mrpy Documentation

Release 1.1.0

Steven Murray

May 02, 2018

Contents

1 Documentation **3**
1.1 Quick Start . . . 3

2 Installation **5**

3 Getting Started **7**
3.1 Features . . . 7
3.2 Examples . . . 7
3.3 Acknowledging . . . 8
3.4 Contents . . . 8
3.5 Indices and tables . . . 126

Python Module Index **127**

A Python package for calculations with the MRP parameterisation of the Halo Mass Function.

See Murray, Robotham, Power (2018) for more details on what the MRP is.

CHAPTER 1

Documentation

Read the docs.

1.1 Quick Start

CHAPTER 2

Installation

```
>> pip install mrpy.
```

This should install the required dependencies automatically.

Note, to use the MCMC fitting features, *emcee* is needed. This is *not* installed automatically.

To get the bleeding edge, use `pip install git+git://github.com/steven-murray/mrpy.git`.

CHAPTER 3

Getting Started

There's a lot of things that you can do with *mrpy*. What you require will depend on the problem at hand. We recommend looking at some of the examples, and the API itself for how to use the code.

3.1 Features

With *mrpy* you can:

- Calculate basic statistics of the truncated generalised gamma distribution (TGGD) with the *TGGD* class: mean, mode, variance, skewness, pdf, cdf, generate random variates etc.
- Generate MRP quantities with the *MRP* class: differential number counts, cumulative number counts, various methods for generating normalisations.
- Generate the MRP-based halo mass function as a function of physical parameters via the *mrp_b13* function.
- Fit MRP parameters to data in the form of arbitrary curves with the *get_fit_curve* function.
- Fit MRP parameters to data in the form of a sample of variates with the *SimFit* class: simulation data is supported with extra efficiency, simulation suites fitted simultaneously is also supported, arbitrary priors on parameters, log-normal uncertainties on variates supported.
- Calculate analytic hessians, jacobians at any point (including the solution of a fit).
- Use alternate parameterisations of the same form via the *reparameterise* module.
- Work with a special entirely analytic model to understand the effects of various parameters in the *analytic_model* module.

3.2 Examples

There are several examples featured in the `docs/examples` directory of the github repository. These can also be found in the official documentation.

3.3 Acknowledging

If you use this code in your work, please cite Murray, Robotham, Power (2018) and/or http://ascl.net/1802.015 (whichever is more appropriate). Also consider starring/following the repo on github so we know how much it is being used. We would also love any input to the code!

3.4 Contents

3.4.1 Examples

To help get you started using `mrpy`, we've compiled a few examples. In fact, many of these examples are used explicitly to produce the figures which appear in the MRP paper (Murray, Robotham, Power 2018). Other simple examples can be found in the API documentation for each object.

So, what would you like to learn?

How to get started

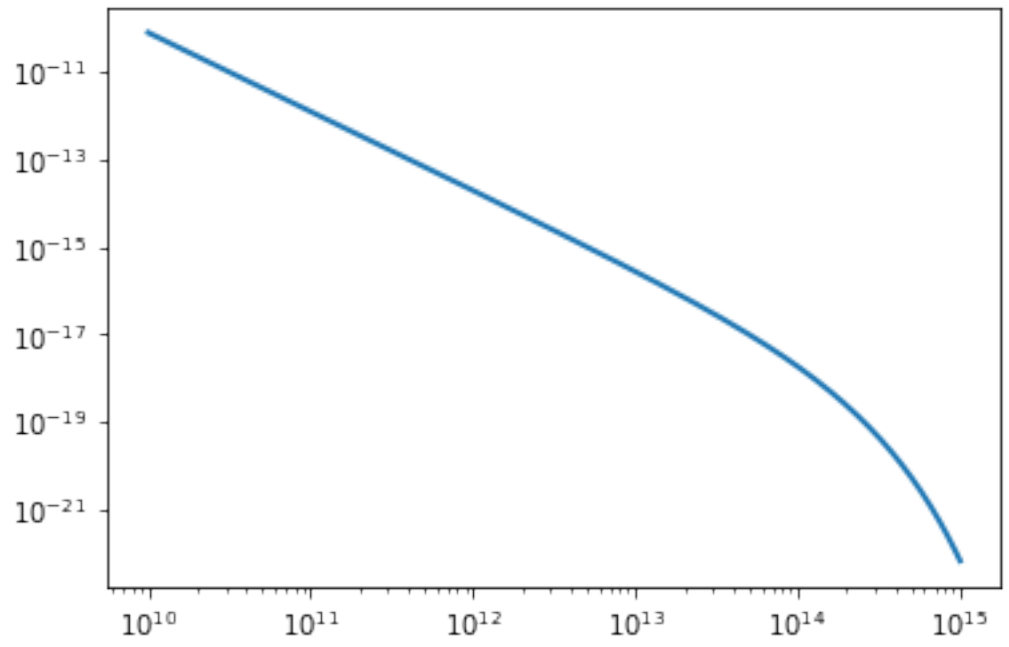
This series of examples shows the very basics of how to get started with MRPY, using the different functionality. These examples aren't "real world" ones, just toy ones to show the basic idea.

```
In [1]: # General Imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
```

Core Functionality

Core functionality (i.e. calculation of the MRP function given input parameters, plus some other functions useful for normalising) is in the `core` module:

```
In [2]: from mrpy import dndm       # resides in the base.core module, but imported into top-level nam
        m = np.logspace(10,15,500) # Create an array of masses
        dn = dndm(m, logHs=14.0, alpha=-1.8, beta=0.85) # Create the MRP mass function
        plt.plot(m,dn,lw=2)
        plt.xscale('log')
        plt.yscale('log')
```

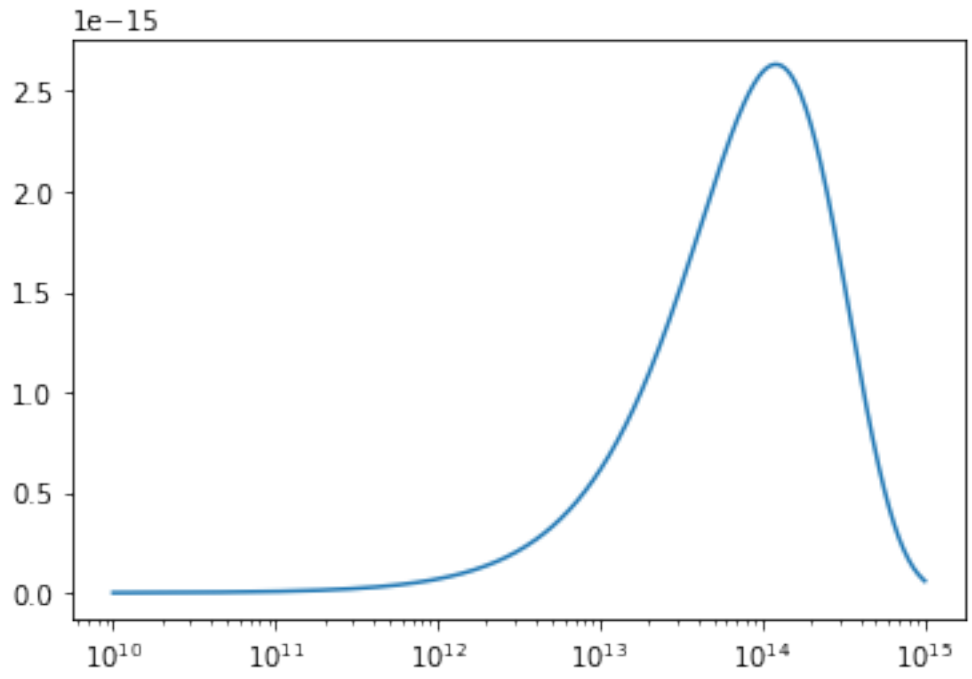


Pure Stats

If you don't care so much about the fact that the MRP is good for halo mass functions (or don't know what a halo mass function is...), but want to use the statistical distribution, you'll want the `stats` module. It contains an object called `TGGD` (short for Truncated Generalised Gamma Distribution), which has many statistical quantities and methods available (such as producing random variates, mean, mode etc.)

```
In [3]: from mrpy import TGGD
        tggd = TGGD(scale=1e14,a=1.0,b=0.85,xmin=1e10)
        print "Mean: ", tggd.mean  # Mean of the distribution
        print "Mode: ", tggd.mode  # Mode of the distirbution
        print "Variance: ", tggd.variance # Variance of the distribution
        print "Mean of sample: ", np.mean(tggd.rvs(1e5))  #Produce 1e5 random variates and take the m
        plt.plot(m, tggd.pdf(m))      #Plot the PDF
        plt.xscale('log')
```

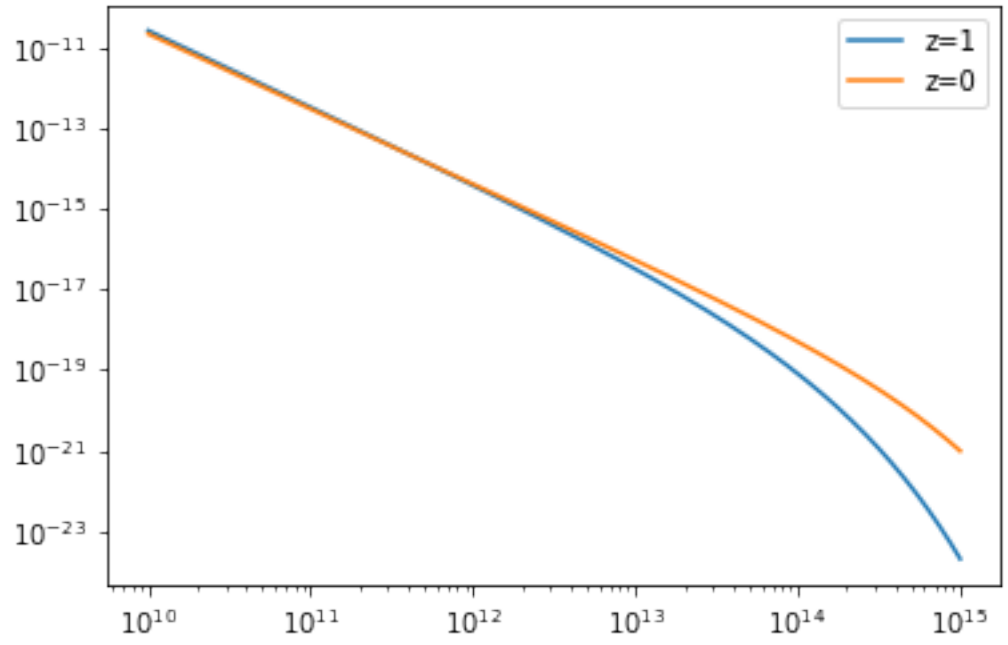
```
Mean:  2.84867015869e+14
Mode:  1.21070004341e+14
Variance:  4.79705281452e+28
Mean of sample:  2.83287193279e+14
```



Physical Dependence

The `physical_dependence` module contains a counterpart to the basic `dndm` function, called `mrp_b13`, which returns the best-fit MRP according to input physical variables (redshift, matter density, rms mass variance). These are derived from fits to the theoretical mass function of Behroozi+2013.

```
In [4]: from mrpy import mrp_b13
        dndm_z1 = mrp_b13(m,z=1)            # HMF at redshift 1
        dndm_z0 = mrp_b13(m,sigma_8=0.85)   # HMF at redshift 0 but sigma_8=0.85
        plt.plot(m,dndm_z1,label="z=1")
        plt.plot(m,dndm_z0,label="z=0")
        plt.legend(loc=0)
        plt.xscale('log')
        plt.yscale('log')
```



Fitting MRP

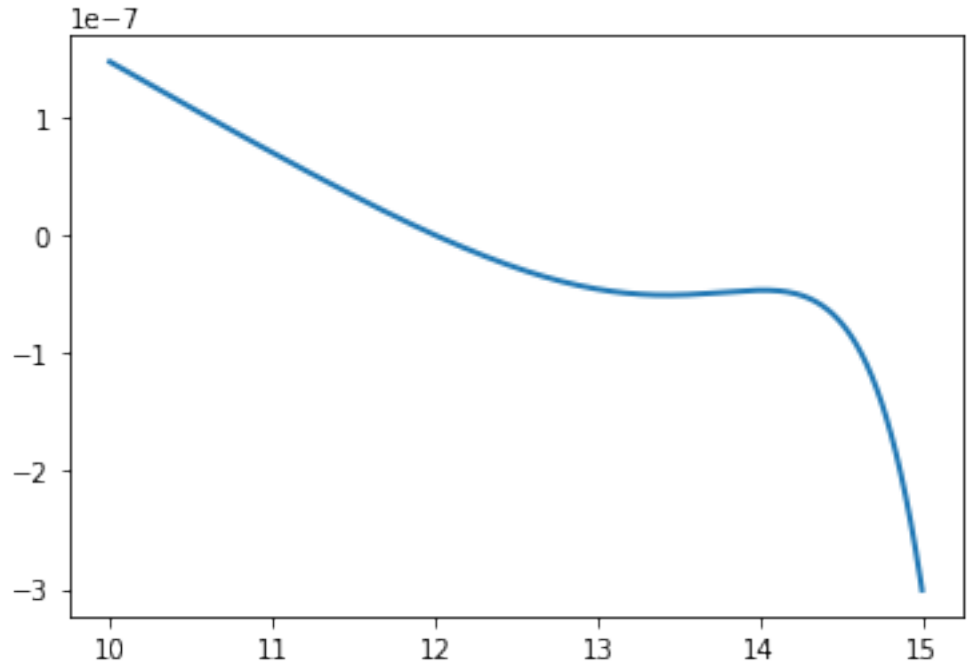
Fitting Curve Data

The `fit_curve` module contains routines to fit the MRP to binned/curve data. This can be a theoretical curve, or binned halos (or other variates). There are several options available, and the gradient of the objective function is specified analytically to improve performance. See Murray, Robotham, Power, 2016 (in prep.) for more details.

```
In [5]: from mrpy.fitting.fit_curve import get_fit_curve
        dn = dndm(m,logHs=14.0,alpha=-1.9,beta=0.75,norm=1)
        result, curve_obj = get_fit_curve(m,dn,[14.5, -1.8, 0.85,0.],
                                          bounds = [[None,None], [-2,-1.5], [0.2,1.5], [None,None]])
        print result
        plt.plot(curve_obj.logm,curve_obj.dndm()/dn-1,lw=2)
```

```
      fun: 1.6704623362075916e-12
 hess_inv: <4x4 LbfgsInvHessProduct with dtype=float64>
      jac: array([ -8.60315661e-05,  -1.13296916e-04,   6.03274702e-05,
        -3.79420248e-06])
  message: 'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 28
      nit: 25
   status: 0
  success: True
        x: array([  1.40000001e+01,  -1.90000003e+00,   7.50000058e-01,
        -5.62234817e-07])
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x7f072e383dd0>]
```



In the previous example, we simply fit the four MRP parameters to the input curve. Options can be specified to constrain the normalisation via the known mean density of the Universe (see MRP, 2017 Appendix C.1 for details).

Fitting Samples

To fit actual samples of halos, use the `fit_sample` module. At this stage, only samples with no measurement uncertainties, and a constant volume (per subsample) are supported. Usefully, this covers the case of output halos from simulations. Within this context, either a downhill-gradient method or MCMC can be used.

We hope to provide more general fitting scenarios via configuration with other packages in the future.

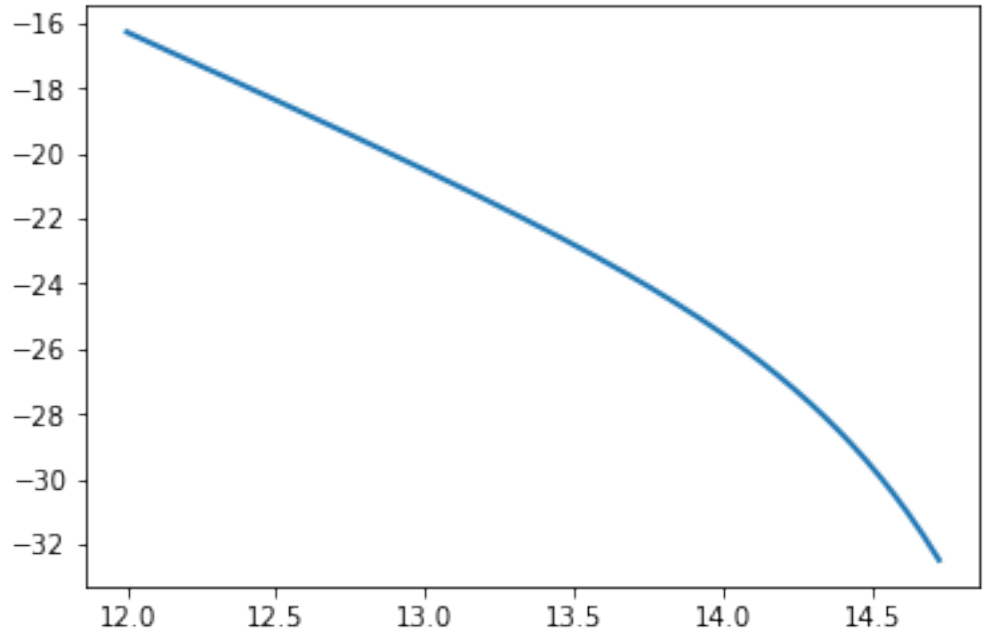
An example of using the downhill method is as follows:

```
In [6]: from mrpy.base.stats import TGGD
        from mrpy.fitting.fit_sample import SimFit

        # Create some mock data to fit
        r = TGGD(scale=1e14,a=-1.8,b=1.0,xmin=1e12).rvs(1e5)
        r = np.sort(r)

        # Create fit object, specifying parameter bounds
        fitobj = SimFit(r,hs_bounds=(12,16),alpha_bounds=(-1.99,-1.6),beta_bounds=(0.5,1.5))

        # Run downhill gradient method
        res, obj = fitobj.run_downhill()

        # Print the resulting parameters
        print "Resulting Parameters: ", res.x

        # The "obj" returned is a list of PerObjLike objects defined at the result, containing lots
        print "Hessian at solution: ", obj[0].hessian
        print "Covariance at solution: ", obj[0].cov
```

```
        # Plot the mass function from obj (the logm contains all the masses from the fit)
        plt.plot(obj[0].logm,obj[0].dndm(log=True),lw=2)
```

```
Resulting Parameters:  [ 13.9900839   -1.79700413   0.94765685 -24.43903982]
Hessian at solution:  [[-1838656.7724181   1485394.57073756  -508029.87769594  -427802.81517913]
 [ 1485394.57073756 -1328344.72287371   412360.36967558   351970.2695528 ]
 [ -508029.87769594   412360.36967558  -142628.15625157  -118826.3858363 ]
 [ -427802.81517913   351970.2695528   -118826.3858363   -100000.08553355]]
Covariance at solution:  [[ 0.00077558 -0.00025638  0.00122131 -0.00567158]
 [-0.00025638  0.00010004 -0.00046645  0.00200319]
 [ 0.00122131 -0.00046645  0.00287933 -0.01028797]
 [-0.00567158  0.00200319 -0.01028797  0.04354857]]
```

Out[6]: [<matplotlib.lines.Line2D at 0x7f072cbebd50>]



Fit MRP parameters to model data

In this example, we do something very simple – fit the MRP parameters using a theoretically produced HMF. This might be one of the first things you'd want to do with the MRP. In addition to the simple fit, we'll also change the truncation scale, to see how the fit performs over different mass ranges. Furthermore, we'll change the redshift and halo overdensity to make sure the fit performs well in all cases.

The resulting plots appear as figures 1,2 in Murray, Robotham, Power (2018)

```
In [28]: #Import necessary libraries
         %matplotlib inline
         import matplotlib.pyplot as plt
         import numpy as np
         from matplotlib.ticker import MaxNLocator
         from mrpy.fitting.fit_curve import get_fit_curve

         from os.path import join, expanduser

         #We'll use the hmf package to produce the theoretical HMF
```

```
         from hmf import MassFunction
         from hmf.cosmo import Planck13

In [17]: fig_folder = expanduser("~")
```

First, we produce a *hmf* model consistent with the Planck 2013 data, and resolved enough to produce a high-quality fit:

```
In [29]: hmf = MassFunction(hmf_model="Tinker08",Mmin=5,Mmax=18,lnk_min=-13,lnk_max=13,
                            transfer_model="CAMB",
                            dlnk=0.01,
                            sigma_8=0.829,n=0.9603,dlog10m=0.1,cosmo_model=Planck13)
```

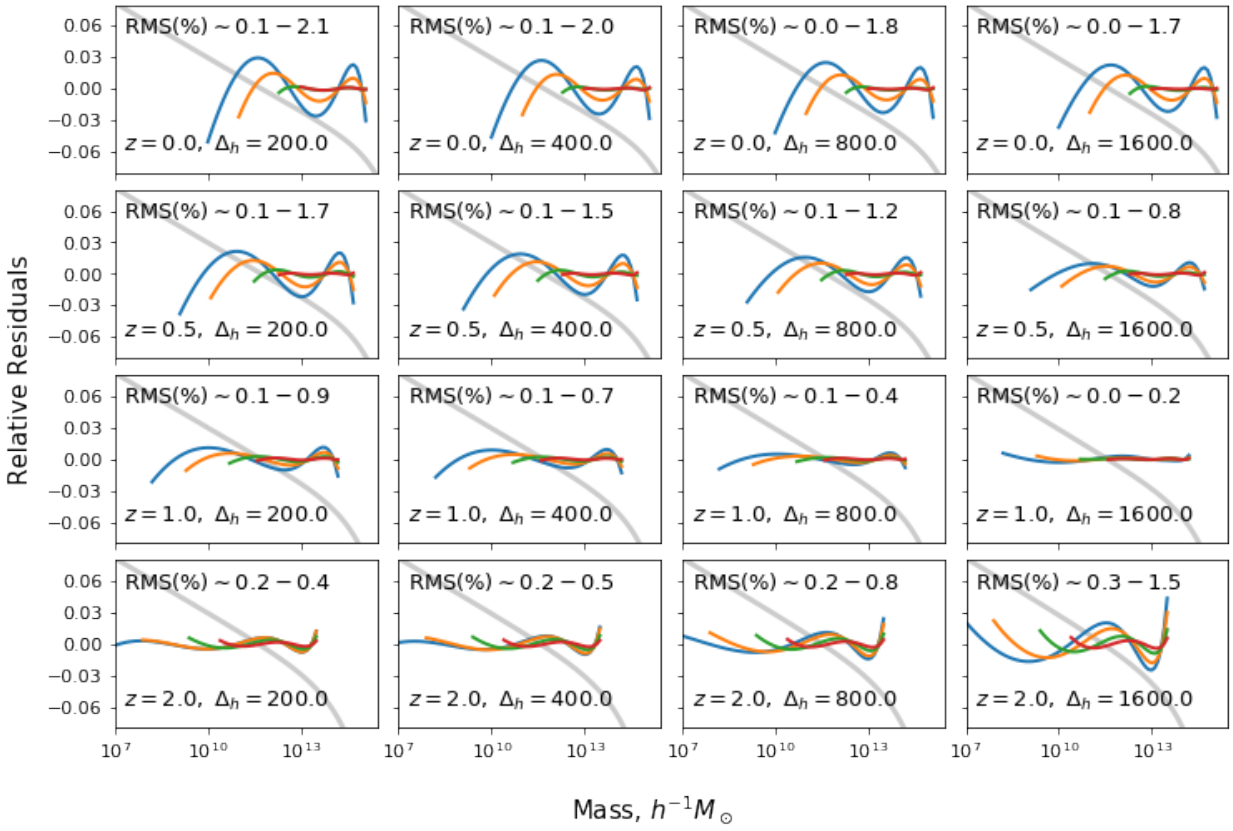
Here's the important part: actually fitting the data.

Fit across redshift and overdensity

```
In [13]: # Create the lists that we'll use to save the results
         res = [0]*64
         obj = [0]*64
         max_dev = np.zeros(64)
         rms_dev = np.zeros(64)
         dndms_whole = [0]*64
         ms_whole = [0]*64

         # 4 different truncation masses
         sigmins = [4,3,2,1.5]
         mmins = [1e10,1e11,1e12,1e13]
         deltahs = [200.0,400.0,800.0,1600.0]
         zs = [0.0,0.5,1.0,2.0]

         dndms = [0]*64
         ii = 0
         for i,z in enumerate(zs):
             for j,deltah in enumerate(deltahs):
                 hmf.update(z=z,delta_h=deltah)

                 for k,sigmin in enumerate(sigmins):
                     # Get theoretical data
                     mask = np.logical_and(hmf.sigma < sigmin, hmf.sigma > 0.5)
                     dn = hmf.dndm[mask]
                     m = hmf.m[mask]

                     # Fit the MRP in the *simplest* way possible.
                     res[ii], obj[ii] = get_fit_curve(m, dn,
                                                      x0 = [14.4,-1.9,0.8,-43],
                                                      bounds=[[None,None],[-2.5,-1.5],[0.2,None],[Nor
                                                      jac=False)

                     max_dev[ii] = np.abs(obj[ii].dndm()/dn-1).max()
                     rms_dev[ii] = np.sqrt(np.mean((obj[ii].dndm()/dn-1)**2))
                     dndms[ii] = dn
                     dndms_whole[ii] = hmf.dndm
                     ms_whole[ii] = hmf.m

                     #print z, deltah, "%1.0e : "%mmin, res[ii].x, max_dev[ii], rms_dev[ii] # the act
                     ii += 1
```

Next the boring stuff... setting up and plotting the figure.

```
In [18]: # Create the figure object
         xmin_plot = 1e7
         xmax_plot = 3e15
         ymin_plot = -0.08
         ymax_plot = 0.08
         fig,ax = plt.subplots(4,4,sharex=True,sharey=True,squeeze=True,figsize=(12,8),
                               subplot_kw={"xscale":"log","ylim":(ymin_plot,ymax_plot),
                                           "xlim":(xmin_plot, xmax_plot)})

         # Contract the space a bit
         plt.subplots_adjust(wspace=0.08,hspace=0.10)

         # Plot the fitted data.
         # Note that 'obj' contains lots of quantities of interest, not least of which is a method
         # to calculate dn/dm!
         ii = 0
         for i,z in enumerate(zs):
             for j,deltah in enumerate(deltahs):
                 ax[i,j].text(2*xmin_plot,0.053, "RMS(%)" + r"$\sim %1.1f - %1.1f$"%(rms_dev[ii:ii+4
                 ax[i,j].text(2*xmin_plot,-0.06,r"$z=%s,\ \Delta_h=%s$"%(z,deltah),fontsize=13)

                 for k,mmin in enumerate(mmins):
                     # Background grey scaled MF
                     if k==0:
                         mask = np.logical_and(ms_whole[ii]>xmin_plot, ms_whole[ii]<xmax_plot)
                         dndm = np.log10(dndms_whole[ii][mask])
                         if ii==0:
                             dndm_range = np.max(dndm) - np.min(dndm)
                         dndm *= (ymax_plot - ymin_plot)/dndm_range
                         dndm += ymax_plot - np.max(dndm)
                         ax[i,j].plot(ms_whole[ii][mask], dndm, color='grey', alpha=0.4,lw=3)

                     # Plot each iteration
                     ax[i,j].plot(obj[ii].m,obj[ii].dndm()/dndms[ii]-1,lw=2)
                     # Modify the ticks for prettiness
                     ax[i,j].tick_params(axis='both', which='major', labelsize=11)
                     ax[i,j].tick_params(axis='both', which='major', labelsize=11)
                     ax[i,j].yaxis.set_major_locator(MaxNLocator(6))



                     ii += 1

         fig.text(0.5, 0.04, r"Mass, $h^{-1}M_\odot$",fontsize=15, ha='center', va='center')
         fig.text(0.06, 0.5, 'Relative Residuals', fontsize=15,ha='center', va='center', rotation='ve

         #Save image
         if fig_folder:
             fig.savefig(join(fig_folder,"comparison_tinker.pdf"))
```



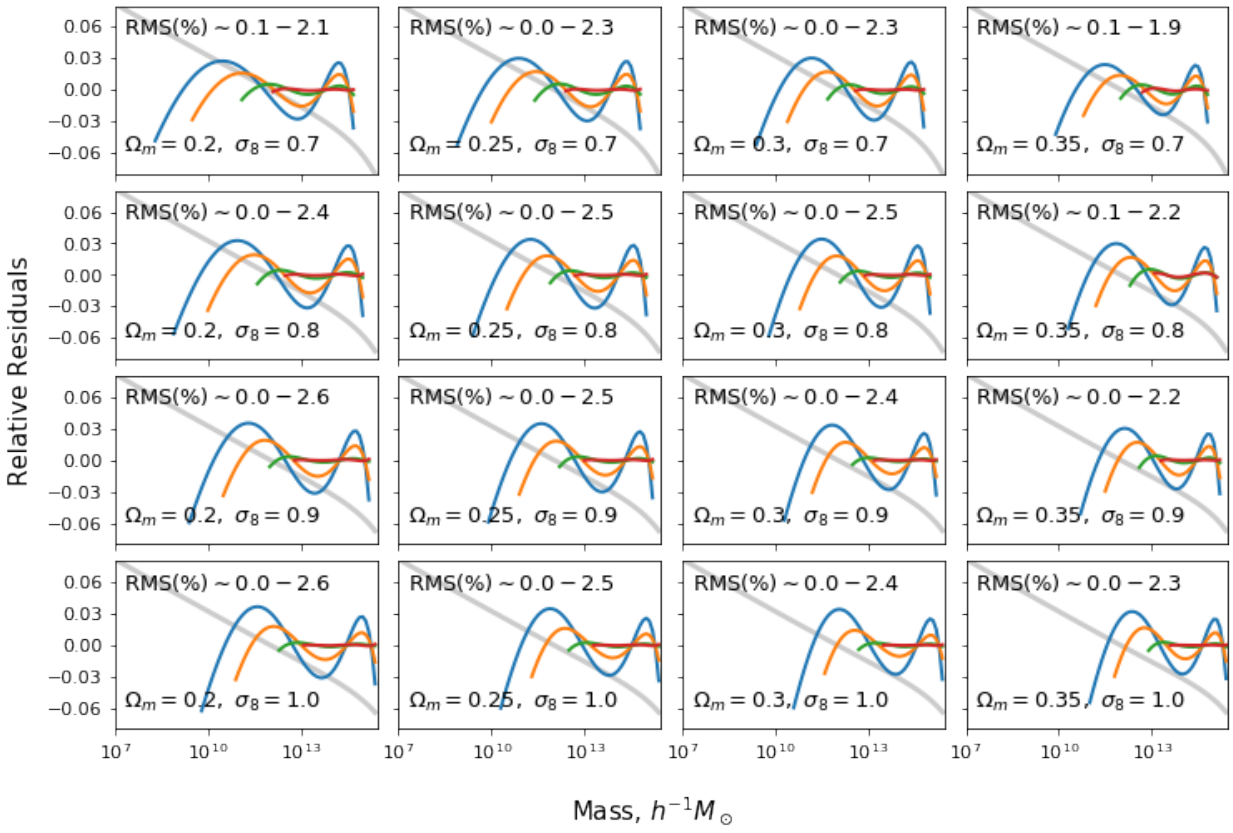
Fit against different cosmology and halo type

```
In [22]: hmf = MassFunction(hmf_model="Warren",Mmin=5,Mmax=18,lnk_min=-13,lnk_max=13,
                            transfer_model="CAMB",
                            dlnk=0.01,
                            sigma_8=0.829,n=0.9603,dlog10m=0.1,cosmo_model=Planck13)

In [23]: # Create the lists that we'll use to save the results
         res = [0]*64
         obj = [0]*64
         max_dev = np.zeros(64)
         rms_dev = np.zeros(64)
         # 4 different truncation masses
         sigmins = [4,3,2,1.5]
         oms = [0.2, 0.25, 0.3, 0.35]
         s8s = [0.7, 0.8, 0.9, 1.0]

         dndms = [0]*64
         ii = 0
         for i,s8 in enumerate(s8s):
             for j,om in enumerate(oms):
                 hmf.update(cosmo_params={"Om0":om}, sigma_8 = s8)

                 for k,sigmin in enumerate(sigmins):


                     # Get theoretical data
                     mask = np.logical_and(hmf.sigma < sigmin, hmf.sigma > 0.5)
```

```
                    dn = hmf.dndm[mask]
                    m = hmf.m[mask]

                    # Fit the MRP in the *simplest* way possible.
                    res[ii], obj[ii] = get_fit_curve(m, dn,
                                                     x0 = [14.4,-1.9,0.8,-43],
                                                     bounds=[[None,None],[-2.5,-1.5],[0.2,None],[Nor
                                                     jac=False)

                    max_dev[ii] = np.abs(obj[ii].dndm()/dn-1).max()
                    rms_dev[ii] = np.sqrt(np.mean((obj[ii].dndm()/dn-1)**2))
                    dndms[ii] = dn

                    dndms_whole[ii] = hmf.dndm
                    ms_whole[ii] = hmf.m

                    #print z, deltah, "%1.0e : "%mmin, res[ii].x, max_dev[ii], rms_dev[ii] # the act
                    ii += 1
```

In [25]:
```
xmin_plot = 1e7
xmax_plot = 3e15
ymin_plot = -0.08
ymax_plot = 0.08
fig,ax = plt.subplots(4,4,sharex=True,sharey=True,squeeze=True,figsize=(12,8),
                      subplot_kw={"xscale":"log","ylim":(ymin_plot,ymax_plot),
                                  "xlim":(xmin_plot, xmax_plot)})
# Contract the space a bit
plt.subplots_adjust(wspace=0.08,hspace=0.10)

# Plot the fitted data.
# Note that 'obj' contains lots of quantities of interest, not least of which is a method
# to calculate dn/dm!
ii = 0
for i,s8 in enumerate(s8s):
    for j,om in enumerate(oms):
        ax[i,j].text(2*xmin_plot,0.053, "RMS(%)" + r"$\sim %1.1f - %1.1f$"%(rms_dev[ii:ii+4
        ax[i,j].text(2*xmin_plot,-0.06,r"$\Omega_m=%s,\ \sigma_8=%s$"%(om,s8),fontsize=13)

        for k,mmin in enumerate(mmins):
            if k==0:
                mask = np.logical_and(ms_whole[ii]>xmin_plot, ms_whole[ii]<xmax_plot)
                dndm = np.log10(dndms_whole[ii][mask])
                if ii==0:
                    dndm_range = np.max(dndm) - np.min(dndm)
                dndm *= (ymax_plot - ymin_plot)/dndm_range
                dndm += ymax_plot - np.max(dndm)
                ax[i,j].plot(ms_whole[ii][mask], dndm, color='grey', alpha=0.4,lw=3)

            # Plot each iteration
            ax[i,j].plot(obj[ii].m,obj[ii].dndm()/dndms[ii]-1,lw=2)

            # Modify the ticks for prettiness
            ax[i,j].tick_params(axis='both', which='major', labelsize=11)
            ax[i,j].tick_params(axis='both', which='major', labelsize=11)
            ax[i,j].yaxis.set_major_locator(MaxNLocator(6))



            ii += 1
```

```
fig.text(0.5, 0.04, r"Mass, $h^{-1}M_\odot$",fontsize=15, ha='center', va='center')
fig.text(0.06, 0.5, 'Relative Residuals', fontsize=15,ha='center', va='center', rotation='ve

#Save image
if fig_folder:
    fig.savefig(join(fig_folder,"comparison_warren.pdf"))
```



Fit MRP parameters to a suite of simulation data simultaneously

In this example, we grab haloes from the publicly available ν^2GC simulation suite and show how MRP can be fit to the haloes of 4 simulations simultaneously. In this case, the 4 simulations have different box sizes, so they probe different parts of the mass function more or less effectively. By combining them, we can get a good handle on a wide range of the mass function.

