Hand-in Exercise 1 solution

Pengyu Liu (s2604531)

October 9, 2020

Abstract

In this document, the solutions for the three problems are given for the Hand-in Exercise 1 in the course Numerical recipes for astrophysics.

1 Cooling rates in cosmological simulations

In this section we look at question Cooling rates in cosmological simulations, we want to do the 3d interpolation to the cooling_rates over Hydrogen density, temperature and redshift. The 3D linear interpolator is given by the function linear_interp3d(cube,x,y,z,xitp,yitp,zitp) in the shared modules. For a given new point(znew,ynew,xnew), we use the bisection method to find the positions where we need to do interpolation for each axis. After finding the nearest values for znew(z1,z2), ynew(y1,y2) and xnew(x1,x2), we do the linear interpolation along each axis. For example, the xnew locates between x1 and x2, then we fix z1,y1 to do 1D linear interpolation along x axis. Then we fix z1,y2 and also do 1D interpolation along x axis. And we use the two interpolated new points to do 1D linear interpolation along y axis. Repeat the steps for z2, and we interpolate the final two points along the z axis.

The shared modules for the sub-question a and b are given by:

```
import h5py
import numpy as np
import os
import matplotlib.pyplot as plt
from tqdm import tqdm

def bisection(x,x_new):
    #find the nearest two points to x_new using bisection
    #return the indexes of these two points in the array x
    N=x.size
    #edge1 is the lower point and edge2 is the upper point
    edge1=0
    edge2=N-1
    while (edge2-edge1)>1:
        middle=int((edge2+edge1)/2)
        if x_new>x[middle]:
            #if x_new>x[middle], update the lower edge
            edge1=middle
        else:
            #if x_new<=x[middle], update the upper edge
            edge2=middle

    return edge1,edge2

def linear_interp3d(cube,x,y,z,xitp,yitp,zitp):
    #cube is the 3d array that to be interpolated. axis=0 is z-axis, axis=1 is y-axis and axis
    =2 is x-axis
    #x,y,z are the grids of the cube;
    #xitp,yitp,zitp are coordinates of new points

    #number of new points
    num=xitp.size
    #p is used to store the interpolated values
    p=np.zeros(num)
    for i in range(num):
        z1=zitp[i]
        y1=yitp[i]
        x1=xitp[i]
        #find the nearest points to the new point
        zl,zu=bisection(z,z1)
```

```
        yl,yu=bisection(y,y1)
        xl,xu=bisection(x,x1)
        #at zl plane
        #interpolate along x axis
        #the below linear interpolation already contains: if x1=x[xl],p1=cube[zl,yl,xl]; if x1=x[xu], p1=cube[zl,yl,xu]
        p1=(x1-x[xl])*(cube[zl,yl,xu]-cube[zl,yl,xl])/(x[xu]-x[xl])+cube[zl,yl,xl]
        p2=(x1-x[xl])*(cube[zl,yu,xu]-cube[zl,yu,xl])/(x[xu]-x[xl])+cube[zl,yu,xl]
        #interpolate along y axis
        p3=(y1-y[yl])*(p2-p1)/(y[yu]-y[yl])+p1
        #at zu plane
        #interpolate along x axis
        p4=(x1-x[xl])*(cube[zu,yl,xu]-cube[zu,yl,xl])/(x[xu]-x[xl])+cube[zu,yl,xl]
        p5=(x1-x[xl])*(cube[zu,yu,xu]-cube[zu,yu,xl])/(x[xu]-x[xl])+cube[zu,yu,xl]
        #interpolate along y axis
        p6=(y1-y[yl])*(p5-p4)/(y[yu]-y[yl])+p4

        #interpolate along z axis
        p[i]=(z1-z[zl])*(p6-p3)/(z[zu]-z[zl])+p3

    return p


#get the filename we need to read
z=[]
for f_name in os.listdir('CoolingTables'):
    if f_name.endswith('.hdf5'):
        if f_name[2:7]=='colli':
            continue
        elif f_name[2:7]=='photo':
            continue
        elif f_name[2:7] not in z:
            z.append(f_name[2:7])

z.sort()

#create arrays
Z=np.array(z,dtype='float')
cool_metalfree=np.zeros((Z.size,352,81))
cool_metal=np.zeros_like(cool_metalfree)
ne_nh=np.zeros_like(cool_metalfree)
ne_nh_sol=np.zeros_like(cool_metalfree)

#read in data
f=h5py.File('CoolingTables/z_{:}.hdf5'.format(z[0]),'r')
#Temperature bins
T=np.array(f['Total_Metals/Temperature_bins'])
#Hydrogen density bins
H=np.array(f['Total_Metals/Hydrogen_density_bins'])
#ne/nH in the solar system
ne_nh_sol[0]=np.array(f['Solar/Electron_density_over_n_h'])
#He to H mass fraction=0.258([2,])
#Because the electron density contributed from heavy elements is very small, I take the electron density from He and H as the ne/nH
ne_nh[0]=np.array(f['Metal_free/Electron_density_over_n_h'][2,])
cool_metalfree[0]=np.array(f['Metal_free/Net_Cooling'][2,])
#Because the coefficient for every metal element is the same, I use the net_cooling from Total_Metals
cool_metal[0]=np.array(f['Total_Metals/Net_cooling'])
```

