Hand-in Exercise 2 solution

Pengyu Liu (s2604531)

October 31, 2020

Abstract

In this document, the solutions for the two problems are given for the Hand-in Exercise 2 in the course Numerical recipes for astrophysics.

1 Question1 Dark Matter Halo

The shared modules for the this question are given by:

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

question1.py

1.1 a

We wrote a random number generator that returns random floating numbers between the lower limit and the upper limit(here[0,1]). We combined 64_bit XOR-shift and MLCG. The higher 32 bits of the output of 64_bit XOR shift is used as the input for MLCG. And we generated 1000 and 1000000 random numbers to test its quality. The code specific to this question is given by:

```python
#(a)
def RNG(seed,low,up,n):
    #Seed is the seed. Low is the lower limit and up is the upper limit.
    #n is the number of random numbers we need to generate
    ran=np.zeros(n)
    init=np.uint64(seed)
    a1=np.uint64(21)
    a2=np.uint64(35)
    a3=np.uint64(4)
    #MLCG
    m=2**64
    a=2685821657736338717
    for i in range(n):
        #64 bit xor shift
        x=init
        x=x^(x>>a1)
        x=x^(x<<a2)
        x=x^(x>>a3)
        #x as input for a multip linear congruential generator
        Id=np.uint64(int(x)*a%m)
        #use the high 32bits
        ran[i]=low+(up-low)*(Id>>np.uint(32))/(2**32)
        #update the state of next number
        init=Id
    return ran

print('run (a)')
seed=31
print('The seed is:',seed)
a1=RNG(seed,0,1,1000)

plt.scatter(a1[:-1],a1[1:])
plt.xlabel('xi')
plt.ylabel('xi+1')
plt.title('1000 random numbers')
plt.savefig('./plots/Q1ascatter.png',dpi=150)
plt.close()
```

```
a2=RNG(seed,0,1,1000000)
yerr=np.sqrt(1e6*0.05)

entries, edges, _=plt.hist(a2,bins=20,range=(0,1),histtype='step',label='sample')
#plot poisson uncertainties(the same for all bins)
bin_centers=0.5*(edges[:-1]+edges[1:])
yexp=np.ones_like(bin_centers)*1e6*0.05
plt.errorbar(bin_centers,yexp,yerr=np.sqrt(1e6*0.05),fmt='r.',capsize=3,label='Poisson')
#only show [45000,55000] to see the number more clearly
plt.ylim(45000,55000)
plt.xlabel('bin')
plt.ylabel('number')
plt.title('1000000 random numbers histogram')
plt.legend()
plt.savefig('./plots/Q1ahistogram.png',dpi=150)
plt.close()
#calculate the Pearson correlation coefficient for 100000 numbers
#r(xi,xi+1)
ri_i1=(np.mean(a2[:99999]*a2[1:100000])-np.mean(a2[:99999])*np.mean(a2[1:100000]))/(np.std(a2
    [:99999])*np.std(a2[1:100000]))
#r(xi,xi+2)
ri_i2=(np.mean(a2[:99998]*a2[2:100000])-np.mean(a2[:99998])*np.mean(a2[2:100000]))/(np.std(a2
    [:99998])*np.std(a2[2:100000]))

print('The Pearson correlation coefficient r(xi,xi+1) is: ',ri_i1)
print('The Pearson correlation coefficient r(xi,xi+2) is: ',ri_i2)
#write out result
file1=open('Q1a.txt','w')
file1.write('\nThe seed is: ')
file1.write(str(seed))
file1.write('\nThe Pearson correlation coefficient r(xi,xi+1) is:{:.6f}'.format(ri_i1))
file1.write('\nThe Pearson correlation coefficient r(xi,xi+2) is:{:.6f}'.format(ri_i2))
file1.close()
```

question1.py

The seed of this entire program is 31. Firstly, we generated 1000 randoms numbers between 0 and 1 and plot them in a scatter plot, see Figure 1.

Then we generate 1e6 random numbers between 0 and 1. And bin them in 20 bins 0.05 wide. For each bin, the theoretical value is 50000, so the poisson uncertainty is $\sqrt{50000}$. The result is shown in Figure 2.

In addittion, we calculate the Pearson correlation coefficient $r_{x_i x_{i+1}}$ and $r_{x_i x_{i+2}}$ for 1e5 numbers. The result is given by:

```

The seed is: 31
The Pearson correlation coefficient r(xi,xi+1) is:0.003535
The Pearson correlation coefficient r(xi,xi+2) is:0.001840
```

Q1a.txt

The absolute values are much smaller than 1, which means that these random numbers have little correlations with each other. Our random number generator has a good quality and can be used for following steps.

1.2 b

We used the transformation method to generate radial distribution of particles. At first, we integrate Hernquist profile from 0 to r and that is $\frac{M*r^2}{(r+a)^2}$. Then we need to normalize it by dividing Mdm(r=∞) and get the CDF=$\frac{r^2}{(r+a)^2}$. The CDF is also the enclosed mass fraction. Then we invert the CDF so that we can sample points: $y = \frac{a*\sqrt{x}}{1-\sqrt{x}}$. x is random numbers between 0 and 1. We use our RNG(random number generators) to generate 1e6 points in [0,1] and use the invert CDF to transfer them to random numbers following the Hernquist distribution.

The code specific to this question is give by:

```
#(b)
print('run (b)')
def Hernquist(r):
    Mdm=1e12
    #unit of a:kpc
    a=80
    return Mdm*a/(2*np.pi*r*np.power(r+a,3))


