NUR Assignment 2

Christiaan van Buchem - s1587064

May 28, 2019

Abstract

In this document I will be giving my answers to the questions of the second assignment for the Numerical Recipes for Astrophysics course. For each question I will give a short introduction, write out any non-coded answers that may be required, produce the print statements and the plots, and finally I will show the script used to produce the results.

1 Normally distributed pseudo-random numbers

1.1 RNG

For exercise 1 we were tasked with writing a random number generator that returns a random floating point number between 0 and 1. At minimum we had to use some combination of an MWC and a 64-bit XOR-shift. The plots made to test the quality of the RNG can be seen in Figures 1(a), 1(b), and 2.

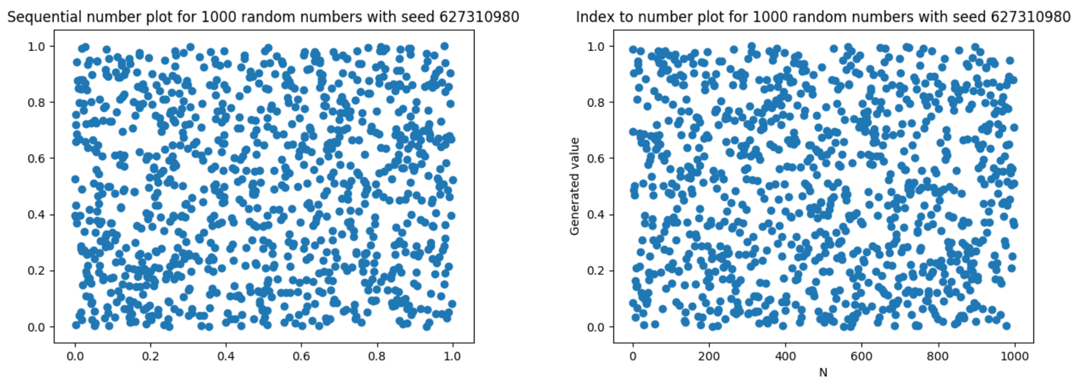

Figure 1: *Left:* Sequential number plot showing that it appears that each number is independent of its predecessor. *Right:* Index to number plot showing that there does not appear to be a relation between the index of a number and its value.

1.2 Box-Muller method

Using the Box-Muller method we had to generate 1000 normally distributed random numbers. In order to check if they follow the expected distribution we make a histogram with an over-plotted Gaussian. The results can be seen in Figure 3.

1.3 KS-test

For this exercise we tested whether or not our function is consistent with the normal distribution. The resulting plot can be seen in Figure 4. The slight difference between the two may be attributed to the fact that in the self written KS-test the following approximation was used:

$$P_{KS}(z) \approx \begin{cases} \frac{\sqrt{2\pi}}{z}[(e^{-\pi^2/(8z^2)}) + (e^{-\pi^2/(8z^2)})^9 + (e^{-\pi^2/(8z^2)})^{2}5], & (z < 1.18) \\ 1 - 2[(e^{-2z^2}) - (e^{-2z^2})^4 + (e^{-2z^2})^9], & (z \geq 1.18) \end{cases}$$

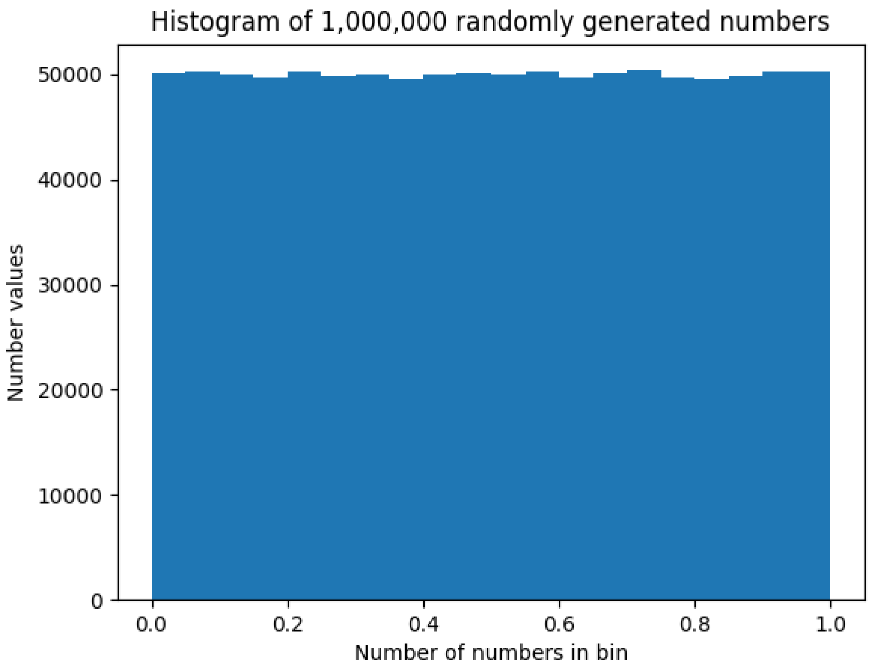

Figure 2: This histogram places the random number generator under a sharper knife, allowing us to see that there are some fluctuations between the bins. Overal it appears to be quite unbiased.

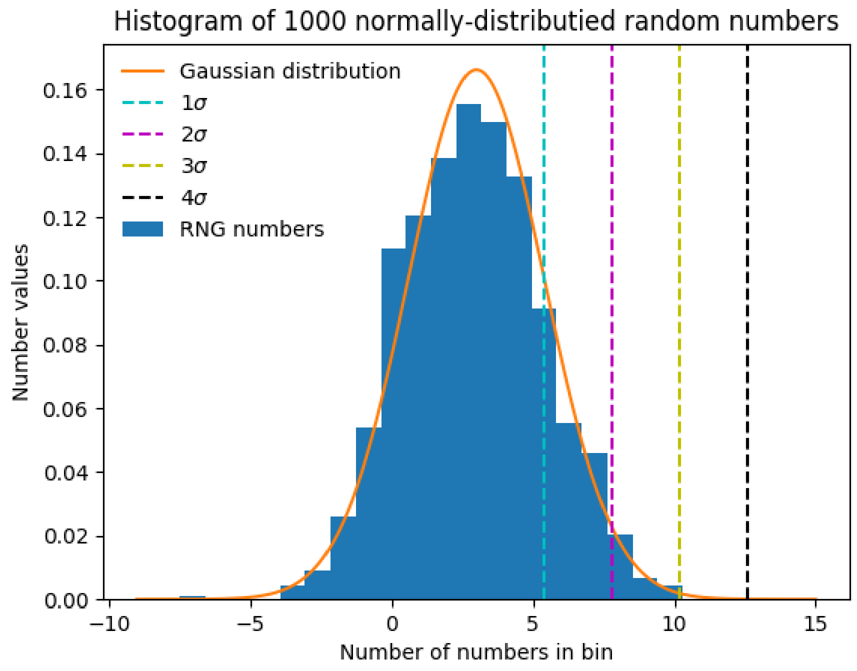

Figure 3: In this figure we can see that numbers generated using the Box-Muller method do indeed follow the Gaussian distribution.

1.4 Kuiper's-test

The same as for the KS-test except that we had to use Kuiper's test. Results can be seen in Figure 5.

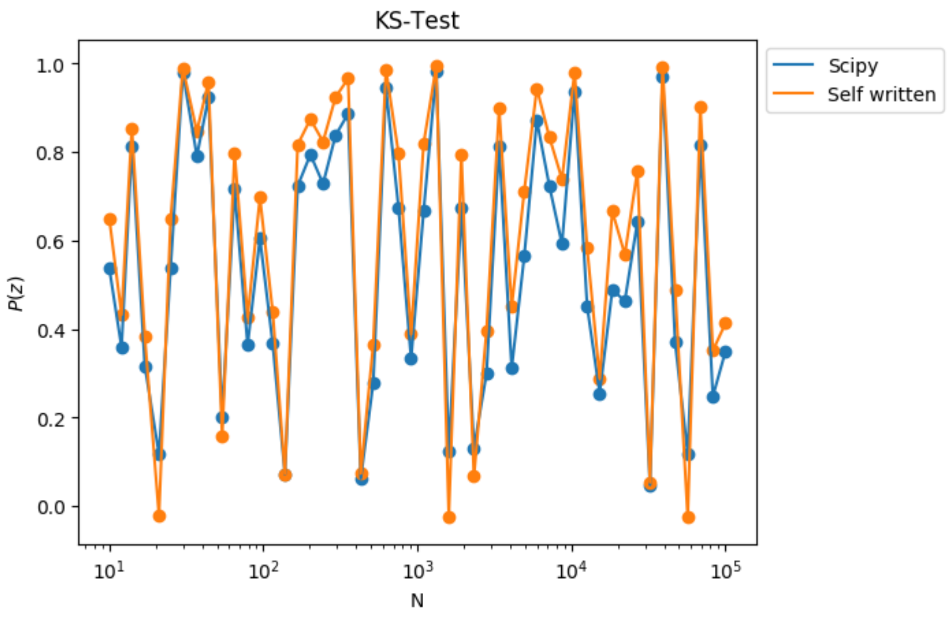

Figure 4: Here we see that the 'self-written' KS-test follows the Scipy KS-test results almost exactly.

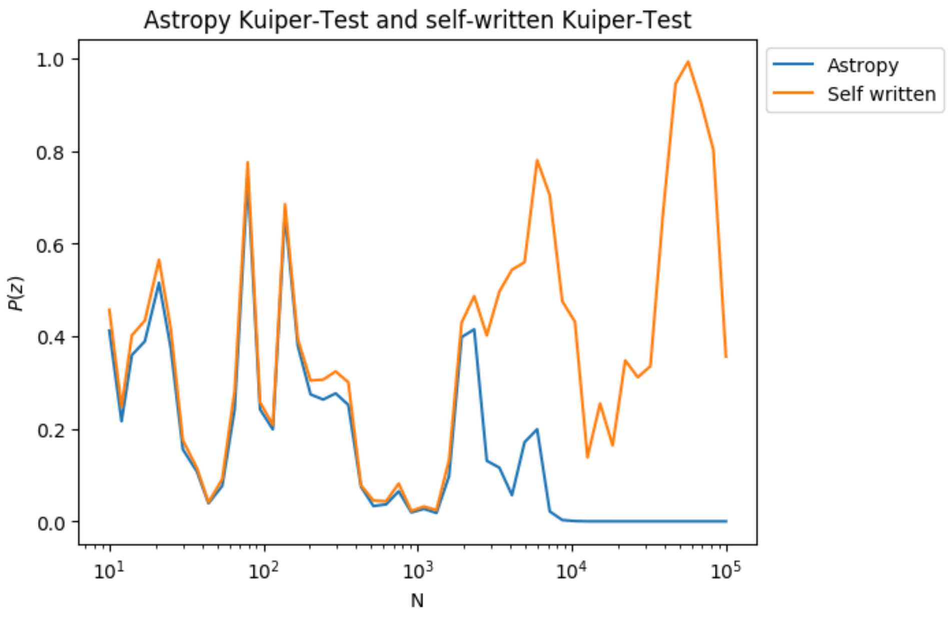

Figure 5: Here we compare the Kuipers test.

1.5 Analysing a dataset

In this exercise we were tasked with analysing a giving data set using either the KS-test or Kuipers test. The results can be seen in Figure 6. We decided to use the KS-Test for this exercise. It appears that the 3rd data set has also been drawn from a normal distribution due to the fact that it is the only one that remains above 0.

1.6 Scripts

Here we can see the terminal output of the script used for this exercise:

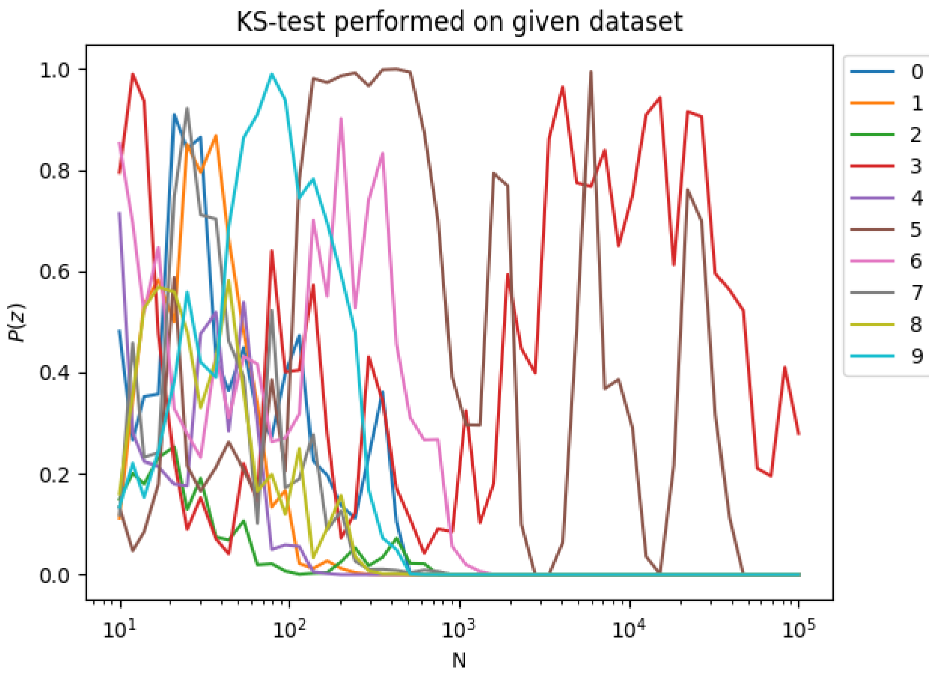

Figure 6: Analysing the different datasets.

