NUR Hand-in excercise 1

Meng Yao (s2308266)

April 9, 2019

Abstract

The source code and the outputs of Hand-in exercise 1 are shown in this report.

1 Source code

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
import sys
sys.stdout = open('outputs.txt', 'w')

'''
Question 1
(a)
Poisson probability distribution function
P_lambmda(k)=(lambda^k)*(e^-lambda)/(k!)
'''
print('Question 1')
print('(a) Poisson function')
def Poisson(p_lam,p_k):
    '''
    Possion Function (lambda^k)*(e^-lambda)/(k!)
    p_lam ------parameter lambda
    p_k   ------parameter k
    '''
    factorial = np.float64(1)
    for i in range(1,p_k+1):       #calculate the factorial of k
        factorial = factorial * np.float(i)
    num1=np.float(p_lam**p_k)      #the 1st item of numerator
    num2=np.float(np.exp(-p_lam))  #the 2nd item of numerator
    den=factorial                  #denominator
    P=np.float64(num1*num2/den)    #calcultae P
    return P
for i in ((1,0),(5,10),(3,21),(2.6,40)):
    print('Poisson{} = '.format(i),Poisson(i[0],i[1]))

'''
(b)
random number generator
'''
```

```
print('\n(b) Random number generator')
def generator(n,seed):
    '''
    combined number generator:
    LCG(XOR-shift)^MWC
    period = period of xorshift * period of MWC
    the parameters will be given later
    n = the amount of numbers
    seed = initial seed
    '''
    #parameters of each generator
    ##XOR-shift 64-bit
    XOR_a1=21
    XOR_a2=35
    XOR_a3=4
    bit64=2**64-1
    ##LCG
    LCG_a=3935559000370003845
    LCG_c=2691343689449507681
    mod = 2**64
    ##MWC
    MWC_a=4294957665
    #seed
    x=seed
    l=seed
    m=seed
    number = np.zeros(n)
    for i in range(n):
        #XORshift
        '''
  XOR-shift : keep the number in 64-bit and do logical left or right bit shift, which means some the bits that are moved out of memory.
  the 'new bits' will be filled by 0
  '''
        x = x ^ (x >> XOR_a1)
        x = x ^ (x << XOR_a2) & bit64 #  do a logical 'and' to cut the number to 64 bits.
        x = x ^ (x >> XOR_a3)
        # LCG part
        '''
        use the output of XOR-shift to get a new number, the new period = the period of XOR which is 2^64-1
        l will not fed back into the XOR-shift, which works unperturbed
  LCG : calculate the remainder and put it back into iteration. Hence, generally the period is a factor of the divisor.
        '''
        l = (LCG_a * x + LCG_c) % mod
        #MWC
        '''
  only the 32 bits of the old number will be manipulated and propagated to the next one
  '''
        m = (MWC_a*(m & (2**32-1))+(m >>32)) & bit64 # constrain the bits of MWC
        #combine them
        number[i] = (l ^ m)
    #normalise in (0,1) /maxnumber of 64.
    #Note that 'period' shows the repeating information (how long the sequence is), not the range of radom number
    number=np.array(number)/(2**64-1)
```

```
    return number
#set the seed
seed = 777
print('seed = ', seed)
#first 1000 numbers
n1000 = generator(1000,seed)
fig1 = plt.figure(1)
ax1_1 = fig1.add_subplot(1,2,1)
ax1_1.scatter(n1000[0:-2], n1000[1:-1])
ax1_1.set_xlabel("$X_i$")
ax1_1.set_ylabel("$X_{i+1}$")
ax1_1.set_title('Sequential 1000 numbers')
# 1 million numbers
n1m=generator(10**6,seed)
ax1_2=fig1.add_subplot(1,2,2)
ax1_2.hist(n1m, bins=np.linspace(0.0, 1.0, 21))
ax1_2.set_title('Histogram of 1 million numbers')
ax1_2.set_xlabel('bins')
ax1_2.set_ylabel('quantity of numbers')
fig1.tight_layout()
fig1.savefig("1.png")
fig1.show()
print('figure of random number generator please see fig.1')
'''
Question 2
(a)
Statellite galaxies
number density profile
solve normalisation coefficient A
'''
print('\n\nQuestion 2')
print('(a) solve A ')
#pick up generated numbers and set them to the required range.
a = (2.5-1.1)*n1000[7]+1.1
b = (2-0.5)*n1000[77]+0.5
c = (4-1.5)*n1000[777]+1.5
print(' a = ', a)
print(' b = ', b)
print(' c = ', c)
'''
BTW, I don't think 2(a) is slovable without viral radius, so I will set it to 1. Then, x = r.
n(x) = A*100*(x/b)^(a-3)*exp(-(x/b)^c)
3rd_integral n(x) d(4*pi*x^3)/3 = 100 ==> 3rd_integral n(x)*4*pi*x^2 dx = 100 ==> 3rd_integral A*(x/b)^(a-3)*exp(-(x/b)^c) * 4*pi*x^2 dx= 1
Hence, A = 1 / 3rd_integral (x/b)^(a-3)*exp(-(x/b)^c) * 4*pi*x^2 dx
'''
N_sat=100.
# Now do the integral, using trapeziodal rules
def integrator(function, lower ,upper, intervals):
    '''
    trapeziodal rule:
    s=h/2(f(x_0)+2sum(f(x_1_to_n-1))+f(x_n))
    function =
    lower = lower limit of the integral
    upper = upper limit
    interbvals = n of intervals
    '''
    h = (upper - lower) / intervals
    S = 0.5*(function(lower) + function(upper))
    for i in range(1,intervals):
```

