

NUR Assignment 2

Christiaan van Buchem - s1587064

May 23, 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.

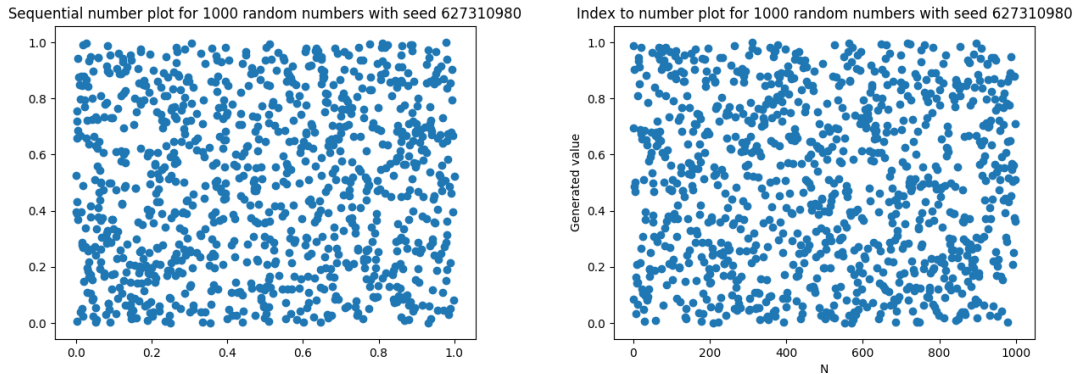


Figure 1: *Left*: Sequential number plot showing that it appears that each number is independent of its predecessor. *Right*: Index to number plot showing that there does not appear to be a relation between the index of a number and its value.

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:

$$P_{KS}(z) \approx \begin{cases} \frac{\sqrt{2\pi}}{z} [(e^{-\pi^2/(8z^2)}) + (e^{-\pi^2/(8z^2)})^9 + (e^{-\pi^2/(8z^2)})^{25}], & (z < 1.18) \\ 1 - 2[(e^{-2z^2}) - (e^{-2z^2})^4 + (e^{-2z^2})^9], & (z \geq 1.18) \end{cases}$$

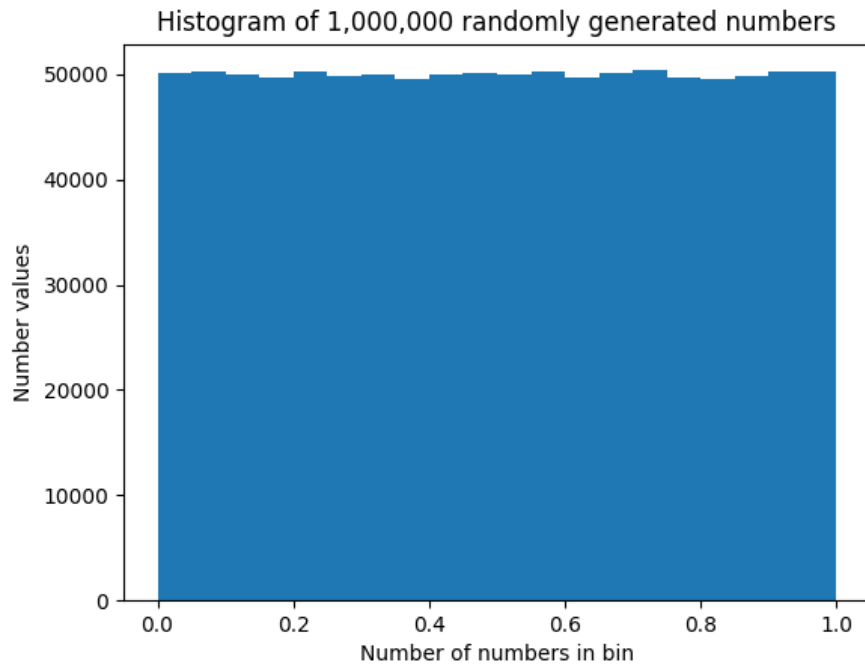


Figure 2: This histogram places the random number generator under a sharper knife, allowing us to see that there are some fluctuations between the bins. Overall it appears to be quite unbiased.

