Numerical Recipes for Astrophysics
Solutions hand-in assignment-2

Luther Algra - s1633376

May 13, 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 and then giving a modified version of the obtained output to the MWC algorithm. The modification of the XOR-shifts output consists of putting the last 32 bits 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 algorithm for this modified value is set as new state of the random number generator. 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.

Code - Plots

The code for generating the plots. The initialization of the created random number generator is not explicitly shown in this piece of code but can be found on page .. where the full code is shown that contains all sub-questions together.

```
    """
        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

    # Print the seed.
    print('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')
```

./Code/assigment1.py

Code - Output text

The text output produced by the code:

```
Initial seed:  16702650
```

./Output/assigment1_out.txt

Code - Output plots

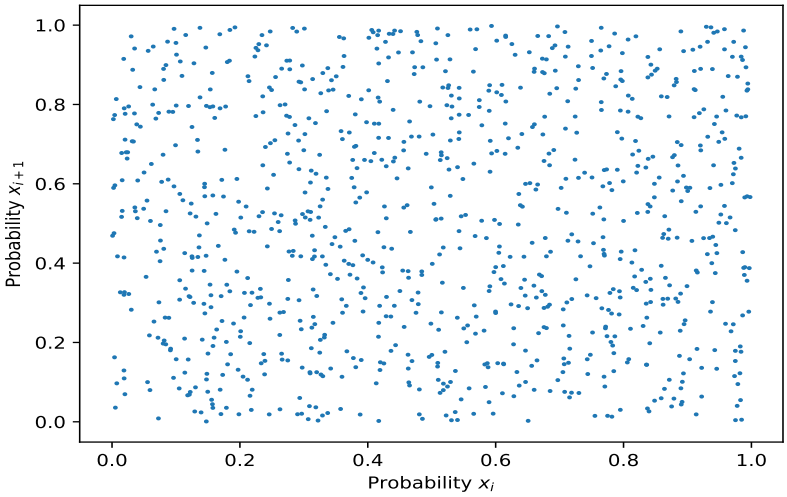

Figure 1: TODO

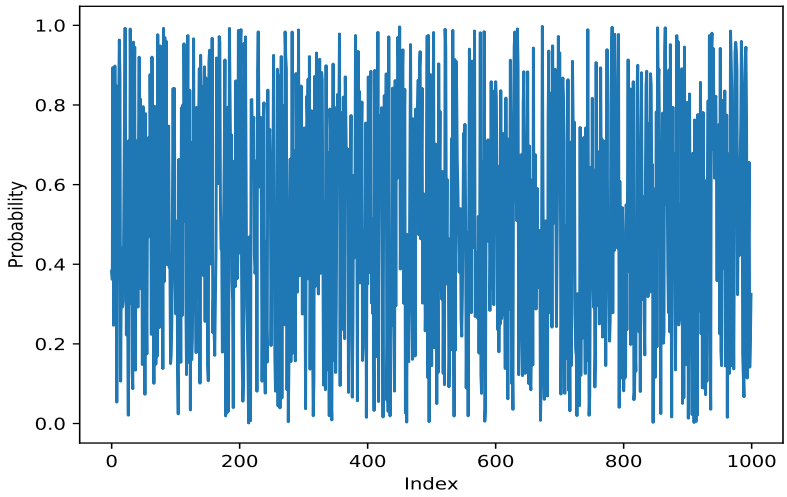

Figure 2: TODO

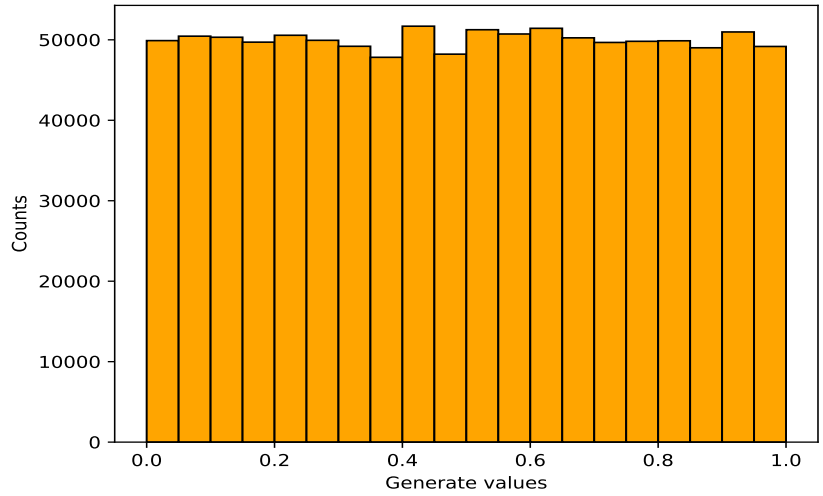

Figure 3: TODO

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 -5 σ 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 correct transformation of two i.i.d uniform variables to two i.i.d normal distributed variables with the Box-Muller method. This is done by first transforming the joined CDF of the two random Gaussian variables to polar coordinates. This transformation makes it possible to find the CDFs for the polar coordinates of the random Gaussian variables. 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 class that contains the random number generator (see page ...). The code for the plots and the created plots can be found below. The code for the creation of the plots makes besides the random number generator use of a function that represents the normal distribution in the file '.....'. This file is treated as shared module and can be found on page ...

