

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 for most questions 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 $(2.6, 40)$.

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

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/utls.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.

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

`./code/assignment1_a.py`

Code - helper

The code for the Poisson distribution and the factorial function.

```
1 import numpy as np
2
3 def factorial(number):
4     """
5     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 bits¹. This is done to fix number from becoming 'infinite' large².

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.

```
1 import mathlib.utils
2 import mathlib.rng
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 # Initialize the random number generator.
7 rng = mathlib.rng.RNG(5332341)
8 print('Initializing seed: ', 5332341)
9
10 # Plot sequential random numbers against each other in a scatter plot.
11 x_values = rng.gen_next_floats(1001)
12 plt.scatter(x_values[0:1000], x_values[1:1001], s=2)
13 plt.savefig('./plots/random_next_current.pdf')
14 plt.figure()
15
16 # Plot 1e6 random points.
17 values = rng.gen_next_floats(int(1e6))
```

¹The 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, except the last 32 bits, to zero.

²Python 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 plt.hist(values, bins=20, range=(0,1), alpha=0.8, edgecolor='black')
19 plt.savefig('./plots/random_uniformness.pdf')
20 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 = 0xBADCODE
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 = 0xFFFFFFFF
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.

```

```

58         """
59         return self._gen_next() * (1.0 / self._uint32_max)
60
61     def gen_next_floats(self, amount):
62         """
63         Generate multiple random floats
64         between 0 and 1.
65         In:
66             param: amount — The amount of floats to generate.
67         Out:
68             return: An array with 'amount' random floats
69                   between 0 and 1.
70         """
71
72         samples = np.zeros(amount)
73
74         for i in range(amount):
75             samples[i] = self.gen_next_float()
76
77         return samples
78
79     def gen_next_float_range(self, low, high):
80         """
81         Generate a float in a specific range.
82
83         In:
84             param: low — The lowest possible value to generate.
85             param: high — The highest possible value to generate.
86         Out:
87             return: A float in the range low, high.
88         """
89
90         ret = self.gen_next_float()
91         return ret * (high - low) + low
92
93     def gen_next_floats_range(self, low, high, amount):
94         """
95         Generate multiple floats in a specific range.
96
97         In:
98             param: low — The lowest possible value to generate.
99             param: high — The highest possible value to generate.
100            param: amount — The amount of floats to generate.
101         Out:
102             return: An array of floats in the range low, high
103         """
104
105         floats = self.gen_next_floats(amount)
106         return floats * (high - low) + low
107
108
109     def LCG(self, number):
110         """
111         Execute the linear congruential algorithm (LGC) on the
112         provided number.
113
114         In:
115             param: number — The number to execute the LCA on.
116         Out:
117             return: The number produced by the LCA.
118         """
119         return (self._lgc_a * number + self._lgc_c) % self._lgc_m
120
121     def _XOR_shift(self, number):
122         """
123         Execute the XOR-shift algorithm on the
124         input number.
125         In:
126             param: number — The number to XOR-shift.
127         Out:

```

```

128         return: The number produced by XOR-shift.
129     """
130
131     # Shift to the right and then bitwise xor.
132     number ^= (number >> self._xor_a1)
133     # Shift to the left and then bitwise xor.
134     number ^= (number << self._xor_a2)
135     # Shift to the right and then bitwise xor.
136     number ^= (number >> self._xor_a3)
137
138     return number
139
140 def _mwc(self, number):
141     """
142     Perform multiply with carry (MWC) on
143     the given input.
144     In:
145         param: number — The number to perform MWC on.
146     Out:
147         return The new number.
148     """
149     return self._mwc_a * (number & (self._uint32_max - 1)) + (number >> 32)
150
151 def _gen_next(self):
152     """
153     Generate the next pseudo random value
154     and update the seed.
155     """
156
157     old_seed = self._seed
158
159     self._seed = self._XOR_shift(old_seed)
160     self._seed ^= self._LCG(old_seed)
161     self._seed -= self._mwc(old_seed)
162
163     # Only keep the last 32 bits to prevent number becoming too large.
164     self._seed &= self._uint32_max
165
166     return abs(self._seed)
167
168 def rejection_sampling(self, sample_dist, encl_func, \
169                       encl_unif_transf, samples):
170     """
171     Perform rejection sampling.
172
173     In:
174         param: sample_dist — The distribution to sample from.
175         param: encl_func — The function that encloses the distribution.
176                             This is the distribution that is chosen to
177                             enclose the sample distribution multiplied
178                             by a scalar x so that
179                             sample_dist(x) < enclosing_func(x) for x in
180                             the interval of which the distribution holds.
181
182         param: encl_unf_transf — The transformation that transforms a
183                                 uniform distributed variable
184                                 between 0 and 1 to a variable
185                                 distributed according to the chosen
186                                 enclosing distribution.
187         param: samples — The amount of samples to sample from 'sample_dist'
188     ,
189
190     Out:
191         return: An array of 'samples' samples sampled
192                 from 'sample_dist'.
193     """
194
195     # An array that stores the generated samples.
196     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).

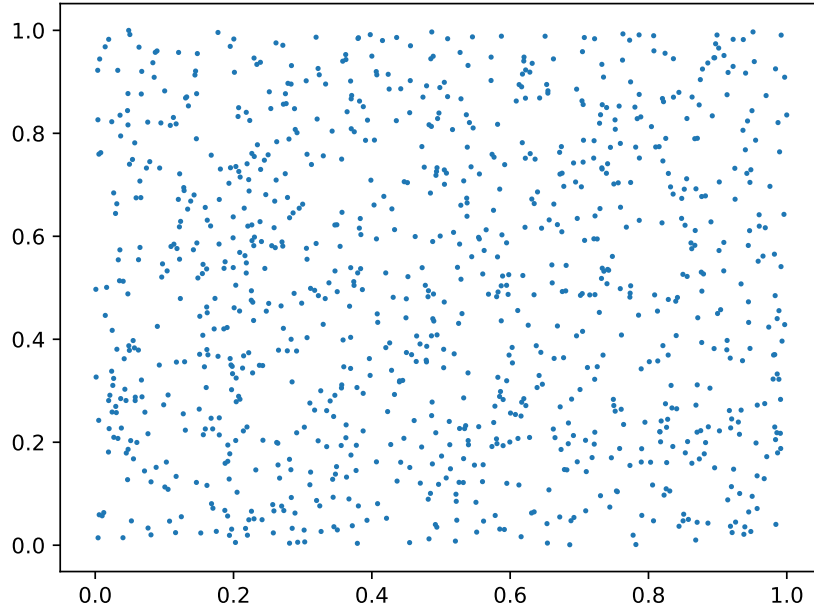


Figure 1: The result of plotting x_{i+1} against x_i for 1000 values.

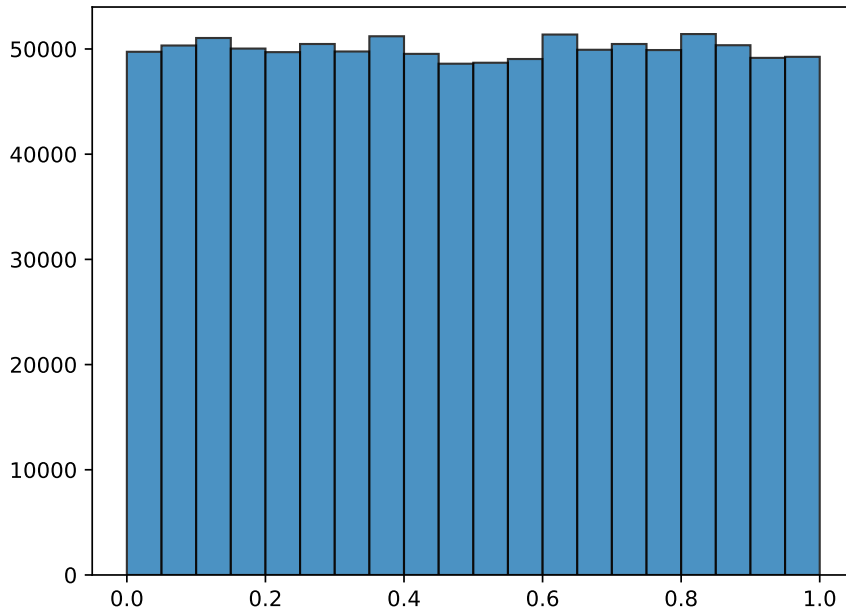


Figure 2: The uniformness of the random number generator for 1 million generated values between 0 and 1. The values are binned in 20 bins

Assignment 2.a

Question

The number density profile for satellite galaxies is given by,

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

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,

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

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.

