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.

```python
def assigment_1a(random):
    """
        Execute assigment 1.a
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:

    # (1) matplotlib.pyplot as plt
    # (2) mathlib.random as random
    # (3) mathlib.stats as ml_stats
    # (3) numpy as np

    # Print the seed.
    print('[1.a] Initial seed: ', random.get_seed())

    # Generate 1000 numbers.
    numbers_1000 = random.gen_uniforms(1000)

    # Plot them agianst each other.
    plt.scatter(numbers_1000[0:999], numbers_1000[1:], s=2)
    plt.ylabel(r'Probability $x_{i+1}$')
    plt.xlabel(r'Probability $x_{i}$')
    plt.savefig('./Plots/1_plot_against.pdf')
    plt.figure()

    # Plot them against the index.
    plt.plot(range(0, 1000), numbers_1000)
    plt.ylabel('Probability p')
    plt.xlabel('Index')
    plt.savefig('./Plots/1_plot_index.pdf')
    plt.figure()

    # Create a histogram for 1e6 points with 20 bins of 0.05 wide.
    numbers_mil = random.gen_uniforms(int(1e6))
    plt.hist(numbers_mil, bins=20, range=(0,1), color='orange',edgecolor='black')
    plt.ylabel('Counts')
    plt.xlabel('Generate values')
    plt.savefig('./Plots/1_hist_uniformnes.pdf')
    plt.figure()

    # Extra, to print the smallest and lagest bin value.
    counts, _ = np.histogram(numbers_mil, bins=20)
    print('[1.a] Max counts: ', max(counts))
```

./Code/assigment_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.a] Initial seed:  78379522
[1.a] Max counts:  50343
[1.a] Min counts:  49557
```

./Output/assigment1_out.txt

Code - Output plots

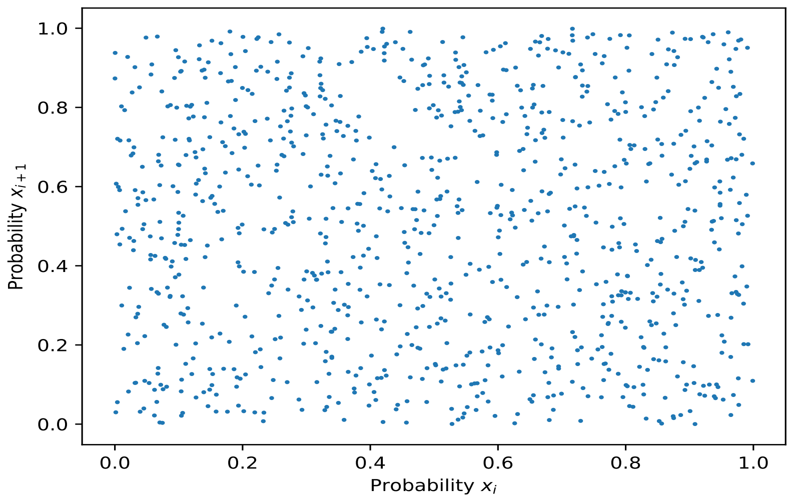

Figure 1: A plot of random number x_{i+1} against random number x_i for the first 1000 random uniforms produced by the random number generator. A good random number generator should produce a homogeneous plot without many (large) empty spots. The largest empty spot in the above plot is at $x_i = 0.4$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

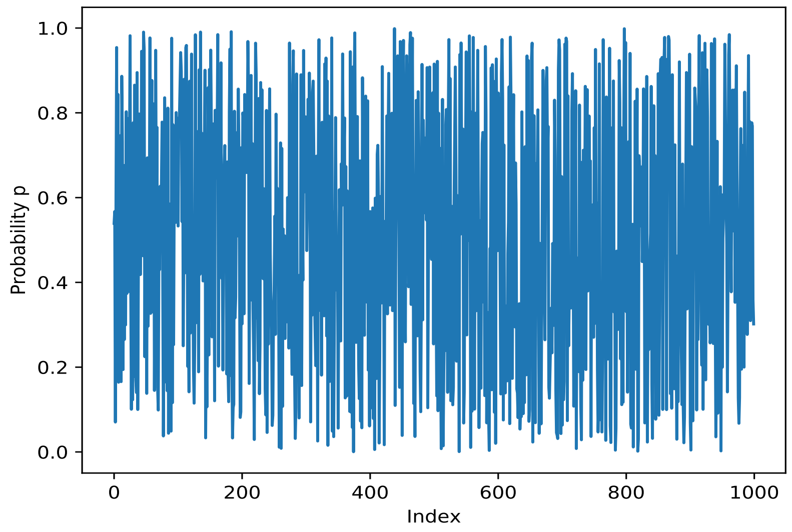

Figure 2: The first 1000 random uniform numbers produced by the random number generator (RNG) against their index. A good random number generator should not have large wide gaps (e.g when moving from index 400 to 450 it should not only produce values larger than 0.8, which would leave a wide gap). In the plot small gaps appear, see for example index $\sim$ 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 $p = 0.2$.

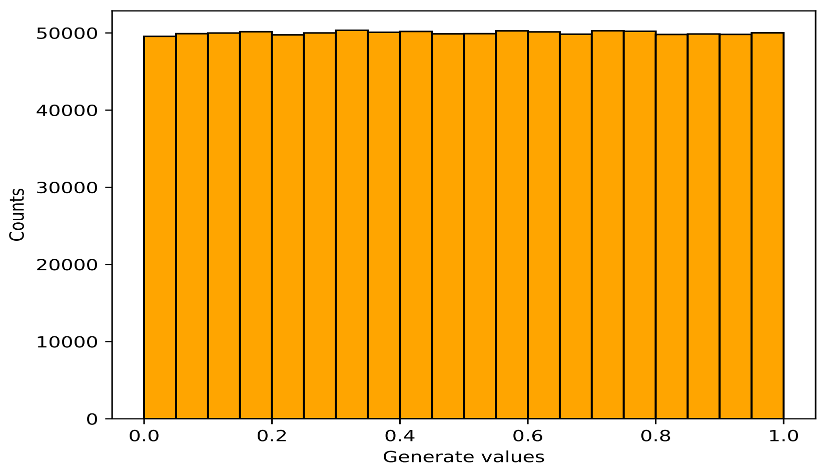

Figure 3: The uniforms of the random number generator for 1 million random values. The values are binned in 20 bins. A good random number generator should fluctuate around 50000 $\pm 2\sqrt{50000} = 50000 \pm 447$ counts per bin (2 sigma). The maximum and minimum amount of counts corresponds to 50343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

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

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

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

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

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

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

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,

$$P(R \leq r_1) = P(U_1 \leq u_1) \rightarrow 1 - e^{-\frac{r_1^2}{2\sigma^2}} = \int_0^{u1} du_1 = u_1 \tag{4}$$

$$P(\Theta \leq \theta) = P(U_2 \leq u_2) \rightarrow \frac{\theta_1}{2\pi} = \int_0^{u2} du_2 = u_2 \tag{5}$$

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

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

$$\theta_1 = 2\pi u_2 \tag{7}$$

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;

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

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

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.