Code - Plots

The code for generating the plots. The code for the function *mathlib.statistics.normal* can be found on page ..., where the shared modules are shown.

```
def assigment_1b(random):
    """
        Execute assigment 1.b
    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.statistics -- This contains the normal distribution is self


    # 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()
```

./Code/assigment1.py

Code - Output plot(s)

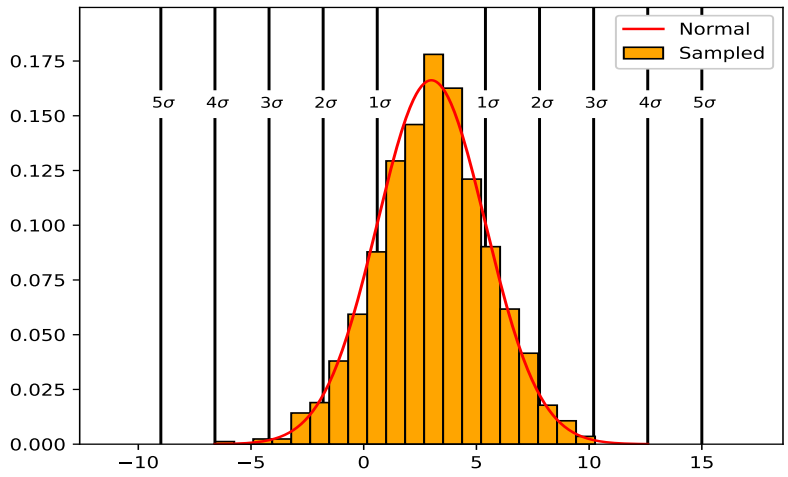

Figure 4: TODO

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,

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

where the erf is given by,

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

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,

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

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 implementation of the KS-tests and using it to test the null hypothesis that the samples follow a normal distribution is with the above approximations implemented. The code for the KS-test and the approximations for the CDF's is located in the file .... at page, as this file is threaded as module. The code for the creation of the plot is given below.

Code - Plots

The code for generating the two plots.

```
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) 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')
    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')
    plt.legend()
    plt.savefig('./Plots/1_plot_ks_test_self_scipy.pdf')
    plt.figure()
```

./Code/assigment1.py

Code - Output plot(s)

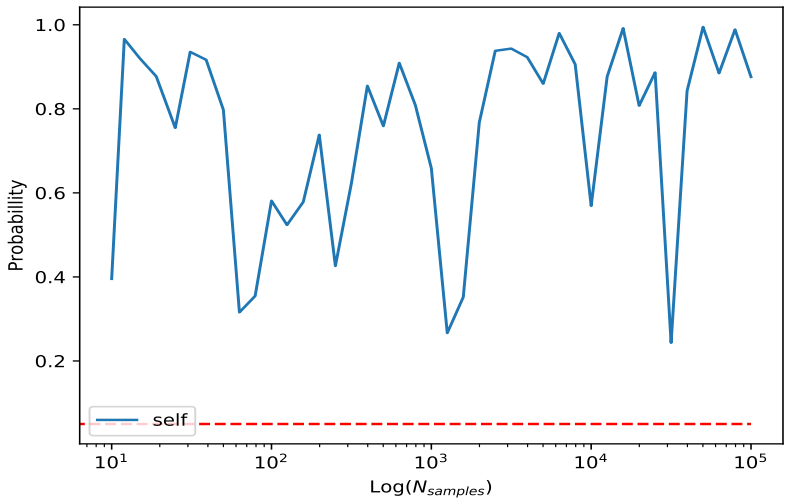

Figure 5: TODO

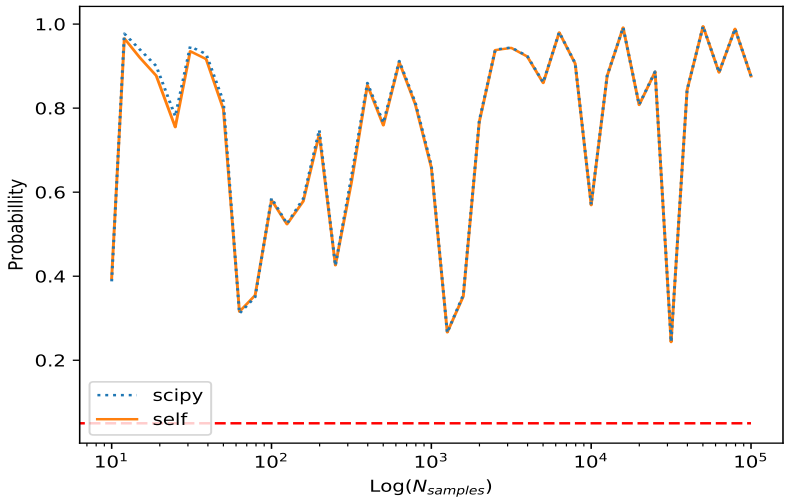

Figure 6: TODO