_ but not _within_ a class dynamically.

Here we loop through all of the possible properties in the metadata file. We then use the builtin property() function and some generators to create setters and getters for those properties. This will allow them to be accessed from outside by using SWIFTParticleDataset.name, where the name is, for example, coordinates.

Parameters

- **particle_metadata** (SWIFTParticleTypeMetadata) – the metadata for the particle type
- **mask** (SWIFTMask) – the mask object for the dataset

class swiftsimio.reader.**SWIFTDataset**(*filename*, *mask=None*)

Bases: object

A collection object for:

- SWIFTUnits,
- SWIFTMetadata,
- SWIFTParticleDataset

This object, in essence, completely represents a SWIFT snapshot. You can access the different particles as follows:

- SWIFTDataset.gas.particle_ids
- SWIFTDataset.dark_matter.masses
- SWIFTDataset.gas.smoothing_lengths

These arrays all have units that are determined by the unit system in the file.

The unit system is available as SWIFTDataset.units and the metadata as SWIFTDataset.metadata.

def get_units(self):

Loads the units from the SWIFT snapshot.

def get_metadata(self):

Loads the metadata from the SWIFT snapshot.

def create_particle_datasets(self):

Creates particle datasets for whatever particle types and names are specified in metadata.particle_types.

get_units()

Loads the units from the SWIFT snapshot.

Ordinarily this happens automatically, but you can call this function again if you mess things up.

get_metadata()

Loads the metadata from the SWIFT snapshot.

Ordinarily this happens automatically, but you can call this function again if you mess things up.

create_particle_datasets()

Creates particle datasets for whatever particle types and names are specified in metadata.particle_types.

These can then be accessed using their underscore names, e.g. gas.

swiftsimio.statistics module

Reader for the statistics file.

class swiftsimio.statistics.**SWIFTStatisticsFile**(*filename: str*)

Bases: object

SWIFT statistics files (e.g. SFR.txt, energy.txt) reader.

header_names: List[str]

header_units: Dict[str, unyt_quantity]

header_snake_case_names: List[str]

raw_lines: List[str]

swiftsimio.subset_writer module

Contains functions for reading a subset of a SWIFT dataset and writing it to a new file.

swiftsimio.subset_writer.**get_swift_name**(*name: str*) → str

Returns the particle type name used in SWIFT

Parameters
name (*str*) – swiftsimio particle name (e.g. gas)

Returns
SWIFT particle type corresponding to *name* (e.g. PartType0)

Return type
str

swiftsimio.subset_writer.**get_dataset_mask**(*mask: SWIFTMask*, *dataset_name: str*, *suffix: str | None = None*) → ndarray

Return appropriate mask or mask size for given dataset

Parameters

- **mask** (*SWIFTMask*) – the mask used to define subset that is written to new snapshot
- **dataset_name** (*str*) – the name of the dataset we're interested in. This is the name from the hdf5 file (i.e. "PartType0", rather than "gas")
- **suffix** (*str, optional*) – specify a suffix string to append to dataset underscore name to return something other than the dataset mask. This is specifically used for returning the mask size by setting suffix="_size", which would return, for example mask.gas_size

Returns
mask for the appropriate dataset

Return type
np.ndarray

swiftsimio.subset_writer.**find_datasets**(*input_file: File*, *dataset_names=[]*, *path=None*, *recurse=False*) → List[str]

Recursively finds all the datasets in the snapshot and writes them to a list

Parameters

- **input_file** (*h5py.File*) – hdf5 file handle for snapshot
- **dataset_names** (*list of str, optional*) – names of datasets found in the snapshot
- **path** (*str, optional*) – the path to the current location in the snapshot
- **recurse** (*bool, optional*) – flag to indicate whether we're recursing or not

Returns
dataset_names – names of datasets in *path* in *input_file*

Return type
list of str

swiftsimio.subset_writer.**find_links**(*input_file: File*, *link_names: List | None = []*, *link_paths: List | None = []*, *path: str | None = None*)