```
1 import mathlib.rng
2 import mathlib.integrate
3 import numpy as np
4
5 # Recreate the random number generator with the last seed of the previous
6   exercise.
7 # This is done to make the execution of the random number generator
8 # continues through out the all assignments. See assignment 1.b in
9 # the report for more information.
10 rng = mathlib.rng.RNG(3197832550)
11
12 # Generate the random numbers.
13 a = rng.gen_next_float_range(1.1, 2.5)
14 b = rng.gen_next_float_range(0.5, 2)
15 c = rng.gen_next_float_range(1.5, 4)
16
17 # The density function in spherical coordinates integrated
18 # over the angles theta and phi.
19 density = lambda x: b**2 * (x/b)**(a-1) * np.exp(-(x/b)**c) * 4 * np.pi
20
21 # Find the normalization constant.
22 A = 1/mathlib.integrate.romberg(density, 0, 5, 15)
23
24 for label, value in zip(['a', 'b', 'c', 'A'], [a, b, c, A]):
25     print(label + ' = {0:.18f}'.format(value))
26
27 # Print the seed, which is needed in assignment 2.d.
28 print('Final seed: ', rng.get_seed())
```

`./code/assignment2_a.py`

Code - romberg

The code for romberg integration.

```
1 import numpy as np
2
3 def trapezoid(function, a, b, num_trap):
4     """
5         Perform trapezoid integration
6
7     In:
8         param: function — The function to integrate.
9         param: a — The value to start the integration at.
10        param: b — The value to integrate the function to.
11        param: num_trap — The number of trapezoids to use.
12
13    Out:
14        return: An approximation of the integral from a to b of
15                the function 'function'
16    """
17
18    # The step size for the integration range.
19    dx = (b-a)/num_trap
20
21    # Find trapezoid points.
22    x_i = np.arange(a, b+dx, dx)
23
24    # Determine the area of all trapezoids.
25    area = dx*(function(a) + function(b))/2
26    area += np.sum(function(x_i[1:num_trap])) * dx
27
28    return area
29
30
31
32 def romberg(function, a, b, num_trap):
33     """
34         Perform romberg integration
35
36    In:
37        param: function — The function to integrate.
38        param: a — The value to start the integration at.
39        param: b — The value to stop the integrate at.
40        param: num_trap — The maximum number of initial trapezoid
41                           to use to approximate the area.
42
43    Out:
44        return: An approximation of the integral from a to b of
45                the function 'function'
46    """
47
48    # The array to store the combined results in.
49    results = np.zeros(num_trap)
50
51    # The current combination:
52    # 0 = combine trapezoids, 1 = combine combined trapezoids,
53    # 2 = combine the combined combined trapezoids etc.
54    for combination in range(0, num_trap-1):
55
56        # Iterate and combine.
57        for j in range(1, num_trap-combination):
58            # Create the initial trapezoids to combine.
59            if combination == 0:
60                # We need two trapezoids to combine the very first time.
61                if j == 1:
62                    results[j-1] = trapezoid(function, a, b, 1)
63
64                    results[j] = trapezoid(function, a, b, 2**j)
65
66            # Combine.
67            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/assignment2_ 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.

```
1 import mathlib.interpolate as interp
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 # The values of abc found in assignment 2.1.
6 a = 1.316496837562065814
7 b = 1.305096297363074420
8 c = 1.790962180074062715
9 A = 0.051464524931584120
10
11 # The volume number density function that is
12 # used to create the y-values of the points.
13 density = lambda x: A*(x/b)**(a-3)* np.exp(-(x/b)**c)
14
15 # The points.
16 points_x = np.array([1e-4, 1e-2, 1e-1, 1, 5.0])
17 points_y = density(points_x)
18
19 # Take the logarithm.
20 points_x_log = np.log10(points_x)
21 points_y_log = np.log10(points_y)
22
23 # Points to interpolate.
24 interpolate_x = np.linspace(1e-4, 5, 2000)
25 interpolate_x_log = np.log10(interpolate_x)
26
27 # Create the interpolated points.
28 lin_interp = list()
29 poly_interp = list()
30
31 for x in interpolate_x_log:
32     lin_interp.append(10**interp.interpolate_linear(points_x_log,\
33                                                     points_y_log, x))
34     poly_interp.append(10**interp.interpolate_neville(points_x_log,\
35                                                       points_y_log, x))
36
37 # First only plot the points and then plot the points
38 # with the interpolated values.
39 plt.loglog()
40 plt.scatter(points_x, points_y, s=15, color = 'black',zorder=1)
41 plt.savefig('./plots/points.pdf')
42 plt.plot(interpolate_x, lin_interp, label = 'linear',
43          linestyle = '—', zorder = 0)
44 plt.plot(interpolate_x, poly_interp, label= 'polynomial',
45          linestyle = '-.', zorder = 0)
46 plt.legend()
47 plt.savefig('./plots/interpolate.pdf')
```

./code/assignment2_b.py

Code - interpolate

The code of the linear and polynomial interpolator.

```
1 import numpy as np
2
3 def _find_bisection_idx(x_vals, points, x):
4     """
5     Find the index of the point that is the closest to x from the left
6     in the provided array of known x points.
7
8     In:
9         param: x_vals — The known x value's.
10         param: points — The minimum number of required points
11                        that are needed to interpolate. For example 2
12                        when using linear interpolation.
```

```

13     param: x — The point to find the closest point for.
14 Out:
15     return: The index of the closest known point that is left of x,
16             corrected by the minimum number of required points needed
17             for interpolation.
18
19 """
20
21
22 # Index of known point of the right.
23 edge_right_idx = len(x_vals) - 1
24 # Index of known point of the left.
25 edge_left_idx = 0
26 # Index of the known point in the center of the left and right point.
27 edge_mid_idx = 0
28
29 # While the known left and right point are not separated
30 # by known points, perform bisection.
31 while edge_right_idx - edge_left_idx > 1:
32
33     # Use the left and right index to find the index
34     # of the point in the center.
35     edge_mid_idx = (edge_right_idx + edge_left_idx) >> 1
36
37     # Is the point x left of the point in the middle?
38     if x_vals[edge_mid_idx] > x: #True -> set right index to middle
39         edge_right_idx = edge_mid_idx
40     else: #False, set left index to middle index
41         edge_left_idx = edge_mid_idx
42
43
44 # The point is now found, the interpolation might need
45 # need a minimum of point 'points' for interpolation. If there are not
46 # enough points on the right of the left_index then decrease the index
47 # until there are enough.
48
49 # Not enough points because the left point is to far to the right.
50 if edge_left_idx + points > len(x_vals):
51
52     # Move to the left until there are enough points.
53     while edge_left_idx + points > len(x_vals):
54         edge_left_idx -= 1
55
56     # Not enough points in the dataset.
57     if edge_left_idx < 0:
58         raise "Dataset is to small to interpolate x. Not enough points"
59
60     return edge_left_idx
61 else: # Enough points are available
62     return edge_left_idx
63
64
65
66 def interpolate_linear(x_vals, y_vals, x):
67     """
68         Perform linear interpolation
69
70     In:
71         param: x_vals — The known x values.
72         param: y_vals — The known y values.
73         param: x — The point to linear interpolate using the known values.
74
75     Out:
76         return: The interpolated y value for x.
77     """
78
79     # Find the index of the closest known value to the left of x.
80     idx = _find_bisection_idx(x_vals, 2, x)
81
82     # Horizontal line, no need to interpolate.