```

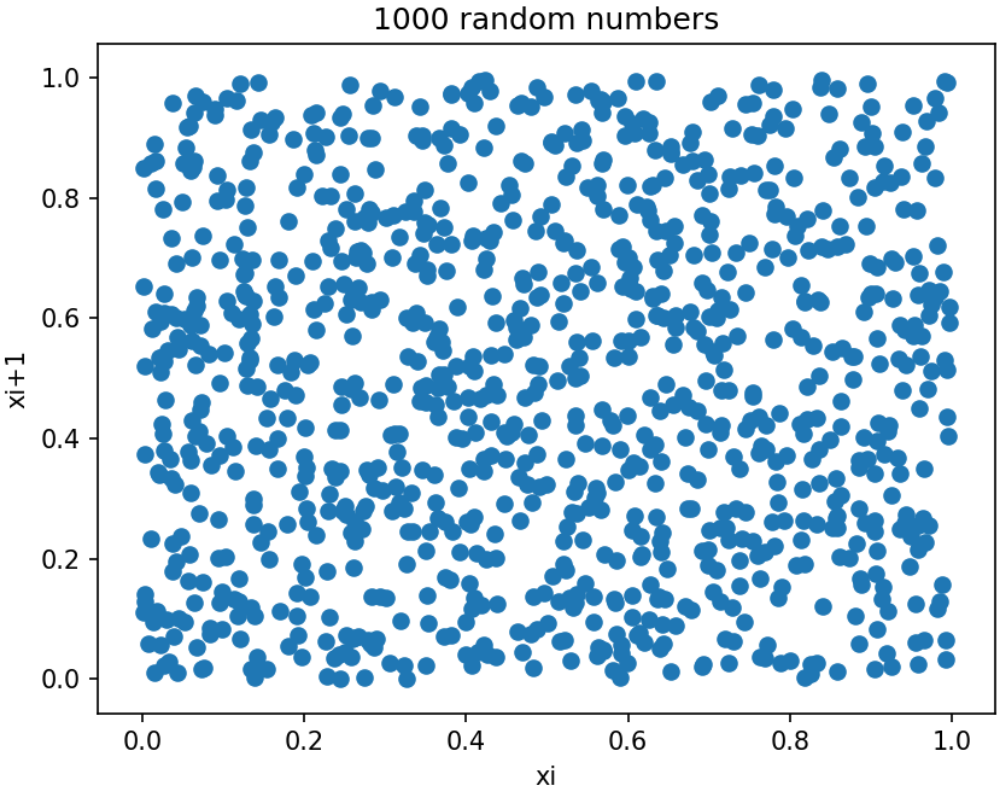

Figure 1: Sequential random numbers against each other(x_{i+1} vs x_i). We can see that these points scattered irregularly, which shows our random numbers have little influence on each other.

```
def cdf_Hernq(r):
    a=80
    #the cumulative distribution function of the Hernquist function(normalized to 1)
    return np.power(r/(a+r),2)

#the inverse of CDF
def invercdf(x):
    a=80
    return a*np.sqrt(x)/(1-np.sqrt(x))
#sampling, a2 is the result of 1a(1e6 uniformed numbers in [0,1])
inver=invercdf(a2)
inver_sum,inver_bin=np.histogram(inver,bins=10000,range=(0,10000))
inver_frac=np.zeros_like(inver_sum)
#the enclosed fraction of particles
for i in range(inver_frac.size):
    inver_frac[i]=np.sum(inver_sum[:i+1])
inver_frac=inver_frac/np.sum(inver_sum)
plt.plot(inver_bin[1:],inver_frac,label='sampled particles fraction')

#the expected amount of enclosed fraction of mass
radius=np.linspace(0,10000,1000)
expected_frac=cdf_Hernq(radius)
plt.plot(radius,expected_frac,label='expect mass fraction')
plt.legend()
plt.xlabel('r[kpc]')
plt.ylabel('fraction')
plt.savefig('./plots/Q1bfraction.png',dpi=150)
plt.close()
```

question1.py

We compare the enclosed fraction of particles at a certain radius with the expected amount of enclosed fraction of mass in Figure 3.

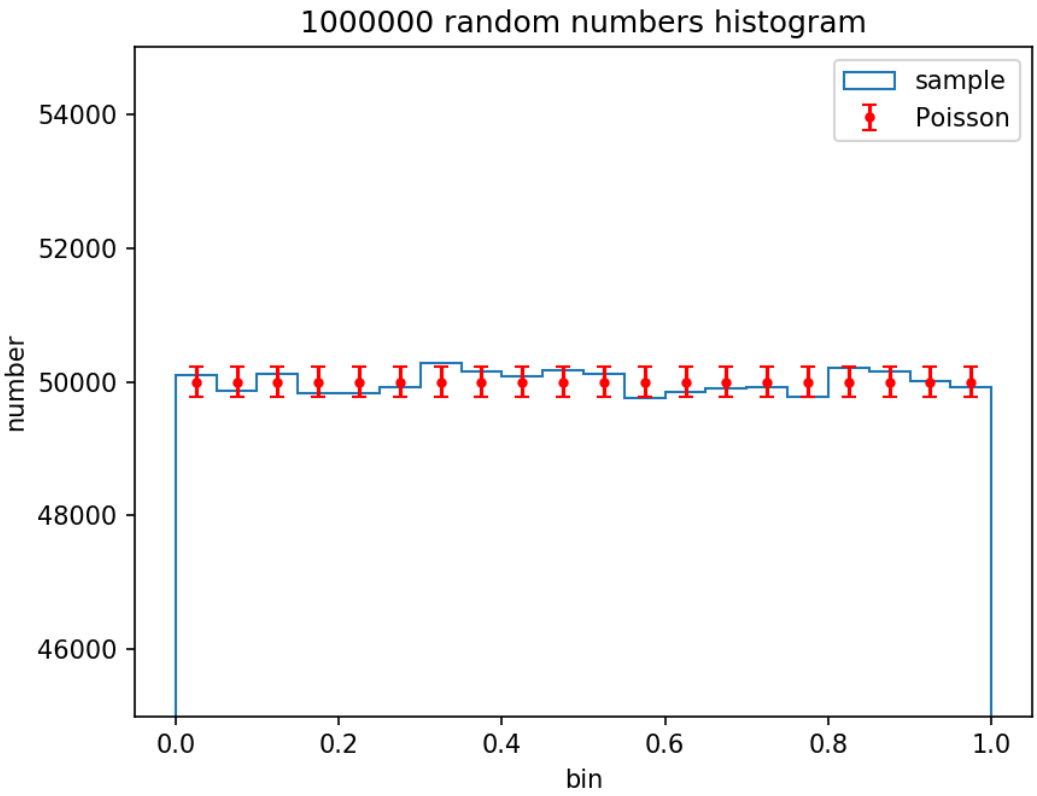

Figure 2: Histogram of 1e6 random numbers. The blue line shows the histogram of the 1e6 numbers and the red error bar shows the poisson uncertainty. We can see that the value of each bin agrees with the mean value within the poisson uncertainty.

1.3 c

We generated a 3D distribution of 1e3 particles in a Hernquist profile. We used 1000 random numbers for r, another 1000 random numbers for θ and another random numbers for ϕ. Because we need θ and ϕ distribute uniformly on a sphere, we should use the inverse transform method to calculate p(θ) and p(ϕ). The probability of having a point in an element area dA should be constant over the sphere. So θ=arccos(1-2*x1) and ϕ=2*pi*x2, where x1 and x2 are uniformed random numbers in [0,1]. The code specific to this question is give by:

```
#(c)
print('run (c)')
#phi using 1000 random numbers in a2
phi=2*np.pi*np.copy(a2[10000:11000])
#theta using another 1000 random numbers in a2
theta=np.arccos(1-2*np.copy(a2[20000:21000]))
x=np.copy(inver[:1000])*np.sin(theta)*np.cos(phi)/(4*np.pi*np.sqrt(inver[:1000]))
y=np.copy(inver[:1000])*np.sin(theta)*np.sin(phi)/(4*np.pi*np.sqrt(inver[:1000]))
z=np.copy(inver[:1000])*np.cos(theta)/(4*np.pi*np.sqrt(inver[:1000]))
#3d scatter plot
fig=plt.figure()
ax=fig.add_subplot(111,projection='3d')
ax.scatter(x,y,z)
ax.set_xlabel('x[kpc]')
ax.set_ylabel('y[kpc]')
ax.set_zlabel('z[kpc]')
ax.set_title('3D scatter plot of 1000 particles')
plt.savefig('./plots/Q1c3d.png',dpi=150)
plt.close()
#make a plot of theta and phi
fig2=plt.figure()
ax2=fig2.add_subplot(111,projection='3d')
ax2.scatter(np.sin(theta)*np.cos(phi),np.sin(theta)*np.sin(phi),np.cos(theta))
ax2.set_xlabel('x[kpc]')
ax2.set_ylabel('y[kpc]')
ax2.set_zlabel('z[kpc]')
```