```
def assigment_1b(random):
    """
        Execute assigment 1.b
    Int:
        param: random -- An instance of the random number generator.
    """

    # The relevant imports for this piece of code are:

    # (1) matplotlib.pyplot as plt
    # (2) mathlib.random as random
    # (3) mathlib.stats as ml_stats
    # (4) numpy as np

    # Sigma and mean for the distribution.
    mean = 3.0
    sigma = 2.4

    # Generate 1000 random normal variables for the given mean and sigma.
    samples = random.gen_normals(mean, sigma, 1000)

    # The true normal distribution for the given mean and sigma.
    gaussian_x = np.linspace(-sigma*4 +mean, sigma*4 +mean, 1000)
    gaussian_y = ml_stats.normal(gaussian_x, mean, sigma)

    # Create a histogram.
    plt.hist(samples, bins=20, density=True, edgecolor='black',
                      facecolor='orange', zorder=0.1, label='Sampled')
    plt.plot(gaussian_x, gaussian_y, c='red', label='Normal')
    plt.xlim(-sigma*6.5 + mean,sigma*6.5 + mean)
    plt.ylim(0, max(gaussian_y)*1.2)

    # Add the sigma lines.

    # The hight of the sigma lines that need to be added.
    lines_height = max(gaussian_y)*1.2

    for i in range(1, 6):
        # Absolute shift from the mean for the given sigma
        shift = i*sigma

        # Sigma right of the mean.
        plt.vlines(mean + shift, 0, lines_height,
                      linestyles='-', color='black', zorder=0.0)
        plt.text(mean + shift -0.4, lines_height/1.3, str(i) + r'$\sigma$',
                 color='black', backgroundcolor='white', fontsize=9)
```

./Code/assigment_1.py

Code - Output plot(s)

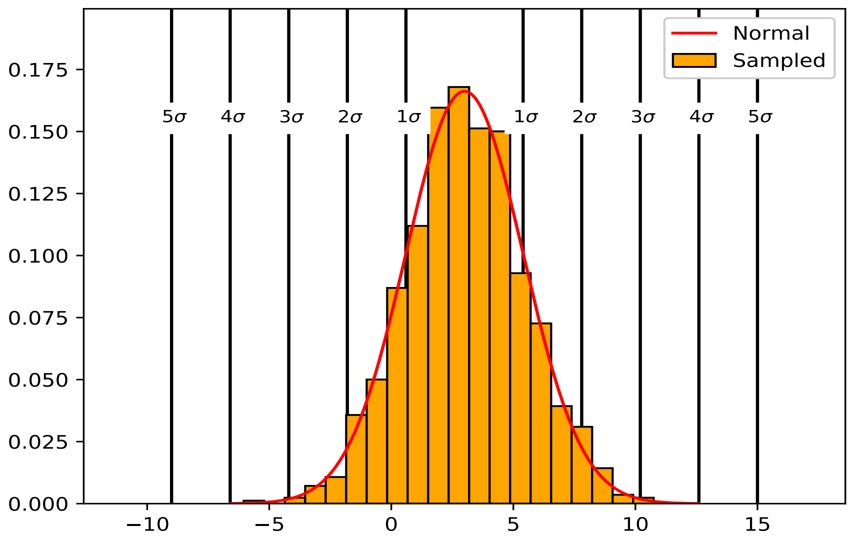

Figure 4: A histogram of the 1000 random nomal distributed variables generated with the box muller method for $\mu = 3$ and $\sigma = 2.4$ (orange). The red line is the true normal distribution for these values of μ and $\sigma$. The histogram appears to approximate the distribution quite well, but displays 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 to 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,

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

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*,

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

(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,

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

where the erf is given by,

$$\text{erf}(x)\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt \tag{13}$$

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*,

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

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.

```python
def assigment_1c(random):
    """
        Execute assigment 1.c
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:
    # (1) matplotlib.pyplot as plt
    # (2) numpy as np
    # (3) astropy.stats
    # (4) mathlib.statistics as ml_stats

    # The values to plot point for.
    plot_values = np.array(10**np.arange(1, 5.1, 0.1),dtype=int)

    # An array in which the p-values are stored for the self created.
    # ks-test and the scipy version.
    p_values_self = np.zeros(len(plot_values))
    p_values_scipy = np.zeros(len(plot_values))

    # Generate the maximum amount of needed random numbers.
    random_numbers = random.gen_normals(0, 1, int(1e5))

    # Calculate the p-values with the ks-test.
    for idx, values in enumerate(plot_values):

        # Calculate the value with scipy.
        p_values_scipy[idx] = sp_stats.kstest(random_numbers[0:values],
                                              'norm')[1]
        # Calculate the p-values with the own implementation.
        p_values_self[idx] = ml_stats.kstest(random_numbers[0:values],
                                             ml_stats.normal_cdf)


    # Plot the probabilities for only my own implementation.
    plt.plot(plot_values, p_values_self, label = 'self', color='orange')
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')
    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_ks_test_self.pdf')
    plt.figure()

    # Plot the probabilities for both the scipy and my own implemntation.
    plt.plot(plot_values, p_values_scipy, label='scipy', linestyle=':',
                                                    zorder=1.1)
    plt.plot(plot_values, p_values_self, label='self', zorder=1.0,
                                                    color='orange')
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')
    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_ks_test_self_scipy.pdf')
```

./Code/assigment_1.py

Code - Output plot(s)

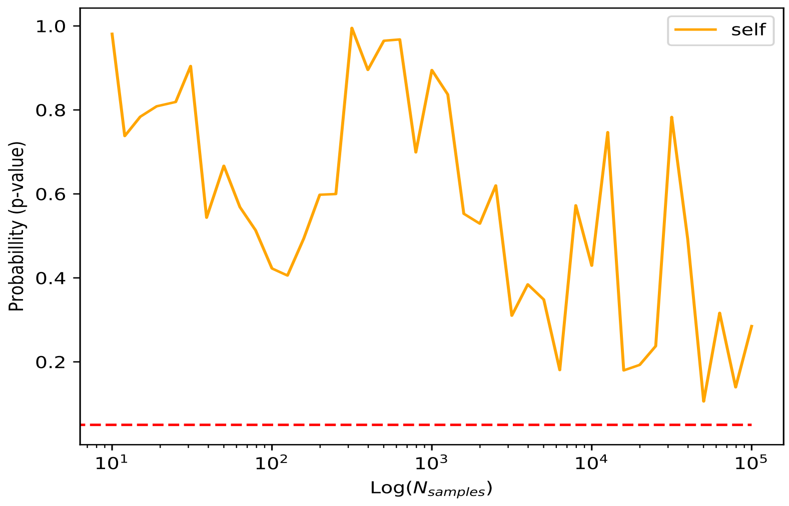

Figure 5: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** the line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the RNG always passes KS-test up to atleast 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.

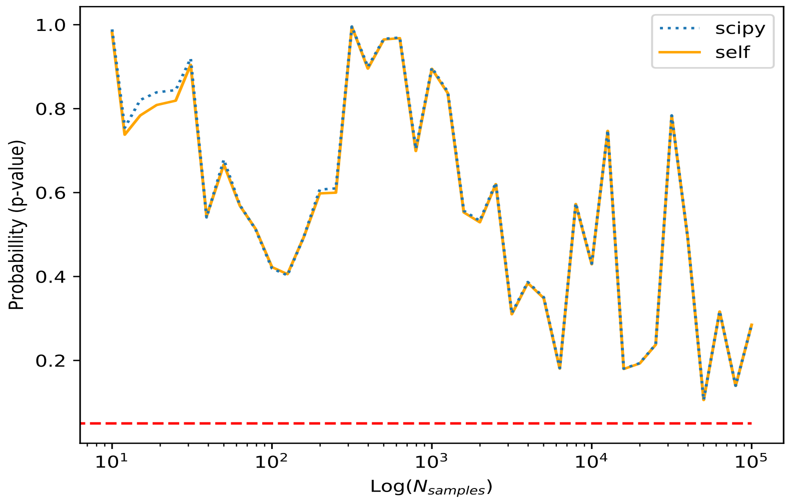

Figure 6: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. The orange line is the self written implementation of the KS-test and the blue line is the scipy version. A point **below** the red line would suggests 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,

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

The sum is 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.

