Numerical Recipes for Astrophysics
Solutions hand-in assignment-2

Luther Algra - s1633376

May 21, 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.

```python
    """
        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

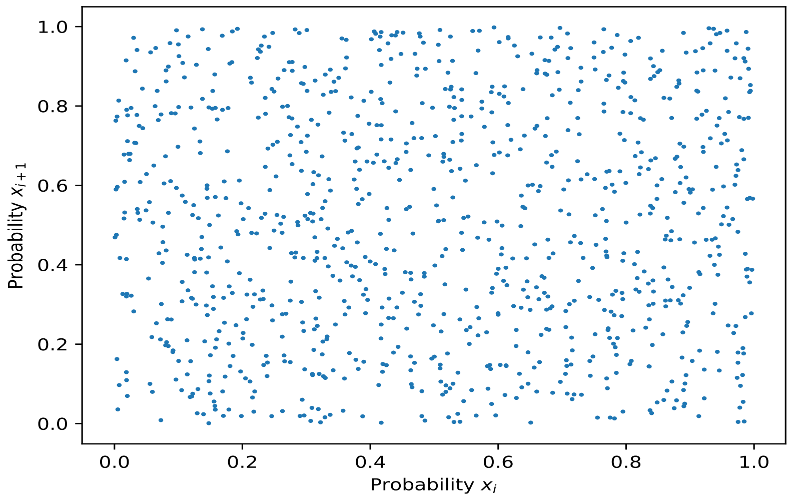

Figure 1: TODO

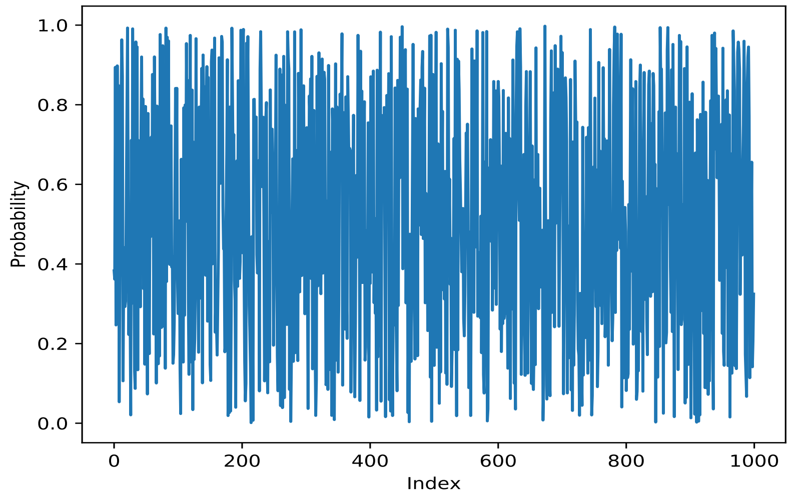

Figure 2: TODO

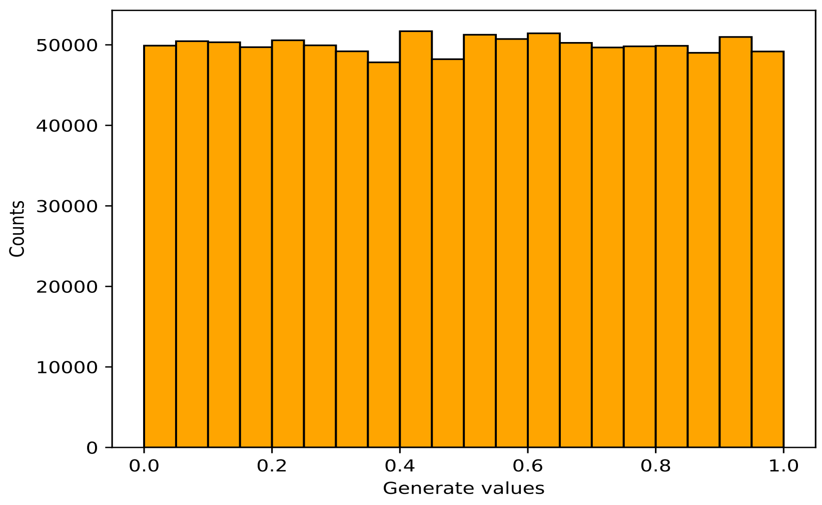

Figure 3: TODO

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,

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

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;

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

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

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)

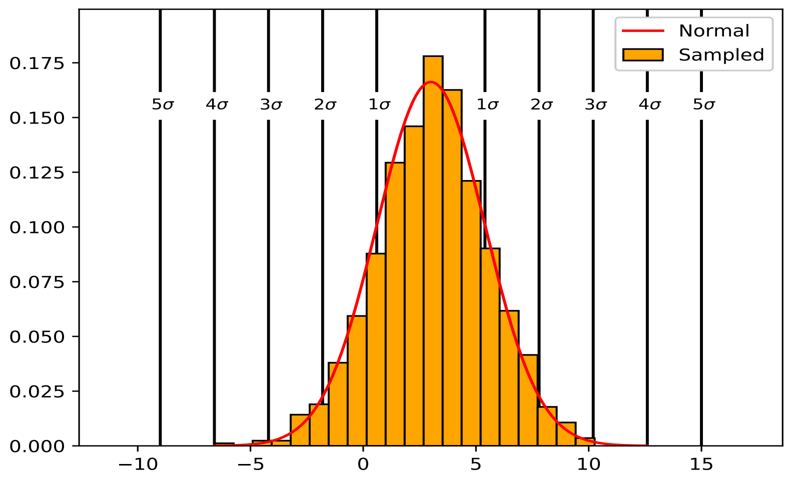

Figure 4: TODO

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,

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

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,

$$\text{erf}(x) \approx 1-(a_1t+a_2t^2+...+a_5t^5)e^{-x^2}\text{x} \quad t = \frac{1}{1+px} \tag{12}$$

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 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.