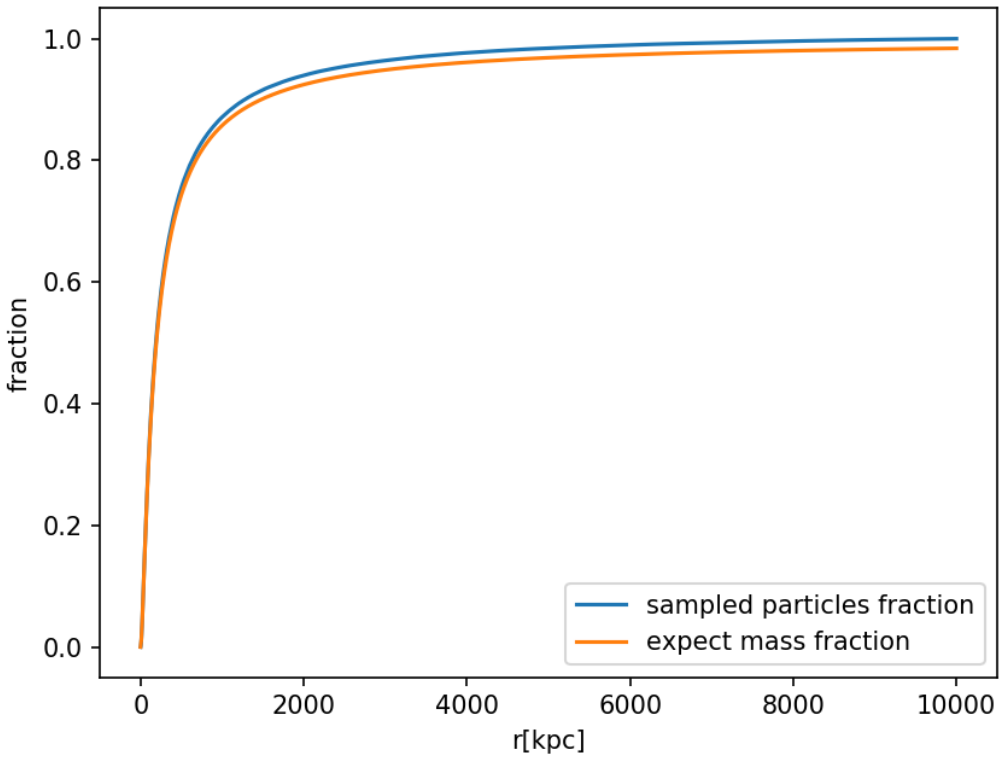

Figure 3: Enclosed fraction of particles and expected encloses mass fraction. The blue is our sampled results and the orange line is the expected value. The sampled results are slightly higher than the expected value at large radius. The reason is that the sampled value reaches 1 at the largest radius we generated but the expected value goes to 1 only when radius reaches infinity. The radius we generate always has a certain value not infinity. When the number of points increases, the error will become smaller and the sampled line will become very close to the expected one.

```
ax2.set_title('random numbers on a sphere')
plt.savefig('./plots/Q1csphere.png',dpi=150)
plt.close()
```

question1.py

The 3D scatter plot is given by Figure 4. And the distribution of θ and ϕ on a sphere is given by Figure 5.

1.4 d

We wrote a differentiation routine using ridder's method and calculated $\frac{d\rho(x)}{dr}$ at r=1.2a numerically and analytically. The code specific to this question is give by:

```
#(d)
print('run (d)')
#ridder's method for differentiation
def ridder_der(f,x,h0=0.1,m=10,d=2):
    #m:order; h0:initial interval(need to be adjusted for different functions)
    #two arrays to store previous and current results
    rdr=np.zeros(m)
    rdr2=np.zeros(m)
    hh=h0
    for i in range(m):
        #central difference
        rdr[i]=0.5*(f(x+hh)-f(x-hh))/hh
        hh=hh/d

    result=np.copy(rdr[0])
    den=1
    mul=d**2
```

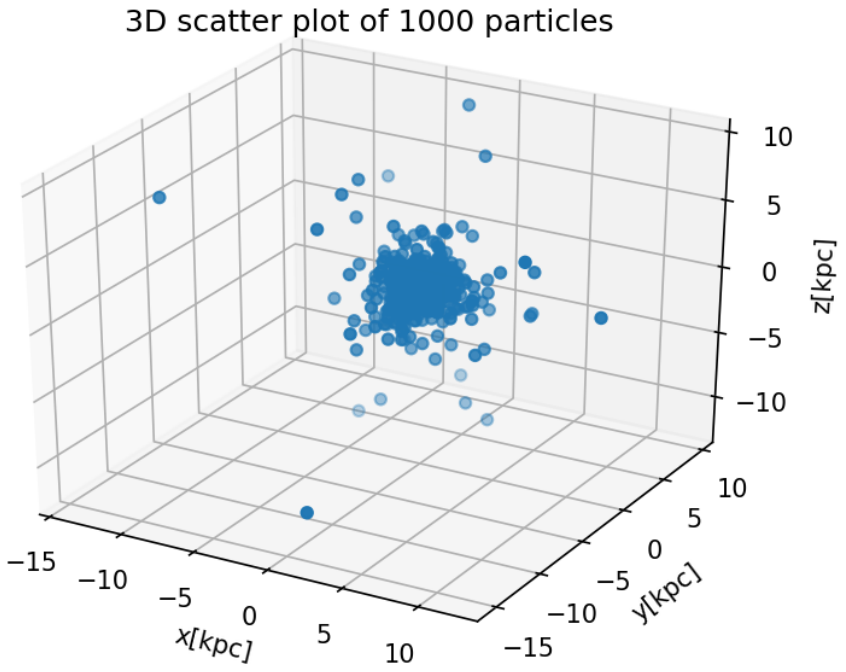

Figure 4: The 3D scatter plot of 1000 random points in a Hernquist profile.

```
    #initial error(larger number)
    err=10
    for i in range(1,m):
        den*=mul
        for j in range(m-i):
            #calculate new values
            rdr2[j]=(den*rdr[j+1]-rdr[j])/(den-1)
            errn=max(abs(rdr2[j]-rdr[j]),abs(rdr2[j]-rdr[j+1]))
            if errn<err:
                #compare recent results: if error goes down, go to higher order
                err=errn
                result=rdr2[j]
            #terminate early if the error grows
            elif errn>3*err:
                return result
        #put new values to the old array
        rdr=np.copy(rdr2)
    return result

numer_result=ridder_der(Hernquist,x=1.2*80,h0=1,m=10,d=2)
print('The numerical result is:{:.10f}'.format(numer_result))

#analytical result of Hernquist
def Hernquist_rd(r):
    Mdm=1e12
    #unit of a:kpc
    a=80
    return (Mdm*a*0.5/np.pi)*(-(1/(np.power(r,2)*np.power(r+a,3)))-3/(r*np.power(r+a,4)))
analy_result=Hernquist_rd(1.2*80)
print('The analytical result is:{:.10f}'.format(analy_result))

#write out result
file2=open('Q1d.txt','w')
file2.write('The numerical result is:{:.10f}'.format(numer_result))
file2.write('\nThe analytical result is:{:.10f}'.format(analy_result))
file2.close()
```

