

Numerical Recipes for Astrophysics

Solutions hand-in assignment-2

Luther Algra - s1633376

May 25, 2019

Abstract

The current document contains the solutions for the second hand-in assignment of Numerical Recipes. The main questions, 1, 2, 3 ..., 7, are in this document all given their own section. Each section contains a subsection for its related sub-questions (1.a, 1.b, 1.c, ..., 1.e) and ends with a final subsection that contains two segments of code. The first segment contains the code for the full main question. The second segment contains the code of shared modules used by the sub-questions. A sub-question itself always starts with a short summary of the question that needs to be answered. The summary is followed by an explanation of how the problem is solved and the code that provides the solution. The output of the code is always presented after the code and is there discussed if necessary.

1 - Normally distributed pseudo-random numbers

Question 1.a

Problem

Write a random number generator that returns a random floating-point number between 0 and 1. At minimum, use some combination of an MWC and a 64-bit XOR-shift. Plot a sequential of random numbers against each other in a scatter plot (x_{i+1} vs x_i) for the first 1000 numbers generated. Also plot the value of the random numbers for the first 1000 numbers vs the index of the random number, this mean the x-axis has a value from 0 through 999 and the y-axis 0 through 1). Finally, have your code generate 1,000,000 random numbers and plot the result of binning these in 20 bins 0.05 wide.

Solution

The state of the random number generator (RNG) is updated by first performing a 64-bit XOR-shift on the current state. Next, a modified version of the 64-bit XOR-shift output is given to the MWC algorithm. The modified XOR-shifts output given to the MWC algorithm is the output of the 64 XOR-shift with the last 32 bits put to zero. This is done by performing the 'AND' operation with the maximum value of an unsigned int32. This modification was performed as the MWC algorithm expects as input a 64-bit unsigned integer with a value between $0 < x < 2^{32}$. The output of the MWC is finally XORd with the unmodified output of the 64-bit XOR-shift. The result is set as the new state of the RNG.

The first 32 bits of the new state are used to provide a random value, as the output of the MWC algorithm only contains 32 significant bits. This random value is obtained by performing the 'AND' operation between the seed and the maximum value of an unsigned int32. The resulting value is then divided by the maximum value of an unsigned int32 to obtain a value between 0 and 1.

The code for the random number generator can be found at the end of this section, as it is treated as a shared module (see page 24). The code for generating the plots and the created plots can be found below. The code does not only print the random seed, but also prints the maximum and minimum number of counts for the binned 1,000,000 values. These values are referred to in the description of the plot that displayed the uniformness (figure 2).

Code - Plots

The code for generating the plots. The used imports and the initialization of the random number generator are not explicit shown in this piece of code, but can be found on page 19. The code for the random number generator can, as mentioned before, be found on page 24.

```
1 def assignment_1a(random):
2     """
3     Execute assignment 1.a
4     Int:
5     param: random — An initialization of the random number generator.
6     """
7
8     # The relevant imports for this piece of code are:
9
10    # (1) matplotlib.pyplot as plt
11    # (2) mathlib.random as random
12    # (3) mathlib.stats as ml_stats
13    # (3) numpy as np
14
15    # Print the seed.
16    print('[1.a] Initial seed: ', random.get_seed())
17
18    # Generate 1000 numbers.
19    numbers_1000 = random.gen_uniforms(1000)
20
21    # Plot them against each other.
22    plt.scatter(numbers_1000[0:999], numbers_1000[1:], s=2)
23    plt.ylabel(r'Probability  $x_{i+1}$ ')
24    plt.xlabel(r'Probability  $x_i$ ')
25    plt.savefig('./Plots/1_plot_against.pdf')
26    plt.figure()
27
28    # Plot them against the index.
29    plt.plot(range(0, 1000), numbers_1000)
30    plt.ylabel('Probability p')
31    plt.xlabel('Index')
32    plt.savefig('./Plots/1_plot_index.pdf')
33    plt.figure()
34
35    # Create a histogram for 1e6 points with 20 bins of 0.05 wide.
36    numbers_mil = random.gen_uniforms(int(1e6))
37    plt.hist(numbers_mil, bins=20, range=(0,1), color='orange', edgecolor='black')
38    plt.ylabel('Counts')
39    plt.xlabel('Generate values')
40    plt.savefig('./Plots/1_hist_uniformnes.pdf')
41    plt.figure()
42
43    # Extra, to print the smallest and lagest bin value.
44    counts, _ = np.histogram(numbers_mil, bins=20)
45    print('[1.a] Max counts: ', max(counts))
```

./Code/assignment_1.py

Code - Output text

The text output produced by the code. The first value is the initial seed of the RNG. The second and third value are the maximum and minimum amount of counts for the histogram displaying the uniformness.

```
1 [1.a] Initial seed: 78379522
2 [1.a] Max counts: 50343
3 [1.a] Min counts: 49557
```

./Output/assignment1_out.txt

Code - Output plots

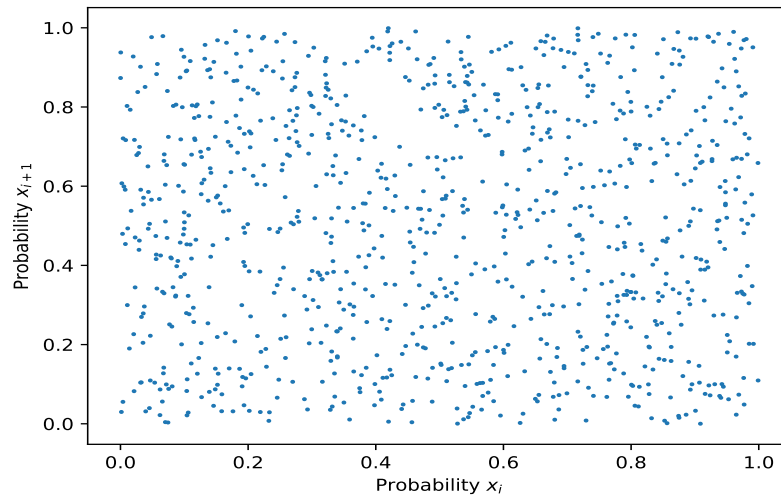


Figure 1: A plot of random number x_{i+1} against random number x_i for the first 1000 random uniforms produced by the random number generator. A good random number generator should produce a homogeneous plot without many (large) empty spots. The largest empty spot in the above plot is at $x_i = 0.4$ and $x_{i+1} = 0.8$. The spot is not significant large, but might point towards an impurity in the RNG.

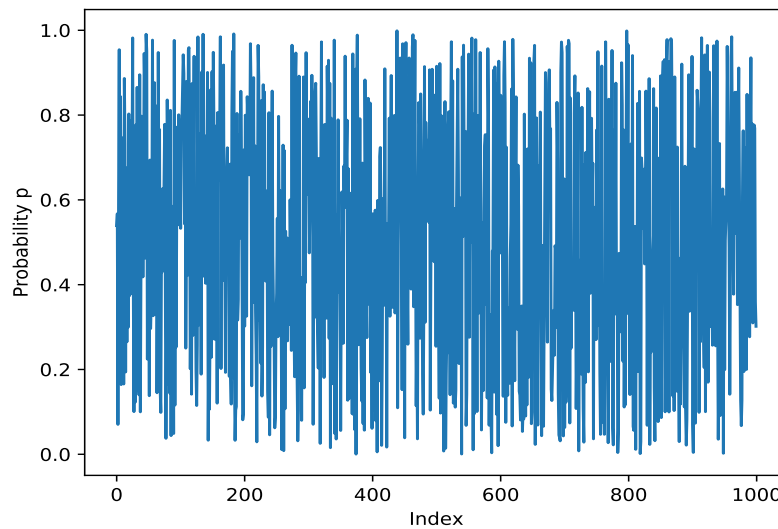


Figure 2: The first 1000 random uniform numbers produced by the random number generator (RNG) against their index. A good random number generator should not have large wide gaps (e.g when moving from index 400 to 450 it should not only produce values larger than 0.8, which would leave a wide gap). In the plot small gaps appear, see for example index ~ 420 at a probability of 0.6. The number of gaps and the width of the gaps do not appear to be significant. This might therefore either be the result of being unlucky, or could point again towards an impurity in the RNG. The average value produced by the RNG should furthermore be 0.5. This corresponds to rapidly moving up and down around the horizontal line corresponding with a probability of $p = 0.5$. In the plot this should, result in a 'dense' region (less white) around the line $p = 0.5$. It can indeed be seen that the plot is denser around the line $p = 0.5$ than at $p = 0.8$ or $p = 0.2$.

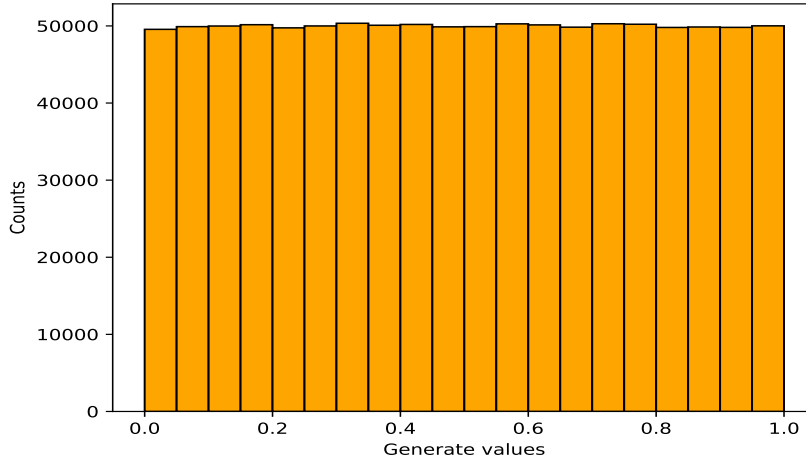


Figure 3: The uniforms of the random number generator for 1 million random values. The values are binned in 20 bins. A good random number generator should fluctuate around $50000 \pm 2\sqrt{50000} = 50000 \pm 447$ counts per bin (2 sigma). The maximum and minimum amount of counts corresponds to 50343 and 49557 counts. These values just lay withing the 2 sigma uncertainty. The uniformness of the random number generator therefore appears to be acceptable.