```

```

83     if y_vals[idx] == y_vals[idx+1]:
84         return y_vals[idx]
85
86     # Linear interpolate.
87     else:
88
89         # Find the slope between the points x_{idx} and x_{idx} + 1
90         a = (y_vals[idx+1] - y_vals[idx]) / (x_vals[idx+1] - x_vals[idx])
91
92         # Find the intersect.
93         b = y_vals[idx] - a*x_vals[idx]
94
95         # Linear interpolate the point x.
96         return a*x + b
97
98
99 def interpolate_neville(x_vals, y_vals, x):
100     """
101     Perform polynomial interpolation using Neveille's algorithm
102
103     In:
104     param: x_vals — The known x values.
105     param: y_vals — The known y values.
106     param: x — The point to linear interpolate using the known values
107
108     Out:
109     return: The interpolated y value for x.
110     """
111
112
113     # Number of x values.
114     values = len(x_vals)
115
116     # The array that contains the initial polynomial values.
117     initials = np.zeros(values)
118
119
120     # Combine the evaluations of order 'order' to approximate
121     # the higher order polynomial. Store the results in the
122     # array with the initial values.
123     for order in range(0, values):
124
125         # Each order had one less element. Combine them to approximate
126         # the higher order polynomial.
127         for element_idx in range(values - order):
128             # The very first time set the initial values.
129             if order == 0:
130                 initials[element_idx] = y_vals[element_idx]
131             else:
132                 top = (x_vals[order+ element_idx] - x)* initials[element_idx]
133                 top += (x - x_vals[element_idx])*initials[element_idx+1]
134                 bottom = x_vals[element_idx+order] - x_vals[element_idx]
135
136                 initials[element_idx] = top/bottom
137
138     return initials[0]

```

./code/mathlib/interpolate.py

Output - Figure

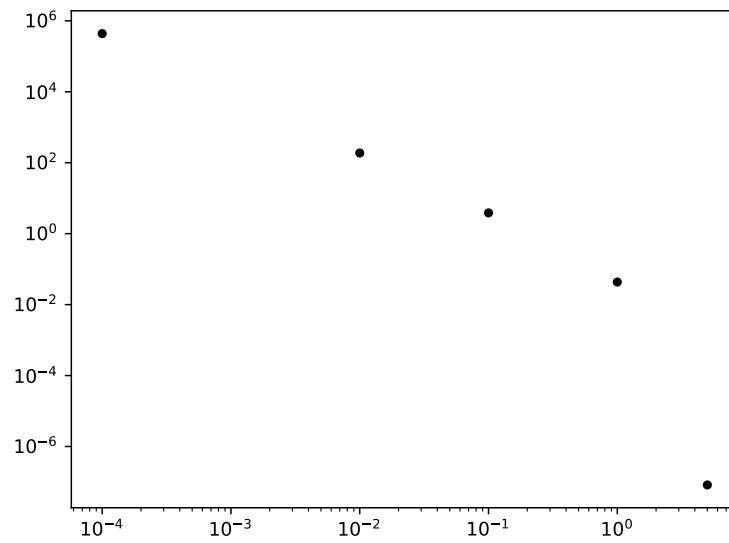


Figure 3: A logarithmic plot of the 5 points that need to be interpolated.

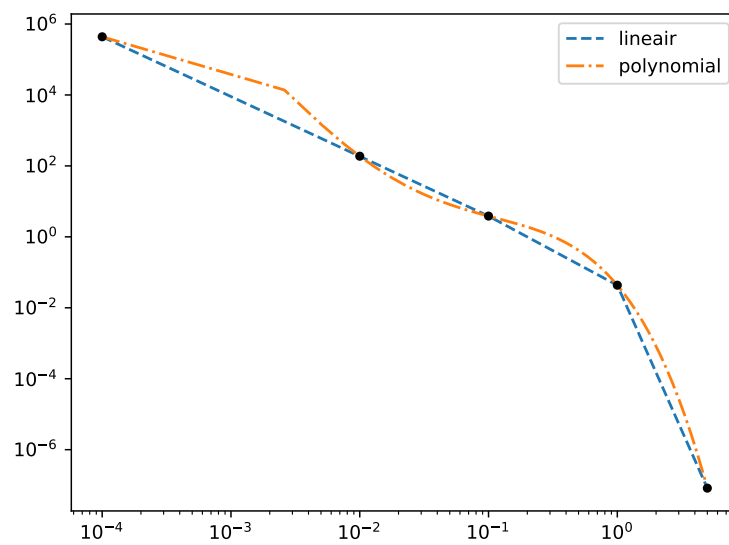


Figure 4: The interpolation of the linear interpolator and the polynomial interpolator.

Assignment 2.c

Question

Numerically calculate $dn(x)/dx$ 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,

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

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,

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

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.

```
1 import mathlib.derivative
2 import numpy as np
3
4 # The values of abc found in assignment 2.1.
5 a = 1.316496837562065814
6 b = 1.305096297363074420
7 c = 1.790962180074062715
8 A = 0.051464524931584120
9
10 # The analytical derivative and the density.
11 analytical = (A*100*((a-3) - c))/(b*np.exp(1))
12 n = lambda x: (x/b)**(a-3)* np.exp(-(x/b)**c)*A*100
13
14 # Print the results.
15 print('Analytical: {0:.12f}'.format(analytical))
16 print('Numeric: {0:.12f}'.format(mathlib.derivative.ridder(n,b,1e-13)))
```

`./code/assignment2_c.py`

Code - ridder

The code for ridder's method

```
1 import numpy as np
2
3 def central_diff(function, x, h):
4     """
5     Use the central difference method to approximate the derivative
6     at a point x.
7
8     In:
9     param: function — The function to calculate the derivative of.
10    param: x — The point to approximate the derivative of the function at.
11    param: h — A small value h that in the limit would go to zero.
12    Out:
13    return: An approximation of the derivative of the provided
14            function at x.
15    """
16
17    #f'(x) approx (f(x+h)-f(x-h))/2h
18    return (function(x+h) - function(x-h))/(2*h)
19
20
21 def ridder(function, x, precision):
22     """
23     Perform ridders differential method to estimate
24     the derivative at a point x
25
26     In:
27     param: function — The function to estimate the derivative for.
28     param: x — The point to estimate the derivative at.
29     param: precision — The precision that must be obtained
30                       in the estimation.
31    Out:
32    return: An approximation of the derivative.
33    """
34
35    # The number of initial approximations to use.
36    approximations = 0
37    # The return value.
38    ret = 0
39    # The current precision.
40    current_precision = 0xFFFFFFFF
41
42    # While we didn't reach the request precision with
43    # the current amount of initial approximations, try again
44    # but with more approximations.
45    while current_precision > precision:
46        # Increase the amount of initial approximations.
47        approximations += 5
48        # Reset the error.
49        current_precision = 0xFFFFFFFF
50
51
52    # The array to store the combined results in.
53    results = np.zeros(approximations)
54
55    # The current combination:
56    # 0 = combine initial central_difference evaluations,
57    # 1 = combine the combined central difference evaluations
58    # 2 = combine the combined combined central difference evaluations etc.
59    for combination in range(0, approximations-1):
60
61        # Combine for the current 'combination'.
62        for j in range(1, approximations-combination):
63
64            # Create the initial central difference to combine.
65            if combination == 0:
66                # We need two central difference's to
67                # combine the very first time.
```

```

68         if j == 1:
69             results[j-1] = central_diff(function, x, 1)
70
71         # Decrease h by a factor of 2 for each next approximation.
72         results[j] = central_diff(function, x, (1/2)**j)
73
74         # Keep the evaluation of the previous combination that
75         # is getting overwritten temporary in memory to update
76         # the current precision.
77         previous = results[j-1]
78
79         # Combine
80         power = 4** (combination+1)
81         results[j-1] = (power*results[j] - results[j-1])/(power-1)
82
83         # Determine the new precision
84         precision_tmp = max(abs(results[j-1] - previous), \
85                             abs(results[j-1] - results[j]))
86
87         # New precision is smaller -> update
88         if precision_tmp < current_precision:
89             current_precision = precision_tmp
90             ret = results[j-1]
91
92         # Terminate if requested precision is reached
93         if current_precision < precision:
94             return ret
95
96         # Abort early if the error of the last combined result is worse
97         # than the previous order by a large amount.
98         if j == (approximations-combination-1) \
99             and abs(ret - previous) > 100*current_precision:
100
101             return ret
102     return ret

```

./code/mathlib/derivative.py

Output - Text

The output from ./code/assignment2_c.py with 12 significant digits.

```

1 Analytical: -5.040329317958
2 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,

$$\phi = 2\pi v \quad (6)$$

$$\theta = \arccos(1 - 2v) \quad (7)$$

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. $p(x) < k p_{enc}(x) \forall x \in [0, 5]$). The chosen distribution is an uniform distribution for $x \in [0, 5]$,

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

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,

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

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

$$z = 5y \quad (10)$$

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 generated³ from an uniform distribution between 0 and $k p_{enc}(z)$ 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.

³Let 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.