cooling.py

1.1 a

We use the equation (2) in Hand in Exercise 1 to calculate the cooling rate. The cooling rate for H and He is obtained from the metal_free file. Because the coefficient for the second term on the right hand side of equation (2) is the same for every elements, we can use the cooling rate in the Total_metal file as the sum of cooling rate of all heavy elements. The metallicity for this question is 0.25. So we need to do 3D linear interpolation for cooling rate(H,He), cooling rate(total metal), ne/nH and (ne/nH)solar separately. Though (ne/nH)solar is the same for all redshifts, we still copy them into a 3D cube so that we can use the 3D linear interpolator. Because electron density contributed from heavy elements is very small, we simply use the (ne/nH) in the metal_free file.

The code specific to this question is given by:

```
metallicity=0.25
for i in range(1,Z.size):
    f=h5py.File('CoolingTables/z_{:}.hdf5'.format(z[i]),'r')
    ne_nh[i]=np.array(f['Metal_free/Electron_density_over_n_h'][2,])
    cool_metalfree[i]=np.array(f['Metal_free/Net_Cooling'][2,])
    cool_metal[i]=np.array(f['Total_Metals/Net_cooling'])
    ne_nh_sol[i]=np.array(f['Solar/Electron_density_over_n_h'])


#(a)
T_new=np.copy(T)
Z_new=np.zeros_like(T)
Z_new[:]=3
H_new=np.zeros_like(T)
H_den=np.array([1,1e-2,1e-4,1e-6])


for i in range(H_den.size):
    H_new[:]=H_den[i]
    #interpolate
    cool_metalfree1=linear_interp3d(cool_metalfree,H,T,Z,H_new,T_new,Z_new)
    cool_metal1=linear_interp3d(cool_metal,H,T,Z,H_new,T_new,Z_new)
    ne_nh1=linear_interp3d(ne_nh,H,T,Z,H_new,T_new,Z_new)
    ne_nh_sol1=linear_interp3d(ne_nh_sol,H,T,Z,H_new,T_new,Z_new)
    #calculate total cooling rate
    total_cool=cool_metalfree1+metallicity*cool_metal1*ne_nh1/ne_nh_sol1
    plt.loglog(T,total_cool,label='nH={:}'.format(H_den[i]))

plt.legend()
plt.xlabel('T(K)')
plt.ylabel('total cooling rate/n_H^2 (erg s^-1 cm^3)')
plt.title('z=3')
plt.savefig('./plots/cooling1a.png')
plt.close()
```

cooling.py

Setting metallicity=0.25 and at at a redshift of z = 3, our script produces the following results for densities of (1 cm^{-3}, 10^{-2} cm^{-3}, 10^{-4} cm^{-3}, 10^{-6} cm^{-3}), see Figure 1.

1.2 b

Now, we set the metallicity to be 0.5, density to 0.0001 cm^{-3} and calculate the cooling rate as a function of temperature for the allowed redshift (0-8.989) range. We need to make a movie to show the variations with redshift. We used 100 different redshifts and a framerate of 10 in this movie.

