SIFDM Data Interpretations Tutorial

SIFDM Data Interpretations Tutorial

Connor Painter, UT Austin Dept. of Astronomy
Last updated: 11/2/2021

This document is meant as a guide and tutorial to `sifdm-data-interpretations.py`, which is Python code that helps analyze and visualize outputs from self-interacting dark matter simulations by Philip Mocz. You can find the accompanying tutorial images saved in a nearby folder located on Github.

Getting Started

1. *Locating output folders*

To be able to run `sifdm-data-interpretations.py`, you'll need to have outputs from a simulation from Philip's MATLAB code accessible in a folder (usually looks something like `outputs/f4L20T4n400r100`). Upon opening the Python document, you need to paste the full path to each output folder as a string, then group the paths as a global list variable called `folders`. The code comes formatted ready for you to do this: you can just paste the path into d1, d2, etc. If you have two output folders, it might look something like:

```
## f15 = Inf; L = 20; T = 4; Nout = 400; N (res) = 100
d1 = r"/Users/me/Documents/sifdm-matlab-main/output/fInfL20T4n400r100"

## f15 = 1; L = 20; T = 4; Nout = 400; N (res) = 100
d2 = r"/Users/me/Documents/sifdm-matlab-main/output/f1L20T4n400r100"

folders = [d1, d2]
```

Compiling the code automatically checks whether the folders within the list exist and will print an error message if any folder is not found.

2. *Preferences for saving images*

By default, compiling the code will create subfolders within the current working directory to organize saved images by the properties of their underlying simulation. If you do not want to create subfolders, toggle `createImgSubfolders`. If you want subfolders created in a different directory (other than the default), change `saveToParent` to a full path of the directory of your choice.

Documentation

class `Sim(snapdir)`

Creates objects that represent entire simulations. Contains attributes and functions that pertain to or require information from every snapshot within an output folder.

Parameters

- `snapdir` : string
  - Full path to an output directory containing snapshots from a simulation.

Attributes

- `snapdir` : str
  - Output directory used to create the object.
- `dir` : str
  - Basename of `snapdir`.
- `snaps` : List of Snap objects
  - All snapshots from the simulation in a list of Snap objects.
- `t` : list of floats
  - Ordered list of timestamps from each snap.
- `Nout` : int
  - Number of snapshots in the output directory.
- `m22` : float
  - Mass of the dark matter particle used in the simulation [$10^{-22}\ eV$].
- `m` : float
  - Mass of the dark matter particle used in the simulation [$M_{\odot}$].
- `f15` : float
  - Strong-CP energy decay constant used in the simulation [$10^{15}\ GeV$].
- `f` : float
  - Strong-CP energy decay constant used in the simulation [$M_{\odot}(km/s)^2$].
- `a_s` : float
  - s-scattering length used in the simulation.
- `Lbox` : float
  - Simulation box side length [kpc].
- `N` : int
  - Linear resolution of the simulation; total number of data points in N^3.
- `dx` : float
  - Distance between data points [kpc]. Equivalent to Lbox/N.
- `critical_M_sol` : float
  - Soliton mass at which the soliton is predicted to collapse.

- `critical_rho0` : float
    - Central soliton density at which the soliton is predicted to collapse.
- `critical_r_c` : float
    - Soliton core radius at which the soliton is predicted to collapse.
- `critical_beta` : float
    - Soliton stability constant value at which the soliton is predicted to collapse

Examples

Consider the output folders provided in the first section. Create a `Sim` object by executing

```
>>    sim = Sim(d1)
```

This may take a moment, as it is loading N^3 complex values of the wavefunction Ψ. Print some attributes of the simulation object by executing

```
>>    sim.f15
>>    inf

>>    sim.Lbox
>>    20.0

>>    np.shape(sim.snaps[142].psi)
>>    (100, 100, 100)
```

Attributed Functions

`get``(q, snaps=None, axis=None, project=False, i=None, iSlice=None, log10=False)`

Retrieves some quantity q (along index keyword arguments) from any snapshots. Makes use of the get(...) function from the Snap class, which stores data in attributes for easy future retrieval.

Parameters

- `q` : str
    - Name of quantity to retrieve.
- `snaps` : (*Optional*) None or list of ints
    - Snap numbers from which to retrieve quantity.
- `axis` : (*Optional*) None, int, or str
    - Axis along which to project or cross-section quantity, if applicable
- `project` : (*Optional*) bool

    - Toggles whether to project the retrieved quantity.
- `i` : (*Optional*) None, int, or tuple
    - Index of the quantity `q`.
- `iSlice` : (*Optional*) None, int, or `'max'`
    - Index at which to retrieve a cross-section of the quantity.
- `log10` : (*Optional*) bool
    - Toggles whether the log base 10 of the quantity is retrieved.

Returns

- `data` : list
    - List of all requested data.

Examples

Get the x component of the Madelung velocity from every snapshot:

```
>>    v = sim.get('v', i=0)
```

While this might take a while on the first execution, every quantity that was found while computing the Madelung velocity is now stored in each snap object for quick future retrieval. For instance,

```
>>    phase = sim.get('phase')
```

should be lightning fast. Notice that to access these quantities, you need to request them from the list of snaps within the Sim object; they are not directly attributed to the Sim object. In other words,

```
>>    sim.v
>>    AttributeError: 'Sim' object has no attribute 'v'

>>    np.shape(sim.snaps[0].v)
>>    (3, 100, 100, 100)
```

`evolutionPlot``(q, i=None, log10=False, ax=None, **kwargs)`

Plots the evolution of any quantities that are scalar-valued at every snap.

Parameters

- `q` : str
    - Name of quantity(s) to plot.
- `i` : (*Optional*) None, int, or tuple
    - Index of the quantity `q` to be plotted.

- `log10` : (*Optional*) bool
  - Toggles whether the log base 10 of the quantity is plotted.
- `ax` : (*Optional*) `matplotlib.axes.Axes`
  - Axes on which to place the plot.
- `**kwargs`
  - `figsize` : 2-tuple of ints
  - `dpi` : int
  - `save` : bool
  - `filename` : str
  - `c` : str
  - `iterproduct` : bool
  - `snaps` : list of ints
  - `legendkws` : dict
  - Other keyword arguments passed to `matplotlib.axes.Axes.plot`

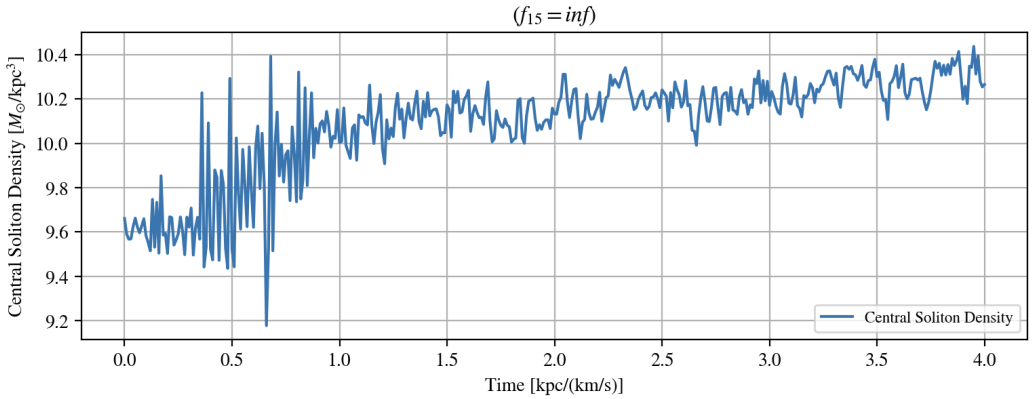
Returns

- `t` : list of floats
  - Time values at which the quantity was plotted.
- `data` : list of floats
  - Ordered list of the requested quantity at each point in time.

Examples

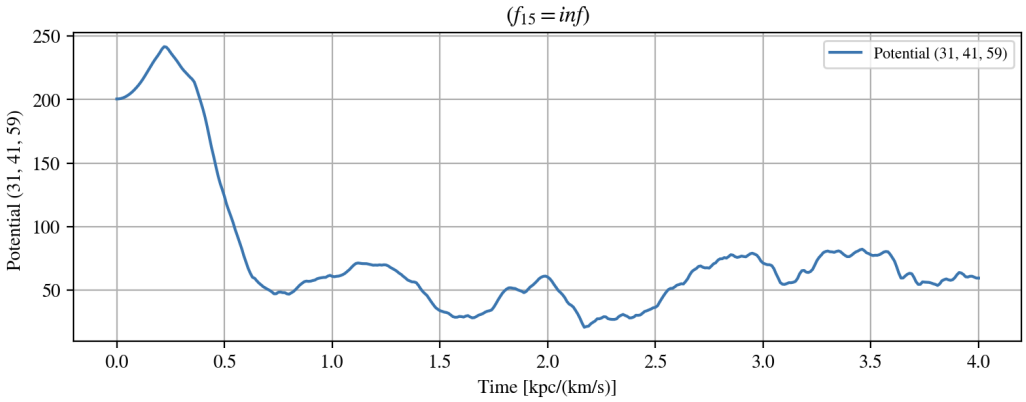
Plot the evolution of the density ρ_0 at the center of the densest soliton.

```
>>    sim.evolutionPlot('rho0', log10=True,
filename="tutorial-rho_0-evolution-plot")
```



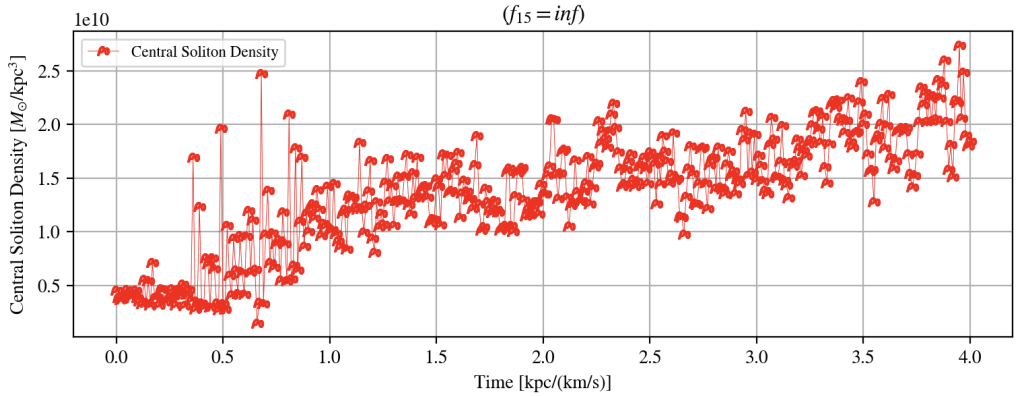
Plot the evolution of the potential at a specific location in space:

```
>>    sim.evolutionPlot('V', i = (31,41,59),
filename="tutorial-V-31-41-59-evolution-plot")
```



