hmf Documentation

Release 3.0.2

Steven Murray

Nov 03, 2018

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.

> **Warning:** Due to the general trend of moving to Python 3 by important projects such as IPython and astropy, from version 3.0, hmf is compatible with Python 3, and from version 3.1, it will drop (official) support for Python 2.

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.
- Python 2 and 3 compatible

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*. Explicitly, the dependencies are numpy, scipy, astropy and camb.

You will only need *emcee* if you are going to be using the fitting capabilities of *hmf*.

Finally the *hmf* package needs to be installed: `pip install hmf`.

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 Tinker (2008) 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: @steven-murray

CHAPTER 7

Contributors

Jordan Mirocha (UCLA): @mirochaj

CHAPTER 8

Comments, corrections and suggestions

Chris Power (UWA) Aaron Robotham (UWA): @asgr Alexander Knebe (UAMadrid) Peter Behroozi (UC Berkeley)

CHAPTER 9

Contents

9.1 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.

9.1.1 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:

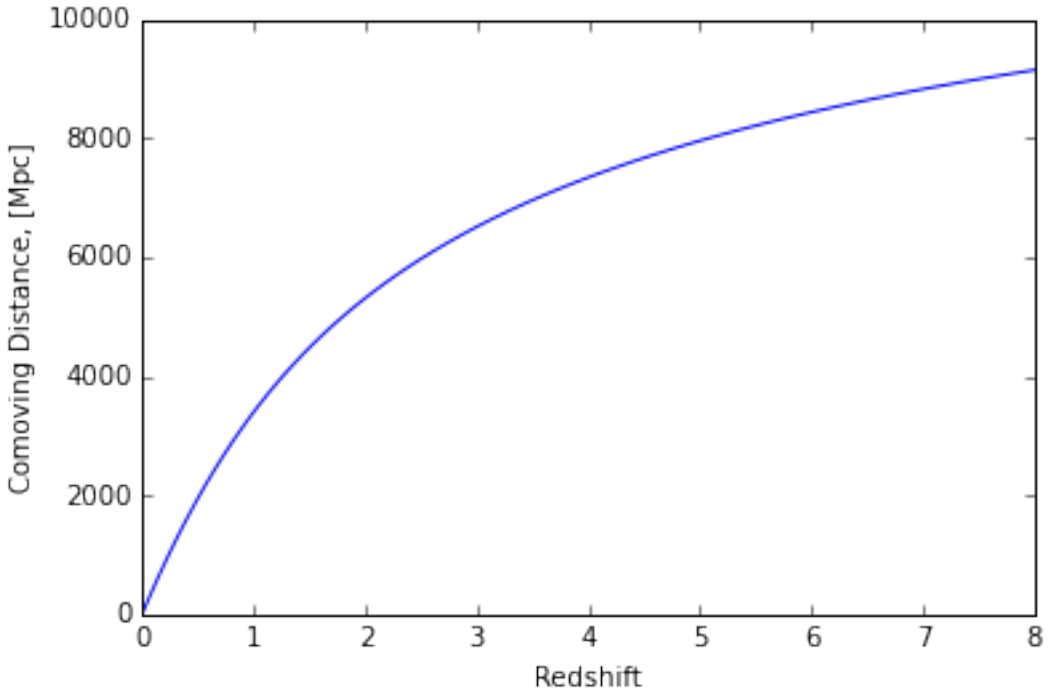
```
print "Matter density: ", my_cosmo.cosmo.Om0
print "Hubble constant: ", my_cosmo.cosmo.H0
print "Dark Energy density: ", my_cosmo.cosmo.Ode0
print "Baryon density: ",  my_cosmo.cosmo.Ob0
print "Curvature density: ", my_cosmo.cosmo.Ok0
```

```
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}
```