a2_1.txt

Here is the script used to produce these results:

```
# a2_1
import numpy as np
import sys
from matplotlib import pyplot as plt
from scipy import stats
import os
from astropy.stats import kuiper

# --- Functions and classes ---

class rng(object):
    # Rng object that is initiated with a give seed
    a1,a2,a3 = np.int64(21),np.int64(35),np.int64(4)
    a = 4294957665


    def __init__(self, seed):
        self.state = np.int64(seed)

    def MWC(self):
        # Multiply with carry generator
        x = np.int64(self.state)
        self.state = self.a*(x&(2**32-1))+(x>>32)

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

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

def box_muller(u1,u2,mu,sigma):
    # Implementation of the Box Muller transform
    x1 = (-2*np.log(u1))**0.5*np.sin(2*np.pi*u2)
    x2 = (-2*np.log(u1))**0.5*np.cos(2*np.pi*u2)
    return x1*sigma+mu,x2*sigma+mu
#end box_muller

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

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

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

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

def KS_Kuip_test(sample,f,mu,sig,Kuip=False):
    # Implementation of the Kalgorov-Smirnov test
    N = len(sample)
    x = np.linspace(mu-5*sig,mu+5*sig,1000)
    F, Fn = np.zeros(len(x)), np.zeros(len(x))
    Dmin = 0
    Dmax = 0
    for i in range(len(Fn)):
        Fn[i] = len(np.where(sample<=x[i])[0])/N
        F[i] = romber_int(f,x[0],x[i])
        Dn = F[i] - Fn[i]
        if Dn > Dmin:
            Dmin = Dn
        Dn = Fn[i] - F[i]
        if Dn > Dmax:
            Dmax = Dn
    # Determine the manner in which D is calculated
    if Kuip:
        D = Dmin+Dmax
        # Calculate the probability
        z = (N**0.5+0.155+0.24*N**(-0.5))*D
        #print(z)
        if z < 0.4:
            P = 1
        else:
            P = 0
            for i in range(1,1000):
                Pi = 2*(4*i**2*z**2-1)*np.exp(-2*i**2*z**2)
                P += Pi
                if Pi <= 0.00001:
                    return D, P
        return D, P
    else:
        D = np.max((Dmin,Dmax))
        # Calculate the probability
        z = (N**0.5+0.12+0.11*N**(-0.5))*D
        if z < 1.18:
            P = (2*np.pi)**0.5*((np.exp(-1*np.pi**2/(8*z**2)))+(np.exp(-1*np.pi**2/(8*z**2)))**9+(np.exp(-1*np.pi**2/(8*z**2)))**25)
        else:
            P = 1-2*((np.exp(-2*z**2))-(np.exp(-2*z**2))**4+(np.exp(-2*z**2))**9)
        return D,1-P
#end KS_test()

def Ks_test_2s(sample1,sample2,mu,sig,Kuip=False):
    # Implementation of the Kalgorov-Smirnov test
    N1,N2 = len(sample1),len(sample2)
    x = np.linspace(mu-5*sig,mu+5*sig,1000)
    F, G = np.zeros(len(x)), np.zeros(len(x))
    Dmin,Dmax = 0,0
    for i in range(len(x)):
        F[i] = len(sample1[sample1<=x[i]])/N1
        G[i] = len(sample2[sample2<=x[i]])/N2

        Dn = F[i] - G[i]
        if Dn > Dmin:
            Dmin = Dn
        Dn = G[i] - F[i]
        if Dn > Dmax:
            Dmax = Dn
    # Determine the manner in which D is calculated
    if Kuip:
        D = Dmin+Dmax
    else:
        D = np.max((Dmin,Dmax))
    # Calculate the probability
    z = (N1**0.5+0.12+0.11*N1**(-0.5))*D
    if z < 1.18:
        P = (2*np.pi)**0.5*((np.exp(-1*np.pi**2/(8*z**2)))+(np.exp(-1*np.pi**2/(8*z**2)))**9+(np.exp(-1*np.pi**2/(8*z**2)))**25)
        return D,1-P
    else:
        P = 1-2*((np.exp(-2*z**2))-(np.exp(-2*z**2))**4+(np.exp(-2*z**2))**9)
```

```python
        return D,1-P
#end KS_test()

def random_field_generator(n,N,rng,mu=0):
    # Prepares a random field in Fourier space
    print(f'Generating a random field with n = {n} of dimension {N}x{N} (mu = {mu})')
    df = np.zeros((N,N),dtype=complex)
    # Setting values of top half of the field
    for j in range((N//2)+1):
    # Determining the value of k_y
        k_y = j*2*np.pi/N
        for i in range(N):
            # Determining the value of k_x and sigma_x
            if i <= (N//2):
                k_x = (i)*2*np.pi/N
            else:
                k_x = (-N+i)*2*np.pi/N
            # Avoid dividing by 0
            if i != 0 or j != 0:
                sig = ((k_x**2+k_y**2)**0.5)**(n/2)
            else:
                sig = 0
            # Drawing a random number from normal distrib
            #df[j][i] = np.random.normal(0,sig)+ 1j*np.random.normal(0,sig)
            rand = box_muller(rng.rand_num(1),rng.rand_num(1),mu,sig)
            df[j][i] = rand[0] + 1j*rand[1]
    # Setting values of points who need to equal their own conjugates
    df[0][0] = 0
    df[0][N//2] = (df[0][N//2].real)**2
    df[N//2][0] = (df[N//2][0].real)**2
    df[N//2][N//2] = (df[N//2][N//2].real)**2
    # Setting values of bottom half of the field using conjugates
    for j in range((N//2)+1):
        for i in range(N):
            df[-j][-i]= df[j][i].conjugate()
    return df
#end random_field generator()

def gauss_cdf(x):
        gauss = lambda x : 1/(2*np.pi*sig**2)**0.5*np.exp(-0.5*(x-mu)**2/sig**2)
        cdf = np.zeros(len(x))
        for i in range(len(x)):
            cdf[i] = romber_int(gauss,-5,x[i])
        return cdf

# --- Commands, prints and plots ---
if __name__ == '__main__':
    print('--- Exercise 1 ---')
    seed = 627310980
    rng = rng(seed)
    print('Original seed:',seed)

    #--- 1.a ---
    # MWC and XOR-Shift
    N = 1000
    rand = rng.rand_num(N)
    # Sequential number plot
    plt.scatter(rand[:(len(rand)-1)],rand[1:])
    plt.title('Sequential number plot for {} random numbers with seed {}'.format(1000,seed))
    plt.savefig('plots/1a.png')
    plt.close()
    print('Generated plots/1a.png')
    # Index to number plot
    plt.scatter(np.arange(0,N,1),rand)
    plt.title('Index to number plot for {} random numbers with seed {}'.format(1000,seed))
    plt.xlabel('N')
    plt.ylabel('Generated value')
    plt.savefig('plots/1b.png')
    plt.close()
    print('Generated plots/1b.png')
    # Histogram
    N = 1000000
    rand = rng.rand_num(N)
    plt.hist(rand,bins=20,range=(0,1))
    plt.title('Histogram of 1,000,000 randomly generated numbers'.format(1000,seed))
    plt.xlabel('Number of numbers in bin')
    plt.ylabel('Number values')
    plt.savefig('plots/1c.png')
    plt.close()
    print('Generated plots/1c.png')

    #--- 1.b ---
    # Box-Muller method
    N = 1000
    mu, sig = 3,2.4
    rand = box_muller(rng.rand_num(N),rng.rand_num(N),mu,sig)
    gauss = lambda x,mu,sig : 1/(2*np.pi*sig**2)**0.5*np.exp(-0.5*(x-mu)**2/sig**2)
    x = np.linspace(mu-(sig*5),mu+(sig*5),1000)
    plt.hist(rand[0],bins=20,label='RNG numbers',density=1)
    plt.plot(x,gauss(x,mu,sig),label='Gaussian distribution')
    plt.title('Histogram of {} normally-distributied random numbers'.format(1000))
    plt.xlabel('Number of numbers in bin')
    plt.ylabel('Number values')
    plt.axvline(x=mu+sig,label='$1\sigma$',color='c',linestyle='--')
    plt.axvline(x=mu+2*sig,label='$2\sigma$',color='m',linestyle='--')
    plt.axvline(x=mu+3*sig,label='$3\sigma$',color='y',linestyle='--')
    plt.axvline(x=mu+4*sig,label='$4\sigma$',color='k',linestyle='--')
    plt.legend(frameon=False)
    plt.savefig('plots/1d.png')
    plt.close()
    print('Generated plots/1d.png')

    #--- 1.c. ---
    # KS-test
    # Setting parameters
    mu,sig = 0,1
    rand = box_muller(rng.rand_num(N),rng.rand_num(N),mu,sig)
    gauss = lambda x : 1/(2*np.pi*sig**2)**0.5*np.exp(-0.5*(x-mu)**2/sig**2)
    n = np.logspace(np.log10(10),np.log10(100000),dtype=int)
    # Preparing arrays
    P,P_s = np.zeros(len(n)),np.zeros(len(n))
    d,d_s = np.zeros(len(n)),np.zeros(len(n))
    # Running test for different values of N
    for i in range(len(n)):
        rand = box_muller(rng.rand_num(n[i]),rng.rand_num(n[i]),mu,sig)
        d[i],P[i] = KS_Kuip_test(rand[0],gauss,mu,sig)
        d_s[i],P_s[i] = stats.kstest(rand[0],'norm')
    # Plotting
    plt.plot(n,P_s,label='Scipy')
    plt.scatter(n,P_s)
    plt.plot(n,P,label='Self written')
    plt.scatter(n,P)
    plt.title('KS-Test')
    plt.ylabel('$P(z)$')
    plt.xlabel('N')
    plt.xscale('log')
    lgd = plt.legend(loc=2, bbox_to_anchor=(1,1))
    plt.savefig('plots/1e.png',bbox_inches='tight')
    plt.close()
    print('Generated plots/1e.png')

    #---1.d---
    # Kuipers test
    # Preparing arrays
    kuip_P,kuip_P_ast = np.zeros(len(n)),np.zeros(len(n))
    kuip_d,kuip_d_ast = np.zeros(len(n)),np.zeros(len(n))

    # Running test for different values of N
    rand_bm = box_muller(rng.rand_num(n[-1]),rng.rand_num(n[-1]),mu,sig)
    for i in range(len(n)):
        rand = rand_bm[0][:n[i]]
        kuip_d[i],kuip_P[i] = KS_Kuip_test(rand,gauss,mu,sig,Kuip=True)
        kuip_d_ast[i],kuip_P_ast[i] = kuiper(rand,gauss_cdf)
    # Plotting
    plt.plot(n,kuip_P_ast,label='Astropy')
    plt.plot(n,kuip_P,label='Self written')
    plt.title('Astropy Kuiper-Test and self-written Kuiper-Test')
    plt.ylabel('$P(z)$')
    plt.xlabel('N')
    plt.xscale('log')
    lgd = plt.legend(loc=2, bbox_to_anchor=(1,1))
    plt.savefig('plots/1f.png', bbox_inches='tight')
    plt.close()
    print('Generated plots/1f.png')

    #---1.e---
    # Testing on given random numbers
    filename = 'randomnumbers.txt'
    url = 'https://home.strw.leidenuniv.nl/~nobels/coursedata/'
    if not os.path.isfile(filename):
        print(f'File not found, downloading {filename}')
        os.system('wget '+url+filename)
    random_num = np.genfromtxt(filename,delimiter=' ',skip_footer=1)

    n = np.logspace(np.log10(10),np.log10(len(random_num)),dtype=int)
    test_P,test_D = np.zeros((10,len(n)),dtype=list),np.zeros((10,len(n)),dtype=list)
    # Applying Kuipers test
    for i in range(10):
        for j in range(len(n)):
            rand = np.array(random_num[:n[j],i])
            test_D[i][j],test_P[i][j] = KS_Kuip_test(rand,gauss,mu,sig,Kuip=True)
    # Plotting
    for i in range(10):
        plt.plot(n,test_P[i],label = i)
    plt.title('KS-test performed on given dataset')
    plt.ylabel('$P(z)$')
    plt.xlabel('N')
    plt.xscale('log')
    plt.legend(loc=2, bbox_to_anchor=(1,1))
    plt.savefig('plots/1g.png',bbox_inches='tight')
    plt.close()
    print('Generated plots/1g.png')
```

a2_1.py

2 Making an initial density field

For this exercise we were asked to generate a Gaussian random field. The field is generated in Fourier Space. The complex Fourier amplitudes are given by $\tilde{Y} = |\tilde{Y} exp(i\phi)$ where *phi* is a random phase. The power spectrum has the following form:

$$P(k) \propto k^n \tag{1}$$

In Figure 7 the generated Gaussian random fields are given for different n values.

Choose a minimum physical size and explain how this impacts the maximum physical size, the minimum k and maximum k.

