NUR Assignment 1

Christiaan van Buchem - s1587064

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:

```
#NR_a1_1_utils.py
import numpy as np
import sys
import math # Only used for math.isnan() in NewRaph_rootfinder()

def poisson_distribution(mean,k):
    fact = np.float(0)
    for i in range(round(k)):
        fact += np.log(k-1)
    fact = np.exp(fact)
    return (mean**k*np.exp(-mean))/fact

def poisson_distribution_new(mean,k):
#Returns probability for given k and mean
    fact = 1
    mag = 0
    prev_n = 0
    for i in range(round(k)):
        temp = str(fact*(k-i))
        n_zeros = len(temp)-1-prev_n
        fact = int(temp[:10])
        mag += n_zeros
        prev_n = len(temp[:10])-1

    a = str(mean**k)
    b = str(np.exp(mean))

    x = (float(a[:10])/1e9 * 1/float(b[:10])/(fact/10**(prev_n-1)))
    mag_tot = -int(np.log10(float(b)))+len(a)-mag
    return x*10**mag_tot
# Only works for massive numbers.......
#end poisson_distribution()

class rng(object):
# rng object that is initiated with a give seed
    def __init__(self, seed):
        self.state = np.int64(seed)

    def LCG_gen(self):
```

```
    #Linear Congruential generator
        x = self.state
        a,c,m = 2**32,1664525,1013904223
        self.state = np.int64((a*x+c)%m)
    #end LCG_gen()

    def XOR_shift(self):
    # XOR-shift generator
        x = self.state
        a1,a2,a3 = 21,35,4
        x = x ^ x >> a1
        x = x ^ x << a2
        x = x ^ x >> a3
        self.state = np.int64(x)
    #end XOR-shift()

    def rand_num(self,l,min=0,max=1):
    # Generates 'l' random numbers between min and max
        output = []
        for i in range(l):
            self.XOR_shift()
            self.XOR_shift()
            self.LCG_gen()
            self.XOR_shift()
            self.XOR_shift()
            output.append(self.state)
        output = np.array(output)/sys.maxsize
        return min+(output*(max-min))
    #end rand_num()
#end rng()

# --- Simple supporting functions ---

def min(l):
    min = 2**64
    for i in l:
        if i < min:
            min = i
    return min
#end min()

def arg_min(l):
    min = 2**64
    arg = None
    for i in range(len(l)):
        if l[i] < min:
            min = l[i]
            arg = i
    return arg
#end arg_min()

def max(l):
    max = -2**64
    for i in l:
        if i > max:
            max = i
    return max
#end max()

def arg_max(l):
    max = -2**64
    arg = None
    for i in range(len(l)):
        if l[i] > max:
            max = l[i]
            arg = i
    return arg
#end arg_max()
```

NR_a1_1_utils.py

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

```python
#NR_a1_1_main.py
import numpy as np
import matplotlib.pyplot as plt
import NR_a1_1_utils as utils

seed = 42
print('Original seed:',seed)

#--- 1.a ---
# Poisson distribution
print('1.a:')
a1 = [[1,0],[5,10],[3,20],[2.6,40]]
print('Simple poisson distribution:')
for i in range(len(a1)):
    print('P({}) with mean {}:'.format(a1[i][1],a1[i][0]),utils.poisson_distribution(a1[
    i][0],a1[i][1]))
print('Large-number poisson distribution (only works for large numbers):')
a1 = [[101,200]]
for i in range(len(a1)):
    print('P({}) with mean {}:'.format(a1[i][1],a1[i][0]),utils.poisson_distribution_new
    (a1[i][0],a1[i][1]))

#--- 1.b ---
print('1.b:')
# RNG
rng = utils.rng(seed)
# Scatter plot
N = 1000
rand = rng.rand_num(N)
plt.scatter(rand[:(len(rand)-1)],rand[1:])
plt.title('Sequential number plot for {} random numbers with seed {}'.format(1000,seed))
plt.savefig('plots/1_b_1.png')

# Histogram
N = 1000000
rand = rng.rand_num(N)
plt.hist(rand,bins=20,range=(0,1))
plt.title('Histogram of 1,000,000 randomly generated numbers'.format(1000,seed))
plt.xlabel('Number of numbers in bin')
plt.ylabel('Number values')
plt.savefig('plots/1_b_2.png')
print('Saving Histogram and scatter plot.')
```