```
        S += function(lower + i*h)
    integral = h * S
    return integral
# def the function
function_x = lambda x: (x/b)**(a-3.)*np.exp(-(x/b)**c)*4.*np.pi*x**2.
# calculate A
'''
since when x = 0, n(x) is imporper
I set the lower limit to a small value
'''
A = 1. / integrator(function_x, 10**(-32), 5., 10000)
print(' A = ',A)



'''
(b)
make a log-log plot , plot 4 points and do interpolations.
I tried cubic, quadratic adn linear spline interpolation.
None of the results is satisfied and linear spline is the 'best' among them.
'''
print('\n(b) log-log plot. See fig.2.')
#def n(x)
n_x = lambda x : A*N_sat*(x/b)**(a-3)*np.exp(-(x/b)**c)
#ture data x = 10^-4, 10^-2, 10^-1, 1, 5
data_ture_x=np.array([10**-4,10**-2,10**-1,1,5])
data_ture_n=n_x(data_ture_x)
fig2 = plt.figure(2)
fig2.suptitle('log-log plot of n(x)')
ax2_1 = fig2.add_subplot(1,1,1)
ax2_1.set_xscale('log')
ax2_1.set_yscale('log')
ax2_1.set_xlabel('$x = r/r_{vir}$ (log)')
ax2_1.set_ylabel('number density (log)')
ax2_1.plot(data_ture_x, data_ture_n,'o',label='true data')
#Neville's method
def neville(x_ture, y_ture, x_new):
    '''
    Neville's Algorithm:
    p_i,i(x) = y_i
    p_i,j(x) = [(x_j-x)*p_i,j-1(x)+(x-x_i)*p_i+1,j(x)]/x_j-x_i
    '''
    n = len(x_ture)
    y = y_ture.copy()
    for m in range(1,n):
        for i in range(n-m):
            y[i] = ((x_new - x_ture[i+m])*y[i]+(x_ture[i]-x_new)*y[i+1])/(x_ture[i]-x_ture[i+m])

    return y[0]
inter_x=np.linspace(10**-4,5,1000)
inter_n_Neville=np.zeros(1000)
for i in range(len(inter_n_Neville)):
    inter_n_Neville[i] = neville(data_ture_x,data_ture_n,inter_x[i])
ax2_1.plot(inter_x,inter_n_Neville,'b',label='Neville')
#linear interpolation
def linear(inter_x):
    '''
    linear interpolation is quite easy. Take the ture points and
```

```
    calculate the slope and constant for every two adjacent points.
    '''
    #calculate k & b in each range
    #[10^-4,10^-2)
    k0 = (data_ture_n[1]-data_ture_n[0])/(data_ture_x[1]-data_ture_x[0])
    b0 = data_ture_n[1]-k0*data_ture_x[1]
    #[10^-2,10^-1)
    k1 = (data_ture_n[2]-data_ture_n[1])/(data_ture_x[2]-data_ture_x[1])
    b1 = data_ture_n[2]-k1*data_ture_x[2]
    #[10^-1,1)
    k2 = (data_ture_n[3]-data_ture_n[2])/(data_ture_x[3]-data_ture_x[2])
    b2 = data_ture_n[3]-k2*data_ture_x[3]
    #[1,5]
    k3 = (data_ture_n[4]-data_ture_n[3])/(data_ture_x[4]-data_ture_x[3])
    b3 = data_ture_n[4]-k3*data_ture_x[4]
    # do interpolation
    inter_n = np.zeros(len(inter_x))
    for i in range(len(inter_x)):
        if  inter_x[i] < data_ture_x[1]:
            inter_n[i] = k0*inter_x[i] + b0
        if  inter_x[i] >= data_ture_x[1] and inter_x[i] < data_ture_x[2]:
            inter_n[i] = k1*inter_x[i] + b1
        if  inter_x[i] >= data_ture_x[2] and inter_x[i] < data_ture_x[3]:
            inter_n[i] = k2*inter_x[i] + b2
        if  inter_x[i] >= data_ture_x[3]:
            inter_n[i] = k3*inter_x[i] + b3
    return inter_n
inter_n_linear = linear(inter_x)
ax2_1.plot(inter_x,inter_n_linear,'r',label='linear spline')
ax2_1.legend(loc='lower left')
fig2.savefig('2b.png')
fig2.show()

'''
(c) Derivative
Analytical Derivative is (b^3*e^(-(x/b)^c)*(x/b)^a * (-3 + a - c*(x/b)^c))/x^4
'''
print('\n(c) Derivative')
dn_x = lambda x: A*N_sat*(b**3*np.exp(-(x/b)**c)*(x/b)**a*(-3+a-c*(x/b)**c))/x**4
print('Analytical dn(x)/dx (x=5) = ', dn_x(b))
# Use central difference to calculate the derivative.
def central_difference(function,x,h):
    '''
    derivative = lim_(h-->0) [f(x+h)-f(x-h)]/2*h
    function =
    x =
    h = step size
    '''
    derivative = (function(x + h) - function(x - h))/(2*h)
    return derivative
print('Numerical  dn(x)/dx (x=5) = ', central_difference(n_x,b,10**-8))

