

NUR solution template

Folkert Nobels

April 9, 2019

Abstract

In this document an example solution template is given for the exercises in the course Numerical recipes for astrophysics.

1 Exercise 1

For this first exercise we had to write our own Poisson probability distribution function and random number generator. The output of the Poisson function can be seen in the printed text (see below) and the tests for the rng can be found in 1 and 2.

1.1 Scripts

The functions that I wrote for this exercise can be found in:

```
1 #NR_a1_1_utils.py
2 import numpy as np
3 import sys
4 import math # Only used for math.isnan() in NewRaph_rootfinder()
5
6 def poisson_distribution(mean,k):
7     fact = np.float(0)
8     for i in range(round(k)):
9         fact += np.log(k-1)
10    fact = np.exp(fact)
11    return (mean**k*np.exp(-mean))/fact
12
13 def poisson_distribution_new(mean,k):
14 #Returns probability for given k and mean
15     fact = 1
16     mag = 0
17     prev_n = 0
18     for i in range(round(k)):
19         temp = str(fact*(k-i))
20         n_zeros = len(temp)-1-prev_n
21         fact = int(temp[:10])
22         mag += n_zeros
23         prev_n = len(temp[:10])-1
24
25     a = str(mean**k)
26     b = str(np.exp(mean))
27
28     x = (float(a[:10])/1e9 * 1/float(b[:10]))/(fact/10**(prev_n-1))
29     mag_tot = -int(np.log10(float(b)))+len(a)-mag
30     return x*10**mag_tot
31 # Only works for massive numbers.....
32 #end poisson_distribution()
33
34 class rng(object):
35 # rng object that is initiated with a give seed
36     def __init__(self, seed):
37         self.state = np.int64(seed)
38
39     def LCG_gen(self):
```

```

40 #Linear Congruential generator
41 x = self.state
42 a,c,m = 2**32,1664525,1013904223
43 self.state = np.int64((a*x+c)%m)
44 #end LCG-gen()
45
46 def XOR_shift(self):
47 # XOR-shift generator
48 x = self.state
49 a1,a2,a3 = 21,35,4
50 x = x ^ x >> a1
51 x = x ^ x << a2
52 x = x ^ x >> a3
53 self.state = np.int64(x)
54 #end XOR-shift()
55
56 def rand_num(self,l,min=0,max=1):
57 # Generates 'l' random numbers between min and max
58 output = []
59 for i in range(l):
60 self.XOR_shift()
61 self.XOR_shift()
62 self.LCG_gen()
63 self.XOR_shift()
64 self.XOR_shift()
65 output.append(self.state)
66 output = np.array(output)/sys.maxsize
67 return min+(output*(max-min))
68 #end rand_num()
69 #end rng()
70
71 # — Simple supporting functions —
72
73 def min(l):
74 min = 2**64
75 for i in l:
76 if i < min:
77 min = i
78 return min
79 #end min()
80
81 def arg_min(l):
82 min = 2**64
83 arg = None
84 for i in range(len(l)):
85 if l[i] < min:
86 min = l[i]
87 arg = i
88 return arg
89 #end arg_min()
90
91 def max(l):
92 max = -2**64
93 for i in l:
94 if i > max:
95 max = i
96 return max
97 #end max()
98
99 def arg_max(l):
100 max = -2**64
101 arg = None
102 for i in range(len(l)):
103 if l[i] > max:
104 max = l[i]
105 arg = i
106 return arg
107 #end arg_max()

```

NR_a1_1_utils.py

The commands used to retrieve the desired results are given by:

```
1 #NR_a1_1_main.py
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import NR_a1_1_utils as utils
5
6 seed = 42
7 print('Original seed:',seed)
8
9 #--- 1.a ---
10 # Poisson distribution
11 print('1.a:')
12 a1 = [[1,0],[5,10],[3,20],[2.6,40]]
13 print('Simple poisson distribution:')
14 for i in range(len(a1)):
15     print('P({}) with mean {}:'.format(a1[i][1],a1[i][0]),utils.poisson_distribution(a1[
16     i][0],a1[i][1]))
17 print('Large-number poisson distribution (only works for large numbers):')
18 a1 = [[101,200]]
19 for i in range(len(a1)):
20     print('P({}) with mean {}:'.format(a1[i][1],a1[i][0]),utils.poisson_distribution_new
21     (a1[i][0],a1[i][1]))
22
23 #--- 1.b ---
24 print('1.b:')
25 # RNG
26 rng = utils.rng(seed)
27 # Scatter plot
28 N = 1000
29 rand = rng.rand_num(N)
30 plt.scatter(rand[:len(rand)-1],rand[1:])
31 plt.title('Sequential number plot for {} random numbers with seed {}'.format(1000,seed))
32 plt.savefig('plots/1_b_1.png')
33
34 # Histogram
35 N = 1000000
36 rand = rng.rand_num(N)
37 plt.hist(rand,bins=20,range=(0,1))
38 plt.title('Histogram of 1,000,000 randomly generated numbers'.format(1000,seed))
39 plt.xlabel('Number of numbers in bin')
40 plt.ylabel('Number values')
41 plt.savefig('plots/1_b_2.png')
42 print('Saving Histogram and scatter plot.')
```

NR_a1_1_main.py

The result of the given script is given by:

```
1 Original seed: 42
2 1.a:
3 Simple poisson distribution:
4 P(0) with mean 1: 0.36787944117144233
5 P(10) with mean 5: 1.887133131720812e-05
6 P(20) with mean 3: 4.618166957543755e-18
7 P(40) with mean 2.6: 6.717133577841103e-49
8 Large-number poisson distribution (only works for large numbers):
9 P(200) with mean 101: 1.2695314379023172e-18
10 1.b:
11 Saving Histogram and scatter plot.
```

