Numerical Recipes for Astrophysics solutions hand-in assignment-1

Luther Algra - s1633376

April 9, 2019

Abstract

The current document contains the solutions for the first hand-in assignment of Numerical Recipes. Each sub-question (1.a, 1.b, 1.c, ..., 3.a, 3.b) is given its own section that consists of the question and its solution. The solution does for most question consists of two files with code, followed by the output of the code. The first file with code prints/saves the answer of the question and the second file contains helper functions to obtain this answer. Some questions have additional information.

Assignment 1.a

Question

Evaluate the Poisson probability function (equation 1) to at least six significant digits for the values: $(\lambda,\ k) = (1, 0)$, $(5, 10)$, $(3, 21)$ and $(2.6, 40)$.

$$P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{1}$$

Solution

The Poisson distribution is evaluated up to 15 significant digits for the requested values of (λ, k). The file ./code/assignment1_ a.py evaluates the distribution and prints the results. The file ./code/mathlib/utils.py contains the Poisson distribution and the factorial function that are necessary to obtain the result. The code and its output can be found below.

Code - output

The code that produces the output.

```python
import mathlib.utils

for mean, events in [(1,0), (5,10), (3,21), (2.6,40)]:
    print('Poisson(mean:{0:.1f}, events:{1:.0f}) = {2:0.15E}'
          .format(mean, events, mathlib.utils.poisson(mean, events)))
```

./code/assignment1_a.py

Code - helper

The code for the Poisson distribution and the factorial function.

```python
import numpy as np

def factorial(number):
    """
        Calculate the factorial for a number
```

```
6
7       In:
8           param: number -- The number to calculate the factorial for.
9       Out:
10          return: The factorial of the input number.
11      """
12
13      # If number < 20 convert it to an int64 to not lose any precision.
14      # else use a np.float64. ( = lose in precision)
15      # Note: Only datatypes with 64 or less bits are allowed.
16      if number < 20:
17          number = np.int64(number)
18      else:
19          number = np.float64(number)
20
21      if number < 0:
22          raise RuntimeError('Invalid input for factorial: {n}. \
23                              Input must be larger or equal than zero.')
24      elif np.int64(number) == 0: #!0 = 1
25          return 1
26      else:
27          return number*factorial(number-1)
28
29
30  def poisson(average, events):
31      """
32          Evaluate the Poisson distribution.
33
34      In:
35          param: average -- The average number of events.
36          param: events -- The number of observed events.
37
38      Out:
39          return: The evaluation of the Poisson distribution
40                  for the given parameters.
41      """
42
43      return (average**events * np.exp(-average))/factorial(events)
```

./code/mathlib/utils.py

Output

The output produced by /code/assignment1_ a.py

```
1  Poisson(mean:1.0, events:0) = 3.678794411714423E-01
2  Poisson(mean:5.0, events:10) = 1.813278870782187E-02
3  Poisson(mean:3.0, events:21) = 1.019339824111019E-11
4  Poisson(mean:2.6, events:40) = 3.615123994937689E-33
```

./output/assignment1_a_out.txt

Assignment 1.b

Question

Write a random number generator that returns a random floating-point number between 0 and 1. At minimum, use some combination of an (M)LCG and a 64-bit XOR-shift. Plot a sequential of random numbers against each other in a scatter plot (x_{i+1} vs x_i) for the first 100 numbers generated. Additionally, have your code generate 1,000,000 random numbers and plot the result of binning these in 20 bins 0.05 wide. The seed value should be the first output when the code is run.

Solution

To create the random number generator three methods are used. The first two methods are the linear congruential generator (LCG) and a 64-bit XOR-shift. The last method is multiply with carry (MWC). The methods are combined to generate a new seed from the current seed. The new seed is generated by first cloning the current seed and then performing a bitwise xor operation between the XOR-shift method and the LCG method evaluated at the cloned seed. The result is set as the new seed. Next, the result of the MWC method for the cloned seed is subtracted from the new seed. Finally, a bitwise 'and' operation is performed on the new seed to only keep the last 32 bits [1]. This is done to fix number from becoming 'infinite' large[2].

The random number generator should only be initialized once and be used in a continues way through out the full assignment. I however splitted each exercise in a small file to make the report look better. The continues usage of the random number generator trough out the full assignment is as result done by initializing it in a new assignment with the final seed of the previous assignment. The code that displays the output will therefore print the seed twice. The first time is the initial seed. The second time is the value of the seed after generating the plots. The value that the seed has the second time is thus used as initial seed in assignment 2.a.

The code for this assignment is located in two files: ./code/assignment1_ b .py and ./code/-mathlib/rng.py. The first file contains the code that creates the plots and displays the output. The second file contains the code for the random number generator. The code for both files and the output can be found below.

Code - output

The code that produces the output.

```
import mathlib.utils
import mathlib.rng
import matplotlib.pyplot as plt
import numpy as np

# Initialize the random number generator.
rng = mathlib.rng.RNG(5332341)
print('Initializing seed: ', 5332341)

# Plot sequential random numbers against each other in a scatter plot.
x_values = rng.gen_next_floats(1001)
plt.scatter(x_values[0:1000], x_values[1:1001], s=2)
plt.savefig('./plots/random_next_current.pdf')
plt.figure()

# Plot 1e6 random points.
values = rng.gen_next_floats(int(1e6))
```

[1] The performed 'and' operation actually changes the number of bits that is used to store the new seed in python. This would not be the case in other languages such as $c++$ or *java*. The performed 'and' operation will in these languages just puts all bits, expect the last 32 bits, to zero.

[2] Python types can grow as large as possible. Continuous performing bit shifts to the left (i.e continues calling the XOR-shift method) will therefore result in larger and larger numbers. Eventually the number becomes so large that simple bitwise operation take a significant of time.

```
18
19 plt.hist(values, bins=20, range=(0,1),alpha=0.8,edgecolor='black')
20 plt.savefig('./plots/random_uniformness.pdf')
21 print('Seed after creating plots: ', rng.get_seed())
```

./code/assignment1_b.py

Code - random number generator

The code for the random number generator. The code contains additional function's that are not used in this exercise, but will be used in the upcoming exercises.

```
1  import numpy as np
2
3  class RNG():
4      """
5          A class representing a random number generator (RNG)
6      """
7
8      def __init__(self, seed):
9          """
10             Create a new instance of the random number generator.
11
12         In:
13             param: seed -- The seed of the random number generator.
14                            This must be an positive integer.
15         """
16
17         self._seed = seed
18         self._uint32_max = 0xFFFFFFFF
19
20         # The values for the Linear congruential generator.
21         self._lgc_a = 0xBEEF
22         self._lgc_c = 0xBADC0DE
23         self._lgc_m = 0xFFFFFFFF
24
25         # The values for the Xor shift.
26         self._xor_a1 = 3
27         self._xor_a2 = 29
28         self._xor_a3 = 7
29
30         # The values for the multiply with carry.
31         self._mwc_a = 0xFFFFFFF
32
33
34     def get_seed(self):
35         """
36             Get the current seed.
37
38         Out:
39             return: The current seed of the generator.
40         """
41         return self._seed
42
43     def gen_next_int(self):
44         """
45             Generate a pseudo random unsigned integer.
46
47         Out:
48             return: A pseudo random unsigned integer,
49         """
50         return self._gen_next()
51
52     def gen_next_float(self):
53         """
54             Generate a random float between 0 and 1.
55
56         Out:
57             return: A pseudo random float between 0 and 1.
```

```
        """
        return self._gen_next() * (1.0 / self._uint32_max)

    def gen_next_floats(self, amount):
        """
            Generate multiple random floats
            between 0 and 1.
        In:
            param: amount -- The amount of floats to generate.
        Out:
            return: An array with 'amount' random floats
                    between 0 and 1.
        """

        samples = np.zeros(amount)

        for i in range(amount):
            samples[i] = self.gen_next_float()

        return samples

    def gen_next_float_range(self, low, high):
        """
            Generate a float in a specific range.

        In:
            param: low -- The lowest possible value to generate.
            param: high -- The highest possible value to generate.
        Out:
            return: A float in the range low, high.
        """

        ret = self.gen_next_float()
        return  ret* (high-low) + low

    def gen_next_floats_range(self, low, high, amount):
        """
            Generate multiple floats in a specific range.

         In:
            param: low -- The lowest possible value to generate.
            param: high -- The highest possible value to generate.
            param: amount -- The amount of floats to generate.
        Out:
            return: An array of floats in the range low, high
        """

        floats = self.gen_next_floats(amount)
        return floats * (high - low) + low


    def _LCG(self, number):
        """
            Execute the linear congurential algorithm (LGC) on the
            provided number.

        In:
            param: number -- The number to execute the LCA on.
        Out:
            return: The number produced by the LCA.
         """
        return (self._lgc_a*number + self._lgc_c ) % self._lgc_m

    def _XOR_shift(self, number):
        """
            Execute the XOR-shift algorithm on the
            input number.
        In:
            param: number -- The number to XOR-shift.
        Out:
```

```
            return: The number produced by XOR-shift.
        """

        # Shift to the right and then bitwise xor.
        number ^= (number >> self._xor_a1)
        # Shift to the left and then bitwise xor.
        number ^= (number << self._xor_a2)
        # Shift to the right and then bitwise xor.
        number ^= (number >> self._xor_a3)

        return number

    def _mwc(self, number):
        """
            Perform multiply with carry (MWC) on
            the given input.
        In:
            param: number -- The number to perform MWC on.
        Out:
            return The new number.
        """
        return self._mwc_a * (number & (self._uint32_max -1)) + (number >> 32)

    def _gen_next(self):
        """
            Generate the next pseudo random value
            and update the seed.
        """

        old_seed = self._seed

        self._seed = self._XOR_shift(old_seed)
        self._seed ^= self._LCG(old_seed)
        self._seed -= self._mwc(old_seed)

        # Only keep the last 32 bits to prevent number becoming to large.
        self._seed &= self._uint32_max

        return abs(self._seed)

    def rejection_sampling(self, sample_dist, encl_func, \
                           encl_unif_transf, samples):
        """
            Perform rejection sampling.

        In:
            param: sample_dist -- The distribution to sample from.
            param: encl_func -- The function that encloses the distribution.
                                This is the distribution that is chosen to
                                enclose the sample distribution multiplied
                                by a scalar x so that
                                sample_dist(x) < enclosing_func(x) for x in
                                the intervall of which the distribution holds.

            param: encl_unf_transf -- The transformation that transforms a
                                      uniform distributed variable
                                      between 0 and 1 to a variable
                                      distributed according to the chosen
                                      enclosing distribution.
            param: samples -- The amount of samples to sample from 'sample_dist
    '.

        Out:
            return: An array of 'samples' samples sampled
                    from 'sample_dist'.
        """

        # An array that stores the generated samples.
        ret = np.zeros(samples)

```

```
197         # While there are still samples to sample
198         while samples > 0:
199 
200             # Generate a uniform variable between 0 and 1.
201             uniform = self.gen_next_float()
202 
203             # Transform it to a variable distributed
204             # by the enclosing distribution.
205             transformed_x = encl_unif_transf(uniform)
206 
207             # Generate a random uniform variable between
208             # 0 and the height of the enclosing function
209             # evaluated at transformed_x.
210             # Note: enclosing functions = enclosing distribution * scalar
211             transformed_y = self.gen_next_float_range(0, \
212                                                       encl_func(transformed_x))
213 
214             # If the generated y value is within the distribution
215             # to sample from, then a sample is found.
216             if transformed_y <= sample_dist(transformed_x):
217                 ret[samples-1] = transformed_x
218                 samples -= 1  #1 sample less to find.
219 
220         return ret
221 
222 
223     def gen_uniform_spherical_surface_coords(self, samples):
224         """
225             Generate pairs of angles phi and theta uniform
226             distributed around the surface of a sphere.
227             Phi is the declination and theta is the ascension.
228 
229         In:
230             param: samples -- The amount of angle pairs to generate.
231         Out:
232             return: An array containing 'samples' pairs
233                     of angles theta and phi that are
234                     uniform distributed around the
235                     surface of a sphere
236         """
237 
238         # Transformations from a uniform variable between 0 and 1
239         # to uniformly distributed angles around the unit sphere.
240         theta_transf = lambda x : np.arccos(1-2*x)
241         phi_transf = lambda x : 2*np.pi*x
242 
243         # Generate 'samples' uniform floats.
244         samples_gen = self.gen_next_floats(samples)
245 
246         # Generate the uniform distributed angles.
247         return theta_transf(samples_gen), phi_transf(samples_gen)
```

./code/mathlib/rng.py

Output - Text

The text output produced by /code/assignment1_ b.py

```
1 Initializing seed:  5332341
2 Seed after creating plots:  3197832550
```

./output/assignment1_b_out.txt

Output - plots

The plots produced by /code/assignment1_ b.py (see next page).