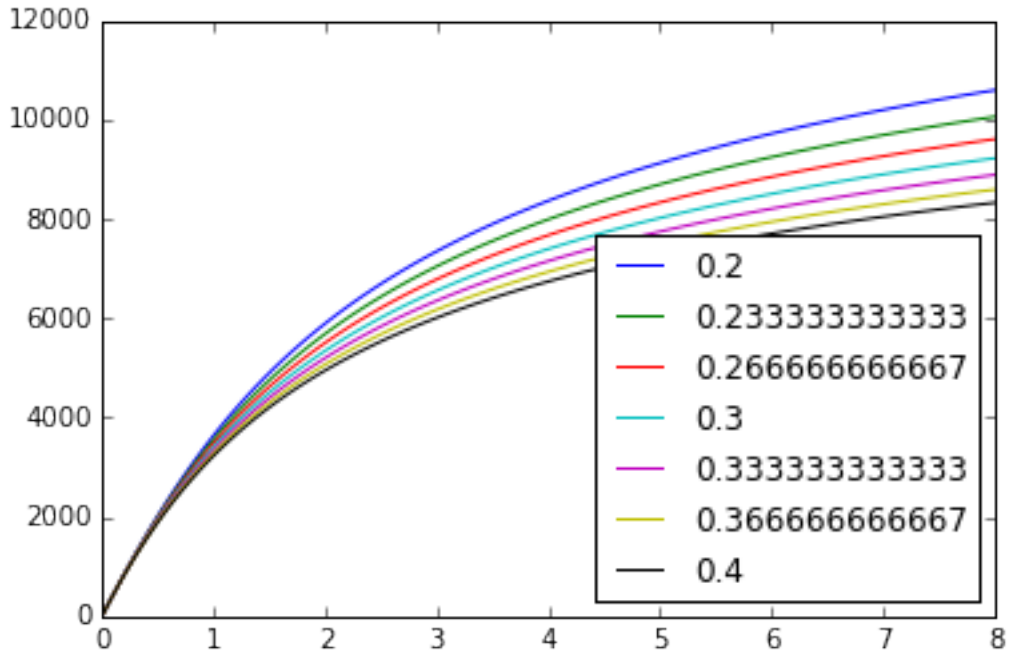
Of course, if we were to update the model to a Flat Lambda CDM model, then the `Ode0` keyword would give an error. To facilitate this, passing an empty dictionary clears all custom values:

```
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)
```



9.1.2 Mass Definitions

`hmf`, as of v3.1, provides a simple way of converting between halo mass definitions, which serves two basic purposes:

1. the mass of a halo in one definition can be converted to its appropriate mass under another definition
2. the halo mass function can be converted between mass definitions.

Introduction

By "mass definition" we mean the way the extent of a halo is defined. In `hmf`, we support two main kinds of definition, which themselves can contain different models. In brief, `hmf` supports Friends-of-Friends (FoF) halos, which are defined via their linking length, b, and spherical-overdensity (SO) halos, which are defined by two criteria: an overdensity Δ_h, and an indicator of what that overdensity is with respect to (usually mean background density ρ_m, or critical density ρ_c). In addition to being able to provide a precise overdensity for a SO halo, `hmf` provides a way to use the so-called "virial" overdensity, as defined in Bryan and Norman (1998).

Converting between SO mass definitions is relatively simple: given a halo profile and concentration for the given halo mass, determine the concentration required to make that profile contain the desired density, and then compute the mass of the halo under such a concentration.

There is no clear way to perform such a conversion for FoF halos. Nevertheless, if one assumes that all linked particles are the same mass, and that halos are spherical and singular isothermal spheres (cf. White (2001)), one can approximate an FoF halo by an SO halo of density $9\rho_m/(2\pi b^3)$. `hmf` will make this approximation if a conversion between mass definitions is desired.

Changing Mass Definitions

Most of the functionality concerning mass definitions is defined in the `hmf.halos.mass_definitions` module:

```
In [1]: import hmf
        from hmf.halos import mass_definitions as md
```

```
        print("Using hmf version v%s"%hmf.__version__)
```

```
Using hmf version v3.0.2
```

While we're at it, import matplotlib and co:

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
        import inspect
```

Different mass definitions exist inside the `mass_definitions` module as `Components`. All definitions are subclassed from the abstract `MassDefinition` class:

```
In [3]: [x[1] for x in inspect.getmembers(md, inspect.isclass) if issubclass(x[1], md.MassDefinition)
```

```
Out[3]: [hmf.halos.mass_definitions.FOF,
         hmf.halos.mass_definitions.MassDefinition,
         hmf.halos.mass_definitions.SOCritical,
         hmf.halos.mass_definitions.SOMean,
         hmf.halos.mass_definitions.SOVirial,
         hmf.halos.mass_definitions.SphericalOverdensity]
```

The Mass Definition Component

To create an instance of any class, optional `cosmo` and `z` arguments can be specified. By default, these are the `Planck15` cosmology at redshift 0. We'll leave them as default for this example. Let's define two mass definitions, both spherical-overdensity definitions with respect to the mean background density:

```
In [4]: mdef_1 = md.SOMean(overdensity=200)
        mdef_2 = md.SOMean(overdensity=500)
```

Each mass definition has its own `model_parameters`, which define the exact overdensity. For both the `SOMean` and `SOCritical` definitions, the `overdensity` can be provided, as above (default is 200). This must be passed as a named argument. For the `FOF` definition, the `linking_length` can be passed (default 0.2). For the `SOVirial`, no parameters are available. Available parameters and their defaults can be checked the same way as *any* `Component` within `hmf`:

```
In [5]: md.SOMean._defaults
```

```
Out[5]: {'overdensity': 200}
```

The explicit halo density for a given mass definition can be accessed:

```
In [6]: mdef_1.halo_density, mdef_1.halo_density/mdef_1.mean_density
```

```
Out[6]: (17068502575484.857, 200.0)
```

Converting Masses

To convert a mass in one definition to a mass in another, use the following:

```
In [7]: mnew, rnew, cnew = mdef_1.change_definition(m=1e12, mdef=mdef_2)
        print("Mass in new definition is %.2f x 10^12 Msun / h"%(mnew/1e12))
        print("Radius of halo in new definition is %.2f Mpc/h"%rnew)
        print("Concentration of halo in new definition is %.2f"%cnew)
```

```
Mass in new definition is 0.76 x 10^12 Msun / h
Radius of halo in new definition is 0.16 Mpc/h
Concentration of halo in new definition is 4.81
```

The input mass argument can be a list or array of masses also. To convert between masses, the concentration of the input halos, and their density profile, must be known. By default, an NFW profile is assumed, with a concentration-mass relation from Duffy et al. (2008).

One can alternatively pass a concentration directly:

```
In [8]: mnew, rnew, cnew = mdef_1.change_definition(m=1e12, mdef=mdef_2, c = 5.0)
        print("Mass in new definition is %.2f x 10^12 Msun / h"%(mnew/1e12))
        print("Radius of halo in new definition is %.2f Mpc/h"%rnew)
        print("Concentration of halo in new definition is %.2f"%cnew)
```

```
Mass in new definition is 0.72 x 10^12 Msun / h
Radius of halo in new definition is 0.16 Mpc/h
Concentration of halo in new definition is 3.31
```

If you have `halomod` installed, you can also pass any `halomod.profiles.Profile` instance (which itself includes a concentration-mass relation) as the `profile` argument.

Converting Mass Functions

All halo mass function fits are measured using halos found using some halo definition. While some fits explicitly include a parameterization for the spherical overdensity of the halo (eg. `Tinker08` and `Watson`), others do not.

By passing a mass definition to the `MassFunction` constructor, `hmf` will attempt to convert the mass function defined by the chosen fitting function to the appropriate mass definition, by solving

$$\int_\infty^{m_{\rm old}} dm\ n(m) = \int_\infty^{m_{\rm new}(m_{\rm old})} dm\ n'(m) \tag{9.1}$$

for $n'(m)$, resulting in

$$n'(m) = n(m_{\rm old}(m_{\rm new})) \left(\frac{dm_{\rm new}}{dm_{\rm old}}\right)^{-1}. \tag{9.2}$$

By default, the mass definition is `None`, which turns off all mass conversion and uses whatever mass definition was employed by the chosen fit. This keeps all existing code running as it was previously. Nevertheless, care should be taken here: many of the different fits have different mass definitions, and should not be compared without some form of mass conversion. To see the mass definition intrinsic to the measurement of each fit, use the following:

```
In [9]: from hmf.mass_function.fitting_functions import SMT, Tinker08, Jenkins
```

