NUR hand-in 2

Meng Yao; s2308266

May 28, 2019

Abstract

The source code and the outputs of Numerical Recipes for Astrophysics Hand-in exercise 2 are shown in this report.

1 Question 1 : Normally distributed pseudo-random numbers

The shared modules

```
'''
q1(a) main to show
'''
'''
Question 1 : Normally distributed pseudo-random numbers
1(a) random number generator
Combine at least MWC and 64-bit shift
'''
sys.stdout = open('outputs1.txt', 'w')
print('Question 1 : Normally distributed pseudo-random numbers')
print('\n 1(a) Random number generator' )
def generator(n,seed):
    '''
    combined number generator:
    XOR-shift ^ MWC, also called Ranq2. In the text book, the
    parameters are given as A3(right)^B1.
    period = period of xorshift * period of MWC. Accoring to the
    text book = 8.5*10^37
    the parameters will be given later
    n = the amount of numbers
    seed = initial seed
    '''
    #parameters of each generator
    ##XOR-shift 64-bit
    XOR_a1=17
    XOR_a2=31
    XOR_a3=8
    bit64=2**64-1
    bit32 = 2**32-1
    ##MWC
    MWC_a=4294957665
    #initial seed
    x=seed
    m=seed
    number = np.zeros(n)
    for i in range(n):
        #XORshift
```

```
        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)
        #MWC
        m = (MWC_a*(m & bit32)+(m >>32))  # use all 64 bits of updated state in bit mix
        #combine them
        number[i] = (x ^ 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
```

1.1 1.a Pseudo-random numbers

```
'''
q1(a) to show
'''
#set the seed
seed = 123456789
print('seed = ', seed)
#first 1000 numbers and plot
n1k_uni = generator(1000,seed)
fig1 = plt.figure(1)
ax1_1 = fig1.add_subplot(2,1,1)
ax1_1.scatter(n1k_uni[0:-2], n1k_uni[1:-1])
ax1_1.set_xlabel("$X_i$")
ax1_1.set_ylabel("$X_{i+1}$")
ax1_1.set_title('Sequential 1000 numbers')
ax1_2 = fig1.add_subplot(2,1,2)
ax1_2.plot(n1k_uni,'.')
ax1_2.set_title('1000 numbers vs indices')
ax1_2.set_xlabel('index')
#fig1.savefig("1a1k.png")
fig1.tight_layout()
# 1 million numbers and plot
fig4 = plt.figure(4)
n1m_uni=generator(10**6,seed)
ax4_1=fig4.add_subplot(1,1,1)
hist_n1m_uni = ax4_1.hist(n1m_uni, bins=np.linspace(0.0, 1.0, 21), edgecolor='black') #plot the histogram and save the elements
ax4_1.set_xlabel('bins')
ax4_1.set_ylabel('quantity of numbers')
ax4_1.set_title('Histogram of 1 million numbers')
fig4.tight_layout()
#fig4.save('1a1m.png')
fig4.show()
print('figure of random number generator please see fig.1')
print('roughly test the quality of RNG by show the largest and smallest number among bins:')
print('max = ',max(hist_n1m_uni[0]),'; min = ',min(hist_n1m_uni[0]))
```

seed = 123456789
figure of random number generator please see fig.1
roughly test the quality of RNG by show the largest and smallest number among bins:

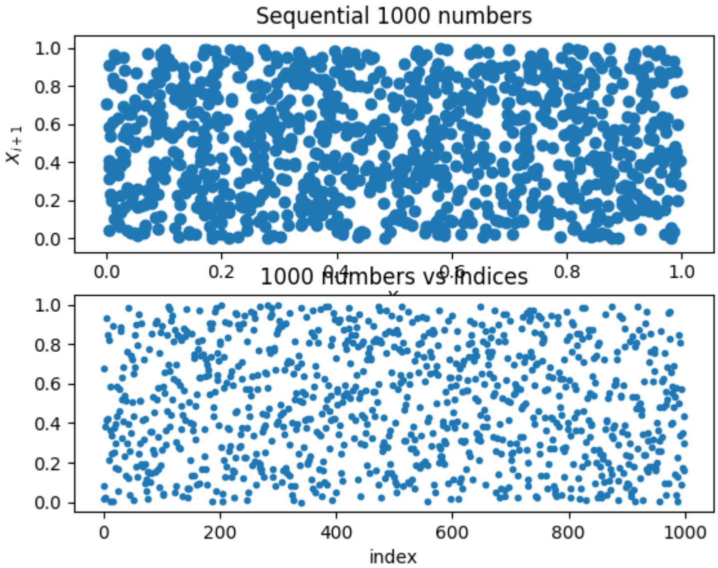

Figure 1: The results of the first 1000 numbers of RNG

max = 50341.0 ; min = 49714.0

1.2 1.b Normally distributed random number

```
'''
1(b) Normally distributed random number
generate normally distributed numbers whose mean=3 sigma=2.4.
Then, compare them with Gaussian probability density function
'''
print('\n 1(b) Normally distributed random number')
def Box_Muller(random_uni):
    '''
    in put a uniformly distributed random number sequence
    out put a nomarlly distributed one
    note that the sequence is still in [0,1)
    '''
    #split into 2 sequences
    u1 = random_uni[0::2]    #sequence 1 is even-th elements
    u2 = random_uni[1::2]    # odd-th elements
    coe = np.sqrt(-2*np.log(u1))
    s1 = coe*np.cos(2*np.pi*u2)  # already normal
    s2 = coe*np.sin(2*np.pi*u2)
    random_normal = np.concatenate([s1,s2])
    return random_normal

def Gaussian_PDF(x,mu,sigma):
    '''
    define Gaussian fuction
    '''
    return 1/(np.sqrt(2*np.pi*sigma**2)) * np.exp(-0.5*(x-mu)**2/
    sigma**2)

```

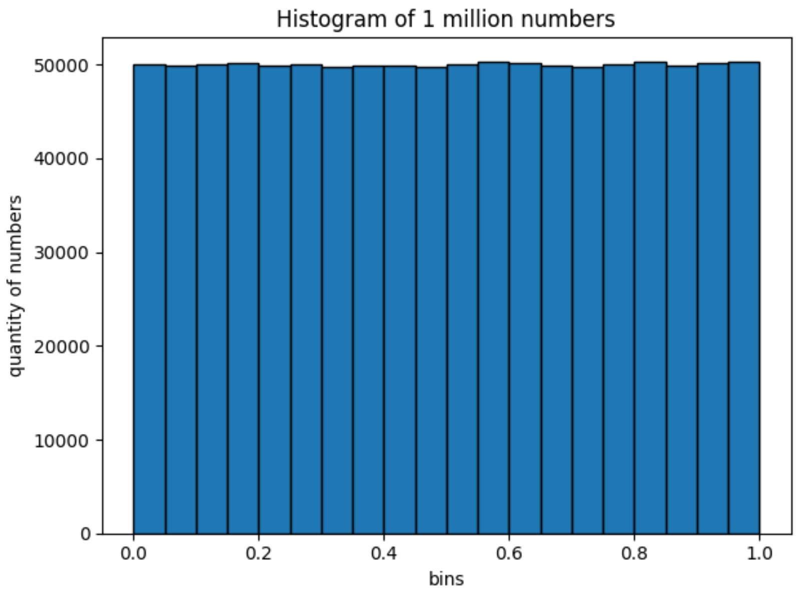

Figure 2: Histogram of 1 million random numbers in 20 bins with a width of 0.05

```
n1k_normal = Box_Muller(n1k_uni)                    #  normally distributed whose mu = 0; sigma =1
n1k_normal_1b = 2.4 * n1k_normal                    # target sigma is 2.4
n1k_normal_1b += 3.                                 # target mean is 3

#plot the normal random number histogram and corresponding Gaussian line
fig2 = plt.figure(2)
#hist
ax2_1 = fig2.add_subplot(1,1,1)
hist_n1k_normal_1b = ax2_1.hist(n1k_normal_1b, bins=np.linspace(3-2.4*5, 3+2.4*5, 21),
                                density='true', label='hist')
ax2_1.set_xlabel('value of the numbers')
ax2_1.set_ylabel('probability')
#Gaussian line
x_GPDF = np.linspace(3-2.4*5, 3+2.4*5,101)
y_GPDF = Gaussian_PDF(x_GPDF,3,2.4)
ax2_1.plot(x_GPDF, y_GPDF,label='Gausian PDF')
#indicated lines
for i in range(1,6):
    ax2_1.axvline(x=3+2.4*i, color='k', linestyle='--')
    ax2_1.axvline(x=3-2.4*i, color='k', linestyle='--')
#show the figure
ax2_1.legend(loc='best')
fig2.suptitle('normally distributed random number test')
#fig2.savefig('1b.png')
fig2.show()
```