'''
```

```
(d)
probability distrubution sampling
'''
print('\n(d) probability distribution and the positions of satellites')
# Choose sampling method to be rejection.
n2m=generator(2*10**6,seed)
def rejection_sample(function, x_min, x_max, y_min, y_max, n):
    '''
    I use rejection sampling here is because it is easy to apply with my random number generator
    according to the slides. If the yi < p(x) then the combination of x_i&y_i will be accepted.

    function =
    x_min, x_max =
    y_min, y_max =
    n = amount of points wanted to be tested < 2 million

    '''
    accepted_x=[]
    accepted_y=[]
    for i in range(n):
        x = n2m[i] * (x_max - x_min) + x_min
        y = n2m[-i] * (y_max - y_min) + y_min   # I don't want the orginal random number are the same.
        if y < function(x):
            accepted_x.append(x)
            accepted_y.append(y)

    return accepted_x, accepted_y

# def probability function
p_x = lambda x: A*(x/b)**(a-3)*np.exp(-(x/b)**c)*4*np.pi*x**2

sample_x, sample_y = rejection_sample(p_x, 0, 3., 0, 4., 2*10**6)
fig3 = plt.figure(3)
fig3.suptitle("rejection sampling")
ax3_1 = fig3.add_subplot(1,1,1)
ax3_1.set_xlabel('x')
ax3_1.set_ylabel('Probability distribution')
ax3_1.scatter(sample_x, sample_y,s=2.7)
ax3_1.plot(np.arange(10**-8, 3, 0.01), [p_x(i) for i in np.arange(10**-8, 3, 0.01)], 'r-',label='p(x)dx')
ax3_1.legend(loc='best')
fig3.savefig('2d.png')
fig3.show()
# position of each galaxy
def halo(n):
    '''
    r = x * viral radius
    as befor, set viral radius = 1
    sample_x is the possible positions, drag out some of them (literally 100)
    n = number_of_satellites
    '''
    satel_x = sample_x[0:n]  # which indicate r
    satel_phi = 2*np.pi*n2m[0:n]
    satel_theta = np.pi*n2m[-n:-1]
    return satel_x, satel_phi, satel_theta

halo_100 = halo(100)
```

```
print('Sampling distribution. See fig.3.')
print('Positions of 100 satellites :')
print('r = ')
print(halo_100[0])
print('phi = ')
print(halo_100[1])
print('theta = ')
print(halo_100[2])

'''
(e)
1000 halos, each contains 100 satellites
'''
print('\n(e) 1000 haloes and histogram')
#zeros 1000 haloes (one halo per row) and each element represents the 'x = r/ r_vir' of the satellite
haloes=np.zeros((1000,100))
for i in range(1000):
    haloes[i]=(sample_x[100*i:100*(i+1)])
radii = np.reshape(haloes,10**5)
#def N_x
N_x = lambda x: A*N_sat*(x/b)**(a-3)*np.exp(-(x/b)**c)*4*np.pi*x**2
# Plot log-log of N(x).
fig4 = plt.figure(4)
fig4.suptitle('Log-log Plot of N(x)')
ax4_1 = fig4.add_subplot(1,1,1)
ax4_1.set_xlabel('log(x)')
ax4_1.set_ylabel('N(x) = n(x)*4*pi*x^2')
ax4_1.set_xscale('log')
ax4_1.set_yscale('log')
ax4_1.plot(np.arange(10**-8, 5, 0.01), N_x(np.arange(10**-8, 5, 0.01)), label='N(x)')
amount,bin_edge,_=ax4_1.hist(radii, bins=np.logspace(np.log10(10**-4),np.log10(5.0), 21))
fig4.savefig('2e.png')
fig4.show()
print('Histogram for 1000 haloes. See fig.4.')
'''
(f)
rooting finding
Newton-Raphson method needs derivative of the original function, which is really difficult to calculate analytically.
Secant method may diverge sometimes.
so i chose the most basic one : bisection N_x1*N_x2
'''
print('\n(f) rooting finding')
def bisection_root(function, x1, x2, epsilon ,iteration):
    '''
    test the sign of f(x1)*f(x2), if it is negative then the root is in this bracket.
    (x1,x2) = range of the root
    epsilo = percision
    iteration = iteration times (in case over-shooting)
    '''
    N_x1 = function(x1)
    N_x2 = function(x2)
    if N_x1*N_x2 > 0:
        print('no root or multiple roots in this range, please reset initial bracket')
        return False
    for i in range(iteration):
        mid = (x1 + x2) / 2.
```