2.1 Scripts

Here we can see the terminal output of the script used for this exercise:

```
--- Exercise 2 ---
Original seed: 627310980
Generating a random field with n = -1 of dimension 1024x1024 (mu = 0)
```

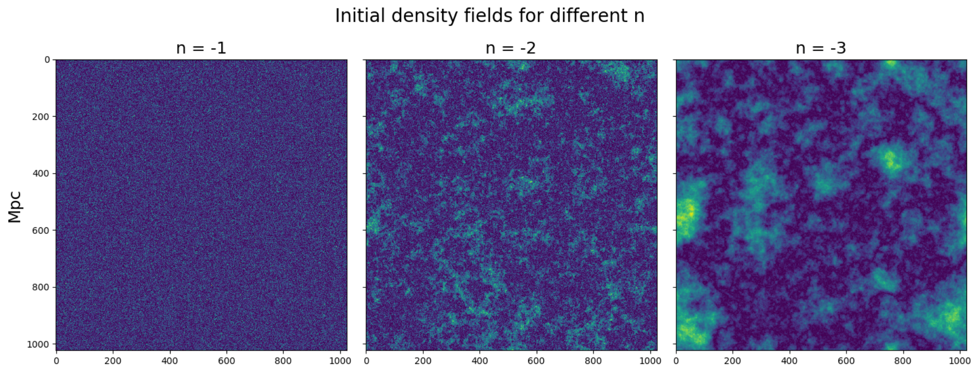

Figure 7: Gaussian random fields for different n values. Notice the clear presence of larger structure when the spectrum is more peaked (lower n).

```
Generating a random field with n = −2 of dimension 1024x1024 (mu = 0)
Generating a random field with n = −3 of dimension 1024x1024 (mu = 0)
Generated plots/2.png
```

a2_2.txt

Here is the script used to produce these results:

```
# a2_2
import numpy as np
import sys
from matplotlib import pyplot as plt
from scipy import stats
import os
from a2_1 import rng,box_muller

# --- Functions and classes ---

def random_field_generator(n,N,rng,mu=0):
    # Prepares a random field in Fourier space
    print(f'Generating a random field with n = {n} of dimension {N}x{N} (mu = {mu})')
    df = np.zeros((N,N),dtype=complex)
    # Setting values of top half of the field
    for j in range((N//2)+1):
    # Determining the value of k_y
        k_y = j*2*np.pi/N
        for i in range(N):
            # Determining the value of k_x and sigma_x
            if i <= (N//2):
                k_x = (i)*2*np.pi/N
            else:
                k_x = (-N+i)*2*np.pi/N
            # Avoid dividing by 0
            if i != 0 or j != 0:
                sig = ((k_x**2+k_y**2)**0.5)**(n/2)
            else:
                sig = 0
            # Drawing a random number from normal distrib
            #df[j][i] = np.random.normal(0,sig)+ 1j*np.random.normal(0,sig)
            rand = box_muller(rng.rand_num(1),rng.rand_num(1),mu,sig)
            df[j][i] = rand[0] + 1j*rand[1]
    # Setting values of points who need to equal their own conjugates
    df[0][0] = 0
    df[0][N//2] = (df[0][N//2].real)**2
    df[N//2][0] = (df[N//2][0].real)**2
    df[N//2][N//2] = (df[N//2][N//2].real)**2
    # Setting values of bottom half of the field using conjugates
    for j in range((N//2)+1):
        for i in range(N):
            df[-j][-i]= df[j][i].conjugate()
    return df
```

```
#end random_field generator()

# --- Commands, prints and plots ---
if __name__ == '__main__':
    print('--- Exercise 2 ---')
    seed = 627310980
    rng = rng(seed)
    print('Original seed:',seed)

    # Making initial density fields for different n values
    N = 1024
    df1 = random_field_generator(-1,N,rng)
    df1_inft = np.fft.ifft2(df1)
    df2 = random_field_generator(-2,N,rng)
    df2_inft = np.fft.ifft2(df2)
    df3 = random_field_generator(-3,N,rng)
    df3_inft = np.fft.ifft2(df3)
    # Plotting fields
    fig, ((ax1,ax2,ax3)) = plt.subplots(1, 3,sharex='col', sharey='row',figsize=(15,15))
    ax1.set_title('n = -1',size=18)
    ax1.imshow(np.abs(df1_inft))
    ax1.set_ylabel('Mpc',size=18)
    ax1.invert_yaxis()
    ax2.set_title('n = -2',size=18)
    ax2.imshow(np.abs(df2_inft ))
    ax3.set_title('n = -3',size=18)
    ax3.imshow(np.abs(df3_inft ))
    fig.suptitle('Initial density fields for different n',y=0.7,size=20)
    fig.tight_layout()
    plt.savefig('plots/2.png',bbox_inches='tight',pad_inches = 0)
    plt.close()
    print('Generated plots/2.png')
```

a2_2.py

3 Linear Structure Growth

The evolution of density perturbations in the initial universe evolves according to the following equation:

$$\frac{\partial^2\delta}{\partial t^2} + 2\frac{\dot{a}}{a}\frac{\partial\delta}{\partial t} = \frac{3}{2}\Omega_0 H_0^2\frac{\delta}{a^3} \tag{2}$$

In the early Universe we can separate the density perturbation as having a spatial part and a temporal part: $\delta = D(t)\Delta(x)$. In the case of a second order equation we have two growth factors. This means that the above partial differential equation becomes:

$$\frac{d^2D}{dt^2} + 2\frac{\dot{a}}{a}\frac{dD}{dt} = \frac{3}{2}\Omega_0 H_0^2\frac{D}{a^3} \tag{3}$$

We were asked to look at a Einstein-de Sitter Universe where $\Omega_m = 1$ and the scale factor is given by:

$$a(t) = (\frac{3}{2}H_0 t)^{2/3} \tag{4}$$

The density growth equation for this Universe is the following:

$$\frac{d^2D}{dt^2} = \frac{-4}{3t}\frac{dD}{dt} + \frac{2}{3t^2}D \tag{5}$$

For this exercise we were to calculate the numerical solutions for three different sets of initial conditions. These results were then to be compared with the analytical solutions of the ODE.

In Table **??** we can see the different cases and their analytical solutions.

In Figure 8 we can see the numerical and analytical solutions for the 3 different cases.

Mention why they do not match.

	D(1)	D'(2)	D(t)
case 1	3	2	$3t^{2/3}$
case 2	10	-10	$10t^{-1}$
case3	5	0	$(3t^{5/3}+2)t^{-1}$

Table 1: The three different sets of initial conditions.

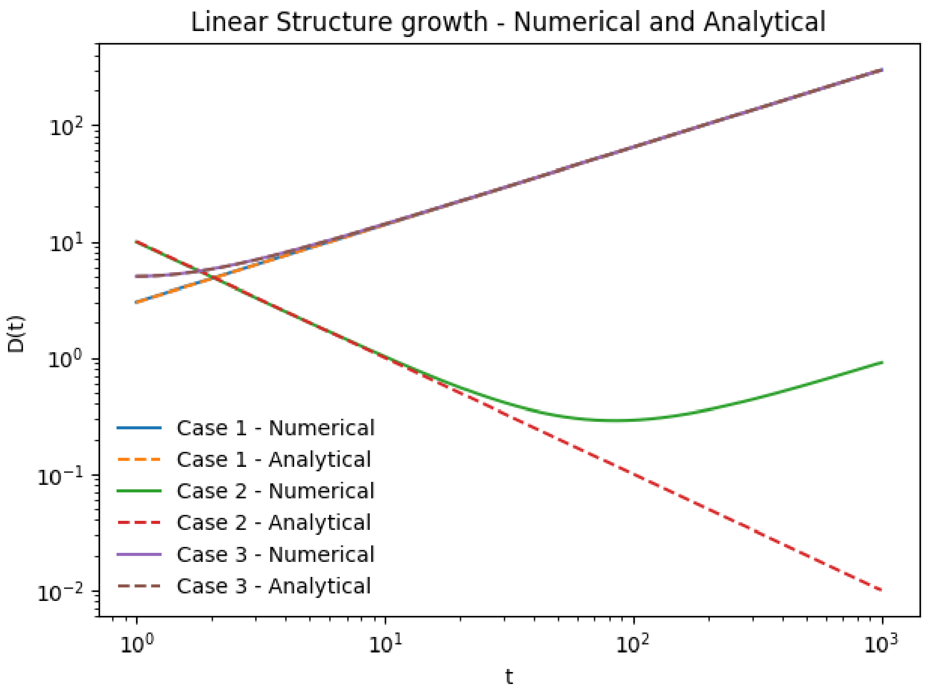

Figure 8: Analytical and numerical solutions to the partial differential equations given in this question.

3.1 Scripts

Here we can see the terminal output of the script used for this exercise:

```
—— Exercise 1 ——
 Original seed: 627310980
 Generated plots/1a.png
 Generated plots/1b.png
 Generated plots/1c.png
 Generated plots/1d.png
 Generated plots/1e.png
 Generated plots/1f.png
 Generated plots/1g.png
—— Exercise 3 ——
```

a2_3.txt

Here is the script used to produce these results:

```
# a2_3
import numpy as np
import sys
import matplotlib.pyplot as plt
from a2_1 import rng,box_muller

def k_calc(h,f,t,x1,x2,xn):
    # Likely soruce of error: What do we do with the second variable when calculating k?
    # To-do: Try to solve the problem by applying it on a simpler function
    k1 = h * f(t,x1,x2)
    k2 = h * f(t+0.5*h,x1+0.5*k1,x2+0.5*k1)
    k3 = h * f(t+0.5*h,x1+0.5*k2,x2+0.5*k2)
    k4 = h * f(t+h,x1+k3,x2+k3)
    return xn+1/6*k1+1/3*k2+1/3*k3+1/6*k4

def runge_kutta(x0,y0,f,xmax,h=0.0001):
    #Implementaiton of the Runge-Kutta method for ODE integration
    # x0,y0 are the starting values and f is the ode()
    xn,yn = x0,y0
    y_out,x_out = [],[]
    while xn < xmax:
        yn_new = k_calc(h,f,xn,yn)
        y_out.append(yn_new)
        xn += h
        x_out.append(xn)
        yn = yn_new

    plt.plot(x_out,y_out)

    return np.sum(y_out)*h

def k_calc2nd(h,f,g,t,x1,x2):
    # Support function for runge_kutta method (for 2nd order ODEs)
    k1 = h * f(t,x1,x2)
    l1 = h * g(t,x1,x2)
    k2 = h * f(t+0.5*h,x1+0.5*k1,x2+0.5*l1)
    l2 = h * g(t+0.5*h,x1+0.5*k1,x2+0.5*l1)
    k3 = h * f(t+0.5*h,x1+0.5*k2,x2+0.5*k2)
    l3 = h * g(t+0.5*h,x1+0.5*k2,x2+0.5*k2)
    k4 = h * f(t+h,x1+k3,x2+k3)
    l4 = h * g(t+h,x1+k3,x2+k3)

    x1_new = x1+1/6*(k1+2*k2+2*k3+k4)
    x2_new = x2+1/6*(l1+2*l2+2*l3+l4)

    return x1_new, x2_new
#end k_calc2nd()

def runge_kutta2nd(x1_0,x2_0,t0,t,f,g,h=0.01):
    #Implementaiton of the Runge-Kutta method for ODE integration
    # x0,y0 are the starting values and f is the ode()
    t = np.arange(t0,t+h,h)
    #print(t)
    x1n,x2n = x1_0,x2_0
    x1_out = np.zeros(len(t))
    for i in range(len(t)):
        x1n,x2n = k_calc2nd(h,f,g,t[i],x1n,x2n)
        x1_out[i] = x1n
    return np.sum(x1_out)*h,x1_out

# --- Commands, prints and plots ---
if __name__ == '__main__':
    print('--- Exercise 3 ---')
    seed = 627310980
    rng = rng(seed)
    print('Original seed:',seed)

    f = lambda t,x1,x2: x2
    g = lambda t,x1,x2: -4/(3*t)*x2 + 2/(3*t**2)*x1
    case1,yt1 = runge_kutta2nd(3,2,1,1000,f,g)
    case2,yt2 = runge_kutta2nd(10,-10,1,1000,f,g)
    case3,yt3 = runge_kutta2nd(5,0,1,1000,f,g)
    print(f'case1: {case1},case2: {case2}, case3: {case3}')

    f = lambda t,x1,x2 : x2
    g = lambda t,x1,x2 : x1*6-x2

    D1 = lambda t : 3*t**(2/3)
    D2 = lambda t : 10/t
    D3 = lambda t : (3*t**(5/3)+2)/t
    t = np.arange(1,1000+0.01,0.01)
    plt.plot(t,yt1,label='Case 1 - Numerical')
    plt.plot(t,D1(t),linestyle='--',label='Case 1 - Analytical')
    plt.plot(t,yt2,label='Case 2 - Numerical')
    plt.plot(t,D2(t),linestyle='--',label='Case 2 - Analytical')
    plt.plot(t,yt3,label='Case 3 - Numerical')
    plt.plot(t,D3(t),linestyle='--',label='Case 3 - Analytical')
    plt.legend(frameon=False)
    plt.xlabel('t')
    plt.ylabel('D(t)')
    plt.title('Linear Structure growth - Numerical and Analytical')
    plt.xscale('log')
    plt.yscale('log')
    plt.tight_layout()
    plt.savefig('plots/3.png')
    plt.close()
    print('Generated plots/3.png')
```

a2_3.py

4 Zeldovich approximation

In this exercise we will be looking at the Zeldovich approximation.

4.1 Calculating the linear growth factor to a given redshift.

Our first task was to integrate the linear growth factor up to a redshift of $z = 50$. The integral to be solved is the following:

$$D(z) = \frac{5\Omega_m H_0^2}{2} H(z) \int_z^\infty \frac{1+z'}{H^3(z')} dz' \tag{6}$$

Where

$$H(z)^2 = H_0^2(\Omega_m(1+z)^3 + \Omega_\Lambda) \tag{7}$$

In order to avoid having to integrate up to ∞ we will be substituting $z = \frac{1}{a} - 1$. This gives us the following equations:

$$D(a) = \frac{5\Omega_m H_0^2}{2} H(a) \int_0^a \frac{1}{a^3 H^3(a')} da' \tag{8}$$

Where

$$H(a)^2 = H_0^2(\frac{\Omega_m}{a^3} + \Omega_\Lambda) \tag{9}$$

The resulting value is: $D(1/51) = 0.0196$. The exact number and the way that it was calculated can be found in the print output below.