1.3 1.c KS-test

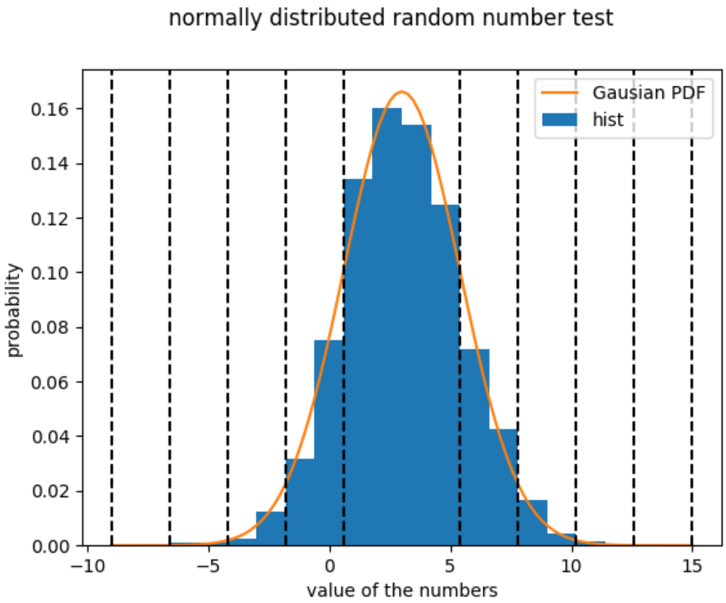

Figure 3: Histogram of 1000 normally distributed random number compared with Guassian distribution

```
'''
1(c) KS-test
KS-test is based on Cumulative distribution function. In our case,
    we use Gaussian CDF.
Gaussian CDF can be given by a error function ' erf(z)' which is a
    special funtion.
m = 0; sigma = 1 gives a standard normal distribution, where
Gaussian_CDF(x) = 0.5*(1+erf(x/sqrt(2))),  erf(z) = 2/sqrt(pi) *
    integral_0^z(e^-t2 dt)
'''

print('\n 1(c) KS-test')
# Now do the integral, using trapeziodal rules
def integrator(function, lower ,upper, n_intervals):
    '''
    trapeziodal rule
    function =
    lower = lower limit of the integral
    upper = upper limit
    interbvals = n of n_intervals
    '''
    h = (upper - lower) / n_intervals
    S = 0.5*(function(lower) + function(upper))
    for i in range(1,n_intervals):
        S += function(lower + i*h)
    integral = h * S
    return integral
#define the error function in order to calculate Gaussian CDF
def erf(z):
    erf_integral = lambda t: np.exp(-t**2)
    erf = 2./np.sqrt(np.pi) * integrator(erf_integral,0,z,10**3)
    return erf
```

```
# Gaussian cumulative distribution funciton
def Gaussian_CDF(x):
    CDF = 0.5*(1 + erf(x/np.sqrt(2.)))
    return CDF
# KS-test needs to sort the array
def quick_sort(array,i,j):
    '''
    i and j are the two elements we want to start with
    '''
    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
#KS-test function


def KS_test(array,CDF):
    '''
    array is the data that we want to test.
    index is the index of this array where we want to get the percentage.
    WARN : index should already be integers
    '''
    #calculate CDF for data
    array = sorted(array)
    N = len(array)
    pECDF = np.arange(0,1,1/N) + 1/N
    pCDF = np.zeros(N)
    for i in range(N):
        pCDF[i] = CDF(array[i])
    D = [abs(x) for x in (pECDF-pCDF)]
    return D


# generate 100,000 normallu distributed numbers
n100k_normal = Box_Muller(n1m_uni[:10**5])
#calculate D for every single point, 10**5 in total
#D100k = KS_test (n100k_normal, Gaussian_CDF)     # this step is a bit slow and cause my laptop heating......
#np.save('D100k',D100k)
D100k = np.load('D100k.npy')
index_ks = [int(x) for x in np.logspace(1,5,num=41,base=10)]   # 10 to 10**5
Dmax = np.zeros(len(index_ks))
for i in range(len(index_ks)):
    Dmax[i] = max(D100k[:index_ks[i]])
# calculate P value
def KS_CDF(D,N):
    '''
    use D value to calculate P_ks(z)
```

```

    '''
    z = (np.sqrt(N)+0.12+0.11/np.sqrt(N))*D
    if z < 1.18:
        exp = np.exp(-np.pi**2/(8*z**2))
        P_ks = np.sqrt(2*np.pi)/z * (exp + exp**9 + exp**25)

    else:
        exp = np.exp(-2*z**2)
        P_ks = 1-2*(exp-exp**4+exp**9)

    return P_ks
P_ks_z = np.zeros(len(Dmax))
for i in range(len(Dmax)):
    P_ks_z[i] = KS_CDF(Dmax[i],index_ks[i])
D_sci = np.zeros(len(index_ks))
P_val_sci = np.zeros(len(index_ks))
for i in range(len(index_ks)):
    D_sci[i], P_val_sci[i] = stats.kstest(n100k_normal[:index_ks[i]],'norm')
    #plot consistentcy
fig3 = plt.figure(3)
#plot Dmax vs the amount of numbers I used to test
ax3_1 = fig3.add_subplot(2,2,1)
ax3_1.scatter(index_ks, Dmax)
ax3_1.set_xscale('log')
ax3_1.set_ylim(-10**-4,0.004)
ax3_1.set_ylabel('maxium D')
ax3_1.set_title('my Dmax')
ax3_2 = fig3.add_subplot(2,2,2)
ax3_2.scatter(index_ks, D_sci )
ax3_2.set_xscale('log')
ax3_2.set_title('Dmax from scipy')
# plot P value
ax3_3 = fig3.add_subplot(2,2,3)
ax3_3.scatter(index_ks, P_ks_z)
ax3_3.set_ylabel('P value')
ax3_3.set_title('my P of z')
ax3_3.set_xscale('log')
ax3_4 = fig3.add_subplot(2,2,4)
ax3_4.scatter(index_ks, P_val_sci)
ax3_4.set_title('P value from scipy')
ax3_4.set_xscale('log')

fig3.tight_layout()
fig3.suptitle('KS-test from 10 to 10000 numbers',y = 1)
#fig3.savefig('1c.png')
fig3.show()

'''
I thought D is the maxium value from all the points that I put into the test.
In this case , I calculated 10^5 distances between ECDF and Gaussion CDF.
Then I select the maximum  from first 10 distance, 10^1.1 distances, 10^1.2 distances......10^5 distances.
My maximum D goes up, besides, values of D also cause a unnormal result to P value.
'''
```