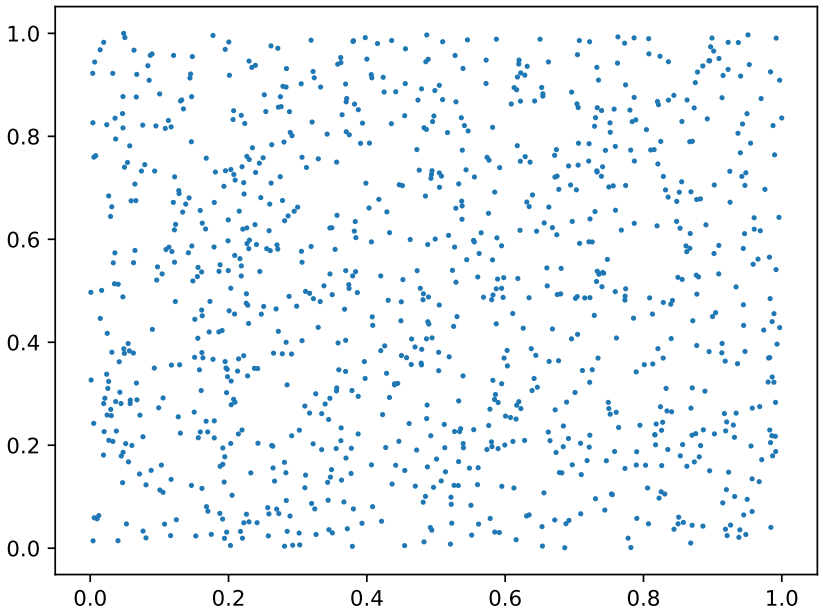

Figure 1: The result of plotting x_{i+1} against x_i for 1000 values.

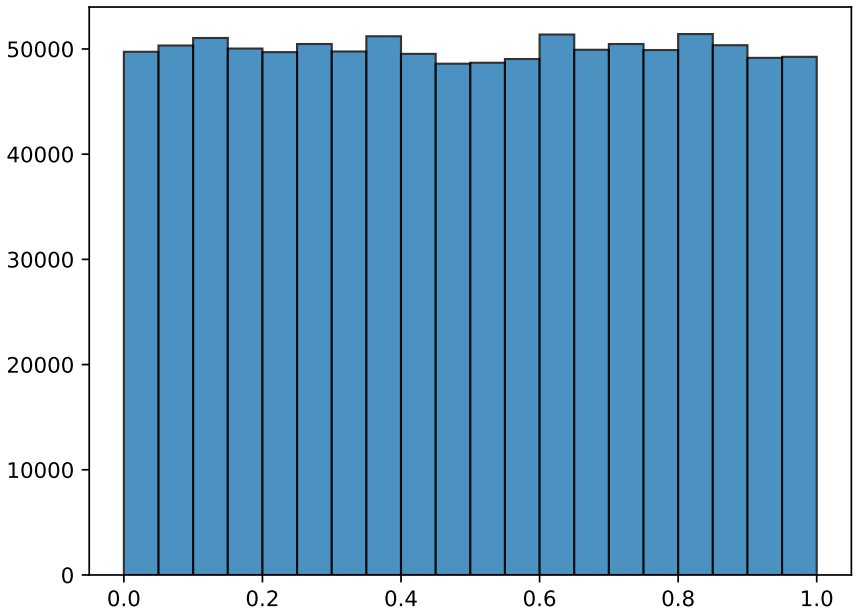

Figure 2: The uniformness of the random number generator for 1 million generated values between 0 and 1. The values are binned in 20 bins

Assignment 2.a

Question

The number density profile for satellite galaxies is given by,

$$n(x) = A\langle N_{sat}\rangle \left(\frac{x}{b}\right)^{a-3} \exp\left[-\left(\frac{x}{b}\right)^{c}\right] \tag{2}$$

where x is the radius relative to the virial radius, and a, b and c are free parameters. The value of A here normalizes the integral such that the integral from $x = 0$ to $x = 5$ gives,

$$\langle N_{sat}\rangle = \iiint_V n(x)dV \tag{3}$$

Have your code randomly generate three numbers $1.1 < a < 2.5$, $0.5 < b < 2$ and $1.5 < c < 4$. Write a numerical integrator to solve equation 3 in spherical coordinates for A given those three parameters, taking $\langle N_{sat}\rangle = 100$. Output the numbers a, b, c and A.

Solution

The value of A is found with the help of Romberg integration. The code for this assignment is located in two files:./code/assignment2_ a.py and ./code/mathlib/integrate.py. The first file contains the code that generates the values a, b, c and A. The second file contains the implementation for Romberg integration. The code for both files and the output can be found below.

Code - output

The code for generating the output.

```python
import mathlib.rng
import mathlib.integrate
import numpy as np

# Recreate the random number generator with the last seed of the previous
    exercise.
# This is done to make the execution of the random number generator
# continues through out the all assigments. See assigment 1.b in
# the report for more information.
rng = mathlib.rng.RNG(3197832550)

# Generate the random numbers.
a = rng.gen_next_float_range(1.1, 2.5)
b = rng.gen_next_float_range(0.5, 2)
c = rng.gen_next_float_range(1.5, 4)

# The density function in spherical coordinates integrated
# over the angles theta and phi.
density = lambda x: b**2  * (x/b)**(a-1)* np.exp(-(x/b)**c) * 4 *np.pi

# Find the normalization constant.
A = 1/mathlib.integrate.romberg(density, 0, 5, 15)

for label,value in zip(['a','b','c','A'], [a,b,c,A]):
    print(label + ' = {0:.18f}'.format(value))

# Print the seed, which is needed in assigment 2.d.
print('Final seed: ', rng.get_seed())
```

./code/assignment2_a.py

Code - romberg

The code for romberg integration.

```
import numpy as np

def trapezoid(function, a, b, num_trap):
    """
        Perform trapezoid integration

    In:
        param: function -- The function to integrate.
        param: a -- The value to start the integration at.
        param: b -- The value to integrate the function to.
        param: num_trap -- The number of trapezoids to use.

    Out:
        return: An approximation of the integral from a to b of
                the function 'function'
    """

    # The step size for the integration range.
    dx = (b-a)/num_trap

    # Find trapezoid points.
    x_i = np.arange(a,b+dx,dx)

    # Determine the area of all trapezoids.
    area = dx*(function(a) + function(b))/2
    area += np.sum(function(x_i[1:num_trap])) *dx

    return area



def romberg(function, a, b, num_trap):
    """
        Perform romberg integration

    In:
        param: function -- The function to integrate.
        param: a -- The value to start the integration at.
        param: b -- The value to stop the integrate at.
        param: num_trap -- The maximum number of initial trapezoid
                           to use to approximate the area.

    Out:
        return: An approximation of the integral from a to b of
                the function 'function'
    """

    # The array to store the combined results in.
    results = np.zeros(num_trap)

    # The current combination:
    # 0 = combine trapezoids, 1 = combine combined trapezoids,
    # 2 = combine the combined combined trapezoids etc.
    for combination in range(0, num_trap-1):

        # Iterate and combine.
         for j in range(1, num_trap-combination):
            # Create the initial trapezoids to combine.
            if combination == 0:
                # We need two trapezoids to combine the very first time.
                if j == 1:
                    results[j-1] = trapezoid(function, a, b, 1)

                results[j] = trapezoid(function,a,b,2**j)

            # Combine.
            power = 4**(combination+1)
```

```
68                 results[j-1] = (power*results[j] - results[j-1])/(power-1)
69
70         return results[0]
```

./code/mathlib/integrate.py

Output - Text

The produced output from ./code/assigment2_ a.py.

```
1 a = 1.316496837562065814
2 b = 1.305096297363074420
3 c = 1.790962180074062715
4 A = 0.051464524931584120
5 Final seed:  499869219
```

./output/assignment2_a_out.txt

Assignment2.b

Question

Make a log-log plot of and plot single points for $n(10^{-4})$, $n(10^{-2})$, $n(10^{-1})$, $n(1)$ and $n(5)$ with an axis range from $x = 10^{-4}$ to $x_{max} = 5$. Interpolate the values in between based on just these points - make sure to argue in the comments of your code why you chose to interpolate in a certain way.

Solution

Four out of the 5 points plotted in logarithmic space seems to follow a straight line (see figure 3 below the code section). If we assume that the error on these points is small, then this suggests that the underlying physical relation might be a linear relation in logarithmic space. The last point deviates from this relation with a rapid drop. If we again assume that the error is small, then this suggests that there might be an exponential decrease.

Most of the data seems to follow a linear relation in log space. If you need to interpolate a value, then you likely need to interpolate withing this linear part, as it wouldn't make much sense to obtained most of your data in a region that is uninteresting. An interpolator should furthermore serve the data as best as possible. Interpolating the data with a linear and for example a polynomial interpolator, shows that the polynomial interpolator better interpolates the expected exponential drop, where as the linear interpolator better interpolates the expected power law (see figure 4 below the code section). Many other interpolation methods, such as cubic splines and nearest neighbor interpolation, also interpolate the expected exponential drop better than the linear interpolator, but fail to interpolate the linear relation with high accuracy (not shown).

Based on the above two arguments there is therefore chosen to interpolate the points with a **linear interpolate** in log space.

The code for this assignment consists of two files: ./code/assignment2_ b.py and ./code/-mathlib/interpolate.py. The first file creates the plots and the second file implements the interpolation methods. The code of the two files and the results can be found below.

Code - output

The code that interpolates the values and creates the plot.

```
import mathlib.interpolate as interp
import matplotlib.pyplot as plt
import numpy as np

# The values of abc found in assigment 2.1.
a = 1.316496837562065814
b = 1.305096297363074420
c = 1.790962180074062715
A = 0.051464524931584120

# The volume number density function that is
# used to create the y-values of the points.
density = lambda x: A*(x/b)**(a-3)* np.exp(-(x/b)**c)

# The points.
points_x = np.array([1e-4, 1e-2, 1e-1, 1, 5.0])
points_y = density(points_x)

# Take the logarithm.
points_x_log = np.log10(points_x)
points_y_log = np.log10(points_y)

# Points to interpolate.
interpolate_x = np.linspace(1e-4, 5, 2000)
interpolate_x_log = np.log10(interpolate_x)

# Create the interpolated points.
lin_interp = list()
poly_interp = list()

for x in interpolate_x_log:
    lin_interp.append(10**interp.interpolate_linear(points_x_log,\
                                                    points_y_log, x))
    poly_interp.append(10**interp.interpolate_neville(points_x_log, \
                                                    points_y_log, x))

# First only plot the points and then plot the points
# with the interpolated values.
plt.loglog()
plt.scatter(points_x, points_y, s=15, color = 'black',zorder=1)
plt.savefig('./plots/points.pdf')
plt.plot(interpolate_x, lin_interp, label = 'lineair',
         linestyle = '--', zorder = 0)
plt.plot(interpolate_x, poly_interp, label= 'polynomial',
         linestyle = '-.', zorder = 0)
plt.legend()
plt.savefig('./plots/interpolate.pdf')
```

./code/assignment2_b.py

Code - interpolate

The code of the linear and polynomial interpolator.

```
import numpy as np

def _find_bisection_idx(x_vals, points, x):
    """
        Find the index of the point that is the closest to x from the left
        in the provided array of known x points.

    In:
        param: x_vals -- The known x value's.
        param: points -- The minimum number of required points
                         that are needed to interpolate. For example 2
                         when using linear interpolation.
```

```
        param: x -- The point to find the closest point for.
    Out:
        return: The index of the closest known point that is left of x,
                corrected by the minimum number of required points needed
                for interpolation.

    """


    # Index of known point of the right.
    edge_right_idx = len(x_vals) - 1
    # Index of known point of the left.
    edge_left_idx = 0
    # Index of the known point in the center of the left and right point.
    edge_mid_idx = 0

    # While the known left and right point are not separated
    # by known points, perform bisection.
    while edge_right_idx - edge_left_idx > 1:

        # Use the left and right index to find the index
        # of the point in the center.
        edge_mid_idx = (edge_right_idx + edge_left_idx) >> 1

        # Is the point x left of the point in the middle?
        if x_vals[edge_mid_idx] > x: #True -> set right index to middle
            edge_right_idx = edge_mid_idx
        else: #False, set left index to middle index
            edge_left_idx = edge_mid_idx


    # The point is now found, the interpolation might need
    # need a minimum of point 'points' for interpolation. If there are not
    # enough points on the right of the left_index then decrease the index
    # until there are enough.

    # Not enough points because the left point is to far to the right.
    if edge_left_idx + points > len(x_vals):

        # Move to the left until there are enough points.
        while edge_left_idx + points > len(x_vals):
            edge_left_idx -= 1

        # Not enough points in the dataset.
        if edge_left_idx < 0:
            raise "Dataset is to small to interpolate x. Not enough points"

        return edge_left_idx
    else: # Enough points are available
        return edge_left_idx



def interpolate_linear(x_vals, y_vals, x):
    """
            Perform linear interpolation

    In:
        param: x_vals -- The known x values.
        param: y_vals -- The known y values.
        param: x -- The point to linear interpolate using the known values.

    Out:
        return: The interpolated y value for x.
    """

    # Find the index of the closest known value to the left of x.
    idx = _find_bisection_idx(x_vals, 2, x)

    # Horizontal line, no need to interpolate.
```

```
    if y_vals[idx] == y_vals[idx+1]:
        return y_vals[idx]

    # Linear interpolate.
    else:

        # Find the slope between the points x_{idx} and x_{idx} + 1
        a = (y_vals[idx+1] - y_vals[idx])/(x_vals[idx+1] - x_vals[idx])

        # Find the intersect.
        b =  y_vals[idx] - a*x_vals[idx]

        # Linear interpolate the point x.
        return a*x + b


def interpolate_neville(x_vals, y_vals, x):
    """
        Perform polynomial interpolation using Neveille's algorithm

       In:
        param: x_vals -- The known x values.
        param: y_vals -- The known y values.
        param: x -- The point to linear interpolate using the known values

    Out:
        return: The interpolated y value for x.
    """


    # Number of x values.
    values = len(x_vals)

    # The array that contains the initial polynomial values.
    initials = np.zeros(values)


    # Combine the evaluations of order 'order' to approximate
    # the higher order polynomial. Store the results in the
    # array with the initial values.
    for order in range(0, values):

        # Each order had one less element. Combine them to approximate
        # the higher order polynomial.
        for element_idx in range(values - order):
            # The very first time set the initial values.
            if order == 0:
                initials[element_idx] = y_vals[element_idx]
            else:
                top = (x_vals[order+ element_idx] - x)* initials[element_idx]
                top += (x - x_vals[element_idx])*initials[element_idx+1]
                bottom = x_vals[element_idx+order] - x_vals[element_idx]

                initials[element_idx] = top/bottom

    return initials[0]
```