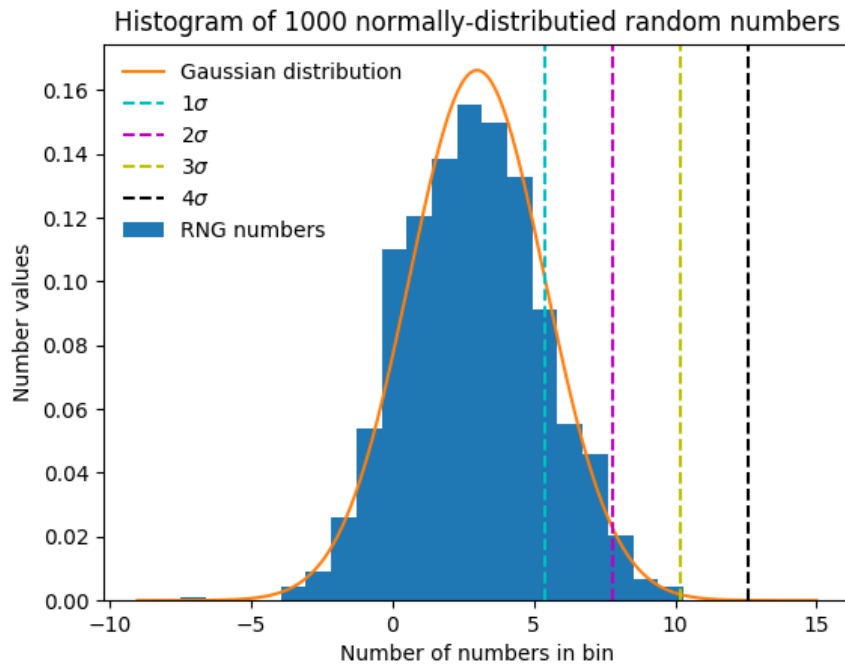


Figure 3: In this figure we can see that numbers generated using the Box-Muller method do indeed follow the Gaussian distribution.

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.

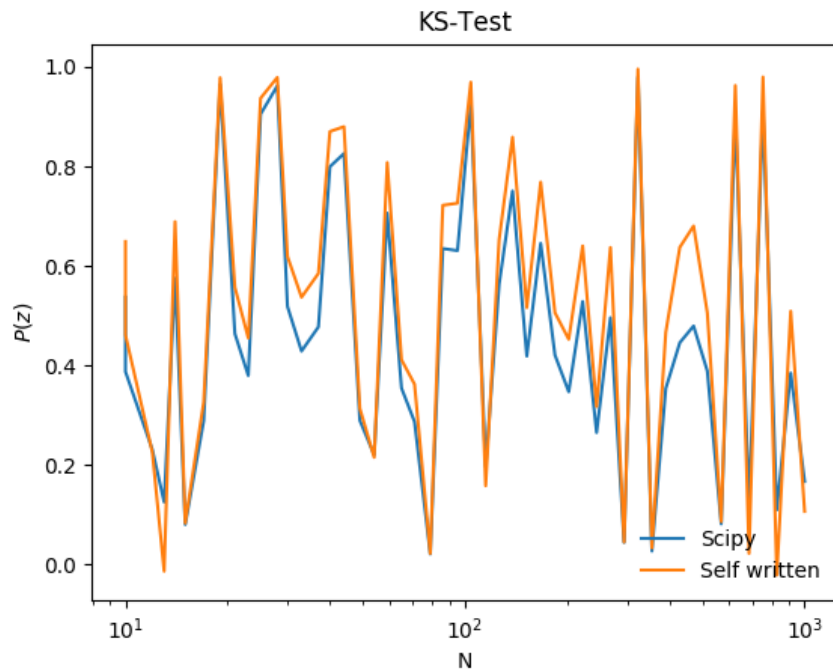


Figure 4: Here we see that the 'self-written' KS-test follows the Scipy KS-test results almost exactly.

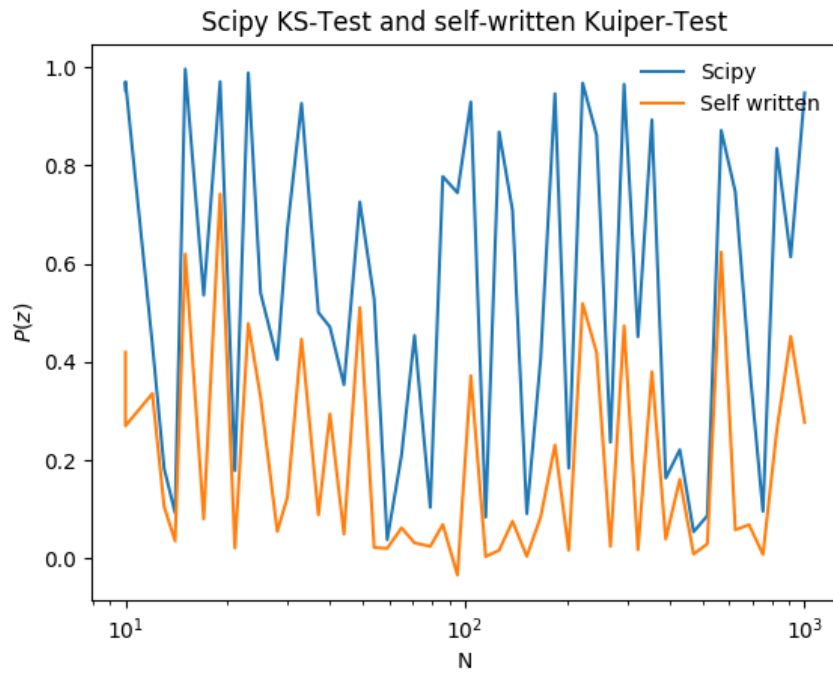


Figure 5: Here we compare the Kuipers test.