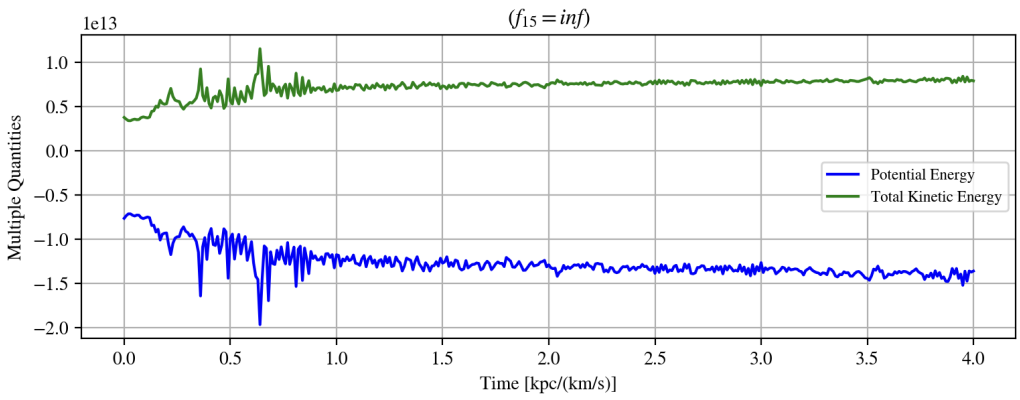
Make use of matplotlib's keyword arguments:

```
>>    sim.evolutionPlot('rho0', c='r', marker=r'$\rho_0$',
linewidth=0.3,
filename='tutorial-rho_0-evolution-plot-silly')
```



Make use of multi-quantity plotting support to compare energy contributions:

```
>>    sim.evolutionPlot(['W','KQ'], c=['b','g'],
filename="tutorial-energy-evolution-plot")
```



`evolutionMovie``(q, axis=1, project=False, i=None, iSlice=None, log10=None, **kwargs)`

Animates the evolution of any quantities defined throughout the box.

Parameters

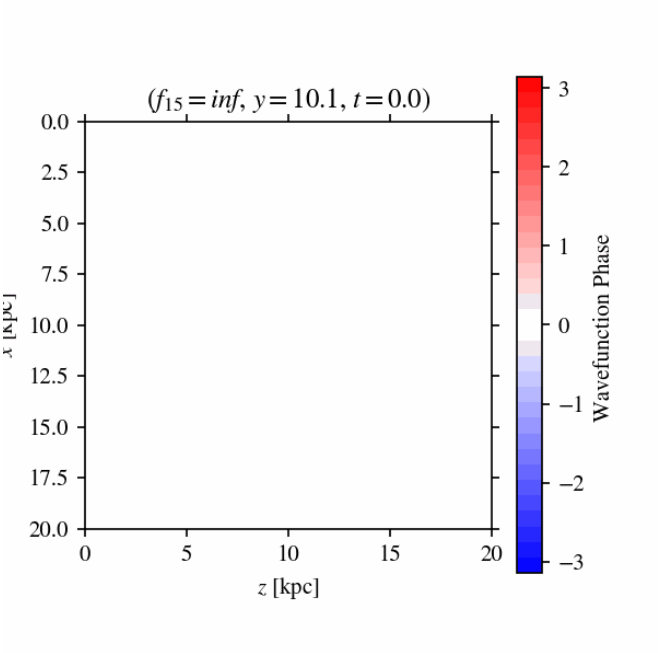
- `q` : str
  - Name of quantity(s) to animate.
- `axis` : (*Optional*) int or `'x'`, `'y'`, or `'z'`
  - Axis along which to retrieve cross-sections or projections.
- `project` : (*Optional*) bool
  - Toggles whether to animate projection plots or cross-sections.
- `i` : (*Optional*) None, int, or tuple
  - Index of the quantity `q` to be animated.
- `iSlice` : (*Optional*) int
  - Index at which to slice (if project is False). Defaults to slice at the index where the `q` is the greatest.
- `log10` : (*Optional*) bool
  - Toggles whether the log base 10 of the quantity is animated.
- `**kwargs`
  - `dpi` : int
  - `wspace` : float
  - `save` : bool
  - `filename` : str
  - `fps` : int
  - `iterproduct` : bool
  - `climfactors` : list of 2 floats
  - `clims` : list of 2 floats
  - `cmap` : str
  - `snaps` : list of ints

- Other keyword arguments passed to `Snap.plot2d(...)`

Examples

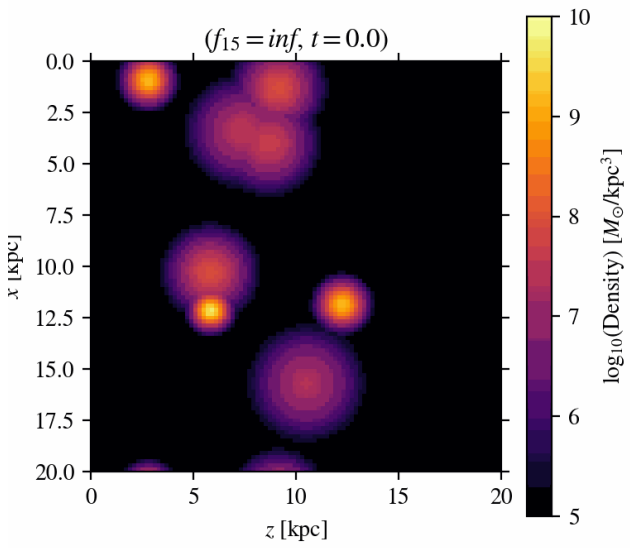
Animate the evolution of the wavefunction phase at a specific cross-section:

```
>>    sim.evolutionMovie('phase', iSlice=50,
filename='tutorial-phase-evolution-movie')
```



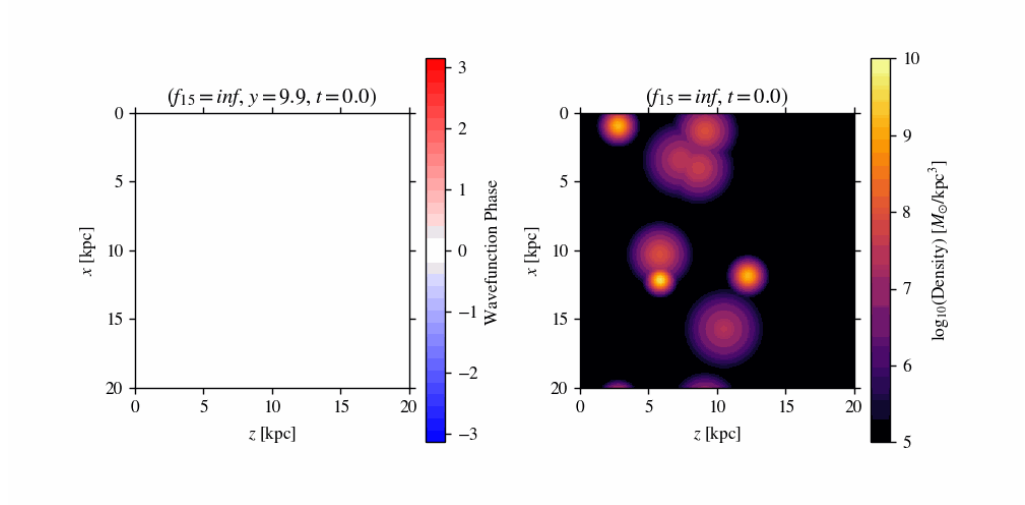
The colormap defaults to a preset based on the animated quantity.
Animate the evolution of the projected density of FDM with color limits adjusted for appropriate saturation:

```
>>    sim.evolutionMovie('rho', project=True, log10=True,
clims=[5,10], filename="tutorial-density-evolution-movie")
```



Combine the previous two animations in a single figure:

```
>>    sim.evolutionMovie(['phase', 'rho'],
project=[False,True], log10=[False,True],
clims=[[None,None], [5,10]],
filename="tutorial-multi-evolution-movie", dpi=125)
```



densityProfileMovie(rmin=None, rmax=None, shells=20, normalize=True, neighbors=1, rands=1e5, fit=False, **kwargs)

Animates the evolution of the soliton density profile.