```python
def assigment_1d(random):
    """
        Execute assigment 1.d
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:
    # (1) matplotlib.pyplot as plt
    # (2) numpy as np
    # (3) scipy.stats as sp_stats
    # (4) mathlib.statistics as ml_stats

    # The values to plot point for.
    plot_values = np.array(10**np.arange(1, 5.1, 0.1), dtype=int)

    # Generate the maximum amount of needed random numbers.
    random_numbers = random.gen_normals(0, 1, int(1e5))

    # An array in which the p-values are stored for the self created
    # kuiper-test and the astropy version.
    p_values_self = np.zeros(len(plot_values))
    p_values_astropy = np.zeros(len(plot_values))

    # Calculate the p-values with the ks-test
    for idx, values in enumerate(plot_values):

        # Calculate the value with the own implemnetation
        p_values_self[idx] = ml_stats.kuiper_test(random_numbers[0:values],
    ml_stats.normal_cdf)
        # Calculare the value with astropy.
        p_values_astropy[idx] = astropy.stats.kuiper(random_numbers[0:values],
                                                     ml_stats.normal_cdf)[1]
```

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

```

    # Plot the probabilities for only my own implementation
    plt.plot(plot_values, p_values_self, label = 'self')
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')
    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_kuiper_test_self.pdf')
    plt.figure()

    # Plot the probabiliteis with both the own implementation and astropy
    plt.plot(plot_values, p_values_astropy, label='astropy', linestyle=':',
    zorder=1.1)
    plt.plot(plot_values, p_values_self, label='self',zorder=1.0)
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')

    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_kuiper_test_self_astropy.pdf')
    plt.figure()
```

./Code/assigment_1.py

Code - Output plot(s)

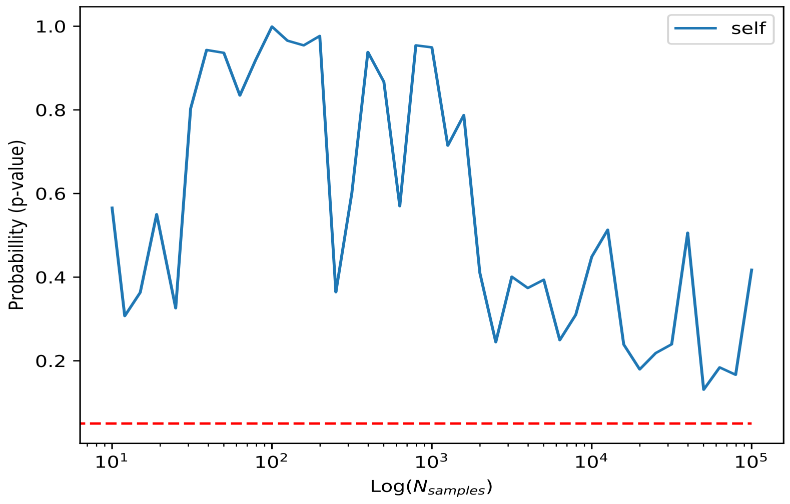

Figure 7: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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.

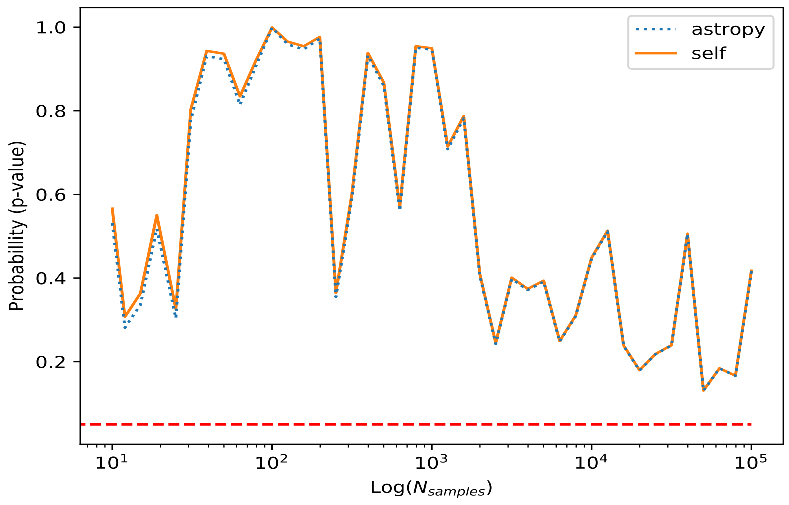

Figure 8: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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 be[2] 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 $\mu = 0$.

[2]A 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.

```python

def assigment_1e(random):
    """
        Execute assigment 1.e
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:
    # (1) matplotlib.pyplot as plt
    # (2) numpy as np
    # (3) scipy.stats as sp_stats
    # (4) mathlib.statistics as ml_stats

    # Load the data.
    data = np.loadtxt('randomnumbers.txt')

    # Generate the maximum amount of needed random numbers.
    random_numbers = random.gen_normals(0, 1, int(1e5))

    # The values to plot point for.
    plot_values = np.array(10**np.arange(1, 5.1, 0.1), dtype=int)

    # Pre-sort the random numbers
    random_nums_sorted = list()

    for idx, values in enumerate(plot_values):
        random_nums_sorted.append(sorting.merge_sort(random_numbers[0:values]))


    # Go over the columns and perform the KS-test2
    for i in range(data.shape[1]):

        # An array in which the p-values are stored for the self created
        # ks-test2 and the scipy version.
        p_values_self = np.zeros(len(plot_values))

        # Calculate the p-values with the ks-test2
        for idx, values in enumerate(plot_values):

            # Perform the ks-test2 with the own implementation.
            p_values_self[idx] = ml_stats.kstest2(data[:,i][0:values],
                                                  random_numbers[0:values],
                                                  random_nums_sorted[idx])

        # Plot the p-values.
        plt.plot(plot_values, p_values_self, label = 'self',color = 'orange')
        plt.hlines(0.05,0,10**5,colors='red',linestyles='--')