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.

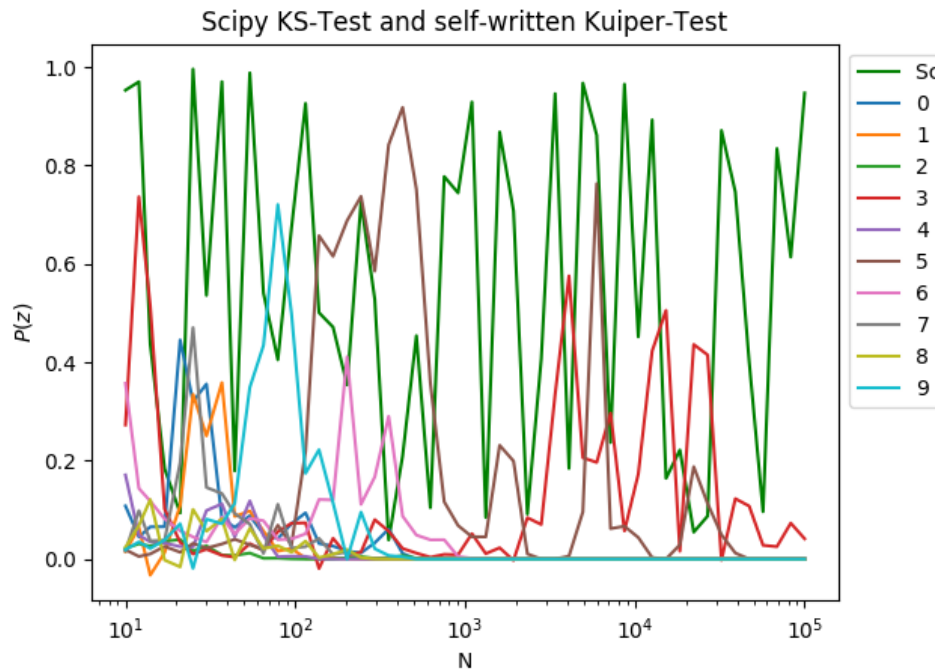


Figure 6: Analysing the different datasets.

1.6 Scripts

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

```

1  — Exercise 1 —
2  Original seed: 627310980
3  Generated plots/1a.png
4  Generated plots/1b.png
5  Generated plots/1c.png
6  Generated plots/1d.png
7  Generated plots/1e.png
8  Generated plots/1f.png
9  Generated plots/1g.png

```

a2.1.txt

Here is the script used to produce these results:

```

1  # a2.1
2  import numpy as np
3  import sys
4  from matplotlib import pyplot as plt
5  from scipy import stats
6  import os
7
8  # — Functions and classes —
9
10 class rng(object):
11     # Rng object that is initiated with a give seed
12     a1,a2,a3 = np.int64(21),np.int64(35),np.int64(4)
13     a = 4294957665
14
15
16     def __init__(self, seed):
17         self.state = np.int64(seed)
18
19     def MWC(self):
20         # Multiply with carry generator

```

```

21         x = np.int64(self.state)
22         self.state = self.a*(x&(2**32-1))+(x>>32)
23
24     def XOR_shift(self):
25         # XOR-shift generator
26         x = np.int64(self.state)
27         x = x ^ x >> self.a1
28         x = x ^ x << self.a2
29         x = x ^ x >> self.a3
30         self.state = np.int64(x)
31     #end XOR_shift()
32
33     def rand_num(self, l, min=0, max=1):
34         # Generates 'l' random numbers between min and max
35         output = []
36         for i in range(l):
37             self.XOR_shift()
38             self.MWC()
39             self.XOR_shift()
40             output.append(self.state)
41         output = np.array(output)/sys.maxsize
42         return min+(output*(max-min))
43     #end rand_num()
44 #end rng()
45
46 def box_muller(u1, u2, mu, sigma):
47     # Implementation of the Box Muller transform
48     x1 = (-2*np.log(u1))*0.5*np.sin(2*np.pi*u2)
49     x2 = (-2*np.log(u1))*0.5*np.cos(2*np.pi*u2)
50     return x1*sigma+mu, x2*sigma+mu
51 #end box_muller
52
53 def central_diff(f, h, x):
54     # Calculates the central difference\n",
55     return (f(x+h)-f(x-h))/(2*h)
56 #end central_diff()
57
58 def ridders_diff(f, x):
59     #Differentiates using Ridder's method
60     m = 10
61     D = np.zeros((m, len(x)))
62     d = 2
63     h = 0.001
64     for i in range(m):
65         D_new = D
66         for j in range(i+1):
67             if j == 0:
68                 D_new[j] = central_diff(f, h, x)
69             else:
70                 D_new[j] = (d**(2*(j+1))*D[j-1]-D_new[j-1])/(d**(2*(j+1))-1)
71         D = D_new
72         h = h/d
73     return D[m-1]
74 #end ridders_diff()
75
76 def comp_trapezoid(f, a, b, n):
77     # Composite trapezoid rule used in romber_int()
78     h = 1/(2**(n-1))*(b-a)
79     sum = 0
80     for i in range(1, 2**(n-1)):
81         sum += f(a+i*h)
82     return (h/2.)*(f(a)+2*sum+f(b))
83 #end comp_trapezoid()
84
85 def romber_int(f, a, b):
86     # Integrates from a to b up to an accuracy of 6 decimals
87     for n in range(1, 10):
88         S_new = np.zeros((n))
89         S_new[0] = comp_trapezoid(f, a, b, n)
90         for j in range(2, n+1):