Question 1.b)

Problem

Now use the Box-Muller method to generate 1000 normally-distributed random numbers. To check if they are following the expected Gaussian distribution, make a histogram (scaled appropriate) with the corresponding true probability distribution (normalized to integrate to 1) as line. This plot should contain the interval of -5σ until 5σ from the theoretical probability distribution. Indicate the theoretical 1σ , 2σ , 3σ and 4σ interval with a line. For this plot, use $\mu = 3$ and $\sigma = 2.4$ and choose bins that are appropriate.

Solution

The solution consists of deriving the transformation of two i.i.d uniform variables to two i.i.d normal distributed variables with the Box-Muller method. A brief form of the derivation can be found below. The final transformation, equation 9, is implemented in the random number generator and used to generate the plot. The final histogram is created with 20 bins and can be found on page 6.

Let $X, Y \sim G(\mu, \sigma^2)$ be two i.i.d Gaussian distributed random variables. Their joined CDF is then given by,

$$P(X \leq x_1, Y \leq y_1) = \int_{-\infty}^{x_1} \int_{-\infty}^{y_1} G(x|\mu, \sigma^2)G(y|\mu, \sigma^2)dxdy \quad (1)$$

Transforming to polar coordinates by substituting $(x - \mu) = r \cos(\theta)$ and $(y - \mu) = r \sin(\theta)$ yields,

$$\begin{aligned} P(R \leq r_1, \Theta \leq \theta_1) &= \int_0^{r_1} \int_0^{\theta_1} G(r \cos(\theta)\sigma + \mu|\mu, \sigma^2)G(r \sin(\theta)\sigma + \mu|\mu, \sigma^2)rdrd\theta \\ &= \frac{1}{2\pi\sigma^2} \int_0^{r_1} \int_0^{\theta_1} r e^{-\frac{1}{2}\left[\left(\frac{r \cos(\theta)}{\sigma}\right)^2 + \left(\frac{r \sin(\theta)}{\sigma}\right)^2\right]} drd\theta \\ &= \frac{1}{2\pi\sigma^2} \int_0^{r_1} \int_0^{\theta_1} r e^{-\frac{r^2}{2\sigma^2}} drd\theta \end{aligned}$$

The CDF's for the polar coordinates are now given by,

$$P(R \leq r_1) = \frac{1}{\sigma^2} \int_0^{r_1} r e^{-\frac{r^2}{2\sigma^2}} dr = \int_0^{r_1} \frac{d}{dr} \left(-e^{-\frac{r^2}{2\sigma^2}} \right) dr = 1 - e^{-\frac{r_1^2}{2\sigma^2}} \quad (2)$$

$$P(\Theta \leq \theta_1) = \frac{1}{2\pi} \left[-e^{-\frac{r^2}{2\sigma^2}} \right]_0^\infty \int_0^{\theta_1} d\theta = \frac{\theta_1}{2\pi} \quad (3)$$

The CDFs can be used to convert two uniform distributed variables to the polar coordinates of the Gaussian distributed variables. Let $U_1, U_2 \sim U(0, 1)$ be two i.i.d uniform variables. From the transformation law of probability we then must have that,

$$P(R \leq r_1) = P(U_1 \leq u_1) \rightarrow 1 - e^{-\frac{r_1^2}{2\sigma^2}} = \int_0^{u_1} du_1 = u_1 \quad (4)$$

$$P(\Theta \leq \theta) = P(U_2 \leq u_2) \rightarrow \frac{\theta_1}{2\pi} = \int_0^{u_2} du_2 = u_2 \quad (5)$$

The transformation from the two uniform distributed variables to the polar coordinates of the Gaussian distributed variables then becomes,

$$r_1 = \sqrt{-2\sigma^2 \ln(1 - u_1)} \quad (6)$$

$$\theta_1 = 2\pi u_2 \quad (7)$$

Converting back to Cartesian coordinates then yields the transformation from two i.i.d uniform distributed variables to two i.i.d Gaussian distributed variables;

$$x_1 = r \cos(\theta) + \mu = \sqrt{-2\sigma^2 \ln(1 - u_1)} \cos(2\pi u_2) + \mu \quad (8)$$

$$y_1 = r \sin(\theta) + \mu = \sqrt{-2\sigma^2 \ln(1 - u_1)} \sin(2\pi u_2) + \mu \quad (9)$$

These transformation are implemented in the random number generator (see page 24) on line 130. The code for the generation of the plot and the created plot can be found below. The code that generates the plot makes besides the RNG use of a function for the normal distribution in the file `./Code/mathlib/statistics.py`. This file is treated as a shared module and can be found on page 31. The called function, `normal`, can be found on line 214 in this file.

Code - Plots

The code for generating the plots. The imports are again not explicit shown, but can be found on page 19. The shared modules can be found on pages 24 and 31.

```

1 def assignement_1b(random):
2     """
3         Execute assignement 1.b
4     Int:
5         param: random — An instance of the random number generator.
6     """
7
8     # The relevant imports for this piece of code are:
9
10    # (1) matplotlib.pyplot as plt
11    # (2) mathlib.random as random
12    # (3) mathlib.stats as ml_stats
13    # (4) numpy as np
14
15    # Sigma and mean for the distribution.
16    mean = 3.0
17    sigma = 2.4
18
19    # Generate 1000 random normal variables for the given mean and sigma.
```

```

20 samples = random.gen_normals(mean, sigma, 1000)
21
22 # The true normal distribution for the given mean and sigma.
23 gaussian_x = np.linspace(-sigma*4 +mean, sigma*4 +mean, 1000)
24 gaussian_y = ml.stats.normal(gaussian_x, mean, sigma)
25
26 # Create a histogram.
27 plt.hist(samples, bins=20, density=True, edgecolor='black',
28          facecolor='orange', zorder=0.1, label='Sampled')
29 plt.plot(gaussian_x, gaussian_y, c='red', label='Normal')
30 plt.xlim(-sigma*6.5 + mean, sigma*6.5 + mean)
31 plt.ylim(0, max(gaussian_y)*1.2)
32
33 # Add the sigma lines.
34
35 # The hight of the sigma lines that need to be added.
36 lines_height = max(gaussian_y)*1.2
37
38 for i in range(1, 6):
39     # Absolute shift from the mean for the given sigma
40     shift = i*sigma
41
42     # Sigma right of the mean.
43     plt.vlines(mean + shift, 0, lines_height,
44               linestyle='-', color='black', zorder=0.0)
45     plt.text(mean + shift -0.4, lines_height/1.3, str(i) + r'$\sigma$',
46             color='black', backgroundcolor='white', fontsize=9)

```

./Code/assignment.1.py

Code - Output plot(s)

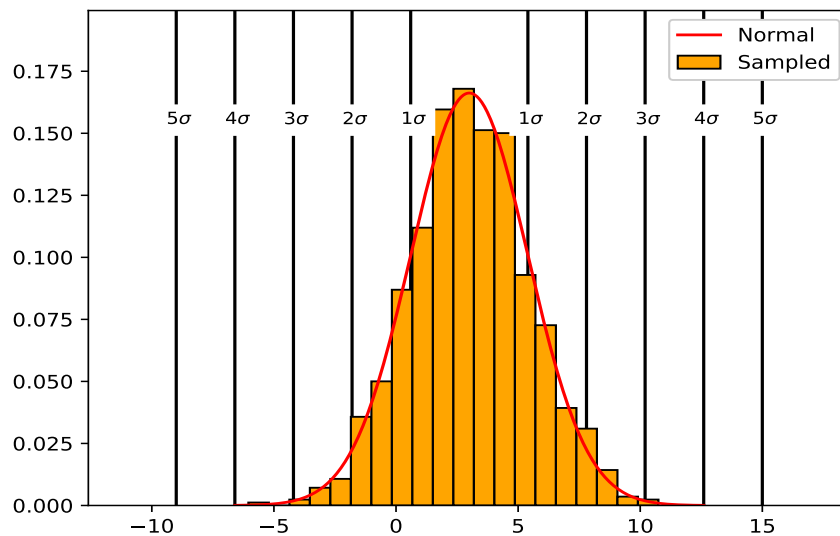


Figure 4: A histogram of the 1000 random normal distributed variables generated with the box muller method for $\mu = 3$ and $\sigma = 2.4$ (orange). The red line is the true normal distribution for these values of μ and σ . The histogram appears to approximate the distribution quite well, but displays small deviations. The bin left of the peak (the highest bin) is larger than it should be and the first two bins right of the peak appear to respectively lack counts and have too many counts. By eye the histogram appears to be acceptable. A statistical test is of course better to determine whether the histogram would truly be acceptable or not.

Question 1.c

Problem

Write a code that can do the KS-test on the your function to determine if it is consistent with a normal distribution. For this, use $\mu = 0$ and $\sigma = 1$. Make a plot of the probability that your Gaussian random number generator is consistent with Gaussian distributed random numbers, start with 10 random numbers and use in your plot a spacing of 0.1 dex until you have calculated it for 10^5 random numbers on the x-axis. Compare your algorithm with the KS-test function from `scipy`, `scipy.stats.kstest` by making an other plot with the result from your KS-test and the KS-test from `scipy`.

Solution

The implementation of the KS-test is in general straight forwards. There are however two points of interest that needs to be discussed. The first point is the implementation of the CDF for the KS-statistic and the second point is the implementation of the CDF for the normal distribution.