./code/mathlib/interpolate.py

Output - Figure

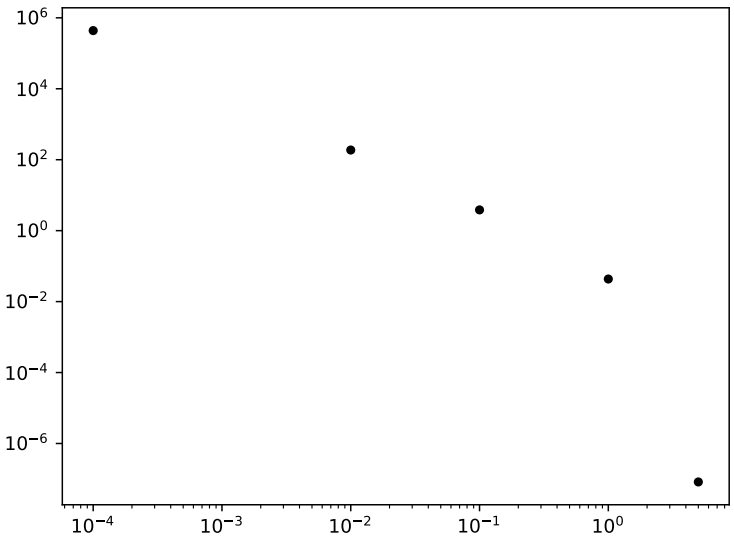

Figure 3: A logarithmic plot of the 5 points that need to be interpolated.

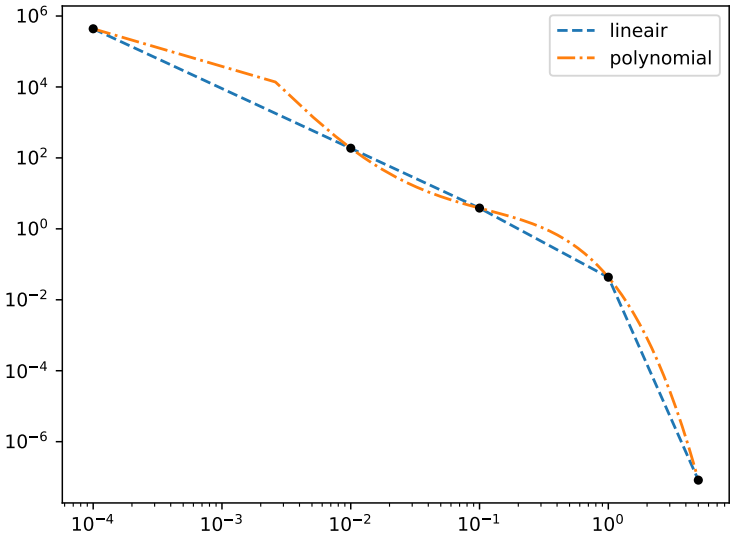

Figure 4: The interpolation of the linear interpolator and the polynomial interpolator.

Assignment 2.c

Question

Numerically calculate $\mathrm{d}n(x)/\mathrm{dx}$ at $x = b$. Output the value found alongside the analytical result, both to at least 12 significant digits. Choose your algorithm such as to get them as close as possible.

Solution

The the volume number density for satellite galaxies is,

$$n(x) = A\langle N_{sat}\rangle \left(\frac{x}{b}\right)^{a-3} \exp\left(-\left(\frac{x}{b}\right)^{c}\right) \tag{4}$$

Calculating the analytical derivative gives,

$$\begin{aligned}\frac{dn(x)}{dx} &= A\langle N_{sat}\rangle \frac{d}{dx}\left[\left(\frac{x}{b}\right)^{a-3} \exp\left(-\left(\frac{x}{b}\right)^{c}\right)\right] \\ &= A\langle N_{sat}\rangle \left((a-3)\left(\frac{x}{b}\right)^{a-4}\frac{1}{b}\exp\left(-\left(\frac{x}{b}\right)^{c}\right) - \left(\frac{x}{b}\right)^{a-3}\exp\left(-\left(\frac{x}{b}\right)^{c}\right) c\left(\frac{x}{b}\right)^{c-1}\frac{1}{b}\right)\end{aligned}$$

For $x = b$ this yields,

$$\left.\frac{dy}{dx}\right|_{x=b} = A\langle N_{sat}\rangle \left((a-3)e^{-1}\frac{1}{b} - e^{-1}\frac{c}{b}\right) = \frac{A\langle N_{sat}\rangle}{be}\left((a-3)-c\right) \tag{5}$$

The code that calculates the analytical and numerical result consists of two files: ./code/assignment2_c.py and ./code/mathlib/derivative.py. The first file calculates the analytical and numerical results and outputs these results with 12 significant digits. The second file contains the code for Ridders method, which is used to make the numerical calculation. The code of both files and the output can be found below.

Code - output

The code that produces the output.

```python
import mathlib.derivative
import numpy as np

# The values of abc found in assigment 2.1.
a = 1.316496837562065814
b = 1.305096297363074420
c = 1.790962180074062715
A = 0.051464524931584120

# The analytical derivative and the density.
analytical = (A*100*((a-3) -c))/(b*np.exp(1))
n = lambda x: (x/b)**(a-3)* np.exp(-(x/b)**c)*A*100

# Print the results.
print('Analytical: {0:.12f}'.format(analytical))
print('Numeric: {0:.12f}'.format(mathlib.derivative.ridder(n,b,1e-13)))
```

./code/assignment2_c.py

Code - ridder

The code for ridder's method

```
import numpy as np

def central_diff(function, x, h):
    """
        Use the central difference method to approximate the derivative
        at a point x.

    In:
        param: function -- The function to calculate the derivative of.
        param: x -- The point to approximate the derivative of the function at.
        param: h -- A small value h that in the limit would go to zero.
    Out:
        return: An approximation of the derivative of the provided
                function at x.
    """

    #f'(x) approx (f(x+h)-f(x-h))/2h
    return (function(x+h) - function(x-h))/(2*h)


def ridder(function, x , precision):
    """
        Perform ridders differential method to estimate
        the derivative at a point x

    In:
        param: function -- The function to estimate the derivative for.
        param: x -- The point to estimate the derivative at.
        param: precision -- The precision that must be obtained
                            in the estimation.
    Out:
        return: An approximation of the derivative.
    """

    # The number of initial approximations to use.
    approximations = 0
    # The return value.
    ret = 0
    # The current precision.
    current_precision = 0xFFFFFFFFFFF

    # While we didn't reach the request precision with
    # the current amount of initial approximations, try again
    # but with more approximations.
    while current_precision > precision:
        # Increase the amount of initial approximations.
        approximations += 5
        # Reset the error.
        current_precision = 0xFFFFFFFFFF


        # The array to store the combined results in.
        results = np.zeros(approximations)

        # The current combination:
        # 0 = combine initial central_difference evaluations,
        # 1 = combine the combined central difference evaluations
        # 2 = combine the combined combined central difference evaluations etc.
        for combination in range(0, approximations-1):

            # Combine for the current 'combination'.
             for j in range(1, approximations-combination):

                # Create the initial central difference to combine.
                if combination == 0:
                    # We need two central difference's to
                    # combine the very first time.
```

```
                    if j == 1:
                        results[j-1] = central_diff(function, x,1)

                    # Decrease h by a factor of 2 for each next approximation.
                    results[j] = central_diff(function, x, (1/2)**j)

                # Keep the evaluation of the previous combination that
                # is getting overwritten temporary in memory to update
                # the current precision.
                previous = results[j-1]

                # Combine
                power = 4**(combination+1)
                results[j-1] = (power*results[j] - results[j-1])/(power-1)

                # Determine the new precision
                precision_tmp = max(abs(results[j-1] - previous), \
                                    abs(results[j-1] - results[j]))

                # New precision is smaller -> update
                if precision_tmp < current_precision:
                    current_precision = precision_tmp
                    ret = results[j-1]

                    # Terminate if requested precision is reached
                    if current_precision < precision:
                        return ret

                # Abort early if the error of the last combined result is worse
                # than the previous order by a large amount.
                if j == (approximations-combination-1) \
                    and abs(ret - previous) > 100*current_precision:

                    return ret
    return ret
```

./code/mathlib/derivative.py

Output - Text

The output from ./code/assignment2_ c.py with 12 significant digits.

```
Analytical: -5.040329317958
Numeric: -5.040329317958
```

./output/assignment2_c_out.txt

Question 2.d

Question

Now we want to generate 3D satellite positions such that they statistically follow the satellite profile in equation (2) of the assignment; that is, the probability distribution of the (relative) radii $x \in [0, x_{max})$ should be $p(x)dx = n(x)4\pi x^2 dx/\langle N_{sat}\rangle$ (with the same a,b and c as before). Use one of the methods discussed in class to sample this distribution. Additionally, for each galaxy generate random angles ϕ and θ such that the resulting positions are (statistically) uniformly distributed as a function of direction. Output the positions (r, ϕ, θ) for 100 such satellites.

Solution

Let v be a uniform random variable between 0 and 1. A uniform distributed ϕ and θ on a sphere are then given by,

$$\phi = 2\pi v \tag{6}$$
$$\theta = \arccos(1 - 2v) \tag{7}$$

These equation can be derived from a uniform distribution for the solid angle. This derivation is not asked in the question and the equations are therefore directly taken from the first slide of the 5th lecture.

The radius x is sampled from the given probability distribution $p(x) = n(x)4\pi x^2/\langle N_{sat}\rangle$ with the help of rejection sampling. To apply rejection sampling first a distribution $p_{enc}(x)$ is chosen that encloses all of $p(x)$ when the distribution is multiplied with a scalar k (i.e $p(x) < kp_{enc}(x)\ \forall\ x \in [0, 5)$). The chosen distribution is an uniform distribution for $x \in [0, 5)$,

$$p_{enc}(x) = \begin{cases} \frac{1}{5} & \text{for } 0 < x < 5 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

The scalar is chosen to be $k = 2$. To apply rejection sampling it must first be possible to generate samples from p_{enc}. To sample from this distribution the fundamental transformation law of probability is applied. Let y be a uniform random variable between 0 and 1. Let z be distributed according to p_{enc}. The fundamental transformation law of probabilities then yields,

$$\int_0^z p_{enc}(z')dz' = \int_0^y 1dy' \tag{9}$$

Solving this for z gives the transformation that is needed to let Y be distributed according to p_{enc},

$$z = 5y \tag{10}$$

Samples distributed according to $p(x)$ can now be found with rejection sampling: Generate a random uniform variable y between 0 and 1. Transform this variable to obtain a random variable z distributed according to p_{enc}. If a new random variable w generated[3] from an uniform distribution between 0 and $kp_{enc}(z)$ is smaller or equal than $p(z)$, then z is a sample from the distribution of which we want to sample. If the condition does not hold reject the sample and repeat the above process.

The code used for the sample generation and the printing of the results consists of two files. The first file is the file that prints the results: ./code/assignment2.d.py. The second file, ./code/mathlib/rng.py, contains the functions rejection_ sample and gen_ uniform_ spherical_ surface_ coords) that are used to obtain the result. **The code for the second file can be found on pages 6 and 7**. The code for the first file and its output can be found on the next page.

[3]Let u be an uniform variable between 0 and 1, then w is given by $w = k * p_{enc}(z) * u$. This can be derived on exactly the same way as equation 10.