NR_a1_1_main.txt

2 Exercise 2

For this section of the assignment we were asked to write a variety of different functions in order to probe the given probability function $n(x)$. These will be discussed per sub-question:

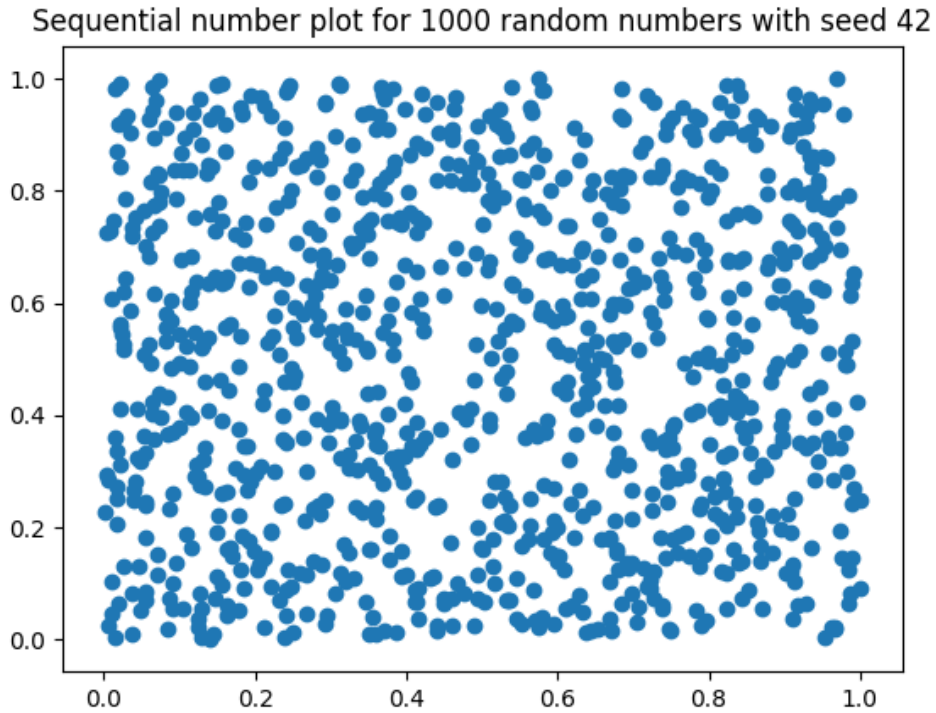


Figure 1: In this figure we can see that it appears that the random number generator is producing numbers without a certain preference.

2.1 Write a numerical integrator

In order to accomplish this task I wrote an integrator that uses Romberg's algorithm in order to numerically integrate a given function from a to b .

The output values are found in the print statement produced by the script.

2.2 Make a log-log plot and interpolate the given values

For this I used a linear-interpolator due to the fact that at first glance these values follow a linear trend in log-log space. Due to having too little time I did not manage to alter my interpolator to also work in log-space. For this reason the interpolated function systematically overshoots the fit one would expect. We can see this well in 3.

2.3 Numerically calculate $dn(x)/dx$ at $x = b$

For this task I wrote an algorithm that numerically differentiates using Ridder's method. The analytic and calculated numerical value can be found in the print statement.

2.4 Generate 3D satellite positions

In order to generate this distribution I used the rejection sampling method. The positions generated using this method can be found in the print statement.

2.5 Repeat (d) for 1000 haloes each containing 100 satellites

The histogram generated for this task can be found in 4. It appears that the generated galaxies match the distribution fairly well. It is only really lacking at the lower end of the distribution.

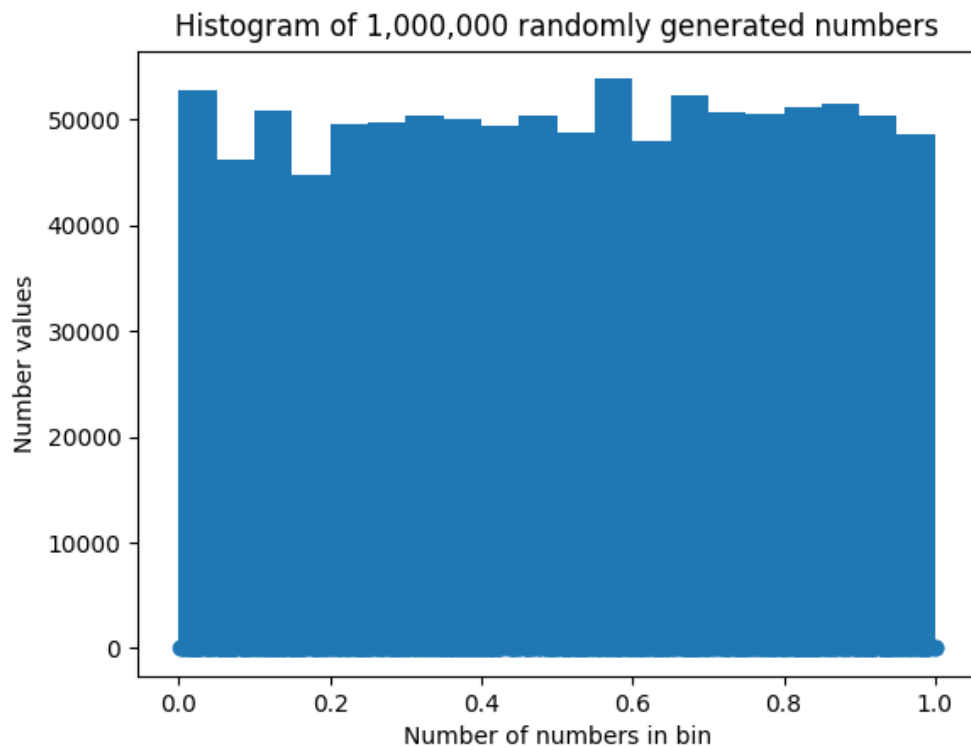


Figure 2: This histogram places the random number generator under a sharper knife, allowing us to see that there are some fluctuations between the bins. Overall it appears to be quite unbiased.