The code specific to this question is give by:

```
#(b)interpolate z from 0 to 8.989
metallicity=0.5
Z_b=np.linspace(0,Z[-1],100)
H_new[:]=1e-4


for i in tqdm(range(100)):
    Z_new[:]=Z_b[i]
    #interpolate
    cool_metalfree1=linear_interp3d(cool_metalfree,H,T,Z,H_new,T_new,Z_new)
    cool_metal1=linear_interp3d(cool_metal,H,T,Z,H_new,T_new,Z_new)
    ne_nh1=linear_interp3d(ne_nh,H,T,Z,H_new,T_new,Z_new)
    ne_nh_sol1=linear_interp3d(ne_nh_sol,H,T,Z,H_new,T_new,Z_new)
    #calculate total cooling rate
    total_cool=cool_metalfree1+metallicity*cool_metal1*ne_nh1/ne_nh_sol1
    plt.loglog(T,total_cool)
    plt.xlabel('T(K)')
    plt.ylabel('total cooling rate/n_H^2 (erg s^-1 cm^3)')
    plt.title('z={:1.4f}'.format(Z_b[i]))
    plt.ylim(1e-24,1e-19)
    plt.savefig('./plots/snap%04d.png'%i)
    plt.close()
```

cooling.py

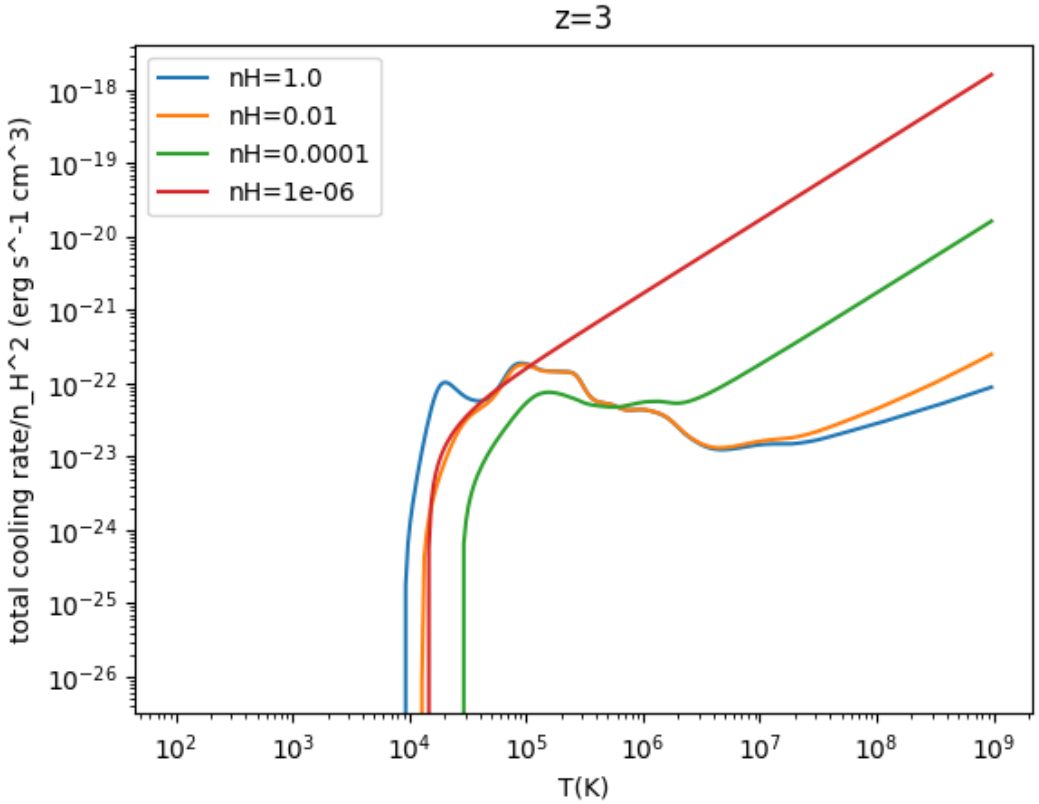

Figure 1: The result of our program shows the total cooling rate as a function of the temperature at z=3. It roughly decreases with the Hydrogen density and increases with temperature. The result is similar to Figure 2 in the referenced literature(Wiersma et al.2009).