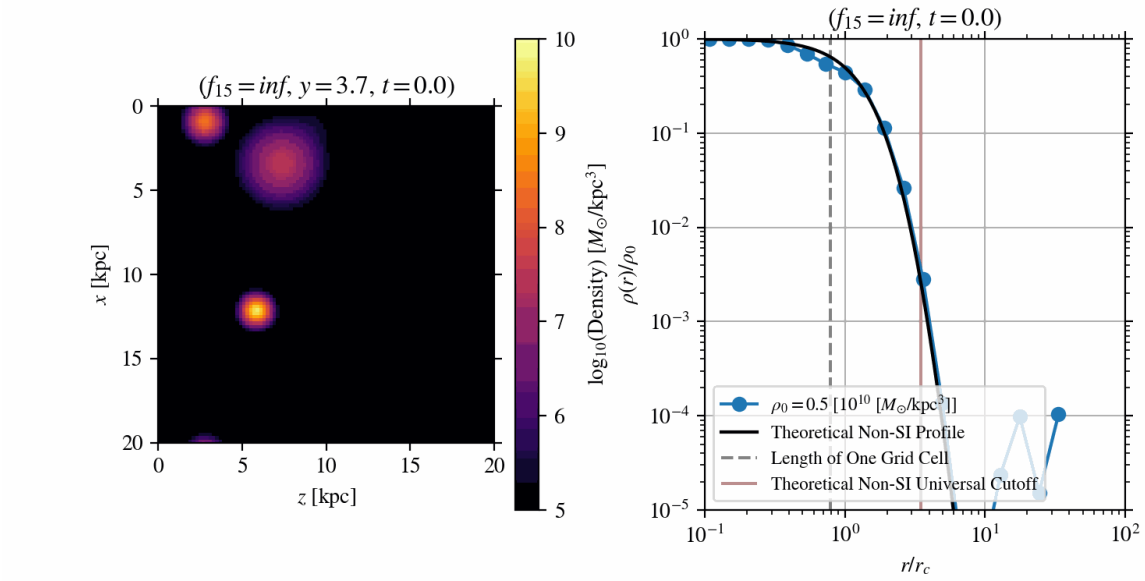
Parameters

- `rmin` : (*Optional*) float
  - Minimum radius at which to compute the average density.
- `rmax` : (*Optional*) float
  - Maximum radius at which to compute the average density.
- `shells` : (*Optional*) int
  - Number of shells around the soliton center in which to sample density.
- `normalize` : (*Optional*) bool
  - Toggles whether to normalize the density measurements by the density at the center of the soliton and the radii by core radius.
- `neighbors` : (*Optional*) int
  - Number of neighboring grid cells to consider in soliton center-of-mass calculation.
- `rands` : (*Optional*) int
  - Number of random coordinates to draw in each shell.
- `fit` : (*Optional*) bool
  - Toggles whether to include the theoretical prediction for a non-self-interacting soliton density profile in the animation.
- `**kwargs`
  - `figsize` : list of two ints
  - `dpi` : int
  - `clims` : list of two floats
  - `climfactors` : list of two floats
  - `axis` : int or str
  - `filename` : str
  - `fps` : int
  - `snaps` : list of ints
  - Other keyword arguments passed to `Snap.slicePlot(...)` and `Snap.densityProfile(...)`.

Examples

Create a density profile animation with the measured data points as well as a curve representing the theoretical profile for a non-self-interacting halo:

```
>>    sim.densityProfileMovie(fit=True,
filename='tutorial-density-profile-movie')
```



`class` **`Snap`**`(snapdir, snapnum, loadall=False)`

Creates objects that represent individual snapshots within simulations. Contains attributes and functions that pertain to quantities defined within the simulation box at a particular moment in time.

Parameters

- `snapdir` : str
  - Full path to an output directory containing snapshots from a simulation.
- `snapnum` : int
  - The number assigned to the snapshot (in its filename).
- `loadall` : (*Optional*) bool
  - Toggles whether to calculate every known quantity associated with the snapshot immediately upon instantiation. If False, only imports basic quantities and does no computations until asked.

Attributes

- `snapdir` : str
  - Output directory used to create the object.
- `num` : int
  - Snap number used to create the object.
- `filename` : str
  - Filename as outputted by Philip's code.
- `path` : str
  - Full path to snapshot file.
- `dir` : str
  - Basename of `snapdir`.

- `psi` : NumPy array of shape (N, N, N)
    - Wavefunction of dark matter.
- `t` : float
    - Time snapshot was taken.
- `m22` : float
    - Mass of the dark matter particle used in the simulation [$10^{-22}\ eV$]
- `m` : float
    - Mass of the dark matter particle used in the simulation [$M_{\odot}$]
- `f15` : float
    - Strong-CP energy decay constant used in the simulation [$10^{15}\ GeV$]
- `f` : float
    - Strong-CP energy decay constant used in the simulation [$M_{\odot}(km/s)^2$]
- `a_s` : float
    - s-scattering length used in the simulation.
- `Lbox` : float
    - Simulation box side length [kpc]
- `N` : int
    - Linear resolution of the simulation; total number of data points in N^3.
- `dx` : float
    - Distance between data points [kpc]. Equivalent to Lbox/N.
- `critical_M_sol` : float
    - Soliton mass at which the soliton is predicted to collapse.
- `critical_rho0` : float
    - Central soliton density at which the soliton is predicted to collapse.
- `critical_r_c` : float
    - Soliton core radius at which the soliton is predicted to collapse.
- `critical_beta` : float
    - Soliton stability constant value at which the soliton is predicted to collapse
- `all_loaded` : bool
    - Value of loadall used to create the object.

(Attributes created when loadall is True or with later analysis)

- `phase` : NumPy array of shape (N, N, N)
    - Wavefunction phase.
- `rho` : NumPy array of shape (N, N, N)
    - Density of dark matter (magnitude squared of the wavefunction)
- `rhobar` : float
    - Mean density of dark matter in the box.
- `rho0` : float
    - Maximum density of dark matter in the box (ideally at a soliton center).

- `i0` : 3-tuple
    - Index of rho0.
- `beta` : float
    - Soliton stability constant.
- `r_c` : float
    - Soliton core radius as calculated from rho0 and m22.
- `M_sol` : float
    - Mass of the soliton.
- `critical_f` : float
    - Self-interaction strength at which a soliton of mass `M_sol` (as computed in parent Snap) collapses.
- `tailindex` : float
    - Index of power law fit to density profile tail.
- `V` : NumPy array of shape (N, N, N)
    - Gravitational potential.
- `v` : NumPy array of shape $(3, N, N, N)$
    - Madelung velocity components.
- `v2` : NumPy array of shape (N, N, N)
    - Magnitude squared of Madelung velocity. Seems to be a great way to visualize vortex rings and turbulence.
- `M` : float
    - Total mass of the particles in the box.
- `W` : float
    - Total potential energy.
- `Kv` : float
    - Classical kinetic energy.
- `Krho` : float
    - Quantum gradient energy.
- `KQ` : float
    - Total kinetic energy.
- `E` : float
    - Total energy, kinetic plus potential.
- `L` : list of length 3
    - Angular momentum components.
- `L2` : float
    - Magnitude squared of angular momentum.

Examples

Suppose you wanted to examine the 315th snapshot in d2, as defined in the first section, but you don't need to load every quantity associated with it. Define the Snap object as

```
>>    snap = Snap(d2, 315)
```

leaving loadall to default to False. Test that this loaded the expected things:

```
>>    snap.num
>>    315

>>    np.shape(snap.psi)
>>    (100, 100, 100)

>>    snap.KQ
>>    AttributeError: 'Snap' object has no attribute 'KQ'
```

To retrieve the value of snap.KQ (and other quantities not yet loaded), you'll need to use the attributed get(...) function (see below).

Attributed Functions

> **get**(q, axis=None, project=False, i=None, iSlice=None, log10=False)
>
> > Retrieves some quantity `q` (along index keyword arguments) from the snapshot. Checks to see if the quantity has already been computed and stored: if so, it does no computations; otherwise, it computes the quantity (along with anything else that is necessary) and stores it for future retrieval.
>
> **Parameters**
>
> - `q` : str
>   - Name of quantity(s) to retrieve.
> - `axis` : (*Optional*) None, int, or str
>   - Axis along which to project or cross-section quantity, if applicable
> - `project` : (*Optional*) bool
>   - Toggles whether to project the retrieved quantity.
> - `i` : (*Optional*) None, int, or tuple
>   - Index of the quantity `q`.
> - `iSlice` : (*Optional*) None, int, or `'max'`
>   - Index at which to retrieve a cross-section of the quantity.
> - `log10` : (*Optional*) bool
>   - Toggles whether the log base 10 of the quantity is retrieved.
>
> **Returns**
>
> - `data` : list

- List of all requested data.

Examples

Following from the example above, use this function to retrieve the total energy due to the quantum pressure term:

```
>>    snap.get('KQ', log10=True)
>>    13.147864633639639
```

Get a specific element from a multi-dimensional array:

```
>>    snap.get('v', i=(2, 71, 82, 81))
>>    310.0823468314939
```

Use Python keyword slice in your tuples to get arbitrary slices:

```
>>    np.shape(snap.get('v', i=tuple((2, slice(71,82),
slice(8,18), 28))))
>>    (11, 10)
```

Get multiple quantities using list comprehension:

```
>>    [snap.get(q) for q in ['rhobar', 'rho0']]
>>    [5.65946975436462, 11.146991894825794]
```

plot2d`(q, axis=1, project=False, i=None, iSlice=None, log10=False, ax=None, **kwargs)`

Plots given quantities defined throughout the simulation box. Generalizes `Snap.slicePlot(...)` and `Snap.projectionPlot(...)`, feel free to use any of them.

Parameters

- `q` : str
  - Name of quantity(s) to plot.
- `axis` : (*Optional*) None, int, or str
  - Axis along which to retrieve cross-sections or projections.
- `project` : (*Optional*) bool
  - Toggles whether to plot a projection or cross-section.
- `i` : (*Optional*) None, int, or tuple
  - Index of the quantity `q` to be plotted.

- `iSlice` : (*Optional*) None, int, or `'max'`
    - Index at which to slice (if project is False). Defaults to slice at index where the `q` is the greatest.
- `log10` : (*Optional*) bool
    - Toggles whether the log base 10 of the quantity is plotted.
- `ax` : (*Optional*) `matplotlib.axes.Axes`
    - Axes on which to place the plot.
- `**kwargs`
    - `figsize` : 2-tuple
    - `dpi` : int
    - `zoom` : list of 4 ints
    - `climfactors` : list of 2 floats
    - `clims` : list of 2 floats
    - `filename` : str
    - `fps` : int
    - `wspace` : float
    - `iterproduct` : bool
    - `cmap` : str
    - `colorbar` : bool
    - Other keyword arguments passed to `matplotlib.axes.Axes.imshow(...)`.