2.6 Write a root-finding algorithm

The method I used in order to find these roots is the Newton-Raphson method. This is because the false-position method and Secant method appeared not to be able to converge for this function. The values of the roots are given in the print output.

2.7 Take the bin from `e` containing the largest number of galaxies. Using sorting calculate the median, 16th and 84th percentile for this bin and output the values and plot the histogram

The calculated values for the median, 16th and 84th percentile can be found in the print statement.

The histogram produced for this task can be found in 5.

As you can (barely) see, the Poisson distribution does not match the histogram. This however is most likely due to a mistake in my Poisson distribution function.

2.8 Write an interpolator that gives values of `a` for different `a,b,c`

In order to accomplish this task I wrote a function for a trilinear-interpolator. The values I tested it with and the resulting `A` value can be found in the print statement.

2.9 Scripts

The functions that I wrote for this exercise can be found in:

```
1 #NR_a1.2_utils.py
2 import numpy as np
3 import sys
```

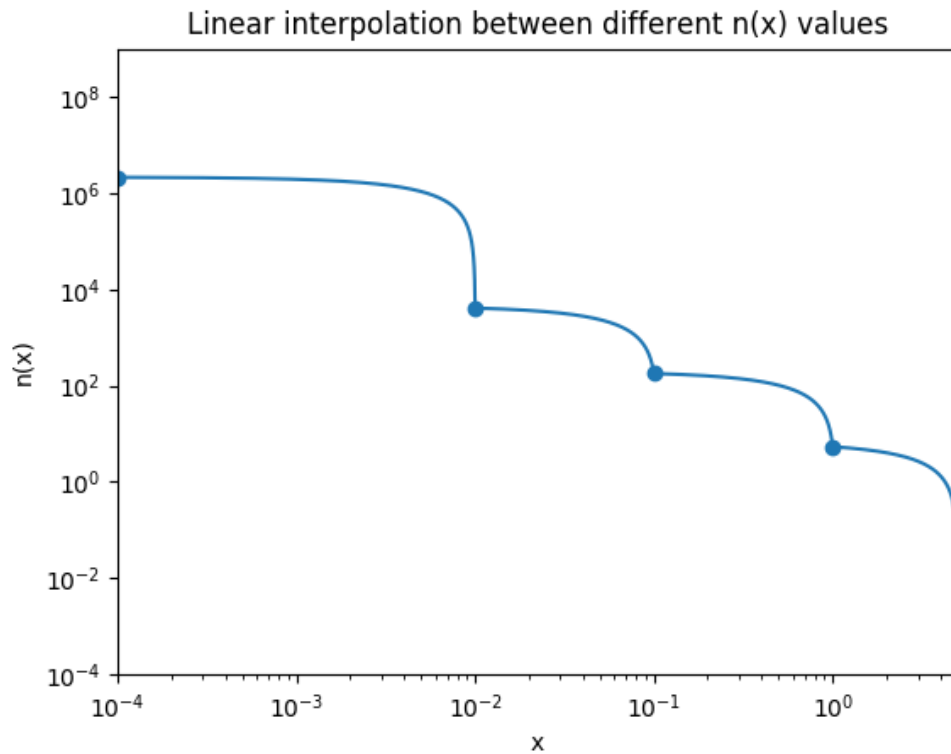


Figure 3: Here we see the way in which a linear interpolator overshoots the expected function when used in log-space.

```

4 import math # Only used for math.isnan() in NewRaph_rootfinder()
5 from NR_al_1_utils import poisson_distribution, rng, min, max, arg_max, arg_min
6
7 def central_diff(f,h,x):
8     # Calculates the central difference\n",
9     return (f(x+h)-f(x-h))/(2*h)
10 #end central_diff()
11
12 def ridders_diff(f,x):
13     # Differentiates using Ridder's method
14     m = 10
15     D = np.zeros((m,len(x)))
16     d = 2
17     h = 0.001
18     for i in range(m):
19         D_new = D
20         for j in range(i+1):
21             if j == 0:
22                 D_new[j] = central_diff(f,h,x)
23             else:
24                 D_new[j] = (d**(2*(j+1))*D[j-1]-D_new[j-1])/(d**(2*(j+1))-1)
25         D = D_new
26         h = h/d
27     return D[m-1]
28 #end ridders_diff()
29
30 def comp_trapezoid(f,a,b,n):
31     # Composite trapezoid rule used in romber_int()
32     h = 1/(2**(n-1))*(b-a)
33     sum = 0
34     for i in range(1,2**(n-1)):
35         sum += f(a+i*h)