Code - output

The code that generates the output.

```
import mathlib.rng
import mathlib.derivative
import numpy as np

# The values of abc found in assigment 2.1.
a = 1.316496837562065814
b = 1.305096297363074420
c = 1.790962180074062715
A = 0.051464524931584120

# Initialize with the seed at the end of assigment 2.1.
rng = mathlib.rng.RNG(499869219)

# The distribution from which we want to sample.
sample_dist = lambda x : (b**2  * (x/b)**(a-1)* np.exp(-(x/b)**c) )*A*4*np.pi
# The enclosing distribution.
enclosing_dist = lambda x : 2
# The transformation for a uniform variable to the enclosing distribution.
enclosed_transform = lambda x : 5*x

# Generate 100 uniform distributed angles.
theta, phi = rng.gen_uniform_spherical_surface_coords(100)
radii = rng.rejection_sampling(sample_dist, \
                               enclosing_dist, enclosed_transform, 100)

# Display the final seed used for the next assigment,
print('Final seed: ', rng.get_seed())
# Empty line to seperate seed from the remaining output
print('')

# Print the results
print('x, phi, theta')
for pair in zip(radii,phi, theta):
    print(pair)
```

./code/assignment2_d.py

Output - text

The generated output.

```
Final seed:  1016249662

x, phi, theta
(0.3579938284489312, 3.6257715658348064, 1.7255319984510258)
(2.485454148260284, 3.1390020926533575, 1.5699717255446173)
(0.4541655596006116, 5.843930975811292, 2.6064208102306603)
(1.5978706294200082, 0.45178480110985847, 0.5429419074656698)
(1.1149836438975724, 3.9722785250354633, 1.8383941540875628)
(0.7093001752414974, 2.0975005512454707, 1.2320076806321651)
(0.5454029505013961, 0.4179331781088591, 0.5217110205747997)
(0.4163416476026973, 1.3657866067437219, 0.9700520190115742)
(0.7441318397745798, 3.946953265513712, 1.8300448344712918)
(0.05379230134510256, 1.8290937441735347, 1.1397943248306432)
(0.8180404584431183, 1.882185879343575, 1.1583173446872264)
(0.476188993890813 76, 0.04835423943941989, 0.17567742622176938)
(1.171271449693309, 0.47408060452189765, 0.556525770562289)
(0.8240872402265871, 5.808786488717012, 2.584875161184929)
(0.7075331396673651, 6.2752330512715, 3.070425943784541)
(0.3125357989483829, 2.249217286617513, 1.2827788731358873)
(0.7580733079831287, 2.420777266647821, 1.3392912948857487)
(0.6215506723666449, 3.8517906521695933, 1.798830519836583)
(1.4673194397397618, 3.980560841594835, 1.8411288191929622)
(0.701170441624981, 4.186329245957005, 1.9098024926329265)
(1.1499401918030203, 4.919064339462517, 2.1721834961232105)
(0.2875093452836176, 3.5907494313570334, 1.714258976950838)
```

```
(0.7690208116008483, 0.7678605758310203, 0.7142535941495242)
(0.5020575633975811, 6.247139421166786, 2.9899630360509617)
(0.3804066114547678, 0.7348390476584241, 0.698055430681658)
(0.34301911488711345, 2.9687685565379285, 1.5157569235344739)
(1.3005203430774903, 1.768384719750567, 1.1184182204321294)
(1.049710961768802, 1.9112587780332704, 1.168396684520003)
(0.6364001498176717, 0.039911573892260445, 0.15956966944064624)
(2.527872090583637, 0.34375757729436734, 0.4721806895663418)
(0.9569772987060661, 3.5388058712071633, 1.697572543733482)
(1.1384866691982576, 4.040522041313443, 1.8609903185074965)
(0.6510213833886714, 2.08623940431945, 1.228204556736262)
(0.7863035206185429, 3.1807348693550206, 1.583256003418326)
(1.377572160302096, 5.305222275643324, 2.3304973411227587)
(0.21021116762659772, 2.261984089311919, 1.2870146113333585)
(0.6828774408630276, 4.4465089557754025, 1.9991433778357142)
(1.8850988608051789, 4.9931440474598086, 2.201071537497057)
(1.1286425662992157, 3.996343891711613, 1.8463458483513429)
(0.9872137408208135, 5.532870662119667, 2.435907593130786)
(0.32931922593370994, 3.6043717631735754, 1.718641509029965)
(1.389728403042473, 0.41313362184410113, 0.518637276661106)
(0.5898687221551939, 1.4601768461551858, 1.0060384873238106)
(0.7752025001158944, 4.105047287966843, 1.8824962985214837)
(2.6754527568527156, 0.3027782589945894, 0.44264313923278487)
(0.09967927474055423, 1.2041681001094282, 0.9062508601035965)
(1.8776655515836704, 2.180716018803088, 1.259958381951315)
(0.08876249079796544, 5.051589061325694, 2.224298534183561)
(1.0556129601447874, 0.011354697611375811, 0.08504694955548202)
(1.0668362633015114, 3.8358833637349052, 1.7936356181624407)
(0.19937052861772722, 0.996492252454145, 0.819198961181742)
(0.1681643561851616, 4.464005243884389, 2.0052743745216035)
(0.08704470356159022, 0.9120842916126898, 0.781760160197898)
(0.5375231698941261, 4.467576446354424, 2.006527921835051)
(0.3597077157720243, 4.625436532114527, 2.062719818095845)
(1.0923313864256095, 0.6592134682747388, 0.6597167976591414)
(1.1161449938351626, 2.8028816806565002, 1.462771300113757)
(1.278946119425573, 3.3990114690680735, 1.6528272483672306)
(1.1000187325058548, 4.662921385312137, 2.076306622905424)
(0.4796606815605565, 3.194492323599985, 1.5876356105524145)
(0.10401859649084941, 3.3699955027612822, 1.6435634122630045)
(0.09232803715679982, 5.908902602658324, 2.648476669835251)
(1.170586959032944, 2.131435513763629, 1.2434377483666312)
(0.4094679002672126, 1.0260995916912727, 0.8320219032742583)
(2.311631084259514, 3.5246704486696423, 1.693037992133638)
(1.157567722526744, 3.024518313540633, 1.5335217760438127)
(2.1493239822213828, 4.410109312764263, 1.9864427443966763)
(0.17980300476304328, 6.278362910374851, 3.0861776835631125)
(0.27707741718671225, 4.974510684329675, 2.193749231978967)
(0.1325431536260394, 3.945136900574399, 1.8294467263330656)
(2.1336953358570336, 6.154539503826982, 2.8544279035598596)
(1.4742452864242 36, 1.3259100902203211, 0.9545822652253781)
(0.3535610461499451, 2.1424972298453664, 1.2471539381354246)
(1.8689941921897684, 1.157530892008798, 0.8872501518212352)
(0.496305306557637, 4.944879537563 72, 2.1821836715333656)
(0.2796433820108984, 4.697434177065051, 2.088907092417107)
(0.8668761772259316, 6.196403675441801, 2.9060015995359376)
(0.13309590987234746, 1.1594475926475194, 0.8880367886252412)
(0.6785562473066514, 1.686372296041978, 1.0891813754537536)
(0.829390833580259, 1.5189879845288539, 1.028048237297669)
(0.024451633222506295, 0.9413026277661871, 0.7948748960867421)
(1.039777572974511, 1.6898077909163576, 1.0904148900447959)
(0.42673284896340524, 1.9779474771807983, 1.1913569678117872)
(1.5269727216863478, 1.847337716546747, 1.146176782855883)
(1.5051165855268753, 0.7103473076563005, 0.6858366829840647)
(0.4047520471282192, 2.8637378863416214, 1.4822366934654974)
(0.5281468051318421, 6.071337982267417, 2.7722556824268483)
(1.2742317622234653, 4.932325124241948, 2.1773117347319153)
(2.86956520468685, 2.0464460927247448, 1.214723628186573)
(0.5294553168419411, 1.4938966821809838, 1.0186945174290796)
(0.40818834104765866, 3.0396934026989695, 1.5383550977692144)
(0.27314911137175496, 2.5331241525441537, 1.3758829636869725)
```

```
(1.5413572805797116, 3.693748172709161, 1.7474705658625795)
(0.5962656113776065, 3.90974897806, 1.8178124555097586)
(0.013182851256146759, 2.372614420780756, 1.323510377583853)
(1.823761988623012, 0.09078962497343887, 0.24099590143646527)
(0.9896303575927462, 5.68253479322631, 2.5129169524463126)
(1.0666621141290902, 0.3479916674450603, 0.475135351696596)
(1.9487270496200602, 3.2649587351224194, 1.6100750693898447)
(0.35688853947373306, 1.2684440216345219, 0.9319848282357273)
```

./output/assignment2_d_out.txt

Assignment 2.e

Question

Repeat the previous exercise for 1000 haloes each containing 100 satellites. Make another log-log plot showing $N(x) = n(x)4\pi x^2$ over the same range as before, but now over-plot a histogram showing the average number of satellites in each bin. Use 20 logarithmically-spaced bins between $x = 10^{-4}$ and x_{max} and don't forget to divide each bin by its width. Do you generated galaxies match this distribution?

Solution

The histogram with the average number of satellites is created by first creating a histogram for each halo and adding each of these histograms together. The obtained "super" histogram is next divided by the width of the bins to correct for the different sized bins. The bin width corrected "super" histogram is finally divided by the number of halos to obtain the histogram with the average number of satellites.

The code that generates the radii and the histogram is located in the file: ./code/assignment2_e.py. The content of this file and its output can be found below[4].

Code - plots

```
import mathlib.rng
import matplotlib.pyplot as plt
import numpy as np

# The values of abc found in assigment 2.1.
a = 1.316496837562065814
b = 1.305096297363074420
c = 1.790962180074062715
A = 0.051464524931584120

# The amount of halo's to generate.
haloes = 1000

# Initialize with the seed at the end of assigment 2.4
# to keep the usage of the random number generator contineus
# through out the full assigment (see report question 1.b).
rng = mathlib.rng.RNG(1016249662)



# The functions from assigment 2.4 and the new function to plot (last one).
sample_dist = lambda x : (b**2  * (x/b)**(a-1)* np.exp(-(x/b)**c) )*A*4*np.pi
enclosing_func = lambda x : 2
enclosed_transform = lambda x : 5*x
N = lambda x : sample_dist(x)*100   #N(x) = p(x)*<N_{sat}> (plot function)

# Create an array with the bins.
bins = np.logspace(-4, np.log10(5), 21)

# Create a matrix that stores all radii for each iteration.
```

[4]The code of the helper functions used in this file can be found in the previous assignment.

```
# This matrix is not used in this assigment, but is needed
# for assigment 2.g.
radi_all = np.zeros((haloes, 100))

#
# Generate the samples
#

# The first halo is generated outside a loop to obtain the bin-widths,
# bin-centers and the initial histogram. This histogram is used to
# create a combined histogram by adding the histograms for each halo
# to it. The final histogram is then obtained by dividing this combined
# histogram by the bin width and the number of haloes.

# Radii of first halo.
radi = rng.rejection_sampling(sample_dist, \
                              enclosing_func, enclosed_transform, 100)
histogram, bin_edges = np.histogram(radi, bins=bins)
bin_widths = bins[1:] - bins[:-1]
bin_centers = (bins[0:-1] + bins[1:])/2

# Remaining haloes are created in a loop
for i in range(1, haloes):
    radi = rng.rejection_sampling(sample_dist, \
                                  enclosing_func, enclosed_transform, 100)
    histogram += np.histogram(radi, bins=bins)[0] # add histograms together
    radi_all[i] = radi

# Divide by the number of haloes to obtain the average and
# divide by bin width to correct for different sized bins.
histogram = np.array(histogram)
histogram_normed = histogram/ (haloes*bin_widths)

# Plot the histogram and the function.
x_plot = np.linspace(0.002, 5, haloes) #x values to plot N(x) for
plt.bar(bin_centers, histogram_normed, width=bin_widths, color='orange',
    edgecolor='black')
plt.plot(x_plot, N(x_plot), c='red')
plt.loglog()
plt.xlim(1e-3, 10)
plt.savefig('./plots/assignment2e.pdf')

# Save the generated radii for assigment 2.g.
np.save('radii_2e', radi_all)

# No need to print the final seed as the random number generator is
# not needed for any other assigment.
```

./code/assignment2_e.py

Output - plot

The output generated by ./code/assignment2_ e.py consists of figure 5 on the next page.

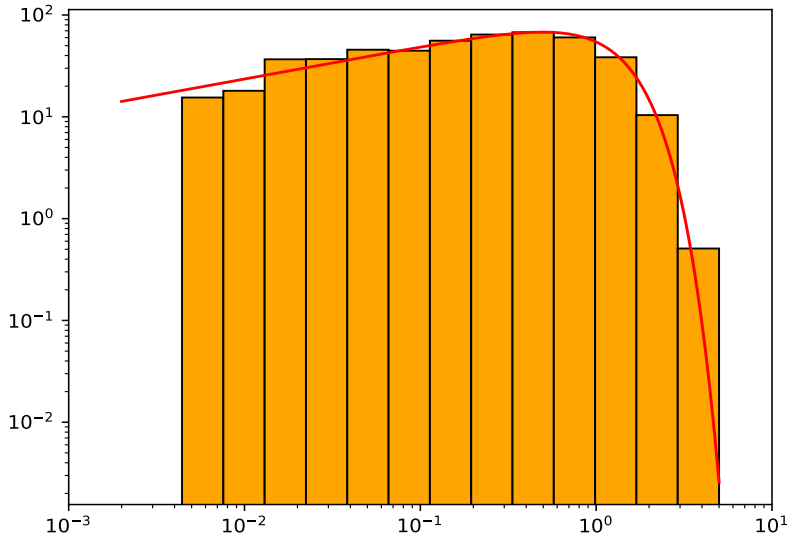

Figure 5: The average number of satellites on a spherical shell at distance $x = r/r_{viral}$. The red line is the original function $N(x)$ that is used to generate the histogram.

