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 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 int 32. 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 new state.
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 int 32. The resulting value is then divided by the maximum value of an uint32 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. 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.

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 .. where the full code is shown that contains all sub-questions together. The code for the random number generator can be found on page,... as it is treated as a shared module.

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

    # 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')
    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()
```

./Code/assigment_1.py

Code - Output text

The text output produced by the code:

```
Initial seed:  1234567
```

./Output/assigment1_out.txt

Code - Output plots

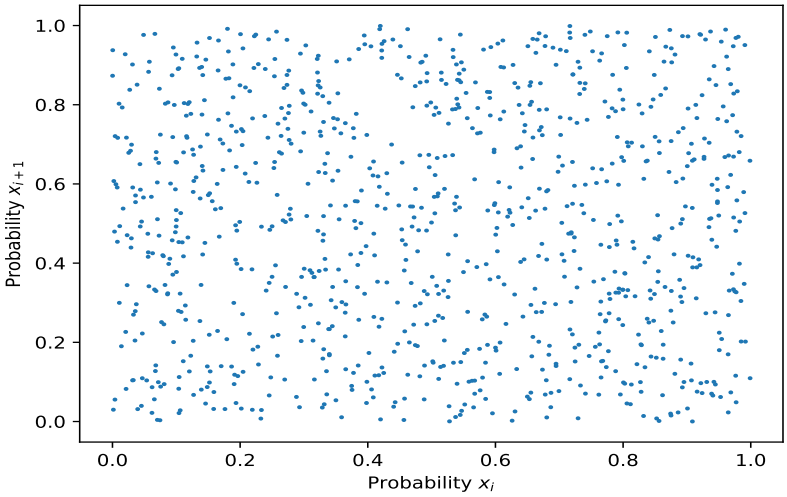

Figure 1: A plot of random number x_{i+1} against random number x_i for the first 1000 random uniforms produced by the random number generator. A good random number generator should produce a homogeneous plot without many (large) empty spots. In the above plot large empty spots appear to be absent, which suggests that the random number generator passes this test.

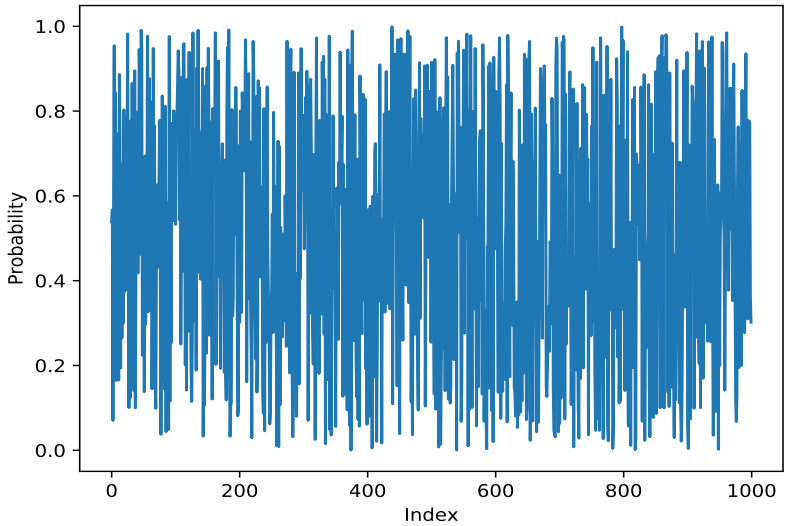

Figure 2: The first 1000 random uniform numbers produced by the random number generator (RNG) against their index. A good random number generator should not have large wide gaps (e.g when moving from index 400 to 450 it should not only produce values larger than 0.8, which would leave a wide gap). In the plot these gaps appear to be absent. The average value produced by the RNG should furthermore be 0.5. This corresponds to rapidly moving up and down the line $y = 0.5$. In the plot this should, result in a 'dense' region (less white) around the line $y = 0.5$. It can indeed be seen that the plot is denser close to $y = 0.5$ than at $y = 0.8$ or $y = 0.2$.

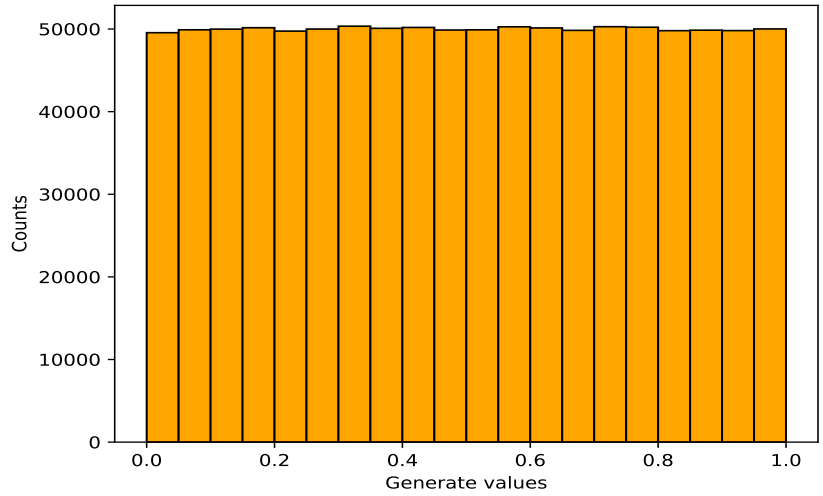

Figure 3: The uniforms of the random number generator for 1 million random values. The values are binned in 20 bins. A good random number generator should fluctuate around 50000 $\pm 2\sqrt{50000} = 50000 \pm 447$ counts per bin (2 sigma). The maximum and minimum amount of counts corresponds to 50444 and 49642 counts. These value's just lay withing the 2 sigma uncertainty. The uniformness of the random number generator therefore appears to be quite 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. The derivation can be found below and the derived transformation is added to the random number generator class to produce normal distributed values.

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

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

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

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

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

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

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

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;

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

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

These transformation are implemented in the random number generator, which is treated as a shared module and can be found on page .... 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 ...

Code - Plots

The code for generating the plots. The imports are not explicit shown, but can be found on page ... where the code for the full assignment is shown. The shared modules include ... and .... .These can be found on pages ....