Recursively finds all the links in the snapshot and writes them to a list

Parameters

- **input_file** (*h5py.File*) – hdf5 file handle for snapshot
- **link_names** (*list of str, optional*) – names of links found in the snapshot
- **link_paths** (*list of str, optional*) – paths where links found in the snapshot point to
- **path** (*str, optional*) – the path to the current location in the snapshot

Returns

link_names, link_paths – lists of the names and links of paths in *input_file*

Return type

list of str, list of str

swiftsimio.subset_writer.**update_metadata_counts**(*infile: File*, *outfile: File*, *mask:* SWIFTMask)

Recalculates the cell particle counts and offsets based on the particles present in the subset

Parameters

- **infile** (*h5py.File*) – File handle for input snapshot
- **outfile** (*h5py.File*) – File handle for output subset of snapshot
- **mask** (SWIFTMask) – the mask being used to define subset

swiftsimio.subset_writer.**write_metadata**(*infile: File*, *outfile: File*, *links_list: List[str]*, *mask:* SWIFTMask)

Copy over all the metadata from snapshot to output file

Parameters

- **infile** (*h5py.File*) – hdf5 file handle for input snapshot
- **outfile** (*h5py.File*) – hdf5 file handle for output snapshot
- **links_list** (*list of str*) – names of links found in the snapshot
- **mask** (SWIFTMask) – the mask being used to define subset

swiftsimio.subset_writer.**write_datasubset**(*infile: File*, *outfile: File*, *mask:* SWIFTMask, *dataset_names: List[str]*, *links_list: List[str]*)

Writes subset of all datasets contained in snapshot according to specified mask :param infile: hdf5 file handle for input snapshot :type infile: h5py.File :param outfile: hdf5 file handle for output snapshot :type outfile: h5py.File :param mask: the mask used to define subset that is written to new snapshot :type mask: SWIFTMask :param dataset_names: names of datasets found in the snapshot :type dataset_names: list of str :param links_list: names of links found in the snapshot :type links_list: list of str

swiftsimio.subset_writer.**connect_links**(*outfile: File*, *links_list: List[str]*, *paths_list: List[str]*)

Connects up the links to the appropriate path

Parameters

- **outfile** (*h5py.File*) – file containing the hdf5 subsnapshot
- **links_list** (*list of str*) – list of names of soft links
- **paths_list** (*list of str*) – list of paths specifying how to link each soft link

swiftsimio.subset_writer.**write_subset**(*output_file: str*, *mask:* SWIFTMask)

Writes subset of snapshot according to specified mask to new snapshot file

Parameters

- **input_file** (*str*) – path to input snapshot
- **output_file** (*str*) – path to output snapshot
- **mask** (SWIFTMask) – the mask used to define subset that is written to new snapshot

swiftsimio.swiftsnap module

swiftsnap allows you to check the metadata of a SWIFT snapshot easily from the command line. See the -h invocation for more details.

swiftsimio.swiftsnap.**decode**(*bytestring: bytes*) → str

swiftsimio.swiftsnap.**swiftsnap**()

swiftsimio.units module

Contains unit systems that may be useful to astronomers. In particular, it contains the cosmo_units which can be considered Gadget-oid default units, with

- Unit length = Mpc
- Unit velocity = km/s
- Unit mass = 10^10 Msun
- Unit temperature = K

Also contains unit conversion factors, to simplify units wherever possible.

swiftsimio.writer module

Contains functions and objects for creating SWIFT datasets.

Essentially all you want to do is use SWIFTWriterDataset and fill the attributes that are required for each particle type. More information is available in the README.

swiftsimio.writer.**get_dimensions**(*dimension: <module 'unyt.dimensions' from '/home/docs/checkouts/readthedocs.org/user_builds/swiftsimio/envs/stable/lib/python3.8/site-packages/unyt/dimensions.py'>*) → dict

Returns exponents corresponding to base dimensions for given unyt dimensions object

Parameters
dimension (*unyt.dimensions*) – dimension for which we're identifying the exponents

Returns
exp_array – array of exponents corresponding to each base dimension

Return type
np.ndarray

Examples

```
>>> get_dimensions(unyt.dimensions.velocity)
{
    "(mass)": 0,
    "(length)": 1,
    "(time)": -1,
    "(temperature)": 0,
    "(current)": 0,
}
```

swiftsimio.writer.**generate_getter**(*name: str*)

Generates a function that gets the unyt array for name.