Assignment 2.f

Question

Write a root-finding algorithm to find the solution(s) to $N(x) = y/2$ in the same x-range, where y is the maximum of $N(x)$. Use the same parameter values as before. Output the root(s).

Solution

The x-cöordinate of the maximum is determined analytically by taking the derivative of $N(x)$,

$$\begin{aligned}
\frac{dN(x)}{dx} &= 4\pi A\langle N_{sat}\rangle \frac{d}{dx}\left[b^2\left(\frac{x}{b}\right)^{a-1}\exp\left(-\left(\frac{x}{b}\right)^c\right)\right] \\
&= 4\pi A\langle N_{sat}\rangle \left(b^2(a-1)\left(\frac{x}{b}\right)^{a-2}\frac{1}{b}\exp\left(-\left(\frac{x}{b}\right)^c\right) - b^2\left(\frac{x}{b}\right)^{a-1}\exp\left(-\left(\frac{x}{b}\right)^c\right)c\left(\frac{x}{b}\right)^{c-1}\frac{1}{b}\right) \\
&= 4\pi\langle N_{sat}\rangle b\exp\left(-\left(\frac{x}{b}\right)^c\right)\left((a-1)\left(\frac{x}{b}\right)^{a-2} - c\left(\frac{x}{b}\right)^{a-1}\left(\frac{x}{b}\right)^{c-1}\right)
\end{aligned}$$

Equating the derivative to zero gives as solutions that,

$$4\pi\langle N_{sat}\rangle b\exp\left(-\left(\frac{x}{b}\right)^c\right) = 0 \quad \text{or} \quad \left((a-1)\left(\frac{x}{b}\right)^{a-2} - c\left(\frac{x}{b}\right)^{a-1}\left(\frac{x}{b}\right)^{c-1}\right) = 0 \qquad (11)$$

The equation on the left is not solvable. Taking the equation on the right and solving it for x gives,

$$\begin{aligned}
(a-1)\left(\frac{x}{b}\right)^{a-2} &= c\left(\frac{x}{b}\right)^{a-1}\left(\frac{x}{b}\right)^{c-1} \\
(a-1)\left(\frac{x}{b}\right)^{a-2} &= c\left(\frac{x}{b}\right)^{a-2}\left(\frac{x}{b}\right)^{c} \\
(a-1) &= c\left(\frac{x}{b}\right)^{c} \\
x_{max} &= \left(\frac{a-1}{c}\right)^{1/c} b
\end{aligned}$$

The maximum value, y, is now given by,

$$y = N(x_{max}) \tag{12}$$

The roots of $N(x) - y/2 = 0$ are found with the Newton-Raphson method. The code that applies the Newton-Raphson method and prints the root is located in the file: ./code/assignment2_ f.py. The implementation of the Newton-Raphson method is found in ./mathlib/roots.py. The content of these files and the output of the first file can be found below.

Code - output

The code that find the roots and prints them.

```
import mathlib.roots as roots
import numpy as np


# The values of abc found in assigment 2.1.
a = 1.316496837562065814
b = 1.305096297363074420
c = 1.790962180074062715
A = 0.051464524931584120


# The maximum x and y coordinate,
x_max = b*((a-1)/c)**(1/c)
y_max = (b**2  * (x_max/b)**(a-1)* np.exp(-(x_max/b)**c) )*A*4*np.pi*100


# The function to put equal to zero and solve.
root_func = lambda x: (b**2 *(x/b)**(a-1)* np.exp(-(x/b)**c))*A*4*np.pi*100 -
    y_max/2
# The derivative of this function, needed for newton_raphson.
root_der = lambda x: 4*np.pi*100*b*np.exp(-(x/b)**c)*((a-1)*(x/b)**(a-2)-c*(x/b
    )**(a-1)*(x/b)**(c-1))

# Print the roots
print('Left root: {0:.7f}'
      .format(roots.newton_raphson(root_func, root_der, 0, 0.3, 1e-10)))
print('Right root: {0:.7f}'
      .format(roots.newton_raphson(root_func, root_der, 1, 2, 1e-10)))
```

./code/assignment2_f.py

Code - Newton- Raphson

The code for the Newton-Raphson method.

```
import numpy as np



def newton_raphson(func, func_deriv, x_left, x_right, xacc):
    """
        Calculate the roots using the Newton-Raphson method

    In:
        param: func -- The function to calculate the roots for.
        param: func_deriv -- The derivative of the function to
                             calculate the roots for.
        param: x_left -- The left bracket value that encloses the root.
        param: x_right -- The right bracket value that encloses the root
        param: xacc -- The accuracy to find the root with.

    Out:
        return: An approximation of the x coordinate of the root.
    """

```

```
21
22      # The initial accuracy and the maximum number
23      # of iterations.
24      acc = 0xFFFFFFFFF
25      iters = 1e4
26
27      # Make sure the left bracket value is the smallest.
28      if x_left > x_right:
29          raise "Left bracked value is larger than right bracked value"
30
31      # Start from the center of the bracket
32      result = 0.5*(x_left+x_right)
33
34      # While the requested accuracy isn't reached and
35      # while the maximum amount of iterations isn't reached,
36      # perform Newton_Raphson
37      while (acc > xacc) and (iters > 0):
38
39          # Keep the previous result temporary in memory
40          # to update the accuracy.
41          old = result
42
43          #x_i+1 = x_i - func(x_i)/func_derv(x_i)
44          result -= func(result)/func_deriv(result)
45
46          # Check if we are still in the bracket
47          if result < x_left or result > x_right:
48              raise "Out of bracket"
49
50          # Update the accuracy
51          acc = abs(abs(result) - abs(old))
52
53      return result
```

./code/mathlib/roots.py

Output - text

The output of ./code/assigment2_ f.py

```
1 Left root: 0.0318795
2 Right root: 1.4515403
```

./output/assignment2_f_out.txt

Assignment 2.g

Question

Take the radial bin from assignment 2.e containing the largest number of galaxies. Using sorting, calculate the median, 16th and 84th percentile for this bin and output these values. Next, make a histogram of the number of galaxies in this radial bin in each halo (so 1000 values), each bin of this histogram should have a width of 1. Plot this histogram, and overplot the Poisson distribution with λ equal to the mean number of galaxies in this radial bin.

Solution

The code does for this exercise consists of three files. The first file (./code/assignment2_ g.py) contains the code that creates the plot and prints the output. The second file (./code/-mathlib/sorting.py) contains the code for merge sorting. The last file (./code/mathlib/utils.py) contains the functions for the median, percentile and Poisson distribution. A part of this last file has earlier been shown in assignment 1.b. The full file, inclusive the earlier shown part, can be found below.

Code - output

The code that produces the output.

```
import mathlib.utils as utils
import matplotlib.pyplot as plt
import numpy as np


# Read in the radii from assigment 2.e.
radii_matrix = np.load('radii_2e.npy')

#
# Create the histograms for each of the 100 generated radii (see assigment 2.e)
# and find the bin with the maximal number of counts over the full dataset
# (i.e the maximum within all the histograms).
#

# The array in which the histograms are stored.
histograms = list()
# The bins for each histogram.
bins = np.logspace(-4, np.log10(5), 21)
# The histogram with the maximum bin.
max_hist_idx = 0
# The bin with the maximum number of counts.
max_bin_idx = 0


for idx in range(radii_matrix.shape[0]):

    # Create the histogram and save it.
    histogram = np.histogram(radii_matrix[idx], bins = bins)[0]
    histograms.append(histogram)

    # Loop over all bins in this new histogram.
    for bin_idx, bin_counts in enumerate(histogram):

        # Check if a bin in the new histogram contains
        # more counts than the known maximum and save the bin index and the
        # histogram index if this is true.
        if histograms[max_hist_idx][max_bin_idx] < bin_counts:
            max_hist_idx = idx
            max_bin_idx = bin_idx

# For each histogram take the bin with the index of max_bin_idx
# and save the counts in an array.
bin_counts = np.array([hist[max_bin_idx] for hist in histograms])
bin_counts =  bin_counts

# Print the median, 16th percentile and 84th percentile for the counts.
print('Median: {0:.1f}'.format(utils.median(bin_counts)))
print('Percentile 16th: {0:.1f}'.format(utils.percentile(bin_counts, 16)))
print('Percentile 84th: {0:.1f}'.format(utils.percentile(bin_counts, 84)))

# Plot the histogram for the counts with bin width of 1.
plt.hist(bin_counts, density=True,
         bins=np.arange(min(bin_counts), max(bin_counts) + 1, 1),
         edgecolor='black')

#
# Overplot the Poisson distribution.
#

# Mean events
mean = int(sum(bin_counts)/len(bin_counts))
x = np.arange(1, 60, 1)

# Poisson function can't take arrays :(
y = [utils.poisson(mean, float(x_val)) for x_val in x]
plt.plot(x, y, label='Poisson(mean:{0:.1f})'.format(mean))
plt.legend()
```

```
68
69 # Save the plot
70 plt.savefig('./plots/poisson.pdf')
```

./code/assignment2_g.py

Code - merge sort

The code for merge sort that is used for the median and the percentile.

```
1  import numpy as np
2
3  def merge_sort(array):
4      """
5          Sort an array using merge sort
6
7      In:
8          param: array -- The array to sort.
9      Out:
10         return: The sorted array.
11     """
12
13     # Get the length of the array.
14     size = len(array)
15
16     # If the length is 1 then  return the input array.
17     # This is an important check as this function is called
18     # recursively.
19     if size == 1:
20         return array
21
22     # Split the array in an sorted left and right segment.
23     left_sorted = merge_sort(array[:size >> 1])
24     right_sorted = merge_sort(array[size >> 1:])
25
26     #
27     # Merge the left and right array.
28     #
29
30     # The final sorted array.
31     result = np.zeros(size)
32
33     # Current index in the left sorted array.
34     left_idx = 0
35     # Current index in the right sorted array.
36     right_idx = 0
37     # Current index in the final sorted array.
38     result_idx = 0
39
40     # While we didn't fill the result array.
41     while result_idx < size:
42
43         # Element from left array is smaller, insert it and increase position.
44         if left_sorted[left_idx] < right_sorted[right_idx]:
45             result[result_idx] = left_sorted[left_idx]
46             left_idx += 1
47             result_idx += 1
48         # Element from right array is smaller, insert it and increase position.
49         else:
50             result[result_idx] = right_sorted[right_idx]
51             right_idx += 1
52             result_idx += 1
53
54         # Only right arrat has elements left, insert the remaining elements.
55         if left_idx == len(left_sorted):
56             result[result_idx:] = right_sorted[right_idx:]
57             break
58
59         # Only left has elements left, insert the remaining elements
60         if right_idx == len(right_sorted):
```

```
            result[result_idx:] = left_sorted[left_idx:]
            break

    return result
```

./code/mathlib/sorting.py

Code - Poisson, median, percentile The code of the Poisson distribution, median function and percentile function.

```
import mathlib.sorting
import numpy as np

def factorial(number):
    """
        Calculate the factorial for a number

    In:
        param: number -- The number to calculate the factorial for.
    Out:
        return: The factorial of the input number.
    """

    # If number < 20 convert it to an int64 to not lose any precision.
    # else use a np.float64. ( = lose in precision)
    # Note: Only datatypes with 64 or less bits are allowed.
    if number < 20:
        number = np.int64(number)
    else:
        number = np.float64(number)

    if number < 0:
        raise RuntimeError('Invalid input for factorial: {n}. \
                            Input must be larger or equal than zero.')
    elif np.int64(number) == 0: #!0 = 1
        return 1
    else:
        return number*factorial(number-1)


def poisson(average, events):
    """
        Evaluate the Poisson distribution.

    In:
        param: average -- The average number of events.
        param: events -- The number of observed events.

    Out:
        return: The evaluation of the Poisson distribution
                for the given parameters.
    """

    return (average**events * np.exp(-average))/factorial(events)


def median(array):
    """
        Determine the median of an array

    In:
        param: array -- The array to determine the median of.
    Out:
        return: The median of the array.
    """

    # Create a sorted version of the array.
    sort = mathlib.sorting.merge_sort(array)
    size = len(sort)
```

```
60      mid = size >> 1
61
62      if size % 2 == 0: # even
63          # An even array doesn't have an exact mid value. We
64          # therefore give the average of the values in the mid back.
65          return (sort[mid] + sort[mid- 1])/2
66      else: # odd
67          return sort[mid] # Give the value in the mid
68
69  def percentile(array, value):
70      """
71          Determine the 'value' percentile of the array
72          using linear interpolation.
73      In:
74          param: array -- The array to find the 'value' percentile for
75          param: value -- A value between 0 and 1 representing the
76                          wanted percentile.
77      Out:
78          return: The 'value' percentile of the array.
79
80      """
81
82      # Sort the array and determine the amount of elements.
83      sort = mathlib.sorting.merge_sort(array)
84      elements = len(array) - 1
85
86      # The index in the array that corresponds with the percentile.
87      # This is a decimal (i.e 1.2345)
88      idx_unrounded = (value*elements/100)
89
90      # The rounded index (i.e 1)
91      idx_rounded = int(idx_unrounded)
92
93      # The decimals of the unrounded index.
94      # This indicates the place between index 'indx_rounded'
95      # and 'idx_rounded+1' at which the interpolated value
96      # should be determined to obtain the percentile.
97      idx_decimals = idx_unrounded - idx_rounded
98
99      # Interpolation fails for 100th percentile, as
100     # idx_rounded+1 doesn't exists.
101     if value == 100:
102         return sort[elements]
103
104     # Determine the interpolated value.
105     return idx_decimals*(sort[idx_rounded+1] - sort[idx_rounded]) + sort[
        idx_rounded]
```