```
        N_mid = function(mid)
        N_x1 = function(x1)
        if N_x1 * N_mid < 0:  # mid point is the new right point x2
            x2 = mid
            if abs(N_mid) < epsilon:  # acheive the percision
                return mid
        else: # mid point is the new left point x1
            x1 = mid
            if abs(N_mid) < epsilon:  # acheive the percision
                return mid
    print("Not converge after {} iterations".format(iteration))
def maximum(a,b,function):
    '''
    use bracket method to find the maximum
    split the bracket into 3 pieces with 2 point and compare them
    if the right one is larger, then use the new left point to replace the original left point
    '''
    while b-a > 10**-8 :
        x=a+(b-a)/3.
        y=a+2*(b-a)/3.
        if function(x) < function(y):
            a=x
        else:
            b=y
    return function(a)
y=maximum(10**-4,5,N_x)
N_x_root = lambda x: A*N_sat*(x/b)**(a-3)*np.exp(-(x/b)**c)*4*np.pi*x**2-y/2
root_1 = bisection_root(N_x_root, 10**-4,1,10**-4,100)
root_2 = bisection_root(N_x_root, 1,5,10**-4,100)
print('Root_1 = ', root_1)
print('Root_2 = ', root_2)

'''
(g) histogram and Poisson distribution
'''
print('\n(g)Histogram and Poisson distribution')
radial_bin_l = bin_edge[amount.argmax(axis=0)]  # located the largrest amount from the histogram, lower limit
radial_bin_r = bin_edge[amount.argmax(axis=0)+1] # upper limit
radial_bin=[] # the radii in that largest bin
#test those satellites one by one , if their radii are in the range , save it.
for i in radii:
    if i >= radial_bin_l and i <= radial_bin_r:
        radial_bin.append(i)

def quick_sort(array,i,j):
    '''
    pick an element as pivot and partition the given array around the picked pivot.
    '''
    if i < j:
        pivot = quick_sort_process(array,i,j)
        quick_sort(array,i,pivot)
        quick_sort(array,pivot+1,j)  # do several times
    return array
def quick_sort_process(array,i,j):
    pivot = array[i]
    while i < j:
        while i < j and array[j] >= pivot:
```

```
            j -= 1
        while i < j and array[j] < pivot:
            array[i] = array[j]
            i += 1
            array[j] = array[i]
        array[i]=pivot
    return i
sorted_radii = sorted(radial_bin) # my sorting function takes too long

median = sorted_radii[int(len(sorted_radii) / 2)]
percent16 = sorted_radii[int(0.16 * len(sorted_radii))]
percent84 = sorted_radii[int(0.84 * len(sorted_radii))]
print('median = ', median)
print('16th percentile = ', percent16)
print('84th percentile = ', percent84)
# histogram : for each halo , test how many satellites' radii are in that range and count it.
hist_each_halo = np.zeros(1000)
for i in range(1000):
    count=0
    for j in haloes[i,:]:
        if j >= radial_bin_l and j <= radial_bin_r:
            count += 1
    hist_each_halo[i] = count
fig5=plt.figure(5)
ax5_1=fig5.add_subplot(1,1,1)
amount_in_bin,_,_=ax5_1.hist(hist_each_halo, bins=np.arange(0.5,101,1),density='true')
poisson_value = np.zeros(100)
for i in range(100):
    poisson_value[i] = Poisson(len(radial_bin)/1000.,i)
ax5_1.plot(np.arange(0,100,1),poisson_value,label='Poisson distribution')
ax5_1.legend(loc='upper right')
fig5.savefig('2g.png')
fig5.show()
print('Histogram and distribution see fig.5')

'''
(h) A(a,b,c)
'''
print('\n(h) A(a,b,c)')
a_h = np.arange(1.1, 2.51, 0.1)
b_h = np.arange(0.5, 2.01, 0.1)
c_h = np.arange(1.5, 4.01, 0.1)
A_h = np.zeros((len(a_h), len(b_h), len(c_h)))
for i in range(len(a_h)):
    for j in range(len(b_h)):
        for k in range(len(c_h)):
            a=a_h[i]
            b=b_h[j]
            c=c_h[k]
            A_h[i,j,k] = 1. / integrator(function_x, 10**(-16), 5., 1000)
print(len(A_h)*len(A_h[0])*len(A_h[0][0]),' values in A table')
print('e.g. A(a=1.1,b=0.5,c=1.5) = ', A_h[0,0,0])