(1) CDF KS-statistic

The p-value produced by the KS-tests requires the evaluation of the CDF for the KS-test statistic,

$$P_{KS}(z) = \frac{2\sqrt{\pi}}{z} \sum_{j=1}^{\infty} \exp\left(-\frac{(2j-1)^2 + \pi^2}{8z^2}\right) \quad (10)$$

This infinite sum needs to be numerically approximated in order to perform the KS-test. The chosen approximation in the implementation of the KS-test for the sum is taken from the book *Numerical Recipes - The art of Scientific Computation*, 3d edition,

$$P_{KS}(z) \approx \begin{cases} \frac{\sqrt{2\pi}}{z} \left[\left(e^{-\pi^2/(8z^2)}\right)^9 + \left(e^{-\pi^2/(8z^2)}\right) \left(e^{-\pi^2/(8z^2)}\right)^{25} \right] & \text{for } z < 1.18 \\ 1 - 2 \left[\left(e^{-2z^2}\right) - \left(e^{-2z^2}\right)^4 + \left(e^{-2z^2}\right)^9 \right] & \text{for } z \geq 1.18 \end{cases} \quad (11)$$

(2) CDF normal distribution

The CDF of the normal distribution is needed in order to perform the KS-test under the null hypothesis that the data follows a normal distribution. The CDF of the normal distribution can in general be written as,

$$\Phi\left(\frac{x-\mu}{\sigma}\right) = \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \right] \quad (12)$$

where the erf is given by,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (13)$$

The integral of the erf function lacks a closed form and therefore also needs to be numerically approximated. The chosen approximation is taken from *Abramowitz and Stegun*,

$$\operatorname{erf}(x) \approx 1 - (a_1 t + a_2 t^2 + \dots + a_5 t^5) e^{-x^2} \quad t = \frac{1}{1 + px} \quad (14)$$

where $p = 0.3275911$, $a_1 = 0.254829592$, $a_2 = -0.284496736$, $a_3 = 1.421413741$, $a_4 = -1.453152027$, $a_5 = 1.061405429$.

The KS-test and the CDF are implemented with these approximations. The code for the KS-test and the CDF is located in the file `./Code/mathlib/statistics.py` at page 31, as this file is threaded as a shared module. The KS-test does require an sorting algorithm, this algorithm is implemented in the file `./Code/matlib/sorting` and can be found on page ???. The code for the generation of the plots and plots are displayed below.

Code - Plots

The code for generating the two plots. The imports for this file are not explicit shown, but can be found on page 19.

```
1 def assignment_1c(random):
2     """
3     Execute assignment 1.c
4     Int:
5     param: random — An initialization of the random number generator.
6     """
7
8     # The relevant imports for this piece of code are:
9     # (1) matplotlib.pyplot as plt
10    # (2) numpy as np
11    # (3) astropy.stats
12    # (4) mathlib.statistics as ml_stats
13
14    # The values to plot point for.
15    plot_values = np.array(10*np.arange(1, 5.1, 0.1), dtype=int)
16
17    # An array in which the p-values are stored for the self created.
18    # ks-test and the scipy version.
19    p_values_self = np.zeros(len(plot_values))
20    p_values_scipy = np.zeros(len(plot_values))
21
22    # Generate the maximum amount of needed random numbers.
23    random_numbers = random.gen_normals(0, 1, int(1e5))
24
25    # Calculate the p-values with the ks-test.
26    for idx, values in enumerate(plot_values):
27
28        # Calculate the value with scipy.
29        p_values_scipy[idx] = sp.stats.kstest(random_numbers[0:values],
30                                             'norm')[1]
31
32        # Calculate the p-values with the own implementation.
33        p_values_self[idx] = ml_stats.kstest(random_numbers[0:values],
34                                             ml_stats.normal_cdf)
35
36    # Plot the probabilities for only my own implementation.
37    plt.plot(plot_values, p_values_self, label='self', color='orange')
38    plt.hlines(0.05, 0, 10**5, colors='red', linestyle='—')
39    plt.xscale('log')
40    plt.xlabel(r'Log($N_{samples}$)')
41    plt.ylabel('Probabillity (p-value)')
42    plt.legend()
43    plt.savefig('./Plots/1_plot_ks-test-self.pdf')
44    plt.figure()
45
46    # Plot the probabilities for both the scipy and my own implemntation.
47    plt.plot(plot_values, p_values_scipy, label='scipy', linestyle=':',
48             zorder=1.1)
49    plt.plot(plot_values, p_values_self, label='self', zorder=1.0,
50             color='orange')
51    plt.hlines(0.05, 0, 10**5, colors='red', linestyle='—')
52    plt.xscale('log')
53    plt.xlabel(r'Log($N_{samples}$)')
54    plt.ylabel('Probabillity (p-value)')
55    plt.legend()
56    plt.savefig('./Plots/1_plot_ks-test-self-scipy.pdf')
```

./Code/assignment_1.py

Code - Output plot(s)

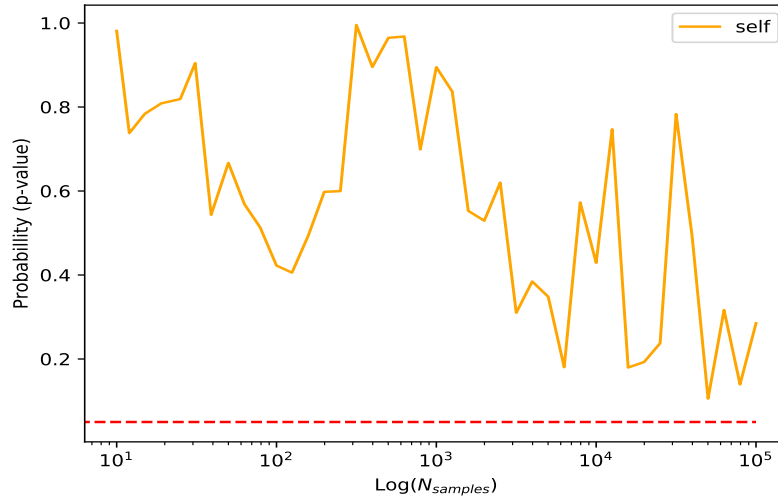


Figure 5: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** the line would suggest that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the RNG always passes KS-test up to at least 10^5 samples. The p-value does however appear to drop for a large number of samples and might even drop further when more samples are used. The drop suggests again that the RNG is likely not perfect.

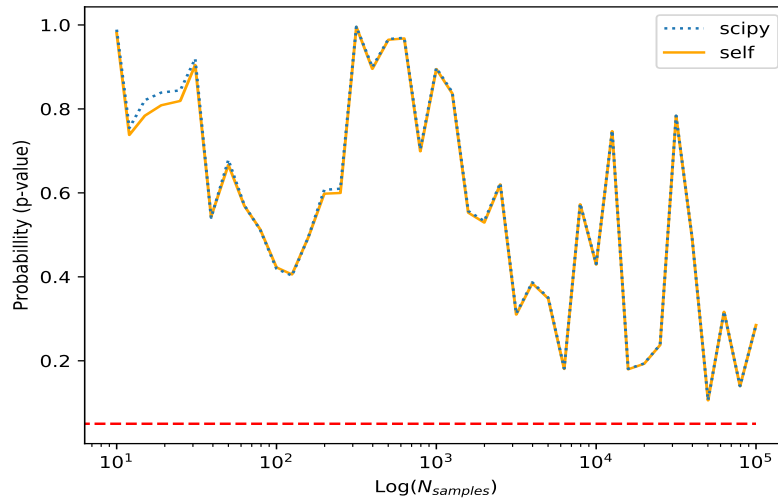


Figure 6: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. The orange line is the self written implementation of the KS-test and the blue line is the scipy version. A point **below** the red line would suggest that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The self written KS-test is close to the scipy version, but shows (small) deviations at small sample sizes (for example at $N_{samples} = 10$ or $N_{samples} = 200$). The self written implementation always has the same shape as the scipy version, even at the deviations. The exact cause for the deviations are unknown, but are likely the result of an approximation that scipy makes that the self written implementation doesn't make. (This is not confirmed by looking at the scipy code.)

Question 1.d)

Problem

Write a code that does the Kuiper's test on your random numbers (see tutorial 8) and make the same plot as for the KS-test.

Solution

The implementation of the Kuiper test does require a numerical approximation of the CDF for the kuiper statistics. The CDF of the kuiper staistic is given by,

$$P_{kuiper}(\lambda) = 1 - 2 \sum_{j=1}^{\infty} (4j^2\lambda^2 - 1)e^{-2j^2\lambda^2} \quad (15)$$

The sum in the above expression negligible compared to the machine error if $\lambda < 0.4$. In this case the numerically approximation thus consist of returning 1. If $\lambda > 0.4$ then the sum is approximated by calculating the first 100 terms of the sum. This should be more than enough for the sum to converge.¹

The kuiper test and the CDF are implemented in the shared module `./Code/mathlib/statistics.py` on page 31. The code that creates the plots and the plots can be found below. The code does make use of **astropy** to compare the self written implementation of the Kuiper-test with the implementation of astropy.