        plt.xlabel(r'Log($N_{samples}$)')
        plt.ylabel('Probabillity (p-value)')
        plt.xscale('log')
        plt.legend()
        plt.savefig("./Plots/1e_plot_column_{0}.pdf".format(i))
```

./Code/assigment_1.py

Code - Output plot(s)

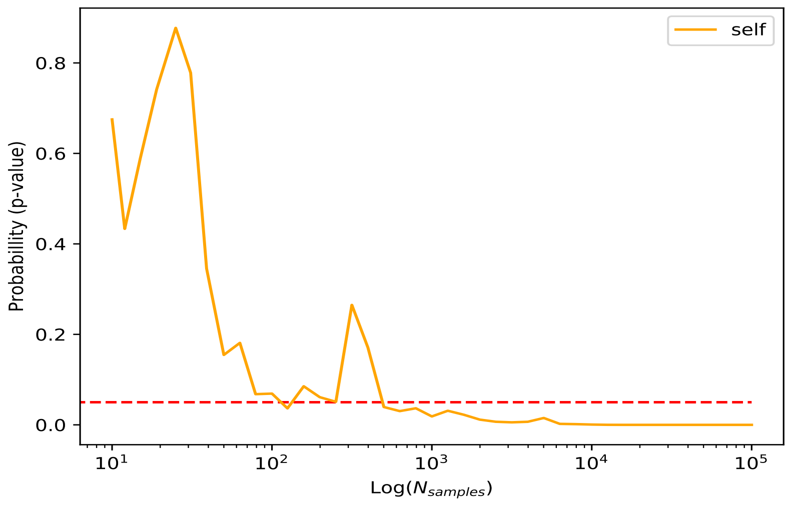

Figure 9: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **first** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

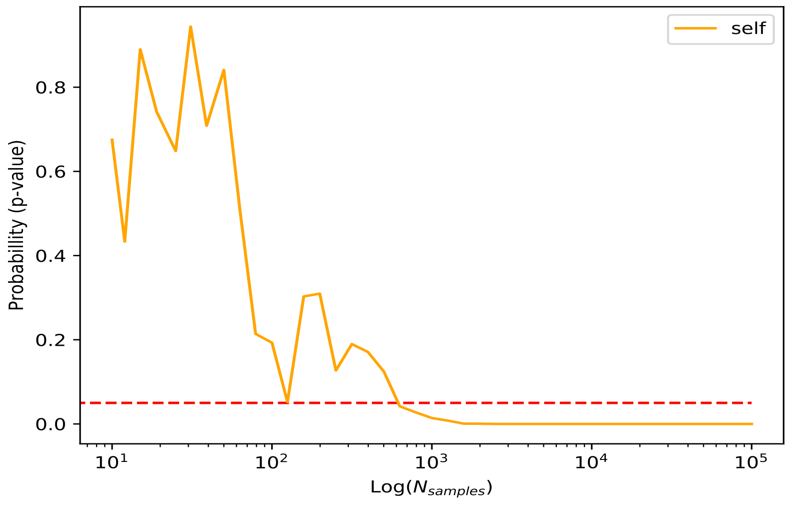

Figure 10: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **second** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

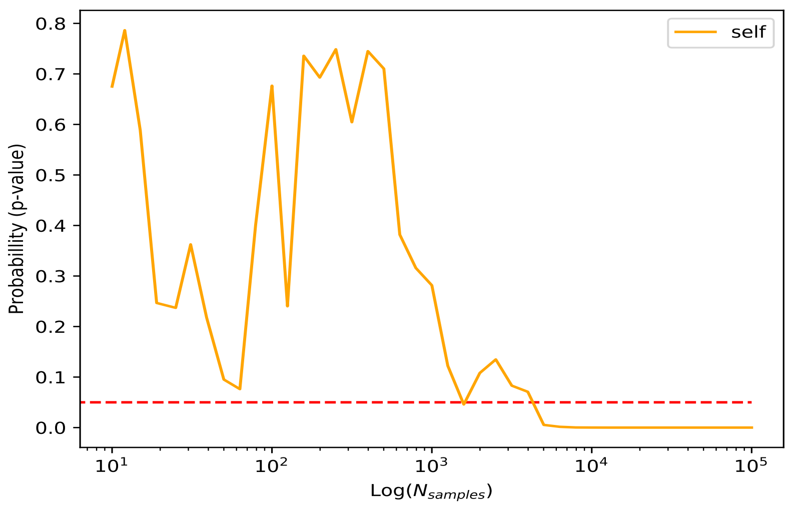

Figure 11: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **third** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

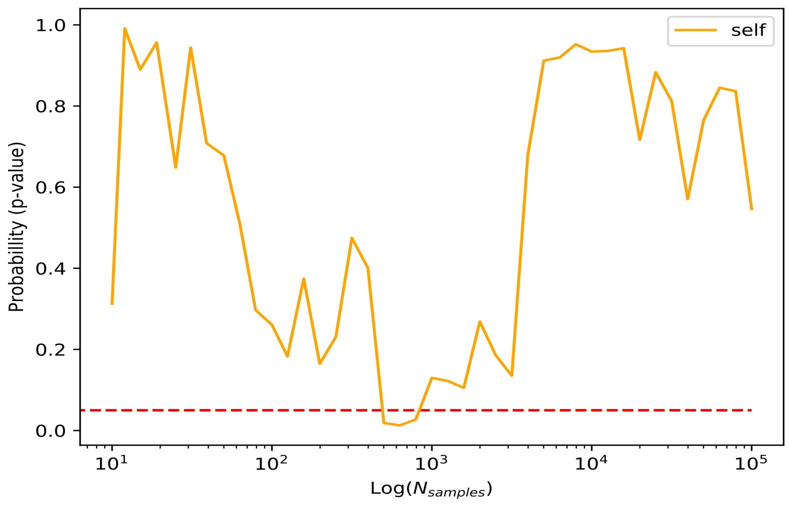

Figure 12: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **forth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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.

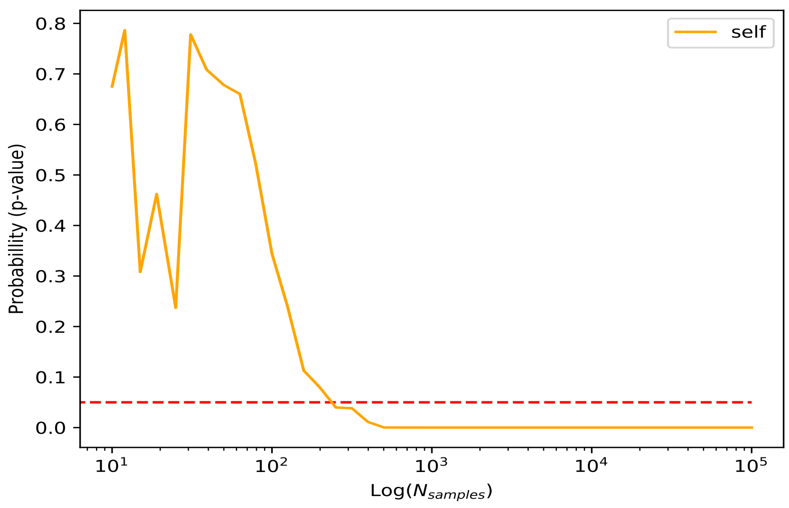

Figure 13: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **fifth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

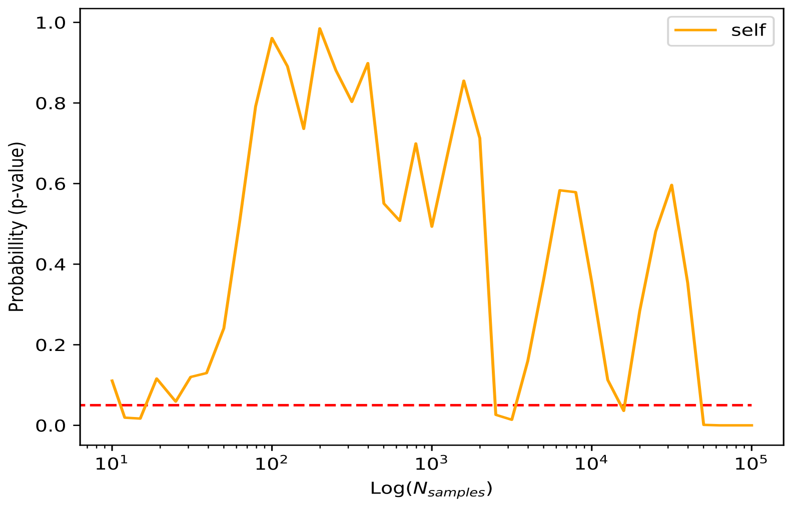

Figure 14: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **sixth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

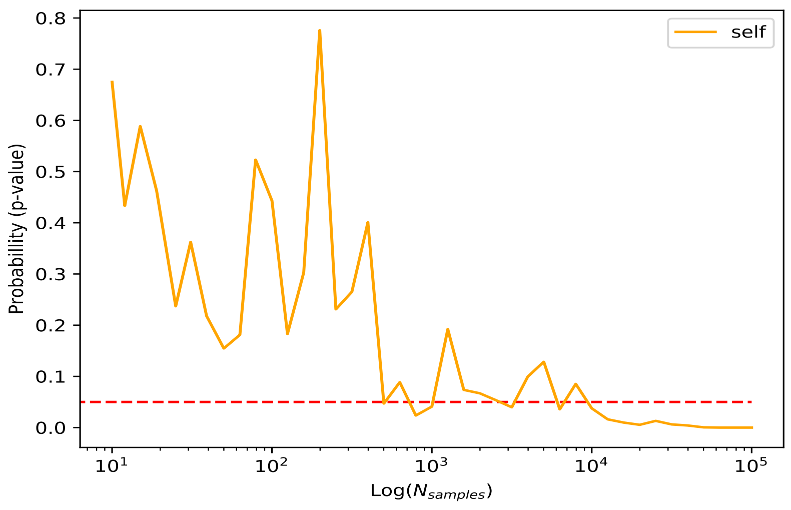

Figure 15: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **seventh** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

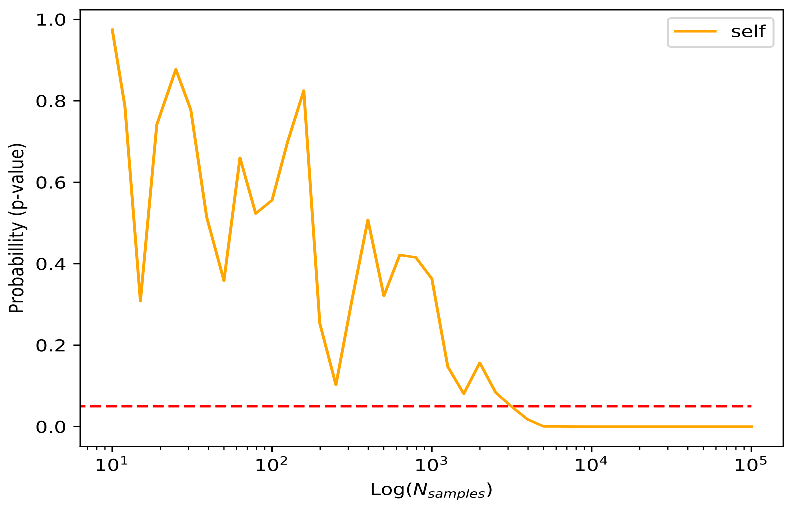

Figure 16: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **eight** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

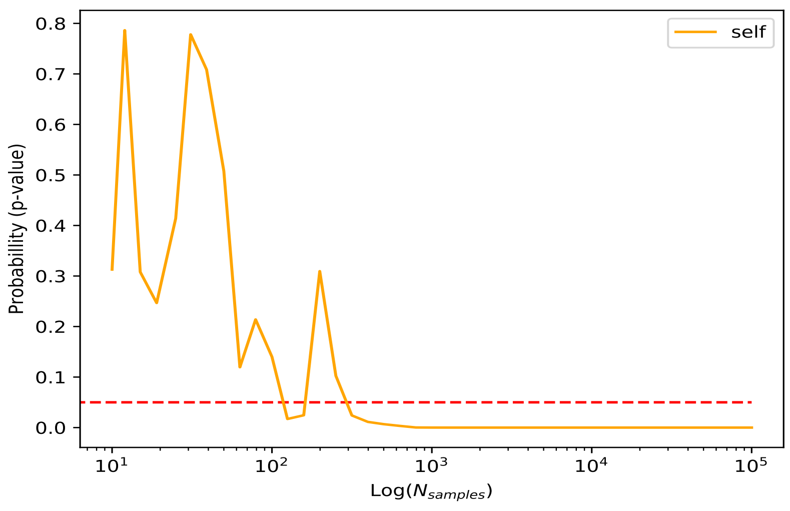

Figure 17: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **ninth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

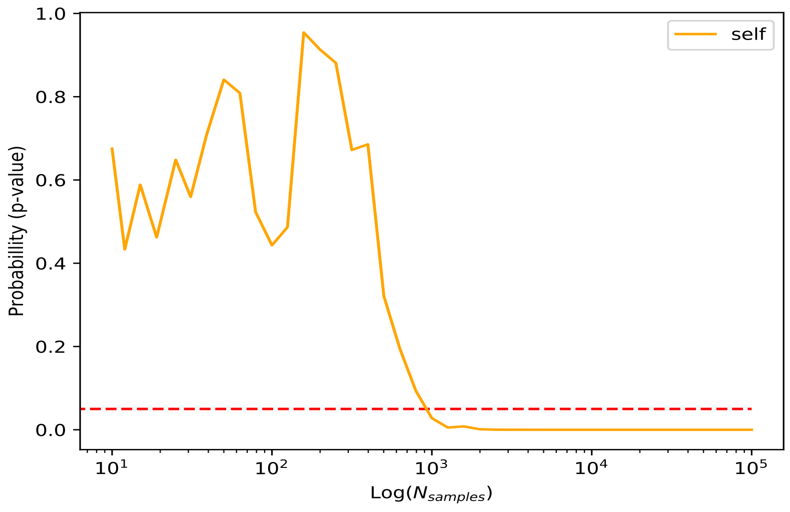

Figure 18: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **tenth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