./code/mathlib/utils.py

Output - text

The text output from ./code/assignment2_ g.py

```
1 Median: 27.0
2 Percentile 16th: 23.0
3 Percentile 84th: 31.0
```

./output/assignment2_g_out.txt

Output - figure

The plot created in ./code/assignment2_ g.py

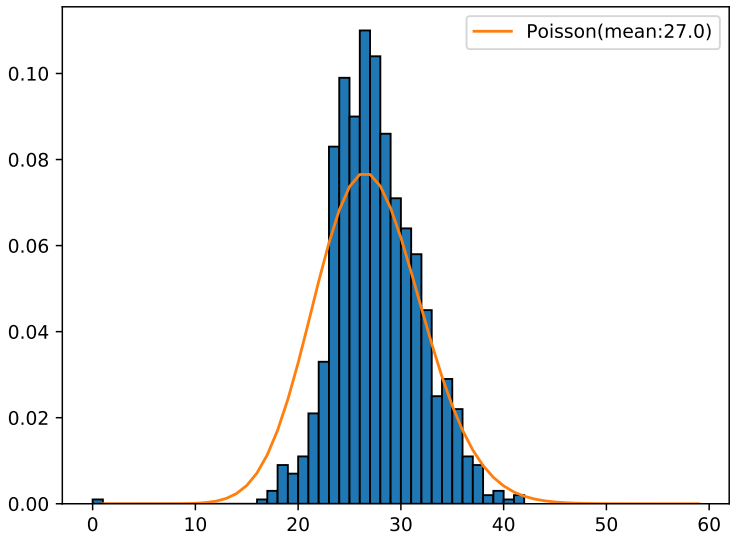

Figure 6: The Poisson distribution over-plotting the created histogram. The figure shows that the number of satellites on a spherical shell at a distance x is approximately distributed according to a Poisson distribution around N(x) - the number of satellites predicted by the model on an infinite thin spherical shell.

Assignment 2.h

Question

The normalization factor A depends on all three parameters. Calculate A at 0.1 wide intervals in the range of a, b and c given above (including the boundaries). You should get a table containing 6240 values. Choose an interpolation scheme and write a 3D interpolator for A as a function of the three parameters based on these calculated values.

Solution

The chosen interpolator is a 3D linear interpolator. The code for the interpolator and the explanation of how it interpolates can be found in the file ./code/mathlib/interpolate3D.py. The file makes use of 1 dimensional linear interpolator of which the code can be found in assignment 2.b. The code that creates the table can be found in the file ./code/assignment2_ h.py.

Code - Table

The code that creates the table and performs a small test to see how well the interpolator approximates the true value.

```
import numpy as np
import mathlib.interpolate3D as intp3d
import mathlib.integrate as intgrt

# Values of the a, b and c parameters.
a_values = np.arange(1.1, 2.6, 0.1) #15
b_values = np.arange(0.5, 2.1, 0.1) #16
c_values = np.arange(1.5, 4.1, 0.1) #26

# Row is a , column is b , depth is c.
table = np.zeros((len(a_values), len(b_values), len(c_values)))

for row, a_val in enumerate(a_values):
    for column, b_val in enumerate(b_values):
        for depth, c_val in enumerate(c_values):

            # The function to normalize (simmilar as in assigment 2.a)
            # This function unfortunatly couldn't be split over multiple
            # lines so it might look slightly ugly in the report.
            density = lambda x: b_val**2 *(x/b_val)**(a_val-1)*np.exp(-(x/b_val
    )**c_val)*4*np.pi

            # Insert the value in the table.
            table[row][column][depth] = 1/intgrt.romberg(density, 0, 5, 15)

# Save the table to the disk to use it later in the next assigment.
np.save('table_2h', table)
```

./code/assignment2_h.py

Code - interpolator

The code for the 3D interpolator

```
import numpy as np
import mathlib.interpolate as intrp

class Interpolate3D(object):
    """
        A class representing a 3D interpolator.
    """

    def __init__(self, tensor, ranges):
        """
            Create a new instance of the 3D interpolator.

        param: tensor -- A tensor containing the known y = f(a,b,c) values
        param: ranges -- A matrix containing the a, b and c values of the
    tensor.
        """

        self._tensor = tensor

        # Unpack the a, b, c values.
        self._row_ranges = ranges[0]
        self._column_ranges = ranges[1]
        self._depth_ranges = ranges[2]


    def interpolateLin(self, a, b, c):
        """
            Estimate the point y = f(a, b, c) using linear interpolation.

        In:
            param: a    The a coordinate to interpolate for.
```

```
            param: b    The b coordinate to interpolate for.
            param: c    The c coordinate to interpolate for.
        Out:
            return: An approximation for the value y = f(a, b, c)
        """

        """
            For linear interpolating we first find the cube that encloses
            the point c_1 = f(a, b, c). (see cube below).
            This is done by finding the indices of the known (a, b, c) values
            that correspond to the left top corner of the cube (point 500).
            Knowing these indices the indices of the other corners
            can easily be found.

            To find the value c_1 we now first interpolate over the 'b' dimension
            to obtain the points x_1, x_4, x_2, x_3. These points are then used
            to interpolate over the c dimension to obtain y_1 and y_2.
            Finally we use y_1 and y_2 to obtain the asked point c_1.

            Note: You don't necessary have to interpolate the b dimension first.
                  You can start with any dimension you want.

                800----x_4-------700
                / |      /       / |
               /  |    y_1      /  |
              /   |    /|      /   |
            500------x_1-----600   |
            |     |     |       |   |
            |     |    C_1      |   |
            |    300---|x_3---|--400
            |   /      |/      |  /
            |  /       y_2     | /
            |/        /        |/
           000------x_2------100
        """

        # Indices of the left top coorner of the cube.
        a_idx = self._find_closest_smaller(a, self._row_ranges)
        b_idx = self._find_closest_smaller(b, self._column_ranges)
        c_idx = self._find_closest_smaller(c, self._depth_ranges)

        # Interpolate b dimension between
        # [500, 600] for x_1,
        # [000, 100] for x_2,
        # [800, 700] for x_4
        # and [300, 400] for x_3.
        x_1 = intrp.interpolate_linear(self._column_ranges[b_idx:b_idx+2],
                                       self._tensor[a_idx][b_idx:b_idx+2][:,c_idx],
                                       b)
        x_2 = intrp.interpolate_linear(self._column_ranges[b_idx:b_idx+2],
                                       self._tensor[a_idx+1][b_idx:b_idx+2][:,c_idx],
                                       b)
        x_3 = intrp.interpolate_linear(self._column_ranges[b_idx:b_idx+2],
                                       self._tensor[a_idx+1][b_idx:b_idx+2][:,c_idx+1],
                                       b)
        x_4 = intrp.interpolate_linear(self._column_ranges[b_idx:b_idx+2],
                                       self._tensor[a_idx][b_idx:b_idx+2][:,c_idx+1],
                                       b)


        # Next interpolate the 'c' dimension
        # between [x_1,x_4] for y_1 and [x_2,x_3] y_2.
        y_1 = intrp.interpolate_linear(self._depth_ranges[c_idx:c_idx+2],
                                       np.array([x_1,x_4]), c)
        y_2 = intrp.interpolate_linear(self._depth_ranges[c_idx:c_idx+2],
                                       np.array([x_2,x_3]), c)

        # Finally interpolate the 'a' dimension
```

```
99          # between [y_1, y_2] to obtain the final result.
100         ret = intrp.interpolate_linear(self._row_ranges[a_idx:a_idx+2],
101                                        np.array([y_1, y_2]), a)
102
103         return ret
104
105
106     def _find_closest_smaller(self, value, array):
107         """
108             Find the closest index of the known value's in the array 'array'
109             that is at the left of 'value'.
110
111         In:
112             param: value -- The value to find the closest index for.
113             param: array -- The array in which to find the closest index
114                             (must be sorted)
115
116         Out:
117             return: The closest index of the known value's in the array
118             that is at the left of 'value'.
119         """
120
121         # The index of the closest value.
122         closest_idx = 0
123         # The difference between value and the current closest value.
124         closest_diff = 0xFFFFFFFFF
125
126         # Go through each element in the array.
127         for idx, array_value in enumerate(array):
128
129             # If the difference between value and array_value is smaller,
130             # then the new closest value is found.
131             # Update the difference and the index in this case.
132             if abs(array_value - value) < closest_diff:
133                 closest_diff = abs(array_value - value)
134                 closest_idx = idx
135
136
137         # If the closest idx is zero then there is no index smaller.
138         if closest_idx == 0:
139             return closest_idx
140
141         # We want the index left. Thus, if the found value is slightly larger
142         # then return the index of the found value - 1.
143         # The factor 1e-7 is to correct for rounding errors.
144         if array[closest_idx] + 1e-7 >= value:
145             return closest_idx - 1
146         else:
147             return closest_idx
```

./code/mathlib/interpolate3D.py

Assignment 3.a

Question

Download the provided file. Each file contains haloes in a certain mass bin with variable numbers of satellites. Write down the log-likelihood corresponding to a set of random realizations of the satellite profile with some unknown $\langle N_{sat} \rangle$. Retain only the terms with a (residual) dependence on a, b and/or c (including A = A(a, b, c)).

Find the a, b and c that maximize this likelihood. Do this separately for each different file/mass bin. Output these values.

Solution

Recall that the number of satellites predicted by the model on an infinite thin spherical shell at a distance x is given by,

$$N(x) = 4\pi A \langle N_{sat} \rangle b^2 \left(\frac{x}{b}\right)^{a-1} \exp\left[-\left(\frac{x}{b}\right)^c\right] \tag{13}$$

here $\langle N_{sat} \rangle$ is the total amount of satellites in the halo and A is a normalization constant such that the integral from $x = 0$ to $x = x_{max}$ of $N(x)$ is $\langle N_{sat} \rangle$.

In exercise 2.g it was found that the number of satellites on a sphere shell at distance x is approximately distributed according to a Poisson distribution around N(x). If we assume that the distribution of satellites for haloes in the same mass bin is similar, then the probability to find the data in one of the files given the model is given by,

$$P(\text{data}|\text{model}) = \prod_{i=0}^{N-1} \frac{N(x_i|a,b,c)^{y_i} e^{-N(x_i|a,b,c)}}{y_i!} \tag{14}$$

here x_i is the distance to a satellite, y_i is the number of satellites at this distance. The optimal value's of a, b and c that maximize this probability can be found minimizing the negative log-likelihood,

$$-\ln(P(\text{data}|\text{model})) = -\sum_{i=0}^{N-1} \left[y_i \ln(N(x_i|a,b,c)) - N(x_i|a,b,c) - \ln(y_i!)\right] \tag{15}$$

The satellites in the haloes of the same mass bin are, as mentioned before, expected to have the same distribution. The data of the haloes of a single file is therefore combined to create a super halo. It is next assume that in the super halo has an infinite amount of bins. In this case y_i is either 0 or 1 and the negative log-likelihood then becomes,

$$-\ln(P(\text{data}|\text{model})) \approx \sum_{i=0}^{M-1} \left[-\ln(N(x_i|a,b,c))\right] + \int_0^{x_{max}} N(x|a,b,c)dx \tag{16}$$

here the sum is taken over the M positions of the satellites in the super halo. The integral over all radii is however equal to the total number of satellites in the super halo and the function to minimize therefore becomes,

$$-\ln(P(\text{data}|\text{model})) \approx \sum_{i=0}^{M-1} \left[-\ln(N(x_i|a,b,c))\right] + \langle N_{sat} \rangle \tag{17}$$

This function is minimized with the help of the downhill simplex method. The function is minimized withing a bounded interval: $1.1 \leq a \leq 2.5$, $0.5 \leq b \leq 2$, $1.5 \leq c \leq 4.0$. The value of x_{max} is furthermore chosen to be 5. These choices make it possible to use the interpolator of the previous question to determine the value of the normalization constant $A(a, b, c)$. The value of $\langle N_{sat} \rangle$ is for each super halo set to the total amount of satellites in the super halo.

The log likelihood function is finally slightly modified to prevent any minimization algorithm from walking outside the bounces. If a minimization algorithm tries an parameter that is outside one of the bounces, then the parameter will be set to the value of the bounce before evaluating the log likelihood (i.e if the downhill simplex method tries $a = 0.47$ then a is first set to 0.5 before evaluating the log likelihood).