'''
3d interpolations
Do interpolation for every single dimensions
a——>b——>c
```

```
'''
def interpolator_3d(a_ture,b_ture,c_ture,A_ture,step_size):
    '''
    do the interpolation on dimensions one by one.
    intepolate a point between two orginal points
    step_size = interpolation intervals, I used 0.05 > 0.01
    this function will generate the more points, which means make the array larger and has a narrower interval.
    '''
    a_inter = np.arange(1.1,2.51,step_size)
    b_inter = np.arange(0.5,2.01,step_size)
    c_inter = np.arange(1.5,4.01,step_size)
    A_inter_a = np.zeros((len(a_inter),len(b_ture),len(c_ture))) # means A after interpolation on a direction
    for i in range(len(b_ture)):
        for j in range(len(c_ture)):
            for k in range(len(a_inter)):
                A_inter_a[k,i,j] = neville(a_ture,A_ture[:,i,j],a_inter[k])
    # A_inter_a is the new 'true data' for b and c
    A_inter_ab = np.zeros((len(a_inter),len(b_inter),len(c_ture)))
    for l in range(len(a_inter)):
        for m in range(len(c_ture)):
            for n in range(len(b_inter)):
                A_inter_ab[l,n,m] = neville(b_ture,A_inter_a[l,:,m],b_inter[n])
    # A_inter_ab is the new 'ture data' for c
    A_inter_abc = np.zeros((len(a_inter),len(b_inter),len(c_inter)))
    for o in range (len(a_inter)):
        for p in range(len(b_inter)):
            for q in range(len(c_inter)):
                A_inter_abc[o,p,q] = neville(c_ture,A_inter_ab[o,p,:],c_inter[q])
    return A_inter_abc
#test 3d interpolation ( this is not in the question so I will not show the results)
#A_inter_3d=interpolator_3d(a_h,b_h,c_h,A_h,0.05)
def A_abc(a_ture,b_ture,c_ture,A_ture,a_new,b_new,c_new):
    '''
    define a function that can use interpolation to output an A value based on any new combination of (a,b,c)
    As same as 3d-interpolator, do one dimension by one dimension.
    '''
    A_inter_a = np.zeros((len(b_ture),len(c_ture))) # after interpolation on 'a' direction
    for i in range(len(b_ture)):
        for j in range(len(c_ture)):
            A_inter_a[i,j] = neville(a_ture,A_ture[:,i,j],a_new)
    # A_inter_a is the new 'true data'
    A_inter_ab = np.zeros(len(c_ture))
    for k in range(len(c_ture)):
        A_inter_ab[k] = neville(b_ture,A_inter_a[:,k],b_new)
    # A_inter_ab is the new 'ture data'
    A_inter_abc = neville(c_ture,A_inter_ab,c_new)
    return A_inter_abc
#test the function build on interpolator
print('test the interpolator function')
print('e.g. A(a=1.27,b=1.27,c=2.27) = ', A_abc(a_h,b_h,c_h,A_h,1.27,1.27,2.27))
```

```
"""
'''
Question 3
(a)find a,b,c
'''
data_m15=np.genfromtxt('satgals_m15.txt',skip_header=5)
#data_m14=np.genfromtxt('satgals_m14.txt',skip_header=5)
#data_m13=np.genfromtxt('satgals_m13.txt',skip_header=5)
#data_m12=np.genfromtxt('satgals_m12.txt',skip_header=5)
#data_m11=np.genfromtxt('satgals_m11.txt',skip_header=5)
#the first column is for x = r/vir
def likelihood(data,a_range,b_range,c_range):
    '''
    Sorry, I can't understand the question clearly so I just wrote something which I think might be interesting.
    For every single satellite, number density 'n' is a function of a,b,c; fixed x; set N_sat to be 100.
    Find the maximum n and the location(a,b,c) for every single satellite and then apply it to all satellites.
    '''
    x_3a = data[0]
    abc = []
    for l in range(len(x_3a)): # for a single satellite
        n = np.zeros((len(a_range),len(b_range),len(c_range)))
        for i in range(len(a_range)):
            for j in range(len(b_range)):
                for k in range(len(c_range)):
                    a = a_range[i]
                    b = b_range[j]
                    c = c_range[k]
                    n[i,j,k] = n_x(x_3a[l])
        abc.append(np.argmax(n, axis=1))
        '''
        I want to indicate the position(i,j,k) of the max value of n
        Then ,I will know the (a,b,c) that maximize the n for every satellite.
        but the 'argmax' is a little complicated in 3 dimensions array.
        It doesn't work well
        '''
    return abc
    '''
    I was looking forward to getting the poistion of peak and revert the index to (a,b,c) value.
    '''
a_3a = np.arange(1.1,2.5,0.1)
b_3a = np.arange(0.5,2,0.1)
c_3a = np.arange(1.5,4,0.1)
#likelihood(data_m15,a_3a,b_3a,c_3a)
'''
Since it didn't work ,I will not make it running now.
'''


'''
(b) Interpolator for a,b,c as function of halo mass, or fitting method
Once I get the a,b,c for each mass bin
I can use following fitting method to fit
'''
def least_square(ls_x,ls_y):
```

```
    '''
    I have written this routine on question 1 of tutorial 7.
    '''
    ls_x_mean=sum(ls_x)/len(ls_x)
    ls_y_mean=sum(ls_y)/len(ls_y)
    numer=0
    denom=0
    for i in range(len(ls_x)):
            numer += (ls_x[i] - ls_x_mean) * (ls_y[i] - ls_y_mean)
            denom += (ls_x[i] - ls_y_mean) ** 2
    w = numer / denom
    b = ls_y_mean - (w * ls_x_mean)
    return w,b
"""
print('\nQuestion 3. \n The code did not run as I had expected it
    to but I wrote the least-square fitting algorithm. Please find
    it in the source code section.')
```

2 Outputs

```
Question 1
(a) Poisson function
Poisson(1, 0) =  0.36787944117144233
Poisson(5, 10) =  0.01813278870782187
Poisson(3, 21) =  1.0193398241110193e-11
Poisson(2.6, 40) =  3.615123994937689e-33

(b) Random number generator
seed =  777
figure of random number generator please see fig.1


Question 2
(a) solve A
 a =  1.6629645031839944
 b =  1.0274077403563826
 c =  2.878139347989354
 A =  0.13691978822510378

(b) log-log plot. See fig.2.