```

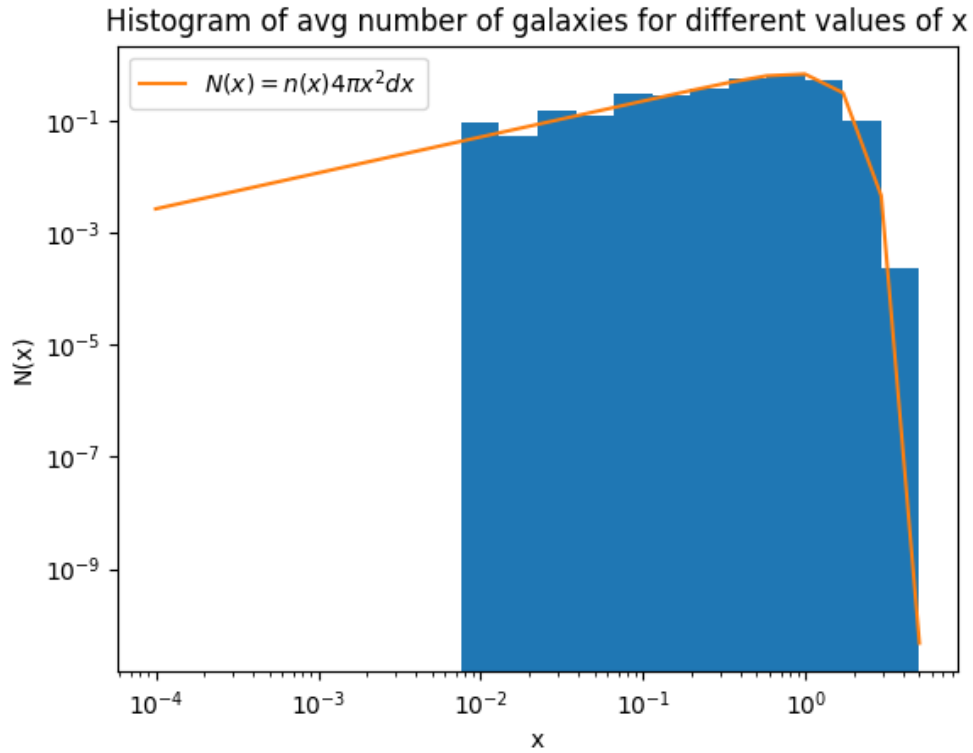


Figure 4: Histogram of number of satellites in each bin with over-plotted probability. Note the discrepancy at the lower end of the distribution.

```

36     return (h/2.)*(f(a)+2*sum+f(b))
37 #end comp_trapezoid()
38
39 def romber_int(f,a,b):
40     # Integrates from a to b up to an accuracy of 6 decimals
41     for n in range(1,10):
42         S_new = np.zeros((n))
43         S_new[0] = comp_trapezoid(f,a,b,n)
44         for j in range(2,n+1):
45             S_new[j-1] = (4**(j-1)*S_new[j-2]-S[j-2])/(4**(j-1)-1)
46         S = S_new
47         if n > 3:
48             if abs(S[-2]-S[-1]) < 1e-6:
49                 return S[-1]
50     return S[-1]
51 #end romber_int()
52
53 def interpol_lin_log(xj,yj,x,y):
54     # Linear interpolator
55     j=0
56     xj,yj,x = np.log10(xj),np.log10(yj),np.log10(x)
57     for i in range(len(x)):
58         if x[i]>xj[j]:
59             j+=1
60         if j>4:
61             return 10**y
62         y[i]=(((yj[j]-yj[j-1])/(xj[j]-xj[j-1]))*(x[i]-xj[j]))+yj[j])
63 #end interpol_lin()
64
65 def interpol_lin(xj,yj,x,y):
66     # Linear interpolator
67     j=0