```
import astropy.stats
import matplotlib.pyplot as plt
import mathlib.random as random
import mathlib.sorting as sorting
import mathlib.statistics as ml_stats
import numpy as np
import scipy.stats as sp_stats

def main():

    # Initialize the random number generator.
    rng = random.Random(78379522)

    # Run assigments
    assigment_1a(rng)
    assigment_1b(rng)
    assigment_1c(rng)
    assigment_1d(rng)
    assigment_1e(rng)


def assigment_1a(random):
    """
        Execute assigment 1.a
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:

    # (1) matplotlib.pyplot as plt
    # (2) mathlib.random as random
    # (3) mathlib.stats as ml_stats
    # (3) numpy as np

    # Print the seed.
    print('[1.a] Initial seed: ', random.get_seed())

    # Generate 1000 numbers.
    numbers_1000 = random.gen_uniforms(1000)

    # Plot them agianst each other.
    plt.scatter(numbers_1000[0:999], numbers_1000[1:], s=2)
    plt.ylabel(r'Probability $x_{i+1}$')
    plt.xlabel(r'Probability $x_{i}$')
    plt.savefig('./Plots/1_plot_against.pdf')
    plt.figure()

    # Plot them against the index.
    plt.plot(range(0, 1000), numbers_1000)
    plt.ylabel('Probability p')
    plt.xlabel('Index')
    plt.savefig('./Plots/1_plot_index.pdf')
    plt.figure()

    # Create a histogram for 1e6 points with 20 bins of 0.05 wide.
    numbers_mil = random.gen_uniforms(int(1e6))
    plt.hist(numbers_mil, bins=20, range=(0,1), color='orange',edgecolor='black')
    plt.ylabel('Counts')
    plt.xlabel('Generate values')
    plt.savefig('./Plots/1_hist_uniformnes.pdf')
    plt.figure()

    # Extra, to print the smallest and lagest bin value.
    counts, _ = np.histogram(numbers_mil, bins=20)
    print('[1.a] Max counts: ', max(counts))
    print('[1.a] Min counts: ', min(counts))

def assigment_1b(random):
    """
        Execute assigment 1.b
    Int:
        param: random -- An instance of the random number generator.
    """

    # The relevant imports for this piece of code are:

    # (1) matplotlib.pyplot as plt
    # (2) mathlib.random as random
    # (3) mathlib.stats as ml_stats
    # (4) numpy as np

