hmf Documentation

Release 2.0.0

Steven Murray

February 19, 2016

Contents

The halo mass function calculator. *hmf* is a python application that provides a flexible and simple way to calculate the Halo Mass Function for a range of varying parameters. It is also the backend to HMFcalc, the online HMF calculator.

CHAPTER 1

Documentation

Read the docs.

CHAPTER 2

Attribution

Please cite Murray, Power and Robotham (2013) if you find this code useful in your research.

CHAPTER 3

Features

- Calculate mass functions and related quantities extremely easily.
- Very simple to start using, but wide-ranging flexibility.
- Caching system for optimal parameter updates, for efficient iteration over parameter space.
- Support for all LambdaCDM cosmologies.
- Focus on flexibility in models. Each "Component", such as fitting functions, filter functions, growth factor models and transfer function fits are implemented as generic classes that can easily be altered by the user without touching the source code.
- Focus on simplicity in frameworks. Each "Framework" mixes available "Components" to derive useful quantities – all given as attributes of the Framework.
- Comprehensive in terms of output quantities: access differential and cumulative mass functions, mass variance, effective spectral index, growth rate, cosmographic functions and more.
- **Comprehensive in terms of implemented Component models**
  - 5+ models of transfer functions including directly from CAMB
  - 4 filter functions
  - 20 hmf fitting functions
- Includes models for Warm Dark Matter
- Nonlinear power spectra via HALOFIT
- Functions for sampling the mass function.
- CLI scripts both for producing any quantity included, or fitting any quantity.

CHAPTER 4

Installation

hmf is built on several other packages, most of which will be familiar to the scientific python programmer. All of these dependencies *should* be automatically installed when installing *hmf*, except for one. Explicitly, the dependencies are numpy, scipy, scitools, cosmolopy and emcee.

You will only need *emcee* if you are going to be using the fitting capabilities of *hmf*. The final, optional, library is pycamb, which can not be installed using pip currently.

Please follow the guidelines on its readme page. installation instructions.

Note: At present, versions of CAMB post March 2013 are not working with *pycamb*. Please use earlier versions until further notice.

Finally the *hmf* package needs to be installed: `pip install hmf`. If you want to install the latest build (not necessarily stable), grab it here.

To go really bleeding edge, install the develop branch using `pip install git+git://github.com/steven-murray/hmf.git@develop`.

CHAPTER 5

Quickstart

Once you have *hmf* installed, you can quickly generate a mass function by opening an interpreter (e.g. IPython) and doing:

```
>>> from hmf import MassFunction
>>> hmf = MassFunction()
>>> mass_func = hmf.dndlnm
```

Note that all parameters have (what I consider reasonable) defaults. In particular, this will return a Sheth-Mo-Tormen (2001) mass function between $10^{10} - 10^{15} M_{\odot}$, at $z = 0$ for the default PLANCK15 cosmology. Nevertheless, there are several parameters which can be input, either cosmological or otherwise. The best way to see these is to do

```
>>> MassFunction.parameter_info()
```

We can also check which parameters have been set in our "default" instance:

```
>>> hmf.parameter_values
```

To change the parameters (cosmological or otherwise), one should use the *update()* method, if a MassFunction() object already exists. For example

```
>>> hmf = MassFunction()
>>> hmf.update(Ob0 = 0.05, z=10) #update baryon density and redshift
>>> cumulative_mass_func = hmf.ngtm
```

For a more involved introduction to *hmf*, check out the tutorials, which are currently under construction, or the API docs.

CHAPTER 6

Author

- Steven Murray

CHAPTER 7

Comments, corrections and suggestions

- Jordan Mirocha (UCLA)
- Chris Power (UWA)
- Aaron Robotham (UWA)
- Alexander Knebe (UAMadrid)
- Peter Behroozi (UC Berkeley)

CHAPTER 8

Contents

8.1 Usage and Tutorials

One way to pick up how to use `hmf` is to directly consult the API documentation.

Here, however, we have compiled several more high-level resources on how to get started with `hmf`, and use it efficiently.

8.1.1 Dealing with Cosmological Models

`hmf` uses the robust `astropy` cosmology framework to deal with cosmological models. This provides a range of cosmographic functionality for free.

Cosmological models are the most basic `Framework` within `hmf`. Every other `Framework` depends on it. So knowing how to specify the models is important (but very simple!).

```
from hmf import cosmo
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Default Settings

Like everything in `hmf`, the `Cosmology` framework has all parameters specified with defaults. In this case, there are only two parameters – a base cosmological model, and a dictionary of cosmological parameters with which to alter it. By default, the cosmological model is a Flat LambdaCDM model infused with the Planck15 parameters. The dictionary is empty, so we don't modify anything:

```
my_cosmo = cosmo.Cosmology()
```

The intrinsic `astropy` object is found as the `cosmo` attribute of the class we just created. Beware, there is also a `cosmo_model` attribute, which should only be treated as a parameter, never used in calculations. It has not been supplemented with any custom parameters. We can check out the parameters defined within the model:

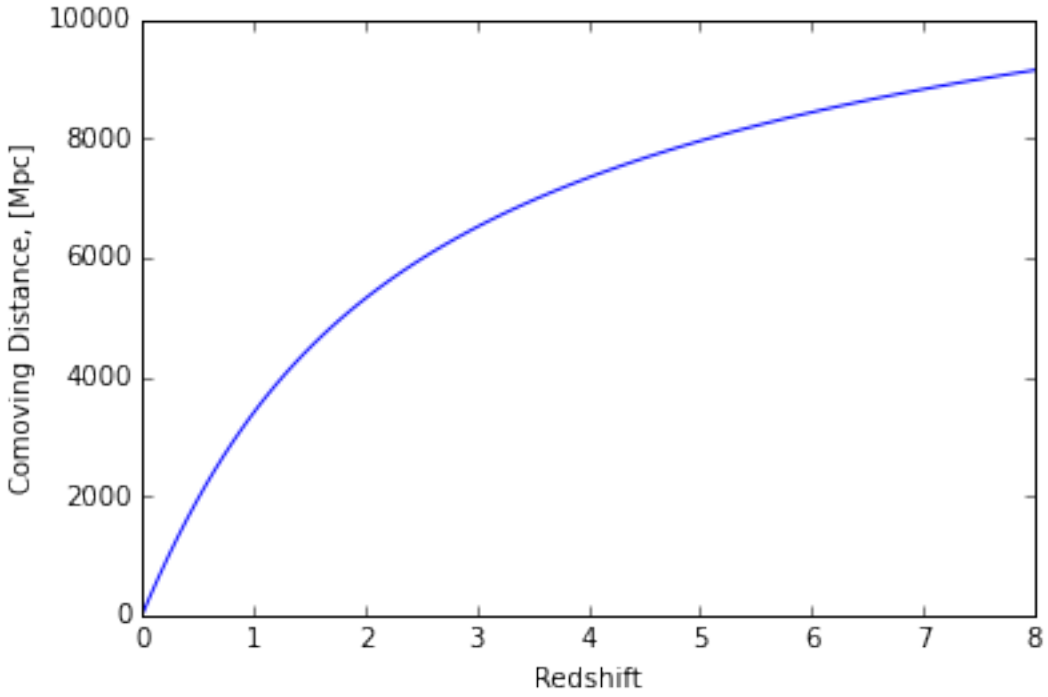
```
print "Matter density: ", my_cosmo.cosmo.Om0
print "Hubble constant: ", my_cosmo.cosmo.H0
print "Dark Energy density: ", my_cosmo.cosmo.Ode0
print "Baryon density: ",  my_cosmo.cosmo.Ob0
print "Curvature density: ", my_cosmo.cosmo.Ok0
```

```
Matter density:  0.3075
Hubble constant:  67.74 km / (Mpc s)
Dark Energy density:  0.691009934459
Baryon density:  0.0486
Curvature density:  0.0
```

Or we can check out some cosmographic quantities, like the comoving distance as a function of redshift:

```
z = np.linspace(0,8,100)
plt.plot(z,my_cosmo.cosmo.comoving_distance(z))
plt.ylabel("Comoving Distance, [Mpc]")
plt.xlabel("Redshift")
```

```
<matplotlib.text.Text at 0x7fa3b8b94a10>
```



Passing a cosmological model

The `cosmo` module contains several pre-made instances of cosmologies which might be useful, which we can input as our default model:

```
my_cosmo = cosmo.Cosmology(cosmo_model=cosmo.WMAP5)

print "WMAP5 baryon density: ", my_cosmo.cosmo.Ob0
```

```
WMAP5 baryon density:  0.0459
```

Alternatively, we can create our own. The `astropy` package contains the basic tools to do this. To create a standard Flat LambdaCDM cosmology:

```
from astropy.cosmology import FlatLambdaCDM
new_model = FlatLambdaCDM(H0 = 75.0, Om0=0.4, Tcmb0 = 5.0, Ob0 = 0.3)
```

This new model can be used as input to the `Cosmology` class:

```
my_cosmo = cosmo.Cosmology(cosmo_model = new_model)
print "Crazy cosmology baryon density: ", my_cosmo.cosmo.Ob0
```

```
Crazy cosmology baryon density:  0.3
```

The `cosmo_model` needn't be a Flat LambdaCDM. It can be any subclass of FLRW. Thus we could use a non-flat model:

```
from astropy.cosmology import LambdaCDM
new_model = LambdaCDM(H0 = 75.0, Om0=0.4, Tcmb0 = 0.0, Ob0 = 0.3, Ode0=0.4)

my_cosmo = cosmo.Cosmology(cosmo_model = new_model)
print "Crazy cosmology curvature density: ", my_cosmo.cosmo.Ok0
```

```
Crazy cosmology curvature density:  0.2
```

Passing custom parameters

Instead of passing a pre-made cosmological model, you can pass custom parameters for the default model. This is passed as a dictionary, in which each entry is a valid parameter for the model that has been passed (i.e., if the model is a FlatLambdaCDM, you can't pass `Ode0`!). This means you can specify the cosmology you want typically in one line, rather than a few. It also means that parameters can be updated in a standard way, so that iterating over parameters, in applications such as fitting models, becomes simple.

When passing the dictionary of parameters, you don't need to specify them all, just whichever ones you want to modify:

```
my_cosmo = cosmo.Cosmology(cosmo_params={"Om0":0.2})
print "Custom cosmology matter density: ", my_cosmo.cosmo.Om0
```

```
Custom cosmology matter density:  0.2
```

New parameters are available for extended cosmological models:

```
my_cosmo = cosmo.Cosmology(new_model,{"Om0":0.2,"Ode0":0.0,"Ob0":0.2})
print "Custom cosmology curvature density: ", my_cosmo.cosmo.Ok0
```

```
Custom cosmology curvature density:  0.8
```

Updating parameters

One of the great things about `hmf` Frameworks is that any parameter can be updated without re-creating the entire object. This is also true of the `Cosmology` class.

Any parameter passed to the constructor may also be updated:

```
my_cosmo = cosmo.Cosmology(new_model)
my_cosmo.update(cosmo_params={"Om0":0.2,"Ode0":0.0,"Ob0":0.2})
print "Custom cosmology curvature density: ", my_cosmo.cosmo.Ok0
```

```
Custom cosmology curvature density:  0.8
```

The parameter dictionary is persistent, so that updating a different parameter doesn't affect the others:

```
my_cosmo.update(cosmo_params={"H0":10.0})
print "Custom cosmology curvature density: ", my_cosmo.cosmo.Ok0
print "Custom parameters: ", my_cosmo.cosmo_params
```

```
Custom cosmology curvature density:  0.8
Custom parameters:  {'H0': 10.0, 'Om0': 0.2, 'Ode0': 0.0, 'Ob0': 0.2}
```

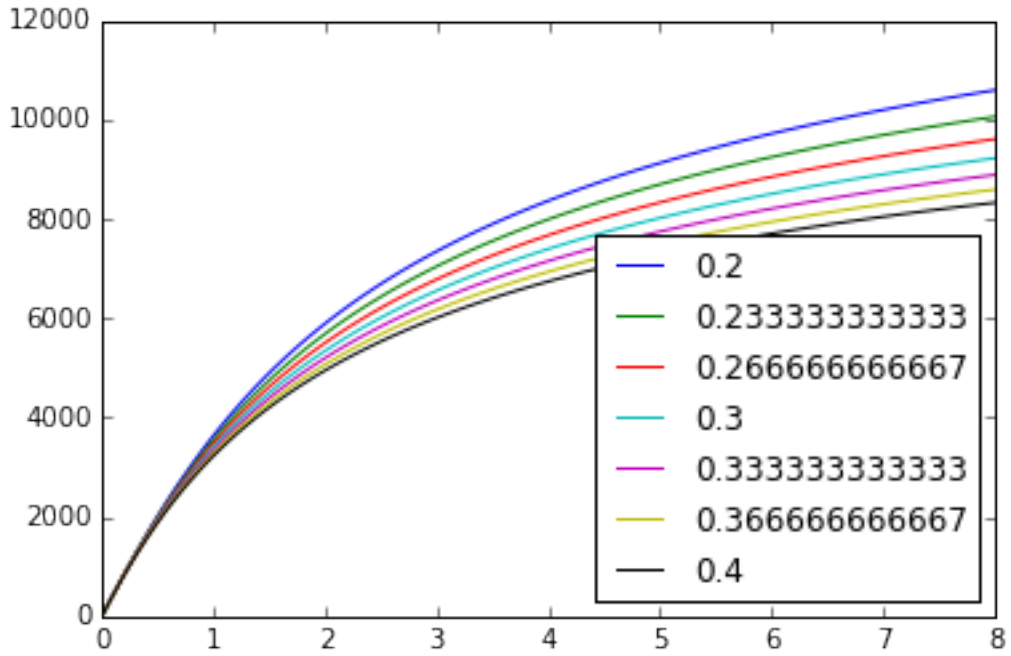
Of course, if we were to update the model to a Flat Lambda CDM model, then the `Ode0` keyword would give an error. To facilitate this, passing an empty dictionary clears all custom values:

```
my_cosmo.update(cosmo_model=cosmo.Planck13,cosmo_params={})
print "Flat cosmology curvature density: ", my_cosmo.cosmo.Ok0
```

```
Flat cosmology curvature density:  0.0
```

In effect, this gives us an easy way to track changes induced by a cosmological variable:

```
for Om0 in np.linspace(0.2,0.4,7):
    my_cosmo.update(cosmo_params={"Om0":Om0})
    plt.plot(z,my_cosmo.cosmo.comoving_distance(z),label="%s"%Om0)
_ = plt.legend(loc=0)
```



8.2 License

8.3 API Summary

hmf.cosmo	Module dealing with cosmological models.
hmf.transfer_models	Various models for computing the transfer function.
hmf.transfer	Module providing a framework for transfer functions.
hmf.halofit	Implements the HALOFIT (Smith+2003, Takahashi+2012) method.
hmf.growth_factor	Module defining the growth factor *Component*.
hmf.filters	A module containing various smoothing filter Component models, including the popular top-hat in r
hmf.fitting_functions	A module defining several mass function fits.
hmf.hmf	The primary module for user-interaction with the `hmf` package.
hmf.tools	
hmf.wdm	Module containing Warm Dark Matter models.
hmf.integrate_hmf	A supporting module that provides a routine to integrate the differential hmf in a robust manner.
hmf.functional	This module provides functions for generating several *hmf.hmf.MassFunction* instances from
hmf.sample	Module for dealing with sampled mass functions.
hmf._framework	Classes defining the overall structure of the hmf framework.

8.3.1 hmf.cosmo

Module dealing with cosmological models.

The main class is *Cosmology*, which is a framework wrapping the astropy cosmology classes, while converting it to a *hmf._framework.Framework* for use in this package.

Also provided in the namespace are the pre-defined cosmologies from *astropy*: *WMAP5*, *WMAP7*, *WMAP9*, *Planck13* and *Planck15*, which may be used as arguments to the Cosmology framework. All custom subclasses of `astropy.cosmology.FLRW` may be used as inputs.

Functions

get_cosmo(name)	Returns a FLRW cosmology given a string (must be one defined in this module).

hmf.cosmo.get_cosmo

hmf.cosmo.**get_cosmo**(*name*)

Returns a FLRW cosmology given a string (must be one defined in this module).

Parameters **name** : str

The class name of the appropriate model

Classes

Cosmology([cosmo_model, H0, Om0, Tcmb0, ...])	Basic Cosmology object.
FLRW(H0, Om0, Ode0[, Tcmb0, Neff, m_nu, ...])	A class describing an isotropic and homogeneous (Friedmann-Lemaitre-Rober

hmf.cosmo.Cosmology

class hmf.cosmo.**Cosmology**(*cosmo_model=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *cosmo_params=None*)

Basic Cosmology object.

This class thinly wraps cosmology objects from the astropy package. The full functionality of the astropy cosmology objects are available in the *cosmo* attribute. What the class adds to the existing astropy implementation is the specification of the cosmological parameters as *parameter* inputs to an over-arching Framework.

In particular, while any instance of a subclass of `astropy.cosmology.FLRW` may be passed as the base cosmology, the specific parameters can be updated individually by passing them through the *cosmo_params* dictionary (both in the constructor and the *update()* method.

This dictionary is kept in memory and so adding a different parameter on a later update will *update* the dictionary, rather than replacing it.

To read a standard documented list of parameters, use `Cosmology.parameter_info()`. If you want to just see the plain list of available parameters, use `Cosmology.get_all_parameters()`. To see the actual defaults for each parameter, use `Cosmology.get_all_parameter_defaults()`.

Methods

__init__([cosmo_model, H0, Om0, Tcmb0, ...])	
get_all_parameter_defaults()	Dictionary of all parameters and defaults
get_all_parameter_names()	Yield all parameter names in the class.
get_all_parameters()	Yield all parameters as tuples of (name,obj)
parameter_info()	
update(**kwargs)	Update parameters of the framework with kwargs.

hmf.cosmo.Cosmology.__init__

Cosmology.**__init__**(*cosmo_model=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *cosmo_params=None*)

hmf.cosmo.Cosmology.get_all_parameter_defaults

Cosmology.**get_all_parameter_defaults**()

Dictionary of all parameters and defaults

hmf.cosmo.Cosmology.get_all_parameter_names

Cosmology.**get_all_parameter_names**()

Yield all parameter names in the class.

hmf.cosmo.Cosmology.get_all_parameters

Cosmology.**get_all_parameters**()
Yield all parameters as tuples of (name,obj)

hmf.cosmo.Cosmology.parameter_info

Cosmology.**parameter_info**()

hmf.cosmo.Cosmology.update

Cosmology.**update**(***kwargs*)
Update parameters of the framework with kwargs.

Attributes

cosmo	Cosmographic object (astropy.cosmology.FLRW object), with custom cosmology from *cosmo_pa*
cosmo_model	**Parameter**: The basis for the cosmology – see astropy documentation. Can be a custom
cosmo_params	**Parameter**: Parameters for the cosmology that deviate from the base cosmology passed.
mean_density0	Mean density of universe at z=0, [Msun h^2 / Mpc**3]
parameter_values	Dictionary of all parameters and their current values

hmf.cosmo.Cosmology.cosmo

Cosmology.**cosmo**
Cosmographic object (astropy.cosmology.FLRW object), with custom cosmology from *cosmo_params* applied.

hmf.cosmo.Cosmology.cosmo_model

Cosmology.**cosmo_model**
Parameter: The basis for the cosmology – see astropy documentation. Can be a custom subclass. Defaults to Planck15.

Type instance of *astropy.cosmology.FLRW* subclass

hmf.cosmo.Cosmology.cosmo_params

Cosmology.**cosmo_params**
Parameter: Parameters for the cosmology that deviate from the base cosmology passed. This is useful for repeated updates of a single parameter (leaving others the same). Default is the empty dict. The parameters passed must match the allowed parameters of *cosmo_model*. For the basic class this is

Tcmb0 Temperature of the CMB at z=0

Neff Number of massless neutrino species

M_nu Mass of neutrino species (list)

H0 The hubble constant at z=0

Om0 The normalised matter density at z=0

Type dict

hmf.cosmo.Cosmology.mean_density0

Cosmology.**mean_density0**
Mean density of universe at z=0, [Msun h^2 / Mpc**3]

hmf.cosmo.Cosmology.parameter_values

Cosmology.**parameter_values**
Dictionary of all parameters and their current values

hmf.cosmo.FLRW

class hmf.cosmo.**FLRW**(*H0*, *Om0*, *Ode0*, *Tcmb0=2.725*, *Neff=3.04*, *m_nu=<Quantity 0.0 eV>*, *Ob0=None*, *name=None*)
A class describing an isotropic and homogeneous (Friedmann-Lemaitre-Robertson-Walker) cosmology.

This is an abstract base class – you can't instantiate examples of this class, but must work with one of its subclasses such as *LambdaCDM* or *wCDM*.

Parameters **H0** : float or scalar ~*astropy.units.Quantity*

Hubble constant at z = 0. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at z=0. Note that this does not include massive neutrinos.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at z=0.

Tcmb0 : float or scalar ~*astropy.units.Quantity*, optional

Temperature of the CMB z=0. If a float, must be in [K]. Default: 2.725 [K]. Setting this to zero will turn off both photons and neutrinos (even massive ones).

Neff : float, optional

Effective number of Neutrino species. Default 3.04.

m_nu : ~*astropy.units.Quantity*, optional

Mass of each neutrino species. If this is a scalar Quantity, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of m_nu if it is not scalar) must be the floor of Neff. Typically this means you should provide three neutrino masses unless you are considering something like a sterile neutrino.

Ob0 : float or None, optional

Omega baryons: density of baryonic matter in units of the critical density at z=0. If this is set to None (the default), any computation that requires its value will raise an exception.

name : str, optional

Name for this cosmological object.

Notes

Class instances are static – you can't change the values of the parameters. That is, all of the attributes above are read only.

Methods

H(z)	Hubble parameter (km/s/Mpc) at redshift z.
Ob(z)	Return the density parameter for baryonic matter at redshift z.
Ode(z)	Return the density parameter for dark energy at redshift z.
Odm(z)	Return the density parameter for dark matter at redshift z.
Ogamma(z)	Return the density parameter for photons at redshift z.
Ok(z)	Return the equivalent density parameter for curvature at redshift z.
Om(z)	Return the density parameter for non-relativistic matter at redshift z.
Onu(z)	Return the density parameter for massless neutrinos at redshift z.
Tcmb(z)	Return the CMB temperature at redshift z.
Tnu(z)	Return the neutrino temperature at redshift z.
__init__(H0, Om0, Ode0[, Tcmb0, Neff, m_nu, ...])	
abs_distance_integrand(z)	Integrand of the absorption distance.
absorption_distance(z)	Absorption distance at redshift z.
age(z)	Age of the universe in Gyr at redshift z.
angular_diameter_distance(z)	Angular diameter distance in Mpc at a given redshift.
angular_diameter_distance_z1z2(z1, z2)	Angular diameter distance between objects at 2 redshifts.
arcsec_per_kpc_comoving(z)	Angular separation in arcsec corresponding to a comoving kpc at redshift z
arcsec_per_kpc_proper(z)	Angular separation in arcsec corresponding to a proper kpc at redshift z.
clone(**kwargs)	Returns a copy of this object, potentially with some changes.
comoving_distance(z)	Comoving line-of-sight distance in Mpc at a given redshift.
comoving_transverse_distance(z)	Comoving transverse distance in Mpc at a given redshift.
comoving_volume(z)	Comoving volume in cubic Mpc at redshift z.
critical_density(z)	Critical density in grams per cubic cm at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.
differential_comoving_volume(z)	Differential comoving volume at redshift z.
distmod(z)	Distance modulus at redshift z.
efunc(z)	Function used to calculate H(z), the Hubble parameter.
inv_efunc(z)	Inverse of efunc.
kpc_comoving_per_arcmin(z)	Separation in transverse comoving kpc corresponding to an arcminute at red
kpc_proper_per_arcmin(z)	Separation in transverse proper kpc corresponding to an arcminute at redshi
lookback_distance(z)	The lookback distance is the light travel time distance to a given redshift.
lookback_time(z)	Lookback time in Gyr to redshift z.
lookback_time_integrand(z)	Integrand of the lookback time.
luminosity_distance(z)	Luminosity distance in Mpc at redshift z.
nu_relative_density(z)	Neutrino density function relative to the energy density in photons.
scale_factor(z)	Scale factor at redshift z.
w(z)	The dark energy equation of state.

hmf.cosmo.FLRW.H

FLRW.**H**(*z*)

Hubble parameter (km/s/Mpc) at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **H** : *~astropy.units.Quantity*

Hubble parameter at each input redshift.

hmf.cosmo.FLRW.Ob

FLRW.**Ob**(*z*)

Return the density parameter for baryonic matter at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Ob** : ndarray, or float if input scalar

The density of baryonic matter relative to the critical density at each redshift.

Raises **ValueError**

If Ob0 is None.

hmf.cosmo.FLRW.Ode

FLRW.**Ode**(*z*)

Return the density parameter for dark energy at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Ode** : ndarray, or float if input scalar

The density of non-relativistic matter relative to the critical density at each redshift.

hmf.cosmo.FLRW.Odm

FLRW.**Odm**(*z*)

Return the density parameter for dark matter at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Odm** : ndarray, or float if input scalar

The density of non-relativistic dark matter relative to the critical density at each redshift.

Raises **ValueError**

If Ob0 is None.

Notes

—

This does not include neutrinos, even if non-relativistic

at the redshift of interest.

hmf.cosmo.FLRW.Ogamma

FLRW.**Ogamma**(*z*)

Return the density parameter for photons at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Ogamma** : ndarray, or float if input scalar

The energy density of photons relative to the critical density at each redshift.

hmf.cosmo.FLRW.Ok

FLRW.**Ok**(*z*)

Return the equivalent density parameter for curvature at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Ok** : ndarray, or float if input scalar

The equivalent density parameter for curvature at each redshift.

hmf.cosmo.FLRW.Om

FLRW.**Om**(*z*)

Return the density parameter for non-relativistic matter at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Om** : ndarray, or float if input scalar

The density of non-relativistic matter relative to the critical density at each redshift.

Notes

This does not include neutrinos, even if non-relativistic at the redshift of interest; see *Onu*.

hmf.cosmo.FLRW.Onu

FLRW.**Onu**(*z*)

Return the density parameter for massless neutrinos at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Onu** : ndarray, or float if input scalar

The energy density of neutrinos relative to the critical density at each redshift. Note that this includes their kinetic energy (if they have mass), so it is not equal to the commonly used $\sum \frac{m_\nu}{94eV}$, which does not include kinetic energy.

hmf.cosmo.FLRW.Tcmb

FLRW.**Tcmb**(*z*)

Return the CMB temperature at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Tcmb** : *~astropy.units.Quantity*

The temperature of the CMB in K.

hmf.cosmo.FLRW.Tnu

FLRW.**Tnu**(*z*)

Return the neutrino temperature at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **Tnu** : *~astropy.units.Quantity*

The temperature of the cosmic neutrino background in K.

hmf.cosmo.FLRW.__init__

FLRW.**__init__**(*H0*, *Om0*, *Ode0*, *Tcmb0=2.725*, *Neff=3.04*, *m_nu=<Quantity 0.0 eV>*, *Ob0=None*, *name=None*)

hmf.cosmo.FLRW.abs_distance_integrand

FLRW.**abs_distance_integrand**(*z*)

Integrand of the absorption distance.

Parameters **z** : float or array

Input redshift.

Returns **X** : float or array

The integrand for the absorption distance

References

See Hogg 1999 section 11.

hmf.cosmo.FLRW.absorption_distance

FLRW.**absorption_distance**(*z*)
Absorption distance at redshift `z`.

This is used to calculate the number of objects with some cross section of absorption and number density intersecting a sightline per unit redshift path.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : float or ndarray

Absorption distance (dimensionless) at each input redshift.

References

Hogg 1999 Section 11. (astro-ph/9905116) Bahcall, John N. and Peebles, P.J.E. 1969, ApJ, 156L, 7B

hmf.cosmo.FLRW.age

FLRW.**age**(*z*)
Age of the universe in Gyr at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **t** : *~astropy.units.Quantity*

The age of the universe in Gyr at each input redshift.

See also:

z_at_value Find the redshift corresponding to an age.

hmf.cosmo.FLRW.angular_diameter_distance

FLRW.**angular_diameter_distance**(*z*)
Angular diameter distance in Mpc at a given redshift.

This gives the proper (sometimes called 'physical') transverse distance corresponding to an angle of 1 radian for an object at redshift `z`.

Weinberg, 1972, pp 421-424; Weedman, 1986, pp 65-67; Peebles, 1993, pp 325-327.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : *~astropy.units.Quantity*

Angular diameter distance in Mpc at each input redshift.

hmf.cosmo.FLRW.angular_diameter_distance_z1z2

FLRW.**angular_diameter_distance_z1z2**(*z1*, *z2*)

Angular diameter distance between objects at 2 redshifts. Useful for gravitational lensing.

Parameters **z1, z2** : array-like, shape (N,)

Input redshifts. z2 must be large than z1.

Returns **d** : *~astropy.units.Quantity*, shape (N,) or single if input scalar

The angular diameter distance between each input redshift pair.

Raises **CosmologyError**

If omega_k is < 0.

Notes

This method only works for flat or open curvature (omega_k >= 0).

hmf.cosmo.FLRW.arcsec_per_kpc_comoving

FLRW.**arcsec_per_kpc_comoving**(*z*)

Angular separation in arcsec corresponding to a comoving kpc at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **theta** : *~astropy.units.Quantity*

The angular separation in arcsec corresponding to a comoving kpc at each input redshift.

hmf.cosmo.FLRW.arcsec_per_kpc_proper

FLRW.**arcsec_per_kpc_proper**(*z*)

Angular separation in arcsec corresponding to a proper kpc at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **theta** : *~astropy.units.Quantity*

The angular separation in arcsec corresponding to a proper kpc at each input redshift.

hmf.cosmo.FLRW.clone

FLRW.**clone**(***kwargs*)

Returns a copy of this object, potentially with some changes.

Returns **newcos** : Subclass of FLRW

A new instance of this class with the specified changes.

Notes

This assumes that the values of all constructor arguments are available as properties, which is true of all the provided subclasses but may not be true of user-provided ones. You can't change the type of class, so this can't be used to change between flat and non-flat. If no modifications are requested, then a reference to this object is returned.

Examples

To make a copy of the Planck13 cosmology with a different Omega_m and a new name:

```
>>> from astropy.cosmology import Planck13
>>> newcos = Planck13.clone(name="Modified Planck 2013", Om0=0.35)
```

hmf.cosmo.FLRW.comoving_distance

FLRW.**comoving_distance**(*z*)

Comoving line-of-sight distance in Mpc at a given redshift.

The comoving distance along the line-of-sight between two objects remains constant with time for objects in the Hubble flow.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : ndarray, or float if input scalar

Comoving distance in Mpc to each input redshift.

hmf.cosmo.FLRW.comoving_transverse_distance

FLRW.**comoving_transverse_distance**(*z*)

Comoving transverse distance in Mpc at a given redshift.

This value is the transverse comoving distance at redshift `z` corresponding to an angular separation of 1 radian. This is the same as the comoving distance if omega_k is zero (as in the current concordance lambda CDM model).

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : *~astropy.units.Quantity*

Comoving transverse distance in Mpc at each input redshift.

Notes

This quantity also called the 'proper motion distance' in some texts.

hmf.cosmo.FLRW.comoving_volume

FLRW.**comoving_volume**(*z*)

Comoving volume in cubic Mpc at redshift `z`.

This is the volume of the universe encompassed by redshifts less than `z`. For the case of omega_k = 0 it is a sphere of radius *comoving_distance* but it is less intuitive if omega_k is not 0.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **V** : *~astropy.units.Quantity*

Comoving volume in Mpc^3 at each input redshift.

hmf.cosmo.FLRW.critical_density

FLRW.**critical_density**(*z*)

Critical density in grams per cubic cm at redshift `z`.

Parameters **z** : array-like

Input redshifts.

Returns **rho** : *~astropy.units.Quantity*

Critical density in g/cm^3 at each input redshift.

hmf.cosmo.FLRW.de_density_scale

FLRW.**de_density_scale**(*z*)

Evaluates the redshift dependence of the dark energy density.

Parameters **z** : array-like

Input redshifts.

Returns **I** : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and is given by

$$I = \exp\left(3\int_a^1 \frac{da'}{a'}\left[1 + w\left(a'\right)\right]\right)$$

It will generally helpful for subclasses to overload this method if the integral can be done analytically for the particular dark energy equation of state that they implement.

hmf.cosmo.FLRW.differential_comoving_volume

FLRW.**differential_comoving_volume**(*z*)

Differential comoving volume at redshift z.

Useful for calculating the effective comoving volume. For example, allows for integration over a comoving volume that has a sensitivity function that changes with redshift. The total comoving volume is given by integrating differential_comoving_volume to redshift z and multiplying by a solid angle.

Parameters **z** : array-like

Input redshifts.

Returns **dV** : *~astropy.units.Quantity*

Differential comoving volume per redshift per steradian at each input redshift.

hmf.cosmo.FLRW.distmod

FLRW.**distmod**(*z*)

Distance modulus at redshift `z`.

The distance modulus is defined as the (apparent magnitude - absolute magnitude) for an object at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **distmod** : *~astropy.units.Quantity*

Distance modulus at each input redshift, in magnitudes

See also:

z_at_value Find the redshift corresponding to a distance modulus.

hmf.cosmo.FLRW.efunc

FLRW.**efunc**(*z*)

Function used to calculate H(z), the Hubble parameter.

Parameters **z** : array-like

Input redshifts.

Returns **E** : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

It is not necessary to override this method, but if de_density_scale takes a particularly simple form, it may be advantageous to.

hmf.cosmo.FLRW.inv_efunc

FLRW.**inv_efunc**(*z*)

Inverse of efunc.

Parameters **z** : array-like

Input redshifts.

Returns **E** : ndarray, or float if input scalar

The redshift scaling of the inverse Hubble constant.

hmf.cosmo.FLRW.kpc_comoving_per_arcmin

FLRW.**kpc_comoving_per_arcmin**(*z*)

Separation in transverse comoving kpc corresponding to an arcminute at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : *~astropy.units.Quantity*

The distance in comoving kpc corresponding to an arcmin at each input redshift.

hmf.cosmo.FLRW.kpc_proper_per_arcmin

FLRW.**kpc_proper_per_arcmin**(*z*)

Separation in transverse proper kpc corresponding to an arcminute at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : *~astropy.units.Quantity*

The distance in proper kpc corresponding to an arcmin at each input redshift.

hmf.cosmo.FLRW.lookback_distance

FLRW.**lookback_distance**(*z*)

The lookback distance is the light travel time distance to a given redshift. It is simply c * lookback_time. It may be used to calculate the proper distance between two redshifts, e.g. for the mean free path to ionizing radiation.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar

Returns **d** : *~astropy.units.Quantity*

Lookback distance in Mpc

hmf.cosmo.FLRW.lookback_time

FLRW.**lookback_time**(*z*)

Lookback time in Gyr to redshift `z`.

The lookback time is the difference between the age of the Universe now and the age at redshift `z`.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar

Returns **t** : *~astropy.units.Quantity*

Lookback time in Gyr to each input redshift.

See also:

z_at_value Find the redshift corresponding to a lookback time.

hmf.cosmo.FLRW.lookback_time_integrand

FLRW.**lookback_time_integrand**(*z*)

Integrand of the lookback time.

Parameters **z** : float or array-like

Input redshift.

Returns **I** : float or array

The integrand for the lookback time

References

Eqn 30 from Hogg 1999.

hmf.cosmo.FLRW.luminosity_distance

FLRW.**luminosity_distance**(*z*)

Luminosity distance in Mpc at redshift `z`.

This is the distance to use when converting between the bolometric flux from an object at redshift `z` and its bolometric luminosity.

Parameters **z** : array-like

Input redshifts. Must be 1D or scalar.

Returns **d** : *~astropy.units.Quantity*

Luminosity distance in Mpc at each input redshift.

See also:

z_at_value Find the redshift corresponding to a luminosity distance.

References

Weinberg, 1972, pp 420-424; Weedman, 1986, pp 60-62.

hmf.cosmo.FLRW.nu_relative_density

FLRW.**nu_relative_density**(*z*)

Neutrino density function relative to the energy density in photons.

Parameters **z** : array like

Redshift

Returns **f** : ndarray, or float if z is scalar

The neutrino density scaling factor relative to the density in photons at each redshift

Notes

The density in neutrinos is given by

$$\rho_\nu(a) = 0.2271\, N_{eff}\, f(m_\nu a/T_{\nu 0})\ \rho_\gamma(a)$$

where

$$f(y) = \frac{120}{7\pi^4}\int_0^\infty dx \frac{x^2\sqrt{x^2+y^2}}{e^x+1}$$

assuming that all neutrino species have the same mass. If they have different masses, a similar term is calculated for each one. Note that f has the asymptotic behavior $f(0) = 1$. This method returns $0.2271f$ using an analytical fitting formula given in Komatsu et al. 2011, ApJS 192, 18.

hmf.cosmo.FLRW.scale_factor

FLRW.**scale_factor**(*z*)

Scale factor at redshift `z`.

The scale factor is defined as $a = 1/(1+z)$.

Parameters **z** : array-like

Input redshifts.

Returns **a** : ndarray, or float if input scalar

Scale factor at each input redshift.

hmf.cosmo.FLRW.w

FLRW.**w**(*z*)

The dark energy equation of state.

Parameters **z** : array-like

Input redshifts.

Returns **w** : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1.

This must be overridden by subclasses.

Attributes

H0	Return the Hubble constant as an ~*astropy.units.Quantity* at z=0
Neff	Number of effective neutrino species
Ob0	Omega baryon; baryonic matter density/critical density at z=0
Ode0	Omega dark energy; dark energy density/critical density at z=0
Odm0	Omega dark matter; dark matter density/critical density at z=0
Ogamma0	Omega gamma; the density/critical density of photons at z=0
Ok0	Omega curvature; the effective curvature density/critical density
Om0	Omega matter; matter density/critical density at z=0
Onu0	Omega nu; the density/critical density of neutrinos at z=0
Tcmb0	Temperature of the CMB as ~*astropy.units.Quantity* at z=0
Tnu0	Temperature of the neutrino background as ~*astropy.units.Quantity* at z=0
critical_density0	Critical density as ~*astropy.units.Quantity* at z=0
h	Dimensionless Hubble constant: h = H_0 / 100 [km/sec/Mpc]
has_massive_nu	Does this cosmology have at least one massive neutrino species?
hubble_distance	Hubble distance as ~*astropy.units.Quantity*
hubble_time	Hubble time as ~*astropy.units.Quantity*
m_nu	Mass of neutrino species

hmf.cosmo.FLRW.H0

FLRW.**H0**
Return the Hubble constant as an ~*astropy.units.Quantity* at z=0

hmf.cosmo.FLRW.Neff

FLRW.**Neff**
Number of effective neutrino species

hmf.cosmo.FLRW.Ob0

FLRW.**Ob0**
Omega baryon; baryonic matter density/critical density at z=0

hmf.cosmo.FLRW.Ode0

FLRW.**Ode0**
Omega dark energy; dark energy density/critical density at z=0

hmf.cosmo.FLRW.Odm0

FLRW.**Odm0**
Omega dark matter; dark matter density/critical density at z=0

hmf.cosmo.FLRW.Ogamma0

FLRW.**Ogamma0**
Omega gamma; the density/critical density of photons at z=0

hmf.cosmo.FLRW.Ok0

FLRW.**Ok0**
Omega curvature; the effective curvature density/critical density at z=0

hmf.cosmo.FLRW.Om0

FLRW.**Om0**
Omega matter; matter density/critical density at z=0

hmf.cosmo.FLRW.Onu0

FLRW.**Onu0**
Omega nu; the density/critical density of neutrinos at z=0

hmf.cosmo.FLRW.Tcmb0

FLRW.**Tcmb0**
Temperature of the CMB as *~astropy.units.Quantity* at z=0

hmf.cosmo.FLRW.Tnu0

FLRW.**Tnu0**
Temperature of the neutrino background as *~astropy.units.Quantity* at z=0

hmf.cosmo.FLRW.critical_density0

FLRW.**critical_density0**
Critical density as *~astropy.units.Quantity* at z=0

hmf.cosmo.FLRW.h

FLRW.**h**
Dimensionless Hubble constant: h = H_0 / 100 [km/sec/Mpc]

hmf.cosmo.FLRW.has_massive_nu

FLRW.**has_massive_nu**
Does this cosmology have at least one massive neutrino species?

hmf.cosmo.FLRW.hubble_distance

FLRW.**hubble_distance**
Hubble distance as *~astropy.units.Quantity*

hmf.cosmo.FLRW.hubble_time

FLRW.**hubble_time**
Hubble time as *~astropy.units.Quantity*

hmf.cosmo.FLRW.m_nu

FLRW.**m_nu**
Mass of neutrino species

8.3.2 hmf.transfer_models

Various models for computing the transfer function.

Note that these are not transfer function "frameworks". The framework is found in *hmf.transfer*.

Classes

BBKS(cosmo, **model_parameters)	BBKS (1986) transfer function.
BondEfs(cosmo, **model_parameters)	Transfer function of Bond and Efstathiou
CAMB(cosmo, **model_parameters)	Transfer function computed by CAMB.
Component(**model_params)	Base class representing a component model.
EH(cosmo, **model_parameters)	Alias of *EH_BAO*
EH_BAO(cosmo, **model_parameters)	Eisenstein & Hu (1998) fitting function with BAO wiggles
EH_NoBAO(cosmo, **model_parameters)	Eisenstein & Hu (1998) fitting function without BAO wiggles
FromFile(cosmo, **model_parameters)	Import a transfer function from file.
TransferComponent(cosmo, **model_parameters)	Base class for transfer models.
spline	alias of InterpolatedUnivariateSpline

hmf.transfer_models.BBKS

class hmf.transfer_models.**BBKS**(*cosmo*, ***model_parameters*)
BBKS (1986) transfer function.

Parameters **cosmo** : astropy.cosmology.FLRW instance

The cosmology used in the calculation

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, available parameters are the following: **a, b, c, d, e**. To see their default values, check the _defaults class attribute.

Notes

The fit is given as

$$T(k) = \frac{\ln(1+aq)}{aq}\left(1 + bq + (cq)^2 + (dq)^3 + (eq)^4\right)^{-1/4},$$

where

$$q = \frac{k}{\Gamma} \exp\left(\Omega_{b,0} + \frac{\sqrt{2h}\Omega_{b,0}}{\Omega_{m,0}}\right)$$

and $\Gamma = \Omega_{m,0}h$.

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.BBKS.__init__

BBKS.**__init__**(*cosmo*, ***model_parameters*)

hmf.transfer_models.BBKS.lnt

BBKS.**lnt**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.BondEfs

class hmf.transfer_models.**BondEfs**(*cosmo*, ***model_parameters*)

Transfer function of Bond and Efstathiou

Parameters **cosmo** : astropy.cosmology.FLRW instance

The cosmology used in the calculation

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, available parameters are the following: **a, b, c, nu**. To see their default values, check the _defaults class attribute.

Notes

The fit is given as

$$T(k) = \left[1 + (\tilde{a}k + (\tilde{b}k)^{3/2} + (\tilde{c}k)^2)^\nu\right]^{-1/\nu}$$

where $\tilde{x} = x\alpha$ and

$$\alpha = \frac{0.3 \times 0.75^2}{\Omega_{m,0}h^2}.$$

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.BondEfs.__init__

BondEfs.**__init__**(*cosmo*, ***model_parameters*)

hmf.transfer_models.BondEfs.lnt

BondEfs.**lnt**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.CAMB

class hmf.transfer_models.**CAMB**(*cosmo*, ***model_parameters*)

Transfer function computed by CAMB.

Parameters **cosmo** : astropy.cosmology.FLRW instance

The cosmology used in the calculation

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, available parameters are the following. To see their default values, check the _defaults class attribute.

Scalar_initial_condition Initial scalar perturbation mode (adiabatic=1, CDM iso=2, Baryon iso=3,neutrino density iso =4, neutrino velocity iso = 5)

lAccuracyBoost Larger to keep more terms in the hierarchy evolution

AccuracyBoost Increase accuracy_boost to decrease time steps, use more k values, etc.Decrease to speed up at cost of worse accuracy. Suggest 0.8 to 3.

w_perturb Whether to perturb the dark energy equation of state.

transfer__k_per_logint Number of wavenumbers estimated per log interval by CAMB Default of 11 gets best performance for requisite accuracy of mass function.

transfer__kmax Maximum value of the wavenumber. Default of 0.25 is high enough for requisite accuracy of mass function.

ThreadNum Number of threads to use for calculation of transfer function by CAMB. Default 0 automatically determines the number.

scalar_amp Amplitude of the power spectrum. It is not recommended to modify this parameter, as the amplitude is typically set by sigma_8.

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.CAMB.__init__

CAMB.**__init__**(*cosmo*, ***model_parameters*)

hmf.transfer_models.CAMB.lnt

CAMB.**lnt**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.Component

class hmf.transfer_models.**Component**(***model_params*)

Base class representing a component model.

All components should be subclassed from this. Components are generally parts of the calculation which can take different models, example the HMF fitting functions, bias models, growth functions, etc.

The feature of this class is that it contains a class variable called _defaults containing the defaults for the parameters of any specific model. These are checked and updated with passed parameters by the __init__ method.

Methods

__init__(**model_params)

hmf.transfer_models.Component.__init__

Component.**__init__**(***model_params*)

hmf.transfer_models.EH

class hmf.transfer_models.**EH**(*cosmo*, ***model_parameters*)

Alias of *EH_BAO*

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.EH.__init__

EH.**__init__**(*cosmo*, ***model_parameters*)

hmf.transfer_models.EH.lnt

EH.**lnt**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.EH_BAO

class hmf.transfer_models.**EH_BAO**(*cosmo*, ***model_parameters*)

Eisenstein & Hu (1998) fitting function with BAO wiggles

From EH1998, Eqs. 26,28-31. Code adapted from CHOMP.

Parameters **cosmo** : astropy.cosmology.FLRW instance

The cosmology used in the calculation

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, there are no model parameters.

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.EH_BAO.__init__

EH_BAO.**__init__**(*cosmo*, ***model_parameters*)

hmf.transfer_models.EH_BAO.lnt

EH_BAO.**lnt**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.EH_NoBAO

class hmf.transfer_models.**EH_NoBAO**(*cosmo*, ***model_parameters*)

Eisenstein & Hu (1998) fitting function without BAO wiggles

From EH 1998 Eqs. 26,28-31. Code adapted from CHOMP project.

Parameters **cosmo** : astropy.cosmology.FLRW instance

The cosmology used in the calculation

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, there are no model parameters.

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.EH_NoBAO.__init__

EH_NoBAO.**__init__**(*cosmo*, ***model_parameters*)

hmf.transfer_models.EH_NoBAO.lnt

EH_NoBAO.**lnt**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.FromFile

class hmf.transfer_models.**FromFile**(*cosmo*, ***model_parameters*)

Import a transfer function from file.

Note: The file should be in the same format as output from CAMB, or else in two-column ASCII format (k,T).

Parameters **cosmo** : astropy.cosmology.FLRW instance

The cosmology used in the calculation

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, available parameters are the following. To see their default values, check the `_defaults` class attribute.

fname str Location of the file to import.

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.FromFile.__init__

`FromFile.`**`__init__`**(*cosmo*, ***model_parameters*)

hmf.transfer_models.FromFile.lnt

`FromFile.`**`lnt`**(*lnk*)

Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.TransferComponent

class `hmf.transfer_models.`**`TransferComponent`**(*cosmo*, ***model_parameters*)

Base class for transfer models.

The only necessary function to specify is `lnt`, which returns the log transfer given `lnk`.

Parameters **cosmo** : `astropy.cosmology.FLRW` instance

The cosmology used in the calculation

****model_parameters :**

Any model-specific parameters.

Methods

__init__(cosmo, **model_parameters)	
lnt(lnk)	Natural log of the transfer function

hmf.transfer_models.TransferComponent.__init__

`TransferComponent.`**`__init__`**(*cosmo*, ***model_parameters*)

hmf.transfer_models.TransferComponent.lnt

TransferComponent.**lnt**(*lnk*)
Natural log of the transfer function

Parameters **lnk** : array_like

Wavenumbers [Mpc/h]

Returns **lnt** : array_like

The log of the transfer function at lnk.

hmf.transfer_models.spline

hmf.transfer_models.**spline**
alias of InterpolatedUnivariateSpline

8.3.3 hmf.transfer

Module providing a framework for transfer functions.

This module contains a single class, *Transfer*, which provides methods to calculate the transfer function, matter power spectrum and several other related quantities.

Functions

cached_property(*parents)	A robust property caching decorator.
get_model(name, mod, **kwargs)	Returns an instance of name from the module mod, with given params.
parameter(f)	A simple cached property which acts more like an input value.

hmf.transfer.cached_property

hmf.transfer.**cached_property**(**parents*)
A robust property caching decorator.

This decorator only works when used with the entire system here....

Usage:: class CachedClass(Cache):

@cached_property("parent_parameter") def amethod(self):

...calculations... return final_value

@cached_property("amethod") def a_child_method(self): #method dependent on amethod

final_value = 3*self.amethod return final_value

This code will calculate amethod on the first call, but return the cached value on all subsequent calls. If any parent parameter is modified, the calculation will be re-performed.

hmf.transfer.get_model

hmf.transfer.**get_model**(*name*, *mod*, ***kwargs*)
Returns an instance of name from the module mod, with given params.

Parameters **name** : str

The class name of the appropriate model

mod : str

The module name of the appropriate module

****kwargs :**

Any parameters for the instantiated model (including model parameters)

hmf.transfer.parameter

hmf.transfer.**parameter**(*f*)

A simple cached property which acts more like an input value.

This cached property is intended to be used on values that are passed in `__init__`, and can possibly be reset later. It provides the opportunity for complex setters, and also the ability to update dependent properties whenever the value is modified.

Usage:: @set_property("amethod") def parameter(self,val):

if isinstance(int,val): return val

else: raise ValueError("parameter must be an integer")

@cached_property() def amethod(self):

return 3*self.parameter

Note that the definition of the setter merely returns the value to be set, it doesn't set it to any particular instance attribute. The decorator automatically sets `self.__parameter = val` and defines the get method accordingly

Classes

Transfer([sigma_8, n, z, lnk_min, lnk_max, ...])	A transfer function framework.

hmf.transfer.Transfer

class hmf.transfer.**Transfer**(*sigma_8=0.8344*, *n=0.9624*, *z=0.0*, *lnk_min=-18.420680743952367*, *lnk_max=9.9034875525361272*, *dlnk=0.05*, *transfer_model=<class 'hmf.transfer_models.EH'>*, *transfer_params=None*, *takahashi=True*, *growth_model=<class 'hmf.growth_factor.GrowthFactor'>*, *growth_params=None*, ***kwargs*)

A transfer function framework.

The purpose of this `hmf._frameworks.Framework` is to calculate transfer functions, power spectra and several tightly associated quantities given a basic model for the transfer function.

As in all frameworks, to update parameters optimally, use the *update()* method. All output quantities are calculated only when needed (but stored after first calculation for quick access).

In addition to the parameters directly passed to this class, others are available which are passed on to its superclass. To read a standard documented list of (all) parameters, use `Transfer.parameter_info()`. If you want to just see the plain list of available parameters, use `Transfer.get_all_parameters()`.To see the actual defaults for each parameter, use `Transfer.get_all_parameter_defaults()`.

Methods

__init__([sigma_8, n, z, lnk_min, lnk_max, ...])	
get_all_parameter_defaults()	Dictionary of all parameters and defaults
get_all_parameter_names()	Yield all parameter names in the class.
get_all_parameters()	Yield all parameters as tuples of (name,obj)
parameter_info()	
update(**kwargs)	Update parameters of the framework with kwargs.

hmf.transfer.Transfer.__init__

Transfer.**__init__**(*sigma_8=0.8344*, *n=0.9624*, *z=0.0*, *lnk_min=-18.420680743952367*, *lnk_max=9.9034875525361272*, *dlnk=0.05*, *transfer_model=<class 'hmf.transfer_models.EH'>*, *transfer_params=None*, *takahashi=True*, *growth_model=<class 'hmf.growth_factor.GrowthFactor'>*, *growth_params=None*, ***kwargs*)

hmf.transfer.Transfer.get_all_parameter_defaults

Transfer.**get_all_parameter_defaults**()
Dictionary of all parameters and defaults

hmf.transfer.Transfer.get_all_parameter_names

Transfer.**get_all_parameter_names**()
Yield all parameter names in the class.

hmf.transfer.Transfer.get_all_parameters

Transfer.**get_all_parameters**()
Yield all parameters as tuples of (name,obj)

hmf.transfer.Transfer.parameter_info

Transfer.**parameter_info**()

hmf.transfer.Transfer.update

Transfer.**update**(***kwargs*)
Update parameters of the framework with kwargs.

Attributes

cosmo	Cosmographic object (astropy.cosmology.FLRW object), with custom cosmology from *cosmo_p*
cosmo_model	**Parameter**: The basis for the cosmology – see astropy documentation. Can be a custom

Continued

Table 8.21 – continued from previous page

cosmo_params	**Parameter**: Parameters for the cosmology that deviate from the base cosmology passed.
delta_k	Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$
dlnk	**Parameter**: Step-size of log wavenumbers
growth	The instantiated growth model
growth_factor	The growth factor
growth_model	**Parameter**: The model to use to calculate the growth function/growth rate.
growth_params	**Parameter**: Relevant parameters of the *growth_model*.
k	Wavenumbers, [h/Mpc]
lnk_max	**Parameter**: Maximum (natural) log wavenumber, *k* [h/Mpc].
lnk_min	**Parameter**: Minimum (natural) log wavenumber, *k* [h/Mpc].
mean_density0	Mean density of universe at z=0, [Msun h^2 / Mpc**3]
n	**Parameter**: Spectral index of fluctations
nonlinear_delta_k	Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$
nonlinear_power	Non-linear log power [units Mpc^3/h^3]
parameter_values	Dictionary of all parameters and their current values
power	Normalised log power spectrum [units Mpc^3/h^3]
sigma_8	**Parameter**: RMS linear density fluctations in spheres of radius 8 Mpc/h
takahashi	**Parameter**: Whether to use updated HALOFIT coefficients from Takahashi+12
transfer	The instantiated transfer model
transfer_model	**Parameter**: Defines which transfer function model to use.
transfer_params	**Parameter**: Relevant parameters of the *transfer_model*.
z	**Parameter**: Redshift.

hmf.transfer.Transfer.cosmo

Transfer.**cosmo**
: Cosmographic object (`astropy.cosmology.FLRW` object), with custom cosmology from *cosmo_params* applied.

hmf.transfer.Transfer.cosmo_model

Transfer.**cosmo_model**
: **Parameter**: The basis for the cosmology – see astropy documentation. Can be a custom subclass. Defaults to Planck15.

  Type instance of *astropy.cosmology.FLRW* subclass

hmf.transfer.Transfer.cosmo_params

Transfer.**cosmo_params**
: **Parameter**: Parameters for the cosmology that deviate from the base cosmology passed. This is useful for repeated updates of a single parameter (leaving others the same). Default is the empty dict. The parameters passed must match the allowed parameters of *cosmo_model*. For the basic class this is

  Tcmb0 Temperature of the CMB at z=0

  Neff Number of massless neutrino species

  M_nu Mass of neutrino species (list)

  H0 The hubble constant at z=0

Om0 The normalised matter density at z=0

Type dict

hmf.transfer.Transfer.delta_k

Transfer.**delta_k**
Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$

hmf.transfer.Transfer.dlnk

Transfer.**dlnk**
Parameter: Step-size of log wavenumbers

Type float

hmf.transfer.Transfer.growth

Transfer.**growth**
The instantiated growth model

hmf.transfer.Transfer.growth_factor

Transfer.**growth_factor**
The growth factor

hmf.transfer.Transfer.growth_model

Transfer.**growth_model**
Parameter: The model to use to calculate the growth function/growth rate.

Type str or *hmf.growth_factor.GrowthFactor* subclass

hmf.transfer.Transfer.growth_params

Transfer.**growth_params**
Parameter: Relevant parameters of the *growth_model*.

Type dict

hmf.transfer.Transfer.k

Transfer.**k**
Wavenumbers, [h/Mpc]

hmf.transfer.Transfer.lnk_max

Transfer.**lnk_max**
Parameter: Maximum (natural) log wavenumber, k [h/Mpc].

Type float

hmf.transfer.Transfer.lnk_min

Transfer.**lnk_min**
Parameter: Minimum (natural) log wavenumber, k [h/Mpc].

Type float

hmf.transfer.Transfer.mean_density0

Transfer.**mean_density0**
Mean density of universe at z=0, [Msun h^2 / Mpc**3]

hmf.transfer.Transfer.n

Transfer.**n**
Parameter: Spectral index of fluctations

Must be greater than -3 and less than 4.

Type float

hmf.transfer.Transfer.nonlinear_delta_k

Transfer.**nonlinear_delta_k**
Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$

hmf.transfer.Transfer.nonlinear_power

Transfer.**nonlinear_power**
Non-linear log power [units Mpc^3/h^3]

Non-linear corrections come from HALOFIT.

hmf.transfer.Transfer.parameter_values

Transfer.**parameter_values**
Dictionary of all parameters and their current values

hmf.transfer.Transfer.power

Transfer.**power**
Normalised log power spectrum [units Mpc^3/h^3]

hmf.transfer.Transfer.sigma_8

Transfer.**sigma_8**
Parameter: RMS linear density fluctations in spheres of radius 8 Mpc/h

Type float

hmf.transfer.Transfer.takahashi

Transfer.**takahashi**
Parameter: Whether to use updated HALOFIT coefficients from Takahashi+12

Type bool

hmf.transfer.Transfer.transfer

Transfer.**transfer**
The instantiated transfer model

hmf.transfer.Transfer.transfer_model

Transfer.**transfer_model**
Parameter: Defines which transfer function model to use.

Built-in available models are found in the *hmf.transfer_models* module. Default is CAMB if installed, otherwise EH.

Type str or *hmf.transfer_models.TransferComponent* subclass, optional

hmf.transfer.Transfer.transfer_params

Transfer.**transfer_params**
Parameter: Relevant parameters of the *transfer_model*.

Type dict

hmf.transfer.Transfer.z

Transfer.**z**
Parameter: Redshift.

Must be greater than 0.

Type float

8.3.4 hmf.halofit

Implements the HALOFIT (Smith+2003, Takahashi+2012) method.

This code was heavily influenced by the *HaloFit* class from the *chomp* python package by Christopher Morrison, Ryan Scranton and Michael Schneider (https://code.google.com/p/chomp/). It has been modified to improve its integration with this package.

Functions

halofit(k, delta_k, sigma_8, z[, cosmo, ...])	Implementation of HALOFIT (Smith+2003).

hmf.halofit.halofit

hmf.halofit.**halofit**(*k*, *delta_k*, *sigma_8*, *z*, *cosmo=None*, *takahashi=True*)

Implementation of HALOFIT (Smith+2003).

Parameters **k** : array_like

Wavenumbers [h/Mpc].

delta_k : array_like

Dimensionless power (linear) at *k*.

sigma_8 : float