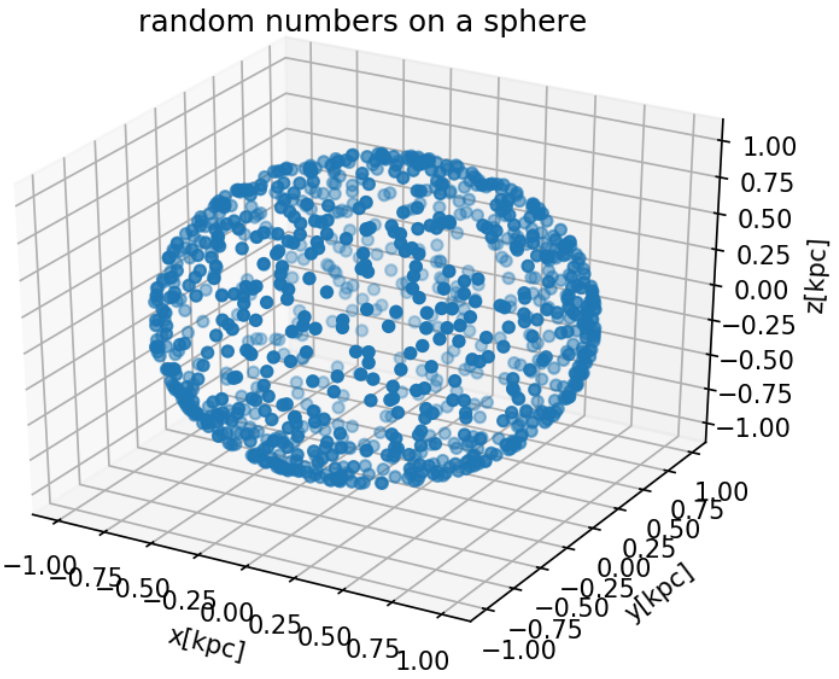

Figure 5: The distribution of θ and ϕ on a sphere. We can see that they are uniformed distributed on a sphere.

question1.py

We chose the maximum order=10 and initial interval h=0.1 for ridder's method. The result is given by:

```
The numerical result is:-668.0899855228
The analytical result is:-668.0899855230
```

Q1d.txt

We can see that the numerical value agrees with the analytical value with a relative error close to 10^{-12}, which is very close to the limit of ridder's method(10^{-14}).

1.5 e

We used Newton_Raphson method to find the root when delta=200 and 500. The code specific to this question is give by:

```
#(e)
print('run (e)')
def NR(f,a,h=0.001):
    #Newton-Raphson method
    #use ridder's differentiation to calculate the derivative, h is the initial h0 for
    ridder_der
    #accuracy
    acc=1e-6
    #the maximum number of bisection iteration
    term=int(1e3)
    x0=a
    for i in range(term):
        #notice: need to adjust h for different function
        x0_der=ridder_der(f,x0,h0=h)
        if x0_der==0:
            #if the derivative ==0, return this point, though it's not a good result
            return x0
        x1=x0-f(x0)/x0_der
```

```
        if abs(f(x1))<acc:
            #find the root successfully
            return x1
        x0=x1
    #don't find a root with an accuracy smaller than acc under the maximum number of
    iterations
    #return the most recent one
    return x0

def f1(x):
    pc=150
    return Hernquist(x)-200*pc

def f2(x):
    pc=150
    return Hernquist(x)-500*pc

def mass(x):
    Mdm=1e12
    a=80
    return Mdm*np.power(x,2)/np.power(a+x,2)

#using Newton-Raphson method to find the root
R200=NR(f1,1,h=0.0001)
R500=NR(f2,1,h=0.0001)
M200=mass(R200)
M500=mass(R500)
#check
print('check function(R200) value:',f1(R200))

print('R200[kpc] is:',R200)
print('M200[Msolar] is:',M200)


#check
print('check function(R500) value:',f2(R500))
print('R500[kpc] is:',R500)
print('M500[Msolar] is:',M500)
#write out results
file3=open('Q1e.txt','w')
file3.write('check function(R200) value:{:}'.format(f1(R200)))
file3.write('\nR200[kpc] is:{:.6f}'.format(R200))
file3.write('\nM200[solar] is:{:.6e}'.format(M200))
file3.write('\ncheck function(R500) value:{:}'.format(f2(R500)))
file3.write('\nR500[kpc] is:{:.6f}'.format(R500))
file3.write('\nM500[solar] is:{:.6e}'.format(M500))
file3.close()
```

question1.py

The results are given by:

```
check function(R200) value:5.275069270282984e-10
R200[kpc] is:88.582774
M200[solar] is:2.761037e+11
check function(R500) value:6.83940015733242e-10
R500[kpc] is:60.808375
M500[solar] is:1.864961e+11
```

Q1e.txt

We also show the function value at the root we found, and they are very close to 0. Our root algorithm finds good results. One thing needs to notice is that because we used ridder's method to calculate the differentiation, the initial interval needs to be adjusted for different functions so that it can give us the best results.