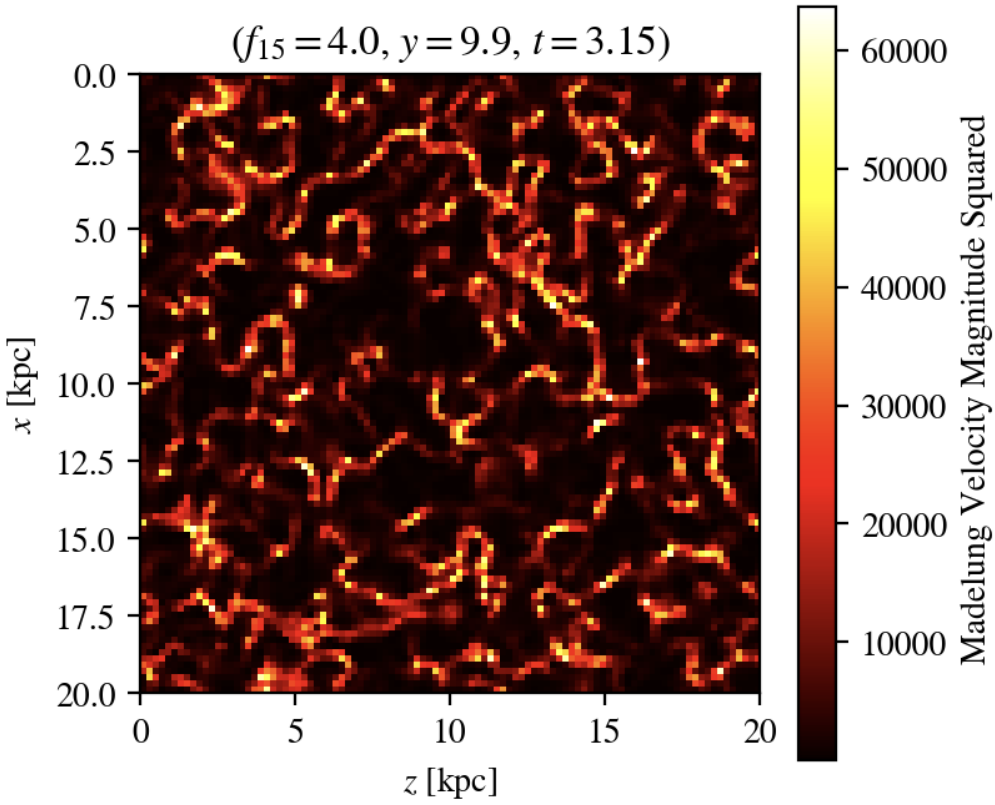
Returns

- `data` : 2-dimensional NumPy array
    - The matrix of plotted data (either a projection or cross-section) related to `q`.

Examples

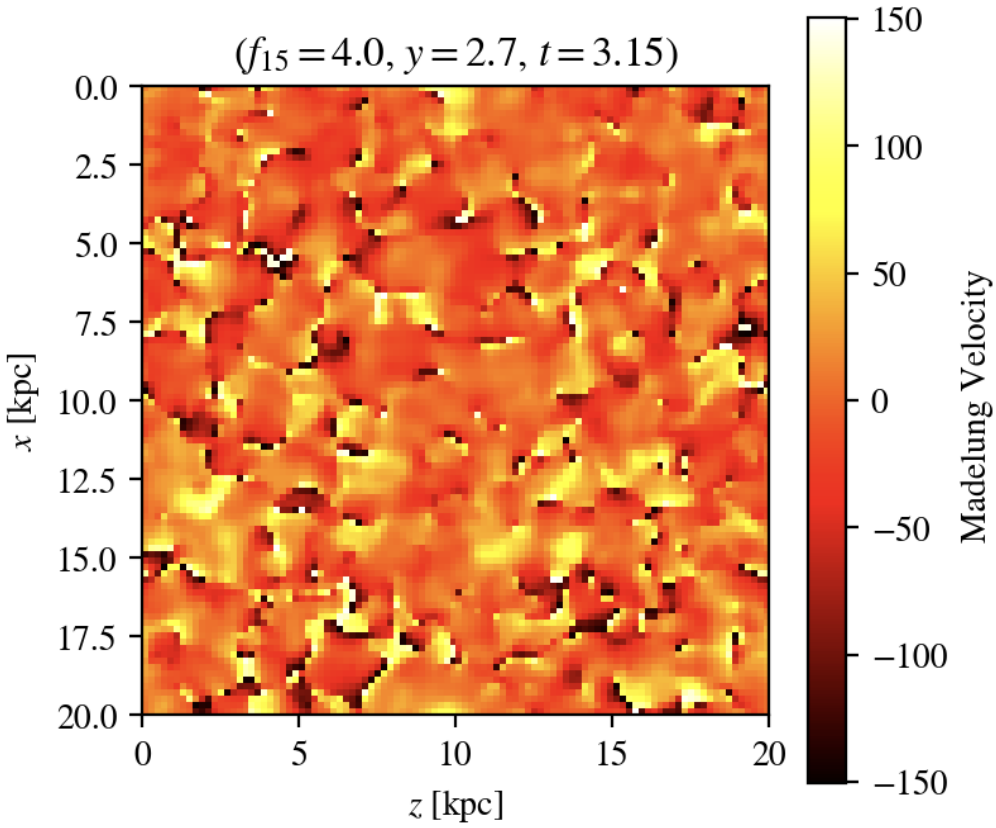
Quickly plot the magnitude squared of the Madelung velocity:

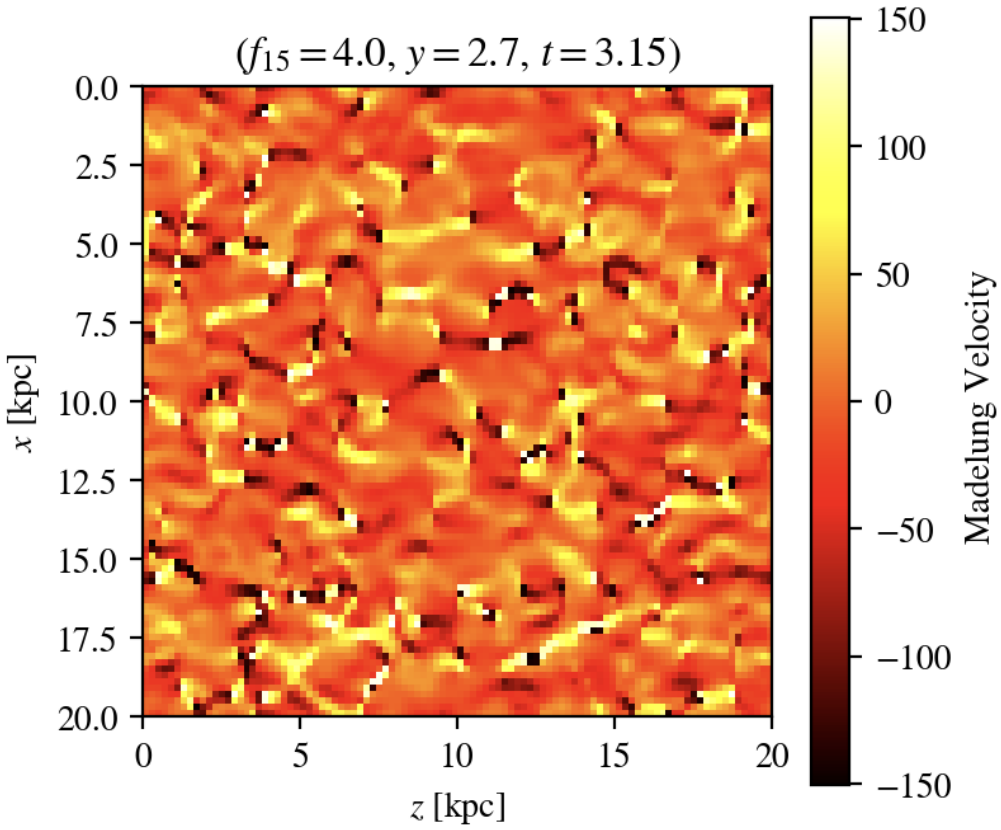
```
>>    snap.plot2d('v2', save=True, filename='tutorial-v2')
```

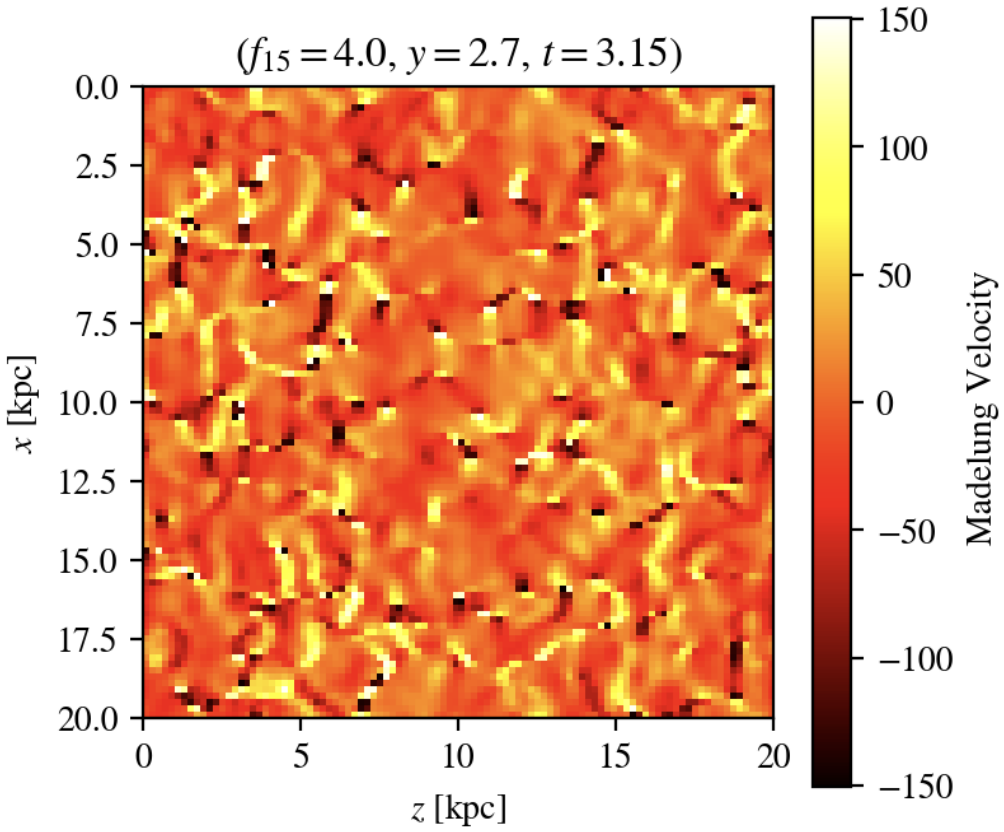


Plot cross-sections of all three components of the Madelung velocity (at a particular slice) separately using list comprehension:

```
>>    [snap.plot2d('v', i=i, iSlice=13, save=True,
filename=f"tutorial-v_{'xyz'[i]}") for i in range(3)]
```

