Numerical Recipes for Astrophysics
Solutions hand-in assignment-2

Luther Algra - s1633376

May 28, 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. If necessary some of the results are discussed here. The output of the code is always presented after the code and plots are discussed in their caption.

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 that is 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 19). 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 19.

```
    """
        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))
```

./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

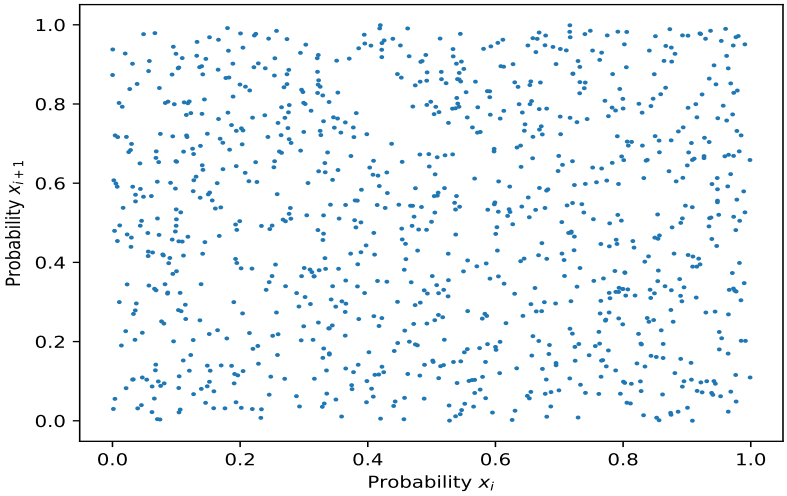

Figure 1: A plot of random number x_{i+1} against random number x_i for the first 1000 random uniforms produced by the random number generator. A good random number generator should produce a homogeneous plot without many (large) empty spots. The largest empty spot in the above plot is at $x_i = 0.4$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

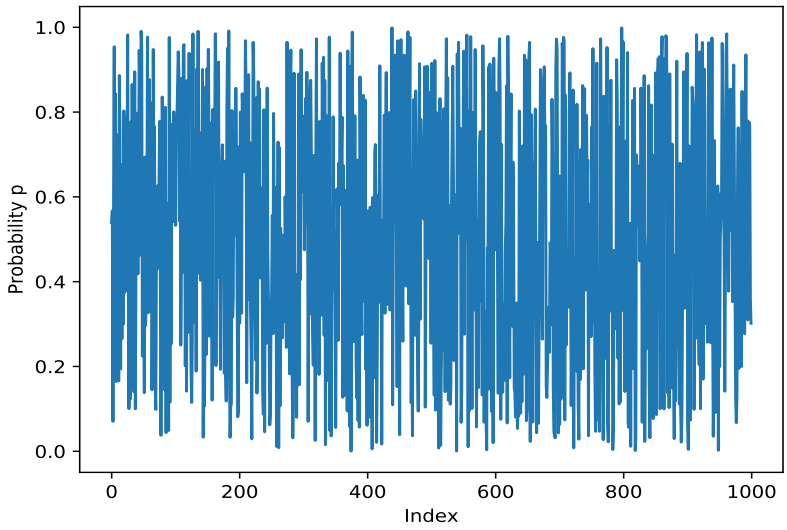

Figure 2: The first 1000 random uniform numbers produced by the random number generator (RNG) against their index. A good random number generator should not have large wide gaps (e.g when moving from index 400 to 450 it should not only produce values larger than 0.8, which would leave a wide gap). In the plot small gaps appear, see for example index ~ 420 at a probability of $p = 0.6$. 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 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 $p = 0.2$.

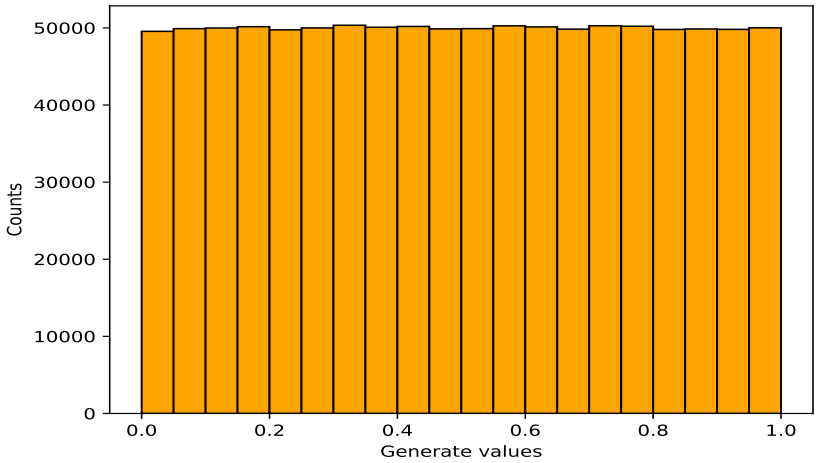

Figure 3: The uniforms of the random number generator for 1 million random values. The values are binned in 20 bins. A good random number generator should fluctuate around 50000 $\pm 2\sqrt{50000} = 50000 \pm 447$ counts per bin (2 sigma). The maximum and minimum amount of counts corresponds to 50343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