```
In [10]: print(SMT.sim_definition.halo_finder_type, SMT.sim_definition.halo_overdensity)
         print(Tinker08.sim_definition.halo_finder_type, Tinker08.sim_definition.halo_overdensity)
         print(Jenkins.sim_definition.halo_finder_type, Jenkins.sim_definition.halo_overdensity)
```

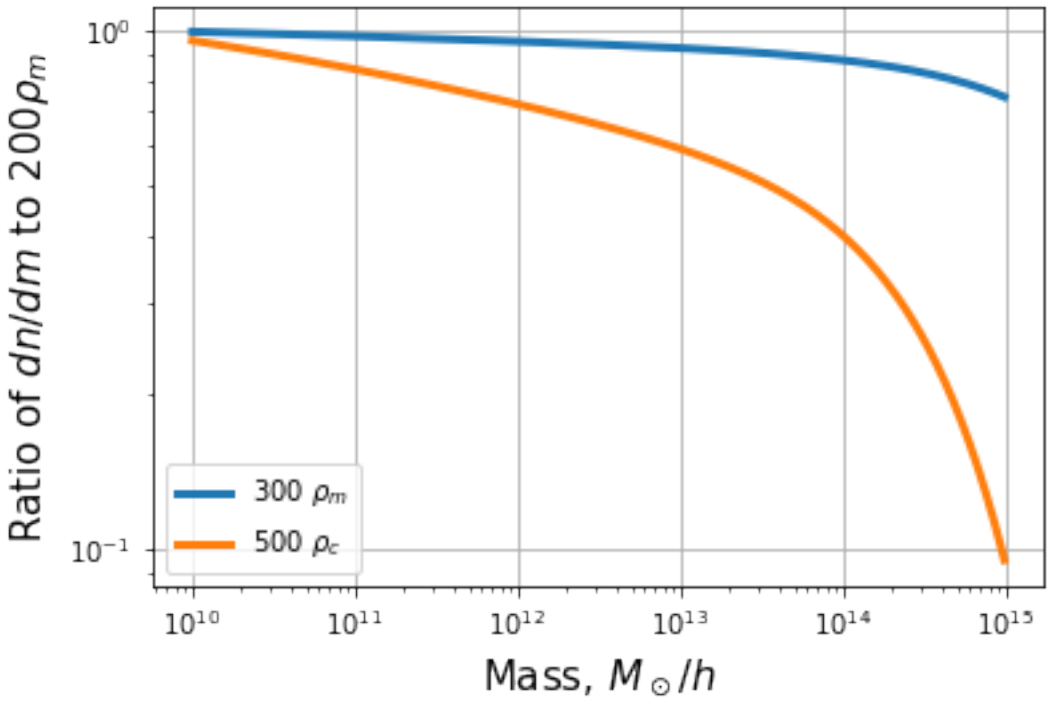
```
SO vir
SO *
FoF 0.2
```

Here "vir" corresponds to the virial definition of Bryan and Norman (1998), an asterisk indicates that the fit is itself parameterized for SO mass definitions, and 0.2 is the FoF linking length.

Let's convert the mass definition of `Tinker08`, which has an intrinsic parameterization:

```
In [21]: # Default mass function object
         mf = hmf.MassFunction()

         dndm0 = mf.dndm

         # Change the mass definition to 300rho_mean
         mf.update(
```

```
    mdef_model   = "SOMean",
    mdef_params = {
        "overdensity": 300
    }
)

plt.plot(mf.m, mf.dndm/dndm0, label=r"300 $\rho_m$", lw=3)

# Change the mass definition to 500rho_crit
mf.update(
    mdef_model   = "SOCritical",
    mdef_params = {
        "overdensity": 500
    }
)

plt.plot(mf.m, mf.dndm/dndm0, label=r"500 $\rho_c$", lw=3)