1.6 f

We used the downhill simplex method to find the minimum of this potential. Because downhill simplex method requires sort algorithm, we also wrote the mergesort to sort the array and returns its index after sorting. The code specific to this question is give by:

```
#(f)
print('run (f)')
def mergesort(arr,left,right,index):
    #mergesort arr from arr[left] to arr[right]
```

```
    #caution: arr will be changed to the sorted array!
    #Copy it before using this function if you want to keep the initial arr.
    #also return the index after sorting
    #recursion
    if left<right:
        mid=int(0.5*(left+right))
        #sort the left part
        mergesort(arr,left,mid,index)
        #sort the right part
        mergesort(arr,mid+1,right,index)

        #merge and sort arr from arr[left] to arr[right]. mid is the last index of the left part
        #Both of arr[left:mid+1] and arr[mid+1:right] are already sorted
        i=left
        j=mid+1
        arr_new=np.zeros_like(arr[left:right+1])
        index_new=np.zeros_like(index[left:right+1])
        d=0
        while i<=mid and j<=right:
            #put the smaller one to arr_new[d]
            if arr[i]<=arr[j]:
                arr_new[d]=arr[i]
                index_new[d]=index[i]
                i+=1
            else:
                arr_new[d]=arr[j]
                index_new[d]=index[j]
                j+=1
            d+=1
        #One part is ended and all of the rest of another part are smaller or larger than previous elements.
        if i<=mid:
            #put the rest of the left part to arr_new(because are they sorted, we put them directly to arr_new)
            arr_new[d:]=np.copy(arr[i:mid+1])
            index_new[d:]=np.copy(index[i:mid+1])
        else:
            #put the rest of the right part to arr_new
            arr_new[d:]=np.copy(arr[j:right+1])
            index_new[d:]=np.copy(index[j:right+1])
        #put arr_new back to arr
        arr[left:right+1]=arr_new
        index[left:right+1]=index_new

#minization
def downhill_simplex(f,init_x):
    #downhill method to find the minimum of f
    #N dimension requires N+1 points
    #dimention
    N=init_x.size
    #generate N+1 points
    step1=-100
    step2=10
    point=np.zeros((N+1,N))
    point[0]=np.copy(init_x)
    fun=np.zeros(N+1)
    fun[0]=f(point[0])
    for i in range(N):
        point[i+1,0]=point[i,0]+step1
        point[i+1,0]=point[i,0]+step2
        fun[i+1]=f(point[i+1])
    #maximum number of iteration
    term=10000
    #target accuracy
    acc=1e-10
    #record the best point of each iterations
    trace=np.copy(point[0])

    #start iteration
    for i in range(term):
        #sort
        fun_ind=np.arange(N+1)
        mergesort(fun,0,N,fun_ind)
        point=point[fun_ind]
        #check if find the minimum
```

```
        ran=abs((fun[0]-fun[-1])/(0.5*(fun[0]+fun[-1])))
        if ran<acc:
            #find the best guess x0
            trace=np.vstack((trace,point[0]))
            return point[0],trace
        #centroid of the first N points
        xcen=np.mean(point[:-1],axis=0)
        #propose a new point by reflecting
        xtry=2*xcen-point[-1]
        if f(xtry)>=fun[0] and f(xtry)<fun[-1]:
            #new points is better but not the best,accept it
            point[-1]=np.copy(xtry)
            fun[-1]=f(xtry)
            trace=np.vstack((trace,point[0]))
        elif f(xtry)<fun[0]:
            #new point is the very best
            #propose a new point by expanding further
            xexp=2*xtry-xcen
            if f(xexp)<f(xtry):
                #xexp is better, accept it
                point[-1]=np.copy(xexp)
                fun[-1]=f(xexp)
                trace=np.vstack((trace,xexp))
            else:
                #accept the reflected one
                point[-1]=np.copy(xtry)
                fun[-1]=f(xtry)
                trace=np.vstack((trace,xtry))
        else:
            #xtry is the baddest
            trace=np.vstack((trace,point[0]))
            #propose a new point by contracting
            xtry=0.5*(xcen+point[-1])
            if f(xtry)<fun[-1]:
                #accept
                point[-1]=np.copy(xtry)
                fun[-1]=f(xtry)
            else:
                #zoom on the best point by contracting all others to it
                point[1:]=0.5*(xcen+point[1:])
                for i in range(N):
                    fun[i+1]=f(point[i+1])
    #the target accuracy is not reached after the maximum of iterations
    #return the current best one and trace
    return point[0],trace

def Hernquist2d(var):
    x,y=var
    a=80
    Mdm=1e12
    G=4.3*1e-6 #unit:kpc km^2 s^-2 Msol^-1
    return -G*Mdm/(a+np.sqrt(np.power(x-1.3,2)+2*np.power(y-4.2,2)))

def distance(point,end):
    return np.sqrt(np.sum(np.power(point-end,2)))

final_point,trace= downhill_simplex(Hernquist2d,np.array([-1000,-200]))

#calculate the distance from the final point
dis=np.zeros(trace.shape[0])
for i in range(trace.shape[0]):
    dis[i]=distance(trace[i],trace[-1])

print('The minimum of this potential is {:}'.format(Hernquist2d(final_point)))
print('The final point is',final_point)

file4=open('Q1f.txt','w')
file4.write('The minimum of this potential is {:.6f}'.format(Hernquist2d(final_point)))
file4.write('\nThe final point(x,y) is:{:}'.format(final_point))
file4.close()

plt.plot(dis)
plt.xlabel('the number of iterations')
plt.ylabel('distance')
plt.savefig('./plots/Q1f.png',dpi=150)
plt.close()
```

question1.py

The final point and its potential is given by:

```
The minimum of this potential is -53749.999979
The final point(x,y) is:[1.30000001 4.20000002]
```

Q1f.txt

It matched with the analytical result (1.3,4.2) with an error about 1e-8. The distance from the final point at each iteration is given by Figure 6.

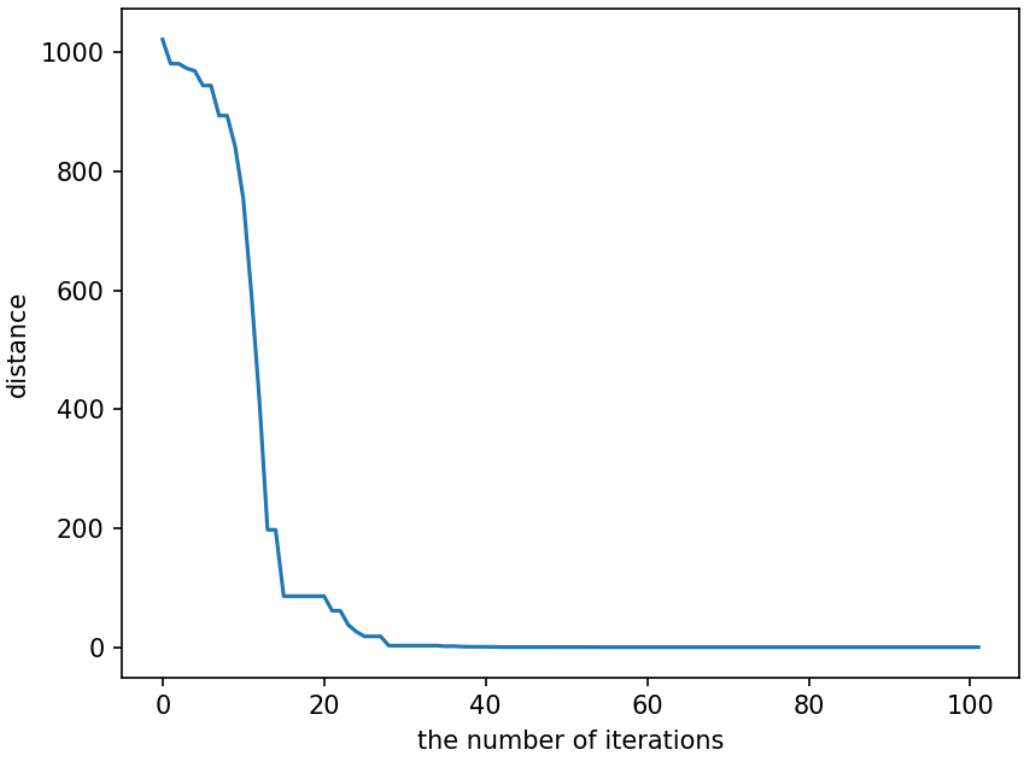

Figure 6: The number of iterations versus the distance from the final point. We can see that it almost successfully reached the final location after 30 iterations. But one thing needs to notice that we need to choose the initial points in a good way. We are already given one point and need to choose another 2 points. If we add the same value to each points in all dimensions, it might not go to the minimum. Instead, these points go close to each other and end at some wrong location. This phenomenon needs to be explored further.

2 question2 Satellite galaxies around a massive central Part2

The shared modules are given by:

```python
#question2
import numpy as np
import matplotlib.pyplot as plt

seed=31
def RNG(seed,low,up,n):
    #Seed is the seed. Low is the lower limit and up is the upper limit.
    #n is the number of random numbers we need to generate
    ran=np.zeros(n)
    init=np.uint64(seed)
    a1=np.uint64(21)
    a2=np.uint64(35)
    a3=np.uint64(4)
```

```
    #MLCG
    m=2**64
    a=2685821657736338717
    for i in range(n):
        #64 bit xor shift
        x=init
        x=x^(x>>a1)
        x=x^(x<<a2)
        x=x^(x>>a3)
        #x as input for a multip linear congruential generator
        Id=np.uint64(int(x)*a%m)
        #use the high 32bits
        ran[i]=low+(up-low)*(Id>>np.uint(32))/(2**32)
        #update the state of next number
        init=Id
    return ran
```

question2.py

2.1 a

We wrote a 1D minimization algorithm using the bracket to find the bracket and using the golden search to tighten the bracket until we reached our target accuracy. Finding the maximum of N(x) equals to finding the minimum of -N(x).