4.2 Calculating the derivative of the linear growth factor at a given redshift

In order to accomplish this task we had to analytically derive the value of $\dot{D}(t)$. One can calculate this indirectly using the following equation:

$$\dot{D}(t) = \frac{dD}{da}\dot{a} \tag{10}$$

Where

$$\dot{a} = \frac{H_0}{\sqrt{a}} \tag{11}$$

If we use the chain rule we get:

$$\frac{dD}{da} = \frac{5\Omega_m H_0^2}{2}[\frac{dH(a)}{da}I + \frac{dI}{da}H(a)] \tag{12}$$

Where

$$I = \int_0^a \frac{1}{a^3 H(a)^3} da \tag{13}$$

Which gives us:

$$\dot{D}(a) = \frac{5\Omega_m H_0^3}{2\sqrt{a}}[\frac{-3\Omega_m}{2\sqrt{a^5(\Omega_m + \Omega_\Lambda a^3)}} \int_0^a \frac{1}{a^3 H(a)^3} da + \frac{1}{a^3 H(a)^3} H_0 \sqrt{\frac{\Omega_m}{a^3} + \Omega_\Lambda}] \tag{14}$$

The resulting value is: $\dot{D}(1/51) = 1239$ REQUIRE UNITS . The exact number and the way that it was calculated can be found in the print output below.

4.3 Evolution of a volume in 2D

For this exercise we were asked to use the Zeldovich approximation to generate a movie of the evolution of a volume in two dimensions from a scale factor of 0.0025 until a scale factor of 1.0. The movie made for this exercise is called *2D.mp4* and can be found in the directory of this assignment. Besides making the movie we were also asked to plot the position and momentum of the first 10 particles along the y-direction vs a. These can be seen in Figure 9(a) and Figure 9(b).

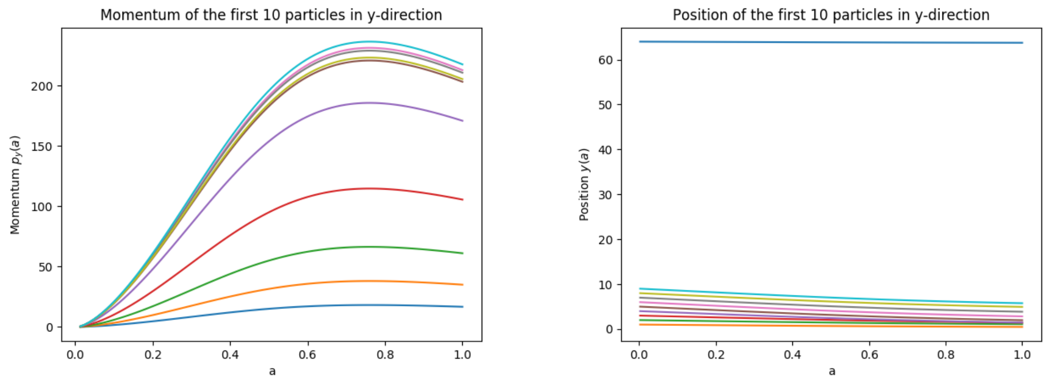

Figure 9: *Left:* Evolution of the momentum of the first 10 particles (in the y-direction). *Right:* Evolution of the y-coordinates of these same 10 particles. Note how we see in the right figure that the outer particles (6 through 10) seem to be moving towards the other particles. This is also reflected in the evolution of the momentum which increases much faster for the outer particles and eventually slows down once they reach 'the rest'.

4.4 Evolution of a volume in 3D

This task was very similar to the previous task except for the fact that we had to make the simulation in 3D. The movies generated for this are named *3D_ xy.mp4*, *3D_ xz.mp4* and *3D_ yz.mp4* for each respective slice. We were also asked to plot the position and momentum of the first 10 particles along the z-direction vs a.

4.5 Scripts

Here we can see the terminal output of the script used for this exercise:

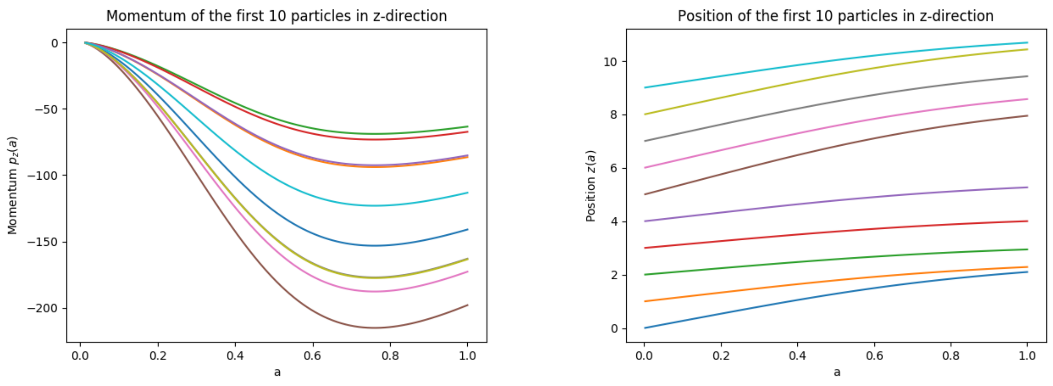

Figure 10: *Left:* Evolution of the momentum of the first 10 particles (in the z-direction). *Right:* Evolution of the y-coordinates of these same 10 particles. Similarly to the particles in the previous exercise, the momentum appears to accurately reflect their movement. More displacement in the spacial coordinates implicates a steeper curve in the momentum graph.

```
--- Exercise 4 ---
Original seed: 627310980
The linear growth factor at z = 50 (a = 1/51) is equal to: 0.01961021426458253
 The analytical value of time derivative of D(z) at z = 50 : <function <lambda> at 0
    x7f7e827e9f28>
 The numerical value of time derivative of D(z) at z = 50 : [499.95564708]
Starting 2D N-body simulation
2D N-body simulation completed
Starting 3D N-body simulation
3D N-body simulation completed
```

a2_4.txt

Here is the script used to produce these results:

```
# a2_4
import numpy as np
import sys
import matplotlib.pyplot as plt
from tqdm import tqdm
from a2_1 import rng, box_muller, romber_int, ridders_diff

def random_field_generator_zeld(N,rng,mu=0,sig=1):
    # Prepares a random field in Fourier space
    #print(f'Generating a random field with n = {n} of dimension {N}x{N} (mu = {mu})')
    ck = np.zeros((N,N),dtype=complex)
    Sx = np.zeros((N,N),dtype=complex)
    Sy = np.zeros((N,N),dtype=complex)
    # Setting values of top half of the field
    for j in range((N//2)+1):
    # Determining the value of k_y
        k_y = j*2*np.pi/N
        for i in range(N):
            # Determining the value of k_x and sigma_x
            if i <= (N//2):
                k_x = (i)*2*np.pi/N
            else:
                k_x = (-N+i)*2*np.pi/N
            # Drawing a random number from normal distrib
            k = (k_x**2+k_y**2)**0.5
            if k == 0:
                k = 1
            rand = box_muller(rng.rand_num(1),rng.rand_num(1),mu,sig)
            ck[j][i] = (rand[0]*k**(-3)) - 1j*(rand[1]*k**(-3))
            Sx[j][i] = ck[j][i]*k_x*1j
            Sy[j][i] = ck[j][i]*k_y*1j
    # Setting values of points who need to equal their own conjugates
    ck[0][0] = 0
    Sx[0][0],Sy[0][0] = 0,0
    ck[0][N//2] = (ck[0][N//2].real)**2
    Sx[0][N//2] = (Sx[0][N//2].real)**2
    Sy[0][N//2] = (Sy[0][N//2].real)**2
    ck[N//2][0] = (ck[N//2][0].real)**2
    Sx[N//2][0] = (Sx[N//2][0].real)**2
    Sy[N//2][0] = (Sy[N//2][0].real)**2
    ck[N//2][N//2] = (ck[N//2][N//2].real)**2
    Sx[N//2][N//2] = (Sx[N//2][N//2].real)**2
    Sy[N//2][N//2] = (Sy[N//2][N//2].real)**2
    # Setting values of bottom half of the field using conjugates
    for j in range((N//2)+1):
        for i in range(N):
            ck[-j][-i]= ck[j][i].conjugate()
            Sx[-j][-i]= Sx[j][i].conjugate()
            Sy[-j][-i]= Sy[j][i].conjugate()
    return Sx,Sy
#end random_field generator()

def random_field_generator_zeld_3D(N,rng,mu=0,sig=1):
    # Prepares a random field in Fourier space
    #print(f'Generating a random field with n = {n} of dimension {N}x{N} (mu = {mu})')
    ck = np.zeros((N,N,N),dtype=complex)
    Sx = np.zeros((N,N,N),dtype=complex)
    Sy = np.zeros((N,N,N),dtype=complex)
    Sz = np.zeros((N,N,N),dtype=complex)

    for l in range(N):
        if l <= (N//2):
            k_z = (l)*2*np.pi/N
        else:
            k_z = (-N+l)*2*np.pi/N
        #print(k_z)
        # Setting values of top half of the field
        for j in range((N//2)+1):
        # Determining the value of k_y
            k_y = j*2*np.pi/N
            for i in range(N):
                # Determining the value of k_x and sigma_x
                if i <= (N//2):
                    k_x = (i)*2*np.pi/N
                else:
                    k_x = (-N+i)*2*np.pi/N
                # Drawing a random number from normal distrib
                k = (k_x**2+k_y**2+k_z**2)**0.5
                if k == 0:
                    k = 1
                rand = box_muller(rng.rand_num(1),rng.rand_num(1),mu,sig)
                ck[l][j][i] = (rand[0]*k**(-3)) - 1j*(rand[1]*k**(-3))
                Sx[l][j][i] = ck[l][j][i]*k_x*1j
                Sy[l][j][i] = ck[l][j][i]*k_y*1j
                Sz[l][j][i] = ck[l][j][i]*k_z*1j
        # Setting values of points who need to equal their own conjugates
        ck[l][0][0] = 0
        Sx[l][0][0],Sy[l][0][0],Sz[l][0][0] = 0,0,0
        ck[l][0][N//2] = (ck[l][0][N//2].real)**2
        Sx[l][0][N//2] = (Sx[l][0][N//2].real)**2
        Sy[l][0][N//2] = (Sy[l][0][N//2].real)**2
        Sz[l][0][N//2] = (Sz[l][0][N//2].real)**2
        ck[l][N//2][0] = (ck[l][N//2][0].real)**2
        Sx[l][N//2][0] = (Sx[l][N//2][0].real)**2
        Sy[l][N//2][0] = (Sy[l][N//2][0].real)**2
        Sz[l][N//2][0] = (Sz[l][N//2][0].real)**2
        ck[l][N//2][N//2] = (ck[l][N//2][N//2].real)**2
        Sx[l][N//2][N//2] = (Sx[l][N//2][N//2].real)**2
        Sy[l][N//2][N//2] = (Sy[l][N//2][N//2].real)**2
        Sz[l][N//2][N//2] = (Sz[l][N//2][N//2].real)**2
        # Setting values of bottom half of the field using conjugates
        for j in range((N//2)+1):
            for i in range(N):
                ck[l][-j][-i]= ck[l][j][i].conjugate()
                Sx[l][-j][-i]= Sx[l][j][i].conjugate()
                Sy[l][-j][-i]= Sy[l][j][i].conjugate()
                Sz[l][-j][-i]= Sz[l][j][i].conjugate()
    return Sx,Sy,Sz
#end random_field generator()

if __name__ == '__main__':
    print('--- Exercise 4 ---')
    seed = 627310980
    rng = rng(seed)
    print('Original seed:',seed)
    # --- 4.a ---
    # Setting the constants
    omega_m = 0.3
    omega_lambda = 0.7
    H0 = 70 # km/s/Mpc
    # Setting the functions
    H = lambda a : H0*((omega_m*(a)**(-3)+omega_lambda))**0.5
    D_prefactor = lambda a : (5*omega_m*H0**2)/2*H(a)
    dIda = lambda a: 1/(a*H(a))**3
    I = lambda a: romber_int(dIda,1e-12,a)
    a = 1/51
    D = lambda a: D_prefactor(a) * I(a)
    print(f'The linear growth factor at z = 50 (a = 1/51) is equal to: {D(a)}')

    # --- 4.b ---
    # Setting the functions
    pre_fact = lambda a: 5*omega_m*H0**3/(2*a**(0.5))
    dHda = lambda a: -3*omega_m/(2*(a**5*(omega_m+omega_lambda*a**3))**0.5)
    dDdt = lambda a: pre_fact(a)*(dHda(a)*I(a)+dIda(a)*H(a))
    dDdt_numerical = ridders_diff(D,np.array([a]))*H0/(a)**0.5
    print(f' The analytical value of time derivative of D(z) at z = 50 : {dDdt}')
    print(f' The numerical value of time derivative of D(z) at z = 50 : {dDdt_numerical}
    ')

    # --- 4.c ---
    print('Starting 2D N-body simulation')
    # Preparing parameters that will be used in both simulations
    N = 64
    a = np.linspace(0.0025,1,90)
    Da = np.zeros(len(a))

    # 2D - Generating S for the x and y dimensions in the Fourier plane
    Sx,Sy = random_field_generator_zeld(N,rng)
    Sx = np.fft.ifft2(Sx).real*N
    Sy = np.fft.ifft2(Sy).real*N

    # Setting the starting coordinates
    q = np.zeros((N,N,2))
    for i in range(len(q)):
        for j in range(len(q)):
            q[i][j] = i,j

    # Preparing values and arrays for the plotting of y vs a
    da = a[1]-a[0]
    p = lambda a,S : -1*(a-da/2)**2*dDdt(a-da/2)*S
    Py = np.zeros((len(a),10))
    Xy = np.zeros((len(a),10))

    # Iterating through all the a values
    x2D = np.zeros((N,N,2))
    for k in tqdm(range(0,90)):
        # Calculating D and D*S
        Da[k] = D(a[k])
        DSx = Da[k]*Sx
        DSy = Da[k]*Sy
        # Calculating the new x positions
        x2D[:,:,0] = (q[:,:,0]+DSx)%N
        x2D[:,:,1] = (q[:,:,1]+DSy)%N
        # Saving for momentum plot
        Xy[k] = x2D[0,:10,1]
        Py[k] = p(a[k],Sy[0,:10])
        # Plotting
        plt.scatter(x2D[:,:,0],x2D[:,:,1],marker='.')
        plt.title('2D N-body simulation')
        plt.ylabel('Mpc')
        plt.xlabel(f'a = {np.round(a[k],3)}')
        plt.savefig('./plots/2Dmovie/snap%04d.png'%k)
        plt.close()

    plt.plot(a,Py)
    plt.xlabel('a')
    plt.ylabel('Momentum $p_y(a)$')
    plt.title('Momentum of the first 10 particles in y-direction')
    plt.savefig('./plots/4a.png')
    plt.close()

    plt.plot(a,Xy)
    plt.xlabel('a')
    plt.ylabel('Position $y(a)$')
    plt.title('Position of the first 10 particles in y-direction')
    plt.savefig('./plots/4b.png')
    plt.close()

    print('2D N-body simulation completed')

    # --- 4.d ---
    print('Starting 3D N-body simulation')

    # 3D - Generating S for the x and y dimensions in the Fourier plane
    Sx,Sy,Sz = random_field_generator_zeld_3D(64,rng)
    Sx = np.fft.ifftn(Sx).real*N**2
    Sy = np.fft.ifftn(Sy).real*N**2
    Sz = np.fft.ifftn(Sz).real*N**2

    # Setting the starting coordinates
    q = np.zeros((N,N,N,3))
    for i in range(N):
        for j in range(N):
            for k in range(N):
                q[i][j][k] = i,j,k

    # Preparing arrays for recording of momentum
    Pz = np.zeros((len(a),10))
    Xz = np.zeros((len(a),10))

    # Iterating through all the a values
    x3D = np.zeros((N,N,N,3))
    for k in tqdm(range(0,90)):
        # Calculating D and D*S
        Da[k] = D(a[k])
        DSx = Da[k]*Sx
        DSy = Da[k]*Sy
        DSz = Da[k]*Sz
        # Calculating the new x positions
        x3D[:,:,:,0] = (q[:,:,:,0]+DSx)%N
        x3D[:,:,:,1] = (q[:,:,:,1]+DSy)%N
        x3D[:,:,:,2] = (q[:,:,:,2]+DSz)%N
        # Saving for momentum plot
        Xz[k] = x3D[0,0,:10,2]
        Pz[k] = p(a[k],Sz[0,0,:10])
        # Producing the slices that will be plotted
        xy = x3D[(x3D[:,:,:,2]>31.5) & (x3D[:,:,:,2]<=32.5)]
        xz = x3D[(x3D[:,:,:,1]>31.5) & (x3D[:,:,:,1]<=32.5)]
        yz = x3D[(x3D[:,:,:,0]>31.5) & (x3D[:,:,:,0]<=32.5)]
        # Plotting
        plt.scatter(xy[:,0],xy[:,1],marker='.')
        plt.title('3D N-body simulation: xy')
```

```
            plt.ylabel('Mpc')
            plt.xlabel(f'a = {np.round(a[k],3)}')
            plt.savefig('./plots/3Dmovie/xy/snap%04d.png'%k)
            plt.close()

            plt.scatter(xz[:,0],xz[:,2],marker='.')
            plt.title('3D N-body simulation: xz')
            plt.ylabel('Mpc')
            plt.xlabel(f'a = {np.round(a[k],3)}')
            plt.savefig('./plots/3Dmovie/xz/snap%04d.png'%k)
            plt.close()

            plt.scatter(yz[:,1],yz[:,2],marker='.')
            plt.title('3D N-body simulation: yz')
            plt.ylabel('Mpc')
            plt.xlabel(f'a = {np.round(a[k],3)}')
            plt.savefig('./plots/3Dmovie/yz/snap%04d.png'%k)
            plt.close()

    plt.plot(a,Pz)
    plt.xlabel('a')
    plt.ylabel('Momentum $p_z(a)$')
    plt.title('Momentum of the first 10 particles in z-direction')
    plt.savefig('./plots/4c.png')
    plt.close()

    plt.plot(a,Xz)
    plt.xlabel('a')
    plt.ylabel('Position $z(a)$')
    plt.title('Position of the first 10 particles in z-direction')
    plt.savefig('./plots/4d.png')
    plt.close()

    print('3D N-body simulation completed')
```