(c) Derivative
Analytical dn(x)/dx (x=5) =  -20.665432423965154
Numerical  dn(x)/dx (x=5) =  -20.66543238754548

(d) probability distribution and the positions of satellites
Sampling distribution. See fig.3.
Positions of 100 satellites :
r =
[0.6946151470329265, 0.7310219267314602, 0.3040205414539694,
    0.25839441810422975, 1.027807936888672, 0.3030914676498381,
    1.1296044727104195, 0.1418091649404141, 0.5447139430869825,
    0.3594026879362262, 1.1172511297291998, 1.0791478707211926,
    0.7510739926994239, 0.17998057128504774, 1.3480897465759845,
    0.7272258440480012, 0.8081205175516876, 0.9496684840229044,
    0.6726386186291384, 0.6465613732093036, 1.778542642017886,
    1.0436059702924707, 0.2808786642008271, 1.0130994165806233,
    1.0801263491575153, 0.7635290007524946, 0.5919230567149196,
    0.43851557084167303, 0.28791892990364143, 1.0760334538853653,
    1.6454329057482096, 0.24594356054573324, 0.597127764080485,
    0.7060998865833868, 0.32809294305515946, 0.8641292598042791,
    0.5092276928025999, 0.9105267804243506, 0.9618478534298651,
```

```
    0.6014645062654661, 1.4384635637646896, 0.2020068565018859,
    0.42768909776691477, 1.2085595963048987, 0.4984193031408748,
    0.6739956343270603, 0.6487096460301438, 1.623763295702238,
    0.5347977552981961, 0.8659995818910158, 0.6538612037521938,
    0.4144757132227146, 0.7553696866825521, 0.029187682490338013,
    0.7506770093837914, 0.7638440221286465, 0.25415217392483763,
    1.2304781456264542, 0.5056984695669674, 0.1655755809133595,
    0.2088337850355816, 0.5507056711852045, 0.9886746874412782,
    0.9082370879945934, 0.6752797823330347, 0.8229865457861786,
    0.44685779279572513, 0.4900311743589316, 1.328554155744372,
    0.24189085749769304, 0.3360224322477655, 0.16421531122654875,
    0.38555691887346866, 1.1892608064204722, 0.37137481220979973,
    0.7608554080057888, 0.17120241301041844, 0.23560776734213823,
    0.22168045529867036, 1.0933155277764832, 1.3342786616653826,
    1.272076192236514, 1.1047358637821365, 0.5829820949222099,
    0.7876707631191733, 1.2223694826673444, 0.16535888752460376,
    0.8010616800301091, 0.8906475979674384, 0.845788408640473,
    0.07204278618255322, 0.7646080402389883, 0.47182057683858614,
    0.2902623359212106, 0.8814734800710415, 0.9613805864733036,
    0.23720991057638469, 0.541979052175256, 0.7844243523236174,
    1.039293217475281]
phi =
[2.8461199  4.86275465 0.89879912 2.67828094 4.80432042 3.12891195
 0.85379011 2.52657878 3.59534553 1.45479856 0.8386021  4.04904598
 3.30001782 5.38044842 2.25022834 0.78996347 1.02857486 1.99422214
 1.46929334 2.18146099 3.80306254 4.96048485 0.31809128 5.41717719
 1.7207461  3.99708778 5.48434005 5.67413449 1.31722548 3.94216344
 5.11182916 1.53104874 0.63673913 2.24014011 4.66422776 0.54118
 2.70926995 2.15263591 3.36180745 5.83480251 3.16749966 5.92929208
 3.44494492 0.49036601 4.15344513 0.13408845 4.40695424 3.44493658
 6.18718287 3.93208215 2.73902472 5.74537343 0.28827685 3.52727101
 2.88778487 5.73621852 2.87117533 4.72401578 5.3700832  5.34997662
 1.38215181 3.64238664 0.44541052 5.61772423 2.74818297 5.56893001
 5.55194144 1.14449953 6.08900042 0.68504765 2.3664095  1.44951018
 0.20915526 2.86916328 0.05459971 4.07910868 2.42636114 2.20920038
 5.29246299 0.63479329 1.23576997 0.57377959 5.88685435 2.81264825
 5.53228425 2.60901237 3.5365992  2.36583808 4.06495589 2.87708493
 1.80626858 4.03653864 0.29700442 0.92615063 1.58280142 1.85299593
 4.63955936 2.71185457 4.97860458 1.14084621]
theta =
[2.46786272 0.29774471 1.87432966 1.24008588 2.75065107 3.02774209
 1.94174344 2.31963649 0.29308535 2.59941275 2.91211977 1.63164754
 1.16817586 0.04673397 2.12340846 2.96914783 1.64782274 1.0867605
 0.84264676 1.598079   2.15326593 0.05491177 1.5447134  0.59505749
 1.54631497 1.78498448 1.2762653  2.24139269 2.95418573 1.66764626
 1.57949114 2.1579593  0.60161071 2.79614538 0.0388627  2.84953686
 0.58134298 2.99508388 2.93018345 2.52870156 2.90360464 0.55186322
 1.74383421 1.05584654 1.9974019  3.07352889 0.3624227  0.80993618
 2.00608724 2.77597198 0.62793029 2.12536689 2.34123979 1.91473246
 2.45125298 0.6128037  2.99320932 1.51300194 2.65007158 2.03684182
 3.09714012 1.71116983 0.40763429 0.34074092 1.01865185 0.25836407
 2.89326949 2.37725183 0.33753698 0.28277509 0.76286683 0.3494638
 1.87392527 0.28267057 2.0942809  1.93551429 0.92519246 1.82554632
 1.2100293  2.51243244 1.78229649 0.98187632 1.9018328  0.85187363
 1.43117794 1.94088237 2.78843993 0.99906496 0.39967171 1.66934138
 1.11630936 0.18233779 1.39821687 1.71619137 1.27651394 0.44921759
 0.09591433 2.12243238 2.48963038]

(e) 1000 haloes and histogram
Histogram for 1000 haloes. See fig.4.