Do note that this example is not quick. It takes a while to *get* the data, let alone run the MCMC on it. You may want to generate some smaller fake datasets to have a play.

The plots from this example are used in MRP as Figures 3 and 4.

```
In [1]: # General imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import pickle

        from os.path import join, splitext, exists, expanduser

        # Mrpy imports
```

```
        from mrpy.fitting.fit_sample import SimFit
        from mrpy import MRP

        from chainconsumer import ChainConsumer

In [2]: fig_folder = expanduser("~")
        data_folder = expanduser("~")
```

Preparing the data

First you'll need to get the data. These are the files you'll need to download (beware, at least one of them is 12Gb alone):

http://www2.ccs.tsukuba.ac.jp/Astro/Members/ishiyama/nngc/Data/n2gc-m_z0.fof.bz2

http://www2.ccs.tsukuba.ac.jp/Astro/Members/ishiyama/nngc/Data/n2gc-h1_z0.fof.bz2

http://www2.ccs.tsukuba.ac.jp/Astro/Members/ishiyama/nngc/Data/n2gc-m_z0.rockstar.bz2

Then unzip them. **NOTE: you don't need to run this section if you've already got the data and compactified it**

First of all, we need to pare down the huge files. We can do this in a few ways:

- We only care about the Mass column, so we can delete everything else
- We only care about unique halo masses (and the quantity of each), so we can "tabulate" the data
- We keep only haloes with 40 or more particles (this limit is taken from the paper accompanying the catalogue)

These operations reduce the file size by about a factor of 100-1000, and make the subsequent MCMC runs much faster.

Something else to consider is that the fastest way to read in the data files and reduce them is to do it in one big chunk with numpy. However, this takes a lot of memory. So instead we read them line by line.

With these considerations, we implement the following functions.

Compress Data

```
In [3]: def strip_and_compress(fname,fout,mpart=None,Nmin=0,Nmax=np.inf, force=False):
            unique_masses = {}
            ftype = splitext(fname)[1][1:]

            if not force and exists(fout):
                return

            with open(fname) as fin:
                for line in fin:
                    l = line.strip()

                    # Skip comments
                    if l.startswith("#"):
                        continue

                    else:
                        if ftype=="fof":
                            npart = int(l.split()[-1])
                        elif ftype=="rockstar":
                            npart = int(l.split()[7])

                        # Reject the entry if it is less than Nmin
```

```
                    if npart < Nmin:
                        continue
                    elif npart > Nmax:
                        continue

                    # Calculate the mass of the halo
                    if ftype=="fof":
                        mvir = mpart * npart
                    elif ftype=="rockstar":
                        mvir = float(l.split()[21]) # Corresponds to M200b

                    # Add it to the final unique mass dict
                    if mvir in unique_masses:
                        unique_masses[mvir] += 1
                    else:
                        unique_masses[mvir] = 1

            # Convert the dict of values into a 2D array of masses and number of occurrences
            out = np.array([[k,v] for k,v in unique_masses.iteritems()])

            print "Compressed {} to {} percent".format(fname,100*len(out[:,1])/sum(out[:,1]))

            # Save the data to a table file
            np.savetxt(fout,out)
```

Now actually do the stripping and compressing of the files. We save the data in new files with an appended ".compact". Note also we limit the size of the halos, to be in line with the quoted values from the I15 paper. There is in fact at least 1 outlier beyond these limits.

```
In [76]: FORCE = False

         strip_and_compress(join(data_folder,"n2gc-h1_z0.fof"),
                            join(data_folder, "n2gc-h1_z0.fof.compact"),2.75e7,100,17476256)
         strip_and_compress(join(data_folder,"n2gc-m_z0.fof"),
                            join(data_folder, "n2gc-m_z0.fof.compact"),2.2e8,100,12120576)

         strip_and_compress(join(data_folder,"n2gc-m_z0.rockstar"),
                            join(data_folder, "n2gc-m_z0.rockstar.compact"),force=FORCE,
                            Nmin=100)

         strip_and_compress(join(data_folder,"n2gc-h1_z0.rockstar"),
                            join(data_folder, "n2gc-h1_z0.rockstar.compact"), force=FORCE,
                            Nmin=100)
```

```
Compressed /home/steven/Documents/DataSets/n2gc/n2gc-m_z0.rockstar to 0.219503952011 percent
Compressed /home/steven/Documents/DataSets/n2gc/n2gc-h1_z0.rockstar to 0.949670196418 percent
```

Read in Tabulated Data

First up, read in the compact data we just created.

```
In [4]: # Read in the data from file
        def get_raw_data(folder, sims=['h1','m'],ftype="fof",mmin=None,mmax=None):
            m = []
            nm = []
            for sim in sims:
                data = np.genfromtxt(join(folder,"n2gc-{}_z0.{}.compact".format(sim,ftype)))
                m.append(data[:,0])
```

```
            nm.append(data[:,1])

        if mmin is not None:
            for i,mm in enumerate(mmin):
                nm[i] = nm[i][m[i]>mm]
                m[i] = m[i][m[i]>mm]
        if mmax is not None:
            for i,mm in enumerate(mmax):
                nm[i] = nm[i][m[i]<mm]
                m[i] = m[i][m[i]<mm]

        return m,nm
```

We read in both FOF and SO halos with similar parameters, and store everything in the `data` dictionary.

```
In [77]: data = {'fof':{}, "so":{}}

         # FOF halos
         data['fof']['m'], data['fof']['nm'] = get_raw_data(data_folder, ['h1','m'],
                                                            mmin=[2.75e9,2.2e10],mmax=[2e13,7e14])
         data['fof']['weights'] = [data['fof']['nm'][0]/140.0**3, data['fof']['nm'][1]/560.0**3]
         print "Total number of FOF haloes: ", np.sum([np.sum(x) for x in data['fof']['nm']])
         print "Total number of *unique* FOF haloes: ", np.sum([len(x) for x in data['fof']['m']])
         print "-"*40
         # SO halos
         data['so']['m'], data['so']['nm'] = get_raw_data(data_folder, ['h1','m'], ftype='rockstar',
                                                          mmin=[2.75e9,2.2e10],mmax=[2e13,7e14])
         data['so']['weights'] = [data['so']['nm'][0]/140.0**3, data['so']['nm'][1]/560.0**3]
         print "Total number of SO haloes: ", np.sum([np.sum(x) for x in data['so']['nm']])
         print "Total number of *unique* SO haloes: ", np.sum([len(x) for x in data['so']['m']])
```

```
Total number of FOF haloes:  24640920.0
Total number of *unique* FOF haloes:  141278
----------------------------------------
Total number of SO haloes:  24245670.0
Total number of *unique* SO haloes:  109419
```

Running the fits

We'll run the fits with the `emcee` package (via a routine built in to `mrpy`), but also with an optimization solver. The in-built function is able to utilise the tabulation of data we have performed already, and can do the suites simultaneously.

Fitting with MCMC

```
In [78]: # Create the fitting class instance. This will have uniform priors.
         fitobj_fof = SimFit(data['fof']['m'],data['fof']['nm'],
                             V=[140.0**3,560.0**3],
                             alpha_bounds = (-1.99,-1.5), hs_bounds=(12,16),
                             beta_bounds=(0.2,1.5),lnA_bounds=(-50,-10))

         fitobj_so  = SimFit(data['so']['m'],data['so']['nm'],
                             V=[140.0**3,560.0**3],
                             alpha_bounds = (-1.99,-1.5), hs_bounds=(12,16),
                             beta_bounds=(0.2,1.5),lnA_bounds=(-50,-10))

In [79]: # We don't use these, but they can be useful if something goes wrong.
         downhill_res_fof = fitobj_fof.run_downhill(lnA0=-40.0)
         downhill_res_so =  fitobj_so.run_downhill(lnA0=-40.0)
```

```
In [80]: # Run the mcmc.
         # We set 300 chains to warmup, but we can extend this later if we need to manually.
         # Also, we start the chains in a small ball around the best (downhill) optimization solution
         #fitobj_fof.run_mcmc(nchains=50,warmup=200,iterations=500,opt_init=True,threads=8)
         fitobj_so.run_mcmc(nchains=50,warmup=200,iterations=500,opt_init=True,threads=8)

Out[80]: <emcee.ensemble.EnsembleSampler at 0x7fc6704bd350>
```

First off we want to look at a few key diagnostics of the chains to check whether everything's okay.

```
In [81]: print "Acceptance fraction for FOF (min, max, mean): ", fitobj_fof.mcmc_res.acceptance_fract
         print "Acceptance fraction for SO (min, max, mean): ", fitobj_so.mcmc_res.acceptance_fractio

Acceptance fraction for FOF (min, max, mean):  0.518 0.622 0.57288
Acceptance fraction for SO (min, max, mean):  0.542 0.648 0.58912
```

These acceptance fractions are somewhat high, but probably okay. We'll check burnin as well soon.

```
In [5]: def gelman_rubin(chain):
            ssq = np.var(chain, axis=1, ddof=1)
            W = np.mean(ssq, axis=0)
            thb = np.mean(chain, axis=1)
            thbb = np.mean(thb, axis=0)
            m = chain.shape[0]
            n = chain.shape[1]
            B = n / (m - 1) * np.sum((thbb - thb)**2, axis=0)
            var_th = (n - 1.) / n * W + 1. / n * B
            R = np.sqrt(var_th / W)
            return R

In [82]: ChainConsumer().add_chain(fitobj_fof.mcmc_res.chain.reshape((-1,4)), walkers = 50).diagnosti
         ChainConsumer().add_chain(fitobj_so.mcmc_res.chain.reshape((-1,4)), walkers = 50).diagnostic

Gelman-Rubin Statistic values for chain 0
Param 0: 1.04796 (Passed)
Param 1: 1.04777 (Passed)
Param 2: 1.09876 (Passed)
Param 3: 1.05742 (Passed)
Gelman-Rubin Statistic values for chain 0
Param 0: 1.06547 (Passed)
Param 1: 1.06030 (Passed)
Param 2: 1.06584 (Passed)
Param 3: 1.06507 (Passed)

Out[82]: True
```

We see that the chains have converged ($R < 1.1$).

Since the fitting takes some time, we save the main results, i.e. the chain, to file here so that we can begin again at any time without running the MCMC. Thus the following analysis only uses the chains as written to file, rather than the full fit objects just created.

```
In [83]: np.savez("n2gc_analysis/n2gc_mcmc_chain_fof",chain=fitobj_fof.mcmc_res.chain)
         np.savez("n2gc_analysis/n2gc_mcmc_chain_so",chain=fitobj_so.mcmc_res.chain)

In [6]: chain_so = np.load("n2gc_analysis/n2gc_mcmc_chain_so.npz")['chain']
        chain_fof = np.load("n2gc_analysis/n2gc_mcmc_chain_fof.npz")['chain']
```

Analysis

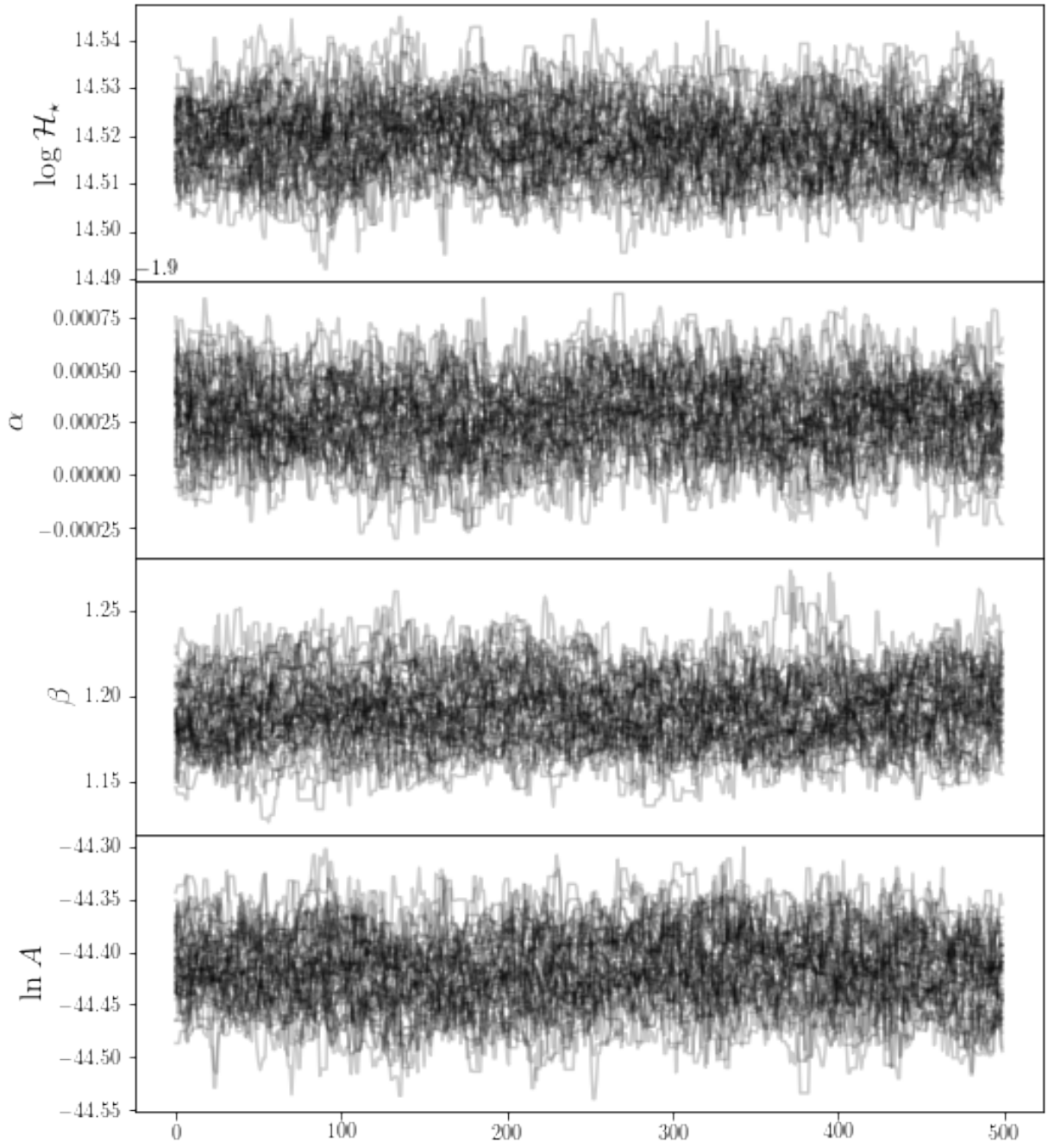
Traceplot

The first thing we might want to do with each fit is to check its traceplot, and determine if the burnin was sufficient.

```
In [7]: def traceplot(keys,chains):
            f, ax = plt.subplots(len(keys), 1, sharex=True, figsize=(8, 2.5 * len(keys)))
            for i, (key, chain) in enumerate(zip(keys,chains.T)):
                ax[i].plot(chain, color="black", alpha=0.2)
                ax[i].set_ylabel(key,fontsize=16)
            f.subplots_adjust(hspace=0)
            plt.setp([a.get_xticklabels() for a in f.axes[:-1]], visible=False)
            return f
```

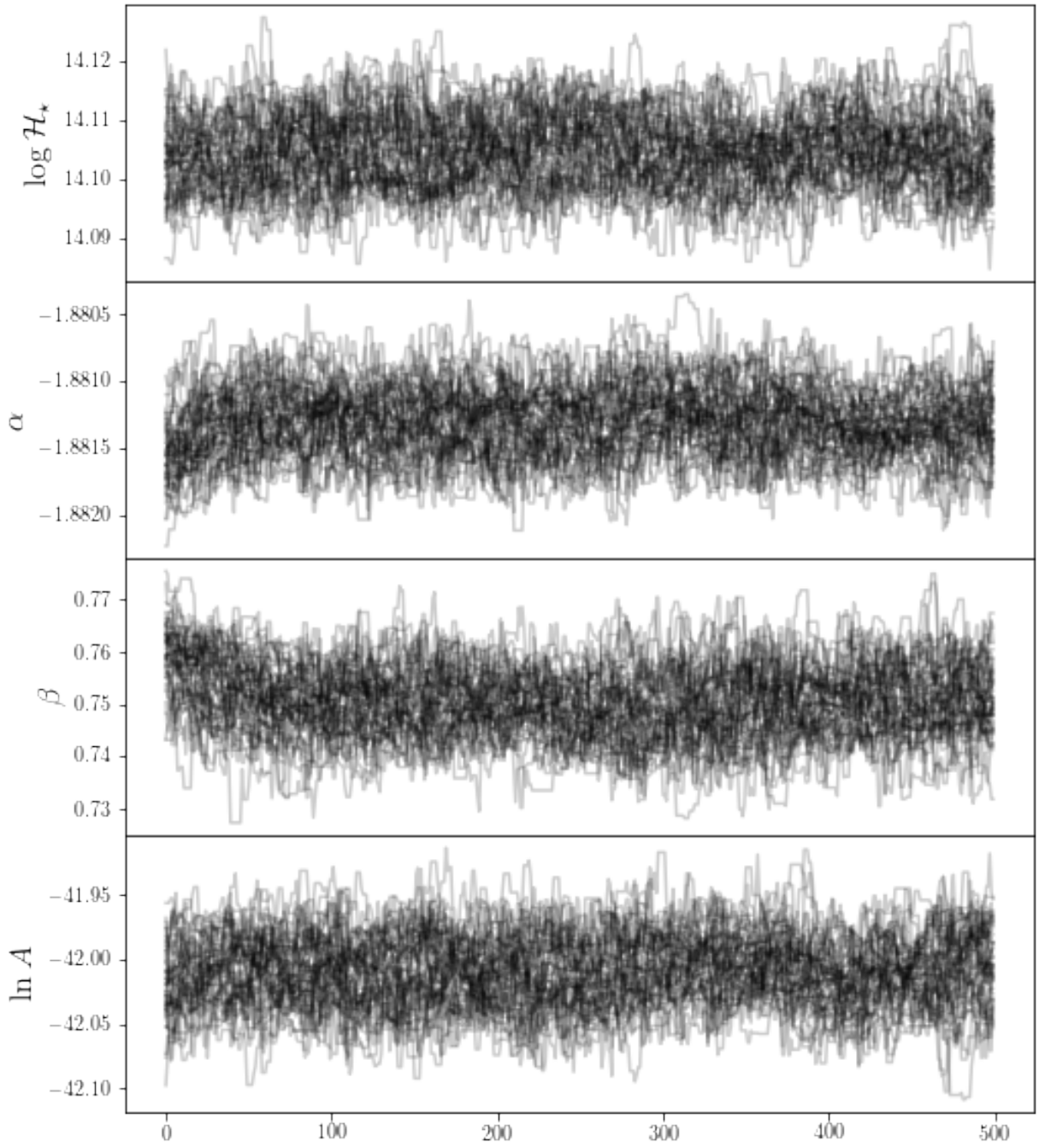
Plot both cases:

```
In [44]: fig0 = traceplot([r"$ \log \mathcal{H}_\star$",r"$\alpha$",r'$\beta$',r"$\ln A$"],chain_fof
         plt.show()
```



It seems like a reasonable burn-in time has been met for the FOF halos, so we're happy we trust our sample.

```
In [86]: fig0 = traceplot([r"$ \log \mathcal{H}_\star$",r"$\alpha$",r'$\beta$',r"$\ln A$"],chain_so)
         plt.show()
```



The SO halos however have moved significantly, possible due to a poor downhill-gradient optimization. We remove the firs parts of the chain:

```
In [87]: chain_so = chain_so[:,100:,:]
```

Basic Results

We'd like to know the pure basic results: mean, median, mode, standard deviation etc.

```
In [51]: print "Mean: ", np.mean(chain_fof,axis=(0,1))
         print "Median: ", np.median(chain_fof,axis=(0,1))
         print "Mode: ", fitobj_fof.mcmc_res.flatchain[np.argmax(fitobj_fof.mcmc_res.flatlnprobabilit
         print "Std Dev.: ", np.std(chain_fof,axis=(0,1))
         print "Covariance: ", np.cov(chain_fof.reshape((-1,4)).T)
         print "Relative Uncertainty: ", np.std(chain_fof,axis=(0,1))*100/np.mean(chain_fof,axis=(0,1
         from mrpy.base.core import log_mass_mode
         print "Log Mass Mode: ", np.log10(log_mass_mode(*np.mean(chain_fof[:,:,:3],axis=(0,1))))
```

```
Mean:  [ 14.51918594  -1.89972215   1.19198794 -44.41676165]
Median:  [ 14.51921005  -1.89971928   1.19135641 -44.41679216]
Mode:  [ 14.51964269  -1.89972233   1.19189829 -44.41885557]
Std Dev.:  [ 0.00715219  0.00016809  0.0194396   0.03367548]
Covariance:  [[  5.11558477e-05  -1.68702393e-07  -1.92940482e-05  -2.09231989e-04]
 [ -1.68702393e-07   2.82549206e-08  -1.01344465e-06   1.83532519e-06]
 [ -1.92940482e-05  -1.01344465e-06   3.77913140e-04  -2.41123515e-04]
 [ -2.09231989e-04   1.83532519e-06  -2.41123515e-04   1.13408347e-03]]
Relative Uncertainty:  [ 0.04926026 -0.00884806  1.63085537 -0.07581706]
Log Mass Mode:  13.617275083
```

```
In [88]: print "Mean: ", np.mean(chain_so,axis=(0,1))
         print "Median: ", np.median(chain_so,axis=(0,1))
         print "Mode: ", fitobj_so.mcmc_res.flatchain[np.argmax(fitobj_so.mcmc_res.flatlnprobability)
         print "Std Dev.: ", np.std(chain_so,axis=(0,1))
         print "Covariance: ", np.cov(chain_so.reshape((-1,4)).T)
         print "Relative Uncertainty: ", np.std(chain_so,axis=(0,1))*100/np.mean(chain_so,axis=(0,1))
         from mrpy.base.core import log_mass_mode
         print "Log Mass Mode: ", np.log10(log_mass_mode(*np.mean(chain_so[:,:,:3],axis=(0,1))))
```

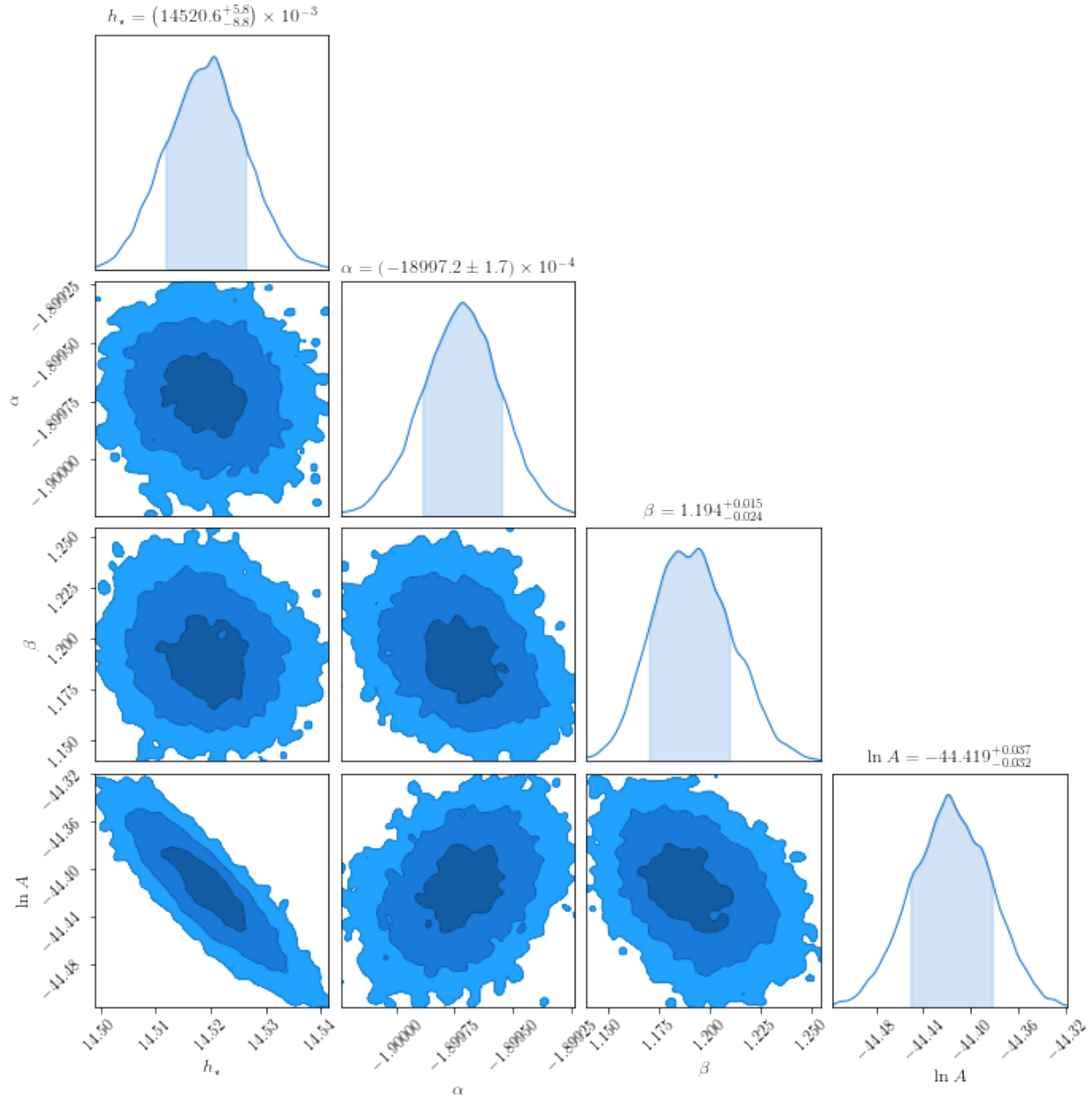
```
Mean:  [ 14.10470648  -1.88130297   0.74993505 -42.00895938]
Median:  [ 14.10465981  -1.88130181   0.7497424  -42.00886146]
Mode:  [ 14.10427937  -1.88133126   0.75064434 -42.00835854]
Std Dev.:  [ 0.00592467  0.00024254  0.00635023  0.02659803]
Covariance:  [[  3.51034875e-05  -4.02509937e-07  -6.81369166e-06  -1.46285387e-04]
 [ -4.02509937e-07   5.88305367e-08  -9.58809971e-07   3.51176102e-06]
 [ -6.81369166e-06  -9.58809971e-07   4.03274777e-05  -3.28788684e-05]
 [ -1.46285387e-04   3.51176102e-06  -3.28788684e-05   7.07490346e-04]]
Relative Uncertainty:  [ 0.04200493 -0.01289234  0.84677108 -0.06331513]
Log Mass Mode:  13.0371689695
```

Corner plot

This produces a "corner" plot which shows the covariance between parameters.

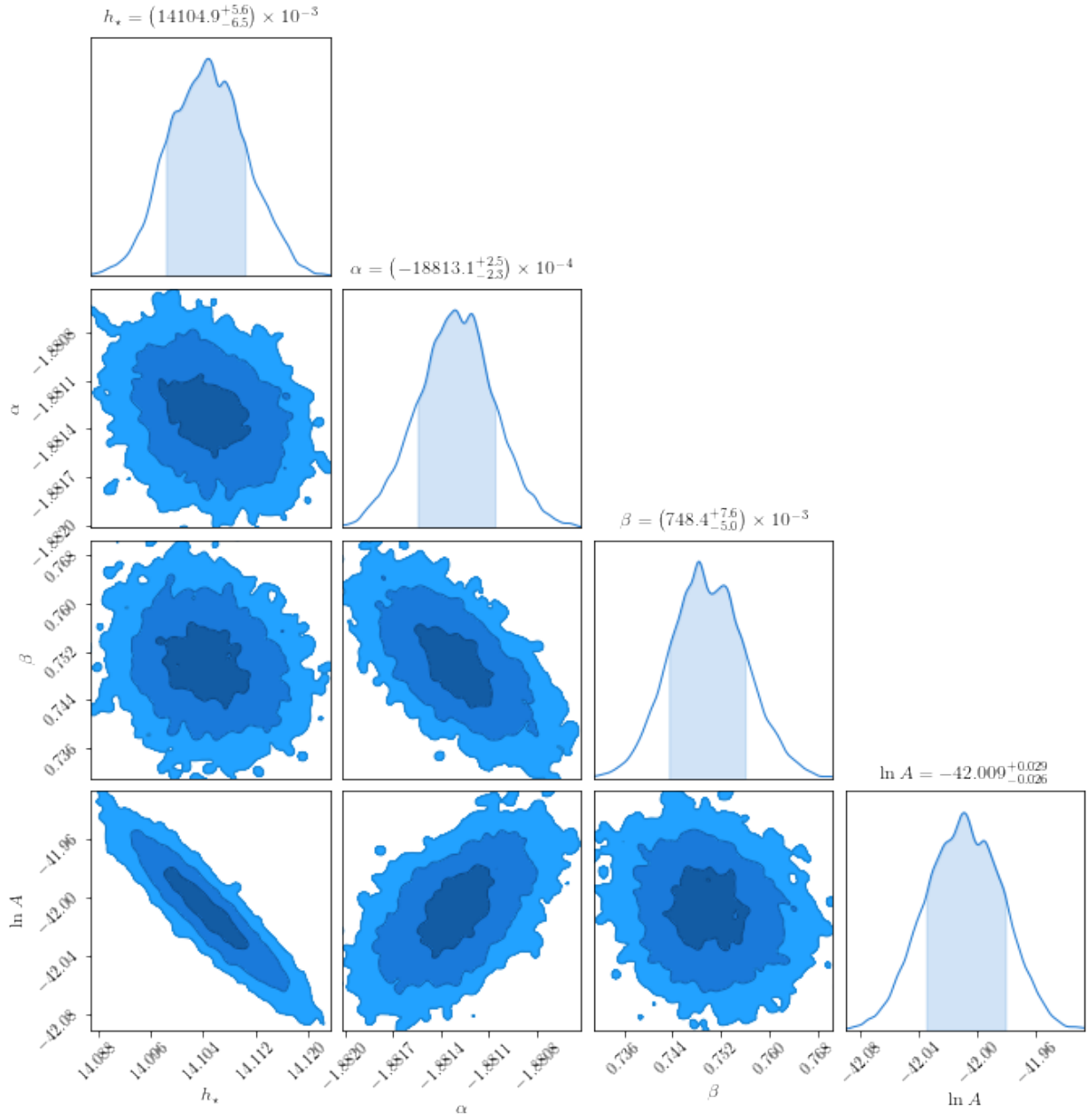
```
In [53]: c = ChainConsumer().add_chain(chain_fof.reshape((-1,4)),
                                       parameters=[r'$h_\star$',r'$\alpha$',r"$\beta$",r'$\ln A$'],
                                       walkers=50)
         fig = c.plot(figsize="PAGE")
         if fig_folder:
             fig.savefig(join(fig_folder,"n2gc_triangle.pdf"))
```

```
WARNING:chainconsumer.chain:This method is deprecated. Please use chainConsumer.plotter.plot instead
```




```
In [89]: c = ChainConsumer().add_chain(chain_so.reshape((-1,4)),
                                       parameters=[r'$h_\star$',r'$\alpha$',r"$\beta$",r'$\ln A$'],
                                       walkers=50)
         fig = c.plot(figsize="PAGE")
```

```
WARNING:chainconsumer.chain:This method is deprecated. Please use chainConsumer.plotter.plot instead
```



Residual Plot

Importantly, we want to check if the actual results look good against the data, when binned.

```
In [90]:
        # A function to create histograms from raw masses, and conver them to dn/dm.
        # It also sets edge values in which a whole bin is not sampled to nan for visual purposes.
        def bin_masses(masses,nm, V, bins=50):
            hist, edges = np.histogram(np.log10(masses), bins,weights=nm)
            centres = (edges[1:] + edges[:-1]) / 2
            dx = centres[1] - centres[0]
            dn = hist.astype("float") / (10 ** centres  *float(V)* dx * np.log(10))#
            poisson_error = np.sqrt(hist.astype("float"))/ (10 ** centres  *float(V)* dx * np.log(10
```

```
            try:
                hist0 = np.where(hist != 0)[0][0]
                dn[hist0] = 0
                hist[hist0] = 0
                poisson_error[hist0] = 0
            except IndexError:
                pass

            try:
                histN = np.where(hist != 0)[0][-1]
                dn[histN] = 0
                hist[histN] = 0
                poisson_error[histN] = 0
            except IndexError:
                pass

            dn[hist == 0] = np.nan
            return centres, dn, hist, poisson_error

        resids = {}
        for jj, ftype in enumerate(['fof','so']):
            resids[ftype] = {}

            m,nm = data[ftype]['m'], data[ftype]['nm']

            # Generate total density of each sim
            resids[ftype]['rho'] = [np.sum(x*nx)/L**3 for x,nx,L in zip(m,nm,[140.0,560.0])]

            # Calculate the total mmin and mmax for all sims in the suite
            mmin = np.min([x.min() for x in m])
            mmax = np.max([x.max() for x in m])

            # Generate the bin structure
            bins = np.linspace(np.log10(mmin), np.log10(mmax),50)
            bin_centres = (bins[1:] + bins[:-1])/2

            # Generate the dn/dm from the sims
            resids[ftype]["dndm"] = []
            resids[ftype]["hist"] = []
            resids[ftype]["err"] = []
            for mi,nmi,L in zip(m,nm,[140.0,560.0]):
                _,dn,h_, err = bin_masses(mi,nmi,L**3,bins)
                resids[ftype]["dndm"].append(dn)
                resids[ftype]["hist"].append(h_)
                resids[ftype]["err"].append(err)


            # The final best-fit object.
            parms = np.mean([chain_fof, chain_so][jj], axis=(0,1))
            norm = parms[3] # downhill_res[0].x[3]
            resids[ftype]['fit'] = MRP(logm = bin_centres,logHs=parms[0],alpha=parms[1],beta=parms[2
```

Along with the best-fit MRP, we want to show the published mass function of the data, which we get from the hmf package.

```
In [57]: from hmf import MassFunction

         h = MassFunction(hmf_model="Ishiyama", cosmo_params={"Om0":0.31, "Ob0":0.048, "H0":68.0},
```

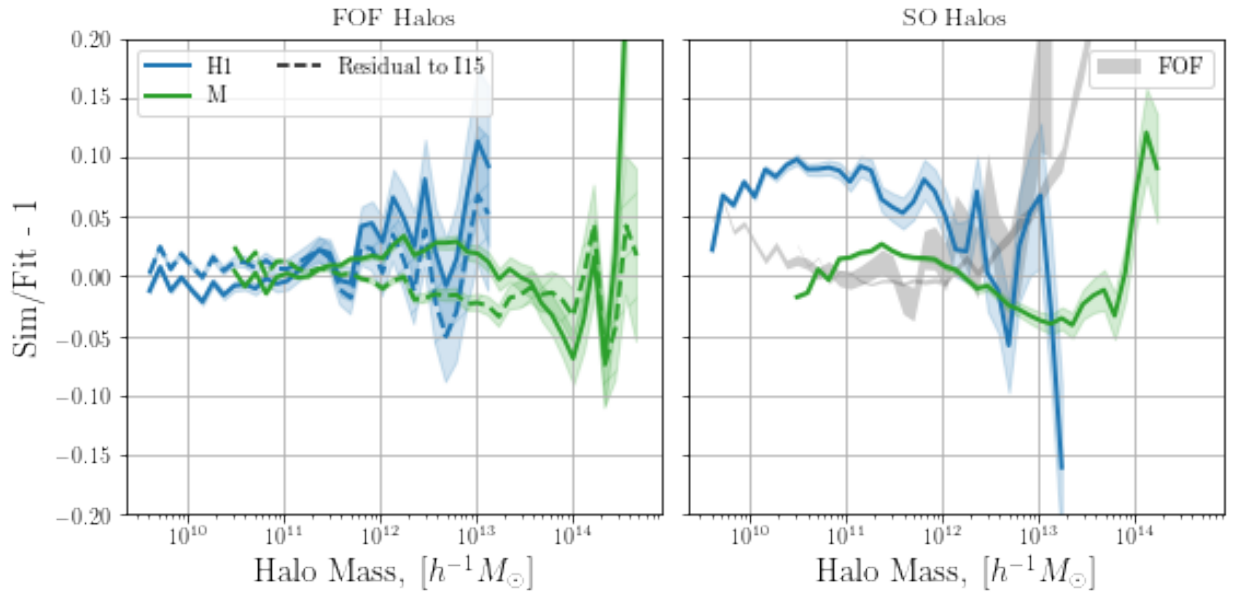
```
                        sigma_8=0.83, n=0.96,lnk_min=-15, lnk_max=15, dlnk=0.01,Mmin=bin_centres[0]
                        dlog10m=bin_centres[1]-bin_centres[0])
```

Finally we draw the actual plot.

```
In [114]: fig, ax = plt.subplots(1,2, figsize=(9,4), sharex=True, sharey=True,
                                  subplot_kw={"xscale":'log', 'ylim':(-0.2,0.2)},
                                  gridspec_kw={"wspace":0.05})
          ftypes = ['fof','so']

          for jj, ftype in enumerate(ftypes):
              for i,(dn,hst,err, label,col) in enumerate(zip(resids[ftype]['dndm'],
                                                             resids[ftype]['hist'],
                                                             resids[ftype]['err'],
                                                             ["H1","M"],
                                                             ["C0",'C2'])):
                  fit = resids[ftype]['fit']

                  # Plot alternative type in grey
                  if jj==1:
                      dn_, err_ = resids[ftypes[(jj+1)%2]]['dndm'][i], resids[ftypes[(jj+1)%2]]['err
                      frac = (dn_/fit.dndm()) - 1
                      err_ = err_/fit.dndm()
                      mask = np.abs(frac)<0.3
                      #ax[jj].plot(10**bin_centres[mask],frac[mask],label=ftypes[(jj+1)%2].upper() i
                      #           color='k',lw=2, alpha=0.1)
                      ax[jj].fill_between(10**bin_centres[mask],
                                          frac[mask] - err_[mask], frac[mask]+err_[mask],
                                             label=ftypes[(jj+1)%2].upper() if i else "",
                                          color='k', alpha=0.2, facecolor=None, edgecolor=None, lw=0)


                  # Residuals to MRP. Mask trailing bits so that poisson noise doesn't dominate the
                  frac = (dn/fit.dndm()) - 1
                  err_ = err/fit.dndm()
                  mask = np.abs(frac)<0.3
                  ax[jj].plot(10**bin_centres[mask],frac[mask],label=label if not jj else "",color=co
                  ax[jj].fill_between(10**bin_centres[mask],
                                      frac[mask] - err_[mask], frac[mask]+err_[mask],
                                      color=col, alpha=0.2)


                  if ftype=="fof":
                      frac = (dn/h.dndm) - 1
                      err_ = err/h.dndm
                      mask = np.abs(frac)<0.3
                      ax[jj].plot(10**bin_centres[mask],frac[mask],color=col,lw=2, ls='--')
                      ax[jj].fill_between(10**bin_centres[mask],
                                          frac[mask] - err_[mask], frac[mask]+err_[mask],
                                          color=col, alpha=0.2, hatch='/')



                  ax[jj].set_xlabel(r"Halo Mass, [$h^{-1}M_\odot$]",fontsize=15)
                  ax[jj].grid(True)
          # Rsidual of Rockstar to MRP
          #frac = dndm_rock/fit.dndm() -1
          #plt.plot(10**bin_centres[np.abs(frac)<0.3],frac[np.abs(frac)<0.3], color="C3",label="SO")

          # Legend item for I15 fit
```

```
        ax[0].plot([0],[0],label="Residual to I15",ls="--",color="k")
        ax[0].set_title("FOF Halos")
        ax[1].set_title("SO Halos")

        # PLOT STYLING
        #ax[0].xscale('log')
        #plt.grid(True)
        #plt.ylim((-0.2,0.2))
        #plt.ylim((-0.05,0.05))
        ax[0].set_ylabel("Sim/Fit - 1",fontsize=15)
        for jj in range(2):
            ax[jj].legend(loc=0,ncol=2)

        # Save for the paper!
        if fig_folder:
            plt.savefig(join(fig_folder,"n2gc_fof_simul.pdf"))
```

```
/home/steven/miniconda3/envs/mrpy/lib/python2.7/site-packages/ipykernel/__main__.py:31: RuntimeWarnir
/home/steven/miniconda3/envs/mrpy/lib/python2.7/site-packages/ipykernel/__main__.py:41: RuntimeWarnir
/home/steven/miniconda3/envs/mrpy/lib/python2.7/site-packages/ipykernel/__main__.py:19: RuntimeWarnir
```



We notice that the residuals from MRP are very similar in magnitude to those from the full EPS-based fit, over a fairly wide range of masses. Note that it seems that the MRP will diverge more significantly below the mass threshold than the EPS fit. In any case, both diverge significantly less than the *same simulation* with haloes found with a spherical overdensity technique.