```
1 import mathlib.rng
2 import mathlib.derivative
3 import numpy as np
4
5 # The values of abc found in assignment 2.1.
6 a = 1.316496837562065814
7 b = 1.305096297363074420
8 c = 1.790962180074062715
9 A = 0.051464524931584120
10
11 # Initialize with the seed at the end of assignment 2.1.
12 rng = mathlib.rng.RNG(499869219)
13
14 # The distribution from which we want to sample.
15 sample_dist = lambda x : (b**2 * (x/b)**(a-1)* np.exp(-(x/b)**c) )*A*4*np.pi
16 # The enclosing distribution.
17 enclosing_dist = lambda x : 2
18 # The transformation for a uniform variable to the enclosing distribution.
19 enclosed_transform = lambda x : 5*x
20
21 # Generate 100 uniform distributed angles.
22 theta, phi = rng.gen_uniform_spherical_surface_coords(100)
23 radii = rng.rejection_sampling(sample_dist, \
24                               enclosing_dist, enclosed_transform, 100)
25
26 # Display the final seed used for the next assignment,
27 print('Final seed: ', rng.get_seed())
28 # Empty line to separate seed from the remaining output
29 print('')
30
31 # Print the results
32 print('x, phi, theta')
33 for pair in zip(radii, phi, theta):
34     print(pair)
```

./code/assignment2_d.py

Output - text

The generated output.

```
1 Final seed: 1016249662
2
3 x, phi, theta
4 (0.3579938284489312, 3.6257715658348064, 1.7255319984510258)
5 (2.485454148260284, 3.1390020926533575, 1.5699717255446173)
6 (0.4541655596006116, 5.843930975811292, 2.6064208102306603)
7 (1.5978706294200082, 0.45178480110985847, 0.5429419074656698)
8 (1.1149836438975724, 3.9722785250354633, 1.8383941540875628)
9 (0.7093001752414974, 2.0975005512454707, 1.2320076806321651)
10 (0.5454029505013961, 0.4179331781088591, 0.5217110205747997)
11 (0.4163416476026973, 1.3657866067437219, 0.9700520190115742)
12 (0.7441318397745798, 3.946953265513712, 1.8300448344712918)
13 (0.05379230134510256, 1.8290937441735347, 1.1397943248306432)
14 (0.8180404584431183, 1.882185879343575, 1.1583173446872264)
15 (0.47618899389081376, 0.04835423943941989, 0.17567742622176938)
16 (1.171271449693309, 0.47408060452189765, 0.556525770562289)
17 (0.8240872402265871, 5.808786488717012, 2.584875161184929)
18 (0.7075331396673651, 6.2752330512715, 3.070425943784541)
19 (0.3125357989483829, 2.249217286617513, 1.2827788731358873)
20 (0.7580733079831287, 2.420777266647821, 1.3392912948857487)
21 (0.6215506723666449, 3.8517906521695933, 1.798830519836583)
22 (1.4673194397397618, 3.980560841594835, 1.8411288191929622)
23 (0.701170441624981, 4.186329245957005, 1.9098024926329265)
24 (1.1499401918030203, 4.919064339462517, 2.1721834961232105)
25 (0.2875093452836176, 3.5907494313570334, 1.714258976950838)
```

26 (0.7690208116008483, 0.7678605758310203, 0.7142535941495242)
 27 (0.5020575633975811, 6.247139421166786, 2.9899630360509617)
 28 (0.3804066114547678, 0.7348390476584241, 0.698055430681658)
 29 (0.34301911488711345, 2.9687685565379285, 1.5157569235344739)
 30 (1.3005203430774903, 1.768384719750567, 1.1184182204321294)
 31 (1.049710961768802, 1.9112587780332704, 1.168396684520003)
 32 (0.6364001498176717, 0.039911573892260445, 0.15956966944064624)
 33 (2.527872090583637, 0.34375757729436734, 0.4721806895663418)
 34 (0.9569772987060661, 3.5388058712071633, 1.697572543733482)
 35 (1.1384866691982576, 4.040522041313443, 1.8609903185074965)
 36 (0.6510213833886714, 2.08623940431945, 1.228204556736262)
 37 (0.7863035206185429, 3.1807348693550206, 1.583256003418326)
 38 (1.377572160302096, 5.305222275643324, 2.3304973411227587)
 39 (0.21021116762659772, 2.261984089311919, 1.2870146113333585)
 40 (0.6828774408630276, 4.4465089557754025, 1.9991433778357142)
 41 (1.8850988608051789, 4.9931440474598086, 2.201071537497057)
 42 (1.1286425662992157, 3.996343891711613, 1.8463458483513429)
 43 (0.9872137408208135, 5.532870662119667, 2.435907593130786)
 44 (0.32931922593370994, 3.6043717631735754, 1.718641509029965)
 45 (1.389728403042473, 0.41313362184410113, 0.518637276661106)
 46 (0.5898687221551939, 1.4601768461551858, 1.0060384873238106)
 47 (0.7752025001158944, 4.105047287966843, 1.8824962985214837)
 48 (2.6754527568527156, 0.3027782589945894, 0.44264313923278487)
 49 (0.09967927474055423, 1.2041681001094282, 0.9062508601035965)
 50 (1.8776655515836704, 2.180716018803088, 1.259958381951315)
 51 (0.08876249079796544, 5.051589061325694, 2.224298534183561)
 52 (1.0556129601447874, 0.011354697611375811, 0.08504694955548202)
 53 (1.0668362633015114, 3.8358833637349052, 1.7936356181624407)
 54 (0.19937052861772722, 0.996492252454145, 0.819198961181742)
 55 (0.1681643561851616, 4.464005243884389, 2.0052743745216035)
 56 (0.08704470356159022, 0.9120842916126898, 0.781760160197898)
 57 (0.5375231698941261, 4.467576446354424, 2.006527921835051)
 58 (0.3597077157720243, 4.625436532114527, 2.062719818095845)
 59 (1.0923313864256095, 0.6592134682747388, 0.6597167976591414)
 60 (1.1161449938351626, 2.8028816806565002, 1.462771300113757)
 61 (1.278946119425573, 3.3990114690680735, 1.6528272483672306)
 62 (1.1000187325058548, 4.662921385312137, 2.076306622905424)
 63 (0.4796606815605565, 3.194492323599985, 1.5876356105524145)
 64 (0.10401859649084941, 3.3699955027612822, 1.6435634122630045)
 65 (0.09232803715679982, 5.908902602658324, 2.648476669835251)
 66 (1.170586959032944, 2.131435513763629, 1.2434377483666312)
 67 (0.4094679002672126, 1.0260995916912727, 0.8320219032742583)
 68 (2.311631084259514, 3.5246704486696423, 1.693037992133638)
 69 (1.157567722526744, 3.024518313540633, 1.5335217760438127)
 70 (2.1493239822213828, 4.410109312764263, 1.9864427443966763)
 71 (0.17980300476304328, 6.278362910374851, 3.0861776835631125)
 72 (0.27707741718671225, 4.974510684329675, 2.193749231978967)
 73 (0.1325431536260394, 3.945136900574399, 1.8294467263330656)
 74 (2.1336953358570336, 6.154539503826982, 2.8544279035598596)
 75 (1.474245286424236, 1.3259100902203211, 0.9545822652253781)
 76 (0.3535610461499451, 2.1424972298453664, 1.2471539381354246)
 77 (1.8689941921897684, 1.157530892008798, 0.8872501518212352)
 78 (0.496305306557637, 4.94487953756372, 2.1821836715333656)
 79 (0.2796433820108984, 4.697434177065051, 2.088907092417107)
 80 (0.8668761772259316, 6.196403675441801, 2.9060015995359376)
 81 (0.13309590987234746, 1.1594475926475194, 0.8880367886252412)
 82 (0.6785562473066514, 1.686372296041978, 1.0891813754537536)
 83 (0.829390833580259, 1.5189879845288539, 1.028048237297669)
 84 (0.024451633222506295, 0.9413026277661871, 0.7948748960867421)
 85 (1.039777572974511, 1.6898077909163576, 1.0904148900447959)
 86 (0.42673284896340524, 1.9779474771807983, 1.1913569678117872)
 87 (1.5269727216863478, 1.847337716546747, 1.146176782855883)
 88 (1.5051165855268753, 0.7103473076563005, 0.6858366829840647)
 89 (0.4047520471282192, 2.8637378863416214, 1.4822366934654974)
 90 (0.5281468051318421, 6.071337982267417, 2.7722556824268483)
 91 (1.2742317622234653, 4.932325124241948, 2.1773117347319153)
 92 (2.86956520468685, 2.0464460927247448, 1.214723628186573)
 93 (0.5294553168419411, 1.4938966821809838, 1.0186945174290796)
 94 (0.40818834104765866, 3.0396934026989695, 1.5383550977692144)
 95 (0.27314911137175496, 2.5331241525441537, 1.3758829636869725)