NR_a1_1_main.py

The result of the given script is given by:

```
Original seed: 42
1.a:
Simple poisson distribution:
P(0) with mean 1: 0.36787944117144233
P(10) with mean 5: 1.8871331317208 12e-05
P(20) with mean 3: 4.618166957543755e-18
P(40) with mean 2.6: 6.717133577841103e-49
Large-number poisson distribution (only works for large numbers):
P(200) with mean 101: 1.2695314379023172e-18
1.b:
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:

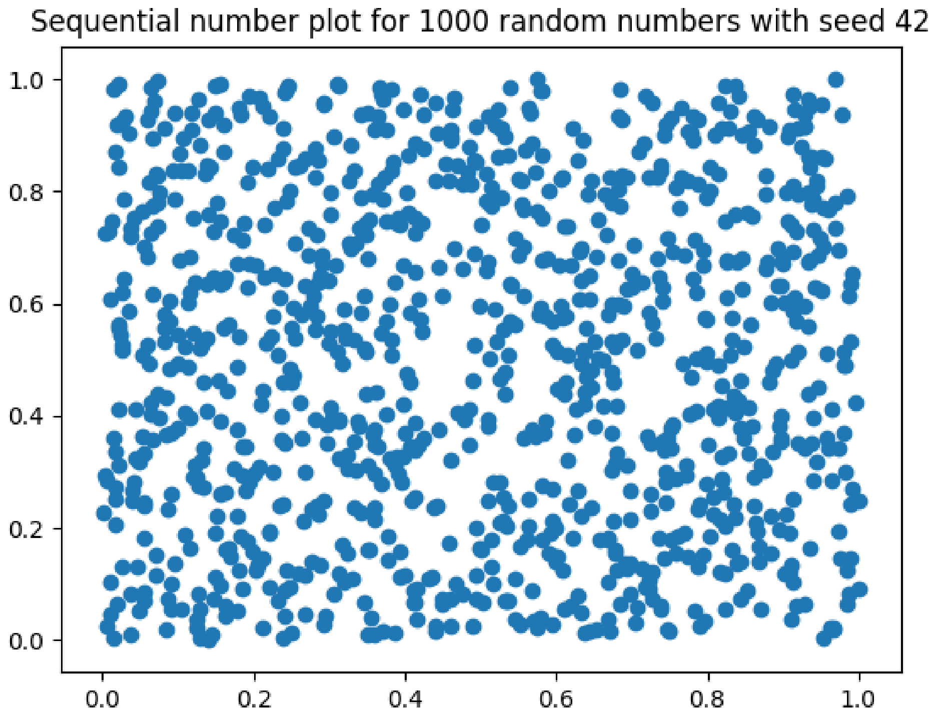

Figure 1: In this figure we can see that it appears that the random number generator is producing numbers without a certain preference.

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 Repeart (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.

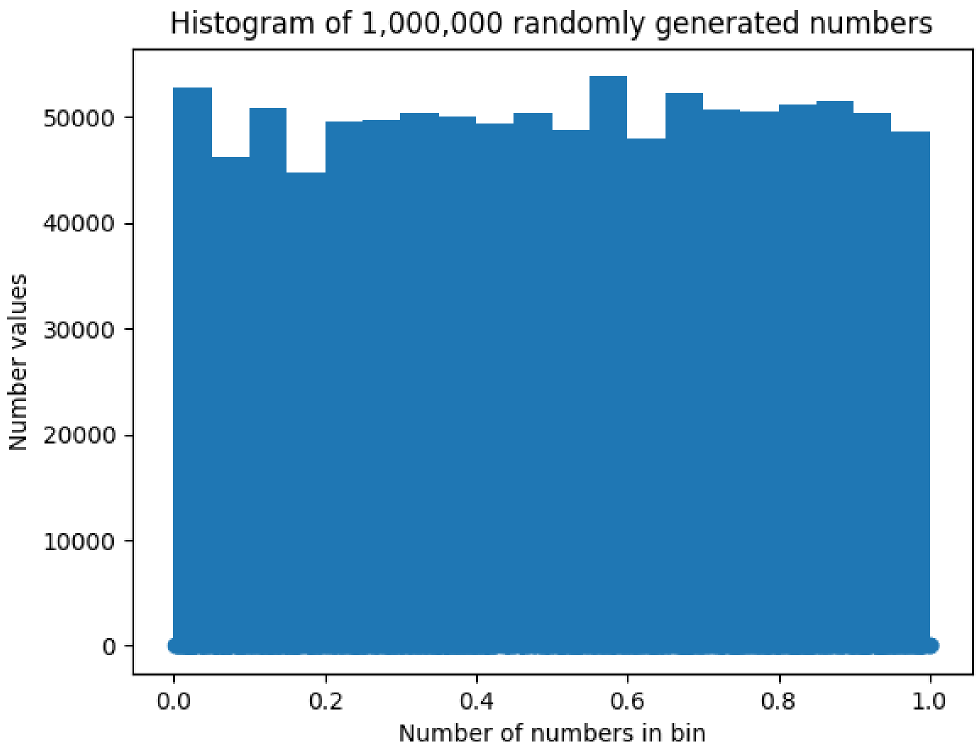

Figure 2: This histogram places the random number generator under a sharper knife, allowing us to see that there are some fluctuations between the bins. Overal it appears to be quite unbiased.

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:

```
#NR_a1_2_utils.py
import numpy as np
import sys
```

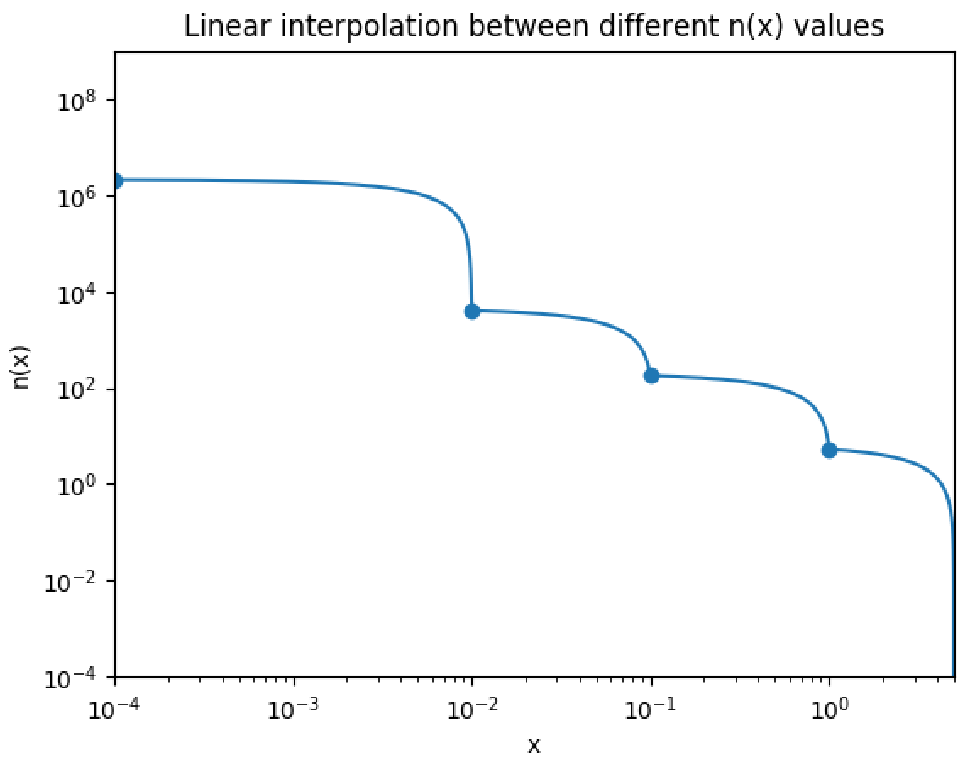

Figure 3: Here we see the way in which a linear interpolator overshoots the expected function when used in log-space.