```
    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.
    """

    # 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):
```

./Code/assigment_1.py

Code - Output plot(s)

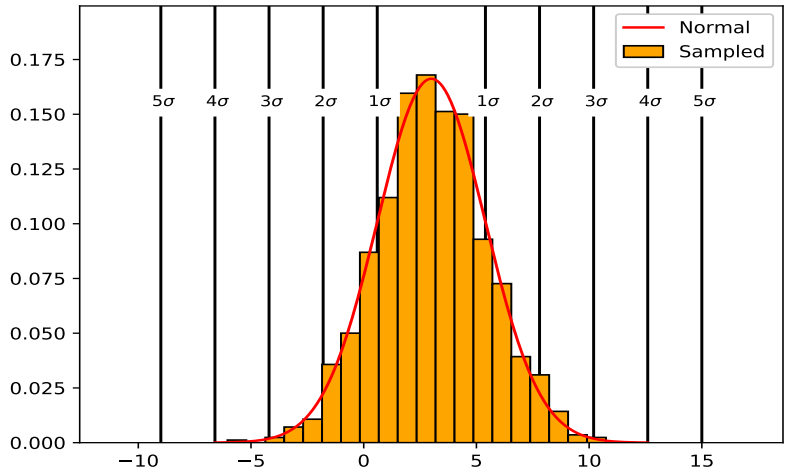

Figure 4: A histogram of the 1000 random nomal distributed variables generated with the box muller method for $\mu = 3$ and $\sigma = 2.4$ (orange) and the true normal distribution (red). The histogram approximates the normal distribution by eye, but displays some deviations. The peak is higher than it should be and slightly left of the peak is a bin with a low amount of counts. The left part of the histogram furthermore appears to be below the true normal distribution. 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,

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

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 ... who states that the sum can be approximated by,

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

(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,

$$\Phi\left(\frac{x-\mu}{\sigma}\right) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right] \tag{10}$$

where the erf is given by,

$$\text{erf}(x)\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{11}$$

The integral of the erf function lacks a closed form and therefore also needs to be numerically approximated. The chosen approximation is taken from ..., who states that the function can be approximated by,

$$\text{erf}(x) \approx 1 - (a_1 t + a_2 t^2 + ... + a_5 t^5) e^{-x^2}\text{x} \quad t = \frac{1}{1+px} \tag{12}$$

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 .... at page, as this file is threaded as a shared module. The KS-test does require an sorting alogirhtm, this algorithm is implemented in the file ... and can be found on page ..., as it is simillar to the previous file treated as a shared module. The code for generating 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 giving in the comments to make the code easier to understand. The code of assigment 1, inclusive imports, can as mentioned before be found on page ....