For the movie see the main directory the file `cooling_ratemovie.mp4`. We can see that the cooling rate increases with redshift.

2 Redshift distribution of galaxies

The shared module is given by:

```
#code for question2
import numpy as np
```

solveLU2.py

2.1 a

We need to write an LU decomposition code to solve the linear equations: wss*f=wgs. wss is the matrix needed to do LU decomposition and f is solved by the LU matrix. The size of wss is (16,16) and both sizes of f and wgs are 16.
We used the improved Crout's algorithm on slide14 in lecture2 to calculate the LU matrix. We loop over from k to i to j. At first, we find the row($\geq$k) with the largest absolute value as the pivot condidate and swap the row k and imax. Record the index of imax because we need it when solve the equations. Then loop over i>k to calculate alpha(ik). In this loop, loop over j>k and calculate alpha(ij) and beta(ij). And we put L and U matrix in one matrix to save memory. Because alpha[ii]=1, we don't need to write them out.

After computing the LU matrix, we can use forward substitution and back substitution to solve f. Because we swapped rows in LU decomposition, we also need to swap wgs in forward substitution.

The code of this subquestion is given by:

```python
#2(a)
def LU_decom(M):
    #M is the Matrix to be decomposed and should have a size of N*N
    A=np.copy(M)
    n1,n2=A.shape
    if n1!=n2:
        print('Error: The size of the matrix should be N*N.')
        return
    #an array to store the permutation
    indexmax=np.zeros(n1,dtype=np.int)
    #Improved Crout's algorithm on slide14 in lecture 2
    for k in range(n1):
        # find the row with the largest pivot candidate from row>=k
        ind=np.argmax(abs(A[k:,k]))+k
        if ind!=k:
            #if ind!=k,ind must be large than k. row[ind] has the largest absolute value and
    swap row ind,k
            A[[ind,k],:]=A[[k,ind],:]
        #record the swap: if ind=k, no swap; if ind>k, ind is the swaped row for row k
        indexmax[k]=ind
        if A[k,k]==0:
            print('Error: The matrix is singular.')
            return
        for i in range(k+1,n1):
            #alpha(ik)
            A[i,k]=A[i,k]/A[k,k]
            #loop over columns j>k to compute alpha(ij) and beta(ij)
            A[i,(k+1):]-=A[i,k]*A[k,(k+1):]

    #return A(the LU matrix) and pivot index array
    return A, indexmax

def LU_solve(A,pivot,b):
    #solve x
    x=np.copy(b)
    n1=x.size
    #forward substitution
    for i in range(n1):
        #Because we swapped rows in LU decomposition, now also need to swap rows for x
        mid=x[pivot[i]]
        x[pivot[i]]=x[i]
        #alpha(ii)=1, so no need to divide
        if i==0:
            x[i]=mid
        else:
            x[i]=mid-np.sum(A[i,0:i]*x[0:i])

    #back substitution
    for i in range(n1-1,-1,-1):
    #In forward substitution, we swapped row. Don't need to swap in back substitution
        if i==(n1-1):
            x[i]=x[i]/A[i,i]
        else:
            x[i]=(x[i]-np.sum(A[i,(i+1):]*x[(i+1):]))/A[i,i]

    return x


#read in data
wgs=np.loadtxt('wgs.dat',dtype=np.float32)
wss=np.loadtxt("wss.dat",dtype=np.float32)
#LU decomposition
LU,piv=LU_decom(wss)
#solve
f=LU_solve(LU,piv,wgs)


np.savetxt('2aLU.txt',LU,fmt='%1.5f')
np.savetxt('2af.txt',f)
file1=open('2af.txt','a')
file1.write('\nThe sum of f is: ')
file1.write(str(sum(f)))
file1.write('\nThe error is: ')
file1.write(str(abs(1-sum(f))))
file1.close()
```