```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) 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)

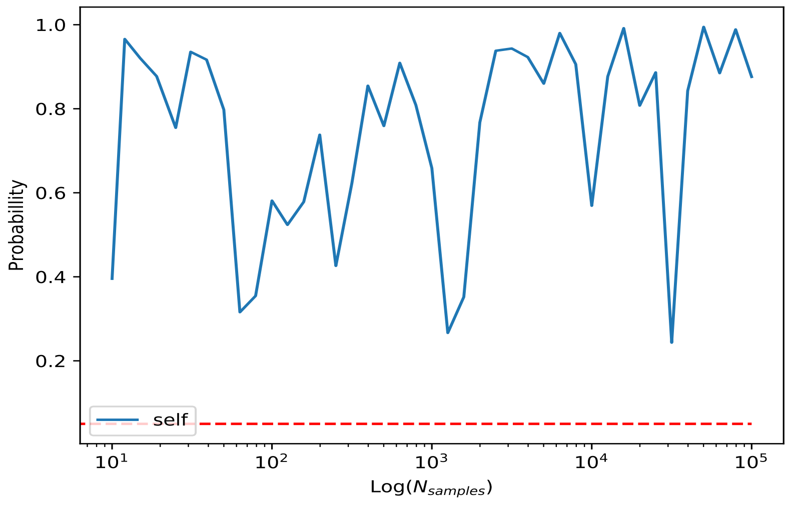

Figure 5: TODO

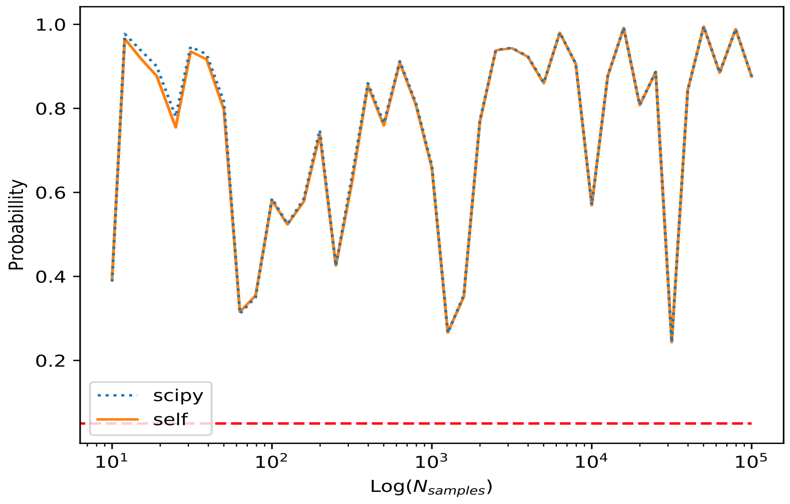

Figure 6: TODO

Question 1.c

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 kuiper statistics,

(1) CDF KS-statistic

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

$$P_{KS}(z) = \frac{2\sqrt{\pi}}{z} \sum_{j=1}^{\infty} \exp\left(-\frac{(2j-1)^2 + \pi^2}{8z^2} \right) \tag{13}$$