The code of this subquestion is given by:

```
#(a)
print('run (a)')
def bracketmin(f,x1,x2):
    a=x1
    b=x2
    if f(a)<f(b):
        #switch a and b to ensure that f(b)<f(a)
        a,b=b,a
    #propose a new point
    #golden ratio
    w=1.618
    c=b+(b-a)*w
    while f(c)<f(b):
        #use a,b,c to find a new point by fitting a parabola
        d=b-0.5*(np.power(b-a,2)*(f(b)-f(c))-np.power(b-c,2)*(f(b)-f(a)))/((b-a)*(f(b)-f(c))-(
    b-c)*(f(b)-f(a)))
        if ((d-c)*(d-b))<0:
            #d is between b and c
            if f(d)<f(c):
                #find the bracket
                return b,d,c
            elif f(d)>f(b):
                #find the bracket
                return a,b,d
            #the parabola is a bad fit
            d=c+(c-b)*w
        else:
            #d is beyond c
            if abs(b-d)>20*abs(c-b):
                #d is too far away. take another step
                d=c+(c-b)*w
        #move all points over
        a,b,c=b,c,d
#         if f(a)<f(b):
#             #always ensure f(b)<f(a)
#             a,b=b,a

    return a,b,c

def golden_search_min(f,bracket):
    #bracket is a tuple(a<c)
    a,b,c=bracket
    acc=1e-6
    w=0.38197
    #maximum of iteration
    term=int(1e6)
    for i in range(term):
```

```
        #identify the larger interval
        #choose d inside the interval in a self-similar way
        if abs(a-b)>abs(b-c):
            d=b+(a-b)*w
        else:
            d=b+(c-b)*w
        if abs(c-a)>acc:
            if f(d)<f(b):
                #tighten towards d
                if ((d-a)*(d-b))<0:
                    #d is in between a and b
                    c,b=b,d
                else:
                    #d is in between b and c
                    a,b=b,d
            else:
                if ((d-a)*(d-b))<0:
                    #d is in between a and b
                    a=d
                else:
                    #d is in between b and c
                    c=d
        else:
            #reached the accuracy
            if f(d)<f(b):
                return d
            else:
                return b

def Nsate(x):
    a=2.4
    b=0.25
    c=1.7
    Nsat=100
    A=256/(5*np.power(np.pi,1.5))
    return -4*np.pi*A*Nsat*np.power(x,a-1)*np.power(1/b,a-3)*np.exp(-np.power(x/b,c))


bracket=bracketmin(Nsate,0,5)
x_max=golden_search_min(Nsate,bracket)
N_max=-Nsate(x_max)
print('The braket is: ',bracket)
print('x at the maximum is:',x_max)
print('N(x) at the maximum is:', N_max)
#write out result
file1=open('Q2a.txt','w')
file1.write('The braket after bracketmin is:{:}'.format(bracket))
file1.write('\nx at the maximum is:{:.10f}'.format(x_max))
file1.write('\nN(x) at the maximum is:{:.10f}'.format(N_max))
file1.close()
```

question2.py

The results are given by:

```
The braket after bracketmin is:(0, 5, 13.09)
x at the maximum is:0.2230177593
N(x) at the maximum is:270.1096030529
```

Q2a.txt

Given (0,5), the bracket algorithm returns (0,5,13.09). Though it exceeds xmax=5, we set N(x)=0 where x≥5, because N(x)≥0 changes smoothly to 0. It does not influence the maximization process.

2.2 b

Because this question only concerns the radial dimension, we can transfer it to 1d sampling, ignoring θ and ϕ. We used the rejection sample to generate the 1000 points.
The code of this subquestion is given by:

```
#(b)
print('run (b)')
def N_norm(x):
    a=2.4
    b=0.25
```

```
    c=1.7
    A=256/(5*np.power(np.pi,1.5))
    return 4*np.pi*A*np.power(x,a-1)*np.power(1/b,a-3)*np.exp(-np.power(x/b,c))

def gx(x):
    #normalize N to 1
    N_max=2.7010960100989907
    return N_norm(x)/N_max
#rejection sample
def reject_sample(f,x_ran,num):
    #num is the number of points we want to generate
    point=np.zeros(num)
    #seed is the seed of we set for the entire program(global variable)
    ths=np.copy(seed)
    a,b=x_ran
    count=0
    #the number of random numbers we generate every time
    smal=10000
    while count<num:
        y1=RNG(seed=ths,low=0,up=1,n=smal)
        ths+=11
        x1=RNG(seed=ths,low=a,up=b,n=smal)
        ths+=3
        i=0
        while i in range(smal) and count<num:
            if y1[i]<=f(x1[i]):
                #accept it
                point[count]=x1[i]
                count+=1
            i+=1
    return point
#generate 10000 points
po_num=10000
point_sam=reject_sample(gx,x_ran=(0,5),num=po_num)
N_sam=np.zeros(po_num)
for i in range(po_num):
    N_sam[i]=-Nsate(point_sam[0])

#make a plot
arra=np.arange(0.0001,5,0.001)
#normalize N(x)
plt.plot(np.log10(arra),np.log10(-Nsate(arra)/100),label='N(x)')
#log bin
bins=np.logspace(np.log10(0.0001),np.log10(5), 21)
hist,bin_edges=np.histogram(point_sam,bins=bins)
hist2=np.copy(hist!=0)
bins_cen=0.5*(bins[:-1]+bins[1:])
width=bin_edges[1:]-bin_edges[:-1]
plt.plot(np.log10(bins_cen[hist2]),np.log10(hist[hist2]/width[hist2])-4,'-o',label='sampled')
plt.legend()
plt.xlabel('log10(x)')
plt.ylabel('log10(N(x))')
plt.savefig('./plots/Q2b.png',dpi=150)
plt.close()
```

question2.py

Figure 7 shows the histogram of our 10000 sampled points and the normalized N(x).

2.3 c

At first, we selected 100 unique random galaxies from the 10000 galaxies in (b) with equal probability and not rejecting any drawn. We assume that the 10000 galaxies are unique even for those whose radii are the same, because the θ and ϕ can be different. Then we can transfer this question to generate 100 unique uniformed integers from 0 and 9999. We draw 100 galaxies out using these 100 indexes. I referred to the Fisher Yates shuffle mentioned in T5.3. The algorithm is:
1.generate a random integer(d1) from 0 and the number of undrawned indexes
2.count from the low end and pick the d1th number out from undrawned indexes
3.put it into the drawn list and inactive it in the original indexes list
4.go back to 1 until we draw 100 indexes