```

```

91         S_new[j-1] = (4**j-1)*S_new[j-2]-S[j-2])/(4**j-1-1)
92     S = S_new
93     if n > 3:
94         if abs(S[-2]-S[-1]) < 1e-6:
95             return S[-1]
96     return S[-1]
97 #end romber_int()
98
99 def KS_Kuip_test(sample, f, mu, sig, Kuip=False):
100     # Implementation of the Kalgorov-Smirnov test
101     N = len(sample)
102     x = np.linspace(mu-5*sig, mu+5*sig, 1000)
103     F, Fn = np.zeros(len(x)), np.zeros(len(x))
104     Dmin = 0
105     Dmax = 0
106     for i in range(len(Fn)):
107         Fn[i] = len(np.where(sample<=x[i])[0])/N
108         F[i] = romber_int(f, x[0], x[i])
109         Dn = F[i] - Fn[i]
110         if Dn > Dmin:
111             Dmin = Dn
112         Dn = Fn[i] - F[i]
113         if Dn > Dmax:
114             Dmax = Dn
115     # Determine the manner in which D is calculated
116     if Kuip:
117         D = Dmin+Dmax
118     else:
119         D = np.max((Dmin, Dmax))
120     # Calculate the probability
121     z = (N**0.5+0.12+0.11*N**(-0.5))*D
122     if z < 1.18:
123         P = (2*np.pi)**0.5*((np.exp(-1*np.pi**2/(8*z**2)))+(np.exp(-1*np.pi**2/(8*z**2))
124         )**9+(np.exp(-1*np.pi**2/(8*z**2)))**25)
125         return D, 1-P
126     else:
127         P = 1-2*((np.exp(-2*z**2))-(np.exp(-2*z**2))**4+(np.exp(-2*z**2))**9)
128         return D, 1-P
129 #end KS_test()
130
131 def random_field_generator(n, N, rng, mu=0):
132     # Prepares a random field in Fourier space
133     print(f'Generating a random field with n = {n} of dimension {N}x{N} (mu = {mu})')
134     df = np.zeros((N, N), dtype=complex)
135     # Setting values of top half of the field
136     for j in range((N//2)+1):
137         # Determining the value of k_y
138         k_y = j*2*np.pi/N
139         for i in range(N):
140             # Determining the value of k_x and sigma_x
141             if i <= (N//2):
142                 k_x = (i)*2*np.pi/N
143             else:
144                 k_x = (-N+i)*2*np.pi/N
145             # Avoid dividing by 0
146             if i != 0 or j != 0:
147                 sig = ((k_x**2+k_y**2)**0.5)**(n/2)
148             else:
149                 sig = 0
150             # Drawing a random number from normal distrib
151             #df[j][i] = np.random.normal(0, sig)+ 1j*np.random.normal(0, sig)
152             rand = box_muller(rng.randnum(1), rng.randnum(1), mu, sig)
153             df[j][i] = rand[0] + 1j*rand[1]
154     # Setting values of points who need to equal their own conjugates
155     df[0][0] = 0
156     df[0][N//2] = (df[0][N//2].real)**2
157     df[N//2][0] = (df[N//2][0].real)**2
158     df[N//2][N//2] = (df[N//2][N//2].real)**2
159     # Setting values of bottom half of the field using conjugates
160     for j in range((N//2)+1):