Code - Plots

The code for generating the two plots. The imports are not explicit shown but can be shown on page 19.

```
1 def assignment_1d(random):
2     """
3     Execute assignment 1.d
4     Int:
5     param: random — An initialization of the random number generator.
6     """
7
8     # The relevant imports for this piece of code are:
9     # (1) matplotlib.pyplot as plt
10    # (2) numpy as np
11    # (3) scipy.stats as sp_stats
12    # (4) mathlib.statistics as ml_stats
13
14    # The values to plot point for.
15    plot_values = np.array(10*np.arange(1, 5.1, 0.1), dtype=int)
16
17    # Generate the maximum amount of needed random numbers.
18    random_numbers = random.gen_normals(0, 1, int(1e5))
19
20    # An array in which the p-values are stored for the self created
21    # kuiper-test and the astropy version.
22    p_values_self = np.zeros(len(plot_values))
23    p_values_astropy = np.zeros(len(plot_values))
24
25    # Calculate the p-values with the ks-test
26    for idx, values in enumerate(plot_values):
27
28        # Calculate the value with the own implemnetation
29        p_values_self[idx] = ml_stats.kuiper_test(random_numbers[0:values],
30        ml_stats.normal_cdf)
31        # Calculare the value with astropy.
32        p_values_astropy[idx] = astropy.stats.kuiper(random_numbers[0:values],
33        ml_stats.normal_cdf)[1]
```

¹In theory less terms are enough. The evaluation of the sum could therefore stop early by checking for a required precision.

```

33
34 # Plot the probabilities for only my own implementation
35 plt.plot(plot_values, p_values_self, label = 'self')
36 plt.hlines(0.05,0,10**5,colors='red',linestyles='—')
37 plt.xscale('log')
38 plt.xlabel(r'Log($N_{samples}$)')
39 plt.ylabel('Probabillity (p-value)')
40 plt.legend()
41 plt.savefig('./Plots/1-plot-kuiper-test-self.pdf')
42 plt.figure()
43
44 # Plot the probabilliteis with both the own implementation and astropy
45 plt.plot(plot_values, p_values_astropy, label='astropy', linestyle=':',
46         zorder=1.1)
47 plt.plot(plot_values, p_values_self, label='self',zorder=1.0)
48 plt.hlines(0.05,0,10**5,colors='red',linestyles='—')
49 plt.xscale('log')
50
51 plt.xlabel(r'Log($N_{samples}$)')
52 plt.ylabel('Probabillity (p-value)')
53 plt.legend()
54 plt.savefig('./Plots/1-plot-kuiper-test-self-astropy.pdf')
55 plt.figure()

```

./Code/assigment_1.py

Code - Output plot(s)

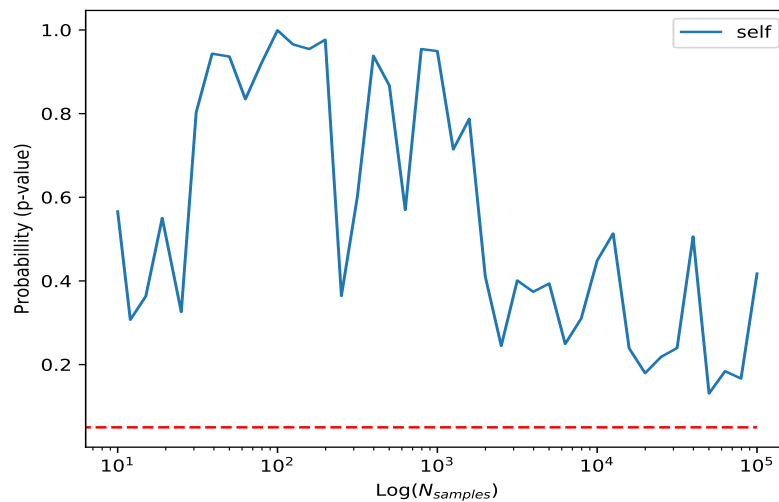


Figure 7: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the RNG always passes kuiper test. It can however be seen that the p-value stays lower for larger sample size, similar as with the KS-test.

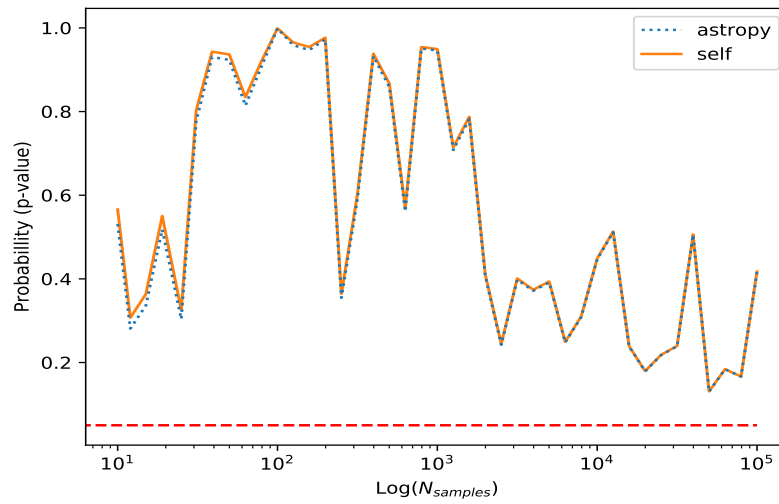


Figure 8: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the self written implementation has (small) deviations from the astropy implementation at small sample sizes. This is similar to the situation with the KS-test and might be caused by an approximation made in astropy.

Question 1.e)

Problem

Download the dataset. The dataset contains 10 sets of random numbers. Compare these 10 sets with your Gaussian pseudo random numbers and make the plot of the probabilities as in either of the previous two exercises (your choice). Which random number arrays is/are consistent with a Gaussian random numbers with $\sigma = 1$ and $\mu = 0$

Solution

The distributions are compared by performing the ks-test2. The random numbers are generated once and presorted to save computation time by performing the ks-test2 10 times. The code that contains the implementation of the ks-test2 can be found on page 31. The code that generates the plots and the generated plots can be found below.

The plots show that there is only one column which might be² a normal distribution with $\sigma = 1$ and $\mu = 0$. This is column 3 (figure ...). In all other plots the p-value drops and stays below $p = 0.05$ when including all samples, which indicates that their is enough statistical evidence to reject the hypothesis that they follow a normal distribution with $\sigma = 1$ and $\mu = 0$.

²A p-value only shows statistical evidence against the null hypothesis, it isn't a measure of how good the null hypothesis is.

Code - Plots

The code for generating the 10 plots.

```
1
2 def assignment_1e(random):
3     """
4         Execute assignment 1.e
5     Int:
6         param: random — An initialization of the random number generator.
7     """
8
9     # The relevant imports for this piece of code are:
10    # (1) matplotlib.pyplot as plt
11    # (2) numpy as np
12    # (3) scipy.stats as sp_stats
13    # (4) mathlib.statistics as ml_stats
14
15    # Load the data.
16    data = np.loadtxt('randomnumbers.txt')
17
18    # Generate the maximum amount of needed random numbers.
19    random_numbers = random.gen_normals(0, 1, int(1e5))
20
21    # The values to plot point for.
22    plot_values = np.array(10*np.arange(1, 5.1, 0.1), dtype=int)
23
24    # Pre-sort the random numbers
25    random_nums_sorted = list()
26
27    for idx, values in enumerate(plot_values):
28        random_nums_sorted.append(sorting.merge_sort(random_numbers[0:values]))
29
30
31    # Go over the columns and perform the KS-test2
32    for i in range(data.shape[1]):
33
34        # An array in which the p-values are stored for the self created
35        # ks-test2 and the scipy version.
36        p_values_self = np.zeros(len(plot_values))
37
38        # Calculate the p-values with the ks-test2
39        for idx, values in enumerate(plot_values):
40
41            # Perform the ks-test2 with the own implementation.
42            p_values_self[idx] = ml_stats.kstest2(data[:, i][0:values],
43                                                random_numbers[0:values],
44                                                random_nums_sorted[idx])
45
46        # Plot the p-values.
47        plt.plot(plot_values, p_values_self, label = 'self', color = 'orange')
48        plt.hlines(0.05, 0, 10*5, colors='red', linestyle='--')
49
50        plt.xlabel(r'Log($N_{samples}$)')
51        plt.ylabel('Probabillity (p-value)')
52        plt.xscale('log')
53        plt.legend()
54        plt.savefig("./Plots/1e-plot-column-{:0}.pdf".format(i))
```

./Code/assignment_1.py

Code - Output plot(s)

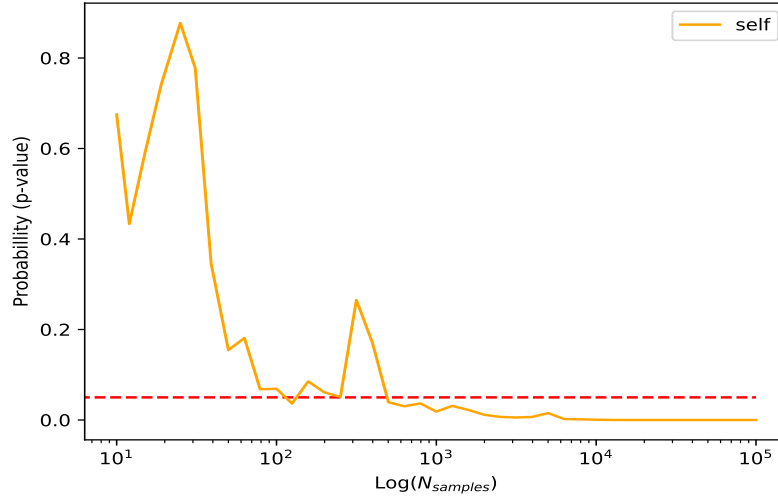


Figure 9: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **first** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

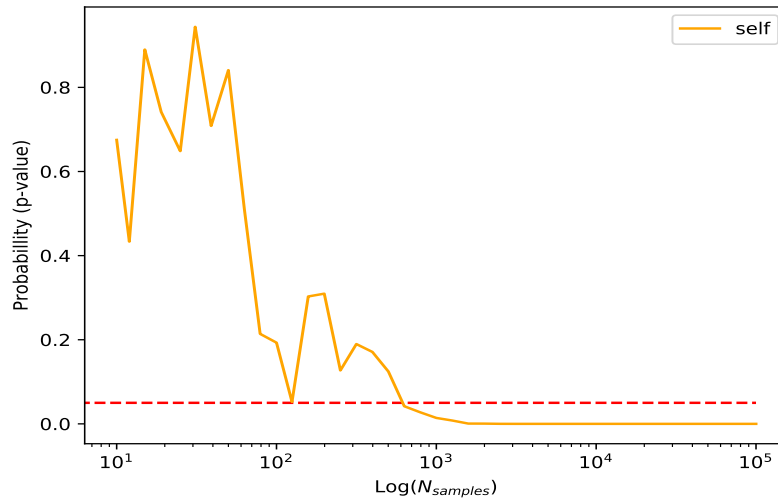


Figure 10: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **second** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