a2_4.py

5 Mass assignment schemes

The aim of this assignment was to generate a particle mesh that could be used to calculate a density mesh which could then be used to calculate the gravitational forces in our simulation.

5.1 Nearest Gird Point method

The most simple way to approach this problem is the to add the mass of each particle to the grid point that is closest to it (hence the name 'Nearest Grid Point method'). My 3D implementation of this method generated the mesh visible in Figure 11.

5.2 Testing robustness

In order to test the robustness of the method we were asked to make a plot of the x position of an individual particle and the value in cell 4 and 0 in 1 dimension (where x varied from the lowest to the highest possible value. In order to accomplish this a 1-D version of the algorithm was coded up (see code below). The resulting graph can be seen in Figure 12.

5.3 The Cloud in Cell method

In the Cloud in Cell method the mass of the particle is distributed among the nearest cells according to how close they are to the particle. The nearer the cell is to a particle, the higher the percentage of the mass assigned to the cell. The implementation of this code can be found below. The z-slices produced using this method may be seen in Figure 13. Finally its robustness is assessed using the plot found in Figure 14.

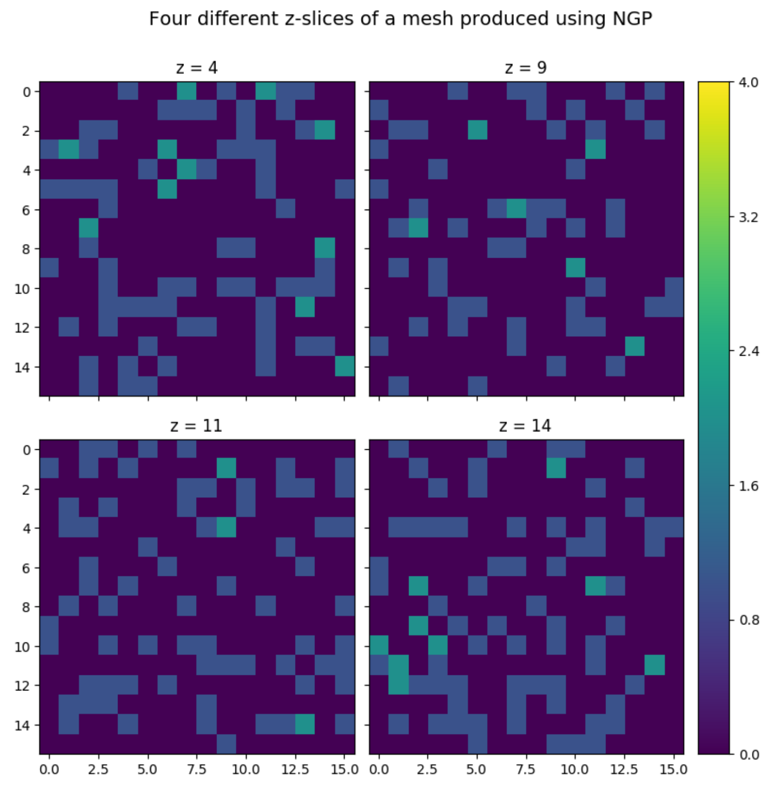

Figure 11: Four different z-slices of the mesh that was produced using the Nearest Grid Point method.

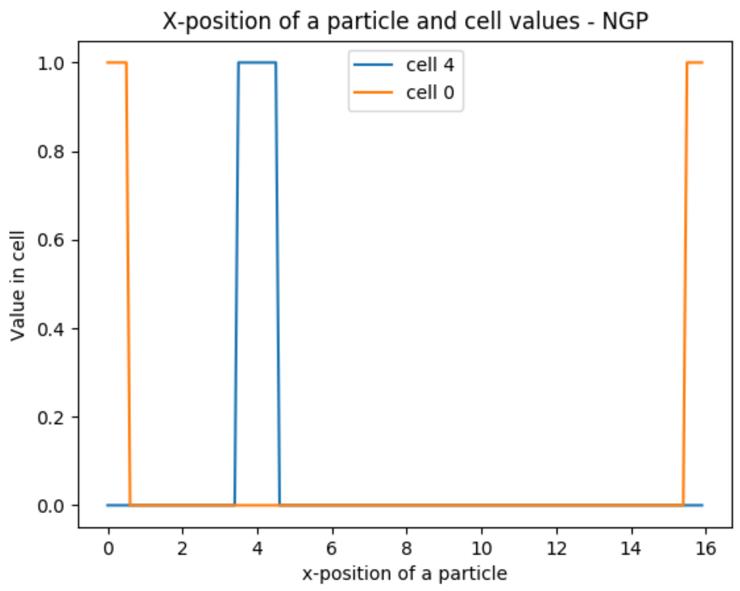

Figure 12: Here we see the values of cell 4 and cell 0 as a function of the position of a particle along the x-axis. Note how the cell takes on a value of 1 once the particle is within 0.5 cell sizes of it's center. This indicates that the code is working properly.

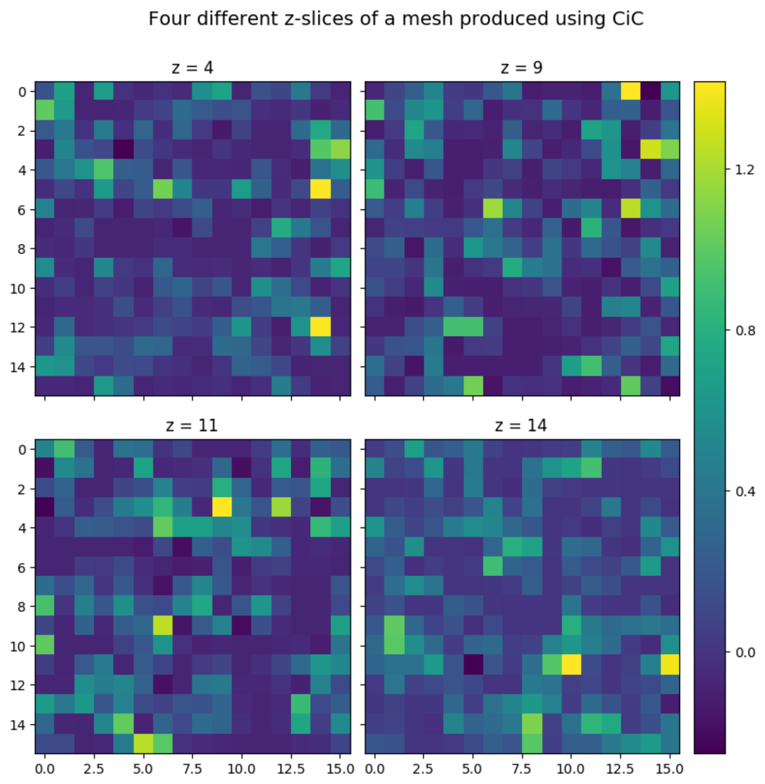

Figure 13: Four different z-slices of the mesh that was produced using the Cloud in Cell method. Note how the mass in this mesh is far more spread out than in the one produced using the NGP method.

5.4 1D Fast Fourier Transform Algorithm

For this exercise the Cooley-Tukey algorithm was used. In Figure 15 we can see that the 'self-written' algorithm produces the same values as the numpy.fft package and the analytical values.

5.5 2D and 3D Fast Fourier Transform Algorithms

The next task was to generalize the FFT to 2 and 3 dimensions. The 2D transform was then to be tested by comparing it with the analytical FFT of the same function. These can be seen in Figure 16. For the 3D FFT we were to make a plot of a 3D multivariate Gaussian function and plot the 3 slices centered at the center for the three different slice options $x-y, x-z$ and $y-z$. These can be seen in Figure 17.

5.6 Calculating the potential for the given particles

For this exercise we were given an equation which we could use to calculate the potential for the given particles. Implementing this required to following thee steps:

1. Taking the Fourier transform of the δ, where δ is the density distribution (the calculated mesh).
2. Dividing the result by k^2.
3. Applying the inverse Fourier transform.

The implementation of these steps may be found in the code below. The resulting slices can be seen in Figure **??** and Figure **??**.

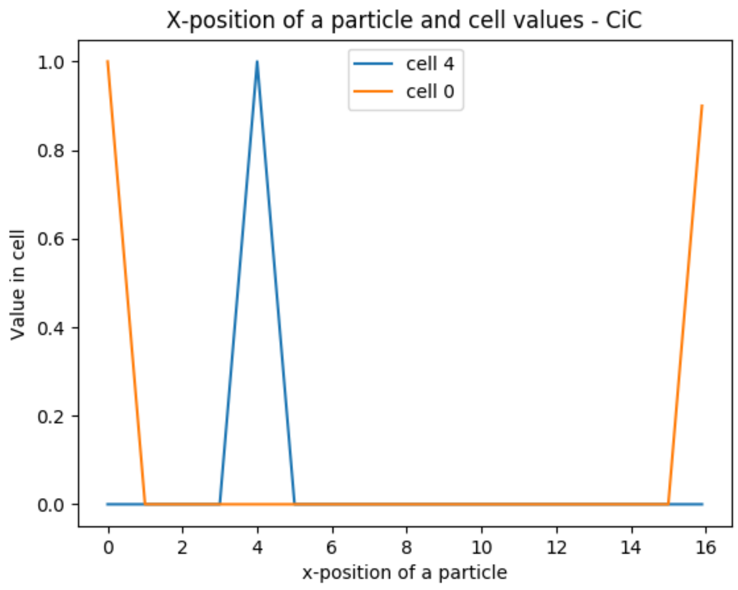

Figure 14: Here we see the values of cell 4 and cell 0 as a function of the position of a particle along the x-axis. Note how the cells gradually take on more and more of the percentage of the mass of the particle, peaking at 1 when the particle is located at the exact same spot as the cell. After this the value gradually drops back down to 0. This indicates that the code is working as intended.