```

96 (1.5413572805797116, 3.693748172709161, 1.7474705658625795)
97 (0.5962656113776065, 3.90974897806, 1.8178124555097586)
98 (0.013182851256146759, 2.372614420780756, 1.323510377583853)
99 (1.823761988623012, 0.09078962497343887, 0.24099590143646527)
100 (0.9896303575927462, 5.68253479322631, 2.5129169524463126)
101 (1.0666621141290902, 0.3479916674450603, 0.475135351696596)
102 (1.9487270496200602, 3.2649587351224194, 1.6100750693898447)
103 (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 below⁴. The output is located at

Code - plots

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # The values of abc found in assignment 2.1.
5 a = 1.316496837562065814
6 b = 1.305096297363074420
7 c = 1.790962180074062715
8 A = 0.051464524931584120
9
10 # The amount of halo's to generate.
11 haloes = 1000
12
13 # Initialize with the seed at the end of assignment 2.4
14 # to keep the usage of the random number generator contineus
15 # through out the full assignment (see report question 1.b).
16 rng = matplotlib.random.RandomState(1016249662)
17
18
19
20
21 # The functions from assignment 2.4 and the new function to plot (last one).
22 sample_dist = lambda x : (b**2 * (x/b)**(a-1)* np.exp(-(x/b)**c) ) * A * 4 * np.pi
23 enclosing_func = lambda x : 2
24 enclosed_transform = lambda x : 5*x
25 N = lambda x : sample_dist(x)*100 #N(x) = p(x)*<N_{sat}> (plot function)
26
27 # Create an array with the bins.
28 bins = np.logspace(-4, np.log10(5), 21)
29
30 # Create a matrix that stores all radii for each iteration.

```

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

```

31 # This matrix is not used in this assignment, but is needed
32 # for assignment 2.g.
33 radi_all = np.zeros((haloes, 100))
34
35 #
36 # Generate the samples
37 #
38
39 # The first halo is generated outside a loop to obtain the bin-widths,
40 # bin-centers and the initial histogram. This histogram is used to
41 # create a combined histogram by adding the histograms for each halo
42 # to it. The final histogram is then obtained by dividing this combined
43 # histogram by the bin width and the number of haloes.
44
45 # Radii of first halo.
46 radi = rng.rejection_sampling(sample_dist, \
47                               enclosing_func, enclosed_transform, 100)
48 histogram, bin_edges = np.histogram(radi, bins=bins)
49 bin_widths = bins[1:] - bins[:-1]
50 bin_centers = (bins[0:-1] + bins[1:])/2
51
52 # Remaining haloes are created in a loop
53 for i in range(1, haloes):
54     radi = rng.rejection_sampling(sample_dist, \
55                                   enclosing_func, enclosed_transform, 100)
56     histogram += np.histogram(radi, bins=bins)[0] # add histograms together
57     radi_all[i] = radi
58
59 # Divide by the number of haloes to obtain the average and
60 # divide by bin width to correct for different sized bins.
61 histogram = np.array(histogram)
62 histogram_normed = histogram / (haloes * bin_widths)
63
64 # Plot the histogram and the function.
65 x_plot = np.linspace(0.002, 5, haloes) #x values to plot N(x) for
66 plt.bar(bin_centers, histogram_normed, width=bin_widths, color='orange',
67         edgecolor='black')
68 plt.plot(x_plot, N(x_plot), c='red')
69 plt.loglog()
70 plt.xlim(1e-3, 10)
71 plt.savefig('./plots/assignment2e.pdf')
72
73 # Save the generated radii for assignment 2.g.
74 np.save('radii_2e', radi_all)
75
76 # No need to print the final seed as the random number generator is
77 # not needed for any other assignment.

```

./code/assignment2_e.py

Output - plot

The output generated by ./code/assignment2_e.py consists of figure 5 on the next page.

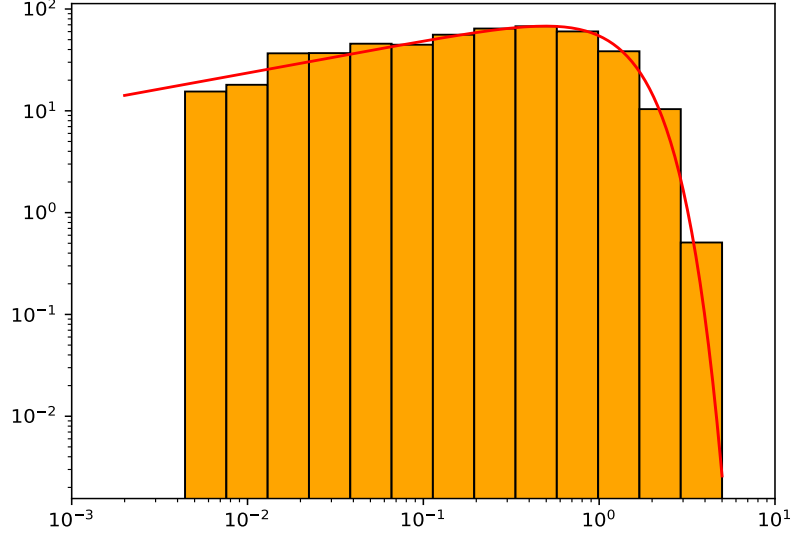


Figure 5: The average number of satellites on a spherical shell at distance $x = r/r_{viral}$. The red line is the original function $N(x)$ that is used to generate the histogram.

Assignment 2.6

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,

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

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,

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

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.

```

1 import mathlib.roots as roots
2 import numpy as np
3
4
5 # The values of abc found in assignment 2.1.
6 a = 1.316496837562065814
7 b = 1.305096297363074420
8 c = 1.790962180074062715
9 A = 0.051464524931584120
10
11
12 # The maximum x and y coordinate ,
13 x_max = b*((a-1)/c)**(1/c)
14 y_max = (b**2 * (x_max/b)**(a-1)* np.exp(-(x_max/b)**c) )*A*4*np.pi*100
15
16
17 # The function to put equal to zero and solve .
18 root_func = lambda x: (b**2 *(x/b)**(a-1)* np.exp(-(x/b)**c))*A*4*np.pi*100 -
    y_max/2
19 # The derivative of this function , needed for newton_raphson.
20 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))
21
22 # Print the roots
23 print('Left root: {0:.7f}'.
    .format(roots.newton_raphson(root_func , root_der , 0 , 0.3 , 1e-10)))
24 print('Right root: {0:.7f}'.
    .format(roots.newton_raphson(root_func , root_der , 1 , 2 , 1e-10)))
25
26

```

`./code/assignment2_f.py`

Code - Newton- Raphson

The code for the Newton-Raphson method.