RMS linear density fluctuations in spheres of radius 8 Mpc/h at z=0.

z : float

Redshift

cosmo : *hmf.cosmo.Cosmology* instance, optional

An instance of either the *Cosmology* class provided in the *hmf* package, or any subclass of *FLRW* from *astropy*. Defualt is the default cosmology from the *hmf.cosmo* module.

takahashi : bool, optional

Whether to use updated parameters from Takahashi+2012. Otherwise use original from Smith+2003.

Returns **nonlinear_delta_k** : array_like

Dimensionless power at *k*, with nonlinear corrections applied.

Classes

csm	alias of Cosmology

hmf.halofit.csm

hmf.halofit.**csm**

alias of Cosmology

8.3.5 hmf.growth_factor

Module defining the growth factor *Component*.

The primary class, *GrowthFactor*, executes a full numerical calculation in standard flat LambdaCDM. Simplifications which may be more efficient, or extensions to alternate cosmologies, may be implemented.

Classes

Cmpt	alias of Component
GenMFGrowth(cosmo, **model_parameters)	Port of growth factor routines found in the genmf code.
GrowthFactor(cosmo, **model_parameters)	General class for a growth factor calculation.

hmf.growth_factor.Cmpt

hmf.growth_factor.**Cmpt**
alias of Component

hmf.growth_factor.GenMFGrowth

class hmf.growth_factor.**GenMFGrowth**(*cosmo*, ***model_parameters*)
Port of growth factor routines found in the genmf code.

Parameters **cosmo** : astropy.cosmology.FLRW instance

Cosmological model.

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, available parameters are as follows.To see their default values, check the _defaults class attribute.

dz Step-size for redshift integration

zmax Maximum redshift to integrate to. Only used for *growth_factor_fn()*.

Methods

__init__(cosmo, **model_parameters)	
growth_factor(z)	The growth factor, $d(a) = D^+(a)/D^+(a=1)$.
growth_factor_fn([zmin, inverse])	Return the growth factor as a callable function.
growth_rate(z)	Growth rate, dln(d)/dln(a) from Hamilton 2000 eq.
growth_rate_fn([zmin])	Growth rate, dln(d)/dln(a) from Hamilton 2000 eq.

hmf.growth_factor.GenMFGrowth.__init__

GenMFGrowth.**__init__**(*cosmo*, ***model_parameters*)

hmf.growth_factor.GenMFGrowth.growth_factor

GenMFGrowth.**growth_factor**(*z*)
The growth factor, $d(a) = D^+(a)/D^+(a=1)$.

This uses an approximation only valid in closed or flat cosmologies, ported from genmf.

Parameters **z** : array_like

Redshift.

Returns **gf** : array_like

The growth factor at z.

hmf.growth_factor.GenMFGrowth.growth_factor_fn

GenMFGrowth.**growth_factor_fn**(*zmin=0.0*, *inverse=False*)

Return the growth factor as a callable function.

Parameters **zmin** : float, optional

The minimum redshift of the function. Default 0.0

inverse: bool, optional

Whether to return the inverse relationship [z(g)]. Default False.

Returns callable

The normalised growth factor as a function of redshift, or redshift as a function of growth factor if `inverse` is True.

hmf.growth_factor.GenMFGrowth.growth_rate

GenMFGrowth.**growth_rate**(*z*)

Growth rate, dln(d)/dln(a) from Hamilton 2000 eq. 4

Parameters **z** : float

The redshift

hmf.growth_factor.GenMFGrowth.growth_rate_fn

GenMFGrowth.**growth_rate_fn**(*zmin=0*)

Growth rate, dln(d)/dln(a) from Hamilton 2000 eq. 4, as callable.

Parameters **zmin** : float, optional

The minimum redshift of the function. Default 0.0

Returns callable

The normalised growth rate as a function of redshift.

hmf.growth_factor.GrowthFactor

class hmf.growth_factor.**GrowthFactor**(*cosmo*, ***model_parameters*)

General class for a growth factor calculation.

Each of the methods in this class is defined using a numerical integral, following *[R21]*.

Parameters **cosmo** : `astropy.cosmology.FLRW` instance

Cosmological model.

****model_parameters** : unpack-dict

Parameters specific to this model. In this case, available parameters are as follows.To see their default values, check the `_defaults` class attribute.

dlna Step-size in log-space for scale-factor integration

amin Minimum scale-factor (i.e.e maximum redshift) to integrate to. Only used for *growth_factor_fn()*.

References

[R21]

Methods

__init__(cosmo, **model_parameters)	
growth_factor(z)	Calculate $d(a) = D^+(a)/D^+(a=1)$, from Lukic et.
growth_factor_fn([zmin, inverse])	Calculate $d(a) = D^+(a)/D^+(a=1)$, from Lukic et.
growth_rate(z)	Growth rate, dln(d)/dln(a) from Hamilton 2000 eq.
growth_rate_fn([zmin])	Growth rate, dln(d)/dln(a) from Hamilton 2000 eq.

hmf.growth_factor.GrowthFactor.__init__

GrowthFactor.**__init__**(*cosmo*, ***model_parameters*)

hmf.growth_factor.GrowthFactor.growth_factor

GrowthFactor.**growth_factor**(*z*)

Calculate $d(a) = D^+(a)/D^+(a=1)$, from Lukic et. al. 2007, eq. 7.

Parameters **z** : float

The redshift

Returns float

The normalised growth factor.

hmf.growth_factor.GrowthFactor.growth_factor_fn

GrowthFactor.**growth_factor_fn**(*zmin=0.0*, *inverse=False*)

Calculate $d(a) = D^+(a)/D^+(a=1)$, from Lukic et. al. 2007, eq. 7.

Returns a function G(z).

Parameters **zmin** : float, optional

The minimum redshift of the function. Default 0.0

inverse: bool, optional

Whether to return the inverse relationship [z(g)]. Default False.

Returns callable

The normalised growth factor as a function of redshift, or redshift as a function of growth factor if `inverse` is True.

hmf.growth_factor.GrowthFactor.growth_rate

GrowthFactor.**growth_rate**(*z*)
Growth rate, dln(d)/dln(a) from Hamilton 2000 eq. 4

Parameters **z** : float

The redshift

hmf.growth_factor.GrowthFactor.growth_rate_fn

GrowthFactor.**growth_rate_fn**(*zmin=0*)
Growth rate, dln(d)/dln(a) from Hamilton 2000 eq. 4, as callable.

Parameters **zmin** : float, optional

The minimum redshift of the function. Default 0.0

Returns callable

The normalised growth rate as a function of redshift.

8.3.6 hmf.filters

A module containing various smoothing filter Component models, including the popular top-hat in real space.

Classes

Filter(k, power, **model_parameters)	Base class for Filter components.
Gaussian(k, power, **model_parameters)	Gaussian window function.
SharpK(k, power, **model_parameters)	Fourier-space top-hat window function
SharpKEllipsoid(k, power, **model_parameters)	Fourier-space top-hat window function with ellipsoidal correction (see Schn
TopHat(k, power, **model_parameters)	Real-space top-hat window function.

hmf.filters.Filter

class hmf.filters.**Filter**(*k*, *power*, ***model_parameters*)
Base class for Filter components.

Filters handle the calculation of the mass variance from the power spectrum, via a window function. Subclasses of Filer implement specific window functions.

The general design is to specify all quantities in terms of length scales, rather than equivalent masses, but conversion methods are provided.

Any explicit subclass need only specify *k_space*, *mass_to_radius*, *radius_to_mass* and *dw_dlnkr*, with *dlnr_dlnm* optional if the mass assignment is unconventional.

Parameters **k** : array_like

Wavenumbers at which the power spectrum is defined.

power : array_like

The power spectrum at *k*.

****model_parameters** : unpacked-dict

As for any *hmf._framework.Component* subclass, any particular parameters of the model may be passed to the constructor. Allowed parameters are found in the `_defaults` attribute.

Notes

Besides the raw filter function itself, two quantities are of primary interest: firstly the mass variance (see *sigma()*), which appears in many cosmological applications, and secondly its logarithmic derivative with mass, which appears in the Press-Schechter formalism for the halo mass function.

To remain extensible and general, the methodology in this class is to calculate the latter quantity as

$$\frac{d\ln\sigma}{d\ln m} = \frac{1}{2}\frac{d\ln\sigma^2}{d\ln R}\frac{d\ln R}{d\ln m}.$$

Each of the quantities on the right can be separately calculated, improving extensibility.

The factor $\frac{d\ln R}{d\ln m}$ is typically 1/3, but this is not necessarily the case for window functions of arbitrary shape.

Methods

__init__(k, power, **model_parameters)	
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order, rk])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

hmf.filters.Filter.__init__

`Filter.`**`__init__`**(*k*, *power*, ***model_parameters*)

hmf.filters.Filter.dlnr_dlnm

`Filter.`**`dlnr_dlnm`**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.filters.Filter.dlnss_dlnm

Filter.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.filters.Filter.dlnss_dlnr

Filter.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in *dlnr_dlnm()*), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.filters.Filter.dw_dlnkr

Filter.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.filters.Filter.k_space

Filter.**k_space**(*kr*)

Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.filters.Filter.mass_to_radius

Filter.**mass_to_radius**(*m*, *rho_mean*)

Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.Filter.nu

Filter.**nu**(*r*, *delta_c=1.686*)

Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.filters.Filter.radius_to_mass

Filter.**radius_to_mass**(*r*, *rho_mean*)

Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.Filter.real_space

`Filter.`**`real_space`**(*R*, *r*)

Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.filters.Filter.sigma

`Filter.`**`sigma`**(*r*, *order=0*, *rk=None*)

Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

hmf.filters.Gaussian

class `hmf.filters.`**`Gaussian`**(*k*, *power*, ***model_parameters*)

Gaussian window function.

This class is based on *Filter*, which can be consulted for details of how to instantiate it.

Notes

The real-space filter is

$$F(r) = \frac{\exp(-r^2/2R^2)}{R^3(2\pi)^{3/2}}$$

for a filter scale of *R*.

The fourier-transform of the filter is

$$W(x = kR) = \exp(-x^2/2).$$

The mass-assignment is

$$m(R) = R^3(2\pi)^{3/2}\bar{\rho},$$

and the derivative of the window function is

$$\frac{dW}{d\ln x}(x = kR) = -xW(x).$$

Methods

__init__(k, power, **model_parameters)	
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order, rk])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

hmf.filters.Gaussian.__init__

Gaussian.**__init__**(*k*, *power*, ***model_parameters*)

hmf.filters.Gaussian.dlnr_dlnm

Gaussian.**dlnr_dlnm**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.filters.Gaussian.dlnss_dlnm

Gaussian.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.filters.Gaussian.dlnss_dlnr

Gaussian.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in *dlnr_dlnm()*), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.filters.Gaussian.dw_dlnkr

Gaussian.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.filters.Gaussian.k_space

Gaussian.**k_space**(*kr*)

Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.filters.Gaussian.mass_to_radius

Gaussian.**mass_to_radius**(*m*, *rho_mean*)

Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.Gaussian.nu

Gaussian.**nu**(*r*, *delta_c=1.686*)

Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.filters.Gaussian.radius_to_mass

Gaussian.**radius_to_mass**(*r*, *rho_mean*)

Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.Gaussian.real_space

Gaussian.**real_space**(*R*, *r*)

Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.filters.Gaussian.sigma

Gaussian.**sigma**(*r*, *order=0*, *rk=None*)

Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

hmf.filters.SharpK

class hmf.filters.**SharpK**(*k*, *power*, ***model_parameters*)

Fourier-space top-hat window function

This class is based on *Filter*, which can be consulted for details of how to instantiate it.

Notes

The real-space filter is

$$F(r) = \frac{\sin(r/R) - (r/R)\cos(r/R)}{2\pi^2 r^3},$$

for a filter scale of *R*.

The fourier-transform of the filter is

$$W(x = kR) = H(kR - 1)$$

where *H* is the Heaviside step-function. The mass-assignment is

$$m(R) = \frac{4\pi}{3}[cR]^3\bar{\rho},$$

where *c* is a free parameter, typically c~2.5. The derivative of the window function is

$$\frac{dW}{d\ln x}(x = kR) = \delta_D(x - 1),$$

where δ_D is the Dirac delta. Furthermore, the derivative of the mass variance takes a particularly simple form in this filter:

$$\frac{d\ln\sigma^2}{d\ln R} = -\frac{P(1/R)}{2\pi^2\sigma^2(R)R^3}.$$

Methods

__init__(k, power, **model_parameters)	
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

hmf.filters.SharpK.__init__

SharpK.**__init__**(*k*, *power*, ***model_parameters*)

hmf.filters.SharpK.dlnr_dlnm

SharpK.**dlnr_dlnm**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.filters.SharpK.dlnss_dlnm

SharpK.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.filters.SharpK.dlnss_dlnr

SharpK.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in *dlnr_dlnm()*), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.filters.SharpK.dw_dlnkr

SharpK.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.filters.SharpK.k_space

SharpK.**k_space**(*kr*)

Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.filters.SharpK.mass_to_radius

SharpK.**mass_to_radius**(*m*, *rho_mean*)

Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.SharpK.nu

SharpK.**nu**(*r*, *delta_c=1.686*)

Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.filters.SharpK.radius_to_mass

SharpK.**radius_to_mass**(*r*, *rho_mean*)

Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.SharpK.real_space

`SharpK.`**`real_space`**(*R*, *r*)

Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.filters.SharpK.sigma

`SharpK.`**`sigma`**(*r*, *order=0*)

Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

hmf.filters.SharpKEllipsoid

class `hmf.filters.`**`SharpKEllipsoid`**(*k*, *power*, ***model_parameters*)

Fourier-space top-hat window function with ellipsoidal correction (see Schneider, Smith, Reed 2013).

Refer to *`Filter`* for more details.

Methods

__init__(k, power, **model_parameters)	
a3(r)	
a3a1(e, p)	The short to long axis ratio of an ellipsoid given its ellipticity and
a3a2(e, p)	The short to medium axis ratio of an ellipsoid given its ellipticity and
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
em(xm)	The average ellipticity of a patch as a function of peak tensor
gamma(r)	Bardeen et al.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
pm(xm)	The average prolateness of a patch as a function of peak tensor
r_a3(rmin, rmax)	
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.
xi(pm, em)	
xm(g, v)	Peak of the distribution of x, where x is the sum of the eigenvalues of the inertia tensor

hmf.filters.SharpKEllipsoid.__init__

SharpKEllipsoid.**__init__**(*k*, *power*, ***model_parameters*)

hmf.filters.SharpKEllipsoid.a3

SharpKEllipsoid.**a3**(*r*)

hmf.filters.SharpKEllipsoid.a3a1

SharpKEllipsoid.**a3a1**(*e*, *p*)

The short to long axis ratio of an ellipsoid given its ellipticity and prolateness

hmf.filters.SharpKEllipsoid.a3a2

SharpKEllipsoid.**a3a2**(*e*, *p*)

The short to medium axis ratio of an ellipsoid given its ellipticity and prolateness

hmf.filters.SharpKEllipsoid.dlnr_dlnm

SharpKEllipsoid.**dlnr_dlnm**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.filters.SharpKEllipsoid.dlnss_dlnm

SharpKEllipsoid.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.filters.SharpKEllipsoid.dlnss_dlnr

SharpKEllipsoid.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in *dlnr_dlnm()*), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.filters.SharpKEllipsoid.dw_dlnkr

SharpKEllipsoid.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.filters.SharpKEllipsoid.em

SharpKEllipsoid.**em**(*xm*)
The average ellipticity of a patch as a function of peak tensor

hmf.filters.SharpKEllipsoid.gamma

SharpKEllipsoid.**gamma**(*r*)
Bardeen et al. 1986 equation 6.17

hmf.filters.SharpKEllipsoid.k_space

SharpKEllipsoid.**k_space**(*kr*)
Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.filters.SharpKEllipsoid.mass_to_radius

SharpKEllipsoid.**mass_to_radius**(*m*, *rho_mean*)
Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.SharpKEllipsoid.nu

SharpKEllipsoid.**nu**(*r*, *delta_c=1.686*)
Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.filters.SharpKEllipsoid.pm

SharpKEllipsoid.**pm**(*xm*)
The average prolateness of a patch as a function of peak tensor

hmf.filters.SharpKEllipsoid.r_a3

SharpKEllipsoid.**r_a3**(*rmin*, *rmax*)

hmf.filters.SharpKEllipsoid.radius_to_mass

SharpKEllipsoid.**radius_to_mass**(*r*, *rho_mean*)
Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.SharpKEllipsoid.real_space

SharpKEllipsoid.**real_space**(*R*, *r*)
Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.filters.SharpKEllipsoid.sigma

SharpKEllipsoid.**sigma**(*r*, *order=0*)
Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

hmf.filters.SharpKEllipsoid.xi

`SharpKEllipsoid.`**`xi`**(*pm*, *em*)

hmf.filters.SharpKEllipsoid.xm

`SharpKEllipsoid.`**`xm`**(*g*, *v*)

Peak of the distribution of x, where x is the sum of the eigenvalues of the inertia tensor (?) of an ellipsoidal peak, divided by the second spectral moment.

Equation A6. in Schneider et al. 2013

hmf.filters.TopHat

class `hmf.filters.`**`TopHat`**(*k*, *power*, ***model_parameters*)

Real-space top-hat window function.

This class is based on *`Filter`*, which can be consulted for details of how to instantiate it.

Notes

This filter specifically implements the real-space filter:

$$F(r) = H(R - r)$$

for a filter size of *R*, where *H* is the Heaviside step-function.

Its fourier-transform is

$$W(x = kR) = 3\frac{\sin x - x\cos x}{x^3}.$$

Furthermore the mass-assignment is

$$m(R) = \frac{4\pi}{3} R^3 \bar{\rho}$$

and the derivative of the window function is

$$\frac{dW}{d\ln x}(x = kR) = \frac{1}{x^3}[9x\cos x + 3(x^2 - 3)\sin x].$$

Methods

__init__(k, power, **model_parameters)	
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order, rk])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

hmf.filters.TopHat.__init__

TopHat.**__init__**(*k*, *power*, ***model_parameters*)

hmf.filters.TopHat.dlnr_dlnm

TopHat.**dlnr_dlnm**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.filters.TopHat.dlnss_dlnm

TopHat.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.filters.TopHat.dlnss_dlnr

TopHat.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in `dlnr_dlnm()`), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.filters.TopHat.dw_dlnkr

TopHat.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.filters.TopHat.k_space

TopHat.**k_space**(*kr*)

Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.filters.TopHat.mass_to_radius

TopHat.**mass_to_radius**(*m*, *rho_mean*)

Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.TopHat.nu

TopHat.**nu**(*r*, *delta_c=1.686*)

Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.filters.TopHat.radius_to_mass

TopHat.**radius_to_mass**(*r*, *rho_mean*)

Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.filters.TopHat.real_space

TopHat.**real_space**(*R*, *r*)

Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.filters.TopHat.sigma

TopHat.**sigma**(*r*, *order=0*, *rk=None*)

Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

8.3.7 hmf.fitting_functions

A module defining several mass function fits.

Each fit is taken from the literature. If there are others out there that are not listed here, please advise via GitHub.

Classes

Angulo(nu2[, m, z, n_eff, delta_halo, ...])	Angulo mass function fit.
AnguloBound(nu2[, m, z, n_eff, delta_halo, ...])	Angulo mass function fit.
Behroozi(**model_parameters)	Behroozi mass function fit [R24].
Bhattacharya(**kwargs)	Bhattacharya mass function fit.
Courtin(nu2[, m, z, n_eff, delta_halo, ...])	Courtin mass function fit.
Crocce(*args, **kwargs)	Crocce mass function fit.
FittingFunction(nu2[, m, z, n_eff, ...])	Base-class for a halo mass function fit.
Ishiyama(nu2[, m, z, n_eff, delta_halo, ...])	Ishiyama mass function fit.
Jenkins(nu2[, m, z, n_eff, delta_halo, ...])	Jenkins mass function fit.
Manera(nu2[, m, z, n_eff, delta_halo, ...])	Manera mass function fit.
PS(nu2[, m, z, n_eff, delta_halo, cosmo, ...])	Press-Schechter mass function fit.
Peacock(nu2[, m, z, n_eff, delta_halo, ...])	Peacock mass function fit.
Pillepich(nu2[, m, z, n_eff, delta_halo, ...])	Pillepich mass function fit.
Reed03(nu2[, m, z, n_eff, delta_halo, ...])	Reed03 mass function fit.
Reed07(nu2[, m, z, n_eff, delta_halo, ...])	Reed07 mass function fit.

Continued on next page

Table 8.33 – continued from previous page

SMT(nu2[, m, z, n_eff, delta_halo, cosmo, ...])	Sheth-Mo-Tormen mass function fit.
ST(nu2[, m, z, n_eff, delta_halo, cosmo, ...])	Alias of *SMT*
Tinker08(**model_parameters)	Tinker08 mass function fit.
Tinker10(**model_parameters)	Tinker10 mass function fit.
Warren(nu2[, m, z, n_eff, delta_halo, ...])	Warren mass function fit.
Watson(nu2[, m, z, n_eff, delta_halo, ...])	Watson mass function fit.
Watson_FoF(nu2[, m, z, n_eff, delta_halo, ...])	Watson FoF mass function fit.

hmf.fitting_functions.Angulo

class hmf.fitting_functions.**Angulo**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Angulo mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using <model>._defaults

Notes

The Angulo *[R1]* form is:

$$f_{\rm Ang}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + c\right]\exp\left(\frac{d}{\sigma^2}\right)$$

References

[R1]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Angulo.__init__

Angulo.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Angulo.cutmask

Angulo.**cutmask**

hmf.fitting_functions.Angulo.fsigma

Angulo.**fsigma**

hmf.fitting_functions.Angulo.req_dhalo

Angulo.**req_dhalo = False**

hmf.fitting_functions.Angulo.req_mass

Angulo.**req_mass** = True

hmf.fitting_functions.Angulo.req_neff

Angulo.**req_neff** = False

hmf.fitting_functions.Angulo.req_omz

Angulo.**req_omz** = False

hmf.fitting_functions.Angulo.req_sigma

Angulo.**req_sigma** = True

hmf.fitting_functions.Angulo.req_z

Angulo.**req_z** = False

hmf.fitting_functions.AnguloBound

class hmf.fitting_functions.**AnguloBound**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Angulo mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift z. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Angulo *[R2]* form is:

$$f_{\rm Ang}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + c\right]\exp\left(\frac{d}{\sigma^2}\right)$$

References

[R2]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.AnguloBound.__init__

AnguloBound.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.AnguloBound.cutmask

AnguloBound.**cutmask**

hmf.fitting_functions.AnguloBound.fsigma

AnguloBound.**fsigma**

hmf.fitting_functions.AnguloBound.req_dhalo

AnguloBound.**req_dhalo** = **False**

hmf.fitting_functions.AnguloBound.req_mass

AnguloBound.**req_mass** = **True**

hmf.fitting_functions.AnguloBound.req_neff

AnguloBound.**req_neff** = **False**

hmf.fitting_functions.AnguloBound.req_omz

AnguloBound.**req_omz** = **False**

hmf.fitting_functions.AnguloBound.req_sigma

AnguloBound.**req_sigma** = **True**

hmf.fitting_functions.AnguloBound.req_z

AnguloBound.**req_z** = **False**

hmf.fitting_functions.Behroozi

class hmf.fitting_functions.**Behroozi**(***model_parameters*)

Behroozi mass function fit *[R3]*.

This is an empirical modification to the *Tinker08* fit, to improve accuracy at high redshift.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

References

[R3]

Methods

__init__(**model_parameters)

hmf.fitting_functions.Behroozi.__init__

Behroozi.**__init__**(***model_parameters*)

Attributes

cutmask
delta_virs
fsigma
normalise
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z
terminate

hmf.fitting_functions.Behroozi.cutmask

Behroozi.**cutmask**

hmf.fitting_functions.Behroozi.delta_virs

Behroozi.**delta_virs = array([200, 300, 400, 600, 800, 1200, 1600, 2400, 3200])**

hmf.fitting_functions.Behroozi.fsigma

Behroozi.**fsigma**

hmf.fitting_functions.Behroozi.normalise

Behroozi.**normalise**

hmf.fitting_functions.Behroozi.req_dhalo

Behroozi.**req_dhalo** = True

hmf.fitting_functions.Behroozi.req_mass

Behroozi.**req_mass** = False

hmf.fitting_functions.Behroozi.req_neff

Behroozi.**req_neff** = False

hmf.fitting_functions.Behroozi.req_omz

Behroozi.**req_omz** = False

hmf.fitting_functions.Behroozi.req_sigma

Behroozi.**req_sigma** = True

hmf.fitting_functions.Behroozi.req_z

Behroozi.**req_z** = True

hmf.fitting_functions.Behroozi.terminate

Behroozi.**terminate** = True

hmf.fitting_functions.Bhattacharya

class hmf.fitting_functions.**Bhattacharya**(***kwargs*)

Bhattacharya mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to m

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Bhattacharya *[R4]* form is:

$$f_{\rm Btc}(\sigma) = f_{\rm SMT}(\sigma)(\nu\sqrt{a})^q$$

References

[R4]

Methods

__init__(**kwargs)
norm()

hmf.fitting_functions.Bhattacharya.__init__

`Bhattacharya.`**`__init__`**`(`*`**kwargs`*`)`

hmf.fitting_functions.Bhattacharya.norm

`Bhattacharya.`**`norm`**`()`

Attributes

cutmask	
fsigma	Calculate $f(\sigma)$ for Bhattacharya form.
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.Bhattacharya.cutmask

Bhattacharya.**cutmask**

hmf.fitting_functions.Bhattacharya.fsigma

Bhattacharya.**fsigma**

Calculate $f(\sigma)$ for Bhattacharya form.

Bhattacharya, S., et al., May 2011. ApJ 732 (2), 122. http://labs.adsabs.harvard.edu/ui/abs/2011ApJ...732..122B

Note: valid for $10^{11.8} M_\odot < M < 10^{15.5} M_\odot$

Returns **vfv** : array_like, len=len(pert.M)

The function :math:‘f(sigma)equiv

u f(

u)‘ defined on `pert.M`

hmf.fitting_functions.Bhattacharya.req_dhalo

Bhattacharya.**req_dhalo** = False

hmf.fitting_functions.Bhattacharya.req_mass

Bhattacharya.**req_mass** = True

hmf.fitting_functions.Bhattacharya.req_neff

Bhattacharya.**req_neff** = False

hmf.fitting_functions.Bhattacharya.req_omz

Bhattacharya.**req_omz** = False

hmf.fitting_functions.Bhattacharya.req_sigma

Bhattacharya.**req_sigma** = **False**

hmf.fitting_functions.Bhattacharya.req_z

Bhattacharya.**req_z** = **True**

hmf.fitting_functions.Courtin

class hmf.fitting_functions.**Courtin**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Courtin mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using <model>._defaults

Notes

The Courtin *[R5]* form is:

$$f_{\rm Ctn}(\sigma) = A\sqrt{2a/\pi}\nu \exp(-a\nu^2/2)(1+(a\nu^2)^{-p})$$

References

[R5]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])	
norm()	

hmf.fitting_functions.Courtin.__init__

Courtin.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

hmf.fitting_functions.Courtin.norm

Courtin.**norm**()

Attributes

cutmask	
fsigma	
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.Courtin.cutmask

Courtin.**cutmask**

hmf.fitting_functions.Courtin.fsigma

Courtin.**fsigma**

hmf.fitting_functions.Courtin.req_dhalo

Courtin.**req_dhalo** = False

hmf.fitting_functions.Courtin.req_mass

Courtin.**req_mass** = False

hmf.fitting_functions.Courtin.req_neff

Courtin.**req_neff** = False

hmf.fitting_functions.Courtin.req_omz

Courtin.**req_omz** = False

hmf.fitting_functions.Courtin.req_sigma

Courtin.**req_sigma** = False

hmf.fitting_functions.Courtin.req_z

Courtin.**req_z** = False

hmf.fitting_functions.Crocce

class hmf.fitting_functions.**Crocce**(**args*, ***kwargs*)

Crocce mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Crocce *[R6]* form is:

$$f_{\rm Cro}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + c\right]\exp\left(\frac{d}{\sigma^2}\right)$$

References

[R6]

Methods

__init__(*args, **kwargs)

hmf.fitting_functions.Crocce.__init__

Crocce.**__init__**(**args*, ***kwargs*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Crocce.cutmask

Crocce.**cutmask**

hmf.fitting_functions.Crocce.fsigma

Crocce.**fsigma**

hmf.fitting_functions.Crocce.req_dhalo

Crocce.**req_dhalo = False**

hmf.fitting_functions.Crocce.req_mass

Crocce.**req_mass** = True

hmf.fitting_functions.Crocce.req_neff

Crocce.**req_neff** = False

hmf.fitting_functions.Crocce.req_omz

Crocce.**req_omz** = False

hmf.fitting_functions.Crocce.req_sigma

Crocce.**req_sigma** = True

hmf.fitting_functions.Crocce.req_z

Crocce.**req_z** = True

hmf.fitting_functions.FittingFunction

class hmf.fitting_functions.**FittingFunction**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Base-class for a halo mass function fit.

This class should not be called directly, rather use a subclass which is specific to a certain fitting formula. The only method necessary to define for any subclass is *fsigma*, as well as a dictionary of default parameters as a class variable *_defaults*. Model parameters defined here are accessed through the `params` instance attribute (and may be overridden at instantiation by the user). A subclass may optionally define a *cutmask* property, to override the default behaviour of returning True for the whole range.

In addition, several class attributes, *req_**, identify the required arguments for a given subclass. These must be set accordingly.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Examples

The following would be an example of defining the Sheth-Tormen mass function (which is already included), showing the basic idea of subclassing this class:

```
>>> class SMT(FittingFunction):
>>>     # Subclass requirements
>>>     req_sigma = False
>>>     req_z     = False
>>>
>>>     # Default parameters
>>>     _defaults = {"a":0.707, "p":0.3, "A":0.3222}
>>>
>>>     @property
>>>     def fsigma(self):
>>>         A = self.params['A']
>>>         a = self.params["a"]
>>>         p = self.params['p']
>>>
>>>         return (A * np.sqrt(2.0 * a / np.pi) * self.nu * np.exp(-(a * self.nu2) / 2.0)
>>>                 * (1 + (1.0 / (a * self.nu2)) ** p))
```

In that example, we did not specify *cutmask*.

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.FittingFunction.__init__

`FittingFunction.`**`__init__`**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask	A logical mask array specifying which elements of *fsigma* are within the fitted range.
fsigma	The function $f(\sigma) \equiv \nu f(\nu)$.
req_dhalo	Whether *delta_halo* is required for this subclass
req_mass	Whether *m* is required for this subclass
req_neff	Whether *n_eff* is required for this subclass
req_omz	Whether *omegam_z* is required for this subclass
req_sigma	Whether *sigma* (via *delta_c*) is required for this subclass
req_z	Whether *z* is required for this subclass

hmf.fitting_functions.FittingFunction.cutmask

FittingFunction.**cutmask**
A logical mask array specifying which elements of *fsigma* are within the fitted range.

hmf.fitting_functions.FittingFunction.fsigma

FittingFunction.**fsigma**
The function $f(\sigma) \equiv \nu f(\nu)$.

hmf.fitting_functions.FittingFunction.req_dhalo

FittingFunction.**req_dhalo = False**
Whether *delta_halo* is required for this subclass

hmf.fitting_functions.FittingFunction.req_mass

FittingFunction.**req_mass = False**
Whether *m* is required for this subclass

hmf.fitting_functions.FittingFunction.req_neff

FittingFunction.**req_neff = False**
Whether *n_eff* is required for this subclass

hmf.fitting_functions.FittingFunction.req_omz

FittingFunction.**req_omz = False**
Whether *omegam_z* is required for this subclass

hmf.fitting_functions.FittingFunction.req_sigma

FittingFunction.**req_sigma = True**
Whether *sigma* (via *delta_c*) is required for this subclass

hmf.fitting_functions.FittingFunction.req_z

FittingFunction.**req_z = True**
Whether z is required for this subclass

hmf.fitting_functions.Ishiyama

class hmf.fitting_functions.**Ishiyama**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Ishiyama mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift z. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using <model>._defaults

Notes

The Ishiyama *[R7]* form is:

$$f_{\rm Ishiyama}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + 1\right]\exp(\frac{d}{\sigma^2})$$

References

[R7]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Ishiyama.__init__

Ishiyama.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Ishiyama.cutmask

Ishiyama.**cutmask**

hmf.fitting_functions.Ishiyama.fsigma

Ishiyama.**fsigma**

hmf.fitting_functions.Ishiyama.req_dhalo

Ishiyama.**req_dhalo** = False

hmf.fitting_functions.Ishiyama.req_mass

Ishiyama.**req_mass** = True

hmf.fitting_functions.Ishiyama.req_neff

Ishiyama.**req_neff** = False

hmf.fitting_functions.Ishiyama.req_omz

Ishiyama.**req_omz** = **False**

hmf.fitting_functions.Ishiyama.req_sigma

Ishiyama.**req_sigma** = **True**

hmf.fitting_functions.Ishiyama.req_z

Ishiyama.**req_z** = **False**

hmf.fitting_functions.Jenkins

class hmf.fitting_functions.**Jenkins**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Jenkins mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using <model>._defaults

Notes

The Jenkins *[R8]* form is:

$$f_{\rm Jenkins}(\sigma) = A \exp\left(-\left|\ln \sigma^{-1} + b\right|^c\right)$$

References

[R8]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Jenkins.__init__

Jenkins.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Jenkins.cutmask

Jenkins.**cutmask**

hmf.fitting_functions.Jenkins.fsigma

Jenkins.**fsigma**

hmf.fitting_functions.Jenkins.req_dhalo

Jenkins.**req_dhalo** = False

hmf.fitting_functions.Jenkins.req_mass

Jenkins.**req_mass** = False

hmf.fitting_functions.Jenkins.req_neff

Jenkins.**req_neff** = **False**

hmf.fitting_functions.Jenkins.req_omz

Jenkins.**req_omz** = **False**

hmf.fitting_functions.Jenkins.req_sigma

Jenkins.**req_sigma** = **True**

hmf.fitting_functions.Jenkins.req_z

Jenkins.**req_z** = **False**

hmf.fitting_functions.Manera

class hmf.fitting_functions.**Manera**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Manera mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Manera *[R9]* form is:

$$f_{\rm Man}(\sigma) = A\sqrt{2a/\pi}\nu \exp(-a\nu^2/2)(1+(a\nu^2)^{-p})$$

References

[R9]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])	
norm()	

hmf.fitting_functions.Manera.__init__

Manera.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

hmf.fitting_functions.Manera.norm

Manera.**norm**()

Attributes

cutmask	A logical mask array specifying which elements of *fsigma* are within the fitted range.
fsigma	
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.Manera.cutmask

Manera.**cutmask**

A logical mask array specifying which elements of *fsigma* are within the fitted range.

hmf.fitting_functions.Manera.fsigma

Manera.**fsigma**

hmf.fitting_functions.Manera.req_dhalo

Manera.**req_dhalo** = False

hmf.fitting_functions.Manera.req_mass

Manera.**req_mass** = False

hmf.fitting_functions.Manera.req_neff

Manera.**req_neff** = False

hmf.fitting_functions.Manera.req_omz

Manera.**req_omz** = False

hmf.fitting_functions.Manera.req_sigma

Manera.**req_sigma** = False

hmf.fitting_functions.Manera.req_z

Manera.**req_z** = False

hmf.fitting_functions.PS

class hmf.fitting_functions.**PS**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Press-Schechter mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift z. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Press-Schechter *[R10]* form is:

$$f_{\rm PS}(\sigma) = \sqrt{\frac{2}{\pi}}\nu \exp(-0.5\nu^2)$$

References

[R10]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.PS.__init__

`PS.`**`__init__`**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask	A logical mask array specifying which elements of *fsigma* are within the fitted range.
fsigma	
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.PS.cutmask

PS.**cutmask**
A logical mask array specifying which elements of *fsigma* are within the fitted range.

hmf.fitting_functions.PS.fsigma

PS.**fsigma**

hmf.fitting_functions.PS.req_dhalo

PS.**req_dhalo** = False

hmf.fitting_functions.PS.req_mass

PS.**req_mass** = False

hmf.fitting_functions.PS.req_neff

PS.**req_neff** = False

hmf.fitting_functions.PS.req_omz

PS.**req_omz** = False

hmf.fitting_functions.PS.req_sigma

PS.**req_sigma** = False

hmf.fitting_functions.PS.req_z

PS.**req_z** = False

hmf.fitting_functions.Peacock

class hmf.fitting_functions.**Peacock**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Peacock mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to m

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift z. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Peacock *[R11]* form is:

$$f_{\rm Pck}(\sigma) = \nu \exp(-c\nu^2)(2cd\nu + ba\nu^{b-1})/d^2$$

References

[R11]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Peacock.__init__

`Peacock.`**`__init__`**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Peacock.cutmask

Peacock.**cutmask**

hmf.fitting_functions.Peacock.fsigma

Peacock.**fsigma**

hmf.fitting_functions.Peacock.req_dhalo

Peacock.**req_dhalo** = False

hmf.fitting_functions.Peacock.req_mass

Peacock.**req_mass** = True

hmf.fitting_functions.Peacock.req_neff

Peacock.**req_neff** = False

hmf.fitting_functions.Peacock.req_omz

Peacock.**req_omz** = False

hmf.fitting_functions.Peacock.req_sigma

Peacock.**req_sigma** = True

hmf.fitting_functions.Peacock.req_z

Peacock.**req_z** = False

hmf.fitting_functions.Pillepich

class hmf.fitting_functions.**Pillepich**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Pillepich mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Pillepich *[R12]* form is:

$$f_{\rm Pillepich}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + c\right]\exp\left(\frac{d}{\sigma^2}\right)$$

References

[R12]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Pillepich.__init__

Pillepich.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Pillepich.cutmask

Pillepich.**cutmask**

hmf.fitting_functions.Pillepich.fsigma

Pillepich.**fsigma**

hmf.fitting_functions.Pillepich.req_dhalo

Pillepich.**req_dhalo** = False

hmf.fitting_functions.Pillepich.req_mass

Pillepich.**req_mass** = True

hmf.fitting_functions.Pillepich.req_neff

Pillepich.**req_neff** = False

hmf.fitting_functions.Pillepich.req_omz

Pillepich.**req_omz** = False

hmf.fitting_functions.Pillepich.req_sigma

Pillepich.**req_sigma** = True

hmf.fitting_functions.Pillepich.req_z

Pillepich.**req_z** = **False**

hmf.fitting_functions.Reed03

class hmf.fitting_functions.**Reed03**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Reed03 mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using <model>._defaults

Notes

The Reed03 *[R13]* form is:

$$f_{\rm R03}(\sigma) = f_{\rm SMT}(\sigma)\exp\left(-\frac{c}{\sigma\cosh^5(2\sigma)}\right)$$

References

[R13]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])	
norm()	

hmf.fitting_functions.Reed03.__init__

Reed03.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

hmf.fitting_functions.Reed03.norm

Reed03.**norm**()

Attributes

cutmask	
fsigma	
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.Reed03.cutmask

Reed03.**cutmask**

hmf.fitting_functions.Reed03.fsigma

Reed03.**fsigma**

hmf.fitting_functions.Reed03.req_dhalo

Reed03.**req_dhalo** = False

hmf.fitting_functions.Reed03.req_mass

Reed03.**req_mass** = False

hmf.fitting_functions.Reed03.req_neff

Reed03.**req_neff** = False

hmf.fitting_functions.Reed03.req_omz

Reed03.**req_omz** = False

hmf.fitting_functions.Reed03.req_sigma

Reed03.**req_sigma** = True

hmf.fitting_functions.Reed03.req_z

Reed03.**req_z** = False

hmf.fitting_functions.Reed07

class hmf.fitting_functions.**Reed07**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Reed07 mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Reed07 *[R14]* form is:

$$f_{\rm R07}(\sigma) = A\sqrt{2a/\pi}\left[1 + (\frac{1}{a\nu^2})^p + 0.6G_1 + 0.4G_2\right]\nu\exp\left(-ca\nu^2/2 - \frac{0.03\nu^{0.6}}{(n_{\rm eff}+3)^2}\right)$$

References

[R14]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Reed07.__init__

Reed07.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Reed07.cutmask

Reed07.**cutmask**

hmf.fitting_functions.Reed07.fsigma

Reed07.**fsigma**

hmf.fitting_functions.Reed07.req_dhalo

Reed07.**req_dhalo** = **False**

hmf.fitting_functions.Reed07.req_mass

Reed07.**req_mass** = **False**

hmf.fitting_functions.Reed07.req_neff

Reed07.**req_neff** = **True**

hmf.fitting_functions.Reed07.req_omz

Reed07.**req_omz** = **False**

hmf.fitting_functions.Reed07.req_sigma

Reed07.**req_sigma** = **True**

hmf.fitting_functions.Reed07.req_z

Reed07.**req_z** = **False**

hmf.fitting_functions.SMT

class hmf.fitting_functions.**SMT**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Sheth-Mo-Tormen mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift z. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Sheth-Mo-Tormen *[R15]* form is:

$$f_{\rm SMT}(\sigma) = A\sqrt{2a/\pi}\nu \exp(-a\nu^2/2)(1 + (a\nu^2)^{-p})$$

References

[R15]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])	
norm()	

hmf.fitting_functions.SMT.__init__

SMT.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

hmf.fitting_functions.SMT.norm

SMT.**norm**()

Attributes

cutmask	A logical mask array specifying which elements of *fsigma* are within the fitted range.
fsigma	
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.SMT.cutmask

SMT.**cutmask**
A logical mask array specifying which elements of *fsigma* are within the fitted range.

hmf.fitting_functions.SMT.fsigma

SMT.**fsigma**

hmf.fitting_functions.SMT.req_dhalo

SMT.**req_dhalo** = False

hmf.fitting_functions.SMT.req_mass

SMT.**req_mass** = False

hmf.fitting_functions.SMT.req_neff

SMT.**req_neff** = False

hmf.fitting_functions.SMT.req_omz

SMT.**req_omz** = False

hmf.fitting_functions.SMT.req_sigma

SMT.**req_sigma** = False

hmf.fitting_functions.SMT.req_z

SMT.**req_z** = False

hmf.fitting_functions.ST

class hmf.fitting_functions.**ST**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)
Alias of *SMT*

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])
norm()

hmf.fitting_functions.ST.__init__

ST.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

hmf.fitting_functions.ST.norm

ST.**norm**()

Attributes

cutmask	A logical mask array specifying which elements of *fsigma* are within the fitted range.
fsigma	
req_dhalo	
req_mass	
req_neff	
req_omz	
req_sigma	
req_z	

hmf.fitting_functions.ST.cutmask

ST.**cutmask**
A logical mask array specifying which elements of *fsigma* are within the fitted range.

hmf.fitting_functions.ST.fsigma

ST.**fsigma**

hmf.fitting_functions.ST.req_dhalo

ST.**req_dhalo** = False

hmf.fitting_functions.ST.req_mass

ST.**req_mass** = False

hmf.fitting_functions.ST.req_neff

ST.**req_neff** = False

hmf.fitting_functions.ST.req_omz

ST.**req_omz** = False

hmf.fitting_functions.ST.req_sigma

ST.**req_sigma** = False

hmf.fitting_functions.ST.req_z

ST.**req_z** = False

hmf.fitting_functions.Tinker08

class hmf.fitting_functions.**Tinker08**(***model_parameters*)

Tinker08 mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using <model>._defaults

Notes

The Tinker08 *[R16]* form is:

$$f_{\rm Tkr}(\sigma) = A\left(\frac{\sigma}{b}^{-a} + 1\right)\exp(-c/\sigma^2)$$

References

[R16]

Methods

__init__(**model_parameters)

hmf.fitting_functions.Tinker08.__init__

Tinker08.**__init__**(***model_parameters*)

Attributes

cutmask
delta_virs
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Tinker08.cutmask

Tinker08.**cutmask**

hmf.fitting_functions.Tinker08.delta_virs

Tinker08.**delta_virs** = array([200, 300, 400, 600, 800, 1200, 1600, 2400, 3200])

hmf.fitting_functions.Tinker08.fsigma

Tinker08.**fsigma**

hmf.fitting_functions.Tinker08.req_dhalo

Tinker08.**req_dhalo** = True

hmf.fitting_functions.Tinker08.req_mass

Tinker08.**req_mass** = False

hmf.fitting_functions.Tinker08.req_neff

Tinker08.**req_neff** = False

hmf.fitting_functions.Tinker08.req_omz

Tinker08.**req_omz** = False

hmf.fitting_functions.Tinker08.req_sigma

Tinker08.**req_sigma** = True

hmf.fitting_functions.Tinker08.req_z

Tinker08.**req_z** = True

hmf.fitting_functions.Tinker10

class hmf.fitting_functions.**Tinker10**(***model_parameters*)

Tinker10 mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Tinker10 *[R17]* form is:

$$f_{\rm Tkr}(\sigma) = (1 + (\beta\nu)^{-2\phi})\nu^{2\eta+1}\exp(-\gamma\nu^2/2)$$

References

[R17]

Methods

__init__(**model_parameters)

hmf.fitting_functions.Tinker10.__init__

Tinker10.**__init__**(***model_parameters*)

Attributes

cutmask
delta_virs
fsigma
normalise
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z
terminate

hmf.fitting_functions.Tinker10.cutmask

Tinker10.**cutmask**

hmf.fitting_functions.Tinker10.delta_virs

Tinker10.**delta_virs** = **array([200, 300, 400, 600, 800, 1200, 1600, 2400, 3200])**

hmf.fitting_functions.Tinker10.fsigma

Tinker10.**fsigma**

hmf.fitting_functions.Tinker10.normalise

Tinker10.**normalise**

hmf.fitting_functions.Tinker10.req_dhalo

Tinker10.**req_dhalo** = True

hmf.fitting_functions.Tinker10.req_mass

Tinker10.**req_mass** = False

hmf.fitting_functions.Tinker10.req_neff

Tinker10.**req_neff** = False

hmf.fitting_functions.Tinker10.req_omz

Tinker10.**req_omz** = False

hmf.fitting_functions.Tinker10.req_sigma

Tinker10.**req_sigma** = True

hmf.fitting_functions.Tinker10.req_z

Tinker10.**req_z** = True

hmf.fitting_functions.Tinker10.terminate

Tinker10.**terminate** = True

hmf.fitting_functions.Warren

class hmf.fitting_functions.**Warren**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Warren mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Warren *[R18]* form is:

$$f_{\rm Warren}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + c\right]\exp\left(\frac{d}{\sigma^2}\right)$$

References

[R18]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Warren.__init__

`Warren.__init__`(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Warren.cutmask

Warren.**cutmask**

hmf.fitting_functions.Warren.fsigma

Warren.**fsigma**

hmf.fitting_functions.Warren.req_dhalo

Warren.**req_dhalo** = False

hmf.fitting_functions.Warren.req_mass

Warren.**req_mass** = True

hmf.fitting_functions.Warren.req_neff

Warren.**req_neff** = False

hmf.fitting_functions.Warren.req_omz

Warren.**req_omz** = False

hmf.fitting_functions.Warren.req_sigma

Warren.**req_sigma** = True

hmf.fitting_functions.Warren.req_z

Warren.**req_z** = False

hmf.fitting_functions.Watson

class hmf.fitting_functions.**Watson**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Watson mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the `cosmo.Cosmology` class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Watson *[R19]* form is:

$$f_{\rm WatS}(\sigma) = \Gamma A \left((\frac{\beta}{\sigma}^{\alpha} + 1 \right) \exp(-\gamma/\sigma^2)$$

References

[R19]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])	
gamma()	Calculate Γ for the Watson fit.

hmf.fitting_functions.Watson.__init__

Watson.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

hmf.fitting_functions.Watson.gamma

Watson.**gamma**()
Calculate Γ for the Watson fit.

Attributes

cutmask
fsigma
req_cosmo
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Watson.cutmask

Watson.**cutmask**

hmf.fitting_functions.Watson.fsigma

Watson.**fsigma**

hmf.fitting_functions.Watson.req_cosmo

Watson.**req_cosmo** = True

hmf.fitting_functions.Watson.req_dhalo

Watson.**req_dhalo** = True

hmf.fitting_functions.Watson.req_mass

Watson.**req_mass** = False

hmf.fitting_functions.Watson.req_neff

Watson.**req_neff** = **False**

hmf.fitting_functions.Watson.req_omz

Watson.**req_omz** = **True**

hmf.fitting_functions.Watson.req_sigma

Watson.**req_sigma** = **True**

hmf.fitting_functions.Watson.req_z

Watson.**req_z** = **True**

hmf.fitting_functions.Watson_FoF

class hmf.fitting_functions.**Watson_FoF**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Watson FoF mass function fit.

For details on attributes, see documentation for *FittingFunction*.

Parameters **nu2** : array_like

A vector of peak-heights, δ_c^2/σ^2 corresponding to *m*

m : array_like, optional

A vector of halo masses [units M_sun/h]. Only necessary if *req_mass* is True. Typically provides limits of applicability. Must correspond to *nu2*.

z : float, optional

The redshift. Only required if *req_z* is True, in which case the default is 0.

n_eff : array_like, optional

The effective spectral index at *m*. Only required if *req_neff* is True.

delta_halo : float, optional

The overdensity of the halo w.r.t. the mean density of the universe. Only required if *req_dhalo* is True, in which case the default is 200.0

cosmo : *hmf.cosmo.Cosmology* instance, optional

A cosmology. Default is the default provided by the cosmo.Cosmology class. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

omegam_z : float, optional

A value for the mean matter density at the given redshift *z*. Either *omegam_z* or *cosmo* is required if *req_omz* is True. If both are passed, omegam_z takes precedence.

****model_parameters** : unpacked-dictionary

These parameters are model-specific. For any model, list the available parameters (and their defaults) using `<model>._defaults`

Notes

The Watson FoF *[R20]* form is:

$$f_{\rm WatF}(\sigma) = A\left[\left(\frac{e}{\sigma}\right)^b + c\right]\exp\left(\frac{d}{\sigma^2}\right)$$

References

[R20]

Methods

__init__(nu2[, m, z, n_eff, delta_halo, ...])

hmf.fitting_functions.Watson_FoF.__init__

Watson_FoF.**__init__**(*nu2*, *m=None*, *z=0*, *n_eff=None*, *delta_halo=200*, *cosmo=None*, *omegam_z=None*, *delta_c=1.686*, ***model_parameters*)

Attributes

cutmask
fsigma
req_dhalo
req_mass
req_neff
req_omz
req_sigma
req_z

hmf.fitting_functions.Watson_FoF.cutmask

Watson_FoF.**cutmask**

hmf.fitting_functions.Watson_FoF.fsigma

Watson_FoF.**fsigma**

hmf.fitting_functions.Watson_FoF.req_dhalo

Watson_FoF.**req_dhalo** = False

hmf.fitting_functions.Watson_FoF.req_mass

Watson_FoF.**req_mass** = False

hmf.fitting_functions.Watson_FoF.req_neff

Watson_FoF.**req_neff** = False

hmf.fitting_functions.Watson_FoF.req_omz

Watson_FoF.**req_omz** = False

hmf.fitting_functions.Watson_FoF.req_sigma

Watson_FoF.**req_sigma** = True

hmf.fitting_functions.Watson_FoF.req_z

Watson_FoF.**req_z** = False

8.3.8 hmf.hmf

The primary module for user-interaction with the `hmf` package.

The module contains a single class, *MassFunction*, which wraps almost all the functionality of `hmf` in an easy-to-use way.

Functions

cached_property(*parents)	A robust property caching decorator.
get_model(name, mod, **kwargs)	Returns an instance of `name` from the module `mod`, with given params.
int_gtm(M, dndm[, mass_density])	Cumulatively integrate dn/dm.
*issubclass_*(arg1, arg2)	Determine if a class is a subclass of a second class.
minimize(fun, x0[, args, method, jac, hess, ...])	Minimization of scalar function of one or more variables.
parameter(f)	A simple cached property which acts more like an input value.

hmf.hmf.cached_property

hmf.hmf.**cached_property**(**parents*)

A robust property caching decorator.

This decorator only works when used with the entire system here....

Usage:: class CachedClass(Cache):

@cached_property("parent_parameter") def amethod(self):

...calculations... return final_value

@cached_property("amethod") def a_child_method(self): #method dependent on amethod

final_value = 3*self.amethod return final_value

This code will calculate `amethod` on the first call, but return the cached value on all subsequent calls. If any parent parameter is modified, the calculation will be re-performed.

hmf.hmf.get_model

`hmf.hmf.`**`get_model`**(*name*, *mod*, ***kwargs*)

Returns an instance of `name` from the module `mod`, with given params.

Parameters **name** : str

The class name of the appropriate model

mod : str

The module name of the appropriate module

****kwargs :**

Any parameters for the instantiated model (including model parameters)

hmf.hmf.int_gtm

`hmf.hmf.`**`int_gtm`**(*M*, *dndm*, *mass_density=False*)

Cumulatively integrate dn/dm.

Parameters **M** : array_like

Array of masses.

dndm : array_like

Array of dn/dm (corresponding to M)

mass_density : bool, *False*

Whether to calculate mass density (or number density).

Returns **ngtm** : array_like

Cumulative integral of dndm.

Examples

Using a simple power-law mass function:

```
>>> import numpy as np
>>> m = np.logspace(10,18,500)
>>> dndm = m**-2
>>> ngtm = hmf_integral_gtm(m,dndm)
>>> np.allclose(ngtm,1/m) #1/m is the analytic integral to infinity.
True
```

The function always integrates to m=1e18, and extrapolates with a spline if data not provided:

```
>>> m = np.logspace(10,12,500)
>>> dndm = m**-2
>>> ngtm = hmf_integral_gtm(m,dndm)
>>> np.allclose(ngtm,1/m) #1/m is the analytic integral to infinity.
True
```

hmf.hmf.issubclass

hmf.hmf.**issubclass_**(*arg1*, *arg2*)

Determine if a class is a subclass of a second class.

*issubclass_* is equivalent to the Python built-in `issubclass`, except that it returns False instead of raising a TypeError is one of the arguments is not a class.

Parameters **arg1** : class

Input class. True is returned if *arg1* is a subclass of *arg2*.

arg2 : class or tuple of classes.

Input class. If a tuple of classes, True is returned if *arg1* is a subclass of any of the tuple elements.

Returns **out** : bool

Whether *arg1* is a subclass of *arg2* or not.

See also:

`issubsctype`, `issubdtype`, `issctype`

Examples

```
>>> np.issubclass_(np.int32, np.int)
True
>>> np.issubclass_(np.int32, np.float)
False
```

hmf.hmf.minimize

hmf.hmf.**minimize**(*fun*, *x0*, *args=()*, *method='BFGS'*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options=None*)

Minimization of scalar function of one or more variables.

New in version 0.11.0.

Parameters **fun** : callable

Objective function.

x0 : ndarray

Initial guess.

args : tuple, optional

Extra arguments passed to the objective function and its derivatives (Jacobian, Hessian).

method : str, optional

Type of solver. Should be one of

- 'Nelder-Mead'
- 'Powell'
- 'CG'
- 'BFGS'
- 'Newton-CG'
- 'Anneal'
- 'L-BFGS-B'
- 'TNC'
- 'COBYLA'
- 'SLSQP'
- 'dogleg'
- 'trust-ncg'

jac : bool or callable, optional

Jacobian of objective function. Only for CG, BFGS, Newton-CG, dogleg, trust-ncg. If *jac* is a Boolean and is True, *fun* is assumed to return the value of Jacobian along with the objective function. If False, the Jacobian will be estimated numerically. *jac* can also be a callable returning the Jacobian of the objective. In this case, it must accept the same arguments as *fun*.

hess, hessp : callable, optional

Hessian of objective function or Hessian of objective function times an arbitrary vector p. Only for Newton-CG, dogleg, trust-ncg. Only one of *hessp* or *hess* needs to be given. If *hess* is provided, then *hessp* will be ignored. If neither *hess* nor *hessp* is provided, then the hessian product will be approximated using finite differences on *jac*. *hessp* must compute the Hessian times an arbitrary vector.

bounds : sequence, optional

Bounds for variables (only for L-BFGS-B, TNC and SLSQP). `(min, max)` pairs for each element in `x`, defining the bounds on that parameter. Use None for one of `min` or `max` when there is no bound in that direction.

constraints : dict or sequence of dict, optional

Constraints definition (only for COBYLA and SLSQP). Each constraint is defined in a dictionary with fields:

type [str] Constraint type: 'eq' for equality, 'ineq' for inequality.

fun [callable] The function defining the constraint.

jac [callable, optional] The Jacobian of *fun* (only for SLSQP).

args [sequence, optional] Extra arguments to be passed to the function and Jacobian.

Equality constraint means that the constraint function result is to be zero whereas inequality means that it is to be non-negative. Note that COBYLA only supports inequality constraints.

tol : float, optional

Tolerance for termination. For detailed control, use solver-specific options.

options : dict, optional

A dictionary of solver options. All methods accept the following generic options:

maxiter [int] Maximum number of iterations to perform.

disp [bool] Set to True to print convergence messages.

For method-specific options, see *show_options('minimize', method)*.

callback : callable, optional

Called after each iteration, as `callback(xk)`, where `xk` is the current parameter vector.

Returns **res** : Result

The optimization result represented as a `Result` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See *Result* for a description of other attributes.

See also:

`minimize_scalar` Interface to minimization algorithms for scalar univariate functions.

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *BFGS*.

Unconstrained minimization

Method *Nelder-Mead* uses the Simplex algorithm *[R52]*, *[R53]*. This algorithm has been successful in many applications but other algorithms using the first and/or second derivatives information might be preferred for their better performances and robustness in general.

Method *Powell* is a modification of Powell's method *[R54]*, *[R55]* which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken.

Method *CG* uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method described in *[R56]* pp. 120-122. Only the first derivatives are used.

Method *BFGS* uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) *[R56]* pp. 136. It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as *hess_inv* in the Result object.

Method *Newton-CG* uses a Newton-CG algorithm *[R56]* pp. 168 (also known as the truncated Newton method). It uses a CG method to the compute the search direction. See also *TNC* method for a box-constrained minimization with a similar algorithm.

Method *Anneal* uses simulated annealing, which is a probabilistic metaheuristic algorithm for global optimization. It uses no derivative information from the function being optimized.

Method *dogleg* uses the dog-leg trust-region algorithm *[R56]* for unconstrained minimization. This algorithm requires the gradient and Hessian; furthermore the Hessian is required to be positive definite.

Method *trust-ncg* uses the Newton conjugate gradient trust-region algorithm *[R56]* for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector.

Constrained minimization

Method *L-BFGS-B* uses the L-BFGS-B algorithm *[R57]*, *[R58]* for bound constrained minimization.

Method *TNC* uses a truncated Newton algorithm *[R56]*, *[R59]* to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *Newton-CG* method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

Method *COBYLA* uses the Constrained Optimization BY Linear Approximation (COBYLA) method *[R60]*, [10], [11]. The algorithm is based on linear approximations to the objective function and each constraint. The method wraps a FORTRAN implementation of the algorithm.

Method *SLSQP* uses Sequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [12]. Note that the wrapper handles infinite values in bounds by converting them into large floating values.

References

[R52], *[R53]*, *[R54]*, *[R55]*, *[R56]*, *[R57]*, *[R58]*, *[R59]*, *[R60]*, [10], [11], [12]

Examples

Let us consider the problem of minimizing the Rosenbrock function. This function (and its respective derivatives) is implemented in *rosen* (resp. *rosen_der*, *rosen_hess*) in the *scipy.optimize*.

```
>>> from scipy.optimize import minimize, rosen, rosen_der
```

A simple application of the *Nelder-Mead* method is:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead')
>>> res.x
[ 1.  1.  1.  1.  1.]
```

Now using the *BFGS* algorithm, using the first derivative and a few options:

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 52
         Function evaluations: 64
         Gradient evaluations: 64
>>> res.x
[ 1.  1.  1.  1.  1.]
>>> print res.message
Optimization terminated successfully.
```

[10] Powell M J D. Direct search algorithms for optimization calculations. 1998. Acta Numerica 7: 287-336.
[11] Powell M J D. A view of algorithms for optimization without derivatives. 2007.Cambridge University Technical Report DAMTP 2007/NA03
[12] Kraft, D. A software package for sequential quadratic programming. 1988. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center – Institute for Flight Mechanics, Koln, Germany.

```
>>> res.hess
[[ 0.00749589  0.01255155  0.02396251  0.04750988  0.09495377]
 [ 0.01255155  0.02510441  0.04794055  0.09502834  0.18996269]
 [ 0.02396251  0.04794055  0.09631614  0.19092151  0.38165151]
 [ 0.04750988  0.09502834  0.19092151  0.38341252  0.7664427 ]
 [ 0.09495377  0.18996269  0.38165151  0.7664427   1.53713523]]
```

Next, consider a minimization problem with several constraints (namely Example 16.4 from *[R56]*). The objective function is:

```
>>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2
```

There are three constraints defined as:

```
>>> cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
...         {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
...         {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})
```

And variables must be positive, hence the following bounds:

```
>>> bnds = ((0, None), (0, None))
```

The optimization problem is solved using the SLSQP method as:

```
>>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
...                constraints=cons)
```

It should converge to the theoretical solution (1.4 ,1.7).

hmf.hmf.parameter

hmf.hmf.**parameter**(*f*)

A simple cached property which acts more like an input value.

This cached property is intended to be used on values that are passed in `__init__`, and can possibly be reset later. It provides the opportunity for complex setters, and also the ability to update dependent properties whenever the value is modified.

Usage:: @set_property("amethod") def parameter(self,val):

if isinstance(int,val): return val

else: raise ValueError("parameter must be an integer")

@cached_property() def amethod(self):

return 3*self.parameter

Note that the definition of the setter merely returns the value to be set, it doesn't set it to any particular instance attribute. The decorator automatically sets `self.__parameter = val` and defines the get method accordingly

Classes

Filter(k, power, **model_parameters)	Base class for Filter components.
MassFunction([Mmin, Mmax, dlog10m, ...])	An object containing all relevant quantities for the mass function.
TopHat(k, power, **model_parameters)	Real-space top-hat window function.
spline	alias of `InterpolatedUnivariateSpline`

hmf.hmf.Filter

class hmf.hmf.**Filter**(*k*, *power*, ***model_parameters*)

Base class for Filter components.

Filters handle the calculation of the mass variance from the power spectrum, via a window function. Subclasses of `Filer` implement specific window functions.

The general design is to specify all quantities in terms of length scales, rather than equivalent masses, but conversion methods are provided.

Any explicit subclass need only specify *k_space*, *mass_to_radius*, *radius_to_mass* and *dw_dlnkr*, with *dlnr_dlnm* optional if the mass assignment is unconventional.

Parameters **k** : array_like

Wavenumbers at which the power spectrum is defined.

power : array_like

The power spectrum at *k*.

****model_parameters** : unpacked-dict

As for any *hmf._framework.Component* subclass, any particular parameters of the model may be passed to the constructor. Allowed parameters are found in the `_defaults` attribute.

Notes

Besides the raw filter function itself, two quantities are of primary interest: firstly the mass variance (see *sigma()*), which appears in many cosmological applications, and secondly its logarithmic derivative with mass, which appears in the Press-Schechter formalism for the halo mass function.

To remain extensible and general, the methodology in this class is to calculate the latter quantity as

$$\frac{d\ln\sigma}{d\ln m} = \frac{1}{2}\frac{d\ln\sigma^2}{d\ln R}\frac{d\ln R}{d\ln m}.$$

Each of the quantities on the right can be separately calculated, improving extensibility.

The factor $\frac{d\ln R}{d\ln m}$ is typically 1/3, but this is not necessarily the case for window functions of arbitrary shape.

Methods

__init__(k, power, **model_parameters)	
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order, rk])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

hmf.hmf.Filter.__init__

Filter.**__init__**(*k*, *power*, ***model_parameters*)

hmf.hmf.Filter.dlnr_dlnm

Filter.**dlnr_dlnm**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.hmf.Filter.dlnss_dlnm

Filter.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.hmf.Filter.dlnss_dlnr

Filter.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in *dlnr_dlnm()*), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.hmf.Filter.dw_dlnkr

Filter.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.hmf.Filter.k_space

`Filter.`**`k_space`**(*kr*)

Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.hmf.Filter.mass_to_radius

`Filter.`**`mass_to_radius`**(*m*, *rho_mean*)

Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.hmf.Filter.nu

`Filter.`**`nu`**(*r*, *delta_c=1.686*)

Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.hmf.Filter.radius_to_mass

`Filter.`**`radius_to_mass`**(*r*, *rho_mean*)

Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.hmf.Filter.real_space

`Filter.`**`real_space`**(*R*, *r*)

Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.hmf.Filter.sigma

`Filter.`**`sigma`**(*r*, *order=0*, *rk=None*)

Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

hmf.hmf.MassFunction

class hmf.hmf.**MassFunction**(*Mmin=10*, *Mmax=15*, *dlog10m=0.01*, *hmf_model=<class 'hmf.fitting_functions.Tinker08'>*, *hmf_params=None*, *delta_h=200.0*, *delta_wrt='mean'*, *delta_c=1.686*, *filter_model=<class 'hmf.filters.TopHat'>*, *filter_params=None*, ***transfer_kwargs*)

An object containing all relevant quantities for the mass function.

The purpose of this class is to calculate many quantities associated with the dark matter halo mass function (HMF). The class is initialized to form a cosmology and takes in various options as to how to calculate all further quantities.

All required outputs are provided as `@property` attributes for ease of access.

Contains an update() method which can be passed arguments to update, in the most optimal manner. All output quantities are calculated only when needed (but stored after first calculation for quick access).

In addition to the parameters directly passed to this class, others are available which are passed on to its superclass. To read a standard documented list of (all) parameters, use `MassFunction.parameter_info()`. If you want to just see the plain list of available parameters, use `MassFunction.get_all_parameters()`.To see the actual defaults for each parameter, use `MassFunction.get_all_parameter_defaults()`.

Examples

Since all parameters have reasonable defaults, the most obvious thing to do is

```
>>> h = MassFunction()
>>> h.dndm
```

Many different parameters may be passed, both models and parameters of those models. For instance:

```
>>> h = MassFunction(z=1.0,Mmin=8,hmf_model="SMT")
>>> h.dndm
```

Once instantiated, changing parameters should be done through the *update()* method:

```
>>> h.update(z=2)
>>> h.dndm
```

Methods

__init__([Mmin, Mmax, dlog10m, hmf_model, ...])	
get_all_parameter_defaults()	Dictionary of all parameters and defaults
get_all_parameter_names()	Yield all parameter names in the class.
get_all_parameters()	Yield all parameters as tuples of (name,obj)

Continued on next page

Table 8.81 – continued from previous page

parameter_info()	
update(**kwargs)	Update parameters of the framework with kwargs.

hmf.hmf.MassFunction.__init__

MassFunction.**__init__**(*Mmin=10*, *Mmax=15*, *dlog10m=0.01*, *hmf_model=<class 'hmf.fitting_functions.Tinker08'>*, *hmf_params=None*, *delta_h=200.0*, *delta_wrt='mean'*, *delta_c=1.686*, *filter_model=<class 'hmf.filters.TopHat'>*, *filter_params=None*, ***transfer_kwargs*)

hmf.hmf.MassFunction.get_all_parameter_defaults

MassFunction.**get_all_parameter_defaults**()
Dictionary of all parameters and defaults

hmf.hmf.MassFunction.get_all_parameter_names

MassFunction.**get_all_parameter_names**()
Yield all parameter names in the class.

hmf.hmf.MassFunction.get_all_parameters

MassFunction.**get_all_parameters**()
Yield all parameters as tuples of (name,obj)

hmf.hmf.MassFunction.parameter_info

MassFunction.**parameter_info**()

hmf.hmf.MassFunction.update

MassFunction.**update**(***kwargs*)
Update parameters of the framework with kwargs.

Attributes

M	Masses (alias of m, deprecated)
Mmax	**Parameter**: Maximum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].
Mmin	**Parameter**: Minimum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].
cosmo	Cosmographic object (`astropy.cosmology.FLRW` object), with custom cosmology from *cosmo_p*
cosmo_model	**Parameter**: The basis for the cosmology – see astropy documentation. Can be a custom
cosmo_params	**Parameter**: Parameters for the cosmology that deviate from the base cosmology passed.
delta_c	**Parameter**: The critical overdensity for collapse, δ_c.
delta_h	**Parameter**: The overdensity for the halo definition, with respect to *delta_wrt*

Continued

Table 8.82 – continued from previous page

delta_halo	Overdensity of a halo w.r.t mean density
delta_k	Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$
delta_wrt	**Parameter**: Defines what the overdensity of a halo is with respect to, mean density
dlnk	**Parameter**: Step-size of log wavenumbers
dlog10m	**Parameter**: log10 interval between mass bins
dndlnm	The differential mass function in terms of natural log of *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]
dndlog10m	The differential mass function in terms of log of *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]
dndm	The number density of haloes, `len=len(m)` [units $h^4 M_\odot^{-1} Mpc^{-3}$]
filter	Instantiated model for filter/window functions.
filter_model	**Parameter**: A model for the window/filter function.
filter_params	**Parameter**: Model parameters for *filter_model*.
fsigma	The multiplicity function, $f(\sigma)$, for *hmf_model*.
growth	The instantiated growth model
growth_factor	The growth factor
growth_model	**Parameter**: The model to use to calculate the growth function/growth rate.
growth_params	**Parameter**: Relevant parameters of the *growth_model*.
hmf	Instantiated model for the hmf fitting function.
hmf_model	**Parameter**: A model to use as the fitting function $f(\sigma)$
hmf_params	**Parameter**: Model parameters for *hmf_model*.
how_big	Size of simulation volume in which to expect one halo of mass m, `len=len(m)` [units $Mpch^{-1}$]
k	Wavenumbers, [h/Mpc]
lnk_max	**Parameter**: Maximum (natural) log wavenumber, *k* [h/Mpc].
lnk_min	**Parameter**: Minimum (natural) log wavenumber, *k* [h/Mpc].
lnsigma	Natural log of inverse mass variance, `len=len(m)`
m	Masses
mass_nonlinear	The nonlinear mass, nu(Mstar) = 1.
mean_density	Mean density of universe at redshift z
mean_density0	Mean density of universe at z=0, [Msun h^2 / Mpc**3]
n	**Parameter**: Spectral index of fluctations
n_eff	Effective spectral index at scale of halo radius, `len=len(m)`
ngtm	The cumulative mass function above *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]
nonlinear_delta_k	Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$
nonlinear_power	Non-linear log power [units Mpc^3/h^3]
nu	The parameter $\nu = \left(\frac{\delta_c}{\sigma}\right)^2$, `len=len(m)`
parameter_values	Dictionary of all parameters and their current values
power	Normalised log power spectrum [units Mpc^3/h^3]
radii	The radii corresponding to the masses *m*.
rho_gtm	Mass density in haloes >*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]
rho_ltm	Mass density in haloes <*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]
sigma	The mass variance at *z*, `len=len(m)`
sigma_8	**Parameter**: RMS linear density fluctations in spheres of radius 8 Mpc/h
takahashi	**Parameter**: Whether to use updated HALOFIT coefficients from Takahashi+12
transfer	The instantiated transfer model
transfer_model	**Parameter**: Defines which transfer function model to use.
transfer_params	**Parameter**: Relevant parameters of the *transfer_model*.
z	**Parameter**: Redshift.

hmf.hmf.MassFunction.M

MassFunction.**M**
Masses (alias of m, deprecated)

hmf.hmf.MassFunction.Mmax

MassFunction.**Mmax**
Parameter: Maximum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].

Type float

hmf.hmf.MassFunction.Mmin

MassFunction.**Mmin**
Parameter: Minimum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].

Type float

hmf.hmf.MassFunction.cosmo

MassFunction.**cosmo**
Cosmographic object (`astropy.cosmology.FLRW` object), with custom cosmology from *cosmo_params* applied.

hmf.hmf.MassFunction.cosmo_model

MassFunction.**cosmo_model**
Parameter: The basis for the cosmology – see astropy documentation. Can be a custom subclass. Defaults to Planck15.

Type instance of *astropy.cosmology.FLRW* subclass

hmf.hmf.MassFunction.cosmo_params

MassFunction.**cosmo_params**
Parameter: Parameters for the cosmology that deviate from the base cosmology passed. This is useful for repeated updates of a single parameter (leaving others the same). Default is the empty dict. The parameters passed must match the allowed parameters of *cosmo_model*. For the basic class this is

Tcmb0 Temperature of the CMB at z=0

Neff Number of massless neutrino species

M_nu Mass of neutrino species (list)

H0 The hubble constant at z=0

Om0 The normalised matter density at z=0

Type dict

hmf.hmf.MassFunction.delta_c

MassFunction.**delta_c**
Parameter: The critical overdensity for collapse, δ_c.

Type float

hmf.hmf.MassFunction.delta_h

MassFunction.**delta_h**
Parameter: The overdensity for the halo definition, with respect to *delta_wrt*

Type float

hmf.hmf.MassFunction.delta_halo

MassFunction.**delta_halo**
Overdensity of a halo w.r.t mean density

hmf.hmf.MassFunction.delta_k

MassFunction.**delta_k**
Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$

hmf.hmf.MassFunction.delta_wrt

MassFunction.**delta_wrt**
Parameter: Defines what the overdensity of a halo is with respect to, mean density of the universe, or critical density.

Type str, {"mean", "crit"}

hmf.hmf.MassFunction.dlnk

MassFunction.**dlnk**
Parameter: Step-size of log wavenumbers

Type float

hmf.hmf.MassFunction.dlog10m

MassFunction.**dlog10m**
Parameter: log10 interval between mass bins

Type float

hmf.hmf.MassFunction.dndlnm

MassFunction.**dndlnm**
The differential mass function in terms of natural log of m, `len=len(m)` [units $h^3 Mpc^{-3}$]

hmf.hmf.MassFunction.dndlog10m

MassFunction.**dndlog10m**
The differential mass function in terms of log of *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]

hmf.hmf.MassFunction.dndm

MassFunction.**dndm**
The number density of haloes, `len=len(m)` [units $h^4 M_\odot^{-1} Mpc^{-3}$]

hmf.hmf.MassFunction.filter

MassFunction.**filter**
Instantiated model for filter/window functions.

hmf.hmf.MassFunction.filter_model

MassFunction.**filter_model**
Parameter: A model for the window/filter function.

Type str or *hmf.filters.Filter* subclass

hmf.hmf.MassFunction.filter_params

MassFunction.**filter_params**
Parameter: Model parameters for *filter_model*.

Type dict

hmf.hmf.MassFunction.fsigma

MassFunction.**fsigma**
The multiplicity function, $f(\sigma)$, for *hmf_model*. `len=len(m)`

hmf.hmf.MassFunction.growth

MassFunction.**growth**
The instantiated growth model

hmf.hmf.MassFunction.growth_factor

MassFunction.**growth_factor**
The growth factor

hmf.hmf.MassFunction.growth_model

MassFunction.**growth_model**
Parameter: The model to use to calculate the growth function/growth rate.

Type str or *hmf.growth_factor.GrowthFactor* subclass

hmf.hmf.MassFunction.growth_params

MassFunction.**growth_params**
Parameter: Relevant parameters of the *growth_model*.

Type dict

hmf.hmf.MassFunction.hmf

MassFunction.**hmf**
Instantiated model for the hmf fitting function.

hmf.hmf.MassFunction.hmf_model

MassFunction.**hmf_model**
Parameter: A model to use as the fitting function $f(\sigma)$

Type str or *hmf.fitting_functions.FittingFunction* subclass

hmf.hmf.MassFunction.hmf_params

MassFunction.**hmf_params**
Parameter: Model parameters for *hmf_model*.

Type dict

hmf.hmf.MassFunction.how_big

MassFunction.**how_big**
Size of simulation volume in which to expect one halo of mass m, `len=len(m)` [units $Mpch^{-1}$]

hmf.hmf.MassFunction.k

MassFunction.**k**
Wavenumbers, [h/Mpc]

hmf.hmf.MassFunction.lnk_max

MassFunction.**lnk_max**
Parameter: Maximum (natural) log wavenumber, k [h/Mpc].

Type float

hmf.hmf.MassFunction.lnk_min

MassFunction.**lnk_min**
Parameter: Minimum (natural) log wavenumber, k [h/Mpc].

Type float

hmf.hmf.MassFunction.lnsigma

MassFunction.**lnsigma**
Natural log of inverse mass variance, `len=len(m)`

hmf.hmf.MassFunction.m

MassFunction.**m**
Masses

hmf.hmf.MassFunction.mass_nonlinear

MassFunction.**mass_nonlinear**
The nonlinear mass, nu(Mstar) = 1.

hmf.hmf.MassFunction.mean_density

MassFunction.**mean_density**
Mean density of universe at redshift z

hmf.hmf.MassFunction.mean_density0

MassFunction.**mean_density0**
Mean density of universe at z=0, [Msun h^2 / Mpc**3]

hmf.hmf.MassFunction.n

MassFunction.**n**
Parameter: Spectral index of fluctations

Must be greater than -3 and less than 4.

Type float

hmf.hmf.MassFunction.n_eff

MassFunction.**n_eff**
Effective spectral index at scale of halo radius, `len=len(m)`

Notes

This function, and any derived quantities, can show small non-physical 'wiggles' at the 0.1% level, if too coarse a grid in ln(k) is used. If applications are sensitive at this level, please use a very fine k-space grid.

Uses eq. 42 in Lukic et. al 2007.

hmf.hmf.MassFunction.ngtm

MassFunction.**ngtm**
The cumulative mass function above *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]

In the case that *m* does not extend to sufficiently high masses, this routine will auto-generate `dndm` for an extended mass range.

In the case of the ff.Behroozi fit, it is impossible to auto-extend the mass range except by the power-law fit, thus one should be careful to supply appropriate mass ranges in this case.

hmf.hmf.MassFunction.nonlinear_delta_k

MassFunction.**nonlinear_delta_k**
Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$

hmf.hmf.MassFunction.nonlinear_power

MassFunction.**nonlinear_power**
Non-linear log power [units Mpc^3/h^3]

Non-linear corrections come from HALOFIT.

hmf.hmf.MassFunction.nu

MassFunction.**nu**
The parameter $\nu = \left(\frac{\delta_c}{\sigma}\right)^2$, `len=len(m)`

hmf.hmf.MassFunction.parameter_values

MassFunction.**parameter_values**
Dictionary of all parameters and their current values

hmf.hmf.MassFunction.power

MassFunction.**power**
Normalised log power spectrum [units Mpc^3/h^3]

hmf.hmf.MassFunction.radii

MassFunction.**radii**
The radii corresponding to the masses *m*.

hmf.hmf.MassFunction.rho_gtm

MassFunction.**rho_gtm**
Mass density in haloes >*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]

In the case that *m* does not extend to sufficiently high masses, this routine will auto-generate `dndm` for an extended mass range.

In the case of the ff.Behroozi fit, it is impossible to auto-extend the mass range except by the power-law fit, thus one should be careful to supply appropriate mass ranges in this case.

hmf.hmf.MassFunction.rho_ltm

MassFunction.**rho_ltm**
Mass density in haloes <*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]

Note: As of v1.6.2, this assumes that the entire mass density of halos is encoded by the `mean_density` parameter (ie. all mass is found in halos). This is not explicitly true of all fitting functions (eg. Warren), in which case the definition of this property is somewhat inconsistent, but will still work.

In the case that *m* does not extend to sufficiently high masses, this routine will auto-generate `dndm` for an extended mass range.

In the case of the ff.Behroozi fit, it is impossible to auto-extend the mass range except by the power-law fit, thus one should be careful to supply appropriate mass ranges in this case.

hmf.hmf.MassFunction.sigma

MassFunction.**sigma**
The mass variance at z, `len=len(m)`

hmf.hmf.MassFunction.sigma_8

MassFunction.**sigma_8**
Parameter: RMS linear density fluctations in spheres of radius 8 Mpc/h

Type float

hmf.hmf.MassFunction.takahashi

MassFunction.**takahashi**
Parameter: Whether to use updated HALOFIT coefficients from Takahashi+12

Type bool

hmf.hmf.MassFunction.transfer

MassFunction.**transfer**
The instantiated transfer model

hmf.hmf.MassFunction.transfer_model

MassFunction.**transfer_model**
Parameter: Defines which transfer function model to use.

Built-in available models are found in the *hmf.transfer_models* module. Default is CAMB if installed, otherwise EH.

Type str or *hmf.transfer_models.TransferComponent* subclass, optional

hmf.hmf.MassFunction.transfer_params

MassFunction.**transfer_params**
Parameter: Relevant parameters of the *transfer_model*.

Type dict

hmf.hmf.MassFunction.z

MassFunction.**z**
Parameter: Redshift.

Must be greater than 0.

Type float

hmf.hmf.TopHat

class hmf.hmf.**TopHat**(*k*, *power*, ***model_parameters*)
Real-space top-hat window function.

This class is based on *Filter*, which can be consulted for details of how to instantiate it.

Notes

This filter specifically implements the real-space filter:

$$F(r) = H(R - r)$$

for a filter size of *R*, where *H* is the Heaviside step-function.

Its fourier-transform is

$$W(x = kR) = 3\frac{\sin x - x\cos x}{x^3}.$$

Furthermore the mass-assignment is

$$m(R) = \frac{4\pi}{3}R^3\bar{\rho}$$

and the derivative of the window function is

$$\frac{dW}{d\ln x}(x = kR) = \frac{1}{x^3}[9x\cos x + 3(x^2 - 3)\sin x].$$

Methods

__init__(k, power, **model_parameters)	
dlnr_dlnm(r)	The derivative of log radius with log mass.
dlnss_dlnm(r)	The logarithmic slope of mass variance with mass.
dlnss_dlnr(r)	The derivative of the mass variance with radius.
dw_dlnkr(kr)	The derivative of the (fourier-transformed) filter with $\ln(kr)$.
k_space(kr)	Fourier-transform of the real-space filter.
mass_to_radius(m, rho_mean)	Return radius of a region of space given its mass.
nu(r[, delta_c])	Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.
radius_to_mass(r, rho_mean)	Return mass of a region of space given its radius
real_space(R, r)	Filter definition in real space.
sigma(r[, order, rk])	Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

hmf.hmf.TopHat.__init__

TopHat.**__init__**(*k*, *power*, ***model_parameters*)

hmf.hmf.TopHat.dlnr_dlnm

TopHat.**dlnr_dlnm**(*r*)

The derivative of log radius with log mass.

For the usual $m \propto r^3$ mass assignment, this is just 1/3.

Parameters **r** : array_like

Radii.

hmf.hmf.TopHat.dlnss_dlnm

TopHat.**dlnss_dlnm**(*r*)

The logarithmic slope of mass variance with mass.

This is an important quantity, and is used directly to calculate $\frac{dn}{dm}$.

Parameters **r** : array_like

Radii.

hmf.hmf.TopHat.dlnss_dlnr

TopHat.**dlnss_dlnr**(*r*)

The derivative of the mass variance with radius.

Parameters **r** : array_like

Radii

Returns **dlnss_dlnr** : array_like

The derivative of the the mass variance with radius.

Notes

Given a prescription for how radius grows with mass (typically with a log-slope of 1/3, and set in `dlnr_dlnm()`), this specifies the quantity $\frac{d\ln\sigma^2}{d\ln m}$.

The general formula is

$$\frac{d\ln\sigma^2}{d\ln R} = \frac{1}{\pi^2\sigma^2}\int_0^\infty W(kR)\frac{dW(kR)}{d\ln(kR)}P(k)k^2dk$$

hmf.hmf.TopHat.dw_dlnkr

TopHat.**dw_dlnkr**(*kr*)

The derivative of the (fourier-transformed) filter with $\ln(kr)$.

Parameters **kr** : array_like

Scale(s) at which the derivative is evaluated.

Notes

In terms of $\frac{dw^2}{dm}$, which is a commonly used quantity, this has the relationship

$$w\frac{dw}{d\ln r} = \frac{2}{r}\frac{dw^2}{dm}\frac{dm}{dr}.$$

hmf.hmf.TopHat.k_space

TopHat.**k_space**(*kr*)

Fourier-transform of the real-space filter.

Parameters **kr** : array_like

The scales at which to return the filter function

Returns **w** : array_like

The filter in fourier space, `len(kr)`

hmf.hmf.TopHat.mass_to_radius

TopHat.**mass_to_radius**(*m*, *rho_mean*)

Return radius of a region of space given its mass.

Parameters **m** : array_like

Masses

rho_mean : float

Mean density of the Universe.

Returns **r** : array_like

The corresponding radii to m

Notes

The units of *m* don't matter as long as they are consistent with *rho_mean*.

hmf.hmf.TopHat.nu

TopHat.**nu**(*r*, *delta_c=1.686*)

Peak height, $\frac{\delta_c^2}{\sigma^2(r)}$.

Parameters **r** : array_like

Radii

delta_c : float, optional

Critical overdensity for collapse.

hmf.hmf.TopHat.radius_to_mass

TopHat.**radius_to_mass**(*r*, *rho_mean*)

Return mass of a region of space given its radius

Parameters **r** : array_like

Radii

rho_mean : float

Mean density of the Universe.

Returns **m** : float or array of floats

The corresponding masses to r

Notes

The units of *r* don't matter as long as they are consistent with *rho_mean*.

hmf.hmf.TopHat.real_space

TopHat.**real_space**(*R*, *r*)

Filter definition in real space.

Parameters **R** : float

The smoothing scale

r : array_like

The radial co-ordinate

hmf.hmf.TopHat.sigma

TopHat.**sigma**(*r*, *order=0*, *rk=None*)
Calculate the nth-moment of the smoothed density field, $\sigma_n(r)$.

Note: This is not $\sigma_n^2(r)$!

Parameters **r** : float or array_like

The radii of the spheres at which to calculate the nth moment.

order : int, optional

The order of the moment. Default 0 corresponds to common mass variance.

Returns **sigma** : array_like

The square root of the moment at *r*.

Notes

The general definition for the nth-moment of the smoothed density field is (see Bardeen et al. 1986, Eq 4.6c)

$$\sigma_n^2(R) = \frac{1}{2\pi^2}\int_0^\infty dk\, k^{2(1+n)} P(k) W^2(kR)$$

hmf.hmf.spline

hmf.hmf.**spline**
alias of InterpolatedUnivariateSpline

8.3.9 hmf.wdm

Module containing Warm Dark Matter models.

This module contains both WDM Components (basic WDM models and also recalibrators for the HMF) and Frameworks (Transfer and MassFunction). The latter inject WDM modelling into the standard CDM Frameworks, and provide an example of how one would go about this for other alternative cosmologies.

Functions

cached_property(*parents)	A robust property caching decorator.
get_model(name, mod, **kwargs)	Returns an instance of name from the module mod, with given params.
parameter(f)	A simple cached property which acts more like an input value.

hmf.wdm.cached_property

hmf.wdm.**cached_property**(**parents*)
A robust property caching decorator.

This decorator only works when used with the entire system here....

Usage:: class CachedClass(Cache):

@cached_property("parent_parameter") def amethod(self):

...calculations... return final_value

@cached_property("amethod") def a_child_method(self): #method dependent on amethod

final_value = 3*self.amethod return final_value

This code will calculate `amethod` on the first call, but return the cached value on all subsequent calls. If any parent parameter is modified, the calculation will be re-performed.

hmf.wdm.get_model

`hmf.wdm.`**`get_model`**(*name*, *mod*, ***kwargs*)

Returns an instance of `name` from the module `mod`, with given params.

Parameters **name** : str

The class name of the appropriate model

mod : str

The module name of the appropriate module

****kwargs :**

Any parameters for the instantiated model (including model parameters)

hmf.wdm.parameter

`hmf.wdm.`**`parameter`**(*f*)

A simple cached property which acts more like an input value.

This cached property is intended to be used on values that are passed in `__init__`, and can possibly be reset later. It provides the opportunity for complex setters, and also the ability to update dependent properties whenever the value is modified.

Usage:: @set_property("amethod") def parameter(self,val):

if isinstance(int,val): return val

else: raise ValueError("parameter must be an integer")

@cached_property() def amethod(self):

return 3*self.parameter

Note that the definition of the setter merely returns the value to be set, it doesn't set it to any particular instance attribute. The decorator automatically sets `self.__parameter = val` and defines the get method accordingly

Classes

Bode01(mx, cosmo, z, **model_params)	
Component(**model_params)	Base class representing a component model.

Continued c

Table 8.85 – continued from previous page

Lovell14(m, dndm0, wdm, **model_parameters)	Lovell+2014 recalibration of the WDM HMF.
MassFunctionWDM([alter_dndm, alter_params])	A subclass of `hmf.MassFunction` that mixes in WDM capabilities.
Schneider12(m, dndm0, wdm, **model_parameters)	Schneider+2012 recalibration of the WDM HMF.
Schneider12_vCDM(m, dndm0, wdm, ...)	Schneider+2012 recalibration of the CDM HMF.
TransferWDM([wdm_mass, wdm_model, wdm_params])	A subclass of *hmf.transfer.Transfer* that mixes in WDM capa
Viel05(mx, cosmo, z, **model_params)	Transfer function from Viel 2005 (which is exactly the same as Bode et
WDM(mx, cosmo, z, **model_params)	Base class for all WDM components.
WDMRecalibrateMF(m, dndm0, wdm, ...)	Base class for Components that emulate the effect of WDM on the HM

hmf.wdm.Bode01

class `hmf.wdm.`**`Bode01`**(*mx*, *cosmo*, *z*, ***model_params*)

Methods

__init__(mx, cosmo, z, **model_params)	
transfer(k)	

hmf.wdm.Bode01.__init__

`Bode01.`**`__init__`**(*mx*, *cosmo*, *z*, ***model_params*)

hmf.wdm.Bode01.transfer

`Bode01.`**`transfer`**(*k*)

Attributes

lam_eff_fs	Effective free-streaming scale.
lam_hm	Half-mode scale.
m_fs	Free-streaming mass scale.
m_hm	Half-mode mass scale.

hmf.wdm.Bode01.lam_eff_fs

`Bode01.`**`lam_eff_fs`**

Effective free-streaming scale.

From Schneider+2013, Eq. 6

hmf.wdm.Bode01.lam_hm

`Bode01.`**`lam_hm`**

Half-mode scale.

From Schneider+2012, Eq. 8.

hmf.wdm.Bode01.m_fs

Bode01.**m_fs**
Free-streaming mass scale.

From Schneider+2012, Eq. 7

hmf.wdm.Bode01.m_hm

Bode01.**m_hm**
Half-mode mass scale.

From Schneider+2013, Eq. 8

hmf.wdm.Component

class hmf.wdm.**Component**(***model_params*)
Base class representing a component model.

All components should be subclassed from this. Components are generally parts of the calculation which can take different models, example the HMF fitting functions, bias models, growth functions, etc.

The feature of this class is that it contains a class variable called _defaults containing the defaults for the parameters of any specific model. These are checked and updated with passed parameters by the __init__ method.

Methods

__init__(**model_params)

hmf.wdm.Component.__init__

Component.**__init__**(***model_params*)

hmf.wdm.Lovell14

class hmf.wdm.**Lovell14**(*m*, *dndm0*, *wdm*, ***model_parameters*)
Lovell+2014 recalibration of the WDM HMF.

Parameters **m** : array_like

Masses at which the HMF is calculated.

dndm0 : array_like

The original HMF at *m*.

wdm : *WDM* subclass instance

An instance of *WDM* providing a Warm Dark Matter model.

****model_parameters** : unpack-dict

Parameters specific to this model: **beta**. To see the default values, check the _defaults class attribute.

Methods

__init__(m, dndm0, wdm, **model_parameters)	
dndm_alter()	

hmf.wdm.Lovell14.__init__

Lovell14.**__init__**(*m*, *dndm0*, *wdm*, ***model_parameters*)

hmf.wdm.Lovell14.dndm_alter

Lovell14.**dndm_alter**()

hmf.wdm.MassFunctionWDM

class hmf.wdm.**MassFunctionWDM**(*alter_dndm=None*, *alter_params=None*, ***kwargs*)

A subclass of hmf.MassFunction that mixes in WDM capabilities.

This replaces the standard CDM quantities with WDM-derived ones, where relevant.

In addition to the parameters directly passed to this class, others are available which are passed on to its superclass. To read a standard documented list of (all) parameters, use MassFunctionWDM.parameter_info(). If you want to just see the plain list of available parameters, use MassFunctionWDM.get_all_parameters().To see the actual defaults for each parameter, use MassFunctionWDM.get_all_parameter_defaults().

Methods

__init__([alter_dndm, alter_params])	
get_all_parameter_defaults()	Dictionary of all parameters and defaults
get_all_parameter_names()	Yield all parameter names in the class.
get_all_parameters()	Yield all parameters as tuples of (name,obj)
parameter_info()	
update(**kwargs)	Update parameters of the framework with kwargs.

hmf.wdm.MassFunctionWDM.__init__

MassFunctionWDM.**__init__**(*alter_dndm=None*, *alter_params=None*, ***kwargs*)

hmf.wdm.MassFunctionWDM.get_all_parameter_defaults

MassFunctionWDM.**get_all_parameter_defaults**()

Dictionary of all parameters and defaults

hmf.wdm.MassFunctionWDM.get_all_parameter_names

MassFunctionWDM.**get_all_parameter_names**()
Yield all parameter names in the class.

hmf.wdm.MassFunctionWDM.get_all_parameters

MassFunctionWDM.**get_all_parameters**()
Yield all parameters as tuples of (name,obj)

hmf.wdm.MassFunctionWDM.parameter_info

MassFunctionWDM.**parameter_info**()

hmf.wdm.MassFunctionWDM.update

MassFunctionWDM.**update**(***kwargs*)
Update parameters of the framework with kwargs.

Attributes

M	Masses (alias of m, deprecated)
Mmax	**Parameter**: Maximum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].
Mmin	**Parameter**: Minimum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].
alter_dndm	**Parameter**: A model for empirical recalibration of the HMF.
alter_params	**Parameter**: Model parameters for *alter_dndm*.
cosmo	Cosmographic object (`astropy.cosmology.FLRW` object), with custom cosmology from *cosmo_p*
cosmo_model	**Parameter**: The basis for the cosmology – see astropy documentation. Can be a custom