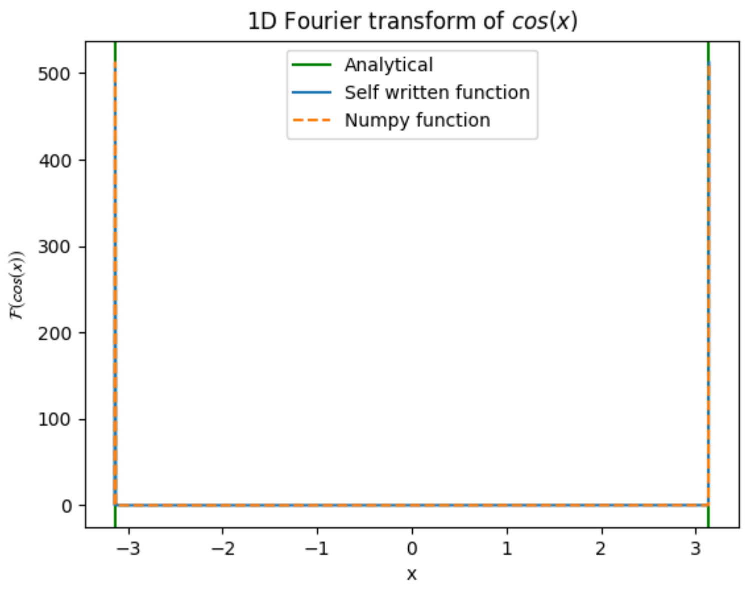

Figure 15: Plot of the Fourier transform of a $cos(t)$ function as calculated with three different methods. Note how they all appear to give the same values.

5.7 Calculating the potential gradient for the particles

Now that we had the potential field, we could use it to calculate the gradient of the potential for the first 10 particles. In order to implement this, the gradient was calculated using the central difference method. Afterwards the potential was assigned to the particles using a sort of 'reverse CiC' algorithm. The code can once again be found below as well as the output of the gradients values for the 10 first particles.

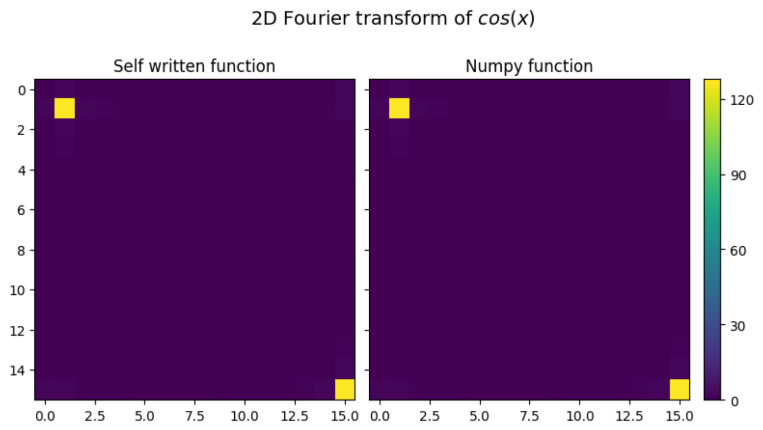

Figure 16: Plot of the 2D transform.

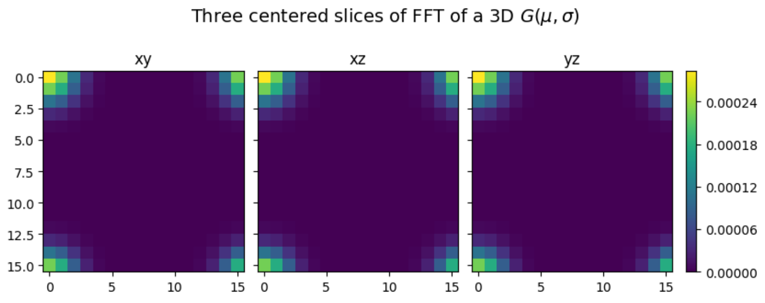

Figure 17: Slices of the 3D transform of a multivariate Gaussian.

5.8 Scripts

Here we can see the terminal output of the script used for this exercise:

```
--- Exercise 5 ---
Original seed: 121
Potential gradient output:
[x,y,z]
[5.08808058e-01 3.08650779e-04 1.68811888e-01]
[ 0.23984587 -0.08231249 -0.32767697]
[-0.0138786   0.01586179 -0.24372081]
[ 0.21567265 -0.10735527  0.29234552]
[ 0.15196136 -0.05566411 -0.08047141]
[0.20694978 0.03244246 0.05488023]
[ 0.11346534 -0.10504644 -0.11604878]
[ 0.00781276 -0.21064205 -0.24514963]
[-0.1521923   0.00329643  0.33940659]
[-0.13377353 -0.11253401 -0.049315   ]
```

a2_5.txt

Here is the script used to produce these results:

```
#a2_5.py
import numpy as np
import sys
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid
```

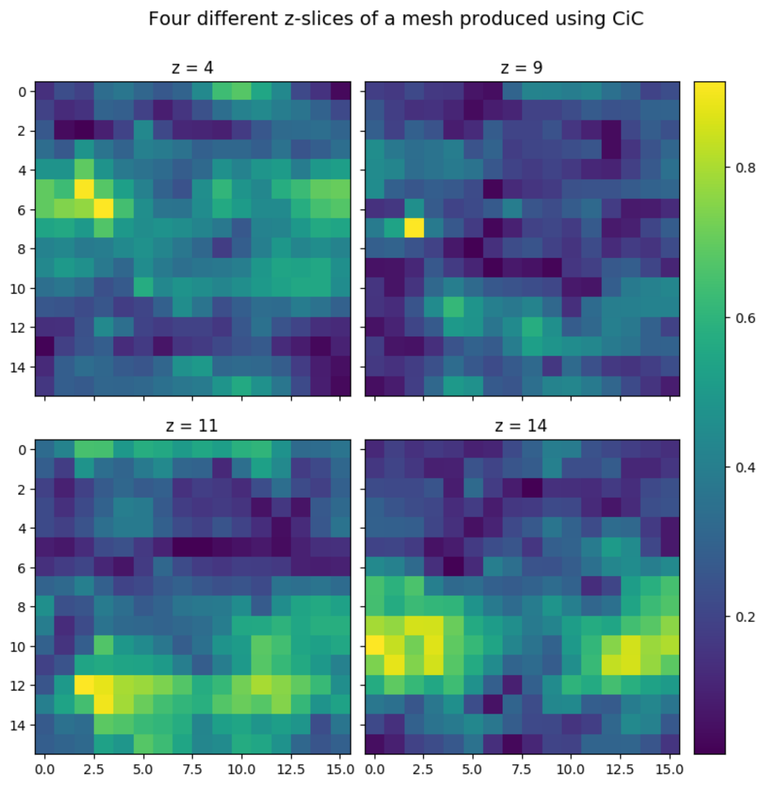

Figure 18: Plot of the 2D transform of $cos(x, y)$ function.

