NUR Assignment 2

Christiaan van Buchem - s1587064

May 24, 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.

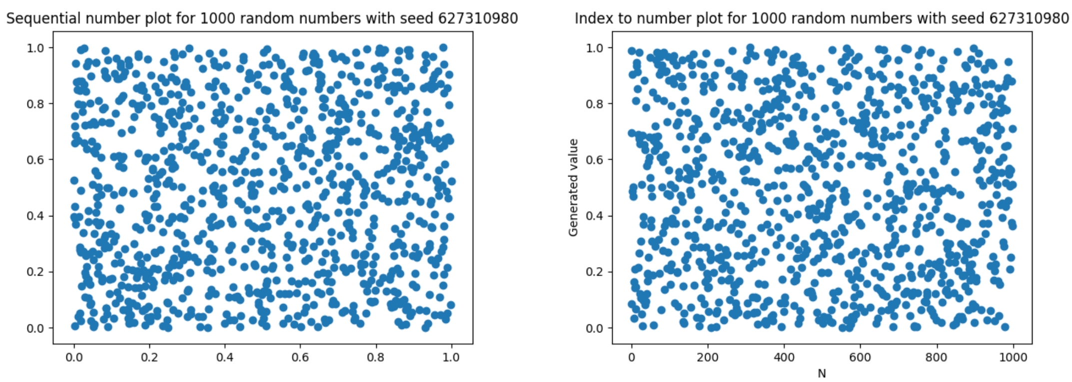

Figure 1: *Left:* Sequential number plot showing that it appears that each number is independent of its predecessor. *Right:* Index to number plot showing that there does not appear to be a relation between the index of a number and its value.

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:

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

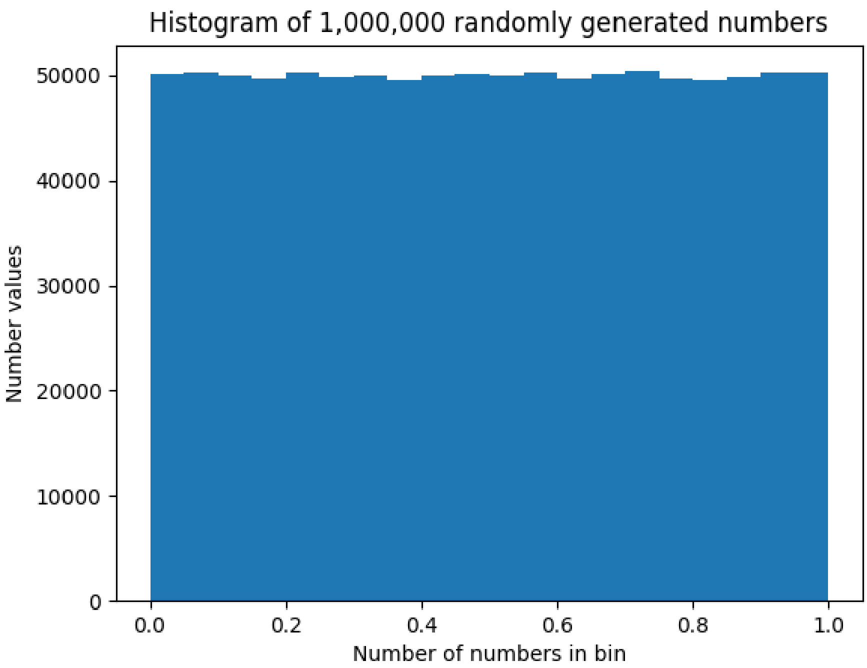

Figure 2: This histogram places the random number generator under a sharper knife, allowing us to see that there are some fluctuations between the bins. Overal it appears to be quite unbiased.