cosmo_params	**Parameter**: Parameters for the cosmology that deviate from the base cosmology passed.
delta_c	**Parameter**: The critical overdensity for collapse, δ_c.
delta_h	**Parameter**: The overdensity for the halo definition, with respect to *delta_wrt*
delta_halo	Overdensity of a halo w.r.t mean density
delta_k	Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$
delta_wrt	**Parameter**: Defines what the overdensity of a halo is with respect to, mean density
dlnk	**Parameter**: Step-size of log wavenumbers
dlog10m	**Parameter**: log10 interval between mass bins
dndlnm	The differential mass function in terms of natural log of *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]
dndlog10m	The differential mass function in terms of log of *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]
dndm	The number density of haloes in WDM, `len=len(m)` [units $h^4 M_\odot^{-1} Mpc^{-3}$]
filter	Instantiated model for filter/window functions.
filter_model	**Parameter**: A model for the window/filter function.
filter_params	**Parameter**: Model parameters for *filter_model*.
fsigma	The multiplicity function, $f(\sigma)$, for *hmf_model*.
growth	The instantiated growth model
growth_factor	The growth factor
growth_model	**Parameter**: The model to use to calculate the growth function/growth rate.
growth_params	**Parameter**: Relevant parameters of the *growth_model*.

Continued

Table 8.91 – continued from previous page

hmf	Instantiated model for the hmf fitting function.
hmf_model	**Parameter**: A model to use as the fitting function $f(\sigma)$
hmf_params	**Parameter**: Model parameters for *hmf_model*.
how_big	Size of simulation volume in which to expect one halo of mass m, `len=len(m)` [units $Mpch^{-1}$]
k	Wavenumbers, [h/Mpc]
lnk_max	**Parameter**: Maximum (natural) log wavenumber, *k* [h/Mpc].
lnk_min	**Parameter**: Minimum (natural) log wavenumber, *k* [h/Mpc].
lnsigma	Natural log of inverse mass variance, `len=len(m)`
m	Masses
mass_nonlinear	The nonlinear mass, nu(Mstar) = 1.
mean_density	Mean density of universe at redshift z
mean_density0	Mean density of universe at z=0, [Msun h^2 / Mpc**3]
n	**Parameter**: Spectral index of fluctations
n_eff	Effective spectral index at scale of halo radius, `len=len(m)`
ngtm	The cumulative mass function above *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]
nonlinear_delta_k	Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$
nonlinear_power	Non-linear log power [units Mpc^3/h^3]
nu	The parameter $\nu = \left(\frac{\delta_c}{\sigma}\right)^2$, `len=len(m)`
parameter_values	Dictionary of all parameters and their current values
power	Normalised log power spectrum [units Mpc^3/h^3]
radii	The radii corresponding to the masses *m*.
rho_gtm	Mass density in haloes >*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]
rho_ltm	Mass density in haloes <*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]
sigma	The mass variance at *z*, `len=len(m)`
sigma_8	**Parameter**: RMS linear density fluctations in spheres of radius 8 Mpc/h
takahashi	**Parameter**: Whether to use updated HALOFIT coefficients from Takahashi+12
transfer	The instantiated transfer model
transfer_model	**Parameter**: Defines which transfer function model to use.
transfer_params	**Parameter**: Relevant parameters of the *transfer_model*.
wdm	The instantiated WDM model.
wdm_mass	**Parameter**: Mass of the WDM particle.
wdm_model	**Parameter**: A model for the WDM effect on the transfer function.
wdm_params	**Parameter**: Parameters of the WDM model.
z	**Parameter**: Redshift.

hmf.wdm.MassFunctionWDM.M

MassFunctionWDM.**M**
Masses (alias of m, deprecated)

hmf.wdm.MassFunctionWDM.Mmax

MassFunctionWDM.**Mmax**
Parameter: Maximum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].

Type float

hmf.wdm.MassFunctionWDM.Mmin

MassFunctionWDM.**Mmin**
Parameter: Minimum mass at which to perform analysis [units $\log_{10} M_\odot h^{-1}$].

Type float

hmf.wdm.MassFunctionWDM.alter_dndm

MassFunctionWDM.**alter_dndm**
Parameter: A model for empirical recalibration of the HMF.

Type None, str, or :class'WDMRecalibrateMF' subclass.

hmf.wdm.MassFunctionWDM.alter_params

MassFunctionWDM.**alter_params**
Parameter: Model parameters for *alter_dndm*.

hmf.wdm.MassFunctionWDM.cosmo

MassFunctionWDM.**cosmo**
Cosmographic object (`astropy.cosmology.FLRW` object), with custom cosmology from *cosmo_params* applied.

hmf.wdm.MassFunctionWDM.cosmo_model

MassFunctionWDM.**cosmo_model**
Parameter: The basis for the cosmology – see astropy documentation. Can be a custom subclass. Defaults to Planck15.

Type instance of *astropy.cosmology.FLRW* subclass

hmf.wdm.MassFunctionWDM.cosmo_params

MassFunctionWDM.**cosmo_params**
Parameter: Parameters for the cosmology that deviate from the base cosmology passed. This is useful for repeated updates of a single parameter (leaving others the same). Default is the empty dict. The parameters passed must match the allowed parameters of *cosmo_model*. For the basic class this is

Tcmb0 Temperature of the CMB at z=0

Neff Number of massless neutrino species

M_nu Mass of neutrino species (list)

H0 The hubble constant at z=0

Om0 The normalised matter density at z=0

Type dict

hmf.wdm.MassFunctionWDM.delta_c

MassFunctionWDM.**delta_c**
Parameter: The critical overdensity for collapse, δ_c.

Type float

hmf.wdm.MassFunctionWDM.delta_h

MassFunctionWDM.**delta_h**
Parameter: The overdensity for the halo definition, with respect to *delta_wrt*

Type float

hmf.wdm.MassFunctionWDM.delta_halo

MassFunctionWDM.**delta_halo**
Overdensity of a halo w.r.t mean density

hmf.wdm.MassFunctionWDM.delta_k

MassFunctionWDM.**delta_k**
Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$

hmf.wdm.MassFunctionWDM.delta_wrt

MassFunctionWDM.**delta_wrt**
Parameter: Defines what the overdensity of a halo is with respect to, mean density of the universe, or critical density.

Type str, {"mean", "crit"}

hmf.wdm.MassFunctionWDM.dlnk

MassFunctionWDM.**dlnk**
Parameter: Step-size of log wavenumbers

Type float

hmf.wdm.MassFunctionWDM.dlog10m

MassFunctionWDM.**dlog10m**
Parameter: log10 interval between mass bins

Type float

hmf.wdm.MassFunctionWDM.dndlnm

MassFunctionWDM.**dndlnm**
The differential mass function in terms of natural log of m, `len=len(m)` [units $h^3 Mpc^{-3}$]

hmf.wdm.MassFunctionWDM.dndlog10m

MassFunctionWDM.**dndlog10m**
The differential mass function in terms of log of *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]

hmf.wdm.MassFunctionWDM.dndm

MassFunctionWDM.**dndm**
The number density of haloes in WDM, `len=len(m)` [units $h^4 M_\odot^{-1} Mpc^{-3}$]

hmf.wdm.MassFunctionWDM.filter

MassFunctionWDM.**filter**
Instantiated model for filter/window functions.

hmf.wdm.MassFunctionWDM.filter_model

MassFunctionWDM.**filter_model**
Parameter: A model for the window/filter function.

Type str or *hmf.filters.Filter* subclass

hmf.wdm.MassFunctionWDM.filter_params

MassFunctionWDM.**filter_params**
Parameter: Model parameters for *filter_model*.

Type dict

hmf.wdm.MassFunctionWDM.fsigma

MassFunctionWDM.**fsigma**
The multiplicity function, $f(\sigma)$, for *hmf_model*. `len=len(m)`

hmf.wdm.MassFunctionWDM.growth

MassFunctionWDM.**growth**
The instantiated growth model

hmf.wdm.MassFunctionWDM.growth_factor

MassFunctionWDM.**growth_factor**
The growth factor

hmf.wdm.MassFunctionWDM.growth_model

MassFunctionWDM.**growth_model**
Parameter: The model to use to calculate the growth function/growth rate.

Type str or *hmf.growth_factor.GrowthFactor* subclass

hmf.wdm.MassFunctionWDM.growth_params

MassFunctionWDM.**growth_params**
Parameter: Relevant parameters of the *growth_model*.

Type dict

hmf.wdm.MassFunctionWDM.hmf

MassFunctionWDM.**hmf**
Instantiated model for the hmf fitting function.

hmf.wdm.MassFunctionWDM.hmf_model

MassFunctionWDM.**hmf_model**
Parameter: A model to use as the fitting function $f(\sigma)$

Type str or *hmf.fitting_functions.FittingFunction* subclass

hmf.wdm.MassFunctionWDM.hmf_params

MassFunctionWDM.**hmf_params**
Parameter: Model parameters for *hmf_model*.

Type dict

hmf.wdm.MassFunctionWDM.how_big

MassFunctionWDM.**how_big**
Size of simulation volume in which to expect one halo of mass m, `len=len(m)` [units $Mpch^{-1}$]

hmf.wdm.MassFunctionWDM.k

MassFunctionWDM.**k**
Wavenumbers, [h/Mpc]

hmf.wdm.MassFunctionWDM.lnk_max

MassFunctionWDM.**lnk_max**
Parameter: Maximum (natural) log wavenumber, *k* [h/Mpc].

Type float

hmf.wdm.MassFunctionWDM.lnk_min

MassFunctionWDM.**lnk_min**
Parameter: Minimum (natural) log wavenumber, k [h/Mpc].

Type float

hmf.wdm.MassFunctionWDM.lnsigma

MassFunctionWDM.**lnsigma**
Natural log of inverse mass variance, `len=len(m)`

hmf.wdm.MassFunctionWDM.m

MassFunctionWDM.**m**
Masses

hmf.wdm.MassFunctionWDM.mass_nonlinear

MassFunctionWDM.**mass_nonlinear**
The nonlinear mass, nu(Mstar) = 1.

hmf.wdm.MassFunctionWDM.mean_density

MassFunctionWDM.**mean_density**
Mean density of universe at redshift z

hmf.wdm.MassFunctionWDM.mean_density0

MassFunctionWDM.**mean_density0**
Mean density of universe at z=0, [Msun h^2 / Mpc**3]

hmf.wdm.MassFunctionWDM.n

MassFunctionWDM.**n**
Parameter: Spectral index of fluctations

Must be greater than -3 and less than 4.

Type float

hmf.wdm.MassFunctionWDM.n_eff

MassFunctionWDM.**n_eff**
Effective spectral index at scale of halo radius, `len=len(m)`

Notes

This function, and any derived quantities, can show small non-physical 'wiggles' at the 0.1% level, if too coarse a grid in ln(k) is used. If applications are sensitive at this level, please use a very fine k-space grid.

Uses eq. 42 in Lukic et. al 2007.

hmf.wdm.MassFunctionWDM.ngtm

MassFunctionWDM.**ngtm**
The cumulative mass function above *m*, `len=len(m)` [units $h^3 Mpc^{-3}$]

In the case that *m* does not extend to sufficiently high masses, this routine will auto-generate `dndm` for an extended mass range.

In the case of the ff.Behroozi fit, it is impossible to auto-extend the mass range except by the power-law fit, thus one should be careful to supply appropriate mass ranges in this case.

hmf.wdm.MassFunctionWDM.nonlinear_delta_k

MassFunctionWDM.**nonlinear_delta_k**
Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$

hmf.wdm.MassFunctionWDM.nonlinear_power

MassFunctionWDM.**nonlinear_power**
Non-linear log power [units Mpc^3/h^3]

Non-linear corrections come from HALOFIT.

hmf.wdm.MassFunctionWDM.nu

MassFunctionWDM.**nu**
The parameter $\nu = \left(\frac{\delta_c}{\sigma}\right)^2$, `len=len(m)`

hmf.wdm.MassFunctionWDM.parameter_values

MassFunctionWDM.**parameter_values**
Dictionary of all parameters and their current values

hmf.wdm.MassFunctionWDM.power

MassFunctionWDM.**power**
Normalised log power spectrum [units Mpc^3/h^3]

hmf.wdm.MassFunctionWDM.radii

MassFunctionWDM.**radii**
The radii corresponding to the masses *m*.

hmf.wdm.MassFunctionWDM.rho_gtm

MassFunctionWDM.**rho_gtm**
Mass density in haloes >*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]

In the case that *m* does not extend to sufficiently high masses, this routine will auto-generate `dndm` for an extended mass range.

In the case of the ff.Behroozi fit, it is impossible to auto-extend the mass range except by the power-law fit, thus one should be careful to supply appropriate mass ranges in this case.

hmf.wdm.MassFunctionWDM.rho_ltm

MassFunctionWDM.**rho_ltm**
Mass density in haloes <*m*, `len=len(m)` [units $M_\odot h^2 Mpc^{-3}$]

Note: As of v1.6.2, this assumes that the entire mass density of halos is encoded by the `mean_density` parameter (ie. all mass is found in halos). This is not explicitly true of all fitting functions (eg. Warren), in which case the definition of this property is somewhat inconsistent, but will still work.

In the case that *m* does not extend to sufficiently high masses, this routine will auto-generate `dndm` for an extended mass range.

In the case of the ff.Behroozi fit, it is impossible to auto-extend the mass range except by the power-law fit, thus one should be careful to supply appropriate mass ranges in this case.

hmf.wdm.MassFunctionWDM.sigma

MassFunctionWDM.**sigma**
The mass variance at z, `len=len(m)`

hmf.wdm.MassFunctionWDM.sigma_8

MassFunctionWDM.**sigma_8**
Parameter: RMS linear density fluctations in spheres of radius 8 Mpc/h

Type float

hmf.wdm.MassFunctionWDM.takahashi

MassFunctionWDM.**takahashi**
Parameter: Whether to use updated HALOFIT coefficients from Takahashi+12

Type bool

hmf.wdm.MassFunctionWDM.transfer

MassFunctionWDM.**transfer**
The instantiated transfer model

hmf.wdm.MassFunctionWDM.transfer_model

MassFunctionWDM.**transfer_model**
Parameter: Defines which transfer function model to use.

Built-in available models are found in the *hmf.transfer_models* module. Default is CAMB if installed, otherwise EH.

Type str or *hmf.transfer_models.TransferComponent* subclass, optional

hmf.wdm.MassFunctionWDM.transfer_params

MassFunctionWDM.**transfer_params**
Parameter: Relevant parameters of the *transfer_model*.

Type dict

hmf.wdm.MassFunctionWDM.wdm

MassFunctionWDM.**wdm**
The instantiated WDM model.

Contains quantities relevant to WDM.

hmf.wdm.MassFunctionWDM.wdm_mass

MassFunctionWDM.**wdm_mass**
Parameter: Mass of the WDM particle.

Type float

hmf.wdm.MassFunctionWDM.wdm_model

MassFunctionWDM.**wdm_model**
Parameter: A model for the WDM effect on the transfer function.

Type str or *WDM* subclass

hmf.wdm.MassFunctionWDM.wdm_params

MassFunctionWDM.**wdm_params**
Parameter: Parameters of the WDM model.

Type dict

hmf.wdm.MassFunctionWDM.z

MassFunctionWDM.**z**
Parameter: Redshift.

Must be greater than 0.

Type float

hmf.wdm.Schneider12

class hmf.wdm.**Schneider12**(*m*, *dndm0*, *wdm*, ***model_parameters*)
Schneider+2012 recalibration of the WDM HMF.

Parameters **m** : array_like

Masses at which the HMF is calculated.

dndm0 : array_like

The original WDM HMF at *m*.

wdm : *WDM* subclass instance

An instance of *WDM* providing a Warm Dark Matter model.

****model_parameters** : unpack-dict

Parameters specific to this model: **alpha**. To see the default values, check the `_defaults` class attribute.

Methods

__init__(m, dndm0, wdm, **model_parameters)
dndm_alter()

hmf.wdm.Schneider12.__init__

Schneider12.**__init__**(*m*, *dndm0*, *wdm*, ***model_parameters*)

hmf.wdm.Schneider12.dndm_alter

Schneider12.**dndm_alter**()

hmf.wdm.Schneider12_vCDM

class hmf.wdm.**Schneider12_vCDM**(*m*, *dndm0*, *wdm*, ***model_parameters*)
Schneider+2012 recalibration of the CDM HMF.

Parameters **m** : array_like

Masses at which the HMF is calculated.

dndm0 : array_like

The CDM HMF at *m*.

wdm : *WDM* subclass instance

An instance of *WDM* providing a Warm Dark Matter model.

****model_parameters** : unpack-dict

Parameters specific to this model: **beta**. To see the default values, check the `_defaults` class attribute.

Methods

__init__(m, dndm0, wdm, **model_parameters)	
dndm_alter()	

hmf.wdm.Schneider12_vCDM.__init__

Schneider12_vCDM.**__init__**(*m*, *dndm0*, *wdm*, ***model_parameters*)

hmf.wdm.Schneider12_vCDM.dndm_alter

Schneider12_vCDM.**dndm_alter**()

hmf.wdm.TransferWDM

class hmf.wdm.**TransferWDM**(*wdm_mass=3.0*, *wdm_model=<class 'hmf.wdm.Viel05'>*, *wdm_params={}*, ***transfer_kwargs*)

A subclass of *hmf.transfer.Transfer* that mixes in WDM capabilities.

This replaces the standard CDM quantities with WDM-derived ones, where relevant.

In addition to the parameters directly passed to this class, others are available which are passed on to its superclass. To read a standard documented list of (all) parameters, use `TransferWDM.parameter_info()`. If you want to just see the plain list of available parameters, use `TransferWDM.get_all_parameters()`.To see the actual defaults for each parameter, use `TransferWDM.get_all_parameter_defaults()`.

Methods

__init__([wdm_mass, wdm_model, wdm_params])	
get_all_parameter_defaults()	Dictionary of all parameters and defaults
get_all_parameter_names()	Yield all parameter names in the class.
get_all_parameters()	Yield all parameters as tuples of (name,obj)
parameter_info()	
update(**kwargs)	Update parameters of the framework with kwargs.

hmf.wdm.TransferWDM.__init__

TransferWDM.**__init__**(*wdm_mass=3.0*, *wdm_model=<class 'hmf.wdm.Viel05'>*, *wdm_params={}*, ***transfer_kwargs*)

hmf.wdm.TransferWDM.get_all_parameter_defaults

TransferWDM.**get_all_parameter_defaults**()

Dictionary of all parameters and defaults

hmf.wdm.TransferWDM.get_all_parameter_names

TransferWDM.**get_all_parameter_names**()
Yield all parameter names in the class.

hmf.wdm.TransferWDM.get_all_parameters

TransferWDM.**get_all_parameters**()
Yield all parameters as tuples of (name,obj)

hmf.wdm.TransferWDM.parameter_info

TransferWDM.**parameter_info**()

hmf.wdm.TransferWDM.update

TransferWDM.**update**(***kwargs*)
Update parameters of the framework with kwargs.

Attributes

cosmo	Cosmographic object (astropy.cosmology.FLRW object), with custom cosmology from *cosmo_p*
cosmo_model	**Parameter**: The basis for the cosmology – see astropy documentation. Can be a custom
cosmo_params	**Parameter**: Parameters for the cosmology that deviate from the base cosmology passed.
delta_k	Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$
dlnk	**Parameter**: Step-size of log wavenumbers
growth	The instantiated growth model
growth_factor	The growth factor
growth_model	**Parameter**: The model to use to calculate the growth function/growth rate.
growth_params	**Parameter**: Relevant parameters of the *growth_model*.
k	Wavenumbers, [h/Mpc]
lnk_max	**Parameter**: Maximum (natural) log wavenumber, *k* [h/Mpc].
lnk_min	**Parameter**: Minimum (natural) log wavenumber, *k* [h/Mpc].
mean_density0	Mean density of universe at z=0, [Msun h^2 / Mpc**3]
n	**Parameter**: Spectral index of fluctations
nonlinear_delta_k	Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$
nonlinear_power	Non-linear log power [units Mpc^3/h^3]
parameter_values	Dictionary of all parameters and their current values
power	Normalised log power spectrum [units Mpc^3/h^3]
sigma_8	**Parameter**: RMS linear density fluctations in spheres of radius 8 Mpc/h
takahashi	**Parameter**: Whether to use updated HALOFIT coefficients from Takahashi+12
transfer	The instantiated transfer model
transfer_model	**Parameter**: Defines which transfer function model to use.
transfer_params	**Parameter**: Relevant parameters of the *transfer_model*.
wdm	The instantiated WDM model.
wdm_mass	**Parameter**: Mass of the WDM particle.
wdm_model	**Parameter**: A model for the WDM effect on the transfer function.

Continued

Table 8.95 – continued from previous page

wdm_params	**Parameter**: Parameters of the WDM model.
z	**Parameter**: Redshift.

hmf.wdm.TransferWDM.cosmo

TransferWDM.**cosmo**
Cosmographic object (`astropy.cosmology.FLRW` object), with custom cosmology from *cosmo_params* applied.

hmf.wdm.TransferWDM.cosmo_model

TransferWDM.**cosmo_model**
Parameter: The basis for the cosmology – see astropy documentation. Can be a custom subclass. Defaults to Planck15.

Type instance of *astropy.cosmology.FLRW* subclass

hmf.wdm.TransferWDM.cosmo_params

TransferWDM.**cosmo_params**
Parameter: Parameters for the cosmology that deviate from the base cosmology passed. This is useful for repeated updates of a single parameter (leaving others the same). Default is the empty dict. The parameters passed must match the allowed parameters of *cosmo_model*. For the basic class this is

Tcmb0 Temperature of the CMB at z=0

Neff Number of massless neutrino species

M_nu Mass of neutrino species (list)

H0 The hubble constant at z=0

Om0 The normalised matter density at z=0

Type dict

hmf.wdm.TransferWDM.delta_k

TransferWDM.**delta_k**
Dimensionless power spectrum, $\Delta_k = \frac{k^3 P(k)}{2\pi^2}$

hmf.wdm.TransferWDM.dlnk

TransferWDM.**dlnk**
Parameter: Step-size of log wavenumbers

Type float

hmf.wdm.TransferWDM.growth

TransferWDM.**growth**
The instantiated growth model

hmf.wdm.TransferWDM.growth_factor

TransferWDM.**growth_factor**
The growth factor

hmf.wdm.TransferWDM.growth_model

TransferWDM.**growth_model**
Parameter: The model to use to calculate the growth function/growth rate.

Type str or *hmf.growth_factor.GrowthFactor* subclass

hmf.wdm.TransferWDM.growth_params

TransferWDM.**growth_params**
Parameter: Relevant parameters of the *growth_model*.

Type dict

hmf.wdm.TransferWDM.k

TransferWDM.**k**
Wavenumbers, [h/Mpc]

hmf.wdm.TransferWDM.lnk_max

TransferWDM.**lnk_max**
Parameter: Maximum (natural) log wavenumber, *k* [h/Mpc].

Type float

hmf.wdm.TransferWDM.lnk_min

TransferWDM.**lnk_min**
Parameter: Minimum (natural) log wavenumber, *k* [h/Mpc].

Type float

hmf.wdm.TransferWDM.mean_density0

TransferWDM.**mean_density0**
Mean density of universe at z=0, [Msun h^2 / Mpc**3]

hmf.wdm.TransferWDM.n

TransferWDM.**n**
Parameter: Spectral index of fluctations

Must be greater than -3 and less than 4.

Type float

hmf.wdm.TransferWDM.nonlinear_delta_k

TransferWDM.**nonlinear_delta_k**
Dimensionless nonlinear power spectrum, $\Delta_k = \frac{k^3 P_{\rm nl}(k)}{2\pi^2}$

hmf.wdm.TransferWDM.nonlinear_power

TransferWDM.**nonlinear_power**
Non-linear log power [units Mpc^3/h^3]

Non-linear corrections come from HALOFIT.

hmf.wdm.TransferWDM.parameter_values

TransferWDM.**parameter_values**
Dictionary of all parameters and their current values

hmf.wdm.TransferWDM.power

TransferWDM.**power**
Normalised log power spectrum [units Mpc^3/h^3]

hmf.wdm.TransferWDM.sigma_8

TransferWDM.**sigma_8**
Parameter: RMS linear density fluctations in spheres of radius 8 Mpc/h

Type float

hmf.wdm.TransferWDM.takahashi

TransferWDM.**takahashi**
Parameter: Whether to use updated HALOFIT coefficients from Takahashi+12

Type bool

hmf.wdm.TransferWDM.transfer

TransferWDM.**transfer**
The instantiated transfer model

hmf.wdm.TransferWDM.transfer_model

TransferWDM.**transfer_model**
Parameter: Defines which transfer function model to use.

Built-in available models are found in the *hmf.transfer_models* module. Default is CAMB if installed, otherwise EH.

Type str or *hmf.transfer_models.TransferComponent* subclass, optional

hmf.wdm.TransferWDM.transfer_params

TransferWDM.**transfer_params**
Parameter: Relevant parameters of the *transfer_model*.

Type dict

hmf.wdm.TransferWDM.wdm

TransferWDM.**wdm**
The instantiated WDM model.

Contains quantities relevant to WDM.

hmf.wdm.TransferWDM.wdm_mass

TransferWDM.**wdm_mass**
Parameter: Mass of the WDM particle.

Type float

hmf.wdm.TransferWDM.wdm_model

TransferWDM.**wdm_model**
Parameter: A model for the WDM effect on the transfer function.

Type str or *WDM* subclass

hmf.wdm.TransferWDM.wdm_params

TransferWDM.**wdm_params**
Parameter: Parameters of the WDM model.

Type dict

hmf.wdm.TransferWDM.z

TransferWDM.**z**
Parameter: Redshift.

Must be greater than 0.

Type float

hmf.wdm.Viel05

class hmf.wdm.**Viel05**(*mx*, *cosmo*, *z*, ***model_params*)

Transfer function from Viel 2005 (which is exactly the same as Bode et al. 2001).

Formula from Bode et. al. 2001 eq. A9.

Parameters **mx** : float

Mass of the particle in keV

cosmo : *hmf.cosmo.Cosmology* instance

A cosmology.

z : float

Redshift.

****model_parameters** : unpack-dict

Parameters specific to a model. Available parameters are as follows. To see the default values, check the `_defaults` class attribute.

mu

g_x

Methods

__init__(mx, cosmo, z, **model_params)	
transfer(k)	

hmf.wdm.Viel05.__init__

Viel05.**__init__**(*mx*, *cosmo*, *z*, ***model_params*)

hmf.wdm.Viel05.transfer

Viel05.**transfer**(*k*)

Attributes

lam_eff_fs	Effective free-streaming scale.
lam_hm	Half-mode scale.
m_fs	Free-streaming mass scale.
m_hm	Half-mode mass scale.

hmf.wdm.Viel05.lam_eff_fs

Viel05.**lam_eff_fs**

Effective free-streaming scale.

From Schneider+2013, Eq. 6

hmf.wdm.Viel05.lam_hm

Viel05.**lam_hm**
Half-mode scale.

From Schneider+2012, Eq. 8.

hmf.wdm.Viel05.m_fs

Viel05.**m_fs**
Free-streaming mass scale.

From Schneider+2012, Eq. 7

hmf.wdm.Viel05.m_hm

Viel05.**m_hm**
Half-mode mass scale.

From Schneider+2013, Eq. 8

hmf.wdm.WDM

class hmf.wdm.**WDM**(*mx*, *cosmo*, *z*, ***model_params*)
Base class for all WDM components.

Do not use this class directly. The primary purpose of the WDM Component is to modify the transfer function. Thus, the only requisite method to define in any given subclass is *transfer()*, which calculates this quantity in the proposed WDM model.

Parameters **mx** : float

Mass of the particle in keV

cosmo : *hmf.cosmo.Cosmology* instance

A cosmology.

z : float

Redshift.

****model_parameters** : unpack-dict

Parameters specific to a model. To see the default values, check the `_defaults` class attribute.

Methods

__init__(mx, cosmo, z, **model_params)	
transfer(lnk)	Transfer function for WDM models

hmf.wdm.WDM.__init__

WDM.**__init__**(*mx*, *cosmo*, *z*, ***model_params*)

hmf.wdm.WDM.transfer

WDM.**transfer**(*lnk*)

Transfer function for WDM models

Parameters **lnk** : array

The wavenumbers *k/h* corresponding to `power_cdm`.

Returns **transfer** : array_like

The WDM transfer function at *lnk*.

hmf.wdm.WDMRecalibrateMF

class hmf.wdm.**WDMRecalibrateMF**(*m*, *dndm0*, *wdm*, ***model_parameters*)

Base class for Components that emulate the effect of WDM on the HMF empirically.

Required method is *dndm_alter()*.

Parameters **m** : array_like

Masses at which the HMF is calculated.

dndm0 : array_like

The original HMF at *m*.

wdm : *WDM* subclass instance

An instance of *WDM* providing a Warm Dark Matter model.

****model_parameters** : unpack-dict

Parameters specific to a model. To see the default values, check the `_defaults` class attribute.

Methods

__init__(m, dndm0, wdm, **model_parameters)
dndm_alter()

hmf.wdm.WDMRecalibrateMF.__init__

WDMRecalibrateMF.**__init__**(*m*, *dndm0*, *wdm*, ***model_parameters*)

hmf.wdm.WDMRecalibrateMF.dndm_alter

WDMRecalibrateMF.**dndm_alter**()

8.3.10 hmf.integrate_hmf

A supporting module that provides a routine to integrate the differential hmf in a robust manner.

Functions

hmf_integral_gtm(M, dndm[, mass_density])	Cumulatively integrate dn/dm.

hmf.integrate_hmf.hmf_integral_gtm

hmf.integrate_hmf.**hmf_integral_gtm**(*M*, *dndm*, *mass_density=False*)

Cumulatively integrate dn/dm.

Parameters **M** : array_like

Array of masses.

dndm : array_like

Array of dn/dm (corresponding to M)

mass_density : bool, *False*

Whether to calculate mass density (or number density).

Returns **ngtm** : array_like

Cumulative integral of dndm.

Examples

Using a simple power-law mass function:

```
>>> import numpy as np
>>> m = np.logspace(10,18,500)
>>> dndm = m**-2
>>> ngtm = hmf_integral_gtm(m,dndm)
>>> np.allclose(ngtm,1/m) #1/m is the analytic integral to infinity.
True
```

The function always integrates to m=1e18, and extrapolates with a spline if data not provided:

```
>>> m = np.logspace(10,12,500)
>>> dndm = m**-2
>>> ngtm = hmf_integral_gtm(m,dndm)
>>> np.allclose(ngtm,1/m) #1/m is the analytic integral to infinity.
True
```

Exceptions

NaNException

8.3.11 hmf.functional

This module provides functions for generating several *hmf.hmf.MassFunction* instances from a combination of lists of parameters, in optimal order.

The underlying idea here is that typically modifying say the redshift has a smaller number of re-computations than modifying the base cosmological parameters. Thus, in a nested loop, the redshift should be the inner loop.

It is not always obvious which order the loops should be, so this module provides functions to determine that order, and indeed perform the loops.

Functions

get_best_param_order(kls[, q])	Get an optimal parameter order for nested loops.
get_hmf(req_qauntities[, get_label, ...])	Yield framework instances for all combinations of parameters supplied.

hmf.functional.get_best_param_order

hmf.functional.**get_best_param_order**(*kls*, *q='dndm'*, ***kwargs*)

Get an optimal parameter order for nested loops.

The underlying idea here is that typically modifying say the redshift has a smaller number of re-computations than modifying the base cosmological parameters. Thus, in a nested loop, the redshift should be the inner loop.

It is not always obvious which order the loops should be, so this function determines that order.

Note: The order is calculated based on actually running one iteration of the loop, so providing arguments which enable a fast calculation is helpful.

Parameters **kls** : hmf._framwork.Framework class

An arbitrary framework for which to determine parameter ordering.

q : str or list of str

A string specifying the desired output (e.g. "dndm"), or a list of such strings.

kwargs : unpacked-dict

Arbitrary keyword arguments to the framework initialiser. These are only for determination of parameter order and so may be very poor in resolution to improve efficiency.

Returns **final_list** : list

An ordered list of parameters, with the first corresponding to the outer-most loop.

Examples

```
>>> from hmf import MassFunction
>>> print get_best_param_order(MassFunction,"dndm",transfer_model="BBKS",dlnk=1,dlog10m=1)[::3]
['z2', 'hmf_model', 'delta_wrt', 'growth_params', 'filter_params', 'Mmin', 'transfer_params', 'd
```

hmf.functional.get_hmf

hmf.functional.**get_hmf**(*req_qauntities*, *get_label=True*, *framework=<class 'hmf.hmf.MassFunction'>*, *fast_kwargs={'Mmax': 11.5, 'lnk_min': -1, 'dlnk': 1, 'lnk_max': 1, 'Mmin': 10, 'transfer_model': 'BBKS', 'dlog10m': 0.5}*, ***kwargs*)

Yield framework instances for all combinations of parameters supplied.

The underlying idea here is that typically modifying say the redshift has a smaller number of re-computations than modifying the base cosmological parameters. Thus, in a nested loop, the redshift should be the inner loop.

It is not always obvious which order the loops should be, but this function internally determines the order, and calculates the requisite quantities in a series of framework instances.

Parameters **req_qauntities** : str or list of str

A string defining the quantities that should be pre-cached in the output instances. It is advisable that *any* required quantities for a given application be provided here, to ensure proper optimization.

get_label : bool, optional

Whether to return a list of string labels designating each combination of parameters.

framework : *hmf._framework.Framework* class, optional

A framework for which to perform the optimization.

fast_kwargs : dict, optional

Parameters to be used in the initial run to determine optimal order. These should be set to provide very quick calculation, and do not affect the final result. This will need to be over-ridden for frameworks other than `hmf.MassFunction`.

kwargs : unpacked-dict

Any of the parameters to the initialiser of *framework* which should be calculated. These may be scalar or lists. The total number of calculations will be the total combination of all parameters.

Yields **quantities** : list

A list of quantities, specified by the *req_quantities* arguments

x : Framework instance

An instance of *framework*, with the requisitie quantities pre-cached.

label : optional

If *get_label* is True, also returns a string label uniquely specifying the current parameter combination.

Examples

The following operation will run 12 iterations, yielding the desired quantities, an instance containing those and other quantities, and a unique label at every iteration.

```
>>> for quants, mf, label in get_hmf(['dndm','ngtm'],z=range(3),hmf_model=["ST","PS"],sigma_8=[0
>>>     print label
sigma.8: 0.7, ST, z: 0
sigma.8: 0.8, ST, z: 0
sigma.8: 0.7, ST, z: 1
sigma.8: 0.8, ST, z: 1
sigma.8: 0.7, ST, z: 2
sigma.8: 0.8, ST, z: 2
sigma.8: 0.7, PS, z: 0
sigma.8: 0.8, PS, z: 0
sigma.8: 0.7, PS, z: 1
sigma.8: 0.8, PS, z: 1
```

```
    sigma.8: 0.7, PS, z: 2
    sigma.8: 0.8, PS, z: 2
```

To calculate all of them and keep the results as a list:

```
    >>> big_list = list(get_hmf('mean_density',z=range(8)))
    >>> print [x[0][0]/1e10 for x in big_list]
    [8.531878308131338, 68.2550264650507, 230.36071431954613, 546.0402117204056, 1066.4847885164174,
```

8.3.12 hmf.sample

Module for dealing with sampled mass functions.

Provides routines for sampling theoretical functions, and for binning sampled data.

Functions

dndm_from_sample(m, V[, nm, bins])	Generate a binned dn/dm from a sample of halo masses.
sample_mf(N, log_mmin[, sort])	Create a sample of halo masses from a theoretical mass function.

hmf.sample.dndm_from_sample

hmf.sample.**dndm_from_sample**(*m*, *V*, *nm=None*, *bins=50*)

Generate a binned dn/dm from a sample of halo masses.

Parameters **m** : array_like

A sample of masses

V : float

Physical volume of the sample

nm : array_like

A multiplicity of each of the masses – useful for samples from simulations in which the number of unique masses is much smaller than the total sample.

bins : int or array

Specifies bins (in log10-space!) for the sample. See *numpy.histogram* for more details.

Returns **centres** : array_like

The centres of the bins.

hist : array_like

The value of dn/dm in each bin.

Notes

The "centres" of the bins are located as the midpoint in log10-space.

If one does not have the volume, it can be calculated as N/n(>mmin).

hmf.sample.sample_mf

hmf.sample.**sample_mf**(*N*, *log_mmin*, *sort=False*, ***mf_kwargs*)

Create a sample of halo masses from a theoretical mass function.

Parameters **N** : int

Number of samples to draw

log_mmin : float

Log10 of the minimum mass to sample [Msun/h]

sort : bool, optional

Whether to sort (in descending order of mass) the output masses.

mf_kwargs : keywords

Anything passed to `hmf.MassFunction` to create the mass function which is sampled.

Returns **m** : array_like

The masses

hmf : *hmf.MassFunction* instance

The instance used to define the mass function.

Notes

Mmax is a free parameter to be sent to *MassFunction*. However, if set at or above 18, the routine will fail. It should be set so that dndm is very small at Mmax, but not numerically 0.

Examples

Simplest example:

```
>>> m,hmf = sample_mf(1e5,11.0)
```

Or change the mass function:

```
>>> m,hmf = sample_mf(1e6,10.0,hmf_model="PS",Mmax=17)
```

8.3.13 hmf._framework

Classes defining the overall structure of the hmf framework.

Functions

get_model(name, mod, **kwargs)	Returns an instance of `name` from the module `mod`, with given params.
*get_model_*(name, mod)	Returns a class `name` from the module `mod`.

hmf._framework.get_model

hmf._framework.**get_model**(*name*, *mod*, ***kwargs*)
Returns an instance of name from the module mod, with given params.

Parameters **name** : str

The class name of the appropriate model

mod : str

The module name of the appropriate module

****kwargs :**

Any parameters for the instantiated model (including model parameters)

hmf._framework.get_model_

hmf._framework.**get_model_**(*name*, *mod*)
Returns a class name from the module mod.

Parameters **name** : str

The class name of the appropriate model

mod : str

The module name of the appropriate module

Classes

Cache()	
Component(**model_params)	Base class representing a component model.
Framework()	Class representing a coherent framework of component models.

hmf._framework.Cache

class hmf._framework.**Cache**

Methods

__init__()

hmf._framework.Cache.__init__

Cache.**__init__**()

hmf._framework.Component

class hmf._framework.**Component**(**model_params*)
Base class representing a component model.

All components should be subclassed from this. Components are generally parts of the calculation which can take different models, example the HMF fitting functions, bias models, growth functions, etc.

The feature of this class is that it contains a class variable called `_defaults` containing the defaults for the parameters of any specific model. These are checked and updated with passed parameters by the __init__ method.

Methods

__init__(**model_params)

hmf._framework.Component.__init__

`Component.__init__`(***model_params*)

hmf._framework.Framework

class `hmf._framework.`**`Framework`**

Class representing a coherent framework of component models.

The specific subclasses of this class should be composed of methods that are decorated with either `@_cache.parameter` for things that are parameters, or `@_cache.cached_property` for derived quantities.

Other methods are permissable, but may complicate matters if a derived quantity uses the non-`cached_property` method. Reserve these for utility methods.

Importantly, any parameter that may be passed to the constructor, *must* be defined as a `parameter` within the class so it may be set properly.

Methods

__init__()	
get_all_parameter_defaults()	Dictionary of all parameters and defaults
get_all_parameter_names()	Yield all parameter names in the class.
get_all_parameters()	Yield all parameters as tuples of (name,obj)
parameter_info()	
update(**kwargs)	Update parameters of the framework with kwargs.

hmf._framework.Framework.__init__

`Framework.__init__`()

hmf._framework.Framework.get_all_parameter_defaults

classmethod `Framework.`**`get_all_parameter_defaults`**`()`

Dictionary of all parameters and defaults

hmf._framework.Framework.get_all_parameter_names

classmethod Framework.**get_all_parameter_names**()
Yield all parameter names in the class.

hmf._framework.Framework.get_all_parameters

classmethod Framework.**get_all_parameters**()
Yield all parameters as tuples of (name,obj)

hmf._framework.Framework.parameter_info

classmethod Framework.**parameter_info**()

hmf._framework.Framework.update

Framework.**update**(***kwargs*)
Update parameters of the framework with kwargs.

Attributes

parameter_values	Dictionary of all parameters and their current values

hmf._framework.Framework.parameter_values

Framework.**parameter_values**
Dictionary of all parameters and their current values

8.4 Releases

8.4.1 Development Version

8.4.2 Older Versions

v2.0.0

v2.0.0 is a (long overdue) major release with several backward-incompatible changes. There are several major features still to come in v2.1.0, which may again be backward incompatible. Though this is not ideal (ideally backwards-incompatible changes will be restricted to increase in the major version number), this has been driven by time constraints.

Known issues with this version, to be addressed by the next, are that both scripts (hmf and hmf-fit) are buggy and untested. Don't use these until the next version unless you're crazy.

Features

- New methods on all frameworks to list all parameters, defaults and current values.
- New general structure for Frameworks and Components makes for simpler delineation and extensibility
- New growth_factor module which adds extensibility to the growth factor calculation
- New transfer_models module which separates the transfer models from the general framework
- New Component which can alter dn/dm in WDM via ad-hoc adjustment
- Added a Prior() abstract base class to the fitting routines
- Added a guess() method to fitting routines
- Added ll() method to Prior classes for future extendibility
- New fit from Ishiyama+2015, Manera+2010 and Pillepich+2009

Enhancements

- Removed nz and z2 from MassFunction. They will return in a later version but better.
- Improved structure for FittingFunction Component, with *cutmask* property defining valid mass range. NOTE: the default MassFunction is no longer to mask values outside the valid range. In fact, the parameter `cut_fit` no longer exists. One can achieve the effect by accessing a relevant array as dndm[MassFunction.hmf.cutmask]
- Renamed some parameters/quantities for more consistency (esp. M –> m)
- No longer dependent on cosmolopy, but rather uses Astropy (v1.0+)
- *mean_dens* now *mean_density0*, as per Astropy
- Added exception to catch when dndm has many NaN values in it.
- Many more tests
- Made the *cosmo* class pickleable by cutting out a method and using it as a function instead.
- Added normalise() to Transfer class.
- Updated fit.py extensively, and provided new example config files
- Send arbitrary kwargs to downhill solver
- New internal _utils module provides inheritable docstrings

Bugfixes

- fixed problem with _gtm method returning nans.
- fixed simple bugs in BBKS and BondEfs transfer models.
- fixes in _cache module
- simple bug when updating sigma_8 fixed.
- Made the EnsembleSampler object pickleable by setting __getstate__
- Major bug fix for EH transfer function without BAO wiggles
- @parameter properties now return docstrings

v1.8.0

February 2, 2015

Features

- Better WDM models
- Added SharpK and SharpKEllipsoid filters and overhauled filter system.

Enhancements

- Separated WDM models from main class for extendibility
- Enhanced caching to deal with subclassing

Bugfixes

- Minor bugfixes

1.7.1

January 28, 2015

Enhancements

- Added warning to docstring of _dlnsdlnm and n_eff for non-physical oscillations.

1.7.0

October 28, 2014

Features

- Very much updated fitting routines, in class structure
- Made fitting_functions more flexible and model-like.

Enhancements

- Modified get_hmf to be more general
- Moved get_hmf and related functions to "functional.py"

1.6.2

September 16, 2014

Features

- New HALOFIT option for original co-oefficients from Smith+03

Enhancements

- Better Singleton labelling in get_hmf
- Much cleaning of mass function integrations. New separate module for it.
- **IMPORTANT**: Removal of nltm routine altogether, as it is inconsistent.
- **IMPORTANT**: mltm now called rho_ltm, and mgtm called rho_gtm
- **IMPORTANT**: Definition of rho_ltm now assumes all mass is in halos.
- Behroozi-specific modifications moved to Behroozi class
- New property hmf which is the actual class for mf_fit

Bugfixes

- Fixed bug in Behroozi fit which caused an infinite recursion
- Tests fixed for new cumulants.

1.6.1

September 8, 2014

Enhancements

- Better get_hmf function

Bugfixes

- Fixed "transfer" property
- Updates fixed for transfer_fit
- Updates fixed for nu
- Fixed cache bug where unexecuted branches caused some properties to be misinterpreted
- Fixed bug in CAMB transfer options, where defaults would overwrite user-given values (introduced in 1.6.0)
- Fixed dependence on transfer_options
- Fixed typo in Tinker10 fit at z = 0

1.6.0

August 19, 2014

Features

- New Tinker10 fit (Tinker renamed Tinker08, but Tinker still available)

Enhancements

- Completely re-worked caching module to be easier to code and faster.
- Better Cosmology class – more input combinations available.

Bugfixes

- Fixed all tests.

1.5.0

May 08, 2014

Features

- Introduced _cache module: Extracts all caching logic to a separate module which defines decorators – much simpler coding!

1.4.5

January 24, 2014

Features

- Added get_hmf function to tools.py – easy iteration over models!
- Added hmf script which provides cmd-line access to most functionality.

Enhancements

- Added Behroozi alias to fits
- Changed kmax and k_per_logint back to have **transfer__** prefix.

Bugfixes

- Fixed a bug on updating delta_c
- Changed default kmax and k_per_logint values a little higher for accuracy.

1.4.4

January 23, 2014

Features

- Added ability to change the default cosmology parameters

Enhancements

- Made updating Cosmology simpler.

Bugfixes

- Fixed a bug in the Tinker function (log was meant to be log10): - thanks to Sebastian Bocquet for pointing this out!
- Fixed a bug in updating n and sigma_8 on their own (introduced in 1.4.0)
- Fixed a bug when using a file for the transfer function.

1.4.3

January 10, 2014

Bugfixes

- Changed license in setup

1.4.2

January 10, 2014

Enhancements

- Mocked imports of cosmolopy for setup
- Cleaner imports of cosmolopy

1.4.1

January 10,2014

Enhancements

- Updated setup requirements and fixed a few tests

1.4.0

January 10, 2014

Enhancements

- Upgraded API once more: - Now Perturbations –> MassFunction
- Added transfer.py which handles all k-based quantities
- Upgraded docs significantly.

1.3.1

January 06, 2014

Bugfixes

- Fixed bug in transfer read-in introduced in 1.3.0

1.3.0

January 03, 2014

Enhancements

- A few more documentation updates (especially tools.py)
- Removed new_k_bounds function from tools.py
- Added *w* parameter to cosmolopy dictionary in *cosmo.py*
- Changed cosmography significantly to use cosmolopy in general
- Generally tidied up some of the update mechanisms.
- **API CHANGE**: cosmography.py no longer exists – I've chosen to utilise cosmolopy more heavily here.
- Added Travis CI usage

Bugfixes

- Fixed a pretty bad bug where updating h/H0 would crash the program if only one of omegab/omegac was updated alongside it
- Fixed a compatibility issue with older versions of numpy in cumulative functions

---

1.2.2

December 10, 2013

Bugfixes

- Bug in "EH" transfer function call

1.2.1

December 6, 2013

Bugfixes

- Small bugfixes to update() method

1.2.0

December 5, 2013

Features

- Addition of cosmo module, which deals with the cosmological parameters in a cleaner way

Enhancements

- Major documentation overhaul – most docstrings are now in Sphinx/numpydoc format
- Some tidying up of several functions.

1.1.10

October 29, 2013

Enhancement

- Better updating – checks if update value is actually different.
- Now performs a check to see if mass range is inside fit range.

Bugfixes

- Fixed bug in mltm property

1.1.9

October 4, 2013

Bugfixes

- Fixed some issues with n(<m) and M(<m) causing them to give NaN's

---

1.1.85

October 2, 2013

Enhancements

- The normalization of the power spectrum now saved as an attribute

---

1.1.8

September 19, 2013

Bugfixes

- Fixed small bug in SMT function which made it crash

1.1.7

September 19, 2013

Enhancements

- Updated "ST" fit to "SMT" fit to avoid confusion. "ST" is still available for now.
- Now uses trapezoid rule for integration as it is faster.

1.1.6

September 05, 2013

Enhancements

- Included an option to use delta_halo as compared to critical rather than mean density (thanks to A. Vikhlinin and anonymous referree)

Bugfixes

- Couple of bugfixes for fitting_functions.py
- Fixed mass range of Tinker (thanks to J. Tinker and anonymous referee for this)

1.1.5

September 03, 2013

Enhancements

-Added a whole suite of tests against genmf that actually work

Bugfixes

- Fixed bug in mgtm (thanks to J. Mirocha)
- Fixed an embarrassing error in Reed07 fitting function
- Fixed a bug in which dndlnm and its dependents (ngtm, etc..) were calculated wrong if dndlog10m was called first.
- Fixed error in which for some choices of M, the whole extension in ngtm would be NAN and give error

---

1.1.4

August 27, 2013

Features

- Added ability to change resolution in CAMB from hmf interface (This requires a re-install of pycamb to the newest version on the fork)

1.1.3

August 7, 2013

Features

- Added Behroozi Fit (thanks to P. Behroozi)

1.1.2

July 02, 2013

Features

- Ability to calculate fitting functions to whatever mass you want (BEWARE!!)

1.1.1

July 02, 2013

Features

- Added Eisenstein-Hu fit to the transfer function

Enhancements

- Improved docstring for Perturbations class

Bugfixes

- Corrections to Watson fitting function from latest update on arXiv (thanks to W. Watson)
- **IMPORTANT**: Fixed units for k and transfer function (Thanks to A. Knebe)

1.1.0

June 27, 2013

Enhancements

- Massive overhaul of structure: Now dependencies are tracked throughout the program, making updates even faster

---

1.0.10

June 24, 2013

Enhancements

- Added dependence on Delta_vir to Tinker

1.0.9

June 19, 2013

Bugfixes

- Fixed an error with an extra ln(10) in the mass function (quoted as dn/dlnM but actually outputting dn/dlog10M)

1.0.8

June 19, 2013

Enhancements

- Took out log10 from cumulative mass functions
- Better cumulative mass function logic

1.0.6

June 19, 2013

Bugfixes

- Fixed cumulative mass functions (extra factor of M was in there)

1.0.4

June 6, 2013

Features

- Added Bhattacharya fitting function

Bugfixes

- Fixed concatenation of list and dict issue

1.0.2

May 21, 2013

Bugfixes

- Fixed some warnings for non-updated variables passed to update()

1.0.1

May 20, 2013

Enhancements

- Added better warnings for non-updated variables passed to update()
- Made default cosmology WMAP7

0.9.99

May 10, 2013

Enhancements

- Added warning for k*R limits

Bugfixes

- Couple of minor bugfixes
- **Important** Angulo fitting function corrected (arXiv version had a typo).

0.9.97

April 15, 2013

Bugfixes

- Urgent Bugfix for updating cosmology (for transfer functions)

0.9.96

April 11, 2013

Bugfixes

- Few bugfixes

0.9.95

April 09, 2013

Features

- Added cascading variable changes for optimization
- Added the README
- Added update() function to simply change parameters using cascading approach

---

0.9.9

April 08, 2013

Features

- First version in its own package
- Added pycamb integration

Enhancements

- Removed fitting function from being a class variable
- Removed overdensity form being a class variable

0.9.7

March 18, 2013

Enhancements

- Modified set_z() so it only does calculations necessary when z changes
- Made calculation of dlnsdlnM in init since it is same for all z
- Removed mean density redshift dependence

0.9.5

March 10, 2013

Features

- The class has been in the works for almost a year now, but it currently will calculate a mass function based on any of several fitting functions.

8.5 Indices and tables

- genindex
- modindex
- search

[R21] Lukic et. al., ApJ, 2007, http://adsabs.harvard.edu/abs/2007ApJ...671.1160L

[R1] Angulo, R. E., et al., 2012. arXiv:1203.3216v1

[R2] Angulo, R. E., et al., 2012. arXiv:1203.3216v1

[R3] Behroozi, P., Weschler, R. and Conroy, C., ApJ, 2013, http://arxiv.org/abs/1207.6105

[R4] Bhattacharya, S., et al., May 2011. ApJ 732 (2), 122. http://labs.adsabs.harvard.edu/ui/abs/2011ApJ...732..122B

[R5] Courtin, J., et al., Oct. 2010. MNRAS 1931. http://doi.wiley.com/10.1111/j.1365-2966.2010.17573.x

[R6] Crocce, M., et al. MNRAS 403 (3), 1353-1367. http://doi.wiley.com/10.1111/j.1365-2966.2009.16194.x

[R7] Ishiyama, T., et al., 2015, arxiv:1412.2860

[R8] Jenkins, A. R., et al., Feb. 2001. MNRAS 321 (2), 372-384. http://doi.wiley.com/10.1046/j.1365-8711.2001.04029.x

[R9] Manera, M., et al., 2010, arxiv:0906.1314

[R10] Press, W. H., Schechter, P., 1974. ApJ 187, 425-438. http://adsabs.harvard.edu/full/1974ApJ...187..425P

[R11] Peacock, J. A., Aug. 2007. MNRAS 379 (3), 1067-1074. http://adsabs.harvard.edu/abs/2007MNRAS.379.1067P

[R12] Pillepich, A., et al., 2010, arxiv:0811.4176

[R13] Reed, D., et al., Dec. 2003. MNRAS 346 (2), 565-572. http://adsabs.harvard.edu/abs/2003MNRAS.346..565R

[R14] Reed, D. S., et al., Jan. 2007. MNRAS 374 (1), 2-15. http://adsabs.harvard.edu/abs/2007MNRAS.374....2R

[R15] Sheth, R. K., Mo, H. J., Tormen, G., May 2001. MNRAS 323 (1), 1-12. http://doi.wiley.com/10.1046/j.1365-8711.2001.04006.x

[R16] Tinker, J., et al., 2008. ApJ 688, 709-728. http://iopscience.iop.org/0004-637X/688/2/709

[R17] Tinker, J., et al., 2010. ApJ 724, 878. http://iopscience.iop.org/0004-637X/724/2/878/pdf/apj_724_2_878.pdf

[R18] Warren, M. S., et al., Aug. 2006. ApJ 646 (2), 881-885. http://adsabs.harvard.edu/abs/2006ApJ...646..881W

[R19] Watson, W. A., et al., MNRAS, 2013. http://adsabs.harvard.edu/abs/2013MNRAS.433.1230W

[R20] Watson, W. A., et al., MNRAS, 2013. http://adsabs.harvard.edu/abs/2013MNRAS.433.1230W

[R52] Nelder, J A, and R Mead. 1965. A Simplex Method for Function Minimization. The Computer Journal 7: 308-13.

[R53] Wright M H. 1996. Direct search methods: Once scorned, now respectable, in Numerical Analysis 1995: Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis (Eds. D F Griffiths and G A Watson). Addison Wesley Longman, Harlow, UK. 191-208.

[R54] Powell, M J D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal 7: 155-162.

[R55] Press W, S A Teukolsky, W T Vetterling and B P Flannery. Numerical Recipes (any edition), Cambridge University Press.

[R56] Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.

[R57] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208.

[R58] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software 23 (4): 550-560.

[R59] Nash, S G. Newton-Type Minimization Via the Lanczos Method. 1984. SIAM Journal of Numerical Analysis 21: 770-778.

[R60] Powell, M J D. A direct search optimization method that models the objective and constraint functions by linear interpolation. 1994. Advances in Optimization and Numerical Analysis, eds. S. Gomez and J-P Hennart, Kluwer Academic (Dordrecht), 51-67.

h

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z