    # Sigma and mean for the distribution.
    mean = 3.0
    sigma = 2.4

    # Generate 1000 random normal variables for the given mean and sigma.
    samples = random.gen_normals(mean, sigma, 1000)

    # The true normal distribution for the given mean and sigma.
    gaussian_x = np.linspace(-sigma*4 +mean, sigma*4 +mean, 1000)
    gaussian_y = ml_stats.normal(gaussian_x, mean, sigma)

    # Create a histogram.
    plt.hist(samples, bins=20, density=True, edgecolor='black',
                      facecolor='orange', zorder=0.1, label='Sampled')
    plt.plot(gaussian_x, gaussian_y, c='red', label='Normal')
    plt.xlim(-sigma*6.5 + mean,sigma*6.5 + mean)
    plt.ylim(0, max(gaussian_y)*1.2)

    # Add the sigma lines.

    # The hight of the sigma lines that need to be added.
    lines_height = max(gaussian_y)*1.2

    for i in range(1, 6):
        # Absolute shift from the mean for the given sigma
        shift = i*sigma

        # Sigma right of the mean.
        plt.vlines(mean + shift, 0, lines_height,
                     linestyles='-', color='black', zorder=0.0)
        plt.text(mean + shift -0.4, lines_height/1.3, str(i) + r'$\sigma$',
                color='black', backgroundcolor='white', fontsize=9)

        # Sigma line left of the mean.
        plt.vlines(mean - shift, 0, lines_height, linestyles='-', zorder=0.0)
        plt.text(mean - shift -0.4, lines_height/1.3, str(i) + r'$\sigma$',
                            color='black', backgroundcolor='white', fontsize=9)

    plt.legend(framealpha=1.0)
    plt.savefig('./Plots/1_hist_gaussian.pdf')
    plt.figure()

def assigment_1c(random):
    """
        Execute assigment 1.c
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:
    # (1) matplotlib.pyplot as plt
    # (2) numpy as np
    # (3) astropy.stats
    # (4) mathlib.statistics as ml_stats

    # The values to plot point for.
    plot_values = np.array(10**np.arange(1, 5.1, 0.1),dtype=int)

    # An array in which the p-values are stored for the self created.
    # ks-test and the scipy version.
    p_values_self = np.zeros(len(plot_values))
    p_values_scipy = np.zeros(len(plot_values))

    # Generate the maximum amount of needed random numbers.
    random_numbers = random.gen_normals(0, 1, int(1e5))

    # Calculate the p-values with the ks-test.
    for idx, values in enumerate(plot_values):

        # Calculate the value with scipy.
        p_values_scipy[idx] = sp_stats.kstest(random_numbers[0:values],
                                              'norm')[1]
        # Calculate the p-values with the own implementation.
        p_values_self[idx] = ml_stats.kstest(random_numbers[0:values],
                                             ml_stats.normal_cdf)


    # Plot the probabilities for only my own implementation.
    plt.plot(plot_values, p_values_self, label = 'self', color='orange')
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')
    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_ks_test_self.pdf')
    plt.figure()

    # Plot the probabilities for both the scipy and my own implemntation.
    plt.plot(plot_values, p_values_scipy, label='scipy', linestyle=':',
                                                    zorder=1.1)
    plt.plot(plot_values, p_values_self, label='self', zorder=1.0,
                                                    color='orange')
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')
    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_ks_test_self_scipy.pdf')
    plt.figure()

def assigment_1d(random):
    """
        Execute assigment 1.d
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:
    # (1) matplotlib.pyplot as plt
    # (2) numpy as np
    # (3) scipy.stats as sp_stats
    # (4) mathlib.statistics as ml_stats

    # The values to plot point for.
    plot_values = np.array(10**np.arange(1, 5.1, 0.1), dtype=int)

    # Generate the maximum amount of needed random numbers.
    random_numbers = random.gen_normals(0, 1, int(1e5))

    # An array in which the p-values are stored for the self created
    # kuiper-test and the astropy version.
    p_values_self = np.zeros(len(plot_values))
    p_values_astropy = np.zeros(len(plot_values))

    # Calculate the p-values with the ks-test
    for idx, values in enumerate(plot_values):

        # Calculate the value with the own implemnetation
        p_values_self[idx] = ml_stats.kuiper_test(random_numbers[0:values],
    ml_stats.normal_cdf)
        # Calculare the value with astropy.
        p_values_astropy[idx] = astropy.stats.kuiper(random_numbers[0:values],
                                                      ml_stats.normal_cdf)[1]

    # Plot the probabilities for only my own implementation
    plt.plot(plot_values, p_values_self, label = 'self')
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')
    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_kuiper_test_self.pdf')
    plt.figure()

    # Plot the probabiliteis with both the own implementation and astropy
    plt.plot(plot_values, p_values_astropy, label='astropy', linestyle=':',
    zorder=1.1)
    plt.plot(plot_values, p_values_self, label='self',zorder=1.0)
    plt.hlines(0.05,0,10**5,colors='red',linestyles='--')
    plt.xscale('log')

    plt.xlabel(r'Log($N_{samples}$)')
    plt.ylabel('Probabillity (p-value)')
    plt.legend()
    plt.savefig('./Plots/1_plot_kuiper_test_self_astropy.pdf')
    plt.figure()

def assigment_1e(random):
    """
        Execute assigment 1.e
    Int:
        param: random -- An initialization of the random number generator.
    """

    # The relevant imports for this piece of code are:
    # (1) matplotlib.pyplot as plt
    # (2) numpy as np
    # (3) scipy.stats as sp_stats
    # (4) mathlib.statistics as ml_stats

    # Load the data.
    data = np.loadtxt('randomnumbers.txt')

    # Generate the maximum amount of needed random numbers.
    random_numbers = random.gen_normals(0, 1, int(1e5))

    # The values to plot point for.
    plot_values = np.array(10**np.arange(1, 5.1, 0.1), dtype=int)

    # Pre-sort the random numbers
    random_nums_sorted = list()

    for idx, values in enumerate(plot_values):
        random_nums_sorted.append(sorting.merge_sort(random_numbers[0:values]))


    # Go over the columns and perform the KS-test2
    for i in range(data.shape[1]):

        # An array in which the p-values are stored for the self created
        # ks-test2 and the scipy version.
```

```python
        p_values_self = np.zeros(len(plot_values))

        # Calculate the p-values with the ks-test2
        for idx, values in enumerate(plot_values):

            # Perform the ks-test2 with the own implementation.
            p_values_self[idx] = ml_stats.kstest2(data[:,i][0:values],
                                                  random_numbers[0:values],
                                                  random_nums_sorted[idx])

        # Plot the p-values.
        plt.plot(plot_values, p_values_self, label = 'self',color = 'orange')
        plt.hlines(0.05,0,10**5,colors='red',linestyles='--')

        plt.xlabel(r'Log($N_{samples}$)')
        plt.ylabel('Probabillity (p-value)')
        plt.xscale('log')
        plt.legend()
        plt.savefig("./Plots/1e_plot_column_{0}.pdf".format(i))
        plt.figure()

if __name__ == '__main__':
    main()
```

./Code/assigment_1.py

Code - Shared-modules

Random Number Generator

```python
import numpy as np

class Random(object):
    """
        A class representing a random number generator (RNG)
    """

    def __init__(self, seed):
        """
            Create a new instance of the random number generator.

        In:
            param: seed -- The seed of the random number generator.
                           This must be a positive integer.