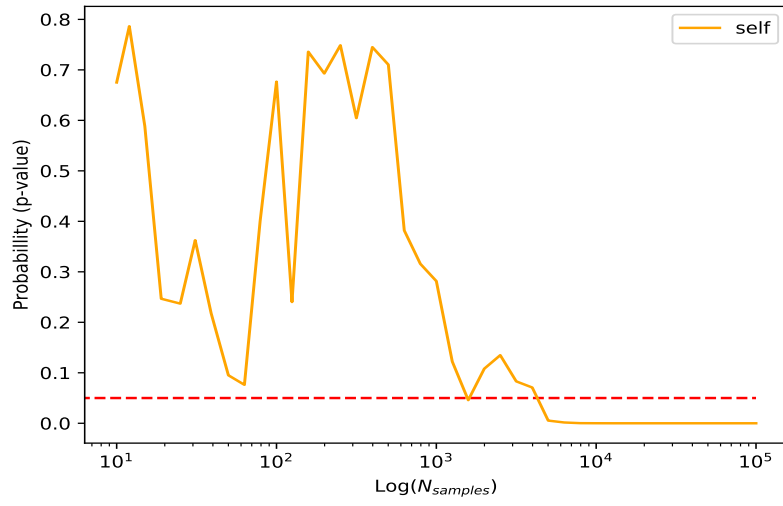


Figure 11: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **third** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

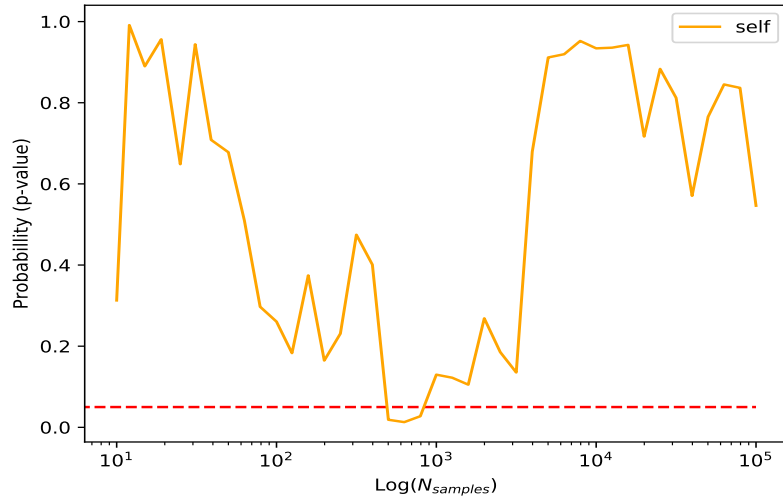


Figure 12: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **forth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 only between 500 – 1000 samples. In all other cases it passes the ks-test.

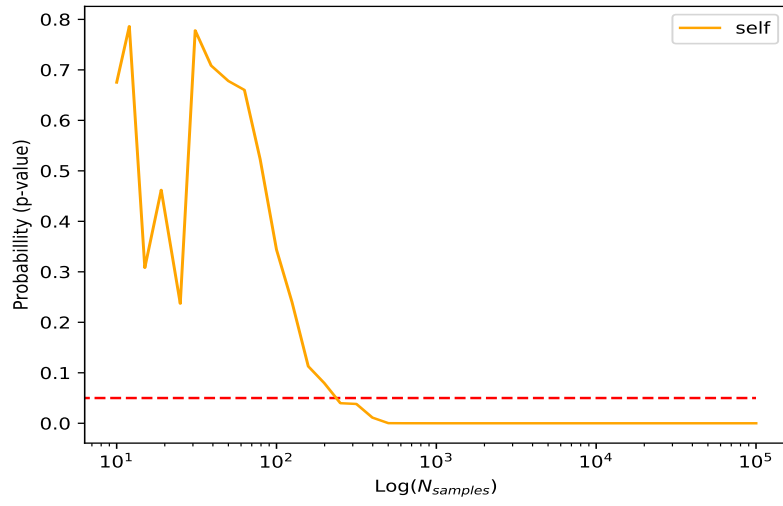


Figure 13: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **fifth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

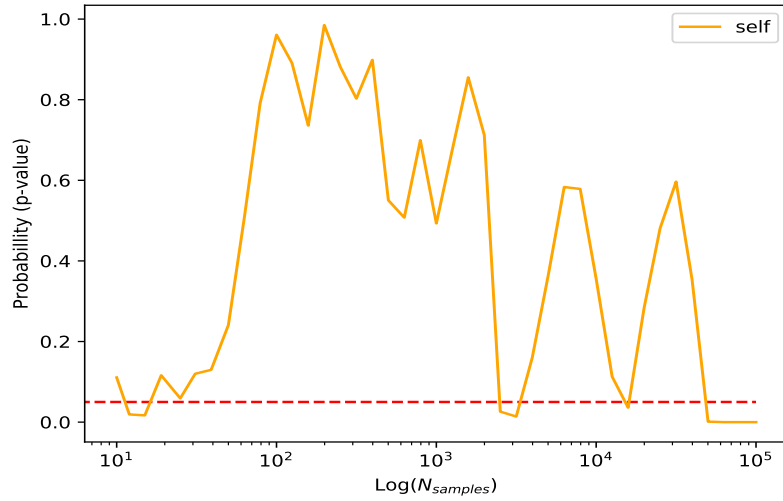


Figure 14: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **sixth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 and stays there when including halve of the samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

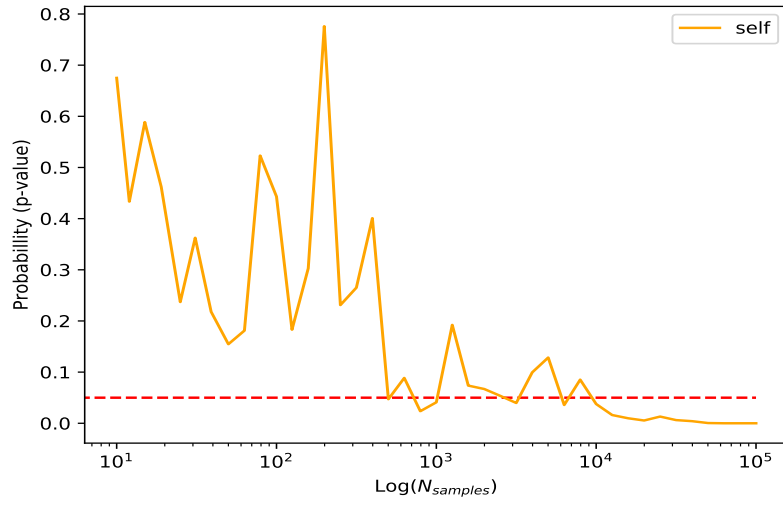


Figure 15: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **seventh** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

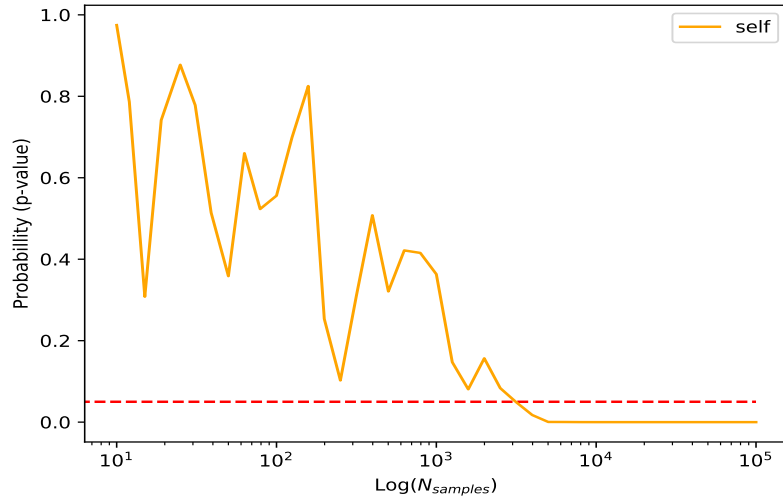


Figure 16: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **eight** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

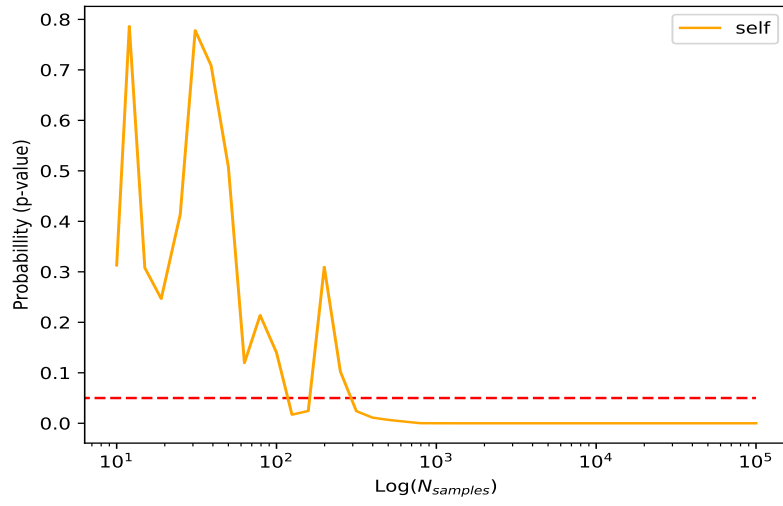


Figure 17: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **ninth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

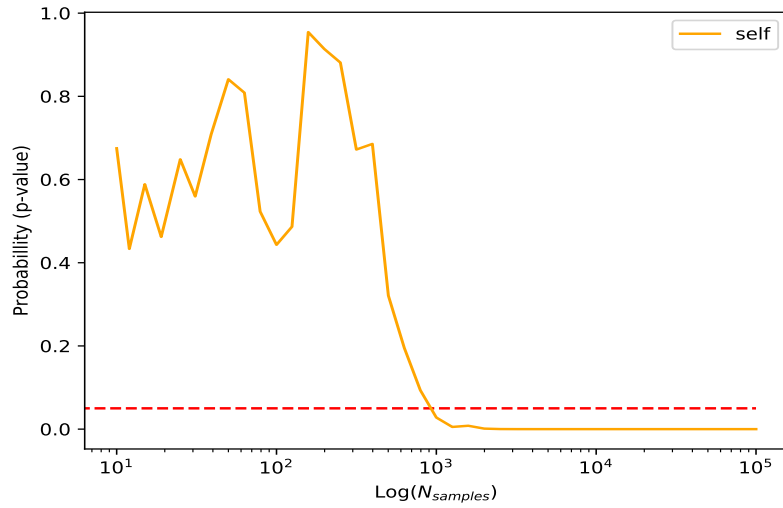


Figure 18: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **tenth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the p-value drops below 0.05 when including all samples. There is thus enough statistical evidence to reject the hypothesis that this column is normal distributed with $\mu = 0$ and $\sigma = 1$.