```
        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) scipy.stats as sp_stats
    # (4) mathlib.sorting as sorting
    # (5) 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]
        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')
    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 beeing consistent under the null hypothesies.
    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)
    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):
```

./Code/assigment_1.py

Code - Output plot(s)

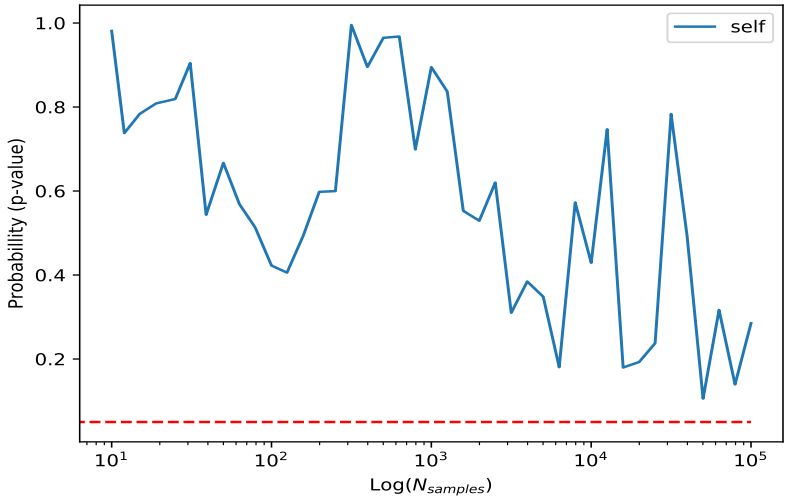

Figure 5: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** thered line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the RNG always passes ks-test up to atleast 10^5 samples.

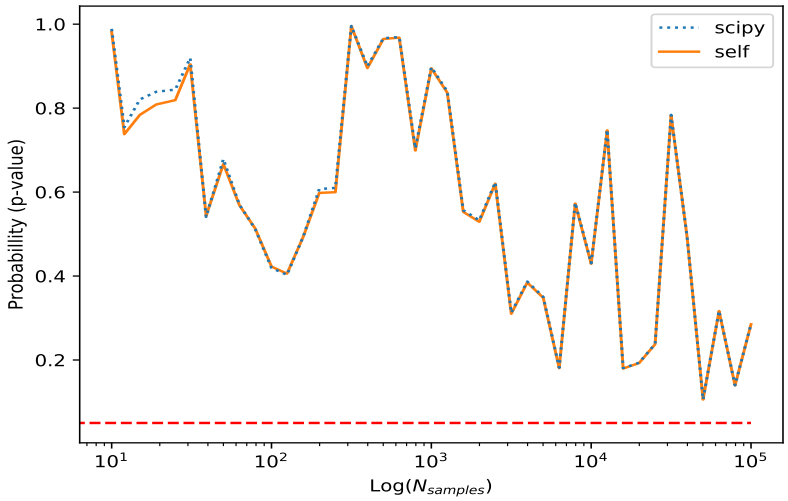

Figure 6: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. The orange line is the self written implementation of the KS-test and the blue line is the scipy version. A point **below** 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 always close to the scipy version. It shows (small) deviations for small sample sizes. The exact cause of this is unknown, but is 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 simmilar to the KS-test require a numerical approximation of the CDF for the kuiper statistics. The CDF of the kuiper staistic is given by,

$$P_{kuiper}(\lambda) = 1 - 2\sum_{j=1}^{\infty}(4j^2\lambda^2 - 1)e^{-2j^2\lambda^2} \tag{13}$$

here the sum is negligible compared to the machine error if $\lambda < 0.4$. In this case the numerically approximation thus consist of simply returning 1. If $\lambda > 0.4$ then the sum is approximated by calculating the first 100 terms of the sum, as 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`, which is located on page ... .The code that creates the plots and the plots self can be found below. This code does make use of **astropy** to compare self written implementation with the implementation of astropy.

Code - Plots

The code for generating the two plots in which the kuiper test is performed. The imports are not explicit shown but can be shown on page ..., where the code of the full assignment is displayed. The comments give an overview of the imports used.