        """

        # The seed and state of the generator
        self._seed = np.uint64(seed)
        self._state = self._seed

        # maximum uint32 value
        self._uint32_max = np.uint64(0xFFFFFFFF)

        # The values for the Xor shift.
        self._xor_a1 = np.uint64(20)
        self._xor_a2 = np.uint64(41)
        self._xor_a3 = np.uint64(5)

        # The values for the multiply with carry.
        self._mwc_a = np.uint64(4294957665)
        self._mwc_base = np.uint64(2**32)

    def get_seed(self):
        """
            Get the seed that is used to initialize this generator.

        Out:
            return: The seed used to initalize the generatorr.
        """
        return self._seed

    def get_state(self):
        """
            Get the state of the generator.

        Out:
            return: The state of the generator.
        """
        return self._state


    def gen_next_int(self):
        """
            Generate a new random 32-bit unsigned integer.
        Out:
            return: A random 32-bit unsigned integer.
        """

        # The state is at the end updated with mwc.
        # We therefore shouldn't use more than 32 bits to generate
        # the number.

        return self._update_state() & self._uint32_max

    def gen_uniform(self):
        """
```

```python
            Generate a random float between 0 and 1.
        Out:
            return: A random float between 0 and 1.
        """

        return self.gen_next_int()*1.0 / self._uint32_max

    def gen_uniforms(self, amount):
        """
            Generate multiple random floats
            between 0 and 1.
        In:
            param: amount -- The amount of floats to generate.
        Out:
            return: An array with 'amount' random floats
                    between 0 and 1.
        """

        samples = np.zeros(amount)

        for i in range(amount):
            samples[i] = self.gen_uniform()

        return samples

    def gen_normal(self, mean, sigma):
        """
            Generate a random normal distributed float.

        In:
            param: mean -- The mean of the gaussian distribution.
            param: sigma -- The squareroot of the variance of the distribution.
        Out:
            return: A random float that is drawn from the parameterized normal
                    distribution.
        """

        # Generate two uniform variables.
        u1 = self.gen_uniform()
        u2 = self.gen_uniform()

        # Use the box muller transformation.
        return sigma*np.sqrt(-2* np.log(1-u1))*np.cos(2*np.pi*u2) + mean

    def gen_normal_uniform(self, mean, sigma, u1, u2):
        """
            Generate a random normal distributed float from two provided
            uniform variables.


        """
        pre_factor= sigma*np.sqrt(-2* np.log(1-u1))

        return pre_factor*np.cos(2*np.pi*u2) + mean, pre_factor*np.sin(2*np.pi*u2) + mean

    def gen_normals(self, mean, sigma, amount):
        """
            Generate multible random normal distributed float.

        In:
            param: mean -- The mean of the gaussian distribution.
            param: sigma -- The squareroot of the variance of the distribution.
            param: amount -- The amount of floats to generate.
        Out:
            return: An array with random floats drawn from the parameterized normal
                    distribution.
        """

        # Pre-factors in the box muller transformation.
        square_pre_factor = -2*sigma**2
        angle_pre_factor = 2*np.pi

        # With the Box-muller two random normals can be generated for two
        # uniforms. If the amount of requested variables is odd then add
        # one to it and later remove it when returning the result.
        elements = amount if amount % 2 == 0 else amount + 1

        # Array in which the drawn normal distributed variables are stored.
        normal_dist = np.zeros(elements)

        # Apply the box muller transformation to generate the samples.
        for i in range(0, elements, 2):

            # Generate the uniforms.
            u1 = self.gen_uniform()
            u2 = self.gen_uniform()

            # Calculate common terms.
            pre_fact = np.sqrt(square_pre_factor*np.log(1-u1))

            # Calculate the samples.
            normal_dist[i] = pre_fact*np.cos(angle_pre_factor*u2) + mean
            normal_dist[i+1] = pre_fact*np.sin(angle_pre_factor*u2) + mean

        # If amount is odd, don't return the last element.
        return normal_dist[0:amount]

    def _update_state(self):
        """
            Update the state of the random number generator.

        Out:
            return: The new state of the random number generator.
        """

        self._state = self._xor_shift(self._state)
        self._state = self._mwc(self._state & self._uint32_max) ^ self._state

        return self._state


    def _xor_shift(self, number):
        """
            Execute the XOR-shift algorithm on the
            input number.
        In:
            param: number -- The number to XOR-shift.
        Out:
            return: The number produced by XOR-shift.
         """

        # Shift to the right and then bitwise xor.
        number ^= (number >> self._xor_a1)
        # Shift to the left and then bitwise xor.
        number ^= (number << self._xor_a2)
        # Shift to the right and then bitwise xor.
        number ^= (number >> self._xor_a3)