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

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,

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

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

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 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 above transformation are implemented in the random number generator (see page 19). 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 22. The called function, `normal`, can be found on line 227 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 19 and 22.

```
1  def assigment_1b(random):
2      """
3          Execute assigment 1.b
4      Int:
5          param: random -- An instance of the random number generator.
6      """
7
8      # The relevant imports for this piece of code are:
9
10     # (1) matplotlib.pyplot as plt
11     # (2) mathlib.random as random
12     # (3) mathlib.stats as ml_stats
13     # (4) numpy as np
14
15     # Sigma and mean for the distribution.
16     mean = 3.0
17     sigma = 2.4
18
19     # Generate 1000 random normal variables for the given mean and sigma.
```

```
    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/1.2, 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/1.2, 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/assigment_1.py

Code - Output plot(s)

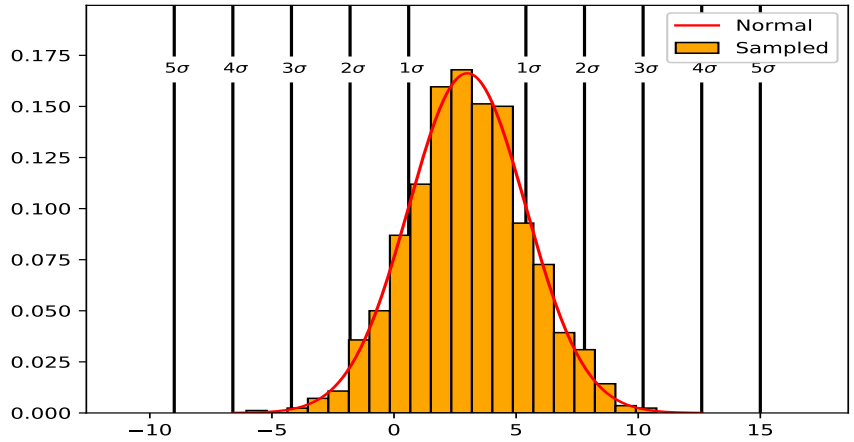

Figure 4: A histogram of the 1000 random normal distributed variables generated with the box muller method for $\mu = 3$ and $\sigma = 2.4$ (orange). The red line is the true normal distribution. The histogram appears to approximate the distribution quite well, but displays small deviations. The bin left of the peak (the highest bin) is larger than it should be. The first bins right of the peak is smaller than it should be and the second bin right of the peak is to high. The histogram does however still appear to be acceptable by eye. 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 for the above equation 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,

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

where the erf is given by,

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

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

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

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 22, 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 26. 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')
    plt.figure()
```

./Code/assigment_1.py

Code - Output plot(s)

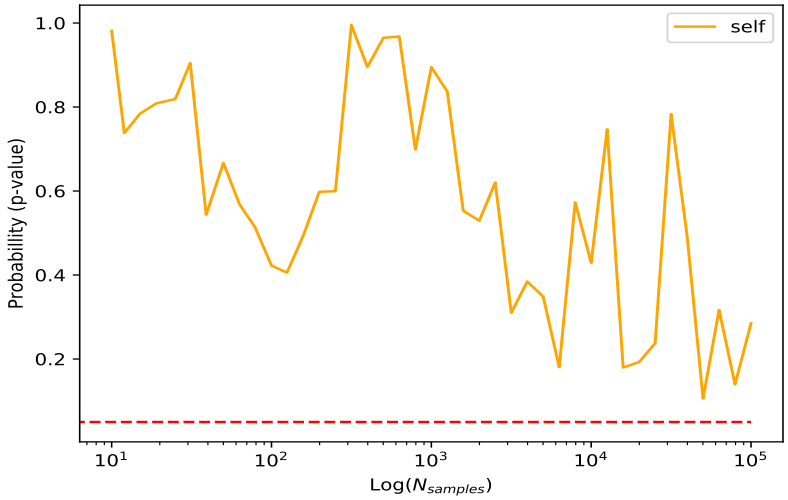

Figure 5: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** the line would suggests that there is enough statistical evidence to reject the (null) hypothesis that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