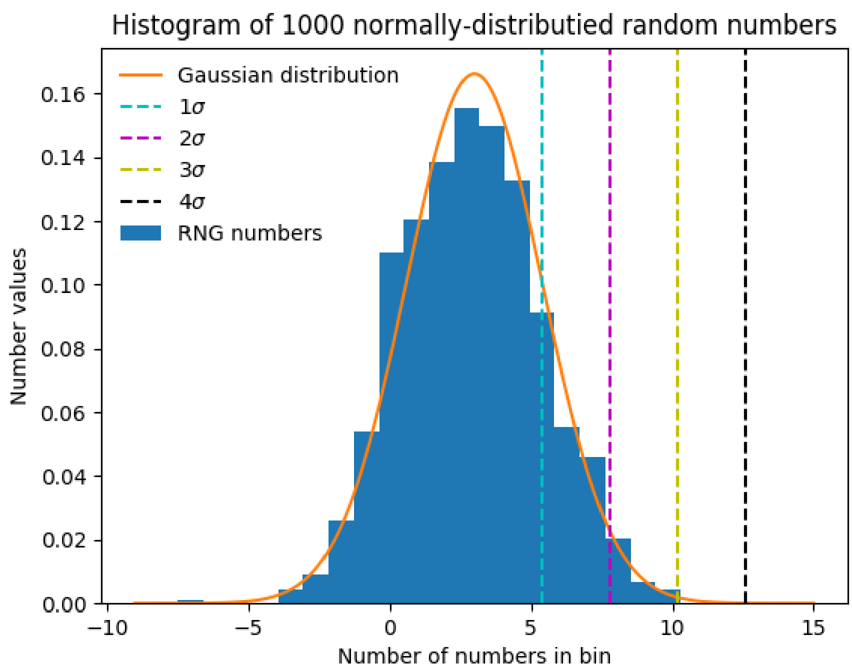

Figure 3: In this figure we can see that numbers generated using the Box-Muller method do indeed follow the Gaussian distribution.

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.

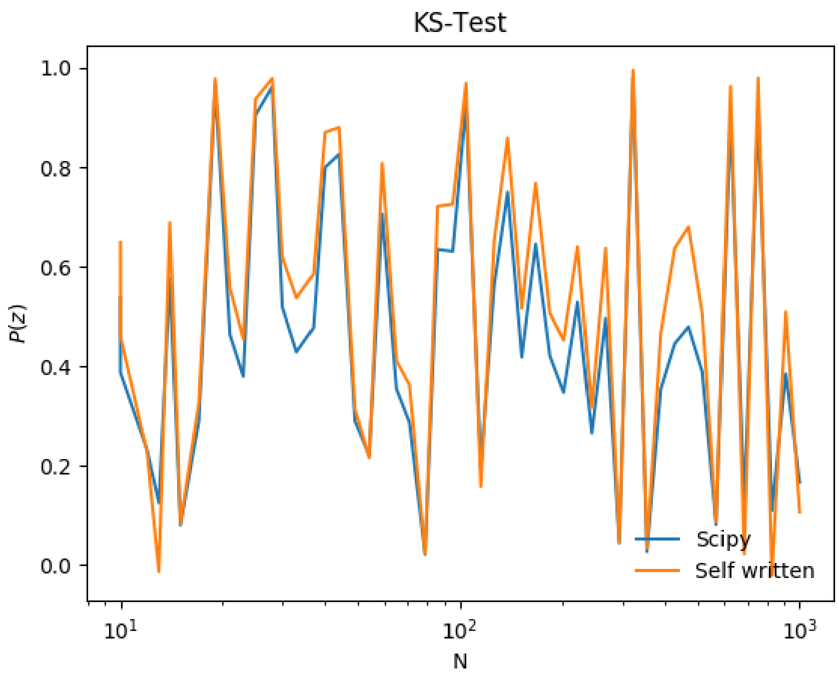

Figure 4: Here we see that the 'self-written' KS-test follows the Scipy KS-test results almost exactly.