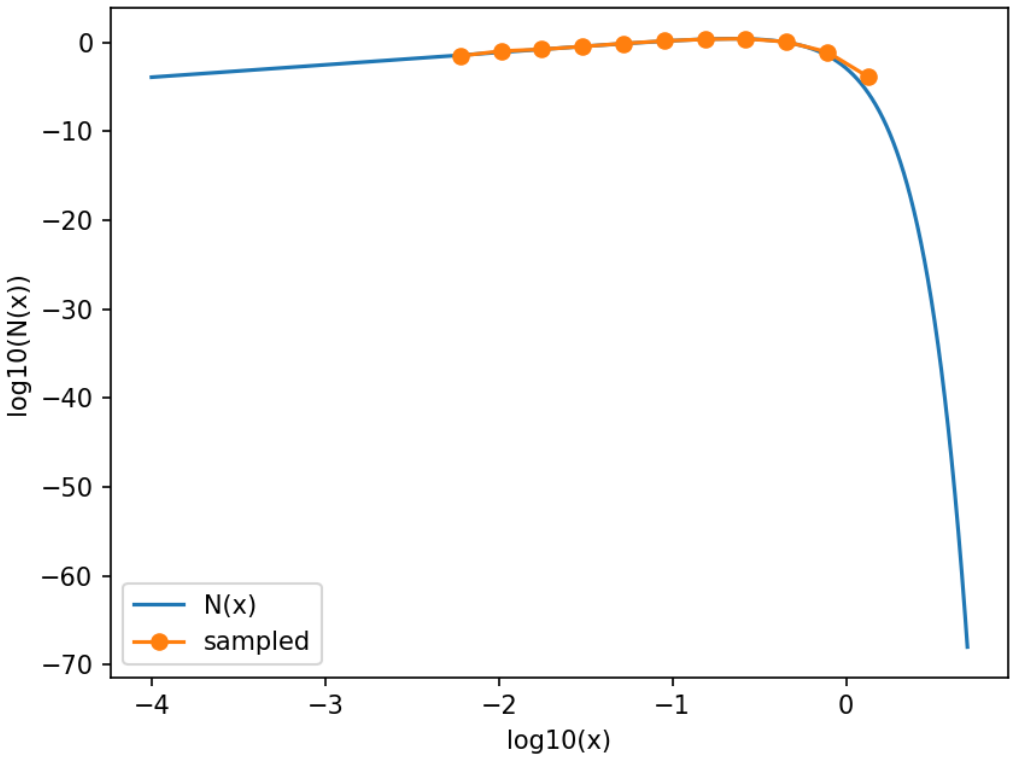

Figure 7: The histogram of our 1000 sampled point and N(x). Both of them are normalized so that the integral equals 1. The blue line is the probability distribution of N(x); the orange line shows the histogram divided by the bin width and the total number 10000. We can see that they agree with each other. Because the probability is very small when x≤1e-3 and x≥1, the rejection sample hardly generate points in these ranges when the number of points is small. That's why we only see points showed by the orange range.

Then we used mergesort to sort the 100 drawn galaxies from smallest to largest and plot the number of galaxies within r.
The code of this subquestion is given by:

```
#(c)
print('run (c)')
def mergesort(arr,left,right,index):
    #mergesort arr from arr[left] to arr[right]
    #caution: arr will be changed to the sorted array!
    #Copy it before using this function if you want to keep the initial arr.
    #also return the index after sorting
    #recursion
    if left<right:
        mid=int(0.5*(left+right))
        #sort the left part
        mergesort(arr,left,mid,index)
        #sort the right part
        mergesort(arr,mid+1,right,index)

        #merge and sort arr from arr[left] to arr[right]. mid is the last index of the left part
        #Both of arr[left:mid+1] and arr[mid+1:right] are already sorted
        i=left
        j=mid+1
        arr_new=np.zeros_like(arr[left:right+1])
        index_new=np.zeros_like(index[left:right+1])
        d=0
        while i<=mid and j<=right:
            #put the smaller one to arr_new[d]
            if arr[i]<=arr[j]:
                arr_new[d]=arr[i]
                index_new[d]=index[i]
```

```
                i+=1
            else:
                arr_new[d]=arr[j]
                index_new[d]=index[j]
                j+=1
            d+=1
        #One part is ended and all of the rest of another part are smaller or larger than previous elements.
        if i<=mid:
            #put the rest of the left part to arr_new(because are they sorted, we put them directly to arr_new)
            arr_new[d:]=np.copy(arr[i:mid+1])
            index_new[d:]=np.copy(index[i:mid+1])
        else:
            #put the rest of the right part to arr_new
            arr_new[d:]=np.copy(arr[j:right+1])
            index_new[d:]=np.copy(index[j:right+1])
        #put arr_new back to arr
        arr[left:right+1]=arr_new
        index[left:right+1]=index_new

#select 100 random satellite galaxies
sele_num=100
drawn=np.zeros(sele_num)
record=np.ones(10000,dtype='bool')
ths=seed
#refer to Fisher Yates shuffle
for i in range(100):
    #generate a random index from 0 and the number of remaining index list
    d1=np.round(RNG(seed=ths,low=0,up=10000-1-i,n=1))
    ths+=1
    temp=0
    #count the d1th number from the remaining index list
    for j in range(10000):
        if temp==d1:
            break
        else:
            if record[i]==True:
                temp+=1
            else:
                continue
    #drawn this d1th number out and inacitive this index
    drawn[i]=np.copy(point_sam[temp])
    record[temp]=False

#sort the 100 drawn galaxies from smallest to largest radius
drawn_ind=np.arange(0,sele_num)
mergesort(drawn,0,sele_num-1,drawn_ind)

r=np.linspace(drawn[0],drawn[-1],100)
num_r=np.zeros_like(r)
for i in range(r.size):
    num_r[i]=np.sum(drawn<=r[i])
#plot the number of galaxies within r
plt.plot(r,num_r)
plt.xscale("log")
plt.xlim(1e-4,5)
plt.xlabel('r')
plt.ylabel('number of galaxies within r')
plt.savefig('./plots/Q2c.png',dpi=150)
plt.close()
```

question2.py

The plot is given by Figure 8.

2.4 d

We used the mergesort to calculate the median, 16th and 84th percentile for this radial bin(containing the largest number of galaxies). And we divided the 10000 points into 100 haloes and made a histogram of the number of galaxies in this radial bin in each halo, where the width of each bin is 1. Then we plot poisson distribution using the poisson function we wrote in Handin1 with the mean value equals to the mean number of galaxies in this radial bin.
The code of this subquestion is given by:

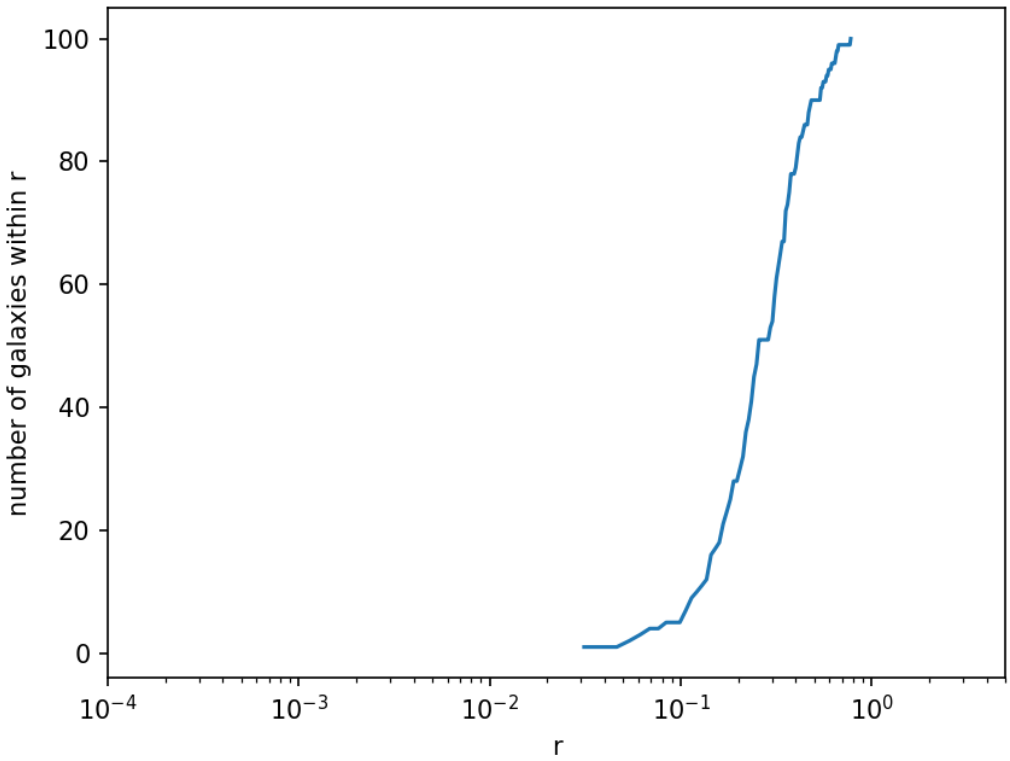

Figure 8: The number of galaxies within r for our 100 selected random samples. The number increases quickly from 0.1 to 1, because this range contains the largest number of galaxies in the original 10000 galaxies. This curve (is CDF in some way) shows that the distribution of the 100 galaxies is similar to the distribution of the original 10000 galaxies. Because we select them randomly following the 3 conditions, it agrees with our expectation.

```
#(d)
print('run (d)')
#find the bin from b containing the largest number of galaxies
histnew=np.copy(hist)
hist_ind=np.arange(histnew.size)
mergesort(histnew,0,histnew.size-1,hist_ind)
in1=hist_ind[-1]
#sorting
mask=(point_sam>=bin_edges[in1])*(point_sam<bin_edges[in1+1])
radial_bin=point_sam[mask]
radial_bin2=np.copy(radial_bin)
radial_size=radial_bin2.size
radial_ind=np.arange(0,radial_size)
mergesort(radial_bin2,0,radial_size-1,radial_ind)
#median of this bin
if radial_size%2==0:
    #radial_size is even
    radial_median=0.5*(radial_bin2[int(radial_size/2)]+radial_bin2[int(radial_size/2)-1])
else:
    #odd
    radial_median=radial_bin2[int(radial_size/2)]
#16th percentile
radial_16=radial_bin2[round(0.16*radial_size)]
#84th percentile
radial_84=radial_bin2[round(0.84*radial_size)]

print('median is:',radial_median)
print('16% is:',radial_16)
print('84% is:',radial_84)

filed=open('Q2d.txt','w')
filed.write('The median of this radial bin is:{:.10f}'.format(radial_median))
```

```
filed.write('\nThe 16th percentile of this radial bin is:{:.10f}'.format(radial_16))
filed.write('\nThe 84th percentile of this radial bin is:{:.10f}'.format(radial_84))
filed.close()
halo=np.zeros(100)
for i in range(100):
    halo[i]=np.sum(mask[100*i:100*(i+1)])

halo_mean=np.mean(halo)
_,_,_=plt.hist(halo,bins=100,label='data')

def poisson(lm,k):
    #lm:lambda(mean), k
    if k==0:
        #because of factorial, we need to calculate differently when k=0
        return np.exp(-lm)
    else:
        #to prevent overflow, calculate at log space
        arr=np.arange(1,k+1)
        p=k*np.log(lm)-lm-np.sum(np.log(arr))
        #return p to linear
        return np.exp(p)

poi_k=np.arange(0,100)
poi_dis=np.zeros_like(poi_k,dtype='float64')
for i in range(poi_k.size):
    poi_dis[i]=poisson(halo_mean,poi_k[i])

plt.plot(poi_k,100*poi_dis,label='poisson')
plt.xlabel('The number of galaxies')
plt.ylabel('frequency')
plt.legend()
plt.savefig('./plots/Q2d.png',dpi=150)
plt.close()
```

question2.py

The median, 16th and 84th percentile for this bin are given by:

```
The median of this radial bin is:0.2607563889
The 16th percentile of this radial bin is:0.2158485143
The 84th percentile of this radial bin is:0.3099758003
```

Q2d.txt

The histogram and poisson distribution are given by Figure 9.

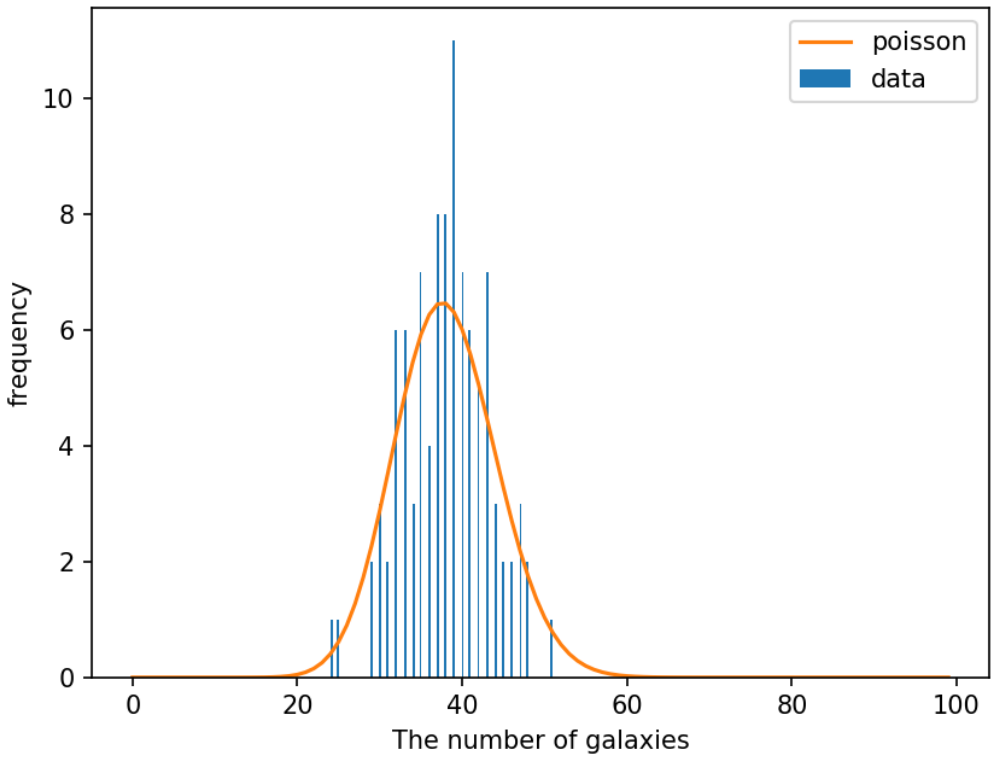

Figure 9: The histogram of the 100 values. The mean number of galaxies in this radial bin in 38.08 in our sample, which is taken as the λ for the poisson function. The orange line Poisson distribution is multiplied by 100 so that we can compare it with the histogram. Our data agree with the Poisson distribution within some uncertainty.