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,

$$P_{KS}(z) \approx \begin{cases} \frac{\sqrt{2\pi}}{z} \left[\left(e^{-\pi^2/(8z^2)} \right) +^9 + \left(e^{-\pi^2/(8z^2)} \right) \left(e^{-\pi^2/(8z^2)} \right)^{25} \right] & \text{for } z < 1.18 \\ 1 - 2\left[\left(e^{-2z^2} \right) - \left(e^{-2z^2} \right)^4 + \left(e^{-2z^2} \right)^9 \right] & \text{for } z >= 1.18 \end{cases} \tag{14}$$

(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,

$$\Phi\left(\frac{x-\mu}{\sigma} \right) = \frac{1}{2} \left[1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}} \right) \right] \tag{15}$$

where the erf is given by,

$$\text{erf}(x) \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{16}$$

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,

$$\text{erf}(x) \approx 1 - (a_1 t + a_2 t^2 + ... + a_5 t^5) e^{-x^2} \text{x} \quad t = \frac{1}{1+px} \tag{17}$$

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.

```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) 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)

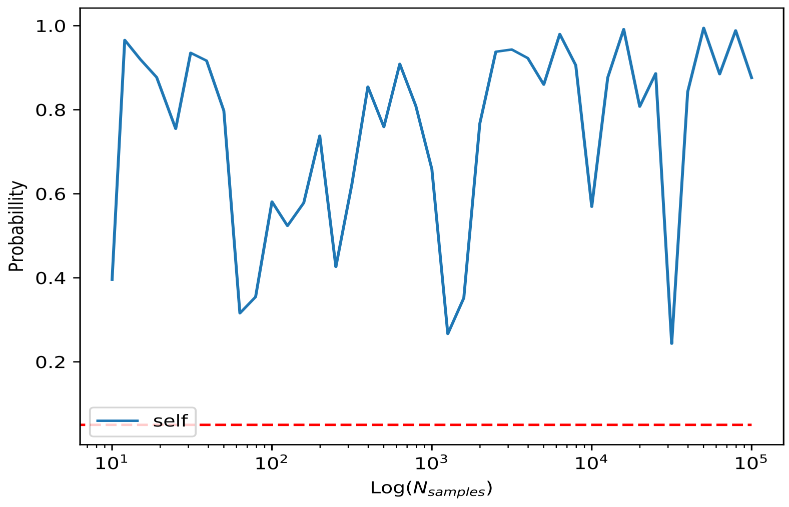

Figure 7: TODO

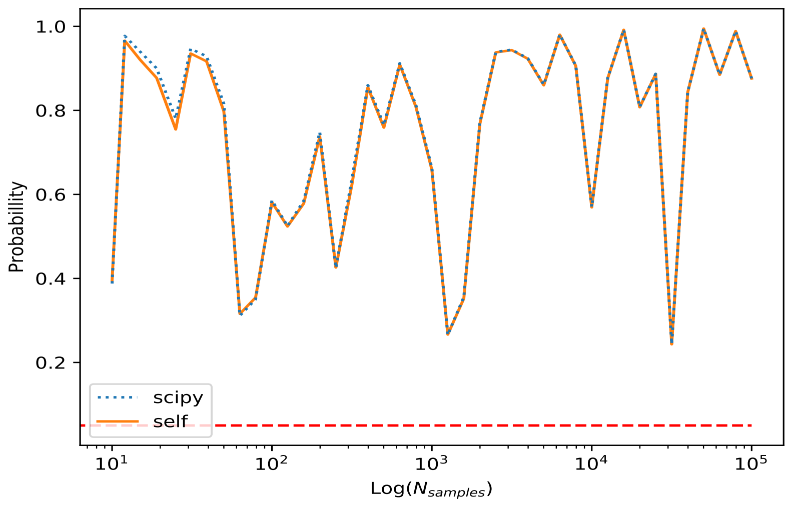

Figure 8: TODO

3 - Linear structure growth

Question 3

Problem

Solve the ODE of equation 18 for the 3 given initial conditions in an matter-dominated Einstein de Sitter Universe. Use an appropriate numerical method. Compare the results with the analytical solution of the ODE. Plot the solution for $t = 1$ until $t = 1000$ yr, use a log log plot.

$$\frac{d^2D}{dt^2} + 2\frac{\dot{a}}{a}\frac{dD}{dt} = \frac{3}{2}\omega_0 H_0^2 \frac{1}{a^3} D \tag{18}$$

Initial conditions:

(1) $D(1) = 3, D'(1) = 2$ (2) $D(1) = 10, D'(1) = -10$ (3) $D(1) = 5, D'(1) = 0$

Solution

The solution of this problem consist of three parts. One, a rewritten version of equation 18 with the scale factor plugged in. Two, a (brief) explanation on how this rewritten version is used numerically. Three, a derivation of the analytical solution.

(1) Rewriting the ODE.

The numerical and analytical solution both require a version of equation 18 with the scale factor plugged in. For an Einstein-de Sitter Universe the scale factor and its derivative are given by,

$$a(t) = \left(\frac{3}{2}H_0 t\right)^{2/3} \qquad \text{and} \qquad \dot{a}(t) = H_0 \left(\frac{3}{2}H_0 t\right)^{-1/3} \tag{19}$$

Plugin this in by equation 18 and using that $\Omega_0 = 1$ yields the rewritten version of equation 18,