Explore an entirely analytic model which includes no Poisson error

In this example, we explore the usefulness of a completely analytic solution to an ideal "sample". For details, see Appendix D.3.3 of the MRP paper. In brief, the "ideal sample" is composed of non-Poisson limited haloes extracted from a pure MRP distribution within some physical volume. As such, the solution is *a priori* the input parameters of the MRP distribution. Finding the covariance of those parameters is our task.

The whole framework of this problem has already been implemented in `mrpy.extra.analytic_model`. We will use that framework to answer some questions.

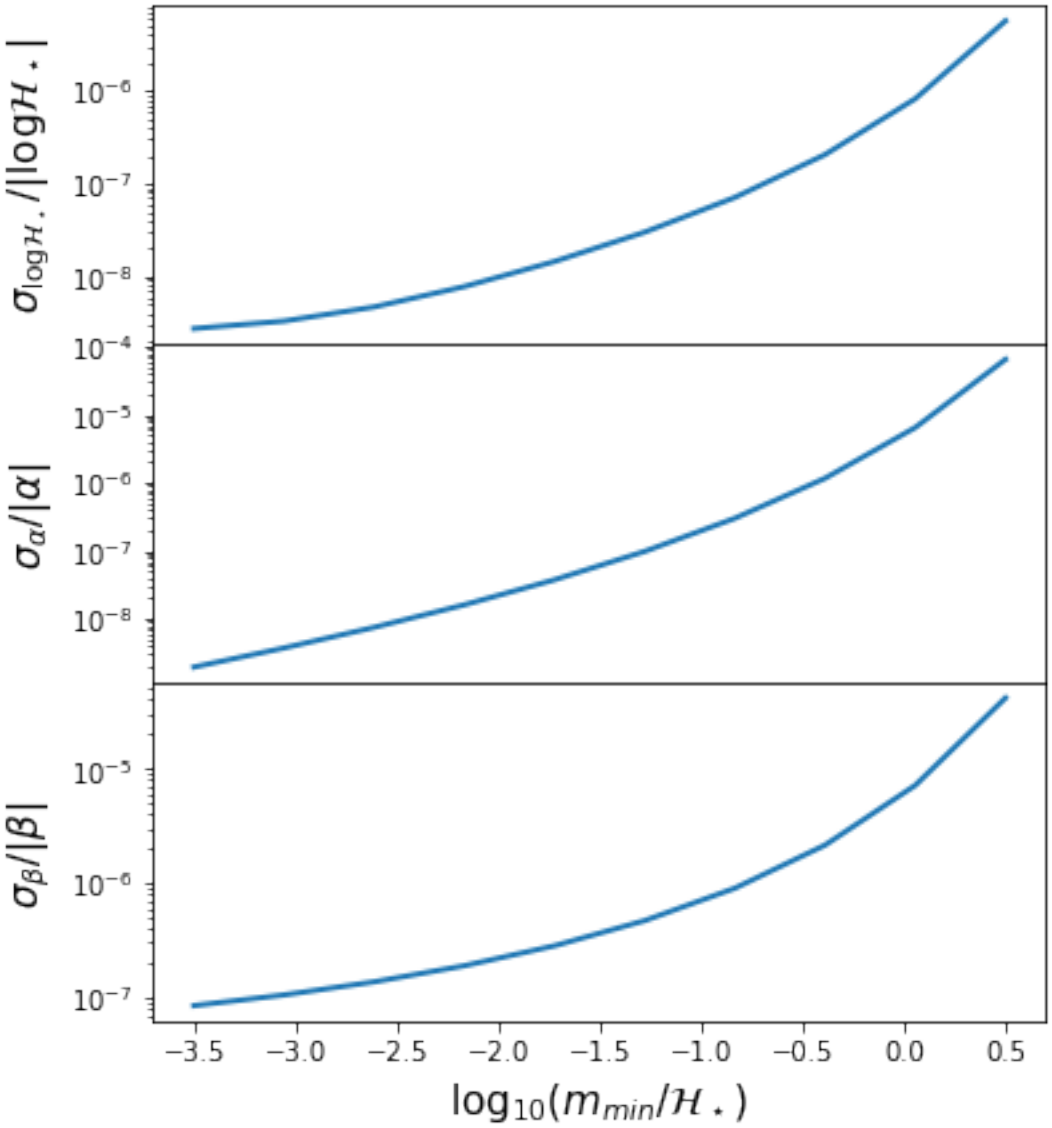
```
In [1]: # Imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np

        from mrpy.extra.analytic_model import IdealAnalytic
```

```
In [2]: # Fiducial Parameters
        beta = 0.75
        alpha = -1.85
        hs = 14.5
        V=400.0**3 # Physical volume

        # Some constants
        log_mmin = np.linspace(11,15,10)
```

What is the expected variance of parameters versustruncation mass?

```
In [3]: fig,ax = plt.subplots(3,1,sharex=True,sharey=False,figsize=(6,7),subplot_kw={"yscale":'log'},
                              gridspec_kw={"hspace":0.0})

        K = IdealAnalytic(log_mmin=log_mmin,logHs=hs,alpha=alpha,beta=beta,lnA=0.0)
        ax[0].plot(log_mmin-hs,np.sqrt(K.cov[:,0,0])/np.abs(hs),lw=2)
        ax[1].plot(log_mmin-hs,np.sqrt(K.cov[:,1,1])/np.abs(alpha),lw=2)
        ax[2].plot(log_mmin-hs,np.sqrt(K.cov[:,2,2])/np.abs(beta),lw=2)

        #ax[0].legend(loc=0,ncol=2)

        ax[0].set_ylabel(r"$\sigma_{\log \mathcal{H}_\star}/|\log \mathcal{H}_\star|$",fontsize=15)
        ax[1].set_ylabel(r"$\sigma_\alpha/|\alpha|$",fontsize=15)
        ax[2].set_ylabel(r"$\sigma_\beta/|\beta|$",fontsize=15)
        ax[2].set_xlabel(r"$\log_{10}\left(m_{min}/\mathcal{H}_\star\right)$",fontsize=15)
```

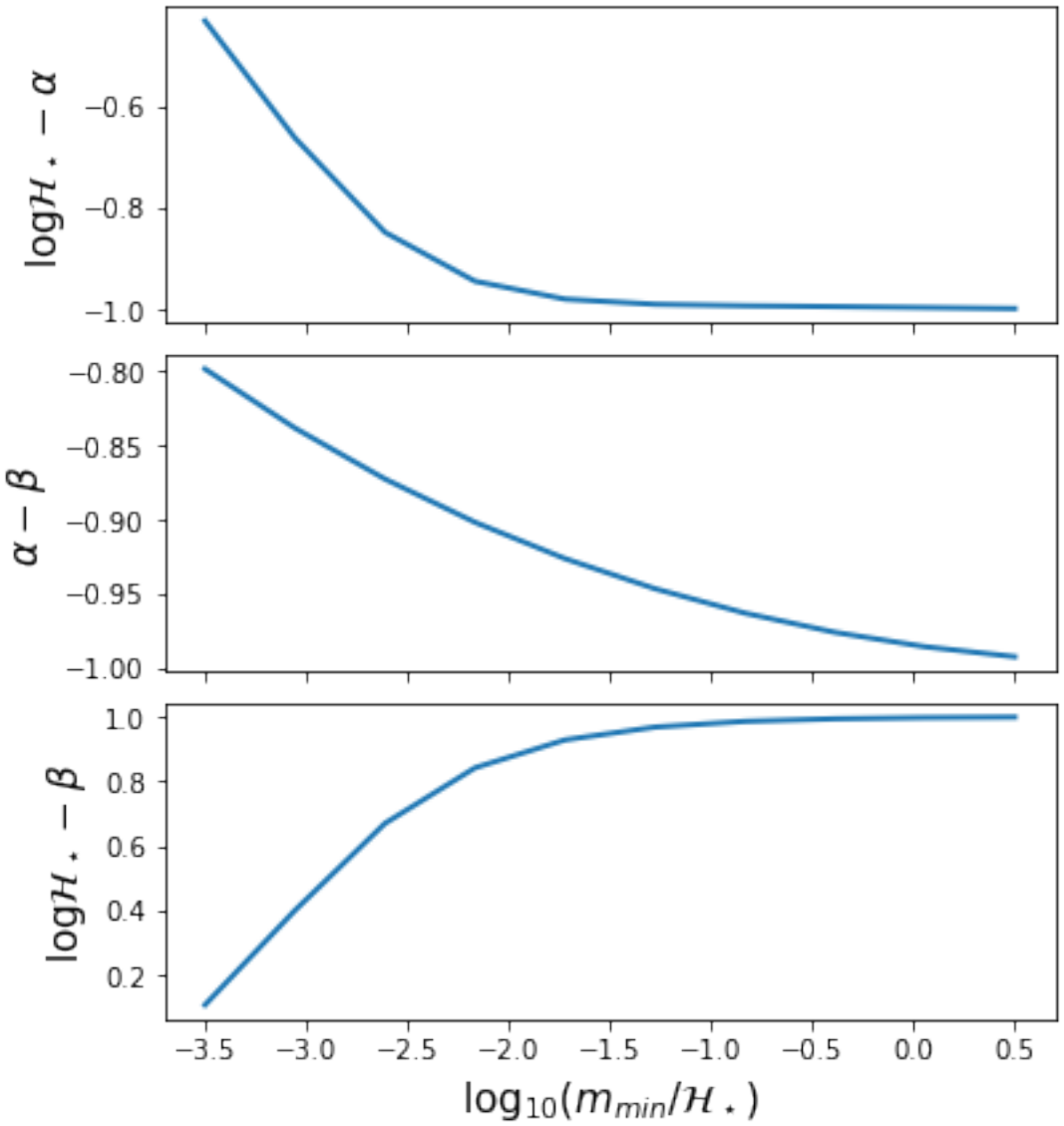
```
Out[3]: <matplotlib.text.Text at 0x7f5681ccfb10>
```



What is the expected correlation of parameters versus truncation mass?

```
In [4]: fig,ax = plt.subplots(3,1,sharex=True,sharey=False,figsize=(6,7),
                              gridspec_kw={"hspace":0.1})

        K = IdealAnalytic(log_mmin=log_mmin,logHs=hs,alpha=alpha,beta=beta,lnA=0.0)
        ax[0].plot(log_mmin-hs,K.corr[:,0,1],lw=2)
        ax[1].plot(log_mmin-hs,K.corr[:,1,2],lw=2)
        ax[2].plot(log_mmin-hs,K.corr[:,0,2],lw=2)

        #ax[0].legend(loc=0,ncol=2)

        ax[0].set_ylabel(r"$\log \mathcal{H}_\star-\alpha$",fontsize=15)
        ax[1].set_ylabel(r"$\alpha-\beta$",fontsize=15)
        ax[2].set_ylabel(r"$\log \mathcal{H}_\star-\beta$",fontsize=15)
        ax[2].set_xlabel(r"$\log_{10}\left(m_{min}/\mathcal{H}_\star\right)$",fontsize=15)
```

Out[4]: <matplotlib.text.Text at 0x7f5681b27090>



Determine relationship of MRP parameters to physical ones

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from hmf import MassFunction
from scipy.integrate import simps
from scipy.interpolate import InterpolatedUnivariateSpline as spline
from mrpy.fitting.fit_curve import get_fit_curve
from mrpy import dndm
import os

from astropy.cosmology import Planck13
```

In [5]:
```
fig_folder=""
```

Our goal in this example is to find functions that relate the MRP parameters, $(\mathcal{H}_s, \alpha, \beta)$ to the physical parameters (z, Ω_m, σ_8). The results of this analysis are *already stored* in the `physical_dependence` module of `mrpy`. This example shows how those models were derived explicitly, and also gives some ideas on how to fit theoretical curves, and the issues that can pop up.

Figures from this example appear in MRP (2018) as figures 5,6 and 7

Our first task is to set up the fiducial cosmology and choose a fitting function. We choose the Behroozi+13 SO fitting function, with a fiducial cosmology of Planck+13 (Planck+WP 68% limits):

```
In [18]: h = MassFunction(cosmo_model=Planck13, hmf_model = "Behroozi",
                          dlnk=0.02, lnk_max=11, lnk_min=-6.5, dlog10m=0.01)
```

Checking Transfer model and Resolution Parameters

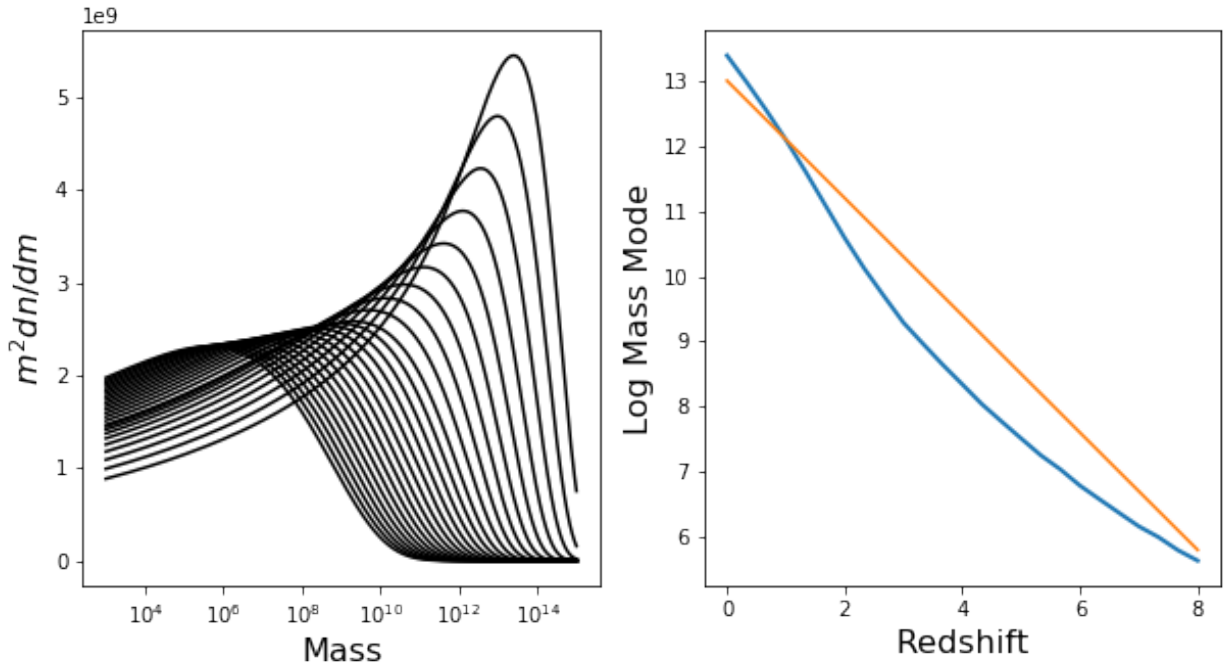
Before we begin our actual analysis, we need to make sure of a few things concerning the models and resolution. Firstly, we want to make sure that using EH isn't a big deal. What we do is plot the resulting mass function ratio of an EH vs CAMB model, for several redshifts (dependence on other parameters will be very small), where the mass is scaled by the log mass mode:

```
In [2]: def get_mass_mode(m,dndm,scalar=True):
            spl = spline(np.log10(m[dndm>0]),(m**2*dndm)[dndm>0],k=4) #k=4 necessary for getting roo
            d = spl.derivative()
            if scalar:
                return d.roots()[0]
            else:
                return d.roots()
```

The mass mode will not change significantly over Ω_m or σ_8, but will change a lot for redshift. Let's have a look at this dependence, to make sure everything is okay:

```
In [19]: fig,ax = plt.subplots(1,2,figsize=(10,5))

         h.update(Mmin=3,Mmax=15)
         Z = np.linspace(0,8,25)
         mode = np.zeros(len(Z))
         for i,z in enumerate(Z):
             h.update(z=z)
             mode[i] = get_mass_mode(h.m,h.dndm)
             ax[0].plot(h.m,h.m**2 * h.dndm,color="k")

         ax[0].set_xscale('log')
         ax[1].plot(Z,mode,lw=2)
         ax[1].plot(Z,13 - 0.9*Z)
         ax[1].set_xlabel("Redshift",fontsize=16)
         ax[0].set_xlabel("Mass",fontsize=16)
         ax[1].set_ylabel("Log Mass Mode",fontsize=16)
         ax[0].set_ylabel(r"$m^2 dn/dm$",fontsize=16)
```

```
Out[19]: <matplotlib.text.Text at 0x7f2a5e564e90>
```

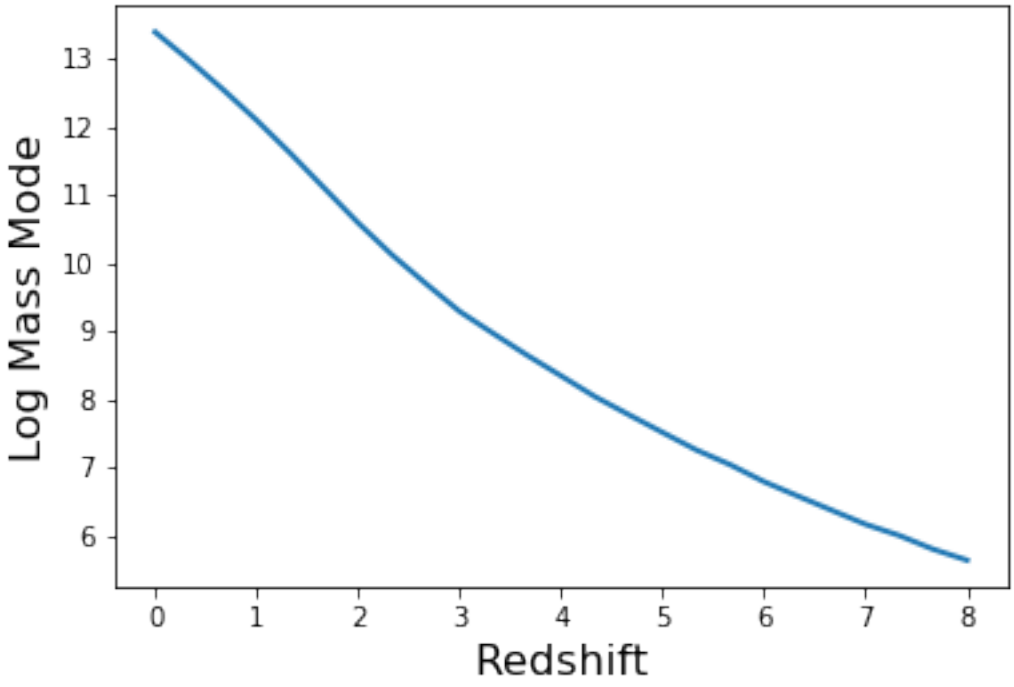


This all looks good (which means that the default resolution parameters are all good for our purposes, but we'll check this more thoroughly soon). Furthermore, we should be able to define a more efficient function which doesn't need to have a huge range of Mmin/Mmax:

```
In [3]: def get_mass_mode_h(h,Mmin=-3,Mmax=2):
            estimate = 13 - 0.9*h.z
            finished = False
            i = 0
            while not finished and i<4:
                i += 1
                h.update(Mmin=estimate+1.5*Mmin*i**0.5,Mmax=estimate+1.5*Mmax*i**0.5)
                mode = get_mass_mode(h.M,h.dndm,scalar=False)
                if len(mode)==1 and mode[0]+Mmin > h.Mmin and mode[0]+Mmax < h.Mmax:
                    finished = True
                else:
                    continue
            return mode[0]
```

Just for testing, let's use this function to again produce a similar plot to above.

```
In [21]: def get_mass_mode_h(h):
             return np.log10(h.M[np.argmax(h.m*h.dndlnm)])

         Z = np.linspace(0,8,25)
         mode = np.zeros(len(Z))
         for i,z in enumerate(Z):
             h.update(z=z)
             mode[i] = get_mass_mode(h.m,h.dndm)

         plt.plot(Z,mode,lw=2)
         plt.xlabel("Redshift",fontsize=16)
         plt.ylabel("Log Mass Mode",fontsize=16)

         if fig_folder:
             plt.savefig(join(fig_folder,"log_mass_mode.pdf"))
```




```
In [4]: def recast_dndm(m,dndm,mode,Mmin=-3,Mmax=2,dm=h.dlog10m):
            mvec = np.arange(mode+Mmin,mode+Mmax,dm)
            sp = spline(np.log10(m[dndm>0]),np.log10(dndm[dndm>0]))
            return 10**sp(mvec)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-b9afd81f536a> in <module>()
----> 1 def recast_dndm(m,dndm,mode,Mmin=-3,Mmax=2,dm=h.dlog10m):
      2     mvec = np.arange(mode+Mmin,mode+Mmax,dm)
      3     sp = spline(np.log10(m[dndm>0]),np.log10(dndm[dndm>0]))
      4     return 10**sp(mvec)

NameError: name 'h' is not defined
```

Mmin and Mmax Determination

We want to choose good mass limits for each parameter set. We do this by setting a constant Mmin and Mmax w.r.t. the mass mode. By setting the mode inside these limits, we ensure a good range to define β in each case. Let's have a look at the mass functions inside these limits:

```
In [ ]: h.update(Mmin=-1,Mmax=18)
        fig,ax = plt.subplots(2,1,sharex=True,figsize=(8,6),gridspec_kw={"hspace":0})

        for z in np.linspace(0,8,9):
            h.update(z=z)
            mode = get_mass_mode(h.m,h.dndm)

            sp = spline(np.log10(h.m[h.dndm>0]),np.log10(h.dndm[h.dndm>0]))

            mvec = np.arange(mode-4,mode+4,0.02)
            hvec = np.arange(-4,4,0.02)
            # The following is necessary since sometimes the vectors have 1 more element
```

```
            N = min(len(mvec),len(hvec))
            mvec = mvec[:N]
            hvec = hvec[:N]

            ax[0].plot(hvec,10**sp(mvec),lw=2)
            ax[1].plot(hvec,10**(2*mvec) * 10**sp(mvec),lw=2)

        ax[0].set_yscale('log')
        ax[0].set_ylim((1e-40,1e14))
        ax[1].set_xlabel(r"$\log_{10} (m/\mathcal{H}_T)$",fontsize=20)
        ax[0].set_ylabel(r"$dn/dm$",fontsize=20)
        ax[1].set_ylabel(r"$m^2 \frac{dn}{dm}$",fontsize=20)

        h.update(z=0)
```

In [ ]:
```
Mmin = -3
Mmax = 2
```

Data Generation

The main idea is to generate a sample of parameters, (z, Ω_m, σ_8), based on a realistic distribution (i.e. Planck13), then for each parameter, generate the HMF between the relevant limits. Then each HMF is to be fit with the MRP. The resulting parameters will need to be fit in Eureqa, after which we'll port the results back here ;)

So, first up, the general parameters of the samples:

Parameter Samples

In [ ]:
```
# General configuration for parameter samples
N_1d = 200      # Number of samples to draw for 1D data
N_3d = 2000     # Number of samples to draw for 3D data
s8_mean = h.sigma_8
s8_sd = 0.012
Om0_mean = h.cosmo.Om0
Om0_sd = 0.017
```

Both σ_8 and Ω_m will be sampled from the Planck distribution, but z will come from a log-linear distribution (which more highly weights low redshifts, but not too much). First, define a function which will generate the samples:

In [ ]:
```
def get_parameter_sample(N, zbound,s8_mean=s8_mean,s8_sd=s8_sd,Om0_mean=Om0_mean,Om0_sd=Om0_s
    sigma_8 = np.random.normal(s8_mean, s8_sd, size=N)
    Om0 = np.random.normal(Om0_mean, Om0_sd, size=N)

    if not hasattr(zbound, "__len__"):
        z = np.repeat(zbound, N)
    else:
        z = np.exp(np.linspace(np.log(1 + zbound[0]), np.log(1 + zbound[1]), N)) - 1

    return sigma_8, Om0, z
```

And a function which generates the EPS mass function from each sample.

In [ ]:
```
def get_hmfs(h, Om0, z, s8,Mmin=-3,Mmax=1):
    # If Om0 is a scalar, then things can be sped up a bit.
    try:
        Om0[3]
        go_for_it = False
    except:
```

```
        go_for_it = True

    # Make sure they're all vectors
    Om0 = np.atleast_1d(Om0)
    z = np.atleast_1d(z)
    s8 = np.atleast_1d(s8)

    N = max(len(Om0),len(z),len(s8))

    if len(Om0)==1: Om0 = np.repeat(Om0,N)
    if len(s8)==1: s8 = np.repeat(s8,N)
    if len(z)==1: z = np.repeat(z,N)

    Nm = len(arange(Mmin,Mmax,h.dlog10m))

    dndm = np.zeros((Nm,N))
    mode = np.zeros(N)

    h.update(Mmin=2,Mmax=16)
    for i, (s, zz, m) in enumerate(zip(s8, z, Om0)):
        if i%100==0:
            print float(i) * 100 / float(len(z)), "% done"

        h.update(cosmo_params={"Om0":m}, z=zz, sigma_8=s)
        if go_for_it:
            mode[i] = get_mass_mode(h.M,h.dndm)
        else:
            mode[i] = get_mass_mode_h(h)

        #get vals at new mass range
        dndm[:,i] = recast_dndm(h.m,h.dndm,mode[i],Mmin=Mmin,Mmax=Mmax,dm=h.dlog10m)[:Nm]
        new_m = arange(mode[i]+Mmin,mode[i]+Mmax,h.dlog10m)

        # for some reason, sometimes new_m is 1 bigger than needed
        if len(new_m)>Nm:
            new_m = new_m[:Nm]


    return dndm,mode
```

Now, try to read the samples in from a data file, otherwise produce them again.

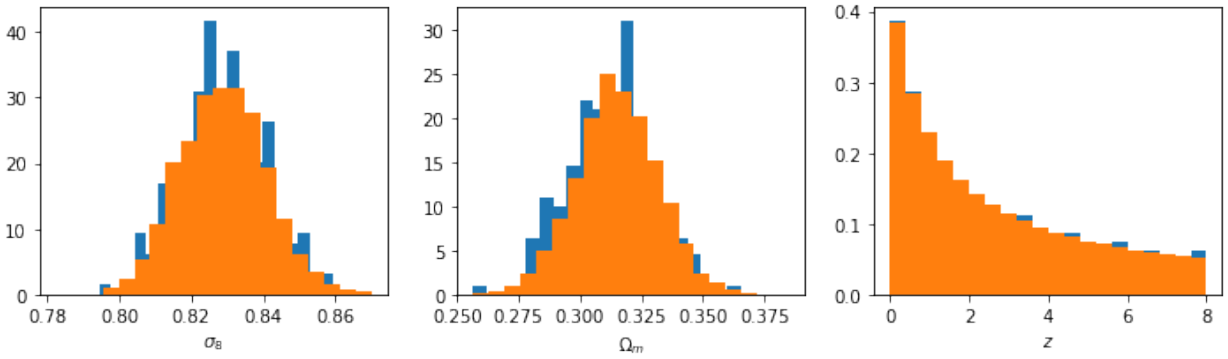
```
In [ ]: if os.path.exists("phys_dep/raw_data.npz"):
    all_data = np.load("phys_dep/raw_data.npz")

    # 3d
    s8_3d = all_data["s8_3d"]
    Om0_3d = all_data["Om0_3d"]
    z_3d = all_data["z_3d"]
    dndm_3d = all_data["dndm_3d"]
    Ht_3d = all_data["Ht_3d"]

    # 1d
    s8_1d = all_data["s8_1d"]
    Om0_1d = all_data["Om0_1d"]
    z_1d = all_data["z_1d"]
    dndm_z_1d = all_data['dndm_z_1d']
    dndm_s8_1d = all_data['dndm_s8_1d']
    dndm_Om0_1d = all_data['dndm_Om0_1d']
    Ht_z_1d = all_data['Ht_z_1d']
```

```
        Ht_s8_1d = all_data['Ht_s8_1d']
        Ht_Om0_1d = all_data['Ht_Om0_1d']
        del all_data
    else:
        # Create Samples
        s8_1d, Om0_1d, z_1d = get_parameter_sample(N_1d,(0,8))
        s8_3d, Om0_3d, z_3d = get_parameter_sample(N_3d,(0,8))

        # Calculate HMF's
        dndm_s8_1d, Ht_s8_1d = get_hmfs(h,Om0,0,s8_1d,Mmin=Mmin,Mmax=Mmax)
        dndm_Om0_1d, Ht_Om0_1d = get_hmfs(h,Om0_1d,0,s8,Mmin=Mmin,Mmax=Mmax)
        dndm_z_1d, Ht_z_1d = get_hmfs(h,Om0,z_1d,s8,Mmin=Mmin,Mmax=Mmax)
        print "done 1d z"

        dndm_3d, Ht_3d = get_hmfs(h,Om0_3d,z_3d,s8_3d,Mmin=Mmin,Mmax=Mmax)
        print "done 3d 8"
```

For good measure, check the distributions:

```
In [33]: fig,ax = plt.subplots(1,3,figsize=(12,3))

         ax[0].hist(s8_1d,normed=True,bins=20)
         ax[1].hist(Om0_1d,normed=True,bins=20)

         ax[0].hist(s8_3d,normed=True,bins=20)
         ax[1].hist(Om0_3d,normed=True,bins=20)

         ax[2].hist(z_1d,normed=True,bins=20)
         ax[2].hist(z_3d,normed=True,bins=20)

         ax[0].set_xlabel(r"$\sigma_8$")
         ax[1].set_xlabel(r"$\Omega_m$")
         ax[2].set_xlabel(r"$z$")
```

```
Out[33]: <matplotlib.text.Text at 0x7f2a5c108090>
```



To me, it looks like the best choice will be something like (-3,2). This will give an upper limit of about 15.5 at z=0, which is pretty close to the largest things we expect to find.At high redshift, this will reach down to $M_{\rm min} \approx 3$, which is incredibly small, but hey, who trusts these scales anyway?

Now save the data for use next time...