(f) rooting finding
```

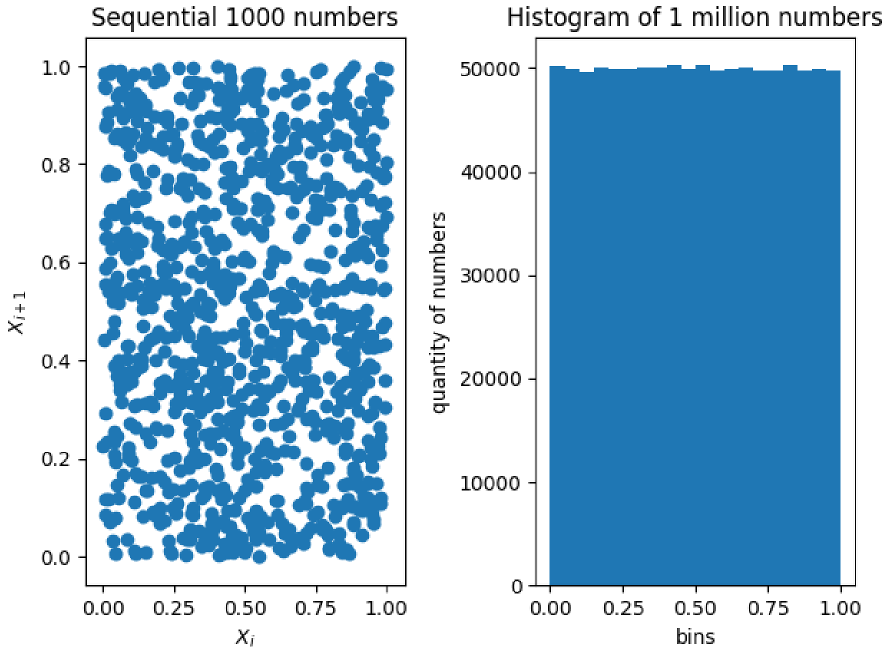

Figure 1: random number generator

```
Root_1 =  0.15418119373321532
Root_2 =  1.1334781646728516

(g)Histogram and Poisson distribution
median =  0.764393189628881
16th percentile =  0.6345954502983239
84th percentile =  0.9084267360075602
Histogram and distribution see fig.5

(h) A(a,b,c)
6240  values in A table
e.g. A(a=1.1,b=0.5,c=1.5) =  0.7673230946253291
test the interpolator function
e.g. A(a=1.27,b=1.27,c=2.27) =  0.05548288251663483