$$\frac{d^2D}{dt^2} + \frac{H_0\left(\frac{3}{2}H_0 t\right)^{-1/3}}{\left(\frac{3}{2}H_0 t\right)^{2/3}}\frac{dD}{dt} - \frac{3}{2}\Omega_0 \frac{H_0^2}{\left(\frac{3}{2}H_0 t\right)^{2/3}} D = 0 \tag{20}$$

$$\frac{d^2D}{dt^2} + \frac{4}{3t}\frac{dD}{dt} - \frac{2}{3t^2} D = 0 \tag{21}$$

(2) Numerical solution

The numerical solution is obtained by first writing equation 21 as a system of first order ODE's and then by applying the Dormand Prince version of the Runde kutta method to solve it. The second order ODE can be written as a system of first order ODE's by substituting $dD/dt = u$. The system then becomes,

$$\begin{cases} \frac{dD}{dt} & = u \\ \frac{d^2D}{d^2} & = -\frac{4}{3t}u + \frac{2}{3t^2}D \end{cases} \tag{22}$$

The above system is as mentioned before solved with the Dormand Prince version of the Runde kutta method. The algorithm uses an adaptive step size that is initial set to $t_{step} = 0.01$ year for all cases. The obtained results c an be found on page .....

(3) Analytical solution

The analytical solution that is required for the plots can be found by solving equation 21. The equation is solved by finding two particular solutions. These can be found by finding the values of lambda for which the the ansatz $D(t) = t^\lambda$ holds. Plugin in the ansatz yields,

$$\lambda(\lambda - 1)t^{\lambda-2} + \frac{4}{3t}\lambda t^{\lambda-1} - \frac{2}{3t^2}t^\lambda = 0 \tag{23}$$

This simplifies to

$$\begin{aligned}0 &= \lambda(\lambda-1)t^{\lambda} + \frac{4}{3}\lambda t^{\lambda} - \frac{2}{3}t^{\lambda}\\ &= \lambda(\lambda-1) + \frac{4}{3}\lambda - \frac{2}{3}\\ &= \lambda^2 + \frac{1}{3}\lambda - \frac{2}{3}\\ &= (\lambda+1)(\lambda-\frac{2}{3})\end{aligned}$$

The peculiar solutions of the ODE are thus given by,

$$D(t) = t^{-1} \qquad\qquad D(t) = t^{2/3} \tag{24}$$

The general solution is the super position of the peculair solutions and can therefore be written as,

$$D(t) = c_1 t^{2/3} + c_2 t^{-1} \tag{25}$$

The constants for the three initial cases can be found by calculating the derivative of the above equation and plugin in t he initial conditions. This yields for the three cases that,

$$(1)\ c_1 = 3, c_2 = 0 \qquad (2)\ c_1 = 0, c_2 = 10, \qquad (3)\ c_1 = 3, c_2 = 2 \tag{26}$$

The code that is used to solve the ODE numerically and generates the plots is split over two files. The first file generates the plots and the second file contains the implementation of the Kunde gutta method. The code and its output can be found below.

Code

The code for generating the three plots.