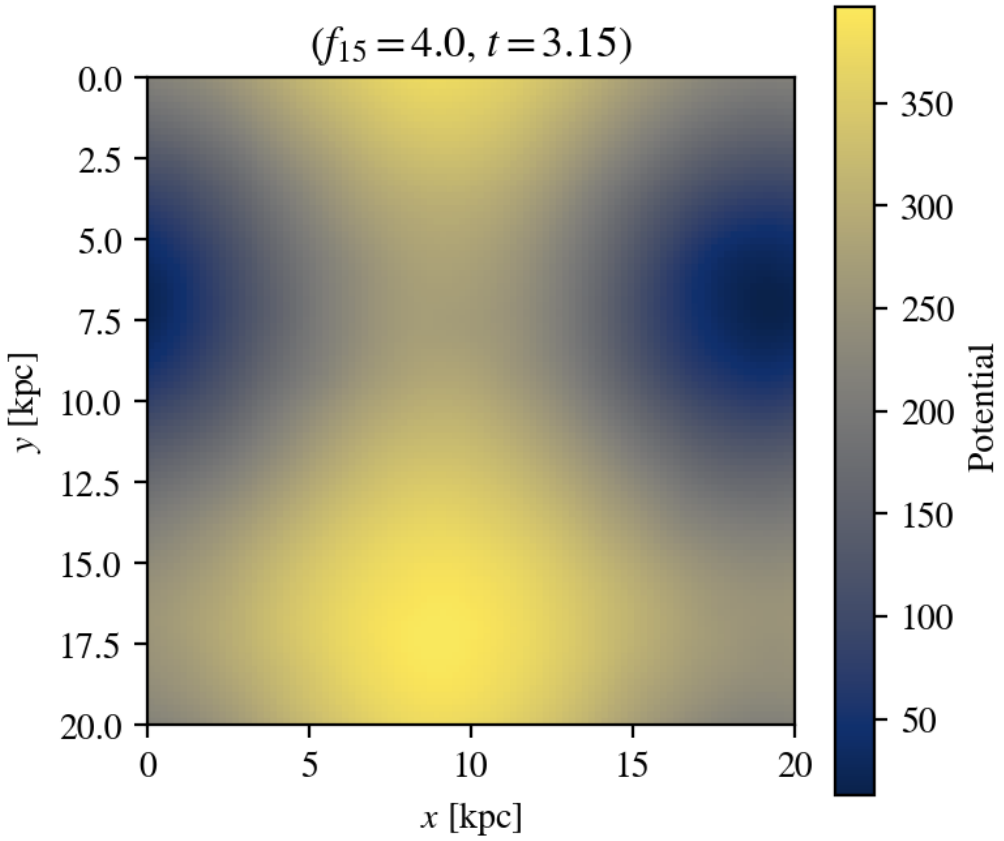

($f_{15} = 4.0$, $y = 2.7$, $t = 3.15$)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x [kpc]
0
5
10
15
20
z [kpc]
150
100
50
0
−50
−100
−150
Madelung Velocity


($f_{15} = 4.0$, $y = 2.7$, $t = 3.15$)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x [kpc]
0
5
10
15
20
z [kpc]
150
100
50
0
−50
−100
−150
Madelung Velocity



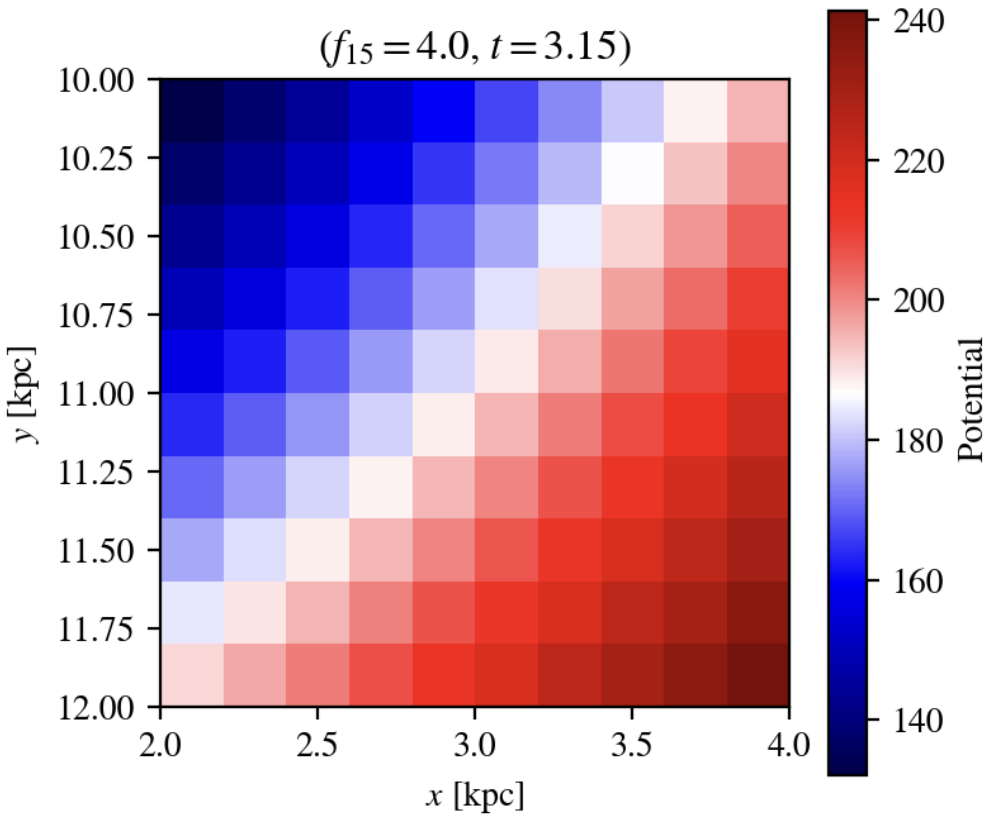
Plot the potential projected onto the z-axis:

```
>>    snap.plot2d('V', 'z', project=True,
filename='tutorial-potential')
```



Zoom in to examine the potential well and change the colormap:

```
>>   snap.plot2d('V', 'z', project=True,
zoom=[50,60,10,20], cmap='seismic',
filename='tutorial-potential')
```



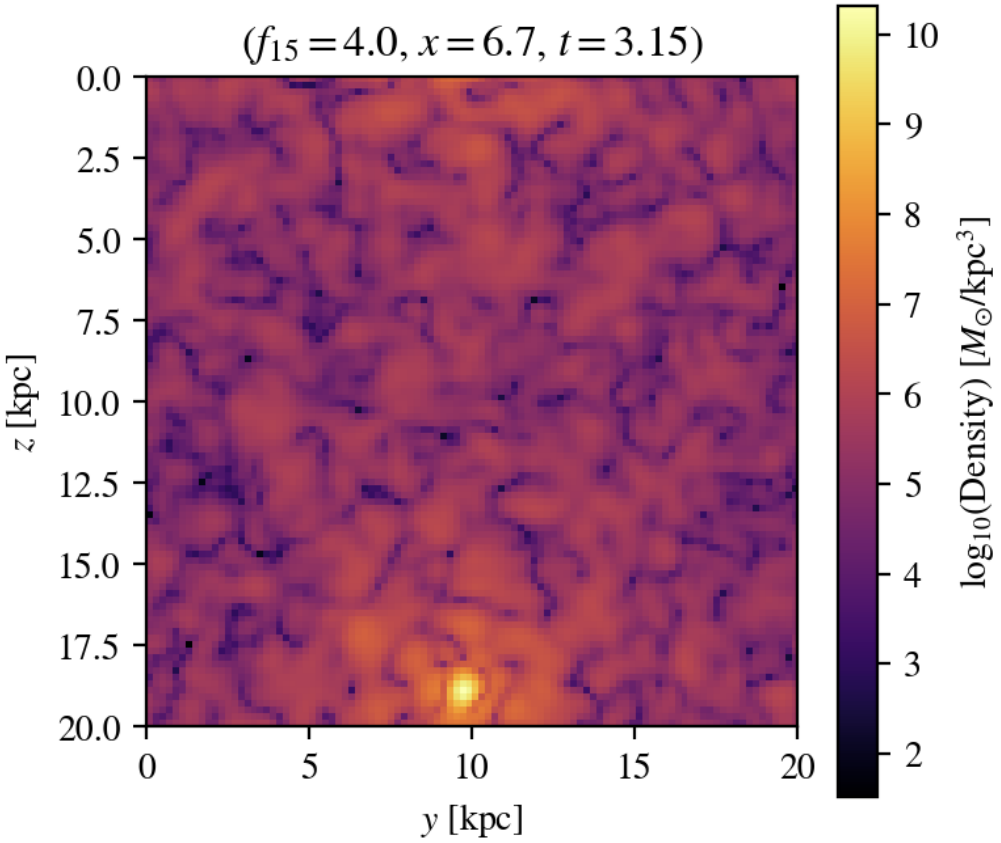
slicePlot(q, axis=1, i=None, iSlice=None, ax=None, **kwargs)

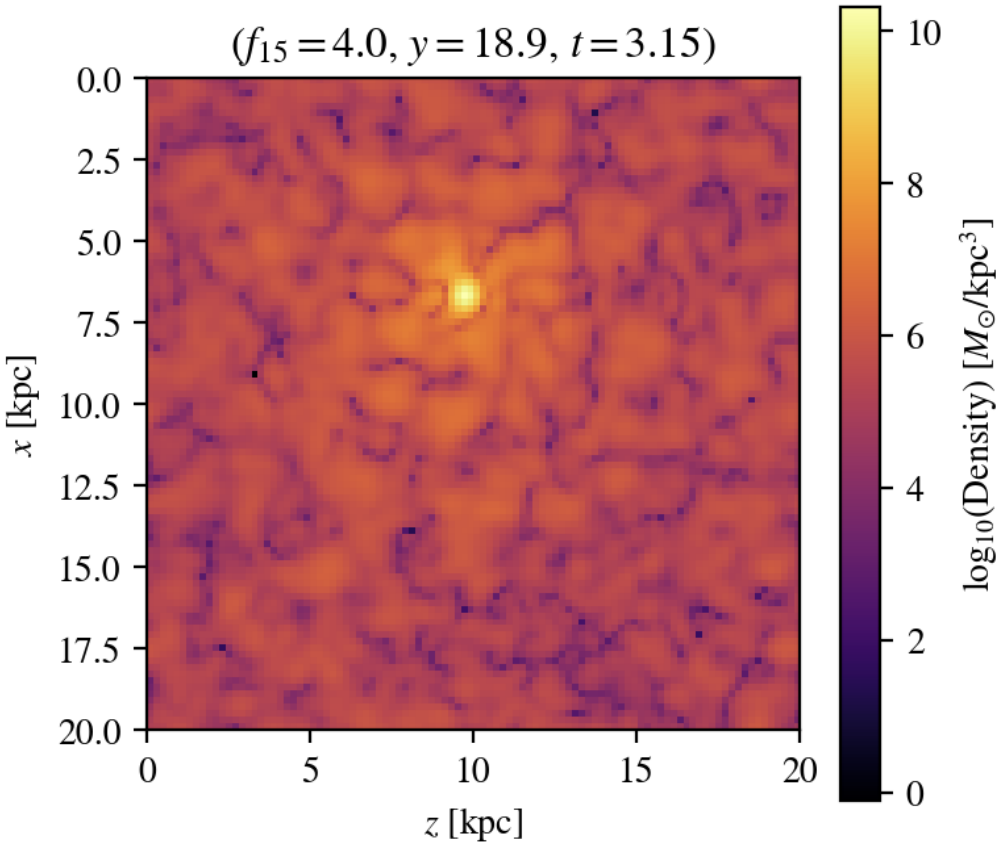
Plots a cross-section of a quantity defined throughout the simulation box. Parameters and output are the same as in Snap.plot2d(...).