```
In [53]: np.savez("phys_dep/raw_data.npz",s8_3d=s8_3d,Om0_3d=Om0_3d,z_3d=z_3d,dndm_3d=dndm_3d,Ht_3d=H
                  z_1d=z_1d,dndm_z_1d=dndm_z_1d,Ht_z_1d=Ht_z_1d,
                  s8_1d = s8_1d,dndm_s8_1d=dndm_s8_1d,Ht_s8_1d=Ht_s8_1d,
                  Om0_1d = Om0_1d,dndm_Om0_1d=dndm_Om0_1d,Ht_Om0_1d=Ht_Om0_1d)
```

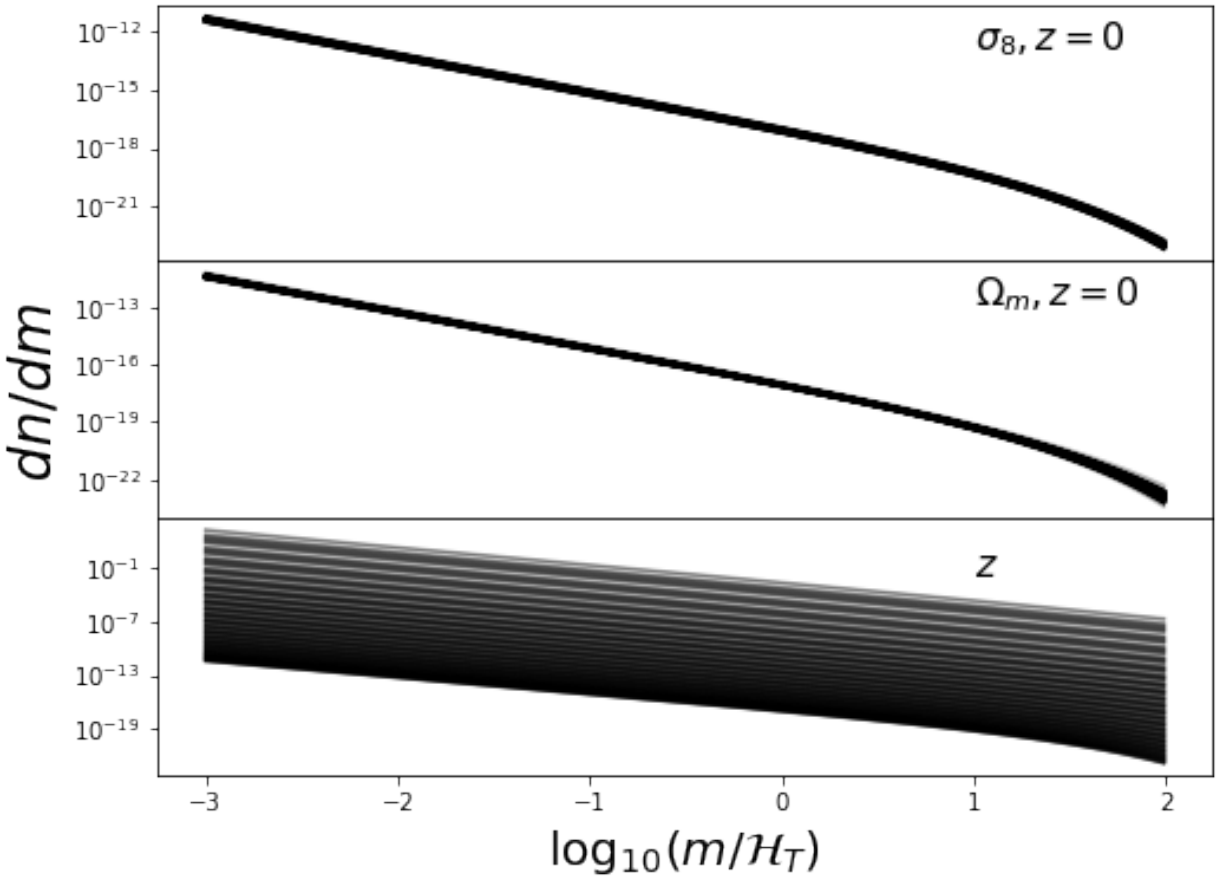
Now it would be good to have a look at all the fits to make sure they look reasonable.

```
In [34]: def plot_sample(sample,m,ax=None,ratio=False,color="k",alpha=0.3,label=None,**kwargs):
             if ax is None:
                 ax = plt.subplot(111)

             for i in range(sample.shape[1]):
                 if i>0:
                     label=None
                 if ratio:
                     ax.plot(m,sample[:,i]/sample[:,0],color=color,alpha=alpha,label=label**kwargs)
                 else:
                     ax.plot(m,sample[:,i],color=color,alpha=alpha,label=label,**kwargs)
```

First look at all the 1d distros

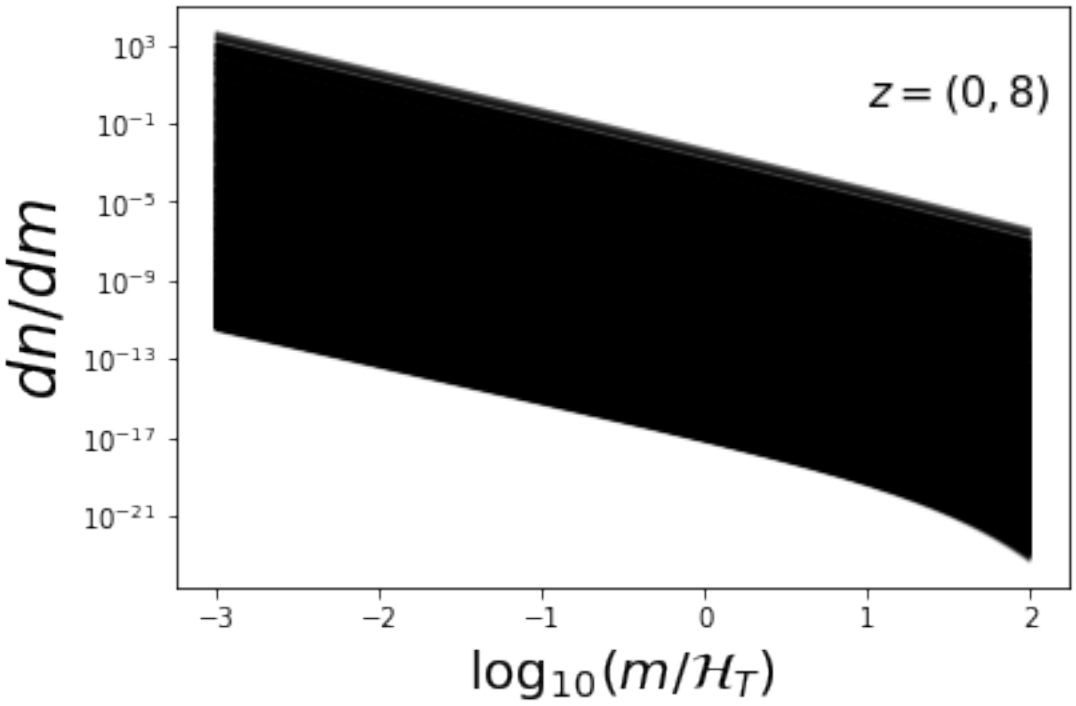
```
In [35]: mvec = np.arange(Mmin,Mmax,h.dlog10m)

         fig,ax=plt.subplots(3,1,sharex=True,gridspec_kw={"hspace":0.0},subplot_kw={"yscale":'log'},f
         for i,sample in enumerate([dndm_s8_1d,dndm_Om0_1d,dndm_z_1d]):
             plot_sample(sample,mvec,ax[i],ratio=False)


         ax[2].set_xlabel(r"$\log_{10}(m/\mathcal{H}_T)$",fontsize=20)
         ax[1].set_ylabel(r"$dn/dm$",fontsize=24)
         ax[0].text(1,1e-13,r"$\sigma_8, z=0$",fontsize=16)
         ax[1].text(1,1e-13,r"$\Omega_m, z=0$",fontsize=16)
         ax[2].text(1,1e-2,r"$z$",fontsize=16)
```

```
Out[35]: <matplotlib.text.Text at 0x7f2a3ebd5610>
```



These look nice and contiguous as we would expect. Now the 3d distros:

```
In [36]: mvec = np.arange(Mmin,Mmax,h.dlog10m)

         plot_sample(dndm_3d,mvec,ratio=False)

         plt.xlabel(r"$\log_{10}(m/\mathcal{H}_T)$",fontsize=20)
         plt.ylabel(r"$dn/dm$",fontsize=24)
         plt.text(1,1e-13,r"$z=0$",fontsize=16)
         plt.text(1,1e0,r"$z=(0,8)$",fontsize=16)
         plt.yscale('log')
```



Fit MRP

First we define a generic function that will fit the dndm vectors.

```
In [ ]: def fit_all_mrp(sample,mode,Om0=None,Mmin=-3,Mmax=1,s=0):
            par = np.zeros((sample.shape[1],4))
            fit = np.zeros_like(sample)

            if Om0 is None:
                Om0 = np.zeros(sample.shape[1])

            guesstimate = [14.5,-1.9,0.8,-40]
            for i,(dndm,m) in enumerate(zip(sample.T,mode)):
                M = 10**np.arange(m+Mmin,m+Mmax,h.dlog10m)[:len(dndm)]
                res,curve = fit_curve(M,dndm,hs0=guesstimate[0],alpha0=guesstimate[1],
                                      beta0=guesstimate[2],lnA0=guesstimate[3])

                par[i] = res.x
                fit[:,i] = curve.dndm()
```

```
            guesstimate = par[i]
        return par,fit
```

And now a couple of functions which plot the results:

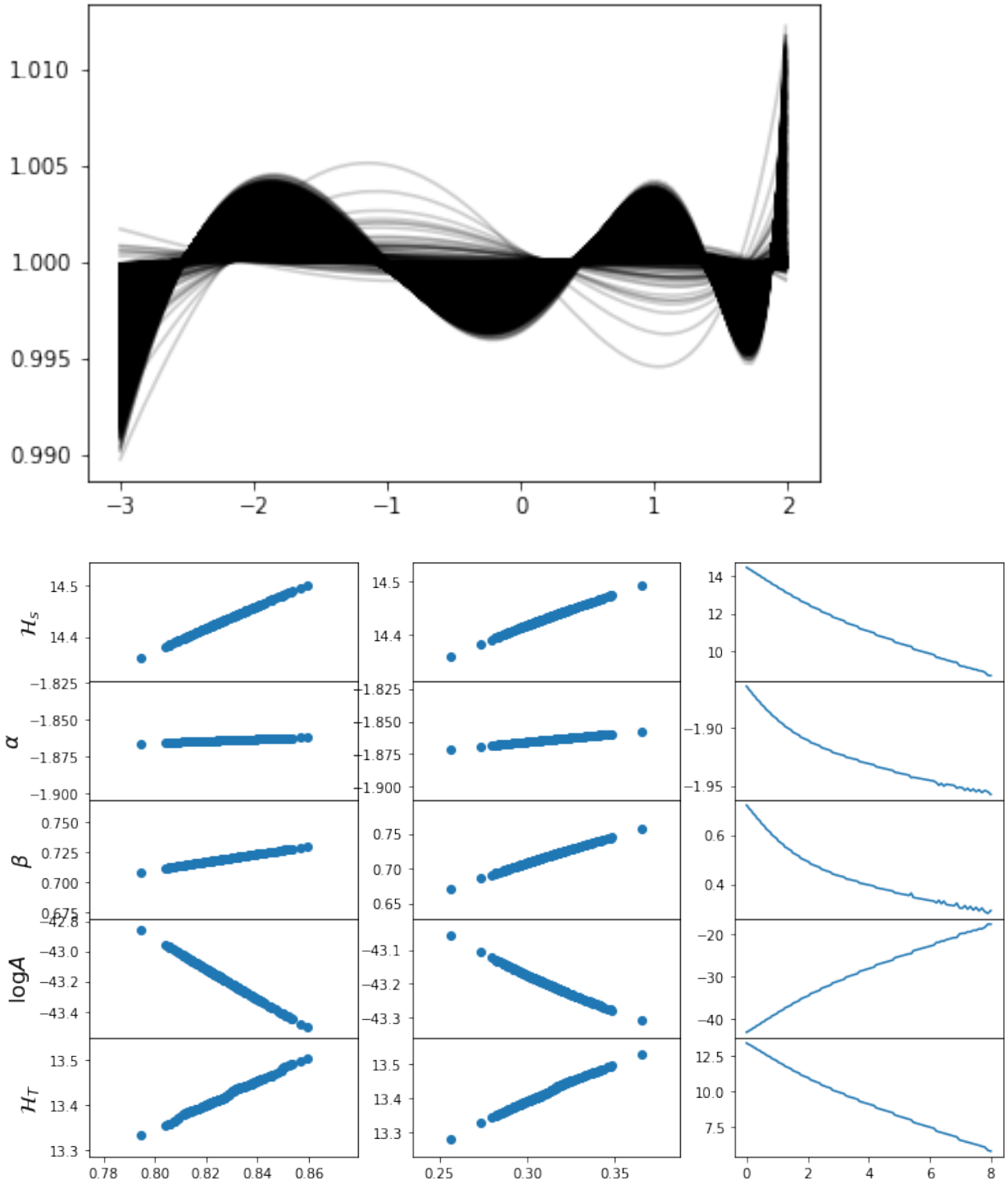
```
In [ ]: def plot_mrp_fit(fit_dict,ax=None,Mmin=Mmin,Mmax=Mmax,dm=h.dlog10m):
            if ax is None:
                ax = plt.subplot(111)

            mvec = np.arange(Mmin,Mmax,dm)[:dndm_3d.shape[0]]
            for i in range(dndm_3d.shape[1]):
                ax.plot(mvec,fit_dict["fit_3d"][:,i]/dndm_3d[:,i],color="k",alpha=0.2)

        def plot_mrp_param_dep(fit_dict):
            fig,ax=plt.subplots(5,3,sharex="col",gridspec_kw={"hspace":0},figsize=(12,8))

            for i in range(5):
                if i<4:
                    ax[i,0].scatter(s8_1d,fit_dict['par_s8_1d'][:,i])
                else:
                    ax[i,0].scatter(s8_1d,Ht_s8_1d)
                ax[i,0].set_ylabel([r"$\mathcal{H}_s$",r"$\alpha$",r"$\beta$",r"$\log A$",r"$\mathcal

            for i in range(5):
                if i<4:
                    ax[i,1].scatter(Om0_1d,fit_dict['par_Om0_1d'][:,i])
                else:
                    ax[i,1].scatter(Om0_1d,Ht_Om0_1d)

            for i in range(5):
                if i<4:
                    ax[i,2].plot(z_1d,fit_dict['par_z_1d'][:,i])
                else:
                    ax[i,2].plot(z_1d,Ht_z_1d)

            for i in range(3):
                ax[3,i].set_xlabel([r"$\sigma_8$",r"$\Omega_m$",r"$z$"][i],fontsize=16)
```

Here we generate the data (again, only if not found in a corresponding file already):

```
In [39]: if os.path.exists('phys_dep/fits.npz'):
             fits = dict(np.load("phys_dep/fits.npz"))
         else:
             fits = {}
             fits["par_s8_1d"], fits["fit_s8_1d"]  = fit_all_mrp(dndm_s8_1d , Ht_s8_1d, Mmin=Mmin,Mma
             fits["par_Om0_1d"],fits["fit_Om0_1d"] = fit_all_mrp(dndm_Om0_1d, Ht_Om0_1d,Mmin=Mmin,Mma
             fits["par_z_1d"],  fits["fit_z_1d"]   = fit_all_mrp(dndm_z_1d  , Ht_z_1d,  Mmin=Mmin,Mma
             fits["par_3d"],    fits["fit_3d"]     = fit_all_mrp(dndm_3d    , Ht_3d,    Mmin=Mmin,Mma

         np.savez("phys_dep/fits.npz",**fits)
         np.savetxt("phys_dep/s8_fits.dat",np.vstack((s8_1d,fits['par_s8_1d'].T)).T)
         np.savetxt("phys_dep/Om0_fits.dat",np.vstack((Om0_1d,fits['par_Om0_1d'].T)).T)
         np.savetxt("phys_dep/z_fits.dat",np.vstack((z_1d,fits['par_z_1d'].T)).T)
         np.savetxt("phys_dep/3d_fits.dat",np.vstack((s8_3d,Om0_3d,z_3d,fits['par_3d'].T)).T)
```

Now plot the resulting functions.

```
In [40]: plot_mrp_fit(fits)
         plot_mrp_param_dep(fits)
```



Return from Eureqa

The data we generated was input to the program "Eureqa" to determine relationships between the physical and MRP parameters. The results were then implemented in `physical_dependence`.

Get new fits for all the data

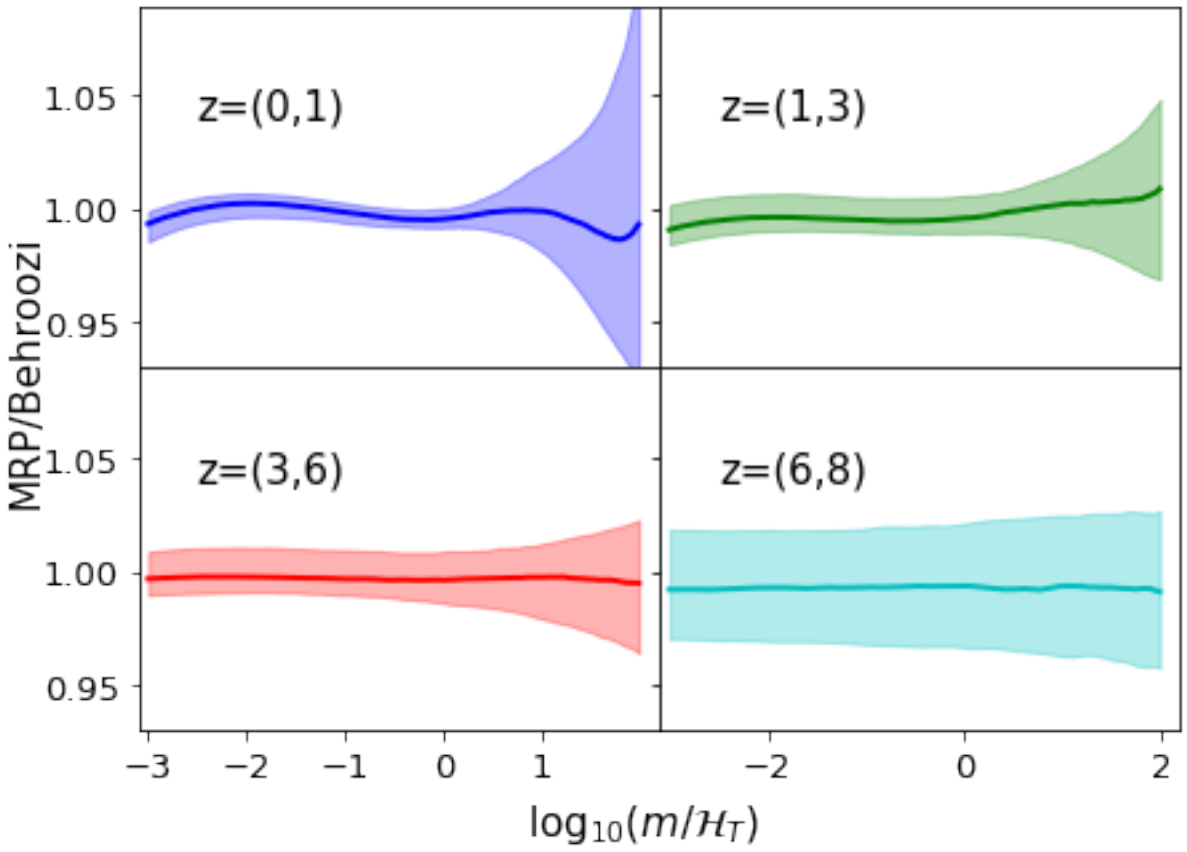
First, we define a helper function for putting labels above entire rows/columns of axes:

```
In [ ]: def suplabel(axis,label,label_prop=None,
                     labelpad=5,
                     ha='center',va='center'):
            ''' Add super ylabel or xlabel to the figure
            Similar to matplotlib.suptitle
            axis       - string: "x" or "y"
            label      - string
            label_prop - keyword dictionary for Text
            labelpad   - padding from the axis (default: 5)
            ha         - horizontal alignment (default: "center")
            va         - vertical alignment (default: "center")
            '''
            fig = plt.gcf()
            xmin = []
            ymin = []
            for ax in fig.axes:
                xmin.append(ax.get_position().xmin)
                ymin.append(ax.get_position().ymin)
            xmin,ymin = min(xmin),min(ymin)
            dpi = fig.dpi
            if axis.lower() == "y":
                rotation=90.
                x = xmin-float(labelpad)/dpi
                y = 0.5
            elif axis.lower() == 'x':
                rotation = 0.
                x = 0.5
                y = ymin - float(labelpad)/dpi
            else:
                raise Exception("Unexpected axis: x or y")
            if label_prop is None:
                label_prop = dict()
            plt.text(x,y,label,rotation=rotation,
                       transform=fig.transFigure,
                       ha=ha,va=va,
                       **label_prop)
```

We basically run through all our samples, generating fits for them based on the parameterisations we created, then we plot each as a ratio to the true HMF. For some more clarity, we do this in several redshift bins.

```
In [43]: # Import the necessary function
         from mrpy.extra.physical_dependence import mrp_b13, _alpha_b13, _beta_b13, _logHs_b13, _lnA_

         # Define each mass function based on our results
         post_fit = np.zeros_like(dndm_3d)
         for i,(z,m,s,mode) in enumerate(zip(z_3d,Om0_3d,s8_3d,Ht_3d)):
             mvec = 10**np.arange(mode+Mmin,mode+Mmax,h.dlog10m)[:len(dndm_3d)]
             post_fit[:,i] = mrp_b13(mvec,z,m,s)

         # Create a figure plotting the average ratio of the functions
         # to the true HMF, with shading giving the uncertainty region.
         ratio = post_fit/dndm_3d
         hvec = np.arange(Mmin,Mmax,0.01)[:len(mvec)]
         cols = ["b","g","r","c"]
         fig,ax = plt.subplots(2,2,sharey=True,sharex="col",figsize=(7,5),gridspec_kw={"hspace":0,"ws
```

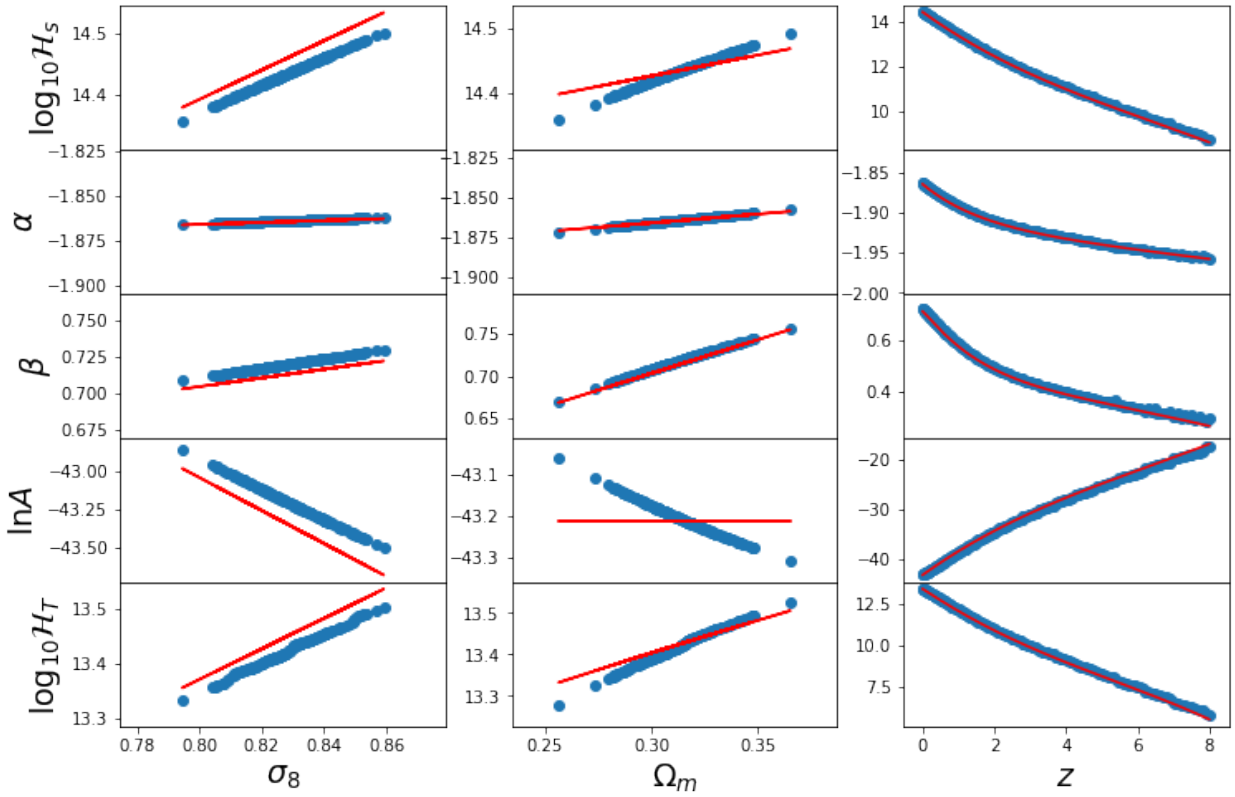
```
                                subplot_kw={"ylim":(0.93,1.09),"xlim":(-3.1,2.2)})

    for j,zrange in enumerate([(0,1),(1,3),(3,6),(6,8)]):
        mask = np.logical_and(z_3d>=zrange[0], z_3d < zrange[-1])
        mean = np.median(ratio[:,mask],axis=1)
        qlow, qhi = np.percentile(ratio[:,mask],(16,84),axis=1)

        ax[divmod(j,2)].plot(hvec,mean,lw=2,color=cols[j],label="z=(%s,%s)"%zrange)
        ax[divmod(j,2)].fill_between(hvec,qlow,qhi,color=cols[j],alpha=0.3)
        ax[divmod(j,2)].text(-2.5,1.04,"z=(%s,%s)"%zrange,fontsize=15)
        ax[divmod(j,2)].tick_params(axis='both', which='major', labelsize=13)
        if j==2:
            ax[divmod(j,2)].xaxis.set_ticks([-3,-2,-1,0,1])
    suplabel('x',r"$\log_{10}(m/\mathcal{H}_T)$",label_prop={"fontsize":15},labelpad=7)
    suplabel("y","MRP/Behroozi",label_prop={"fontsize":15},labelpad=6)

    # Save for the paper!
    if fig_folder:
        plt.savefig(fig_folder+"param_3d.pdf")
```



Also, we can re-look at the parameter dependence with included parameterisations:

```
In [45]: fig,ax=plt.subplots(5,3,sharex="col",gridspec_kw={"hspace":0},figsize=(12,8))

         Om0 = h.cosmo.Om0
         s8 = h.sigma_8

         ## Dependence on s8
```

```
Hs = _logHs_b13(0,Om0,s8_1d)
alpha = _alpha_b13(0,Om0,s8_1d)
beta = _beta_b13(0,Om0,s8_1d)
lnA = _lnA_b13(0,Om0,s8_1d)
param = [Hs,alpha,beta,lnA]
for i in range(5):
    if i<4:
        ax[i,0].scatter(s8_1d,fits['par_s8_1d'][:,i])
        ax[i,0].plot(s8_1d,param[i],color="r")
    else:
        ax[i,0].scatter(s8_1d,Ht_s8_1d)
        ax[i,0].plot(s8_1d,Hs+np.log10((alpha+2)/beta)/beta,color="r")

    ax[i,0].set_ylabel([r"$\log_{10}\mathcal{H}_s$",r"$\alpha$",r"$\beta$",r"$\ln A$",r"$\lo

## Dependence on Om0
Hs = _logHs_b13(0,Om0_1d,s8)
alpha = _alpha_b13(0,Om0_1d,s8)
beta = _beta_b13(0,Om0_1d,s8)
lnA = _lnA_b13(0,Om0_1d,s8)
param = [Hs,alpha,beta,lnA]
for i in range(5):
    if i<4:
        ax[i,1].scatter(Om0_1d,fits['par_Om0_1d'][:,i])
        ax[i,1].plot(Om0_1d,param[i],color="r")
    else:
        ax[i,1].scatter(Om0_1d,Ht_Om0_1d)
        ax[i,1].plot(Om0_1d,Hs+np.log10((alpha+2)/beta)/beta,color="r")

## Dependence on z
Hs = _logHs_b13(z_1d,Om0,s8)
alpha = _alpha_b13(z_1d,Om0,s8)
beta = _beta_b13(z_1d,Om0,s8)
lnA = _lnA_b13(z_1d,Om0,s8)
param = [Hs,alpha,beta,lnA]
for i in range(5):
    if i<4:
        ax[i,2].scatter(z_1d,fits['par_z_1d'][:,i])
        ax[i,2].plot(z_1d,param[i],color="r")
    else:
        ax[i,2].scatter(z_1d,Ht_z_1d)
        ax[i,2].plot(z_1d,Hs+np.log10((alpha+2)/beta)/beta,color="r")


for i in range(3):
    ax[4,i].set_xlabel([r"$\sigma_8$",r"$\Omega_m$",r"$z$"][i],fontsize=19)

for i in range(5):
    for j in range(3):
        yticks = ax[i,j].yaxis.get_major_ticks()
        yticks[0].label1.set_visible(False)
        yticks[-1].label1.set_visible(False)

# Save for the paper!
if fig_folder:
    plt.savefig(fig_folder+"param_dep.pdf")
```



Use different parameterisations of the MRP and check how well they perform

In this example, we use the framework of the `reparameterise` module to check the performance of some of the different parameterizations of the MRP. The idea is to set up large samples of halos with the same MRP parameters at various truncation masses, and check the covariance of the parameters at the solution in each case, for different parameterisations. Then compare the covariance in each case.

Figure from this example is found in MRP appendix B

Create Data

First we're going to set some parameters which we'll use throughout. These typify what we expect the HMF to be around a redshift of 0 (roughly!).

```
In [2]: # Imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import os

        from mrpy.extra import reparameterise as repar
        from mrpy import MRP
        from mrpy.fitting.fit_sample import SimFit
```

```
In [7]: fig_folder=""
```

```
In [3]: # MRP parameters
        hs = 14.5
```

```
        alpha = -1.85
        beta = 0.72
        mmins = [10.0,10.5,11.0,11.5,12.0,12.5,13.0]
        V = 1

        nbar = 1e6

        # Number of samples
        Nreal = 20

        # Parameters defining outcomes
        maxj = 1
```

In [6]:
```
        force_recalc = False

        if os.path.exists("parameterisations/data.npz") and not force_recalc:
            f = np.load("parameterisations/data.npz")
            gg2_mean = f['gg2_mean']
            gg3_mean = f['gg3_mean']
            ht_mean =  f['ht_mean']
            gg2_sd   = f['gg2_sd']
            gg3_sd   = f['gg3_sd']
            ht_sd    = f['ht_sd']

        else:
            # Instantiate our final result arrays
            gg2 = np.zeros((Nreal,len(mmins),3,2))
            gg3 = np.zeros((Nreal,len(mmins),3,2))
            ht = np.zeros((Nreal,len(mmins),3,2))

            np.random.seed(1010)


            for im,mmin in enumerate(mmins):
                mm = MRP(None, logHs=hs, alpha=alpha,beta=beta, norm=0, log_mmin=mmin)
                lnA = np.log(nbar/mm.nbar)

                for i in range(Nreal):

                    # The following checks if there are any oddities with the covariance.
                    # Since the covariance is derived numerically from the analytic hessian,
                    # if the hessian is extremely correlated, the matrix inversion can fail.
                    # In such a case, we re-draw the samples.
                    do = True
                    j=0
                    while do and j<maxj:
                        logm = np.log10(mm.stats.rvs(np.random.poisson(nbar)))

                        #Fit the data
                        fit = SimFit(10**logm, mmin=10**mmin).run_downhill(hs,alpha,beta,lnA)[0].x

                        gg2c = repar.GG2Sample(logm=logm,log_mmin=mmin,logHs=fit[0],alpha=fit[1],beta

                        gg2[i,im,:,0] = np.sqrt(np.diag(gg2c.cov_ratio)[:3]) * gg2c.theta_T()/gg2c.p
                        do = np.any(np.logical_or(np.isnan(gg2[i,im,:,0]),np.isinf(gg2[i,im,:,0])))
                        j+=1
                    if j>maxj:
                        print "ITERATION %s ON MMIN=%s HAD NANS: "%(i,mmin)
```

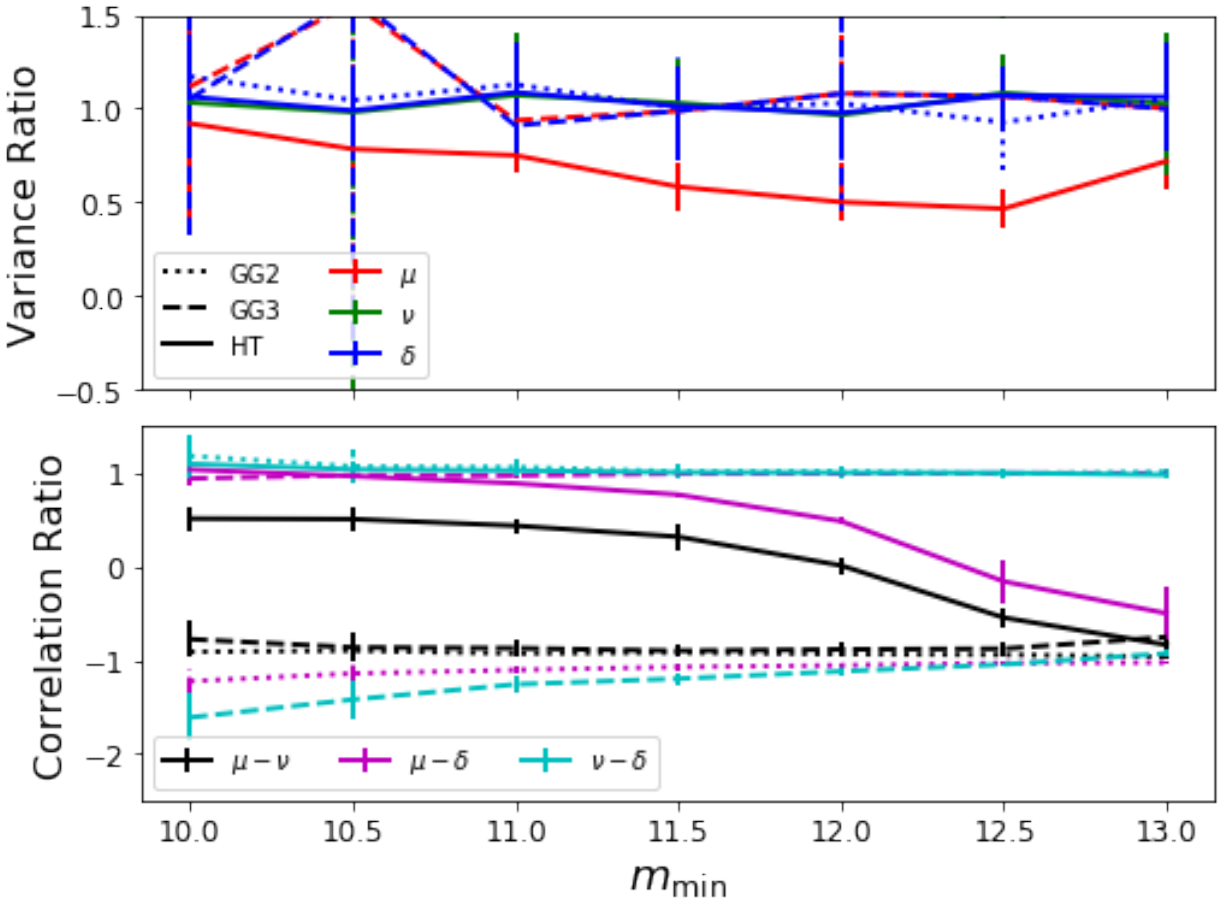
```
                # GG2 results
                gg2[i,im,:,1] = gg2c.corr_ratio[0,1],gg2c.corr_ratio[0,2],gg2c.corr_ratio[1,2]

                # GG3 results
                gg3c = repar.GG3Sample(logm=logm,log_mmin=mmin,logHs=fit[0],alpha=fit[1],beta=fit
                gg3[i,im,:,0] = np.sqrt(np.diag(gg3c.cov_ratio)[:3])* gg3c.theta_T()/gg3c.p_T()
                gg3[i,im,:,1] = gg3c.corr_ratio[0,1],gg3c.corr_ratio[0,2],gg3c.corr_ratio[1,2]

                # HT results
                htc = repar.HTSample(logm=logm,log_mmin=mmin,logHs=fit[0],alpha=fit[1],beta=fit[2
                ht[i,im,:,0] = np.sqrt(np.diag(htc.cov_ratio)[:3])* htc.theta_T()/htc.p_T()
                ht[i,im,:,1] = htc.corr_ratio[0,1],htc.corr_ratio[0,2],htc.corr_ratio[1,2]

        # Take the mean over the first axis
        gg2_mean = np.nanmean(gg2,axis=0)
        gg3_mean = np.nanmean(gg3,axis=0)
        ht_mean = np.nanmean(ht,axis=0)
        gg2_sd = np.nanstd(gg2,axis=0)
        gg3_sd = np.nanstd(gg3,axis=0)
        ht_sd = np.nanstd(ht,axis=0)

        # Save the data
        np.savez("parameterisations/data.npz",gg2_mean=gg2_mean,gg3_mean=gg3_mean,
                 ht_mean=ht_mean,gg2_sd=gg2_sd,gg3_sd=gg3_sd,ht_sd=ht_sd)
```

Make a Plot

We want to make a plot which shows the relative variance/covariance in parameters of the different forms over the different truncation masses. In the top panel we'll plot the difference in variance of the parameters. The smaller the better. In the bottom panel, we'll plot the correlation of the parameter combinations. In this case, the closer to 0 the better. In each case, what is plotted is the *ratio* of the value to that in the case of the MRP. We notate the transformation parameters as μ, ν, δ, and compare directly to $\log \mathcal{H}_\star, \alpha, \beta$.