```
import math # Only used for math.isnan() in NewRaph_rootfinder()
from NR_a1_1_utils import poisson_distribution, rng, min, max, arg_max,arg_min

def central_diff(f,h,x):
# Calculates the central difference\n",
    return (f(x+h)-f(x-h))/(2*h)
#end central_diff()

def ridders_diff(f,x):
# Differentiates using Ridder's method
    m = 10
    D = np.zeros((m,len(x)))
    d = 2
    h = 0.001
    for i in range(m):
        D_new = D
        for j in range(i+1):
            if j == 0:
                D_new[j] = central_diff(f,h,x)
            else:
                D_new[j] = (d**(2*(j+1))*D[j-1]-D_new[j-1])/(d**(2*(j+1))-1)
        D = D_new
        h = h/d
    return D[m-1]
#end ridders_diff()

def comp_trapezoid(f,a,b,n):
# Composite trapezoid rule used in romber_int()
    h = 1/(2**(n-1))*(b-a)
    sum = 0
    for i in range(1,2**(n-1)):
        sum += f(a+i*h)
```

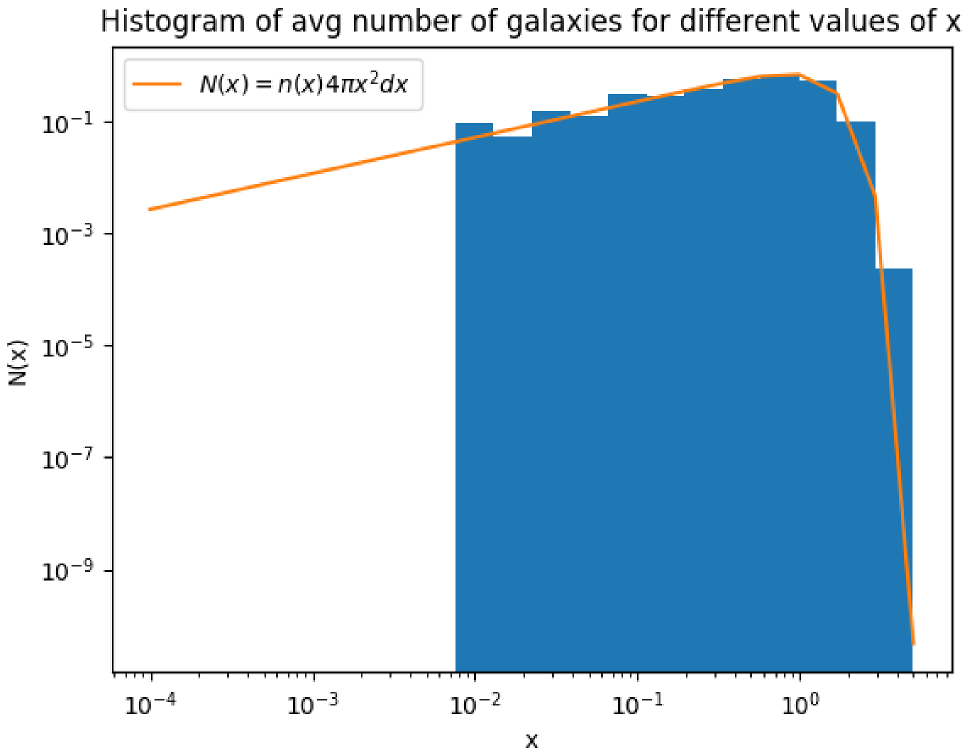

Figure 4: Histogram of number of satellites in each bin with over-plotted probability. Note the discrepancy at the lower end of the distribution.