```

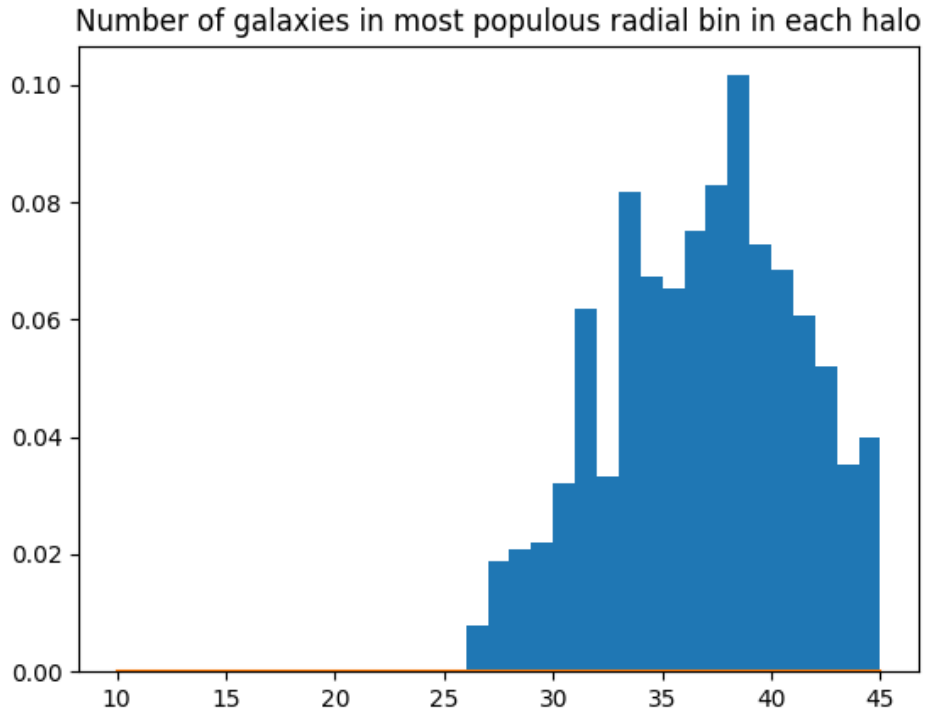


Figure 5: Number of galaxies in most populous radial bin in each halo. Note that the Poisson distribution is most likely calculated wrongly.

```

68     for i in range(len(x)):
69         if x[i] > xj[j]:
70             j += 1
71         if j > 4:
72             return y
73         y[i] = (((y[j] - y[j - 1]) / (x[j] - x[j - 1])) * (x[i] - x[j]) + y[j])
74 #end interpol_lin()
75
76 def neville_alg(x, xj, yj, i, j):
77 # Neville's Algorithm used in interpol_neville
78     if i == j:
79         return yj[i]
80     else:
81         return ((x - xj[j]) * neville_alg(x, xj, yj, i, j - 1) - (x - xj[i]) * neville_alg(x, xj, yj, i + 1, j))
82         / (xj[i] - xj[j])
83 #end neville_alg()
84
85 def interpol_neville(xj, yj, x, y):
86 # Interpolator that uses Neville's Algorithm
87     for z in range(len(x)):
88         y[z] = neville_alg(x[z], xj, yj, 0, len(xj) - 1)
89     return y
90 #end interpol_neville()
91
92 def rejection_sampler(n, p, max_x, max_y, rng):
93 # Rejection sampler that uses max_y value as g
94     sample_x = []
95     sample_y = []
96     n_acpt = 0
97     while n_acpt < n:
98         pot_x = np.float(rng.rand_num(1, max=max_x))
99         pot_y = np.float(rng.rand_num(1, max=max_y))

```



```

99         if pot_y <= p(pot_x):
100             sample_x.append(pot_x)
101             sample_y.append(pot_y)
102             n_accpt += 1
103     return sample_x, sample_y
104 #end rejection_sampler
105
106 def secant_rootfinder(f,a,b):
107     # Secant method root-finder
108     it = 0
109     while np.abs(b-a) > 0.00001:
110         it += 1
111         c = -(b-a)/float((f(b)-f(a))*f(a)+a
112         a = b
113         b = c
114     return a
115 #end secant
116
117 def falspos_rootfinder(f,a,b):
118     # False position method root-finder
119     it = 0
120     c = 0
121     while np.abs(b-a) > 0.001:
122         it += 1
123         c = -(b-a)/float((f(b)-f(a))*f(a)+a
124         print('a',a, ', b', b)
125         print('c',c)
126         if f(a)*f(c) > 0:
127             a = c
128         else:
129             b = c
130     return c, it
131 #end falspos_rootfinder()
132
133 def NewRaph_rootfinder(f,a,b,rng):
134     # Finds root in given range (a,b) for f
135     x = rng.rand_num(1,min=a,max=b)
136     x_new = sys.maxsize
137     for i in range(1000):
138         f_deriv = ridders_diff(f,np.array([x]))
139         if f_deriv == 0 or math.isnan(f_deriv):
140             x = rng.rand_num(1,min=a,max=b)
141             i = 0
142             continue
143         x_new = x - f(x)/f_deriv
144         if abs(x_new-x) < 1e-12:
145             return x_new
146         else:
147             x = x_new
148     return x
149 #end NewRaph_rootfinder()
150
151 def selection_sort(l):
152     # Ascending sorting of l using selection sort
153     sl = []
154     for i in range(len(l)):
155         min = arg_min(l)
156         sl.append(l[min])
157         l = l[:min]+l[(min+1):]
158     return sl
159 #end selection_sort()
160
161 def A_calc(a,b,c):
162     # Calculates A with given values of a,b,c
163     f = lambda x: 4*np.pi*(x**(a-1))/(b**(a-3))*np.exp(-(x/b)**c)
164     f_int = romber_int(f,0,5)
165     return 1/f_int
166 #end A_calc()
167
168 def trilinear_interpolator(al,bl,cl,Al,x,y,z):

```

```

169 # Returns an interpolated value for A1 based on a,b,c values
170 def bracket_finder(xl,x):
171     for i in range(len(xl)):
172         if xl[i] > x:
173             return xl[i-1],xl[i],i
174     print('Could not find a bucket, returning nan.')
175     return float('nan'),float('nan')
176
177 a0,a1,ai = bracket_finder(a1,x)
178 b0,b1,bi = bracket_finder(b1,y)
179 c0,c1,ci = bracket_finder(c1,z)
180
181 xd = (x-a0)/(a1-a0)
182 yd = (y-b0)/(b1-b0)
183 zd = (z-c0)/(c1-c0)
184
185 c00 = A1[ai-1][bi-1][ci-1]*(1-xd)+A1[ai][bi-1][ci-1]*xd
186 c01 = A1[ai-1][bi-1][ci]*(1-xd)+A1[ai][bi-1][ci]*xd
187 c10 = A1[ai-1][bi][ci-1]*(1-xd)+A1[ai][bi][ci-1]*xd
188 c11 = A1[ai-1][bi][ci]*(1-xd)+A1[ai][bi][ci]*xd
189
190 c0 = c00*(1-yd)+c10*yd
191 c1 = c01*(1-yd)+c11*yd
192
193 return c0*(1-zd)+c1*zd
194 #end trilinear_interpolator()