```
In [8]: fig,ax=plt.subplots(2,1,sharex=True,figsize=(8,6))

        ax[0].plot([np.nan],[np.nan],label="GG2",color="k",ls=":",lw=2)
        ax[0].plot([np.nan],[np.nan],label="GG3",color="k",ls="--",lw=2)
        ax[0].plot([np.nan],[np.nan],label="HT",color="k",lw=2)

        for ip,par in enumerate([r"$\mu$",r"$\nu$",r"$\delta$"]):
            col1=['r','g','b'][ip]
            eb = ax[0].errorbar(mmins,gg2_mean[:,ip,0],gg2_sd[:,ip,0],ls=':',color=col1,lw=2)
            eb[-1][0].set_linestyle(":")
            eb=ax[0].errorbar(mmins,gg3_mean[:,ip,0],gg3_sd[:,ip,0],ls="--",color=col1,lw=2)
            eb[-1][0].set_linestyle("--")
            ax[0].errorbar(mmins,ht_mean[:,ip,0],ht_sd[:,ip,0],color=col1,label=par,lw=2)

        for ip, cmb in enumerate([r"$\mu-\nu$",r"$\mu-\delta$",r"$\nu-\delta$"]):
            col2=['k','m','c'][ip]
            eb=ax[1].errorbar(mmins,gg2_mean[:,ip,1],gg2_sd[:,ip,1],ls=':',color=col2,lw=2)
            eb[-1][0].set_linestyle(":")
            ax[1].errorbar(mmins,gg3_mean[:,ip,1],gg3_sd[:,ip,1],ls="--",color=col2,lw=2)
            eb[-1][0].set_linestyle("--")
            ax[1].errorbar(mmins,ht_mean[:,ip,1],ht_sd[:,ip,1],label=cmb,color=col2,lw=2)

        ax[0].set_ylim((-0.5,1.5))
        ax[0].legend(loc=0,ncol=2)
```

```
ax[1].legend(loc=0,ncol=3)
plt.subplots_adjust(hspace=0.1)
ax[1].set_ylim((-2.5,1.5))
ax[-1].set_xlabel(r"$m_{\rm min}$",fontsize=18)

for i in range(2):
    ax[i].tick_params(axis='both', which='major', labelsize=12)

ax[0].set_ylabel("Variance Ratio",fontsize=16)
ax[1].set_ylabel("Correlation Ratio",fontsize=16)

#Save for the paper!
if fig_folder:
    plt.savefig(join(fig_folder,"reparameterisations_vs_mmin.pdf"))
```



Use the MRP to explore the relation between halo mass and stellar mass

In this example, we'll use the MRP function for something other than pure halo mass functions. We'll use it to find the subhalo abundance-matched (SHAM) stellar-mass halo-mass relation (i.e. how much stellar mass does a halo of mass m typically contain?).

This can be done purely numerically, but we'll show (as per section 5 of the MRP paper) that using the analytic nature of the MRP means we can do slightly better than pure numerics.

Plots from this example appear in MRP as figures 14, 15

Motivation and numerical solution

The SHAM approximation for calculating the relationship between stellar and halo mass is simply to equate the cumulative number densities at every mass:

$$n_g(> m_\star) = n_h(> m_h).$$

The galaxy stellar mass function (GSMF) is commonly parameterised as a double-Schechter function, which has the integral

$$n_g(> m_\star) = \Phi_1 \Gamma\left(\alpha_1 + 1, \frac{m_\star}{M_\star}\right) + \Phi_2 \Gamma\left(\alpha_2 + 1, \frac{m_\star}{M_\star}\right),$$

which when equated to the MRP gives us the following equation to solve:

$$A\mathcal{H}_\star \Gamma(z_h, x_h) = \Phi_1 \Gamma\left(\alpha_1 + 1, \frac{m_\star}{M_\star}\right) + \Phi_2 \Gamma\left(\alpha_2 + 1, \frac{m_\star}{M_\star}\right),$$

where we use the common definitions from the MRP paper: $z_h = (\alpha_h + 1)/\beta$ and $x_h = (m_h/\mathcal{H}_\star)^\beta$.

NOTE: the *subhalo* mass function is to be used here, since it is the subhalos which contain galaxies. The MRP is quite able to fit the sHMF as well.

So, we'd hope we could just go and solve this equation for $m_\star$ as a function of x_h. However, unfortunately, the solution is analytically impossible (though maybe you could try?).

The first idea then would be to solve it purely numerically, which is what we're going to do in this section.

First, some general imports and parameters:

```
In [4]: # Some imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        from mrpy.extra.physical_dependence import mrp_params_b13
        from mrpy import MRP
        from scipy.interpolate import InterpolatedUnivariateSpline as spline
        from scipy.optimize import newton, curve_fit
        from mpmath import gammainc, log, exp
```

```
In [5]: # Some parameters
        z = 0
        H0 = 67.04
        mhalo = np.linspace(9,15.5,200) # vector of halo masses to plot.
```

Now define the MRP explicitly, based on best parameters for z=0

```
In [6]: # Get relevant MRP parameters at z=0 for Planck cosmology
        (logHs,alpha,beta,lnA) = mrp_params_b13(z=0)

        # Create an MRP instance with those params
        mrp = MRP(mhalo,logHs,alpha,beta,norm=lnA)
```

We'll also want to define the GSMF explicitly. We use the data from Baldry+12 to define it:

```
In [7]: # Parameters (Baldry+12 defines densities in H0=100 units and masses in H0=70.0 units)
        logMs=10.66 + np.log10(0.7) + np.log10(70.0/H0)
        alpha1=-0.35
        log_phi1=-2.402+3*np.log10(100./H0)
        alpha2=-1.47
        log_phi2=-3.1+3*np.log10(100./H0)
```

```
        # One way to define a double-schechter function is to add two MRP's with beta=1:
        def ngtm_dblschech(m,logMs,alpha1,log_phi1,alpha2,log_phi2):
            schech1 = MRP(m,logMs,alpha1,1.0,norm=np.log(10)*(log_phi1 - logMs))
            schech2 = MRP(m,logMs,alpha2,1.0,norm=np.log(10)*(log_phi2 - logMs))

            return schech1.ngtm() + schech2.ngtm()

        # The GMSF
        ngtm_baldry = ngtm_dblschech(mhalo,logMs,alpha1,log_phi1,alpha2,log_phi2)
```

Now that we have the left-hand and right hand sides, we can solve.

The basic idea is to find which mass in the GSMF has the same value for its $n(> m_\star)$ as a particular mass in the MRP. You can think of this as plotting the two functions together, and then drawing horizontal arrow from one to the other (see below!).

The obvious way to do this is by using splines, and that is the best way to think about it. However, numerically, this is pretty non-robust, since then the GSMF has to be calculated over a range which covers the whole HMF, before we know what that range is.

We rather do this calculation by root-finding. So we write a function which generally solves the double schechter function for x given an output q. Then we relate this to our actual problem. For the root-finding, we use the `gammainc` function straight from `mpmath`, since we need to retain as much precision as possible, to enlarge the mass range within which this works. Also, to get more reliability, we implement an analytic derivative.

```
In [8]: def inv_inc_gamma_dbl(a1,a2,n1,n2,q,x0=0):
            """
            Returns x for q = n1*Gamma(a1,x)+n2*Gamma(a2,x).

            This is general, so supports a<0, but is slow, uses root finding to generate
            the solution.
            """

            def obj_func(x):

                x=np.exp(x)
                p1 = n1*gammainc(a1,x)
                p2 = n2*gammainc(a2,x)
                rat = float(p1/p2)

                if rat>5:
                    func = float(log(p1) + np.log1p(1/rat) - log(q))
                elif rat<0.2:
                    func = float(log(p2) + np.log1p(rat) - log(q))
                else:
                    func = float(log(p1+p2) - log(q))

                return func#,fprime#,fprime2

            def fprime(x):
                x=np.exp(x)
                p1 = n1*gammainc(a1,x)
                p2 = n2*gammainc(a2,x)

                bottom = (p1+p2)
                top = n1*x**a1 + n2*x**a2
                y = exp(-x)
                fprime = -float(y*top/(bottom))

                return fprime
```

```
        res = newton(obj_func,x0=x0,fprime=fprime)#,fprime2=True)
        return np.exp(res)

    def mh_to_ms_schech(log_mh,ngtm,logMs,alpha1,log_phi1,alpha2,log_phi2):
        """
        Generate the SMHM relation ms(mh) for a black-box mass function and single or
        double Schechter SMF.

        Parameters
        ----------
        log_mh : array_like
            log10 halo masses

        ngtm : array_like
            Cumulative mass function.

        Ms, alpha1, alpha2, log_phi1, log_phi2 : float
            Double-schechter parameters. log_phi is in log10.

        Returns
        -------
        ms : array_like
            The log10 stellar masses corresponding to mh.
        """
        # Make sure mh is an array
        log_mh = np.atleast_1d(log_mh)

        ms = np.zeros_like(log_mh)
        for i,ng in enumerate(ngtm):
            ms[i] = inv_inc_gamma_dbl(alpha1+1,alpha2+1,10**log_phi1,10**log_phi2,ng,x0=ms[min(i-

        return np.log10(ms)+logMs
```

Finally we write a convenience function that calculates the transform from mhalo to mstar in both directions, and also from either to the fraction mstar/mhalo.

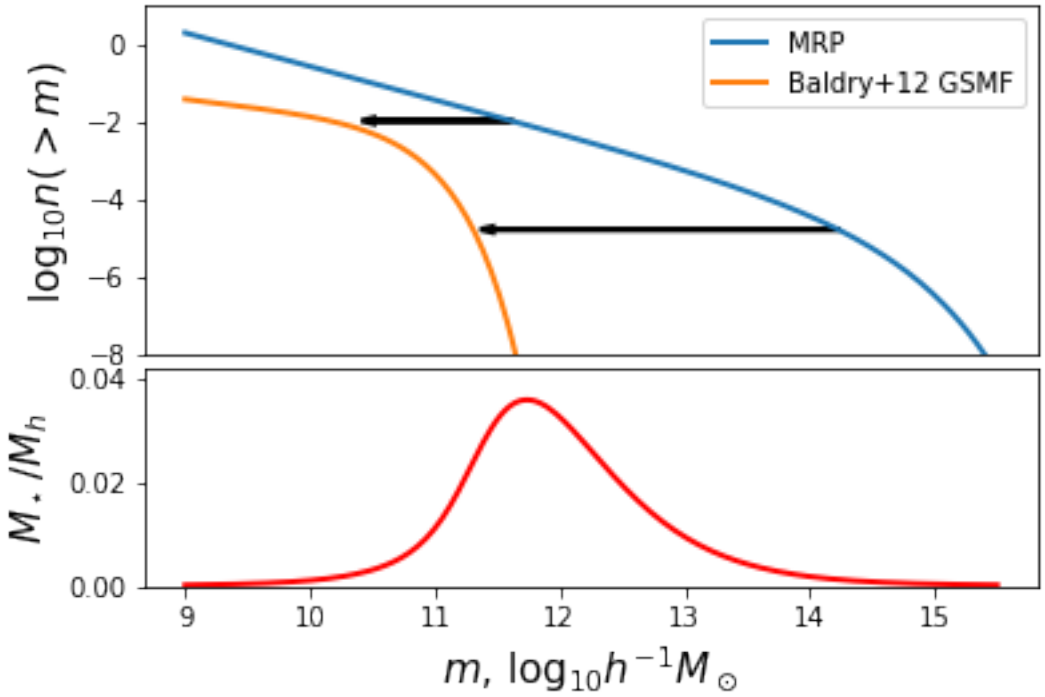
```
In [9]: def get_all_relations(log_mhalo, mrp, **gsmf_kwargs):
            ms = mh_to_ms_schech(log_mhalo, mrp, **gsmf_kwargs)

            ms_to_mh = lambda sm : spline(ms,log_mhalo)(sm)
            mh_to_ms = spline(log_mhalo,ms)
            ms_to_frac = spline(ms,10**(ms-log_mhalo))
            mh_to_frac = spline(log_mhalo,10**(ms-log_mhalo))
            return ms_to_mh, mh_to_ms,ms_to_frac,mh_to_frac

In [10]: mstar_to_mhalo, mhalo_to_mstar, mstar_to_frac, mhalo_to_frac = get_all_relations(mhalo, mrp
                                                                                          alpha1=alph
                                                                                         alpha2=alpha
```

And we can make a plot. We'll make a two-panel plot with the upper panel showing the schematic of what we're doing, and the bottom showing the numerical result.

```
In [11]: # Set up the basic figure with two subplots (one will be blank for now)
         fig,ax= plt.subplots(2,1,gridspec_kw={"height_ratios":(1.6,1)})

         # Plot both n(>m)
         ax[0].plot(mhalo,np.log10(mrp.ngtm()),label="MRP",lw=2)
         ax[0].plot(mhalo,np.log10(ngtm_baldry),label="Baldry+12 GSMF",lw=2)
```

```
        # Axis stylings
        ax[0].legend(loc=0)
        ax[0].set_ylim((-8,1))
        ax[0].set_ylabel(r"$\log_{10} n(>m)$",fontsize=15)

        # Here we want to draw horizontal arrows connecting the MRP to the GSMF.
        # These will signify the masses of each that have the same value of n(>m), i.e., the SMHM re
        n1 = int(len(mhalo)*2.0/5.0)
        dx1 = np.log10(mhalo_to_frac(mhalo[n1])) + 0.355
        ax[0].arrow(mhalo[n1],np.log10(mrp.ngtm()[n1]),dx1,0,head_width=0.15,head_length=0.1,lw=2)

        n2 = int(len(mhalo)*4.0/5.0)
        dx2 = np.log10(mhalo_to_frac(mhalo[n2])) + 0.155
        ax[0].arrow(mhalo[n2],np.log10(mrp.ngtm()[n2]),dx2,0,head_width=0.15,head_length=0.1,lw=2)


        # Now plot the actual SMHM relation in the bottom panel
        ax[1].plot(mhalo,mhalo_to_frac(mhalo),color="r",lw=2)

        # Bottom axis stylings
        ax[0].xaxis.set_ticks([])
        ax[1].set_ylabel(r"$M_\star/M_h$",fontsize=15)
        ax[1].set_xlabel(r"$m$, $\log_{10}h^{-1}M_\odot$",fontsize=15)
        ax[1].set_ylim((0,0.042))
        plt.subplots_adjust(hspace=0.05)


        # Save for the paper!
        fig.savefig("../../../mrpArticle/figures/numerical_solution.pdf")
```

```
/home/steven/miniconda3/envs/mrpy/lib/python2.7/site-packages/ipykernel/__main__.py:6: RuntimeWarning
```



Empirical Models from Literature

There are several models in the literature which attempt to parameterise the ratio m_star/m_h (i.e. to fit the red curve above). In this section, we'll define three of those, and show how they stack up to our numerical solution.

In this case all we really need is the conversion from m_h to the fraction, $f = m_star/m_h$. We'll take the models of Moster+09, Mutch+13 and Behroozi+10:

```
In [12]: ## Moster+09 (default arguments are those given in the paper)
         def mh_to_frac_moster(mh,f0=0.0282,m1=11.884,beta=1.057,gamma=0.556):
             return 2*f0/(10**(-beta*(mh-m1)) +10**(gamma*(mh-m1)))

         ## Mutch+13
         def mh_to_frac_mutch(mh,eps=0.17*0.9,mpeak=11.6,sigma=0.56):
             return  eps * np.exp(-(mh-mpeak)**2/sigma**2)


         ## Behroozi+10 -- this is a bit more complicated since Behroozi defines m_h(m_s) rather thar
         def ms_to_mh_behroozi(ms,m1=10.72,m1_a=0.55,m0=12.35,m0_a=0.28,beta=0.44,
                               beta_a=0.18,delta=0.57,delta_a=0.17,gamma=1.56,gamma_a=2.51,
                               z=0):
             # Get evolution of parameters
             a=1./(1.+z)
             m1 += m1_a*(a-1)
             m0 += m0_a*(a-1)
             beta += beta_a*(a-1)
             delta += delta_a*(a-1)
             gamma += gamma_a*(a-1)

             logx = ms - m1
             x = 10**logx
             return m0 + beta*logx + x**delta/(1+x**-gamma) - 0.5

         def mh_to_ms_func_behroozi(**kwargs):
             ms = np.linspace(6,13,1000)
             mh = ms_to_mh_behroozi(ms,**kwargs)
             return spline(mh,ms)

         def mh_to_frac_behroozi(mh,**kwargs):
             msfunc = mh_to_ms_func_behroozi(**kwargs)
             return 10**(msfunc(mh)-mh)
```

With these defined, we can fit their parameters to our numerical solution, since in this case we care more about how the model itself performs, rather than the actual values of the parameters. To do this, we just use a simple curve_fit method. We perform the fit in log-log space, since that performs the best over a large mass range.

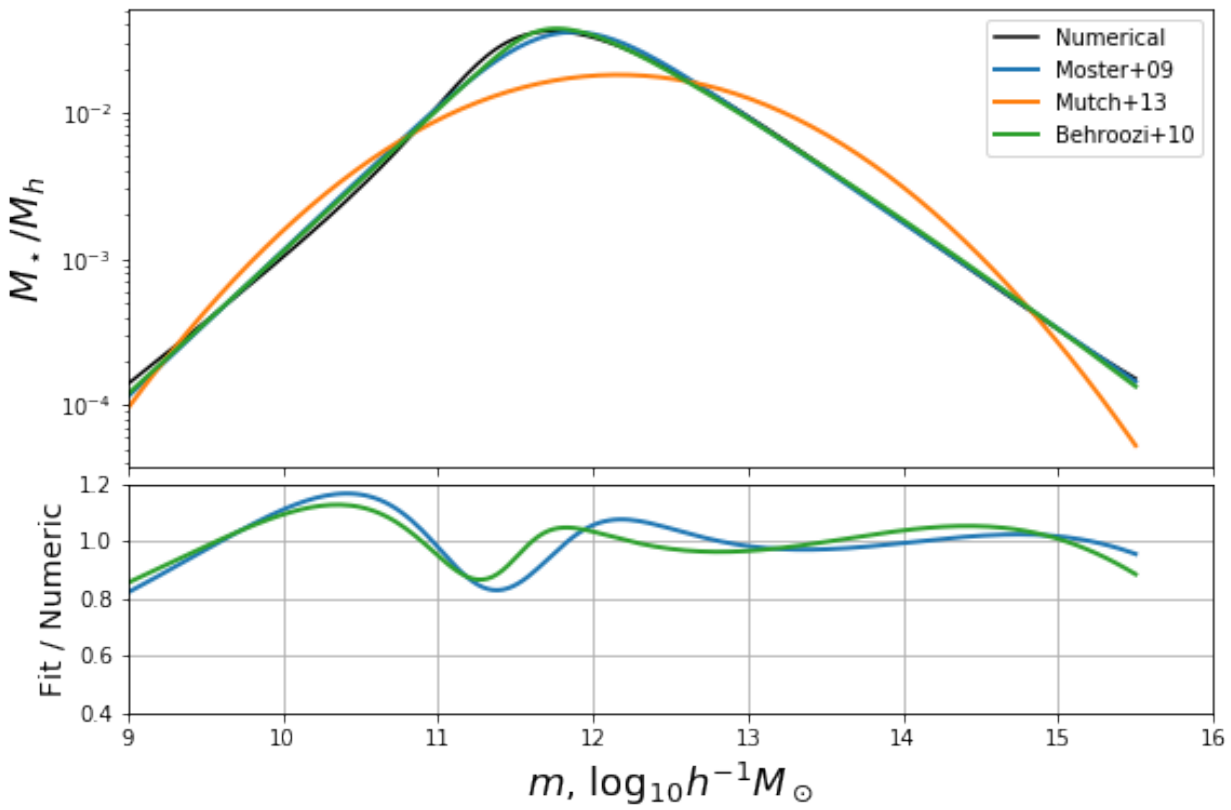
```
In [13]: # Log of the numerical solution
         logf = np.log(mhalo_to_frac(mhalo))

         # ---------- MOSTER ------------------------------------------------------
         # Define an objective function to minimize for Moster+09
         def moster_obj_func(mh, *args) :
             return np.log(mh_to_frac_moster(mh,*args))

         # Minimize the function
         moster_res = curve_fit(moster_obj_func,mhalo,logf,p0=(0.0282,11.884,1.057,0.556))[0]

         # ---------- MUTCH -------------------------------------------------------
```

```
        # Define an objective function to minimize for Moster+09
        def mutch_obj_func(mh, *args) :
            return np.log(mh_to_frac_mutch(mh,*args))

        # Minimize the function
        mutch_res = curve_fit(mutch_obj_func,mhalo,logf,p0=(0.15,11.6,0.56))[0]


        # ---------- BEHROOZI -----------------------------------------------
        # Define an objective function to minimize for Moster+09
        def behroozi_obj_func(mh,m0,m1,beta,delta,gamma) :
            return np.log(mh_to_frac_behroozi(mh,m0=m0,m1=m1,beta=beta,delta=delta,gamma=gamma))

        # Minimize the function
        behroozi_res = curve_fit(behroozi_obj_func,mhalo,logf,p0=(12.35,10.72,0.44,0.57,1.56))[0]
```

```
/home/steven/miniconda3/envs/mrpy/lib/python2.7/site-packages/ipykernel/__main__.py:15: RuntimeWarnin
```

Now plot the solutions against the numerical result:

```
In [14]: def plotbase():
             fig,ax = plt.subplots(2,1,sharex=True,gridspec_kw={"height_ratios":(2,1)},figsize=(9,6))

             ax[0].plot(mhalo,mhalo_to_frac(mhalo),label="Numerical",color="k")

             # Plot Moster results
             ax[0].plot(mhalo,mh_to_frac_moster(mhalo,*moster_res),label="Moster+09",color="C0",lw=2)
             ax[1].plot(mhalo,mh_to_frac_moster(mhalo,*moster_res)/mhalo_to_frac(mhalo),color="C0",lw

             # Plot Mutch results
             ax[0].plot(mhalo,mh_to_frac_mutch(mhalo,*mutch_res),label="Mutch+13",color="C1",lw=2)

             # Plot Behroozi Results
             ax[0].plot(mhalo,np.exp(behroozi_obj_func(mhalo,*behroozi_res)),label="Behroozi+10",colo
             ax[1].plot(mhalo,np.exp(behroozi_obj_func(mhalo,*behroozi_res))/mhalo_to_frac(mhalo),col

             # Plot Stylings
             ax[0].legend(loc=0)
             ax[0].set_yscale('log')
             ax[1].set_ylim((0.4,1.2))
             ax[0].set_xlim((9,16))
             ax[1].grid(True)
             ax[0].set_ylabel(r"$M_\star/M_h$",fontsize=18)
             ax[1].set_ylabel("Fit / Numeric",fontsize=14)
             ax[1].set_xlabel(r"$m$, $\log_{10}h^{-1}M_\odot$",fontsize=18)
             plt.subplots_adjust(hspace=0.05)
             return fig,ax
         fig,ax = plotbase()
```



Constraining Parameters via Analytic Limits

We now investigate the behaviour of the parameterisations based on what we can learn from analytic approximations to the solution. Though we cannot solve the full equation analytically for all m_h, we do well to identify its behaviour in the limits.

As $x \to 0$, we have the identity $\Gamma(z, x) \to -x^z/z$, for $z < 0$. The speed of convergence of this limit depends heavily on the value of z (the more negative, the faster the convergence).

In this case, at small mass, the α_2 term dominates the α_1 term (which generally has positive shape parameter), so we can simply write

$$-\frac{\Phi_2 \left(\frac{m_\star}{M_\star}\right)^{(\alpha_2+1)}}{\alpha_2+1} = -\frac{A\mathcal{H}_\star x_h^{z_h}}{z_h},$$

so that we find that the ratio is a power law:

$$\frac{m_\star}{m_h} = K m_h^p, \tag{3.1}$$

$$p = \frac{\alpha_h - \alpha_2}{\alpha_2 + 1} \tag{3.2}$$

$$K = M_\star \mathcal{H}_\star^{-\frac{\alpha_h}{\alpha_2+1}} \left(\frac{A(\alpha_2+1)}{\Phi_2 z_h}\right)^{\frac{1}{\alpha_2+1}}, \tag{3.3}$$

While M13 cannot replicate this behaviour in the low-mass limit, both M09 and B10 can. Specifically, M09 is equivalent in the low-mass limit if $\beta = p$ and $2f_0/m_1^p = K$, while B10 requires $\beta = 1/(p+1)$ and $m_1(\sqrt{10}/m_0)^{1/\beta} = K$.

Let's write a function to get these constants of the low-mass limit:

```
In [15]: def set_lowmass_const(alpha_num,alpha_den, Ms_num,Ms_den,norm_num,norm_den,
                               beta_num,beta_den):
             """
             Sets p, K for analytic approximation of low-mass relationship.

             Parameters
             ----------
             alpha_num, alpha_den : float
                 slopes of the numerator and denominator mass functions respectively
                 (eg. for SMHM, this would alpha_smf, and alpha_hmf).

             Ms_num, Ms_den : float
                 Characteristic log10 masses of numerator and denominator mf's.
                 (eg. for SMHM, this would be Ms, Hs)

             norm_num, norm_den : float
                 Normalisation of numerator and denominator mf's.
                 (eg. for SMHM, this would be phi, A*10^Hs)

             beta_num, beta_den : float
                 Cut-off parameter for num and den mf's.
                 (eg. for SMHM, this would be 1, beta)
             """
             z_num = (alpha_num + 1)/beta_num
             z_den = (alpha_den + 1)/beta_den
             p = (alpha_den-alpha_num)/z_num
             scale = 10**(Ms_num-Ms_den)
             z_rat = z_num/z_den
             N = norm_den/norm_num
             K = scale * (N * z_rat)**(1.0/z_num)*10**(-Ms_den*p)

             return p,K

In [16]: # Actually set the p,K in our case
         p,K = set_lowmass_const(alpha2,alpha, logMs,logHs,10**log_phi2,np.exp(lnA)*10**logHs,1.0,bet
```

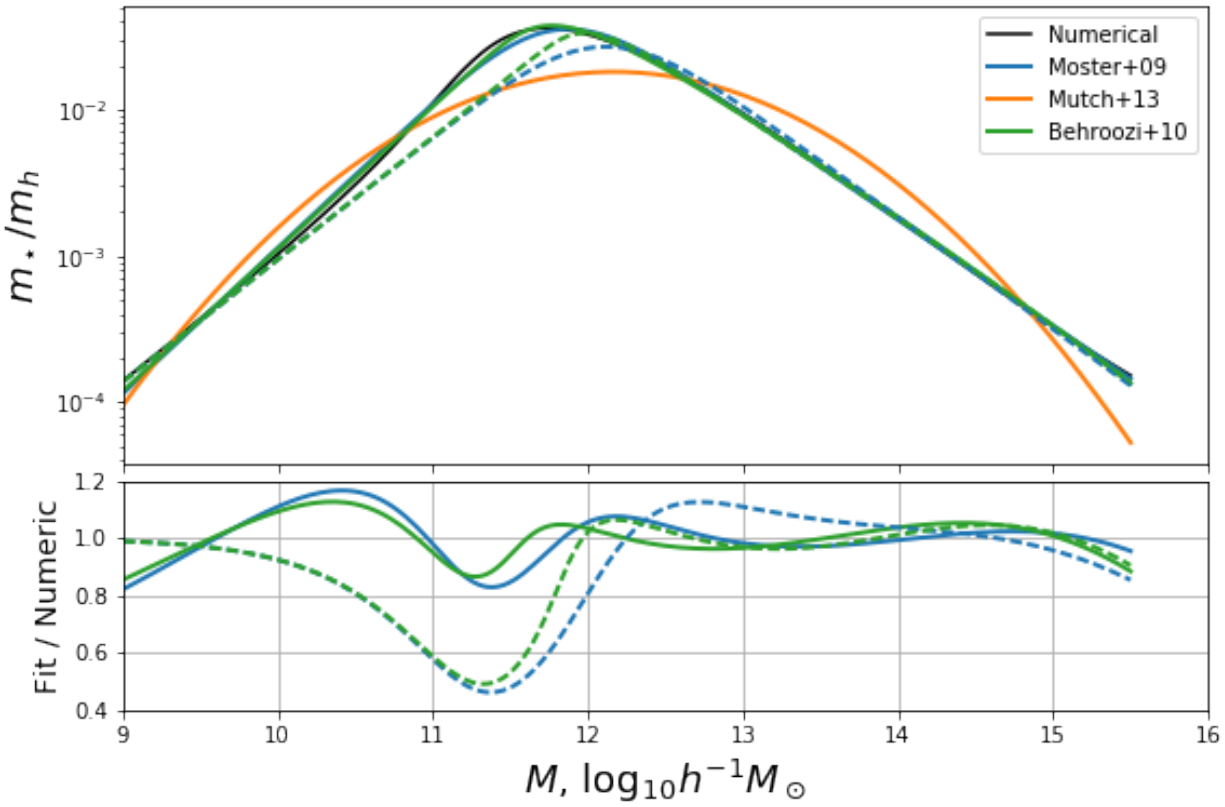
Now we can re-write our objective functions for Moster+09 and Behroozi+10 so that their constants are set by the low-mass behaviour, and rederive solutions.

```
In [17]: def moster_fixed_obj_func(mh,m1,gamma):
             f0 = 10**(m1*p) * K/2
             beta = p
             return np.log(mh_to_frac_moster(mh,f0,m1,beta,gamma))

         moster_fixed_res = curve_fit(moster_fixed_obj_func,mhalo,logf,p0=(11.884,0.6))[0]

         def behroozi_fixed_obj_func(mh,m0,delta,gamma):
             beta = 1./(p+1)
             m1 = np.log10(K) + (m0-0.5)/beta
             return np.log(mh_to_frac_behroozi(mh,m1=m1,m0=m0,beta=beta,delta=delta,gamma=gamma))

         behroozi_fixed_res = curve_fit(behroozi_fixed_obj_func,mhalo,logf,p0=(12.35,0.57,1.56))[0]
```

Plot the solutions along with the plot from before. For some reason in the notebook the figures get closed after every cell so we have to recall the earlier plot.