```

1 import numpy as np
2
3
4
5 def newton_raphson(func , func_deriv , x_left , x_right , xacc):
6     """
7         Calculate the roots using the Newton-Raphson method
8
9     In:
10         param: func — The function to calculate the roots for.
11         param: func_deriv — The derivative of the function to
12             calculate the roots for.
13         param: x_left — The left bracket value that encloses the root.
14         param: x_right — The right bracket value that encloses the root
15         param: xacc — The accuracy to find the root with.
16
17     Out:
18         return: An approximation of the x coordinate of the root.
19     """
20

```

```

21
22 # The initial accuracy and the maximum number
23 # of iterations.
24 acc = 0xFFFFFFFF
25 iters = 1e4
26
27 # Make sure the left bracket value is the smallest.
28 if x_left > x_right:
29     raise "Left bracketed value is larger than right bracketed 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_derv(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/assignment2_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/utlis.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.

```
1 import mathlib.utils as utils
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5
6 # Read in the radii from assignment 2.e.
7 radii_matrix = np.load('radii_2e.npy')
8
9 #
10 # Create the histograms for each of the 100 generated radii (see assignment 2.e)
11 # and find the bin with the maximal number of counts over the full dataset
12 # (i.e the maximum within all the histograms).
13 #
14
15 # The array in which the histograms are stored.
16 histograms = list()
17 # The bins for each histogram.
18 bins = np.logspace(-4, np.log10(5), 21)
19 # The histogram with the maximum bin.
20 max_hist_idx = 0
21 # The bin with the maximum number of counts.
22 max_bin_idx = 0
23
24
25 for idx in range(radii_matrix.shape[0]):
26
27     # Create the histogram and save it.
28     histogram = np.histogram(radii_matrix[idx], bins = bins)[0]
29     histograms.append(histogram)
30
31     # Loop over all bins in this new histogram.
32     for bin_idx, bin_counts in enumerate(histogram):
33
34         # Check if a bin in the new histogram contains
35         # more counts than the known maximum and save the bin index and the
36         # histogram index if this is true.
37         if histograms[max_hist_idx][max_bin_idx] < bin_counts:
38             max_hist_idx = idx
39             max_bin_idx = bin_idx
40
41 # For each histogram take the bin with the index of max_bin_idx
42 # and save the counts in an array.
43 bin_counts = np.array([hist[max_bin_idx] for hist in histograms])
44 bin_counts = bin_counts
45
46 # Print the median, 16th percentile and 84th percentile for the counts.
47 print('Median: {0:.1f}'.format(utils.median(bin_counts)))
48 print('Percentile 16th: {0:.1f}'.format(utils.percentile(bin_counts, 16)))
49 print('Percentile 84th: {0:.1f}'.format(utils.percentile(bin_counts, 84)))
50
51 # Plot the histogram for the counts with bin width of 1.
52 plt.hist(bin_counts, density=True,
53         bins=np.arange(min(bin_counts), max(bin_counts) + 1, 1),
54         edgecolor='black')
55
56 #
57 # Overplot the Poisson distribution.
58 #
59
60 # Mean events
61 mean = int(sum(bin_counts)/len(bin_counts))
62 x = np.arange(1, 60, 1)
63
64 # Poisson function can't take arrays :(
65 y = [utils.poisson(mean, float(x_val)) for x_val in x]
66 plt.plot(x, y, label='Poisson(mean:{0:.1f})'.format(mean))
67 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 array 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):

```

```

61         result[result_idx:] = left_sorted[left_idx:]
62         break
63
64     return result

```

./code/mathlib/sorting.py

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

```

1  import mathlib.sorting
2  import numpy as np
3
4  def factorial(number):
5      """
6          Calculate the factorial for a number
7
8      In:
9          param: number — The number to calculate the factorial for.
10     Out:
11         return: The factorial of the input number.
12     """
13
14     # If number < 20 convert it to an int64 to not lose any precision.
15     # else use a np.float64. (= lose in precision)
16     # Note: Only datatypes with 64 or less bits are allowed.
17     if number < 20:
18         number = np.int64(number)
19     else:
20         number = np.float64(number)
21
22     if number < 0:
23         raise RuntimeError('Invalid input for factorial: {n}. \
24                               Input must be larger or equal than zero.')
25     elif np.int64(number) == 0: #!0 = 1
26         return 1
27     else:
28         return number*factorial(number-1)
29
30
31  def poisson(average, events):
32      """
33          Evaluate the Poisson distribution.
34
35      In:
36          param: average — The average number of events.
37          param: events — The number of observed events.
38
39      Out:
40          return: The evaluation of the Poisson distribution
41                  for the given parameters.
42      """
43
44     return (average**events * np.exp(-average))/factorial(events)
45
46
47  def median(array):
48      """
49          Determine the median of an array
50
51      In:
52          param: array — The array to determine the median of.
53      Out:
54          return: The median of the array.
55      """
56
57     # Create a sorted version of the array.
58     sort = mathlib.sorting.merge_sort(array)
59     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     # Sort the array and determine the amount of elements.
82     sort = mathlib.sorting.merge_sort(array)
83     elements = len(array) - 1
84
85     # The index in the array that corresponds with the percentile.
86     # This is a decimal (i.e 1.2345)
87     idx_unrounded = (value*elements/100)
88
89     # The rounded index (i.e 1)
90     idx_rounded = int(idx_unrounded)
91
92     # The decimals of the unrounded index.
93     # This indicates the place between index 'idx_rounded'
94     # and 'idx_rounded+1' at which the interpolated value
95     # should be determined to obtain the percentile.
96     idx_decimals = idx_unrounded - idx_rounded
97
98     # Interpolation fails for 100th percentile, as
99     # idx_rounded+1 doesn't exist.
100     if value == 100:
101         return sort[elements]
102
103     # Determine the interpolated value.
104     return idx_decimals*(sort[idx_rounded+1] - sort[idx_rounded]) + sort[
105         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`

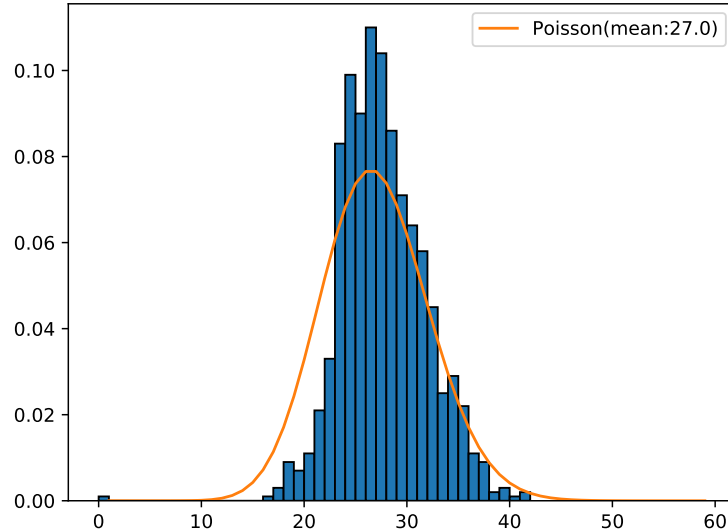


Figure 6: The Poisson distribution over-plotting the created histogram. The figure shows that the number of satellites on a spherical shell at a distance x is approximately distributed according to a Poisson distribution around $N(x)$ - the number of satellites predicted by the model on an infinite thin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

Question 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 question 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.