```python
import numpy as np
import mathlib.ode as ml_ode
import matplotlib.pyplot as plt

def main():

    # The constants of the anlytical solution for the 3 cases.
    c1_cases = [3, 0, 3]
    c2_cases = [0, 10, 2]

    # The initial conditions for the ODE solver of the 3 cases.
    initial = [[3,2],[10, -10], [5,0]]

    # The start and stop time to solve the ODE for.
    t_start = 1 # year
    t_stop = 1000 # years

    # Initial step size for the numerical solution
    t_step = 0.01 # year

    # The time values to plot the anlytical solution for.
    t_plot = np.arange(t_start, t_stop+t_step, t_step)

    # Create the plots
    for case in range(len(c1_cases)):

        # Constants for the anlytical solution.
        c1 = c1_cases[case]
        c2 = c2_cases[case]

        # Initial conditions for the numerical solution
        initial_cond = np.array(initial[case])

        # The analytical solution.
        analytical = lambda t: c1*t**(2/3)+ c2*t**(-1)

        # The numerical solutions.
        sol_num, time = ml_ode.runge_kutta_54(_linear_density_growth,
    initial_cond, t_start, t_stop, t_step, 1e-6, 1e-3)

        # Plot the analytical and numeric solution.
        plt.plot(t_plot, analytical(t_plot), label='Analytical', linestyle=':',
    zorder=0.1)
        plt.plot(time, sol_num[:,0], label='Numeric',zorder=0)
        plt.xlabel('Time [year]')
        plt.ylabel('D(t)')
        plt.loglog()
        plt.legend()
        plt.savefig('./Plots/3_ode_{0}.pdf'.format(case))
        plt.figure()


def _linear_density_growth(values,t):
    """
        A function representing the sytem of ODE's that needs
        to be solved for the lineer density growth equation.
    In:
        param: values -- The current values of the linear growth function and its
     derivatie.
        param: t -- The current time step for which the ODE is integrated.
    Out:
        return: An array representing the system of first order  ODE's for the
    given parameters.
    """

    # Current value of the linear growth function.
    d = values[0]
    # Current value of the derivative of the linear growth function.
    u = values[1]

    # The two systems of first order ODE's.
    first = u
    second = -(4/(3*t))*u + (2/(3*t**2))*d

    return np.array([first, second])


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

./Code/assigment_3.py

The code for the Runde kutta method

```python
import numpy as np


def runge_kutta_54(func, y0, t_start, t_stop, t_step, atol=1e-6, rtol=1e-3):
    """
        Perform the 4the order runga_kutta method for first order ODE integration.
    In:
        param: func -- The function describing the differential equation or the system
                       of first order ODEs to integrate. Must return a numpy array.
        param: y0   -- The initial conditions.
        param: t_start -- The time to start integration at.
        param: t_stop -- The time to stop integration at.
        param: t_step -- The initial step size to use.
        param: steps -- The step size to use for integration.
        param: order -- The order of the algorithm to use.
    """


    # If true, solving a single ODE, else solving a system.
    if type(y0) is not np.ndarray:
        y0 = np.array([y0]) # convert to array


    # Array with values to return, for both the integrated
    # values and the time stemps. The size is increased by a
    # factor of 2 when needed.

    ret = np.zeros((int((t_stop-t_start)/t_step)+1, len(y0)))
    time = np.zeros(int((t_stop-t_start)/t_step)+1)

    # Set initial state.
    ret[0] = y0
    time[0] = t_start

    # Solve the ODE or the system of ODEs
    min_update_scale = 0.2
    max_update_scale = 10

    # Current time at the integration
    t_now = t_start
    # Total amount of executed steps
    steps = 1  # skip zero

    # Current error
    error = 1.1
    y_next = 0

    while t_now <= t_stop:

        # Check if we need to expand the return arrays
        if steps >= ret.shape[0]:
            ret_old = ret.copy()
            ret = np.zeros((ret_old.shape[0]*2, ret_old.shape[1]))
            ret[0:steps] = ret_old

            time_old = time.copy()
            time = np.zeros(time_old.shape[0]*2)
            time[0:steps] = time_old


        # Get the value found at the previous step
        previous = ret[steps-1]

        # Calculate the constants for the runge kutta method. TODO exact name
        k1 = t_step*func(previous, t_now)
        k2 = t_step*func(previous + (1/5)*k1, t_now + (1/5)*t_step)
        k3 = t_step*func(previous + (3/40)*k1 + (9/40)*k2, t_now + (3/10)*t_step)
        k4 = t_step*func(previous + (44/45)*k1 - (56/15)*k2 + (32/9)*k3, t_now + (4/5)*t_step)
        k5 = t_step*func(previous + (19372/6561)*k1 - (25360/2187)*k2 + (64448/6561)*k3 - (212/729)*k4, t_now + (8/9)*t_step)
        k6 = t_step*func(previous + (9017/3168)*k1 - (355/33)*k2 + (46732/5247)*k3 + (49/176)*k4 - (5103/18656)*k5, t_now + t_step)

        # Calculate the new value.
        y_next = previous + ( (35/384)*k1 + (500/1113)*k3 + (125/192)*k4 - (2187/6784)*k5 + (11/84)*k6)
        y_embedded = previous + ( (5179/57600)*k1 + (7571/16695)*k3 + (393/640)*k4 -(92097/339200)*k5 + (187/2100)*k6 )

        # Calculate error
        delta = abs(y_embedded  - y_next)
        scale = atol+np.maximum(abs(previous),abs(y_next))*rtol
        error = np.sqrt(np.sum((delta/scale)**2)/len(k1))

        # The factor used to calculate the new step size.
        update_scale = 0.9*(error)**(-0.2)

        # Make sure the factor is not to large or to small.
        if error == 0:
            update_scale = max_update_scale
        elif update_scale < min_update_scale:
            update_scale = min_update_scale
        elif update_scale > max_update_scale:
            update_scale = max_update_scale

        # Check if the current step should be accepted.
        if error > 1: # reject
            t_step*= min(update_scale, 1.0)
        else: # accept
            t_now += t_step
            t_step *= update_scale

            ret[steps] = y_next
            time[steps] = t_now
            steps += 1


    return ret[0:steps], time[0:steps]
```