```

NR_a1.2_utils.py

The commands used to retrieve the desired results are given by:

```

1 #NR_a1.2_main.py
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import NR_a1.2_utils as utils
5
6 seed = 42
7 print('Original seed:',seed)
8 rng = utils.rng(seed)
9
10 #--- 2.a ---
11 print('2.a:')
12 a = rng.rand_num(1,min=1.1,max=2.5)
13 b = rng.rand_num(1,min=0.5,max=2)
14 c = rng.rand_num(1,min=1.5,max=4)
15 f = lambda x: 4*np.pi*(x**(a-1))/(b**(a-3))*np.exp(-(x/b)**c)
16 f_int = utils.romber_int(f,0,5)
17 A = 1/f_int
18 print('A = {}; a,b,c = {},{},{},{}'.format(A,float(a),float(b),float(c)))
19
20 #--- 2.b ---
21 print('2.b:')
22 xj = [10**-4,10**-2,10**-1,1,5]
23 n_x = lambda x: A*100*(x/b)**(a-3)*np.exp(-(x/b)**c)
24 n = n_x(xj)
25 x = np.logspace(np.log10(1e-4),np.log10(5),10000)
26 y = np.zeros(10000)
27 y_lin = utils.interpol_lin(xj,n,x,y)
28 plt.scatter(xj,n)
29 plt.plot(x,y_lin)
30 plt.xlim(left=10**-4,right=5)
31 plt.ylim(bottom=1e-4,top = 1e9)
32 plt.xscale('log')
33 plt.yscale('log')
34 plt.xlabel('x')
35 plt.ylabel('n(x)')
36 plt.title('Linear interpolation between different n(x) values')
37 plt.savefig('plots/2_b.png')
38 plt.close()
39 print('Saved interpolation plot to \'plots/2_b.png\'')
40

```

```

41 #--- 2.c ---
42 print('2.c:')
43 n = lambda x: A*100*(x/b)**(a-3)*np.exp(-(x/b)**c)
44 x = b
45 dndx = utils.ridders_diff(n,np.array([b]))
46 dndx_analitic = lambda x: (A*100) * (((a-3)*(x/b)**(a-4)*np.exp(-(x/b)**c))/b - ((c*np.
47     exp(-(x/b)**c)*(x/b)**(a+c-4))/b))
48 dndx_an = dndx_analitic(x)
49 print('dn/dx at x = b: analytic = {0:.12f}; numerical = {1:.12f}'.format(float(dndx_an),
50     float(dndx)))
51 #--- 2.d ---
52 print('2.d:')
53 N = 100
54 xmax = 5
55 # Drawing random samples from n(x)
56 pn = lambda x: (n(x)*4*np.pi*x**2)/100
57 x_p = np.linspace(0,xmax,200)
58 g = np.max(pn(x_p)[1:])+0.01
59 samples = utils.rejection_sampler(N,pn,5,g,rng)
60 r = samples[0]
61 # Generating random angles:
62 phi = rng.rand_num(N,min=0,max=2*np.pi)
63 theta = np.arccos(2*rng.rand_num(N)-1)
64 x,y,z = r*np.sin(theta)*np.cos(phi),r*np.sin(theta)*np.sin(phi),r*np.cos(theta)
65 # Plotting positions for N galaxies
66 #ax = plt.figure().add_subplot(111,projection='3d')
67 #ax.scatter(x,y,z)
68 #plt.show()
69 print()
70 print('r,phi,theta:')
71 for i in range(len(r)):
72     print(r[i],phi[i],theta[i])
73
74 #--- 2.e ---
75 print('2.e:')
76 N = 100000
77 samples = utils.rejection_sampler(N,pn,5,g,rng)
78 r = samples[0]
79 bins = np.logspace(np.log10(1e-4),np.log10(xmax),num=21)
80 plt.hist(r,bins=bins,density=True)
81 plt.plot(bins,pn(bins),label = '$N(x) = n(x)4\pi x^2 dx$')
82 plt.yscale('log')
83 plt.xscale('log')
84 plt.ylabel('N(x)')
85 plt.xlabel('x')
86 plt.legend()
87 plt.title('Histogram of avg number of galaxies for different values of x')
88 plt.savefig('plots/2_e.png')
89 plt.close()
90 print('Saved histogram to \'plots/2_e.jpg\'')
91
92 #--- 2.f ---
93 print('2.f:')
94 dpndx = utils.ridders_diff(pn,x)
95 dpndx_analytic = lambda x: A*4*np.pi*(np.exp(-(x/b)**c)*(((a-1)*b**(3-a)*x**(a-2))-(c*b
96     *(2-a)*x**(a-1)*(x/b)**(c-1))))
97 dpndx_0 = float(utils.NewRaph_rootfinder(dpndx_analytic,1e-4,1,rng))
98 new_floor = float(pn(dpndx_0)/2)
99 pn_new_floor = lambda x: pn(x) - new_floor
100 root1 = float(utils.NewRaph_rootfinder(pn_new_floor,1e-4,dpndx_0,rng))
101 root2 = float(utils.NewRaph_rootfinder(pn_new_floor,dpndx_0,5,rng))
102 print('Roots:', root1,root2)
103
104 #--- 2.g ---
105 print('2.g:')
106 counts = np.zeros((len(bins)-1))
107 for i in r:

```

```

108     for j in range(len(bins)-1):
109         if i < bins[j+1] and i > bins[j]:
110             counts[j] += 1
111 r_list = []
112 r_halo_distrib = np.zeros((1000))
113
114 for i in range(len(r)):
115     if r[i] < bins[utils.argmax(counts)+1] and r[i] > bins[utils.argmax(counts)]:
116         r_list.append(r[i])
117         r_halo_distrib[i//100] += 1
118 mean = sum(r_halo_distrib)/len(r_halo_distrib)
119 halo_bins = np.linspace(10,45,36)
120 poissd = []
121
122 for i in range(len(halo_bins)):
123     poissd.append(utils.poisson_distribution(round(mean),int(halo_bins[i])))
124
125 plt.hist(r_halo_distrib,halo_bins,density=True)
126 plt.plot(halo_bins,poissd)
127 plt.title('Number of galaxies in most populous radial bin in each halo')
128 plt.savefig('plots/2-g.png')
129 print('Saved histogram to \'plots/2-g.jpg\'')
130 print('For some reason the poisson distribution does not work correctly here and I don\'
131       t know why. It does appear that the histogram follows a fairly normal distribution,
132       which is a good sign.')
131 plt.close()
132
133 sr = utils.selection_sort(r_list)
134
135 median = sr[int(len(sr)/2-0.5)]
136 p16th = sr[round(len(sr)*0.16)-1]
137 p84th = sr[round(len(sr)*0.84)-1]
138 print('Length: {}, median: {}, 16th: {}, 84th: {}'.format(len(sr),median,p16th,p84th))
139
140 # --- 2.h ---
141 print('2.h:')
142 al = np.linspace(1.1,2.5,15)
143 bl = np.linspace(0.5,2,16)
144 cl = np.linspace(1.5,4,26)
145 param = np.array((al,bl,cl))
146 Al = np.zeros([len(al),len(bl),len(cl)])
147 for i in range(len(al)):
148     for j in range(len(bl)):
149         for k in range(len(cl)):
150             Al[i][j][k] = utils.A_calc(al[i],bl[j],cl[k])
151
152 interpol = utils.trilinear_interpolator(al,bl,cl,Al,2.05,1.05,3.05)
153 print('Interpolator was tested with following values:')
154 print('For a = {}, b = {}, c = {}, the interpolator returned A = {}'.format
155       (2.05,1.05,3.05,interpol))

```

NR_a1_2_main.py

The result of the given script is given by:

```

1 Original seed: 42
2 2.a:
3 A = 0.04701878985790245; a,b,c =
4     1.6419613428029698,1.4577252811360346,2.6047059307084877
5 2.b:
6 Saved interpolation plot to 'plots/2-b.png'.
7 2.c:
8 dn/dx at x = b: analytic = -4.702159549813; numerical = -4.702159549822
9 2.d:
10 r,phi,theta:
11 0.7311141707122876 1.6028076635948272 0.2270265285598231
12 2.197278770187982 5.29883706734633 1.3425005017754366
13 1.261066210848445 0.4705410840621936 1.652797361141153
14 0.6558716361204144 5.012800547859235 2.280159240554945
15 0.7824897836202369 3.2051614706482767 1.109519113847493