1.4 1.d Kuiper's test

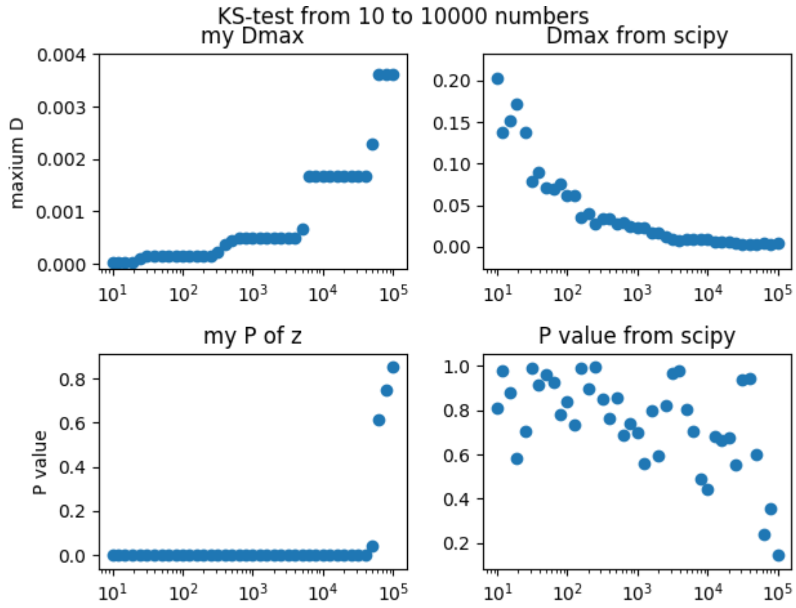

Figure 4: The upper panels show the maxium distance of KS-test for my test and scipy test; the lower panels show the P value

```
'''
1 (d) Kuiper test:
I wll present the statistic of Kuiper test
'''
print('\n 1(d) Kuiper test')

def Kuiper_test(array, CDF):
    '''
    array : array we want to test
    CDF : targert distribution

    '''

    N = len(array)
    array = sorted(array)
    # empirical cdf
    pECDF = np.arange(0,1,1/N) + 1/N
    pCDF = np.zeros(N)
    for i in range(N):
        pCDF[i] = CDF(array[i])
    # Maximum distance when p_data > p CDF
    D_plus= max(pECDF-pCDF)
    # p_data < p CDF
    D_minus = max(CDF-ECDF)
    V = D_plus + D_minus   # Kuiper statistic

    return V
```

My algorithm takes too long during my testing, I will just show it without resluts.

1.5 1.e Compare random numbers with exampels

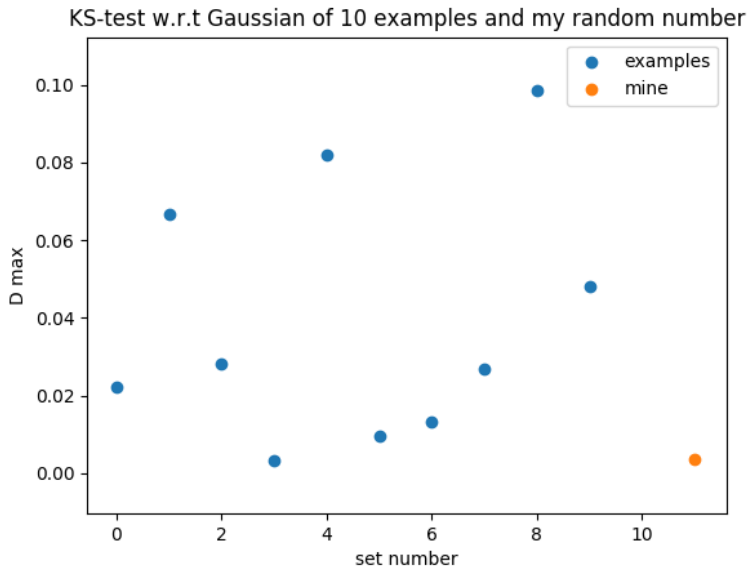

Figure 5: Comparision of my random numbers and examples based on D values of KS-test

```
num_examples=np.genfromtxt('randomnumbers.txt')
D_examples = np.zeros(10)
for i in range(10):
    D_examples[i],_ = stats.kstest(num_examples[:,i],'norm')
plt.figure()
plt.scatter(range(10),D_examples, label='examples')
plt.scatter(11,D_sci[-1],label = 'mine')
plt.legend(loc='best')
plt.xlabel('set number')
plt.ylabel('D max')
plt.title('KS-test w.r.t Gaussian of 10 examples and my random
    number')
#plt.savefig('1d.png')
plt.show()
print('from the figure, we can see that my random number has a
    lower D value of KS-test compared to the examples.')
```

According to the figure, my set of random numbers has a lower D value of KS-test compared to the examples. This means it shows more consistency with corresponding Gaussian distribution.

2 Question 2 : Gaussian random field

I am sorry I don't understand the symmetric thing very well. I asked TA about this but still not very clear. I know that if I take a 2-D matrix of real numbers and do Fourier transform, I will find conjugate symmetry in Fourier space. However, when I did this, I generated k**2 = kx**2 + ky**2 for right half of Fourier space and take P(k) = k**power as an amplitude. I did this separetly in two (512,512) array. Then I have four (512,512) Gaussian distributed random numbers arrays which were generated and saved in question. Those Gaussian arrays represent real part and imaginary part for two amplitude arrays. I took conjugate symmetric matrix, with respect to the center, of region 1 and region 2. Finally I combined those 4 arrays to get a (1024,1024) matrix which represented the field in Fourier space.

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

# In[185]:


import numpy as np
import matplotlib.pyplot as plt
from scipy import fftpack
plt.ioff()

print('\n Question 2 : Gaussian random field ')
def gaussian_random_field(power, size):
    '''
    power : power of the spectrum
    size : image size in per axis
    '''
    N=int(size/2)
    #because of conjuate symmetry, I only generate N = size/2 matrix
    Grn = np.load('normal_rn_q2.npy').reshape((4,N,N))
    Fourier_p1_real = Grn[0]
    Fourier_p1_imag = Grn[1]
    Fourier_p2_real = Grn[2]
    Fourier_p2_imag = Grn[3]
    Fourier_p1 = Fourier_p1_real + 1j*Fourier_p1_imag
    Fourier_p2 = Fourier_p2_real + 1j*Fourier_p2_imag
    #because of conjugate symmetry, I only generate N = size/2 matrix for sub-plane 1 and sub-plane 2
    #and sub-plane 3 is the conjugate symmetric matrix of 1. 4 is of 2.
    kx = np.linspace(np.pi/N,np.pi,N)
    ky = kx
    k = np.zeros((N,N))
    for i in range(N):
        for j in range(N):
            k[i,j] = (kx[i]**2+ky[j]**2)**0.5
    Pk = k ** power    # I get Pk now.
    # Gaussian random number will be used to scale the Pk
    Fourier_p1 *= Pk
    Fourier_p2 *= Pk
    #Fourier_p2 = Fourier_p1.conjugate()
    Fourier_p2 = np.flip(Fourier_p2, axis = 0)
    Fourier_p12 = np.concatenate((Fourier_p2,Fourier_p1))
    # take conjuagate
    Fourier_p3 = Fourier_p1.conjugate()
    Fourier_p3 = np.flip(Fourier_p3, axis = 0)
    Fourier_p3 = np.flip(Fourier_p3, axis = 1)
    Fourier_p4 = Fourier_p2.conjugate()
    Fourier_p4 = np.flip(Fourier_p4, axis = 0)
    Fourier_p4 = np.flip(Fourier_p4, axis = 1)
    Fourier_p34 = np.concatenate((Fourier_p4,Fourier_p3))
    Fourier_p = np.concatenate((Fourier_p34,Fourier_p12),axis = 1)
    GRF = fftpack.ifft2(Fourier_p)
    return GRF, Fourier_p

for n in [-1.0, -2.0, -3.0]:
```

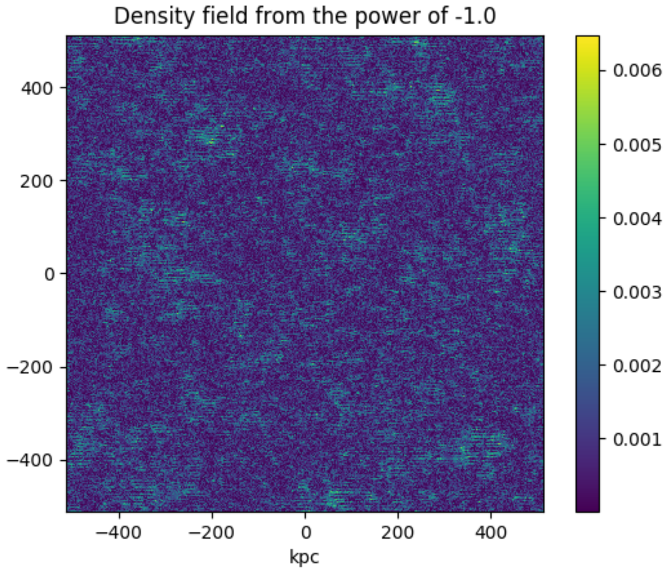

Figure 6: Density field 1