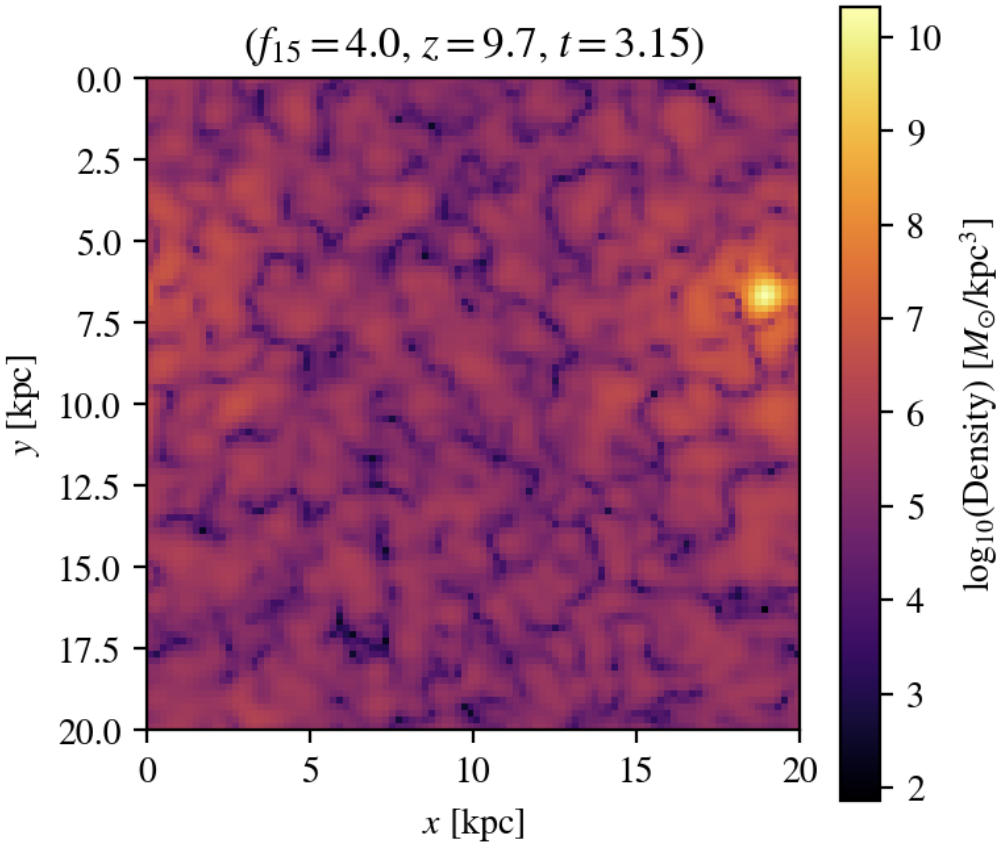
Examples

Get cross-sections of the soliton along all three axes using list comprehension (two equivalent loops):

```
>>    [snap.slicePlot('rho', axis, iSlice='max', log10=True,
filename=f"tutorial-rho_{'xyz'[axis]}") for axis in
range(3)]
>>    [snap.slicePlot('rho', axis, iSlice='max', log10=True,
filename=f"tutorial-rho_{axis}") for axis in 'xyz']
```


($f_{15} = 4.0$, $x = 6.7$, $t = 3.15$)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
z [kpc]
0
5
10
15
20
y [kpc]
10
9
8
7
6
5
4
3
2
log10(Density) [M⊙/kpc3]


($f_{15} = 4.0$, $y = 18.9$, $t = 3.15$)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x [kpc]
0
5
10
15
20
z [kpc]
10
8
6
4
2
0
$\log_{10}$(Density) [$M_\odot$/kpc^3]



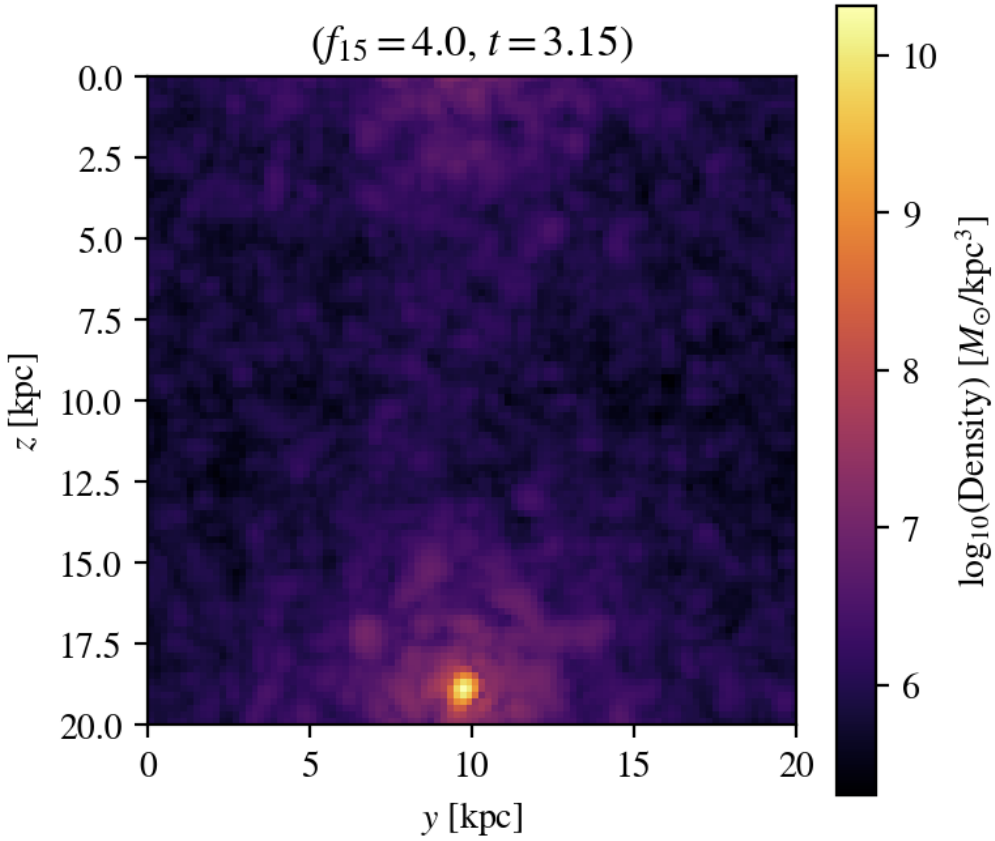
projectionPlot(q, axis=1, ax=None, **kwargs)

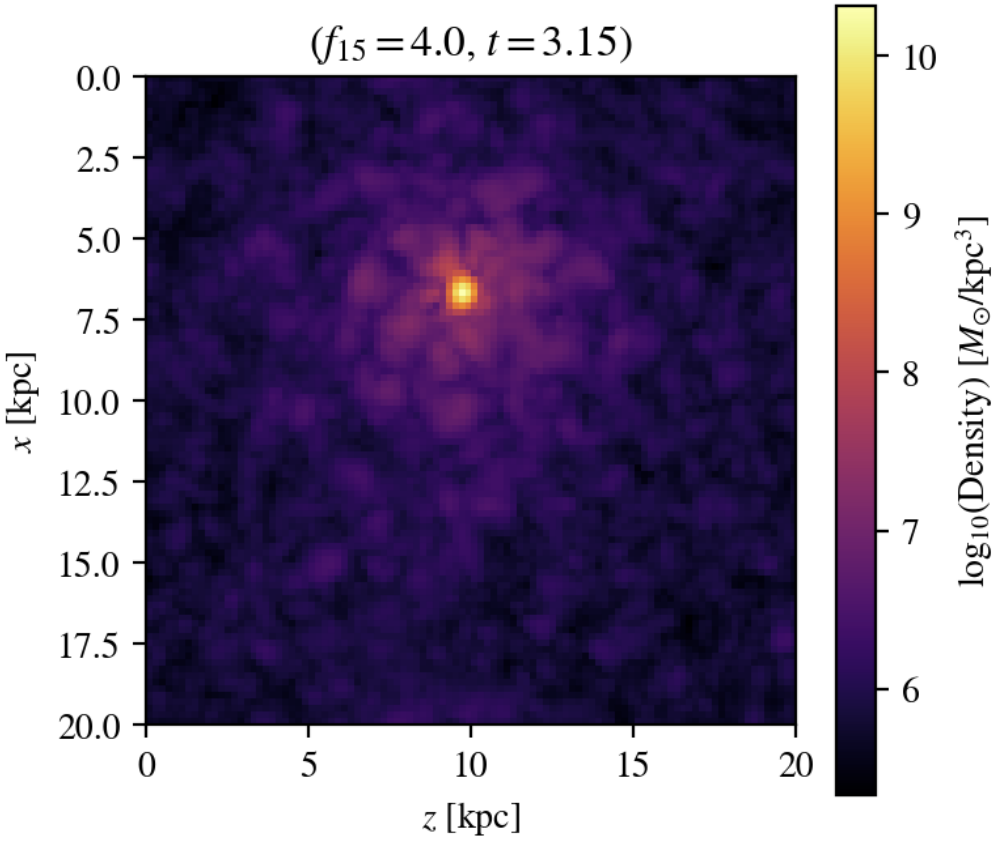
Plots a projection of a quantity defined throughout the simulation box. Parameters and output are the same as in Snap.plot2d(...).