```
    return (h/2.)*(f(a)+2*sum+f(b))
#end comp_trapezoid()

def romber_int(f,a,b):
# Integrates from a to b up to an accuracy of 6 decimals
    for n in range(1,10):
        S_new = np.zeros((n))
        S_new[0] = comp_trapezoid(f,a,b,n)
        for j in range(2,n+1):
            S_new[j-1] = (4**(j-1)*S_new[j-2]-S[j-2])/(4**(j-1)- 1)
        S = S_new
        if n > 3:
            if abs(S[-2]-S[-1]) < 1e-6:
                return S[-1]
    return S[-1]
#end romber_int()

def interpol_lin_log(xj,yj,x,y):
# Linear interpolator
    j=0
    xj,yj,x = np.log10(xj),np.log10(yj),np.log10(x)
    for i in range(len(x)):
        if x[i]>xj[j]:
            j+=1
        if j>4:
            return 10**y
        y[i]=(((yj[j]-yj[j-1])/(xj[j]-xj[j-1]))*(x[i]-xj[j]))+yj[j]
#end interpol_lin()

def interpol_lin(xj,yj,x,y):
# Linear interpolator
    j=0
```

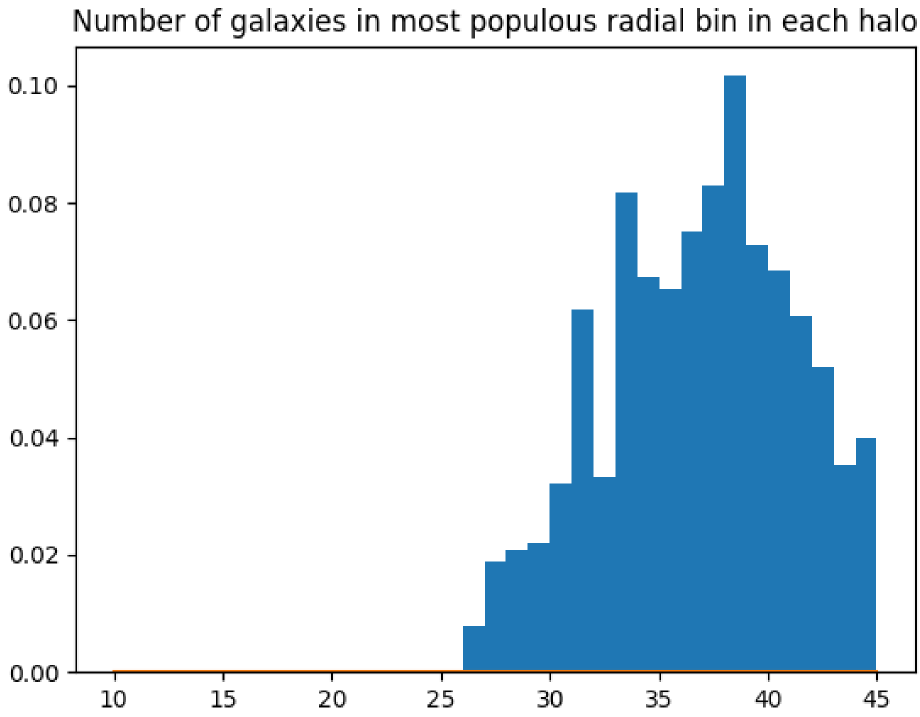

Figure 5: Number of galaxies in most populous radial bin in each halo. Note that the Poisson distribution is most likely calculated wrongly.

```
    for i in range(len(x)):
        if x[i]>xj[j]:
            j+=1
        if j>4:
            return y
        y[i]=(((yj[j]-yj[j-1])/(xj[j]-xj[j-1]))*(x[i]-xj[j]))+yj[j]
#end interpol_lin()

def nevils_alg(x,xj,yj,i,j):
# Neville's Algorithm used in interpol_neville
    if i == j:
        return yj[i]
    else:
        return ((x-xj[j])*nevils_alg(x,xj,yj,i,j-1)-(x-xj[i])*nevils_alg(x,xj,yj,i+1,j))
    /(xj[i]-xj[j])
#end nevils_alg()

def interpol_neville(xj,yj,x,y):
# Interpolator that uses Neville's Algorithm
    for z in range(len(x)):
        y[z] = nevils_alg(x[z],xj,yj,0,len(xj)-1)
    return y
#end interpol_neville()

def rejection_sampler(n,p,max_x,max_y,rng):
# Rejection sampler that uses max_y value as g
    sample_x = []
    sample_y = []
    n_accpt = 0
    while n_accpt < n:
        pot_x = np.float(rng.rand_num(1,max=max_x))
        pot_y = np.float(rng.rand_num(1,max=max_y))
```

```
        if pot_y <= p(pot_x):
            sample_x.append(pot_x)
            sample_y.append(pot_y)
            n_accpt += 1
    return sample_x,sample_y
#end rejection_sampler

def secant_rootfinder(f,a,b):
# Secant method root-finder
    it = 0
    while np.abs(b-a) > 0.00001:
        it += 1
        c = -(b-a)/float((f(b)-f(a)))*f(a)+a
        a = b
        b = c
    return a
#end secant

def falspos_rootfinder(f,a,b):
# False position method root-finder
    it = 0
    c = 0
    while np.abs(b-a) > 0.001:
        it += 1
        c = -(b-a)/float((f(b)-f(a)))*f(a)+a
        print('a',a,', b',b)
        print('c',c)
        if f(a)*f(c) > 0:
            a = c
        else:
            b = c
    return c,it
#end falspos_rootfiner()

def NewRaph_rootfinder(f,a,b,rng):
# Finds root in given range (a,b) for f
    x = rng.rand_num(1,min=a,max=b)
    x_new = sys.maxsize
    for i in range(1000):
        f_deriv = ridders_diff(f,np.array([x]))
        if f_deriv == 0 or math.isnan(f_deriv):
            x = rng.rand_num(1,min=a,max=b)
            i = 0
            continue
        x_new = x - f(x)/f_deriv
        if abs(x_new-x) < 1e-12:
            return x_new
        else:
            x = x_new
    return x
#end NewRaph_rootfinder()

def selection_sort(l):
# Ascending sorting of l using selection sort
    sl = []
    for i in range(len(l)):
        min = arg_min(l)
        sl.append(l[min])
        l = l[:min]+l[(min+1):]
    return sl
#end selection_sort()