plt.xscale('log')
plt.yscale('log')
plt.xlabel(r"Mass, $M_\odot/h$", fontsize=15)
plt.ylabel(r"Ratio of $dn/dm$ to $200 \rho_m$", fontsize=15)
plt.grid(True)
plt.legend();
```

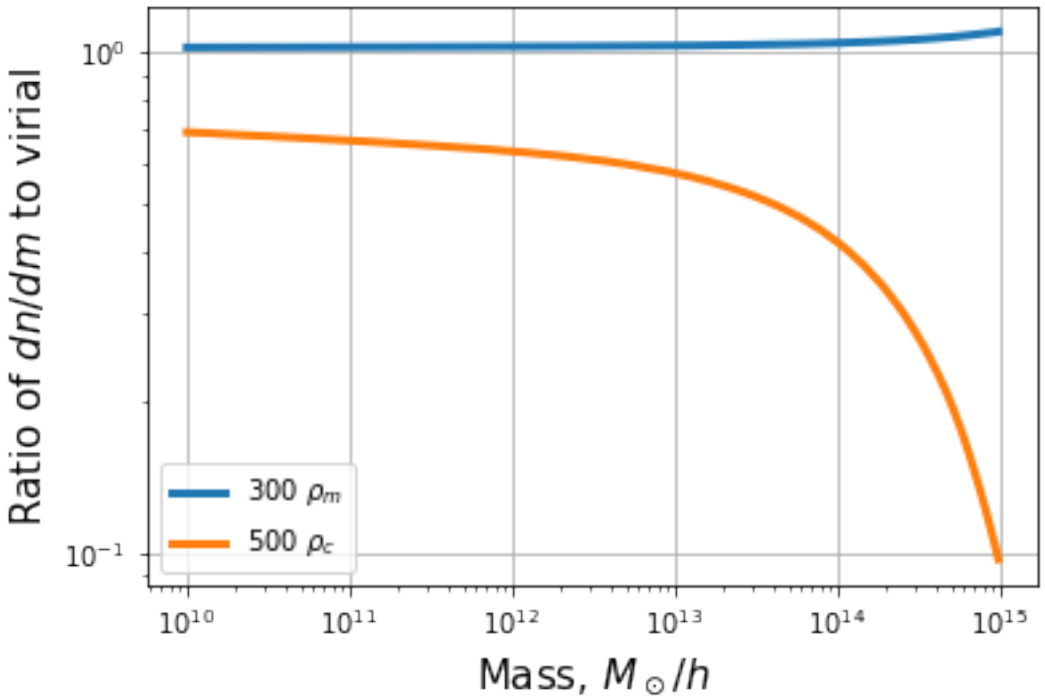


This did not require any internal conversion. Let's try converting a fit that has no explicit parameterization for overdensity:

```
In [27]: # Default mass function object.
         mf = hmf.MassFunction(hmf_model = "SMT", Mmax=15)
         dndm0 = mf.dndm

         # Change the mass definition to 300rho_mean
```

```
mf.update(
    mdef_model  = "SOMean",
    mdef_params = {
        "overdensity": 300
    }
)
plt.plot(mf.m, mf.dndm/dndm0, label=r"300 $\rho_m$", lw=3)


# Change the mass definition to 500rho_crit
mf.update(
    mdef_model  = "SOCritical",
    mdef_params = {
        "overdensity": 500
    }
)
plt.plot(mf.m, mf.dndm/dndm0, label=r"500 $\rho_c$", lw=3)

plt.xscale('log')
plt.yscale('log')
plt.xlabel(r"Mass, $M_\odot/h$", fontsize=15)
plt.ylabel(r"Ratio of $dn/dm$ to virial", fontsize=15)
plt.grid(True)
plt.legend();
```

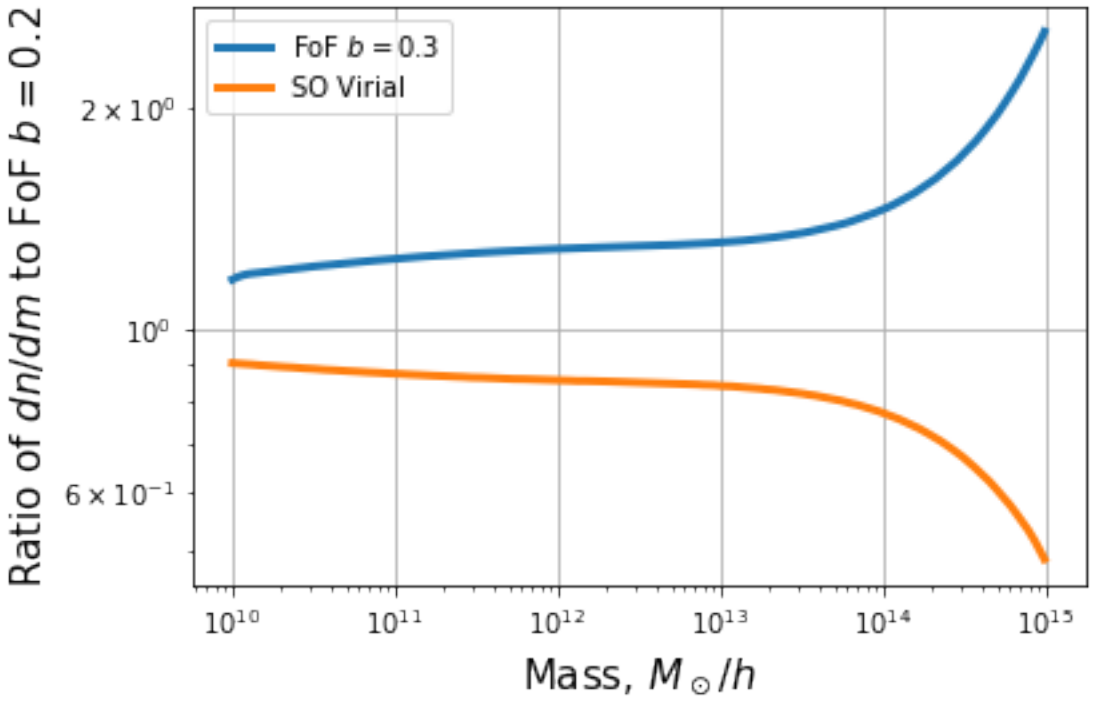
```
332.2445055922226 0.0
300
```



Here, the measured mass function uses "virial" halos, which have an overdensity of ~330ρ_m, so that converting to 300ρ_m is a down-conversion of density. Finally, we convert a FoF mass function:

```
In [33]: # Default mass function object.
         mf = hmf.MassFunction(hmf_model = "Jenkins")
         dndm0 = mf.dndm
```

```
# Change the mass definition to 300rho_mean
mf.update(
    mdef_model  = "FOF",
    mdef_params = {
        "linking_length": 0.3
    }
)
plt.plot(mf.m, mf.dndm/dndm0, label=r"FoF $b=0.3$", lw=3)


# Change the mass definition to 500rho_crit
mf.update(
    mdef_model  = "SOVirial",
    mdef_params = {}    # NOTE: we need to pass an empty dict here
                        # so that the "linking_length" argument is removed.
)
plt.plot(mf.m, mf.dndm/dndm0, label=r"SO Virial", lw=3)

plt.xscale('log')
plt.yscale('log')
plt.xlabel(r"Mass, $M_\odot/h$", fontsize=15)
plt.ylabel(r"Ratio of $dn/dm$ to FoF $b=0.2$", fontsize=15)
plt.grid(True)
plt.legend();
```



9.2 License

9.3 API Summary

`hmf.cosmology.cosmo`
`hmf.cosmology.growth_factor`
`hmf.density_field.transfer`
`hmf.density_field.transfer_models`
`hmf.density_field.halofit`
`hmf.density_field.filters`
`hmf.halos.mass_definitions`
`hmf.mass_function.hmf`
`hmf.mass_function.fitting_functions`
`hmf.mass_function.integrate_hmf`
`hmf.alternatives.wdm`
`hmf.helpers.sample`
`hmf.helpers.functional`

9.4 Releases

9.4.1 dev

Features

- Added new `CambGrowth` growth factor model, which computes the growth using CAMB. This is useful especially when using w > -1, for which the other growth factor models are inadequate. Solves issue #19 raised by @tijmen.
- Added new module `mass_definitions` which more robustly deals with various halo mass definitions, and also includes ability to convert mass functions between different definitions.

Enhancement

- Added *get_dependencies* method to _Framework, to enable finding all parameters that a quantity depends on.

Bugfixes

- When using camb for the transfer function, some cosmologies would lead to a segfault (i.e. when Ob0 or Tcmb0 are not set explicitly). This now raises a helpful error.

Internals

- Removed logging, which was redundant.

- Moved from nose to pytest
- Significant overhaul of package structure to more modularised form.

9.4.2 v3.0.3 [1st Dec 2017]

Bugfixes

- Fixed usage of deprecated MsolMass in wdm
- Fixed Bhattachrya fitting function (thanks to Benedikt Diemer!)
- Fixed typo in Watson fitting function (thanks to Benedikt Diemer!)
- Update cosmo test to use new Astropy constants.
- Fixed issue with sampling function where zeros in ngtm would yield an error.

9.4.3 v3.0.2 [3rd Nov 2017]

Bugfixes

- Changed parameter checks on instantiation to post-conversion.

9.4.4 v3.0.1 [31st Oct 2017]

Enhancement

- Normalised all <>_model properties to be actual classes, rather than either class or string.
- Added consistent checking of dictionaries for <>_params parameters.

9.4.5 v3.0.0 [7th June 2017]

Features

- Now provides compatibility with Python 3.x. Support for 2.x will be removed in hmf v3.1 (whenever that comes).
- Complete overhaul of the caching system. Should be invisible to the user, but streamlines writing of framework code considerably. Removes required manual specification of dependencies between quantities, and adds ability to specify parameter kinds (model, param, res, option, switch).

Bugfixes

- Fixed bug in Caroll1992 GrowthFactor class which affected high-redshift growth factors (thanks to Olmo Piana).
- Fixed astropy dependency to be >= 1.1
- Fixed bug where Takahashi parameters were always passed through regardless of `takahashi` setting.
- Fixed small bug where the functional.get_label method returned differently ordered parameters because of dicts.
- Note that the fitting subpackage is temporarily unsupported and I discourage its use for the time being.

Enhancement

- Completely removes dependence on archaic pycamb package. Now supports natively supplied python interface to CAMB. Install camb with `pip install --egg camb`. This means that much more modern versions of CAMB can be used.

- Many new tests, to bring total coverage up to >80%, and continuous testing on Python 2.7, 3.5 and 3.6