./Code/mathlib/ode.py

Code - Output plot(s)

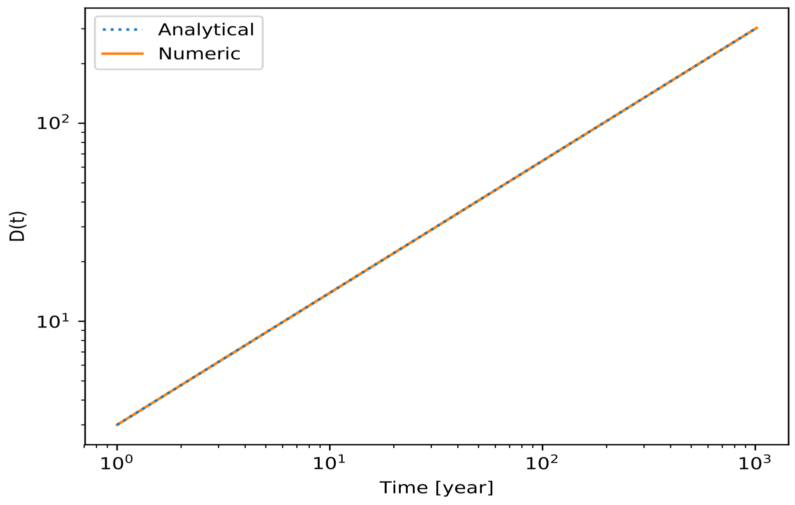

Figure 9: TODO

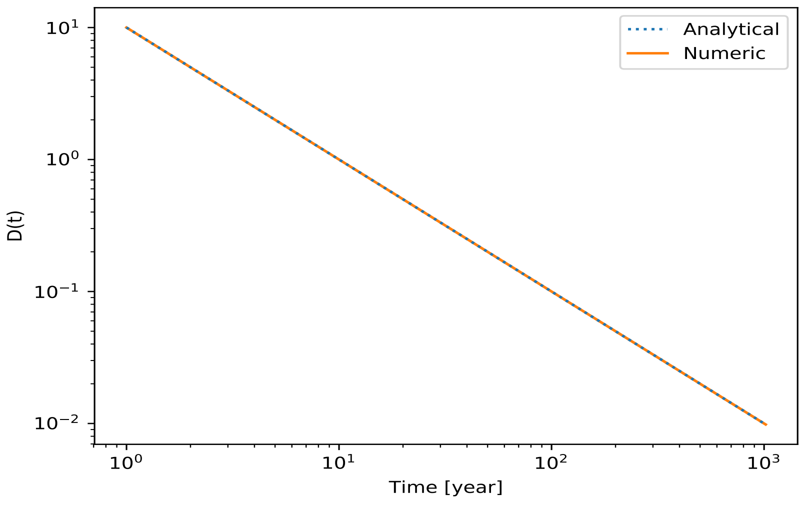

Figure 10: TODO

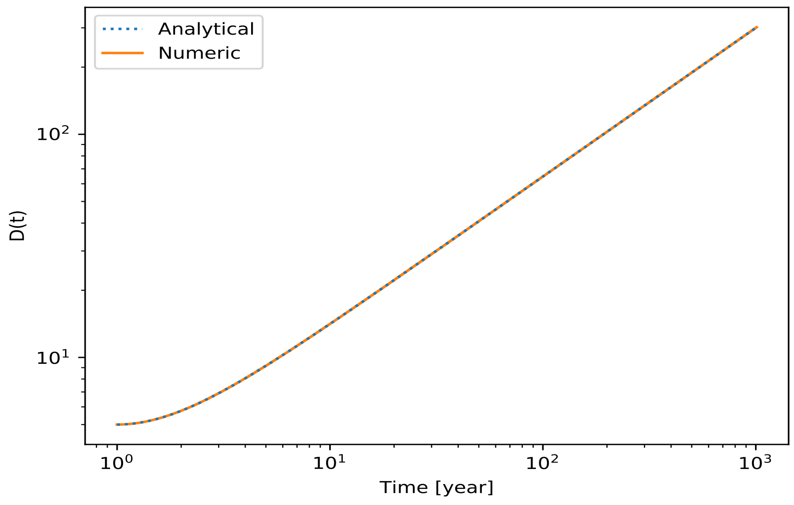

Figure 11: TODO

4 - Zeldovich approximation

Question 4.a

Problem

The linear growth factor is expressed in terms of a integral expression given by,

$$D(z) = \frac{5\Omega_m H_0^2}{2} H(z) \int_z^{\infty} \frac{1+z'}{H^3(z')} dz' \tag{27}$$

Here z is the redshift, Ω_m is the matter fraction of the Universe at $z = 0$ ($\Omega_m = 0.3$), H_0 is the Hubble constant at $z = 0$ and $H(z)$ is the redshift dependent Hubble parameter given by,

$$H(z)^2 = H_0^2 \left(\Omega_m(1+z)^3 + \Omega_\Lambda\right) \tag{28}$$

Here Ω_λ is the dark energy fraction of the Universe given by $\Omega_\lambda = 0.7$. Use numerical integration to calculate the growth factor at $z = 50$ with a relative accuracy of 10^{-5}. Note that $D(a(z = 50)) = D(z = 50)$, so use either variable.