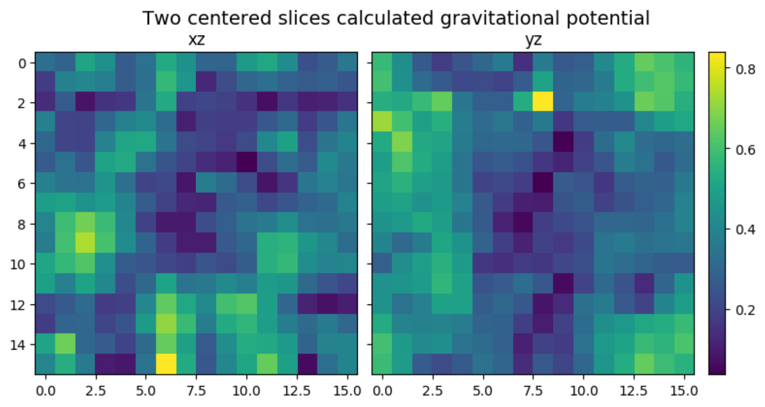

Figure 19: Slices of the 3D transform of a multivariate Gaussian.

```

def NGP(p,N):

    mesh = np.zeros((N,N,N))
    for i in range(len(p[0])):
        x,y,z = np.round(p[:,i])%N
        mesh[int(x)][int(y)][int(z)] += 1

    return mesh
```

```

def CiC(p,N):

    mesh = np.zeros((N,N,N))
    for i in range(len(p[0])):
        w = np.zeros(8)
        x,y,z = np.round(p[:,i])
        dx,dy,dz = (x-p[0,i]),(y-p[1,i]),(z-p[2,i])
        #print(x,p[0,i],y,p[1,i],z,p[2,i])
        #print('delta:',dx,dy,dz)
        sx,sy,sz = np.sign(dx),np.sign(dy),np.sign(dz)
        dx,dy,dz = np.abs(dx),np.abs(dy),np.abs(dz)
        x,y,z = x%N,y%N,z%N
        # Calculating all of the weights
        w[0] = (1-dx)*(1-dy)*(1-dz)
        w[1] = (dx)*(1-dy)*(1-dz)
        w[2] = (1-dx)*(dy)*(1-dz)
        w[3] = (1-dx)*(1-dy)*(dz)
        w[4] = (dx)*(dy)*(dz-1)
        w[5] = (dx)*(1-dy)*(dz)
        w[6] = (1-dx)*(dy)*(dz)
        w[7] = (dx)*(dy)*(dz)
        #print(w)
        # Assigning the weights
        mesh[np.int(x)%N][np.int(y)%N][np.int(z)%N] += w[0]
        mesh[np.int(x-sx)%N][np.int(y)%N][np.int(z)%N] += w[1]
        mesh[np.int(x)%N][np.int(y-sy)%N][np.int(z)%N] += w[2]
        mesh[np.int(x)%N][np.int(y)%N][np.int(z-sz)%N] += w[3]
        mesh[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z)%N] += w[4]
        mesh[np.int(x-sx)%N][np.int(y)%N][np.int(z-sz)%N] += w[5]
        mesh[np.int(x)%N][np.int(y-sy)%N][np.int(z-sz)%N] += w[6]
        mesh[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z-sz)%N] += w[7]

    return mesh

def CiC_reverse(p,gx,gy,gz,N):

    p_out = np.zeros((p.shape))

    for i in range(len(p[0])):
        w = np.zeros(8)
        x,y,z = np.round(p[:,i])
        dx,dy,dz = (x-p[0,i]),(y-p[1,i]),(z-p[2,i])
        #print(x,p[0,i],y,p[1,i],z,p[2,i])
        #print('delta:',dx,dy,dz)
        sx,sy,sz = np.sign(dx),np.sign(dy),np.sign(dz)
        dx,dy,dz = np.abs(dx),np.abs(dy),np.abs(dz)
        x,y,z = x%N,y%N,z%N
        # Calculating all of the weights
        w[0] = (1-dx)*(1-dy)*(1-dz)
        w[1] = (dx)*(1-dy)*(1-dz)
        w[2] = (1-dx)*(dy)*(1-dz)
        w[3] = (1-dx)*(1-dy)*(dz)
        w[4] = (dx)*(dy)*(dz-1)
        w[5] = (dx)*(1-dy)*(dz)
        w[6] = (1-dx)*(dy)*(dz)
        w[7] = (dx)*(dy)*(dz)
        #print(w)
        # Assigning the weights
        # (I am aware of the fact that this is extremely ugly coding, would be the first
 that I would fix if I had more time to spend on this)
        p_out[:,i] += (gx[np.int(x)%N][np.int(y)%N][np.int(z)%N]* w[0],gy[np.int(x)%N][
    np.int(y)%N][np.int(z)%N]* w[0],gz[np.int(x)%N][np.int(y)%N][np.int(z)%N]* w[0])
        p_out[:,i] += (gx[np.int(x-sx)%N][np.int(y)%N][np.int(z)%N]* w[1],gy[np.int(x-sx
    )%N][np.int(y)%N][np.int(z)%N]* w[1],gz[np.int(x-sx)%N][np.int(y)%N][np.int(z)%N]* w
    [1])
        p_out[:,i] += (gx[np.int(x)%N][np.int(y-sy)%N][np.int(z)%N]* w[2],gy[np.int(x)%N
    ][np.int(y-sy)%N][np.int(z)%N]* w[2],gz[np.int(x)%N][np.int(y-sy)%N][np.int(z)%N]* w
    [2])
        p_out[:,i] += (gx[np.int(x)%N][np.int(y)%N][np.int(z-sz)%N]* w[3],gy[np.int(x)%N
][np.int(y)%N][np.int(z-sz)%N]* w[3],gz[np.int(x)%N][np.int(y)%N][np.int(z-sz)%N]* w[3])
        p_out[:,i] += (gx[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z)%N]* w[4],gy[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z)%N]* w[4],gz[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z)%N]* w[4])
        p_out[:,i] += (gx[np.int(x-sx)%N][np.int(y)%N][np.int(z-sz)%N]* w[5],gy[np.int(x-sx)%N][np.int(y)%N][np.int(z-sz)%N]* w[5],gz[np.int(x-sx)%N][np.int(y)%N][np.int(z-sz)%N]* w[5])
        p_out[:,i] += (gx[np.int(x)%N][np.int(y-sy)%N][np.int(z-sz)%N]* w[6],gy[np.int(x)%N][np.int(y-sy)%N][np.int(z-sz)%N]* w[6],gz[np.int(x)%N][np.int(y-sy)%N][np.int(z-sz)%N]* w[6])
        p_out[:,i] += (gx[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z-sz)%N]* w[7],gy[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z-sz)%N]* w[7],gz[np.int(x-sx)%N][np.int(y-sy)%N][np.int(z-sz)%N]* w[7])

    return p_out


def fft1D(x,Nj,x0=0,step=1,inv=False):
    if inv:
        j2 = 2j
    else:
        j2 = -2j
    if Nj == 1:
        #print('Reached bottom',[x[x0]])
        return [x[x0]]
    new_step = step*2
    hNj = Nj//2
    rs = fft1D(x,hNj,x0,new_step,inv=inv)+fft1D(x,hNj,x0+step,new_step,inv=inv)
    rs_new = np.copy(rs)
    for i in range(hNj):
        rs[i],rs[i+hNj]=rs[i]+np.exp(j2*np.pi*i/Nj)*rs[i+hNj],rs[i]-np.exp(j2*np.pi*i/Nj)*rs[i+hNj]
    return rs

def fft2D(x,inv=False):

    x = np.array(x,dtype=complex)
    if len(x.shape) == 2:
        for i in range(x.shape[1]):
            x[:,i] = fft1D(x[:,i],len(x[1]),inv=inv)
        for j in range(x.shape[0]):
            x[j] = fft1D(x[j],len(x[0]),inv=inv)
        return x

def fft3D(x,inv=False):

    x = np.array(x,dtype=complex)

    for k in range(x.shape[2]):
        for i in range(x.shape[1]):
            x[k,:,i] = fft1D(x[k,:,i],len(x[1]),inv=inv)
        for j in range(x.shape[0]):
            x[k][j] = fft1D(x[k][j],len(x[0]),inv=inv)

    for i in range(x.shape[1]):
        for j in range(x.shape[0]):
            x[:,i,j] = fft1D(x[:,i,j],len(x[2]),inv=inv)

    return x

def central_diff_3D(a):
    # Setting constants and array
    N = len(a[0])
    gradx = np.zeros((a.shape))
    grady = np.zeros((a.shape))
    gradz = np.zeros((a.shape))
    # Running through all values in array
    for i in range(N):
        for j in range(N):
            for k in range(N):
                gradx[i][j][k] = a[(i+1)%N][j][k]-a[(i-1)%N][j][k]
                grady[i][j][k] = a[i][(j+1)%N][k]-a[i][(j-1)%N][k]
                gradz[i][j][k] = a[i][j][(k+1)%N]-a[i][j][(k-1)%N]
    return gradx,grady,gradz



if __name__ == '__main__':
    print('--- Exercise 5 ---')

    # --- 5.a ---
    print('Original seed:',121)
    np.random.seed(121)
    N = 16
    positions = np.random.uniform(low=0,high=16,size=(3,1024))
    # Calculating the mesh
    mesh_ngp = NGP(positions,N)
    vmax = np.max(mesh_ngp)
    # Plotting the mesh
    fig = plt.figure(1,(30,30))
    grid = AxesGrid(fig, 142,
                    nrows_ncols=(2, 2),
                    axes_pad=(0.15,0.45),
                    share_all=True,
                    label_mode="L",
                    cbar_location="right",
                    cbar_mode="single",
                    )
    im = grid[0].imshow(mesh_ngp[:,:,3],vmin=0, vmax=vmax)
    grid[0].set_title('z = 4')
    im = grid[1].imshow(mesh_ngp[:,:,8],vmin=0, vmax=vmax)
    grid[1].set_title('z = 9')
    im = grid[2].imshow(mesh_ngp[:,:,10],vmin=0, vmax=vmax)
    grid[2].set_title('z = 11')
    im = grid[3].imshow(mesh_ngp[:,:,13],vmin=0, vmax=vmax)
    grid[3].set_title('z = 14')
    grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
            cax.toggle_label(True)

    fig.suptitle('Four different z-slices of a mesh produced using NGP',x=0.38,y=0.64,
    fontsize=14)
    fig.tight_layout()
    plt.savefig('./plots/5a.png',bbox_inches='tight',pad_inches = 0.5)
    plt.close()

    # --- 5.b ---
    test_points = np.arange(0,16,0.1)
    cell4 = np.zeros(len(test_points))
    cell0 = np.zeros(len(test_points))

    # 1-D implementation of the NGP method
    for i in range(len(test_points)):
        mesh1d = np.zeros(N)
        x = np.round(test_points[i])%N
        mesh1d[int(x)] += 1
        cell4[i] = mesh1d[4]
        cell0[i] = mesh1d[0]
    # Plotting
    plt.plot(test_points,cell4,label='cell 4')
    plt.plot(test_points,cell0,label='cell 0')
    plt.xlabel('x-position of a particle')
    plt.ylabel('Value in cell')
    plt.title('X-position of a particle and cell values - NGP')
    plt.legend()
    plt.savefig('./plots/5b.png')
    plt.close()

    # --- 5.c ---
    # Calculating the mesh
    mesh = CiC(positions,16)
    vmax = np.max(mesh)
    fig = plt.figure(1,(30,30))
    grid = AxesGrid(fig, 142,
                    nrows_ncols=(2, 2),
                    axes_pad=(0.15,0.45),
                    share_all=True,
                    label_mode="L",
                    cbar_location="right",
                    cbar_mode="single",
                    )

    im = grid[0].imshow(mesh[:,:,4])#,vmin=0, vmax=vmax)
    grid[0].set_title('z = 4')
    im = grid[1].imshow(mesh[:,:,9])#,vmin=0, vmax=vmax)
    grid[1].set_title('z = 9')
    im = grid[2].imshow(mesh[:,:,11])#,vmin=0, vmax=vmax)
    grid[2].set_title('z = 11')
    im = grid[3].imshow(mesh[:,:,14])#,vmin=0, vmax=vmax)
    grid[3].set_title('z = 14')
    grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
            cax.toggle_label(True)
    fig.suptitle('Four different z-slices of a mesh produced using CiC',x=0.38,y=0.64,
    fontsize=14)
    fig.tight_layout()
    plt.savefig('./plots/5c.png',bbox_inches='tight',pad_inches = 0.5)
    plt.close()

    # 1-D implementation of the CiC method
    cell4 = np.zeros(len(test_points))
    cell0 = np.zeros(len(test_points))
    for i in range(len(test_points)):
        w = np.zeros(2)
        mesh1d = np.zeros(N)
        x = np.round(test_points[i])
        dx = x-test_points[i]
        sx = np.sign(dx)
        dx = np.abs(dx)
        x=x%N
        w[0] = 1-dx
        w[1] = dx
        mesh1d[np.int(x)%N]+=w[0]
        mesh1d[np.int(x-sx)%N]+=w[1]
        cell4[i] = mesh1d[4]
        cell0[i] = mesh1d[0]

    plt.plot(test_points,cell4,label='cell 4')
    plt.plot(test_points,cell0,label='cell 0')
    plt.xlabel('x-position of a particle')
    plt.ylabel('Value in cell')
    plt.title('X-position of a particle and cell values - CiC')
    plt.legend()
    plt.savefig('./plots/5d.png')
    plt.close()

    # --- 5.d ---
    f = lambda x: np.cos(x)
    x = np.linspace(-np.pi,np.pi,1024)
    fx = f(x)
    fftself = fft1D(fx,len(fx))
    fftnumpy = np.fft.fft(fx)
    plt.axvline(x=np.pi,color='green')
    plt.axvline(x=-np.pi,color='green',label='Analytical')
    plt.plot(x,np.abs(fftself),label='Self written function')
    plt.plot(x,np.abs(fftnumpy),ls='--',label='Numpy function')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('$\mathcal{F(cos(x))}$')
    plt.title('1D Fourier transform of $cos(x)$')
    plt.savefig('./plots/5e.png')
    plt.close()

    # --- 5.e ---
    f2d = lambda x,y: np.cos(x+y)
    N = 16
    x = np.linspace(-3,3,N)
    y = np.linspace(-3,3,N)
    f2dxy = np.zeros((N,N))
    for i in range(N):
        for j in range(N):
            f2dxy[i][j] = f2d(x[i],y[j])

    fftself2 = fft2D(f2dxy)
    fftnumpy2 = np.fft.fft2(f2dxy)

    fig = plt.figure(1,(30,30))
    grid = AxesGrid(fig, 142,
                    nrows_ncols=(1, 2),
                    axes_pad=(0.15,0.45),
                    share_all=True,
                    label_mode="L",
                    cbar_location="right",
                    cbar_mode="single",
                    )

    im = grid[0].imshow(np.abs(fftself2))#,vmin=0, vmax=vmax)
    grid[0].set_title('Self written function')
    im = grid[1].imshow(np.abs(fftnumpy2))#,vmin=0, vmax=vmax)
    grid[1].set_title('Numpy function')
    grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
        cax.toggle_label(True)
    fig.suptitle('2D Fourier transform of $cos(x)$',x=0.38,y=0.58,fontsize=14)
    fig.tight_layout()
    plt.savefig('./plots/5f.png',bbox_inches='tight',pad_inches = 0.5)
    plt.close()

    g3D = lambda x,y,z,mu,sig: 1/(sig*(2*np.pi)**0.5)*np.exp((-(x-mu)**2-(y-mu)**2-(z-mu
)**2)/sig**2)
    N = 16
    x = np.linspace(-3,3,N)
    y = np.linspace(-3,3,N)
    z = np.linspace(-3,3,N)
    f3d = np.zeros((N,N,N))
    for i in range(N):
        for j in range(N):
            for k in range(N):
                f3d[i][j][k] = g3D(x[i],y[j],z[k],0,1)

    fft_f3d = np.abs(fft3D(f3d))

    vmax = np.max(fft_f3d)
    fig = plt.figure(1,(30,30))
    grid = AxesGrid(fig, 142,
                    nrows_ncols=(1, 3),
                    axes_pad=(0.15,0.45),
                    share_all=True,
                    label_mode="L",
                    cbar_location="right",
                    cbar_mode="single",
                    )

    im = grid[0].imshow(fft_f3d[:,:,7])
    grid[0].set_title('xy')
    im = grid[1].imshow(fft_f3d[:,7,:])
    grid[1].set_title('xz')
    im = grid[2].imshow(fft_f3d[7,:,:])
    grid[2].set_title('yz')
```

```python
    grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
            cax.toggle_label(True)
    fig.suptitle('Three centered slices of FFT of a 3D $G(\mu,\sigma)$',x=0.38,y=0.56,
    fontsize=14)
    fig.tight_layout()
    plt.savefig('./plots/5g.png',bbox_inches='tight',pad_inches = 0.5)
    plt.close()

    # --- 5.f ---
    # Calculating the gravitational potential
    mean = np.mean(mesh)
    mesh_n = (mesh-mean)/mean
    fft_mesh = fft3D(mesh_n)/N**3
    N = 16
    for l in range(N):
        if l <= (N//2):
            k_z = (l)*2*np.pi/N
        else:
            k_z = (-N+l)*2*np.pi/N
        for j in range(N):
            if i <= (N//2):
                k_y = (j)*2*np.pi/N
            else:
                k_y = (-N+j)*2*np.pi/N
            for i in range(N):
                if i <= (N//2):
                    k_x = (i)*2*np.pi/N
                else:
                    k_x = (-N+i)*2*np.pi/N
                # Calculating k
                k = (k_x**2+k_y**2+k_z**2)**0.5
                if k == 0:
                    k = 1
                fft_mesh[i][j][l] = fft_mesh[i][j][l]*k**(-2)
    grav_p = fft3D(fft_mesh,inv=True)
    grav_p = np.abs(grav_p)

    vmax = np.max(grav_p)
    fig = plt.figure(1,(30,30))
    grid = AxesGrid(fig, 142,
                    nrows_ncols=(2, 2),
                    axes_pad=(0.15,0.45),
                    share_all=True,
                    label_mode="L",
                    cbar_location="right",
                    cbar_mode="single",
                    )

    im = grid[0].imshow(grav_p[:,:,0])#,vmin=0, vmax=vmax)
    grid[0].set_title('z = 4')
    im = grid[1].imshow(grav_p[:,:,8])#,vmin=0, vmax=vmax)
    grid[1].set_title('z = 9')
    im = grid[2].imshow(grav_p[:,:,10])#,vmin=0, vmax=vmax)
    grid[2].set_title('z = 11')
    im = grid[3].imshow(grav_p[:,:,13])#,vmin=0, vmax=vmax)
    grid[3].set_title('z = 14')
    grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
            cax.toggle_label(True)
    fig.suptitle('Four different z-slices of a mesh produced using CiC',x=0.38,y=0.64,
    fontsize=14)
    fig.tight_layout()
    plt.savefig('./plots/5h.png',bbox_inches='tight',pad_inches = 0.5)
    plt.close()

    fig = plt.figure(1,(30,30))
    grid = AxesGrid(fig, 142,
                    nrows_ncols=(1, 2),
                    axes_pad=(0.15,0.45),
                        share_all=True,
                        label_mode="L",
                        cbar_location="right",
                        cbar_mode="single",
                        )

    im = grid[0].imshow(grav_p[:,7,:])#,vmin=0, vmax=vmax)
    grid[0].set_title('xz')
    im = grid[1].imshow(grav_p[7,:,:])#,vmin=0, vmax=vmax)
    grid[1].set_title('yz')
    grid.cbar_axes[0].colorbar(im)
    for cax in grid.cbar_axes:
            cax.toggle_label(True)
    fig.suptitle('Two centered slices calculated gravitational potential',x=0.38,y=0.57,
    fontsize=14)
    fig.tight_layout()
    plt.savefig('./plots/5i.png',bbox_inches='tight',pad_inches = 0.5)
    plt.close()

    # --- 5.g ---
    # Calculating the gradients for the 3 dimensions
    gradx,grady,gradz = central_diff_3D(grav_p)
    # Calculating the potential for each position
    positions_grad = CiC_reverse(positions[:,:10],gradx,grady,gradz,16)
    print('Potential gradient output:')
    print('[x,y,z]')
    for i in range(len(positions_grad[0])):
        print(positions_grad[:,i])
```

a2_5.py

6 Classifying γ-ray bursts

Using the given dataset we were asked to use logistic regression to make a model of the data, using a binary classification for short (0) or long (1) GRB's.