```
In [23]: fig,ax = plotbase()

         def fixed_additions():
             ax[0].plot(mhalo,np.exp(moster_fixed_obj_func(mhalo,*moster_fixed_res)),color="C0",ls='-
             ax[1].plot(mhalo,np.exp(moster_fixed_obj_func(mhalo,*moster_fixed_res))/mhalo_to_frac(mh
```

```
        ax[0].plot(mhalo,np.exp(behroozi_fixed_obj_func(mhalo,*behroozi_fixed_res)),color="C2",l
        ax[1].plot(mhalo,np.exp(behroozi_fixed_obj_func(mhalo,*behroozi_fixed_res))/mhalo_to_fra

        # add label to fig
        ax[0].plot([0],[0],color='k',ls='--',label="Fixed low-mass")
        return fig, ax

fig,ax = fixed_additions()
```



We can see clearly that the asymptotic behaviour at low mass for the dashed curves we just added tends towards unity.

Defining our own extended model

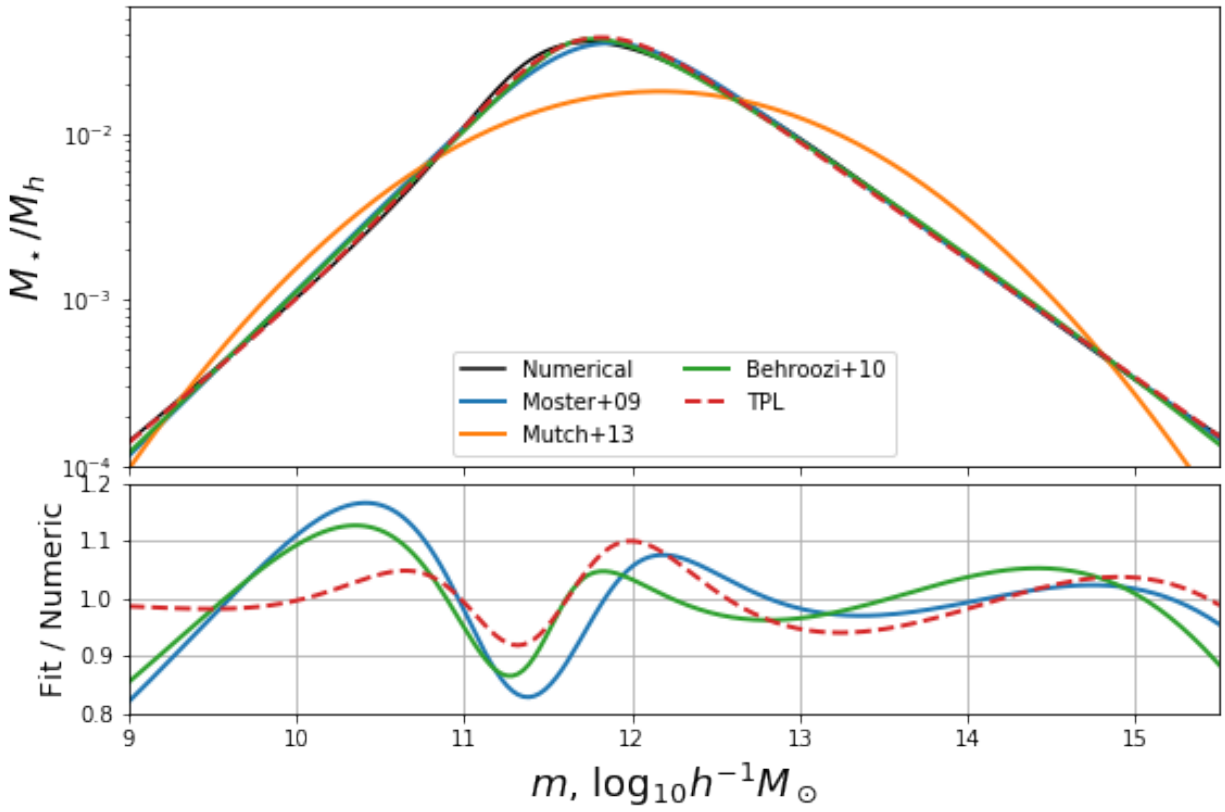
The most severe deficiency of both models, and this is exacerbated by correctly setting the low-mass behaviour, is the position of the turning point. A simple way to change the position of the turning point without affecting the behaviour in the limits, is to use an extension of M09:

$$\frac{m_\star}{m_h} = \frac{w\left(\frac{m_h}{m_1}\right)^{-\delta} + 2f_0}{\left(\frac{m_h}{m_1}\right)^{-\beta} + w\left(\frac{m_h}{m_1}\right)^{\gamma} + k}, \tag{3.4}$$

where w controls the peak position, k is able to correct the amplitude of the high-mass power-law, and δ adds the flexibility needed to induce an upturn left of the turning point. This extension obeys the same relations as M09 in terms of the low-mass approximation, but has the difference that the high-mass power law has the slope $\delta - \gamma$.

We can define this function, and its objective function for optimization:

```
In [18]: def mh_to_frac_triple_pl(mh,f0=0.0282,m1=11.884,beta=1.057,gamma=0.556,delta=0,k=0,w=1):
             return (10**(w*delta*(mh-m1)) +2*f0)/(10**(-beta*(mh-m1)) +w*10**(gamma*(mh-m1))+k)

         def triple_pl_obj_func( mh,m1,gamma,delta,k,w):
             f0 = 10**(m1*p) * K/2
             beta = p
             return np.log(mh_to_frac_triple_pl(mhalo,f0,m1,beta,gamma,delta,k,w))

         triple_pl_res = curve_fit(triple_pl_obj_func,mhalo,logf,p0=(11.884,2.6,2,1,1))

In [21]: fig,ax = plotbase()
         #fig,ax = fixed_additions()

         ax[0].plot(mhalo,np.exp(triple_pl_obj_func(mhalo,*triple_pl_res[0])),label="TPL",color="C3",
         ax[1].plot(mhalo,np.exp(triple_pl_obj_func(mhalo,*triple_pl_res[0]))/mhalo_to_frac(mhalo),co
         ax[0].legend(loc=0,ncol=2)

         ax[0].set_ylim((1e-4,0.06))
         ax[0].set_xlim((9,15.5))
         ax[1].set_ylim(0.8,1.2)
         # Save for the paper!
         fig.savefig("../../../mrpArticle/figures/compare_mine_fixed.pdf")
```



Construct a custom fit for an extension of the MRP: a double-Schechter function.

Note: this example is almost identical to that found in the R package ‘‘tggd‘‘ in the ‘‘tggd_log‘‘ documentation.

While `mrpy` has several in-built routines which aid in fitting the MRP function to data, it does not natively support fitting *extensions* of the MRP, such as double-Schechter functions. For such functions, one can fairly simply create

custom fits using the methods found in Scipy, for example.

In this example, we create a double-Schechter galaxy stellar mass function (GSMF) down to a target stellar mass (xmin) of log10(SM) = 8. We use data from Baldry+2012 to define the function:

Both mixtures have

$$M_\star \equiv \log_{10} \mathcal{H}_\star = 10.66$$

$$and$$

$$\beta = 1$$

Mixture 1 has:

$$\alpha_1 = -1.47$$

$$\phi_1^\star = 0.79 \times 10^{-3}$$

Mixture 2 has:

$$\alpha_2 = -0.35$$

$$\phi_2^\star = 3.96 \times 10^{-3}$$

Furthermore, we use only the purely statistical routines of `mrpy` to achieve our results:

```
In [1]: # Imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        from scipy.stats import gaussian_kde
        from scipy.optimize import minimize

        from mrpy.base import stats
```

First we define objects which capture the statistical quantities of each mixture:

```
In [2]: mix1 = stats.TGGDlog(10.66,-1.47,1.0,8)
        mix2 = stats.TGGDlog(10.66,-0.35,1.0,8)
```

$\phi^\star$ is defined as the value of the pdf *in log-space* at the pivot scale by e, i.e.

$$\phi^\star = PDF(M_\star) \times \frac{e}{\ln(10)}$$

This normalisation is important in our sampling since it gives the ratio of samples produced by each mixture. We can produce the relevant normalisation using:

```
In [3]: M1norm=0.79/mix1.pdf(10.66)
        M2norm=3.96/mix2.pdf(10.66)
        Mtot=M1norm+M2norm
```

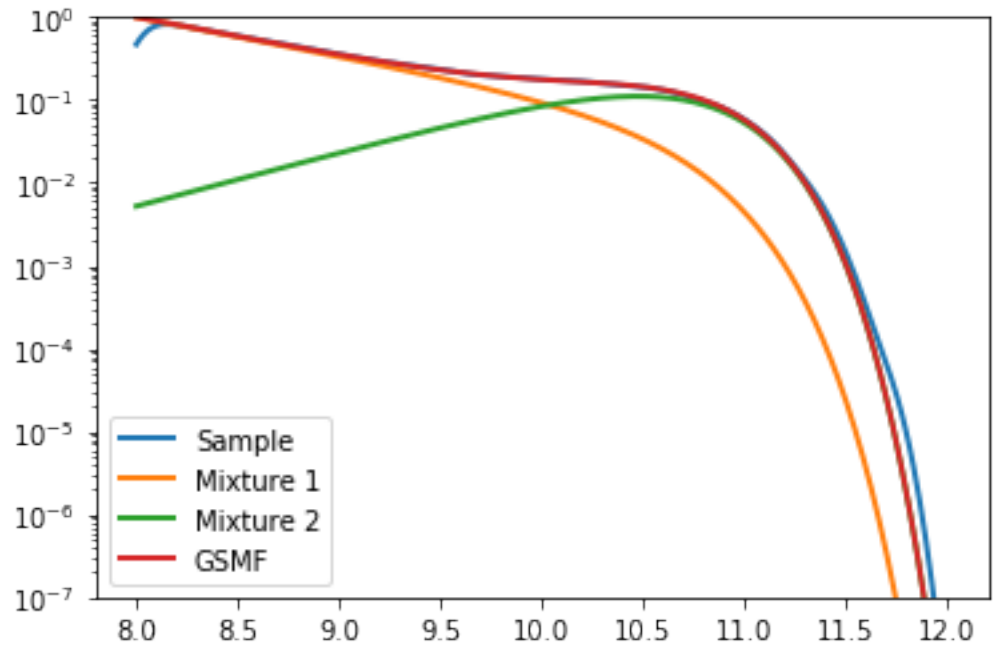
Now say we would like to sample 1e5 galaxies, we can produce these like so:

```
In [4]: Nsamp=1e5

        np.random.seed(100)

        mix1_sample = mix1.rvs(int(Nsamp*M1norm/Mtot))
        mix2_sample = mix2.rvs(int(Nsamp*M2norm/Mtot))

        gal_sample = np.concatenate((mix1_sample,mix2_sample))
```

Let's plot the distribution we've just created, to check if everything has worked okay:

```
In [5]: # Create x array to plot against
        x = np.linspace(8,12,401)

        plt.plot(x,gaussian_kde(gal_sample)(x),label="Sample", lw=2)

        plt.plot(x,mix1.pdf(x) * M1norm/Mtot,label="Mixture 1", lw=2)
        plt.plot(x,mix2.pdf(x) * M2norm/Mtot,label="Mixture 2", lw=2)
        plt.plot(x,mix1.pdf(x) * M1norm/Mtot + mix2.pdf(x) * M2norm/Mtot, label="GSMF", lw=2)
        plt.yscale('log')
        plt.ylim((1e-7,1))
        plt.legend(loc=0)
```

```
Out[5]: <matplotlib.legend.Legend at 0x7fa87a9d5210>
```



Now we can try to fit the mixed model. The trick here is we fit for the mixture using an additional parameter λ, where one component is multiplied by λ and the other $1 - \lambda$. We define it so that `M1norm/Mtot=lambda` and `M1norm/Mtot=1-lambda`. Here's our model:

```
In [6]: def mix_lnl(par,data):
            mix1 = stats.TGGDlog(par[0],par[1],1.0,8)
            mix2 = stats.TGGDlog(par[0],par[2],1.0,8)

            return -np.sum(np.log(mix1.pdf(data)*par[3] + mix2.pdf(data)*(1-par[3])))
```

Now we can perform the fit, using a downhill-gradient method of our choice. The fit is probably not fantastic though. Generalised Gamma distributions (including truncated ones) display poor convergence properties using ML. Full MCMC is a better route when trying to fit GSMF type data. And the data certainly should *not* be binned!

```
In [7]: GSMFfit = minimize(mix_lnl, x0=[10,-2,0,0.5], args=(gal_sample,), bounds=[(9,12),(-2.5,-0.5),

        print "Maximum likelihood parameters: ", GSMFfit.x
```

```
Maximum likelihood parameters:  [ 10.65427592  -1.46844631  -0.34935003   0.83445411]
```

This accords very well with our input parameters!

3.4.2 License

3.4.3 Changelog

Development Version

v1.1.0 [8th Jan 2018]

This version is the version used for all plots in Murray, Robotham, Power (2018), and is released along with that paper. There are many changes in the code from previous versions, the result of a couple of years of sporadic work.

v1.0.0

Features

- New `PerObjFit` class supersedes `get_fit_perobj` function, providing more coherent fitting capabilities.
- **Added heaps of "real-world" examples (used in MRP paper):**
  - https://github/steven-murray/mrpy/docs/examples/fit_curve_against_analytic.ipynb
  - https://github/steven-murray/mrpy/docs/examples/fit_simulation_suite.ipynb
  - https://github/steven-murray/mrpy/docs/examples/heirarchical_model_stan.ipynb
  - https://github/steven-murray/mrpy/docs/examples/explore_analytic_model.ipynb
  - https://github/steven-murray/mrpy/docs/examples/mmin_dependence.ipynb
  - https://github/steven-murray/mrpy/docs/examples/physical_dependence.ipynb
  - https://github/steven-murray/mrpy/docs/examples/parameterization_performance.ipynb
  - https://github/steven-murray/mrpy/docs/examples/SMHM.ipynb
- Added `model` argument to `fit_perobj_stan` to facilitate pickling of multiple fits.
- Added ability to send keyword arguments to priors in `PerObjFit` class
- Added a `normal_prior` function for simple normal priors.

Enhancements

- Changed default weighting from 1 to 0 in `get_fit_curve`.
- Added tests for the `PerObjLikeWeights` class.
- Added tests for `nbar` and `rhobar` for general `m` in ‘‘MRP‘ subclasses.
- Changed imports so that they wouldn’t show up in docs
- Many improvements to documentation (including this file!)

Bugfixes

- Fixed issue setting `log_mmin` in `IdealAnalytic`
- Fixed issue in which `nbar` and `rhobar` are wrong if `mmin` is not `m.min()` in `MRP` subclasses.

3.4.4 API Summary

mrpy.base.special	Definitions of all special functions used throughout *mrpy*.
mrpy.base.stats	A module defining the TGGD distribution (as well as in log and ln space) in standard R style.
mrpy.base.core	Basic MRP functionality, such as functions to generate the MRP with different normalisations.
mrpy.extra.physical_dependence	Module containing functions for the dependence of MRP parameters on physical parameters, defined with respect to Behroozi+13.
mrpy.extra.likelihoods	Provides classes which extend the basic `mrpy.core.MRP` class.
mrpy.extra.analytic_model	A module defining the likelihoods and related quantities involved when the data is purely ideal and analytic.
mrpy.extra.reparameterise	Provides classes which implement variations of the MRP, in which the parameters have been transformed.
mrpy.fitting.fit_curve	Routines that implement simple least-squares fits directly to dn/dm without errors.
mrpy.fitting.fit_sample	Routines that implement fits directly to samples of masses (or other values drawn from a TGGD).

mrpy.base.special

Definitions of all special functions used throughout *mrpy*.

Generally, these are adapted from *mpmath*, but return standard floats/arrays, and can take *array_like* input.

Functions

G1(z, x)	The Meijer-G function with specific arguments: `meijerg([[], [1, 1]], [[0, 0, z], []], x)`, normalised by the incomplete gamma function with arguments *z,x*.
G2(z, x)	The Meijer-G function with specific arguments: `meijerg([[], [1, 1,1]], [[0, 0, 0, z], []], x)`, normalised by the incomplete gamma function with arguments *z,x*.
gamma(z)	Gamma function.
gammainc(z, x)	Upper incomplete gamma function.
hyperReg_2F2(z, x)	The regularised hypergeometric function with specific arguments: `hyper([z,z],[z+1,z+1],-x)`.
polygamma(m, z)	Polygamma function.

mrpy.base.special.G1

mrpy.base.special.**G1**(*z*, *x*)

The Meijer-G function with specific arguments: `meijerg([[], [1, 1]], [[0, 0, z], []], x)`, normalised by the incomplete gamma function with arguments *z,x*.

Either *z* or *x* can be *array_like*.

Notes

This quantity arises in the derivative of the natural log of the incomplete gamma function.

$$\frac{d}{dz}\ln\Gamma(z,x) = G1(z,x) + \ln x$$

mrpy.base.special.G2

mrpy.base.special.**G2**(*z*, *x*)

The Meijer-G function with specific arguments: `meijerg([[], [1, 1,1]], [[0, 0, 0, z], []], x)`, normalised by the incomplete gamma function with arguments *z,x*.

Either *z* or *x* can be *array_like*.

Notes

This quantity arises in the derivative of *G1()*.

mrpy.base.special.gamma

mrpy.base.special.**gamma**(*z*)

Gamma function.

Note: This is exactly as defined by *mpmath*, but modified to take `array_like` arguments, and return results with type *float*.

Notes

Computes the gamma function, *Gamma(x)*. The gamma function is a shifted version of the ordinary factorial, satisfying *Gamma(n) = (n-1)!* for integers *n > 0*. More generally, it is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt$$

for any real or complex *x* with *Re(x) > 0* and for *Re(x) < 0* by analytic continuation.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(1, 6):
...     print("%s %s" % (k, gamma(k)))
...
1 1.0
2 1.0
3 2.0
4 6.0
5 24.0
>>> gamma(inf)
+inf
>>> gamma(0)
Traceback (most recent call last):
  ...
ValueError: gamma function pole
```

The gamma function of a half-integer is a rational multiple of *sqrt{pi}*:

```
>>> gamma(0.5), sqrt(pi)
(1.77245385090552, 1.77245385090552)
>>> gamma(1.5), sqrt(pi)/2
(0.886226925452758, 0.886226925452758)
```

We can check the integral definition:

```
>>> gamma(3.5)
3.32335097044784
>>> quad(lambda t: t**2.5*exp(-t), [0,inf])
3.32335097044784
```

`gamma()` supports arbitrary-precision evaluation and complex arguments:

```
>>> mp.dps = 50
>>> gamma(sqrt(3))
0.91510229697308632046045539308226554038315280564184
>>> mp.dps = 25
>>> gamma(2j)
(0.009902440080927490985955066 - 0.07595200133501806872408048j)
```

Arguments can also be large. Note that the gamma function grows very quickly:

```
>>> mp.dps = 15
>>> gamma(10**20)
1.9328495143101e+1956570551809674817225
```

mrpy.base.special.gammainc

mrpy.base.special.**gammainc**(*z*, *x*)

Upper incomplete gamma function.

Note: This is exactly as defined by *mpmath*, but modified to take `array_like` arguments, and return results with type *float*.

Notes

`gammainc(z, a=0, b=inf)` computes the (generalized) incomplete gamma function with integration limits *[a, b]*:

$$\Gamma(z,a,b) = \int_a^b t^{z-1} e^{-t}\, dt$$

The generalized incomplete gamma function reduces to the following special cases when one or both endpoints are fixed:

- *Gamma(z,0,infty)* is the standard ("complete") gamma function, *Gamma(z)* (available directly as the mpmath function `gamma()`)
- *Gamma(z,a,infty)* is the "upper" incomplete gamma function, *Gamma(z,a)*
- *Gamma(z,0,b)* is the "lower" incomplete gamma function, *gamma(z,b)*.

Of course, we have *Gamma(z,0,x)* + *Gamma(z,x,infty)* = *Gamma(z)* for all z and x.

Note however that some authors reverse the order of the arguments when defining the lower and upper incomplete gamma function, so one should be careful to get the correct definition.

If also given the keyword argument `regularized=True`, `gammainc()` computes the "regularized" incomplete gamma function

$$P(z,a,b) = \frac{\Gamma(z,a,b)}{\Gamma(z)}.$$

Examples

We can compare with numerical quadrature to verify that `gammainc()` computes the integral in the definition:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> gammainc(2+3j, 4, 10)
(0.00977212668627705160602312 - 0.0770637306312989892451977j)
>>> quad(lambda t: t**(2+3j-1) * exp(-t), [4, 10])
(0.00977212668627705160602312 - 0.0770637306312989892451977j)
```

Argument symmetries follow directly from the integral definition:

```
>>> gammainc(3, 4, 5) + gammainc(3, 5, 4)
0.0
>>> gammainc(3,0,2) + gammainc(3,2,4); gammainc(3,0,4)
1.523793388892911312363331
1.523793388892911312363331
>>> findroot(lambda z: gammainc(2,z,3), 1)
3.0
```

Evaluation for arbitrarily large arguments:

```
>>> gammainc(10, 100)
4.083660630910611272288592e-26
>>> gammainc(10, 10000000000000000)
5.2904024499011747529724 86e-4342944819032375
>>> gammainc(3+4j, 1000000+1000000j)
(-1.257913707524362408877881e-434284 + 2.556691003883483531962095e-434284j)
```

Evaluation of a generalized incomplete gamma function automatically chooses the representation that gives a more accurate result, depending on which parameter is larger:

```
>>> gammainc(10000000, 3) - gammainc(10000000, 2)    # Bad
0.0
>>> gammainc(10000000, 2, 3)    # Good
1.755146243738946045873491e+4771204
>>> gammainc(2, 0, 100000001) - gammainc(2, 0, 100000000)    # Bad
0.0
>>> gammainc(2, 100000000, 100000001)    # Good
4.078258353474186729184421e-43429441
```

The incomplete gamma functions satisfy simple recurrence relations:

```
>>> mp.dps = 25
>>> z, a = mpf(3.5), mpf(2)
>>> gammainc(z+1, a); z*gammainc(z,a) + a**z*exp(-a)
10.60130296933533459267329
10.60130296933533459267329
>>> gammainc(z+1,0,a); z*gammainc(z,0,a) - a**z*exp(-a)
1.030425427232114336470932
1.030425427232114336470932
```

Evaluation at integers and poles:

```
>>> gammainc(-3, -4, -5)
(-0.22145770489677985662341 92 + 0.0j)
>>> gammainc(-3, 0, 5)
+inf
```

If z is an integer, the recurrence reduces the incomplete gamma function to $P(a) \exp(-a) + Q(b) \exp(-b)$ where P and Q are polynomials:

```
>>> gammainc(1, 2); exp(-2)
0.13533528323661269189399 95
0.13533528323661269189399 95
>>> mp.dps = 50
>>> identify(gammainc(6, 1, 2), ['exp(-1)', 'exp(-2)'])
'(326*exp(-1) + (-872)*exp(-2))'
```

The incomplete gamma functions reduce to functions such as the exponential integral Ei and the error function for special arguments:

```
>>> mp.dps = 25
>>> gammainc(0, 4); -ei(-4)
0.0037793524098489064788 7486
0.0037793524098489064788 7486
>>> gammainc(0.5, 0, 2); sqrt(pi)*erf(sqrt(2))
```

```
1.691806732945198336509541
1.691806732945198336509541
```

mrpy.base.special.hyperReg_2F2

mrpy.base.special.**hyperReg_2F2**(*z*, *x*)

The regularised hypergeometric function with specific arguments: `hyper([z,z],[z+1,z+1],-x)`.

Parameters

z [array_like]

x [array_like]

Returns

array_like

mrpy.base.special.polygamma

mrpy.base.special.**polygamma**(*m*, *z*)

Polygamma function.

Note: This is exactly as defined by *mpmath*, but modified to take `array_like` arguments, and return results with type *float*.

Notes

Gives the polygamma function of order *m* of *z*, *psi^{(m)}(z)*. Special cases are known as the *digamma function* (*psi^{(0)}(z)*), the *trigamma function* (*psi^{(1)}(z)*), etc. The polygamma functions are defined as the logarithmic derivatives of the gamma function:

$$\psi^{(m)}(z) = \left(\frac{d}{dz}\right)^{m+1} \log \Gamma(z)$$

In particular, *psi^{(0)}(z) = Gamma'(z)/Gamma(z)*. In the present implementation of `psi()`, the order *m* must be a nonnegative integer, while the argument *z* may be an arbitrary complex number (with exception for the polygamma function's poles at *z = 0, -1, -2, ldots*).

Examples

For various rational arguments, the polygamma function reduces to a combination of standard mathematical constants:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> psi(0, 1), -euler
(-0.5772156649015328606065121, -0.5772156649015328606065121)
>>> psi(1, '1/4'), pi**2+8*catalan
(17.19732915450711073927132, 17.19732915450711073927132)
>>> psi(2, '1/2'), -14*apery
(-16.82879664423431999559633, -16.82879664423431999559633)
```

The polygamma functions are derivatives of each other:

```
>>> diff(lambda x: psi(3, x), pi), psi(4, pi)
(-0.110574931257886273452695 2, -0.110574931257886273452695 2)
>>> quad(lambda x: psi(4, x), [2, 3]), psi(3,3)-psi(3,2)
(-0.375, -0.375)
```

The digamma function diverges logarithmically as *z to infty*, while higher orders tend to zero:

```
>>> psi(0,inf), psi(1,inf), psi(2,inf)
(+inf, 0.0, 0.0)
```

Evaluation for a complex argument:

```
>>> psi(2, -1-2j)
(0.03902435405364952654838445 + 0.15743252404130299546853 66j)
```

Evaluation is supported for large orders *m* and/or large arguments *z*:

```
>>> psi(3, 10**100)
2.0e-300
>>> psi(250, 10**30+10**20*j)
(-1.2931425043636426872048 65e-7010 + 3.2328562609091073915131 08e-7018j)
```

Application to infinite series

Any infinite series where the summand is a rational function of the index *k* can be evaluated in closed form in terms of polygamma functions of the roots and poles of the summand:

```
>>> a = sqrt(2)
>>> b = sqrt(3)
>>> nsum(lambda k: 1/((k+a)**2*(k+b)), [0, inf])
0.40496689275178570619175 31
>>> (psi(0,a)-psi(0,b)-a*psi(1,a)+b*psi(1,a))/(a-b)**2
0.40496689275178570619175 31
```

This follows from the series representation ($m > 0$)

$$\psi^{(m)}(z) = (-1)^{m+1} m! \sum_{k=0}^{\infty} \frac{1}{(z+k)^{m+1}}.$$

Since the roots of a polynomial may be complex, it is sometimes necessary to use the complex polygamma function to evaluate an entirely real-valued sum:

```
>>> nsum(lambda k: 1/(k**2-2*k+3), [0, inf])
1.6943614339070612561546 65
>>> nprint(polyroots([1,-2,3]))
[(1.0 - 1.41421j), (1.0 + 1.41421j)]
>>> r1 = 1-sqrt(2)*j
>>> r2 = r1.conjugate()
>>> (psi(0,-r2)-psi(0,-r1))/(r1-r2)
(1.6943614339070612561546 65 + 0.0j)
```

mrpy.base.stats

A module defining the TGGD distribution (as well as in log and ln space) in standard R style.

See the R package *tggd* for a similar implementation.

The distribution functions implemented here are described in detail in Murray, Robotham and Power, 2016.

Note: *Which distribution should I choose?* The log/ln versions in this module are intended to provide the correct distribution when variates are drawn from a real-space TGGD, but there are priors on their uncertainty which operate in log-space (eg. a log-normal distribution). The likelihood of a given set of parameters is incorrect in such a case if the real-space version is used without an adjustment to the Jacobian.

Short answer: generally use *TGGD*, but if your variates must form a proper PDF in *log-space*, use the appropriate log-space version.

Classes

TGGD([scale, a, b, xmin])	The Truncated Generalised Gamma Distribution.
TGGDln([scale, a, b, xmin])	The Truncated Generalised Gamma Distribution in ln space.
TGGDlog([scale, a, b, xmin])	The Truncated Generalised Gamma Distribution in log10 space.

mrpy.base.stats.TGGD

class mrpy.base.stats.**TGGD**(*scale=1*, *a=-1.0*, *b=1.0*, *xmin=0.1*)

The Truncated Generalised Gamma Distribution.

The TGGD has the following PDF:

$$f(x) = \frac{b\left(\frac{x}{s}\right)^{a}\exp\left(-\left(\frac{m}{s}\right)^{b}\right)}{\Gamma\left(\frac{a+1}{b},\left(\frac{x_{\min}}{s}\right)^{b}\right)}$$

where $s > 0$ corresponds to the scale argument of this class, $a \in \Re$, $b > 0$, $x_{\min} > 0$ is the truncation value, and Γ is the incomplete gamma function, provided by *mpmath*.

Parameters

scale [array_like, optional] Transition scale from power-law to exponential cut-off. Analogous to the scale parameter for the standard Gamma distribution.

a [float or array_like, optional] Power-law slope of the TGGD.

b [float or array_like, optional] Exponential cut-off parameter of the TGGD.

xmin [float or array_like, optional] Truncation value of the TGGD.

Examples

The following should create a sample and plot its histogram. The histogram should have a slope of -1.

```
>>> from mrpy.base.stats import TGGD
>>> import matplotlib.pyplot as plt
>>> tggd = TGGD(a=-2)
>>> r = tggd.rvs(100)
>>> plt.hist(np.log10(r))
```

Taking the quantile of the cumulative probability at each variate should return something close to the variate.

```
>>> a = tggd.quantile(tggd.cdf(r))/r  #should be close to 1
>>> np.all(np.isclose(a,1))
True
```

Show that the numerical integral is equal to the CDF. The following uses a *scale* more appropriate to halo mass functions.

```
>>> from scipy.integrate import quad
>>> tggd = TGGD(scale=1e14,a=-1.5,b=0.7,xmin=1e10)
>>> a = quad(tggd.pdf,1e10,1e11)[0]/tggd.cdf(1e11) # should be close to 1
>>> np.isclose(a,1)
True
```

The CDF should approach unity when `x >> scale`

```
>>> a = tggd.cdf(1e18) #Should be close to 1
>>> np.isclose(a,1)
True
```

To show the link to the `log` and `ln` variants, the following should be a sequence from 0 to 1 (by 0.1)

```
>>> from mrpy.base.stats import TGGDlog, TGGDln
>>> tggd = TGGD()
>>> tggd_log = TGGDlog()
>>> tggd_ln = TGGDln()
>>> tggd.cdf(10**tggd_log.quantile(np.arange(0,1,0.1)))
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

```
>>> b = tggd.cdf(np.exp(tggd_ln.quantile(np.arange(0,1,0.1))))
>>> np.all(np.isclose(b,np.arange(0,1,0.1)))
True
```

Methods

__init__([scale, a, b, xmin])	
cdf(q[, lower_tail, log_p])	The cdf of the distribution.
central_moments(n)	Calculate the nth central moment, E[(X-mu)^n].
normalised_central_moments(n)	Calculate the nth standardized central moment, E[(X-mu)^n]/sigma^n.
pdf(x[, log])	The pdf of the distribution.
quantile(p[, lower_tail, log_p, res_approx])	The quantile of the distribution.
raw_moments(n)	Calculate the nth raw moment, E[X^n].
rvs(n[, res_approx])	Generate random variates from the distribution.

mrpy.base.stats.TGGD.__init__

TGGD.**__init__**(*scale=1*, *a=-1.0*, *b=1.0*, *xmin=0.1*)

mrpy.base.stats.TGGD.cdf

TGGD.**cdf**(*q*, *lower_tail=True*, *log_p=False*)

The cdf of the distribution.

Parameters

q [array_like] Variates at which to calculate the cdf. If any of *shape*, *a*, *b* or *xmin* are arrays, *q* must have the same length.

lower_tail [logical, optional] If *True* (default), probabilities are P[X <= q], otherwise, P[X > q].

log_p [logical, optional] If *True*, probabilities *p* are interpreted as log(*p*).

Returns

p [array_like] The integrated probability of a variate being smaller than *q*.

mrpy.base.stats.TGGD.central_moments

TGGD.**central_moments**(*n*)

Calculate the nth central moment, E[(X-mu)^n].

Parameters

n [integer] The order of the moment desired.

Returns

mu_n [float] The nth central moment.

Notes

The 2nd central moment is equivalent to the variance.

Examples

```
>>> from mrpy.base.stats import TGGD
>>> t = TGGD()
>>> variance = t.central_moments(2)
>>> t.raw_moments(10)
199064.8037313875
```

mrpy.base.stats.TGGD.normalised_central_moments

TGGD.**normalised_central_moments**(*n*)

Calculate the nth standardized central moment, E[(X-mu)^n]/sigma^n.

Parameters

n [integer] The order of the moment desired.

Returns

mu_n [float] The nth standardized central moment.

Notes

The 3rd standardized central moment is equivalent to the skewness.

mrpy.base.stats.TGGD.pdf

TGGD.**pdf**(*x*, *log=False*)

The pdf of the distribution.

Parameters

x [array_like] Variates at which to calculate the pdf. If any of *shape*, *a*, *b* or *xmin* are arrays, *x* must have the same length.

log [logical, optional] Whether to return the log of the pdf (if so, uses a better method than taking log of the final result).

Returns

d [float or array_like] Values of the pdf corresponding to the variates *x*.

mrpy.base.stats.TGGD.quantile

TGGD.**quantile**(*p*, *lower_tail=True*, *log_p=False*, *res_approx=0.01*)

The quantile of the distribution.

The quantile at a given probability value *p* is defined as *q*, where

$$p = P(X \leq q).$$

Parameters

p [array_like] Probabilities at which to calculate the quantiles. If any of *shape*, *a*, *b* or *xmin* are arrays, *p* must have the same length.

lower_tail [logical, optional] If *True* (default), probabilities are P[X <= q], otherwise, P[X > q].

log_p [logical, optional] If *True*, probabilities *p* are returned as log(*p*).

res_approx: float, optional Sets the resolution for interpolating the CDF, which is inverted to yield the quantile.

Returns

q [array_like] The quantiles corresponding to *p*.

mrpy.base.stats.TGGD.raw_moments

TGGD.**raw_moments**(*n*)

Calculate the nth raw moment, E[X^n].

Parameters

n [array_like] The order(s) of the moment desired.

Returns

mu_n [array_like] The raw moment(s) corresponding to the order(s) *n*.

Notes

The 1st raw moment is equivalent to the mean.

Examples

```
>>> from mrpy.base.stats import TGGD
>>> t = TGGD()
>>> mean = t.raw_moments(1)
>>> ten_moments = t.raw_moments(np.arange(10))
```

mrpy.base.stats.TGGD.rvs

TGGD.**rvs**(*n*, *res_approx=0.01*)

Generate random variates from the distribution.

Parameters

n [integer or tuple of integers] The size/shape of the returned variates. If an integer, specifies the number of returned variates. If a tuple of integers, the return array will have shape *n*.

res_approx: float, optional Sets the resolution for interpolating the CDF, which is inverted to yield the quantile.

Returns

r [array_like] Random variates from the distribution, with shape *n*.

mrpy.base.stats.TGGDln

class mrpy.base.stats.**TGGDln**(*scale=0.0*, *a=-1.0*, *b=1.0*, *xmin=-2.3025850929940459*)

The Truncated Generalised Gamma Distribution in ln space.

Specifically, if *exp(x)* is drawn from a TGGD distribution (in real space), this function gives the distribution of *x*, using the same parameter values.

The ln TGGD has the following PDF:

$$\frac{b(\exp((x-s)(a+1))\exp(-\exp(b(x-s)))}{s\Gamma(\frac{a+1}{b},\exp(b(x-s)))}$$

where *s* corresponds to the scale argument of this class, and Γ is the incomplete gamma function, provided by *mpmath*.

Parameters

scale [array_like, optional] Transition scale from power-law to exponential cut-off. Analogous to the scale parameter for the standard Gamma distribution.

a [float or array_like, optional] Power-law slope of the TGGD.

b [float or array_like, optional] Exponential cut-off parameter of the TGGD.

xmin [float or array_like, optional] Truncation value of the TGGD.

Examples

The following should create a sample and plot its histogram. The histogram should have a slope of -1.

```
>>> from mrpy.base.stats import TGGDln
>>> import matplotlib.pyplot as plt
>>> tggdln = TGGDln(a=-2)
>>> r = tggdln.rvs(100)
>>> plt.hist(r)
```

Taking the quantile of the cumulative probability at each variate should return something close to the variate.

```
>>> a = tggdln.quantile(tggdln.cdf(r))/r  #should be close to 1
>>> np.all(np.isclose(a,1))
True
```

Show that the numerical integral is equal to the CDF.