The code that finds the optimum values of a, b and c is located in 3 files: ./mathlib/code/assignment3.a.py, ./mathlib/code/minimise.py, ./mathlib/code.interpolate.py. The first file finds the optimal values of a, b and c for each mass bin. The second file contains the implementation of the downhill simplex method. The last file contains the code for the interpolator. The produced output of the code and the code in the first two files can be found below. The code for the last file can be found in the previous assignment.

Code - output

```
import mathlib.utils
import mathlib.integrate
import mathlib.minimize as minimize
import mathlib.interpolate3D
import matplotlib.pyplot as plt
import numpy as np

#The 3D interpolator for the normalization constant
interpolator = None

def main():
    global interpolator

    # Load the 'table' created in the previous assigment and use it
    # to create the 3d interpolator.
    table = np.load('table_2h.npy')
    # The a, b, c ranges of the 'table'.
    ranges = [np.arange(1.1, 2.6, 0.1),
              np.arange(0.5, 2.1, 0.1),
              np.arange(1.5, 4.1, 0.1)]

    # Create the interpolator.
    interpolator = mathlib.interpolate3D.Interpolate3D(table, ranges)

    # Go through each file and find the optimum.
    for file in ['satgals_m11.txt', 'satgals_m12.txt', 'satgals_m13.txt',
                 'satgals_m14.txt', 'satgals_m15.txt']:


        # Read the file with halos and create a super halo.
        super_halo = np.loadtxt(file, skiprows=4)[:, 0]

        # The number of satelites in the super halo.
        Nsat = len(super_halo)

        # A wrapper around the likelihood function. This is used
        # function that doesn't deppand on Nsat and super halo.
        likelihood_wrapper = lambda a, b, c: likelihood(Nsat, super_halo, a, b,
    c)

        # Find the optimal values.
        a_opt, b_opt, c_opt = minimize.downhill_simplex(likelihood_wrapper,
                                                        [1.1, 1.3, 3.1],
                                                         max_iters=200)

        # Print the file and the output.
        print('File: {0} a_opt:{1:.5f} b_opt:{2:.5f} c_opt:{3:.5f}'
              .format(file, a_opt, b_opt, c_opt))


        # Plot the results  for the optimal found values.
        bins = np.logspace(-4, np.log10(5), 21)
        x = np.linspace(min(super_halo), 5, 100)
```

```
53
54          plt.plot(x, N(Nsat,x,a_opt,b_opt,c_opt)/Nsat ,
55                   c = 'red', label = 'Model')
56          plt.hist(super_halo, bins= bins, density = True,
57                   label = 'Data', edgecolor = 'black')
58
59          plt.xlim(min(super_halo)/5, max(super_halo)*30)
60          plt.ylim(bottom=10**(-20), top=10)
61          plt.legend(framealpha=1, loc= 1)
62          plt.loglog()
63          plt.savefig('./plots/' + file[0:-4] + '.pdf')
64          plt.figure()
65
66
67
68
69
70
71
72
73  def N(Nsat, x, a, b, c):
74      """
75          Calculate the number of satelites on an infinite thin shell
76          with radius x.
77
78      In:
79          param: Nsat -- The total number of satelites in the halo.
80          param: x -- The radius of the shell.
81          param: a -- The first model parameter.
82          param: b -- The second model parameter.
83          param: c -- The third model parameter.
84
85      Out:
86          return:  The number of satelites on an infinite thin shell
87                   with radius x.
88      """
89
90      global interpolator
91
92      return interpolator.interpolateLin(a,b,c)*b**2*(x/b)**(a-1)* np.exp(-(x/b)
        **c)*Nsat*4*np.pi
93
94  def likelihood(Nsat, x_values, a, b, c):
95      """
96          Calculate the negative loglikelihood for
97          the model given the total amount of satelites,
98          their x positions and the model parameters.
99
100     In:
101         param: Nsat -- The total number of satelites.
102         param: x_values -- The position of all the satelites.
103         param: a -- The first model parameter.
104         param: b -- The secon model parameter.
105         param: c -- The third model parameter.
106
107     Out:
108         return: The value of the negative loglikelihood evaluated for
109                 the given input.
110     """
111
112
113     # Only optimise within the bounded region 1 <= a <= 2.5. Values
114     # outside the bounds are put at the edge.
115     if a < 1:
116         a = 1
117     elif a > 2.5:
118         a = 2.5
119
120     # Only optimise within the bounded region 0.5 <= b <= 2.0 Values
121     # outside the bounds are put at the edge.
```

```
    if b < 0.5:
        b = 0.5
    elif b > 2.0:
        b = 2

    # Only optimise within the bounded region 1.5 <= c <= 4. Values
    # outside the bounds are put at the edge.
    if c > 4:
        c = 4
    elif c < 1.5:
        c = 1.5

    #return the negative loglikelihood.
    return - np.sum(np.log(N(Nsat, x_values, a, b, c))) + Nsat



if __name__ == "__main__":
    # call the entry function of this script
    main()
```

./code/assignment3_a.py

Code - downhill

```
import numpy as np
import matplotlib.pyplot as plt

def downhill_simplex(function, start, max_iters = 100, toll = 1e-10):
    """
        Try to find the minimum of the provided function
        with the help of the downhill simplex method
    In:
        param:function -- The function to minimize.
        param:start --  The best guess for the minimum.
        param:max_iters -- The maximum number of iterations to perform.
        param:toll -- The minimum tolerance for convergence.
    Out:
        return: An approximation for the minimum.
    """


    # The dimension in which we are.
    dim = len(start)

    # A matrix representing the vertices.
    vertices = np.zeros((dim+1, dim))
    vertices[0] = start #first one is start point

    # A matrix with function evaluations of the vertices.
    evaluations = np.zeros(dim+1)
    evaluations[0] = function(*vertices[0])

    # An array representing a direction
    # vector in our n-dimensional space.
    direction_vector = np.zeros(dim)

    # Create the simplex by selecting
    # vertices around the starting point.
    for i in range(1, dim+1):
        direction_vector[i-1] = 1
        vertices[i] = start + 0.1*direction_vector
        evaluations[i] = function(*vertices[i])
        direction_vector[i-1] = 0

    while max_iters >= 0:

        max_iters -= 1

```

```
        # Order the vertices such that f(x_0) <= f(x_1) <= f(x_2) ... <= f(x_n)
        # Just use selection sorts, even in 50 dimensions it is just of O(2500)
        for i in range(0, vertices.shape[0]):

            # Loop over the remaining vertices
            for j in range(i+1, vertices.shape[0]):

                # If the function evaluation is smaller
                # at the remaining vertex, then swap the vertices
                if evaluations[j] < evaluations[i]:
                    tmp_1 = vertices[i].copy()
                    tmp_2 = evaluations[i]
                    vertices[i] = vertices[j]
                    evaluations[i] = evaluations[j]
                    vertices[j] = tmp_1
                    evaluations[j] = tmp_2

        # Check if 'converged'
        if (abs(evaluations[dim] - evaluations[0]))/(abs(evaluations[dim] +
    evaluations[0])/2) < tol1:
            return vertices[0]

        # Determine the centroid.
        centroid = (1/(dim))*np.sum(vertices[0:dim], axis = 0)

        # Try a new reflected point.
        try_point = 2*centroid - vertices[dim]
        try_eval = function(*try_point)

        # if f(x_0) <= f(x_try) < f(x_n)
        # then the new point is better but not the best, accept it.
        if evaluations[0] <= try_eval < evaluations[dim]:
            vertices[dim] = try_point.copy()
            evaluations[dim] = try_eval
            continue


        # Expansion

        # New point is the very best, propose
        # a second point by expanding further in that
        # direction.
        if try_eval < evaluations[0]:
            expand = 2*try_point - centroid
            expand_eval = function(*expand)

            # Expansion is even better
            if expand_eval < try_eval:
                vertices[dim] = expand.copy()
                evaluations[dim] = expand_eval
            else:
                vertices[dim] = try_point.copy()
                evaluations[dim] = try_eval

            continue

        # Contraction

        # We now know that f(x_try) > f(x_{n-1}) and
        # therefore propose new point by contraction.

        try_point = (centroid + vertices[dim])/2
        try_eval = function(*try_point)

        # Is it better than the worst point?
        if try_eval < evaluations[dim]: # true
            vertices[dim] = try_point.copy()
            evaluations[dim] = try_eval
            continue

```

```
        # Shrink

        # All previous points where bad, zoom in on the best point
        # by shrinking.
        for i in range(1, dim+1):
            vertices[i] = (vertices[i]+ vertices[0])/2
            evaluations[i] = function(*(vertices[i]))


    return vertices[0]
```

./code/mathlib/minimize.py

Output - text

The optimal values found by ./mathlib/code/assignment3.a.py:

```
File: satgals_m11.txt a_opt:1.28368 b_opt:1.14950 c_opt:3.33632
File: satgals_m12.txt a_opt:1.54045 b_opt:0.94998 c_opt:3.74252
File: satgals_m13.txt a_opt:1.40908 b_opt:0.85051 c_opt:3.20000
File: satgals_m14.txt a_opt:1.61791 b_opt:0.74704 c_opt:3.48682
File: satgals_m15.txt a_opt:1.60000 b_opt:0.94610 c_opt:2.90000
```

./output/assignment3_a_out.txt

Output - plots

The fit of equation 13 to the data in the different files for the found optimal values.

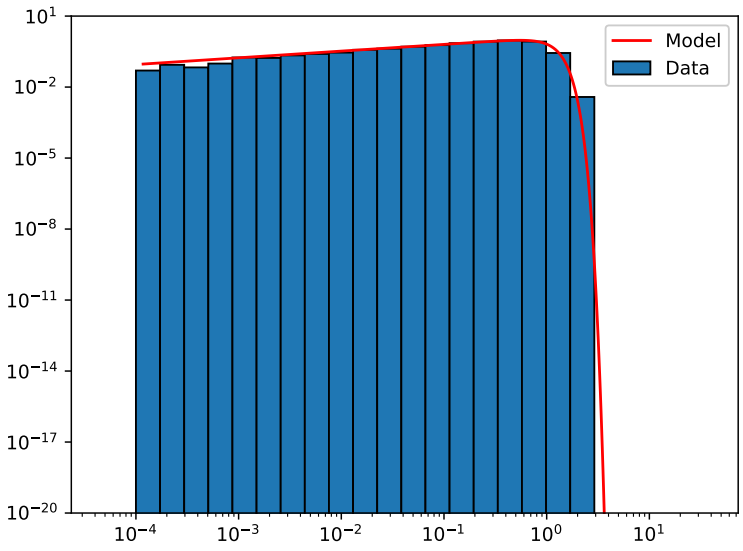

Figure 7: The model (equation 13) with the found parameters of a, b and c fitted to the data for the mass bin of 10^{11} $M_{\odot}$.

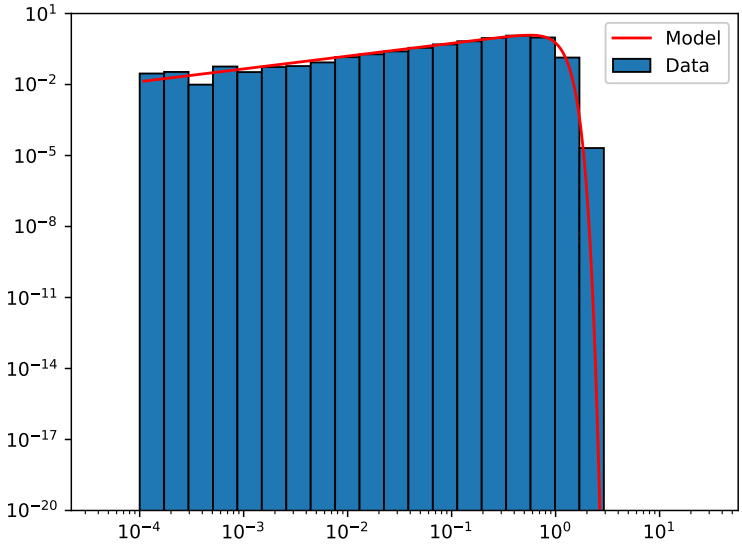

Figure 8: The model (equation 13) with the found parameters of a, b and c fitted to the data for the mass bin of 10^{12} $M_{\odot}$.

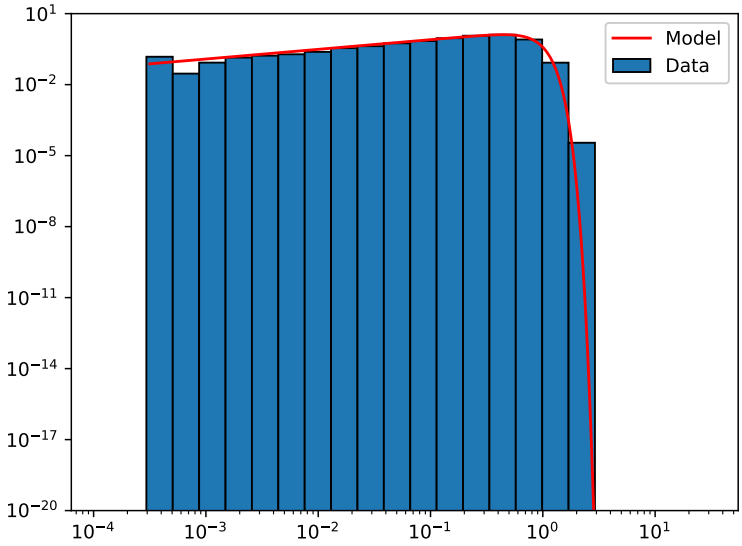

Figure 9: The model (equation 13) with the found parameters of a, b and c fitted to the data for the mass bin of 10^{13} $M_{\odot}$.