```
        field = gaussian_random_field(n, size=1024)[0]
        plt.figure()
        plt.imshow(np.absolute(field),extent=[-512,512,-512,512])
        plt.title('Density field from the power of {0}'.format(n))
        plt.xlabel('kpc')
        plt.xlabel('kpc')
        plt.colorbar()
        plt.savefig('q2_{0}.png'.format(int(-n)))
       #plt.show()

#print('\n show a row of Fourier plane = ',gaussian_random_field(n,
    size=1024)[1][0])
```

The mistakes are caused by my misunderstanding on symmetry.

3 Question 3 : Linear structure growth

Basically, this question is to solve a ODE, we will use 4th order Runge-Kutta method. Since $\Omega_m = 1$, the differential equation becomes : D"+4/(3t)*D' = 2/(3t**2)*D.

In terms of Runge-Kutta, for every iteration, I use a pair of D and D' as an input to calculate D". Then I will have all information of this present state in order to calculate next state. The pair of D and D' will be updated by R-K method and used in next iteration.

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

# In[1]:


```

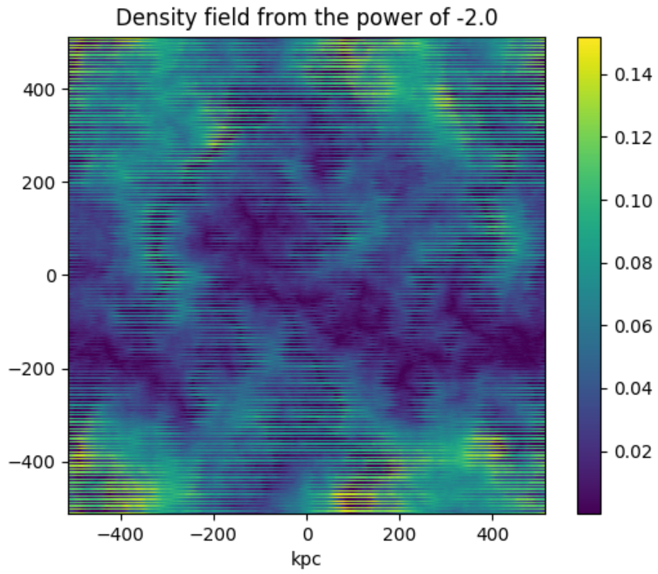

Figure 7: Density field 2

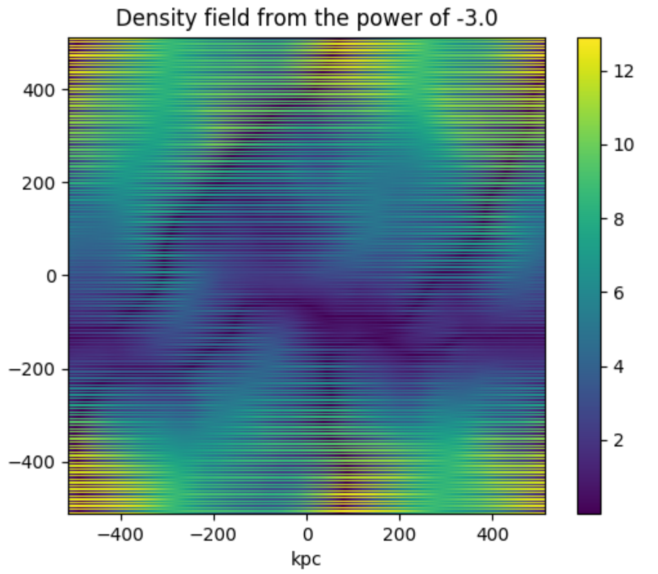

Figure 8: Density field 3

```
import numpy as np
import matplotlib.pyplot as plt
import sys
from scipy import stats
plt.ioff()


# In[ ]:


'''
Question 3: Linear structure growth
'''
print('Question 3: Linear structure growth')


def R_K_4 (ODE, stepsize,t,D_duo):
    '''
    Runge_Kutta methor (4th order)
    ODE : the ODE we want to solve
    initial : initial state. In our case D(1) & D'(1)
    '''
    h= stepsize
    # calculate yn+1 - yn = ?
    # This will use the old state D_duo which is so-called yn
    # D_duo cantains D0 and D1, which can be used to calculate D2
    # D1 and D2 determine the k and growth of D0 and D1
    k1 = h * ODE(t,D_duo)                          # k1 = h* f(x,y,y1,y2.......)
    k2 = h * ODE(t + h*0.5, D_duo + k1*0.5)
    k3 = h * ODE(t + h*0.5, D_duo + k2*0.5)
    k4 = h * ODE(t + h, D_duo + k3)
    growth = k1/6. + k2/3. + k3/3. + k4/6.
    #update the state to yn+1 = D_duo
    D_duo += growth
    return D_duo
def ODE(t, D_duo):
    '''
    as same as the f(x,y) in slides
    note that we have second order derivative here.
    D_duo will include both D and D', will call them D0 and D1
    and be propagated togather.
    '''
    #initial state
    D0 = D_duo[0]
    D1 = D_duo[1]

    D2 = 2./(3.*t**2)*D0 - 4./(3.*t)*D1
    return np.array((D1,D2))
def D_analytical(D0,D1,t):
    term1 = 3/5 * (D0+D1)
    term2 = D0 - term1
    return term1*t**(2/3) + term2*t**(-1)


# In[28]:


# solve case 1
D_duo_case1 = np.array((3.,2.))
stepsize = 0.1
t = np.arange(1,1000,stepsize)
```

```
D_case1_rk = np.zeros(len(t))
D_case1_a = np.zeros(len(t))
for i in range(len(t)):
    D_case1_rk[i] = R_K_4(ODE, 0.1,t[i],D_duo_case1)[0]
    D_case1_a[i] = D_analytical(3,2,t[i])
fig1 = plt.figure(1)
ax1_1 = fig1.add_subplot(111)
ax1_1.loglog(t,D_case1_rk, label = 'Runge-Kutta')
ax1_1.loglog(t,D_case1_a, label = 'Analytical' )
ax1_1.legend(loc='best')
ax1_1.set_xlabel('t')
ax1_1.set_ylabel('D(t)')
fig1.suptitle('case1')
fig1.savefig('q3_case1.png')
#fig1.show()


# In[32]:


#case 2
D_duo_case2 = np.array((10.,-10.))
stepsize = 0.1
t = np.arange(1,1000,stepsize)
D_case2_rk = np.zeros(len(t))
D_case2_a = np.zeros(len(t))
for i in range(len(t)):
    D_case2_rk[i] = R_K_4(ODE, 0.1,t[i],D_duo_case2)[0]
    D_case2_a[i] = D_analytical(10.,-10.,t[i])
fig2 = plt.figure(2)
ax2_1 = fig2.add_subplot(111)
ax2_1.loglog(t,D_case2_rk, label = 'Runge-Kutta')
ax2_1.loglog(t,D_case2_a, label = 'Analytical' )
ax2_1.legend(loc='best')
ax2_1.set_xlabel('t')
ax2_1.set_ylabel('D(t)')
fig2.suptitle('case2')
fig2.savefig('q3_case2.png')
#fig2.show()


# In[34]:


#case 3

D_duo_case3 = np.array((5.,0.))
stepsize = 0.1
t = np.arange(1,1000,stepsize)
D_case3_rk = np.zeros(len(t))
D_case3_a = np.zeros(len(t))
for i in range(len(t)):
    D_case3_rk[i] = R_K_4(ODE, 0.1,t[i],D_duo_case3)[0]
    D_case3_a[i] = D_analytical(5.,0.,t[i])
fig3 = plt.figure(3)
ax3_1 = fig3.add_subplot(111)
ax3_1.loglog(t,D_case3_rk, label = 'Runge-Kutta')
ax3_1.loglog(t,D_case3_a, label = 'Analytical' )
ax3_1.legend(loc='best')
ax3_1.set_xlabel('t')
ax3_1.set_ylabel('D(t)')
fig3.suptitle('case3')
```