```

16	0.5641889930754497	4.239148403641524	2.5313275914354216
17	0.7455003237080227	1.3378904226799089	2.07121896558909
18	0.9233498635882491	0.13554910534546938	0.8080191138718723
19	0.6646036870484072	2.593921050894136	0.6430905972730414
20	1.422587656024743	1.5936802052506487	1.1961750855480042
21	1.2151205803427914	3.3866416280089577	1.371971552370749
22	0.2683698883997805	3.983288228149664	2.34892948705019
23	0.5195498142643846	0.3023185592993231	0.20783067746454917
24	0.43194175748799657	2.0002705470795115	2.114834963001796
25	0.45944451540419623	3.701972044276426	2.720290327493936
26	0.8006572399002244	4.274590237546436	1.7464117024831116
27	0.2804255826180151	2.086678285518562	1.2565200536872243
28	1.7281651532360687	1.493229941026918	1.5682490721280593
29	0.4403162291018148	5.51406175002302	2.3134558412397968
30	0.28005365086933415	4.1667562336864306	1.2754577518288508
31	0.629885215859386	3.126371193501809	0.8823133533723011
32	1.3341331154888179	3.744806626025325	2.431634242424714
33	1.8027138385254526	3.577180243900044	1.47799705232885
34	0.2378761548099098	0.21519768812236728	2.7532126716994534
35	0.7747006558042651	3.686789510644586	1.1150685587388856
36	0.6869077696133774	3.636542606711427	1.8470806713450534
37	1.4117884436667754	3.7242596526259963	0.891581350772273
38	0.6294095410752514	1.2088053560981842	0.8448593580067348
39	1.2764752027254045	1.098289778440001	1.4188595428183999
40	1.6150927683803165	1.9199926678160373	2.8470316361511516
41	0.4186689992245968	5.637045372231429	1.9593635789694905
42	0.5461799853935555	2.2252698686054537	0.17807153021610092
43	0.6829810271722446	0.24992055211606554	1.2590866128292584
44	0.6234502914389102	1.5575463076869536	1.195224966142777
45	1.3366949341522365	3.0485171580618378	0.5212172093727148
46	0.6499529248316984	2.5940784042398093	1.453743194278108
47	0.9067285203917376	2.132532890780963	0.7257116272553009
48	1.5146136209601937	2.404549315401679	1.978017257685901
49	0.2770245026144908	6.164596891659123	1.4694682169604514
50	0.5159651931040894	2.3253515653634365	1.1861233069219161
51	0.4759216002517306	6.047620429111068	0.5387480016708217
52	2.247016406983809	0.13409749171900837	1.3807219348619766
53	1.5602946396462583	6.218389640083777	2.186858237035807
54	2.3029029642531844	3.3462652294283277	0.4412630335676338
55	0.6965374980804753	0.4498323703870266	0.43329541725053206
56	1.1409926578391123	0.7249796633728953	0.9982803668408025
57	0.6653952740755197	1.0253133385140492	1.9986788291368351
58	1.6205144185041889	1.005557374832275	0.4313321487854299
59	1.0420429483652858	3.4936712489898514	1.1563665983921778
60	1.8162715311433768	4.398841225589391	1.257522673876321
61	1.753618683902485	3.9458599495477777	2.8651583133590033
62	0.9075665701160043	5.367493156386626	2.631654291756891
63	2.196964042542822	2.597252707137583	1.724099860613247
64	1.335804466710699	0.4204979764301613	0.7175964769213862
65	0.1827132537627097	2.241282270808069	0.853186798298481
66	0.7038700495210275	2.3096897199004744	1.9112929888501538
67	1.7123198598059761	6.120340636505855	1.8682558561336253
68	2.079763440168249	2.150644043039724	2.314405830493339
69	1.8501341524498836	4.948950050730439	2.7251108593102016
70	0.8911001722393672	0.8143685684732586	2.022113824428344
71	1.091892711619008	5.552002315953939	1.7871299593017478
72	2.0542049463220273	3.7792343179439785	2.455513735502
73	1.3211167003377824	2.8219224435250907	0.8302843336903301
74	1.5889036994914583	3.862410859025583	0.20885659658253541
75	0.9993672674922676	4.641443887553411	1.2991381763462724
76	0.21591307901877496	5.32616256645649	2.161269458308147
77	1.6492796030749581	3.2963015420602892	2.239559557821826
78	1.4581203252216812	2.927113955421931	1.8842855313416325
79	2.4482548492097473	4.3877153728887714	1.532180548712544
80	1.9948482716434734	0.8685057783736273	0.5763854455089538
81	0.9218681038047252	3.987296975053835	0.5972292423329523
82	0.44154526532833743	3.2402812512358494	1.1749082354327303
83	0.18173811352986738	1.7758883998026385	1.3174654425683672
84	0.5187689966449206	4.543296053760293	1.7492149796888092
85	1.3650977926903596	0.6726655795284455	2.6194128692669856

```

86 1.4861118717435184 5.627875870996373 0.6574243816564097
87 1.5804517116615513 1.2429693794730885 1.8099719124226885
88 0.7420420489988011 5.30214916092477 0.8281478177090664
89 1.3314795740982264 2.3793285612306017 2.3801303781445413
90 0.6407129680532632 0.8940786201724139 1.7575270804295209
91 0.7322550000028955 5.290332743081566 1.6484916266608831
92 0.3504764783590363 6.087384004395324 2.8246659302385067
93 1.4659929210709874 6.282353866008897 2.402429041978465
94 1.9332396764300799 0.5667864302431423 1.5154492416588403
95 1.8020963918757016 0.6313214477879985 0.9514025960075138
96 1.1281752377936851 3.4268531372170266 2.1126890895625485
97 0.8940159908580776 4.507780842667583 1.2803535540003392
98 0.5783915184826695 6.095133351100733 1.2642369297377836
99 0.8172476257165974 4.896468153276877 1.5065904525286185
100 1.5578782882469553 2.94710820289594 0.653235189348019
101 0.7787656442861324 5.320419139190734 1.771281526346205
102 0.4080652971728931 4.270079657246644 1.8380359213790045
103 0.07337931053185985 2.6303052090296744 2.031693926313672
104 0.9549057503078782 2.189789006551835 2.1302858225822474
105 1.0740411498593938 4.667855605228215 2.1200254410634685
106 1.181490451901903 0.912168796509376 3.002947455960485
107 1.970841873281359 0.05396950164989878 2.0157468651014
108 1.415274167720679 4.574893947574355 1.2887032759180237
109 0.8888173153903209 0.7663566943066645 1.6355426458896933
110 1.4996123211828571 1.9576845334443223 2.49707290293986
111 2.e:
112 Saved histogram to 'plots/2-e.jpg'.
113 2.f:
114 Roots: 0.1987318674472368 1.6407030708001724
115 2.g:
116 Saved histogram to 'plots/2-g.jpg'.
117 For some reason the poisson distribution does not work correctly here and I don't know
    why. It does appear that the histogram follows a fairly normal distribution, which
    is a good sign.
118 Length: 37615, median: 1.2773704178216243, 16th: 1.0771942122640141, 84th:
    1.5217983937551454
119 2.h:
120 Interpolator was tested with following values:
121 For a = 2.05, b = 1.05, c = 3.05, the interpolator returned A = 0.14501036855034055.

```

NR_a1_2.main.txt

3 Exercise 3

For the third exercise we were to work with the data given to us. We also had to write down the log-likelihood corresponding to a set of random realizations of the satellite profile. This is given by:

$$\log L = N \log(A(a, b, c) + (a - 1) \left(\sum_{i=1}^{N-1} \log(x_i) - N \log(b) \right) + \sum_{i=1}^{N-1} -\left(\frac{x_i}{b}\right)^c \quad (1)$$

3.1 Find the a,b,c that maximize this likelihood

In order to accomplish this task I decided to find the minimum of the $-\log$ likelihood. For this I wrote an algorithm that uses the Simplex (also known as the Nelder-Mead method). This function worked well for the test function that I gave it, but I did not manage to keep it within the range given for a,b,c. Given a bit more time, this function would most likely have worked. In order to test it I used the data found in *satgals_m15*.

3.2 Write an interpolator for a,b,c as a function of halo mass, or fit a function to each.

I did not manage to accomplish this task due to time constraints.