```

```

160         for i in range(N):
161             df[-j][-i]= df[j][i].conjugate()
162         return df
163 #end random_field generator()
164
165 # --- Commands, prints and plots ---
166 if __name__ == '__main__':
167     print('--- Exercise 1 ---')
168     seed = 627310980
169     rng = rng(seed)
170     print('Original seed:',seed)
171
172     #--- 1.a ---
173     # MWC and XOR-Shift
174     N = 1000
175     rand = rng.rand_num(N)
176     # Sequential number plot
177     plt.scatter(rand[: (len(rand)-1)], rand[1:])
178     plt.title('Sequential number plot for {} random numbers with seed {}'.format(1000,
179 seed))
179     plt.savefig('plots/1a.png')
180     plt.close()
181     print('Generated plots/1a.png')
182     # Index to number plot
183     plt.scatter(np.arange(0,N,1), rand)
184     plt.title('Index to number plot for {} random numbers with seed {}'.format(1000,seed
185 ))
186     plt.xlabel('N')
187     plt.ylabel('Generated value')
188     plt.savefig('plots/1b.png')
189     plt.close()
190     print('Generated plots/1b.png')
191     # Histogram
192     N = 1000000
193     rand = rng.rand_num(N)
194     plt.hist(rand, bins=20, range=(0,1))
195     plt.title('Histogram of 1,000,000 randomly generated numbers'.format(1000,seed))
196     plt.xlabel('Number of numbers in bin')
197     plt.ylabel('Number values')
198     plt.savefig('plots/1c.png')
199     plt.close()
200     print('Generated plots/1c.png')
201
202     #--- 1.b ---
203     # Box-Muller method
204     N = 1000
205     mu, sig = 3,2.4
206     rand = box_muller(rng.rand_num(N), rng.rand_num(N), mu, sig)
207     gauss = lambda x, mu, sig : 1/(2*np.pi*sig**2)**0.5*np.exp(-0.5*(x-mu)**2/sig**2)
208     x = np.linspace(mu-(sig*5), mu+(sig*5), 1000)
209     plt.hist(rand[0], bins=20, label='RNG numbers', density=1)
210     plt.plot(x, gauss(x, mu, sig), label='Gaussian distribution')
211     plt.title('Histogram of {} normally-distributed random numbers'.format(1000))
212     plt.xlabel('Number of numbers in bin')
213     plt.ylabel('Number values')
214     plt.axvline(x=mu+sig, label='$1\sigma$', color='c', linestyle='--')
215     plt.axvline(x=mu+2*sig, label='$2\sigma$', color='m', linestyle='--')
216     plt.axvline(x=mu+3*sig, label='$3\sigma$', color='y', linestyle='--')
217     plt.axvline(x=mu+4*sig, label='$4\sigma$', color='k', linestyle='--')
218     plt.legend(frameon=False)
219     plt.savefig('plots/1d.png')
220     plt.close()
221     print('Generated plots/1d.png')
222
223     #--- 1.c. ---
224     # KS-test
225     # Setting parameters
226     mu, sig = 0,1
227     rand = box_muller(rng.rand_num(N), rng.rand_num(N), mu, sig)
228     gauss = lambda x : 1/(2*np.pi*sig**2)**0.5*np.exp(-0.5*(x-mu)**2/sig**2)