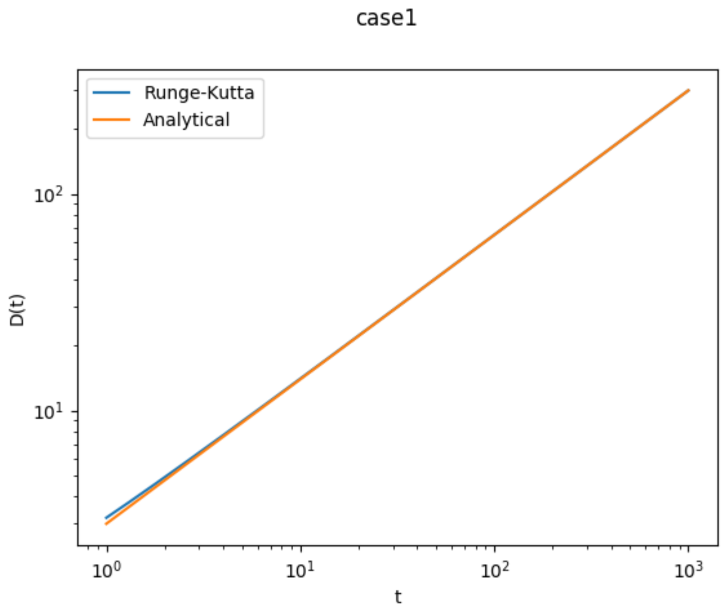

Figure 9: Result provied by Runge-Kutta method and analytical solution of case1.

```
fig3.savefig('q3_case3.png')
#fig3.show()


# In[ ]:
```

The fugures show that my results fit with analytical solution very well.

4 Question 4 : Zeldovich approximation

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

# In[15]:


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


# In[39]:


'''
Question 4 : Zeldovich approximation
4(a) : calculte the growth factor
'''
print('\n Question 4 : Zeldovich approximation')
print('\n 4(a) calculte the growth factor')
```

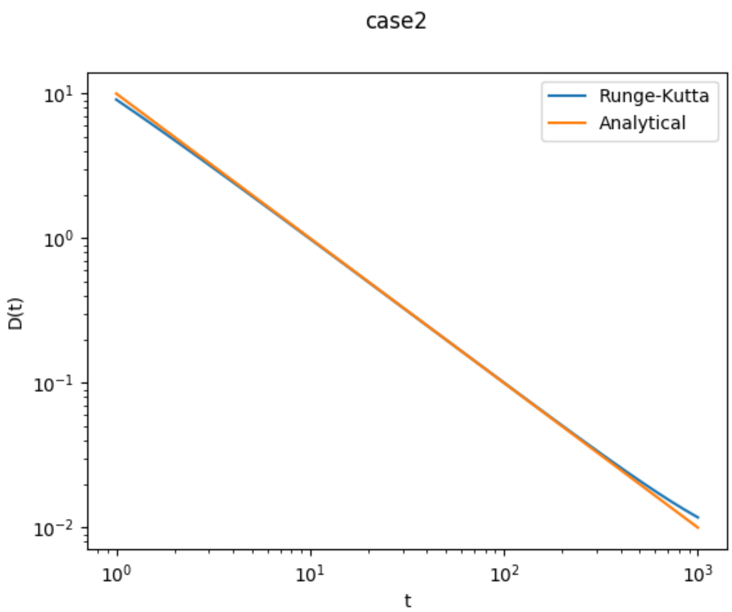

Figure 10: Result provied by Runge-Kutta method and analytical solution of case2.

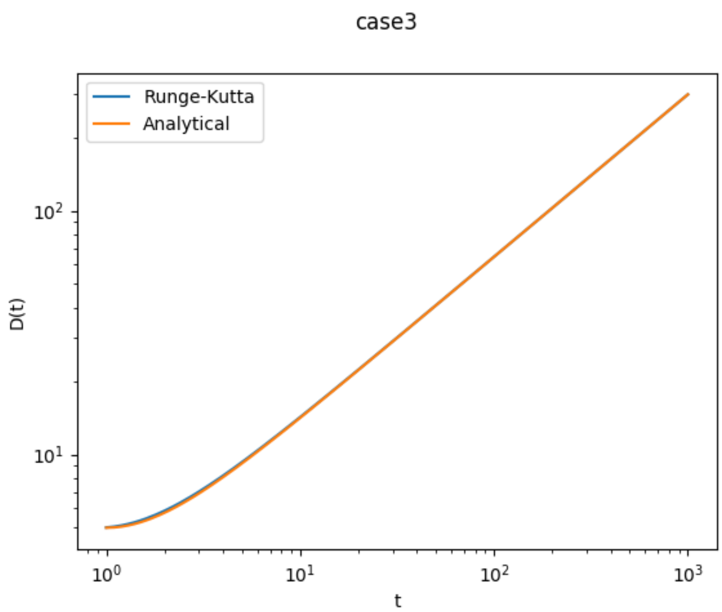

Figure 11: Result provied by Runge-Kutta method and analytical solution of case3.

```
def integrator(function, lower ,upper, n_intervals):
    '''
    trapeziodal rule
    function =
    lower = lower limit of the integral
    upper = upper limit
    interbvals = n of n_intervals
    '''
    h = (upper - lower) / n_intervals
    S = 0.5*(function(lower) + function(upper))
    for i in range(1,n_intervals):
        S += function(lower + i*h)
    integral = h * S
    return integral
def D_of_a_int(a_prime):
    '''
    use a, because 1/1+z will become a, which is more simpler.
    hereby I define the integral part of D(a)

    '''
    Omega_m=0.3
    Omega_lambda = 0.7
    return 1/(Omega_m/a_prime + Omega_lambda*a_prime**2)**(3/2)
z = 50
a = 1/(1+z)
Omega_m=0.3
Omega_lambda = 0.7
#calculte the coefficient part
coeffi = 5*Omega_m/2. *np.sqrt(Omega_m*(1+z)**3 + Omega_lambda) # for accuracy , use z to calculte.
integral = integrator(D_of_a_int,10**-8,a,10**6)
growth = coeffi * integral
print('growth factor = ' , growth)


# In[44]:


'''
4 (b)
dD/dt = 5*Omega_m*H0**2/(2*a**3*H(a)) * (-3*Omega_m*H(a)*integral/2*H0 + 1)
'''
print('\n 4(b)')
z=50
a = 1/(1+z)
H0 = 70
Ha = 70*(Omega_m*(1+z)**3+Omega_lambda)**0.5
analytical_d = 5*Omega_m*H0**2/(2*a**3*Ha) * (-3*Omega_m*Ha*integral/(2*H0) + 1)
print('analytical derivative at (z=50) = ', analytical_d)


# In[ ]:
```

4.1 Result

(a) calculte the growth factor
growth factor = 0.01960778042827206
. (b) analytical derivative at (z=50) = 34499.31702139651.

5 Question 5 : Mass assignment schemes

5.1 a. Nearest Grid Point

I loop over the particles and assign their densties to the cells and the fraction of particle's mass assigned to a cell 'ijk' is the S(x) averaged over this cell.For 3 dimensions, W(ijk) = W(X)*W(Y)*W(Z). NGP is the simplest PM algorithm that assume particles are point-like and all of particles's mass is assigned to the single grid cell that contains it.