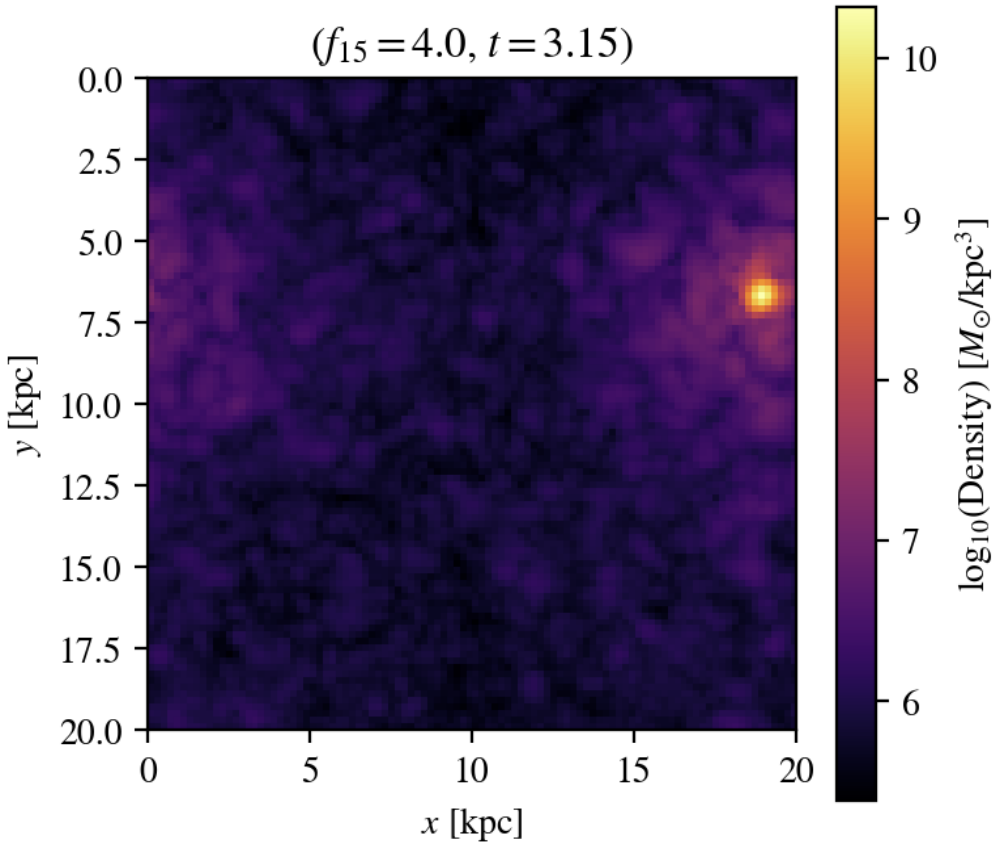
Examples

Get projections of the density along all three axes using list comprehension:

```
>>    [snap.projectionPlot('rho', axis, log10=True,
filename=f'tutorial-rho_{axis}-projected') for axis in
'xyz']
```


($f_{15} = 4.0, t = 3.15$)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
z [kpc]
0
5
10
15
20
y [kpc]
10
9
8
7
6
$\log_{10}$(Density) [$M_\odot$/kpc^3]


($f_{15} = 4.0, t = 3.15$)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x [kpc]
0
5
10
15
20
z [kpc]
10
9
8
7
6
$\log_{10}$(Density) [$M_\odot$/kpc^3]



scan3d(q, axis=1, i=None, log10=False **kwargs)

Animates cross-sections of a quantity through a simulation box, as if "scanning" the domain. Parameters are the same as in `Snap.plot2d(...)`.

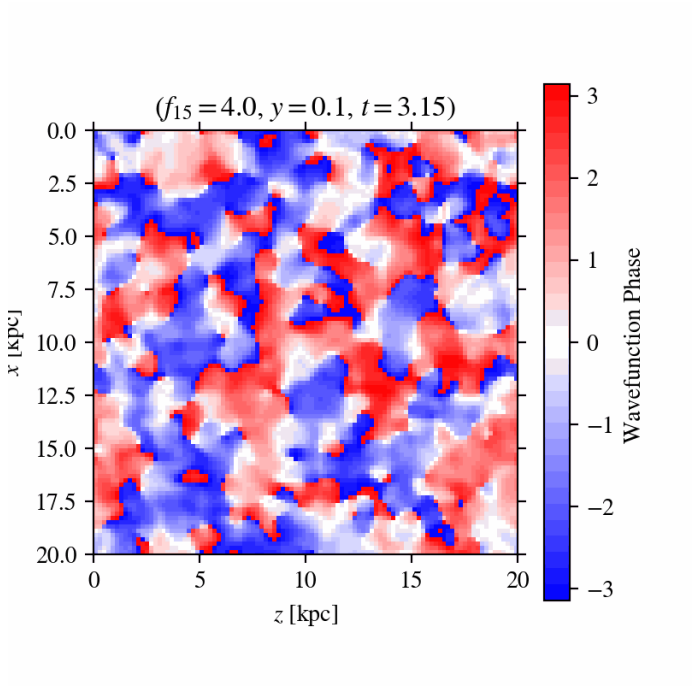
Returns

- `data` : 3-dimensional NumPy array
  - All animated data, equivalent to `snap.get(q, i=i, log10=log10)`.

Examples

Scan the wavefunction phase along the default axis:

```
>>    snap.scan3d('phase', filename='tutorial-phase-scan')
```



densityProfile(rmin=None, rmax=None, shells=20, normalize=True, neighbors=1, rands=1e5, fit=False, plot=True, ax=None, **kwargs)

Computes the soliton density profile. Finds the soliton center by identifying the densest pixel, then performing a center of mass calculation of that pixel and its neighbors (sub-pixel centering). Densities are computed within log-spaced shells using random coordinate sampling (sub-pixel density averaging).

Parameters

- `rmin` : (*Optional*) float
  - Minimum radius at which to compute the average density.
- `rmax` : (*Optional*) float
  - Maximum radius at which to compute the average density.
- `shells` : (*Optional*) int
  - Number of shells around the soliton center in which to sample density.
- `normalize` : (*Optional*) bool
  - Toggles whether to normalize the density measurements by the density at the center of the soliton and the radii by core radius.
- `neighbors` : (*Optional*) int
  - Number of neighboring grid cells to consider in soliton center-of-mass calculation.
- `rands` : (*Optional*) int
  - Number of random coordinates to draw in each shell.

- `fit` : (*Optional*) bool
    - Toggles whether to include the theoretical prediction for a non-self-interacting soliton density profile in the animation.
- `plot` : (*Optional*) bool
    - Toggles whether to display a plot in addition to returning profile data.
- `ax` : `matplotlib.axes.Axes`
    - Axes on which to plot the profile.
- `**kwargs`
    - `figsize` : list of two ints
    - `dpi` : int
    - `lims` : list of four floats
    - `filename` : str
    - `save` : bool
    - Other keyword arguments passed to `matplotlib.axes.Axes.loglog(...)`

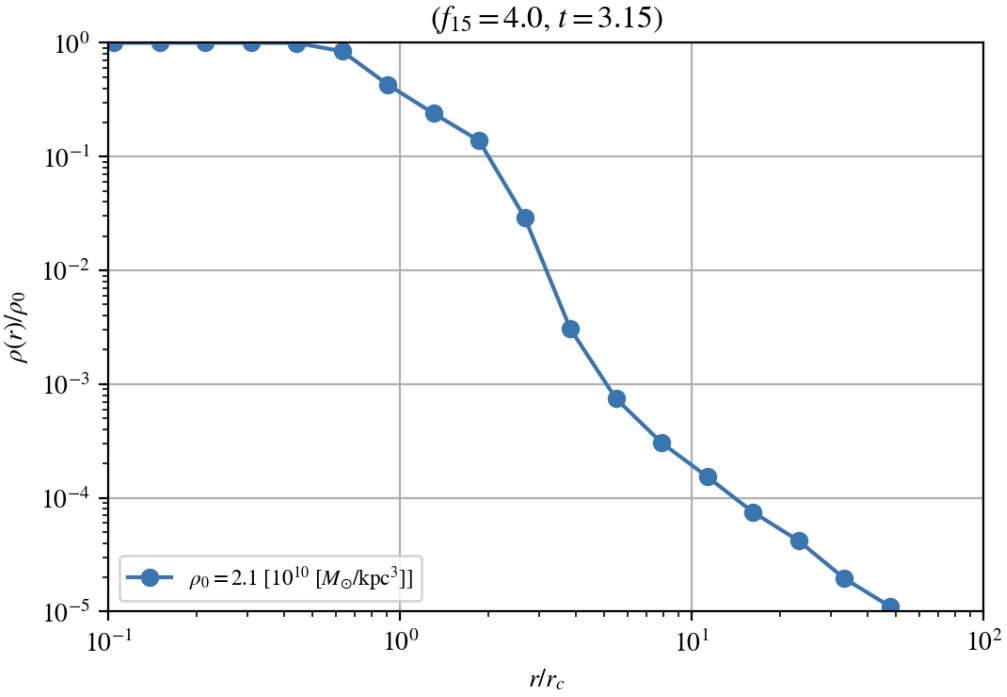
Returns

- `mids` : list of length `shells`
    - List of radii of the log-center of each created shell. In other words, the approximate radius at which the computed mean density is accurate.
- `rho_r` : list of length `shells`
    - List of mean density values computed in each shell.