```
1 import numpy as np
2 import mathlib.interpolate3D as intp3d
3 import mathlib.integrate as intgrt
4
5 # Values of the a, b and c parameters.
6 a_values = np.arange(1.1, 2.6, 0.1) #15
7 b_values = np.arange(0.5, 2.1, 0.1) #16
8 c_values = np.arange(1.5, 4.1, 0.1) #26
9
10 # Row is a , column is b , depth is c.
11 table = np.zeros((len(a_values), len(b_values), len(c_values)))
12
13 for row, a_val in enumerate(a_values):
14     for column, b_val in enumerate(b_values):
15         for depth, c_val in enumerate(c_values):
16
17             # The function to normalize (simmlar as in assigment 2.a)
18             # This function unfortunatly couldn't be split over multiple
19             # lines so it might look slightly ugly in the report.
20             density = lambda x: b_val**2 *(x/b_val)**(a_val-1)*np.exp(-(x/b_val
21 )**c_val)*4*np.pi
22
23             # Insert the value in the table.
24             table[row][column][depth] = 1/intgrt.romberg(density, 0, 5, 15)
25
26 # Save the table to the disk to use it later in the next assignment.
27 np.save('table_2h', table)
```

./code/assignment2_h.py

Code - interpolator

The code for the 3D interpolator

```
1 import numpy as np
2 import mathlib.interpolate as intrp
3
4 class Interpolate3D(object):
5     """
6     A class representing a 3D interpolator.
7     """
8
9     def __init__(self, tensor, ranges):
10         """
11         Create a new instance of the 3D interpolator.
12
13         param: tensor — A tensor containing the known  $y = f(a,b,c)$  values
14         param: ranges — A matrix containing the a, b and c values of the
15         tensor.
16         """
17
18         self._tensor = tensor
19
20         # Unpack the a, b, c values.
21         self._row_ranges = ranges[0]
22         self._column_ranges = ranges[1]
23         self._depth_ranges = ranges[2]
24
25     def interpolateLin(self, a, b, c):
26         """
27         Estimate the point  $y = f(a, b, c)$  using linear interpolation.
28
29         In:
30         param: a      The a coordinate to interpolate for.
```



```

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 = 0xFFFFFFFF
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

Question 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,

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

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,

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

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,

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

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,

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

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,

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

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

```

1 import mathlib.utils
2 import mathlib.integrate
3 import mathlib.minimize as minimize
4 import mathlib.interpolate3D
5 import matplotlib.pyplot as plt
6 import numpy as np
7
8 #The 3D interpolator for the normalization constant
9 interpolator = None
10
11 def main():
12     global interpolator
13
14     # Load the 'table' created in the previous assignment and use it
15     # to create the 3d interpolator.
16     table = np.load('table_2h.npy')
17     # The a, b, c ranges of the 'table'.
18     ranges = [np.arange(1.1, 2.6, 0.1),
19               np.arange(0.5, 2.1, 0.1),
20               np.arange(1.5, 4.1, 0.1)]
21
22     # Create the interpolator.
23     interpolator = mathlib.interpolate3D.Interpolate3D(table, ranges)
24
25     # Go through each file and find the optimum.
26     for file in ['satgals-m15.txt', 'satgals-m14.txt', 'satgals-m13.txt',
27                 'satgals-m12.txt', 'satgals-m11.txt']:
28
29
30         # Read the file with halos and create a super halo.
31         super_halo = np.loadtxt(file, skiprows=4)[: , 0]
32
33         # The number of satelites in the super halo.
34         Nsat = len(super_halo)
35
36         # A wrapper around the likelihood function. This is used
37         # function that doesn't deppand on Nsat and super halo.
38         likelihood_wrapper = lambda a, b, c: likelihood(Nsat, super_halo, a, b,
39 c)
40
41         # Find the optimal values.
42         a_opt, b_opt, c_opt = minimize.downhill-simplex(likelihood_wrapper,
43                                                         [1.1, 1.3, 3.1],
44                                                         max_iters=200)
45
46         # Print the file and the output.
47         print('File: {0} a_opt:{1:.5f} b_opt:{2:.5f} c_opt:{3:.5f}'
48               .format(file, a_opt, b_opt, c_opt))
49
50         # Plot the results for the optimal found values.
51         bins = np.logspace(-4, np.log10(5), 21)
52         x = np.linspace(min(super_halo), 5, 100)

```

```

53     plt.plot(x, N(Nsat,x,a_opt,b_opt,c_opt)/Nsat ,
54              c = 'red', label = 'Model')
55     plt.hist(super_halo, bins= bins, density = True,
56              label = 'Data', edgecolor = 'black')
57
58     plt.xlim(min(super_halo)/5, max(super_halo)*30)
59     plt.ylim(bottom=10*(-20), top=10)
60     plt.legend(framealpha=1, loc= 1)
61     plt.loglog()
62     plt.savefig('./plots/' + file[0:-4] + '.pdf')
63     plt.figure()
64
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)
93     **c)*Nsat*4*np.pi
94
95 def likelihood(Nsat, x_values, a, b, c):
96     """
97     Calculate the negative loglikelihood for
98     the model given the total amount of satelites ,
99     their x positions and the model parameters.
100
101     In:
102     param: Nsat — The total number of satelites.
103     param: x_values — The position of all the satelites.
104     param: a — The first model parameter.
105     param: b — The secon model parameter.
106     param: c — The third model parameter.
107
108     Out:
109     return: The value of the negative loglikelihood evaluated for
110            the given input.
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.

```

```

122     if b < 0.5:
123         b = 0.5
124     elif b > 2.0:
125         b = 2
126
127     # Only optimise within the bounded region 1.5 <= c <= 4. Values
128     # outside the bounds are put at the edge.
129     if c > 4:
130         c = 4
131     elif c < 1.5:
132         c = 1.5
133
134     #return the negative loglikelihood.
135     return - np.sum(np.log(N(Nsat, x_values, a, b, c))) + Nsat
136
137
138
139 if __name__ == "__main__":
140     # call the entry function of this script
141     main()

```

./code/assignment3_a.py

Code - downhill

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def downhill-simplex(function, start, max_iters = 100, toll = 1e-10):
5     """
6         Try to find the minimum of the provided function
7         with the help of the downhill simplex method
8     In:
9         param:function — The function to minimize.
10        param:start — The best guess for the minimum.
11        param:max_iters — The maximum number of iterations to perform.
12        param:toll — The minimum tolerance for convergence.
13    Out:
14        return: An approximation for the minimum.
15    """
16
17
18    # The dimension in which we are.
19    dim = len(start)
20
21    # A matrix representing the vertices.
22    vertices = np.zeros((dim+1, dim))
23    vertices[0] = start #first one is start point
24
25    # A matrix with function evaluations of the vertices.
26    evaluations = np.zeros(dim+1)
27    evaluations[0] = function(*vertices[0])
28
29    # An array representing a direction
30    # vector in our n-dimensional space.
31    direction_vector = np.zeros(dim)
32
33    # Create the simplex by selecting
34    # vertices around the starting point.
35    for i in range(1, dim+1):
36        direction_vector[i-1] = 1
37        vertices[i] = start + 0.1*direction_vector
38        evaluations[i] = function(*vertices[i])
39        direction_vector[i-1] = 0
40
41    while max_iters >= 0:
42
43        max_iters -= 1
44

```

```

45 # Order the vertices such that  $f(x_0) \leq f(x_1) \leq f(x_2) \dots \leq f(x_n)$ 
46 # Just use selection sorts, even in 50 dimensions it is just of  $O(2500)$ 
47 for i in range(0, vertices.shape[0]):
48
49     # Loop over the remaining vertices
50     for j in range(i+1, vertices.shape[0]):
51
52         # If the function evaluation is smaller
53         # at the remaining vertex, then swap the vertices
54         if evaluations[j] < evaluations[i]:
55             tmp_1 = vertices[i].copy()
56             tmp_2 = evaluations[i]
57             vertices[i] = vertices[j]
58             evaluations[i] = evaluations[j]
59             vertices[j] = tmp_1
60             evaluations[j] = tmp_2
61
62     # Check if 'converged'
63     if (abs(evaluations[dim] - evaluations[0]))/(abs(evaluations[dim] +
64 evaluations[0])/2) < toll:
65         return vertices[0]
66
67     # Determine the centroid.
68     centroid = (1/(dim))*np.sum(vertices[0:dim], axis = 0)
69
70     # Try a new reflected point.
71     try_point = 2*centroid - vertices[dim]
72     try_eval = function(*try_point)
73
74     # if  $f(x_0) \leq f(x_{try}) < f(x_n)$ 
75     # then the new point is better but not the best, accept it.
76     if evaluations[0] <= try_eval < evaluations[dim]:
77         vertices[dim] = try_point.copy()
78         evaluations[dim] = try_eval
79         continue
80
81     # Expansion
82
83     # New point is the very best, propose
84     # a second point by expanding further in that
85     # direction.
86     if try_eval < evaluations[0]:
87         expand = 2*try_point - centroid
88         expand_eval = function(*expand)
89
90     # Expansion is even better
91     if expand_eval < try_eval:
92         vertices[dim] = expand.copy()
93         evaluations[dim] = expand_eval
94     else:
95         vertices[dim] = try_point.copy()
96         evaluations[dim] = try_eval
97
98     continue
99
100    # Contraction
101
102    # We now know that  $f(x_{try}) > f(x_{n-1})$  and
103    # therefore propose new point by contraction.
104
105    try_point = (centroid + vertices[dim])/2
106    try_eval = function(*try_point)
107
108    # Is it better than the worst point?
109    if try_eval < evaluations[dim]: # true
110        vertices[dim] = try_point.copy()
111        evaluations[dim] = try_eval
112        continue
113

```