```
print('Question 5 : Mass assignment schemes')
print('\n 5(a)')
def NGP(cell_size, positions):
    '''
    cell_size = the size of the cells ; (n,n,n)
    position : positions of the particles
    '''
    grid = np.zeros(cell_size)
    #assign the particles to the cell, indices is the cell that the
     particle belongs to, is int()
    indices = positions.astype(np.int)
    for i in range(indices.shape[1]):      #for every particles
        grid[indices[:,i][0],indices[:,i][1],indices[:,i][2]] += 1
       # located in a grid and count it
    return grid

# Particles' positions
np.random.seed(121)
positions = np.random.uniform(low=0,high=16,size=(3,1024))
grid = NGP((16,16,16),positions)

# Plot x-y slices of the grid
z=[4,9,11,14]
for i in range(4):
    plt.title('z={0} layer'.format(z[i]))
    plt.imshow(ngp[:,:,z[i]],extent=[0,16,16,0])
    plt.colorbar()
    #plt.savefig('q5_a{}.png'.format(i))
    plt.show()
```

5.2 b. Test NGP

I moved one particle along x axis and test the number in cell0 and cell4. When x is from 0 to 1, particle number in cell0 should be equal to 1. When x is from 4 to 5, particle number in cell4 should be equal to 1.

```
'''
(b) test the robustness
'''
print('\n 5(b) test the robustness')
x_test = np.linspace(0.1,16,30)   # move the particle along x-axis
cell4 = np.zeros(30)
cell0 = np.zeros(30)
for i in range (16):
    position_test = np.array(([x_test[i]],[0],[0]))
    grid_test = NGP((16,16,16),position_test)
    cell4[i] = grid_test[4,0,0]
    cell0[i] = grid_test[0,0,0]

#plot numbers in cell 4 first, when x = 4 to 5 , cell 4 should be 1
```

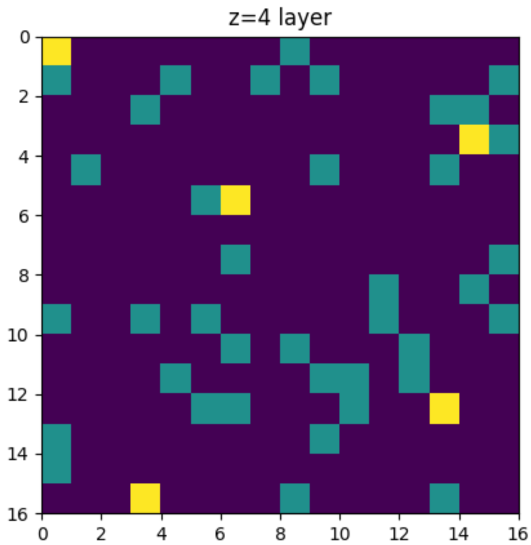

Figure 12: Slice of NGP at z = 4.

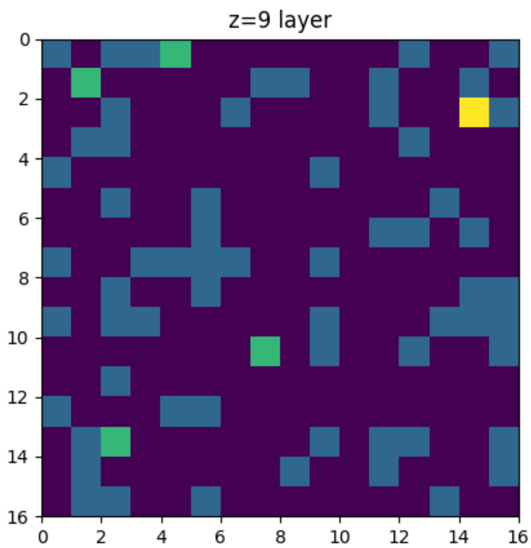

Figure 13: Slice of NGP at z = 9.

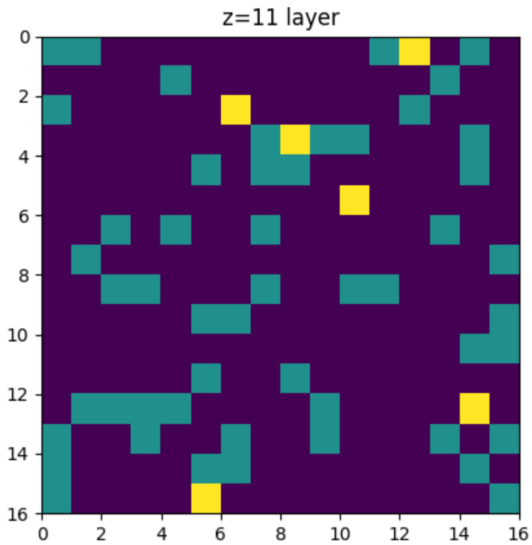

Figure 14: Slice of NGP at z = 11.

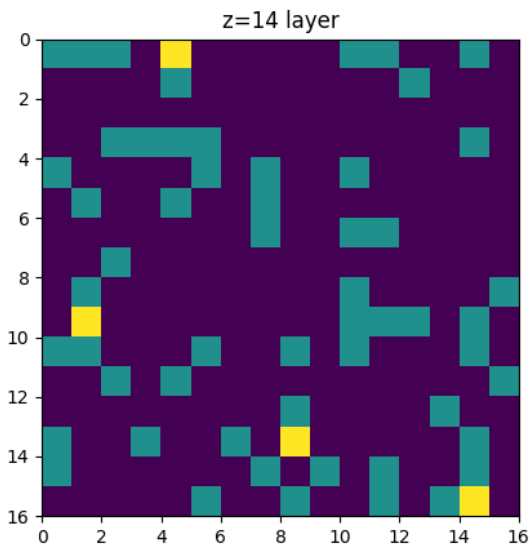

Figure 15: Slice of NGP at z = 14.

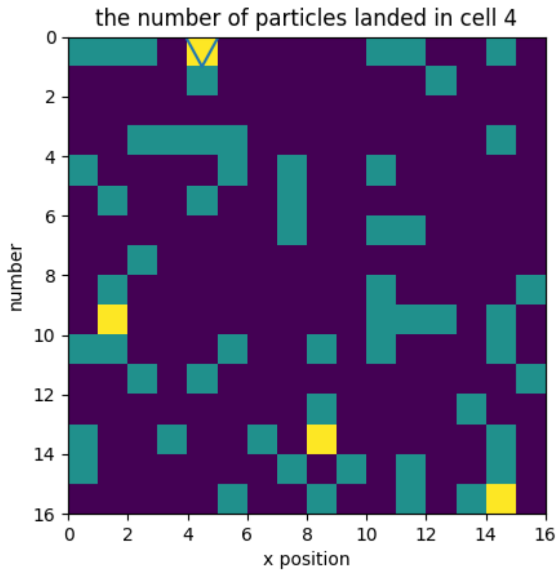

Figure 16: Robustness test on cell0

```
plt.plot(x_test, cell4)
plt.title('the number of particles landed in cell 4')
plt.ylabel('number')
plt.xlabel('x position')
#plt.savefig('q5_b1.png')
plt.show()


# repeat for cell 0, when x = 0 to 1, cell 0 should be equal to 1
plt.plot(x_test, cell0)
plt.title('the number of particles landed in cell 0')
plt.ylabel('number')
plt.xlabel('x position')
#plt.savefig('q5_b2.png')
plt.show()
```

5.3 d. Fast Fourier transform

```
'''
5(d) FFT
'''
def DFT_slow(x):
    #1-D discrete Fourier Transform
    x = np.array(x, dtype=float)
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)      # use vector multiplication
def FFT_1D(x):
    # 1-D Fast FT
    x = np.array(x, dtype=float)
```

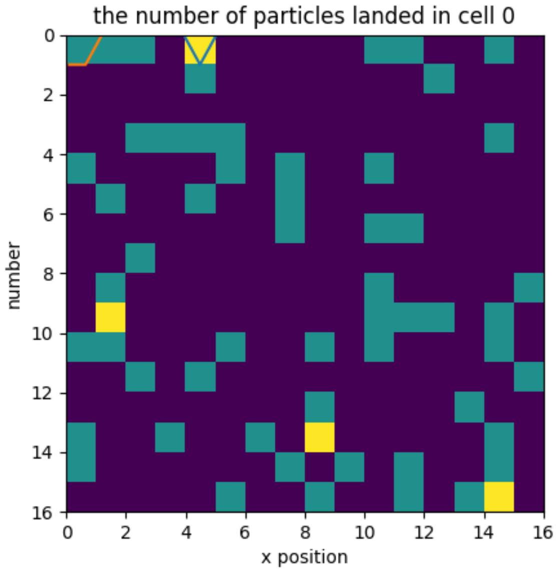

Figure 17: Robustness test on cell4