Question 3.
 The code did not run as I had expected it to but I wrote the least
    -square fitting algorithm. Please find it in the source code
    section.
```

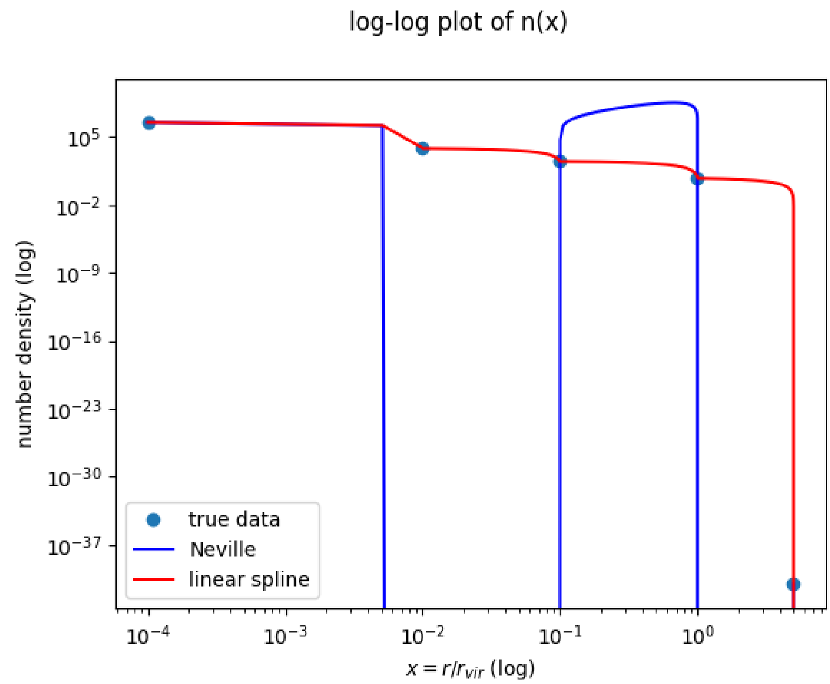

Figure 2: number density and interpolations

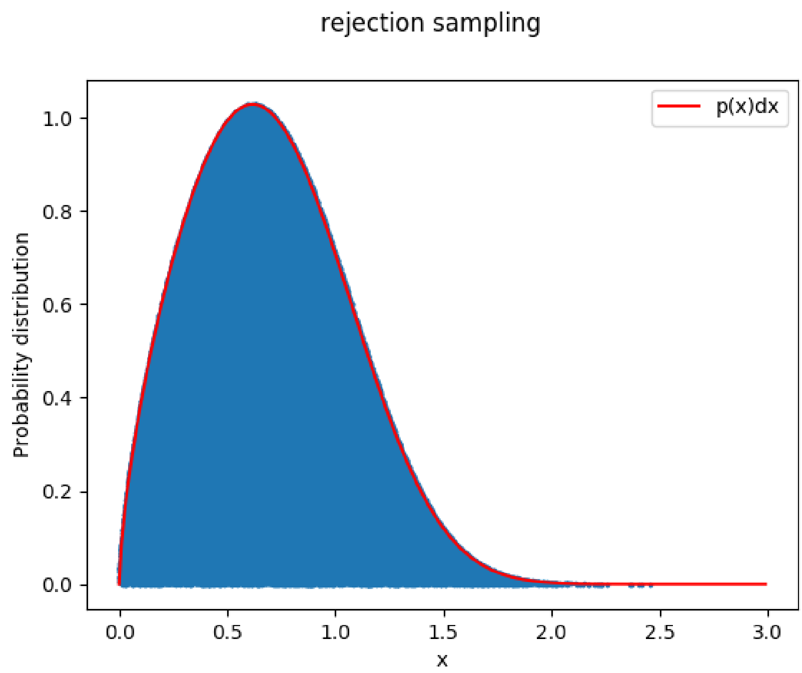

Figure 3: probability distribution

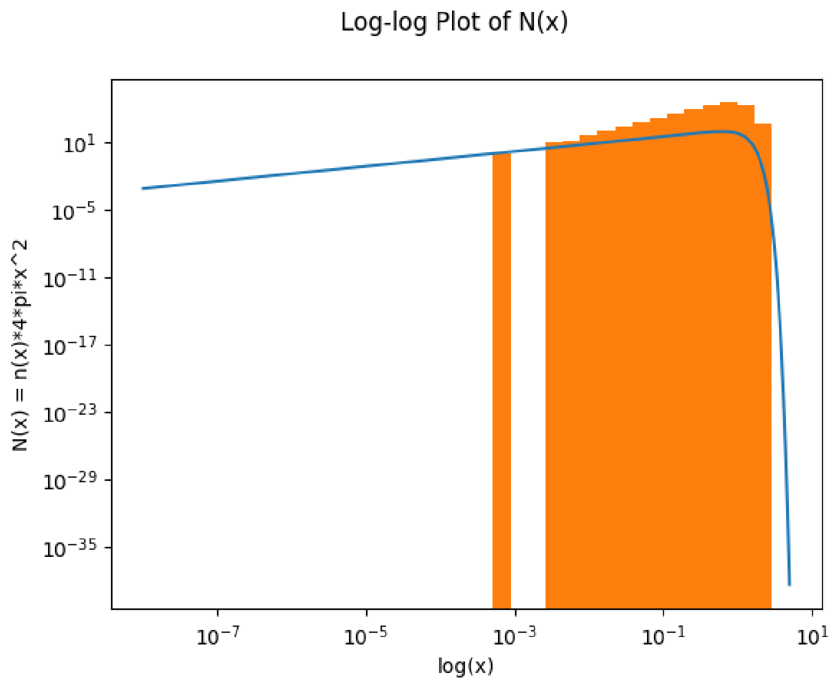

Figure 4: 1000 haloes and number of satellites in each bin

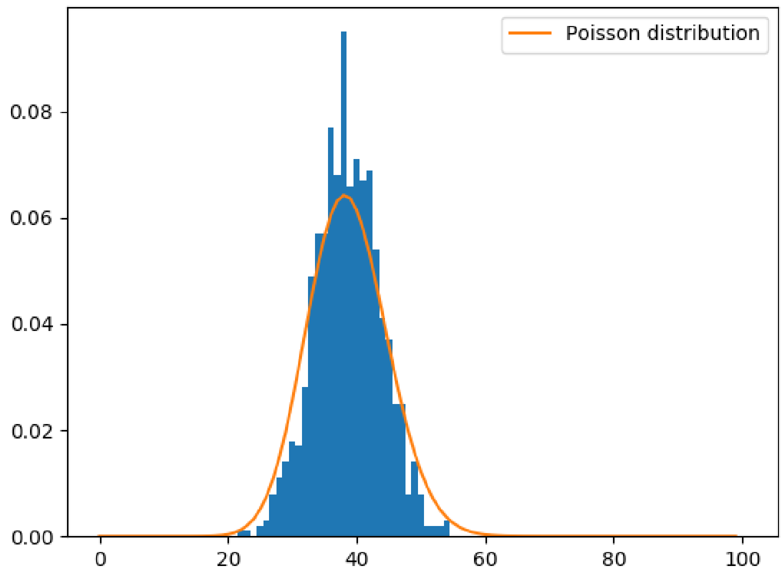

Figure 5: Poisson distribution