Question 1- Summary

Summary

The current sub-section contains the summary of the code used for assignment 1. This includes the file containing all sub-questions and all used shared modules. The shared modules include

Code - Assignment

The full code, inclusive the imports for the full assignment.

```
1 import astropy.stats
2 import matplotlib.pyplot as plt
3 import mathlib.random as random
4 import mathlib.sorting as sorting
5 import mathlib.statistics as ml_stats
6 import numpy as np
7 import scipy.stats as sp_stats
8
9 def main():
10
11     # Initialize the random number generator.
12     rng = random.Random(78379522)
13
14     # Run assignments
15     assignment_1a(rng)
16     assignment_1b(rng)
17     assignment_1c(rng)
18     assignment_1d(rng)
19     assignment_1e(rng)
20
21
22 def assignment_1a(random):
23     """
24     Execute assignment 1.a
25
26     Int:
27     param: random — An initialization of the random number generator.
28     """
29
30     # The relevant imports for this piece of code are:
31
32     # (1) matplotlib.pyplot as plt
33     # (2) mathlib.random as random
34     # (3) mathlib.stats as ml_stats
35     # (3) numpy as np
36
37     # Print the seed.
38     print('[1.a] Initial seed: ', random.get_seed())
39
40     # Generate 1000 numbers.
41     numbers_1000 = random.gen_uniforms(1000)
42
43     # Plot them against each other.
44     plt.scatter(numbers_1000[0:999], numbers_1000[1:], s=2)
45     plt.ylabel(r'Probability  $x_{i+1}$ ')
46     plt.xlabel(r'Probability  $x_i$ ')
47     plt.savefig('./Plots/1_plot_against.pdf')
48     plt.figure()
49
50     # Plot them against the index.
51     plt.plot(range(0, 1000), numbers_1000)
52     plt.ylabel('Probability p')
53     plt.xlabel('Index')
54     plt.savefig('./Plots/1_plot_index.pdf')
55     plt.figure()
56
57     # Create a histogram for 1e6 points with 20 bins of 0.05 wide.
58     numbers_mil = random.gen_uniforms(int(1e6))
59     plt.hist(numbers_mil, bins=20, range=(0,1), color='orange', edgecolor='black')
60     plt.ylabel('Counts')
61     plt.xlabel('Generate values')
```

```

61 plt.savefig('./Plots/1_hist_uniformnes.pdf')
62 plt.figure()
63
64 # Extra, to print the smallest and largest bin value.
65 counts, _ = np.histogram(numbers_mil, bins=20)
66 print('[1.a] Max counts: ', max(counts))
67 print('[1.a] Min counts: ', min(counts))
68
69 def assignment_1b(random):
70     """
71     Execute assignment 1.b
72     Int:
73     param: random — An instance of the random number generator.
74     """
75
76     # The relevant imports for this piece of code are:
77
78     # (1) matplotlib.pyplot as plt
79     # (2) mathlib.random as random
80     # (3) mathlib.stats as ml_stats
81     # (4) numpy as np
82
83     # Sigma and mean for the distribution.
84     mean = 3.0
85     sigma = 2.4
86
87     # Generate 1000 random normal variables for the given mean and sigma.
88     samples = random.gen_normals(mean, sigma, 1000)
89
90     # The true normal distribution for the given mean and sigma.
91     gaussian_x = np.linspace(-sigma*4 + mean, sigma*4 + mean, 1000)
92     gaussian_y = ml_stats.normal(gaussian_x, mean, sigma)
93
94     # Create a histogram.
95     plt.hist(samples, bins=20, density=True, edgecolor='black',
96             facecolor='orange', zorder=0.1, label='Sampled')
97     plt.plot(gaussian_x, gaussian_y, c='red', label='Normal')
98     plt.xlim(-sigma*6.5 + mean, sigma*6.5 + mean)
99     plt.ylim(0, max(gaussian_y)*1.2)
100
101     # Add the sigma lines.
102
103     # The hight of the sigma lines that need to be added.
104     lines_height = max(gaussian_y)*1.2
105
106     for i in range(1, 6):
107         # Absolute shift from the mean for the given sigma
108         shift = i*sigma
109
110         # Sigma right of the mean.
111         plt.vlines(mean + shift, 0, lines_height,
112                 linestyle='-', color='black', zorder=0.0)
113         plt.text(mean + shift - 0.4, lines_height/1.3, str(i) + r'$\sigma$',
114                 color='black', backgroundcolor='white', fontsize=9)
115
116         # Sigma line left of the mean.
117         plt.vlines(mean - shift, 0, lines_height, linestyle='-', zorder=0.0)
118         plt.text(mean - shift - 0.4, lines_height/1.3, str(i) + r'$\sigma$',
119                 color='black', backgroundcolor='white', fontsize=9)
120
121     plt.legend(framealpha=1.0)
122     plt.savefig('./Plots/1_hist_gaussian.pdf')
123     plt.figure()
124
125 def assignment_1c(random):
126     """
127     Execute assignment 1.c
128     Int:
129     param: random — An initialization of the random number generator.
130     """
131

```

```

132 # The relevant imports for this piece of code are:
133 # (1) matplotlib.pyplot as plt
134 # (2) numpy as np
135 # (3) astropy.stats
136 # (4) mathlib.statistics as ml_stats
137
138 # The values to plot point for.
139 plot_values = np.array(10*np.arange(1, 5.1, 0.1), dtype=int)
140
141 # An array in which the p-values are stored for the self created.
142 # ks-test and the scipy version.
143 p_values_self = np.zeros(len(plot_values))
144 p_values_scipy = np.zeros(len(plot_values))
145
146 # Generate the maximum amount of needed random numbers.
147 random_numbers = random.gen_normals(0, 1, int(1e5))
148
149 # Calculate the p-values with the ks-test.
150 for idx, values in enumerate(plot_values):
151
152     # Calculate the value with scipy.
153     p_values_scipy[idx] = sp_stats.kstest(random_numbers[0:values],
154                                           'norm')[1]
155
156     # Calculate the p-values with the own implementation.
157     p_values_self[idx] = ml_stats.kstest(random_numbers[0:values],
158                                         ml_stats.normal_cdf)
159
160 # Plot the probabilities for only my own implementation.
161 plt.plot(plot_values, p_values_self, label='self', color='orange')
162 plt.hlines(0.05, 0, 10*5, colors='red', linestyle='—')
163 plt.xscale('log')
164 plt.xlabel(r'Log($N_{samples}$)')
165 plt.ylabel('Probabillity (p-value)')
166 plt.legend()
167 plt.savefig('./Plots/1_plot_ks_test_self.pdf')
168 plt.figure()
169
170 # Plot the probabilities for both the scipy and my own implemntation.
171 plt.plot(plot_values, p_values_scipy, label='scipy', linestyle=':',
172          zorder=1.1)
173 plt.plot(plot_values, p_values_self, label='self', zorder=1.0,
174          color='orange')
175 plt.hlines(0.05, 0, 10*5, colors='red', linestyle='—')
176 plt.xscale('log')
177 plt.xlabel(r'Log($N_{samples}$)')
178 plt.ylabel('Probabillity (p-value)')
179 plt.legend()
180 plt.savefig('./Plots/1_plot_ks_test_self_scipy.pdf')
181 plt.figure()
182
183 def assignment_1d(random):
184     """
185     Execute assignment 1.d
186     Int:
187     param: random — An initialization of the random number generator.
188     """
189
190     # The relevant imports for this piece of code are:
191     # (1) matplotlib.pyplot as plt
192     # (2) numpy as np
193     # (3) scipy.stats as sp_stats
194     # (4) mathlib.statistics as ml_stats
195
196     # The values to plot point for.
197     plot_values = np.array(10*np.arange(1, 5.1, 0.1), dtype=int)
198
199     # Generate the maximum amount of needed random numbers.
200     random_numbers = random.gen_normals(0, 1, int(1e5))
201
202     # An array in which the p-values are stored for the self created

```

```

203 # kuiper-test and the astropy version.
204 p_values_self = np.zeros(len(plot_values))
205 p_values_astropy = np.zeros(len(plot_values))
206
207 # Calculate the p-values with the ks-test
208 for idx, values in enumerate(plot_values):
209
210     # Calculate the value with the own implemnetation
211     p_values_self[idx] = ml_stats.kuiper_test(random_numbers[0:values],
ml_stats.normal_cdf)
212     # Calculate the value with astropy.
213     p_values_astropy[idx] = astropy.stats.kuiper(random_numbers[0:values],
ml_stats.normal_cdf)[1]
214
215
216 # Plot the probabilities for only my own implementation
217 plt.plot(plot_values, p_values_self, label='self')
218 plt.hlines(0.05,0,10**5,colors='red',linestyles='—')
219 plt.xscale('log')
220 plt.xlabel(r'Log($N_{samples}$)')
221 plt.ylabel('Probabillity (p-value)')
222 plt.legend()
223 plt.savefig('./Plots/1_plot_kuiper_test_self.pdf')
224 plt.figure()
225
226 # Plot the probabilitéis with both the own implementation and astropy
227 plt.plot(plot_values, p_values_astropy, label='astropy', linestyle=':',
zorder=1.1)
228 plt.plot(plot_values, p_values_self, label='self',zorder=1.0)
229 plt.hlines(0.05,0,10**5,colors='red',linestyles='—')
230 plt.xscale('log')
231
232 plt.xlabel(r'Log($N_{samples}$)')
233 plt.ylabel('Probabillity (p-value)')
234 plt.legend()
235 plt.savefig('./Plots/1_plot_kuiper_test_self_astropy.pdf')
236 plt.figure()
237
238 def assignent_1e(random):
239     """
240     Execute assignent 1.e
241     Int:
242     param: random — An initialization of the random number generator.
243     """
244
245     # The relevant imports for this piece of code are:
246     # (1) matplotlib.pyplot as plt
247     # (2) numpy as np
248     # (3) scipy.stats as sp_stats
249     # (4) mathlib.statistics as ml_stats
250
251     # Load the data.
252     data = np.loadtxt('randomnumbers.txt')
253
254     # Generate the maximum amount of needed random numbers.
255     random_numbers = random.gen_normals(0, 1, int(1e5))
256
257     # The values to plot point for.
258     plot_values = np.array(10*np.arange(1, 5.1, 0.1), dtype=int)
259
260     # Pre-sort the random numbers
261     random_nums_sorted = list()
262
263     for idx, values in enumerate(plot_values):
264         random_nums_sorted.append(sorting.merge_sort(random_numbers[0:values]))
265
266
267 # Go over the columns and perform the KS-test2
268 for i in range(data.shape[1]):
269
270     # An array in which the p-values are stored for the self created
271     # ks-test2 and the scipy version.