```
    N = len(x)

    if N % 2 != 0:
        raise ValueError("size of x must be a power of 2")
    elif N <= 4 :    # end of the recurse
        return DFT_slow(x)
    else:
        X_even = FFT_1D(x[::2])
        X_odd = FFT_1D(x[1::2])
        factor = np.exp(-2j * np.pi * np.arange(N) / N)
        return np.concatenate([X_even + factor[:int(N*0.5)] * X_odd
    ,
                               X_even + factor[int(N*0.5):] * X_odd
    ])
#test
#use sin(t)
fun1 = lambda t : np.sin(t)
N = 64
t_test = np.linspace(0,4*np.pi,N)
fun1t = fun1(t_test)
Xk = FFT_1D(fun1t)
fft_np = np.fft.fft(fun1t)

#plot
fs = 64/(4*np.pi)    # sampling frequency
fk = fs/N*np.arange(0,N*0.5,1)    #  fs/N = interval
plt.figure()
plt.plot(fk,np.abs(Xk)[:int(N*0.5)],label='My FFT')
plt.axvline(x=1/(2*np.pi), color='k', linestyle='--', label = '
    analytical')
plt.plot(fk,np.abs(fft_np[:int(N*0.5)]), linestyle='--', label = '
    FFT by numpy' )
plt.xlim(0,1)
```

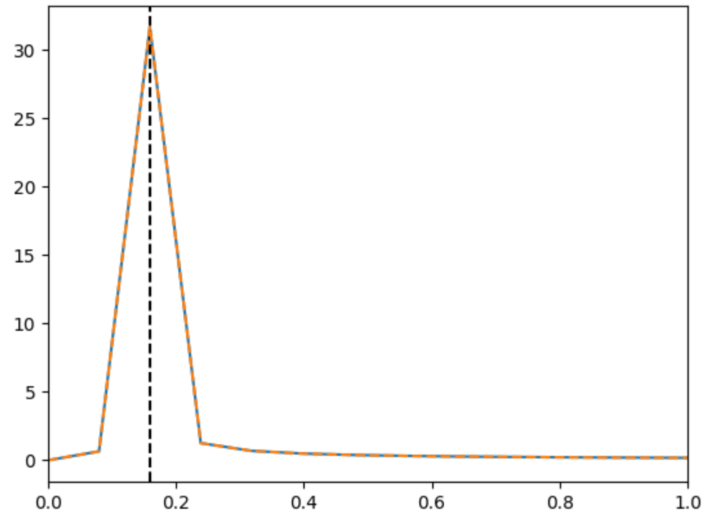

Figure 18: Robustness test on cell0

```
#plt.savefig('q5_d.png')
plt.legend(loc='best')
```

I used y = sin(x) to do the test, whose analytical frequency should be 1/2pi.

My result's peak shows great consistency with both np.fft and analytical Fourier transform.

5.4 e. FTT in 2-D and 3-D

```
'''
5(e): 2&3-D FFT
'''
def FFT_2D(x):
    '''
    2-D FFT. Becuase FFT_2D = FFT(FFT(x),y)
    '''
    F_xy = np.array(np.zeros(x.shape))
    # 1-D Fourier transform through the rows
    for i in range(len(x)):
        F_xy[i,:] = FFT_1D(x[i,:])
    # 1-D Fourier transform through the columns
    for j in range(len(x[0])):
        F_xy[:,j] = FFT_1D(F_xy[:,j])
    return F_xy

def FFT_3D(x):
    '''
    3-D FFT. Becuase FFT_3D = FFT_2D(FFT_2D(FFT_2D(x,y)),(y,z)),(x,
    z)))?
    I thought I am wrong here, sorry.
    '''
    F_xyz = np.array(np.zeros(x.shape))
    # 2-D Fourier transform through the x
```

```
    for i in range(len(x)):
        F_xyz[i,:,:] = FFT_2D(x[i,:,:])
    # 2-D Fourier transform through the y
    for j in range(len(x[0])):
        F_xyz[:,j,:] = FFT_2D(F_xyz[:,j,:])
    # 3-D Fourier transform through the z
    for k in range(len(x[1])):
        F_xyz[:,:,k] = FFT_2D(F_xyz[:,:,k])
    return F_xyz


# chose function f(x,y) = sin(x+y)
fun2 = lambda x,y : np.sin(x+y)
fun2_xy = np.zeros((64,64))
x_test,y_test = t_test ,t_test
for i in range(64):
    for j in range(64):
        fun2_xy[i,j] = fun2(x_test[i],y_test[j])

F_2D_result = FFT_2D(fun2_xy)

#plot function
plt.figure()
plt.imshow(fun2_xy)
plt.colorbar()
plt.title('Function')
#plt.savefig('q5_e1.png')
plt.show()
#plot fourier space
plt.figure()
plt.title('Fourier sapce')
plt.imshow(F_2D_result)
plt.colorbar()
#plt.savefig('q5_e2.png')
plt.show()
```

I used sin(x+y) as a testing funtion.

6 Question 6 : Classifying gamma-ray bursts

The work has three part as follow: First, I pre-processing the data including label the data and processing the missing data. Second, I conducting the gradiant ascend algorithm to train the classifer. Conclusion and discussion are followed in the last part.

6.1 Data Processing

Data are labelled first according to the parameter 'T90' showing the amount of long(label = 1) object is much more than otherwise. I revise the other features and find that there are many missing values, therefore, handling the missing data is of greaat importance.

I first plot the histogram to check the feature values.

It's seems that the mass M and metality Z conform to the gauss distribution. Meanwhile, SFR fits the exoponential distribution. Therefore, I choose to fill the non-determinded variables with random number with specific distribution. The data with SSFR and AV are few, so I decided to give up these two features.

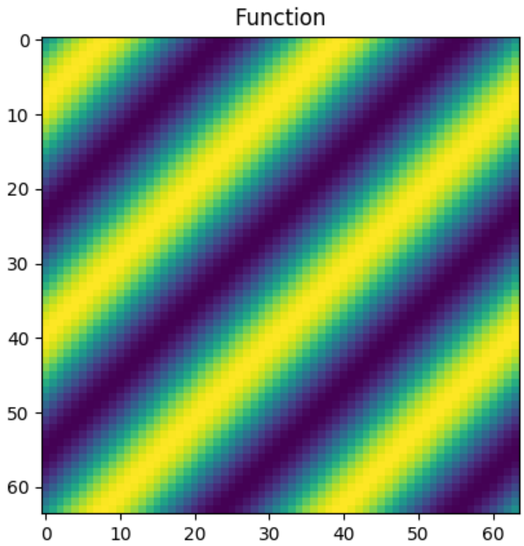

Figure 19: Testing function

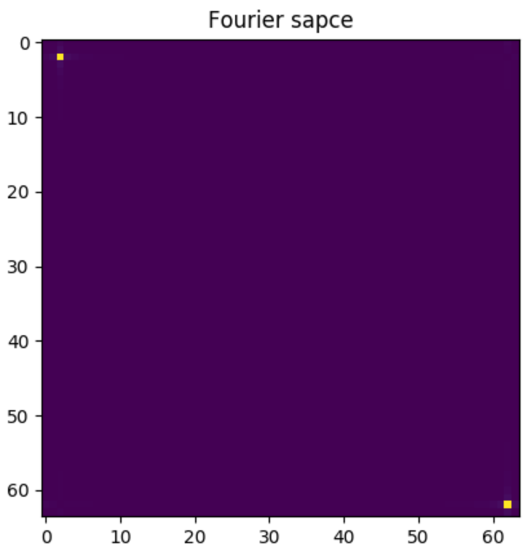

Figure 20: My FFT on testing function

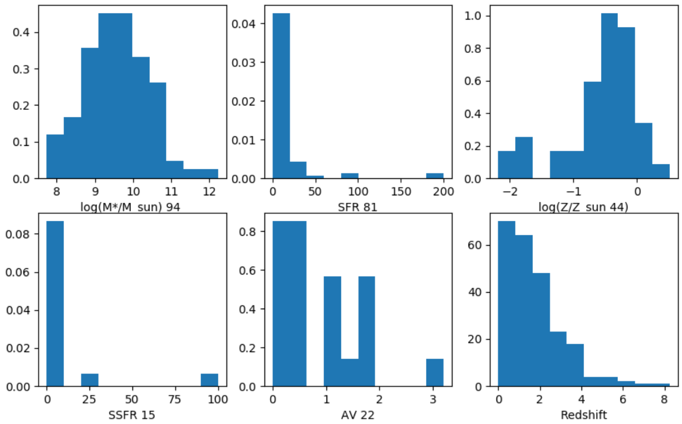

Figure 21: Histograms of the features

6.2 Train the classification

Applying the gradiant ascend algorithm, I plot two figures to show my results.

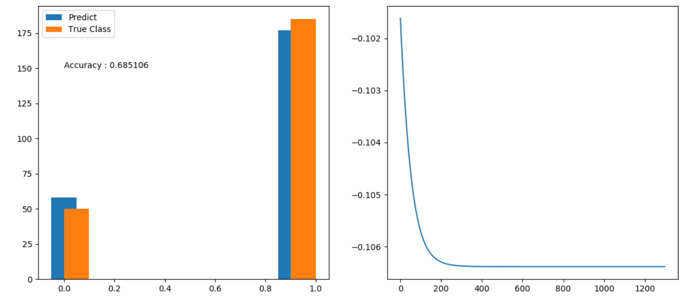

Figure 22: Accuracy and loss

The figures show that the classifer is well trained even if the result is not relative good enough. The histogram displays the similarity between true label and predicted result, however, the accuracy is merely 70 percent, which could be better.

6.3 Discussion

I think this lower accuracy is due to my handling of missing values. Because of wrong filled value, the intrinsic feature of some objects changed which lead to a bad result. I can't totally blame it on the bad algorithm in which play a role. Based on this, I can not discuss which features play the key role in classifying the class.