```
>>> from scipy.integrate import quad
>>> tggdln = TGGDln(scale=14,a=-1.5,b=0.7,xmin=10)
>>> a = quad(tggdln.pdf,10,11)[0]/tggdln.cdf(11) # should be close to 1
>>> np.isclose(a,1)
True
```

The CDF should approach unity when `x >> scale`

```
>>> a = tggdln.cdf(np.log(1e18)) #Should be close to 1
>>> np.isclose(a,1)
True
```

To show the link to the `log` and `ln` variants, the following should be a sequence from 0 to 1 (by 0.1)

```
>>> from mrpy.base.stats import TGGDlog, TGGDln
>>> tggd = TGGD()
>>> tggd_log = TGGDlog()
>>> tggd_ln = TGGDln()
>>> a = tggd_log.cdf(np.log10(tggd.quantile(np.arange(0,1,0.1))))
>>> np.all(np.isclose(a,np.arange(0,1,0.1)))
True
```

```
>>> a = tggd_log.cdf(tggd_ln.quantile(np.arange(0,1,0.1))/np.log(10))
>>> np.all(np.isclose(a,np.arange(0,1,0.1)))
True
```

Methods

__init__([scale, a, b, xmin])	
cdf(q[, lower_tail, log_p])	The cdf of the distribution.
central_moments(n)	Calculate the nth central moment, E[(X-mu)^n].
normalised_central_moments(n)	Calculate the nth standardized central moment, E[(X-mu)^n]/sigma^n.
pdf(x[, log])	The pdf of the distribution.
quantile(p[, lower_tail, log_p, res_approx])	The quantile of the distribution.

Continued on next page

Table 3.5 – continued from previous page

raw_moments(n)	Calculate the nth raw moment, E[X^n].
rvs(n[, res_approx])	Generate random variates from the distribution.

mrpy.base.stats.TGGDln.__init__

TGGDln.**__init__**(*scale=0.0*, *a=-1.0*, *b=1.0*, *xmin=-2.3025850929940459*)

mrpy.base.stats.TGGDln.cdf

TGGDln.**cdf**(*q*, *lower_tail=True*, *log_p=False*)

The cdf of the distribution.

Parameters

q [array_like] Variates at which to calculate the cdf. If any of *shape*, *a*, *b* or *xmin* are arrays, *q* must have the same length.

lower_tail [logical, optional] If *True* (default), probabilities are P[X <= q], otherwise, P[X > q].

log_p [logical, optional] If *True*, probabilities *p* are interpreted as log(*p*).

Returns

p [array_like] The integrated probability of a variate being smaller than *q*.

mrpy.base.stats.TGGDln.central_moments

TGGDln.**central_moments**(*n*)

Calculate the nth central moment, E[(X-mu)^n].

Parameters

n [integer] The order of the moment desired.

Returns

mu_n [float] The nth central moment.

Notes

The 2nd central moment is equivalent to the variance.

Examples

```
>>> from mrpy.base.stats import TGGD
>>> t = TGGD()
>>> variance = t.central_moments(2)
>>> t.raw_moments(10)
199064.8037313875
```

mrpy.base.stats.TGGDln.normalised_central_moments

TGGDln.**normalised_central_moments**(*n*)

Calculate the nth standardized central moment, E[(X-mu)^n]/sigma^n.

Parameters

n [integer] The order of the moment desired.

Returns

mu_n [float] The nth standardized central moment.

Notes

The 3rd standardized central moment is equivalent to the skewness.

mrpy.base.stats.TGGDln.pdf

TGGDln.**pdf**(*x*, *log=False*)

The pdf of the distribution.

Parameters

x [array_like] Variates at which to calculate the pdf. If any of *shape*, *a*, *b* or *xmin* are arrays, *x* must have the same length.

log [logical, optional] Whether to return the log of the pdf (if so, uses a better method than taking log of the final result).

Returns

d [float or array_like] Values of the pdf corresponding to the variates *x*.

mrpy.base.stats.TGGDln.quantile

TGGDln.**quantile**(*p*, *lower_tail=True*, *log_p=False*, *res_approx=0.01*)

The quantile of the distribution.

The quantile at a given probability value *p* is defined as *q*, where

$$p = P(X \leq q).$$

Parameters

p [array_like] Probabilities at which to calculate the quantiles. If any of *shape*, *a*, *b* or *xmin* are arrays, *p* must have the same length.

lower_tail [logical, optional] If *True* (default), probabilities are P[X <= q], otherwise, P[X > q].

log_p [logical, optional] If *True*, probabilities *p* are returned as log(*p*).

res_approx: float, optional Sets the resolution for interpolating the CDF, which is inverted to yield the quantile.

Returns

q [array_like] The quantiles corresponding to *p*.

mrpy.base.stats.TGGDln.raw_moments

TGGDln.**raw_moments**(*n*)

Calculate the nth raw moment, E[X^n].

Parameters

n [array_like] The order(s) of the moment desired.

Returns

mu_n [array_like] The raw moment(s) corresponding to the order(s) *n*.

Notes

The 1st raw moment is equivalent to the mean.

Examples

```
>>> from mrpy.base.stats import TGGD
>>> t = TGGD()
>>> mean = t.raw_moments(1)
>>> ten_moments = t.raw_moments(np.arange(10))
```

mrpy.base.stats.TGGDln.rvs

TGGDln.**rvs**(*n*, *res_approx=0.01*)

Generate random variates from the distribution.

Parameters

n [integer or tuple of integers] The size/shape of the returned variates. If an integer, specifies the number of returned variates. If a tuple of integers, the return array will have shape *n*.

res_approx: float, optional Sets the resolution for interpolating the CDF, which is inverted to yield the quantile.

Returns

r [array_like] Random variates from the distribution, with shape *n*.

mrpy.base.stats.TGGDlog

class mrpy.base.stats.**TGGDlog**(*scale=0.0*, *a=-1.0*, *b=1.0*, *xmin=-1.0*)

The Truncated Generalised Gamma Distribution in log10 space.

Specifically, if *10**x* is drawn from a TGGD distribution (in real space), this function gives the distribution of *x*, using the same parameter values.

The log10 TGGD has the following PDF:

$$\frac{\ln(10)b(10^{(x-s)(a+1)}\exp(-10^{b(x-s)})}{s\Gamma(\frac{a+1}{b},10^{b(m-s)})}$$

where *s* corresponds to the scale argument of this class, and Γ is the incomplete gamma function, provided by *mpmath*.

Parameters

scale [array_like, optional] Transition scale from power-law to exponential cut-off. Analogous to the scale parameter for the standard Gamma distribution.

a [float or array_like, optional] Power-law slope of the TGGD.

b [float or array_like, optional] Exponential cut-off parameter of the TGGD.

xmin [float or array_like, optional] Truncation value of the TGGD.

Examples

The following should create a sample and plot its histogram. The histogram should have a slope of -1.

```
>>> from mrpy.base.stats import TGGDlog
>>> import matplotlib.pyplot as plt
>>> tggdlog = TGGDlog(a=-2)
>>> r = tggdlog.rvs(100)
>>> plt.hist(r)
```

Taking the quantile of the cumulative probability at each variate should return something close to the variate.

```
>>> a = tggdlog.quantile(tggdlog.cdf(r))/r  #should be close to 1
>>> np.all(np.isclose(a,1))
True
```

Show that the numerical integral is equal to the CDF. We use a *scale* appropriate for an example such as a halo mass function.

```
>>> from scipy.integrate import quad
>>> tggdlog = TGGDlog(scale=14,a=-1.5,b=0.7,xmin=10)
>>> a = quad(tggdlog.pdf,10,11)[0]/tggdlog.cdf(11) # should be close to 1
>>> np.isclose(a,1)
True
```

The CDF should approach unity when `10**x >> 10**scale`

```
>>> a = tggdlog.cdf(18) #Should be close to 1
>>> np.isclose(a,1)
True
```

To show the link to the `log` and `ln` variants, the following should be a sequence from 0 to 1 (by 0.1)

```
>>> from mrpy.base.stats import TGGDlog, TGGDln
>>> tggd = TGGD()
>>> tggd_log = TGGDlog()
>>> tggd_ln = TGGDln()
>>> a = tggd_ln.cdf(np.log(tggd.quantile(np.arange(0,1,0.1))))
>>> np.all(np.isclose(a,np.arange(0,1,0.1)))
True
```

```
>>> b = tggd_ln.cdf(tggd_log.quantile(np.arange(0,1,0.1))*np.log(10))
>>> np.all(np.isclose(b,np.arange(0,1,0.1)))
True
```

Methods

__init__([scale, a, b, xmin])	
cdf(q[, lower_tail, log_p])	The cdf of the distribution.
central_moments(n)	Calculate the nth central moment, E[(X-mu)^n].
normalised_central_moments(n)	Calculate the nth standardized central moment, E[(X-mu)^n]/sigma^n.
pdf(x[, log])	The pdf of the distribution.
quantile(p[, lower_tail, log_p, res_approx])	The quantile of the distribution.
raw_moments(n)	Calculate the nth raw moment, E[X^n].
rvs(n[, res_approx])	Generate random variates from the distribution.

mrpy.base.stats.TGGDlog.__init__

TGGDlog.**__init__**(*scale=0.0*, *a=-1.0*, *b=1.0*, *xmin=-1.0*)

mrpy.base.stats.TGGDlog.cdf

TGGDlog.**cdf**(*q*, *lower_tail=True*, *log_p=False*)

The cdf of the distribution.

Parameters

q [array_like] Variates at which to calculate the cdf. If any of *shape*, *a*, *b* or *xmin* are arrays, *q* must have the same length.

lower_tail [logical, optional] If *True* (default), probabilities are P[X <= q], otherwise, P[X > q].

log_p [logical, optional] If *True*, probabilities *p* are interpreted as log(*p*).

Returns

p [array_like] The integrated probability of a variate being smaller than *q*.

mrpy.base.stats.TGGDlog.central_moments

TGGDlog.**central_moments**(*n*)

Calculate the nth central moment, E[(X-mu)^n].

Parameters

n [integer] The order of the moment desired.

Returns

mu_n [float] The nth central moment.

Notes

The 2nd central moment is equivalent to the variance.

Examples

```
>>> from mrpy.base.stats import TGGD
>>> t = TGGD()
>>> variance = t.central_moments(2)
>>> t.raw_moments(10)
199064.8037313875
```

mrpy.base.stats.TGGDlog.normalised_central_moments

TGGDlog.**normalised_central_moments**(*n*)

Calculate the nth standardized central moment, E[(X-mu)^n]/sigma^n.

Parameters

n [integer] The order of the moment desired.

Returns

mu_n [float] The nth standardized central moment.

Notes

The 3rd standardized central moment is equivalent to the skewness.

mrpy.base.stats.TGGDlog.pdf

TGGDlog.**pdf**(*x*, *log=False*)

The pdf of the distribution.

Parameters

x [array_like] Variates at which to calculate the pdf. If any of *shape*, *a*, *b* or *xmin* are arrays, *x* must have the same length.

log [logical, optional] Whether to return the log of the pdf (if so, uses a better method than taking log of the final result).

Returns

d [float or array_like] Values of the pdf corresponding to the variates *x*.

mrpy.base.stats.TGGDlog.quantile

TGGDlog.**quantile**(*p*, *lower_tail=True*, *log_p=False*, *res_approx=0.01*)

The quantile of the distribution.

The quantile at a given probability value p is defined as q, where

$$p = P(X \leq q).$$

Parameters

p [array_like] Probabilities at which to calculate the quantiles. If any of *shape*, *a*, *b* or *xmin* are arrays, *p* must have the same length.

lower_tail [logical, optional] If *True* (default), probabilities are P[X <= q], otherwise, P[X > q].

log_p [logical, optional] If *True*, probabilities *p* are returned as log(*p*).

res_approx: float, optional Sets the resolution for interpolating the CDF, which is inverted to yield the quantile.

Returns

q [array_like] The quantiles corresponding to *p*.

mrpy.base.stats.TGGDlog.raw_moments

TGGDlog.**raw_moments**(*n*)

Calculate the nth raw moment, E[X^n].

Parameters

n [array_like] The order(s) of the moment desired.

Returns

mu_n [array_like] The raw moment(s) corresponding to the order(s) *n*.

Notes

The 1st raw moment is equivalent to the mean.

Examples

```
>>> from mrpy.base.stats import TGGD
>>> t = TGGD()
>>> mean = t.raw_moments(1)
>>> ten_moments = t.raw_moments(np.arange(10))
```

mrpy.base.stats.TGGDlog.rvs

TGGDlog.**rvs**(*n*, *res_approx=0.01*)

Generate random variates from the distribution.

Parameters

n [integer or tuple of integers] The size/shape of the returned variates. If an integer, specifies the number of returned variates. If a tuple of integers, the return array will have shape *n*.

res_approx: float, optional Sets the resolution for interpolating the CDF, which is inverted to yield the quantile.

Returns

r [array_like] Random variates from the distribution, with shape *n*.

mrpy.base.core

Basic MRP functionality, such as functions to generate the MRP with different normalisations.

Note that in this package, the *MRP* will refer to the truncated generalised gamma distribution (TGGD), with added arbitrary normalisation. We will always directly apply it to halo mass functions, (HMFs), and so the variate will generally be mass, *m*, and the relevant default scales will be large.

This does not in principle restrict the usage of the MRP for other applications, such as luminosity functions or other data.

Functions

A_rhom(logHs, alpha, beta[, rhom])	The normalisation required to bind all matter in halos of some scale:
dndm(m, logHs, alpha, beta[, mmin, norm, log])	The MRP distribution.
entire_integral(logHs, alpha, beta)	The entire integral of the un-normalised mass-weighted *non-truncated* MRP:
log_mass_mode(logHs, alpha, beta)	The mode of the log-space MRP weighted by mass.
ngtm(m, logHs, alpha, beta[, mmin, norm, log])	The integral of the MRP, in reverse (i.e.
rho_gtm(m, logHs, alpha, beta[, mmin, norm, log])	The mass-weighted integral of the MRP, in reverse (ie.

mrpy.base.core.A_rhom

mrpy.base.core.**A_rhom**(*logHs*, *alpha*, *beta*, *rhom=83265000000.0*)

The normalisation required to bind all matter in halos of some scale:

$$A_{\rho_c} = \Omega_m \rho_c / k(\vec{\theta})$$

where *k* is the *entire_integral()* of the MRP, with a mass-weighting (or scaling) of s=1.

Parameters

logHs [array_like] The base-10 logarithm of the scale mass, H_s.

alpha [array_like] The power-law index

beta [array_like] Exponential cutoff parameter

rhom [float, optional] The mass density of the Universe.

mrpy.base.core.dndm

mrpy.base.core.**dndm**(*m*, *logHs*, *alpha*, *beta*, *mmin=None*, *norm='pdf'*, *log=False*, ***Arhoc_kw*)

The MRP distribution.

Parameters

m [array_like] Vector of masses at which to evaluate the MRP

logHs [float] The base-10 logarithm of the scale mass, $H_\star$.

alpha [float] The power-law index

beta [float] Exponential cutoff parameter

mmin [float, optional] The lower-truncation mass. Default is the minimum mass in m.

norm :string or float, optional Gives the normalisation of the MRP, A. If set to a *float*, it is directly the normalisation. If set to `"pdf"`, it will automatically render the MRP as a statistical distribution. If set to `"rhoc"`, it will yield the correct total mass density across all masses, down to `m=0`.

log [logical] Whether to return the natural log of the MRP (suitable for Bayesian likelihoods).

****Arhom_kw :** Arguments directly forwarded to the mean-density normalisation, *A_rhom()*.

mrpy.base.core.entire_integral

`mrpy.base.core.`**`entire_integral`**(*logHs*, *alpha*, *beta*)

The entire integral of the un-normalised mass-weighted *non-truncated* MRP:

$$\int_0^\infty m^s f(m) = \mathcal{H}_\star^{s+1}\Gamma\left(\frac{\alpha+1+s}{\beta}\right)\, dm.$$

where s defines a weighting of the integral, in which the immediate application is that `s=1` gives the total mass density.

Note: The sum of *alpha* and s must be greater than -1.

Parameters

logHs [array_like] The base-10 logarithm of the scale mass, $H_\star$.

alpha [array_like] The power-law index

beta [array_like] Exponential cutoff parameter

mrpy.base.core.log_mass_mode

`mrpy.base.core.`**`log_mass_mode`**(*logHs*, *alpha*, *beta*)

The mode of the log-space MRP weighted by mass.

Parameters

logHs, alpha, beta: array_like Shape parameters of the MRP distribution.

Returns

lmm [array_like] The log-space mass mode of the MRP.

Examples

This function:

```
>>> log_mass_mode(14.0,-1.8,0.7)
1.67016714698e+13
```

yields the same result as generating the mode via differentiation:

```
>>> from mrpy.base import stats
>>> from scipy.interpolate import InterpolatedUnivariateSpline as spline
>>> m = np.linspace(13.0,14.0,200)
>>> # Add 1 to alpha to generate mass weighting
>>> mrp = stats.TGGDlog(14.0,-1.8+1,0.7,m[0]).pdf(m,log=True)
>>> s = spline(m,mrp,k=4)
>>> 10**s.derivative().roots()[0]
1.67016715e+13
```

mrpy.base.core.ngtm

mrpy.base.core.**ngtm**(*m*, *logHs*, *alpha*, *beta*, *mmin=None*, *norm='pdf'*, *log=False*, ***Arhom_kw*)

The integral of the MRP, in reverse (i.e. CDF=1 at mmin).

Parameters

m [array_like] Vector of masses at which to evaluate the MRP

logHs [float] The base-10 logarithm of the scale mass, $H_\star$.

alpha [float] The power-law index

beta [float] Exponential cutoff parameter

mmin [float, optional] The lower-truncation mass. Default is the minimum mass in `m`.

norm :string or float, optional Gives the normalisation of the MRP, *A*. If set to a *float*, it is directly the normalisation. If set to `"pdf"`, it will automatically render the MRP as a statistical distribution. If set to `"rhoc"`, it will yield the correct total mass density across all masses, down to `m=0`.

log [logical] Whether to return the natural log of the MRP (suitable for Bayesian likelihoods).

****Arhom_kw :** Arguments directly forwarded to the mean-density normalisation, *A_rhom()*.

mrpy.base.core.rho_gtm

mrpy.base.core.**rho_gtm**(*m*, *logHs*, *alpha*, *beta*, *mmin=None*, *norm='pdf'*, *log=False*, ***Arhom_kw*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass)

Parameters

m [array_like] Vector of masses at which to evaluate the MRP

logHs [float] The base-10 logarithm of the scale mass, $H_\star$.

alpha [float] The power-law index

beta [float] Exponential cutoff parameter

mmin [float, optional] The lower-truncation mass. Default is the minimum mass in `m`.

norm :string or float, optional Gives the normalisation of the MRP, *A*. If set to a *float*, it is directly the normalisation. If set to `"pdf"`, it will automatically render the MRP as a statistical distribution. If set to `"rhoc"`, it will yield the correct total mass density across all masses, down to `m=0`.

log [logical] Whether to return the natural log of the MRP (suitable for Bayesian likelihoods).

****Arhom_kw :** Arguments directly forwarded to the mean-density normalisation, *A_rhom()*.

Classes

MRP(logm, logHs, alpha, beta[, norm, ...])	An MRP object.

mrpy.base.core.MRP

class mrpy.base.core.**MRP**(*logm*, *logHs*, *alpha*, *beta*, *norm='pdf'*, *log_mmin=None*, *rhom=83265000000.0*)

An MRP object.

This class contains methods for calculating typical quantities of interest: the differential/cumulative number densities, as well as mass densities, and several types of normalisation. Also included is a pointer to underlying statistical quantities, such as mean, median, mode etc.

Parameters

logm [array_like] Vector of log10 masses.

logHs, alpha, beta [array_like] The shape parameters of the MRP.

norm [float or string] Gives the normalisation of the MRP, *lnA*. If set to a *float*, it is directly the (log) normalisation. If set to `"pdf"`, it will automatically render the MRP as a statistical distribution. If set to `"rhom"`, it will yield the correct total mass density across all masses, down to `M=0`.

log_mmin [array_like, optional] Log-10 truncation mass of the MRP. By default is set to the minimum mass in `logm`.

rhom [float, optional] Mass density of the Universe. Only required if *norm* is set to `Arhom`.

Methods

__init__(logm, logHs, alpha, beta[, norm, ...])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.

mrpy.base.core.MRP.__init__

MRP.**__init__**(*logm*, *logHs*, *alpha*, *beta*, *norm='pdf'*, *log_mmin=None*, *rhom=83265000000.0*)

mrpy.base.core.MRP.dndlog10m

MRP.**dndlog10m**(*log=False*)

Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.base.core.MRP.dndm

MRP.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.base.core.MRP.ngtm

MRP.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.base.core.MRP.rho_gtm

MRP.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.physical_dependence

Module containing functions for the dependence of MRP parameters on physical parameters, defined with respect to Behroozi+13.

The values output by the functions in this module were derived as part of the study reported in Murray, Robotham and Power (2016). See that paper for details on the fits used.

In brief, three physical parameters were varied – matter density, rms fluctuation σ_8 and redshift – around the best-fit values from Planck+13, and up to a redshift of 8. For each, the Behroozi+13 mass function was calculated using the *hmf* python package over a range of masses corresponding to (lm-3, lm+2), where *lm* is the log mass mode. The MRP was fit to each curve, and then the best-fit parameters were parameterised as a function of the physical parameters using the Eureqa software.

Note: Given the way these parameterisations were determined, it is not advised to trust these values far from the log-mass-mode of each fit.

Functions

mrp_b13(m[, z, Om0, sig8, Hs_kw, alpha_kw, ...])	Return the MRP defined at m for the given physical parameters.
mrp_params_b13([z, Om0, sig8, Hs_kw, ...])	Return all 4 MRP parameters as a function of physical parameters.

mrpy.extra.physical_dependence.mrp_b13

mrpy.extra.physical_dependence.**mrp_b13**(*m*, *z=0*, *Om0=0.315*, *sig8=0.829*, *Hs_kw={}*, *alpha_kw={}*, *beta_kw={}*, *logA_kw={}*, *mmin=None*, *norm=None*, *log=False*, ***Arhoc_kw*)

Return the MRP defined at `m` for the given physical parameters.

Note: Calls `mrpy.core.mrp()` in the background, and takes all of those parameters.

mrpy.extra.physical_dependence.mrp_params_b13

mrpy.extra.physical_dependence.**mrp_params_b13**(*z=0*, *Om0=0.315*, *sig8=0.829*, *Hs_kw={}*, *alpha_kw={}*, *beta_kw={}*, *logA_kw={}*)

Return all 4 MRP parameters as a function of physical parameters.

mrpy.extra.likelihoods

Provides classes which extend the basic `mrpy.core.MRP` class.

Adds methods for calculating the likelihood and its derivatives in special cases of interest. Specifically, the two main cases of interest are fitting the MRP to a sample of data (`SampleLike`) or fitting to a binned (or theoretical) curve.

At this point, the classes here only support the simplest possible cases, in which the effective volume is constant as a function of mass, down to some threshold truncation mass. Furthermore, only data without measurement error is supported at this point.

At this time, we don't directly support fitting MRP extensions, such as a double-MRP.

Functions

expected_likelihood(theta, data_m, data_mf)	

mrpy.extra.likelihoods.expected_likelihood

mrpy.extra.likelihoods.**expected_likelihood**(*theta*, *data_m*, *data_mf*, *kappa=None*, *V0=1*, *mmin=None*)

Classes

CurveLike(logm, data_dndm, logHs, alpha, ...)	A subclass of `mrpy.core.MRP_PO_Likelihood` which adds the likelihood (and derivatives) of a model given data in the form of a curve.
SampleLike(logm, logHs, alpha, beta, lnA[, ...])	A subclass of `mrpy.core.MRP` which adds the likelihood (and derivatives) of a model given a sample of masses.
SampleLikeWeights(weights, *args, **kwargs)	Compactified version of `MRP_PO_Likelihood` useful for simulated haloes.

mrpy.extra.likelihoods.CurveLike

class mrpy.extra.likelihoods.**CurveLike**(*logm*, *data_dndm*, *logHs*, *alpha*, *beta*, *lnA*, *sig_data=1*, *sig_rhomean=inf*, *rhom=83265000000.0*)

A subclass of `mrpy.core.MRP_PO_Likelihood` which adds the likelihood (and derivatives) of a model given data in the form of a curve.

See Murray, Robotham, Power (2017), Appendix C.1 for a description.

Parameters

logm [array_like] Vector of log10 masses.

data_dndm [array_like] Array of the same length as *logm*, giving the value of the differential mass function.

hs, alpha, beta, lnA [float] The parameters of the MRP.

sig_data [array_like, optional] The uncertainty of the data (standard deviation). This is used in the likelihood to weight different mass scales. If scalar, all mass scales are weighted evenly.

sig_rhomean,: float, optional This controls how much influence the total mean density of the universe has on the likelihood. The default value of *inf* means it is completely ignored. If it is 0, it becomes an absolute constraint, so that the total mass density of the universe is perfectly matched (setting the normalisation). In between, it acts as an uncertainty on this value.

rhom [float, optional] Mass density of the Universe. Only used if 'sig_rhomean' not infinite.

Methods

__init__(logm, data_dndm, logHs, alpha, ...)	
dndlog10m([log])	Return the MRP in log10 space at 'm'.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.

mrpy.extra.likelihoods.CurveLike.__init__

CurveLike.**__init__**(*logm*, *data_dndm*, *logHs*, *alpha*, *beta*, *lnA*, *sig_data=1*, *sig_rhomean=inf*, *rhom=83265000000.0*)

mrpy.extra.likelihoods.CurveLike.dndlog10m

CurveLike.**dndlog10m**(*log=False*)

Return the MRP in log10 space at 'm'.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.likelihoods.CurveLike.dndm

CurveLike.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.likelihoods.CurveLike.ngtm

CurveLike.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.likelihoods.CurveLike.rho_gtm

CurveLike.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.likelihoods.SampleLike

class mrpy.extra.likelihoods.**SampleLike**(*logm*, *logHs*, *alpha*, *beta*, *lnA*, *log_mmin=None*, *rhom=83265000000.0*)
A subclass of `mrpy.core.MRP` which adds the likelihood (and derivatives) of a model given a sample of masses.

The likelihoods in this class are true under a number of simplifications. Firstly, the effective volume is constant down to a threshold minimum mass. Secondly, the masses have no measurement uncertainty. While these simplifications are rather restrictive, nevertheless the quantities here are suprisingly useful, for instance with estimating parameters for simulations, or as an ingredient in a more sophisticated analysis.

Parameters

logm [array_like] Vector of log10 masses.

logHs, alpha, beta, lnA [array_like] The parameters of the MRP.

log_mmin [array_like, optional] Log-10 truncation mass of the MRP. By default is set to the minimum mass in `logm`.

rhom [float] Mass density of the universe. Only used if the normalisation is set to `Arhom`.

Methods

__init__(logm, logHs, alpha, beta, lnA[, ...])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.

Continued on next page

Table 3.14 – continued from previous page

dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.

mrpy.extra.likelihoods.SampleLike.__init__

SampleLike.**__init__**(*logm*, *logHs*, *alpha*, *beta*, *lnA*, *log_mmin=None*, *rhom=83265000000.0*)

mrpy.extra.likelihoods.SampleLike.dndlog10m

SampleLike.**dndlog10m**(*log=False*)

Return the MRP in log10 space at 'm'.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.likelihoods.SampleLike.dndm

SampleLike.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.likelihoods.SampleLike.ngtm

SampleLike.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.likelihoods.SampleLike.rho_gtm

SampleLike.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.likelihoods.SampleLikeWeights

class mrpy.extra.likelihoods.**SampleLikeWeights**(*weights*, **args*, ***kwargs*)

Compactified version of MRP_PO_Likelihood useful for simulated haloes.

Effectively, this is the same as MRP_PO_Likelihood, but instead of passing a full array of halo masses, one can pass an array of unique masses, and a *weights* array which specifies the number of each mass in the sample.

This should be useful for simulation-like data, which has many halos of the same mass.

Parameters

weights [array_like] Array of the same length as `m`, giving the number of each mass in the distribution.

Other Parameters

args, kwargs : Other parameters are necessary, to be passed through to `MRP_PO_Likelihood`.

Methods

__init__(weights, *args, **kwargs)	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.

mrpy.extra.likelihoods.SampleLikeWeights.__init__

`SampleLikeWeights.`**`__init__`**(*weights*, **args*, ***kwargs*)

mrpy.extra.likelihoods.SampleLikeWeights.dndlog10m

`SampleLikeWeights.`**`dndlog10m`**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.likelihoods.SampleLikeWeights.dndm

`SampleLikeWeights.`**`dndm`**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.likelihoods.SampleLikeWeights.ngtm

`SampleLikeWeights.`**`ngtm`**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.likelihoods.SampleLikeWeights.rho_gtm

SampleLikeWeights.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.analytic_model

A module defining the likelihoods and related quantities involved when the data is purely ideal and analytic.

See appendix of Murray, Power and Robotham for more details.

It is intended to provide a fast way to estimate the strength of fits using different parameters (such as the scaling).

Classes

IdealAnalytic([logHsd, alphad, betad, lnAd])	

mrpy.extra.analytic_model.IdealAnalytic

class mrpy.extra.analytic_model.**IdealAnalytic**(*logHsd=None*, *alphad=None*, *betad=None*, *lnAd=None*, ***kwargs*)

Methods

__init__([logHsd, alphad, betad, lnAd])	Subclass of *SampleLike*, defining the expected likelihood and covariance for a sample of variates drawn directly from an MRP distribution.
dndlog10m([log])	Return the MRP in log10 space at 'm'.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.

mrpy.extra.analytic_model.IdealAnalytic.__init__

IdealAnalytic.**__init__**(*logHsd=None*, *alphad=None*, *betad=None*, *lnAd=None*, ***kwargs*)

Subclass of *SampleLike*, defining the expected likelihood and covariance for a sample of variates drawn directly from an MRP distribution.

See Murray, Robotham, Power (2016) for details.

In this class, no vector of masses is passed, but rather mmin is used everywhere. To do this, it automatically creates a length-1 vector of masses, equalling mmin, to use the methods of its superclass. To invoke methods for general vectors of m, do not use this class.

Parameters

logHsd, alphad, betad [float, optional] Values of the MRP parameters for the *data*. If not given, they are set to the corresponding values of the variable parameters (i.e. the solution).

Other Parameters

args : Other parameters are passed directly to `PerObjLike`

mrpy.extra.analytic_model.IdealAnalytic.dndlog10m

`IdealAnalytic.`**`dndlog10m`**(*log=False*)

Return the MRP in log10 space at 'm'.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.analytic_model.IdealAnalytic.dndm

`IdealAnalytic.`**`dndm`**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.analytic_model.IdealAnalytic.ngtm

`IdealAnalytic.`**`ngtm`**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.analytic_model.IdealAnalytic.rho_gtm

`IdealAnalytic.`**`rho_gtm`**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise

Provides classes which implement variations of the MRP, in which the parameters have been transformed.

Each transformation has three available classes:

- one with a suffix *MRP*, which implements the core MRP quantities (i.e. is a subclass of `mrpy.core.MRP`)
- one with a suffix *PerObj*, which extends the previous one for likelihoods based on samples of variates (i.e. is a subclass of `mrpy.likelihoods.PerObjLike`)
- one with a suffix *Curve*, which extends the base one for likelihoods based on chi-squared minimzation against binned data (i.e. a subclass of `mrpy.likelihoods.CurveLike`)

In all, the transformed parameters are denoted `p1,p2,p3`.

In addition, base classes for each are provided, which makes it easy to implement arbitrary transformations. See the docs for *ReparameteriseMRP* for more details.

Classes

AP1Curve([p1, p2, p3, logHs, alpha, beta])	
AP1Sample([p1, p2, p3, logHs, alpha, beta])	
Ap1MRP([p1, p2, p3, logHs, alpha, beta])	A fairly standard parameterisation of the TGGD (eg.
GG2Curve([p1, p2, p3, logHs, alpha, beta])	
GG2MRP([p1, p2, p3, logHs, alpha, beta])	A reparameterisation of the standard MRP form of the TGGD (eg.
GG2Sample([p1, p2, p3, logHs, alpha, beta])	
GG3Curve([p1, p2, p3, logHs, alpha, beta])	
GG3MRP([p1, p2, p3, logHs, alpha, beta])	A reparameterisation of the standard MRP form of the TGGD (eg.
GG3Sample([p1, p2, p3, logHs, alpha, beta])	
HTCurve([p1, p2, p3, logHs, alpha, beta])	
HTMRP([p1, p2, p3, logHs, alpha, beta])	A reparameterisation of the standard MRP form of the TGGD.
HTSample([p1, p2, p3, logHs, alpha, beta])	
ReparameteriseCurveLike([p1, p2, p3, logHs, ...])	An extension of *ReparameteriseMRP* which adds necessary methods for calculating jacobians and hessians for chi-square likelihoods.
ReparameteriseMRP([p1, p2, p3, logHs, ...])	Base class for reparameterising the MRP.
ReparameteriseSampleLike([p1, p2, p3, ...])	An extension of *ReparameteriseMRP* which adds necessary methods for calculating jacobians and hessians for per-object likelihoods.

mrpy.extra.reparameterise.AP1Curve

class mrpy.extra.reparameterise.**AP1Curve**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at 'm'.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.AP1Curve.__init__

AP1Curve.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.AP1Curve.dndlog10m

AP1Curve.**dndlog10m**(*log=False*)

Return the MRP in log10 space at 'm'.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.AP1Curve.dndm

AP1Curve.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.AP1Curve.ngtm

AP1Curve.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.AP1Curve.p_T

AP1Curve.**p_T**(***kwargs*)

New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.AP1Curve.rho_gtm

AP1Curve.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.AP1Curve.theta_T

AP1Curve.**theta_T**(*p1=None*, *p2=None*, *p3=None*)

Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.AP1Sample

class mrpy.extra.reparameterise.**AP1Sample**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.AP1Sample.__init__

AP1Sample.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.AP1Sample.dndlog10m

AP1Sample.**dndlog10m**(*log=False*)

Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.AP1Sample.dndm

AP1Sample.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.AP1Sample.ngtm

AP1Sample.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.AP1Sample.p_T

AP1Sample.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.AP1Sample.rho_gtm

AP1Sample.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.AP1Sample.theta_T

AP1Sample.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.Ap1MRP

class mrpy.extra.reparameterise.**Ap1MRP**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)
A fairly standard parameterisation of the TGGD (eg. Lagos et al. Eq. 2.1)

Notes

This parameterisation is determined by setting

$$(\log \mathcal{H}_\star, lpha, eta) = (p_1, p_2 - 1, p_3).$$

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.Ap1MRP.__init__

Ap1MRP.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.Ap1MRP.dndlog10m

Ap1MRP.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.Ap1MRP.dndm

Ap1MRP.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.Ap1MRP.ngtm

Ap1MRP.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.Ap1MRP.p_T

Ap1MRP.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.Ap1MRP.rho_gtm

Ap1MRP.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.Ap1MRP.theta_T

Ap1MRP.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG2Curve

class mrpy.extra.reparameterise.**GG2Curve**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG2Curve.__init__

GG2Curve.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.GG2Curve.dndlog10m

GG2Curve.**dndlog10m**(*log=False*)

Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG2Curve.dndm

GG2Curve.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG2Curve.ngtm

GG2Curve.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.GG2Curve.p_T

GG2Curve.**p_T**(***kwargs*)

New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.GG2Curve.rho_gtm

GG2Curve.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.GG2Curve.theta_T

GG2Curve.**theta_T**(*p1=None*, *p2=None*, *p3=None*)

Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG2MRP

class mrpy.extra.reparameterise.**GG2MRP**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

A reparameterisation of the standard MRP form of the TGGD (eg. Lagos et al. Eq. 2.3)

Notes

This has the form

$$(\log \mathcal{H}_\star, lpha, eta) = (-p_1/p_3, p_2 - 1, p_3).$$

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG2MRP.__init__

GG2MRP.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.GG2MRP.dndlog10m

GG2MRP.**dndlog10m**(*log=False*)

Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG2MRP.dndm

GG2MRP.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG2MRP.ngtm

GG2MRP.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.GG2MRP.p_T

GG2MRP.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.GG2MRP.rho_gtm

GG2MRP.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.GG2MRP.theta_T

GG2MRP.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG2Sample

class mrpy.extra.reparameterise.**GG2Sample**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.

Continued on next page

Table 3.24 – continued from previous page

dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG2Sample.__init__

GG2Sample.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.GG2Sample.dndlog10m

GG2Sample.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG2Sample.dndm

GG2Sample.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG2Sample.ngtm

GG2Sample.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.GG2Sample.p_T

GG2Sample.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.GG2Sample.rho_gtm

GG2Sample.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.GG2Sample.theta_T

GG2Sample.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG3Curve

class mrpy.extra.reparameterise.**GG3Curve**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG3Curve.__init__

GG3Curve.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.GG3Curve.dndlog10m

GG3Curve.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG3Curve.dndm

GG3Curve.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG3Curve.ngtm

GG3Curve.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.GG3Curve.p_T

GG3Curve.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.GG3Curve.rho_gtm

GG3Curve.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.GG3Curve.theta_T

GG3Curve.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG3MRP

class mrpy.extra.reparameterise.**GG3MRP**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)
A reparameterisation of the standard MRP form of the TGGD (eg. Lagos et al. Eq. 2.4)

Notes

This has the form

$$(\log \mathcal{H}_\star, lpha, eta) = (p_1, p_2 p_3 - 1, p_3).$$

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.

Continued on next page

Table 3.26 – continued from previous page

rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG3MRP.__init__

GG3MRP.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.GG3MRP.dndlog10m

GG3MRP.**dndlog10m**(*log=False*)

Return the MRP in log10 space at 'm'.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG3MRP.dndm

GG3MRP.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG3MRP.ngtm

GG3MRP.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.GG3MRP.p_T

GG3MRP.**p_T**(***kwargs*)

New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.GG3MRP.rho_gtm

GG3MRP.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.GG3MRP.theta_T

GG3MRP.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG3Sample

class mrpy.extra.reparameterise.**GG3Sample**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.GG3Sample.__init__

GG3Sample.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.GG3Sample.dndlog10m

GG3Sample.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG3Sample.dndm

GG3Sample.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.GG3Sample.ngtm

GG3Sample.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.GG3Sample.p_T

GG3Sample.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.GG3Sample.rho_gtm

GG3Sample.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.GG3Sample.theta_T

GG3Sample.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.HTCurve

class mrpy.extra.reparameterise.**HTCurve**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.HTCurve.__init__

HTCurve.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.HTCurve.dndlog10m

HTCurve.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.HTCurve.dndm

HTCurve.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.HTCurve.ngtm

HTCurve.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.HTCurve.p_T

HTCurve.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.HTCurve.rho_gtm

HTCurve.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.HTCurve.theta_T

HTCurve.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.HTMRP

class mrpy.extra.reparameterise.**HTMRP**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)
A reparameterisation of the standard MRP form of the TGGD.

rac{2+p_2}{p_3}),p_2,p_3),

where the transformation of $\log \mathcal{H}_\star$ is to the logarithmic mass mode.

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.HTMRP.__init__

HTMRP.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.HTMRP.dndlog10m

HTMRP.**dndlog10m**(*log=False*)

Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.HTMRP.dndm

HTMRP.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.HTMRP.ngtm

HTMRP.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.HTMRP.p_T

HTMRP.**p_T**(***kwargs*)

New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.HTMRP.rho_gtm

HTMRP.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.HTMRP.theta_T

HTMRP.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.HTSample

class mrpy.extra.reparameterise.**HTSample**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	New parameters as functions of the standard MRP parameters.
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T([p1, p2, p3])	Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.HTSample.__init__

HTSample.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.HTSample.dndlog10m

HTSample.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.HTSample.dndm

HTSample.**dndm**(*log=False*)
Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.HTSample.ngtm

HTSample.**ngtm**(*log=False*)
The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.HTSample.p_T

HTSample.**p_T**(***kwargs*)
New parameters as functions of the standard MRP parameters.

mrpy.extra.reparameterise.HTSample.rho_gtm

HTSample.**rho_gtm**(*log=False*)
The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.HTSample.theta_T

HTSample.**theta_T**(*p1=None*, *p2=None*, *p3=None*)
Standard MRP parameters as functions of the new parameters.

mrpy.extra.reparameterise.ReparameteriseCurveLike

class mrpy.extra.reparameterise.**ReparameteriseCurveLike**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)
An extension of *ReparameteriseMRP* which adds necessary methods for calculating jacobians and hessians for chi-square likelihoods.

See *ReparameteriseMRP* for arguments.

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.

Continued on next page

Table 3.31 – continued from previous page

p_T(**kwargs)	The new parameters as functions of theta
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T(**kwargs)	theta as functions of the new parameters p1,p2,p3

mrpy.extra.reparameterise.ReparameteriseCurveLike.__init__

ReparameteriseCurveLike.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.ReparameteriseCurveLike.dndlog10m

ReparameteriseCurveLike.**dndlog10m**(*log=False*)

Return the MRP in log10 space at 'm'.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.ReparameteriseCurveLike.dndm

ReparameteriseCurveLike.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.ReparameteriseCurveLike.ngtm

ReparameteriseCurveLike.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.ReparameteriseCurveLike.p_T

ReparameteriseCurveLike.**p_T**(***kwargs*)

The new parameters as functions of theta

mrpy.extra.reparameterise.ReparameteriseCurveLike.rho_gtm

ReparameteriseCurveLike.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.ReparameteriseCurveLike.theta_T

ReparameteriseCurveLike.**theta_T**(***kwargs*)
theta as functions of the new parameters p1,p2,p3

mrpy.extra.reparameterise.ReparameteriseMRP

class mrpy.extra.reparameterise.**ReparameteriseMRP**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

Base class for reparameterising the MRP.

Reparameterisations take the form of a transformation of parameters. Two transformations are required, one from the new parameters, `p1, p2, p3` to the standard ones, `logHs, alpha, beta`, and also the inverse of this transform.

Actual reparameterisations should be based on this class and provide explicit functions for these transformations in the methods *p_T()* and *theta_T()*. This class provides the identity transforms.

The class may be initialised either with the new parameters or standard ones, which is useful for different applications.

Parameters

p1, p2, p3 [array_like, optional] Values of the transformed parameters. Either all of these *or* all of the standard params must be provided. If both are provided, these are used.

logHs, alpha, beta [array_like, optional] Values of the un-transformed parameters. Either all of these *or* all of the transformed params must be provided. If both are provided, the transformed params are used.

kwargs All of the other parameters are passed directly to the super-class.

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	The new parameters as functions of theta
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T(**kwargs)	theta as functions of the new parameters p1,p2,p3

mrpy.extra.reparameterise.ReparameteriseMRP.__init__

ReparameteriseMRP.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.ReparameteriseMRP.dndlog10m

ReparameteriseMRP.**dndlog10m**(*log=False*)
Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.ReparameteriseMRP.dndm

`ReparameteriseMRP.`**`dndm`**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.ReparameteriseMRP.ngtm

`ReparameteriseMRP.`**`ngtm`**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.ReparameteriseMRP.p_T

`ReparameteriseMRP.`**`p_T`**(***kwargs*)

The new parameters as functions of theta

mrpy.extra.reparameterise.ReparameteriseMRP.rho_gtm

`ReparameteriseMRP.`**`rho_gtm`**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.ReparameteriseMRP.theta_T

`ReparameteriseMRP.`**`theta_T`**(***kwargs*)

theta as functions of the new parameters p1,p2,p3

mrpy.extra.reparameterise.ReparameteriseSampleLike

`class` `mrpy.extra.reparameterise.`**`ReparameteriseSampleLike`**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

An extension of *ReparameteriseMRP* which adds necessary methods for calculating jacobians and hessians for per-object likelihoods.

See *ReparameteriseMRP* for arguments.

Methods

__init__([p1, p2, p3, logHs, alpha, beta])	
dndlog10m([log])	Return the MRP in log10 space at ‘m’.
dndm([log])	Return the MRP at *m*.
ngtm([log])	The number density greater than *mmin*.
p_T(**kwargs)	The new parameters as functions of theta
rho_gtm([log])	The mass-weighted integral of the MRP, in reverse (ie.
theta_T(**kwargs)	theta as functions of the new parameters p1,p2,p3

mrpy.extra.reparameterise.ReparameteriseSampleLike.__init__

ReparameteriseSampleLike.**__init__**(*p1=None*, *p2=None*, *p3=None*, *logHs=None*, *alpha=None*, *beta=None*, ***kwargs*)

mrpy.extra.reparameterise.ReparameteriseSampleLike.dndlog10m

ReparameteriseSampleLike.**dndlog10m**(*log=False*)

Return the MRP in log10 space at ‘m’.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.ReparameteriseSampleLike.dndm

ReparameteriseSampleLike.**dndm**(*log=False*)

Return the MRP at *m*.

Parameters

log [logical, optional] Whether to return the natural log of the MRP.

mrpy.extra.reparameterise.ReparameteriseSampleLike.ngtm

ReparameteriseSampleLike.**ngtm**(*log=False*)

The number density greater than *mmin*.

Parameters

log [logical] Whether to return the natural log of the number density.

mrpy.extra.reparameterise.ReparameteriseSampleLike.p_T

ReparameteriseSampleLike.**p_T**(***kwargs*)

The new parameters as functions of theta

mrpy.extra.reparameterise.ReparameteriseSampleLike.rho_gtm

ReparameteriseSampleLike.**rho_gtm**(*log=False*)

The mass-weighted integral of the MRP, in reverse (ie. from high to low mass).

Parameters

log [logical] Whether to return the natural log of the density.

mrpy.extra.reparameterise.ReparameteriseSampleLike.theta_T

ReparameteriseSampleLike.**theta_T**(***kwargs*)

theta as functions of the new parameters p1,p2,p3

mrpy.fitting.fit_curve

Routines that implement simple least-squares fits directly to dn/dm without errors. Typically used for fitting directly to theoretical functions from EPS theory.

For more involved fits to non-binned data, see **:module:'mrpy.fit_perobj'**. For the definition of the likelihood involved in the fits within this module see `mrpy.likelihoods.CurveLike`.

Functions

get_fit_curve(data_m, data_mf, x0[, bounds, ...])	Perform basic LSQ fitting of the MRP curve to the given curve.
get_fit_expected(data_m, data_mf, x0[, ...])	Generate the expected MLE of MRP parameters, given a mass function from which the samples should be drawn.

mrpy.fitting.fit_curve.get_fit_curve

mrpy.fitting.fit_curve.**get_fit_curve**(*data_m*, *data_mf*, *x0*, *bounds=None*, *jac=True*, *sig_rhomean=inf*, *sig_data=1*, ***minimize_kw*)

Perform basic LSQ fitting of the MRP curve to the given curve. The fit is performed in log-log space

Parameters

data_m [array] Masses (not log)

data_mf [array] Mass function corresponding to m.

x0 [float, optional] Initial guess for each of the MRP parameters (hs, alpha, beta, lnA)

bounds [list of tuples, optional] If None, don't use bounds. If true, set bounds based on bounds passed.

jac [bool, optional] Whether to use analytic jacobian (usually a good idea)

sig_data [array_like, optional] The uncertainty of the data (standard deviation). This is used in the likelihood to weight different mass scales. If scalar, all mass scales are weighted evenly.

sig_rhomean,: float, optional This controls how much influence the total mean density of the universe has on the likelihood. The default value of *inf* means it is completely ignored. If it is 0, it becomes an absolute constraint, so that the total mass density of the universe is

perfectly matched (setting the normalisation). In between, it acts as an uncertainty on this value.

minimize_kw [dict] Any other parameters to `scipy.minimize()`.

Returns

res [*OptimizeResult*] The optimization result represented as a `OptimizeResult` object (see scipy documentation). 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.

The parameters are ordered by *logHs*, *alpha*, *beta*, *[lnA]*.

curve [`mrpy.likelihoods.CurveLike` object] An object containing the solution parameters and methods to access relevant quantities, such as the mass function, or jacobian and hessian at the solution.

Notes

Though the option to *not* use bounds for the fit is available, at this point it seems to yield unpredictable results. Unless the problem at hand is so poorly known as to be impossible to set appropriate bounds, it is encouraged to use them.

Furthermore, use as stringent bounds as possible, since the algorithm explores the edges, which can induce numerical error if values far from the solution are chosen.

Examples

The most obvious example is to generate a mass function curve from given parameters:

```
>>> from mrpy import MRP
>>> import numpy as np
>>> m = np.logspace(10,15,500)
>>> d = MRP(m,14.0,-1.9,0.75,norm=0).dndlog10m()
```

Then find the best-fit parameters for the resulting data:

```
>>> from mrpy.fitting.fit_curve import get_fit_curve
>>> res,curve = get_fit_curve(m,d,x0=[14.2,-1.8,0.8,0.5])
>>> print res.x
```

We can also use the `curve` object to explore some of the qualities of the fit

```
>>> print curve.hessian
>>> from matplotlib.pyplot import plot
>>> plot(curve.logm,curve.dndm(log=True))
>>> plot(curve.logm,np.log(d))
>>> print curve.stats.mean
```

mrpy.fitting.fit_curve.get_fit_expected

`mrpy.fitting.fit_curve.`**`get_fit_expected`**(*data_m*, *data_mf*, *x0*, *bounds=None*, *jac=True*, *V0=1*, *kappa=None*, ***minimize_kw*)

Generate the expected MLE of MRP parameters, given a mass function from which the samples should be drawn.

This integrates over the input mass function, so as to naturally weight each mass scale as if fitting an actual sample. It assumes that there are no uncertainties on the mass estimates.

Parameters

data_m [array] Masses (not log)

data_mf [array] Mass function corresponding to m.

x0 [float, optional] Initial guess for each of the MRP parameters (hs, alpha, beta, lnA)

bounds [list of tuples, optional] If None, don't use bounds. If true, set bounds based on bounds passed.

jac [bool, optional] Whether to use analytic jacobian (usually a good idea)

minimize_kw [dict] Any other parameters to `scipy.minimize()`.

Returns

res [*OptimizeResult*] The optimization result represented as a `OptimizeResult` object (see scipy documentation). 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.

The parameters are ordered by *logHs*, *alpha*, *beta*, *[lnA]*.

curve [`mrpy.likelihoods.CurveLike` object] An object containing the solution parameters and methods to access relevant quantities, such as the mass function, or jacobian and hessian at the solution.

Notes

Though the option to *not* use bounds for the fit is available, at this point it seems to yield unpredictable results. Unless the problem at hand is so poorly known as to be impossible to set appropriate bounds, it is encouraged to use them.

Furthermore, use as stringent bounds as possible, since the algorithm explores the edges, which can induce numerical error if values far from the solution are chosen.

Examples

The most obvious example is to generate a mass function curve from given parameters:

```
>>> from mrpy import MRP
>>> import numpy as np
>>> m = np.logspace(10,15,500)
>>> d = MRP(m,14.0,-1.9,0.75,norm=0).dndlog10m()
```

Then find the best-fit parameters for the resulting data:

```
>>> from mrpy.fitting.fit_curve import get_fit_expected
>>> res,curve = get_fit_curve(m,d,x0=[14.2,-1.8,0.8,0.5])
>>> print res.x
```

We can also use the `curve` object to explore some of the qualities of the fit

```
>>> print curve.hessian
>>> from matplotlib.pyplot import plot
>>> plot(curve.logm,curve.dndm(log=True))
>>> plot(curve.logm,np.log(d))
>>> print curve.stats.mean
```

mrpy.fitting.fit_sample

Routines that implement fits directly to samples of masses (or other values drawn from a TGGD).

For fits to binned data, see **:module:'mrpy.fit_curve'**. For the definition of the likelihood involved in the fits within this module see `mrpy.likelihoods.PerObjLike`.

This module also provides pre-defined prior functions, specifically, the `normal_prior`.

Functions

normal_prior(p, mean, sd)	A normal prior on each parameter.

mrpy.fitting.fit_sample.normal_prior

mrpy.fitting.fit_sample.**normal_prior**(*p*, *mean*, *sd*)

A normal prior on each parameter.

Parameters

p [list] Values of the parameters

mean [list] Values of the prior mean

sd [list] Values of the prior standard deviations. Set to inf if uniform prior required on a single parameter.

Returns

ll : The likelihood of the parameters given the priors

jac : The jacobian of the likelihood.

Classes

SimFit(m[, nm, mmin, V, hs_bounds, ...])	Per-object fits.

mrpy.fitting.fit_sample.SimFit

class mrpy.fitting.fit_sample.**SimFit**(*m*, *nm=None*, *mmin=None*, *V=1.0*, *hs_bounds=(10, 16)*, *alpha_bounds=(-1.99, -1.3)*, *beta_bounds=(0.1, 2.0)*, *lnA_bounds=(-40, -10)*, *prior_func=None*, *prior_kwargs=None*)

Per-object fits.

Parameters

m [array or list of arrays] Masses. Either an array or a list of arrays, each of which is a sample *to be analysed simultaneously*. In the latter case the samples should have the same underlying distribution, but may have differing truncation scales.

nm [array, optional] Specifies the number of occurrences of each variate in *m* (which should then ideally be unique). If not passed, each variate is assumed to occur once. This is useful for speeding up fits on quantized simulations. If *m* is a list of arrays, this should be also.

mmin [array_like, optional] The truncation mass of the sample. By default takes the lowest value of *m*. If *m* is a list of arrays, this should be a list.

V [array, optional] The volume of each subsample

hs_bounds, alpha_bounds, beta_bounds [2-tuple] 2-tuples specifying minimum and maximum values for each bound.

prior_func [function, optional] A function to calculate the likelihood arising from priors on the parameters. By default, uniform priors are assumed, which add nothing to the likelihood. The first parameter taken by the function should be the vector of parameter values, `[logHs,alpha,beta]`, after which arbitrary values are passed via *prior_kwargs*. It should return a tuple, with the first value being a float and constituting the likelihood arising from the prior, and the second being a 3-vector constituting the modification to the jacobian. This can be zero if no jacobian is desired.

prior_kwargs [dict] Arguments sent to the *prior_func*.

Notes

Use as stringent bounds as possible, since the algorithm explores the edges, which can induce numerical error if values far from the solution are chosen.

Methods

__init__(m[, nm, mmin, V, hs_bounds, ...])	
lnL(p[, ret_jac, debug])	Return the log-likelihood of the current model at the parameters *p*.
run_downhill([hs0, alpha0, beta0, lnA0, ...])	Downhill-gradient optimization.
run_mcmc([nchains, warmup, iterations, hs0, ...])	Per-object MCMC fit for masses *m*, using the *emcee* package.

mrpy.fitting.fit_sample.SimFit.__init__

SimFit.**__init__**(*m*, *nm=None*, *mmin=None*, *V=1.0*, *hs_bounds=(10, 16)*, *alpha_bounds=(-1.99, -1.3)*, *beta_bounds=(0.1, 2.0)*, *lnA_bounds=(-40, -10)*, *prior_func=None*, *prior_kwargs=None*)

mrpy.fitting.fit_sample.SimFit.lnL

SimFit.**lnL**(*p*, *ret_jac=False*, *debug=0*)

Return the log-likelihood of the current model at the parameters *p*.

Parameters

p [array] The values of the parameters, `[logHs, alpha, beta]`.

ret_jac [bool] Whether to return the jacobian as the second arg.

debug [int, optional] Set the level of info printed out throughout the function. Highest current level that is useful is 2.

Returns

ll [float] The log-likelihood at the parameters. This is exactly the same as used in the fitting process.

jac [length-3 array, optional] Returned only if *ret_jac* is *True*. The jacobian at the current parameter vector.

mrpy.fitting.fit_sample.SimFit.run_downhill

`SimFit.`**`run_downhill`**(*hs0=14.5*, *alpha0=-1.9*, *beta0=0.8*, *lnA0=-40.0*, *debug=0*, *jac=True*, ***minimize_kw*)

Downhill-gradient optimization.

Parameters

hs0, alpha0, beta0, lnA0: float, optional Initial guess for each of the MRP parameters.

debug [int, optional] Set the level of info printed out throughout the function. Highest current level that is useful is 2.

jac [bool, optional] Whether to use analytic jacobian (usually a good idea)

minimize_kw [dict] Any other parameters to `scipy.optimize.minimize()`.

Returns

downhill_res [*OptimizeResult*] The optimization result represented as a `OptimizeResult` object (see scipy documentation). 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.

The parameters are ordered by *logHs*, *alpha*, *beta*, *[lnA]*.

downhill_obj [`mrpy.likelihoods.PerObjLikeWeights` object] An object containing the solution parameters and methods to access relevant quantities, such as the mass function, or jacobian and hessian at the solution.

Examples

The most obvious example is to generate a sample of variates from given parameters:

```
>>> from mrpy.base.stats import TGGD
>>> r = TGGD(scale=1e14,a=-1.8,b=1.0,xmin=1e12).rvs(1e5)
```

Then find the best-fit parameters for the resulting data:

```
>>> from mrpy.fitting.fit_sample import SampleFit
>>> FitObj = SampleFit(r)
>>> res,obj = FitObj.run_downhill()
>>> print res.x
```

We can also use the `obj` object to explore some of the qualities of the fit

```
>>> print obj.hessian
>>> print obj.cov
```

The results are also stored in the class as *downhill_obj* and *downhill_res*.

```
>>> from matplotlib.pyplot import plot
>>> plot(FitObj.downhill_obj.logm,FitObj.downhill_obj.dndm(log=True))
>>> print obj.stats.mean, r.mean()
```

mrpy.fitting.fit_sample.SimFit.run_mcmc

`SimFit.`**`run_mcmc`**(*nchains=50*, *warmup=1000*, *iterations=1000*, *hs0=14.5*, *alpha0=-1.9*, *beta0=0.8*, *lnA0=-26.0*, *logm0=None*, *debug=0*, *opt_init=False*, *opt_kw=None*, *chainfile='chain.dat'*, *save_latent=True*, ***kwargs*)

Per-object MCMC fit for masses *m*, using the *emcee* package.

This creates an `emcee.EnsembleSampler` object with a correct model, and runs warmup and specified iterations. The entire `emcee.EnsembleSampler` object is returned, and stored in the instance of this class as `mcmc_res`. This affords greater flexibility, with the ability to run no warmup and 0 iterations, and run the iterations oneself.

Parameters

nchains [int, optional] Number of chains to use in the AIES MCMC algorithm

warmup [int, optional] Number (discarded) warmup iterations.

iterations [int, optional] Number of iterations to keep in the chain.

hs0, alpha0, beta0: float, optional Initial guess for each of the MRP parameters.

debug [int, optional] Set the level of info printed out throughout the function. Highest current level that is useful is 2.

opt_init [bool, optional] Whether to run a downhill optimization routine to get the best starting point for the MCMC.

opt_kw [dict, optional] Any arguments to pass to the downhill run.

kwargs : Any other parameters to `emcee.EnsembleSampler`.

Returns

mcmc_res [`emcee.EnsembleSampler` object] This object contains the stored chains, and other attributes.

Examples

The most obvious example is to generate a sample of variates from given parameters:

```
>>> from mrpy.base.stats import TGGD
>>> r = TGGD(scale=1e14,a=-1.8,b=1.0,xmin=1e12).rvs(1e5)
```

Then find the best-fit parameters for the resulting data:

```
>>> from mrpy.fitting.fit_sample import SampleFit
>>> FitObj = SampleFit(r)
```

```
>>> mcmc_res = FitObj.run_mcmc(nchains=10,warmup=100,iterations=100)
>>> print mcmc_res.flatchain.mean(axis=0)
```

3.5 Indices and tables

- genindex
- modindex
- search

Python Module Index

m

mrpy.base.core, 85
mrpy.base.special, 65
mrpy.base.stats, 71
mrpy.extra.analytic_model, 95
mrpy.extra.likelihoods, 90
mrpy.extra.physical_dependence, 89
mrpy.extra.reparameterise, 96
mrpy.fitting.fit_curve, 119
mrpy.fitting.fit_sample, 122

Index

Symbols

__init__() (mrpy.base.core.MRP method), 88
__init__() (mrpy.base.stats.TGGD method), 73
__init__() (mrpy.base.stats.TGGDln method), 78
__init__() (mrpy.base.stats.TGGDlog method), 82
__init__() (mrpy.extra.analytic_model.IdealAnalytic method), 95
__init__() (mrpy.extra.likelihoods.CurveLike method), 91
__init__() (mrpy.extra.likelihoods.SampleLike method), 93
__init__() (mrpy.extra.likelihoods.SampleLikeWeights method), 94
__init__() (mrpy.extra.reparameterise.AP1Curve method), 98
__init__() (mrpy.extra.reparameterise.AP1Sample method), 99
__init__() (mrpy.extra.reparameterise.Ap1MRP method), 100
__init__() (mrpy.extra.reparameterise.GG2Curve method), 102
__init__() (mrpy.extra.reparameterise.GG2MRP method), 103
__init__() (mrpy.extra.reparameterise.GG2Sample method), 105
__init__() (mrpy.extra.reparameterise.GG3Curve method), 106
__init__() (mrpy.extra.reparameterise.GG3MRP method), 108
__init__() (mrpy.extra.reparameterise.GG3Sample method), 109
__init__() (mrpy.extra.reparameterise.HTCurve method), 110
__init__() (mrpy.extra.reparameterise.HTMRP method), 112
__init__() (mrpy.extra.reparameterise.HTSample method), 113
__init__() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 115
__init__() (mrpy.extra.reparameterise.ReparameteriseMRP method), 116
__init__() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 118
__init__() (mrpy.fitting.fit_sample.SimFit method), 123

A

A_rhom() (in module mrpy.base.core), 85
AP1Curve (class in mrpy.extra.reparameterise), 97
Ap1MRP (class in mrpy.extra.reparameterise), 100
AP1Sample (class in mrpy.extra.reparameterise), 99

C

cdf() (mrpy.base.stats.TGGD method), 74
cdf() (mrpy.base.stats.TGGDln method), 78
cdf() (mrpy.base.stats.TGGDlog method), 82
central_moments() (mrpy.base.stats.TGGD method), 74
central_moments() (mrpy.base.stats.TGGDln method), 78
central_moments() (mrpy.base.stats.TGGDlog method), 82
CurveLike (class in mrpy.extra.likelihoods), 91

D

dndlog10m() (mrpy.base.core.MRP method), 88
dndlog10m() (mrpy.extra.analytic_model.IdealAnalytic method), 96
dndlog10m() (mrpy.extra.likelihoods.CurveLike method), 91
dndlog10m() (mrpy.extra.likelihoods.SampleLike method), 93
dndlog10m() (mrpy.extra.likelihoods.SampleLikeWeights method), 94
dndlog10m() (mrpy.extra.reparameterise.AP1Curve method), 98
dndlog10m() (mrpy.extra.reparameterise.Ap1MRP method), 101
dndlog10m() (mrpy.extra.reparameterise.AP1Sample method), 99
dndlog10m() (mrpy.extra.reparameterise.GG2Curve method), 102

dndlog10m() (mrpy.extra.reparameterise.GG2MRP method), 103
dndlog10m() (mrpy.extra.reparameterise.GG2Sample method), 105
dndlog10m() (mrpy.extra.reparameterise.GG3Curve method), 106
dndlog10m() (mrpy.extra.reparameterise.GG3MRP method), 108
dndlog10m() (mrpy.extra.reparameterise.GG3Sample method), 109
dndlog10m() (mrpy.extra.reparameterise.HTCurve method), 111
dndlog10m() (mrpy.extra.reparameterise.HTMRP method), 112
dndlog10m() (mrpy.extra.reparameterise.HTSample method), 113
dndlog10m() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 115
dndlog10m() (mrpy.extra.reparameterise.ReparameteriseMRP method), 116
dndlog10m() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 118
dndm() (in module mrpy.base.core), 85
dndm() (mrpy.base.core.MRP method), 89
dndm() (mrpy.extra.analytic_model.IdealAnalytic method), 96
dndm() (mrpy.extra.likelihoods.CurveLike method), 92
dndm() (mrpy.extra.likelihoods.SampleLike method), 93
dndm() (mrpy.extra.likelihoods.SampleLikeWeights method), 94
dndm() (mrpy.extra.reparameterise.AP1Curve method), 98
dndm() (mrpy.extra.reparameterise.Ap1MRP method), 101
dndm() (mrpy.extra.reparameterise.AP1Sample method), 99
dndm() (mrpy.extra.reparameterise.GG2Curve method), 102
dndm() (mrpy.extra.reparameterise.GG2MRP method), 104
dndm() (mrpy.extra.reparameterise.GG2Sample method), 105
dndm() (mrpy.extra.reparameterise.GG3Curve method), 106
dndm() (mrpy.extra.reparameterise.GG3MRP method), 108
dndm() (mrpy.extra.reparameterise.GG3Sample method), 109
dndm() (mrpy.extra.reparameterise.HTCurve method), 111
dndm() (mrpy.extra.reparameterise.HTMRP method), 112
dndm() (mrpy.extra.reparameterise.HTSample method), 113
dndm() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 115
dndm() (mrpy.extra.reparameterise.ReparameteriseMRP method), 117
dndm() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 118

E

entire_integral() (in module mrpy.base.core), 86
expected_likelihood() (in module mrpy.extra.likelihoods), 90

G

G1() (in module mrpy.base.special), 66
G2() (in module mrpy.base.special), 66
gamma() (in module mrpy.base.special), 66
gammainc() (in module mrpy.base.special), 68
get_fit_curve() (in module mrpy.fitting.fit_curve), 119
get_fit_expected() (in module mrpy.fitting.fit_curve), 120
GG2Curve (class in mrpy.extra.reparameterise), 101
GG2MRP (class in mrpy.extra.reparameterise), 103
GG2Sample (class in mrpy.extra.reparameterise), 104
GG3Curve (class in mrpy.extra.reparameterise), 106
GG3MRP (class in mrpy.extra.reparameterise), 107
GG3Sample (class in mrpy.extra.reparameterise), 109

H

HTCurve (class in mrpy.extra.reparameterise), 110
HTMRP (class in mrpy.extra.reparameterise), 111
HTSample (class in mrpy.extra.reparameterise), 113
hyperReg_2F2() (in module mrpy.base.special), 70

I

IdealAnalytic (class in mrpy.extra.analytic_model), 95

L

lnL() (mrpy.fitting.fit_sample.SimFit method), 123
log_mass_mode() (in module mrpy.base.core), 86

M

MRP (class in mrpy.base.core), 88
mrp_b13() (in module mrpy.extra.physical_dependence), 90
mrp_params_b13() (in module mrpy.extra.physical_dependence), 90
mrpy.base.core (module), 85
mrpy.base.special (module), 65
mrpy.base.stats (module), 71
mrpy.extra.analytic_model (module), 95
mrpy.extra.likelihoods (module), 90
mrpy.extra.physical_dependence (module), 89
mrpy.extra.reparameterise (module), 96
mrpy.fitting.fit_curve (module), 119

mrpy.fitting.fit_sample (module), 122

N

ngtm() (in module mrpy.base.core), 87
ngtm() (mrpy.base.core.MRP method), 89
ngtm() (mrpy.extra.analytic_model.IdealAnalytic method), 96
ngtm() (mrpy.extra.likelihoods.CurveLike method), 92
ngtm() (mrpy.extra.likelihoods.SampleLike method), 93
ngtm() (mrpy.extra.likelihoods.SampleLikeWeights method), 94
ngtm() (mrpy.extra.reparameterise.AP1Curve method), 98
ngtm() (mrpy.extra.reparameterise.Ap1MRP method), 101
ngtm() (mrpy.extra.reparameterise.AP1Sample method), 99
ngtm() (mrpy.extra.reparameterise.GG2Curve method), 102
ngtm() (mrpy.extra.reparameterise.GG2MRP method), 104
ngtm() (mrpy.extra.reparameterise.GG2Sample method), 105
ngtm() (mrpy.extra.reparameterise.GG3Curve method), 107
ngtm() (mrpy.extra.reparameterise.GG3MRP method), 108
ngtm() (mrpy.extra.reparameterise.GG3Sample method), 109
ngtm() (mrpy.extra.reparameterise.HTCurve method), 111
ngtm() (mrpy.extra.reparameterise.HTMRP method), 112
ngtm() (mrpy.extra.reparameterise.HTSample method), 114
ngtm() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 115
ngtm() (mrpy.extra.reparameterise.ReparameteriseMRP method), 117
ngtm() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 118
normal_prior() (in module mrpy.fitting.fit_sample), 122
normalised_central_moments() (mrpy.base.stats.TGGD method), 74
normalised_central_moments() (mrpy.base.stats.TGGDln method), 79
normalised_central_moments() (mrpy.base.stats.TGGDlog method), 83

P

p_T() (mrpy.extra.reparameterise.AP1Curve method), 98
p_T() (mrpy.extra.reparameterise.Ap1MRP method), 101
p_T() (mrpy.extra.reparameterise.AP1Sample method), 100
p_T() (mrpy.extra.reparameterise.GG2Curve method), 102
p_T() (mrpy.extra.reparameterise.GG2MRP method), 104
p_T() (mrpy.extra.reparameterise.GG2Sample method), 105
p_T() (mrpy.extra.reparameterise.GG3Curve method), 107
p_T() (mrpy.extra.reparameterise.GG3MRP method), 108
p_T() (mrpy.extra.reparameterise.GG3Sample method), 110
p_T() (mrpy.extra.reparameterise.HTCurve method), 111
p_T() (mrpy.extra.reparameterise.HTMRP method), 112
p_T() (mrpy.extra.reparameterise.HTSample method), 114
p_T() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 115
p_T() (mrpy.extra.reparameterise.ReparameteriseMRP method), 117
p_T() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 118
pdf() (mrpy.base.stats.TGGD method), 75
pdf() (mrpy.base.stats.TGGDln method), 79
pdf() (mrpy.base.stats.TGGDlog method), 83
polygamma() (in module mrpy.base.special), 70

Q

quantile() (mrpy.base.stats.TGGD method), 75
quantile() (mrpy.base.stats.TGGDln method), 79
quantile() (mrpy.base.stats.TGGDlog method), 83

R

raw_moments() (mrpy.base.stats.TGGD method), 75
raw_moments() (mrpy.base.stats.TGGDln method), 80
raw_moments() (mrpy.base.stats.TGGDlog method), 84
ReparameteriseCurveLike (class in mrpy.extra.reparameterise), 114
ReparameteriseMRP (class in mrpy.extra.reparameterise), 116
ReparameteriseSampleLike (class in mrpy.extra.reparameterise), 117
rho_gtm() (in module mrpy.base.core), 87
rho_gtm() (mrpy.base.core.MRP method), 89
rho_gtm() (mrpy.extra.analytic_model.IdealAnalytic method), 96
rho_gtm() (mrpy.extra.likelihoods.CurveLike method), 92
rho_gtm() (mrpy.extra.likelihoods.SampleLike method), 93
rho_gtm() (mrpy.extra.likelihoods.SampleLikeWeights method), 95
rho_gtm() (mrpy.extra.reparameterise.AP1Curve method), 98
rho_gtm() (mrpy.extra.reparameterise.Ap1MRP method), 101

rho_gtm() (mrpy.extra.reparameterise.AP1Sample method), 100
rho_gtm() (mrpy.extra.reparameterise.GG2Curve method), 103
rho_gtm() (mrpy.extra.reparameterise.GG2MRP method), 104
rho_gtm() (mrpy.extra.reparameterise.GG2Sample method), 105
rho_gtm() (mrpy.extra.reparameterise.GG3Curve method), 107
rho_gtm() (mrpy.extra.reparameterise.GG3MRP method), 108
rho_gtm() (mrpy.extra.reparameterise.GG3Sample method), 110
rho_gtm() (mrpy.extra.reparameterise.HTCurve method), 111
rho_gtm() (mrpy.extra.reparameterise.HTMRP method), 113
rho_gtm() (mrpy.extra.reparameterise.HTSample method), 114
rho_gtm() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 115
rho_gtm() (mrpy.extra.reparameterise.ReparameteriseMRP method), 117
rho_gtm() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 119
run_downhill() (mrpy.fitting.fit_sample.SimFit method), 124
run_mcmc() (mrpy.fitting.fit_sample.SimFit method), 125
rvs() (mrpy.base.stats.TGGD method), 76
rvs() (mrpy.base.stats.TGGDln method), 80
rvs() (mrpy.base.stats.TGGDlog method), 84

S

SampleLike (class in mrpy.extra.likelihoods), 92
SampleLikeWeights (class in mrpy.extra.likelihoods), 93
SimFit (class in mrpy.fitting.fit_sample), 122

T

TGGD (class in mrpy.base.stats), 72
TGGDln (class in mrpy.base.stats), 76
TGGDlog (class in mrpy.base.stats), 80
theta_T() (mrpy.extra.reparameterise.AP1Curve method), 98
theta_T() (mrpy.extra.reparameterise.Ap1MRP method), 101
theta_T() (mrpy.extra.reparameterise.AP1Sample method), 100
theta_T() (mrpy.extra.reparameterise.GG2Curve method), 103
theta_T() (mrpy.extra.reparameterise.GG2MRP method), 104
theta_T() (mrpy.extra.reparameterise.GG2Sample method), 106
theta_T() (mrpy.extra.reparameterise.GG3Curve method), 107
theta_T() (mrpy.extra.reparameterise.GG3MRP method), 109
theta_T() (mrpy.extra.reparameterise.GG3Sample method), 110
theta_T() (mrpy.extra.reparameterise.HTCurve method), 111
theta_T() (mrpy.extra.reparameterise.HTMRP method), 113
theta_T() (mrpy.extra.reparameterise.HTSample method), 114
theta_T() (mrpy.extra.reparameterise.ReparameteriseCurveLike method), 116
theta_T() (mrpy.extra.reparameterise.ReparameteriseMRP method), 117
theta_T() (mrpy.extra.reparameterise.ReparameteriseSampleLike method), 119