```

```

228 n = np.logspace(np.log10(10), np.log10(1000), dtype=int)
229 # Preparing arrays
230 P, P_s = np.zeros(len(n)), np.zeros(len(n))
231 d, d_s = np.zeros(len(n)), np.zeros(len(n))
232 # Running test for different values of N
233 for i in range(len(n)):
234     rand = box_muller(rng.rand_num(n[i]), rng.rand_num(n[i]), mu, sig)
235     d[i], P[i] = KS_Kuip_test(rand[0], gauss, mu, sig)
236     d_s[i], P_s[i] = stats.kstest(rand[0], 'norm')
237 # Plotting
238 plt.plot(n, P_s, label='Scipy')
239 plt.plot(n, P, label='Self written')
240 plt.title('KS-Test')
241 plt.ylabel('$P(z)$')
242 plt.xlabel('N')
243 plt.xscale('log')
244 plt.legend(loc = 'lower right', frameon=False)
245 plt.savefig('plots/1e.png')
246 plt.close()
247 print('Generated plots/1e.png')
248
249 #---1.d---
250 # Kuipers test
251 # Preparing arrays
252 kuip_P, kuip_P_s = np.zeros(len(n)), np.zeros(len(n))
253 kuip_d, kuip_d_s = np.zeros(len(n)), np.zeros(len(n))
254 # Running test for different values of N
255 for i in range(len(n)):
256     rand = box_muller(rng.rand_num(n[i]), rng.rand_num(n[i]), mu, sig)
257     kuip_d[i], kuip_P[i] = KS_Kuip_test(rand[0], gauss, mu, sig, Kuip=True)
258     kuip_d_s[i], kuip_P_s[i] = stats.kstest(rand[0], 'norm')
259 # Plotting
260 plt.plot(n, kuip_P_s, label='Scipy')
261 plt.plot(n, kuip_P, label='Self written')
262 plt.title('Scipy KS-Test and self-written Kuiper-Test')
263 plt.ylabel('$P(z)$')
264 plt.xlabel('N')
265 plt.xscale('log')
266 plt.legend(loc = 'upper right', frameon=False)
267 plt.savefig('plots/1f.png')
268 plt.close()
269 print('Generated plots/1f.png')
270
271 #---1.e---
272 # Testing on given random numbers
273 filename = 'randomnumbers.txt'
274 url = 'https://home.strw.leidenuniv.nl/~nobels/coursedata/'
275 if not os.path.isfile(filename):
276     print(f'File not found, downloading {filename}')
277     os.system('wget '+url+filename)
278 random_num = np.genfromtxt(filename, delimiter=' ', skip_footer=1)
279
280 n = np.logspace(np.log10(10), np.log10(len(random_num)), dtype=int)
281 test_P, test_D = np.zeros((10, len(n)), dtype=list), np.zeros((10, len(n)), dtype=list)
282 # Applying Kuipers test
283 for i in range(10):
284     for j in range(len(n)):
285         rand = np.array(random_num[:n[j], i])
286         test_D[i][j], test_P[i][j] = KS_Kuip_test(rand, gauss, mu, sig, Kuip=True)
287 # Plotting
288 plt.plot(n, kuip_P_s, label='Scipy (KS)', color = 'g')
289 for i in range(10):
290     plt.plot(n, test_P[i], label = i)
291 plt.title('Scipy KS-Test and self-written Kuiper-Test')
292 plt.ylabel('$P(z)$')
293 plt.xlabel('N')
294 plt.xscale('log')
295 plt.legend(loc=2, bbox_to_anchor=(1,1))
296 plt.savefig('plots/1g.png')
297 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 ϕ is a random phase. The power spectrum has the following form:

$$P(k) \propto k^n \quad (1)$$

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 .

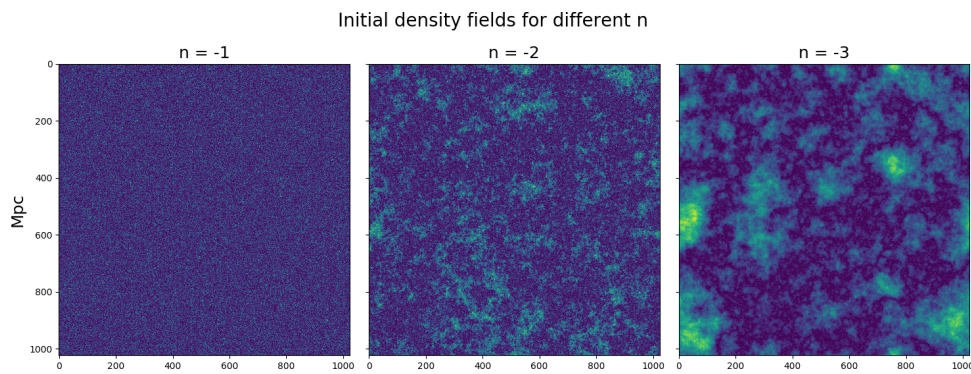


Figure 7: Gaussian random fields for different n values. Notice the clear presence of larger structure when the spectrum is more peaked (lower n).

2.1 Scripts

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

```
1  --- Exercise 2 ---
2  Original seed: 627310980
3  Generating a random field with n = -1 of dimension 1024x1024 (mu = 0)
4  Generating a random field with n = -2 of dimension 1024x1024 (mu = 0)
5  Generating a random field with n = -3 of dimension 1024x1024 (mu = 0)
6  Generated plots/2.png
```

a2.2.txt

Here is the script used to produce these results:

```
1  # a2.2
2  import numpy as np
3  import sys
4  from matplotlib import pyplot as plt
5  from scipy import stats
6  import os
7  from a2.1 import rng, box_muller
8
9  # --- Functions and classes ---
10
11 def random_field_generator(n, N, rng, mu=0):
12     # Prepares a random field in Fourier space
13     print(f'Generating a random field with n = {n} of dimension {N}x{N} (mu = {mu})')
14     df = np.zeros((N, N), dtype=complex)
15     # Setting values of top half of the field
```