```
    plt.figure()

def assigment_1d(random):

    # 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
    # ks-test and the scipy 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 both astropy and the own implemnetation
        p_values_self[idx] = ml_stats.kuiper_test(random_numbers[0:values],
    ml_stats.normal_cdf) # discard distance
        p_values_astropy[idx] = astropy.stats.kuiper(random_numbers[0:values],
    ml_stats.normal_cdf)[1] # stats.kuper_test(random_numbers,  stats.normal_cdf)
    #scipy_stats.norm.cdf)

    # 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()
```

[1] In theory less terms should be more than enough. The evaluation of the sum could therefore stop early by checking for a required precision. This is however not implemented as summing 100 terms is cheap.

./Code/assigment_1.py

Code - Output plot(s)

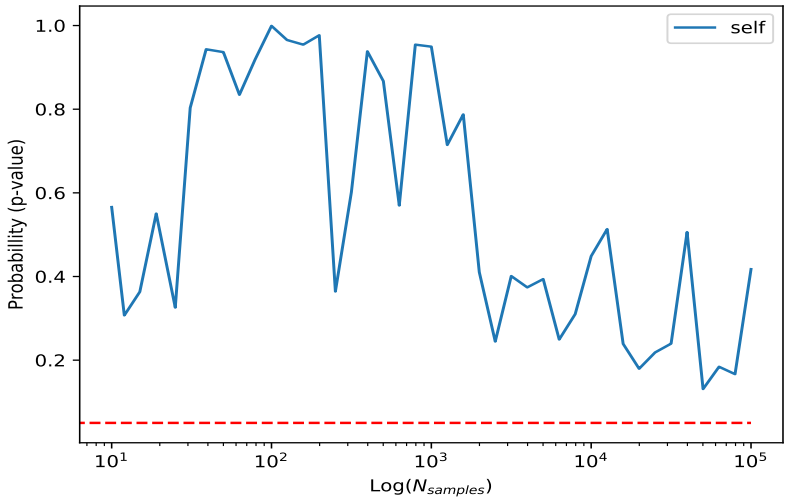

Figure 7: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the RNG always passes kuiper test. IT can however be seen that the value of the statistic stays lower for a significant amount of samples ($N_{samples} > 10^4$). This might indicate that there is flaw in the random number generator.

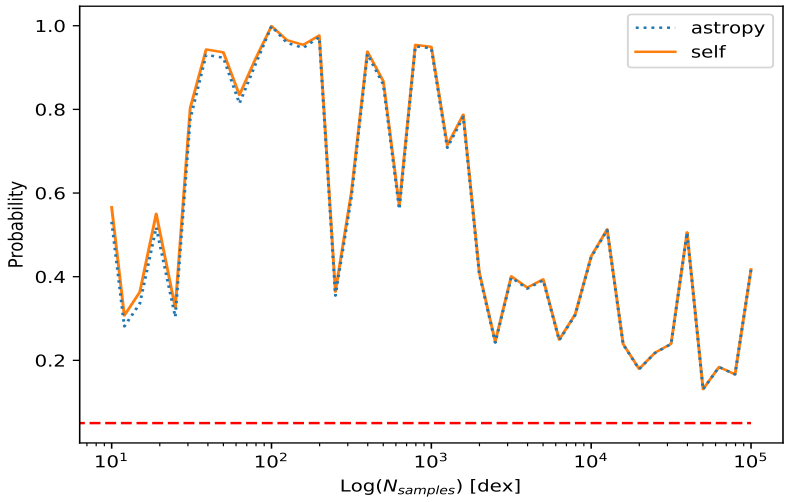

Figure 8: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the self written implementation deviates from the astropy implementation at the start, similar to what happend in the KS-test.

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 distirbutions are compared with the ks-test2. The

Code - Plots

The code for generating the two plots in which the kuiper test is performed. The imports are not explicit shown but can be shown on page ..., where the code of the full assignment is displayed. The comments give an overview of the imports used.