```
#!/usr/bin/env python
```

```
# coding: utf-8



# In[1]:


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
plt.ioff()


# ### Read the data file and label the data

# In[2]:


'''
Question 6 : Classifying gamma-ray bursts
'''
print('Question 6 : Classifying gamma-ray bursts')
#data = pd.read_csv('GRBs.txt', sep='\s+')
data = np.genfromtxt('GRBs.txt', usecols=(2,3,4,5,6,7,8))
data = pd.DataFrame(data, columns=['Redshift', 'T90', 'log(M*/M_sun)', 'SFR',
                                    'log(Z/Z_sun)', 'SSFR', 'AV'])
data['label'] = None
#Assign the label based on T90 parameter
data['label'][data['T90'] < 10] = 0
data['label'][data['T90'] >= 10] = 1
data['label'] = data['label'].convert_objects(convert_numeric=True)
print('Data Shape:', data.shape)


# ### Histogram the features.
# I first plot the histogram to check the feature values.

# In[3]:


# Check the missing data
index_M = data['log(M*/M_sun)'] != -1
index_SFR = data['SFR'] != -1
index_Z = data['log(Z/Z_sun)'] != -1
index_SSFR = data['SSFR'] != -1
index_AV = data['AV'] != -1

fig = plt.figure(figsize=(10,6))

ax1 = fig.add_subplot(2,3,1)
M = ax1.hist(data['log(M*/M_sun)'][index_M], density=True)
ax1.set_xlabel('log(M*/M_sun) %s '%len(data['log(M*/M_sun)'][
    index_M]))

ax2 = fig.add_subplot(2,3,2)
SFR = ax2.hist(data['SFR'][index_SFR], density=True)
ax2.set_xlabel('SFR %s' %len(data['SFR'][index_SFR]))

ax3 = fig.add_subplot(2,3,3)
Z = ax3.hist(data['log(Z/Z_sun)'][index_Z], density=True)
```

```
ax3.set_xlabel('log(Z/Z_sun %s)' %len(data['log(Z/Z_sun)'][index_Z
    ]))

ax4 = fig.add_subplot(2,3,4)
SSFR = ax4.hist(data['SSFR'][index_SSFR], density=True)
ax4.set_xlabel('SSFR %s' %len(data['SSFR'][index_SSFR]))

ax5 = fig.add_subplot(2,3,5)
AV = ax5.hist(data['AV'][index_AV], density=True)
ax5.set_xlabel('AV %s' %len(data['AV'][index_AV]))

ax6 = fig.add_subplot(2,3,6)
ax6.hist(data['Redshift'])
ax6.set_xlabel('Redshift')
fig.savefig('q6_1.png')


# It's seems that the mass M and metality Z conform to the gauss
    distribution. Meanwhile, SFR fits the exoponential distribution
    . Therefore, I choose to fill the non-determinded variables
    with random number with specific distribution.
# The data with SSFR and AV are few, so I decided to give up these
    two features.

# In[4]:


#Processing the missing data
miu_M = np.mean(data['log(M*/M_sun)'][index_M])
sigma_M = np.std(data['log(M*/M_sun)'][index_M])
lambda_SFR = np.mean(data['SFR'][index_SFR])
miu_Z = np.mean(data['log(Z/Z_sun)'][index_Z])
sigma_Z = np.std(data['log(Z/Z_sun)'][index_Z])

index = data['log(M*/M_sun)'] == -1
index_len = len(index)
data['log(M*/M_sun)'][index] = np.random.normal(miu_M, sigma_M,
    index_len)

index = data['log(Z/Z_sun)'] == -1
index_len = len(index)
data['log(Z/Z_sun)'][index] = np.random.normal(miu_Z, sigma_Z, size
    =index_len)

index = data['SFR'] == -1
index_len = len(index)
data['SFR'][index] = np.random.exponential(lambda_SFR, index_len)


# ## Part 2, Train the classification applying the gradiant ascend
    algorithm
# I plot two figures to show my results.

# In[5]:


def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# def binary_crossentropy(y_true, y_predict):
#     m = y_true.shape[0]
#     return -1/m * ()
```

```

def load_data(data):
    cols = ['Redshift', 'log(M*/M_sun)', 'SFR', 'log(Z/Z_sun)']
    data_Input = pd.DataFrame(data, columns=cols)
    data_Input = np.array(data_Input)
    data_Label = data['label']
    data_Label = np.array(data_Label)
    data_Input = np.insert(data_Input, 0, 1, axis=1)
    return data_Input, data_Label

def grad_ascent(data_Input, data_Label, alpha, epochs, loss=False):
    data_Mat = np.mat(data_Input)
    label_Mat = np.mat(data_Label).transpose()
    m, n = np.shape(data_Mat)
    weights = np.random.normal(0.5,0.2,(n,1))
    Loss = []
    for i in range(epochs):
        h = sigmoid(data_Mat * weights)
        weights = weights + alpha * data_Mat.transpose() * (h - label_Mat) / m
        if loss == True:
            if i % 2 == 0:
                Loss.append(-np.sum((np.array(label_Mat) - np.array(h))**2) / (2*m))

    return weights, Loss


# In[6]:


data_in, data_lab = load_data(data)
epoch = 1300
W, loss = grad_ascent(data_in, data_lab, alpha=0.001, epochs=epoch, loss=True)

W = np.array(W)
z = np.dot(data_in, W)
z = z.astype(float)
Prediction = sigmoid(z)
Prediction = np.array(Prediction, dtype=int)

fig = plt.figure(figsize=(14,6))
ax = fig.add_subplot(121)
ax.hist(Prediction, label='Predict',align='left')
ax.hist(data['label'], label='True Class')#, align='right')
ax.legend(loc = 2)
acc = np.sum(data['label'] == Prediction.flatten()) / data.shape[0]
ax.text(0, 150, 'Accuracy : %f' %acc)

ax2 = fig.add_subplot(122)
ax2.plot(np.arange(0,epoch,2),loss)
fig.savefig('q6_2.png')
```