Examples

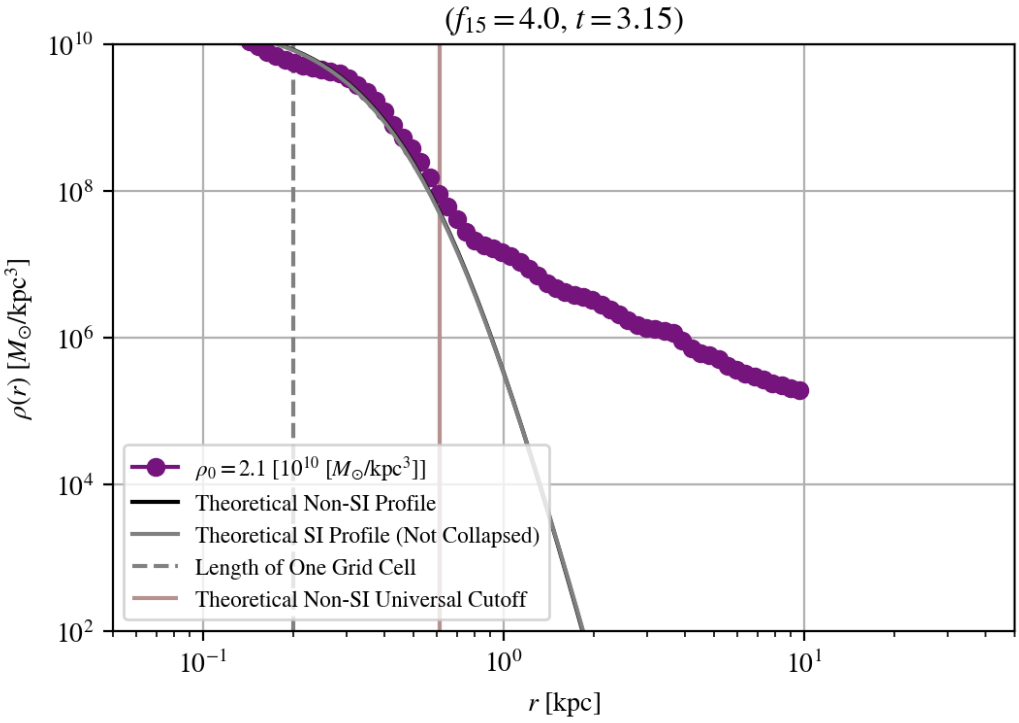
Get the normalized density profile and plot it without a fit (all default parameters):

```
>>    snap.densityProfile(filename=
'tutorial-density-profile-default')
```



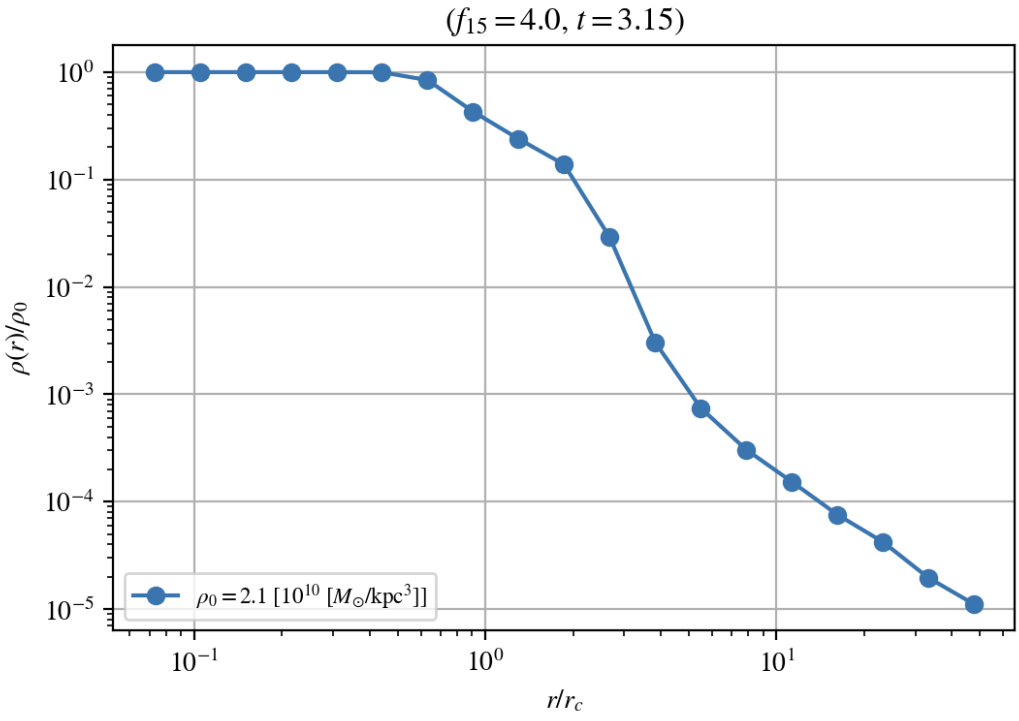
Increase the number of data points, include the fit, make the axes dimensionful, and color it purple:

```
>>    snap.densityProfile(shells=100, normalize=False,
fit=True, c='purple',
filename='tutorial-density-profile-fun')
```



Reset the limits of the plot to whatever matplotlib chooses:

```
>>    snap.densityProfile(lims=[None]*4,
filename='tutorial-density-profile-limits')
```



`solitonDensityFunction``(r, normalize=True, **kwargs)`

Theoretical soliton density function of radius (before collapse). In the future, this will incorporate theoretical predictions for profiles of collapsed solitons, too (polytrope star power laws, etc.).

Parameters

- `r` : float or numpy.ndarray
  - Radius at which to compute the density.
- `normalize` : (*Optional*) bool
  - Toggles whether to normalize the density measurements by the density at the center of the soliton and the radii by core radius.
- `**kwargs`
  - `noSI` : bool
  - `rho0` : float
  - `r_c` : float
  - `beta` : float

Returns

- `rho` : float
  - Density evaluated at given radius.

Examples

Predict the fractional density at the soliton core radius:

```
>>    snap.solitonDensityFunction(1)
>>    0.49157508628080004
```

`fitProfileTail``(rmin=7.0, plot=True, ax=None, **kwargs)`

Fits the tail of the density profile to a power law with the option of plotting.

Parameters

- `rmin` : float
  - Radius at which the power law behavior of the tail begins.
- `plot` : (*Optional*) bool
  - Toggles whether to plot the tail data and its fit.
- `ax` : `matplotlib.axes.Axes`
  - Axes on which to plot the profile.

- `**kwargs`
  - `dpi` : bool
  - `save` : float
  - `filename` : float
  - Other keyword arguments passed to `matplotlib.axes.Axes.loglog(...)`

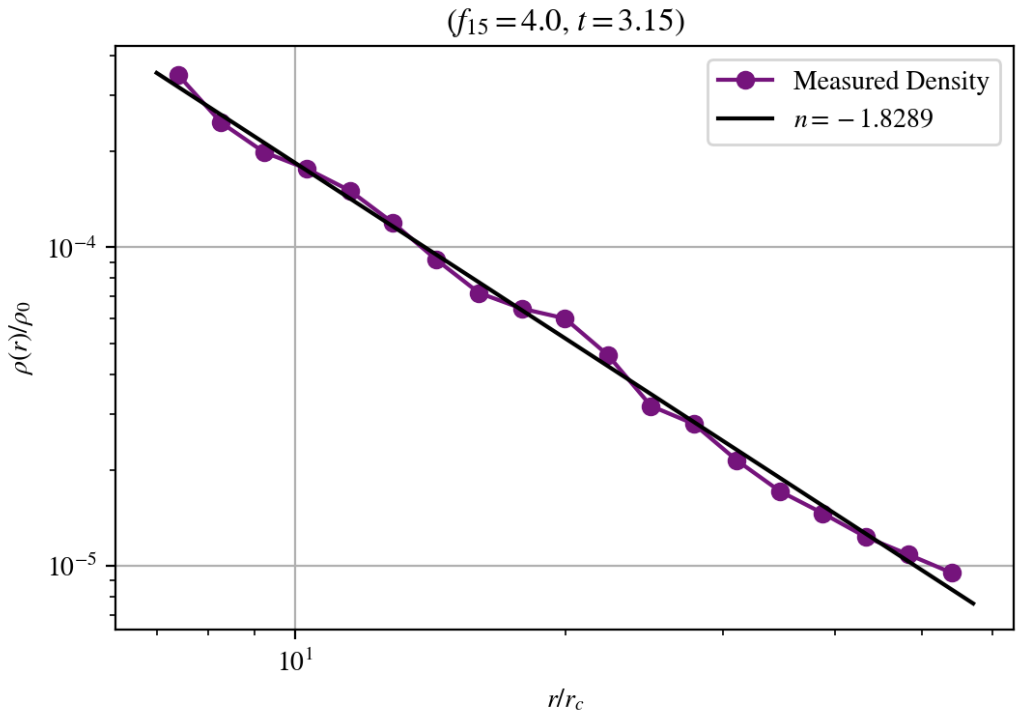
Returns

- `popt` : numpy.ndarray
  - Optimal values for the power law amplitude and index of the density profile tail.
- `pcov` : numpy.ndarray
  - Estimated covariance matrix of `popt`.

Examples

Plot the density profile tail (in purple) and its power law fit:

```
>>    snap.fitProfileTail(c='purple',
filename='tutorial-density-profile-tail')
```



Miscellaneous

Helpful Dictionaries

There are several helpful arrays and dictionaries stored near the top of the document as global variables and used throughout the code. They can be called from the console, as well.

- `Q` : dict
    - Dictionary that takes as input the recognized string name of a quantity and outputs a more descriptive, formatted title of it.
    - This is a good place to reference all quantities recognized by the code (and to clarify what they mean).
- `Q0, Q1, Q2, Q3, Q4` : lists
    - Lists that contain the recognized string names of quantities separated by the natural dimension of their array at each snapshot.
    - For instance, quantities in `Q0` are scalars, `Q1` are vectors, `Q2` matrices, etc.
- `U` : dict
    - Dictionary that takes as input either the recognized string name of a quantity or one of 'length', 'mass', 'time', 'density', 'energy', or 'velocity', and outputs the corresponding units formatted in an r-string.
    - For instance, `U['rho']` yields `'density'`, since `'rho'` has units of `'density'`. `U[U['rho']]` yields `r"[$M_{\odot}/\mathrm{kpc}^3$]"`, the units of density used in the simulation.
- `C` : dict
    - Dictionary containing the default colormaps used for visualizing a select few quantities.
- `LTX` : dict
    - Dictionary that takes as input the recognized string name of a quantity and outputs LaTeX text (for input into matplotlib functions, etc.).

Helper Functions

After the main code, there are a few helper functions that are probably not of use to anyone except the functions in the main code. There are some brief comments about them in the document. If it becomes convenient or necessary, I may add their documentation here later.