Parameters
name (*str*) – name of data field

Returns
getter – function that returns unyt array for *name*

Return type
function

swiftsimio.writer.**generate_setter**(*name: str*, *dimensions*, *unit_system: UnitSystem | str*)

Generates a function that sets self._name to the value that is passed to it.

Parameters

- **name** (*str*) – string to set self._name to
- **dimensions** (*unyt.dimensions*) – physical dimension of self._name (for consistency check)
- **unit_system** (*unyt.UnitSystem or str*) – unit system of self._name

Returns
setter – function to set self._name

Return type
function

swiftsimio.writer.**generate_deleter**(*name: str*)

Generates a function that destroys self._name (sets it back to None).

Parameters
name (*str*) – name of object to be destroyed

Returns
deleter – function to delete self._name

Return type
function

swiftsimio.writer.**generate_dataset**(*unit_system: ~unyt.unit_systems.UnitSystem | str, particle_type: int, unit_fields_generate_units: ~typing.Callable[[...], dict] = <function generate_units>*)

Generates a SWIFTWriterParticleDataset _class_ that corresponds to the particle type given.

We _must_ do the following _outside_ of the class itself, as one can assign properties to a _class_ but not _within_ a class dynamically.

Here we loop through all of the possible properties in the metadata file. We then use the builtin property() function and some generators to create setters and getters for those properties. This will allow them to be accessed from outside by using SWIFTWriterParticleDataset.name, where the name is, for example, coordinates.

Parameters

- **unit_system** (*unyt.UnitSystem or str*) – unit system of the dataset
- **particle_type** (*int*) – the particle type of the dataset. Numbering convention is the same as SWIFT, with 0 corresponding to gas, etc. as usual.
- **unit_fields_generate_units** (*callable, optional*) – collection of properties in metadata file for which to create setters and getters

Returns
an empty dataset for the given particle type

Return type
SWIFTWriterParticleDataset

class swiftsimio.writer.**SWIFTWriterDataset**(*unit_system: ~unyt.unit_systems.UnitSystem | str, box_size: list | ~unyt.array.unyt_quantity, dimension=3, compress=True, extra_header: None | dict = None, unit_fields_generate_units: ~typing.Callable[[...], dict] = <function generate_units>, scale_factor: ~numpy.float32 = 1.0*)

Bases: object

The SWIFT writer dataset. This is used to store all particle arrays and do some extra processing before writing a HDF5 file containing:

- Fully consistent unit system
- All required arrays for SWIFT to start
- Required metadata (all automatic, apart from those required by __init__)

create_particle_datasets()

Creates particle dataset for each particle type in the metadata with associated units

write(*filename: str*)

Writes the information in the dataset to file.

Parameters
filename (*str*) – file to write to

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

ELEVEN

CITING SWIFTSIMIO

Please cite `swiftsimio` using the JOSS paper:

```
@article{Borrow2020,
  doi = {10.21105/joss.02430},
  url = {https://doi.org/10.21105/joss.02430},
  year = {2020},
  publisher = {The Open Journal},
  volume = {5},
  number = {52},
  pages = {2430},
  author = {Josh Borrow and Alexei Borrisov},
  title = {swiftsimio: A Python library for reading SWIFT data},
  journal = {Journal of Open Source Software}
}
```

If you use any of the subsampled projection backends, we ask that you cite our relevant SPHERIC article. Note that citing the arXiv version here is recommended as the ADS cannot track conference proceedings well.

```
@article{Borrow2021,
  title={Projecting SPH Particles in Adaptive Environments},
  author={Josh Borrow and Ashley J. Kelly},
  year={2021},
  eprint={2106.05281},
  archivePrefix={arXiv},
  primaryClass={astro-ph.GA}
}
```

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

G

H

I

K

L

M

N

P

R

S

T

U

V

W