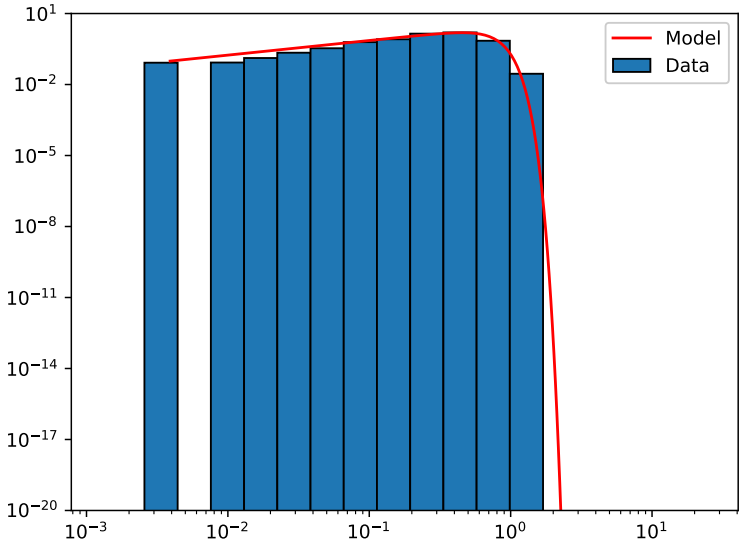

Figure 10: The model (equation 13) with the found parameters of a, b and c fitted to the data for the mass bin of 10^{14} $M_{\odot}$.

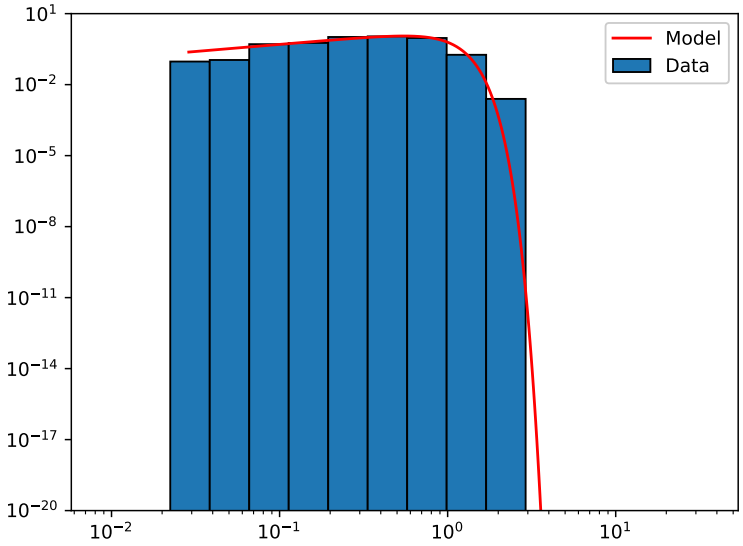

Figure 11: The model (equation 13) with the found parameters of a, b and c fitted to the data for the mass bin of 10^{15} $M_{\odot}$.

Assignment 3.b

Question

Write an interpolator for a, b and c as a function of halo mass **or** fir a function to each. Argue your specific choice! Plot the results.

Solution

A fit should be used if the data has an error, as the data is in this case approximate. A model that is fitted to the data should thus not pass through the data exactly, but should come as close as possible on average. If the error is significant small, then the data is 'exact'. In this case interpolation should be used, as an interpolator selects a curve that exactly passes through all data points and uses it to interpolate the values.

The choice of a fit or an interpolator does thus depend on the error of the data. The optimal parameters of a, b and c where found with the downhill simplex method. The error on these parameters depends on the flatness around the minimum of the negative log likelihood. If the negative log likelihood is flat around the minimum, then the error is larger than if the negative log likelihood is steep around the minimum. To understand this consider the following example. Let $a = 2$, $b = 2$ and $c = 3$ be the parameters for the absolute minimum and let the negative log likelihood be flat around this minimum. The value of the negative log likelihood will for $a = 1.5$, $b = 2$ and $c = 2.5$ be close to the value of the true parameters. The simplex will in this case not be able to distinguishing the optimal values from the non optimal values. If the negative log likelihood is however steep around the minimum then, there is a large difference between the optimal and the non optimal variables. The simplex can in this case distinguish the optimal from the non optimal values and 'walk' towards the true optimum.

The negative log likelihood is 6 times evaluated around the found values of the minimum to check whether the error on the minimum is large or small. Let a_{opt}, b_{opt} and c_{opt} be the optimal values and let $h = 0.1$ be a small stepsize. Te negative log likelihood is then evaluated at the 6 points $(a_{opt} \pm h,\, b_{opt} \pm h,\, c_{opt} \pm h)$. These 6 points are used to calculate the flatness,

$$F = \frac{1}{24 L(a_{opt}, b_{opt}, c_{opt})^2} \sum_{\text{points}} \left(L_i + L(a_{opt}, b_{opt}, c_{opt})\right)^2 \tag{18}$$

here $L(a, b, c)$ is the negative log likelihood and L_i is the value of the negative log likelihood evaluated for point i. If the flatness is approximately equal to 1 (i.e $F \approx 1$), then the function is flat, else it is steep. The results after the code section show that the function is flat around the found minimum for each of the mass bins. This thus suggests that found minimum has an non negligible error and that a fit should be used.

Note: I didn't had enough time to finish this assignment. I therefore below explain what I would have done if I had enough time.

The points plotted in logarithmic space (figure 12 below the code section) show that there might be a linear relation between the coefficients and the mass of the halo. All points are extremely flat and it is therefore assumed that all of them have the same error. A least square fit can in this case be used to fit the data with straight lines. The minimum of the least square fit can be found with one of the 1 dimensional algorithms discussed in class, or with the downhill simplex method.

The code that determines the flatness for each of the mass bins and creates the plot is located in the file: ./code/assignment3_ b.py [5]. The content of the file is shown below.

[5] All helper functions used in this file are already shown through out this report.

Code - output

The code that determines the flatness and creates the log-log point of the plot.

```
import numpy as np
import matplotlib.pyplot as plt
import mathlib.interpolate3D

# The a,b,c found in the previous assigment for
# m11, m12, m13, m14, m15.
abc_values = np.array([[1.28368, 1.14960, 3.33632],
               [1.54045, 0.94998, 3.74252],
               [1.40908, 0.85051, 3.20000],
               [1.61791, 0.74704, 3.48682],
               [1.60000, 0.94610, 2.90000]])

# Massa's of the halo's in suns masses.
masses = [10e11, 10e12, 10e13, 10e14, 10e15]


#The 3D interpolator for the normalization constant
interpolator = None

def main():
    global interpolator

    # Load the 'table' created in the previous assigment and use it
    # to create the 3d interpolator.
    table = np.load('table_2h.npy')
    # The a, b, c ranges of the 'table'.
    ranges = [np.arange(1.1, 2.6, 0.1),
              np.arange(0.5, 2.1, 0.1),
              np.arange(1.5, 4.1, 0.1)]

    # Create the interpolator.
    interpolator = mathlib.interpolate3D.Interpolate3D(table, ranges)

    # Go through all mass bins and caclulate the flatness around the optimum
    for idx, file in enumerate(['satgals_m11.txt', 'satgals_m12.txt', '
    satgals_m13.txt',
                                'satgals_m14.txt', 'satgals_m15.txt']):


        # Read the file with halos and create a super halo.
        super_halo = np.loadtxt(file, skiprows=4)[:, 0]

        # The number of satelites in the super halo.
        Nsat = len(super_halo)

        # A wrapper around the likelihood function. This is used
        # function that doesn't deppand on Nsat and super halo.
        likelihood_wrapper = lambda a, b, c: likelihood(Nsat, super_halo, a, b,
     c)

        # Calculate the flatness;

        # Stepsize
        h = 0.1
        # Optimal parameters found in the previous assignment.
        a_opt, b_opt, c_opt = abc_values[idx]
        # Evaluation of the negative loglikelihood around the optimum
        L_opt = likelihood_wrapper(a_opt, b_opt, c_opt)
        # Flatness
        F = 0

        # a dimension
        F += (likelihood_wrapper(a_opt + h, b_opt, c_opt) + L_opt)**2
        F += (likelihood_wrapper(a_opt - h, b_opt, c_opt) + L_opt)**2

        # b dimension
        F += (likelihood_wrapper(a_opt, b_opt + h, c_opt) + L_opt)**2
```

```
66          F += (likelihood_wrapper(a_opt, b_opt -h, c_opt) + L_opt)**2
67
68          # c dimension
69          F += (likelihood_wrapper(a_opt, b_opt, c_opt + h) + L_opt)**2
70          F += (likelihood_wrapper(a_opt, b_opt, c_opt - h) + L_opt)**2
71
72
73
74          # normalization
75          F *= 1/(24*L_opt**2)
76
77          # print the flatness.
78
79          # Print the file and the output.
80          print('File: {0} Flatness: {1:.10f}'.format(file, F))
81
82
83      # plot the points
84      labels = ['a', 'b', 'c']
85      colors = ['red', 'green', 'orange']
86      for i in range(0, 3):
87          plt.scatter(masses, abc_values[:, i], label = labels[i], c = colors[i])
88      plt.loglog()
89      plt.legend()
90      plt.savefig('./plots/3b.pdf')
91
92
93  def N(Nsat, x, a, b, c):
94      """
95          Calculate the number of satelites on an infinite thin shell
96          with radius x.
97
98      In:
99          param: Nsat -- The total number of satelites in the halo.
100         param: x -- The radius of the shell.
101         param: a -- The first model parameter.
102         param: b -- The second model parameter.
103         param: c -- The third model parameter.
104
105     Out:
106         return:  The number of satelites on an infinite thin shell
107                  with radius x.
108     """
109
110     global interpolator
111
112     return interpolator.interpolateLin(a,b,c)*b**2*(x/b)**(a-1)* np.exp(-(x/b)
        **c)*Nsat*4*np.pi
113
114 def likelihood(Nsat, x_values, a, b, c):
115     """
116         Calculate the negative loglikelihood for
117         the model given the total amount of satelites,
118         their x positions and the model parameters.
119
120     In:
121         param: Nsat -- The total number of satelites.
122         param: x_values -- The position of all the satelites.
123         param: a -- The first model parameter.
124         param: b -- The secon model parameter.
125         param: c -- The third model parameter.
126
127     Out:
128         return: The value of the negative loglikelihood evaluated for
129                 the given input.
130     """
131
132
133     # Only optimise within the bounded region 1 <= a <= 2.5. Values
134     # outside the bounds are put at the edge.
```

```
    if a < 1:
        a = 1
    elif a > 2.5:
        a = 2.5

    # Only optimise within the bounded region 0.5 <= b <= 2.0 Values
    # outside the bounds are put at the edge.
    if b < 0.5:
        b = 0.5
    elif b > 2.0:
        b = 2

    # Only optimise within the bounded region 1.5 <= c <= 4. Values
    # outside the bounds are put at the edge.
    if c > 4:
        c = 4
    elif c < 1.5:
        c = 1.5

    #return the negative loglikelihood.
    return - np.sum(np.log(N(Nsat, x_values, a, b, c))) + Nsat



if __name__ == "__main__":
    # call the entry function of this script
    main()
```

./code/assignment3_b.py

Output - text

The found flatness coefficients. The closer the value to 1, the flatter the minimum of the negative log likelihood is.

```
File: satgals_m11.txt Flatness: 0.9994103699
File: satgals_m12.txt Flatness: 0.9990262935
File: satgals_m13.txt Flatness: 0.9988925551
File: satgals_m14.txt Flatness: 0.9978142293
File: satgals_m15.txt Flatness: 0.9982462739
```

./output/assignment3_b_out.txt

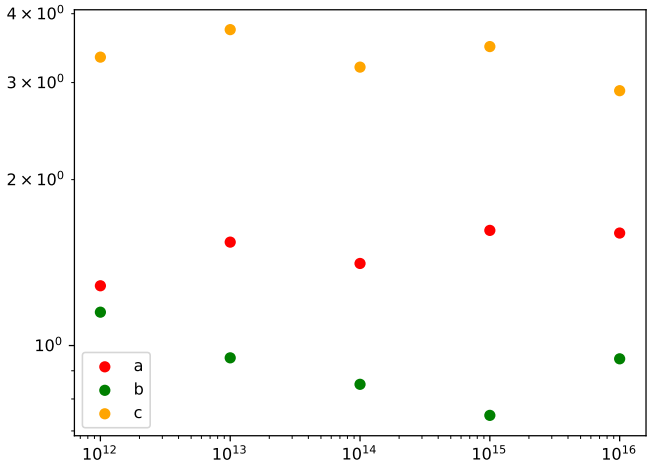

Figure 12: The optimal points plotted against the mass of the halo. The mass is in solar masses.