solveLU2.py

The LU matrix is given by:

```
0.38730 0.35512 0.24105 0.12011 0.09577 0.05575 0.04318 0.04865 0.04628 0.04175 0.01544 0.03499 0.00004 0.03527
    0.01875 0.01420
0.48468 0.64230 0.05264 0.11845 0.05795 -0.01127 0.02097 0.01200 0.02427 0.01136 0.03691 0.01186 0.00893 -0.00717
    0.02573 0.01420
0.34631 0.47914 0.45729 -0.06931 0.14413 0.04300 0.05224 0.00903 0.04434 -0.01293 0.01302 0.01681 0.02572 0.01892
    -0.00786 0.00789
0.29687 0.03688 0.86327 3.00824 0.14909 0.08559 -0.00424 0.00998 -0.02379 0.04312 -0.01242 -0.00905 0.01959 0.02054
    0.02228 0.00937
0.27315 0.03547 -0.02050 0.01041 3.61547 0.21858 0.13999 -0.00797 0.08632 -0.01025 0.06509 0.02821 0.03432 0.03501
    0.03491 -0.00230
0.06264 0.02165 0.28763 0.05982 0.02579 0.92191 0.46947 0.23898 0.07215 0.11446 0.07023 0.01611 0.00997 0.01270
    0.02929 0.04215
0.13199 -0.06135 0.14979 0.04944 0.01549 0.04301 5.43577 0.09383 0.21317 0.14065 0.11458 0.05146 0.05415 0.04697
    0.04189 0.02309
0.13105 -0.06955 -0.05715 0.01569 0.00181 0.20997 0.00004 5.45075 0.35154 0.03294 0.08739 0.09404 0.06962 0.01980
    0.04656 0.01758
0.07501 0.06089 0.12022 0.01315 -0.00763 0.16205 0.01496 0.00252 4.54485 0.30815 0.17478 0.05789 0.01880 0.03235
    0.01482 0.04238
0.03086 0.00936 0.07206 0.00705 0.00526 0.00501 0.02505 0.03606 0.09100 0.85001 0.04817 0.01847 0.07859 0.11493
    0.04748 0.04253
0.01848 0.03526 0.00269 0.01675 0.01850 0.01903 0.00853 0.02217 0.01640 0.51601 10.54105 0.18096 -0.03586 0.07117
    0.03206 0.00908
0.11546 0.00373 0.00989 0.00437 0.01113 0.03540 -0.00266 0.00607 -0.00043 0.02096 0.02787 11.15995 0.39806 0.12652
    0.09428 0.06148
0.01513 0.00448 0.01951 0.01212 0.00302 0.03328 0.00982 0.01611 0.01928 0.14340 0.01979 0.01877 1.63296 0.21451
    0.18161 0.00147
0.05962 0.01924 -0.01971 0.00723 0.00466 0.00218 0.01035 0.00890 0.00775 0.10053 0.01075 0.01673 0.18572 7.57390
    0.43910 0.19498
0.02754 0.01076 0.00710 0.01152 0.00588 0.04470 -0.00359 0.00446 -0.00108 0.06448 0.00682 0.01368 0.08621 0.03908
    3.69338 0.27488
0.02841 0.01361 0.01126 0.01238 0.00315 0.04752 -0.00320 -0.00182 0.00447 0.05138 0.00161 0.00566 0.07079 -0.00065
    0.04114 3.99064
```

2aLU.txt

The upper triangle represents the U matrix(beta) and the lower triangle represents the L matrix(alpha). The solution of f is given by:

```
3.459609113633632660e-03
3.231730684638023376e-02
8.087178319692611694e-02
1.002292484045028687e-01
1.333132982254028320e-01
1.232234090566635132e-01
1.315599232912063599e-01
1.045399680733680725e-01
8.667092025279998779e-02
5.877323448657989502e-02
4.246465489268302917e-02
3.845600783824920654e-02
1.786364242434501648e-02
2.487732656300067902e-02
1.350274961441755295e-02
7.876945659518241882e-03

The sum of f is: 1.0000000279396772
The error is: 2.7939677238464355e-08
```

2af.txt

We can see that the sum of f is very close to 1. The error is very close to the single precision.