9.4.6 v2.0.5 [12th January 2017]

Bugfixes

- Fixed bug in GrowthFactor which gave ripples in functions of z when a coarse grid was used. Thanks to @mirochaj and @thomasguillet!

Enhancments

- Streamlined the caching framework a bit (removing cruft)
- Totally removed dependency on the Cache (super)class – caching indexes now inherent to the called class.
- More robust parameter information based on introspection.

9.4.7 v2.0.4 [11th November, 2016]

Bugfixes

- **IMPORTANT**: Fixed a bug in which updating the cosmology after creation did not update the transfer function.

9.4.8 v2.0.3 [22nd September, 2016]

Bugfixes

- SharpK filter integrated over incorrect range of k, now fixed.

Enhancments

- WDM models now more consistent with MassFunction API.
- Better warning in HALOFIT module when derivatives don't work first time.

9.4.9 v2.0.2 [2nd August, 2016]

Features

- Added a bunch of information to each hmf_model, indicating simulation parameters from which the fit was derived.
- Added `FromArray` transfer model, allowing an array to be passed programmatically for k and T.
- Added `Carroll1992` growth factor approximation model.

Enhancments

- how_big now gives the boxsize required to expect at least one halo above m in 95% of boxes.

Bugfixes

- Removed unnecessary multiplication by 1e6 in cosmo.py (thanks @iw381)
- **IMPORTANT**: normalisation now calculated using convergent limits on k, rather than user-supplied values.
- **IMPORTANT**: fixed bug in Bhattacharya fit, which was multiplying by an extra delta_c/sigma.
- fixed issue with `nonlinear_mass` raising exception when mass outside given mass range.

9.4.10 v2.0.1 [2nd May, 2016]

Bugfixes

- Corrects the default sigma_8 and n (spectral index) parameters to be from Planck15 (previously from Planck13), which corresponds to the default cosmology. **NOTE:** this will change user-code output silently unless sigma_8 and n are explicitly set.

9.4.11 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

Enhancments

- 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

9.4.12 v1.8.0 [February 2, 2015]

Features

- Better WDM models
- Added SharpK and SharpKEllipsoid filters and overhauled filter system.

Enhancments

- Separated WDM models from main class for extendibility
- Enhanced caching to deal with subclassing

Bugfixes

- Minor bugfixes

9.4.13 1.7.1 [January 28, 2015]

Enhancments

- Added warning to docstring of _dlnsdlnm and n_eff for non-physical oscillations.

9.4.14 1.7.0 [October 28, 2014]

Features

- Very much updated fitting routines, in class structure
- Made fitting_functions more flexible and model-like.

Enhancments

- Modified get_hmf to be more general
- Moved get_hmf and related functions to "functional.py"

9.4.15 1.6.2 [September 16, 2014]

Features

- New HALOFIT option for original co-oefficients from Smith+03

Enhancments

- 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.

9.4.16 1.6.1 [September 8, 2014]

Enhancments

- 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

9.4.17 1.6.0 [August 19, 2014]

Features

- New Tinker10 fit (Tinker renamed Tinker08, but Tinker still available)

Enhancments

- Completely re-worked caching module to be easier to code and faster.
- Better Cosmology class – more input combinations available.

Bugfixes

- Fixed all tests.

9.4.18 1.5.0 [May 08, 2014]

Features

- Introduced _cache module: Extracts all caching logic to a separate module which defines decorators – much simpler coding!

9.4.19 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.

Enhancments

- 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.

9.4.20 1.4.4 [January 23, 2014]

Features

- Added ability to change the default cosmology parameters

Enhancments

- 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.

9.4.21 1.4.3 [January 10, 2014]

Bugfixes

- Changed license in setup

9.4.22 1.4.2 [January 10, 2014]

Enhancments

- Mocked imports of cosmolopy for setup