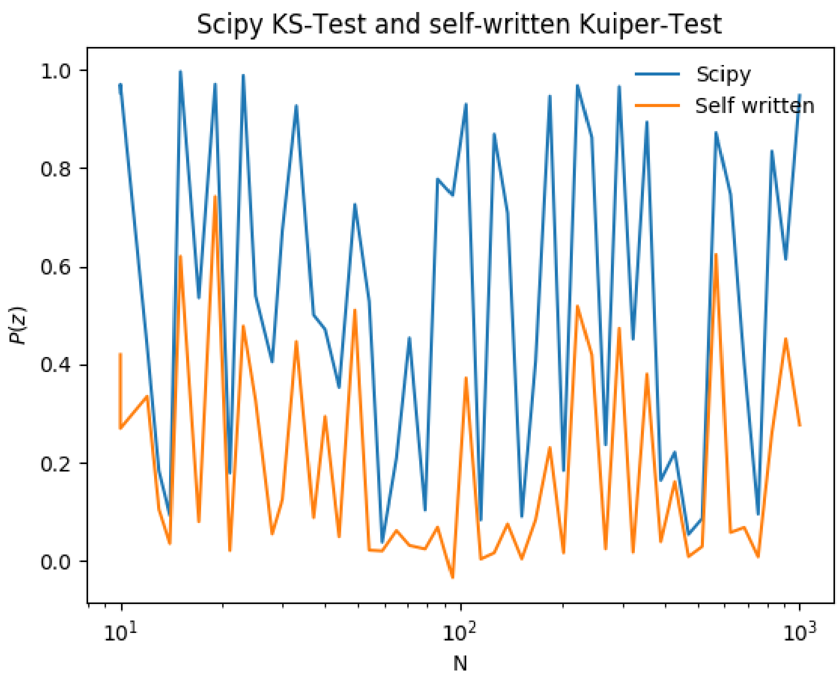

Figure 5: Here we compare the Kuipers test.

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.

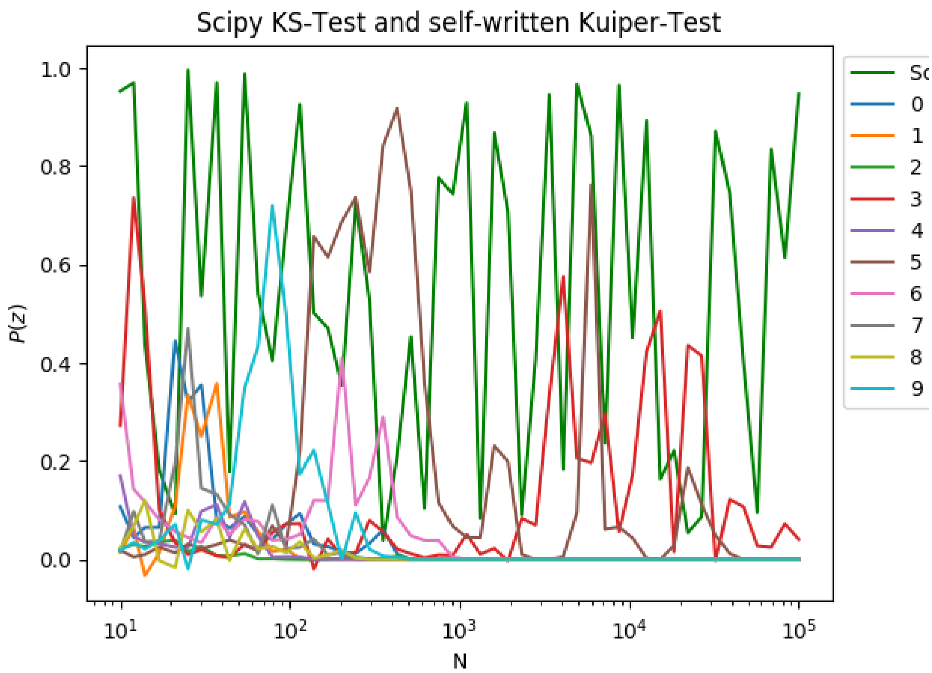

Figure 6: Analysing the different datasets.