```
    plt.figure()

def assigment_1d(random):

    # 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
    # ks-test and the scipy 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 both astropy and the own implemnetation
        p_values_self[idx] = ml_stats.kuiper_test(random_numbers[0:values],
    ml_stats.normal_cdf) # discard distance
        p_values_astropy[idx] = astropy.stats.kuiper(random_numbers[0:values],
    ml_stats.normal_cdf)[1] # stats.kuper_test(random_numbers,  stats.normal_cdf)
    #scipy_stats.norm.cdf)

    # 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()
```

./Code/assigment_1.py

Code - Output plot(s)

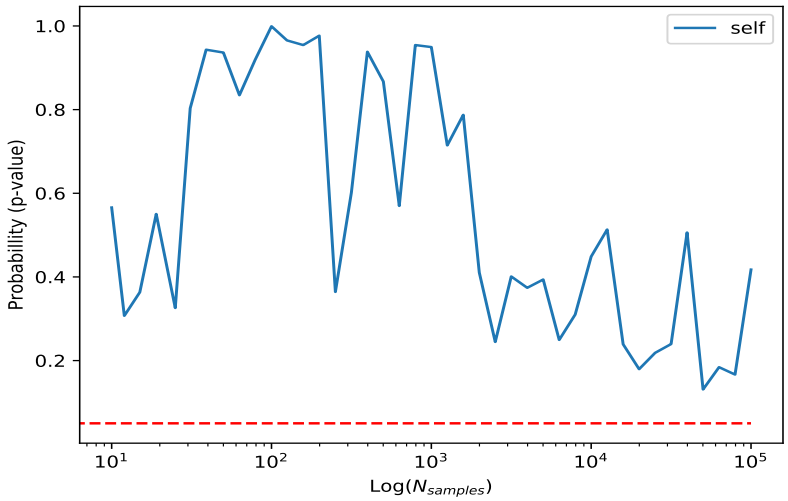

Figure 9: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the RNG always passes kuiper test. IT can however be seen that the value of the statistic stays lower for a significant amount of samples ($N_{samples} > 10^4$). This might indicate that there is flaw in the random number generator.

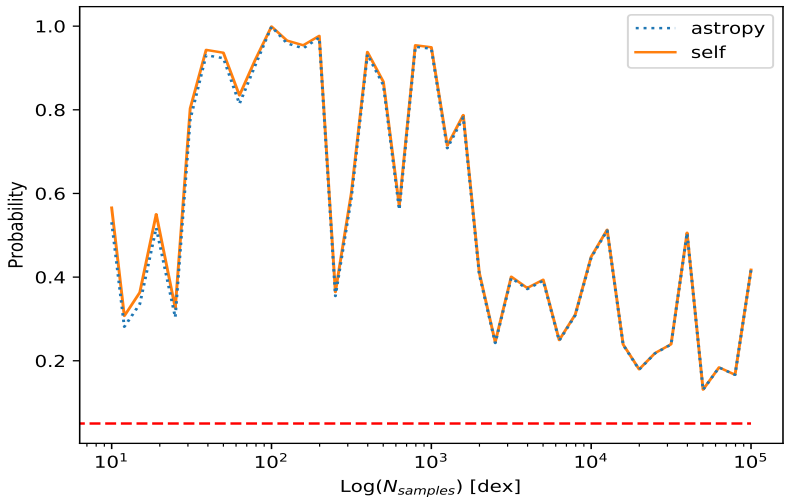

Figure 10: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the self written implementation deviates from the astropy implementation at the start, similar to what happend in the KS-test.