```

```

272     p_values_self = np.zeros(len(plot_values))
273
274     # Calculate the p-values with the ks-test2
275     for idx, values in enumerate(plot_values):
276
277         # Perform the ks-test2 with the own implementation.
278         p_values_self[idx] = ml_stats.kstest2(data[:, i][0: values],
279                                             random_numbers[0: values],
280                                             random_nums_sorted[idx])
281
282     # Plot the p-values.
283     plt.plot(plot_values, p_values_self, label = 'self', color = 'orange')
284     plt.hlines(0.05, 0, 10**5, colors='red', linestyle='—')
285
286     plt.xlabel(r'Log($N_{samples}$)')
287     plt.ylabel('Probabillity (p-value)')
288     plt.xscale('log')
289     plt.legend()
290     plt.savefig("./Plots/1e-plot_column_{0}.pdf".format(i))
291     plt.figure()
292
293 if __name__ == '__main__':
294     main()

```

./Code/assigment_1.py

Random Number Generator

```

1 import numpy as np
2
3 class Random(object):
4     """
5     A class representing a random number generator (RNG)
6     """
7
8     def __init__(self, seed):
9         """
10         Create a new instance of the random number generator.
11
12         In:
13         param: seed — The seed of the random number generator.
14                     This must be a positive integer.
15
16         """
17
18         # The seed and state of the generator
19         self._seed = np.uint64(seed)
20         self._state = self._seed
21
22         # maximum uint32 value
23         self._uint32_max = np.uint64(0xFFFFFFFF)
24
25         # The values for the Xor shift.
26         self._xor_a1 = np.uint64(20)
27         self._xor_a2 = np.uint64(41)
28         self._xor_a3 = np.uint64(5)
29
30         # The values for the multiply with carry.
31         self._mwc_a = np.uint64(4294957665)
32         self._mwc_base = np.uint64(2**32)
33
34     def get_seed(self):
35         """
36         Get the seed that is used to initialize this generator.
37
38         Out:
39         return: The seed used to initialize the generator.
40         """
41         return self._seed
42
43     def get_state(self):
44         """
45         Get the state of the generator.
46
47         Out:
48         return: The state of the generator.
49         """
50         return self._state
51
52     def gen_next_int(self):
53         """
54         Generate a new random 32-bit unsigned integer.
55
56         Out:
57         return: A random 32-bit unsigned integer.
58         """
59
60         # The state is at the end updated with mwc.
61         # We therefore shouldn't use more than 32 bits to generate
62         # the number.
63
64         return self._update_state() & self._uint32_max
65
66     def gen_uniform(self):
67         """

```



```

68         Generate a random float between 0 and 1.
69     Out:
70         return: A random float between 0 and 1.
71     """
72
73     return self.gen_next_int()*1.0 / self._uint32_max
74
75 def gen_uniforms(self, amount):
76     """
77         Generate multiple random floats
78         between 0 and 1.
79     In:
80         param: amount — The amount of floats to generate.
81     Out:
82         return: An array with 'amount' random floats
83                between 0 and 1.
84     """
85
86     samples = np.zeros(amount)
87
88     for i in range(amount):
89         samples[i] = self.gen_uniform()
90
91     return samples
92
93 def gen_normal(self, mean, sigma):
94     """
95         Generate a random normal distributed float.
96
97     In:
98         param: mean — The mean of the gaussian distribution.
99         param: sigma — The squareroot of the variance of the distribution.
100    Out:
101        return: A random float that is drawn from the parameterized normal
102                distribution.
103    """
104
105    # Generate two uniform variables.
106    u1 = self.gen_uniform()
107    u2 = self.gen_uniform()
108
109    # Use the box muller transformation.
110    return sigma*np.sqrt(-2* np.log(1-u1))*np.cos(2*np.pi*u2) + mean
111
112 def gen_normal_uniform(self, mean, sigma, u1, u2):
113     """
114         Generate a random normal distributed float from two provided
115         uniform variables.
116
117     """
118
119     pre_factor= sigma*np.sqrt(-2* np.log(1-u1))
120
121     return pre_factor*np.cos(2*np.pi*u2) + mean, pre_factor*np.sin(2*np.pi*u2
122 ) + mean
123
124 def gen_normals(self, mean, sigma, amount):
125     """
126         Generate multible random normal distributed float.
127
128     In:
129         param: mean — The mean of the gaussian distribution.
130         param: sigma — The squareroot of the variance of the distribution.
131         param: amount — The amount of floats to generate.
132     Out:
133         return: An array with random floats drawn from the parameterized
134                normal
135                distribution.
136     """
137
138     # Pre-factors in the box muller transformation.

```

```

137     square_pre_factor = -2*sigma**2
138     angle_pre_factor = 2*np.pi
139
140     # With the Box-muller two random normals can be generated for two
141     # uniforms. If the amount of requested variables is odd then add
142     # one to it and later remove it when returning the result.
143     elements = amount if amount % 2 == 0 else amount + 1
144
145     # Array in which the drawn normal distributed variables are stored.
146     normal_dist = np.zeros(elements)
147
148     # Apply the box muller transformation to generate the samples.
149     for i in range(0, elements, 2):
150
151         # Generate the uniforms.
152         u1 = self.gen_uniform()
153         u2 = self.gen_uniform()
154
155         # Calculate common terms.
156         pre_fact = np.sqrt(square_pre_factor*np.log(1-u1))
157
158         # Calculate the samples.
159         normal_dist[i] = pre_fact*np.cos(angle_pre_factor*u2) + mean
160         normal_dist[i+1] = pre_fact*np.sin(angle_pre_factor*u2) + mean
161
162     # If amount is odd, don't return the last element.
163     return normal_dist[0:amount]
164
165 def _update_state(self):
166     """
167     Update the state of the random number generator.
168
169     Out:
170     return: The new state of the random number generator.
171     """
172
173     self._state = self._xor_shift(self._state)
174     self._state = self._mwc(self._state & self._uint32_max) ^ self._state
175
176     return self._state
177
178
179 def _xor_shift(self, number):
180     """
181     Execute the XOR-shift algorithm on the
182     input number.
183
184     In:
185     param: number — The number to XOR-shift.
186
187     Out:
188     return: The number produced by XOR-shift.
189     """
190
191     # Shift to the right and then bitwise xor.
192     number ^= (number >> self._xor_a1)
193     # Shift to the left and then bitwise xor.
194     number ^= (number << self._xor_a2)
195     # Shift to the right and then bitwise xor.
196     number ^= (number >> self._xor_a3)
197
198     return number
199
200
201 def _mwc(self, number):
202     """
203     Perform multiply with carry (MWC) on
204     the given input.
205
206     In:
207     param: number — The number to perform MWC on, must be an uint64.
208
209     Out:
210     return: The new number.
211     """
212
213     return self._mwc_a * (number & (self._uint32_max - np.uint64(1))) + (