def A_calc(a,b,c):
# Calculates A with given values of a,b,c
    f = lambda x: 4*np.pi* (x**(a-1))/(b**(a-3)) *np.exp(-(x/b)**c)
    f_int = romber_int(f,0,5)
    return 1/f_int
#end A_calc()

def trilinear_interpolator(al,bl,cl,Al,x,y,z):
```

```python
# Returns an interpolated value for Al based on a,b,c values
    def bracket_finder(xl,x):
        for i in range(len(xl)):
            if xl[i] > x:
                return xl[i-1],xl[i],i
        print('Could not find a backet, returning nan.')
        return float('nan'),float('nan')

    a0,a1,ai = bracket_finder(al,x)
    b0,b1,bi = bracket_finder(bl,y)
    c0,c1,ci = bracket_finder(cl,z)

    xd = (x-a0)/(a1-a0)
    yd = (y-b0)/(b1-b0)
    zd = (y-c0)/(c1-c0)

    c00 = Al[ai-1][bi-1][ci-1]*(1-xd)+Al[ai][bi-1][ci-1]*xd
    c01 = Al[ai-1][bi-1][ci]*(1-xd)+Al[ai][bi-1][ci]*xd
    c10 = Al[ai-1][bi][ci-1]*(1-xd)+Al[ai][bi][ci-1]*xd
    c11 = Al[ai-1][bi][ci]*(1-xd)+Al[ai][bi][ci]*xd

    c0 = c00*(1-yd)+c10*yd
    c1 = c01*(1-yd)+c11*yd

    return c0*(1-zd)+c1*zd
#end trilinear_interpolator()
```

NR_a1_2_utils.py

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

```python
#NR_a1_2_main.py
import numpy as np
import matplotlib.pyplot as plt
import NR_a1_2_utils as utils

seed = 42
print('Original seed:',seed)
rng = utils.rng(seed)

#--- 2.a ---
print('2.a:')
a = rng.rand_num(1,min=1.1,max=2.5)
b = rng.rand_num(1,min=0.5,max=2)
c = rng.rand_num(1,min=1.5,max=4)
f = lambda x: 4*np.pi* (x**(a-1))/(b**(a-3)) *np.exp(-(x/b)**c)
f_int = utils.romber_int(f,0,5)
A = 1/f_int
print('A = {}; a,b,c = {},{},{}'.format(A,float(a),float(b),float(c)))

#--- 2.b ---
print('2.b:')
xj = [10**-4,10**-2,10**-1,1,5]
n_x = lambda x: A*100*(x/b)**(a-3)*np.exp(-(x/b)**c)
n = n_x(xj)
x = np.logspace(np.log10(1e-4),np.log10(5),10000)
y = np.zeros(10000)
y_lin = utils.interpol_lin(xj,n,x,y)
plt.scatter(xj,n)
plt.plot(x,y_lin)
plt.xlim(left=10**-4,right=5)
plt.ylim(bottom=1e-4,top = 1e9)
plt.xscale('log')
plt.yscale('log')
plt.xlabel('x')
plt.ylabel('n(x)')
plt.title('Linear interpolation between different n(x) values')
plt.savefig('plots/2_b.png')
plt.close()
print('Saved interpolation plot to \'plots/2_b.png\'.')

```

```
#--- 2.c ---
print('2.c:')
n = lambda x: A*100*(x/b)**(a-3)*np.exp(-(x/b)**c)
x = b
dndx = utils.ridders_diff(n,np.array([b]))
dndx_analitic = lambda x: (A*100) * (((a-3)*(x/b)**(a-4)*np.exp(-(x/b)**c))/b - ((c*np.exp(-(x/b)**c)*(x/b)**(a+c-4))/b))
dndx_an = dndx_analitic(x)
print('dn/dx at x = b: analytic = {0:.12f}; numerical = {1:.12f}'.format(float(dndx_an),float(dndx)))

#--- 2.d ---
print('2.d:')
N = 100
xmax = 5
# Drawing random samples from n(x)
pn = lambda x: (n(x)*4*np.pi*x**2)/100
x_p = np.linspace(0,xmax,200)
g = np.max(pn(x_p)[1:])+0.01
samples = utils.rejection_sampler(N,pn,5,g,rng)
r = samples[0]
# Generating random angles:
phi = rng.rand_num(N,min=0,max=2*np.pi)
theta = np.arccos(2*rng.rand_num(N)-1)
x,y,z = r*np.sin(theta)*np.cos(phi),r*np.sin(theta)*np.sin(phi),r*np.cos(theta)
# Plotting positions for N galaxies
#ax = plt.figure().add_subplot(111,projection='3d')
#ax.scatter(x,y,z)
#plt.show()
print()
print('r,phi,theta:')
for i in range(len(r)):
    print(r[i],phi[i],theta[i])


#--- 2.e ---
print('2.e:')
N = 100000
samples = utils.rejection_sampler(N,pn,5,g,rng)
r = samples[0]
bins = np.logspace(np.log10(1e-4),np.log10(xmax),num=21)
plt.hist(r,bins=bins,density=True)
plt.plot(bins,pn(bins),label = '$N(x) = n(x)4\pi x^2 dx$')
plt.yscale('log')
plt.xscale('log')
plt.ylabel('N(x)')
plt.xlabel('x')
plt.legend()
plt.title('Histogram of avg number of galaxies for different values of x')
plt.savefig('plots/2_e.png')
plt.close()
print('Saved histogram to \'plots/2_e.jpg\'.')