- Cleaner imports of cosmolopy

9.4.23 1.4.1 [January 10,2014]

Enhancments

- Updated setup requirements and fixed a few tests

9.4.24 1.4.0 [January 10, 2014]

Enhancments

- Upgraded API once more: - Now Perturbations –> MassFunction
- Added transfer.py which handles all k-based quantities
- Upgraded docs significantly.

9.4.25 1.3.1 [January 06, 2014]

Bugfixes

- Fixed bug in transfer read-in introduced in 1.3.0

9.4.26 1.3.0 [January 03, 2014]

Enhancments

- 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

---

9.4.27 1.2.2 [December 10, 2013]

Bugfixes

- Bug in "EH" transfer function call

9.4.28 1.2.1 [December 6, 2013]

Bugfixes

- Small bugfixes to update() method

9.4.29 1.2.0 [December 5, 2013]

Features

- Addition of cosmo module, which deals with the cosmological parameters in a cleaner way

Enhancments

- Major documentation overhaul – most docstrings are now in Sphinx/numpydoc format
- Some tidying up of several functions.

9.4.30 1.1.10 [October 29, 2013]

Enhancements - 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

9.4.31 1.1.9 [October 4, 2013]

Bugfixes

- Fixed some issues with n(<m) and M(<m) causing them to give NaN's

---

9.4.32 1.1.85 [October 2, 2013]

Enhancments

- The normalization of the power spectrum now saved as an attribute

---

9.4.33 1.1.8 [September 19, 2013]

Bugfixes

- Fixed small bug in SMT function which made it crash

9.4.34 1.1.7 [September 19, 2013]

Enhancments

- 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.

9.4.35 1.1.6 [September 05, 2013]

Enhancments

- 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)

9.4.36 1.1.5 [September 03, 2013]

Enhancments

-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

---

9.4.37 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)

9.4.38 1.1.3 [August 7, 2013]

Features

- Added Behroozi Fit (thanks to P. Behroozi)

9.4.39 1.1.2 [July 02, 2013]

Features

- Ability to calculate fitting functions to whatever mass you want (BEWARE!!)

9.4.40 1.1.1 [July 02, 2013]

Features

- Added Eisenstein-Hu fit to the transfer function

Enhancments

- 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)

9.4.41 1.1.0 [June 27, 2013]

Enhancments

- Massive overhaul of structure: Now dependencies are tracked throughout the program, making updates even faster

---

9.4.42 1.0.10 [June 24, 2013]

Enhancments

- Added dependence on Delta_vir to Tinker

9.4.43 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)

9.4.44 1.0.8 [June 19, 2013]

Enhancments

- Took out log10 from cumulative mass functions
- Better cumulative mass function logic

9.4.45 1.0.6 [June 19, 2013]

Bugfixes

- Fixed cumulative mass functions (extra factor of M was in there)

9.4.46 1.0.4 [June 6, 2013]

Features

- Added Bhattacharya fitting function

Bugfixes

- Fixed concatenation of list and dict issue

9.4.47 1.0.2 [May 21, 2013]

Bugfixes

- Fixed some warnings for non-updated variables passed to update()

9.4.48 1.0.1 [May 20, 2013]

Enhancments

- Added better warnings for non-updated variables passed to update()
- Made default cosmology WMAP7

9.4.49 0.9.99 [May 10, 2013]

Enhancments

- Added warning for k*R limits

Bugfixes

- Couple of minor bugfixes
- **Important** Angulo fitting function corrected (arXiv version had a typo).

9.4.50 0.9.97 [April 15, 2013]

Bugfixes

- Urgent Bugfix for updating cosmology (for transfer functions)

9.4.51 0.9.96 [April 11, 2013]

Bugfixes

- Few bugfixes

9.4.52 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

---

9.4.53 0.9.9 [April 08, 2013]

Features

- First version in its own package
- Added pycamb integration

Enhancments

- Removed fitting function from being a class variable
- Removed overdensity form being a class variable

9.4.54 0.9.7 [March 18, 2013]

Enhancments

- 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

9.4.55 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.

9.5 Indices and tables

- genindex
- modindex
- search