2.2 b

We conducted a single iteration to improve f. Because we already solved the LU matrix, we can use the same LU matrix and pivot permutation from 2(a) to solve this question. The method is described on slide 17 in lecture2. We solve delta_f by LU*delta_f=LU*f-wgs and the improved f_new=f-delta_f.

The code of this subquestion is given by:

```
#2(b) single iterative
wgs_new=np.dot(wss,f)
#use the same LU and piv, no need to do LU deecomposition again
delt_f=LU_solve(LU,piv,wgs_new-wgs)
f_new=f-delt_f

np.savetxt('2bf.txt',f_new)
file2=open('2bf.txt','a')
file2.write('\nThe sum of the imroved f is: ')
file2.write(str(sum(f_new)))
file2.write('\nThe error is: ')
```

```
file2.write(str(abs(1-sum(f_new))))
file2.close()
```

solveLU2.py

The solution of the improved f is given by:

```
3.459599567577242851e-03
3.231729567050933838e-02
8.087176829576492310e-02
1.002292558550834656e-01
1.333133280277252197e-01
1.232233867049217224e-01
1.315599083900451660e-01
1.045399606227874756e-01
8.667092025279998779e-02
5.877323821187019348e-02
4.246464744210243225e-02
3.845601528882980347e-02
1.786363869905471802e-02
2.487732656300067902e-02
1.350274775177240372e-02
7.876944728195667267e-03

The sum of the imroved f is: 0.9999999820720404
The error is: 1.792795956134796e-08
```

2bf.txt

We can see that the error is very close to that in question a. It seems that f is only slighlty improved. I think is the reason is that the error almost reached the single precision after the first calculation. So after one iteration, f is improved little under the single precision data type. It shows that this LU decomposition works quite well.

3 Satellite galaxies around a massive central

In this question, we need to write a numerical integrator to solve for A and interpolate the function. In the end, we need to write a function that can return the Poisson probability distribution for a given positive mean λ and integer k.

The shared code is given by:

```
#Question3

import numpy as np
import matplotlib.pyplot as plt
```

3integral.py

3.1 a

This a 3D integral, but we can reduce it to a 1D integral. In this case, dV=$4\pi x^2 dx$. The integral becomes $4\pi b^{3-a} \int_0^5 (x^{a-1} exp(-(\frac{x}{b})^c)\, dx$. Because the integral does not contain any singularities from 0 to 5 at a given a=2.2, b=0.5 and c=3.1, we wrote an extended Simpson integrator to solve the integral. The we can calculate A=$\frac{1}{integral}$. The algorithm of extended Simpson's rule is given on slide7 in lecture3.

The script of this subquestion is given by:

```
#(a)
#function that need to be integrated
def f_itg(x,a=2.2,b=0.5,c=3.1):
    return np.power(x,a-1)*np.exp(-np.power(x/b,c))

def simpson(f,a,b,N):
    #simpson integration
    #f: integrand; a: lower limit; b: upper limit; N: number of intervals
    h=(b-a)/N
    #N+1 points
    x=np.linspace(a,b,N+1)
    y=f(x)

    #if N is even
```

```
    if N%2 ==0:
        slice1=np.arange(0,N-1,2)
        slice2=np.arange(1,N,2)
        slice3=np.arange(2,N+1,2)
        result=(h/3)*(np.sum(y[slice1])+np.sum(y[slice3])+4*np.sum(y[slice2]))

    else:
        #if N is odd, the last interval uses the trapzoid
        slice1=np.arange(0,N-2,2)
        slice2=np.arange(1,N-1,2)
        slice3=np.arange(2,N,2)
        result=(h/3)*(np.sum(y[slice1])+np.sum(y[slice3])+4*np.sum(y[slice2]))+h*0.5*(y[N-1]+y
    [N])

    return result


b=0.5
a=2.2
#1000 intervals can already gives a very good resutls
integ=np.power(b,3-a)*4*np.pi*simpson(f_itg,0,5,1000)
A=1/integ
fA=open('A.txt','w')
fA.write(str(A))
fA.close()
```

3integral.py

We set the number of intervals between 0 and 5 to be 1000. A is given by:

```
1.5382202236720077
```

A.txt

3.2 b

Because we only have 5 points that span in a large range, I chose to do interpolation in loglog space. I used the linear interpolator because there are only 5 points and they show piecewise patterns: changes relatively slowly when x<1 while changes dramatically when x>1.