Solution

The equation is before integrating first written in terms of the scale factor a. Substituting $a = 1/(1+z)$ yields,

$$dz = -(1+z)^2 da = -a^{-2} da \tag{29}$$

Plugin this in by equation 27 results in,

$$D(a) = \frac{5\Omega_m H_0^2}{2} H(a) \int_a^0 \frac{-a'^{-3}}{H^3(a')} da' = \frac{5\Omega_m H_0^2}{2} H(a) \int_0^a \frac{a'^{-3}}{H^3(a')} da' \tag{30}$$

Here the Hubble parameter in terms of the scale factor a is given by,

$$H(a) = H_0^2(\Omega_m a^{-3} + \Omega_\Lambda) \tag{31}$$

Filling this in by equation 30 and simplifying yields,

$$D(a) = \frac{5\Omega_m H_0^3}{2} (\Omega_m a^{-3} + \Omega_\Lambda) \int_0^a \frac{a'^{-3}s}{(H_0^2(\Omega_m a^{-3} + \Omega_\Lambda))^{3/2}} da' \tag{32}$$

$$= \frac{5\Omega_m}{2} (\Omega_m a^{-3} + \Omega_\Lambda) \int_0^{a'} \frac{a'^{-3}}{(\Omega_m a^{-3} + \Omega_\lambda)^{3/2}} da' \tag{33}$$

The above integral is with the help of Romberg integration solved for $\Omega_m = 0.3$ and $\Omega_\lambda = 0.7$.