In order to accomplish this, the data first had to be cleaned up a bit. We threw out all the rows that were not classified as GRB's in the first place. We dealt with the missing data by simply setting those values to 0 (instead of -1). The time (T90) was used in order to produce the labels for the data, but was also thrown out once this was done. The remaining data was then split up into a training and a test set (80% and 20% respectively). In Figure 20 we can see the results. After 4000 epochs we reached an accuracy of 69.3% on the training set, 70.5% on the test set and 69.5% on the entire data set. Training for more epochs would result in a drop of accuracy in the test set, meaning that the models was probably over fitting at that point.

6.1 Scripts

Here we can see the terminal output of the script used for this exercise:

```
--- Exercise 5 ---
Seed: 627310980
Iterations:4000
Training set
Accuracy:69.3%

Test set
Accuracy:70.5%

Entire data set
Accuracy:69.5%
```

a2_6.txt

Here is the script used to produce these results:

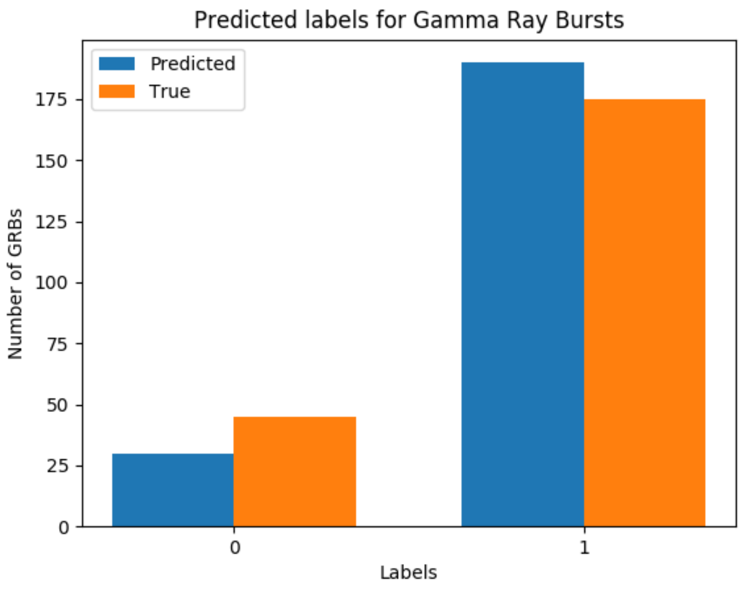

Figure 20: These are the results of the classifier after 4000 training epochs. Label 0 represents the short bursts while 1 represents the long bursts.

```python
#a2_6.py
import numpy as np
import sys
import matplotlib.pyplot as plt
import os
from a2_1 import rng,box_muller


def train_perceptron (data_in,data_out,rng):
    bias = np.zeros ((len(data_in),1)) # Initialise bias array.
    data_in = np.append(data_in,bias,axis=1) # Merge data_in and weights.
    weights = np.array([rng.rand_num(len(data_in[0])),rng.rand_num(len(data_in[0]))]).
    reshape(len(data_in[0]),2) # initialise random weights
    weighted_sum = np.dot(data_in,weights) # Compute weighted sum.
    output_indices = np.argmax(weighted_sum,axis=1) # Select maximum value.

    epoch = 0


    while epoch < 4000:
        wrongly_classified_indices = []
        for i in range(len(data_in)):
            if output_indices[i] != data_out[i]:
                wrongly_classified_indices.append(i)

        rand = rng.rand_num(len(wrongly_classified_indices)-1)*10
        k = wrongly_classified_indices[int(rand[0])]

        for j in range (2):
            if weighted_sum[k][j] > weighted_sum[k][int(data_out[k])]:
                weights[:,j] -= data_in[k]
            if j == int ( data_out[k] ):
                weights[:,j] += data_in[k]

        weighted_sum = np.dot(data_in,weights)
        output_indices = np.argmax(weighted_sum,axis = 1)

        epoch += 1

    sys.stdout.write("Iterations:{0}\n".format(epoch))

    #plt.show() # Toggle

    return weights

def test_perceptron(data_in,data_out,weights,hist=False):
    bias = np.zeros ((len(data_in ),1)) # Initialise bias array.
    data_in = np.append(data_in,bias,axis=1) # Merge data_in and weights.
    weighted_sum = np.dot(data_in, weights)
    output_indices = np.argmax(weighted_sum,axis = 1)
    correct_counter = 0
    for i in range (len(data_in)):
        if output_indices[i] == data_out[i]:
            correct_counter += 1
    print('Accuracy:{:03.1f}%\n'.format(correct_counter*100/len(data_out)))

    if hist:
        short_true = len(data_out[data_out==0])
        short_predicted = len(output_indices[output_indices==0])
        long_true = len(data_out[data_out==1])
        long_predicted = len(output_indices[output_indices==1])
        bar_width = 0.35
        plt.bar(np.array([0,1])-bar_width/2,np.array([short_predicted,long_predicted]),
    bar_width,label='Predicted')
        plt.bar(np.array([0,1])+bar_width/2,np.array([short_true,long_true]),bar_width,
    label='True')
        plt.xticks((0,1))
        plt.xlabel('Labels')
        plt.ylabel('Number of GRBs')
        plt.legend()
        plt.title('Predicted labels for Gamma Ray Bursts')
        #plt.show()
        plt.savefig('./plots/6.png')
        plt.close()

if __name__ == '__main__':
    print('--- Exercise 6 ---')

    seed = 627310980
    print('Seed:',seed)
    rng = rng(seed)

    filename = 'GRBs.txt'
    url = 'https://home.strw.leidenuniv.nl/~nobels/coursedata/'
    if not os.path.isfile(filename):
        print(f'File not found, downloading {filename}')
        os.system('wget '+url+filename)

    data = np.genfromtxt(filename,skip_header=2,usecols = (2,3,4,5,6,7))
    data[data==-1.0] = 0
    names = np.genfromtxt(filename,skip_header=2,usecols=0,dtype=str)
    data = data[names!='XRF']
    labels = np.zeros(len(data))
    labels[data[:,1]>=10] += 1
    data = data[:,[0,2,3,4,5]]
    train_percent = 0.8
    train_in = data[:int(len(data)*train_percent)]
    train_out = labels[:int(len(labels)*train_percent)]
    test_in = data[int(len(data)*train_percent):]
    test_out = labels[int(len(labels)*train_percent):]

    for i in range(1):
        #sys.stdout.write("Run {0}\n".format(i+1))
        weights = train_perceptron(train_in, train_out,rng)
        print('Training set')
        test_perceptron(train_in,train_out,weights)
        print('Test set')
        test_perceptron(test_in,test_out,weights)
        print('Entire data set')
        test_perceptron(data, labels, weights, hist=True)
```

a2_6.py

7 Building a quadtree

For this exercise our goal was to build a Barnes-Hut quadtree with at most 12 particles per leaf node. When writing the code for this exercise I based my class object structure on the one used in this online example: *https://kpully.github.io/Quadtrees/*. The particles and their corresponding nodes are plotted in Figure 21.

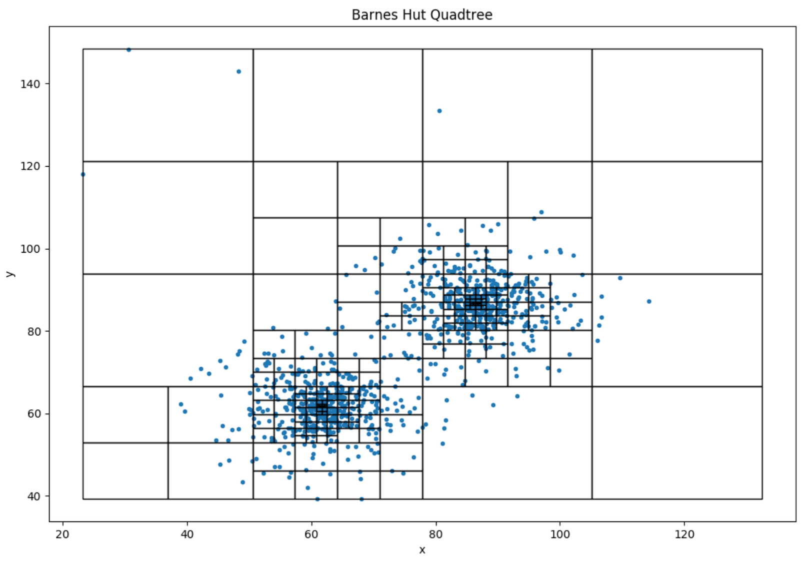

Figure 21: In this figure we can see the Barnes Hut Quadtree built for the given dataset.

7.1 Scripts

Here we can see the terminal output of the script used for this exercise:

```
--- Exercise 7 ---
```

a2_7.txt

Here is the script used to produce these results:

```
#a2_7.py
import numpy as np
import sys
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import h5py
import os


class Point():
    def __init__(self,coordinates):
        self.x = coordinates[0]
        self.y = coordinates[1]
        #self.type = o_type

```

```python
class Node():
    def __init__(self,center,height,width,points):
        self.center = center
        self.height = height
        self.width = width
        self.points = points
        self.children = []

    def height(self):
        return self.height
    def width(self):
        return self.width
    def points(self):
        return self.points
    def print_all(self):
        print(f'center:{self.center},height:{self.height},width:{self.width}')

class Tree():
    def __init__(self,threshold,data):
        self.threshold = threshold

        # Make the point objects and store in list
        self.points = []
        for i in range(len(data)):
            self.points.append(Point(data[i]))
        # Determine the dimensions of the original box
        # We are assuming that the box is square for now
        dx = np.max(data[:,0])-np.min(data[:,0])+0.1
        dy = np.max(data[:,1])-np.min(data[:,1])+0.1
        self.d = np.max((dx,dy))
        center = (np.min(data[:,0])+self.d/2,np.min(data[:,1])+self.d/2)
        # Make the root node
        self.root = Node(center,self.d,self.d,self.points)
        # Now we are going to build the tree:
        builder(self.root,self.threshold)

    def get_points(self):
        return self.points

    def graph(self):

        fig = plt.figure(figsize=(12, 8))
        ax = fig.add_subplot(111)
        root = self.root
        # Root:
        c = find_children(self.root)
        for n in c:
            x0,y0 = n.center[0]-n.width/2,n.center[1]-n.height/2
            ax.add_patch(patches.Rectangle((x0,y0), n.width, n.height, fill=False))
        x = [point.x for point in self.points]
        y = [point.y for point in self.points]
        plt.scatter(x, y, marker='.')
        plt.title('Barnes Hut Quadtree')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.savefig('./plots/7.png')
        plt.close()
        return

def find_children(node):
    if not node.children:
        return [node]
    else:
        children = []
        for child in node.children:
            children += (find_children(child))
    return children

def builder(parent,threshold):
    subdivision(parent,threshold)
    for i in range(len(parent.children)):
        builder(parent.children[i],threshold)

def subdivision(node,threshold):
    if len(node.points) <= threshold:
        return
    dx,dy = node.width/2,node.height/2
    c1 = node.center[0]+dx/2,node.center[1]+dy/2
    p1 = point_selector(c1,dx,dy,node.points)
    n1 = Node(c1,dx,dy,p1)
    #print(f'Found {len(p1)} points top right')
    c2 = node.center[0]-dx/2,node.center[1]+dy/2
    p2 = point_selector(c2,dx,dy,node.points)
    n2 = Node(c2,dx,dy,p2)
    #print(f'Found {len(p2)} points top left')
    c3 = node.center[0]-dx/2,node.center[1]-dy/2
    p3 = point_selector(c3,dx,dy,node.points)
    n3 = Node(c3,dx,dy,p3)
    #print(f'Found {len(p3)} points bottom left')
    c4 = node.center[0]+dx/2,node.center[1]-dy/2
    p4 = point_selector(c4,dx,dy,node.points)
    n4 = Node(c4,dx,dy,p4)
    #print(f'Found {len(p4)} points bottom right')

    node.children = [n1,n2,n3,n4]

def point_selector(center,dx,dy,points):
    xmin,xmax = center[0]-dx/2,center[0]+dx/2
    ymin,ymax = center[1]-dy/2,center[1]+dy/2
    p = []
    #print(f'Box dim: {np.round(xmin,2)} < x < {np.round(xmax,2)}, {np.round(ymin,2)} <
    y < {np.round(ymax,2)} ')
    for i in points:
        if i.x >= xmin and i.y >= ymin and i.x < xmax and i.y < ymax:
            p.append(i)
    return p

if __name__ == '__main__':
    print('--- Exercise 7 ---')

    filename = 'colliding.hdf5'
    url = 'https://home.strw.leidenuniv.nl/~nobels/coursedata/'
    if not os.path.isfile(filename):
        print(f'File not found, downloading {filename}')
        os.system('wget '+url+filename)

    f = h5py.File(filename,'r')
    #print(list(f.keys()))
    a_group_key = list(f.keys())[1]
    data_type4 = f['PartType4']['Coordinates']
    data_type4 = data_type4[:,:2]

    t = Tree(12,data_type4)
    t.graph()
```

a2_7.py