```

114     # Shrink
115
116     # All previous points where bad, zoom in on the best point
117     # by shrinking.
118     for i in range(1, dim+1):
119         vertices[i] = (vertices[i]+ vertices[0])/2
120         evaluations[i] = function(*(vertices[i]))
121
122
123     return vertices[0]

```

./code/mathlib/minimize.py

Output - text

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

```

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

```

./output/assignment3_a_out.txt

Output - plots

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

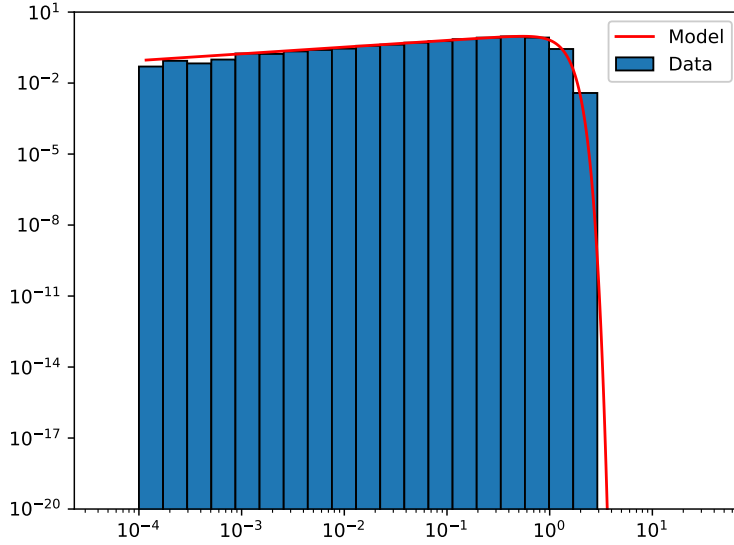


Figure 7: The model (equation 13) with the found parameters of a , b and c fitted to the data for the mass bin of $10^{11} M_{\odot}$.

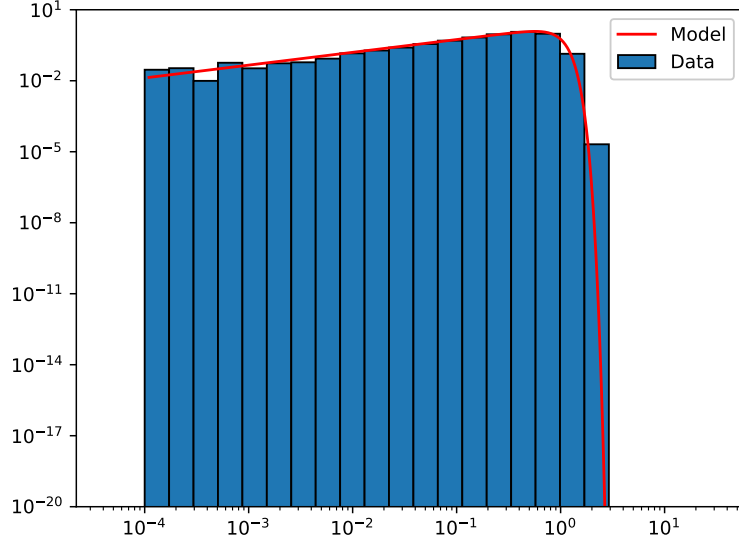


Figure 8: The model (equation 13) with the found parameters of a , b and c fitted to the data for the mass bin of $10^{12} M_{\odot}$.

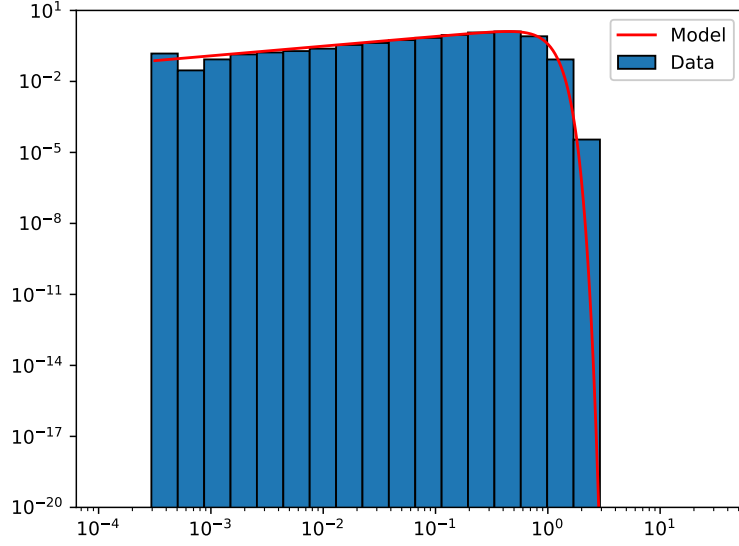


Figure 9: The model (equation 13) with the found parameters of a , b and c fitted to the data for the mass bin of $10^{13} M_{\odot}$.

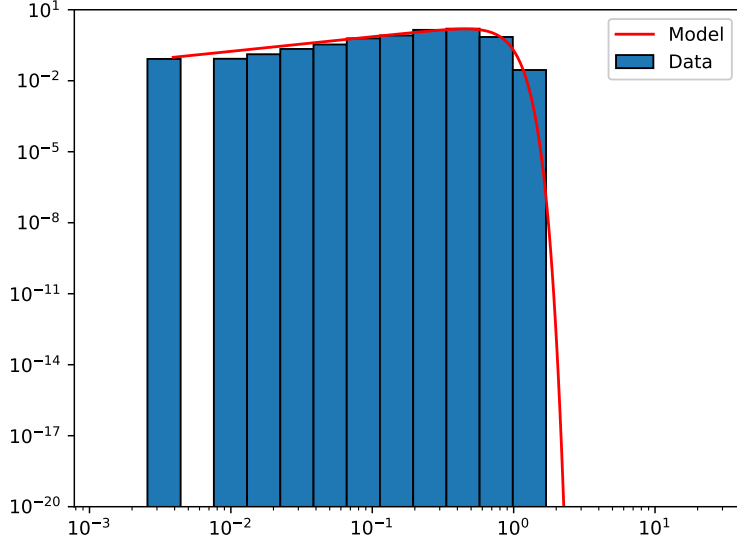


Figure 10: The model (equation 13) with the found parameters of a , b and c fitted to the data for the mass bin of $10^{14} M_{\odot}$.

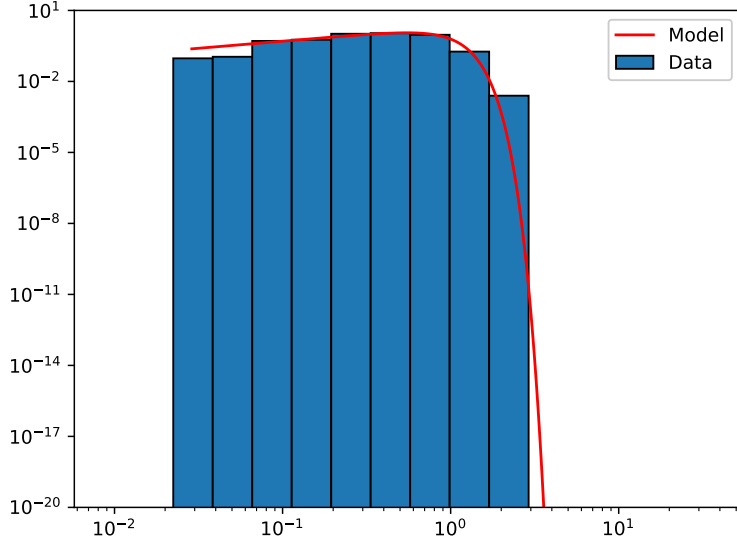


Figure 11: The model (equation 13) with the found parameters of a , b and c fitted to the data for the mass bin of $10^{15} M_{\odot}$.