#--- 2.f ---
print('2.f:')
dpndx = utils.ridders_diff(pn,x)
dpndx_analytic = lambda x: A*4*np.pi*(np.exp(-(x/b)**c)*(((a-1)*b**(3-a)*x**(a-2))-(c*b**(2-a)*x**(a-1)*(x/b)**(c-1))))

dpndx_0 = float(utils.NewRaph_rootfinder(dpndx_analytic,1e-4,1,rng))
new_floor = float(pn(dpndx_0)/2)
pn_new_floor = lambda x: pn(x) - new_floor
root1 = float(utils.NewRaph_rootfinder(pn_new_floor,1e-4,dpndx_0,rng))
root2 = float(utils.NewRaph_rootfinder(pn_new_floor,dpndx_0,5,rng))
print('Roots:', root1,root2)

#--- 2.g ---
print('2.g:')
counts = np.zeros((len(bins)-1))
for i in r:
```

```
    for j in range(len(bins)-1):
        if i < bins[j+1] and i > bins[j]:
            counts[j] += 1
r_list = []
r_halo_distrib = np.zeros((1000))

for i in range(len(r)):
    if r[i] < bins[utils.arg_max(counts)+1] and r[i] > bins[utils.arg_max(counts)]:
        r_list.append(r[i])
        r_halo_distrib[i//100] += 1
mean = sum(r_halo_distrib)/len(r_halo_distrib)
halo_bins = np.linspace(10,45,36)
poissd = []

for i in range(len(halo_bins)):
    poissd.append(utils.poisson_distribution(round(mean),int(halo_bins[i])))

plt.hist(r_halo_distrib,halo_bins,density=True)
plt.plot(halo_bins,poissd)
plt.title('Number of galaxies in most populous radial bin in each halo')
plt.savefig('plots/2_g.png')
print('Saved histogram to \'plots/2_g.jpg\'.')
print('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.')
plt.close()

sr = utils.selection_sort(r_list)

median = sr[int(len(sr)/2-0.5)]
p16th = sr[round(len(sr)*0.16)-1]
p84th = sr[round(len(sr)*0.84)-1]
print('Length: {}, median: {}, 16th: {}, 84th: {}'.format(len(sr),median,p16th,p84th))

#--- 2.h ---
print('2.h:')
al = np.linspace(1.1,2.5,15)
bl = np.linspace(0.5,2,16)
cl = np.linspace(1.5,4,26)
param = np.array((al,bl,cl))
Al = np.zeros([len(al),len(bl),len(cl)])
for i in range(len(al)):
    for j in range(len(bl)):
        for k in range(len(cl)):
            Al[i][j][k] = utils.A_calc(al[i],bl[j],cl[k])

interpol = utils.trilinear_interpolator(al,bl,cl,Al,2.05,1.05,3.05)
print('Interpolator was tested with following values:')
print('For a = {}, b = {}, c = {}, the interpolator returned A = {}.'.format
    (2.05,1.05,3.05,interpol))
```

NR_a1_2_main.py

The result of the given script is given by:

```
Original seed: 42
2.a:
A = 0.04701878985790245; a,b,c =
    1.6419613428029698,1.4577252811360346,2.6047059307084877
2.b:
Saved interpolation plot to 'plots/2_b.png'.
2.c:
dn/dx at x = b: analytic = -4.702159549813; numerical = -4.702159549822
2.d:

r,phi,theta:
0.7311141707122876 1.6028076635948272 0.2270265285598231
2.197278770187982 5.29883706734633 1.3425005017754366
1.261066210848445 0.4705410840621936 1.652797361141153
0.6558716361204144 5.012800547859235 2.280159240554945
0.7824897836202369 3.2051614706482767 1.109519113847493
```