```

16     for j in range((N//2)+1):
17         # Determining the value of k_y
18         k_y = j*2*np.pi/N
19         for i in range(N):
20             # Determining the value of k_x and sigma_x
21             if i <= (N//2):
22                 k_x = (i)*2*np.pi/N
23             else:
24                 k_x = (-N+i)*2*np.pi/N
25             # Avoid dividing by 0
26             if i != 0 or j != 0:
27                 sig = ((k_x**2+k_y**2)**0.5)**(n/2)
28             else:
29                 sig = 0
30             # Drawing a random number from normal distrib
31             #df[j][i] = np.random.normal(0,sig)+ 1j*np.random.normal(0,sig)
32             rand = box_muller(rng.rand_num(1),rng.rand_num(1),mu,sig)
33             df[j][i] = rand[0] + 1j*rand[1]
34         # Setting values of points who need to equal their own conjugates
35         df[0][0] = 0
36         df[0][N//2] = (df[0][N//2].real)**2
37         df[N//2][0] = (df[N//2][0].real)**2
38         df[N//2][N//2] = (df[N//2][N//2].real)**2
39         # Setting values of bottom half of the field using conjugates
40         for j in range((N//2)+1):
41             for i in range(N):
42                 df[-j][-i] = df[j][i].conjugate()
43         return df
44 #end random_field generator()
45
46 # — Commands, prints and plots —
47 if __name__ == '__main__':
48     print('— Exercise 2 —')
49     seed = 627310980
50     rng = rng(seed)
51     print('Original seed:',seed)
52
53     # Making initial density fields for different n values
54     N = 1024
55     df1 = random_field_generator(-1,N,rng)
56     df1_inft = np.fft.ifft2(df1)
57     df2 = random_field_generator(-2,N,rng)
58     df2_inft = np.fft.ifft2(df2)
59     df3 = random_field_generator(-3,N,rng)
60     df3_inft = np.fft.ifft2(df3)
61     # Plotting fields
62     fig, ((ax1,ax2,ax3)) = plt.subplots(1, 3,sharex='col', sharey='row',figsize=(15,15))
63     ax1.set_title('n = -1',size=18)
64     ax1.imshow(np.abs(df1_inft))
65     ax1.set_ylabel('Mpc',size=18)
66     ax1.invert_yaxis()
67     ax2.set_title('n = -2',size=18)
68     ax2.imshow(np.abs(df2_inft))
69     ax3.set_title('n = -3',size=18)
70     ax3.imshow(np.abs(df3_inft))
71     fig.suptitle('Initial density fields for different n',y=0.7,size=20)
72     fig.tight_layout()
73     plt.savefig('plots/2.png',bbox_inches='tight',pad_inches = 0)
74     plt.close()
75     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:

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

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:

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

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

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

The density growth equation for this Universe is the following:

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

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:

```

1  ——— Exercise 1 ———
2  Original seed: 627310980
3  Generated plots/1a.png
4  Generated plots/1b.png
5  Generated plots/1c.png
6  Generated plots/1d.png
7  Generated plots/1e.png
8  Generated plots/1f.png
9  Generated plots/1g.png
10 ——— Exercise 3 ———

```

a2_3.txt

Here is the script used to produce these results:

```

1  # a2_3
2  import numpy as np
3  import sys
4  import matplotlib.pyplot as plt
5  from a2_1 import rng, box_muller
6
7  def k_calc(h, f, t, x1, x2, xn):
8      # Likely source of error: What do we do with the second variable when calculating k?
9      # To-do: Try to solve the problem by applying it on a simpler function
10     k1 = h * f(t, x1, x2)