        return number

    def _mwc(self, number):
        """
           Perform multiply with carry (MWC) on
           the given input.
        In:
            param: number -- The number to perform MWC on, must be an uint64.
        Out:
            return: The new number.
        """
        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 functioned that where needed for the sub-questions.

```
import numpy as np
import mathlib.sorting as sorting


def kstest(x, cdf):
    """
        Perform the Kolmogorov-Smirnov test for goodness of fit
        and return the p-value.
    In:
        param: x   -- An array with value's who's CDF is expected to be
                      the same as the provided CDF. Must be atleast size 4

        param: cdf -- A function that is the expected cdf under the null
    hypothesis.
    Out:
        return: The p-value obtained by performing the KS-test.
    """

    # Amount of values in the input array.
    x_size = len(x)

    # Sort the values and evaluate the cdf.
    x_sorted = sorting.merge_sort(x)
    x_sorted_cdf = cdf(x_sorted)

    # Maximum distance.
    max_dist = 0

    # Value of the emperical cdf at step i-1.
    x_cdf_emperical_previous = 0

    # Find the maximum distance.
    for idx in range(0,x_size):

        # Calculate the emperical cdf.
        x_cdf_emperical = (idx+1)/x_size
        # The true cdf evaluation at the given point.
        x_cdf_true = x_sorted_cdf[idx]

        # Find the distance. The emperical
        # CDF is a step function so there are two distances
        # that need to be checked at each step.

        # Calculate the two distances
        distance_one = abs(x_cdf_emperical - x_cdf_true)
        distance_two = abs(x_cdf_emperical_previous - x_cdf_true)


        # Find the maximum of those two distances and
        # check if it is larger than the current know maximum distance.
        max_dist = max(max_dist, max(distance_one, distance_two))

        # Save the current value of the emperical cdf.
        x_cdf_emperical_previous = x_cdf_emperical

    # Calculate the p-value with the help of the CDF.
    sqrt_elemens = np.sqrt(x_size)
    cdf = _ks_statistic_cdf((sqrt_elemens + 0.12+0.11/sqrt_elemens)*max_dist)
    return 1 - cdf


def kstest2(x1, x2, x2_sorted = None):
    """
        Perform the Kolmogorov-Smirnov test for goodness of fit
        and return the p-value.
    In:
        param: x1   -- An array with value's who's CDF is expected to be
                       the same as the CDF of the proviced values.
                       Must be atleast size 4.

        param: x2 -- A discretized pdf of the expected distribution under the
    null hypothesis.
    Out:
        return: The p-value obtained by performing the KS-test
    """

    # Amount of values in the input distributions.
    x1_size = len(x1)
    x2_size = len(x2)

    # Sort both arrays.
    x1 = sorting.merge_sort(x1)
    x2 = sorting.merge_sort(x2) if type(x2_sorted) is not None else x2_sorted

    # The maximum distance
    max_dist = 0

    # The iteration values used to determine
    # the emperical pdf's and the max distance.
    x1_i, x2_j = 0,0

    # Find the maximum distance by updating the emperical CDFs.
    while x1_i < x1_size and x2_j < x2_size:

        # Update the indices used for the emperical CDF's.

        if x1[x1_i] < x2[x2_j]:
            x1_i += 1
        else:
            x2_j += 1

        # Find the max distance
        max_dist = max(abs(x1_i/x1_size-x2_j/x2_size), max_dist)

    sqrt_factor = np.sqrt((x1_size*x2_size)/(x1_size+x2_size))
    cdf = _ks_statistic_cdf((sqrt_factor + 0.12+0.11/sqrt_factor)*max_dist)

    return 1 - cdf

def kuiper_test(x, cdf):
    """
        Perform the Kuiper test for goodness of fit
        and return the p-value.
    In:
        param: x   -- An array with value's who's CDF is expected to be
                      the same as the provided CDF. Must be atleast size 4

        param: cdf -- A function that is the expected cdf under the null
    hypothesis.
    Out:
        return: The p-value obtained by performing the kuiper-test
    """

    # Sort the data in ascending order, calculate the
    # cdf and the emperical cdf for the sorted values and
    # save the total amount of  elements we have.
    x_sorted = sorting.merge_sort(x)
    x_sorted_cdf = cdf(x_sorted)
    x_elements = len(x)

    # Find the maximum distance above and below
    # the true cdf.
    max_dist_above = 0
    max_dist_below = 0

    # Value of the cdf at step i-1.
    x_cdf_emperical_previous = 0


    for idx, x in enumerate(x_sorted):

        # Calculate the emperical cdf.
        x_cdf_emperical = (idx+1)/x_elements
        # Calculate the true cdf.
        x_cdf_true = x_sorted_cdf[idx]

        # Find the maximum distance above and below
        max_dist_above = max(x_cdf_emperical - x_cdf_true, max_dist_above)
        max_dist_below = max(x_cdf_true - x_cdf_emperical_previous,
    max_dist_below)

        # Update previous cdf
        x_cdf_emperical_previous = x_cdf_emperical

    sqrt_elem = np.sqrt(x_elements)
    v = max_dist_above + max_dist_below
    cdf = _kuiper_statistic_cdf((sqrt_elem + 0.155+0.24/sqrt_elem)*v)

    return 1 - cdf


def _ks_statistic_cdf(z):
    """
        An approximation for the cdf of the
        Kolmogorov-Smirnov (KS) test staistic.
    In:
        param: z -- The value to calculate the cdf at.
    Out:
        return: An approximation of the cdf for the given value.    print(
    max_dist_above + max_dist_below)
    """

    # Numerical approximation taken from:
    # Numerical methods - The art of scientific computation.
    # Third edition.

    if z < 1.18:
        exponent = np.exp(-np.pi**2/(8*z**2))
        pre_factor = np.sqrt(2*np.pi)/z

        return pre_factor*exponent*(1+ exponent**8)*(1+exponent**16)
    else:
        exponent = np.exp(-2*z**2)
        return 1-2*exponent*(1-exponent**3)*(1-exponent**6)

def _kuiper_statistic_cdf(z):
    """
        An approximation for the cdf of the
        Kuiper test statistic
    In:
        param: z -- The value to calculate the cdf at.
    Out:
        return: An approximation of the cdf for the given value.
    """

    # Value of z is to small, sum will be 1 up to 7 digits
    if z < 0.4:
        return 1

    # Approximateed value of the sum by performing 100 iterations

    # The value to return
    ret = 0
    # A term often needed in the sum.
    z_squared = z**2

    # Evaluate the first 100 terms in the sum.
    for j in range(1, 100):
        power = j**2 * z_squared
        ret += (4 * power -1)*np.exp(-2*power)

    return 1- 2*ret



def normal_cdf(x, mean = 0, sigma = 1):
    """
        Evaluate the cummulative normal distribution for
        the given parameters
    In:
        param: x -- The point to evaluate the cdf at or an array of points to
    evaluate it for.
        param: mean -- The mean of the normal distribution.
        param: sigma -- The square root of the variance for the normal
    distribution.
    Out:
        return: The cummulative normal distribution evaluated at.
    """

    # Calculate the CDF using the erf function (defined below).
    return 0.5 + 0.5*erf((x-mean)/(np.sqrt(2)*sigma))


def normal(x, mean = 0, sigma = 1):
    """
        Evaluate the normal distrbution for the given
        parameters.
    In:
        param: x -- The point to evaulte the distribution at.
        param: mean -- The mean of the distribution.
        param: sigma -- The square root of the variance for the distribution.
    Out:
        return: The value of the parameterized distribution evaluated at the
    given point.
    """

    return 1/((np.sqrt(2*np.pi)*sigma))*np.exp(-0.5*((x - mean)/sigma)**2)

def erf(x):
    """
        Evaluate the erf function for a value of x.
    In:
        param: x -- The value to evaluate the erf function for.
    Out:
        return: The erf function evaluated for the given value of x.
    """

    # Numerical approximation taken from Abramowits and Stegun.

    # Constants for the numerical approximation
    p = 0.3275911
    a1 = 0.254829592
    a2 = -0.284496736
    a3 = 1.421413741
    a4 = -1.453152027
    a5 = 1.061405429

    # Array in which the result is stored
    ret = np.zeros(len(x))

    # The approximation functions
    erf_func_t_val = lambda x: 1/(1+ p*x)
    erf_func_approx = lambda t, x : 1 -  t*(a1 + t*(a2 + t*(a3 + t*(a4 + t*a5))))
    *np.exp(-x**2)

    # Evaluate for both positive and negative
    neg_mask = x < 0
    neg_x = x[neg_mask]
    pos_mask = x >= 0
    pos_x = x[pos_mask]

    ret[neg_mask] = -erf_func_approx(erf_func_t_val(-neg_x), -neg_x)
    ret[pos_mask] = erf_func_approx(erf_func_t_val(pos_x), pos_x)

    return ret
```

./Code/mathlib/statistics.py

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

```
import numpy as np

def merge_sort(array):
    """
        Sort an array using merge sort

    In:
        param: array -- The array to sort.
    Out:
        return: The sorted array.
    """

    # Get the length of the array.
    size = len(array)

    # If the length is 1 then  return the input array.
    # This is an important check as this function is called
    # recursively.
    if size == 1:
        return array

    # Split the array in an sorted left and right segment.
    left_sorted = merge_sort(array[:size >> 1])
    right_sorted = merge_sort(array[size >> 1:])
    left_sorted_len = len(left_sorted)
    right_sorted_len = len(right_sorted)

    #
    # Merge the left and right array.
    #

    # The final sorted array.
    result = np.zeros(size)

    # Current index in the left sorted array.
    left_idx = 0
    # Current index in the right sorted array.
    right_idx = 0
    # Current index in the final sorted array.
    result_idx = 0

    # While we didn't fill the result array.
    while result_idx < size:

        # Element from left array is smaller, insert it and increase position.
        if left_sorted[left_idx] < right_sorted[right_idx]:
            result[result_idx] = left_sorted[left_idx]
            left_idx += 1
            result_idx += 1
        # Element from right array is smaller, insert it and increase position.
        else:
            result[result_idx] = right_sorted[right_idx]
            right_idx += 1
            result_idx += 1

        # Only right arrat has elements left, insert the remaining elements.
        if left_idx == left_sorted_len:
```

```python
            result[result_idx:] = right_sorted[right_idx:]
            break

        # Only left has elements left, insert the remaining elements
        if right_idx == right_sorted_len:
            result[result_idx:] = left_sorted[left_idx:]
            break

    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