```

```
number >> np.uint64(32))
```

./Code/mathlib/random.py

Statistical functions

The code containing all statistical functions that were needed for the sub-questions.

```
1 import numpy as np
2 import mathlib.sorting as sorting
3
4
5 def kstest(x, cdf):
6     """
7     Perform the Kolmogorov-Smirnov test for goodness of fit
8     and return the p-value.
9     In:
10     param: x    — An array with values whose CDF is expected to be
11                   the same as the provided CDF. Must be at least size 4
12
13     param: cdf — A function that is the expected cdf under the null
14                   hypothesis.
15     Out:
16     return: The p-value obtained by performing the KS-test.
17     """
18
19     # Amount of values in the input array.
20     x_size = len(x)
21
22     # Sort the values and evaluate the cdf.
23     x_sorted = sorting.merge_sort(x)
24     x_sorted_cdf = cdf(x_sorted)
25
26     # Maximum distance.
27     max_dist = 0
28
29     # Value of the empirical cdf at step i-1.
30     x_cdf_emperical_previous = 0
31
32     # Find the maximum distance.
33     for idx in range(0, x_size):
34
35         # Calculate the empirical cdf.
36         x_cdf_emperical = (idx+1)/x_size
37         # The true cdf evaluation at the given point.
38         x_cdf_true = x_sorted_cdf[idx]
39
40         # Find the distance. The empirical
41         # CDF is a step function so there are two distances
42         # that need to be checked at each step.
43
44         # Calculate the two distances
45         distance_one = abs(x_cdf_emperical - x_cdf_true)
46         distance_two = abs(x_cdf_emperical_previous - x_cdf_true)
47
48         # Find the maximum of those two distances and
49         # check if it is larger than the current known maximum distance.
50         max_dist = max(max_dist, max(distance_one, distance_two))
51
52         # Save the current value of the empirical cdf.
53         x_cdf_emperical_previous = x_cdf_emperical
54
55     # Calculate the p-value with the help of the CDF.
56     sqrt_elements = np.sqrt(x_size)
57     cdf = _ks_statistic_cdf((sqrt_elements + 0.12+0.11/sqrt_elements)*max_dist)
58     return 1 - cdf
59
60
61 def kstest2(x1, x2, x2_sorted = None):
62     """
63     Perform the Kolmogorov-Smirnov test for goodness of fit
```

```

64         and return the p-value.
65 In:
66     param: x1    — An array with value's who's CDF is expected to be
67                   the same as the CDF of the provided values.
68                   Must be atleast size 4.
69
70     param: x2 — A discretized pdf of the expected distribution under the
71     null hypothesis.
72 Out:
73     return: The p-value obtained by performing the KS-test
74     """
75
76     # Amount of values in the input distributions.
77     x1_size = len(x1)
78     x2_size = len(x2)
79
80     # Sort both arrays.
81     x1 = sorting.merge_sort(x1)
82     x2 = sorting.merge_sort(x2) if type(x2_sorted) is not None else x2_sorted
83
84     # The maximum distance
85     max_dist = 0
86
87     # The iteration values used to determine
88     # the emperical pdf's and the max distance.
89     x1_i, x2_j = 0,0
90
91     # Find the maximum distance by updating the emperical CDFs.
92     while x1_i < x1_size and x2_j < x2_size:
93
94         # Update the indices used for the emperical CDF's.
95
96         if x1[x1_i] < x2[x2_j]:
97             x1_i += 1
98         else:
99             x2_j += 1
100
101         # Find the max distance
102         max_dist = max(abs(x1_i/x1_size-x2_j/x2_size), max_dist)
103
104     sqrt_factor = np.sqrt((x1_size*x2_size)/(x1_size+x2_size))
105     cdf = _ks_statistic_cdf((sqrt_factor + 0.12+0.11/sqrt_factor)*max_dist)
106
107     return 1 - cdf
108
109 def kuiper_test(x, cdf):
110     """
111     Perform the Kuiper test for goodness of fit
112     and return the p-value.
113 In:
114     param: x    — An array with value's who's CDF is expected to be
115                   the same as the provided CDF. Must be atleast size 4
116
117     param: cdf — A function that is the expected cdf under the null
118     hypothesis.
119 Out:
120     return: The p-value obtained by performing the kuiper-test
121     """
122
123     # Sort the data in ascending order, calculate the
124     # cdf and the emperical cdf for the sorted values and
125     # save the total amount of elements we have.
126     x_sorted = sorting.merge_sort(x)
127     x_sorted_cdf = cdf(x_sorted)
128     x_elements = len(x)
129
130     # Find the maximum distance above and below
131     # the true cdf.
132     max_dist_above = 0
133     max_dist_below = 0

```

```

133 # Value of the cdf at step i-1.
134 x_cdf_emperical_previous = 0
135
136
137 for idx, x in enumerate(x_sorted):
138
139     # Calculate the emperical cdf.
140     x_cdf_emperical = (idx+1)/x_elements
141     # Calculate the true cdf.
142     x_cdf_true = x_sorted_cdf[idx]
143
144     # Find the maximum distance above and below
145     max_dist_above = max(x_cdf_emperical - x_cdf_true, max_dist_above)
146     max_dist_below = max(x_cdf_true - x_cdf_emperical_previous,
147                           max_dist_below)
148
149     # Update previous cdf
150     x_cdf_emperical_previous = x_cdf_emperical
151
152 sqrt_elem = np.sqrt(x_elements)
153 v = max_dist_above + max_dist_below
154 cdf = _kuiper_statistic_cdf((sqrt_elem + 0.155+0.24/sqrt_elem)*v)
155
156 return 1 - cdf
157
158 def _ks_statistic_cdf(z):
159     """
160     An approximation for the cdf of the
161     Kolmogorov-Smirnov (KS) test staistic.
162     In:
163         param: z — The value to calculate the cdf at.
164     Out:
165         return: An approximation of the cdf for the given value.    print(
166         max_dist_above + max_dist_below)
167     """
168     # Numerical approximation taken from:
169     # Numerical methods — The art of scientific computation.
170     # Third edition.
171
172     if z < 1.18:
173         exponent = np.exp(-np.pi**2/(8*z**2))
174         pre_factor = np.sqrt(2*np.pi)/z
175
176         return pre_factor*exponent*(1+ exponent**8)*(1+exponent**16)
177     else:
178         exponent = np.exp(-2*z**2)
179         return 1-2*exponent*(1-exponent**3)*(1-exponent**6)
180
181 def _kuiper_statistic_cdf(z):
182     """
183     An approximation for the cdf of the
184     Kuiper test statistic
185     In:
186         param: z — The value to calculate the cdf at.
187     Out:
188         return: An approximation of the cdf for the given value.
189     """
190
191     # Value of z is to small, sum will be 1 up to 7 digits
192     if z < 0.4:
193         return 1
194
195     # Approximateed value of the sum by performing 100 iterations
196
197     # The value to return
198     ret = 0
199     # A term often needed in the sum.
200     z_squared = z**2
201

```

```

202     # Evaluate the first 100 terms in the sum.
203     for j in range(1, 100):
204         power = j**2 * z_squared
205         ret += (4 * power - 1)*np.exp(-2*power)
206
207     return 1- 2*ret
208
209
210
211 def normal_cdf(x, mean = 0, sigma = 1):
212     """
213         Evaluate the cummulative normal distribution for
214         the given parameters
215     In:
216         param: x — The point to evaluate the cdf at or an array of points to
217         evaluate it for.
218         param: mean — The mean of the normal distribution.
219         param: sigma — The square root of the variance for the normal
220         distribution.
221     Out:
222         return: The cummulative normal distribution evaluated at.
223     """
224     # Calculate the CDF using the erf function (defined below).
225     return 0.5 + 0.5*erf((x-mean)/(np.sqrt(2)*sigma))
226
227 def normal(x, mean = 0, sigma = 1):
228     """
229         Evaluate the normal distribution for the given
230         parameters.
231     In:
232         param: x — The point to evaulte the distribution at.
233         param: mean — The mean of the distribution.
234         param: sigma — The square root of the variance for the distribution.
235     Out:
236         return: The value of the parameterized distribution evaluated at the
237         given point.
238     """
239     return 1/((np.sqrt(2*np.pi)*sigma))*np.exp(-0.5*((x - mean)/sigma)**2)
240
241 def erf(x):
242     """
243         Evaluate the erf function for a value of x.
244     In:
245         param: x — The value to evaluate the erf function for.
246     Out:
247         return: The erf function evaluated for the given value of x.
248     """
249
250     # Numerical approximation taken from Abramowits and Stegun.
251
252     # Constants for the numerical approximation
253     p = 0.3275911
254     a1 = 0.254829592
255     a2 = -0.284496736
256     a3 = 1.421413741
257     a4 = -1.453152027
258     a5 = 1.061405429
259
260     # Array in which the result is stored
261     ret = np.zeros(len(x))
262
263     # The approximation functions
264     erf_func_t_val = lambda x: 1/(1+ p*x)
265     erf_func_approx = lambda t, x : 1 - t*(a1 + t*(a2 + t*(a3 + t*(a4 + t*a5))))
266     *np.exp(-x**2)
267
268     # Evaluate for both positive and negative
269     neg_mask = x < 0

```

```

269     neg_x = x[neg_mask]
270     pos_mask = x >= 0
271     pos_x = x[pos_mask]
272
273     ret[neg_mask] = -erf_func_approx(erf_func_t_val(-neg_x), -neg_x)
274     ret[pos_mask] = erf_func_approx(erf_func_t_val(pos_x), pos_x)
275
276     return ret

```

./Code/mathlib/statistics.py

Sorting

The sorting algorithm used to sort the input of the KS-test and Kuiper-test.

```

1  import numpy as np
2
3  def merge_sort(array):
4      """
5          Sort an array using merge sort
6
7          In:
8              param: array — The array to sort.
9          Out:
10             return: The sorted array.
11         """
12
13         # Get the length of the array.
14         size = len(array)
15
16         # If the length is 1 then return the input array.
17         # This is an important check as this function is called
18         # recursively.
19         if size == 1:
20             return array
21
22         # Split the array in an sorted left and right segment.
23         left_sorted = merge_sort(array[:size >> 1])
24         right_sorted = merge_sort(array[size >> 1:])
25         left_sorted_len = len(left_sorted)
26         right_sorted_len = len(right_sorted)
27
28         #
29         # Merge the left and right array.
30         #
31
32         # The final sorted array.
33         result = np.zeros(size)
34
35         # Current index in the left sorted array.
36         left_idx = 0
37         # Current index in the right sorted array.
38         right_idx = 0
39         # Current index in the final sorted array.
40         result_idx = 0
41
42         # While we didn't fill the result array.
43         while result_idx < size:
44
45             # Element from left array is smaller, insert it and increase position.
46             if left_sorted[left_idx] < right_sorted[right_idx]:
47                 result[result_idx] = left_sorted[left_idx]
48                 left_idx += 1
49                 result_idx += 1
50             # Element from right array is smaller, insert it and increase position.
51             else:
52                 result[result_idx] = right_sorted[right_idx]
53                 right_idx += 1
54                 result_idx += 1
55
56         # Only right array has elements left, insert the remaining elements.
57         if left_idx == left_sorted_len:

```

```
58         result[result_idx:] = right_sorted[right_idx:]
59         break
60
61     # Only left has elements left, insert the remaining elements
62     if right_idx == right_sorted_len:
63         result[result_idx:] = left_sorted[left_idx:]
64         break
65
66     return result
```

./Code/mathlib/sorting.py

2 - Normally distributed pseudo-random numbers

Question 2

Problem

Make a Gaussian field using the provided information.

Solution

43