```

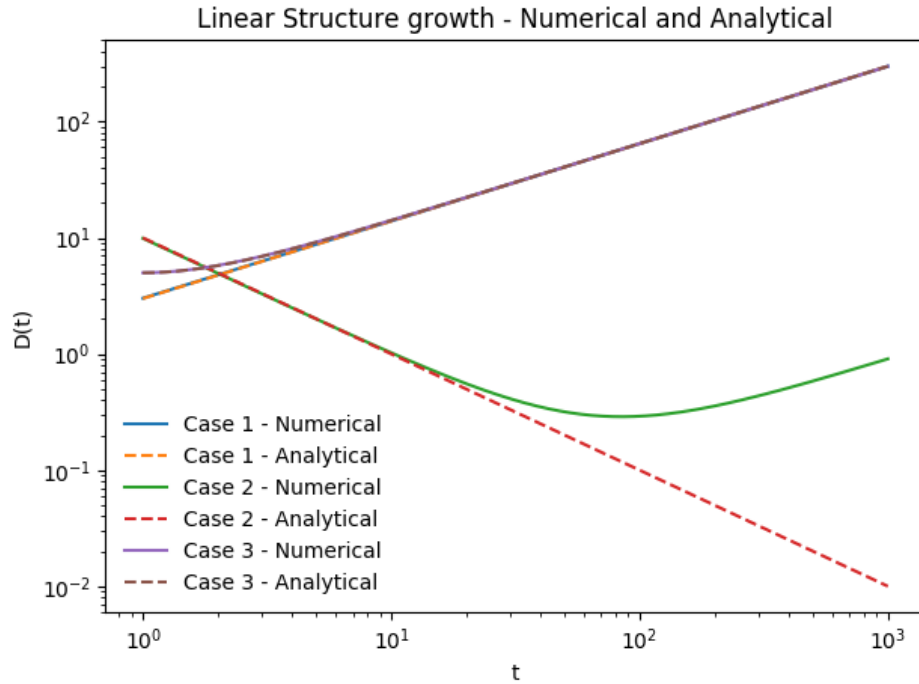


Figure 8: Analytical and numerical solutions to the partial differential equations given in this question.

```

11     k2 = h * f(t+0.5*h, x1+0.5*k1, x2+0.5*k1)
12     k3 = h * f(t+0.5*h, x1+0.5*k2, x2+0.5*k2)
13     k4 = h * f(t+h, x1+k3, x2+k3)
14     return xn+1/6*k1+1/3*k2+1/3*k3+1/6*k4
15
16 def runge_kutta(x0,y0,f,xmax,h=0.0001):
17     #Implementaiton of the Runge-Kutta method for ODE integration
18     # x0,y0 are the starting values and f is the ode()
19     xn,yn = x0,y0
20     y_out,x_out = [],[]
21     while xn < xmax:
22         yn_new = k_calc(h,f,xn,yn)
23         y_out.append(yn_new)
24         xn += h
25         x_out.append(xn)
26         yn = yn_new
27
28     plt.plot(x_out,y_out)
29
30     return np.sum(y_out)*h
31
32 def k_calc2nd(h,f,g,t,x1,x2):
33     # Support function for runge-kutta method (for 2nd order ODEs)
34     k1 = h * f(t,x1,x2)
35     l1 = h * g(t,x1,x2)
36     k2 = h * f(t+0.5*h, x1+0.5*k1, x2+0.5*l1)
37     l2 = h * g(t+0.5*h, x1+0.5*k1, x2+0.5*l1)
38     k3 = h * f(t+0.5*h, x1+0.5*k2, x2+0.5*k2)
39     l3 = h * g(t+0.5*h, x1+0.5*k2, x2+0.5*k2)
40     k4 = h * f(t+h, x1+k3, x2+k3)
41     l4 = h * g(t+h, x1+k3, x2+k3)
42
43     x1_new = x1+1/6*(k1+2*k2+2*k3+k4)
44     x2_new = x2+1/6*(l1+2*l2+2*l3+l4)
45
46     return x1_new, x2_new
47 #end k_calc2nd()

```

```

48
49 def runge_kutta2nd(x1_0,x2_0,t0,t,f,g,h=0.01):
50     #Implementaiton of the Runge-Kutta method for ODE integration
51     # x0,y0 are the starting values and f is the ode()
52     t = np.arange(t0,t+h,h)
53     #print(t)
54     x1n,x2n = x1_0,x2_0
55     x1_out = np.zeros(len(t))
56     for i in range(len(t)):
57         x1n,x2n = k_calc2nd(h,f,g,t[i],x1n,x2n)
58         x1_out[i] = x1n
59     return np.sum(x1_out)*h,x1_out
60
61 # — Commands, prints and plots —
62 if __name__ == '__main__':
63     print('— Exercise 3 —')
64     seed = 627310980
65     rng = rng(seed)
66     print('Original seed:',seed)
67
68     f = lambda t,x1,x2: x2
69     g = lambda t,x1,x2: -4/(3*t)*x2 + 2/(3*t**2)*x1
70     case1,yt1 = runge_kutta2nd(3,2,1,1000,f,g)
71     case2,yt2 = runge_kutta2nd(10,-10,1,1000,f,g)
72     case3,yt3 = runge_kutta2nd(5,0,1,1000,f,g)
73     print(f'case1: {case1},case2: {case2}, case3: {case3}')
74
75     f = lambda t,x1,x2 : x2
76     g = lambda t,x1,x2 : x1*6-x2
77
78     D1 = lambda t : 3*t**(2/3)
79     D2 = lambda t : 10/t
80     D3 = lambda t : (3*t**(5/3)+2)/t
81     t = np.arange(1,1000+0.01,0.01)
82     plt.plot(t,yt1,label='Case 1 - Numerical')
83     plt.plot(t,D1(t),linestyle='—',label='Case 1 - Analytical')
84     plt.plot(t,yt2,label='Case 2 - Numerical')
85     plt.plot(t,D2(t),linestyle='—',label='Case 2 - Analytical')
86     plt.plot(t,yt3,label='Case 3 - Numerical')
87     plt.plot(t,D3(t),linestyle='—',label='Case 3 - Analytical')
88     plt.legend(frameon=False)
89     plt.xlabel('t')
90     plt.ylabel('D(t)')
91     plt.title('Linear Structure growth - Numerical and Analytical')
92     plt.xscale('log')
93     plt.yscale('log')
94     plt.tight_layout()
95     plt.savefig('plots/3.png')
96     plt.close()
97     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:

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

Where

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

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

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

Where

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

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