The code is given by:

```
#(b)
def linear_interp(x1,y1,x_new1,log):
    #boolean:log. If log=True, then do interpolation in log space
    #x1 and y1 are known points, x_new1 are x coordinates of new points
    if log==True:
        x=np.log(x1)
        y=np.log(y1)
        x_new=np.log(x_new1)
    else:
        #otherwise do interpolation in linear space
        x=np.copy(x1)
        y=np.copy(y1)
        x_new=np.copy(x_new1)

    num=x_new.size
    y_new=np.zeros_like(x_new)
    N=x.size
    for i in range(num):
        #bisection: find the nearest two points
        edge1=0
        edge2=N-1
        while (edge2-edge1)>1:
            middle=int((edge2+edge1)/2)
            if x_new[i]>x[middle]:
                edge1=middle
            else:
                edge2=middle
        #calculate slope and interpolate
        y_new[i]=(x_new[i]-x[edge1])*(y[edge2]-y[edge1])/(x[edge2]-x[edge1])+y[edge1]

    if log==True:
        return np.exp(y_new)
    else:
        return y_new
```

```
35
36 #function n(x)
37 def f(x,a=2.2,b=0.5,c=3.1):
38     return np.power(x/b,a-3)*np.exp(-np.power(x/b,c))
39
40 Nsat=100
41 x1=np.array([1e-4,1e-2,1e-1,1,5])
42 y1=A*Nsat*f(x1)
43 #Because f(5) is extremly close to 0 and underflows in the 64bit float type, I truncate f(5)
       to be 1e-30.
44 #Though it causes an error, I know the error instead of unknown underflow error.
45 y1[-1]=1e-30
46 plt.loglog(x1,y1,'o')
47 #The five data points change dramatically in linear space. When we plot them in log space,
48 #they show piecewise patterns. Point 1-4 decreases slowly, but the last point is very small
49 #compared to the former 4 points.
50 #Because there are only 5 points and x,y span in a large range, I chose to do interpolation in
       loglog space.
51 #It is a piecewise function based on just these points, lagrange polynomial like Neville's
       algorithm can
52 #produce large wrinkles between these points. Cubic spline can interpolate piecewise functions
       when
53 #the functions are smooth (1st and 2nd derivatives are continuous.)
54 #Cubic spline takes all points into consideration and
55 #can also cause large wrinkles between points if points change dramatically, which is our case
       .
56 #So cubic spline is not a good choice. Akima spline should be a good choice in this case
       because it
57 #can give natural and smooth results based on a small number of points. However, I have little
       time
58 #to implement it. Instead, I chose to do the linear interpolation in log space, because it can
       also produce
59 #results that do not deviate far from the known points, which is a convenient way to do
       interpolation when
60 #there are only a few points and changes dramatically at some range.
61
62 #produce points for interpolation
63 x=np.linspace(0.0001,0.01,10)
64 x=np.append(x,np.linspace(0.02,0.1,10))
65 x=np.append(x,np.linspace(0.2,1.3,10))
66 x=np.append(x,5)
67
68 #linear interpolate in loglog space
69 yitp=linear_interp(x1,y1,x,log=True)
70
71 plt.loglog(x,yitp,'-.',label='linear interp')
72 plt.xlabel('x')
73 plt.ylabel('n(x)')
74 plt.legend()
75 plt.savefig('./plots/3binterp.png')
76 plt.close()
```

3integral.py

Figure 2 shows the five points and the results of linear interpolation.

3.3 c

The Poisson probability distribution $P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!}$ is easy to underflow because its denominator k! can be very large and overflows even in float64 type. The numerator λ^k can also be very large and exp(-λ) can be very small. Because we calculate the three parts separately in the computer, any part overflows or underflows will cause a huge relative error to P, though P itself is not easy to underflow. So I chose to calculate P at log space and return it back to linear space.