```
0.5641889930754497 4.239148403641524 2.5313275914354216
0.7455003237080227 1.3378904226799089 2.07121896558909
0.9233498635882491 0.13554910534546938 0.8080191138718723
0.6646036870484072 2.593921050894136 0.6430905972730414
1.422587656024743 1.5936802052506487 1.1961750855480042
1.2151205803427914 3.3866416280089577 1.371971552370749
0.2683698883997805 3.983288228149664 2.34892948705019
0.5195498142643846 0.3023185592993231 0.20783067746454917
0.4319417574879965 7 2.0002705470795115 2.114834963001796
0.45944451540419623 3.701972044276426 2.720290327493936
0.8006572399002244 4.274590237546436 1.7464117024831116
0.2804255826180151 2.086678285518562 1.2565200536872243
1.7281651532360687 1.493229941026918 1.5682490721280593
0.4403162291018148 5.51406175002302 2.3134558412397968
0.28005365086933415 4.1667562336864306 1.2754577518288508
0.629885215859386 3.126371193501809 0.8823133533723011
1.3341331154888179 3.744806626025325 2.431634242424714
1.8027138385254526 3.577180243900044 1.47799705232885
0.2378761548099098 0.21519768812236728 2.7532126716994534
0.7747006558042651 3.686789510644586 1.1150685587388856
0.6869077696133774 3.636542606711427 1.8470806713450534
1.4117884436667754 3.7242596526259963 0.891581350772273
0.6294095410752514 1.2088053560981842 0.8448593580067348
1.2764752027254045 1.098289778440001 1.4188595428183999
1.6150927683803165 1.9199926678160373 2.8470316361511516
0.4186689992245968 5.637045372231429 1.9593635789694905
0.5461799853935555 2.2252698686054537 0.17807153021610092
0.6829810271722446 0.24992055211606554 1.2590866128292584
0.6234502914389102 1.5575463076869536 1.195224966142777
1.3366949341522365 3.0485171580618378 0.5212172093727148
0.6499529248316984 2.5940784042398093 1.453743194278108
0.9067285203917376 2.132532890780963 0.7257116272553009
1.5146136209601937 2.404549315401679 1.978017257685901
0.2770245026144908 6.164596891659123 1.4694682169604514
0.5159651931040894 2.3253515653634365 1.1861233069219161
0.4759216002517306 6.047620429111068 0.5387480016708217
2.247016406983809 0.13409749171900837 1.3807219348619766
1.5602946396462583 6.218389640083777 2.186858237035807
2.3029029642531844 3.3462652294283277 0.4412630335676338
0.6965374980804753 0.4498323703870266 0.43329541725053206
1.1409926578391123 0.7249796633728953 0.9982803668408025
0.6653952740755197 1.0253133385140492 1.9986788291368351
1.6205144185041889 1.005557374832275 0.4313321487854299
1.0420429483652858 3.4936712489898514 1.1563665983921778
1.8162715311433768 4.398841225589391 1.257522673876321
1.753618683902485 3.9458599495477777 2.8651583133590033
0.9075665701160043 5.367493156386626 2.631654291756891
2.196964042542822 2.597252707137583 1.724099860613247
1.335804466710699 0.4204979764301613 0.7175964769213862
0.1827132537627097 2.241282270808069 0.853186798298481
0.7038700495210275 2.3096897199004744 1.9112929888501538
1.7123198598059761 6.120340636505855 1.8682558561336253
2.079763440168249 2.150644043039724 2.314405830493339
1.8501341524498836 4.948950050730439 2.7251108593102016
0.8911001722393672 0.8143685684732586 2.022113824428344
1.091892711619008 5.552002315953939 1.7871299593017478
2.0542049463220273 3.7792343179439785 2.455513735502
1.3211167003377824 2.8219224435250907 0.8302843336903301
1.5889036994914583 3.862410859025583 0.20885659658253541
0.9993672674922676 4.641443887553411 1.2991381763462724
0.21591307901877496 5.32616256645649 2.161269458308147
1.6492796030749581 3.2963015420602892 2.239559557821826
1.4581203252216812 2.927113955421931 1.8842855313416325
2.4482548492097473 4.3877153728887714 1.532180548712544
1.9948482716434734 0.8685057783736273 0.5763854455089538
0.9218681038047252 3.987296975053835 0.5972229423329523
0.44154526532833743 3.2402812512358494 1.1749082354327303
0.18173811352986738 1.7758883998026385 1.3174654425683672
0.5187689966449206 4.543296053760293 1.7492149796888092
1.3650977926903596 0.6726655795284455 2.6194128692669856
```

```
1.4861118717435184 5.627875870996373 0.6574243816564097
1.5804517116615513 1.2429693794730885 1.8099719124226885
0.7420420489988011 5.30214916092477 0.8281478177090664
1.3314795740982264 2.3793285612306017 2.3801303781445413
0.6407129680532632 0.8940786201724139 1.7575270804295209
0.7322550000028955 5.290332743081566 1.6484916266608831
0.3504764783590363 6.087384004395324 2.8246659302385067
1.4659929210709874 6.282353866008897 2.402429041978465
1.9332396764300799 0.5667864302431423 1.5154492416588403
1.8020963918757016 0.6313214477879985 0.9514025960075138
1.1281752377936851 3.4268531372170266 2.1126890895625485
0.8940159908580776 4.507780842667583 1.2803535540003392
0.5783915184826695 6.095133351100733 1.2642369297377836
0.8172476257165974 4.896468153276877 1.5065904525286185
1.5578782882469553 2.94710820289594 0.653235189348019
0.7787656442861324 5.320419139190734 1.771281526346205
0.4080652971728931 4.270079657246644 1.8380359213790045
0.07337931053185985 2.6303052090296744 2.031693926313672
0.9549057503078782 2.189789006551835 2.1302858225822474
1.0740411498593938 4.667855605228215 2.1200254410634685
1.181490451901903 0.912168796509376 3.002947455960485
1.970841873281359 0.05396950164989878 2.0157468651014
1.415274167720679 4.574893947574355 1.2887032759180237
0.8888173153903209 0.7663566943066645 1.6355426458896933
1.4996123211828571 1.9576845334443223 2.49707290293986
2.e:
Saved histogram to 'plots/2_e.jpg'.
2.f:
Roots: 0.1987318674472368 1.6407030708001724
2.g:
Saved histogram to 'plots/2_g.jpg'.
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.
Length: 37615, median: 1.2773704178216243, 16th: 1.0771942122640141, 84th: 1.5217983937551454
2.h:
Interpolator was tested with following values:
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 give by:

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

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 $satgals_m 15$.