1.6 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
```

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

# --- 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):
```

```python
            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
    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)
        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)
        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):
```

```python
        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 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(1000),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.plot(n,P,label='Self written')
    plt.title('KS-Test')
    plt.ylabel('$P(z)$')
    plt.xlabel('N')
    plt.xscale('log')
    plt.legend(loc = 'lower right',frameon=False)
    plt.savefig('plots/1e.png')
    plt.close()
    print('Generated plots/1e.png')

    #---1.d---
    # Kuipers test
    # Preparing arrays
    kuip_P,kuip_P_s = np.zeros(len(n)),np.zeros(len(n))
    kuip_d,kuip_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)
        kuip_d[i],kuip_P[i] = KS_Kuip_test(rand[0],gauss,mu,sig,Kuip=True)
        kuip_d_s[i],kuip_P_s[i] = stats.kstest(rand[0],'norm')
    # Plotting
    plt.plot(n,kuip_P_s,label='Scipy')
    plt.plot(n,kuip_P,label='Self written')
    plt.title('Scipy KS-Test and self-written Kuiper-Test')
    plt.ylabel('$P(z)$')
    plt.xlabel('N')
    plt.xscale('log')
    plt.legend(loc = 'upper right',frameon=False)
    plt.savefig('plots/1f.png')
    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
    plt.plot(n,kuip_P_s,label='Scipy (KS)',color = 'g')
    for i in range(10):
        plt.plot(n,test_P[i],label = i)
    plt.title('Scipy KS-Test and self-written Kuiper-Test')
    plt.ylabel('$P(z)$')
    plt.xlabel('N')
    plt.xscale('log')
    plt.legend(loc=2, bbox_to_anchor=(1,1))
    plt.savefig('plots/1g.png')
    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.

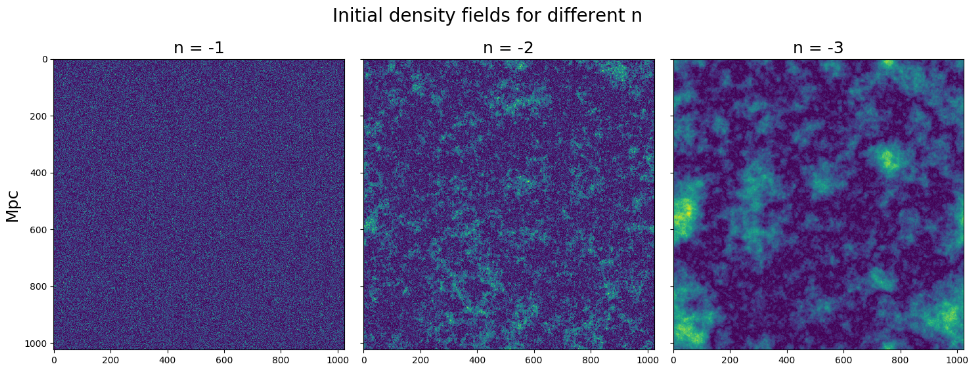

Figure 7: Gaussian random fields for different n values. Notice the clear presence of larger structure when the spectrum is more peaked (lower n).

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)
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
```

```python
    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:

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

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:

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

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.

	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.

In Figure 8 we can see the numerical and analytical solutions for the 3 different cases.
Mention why they do not match.

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)
```

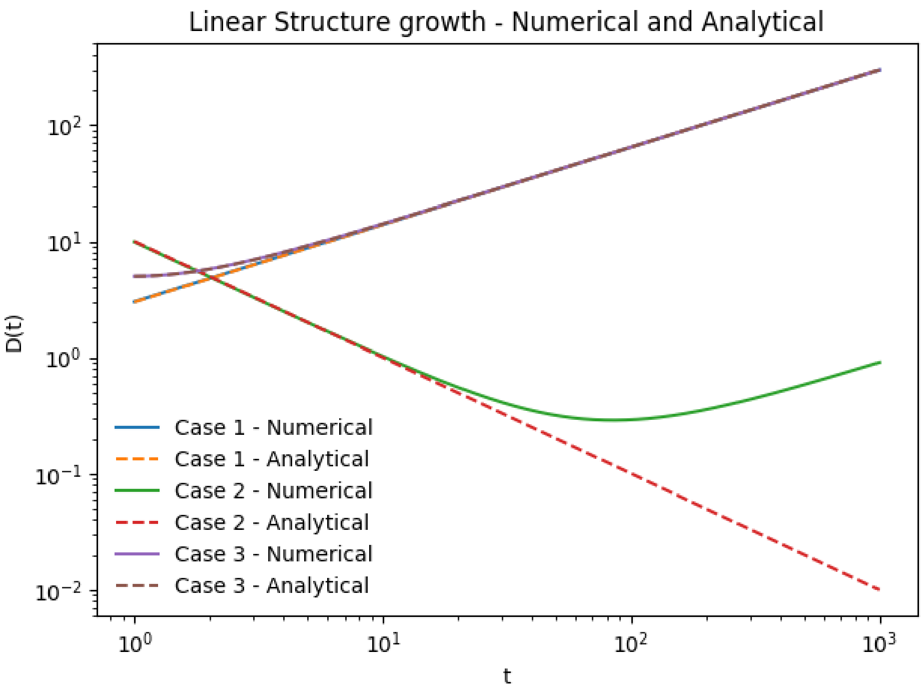

Figure 8: Analytical and numerical solutions to the partial differential equations given in this question.

```
    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