The code is given by:

```
1 #c
2 def poisson(lm,k):
3     #lm:lambda(mean), k
4     if k==0:
5         #because of factorial, we need to calculate differently when k=0
6         return np.exp(-lm)
7     else:
8         #to prevent overflow, calculate at log space
9         arr=np.arange(1,k+1)
10        p=k*np.log(lm)-lm-np.sum(np.log(arr))
```

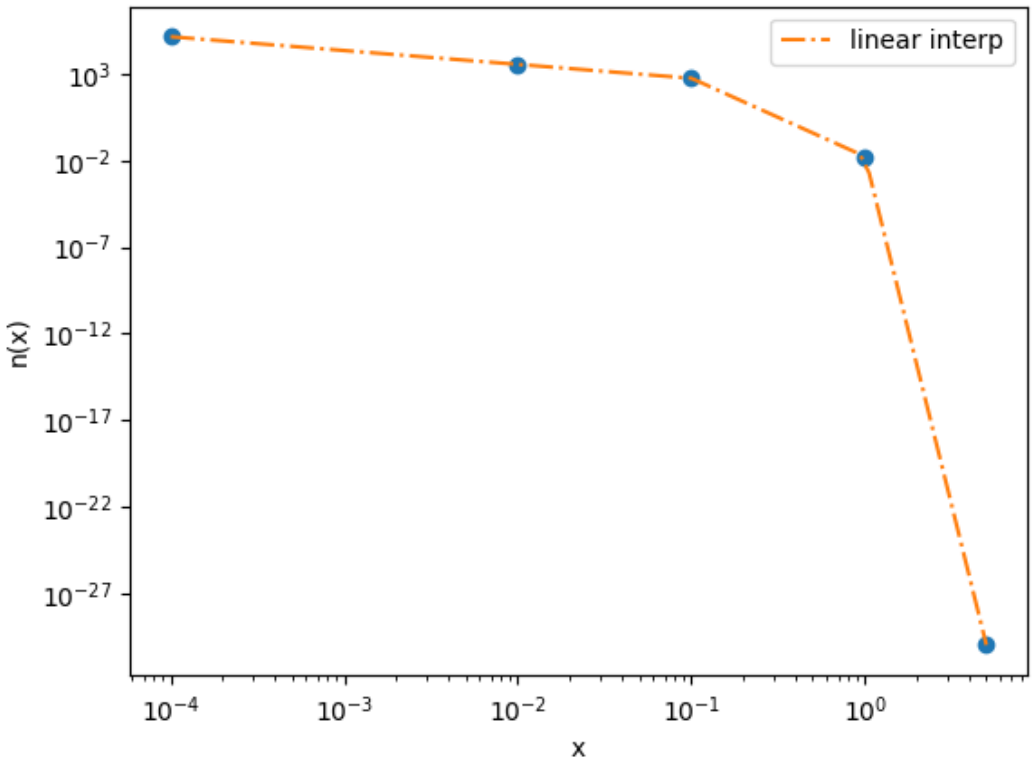

Figure 2: The loglog plot of interpolation. I chose to do linear interpolation at loglog space and truncate at x=5. Although the curve of linear interpolation is not smooth, it does not derivate far from the five points and does not cause large wrinkles between points. It also follows the overall trend of these points.

```
        #return p to linear
        return np.exp(p)

#write out results to 3c.txt
file=open('3c.txt','w')
file.write('\nP(1,0)='+str(poisson(1,0)))
file.write('\nP(5,10)='+str(poisson(5,10)))
file.write('\nP(3,21)='+str(poisson(3,21)))
file.write('\nP(2.6,40)='+str(poisson(2.6,40)))
file.write('\nP(101,200)='+str(poisson(101,200)))
file.close()
```

3integral.py

The results of given (λ,k) are given by:

```

P(1,0)=0.36787944117144233
P(5,10)=0.018132788707821885
P(3,21)=1.01933982411102e-11
P(2.6,40)=3.6151239949376635e-33
P(101,200)=1.2695313920467071e-18
```

3c.txt

We can see that this log function can return $P_\lambda(k)$ correctly in a large range by using numpy float64 type, which is much better than calculating P straight from its definition.