3.2 Write an interpolator fora,bandcas a function of halo mass,orfit a function to each.

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

3.3 Scripts

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

```python
#NR_a1_3_utils.py
import numpy as np
import sys
from NR_a1_1_utils import poisson_distribution, rng, min, max, arg_max,arg_min
from NR_a1_2_utils import romber_int

def selection_sort4simplex(xs,fxs):
# Ascending sorting of xs and fxs based on fxs values using selection sort
    sfxs = []
    sxs = []
    index = []
    #print(fxs)
    for i in range(len(fxs)):
        min = arg_min(fxs)
        sfxs.append(fxs[min])
        sxs.append(xs[min])
        #print(sxs,sfxs)
        if sfxs[i] == fxs[0]:
            fxs = fxs[(min+1):]
            xs = xs[(min+1):]
        elif sfxs[i] == fxs[-1]:
            fxs = fxs[:min]
            xs = xs[:min]
        else:
          #print('min',min)
           #print('xs',xs[:min],xs[(min+1):])
            fxs = np.append(fxs[:min],fxs[(min+1):])
            xs = np.concatenate((xs[:min],xs[(min+1):]))
        #print(xs,fxs)
    return np.array(sxs),np.array(sfxs)
#end selection_sort4simplex()

def downhill_simplex(f,a,b,c,la = [-2**32,2**32],lb = [-2**32,2**32],lc = [-2**32,2**32]):
#Optimizing function
    alpha,beta,gamma = 1,1,0.3
    rngen = rng(14)

    # Generate starting points
    xs = [[a,b,c]]
    for i in range(3):
        xs.append([np.round(float(a+rngen.rand_num(1)),2),np.round(float(b+rngen.rand_num(1)),2),np.round(float(c+rngen.rand_num(1)),2)])
        xs.append([float(a+rngen.rand_num(1)),float(b+rngen.rand_num(1)),float(c+rngen.rand_num(1))])
    xs = np.array(xs)

    #Centroid function
    centroid = lambda xs: np.array([sum(xs[:,0]),sum(xs[:,1]),sum(xs[:,2])])/len(xs)
    it = 0
    N = 1000
    while it < N:
        it += 1
        # Reorder the points from best to worst
        fxs = f(xs)
        xs,fxs = selection_sort4simplex(xs,fxs)
        #print('-------------------')
        #print('xs:',xs)
        #print('fxs:',fxs)
        #Generate a trial point by reflection
        c = centroid(xs)
        #print('Centroid:',c)
        x_new = np.array([c + alpha*(c-xs[-1])])
        #print('x_new:',x_new)
        if x_new[0][0] < la[0] or x_new[0][0] > la[1] or x_new[0][1] < lb[0] or x_new[0][1] > lb[1] or x_new[0][2] < lc[0] or x_new[0][2] > lc[1]:
```

```
            fx_new = 2**32
        else:
            fx_new = f(x_new)
        #print('fx_new',fx_new)

        if fx_new < fxs[0]: #Expansion
            #print('Expanding')
            x_exp = x_new + beta*(x_new-c)
            if f(x_exp) < f(x_new):
                xs[-1] = x_exp
            else:
                xs[-1] = x_new

        elif fx_new > fxs[-1]: #Contraction
            fx_cont = sys.maxsize
            wit = 0
            while fx_cont > fxs[-1] and wit < 10:
                wit += 1
                #print('Contracting')
                x_cont = np.array([c +gamma*(xs[-1]-c)])
                if x_cont[0][0] < la[0] or x_cont[0][0] > la[1] or x_cont[0][1] < lb[0]
    or x_cont[0][1] > lb[1] or x_cont[0][2] < lc[0] or x_cont[0][2] > lc[1]:
                    fx_cont = 2**32
                    gamma = gamma*0.9
                else:
                    fx_cont = f(x_cont)
            xs[-1] = x_cont

        else: #Back to reflection
            #print('Back to top')
            xs[-1] = x_new

        if np.abs(fxs[0]-fxs[1]) < 1e-15:
            print('Found minimum after {} iterations'.format(it))
            print(xs[0],fxs[0])
            return xs[0],fxs[0]
    print('Reached maximum number of iterations, returning best coordinates...')
    return xs[0],fxs[0]
#end downhill_simplex()

def minlog_likelyhood(abc):
# Returns the log_likelyhood of n(x) for given parameters
    a,b,c = abc[:,0],abc[:,1],abc[:,2]

    #Calculating A
    A = []
    for i in range(len(abc)):
        f = lambda x: 4*np.pi* (x**(a[i]-1))/(b[i]**(a[i]-3)) *np.exp(-(x/b[i])**c[i])
        A.append(1/romber_int(f,0,5))
    print(A)

    #Calculating rest of the formula
    N = len(data)
    logSum,expSum = 0,0
    for i in data:
        logSum += np.log(i)
        expSum += -1*(i/b)**c
    return -(N*np.log(A) + (a-1)*(logSum-N*np.log(b)) + expSum)
#end log_likelyhood()
```

NR_a1_3_utils.py

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

```
#NR_a1_3_main.py
import numpy as np
import matplotlib.pyplot as plt
import NR_a1_3_utils as utils

#--- 3.a ---
print('3.a:')
```

```
a,b,c = 10.,3.,3.
xs = [[a,b,c]]
f = lambda xs : (xs[:,0]-1/3)**2+(xs[:,1]-2/3)**2+(xs[:,2]-1/3)**2
min = utils.downhill_simplex(f,a,b,c)
#print(min)
#print('Found the minimum value {4} at a,b,c = {0},{1},{2}')(min[0][0],min[0][1],min
    [0][2],min[1])

x = utils.data_unpacker('satgals_m15.txt')
la,lb,lc = [1.1,2.5],[0.5,2.],[1.5,4.] #Giving a,b,c value range
min = utils.downhill_simplex(utils.minlog_likelyhood,1.8,1.25,2.75,la,lb,lc)
#print('Found the minimum value {4} at a,b,c = {0},{1},{2}')(min[0][0],min[0][1],min
    [0][2],min[1])
print('This almost works correctly, only thing that needs to be done is to properly
    enforce range of abc, values.')
```

NR_a1_3_main.py

The result of the given script is given by:

```
3.a:
Found minimum after 614 iterations
[0.33333338 0.6666667  0.33333334] 4.039755480067767e-15
```

NR_a1_3_main.txt