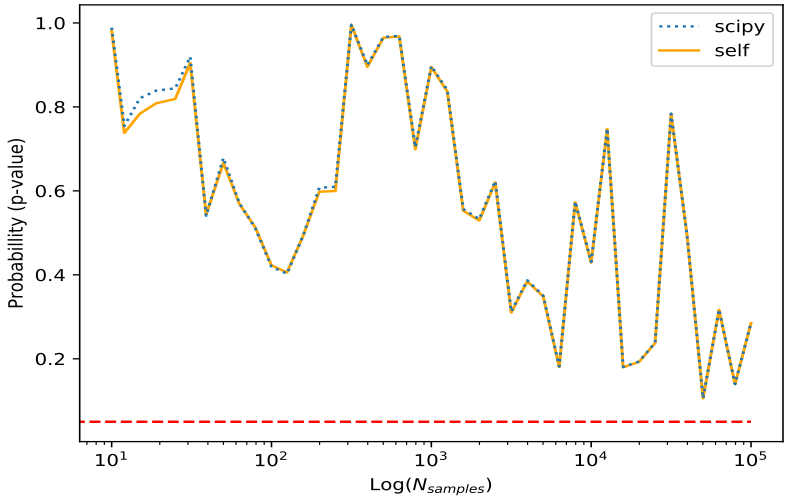

Figure 6: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. The orange line is the self written implementation of the KS-test and the blue line is the scipy version. A point **below** 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 (see $N_{samples} = 10$ 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.

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` that can be found on page 22. 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.

```
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] Actually, less terms are enough. The evaluation of the sum could therefore stop early by checking for a required precision.

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

./Code/assigment_1.py

Code - Output plot(s)

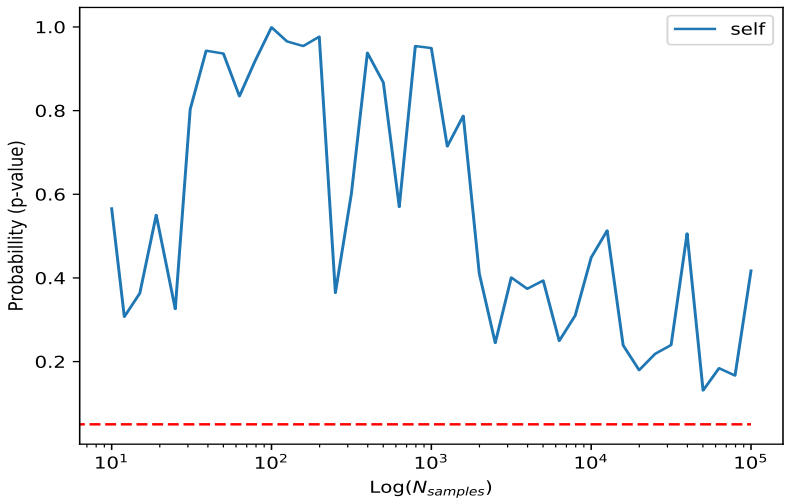

Figure 7: The P-value produced by the Kuiper-test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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 drops for larger sample size, similar as what happens by the KS-test. This might, as mentioned before, indicate that there is an flaw in the random number generator.

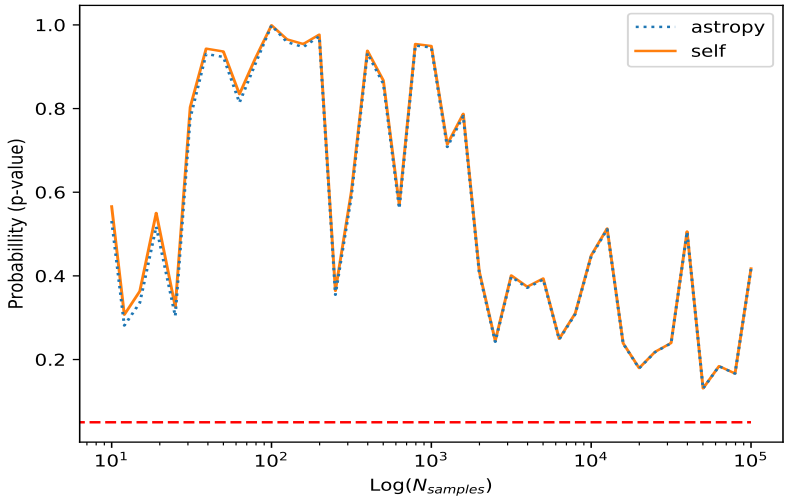

Figure 8: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis that the data is normal distributed. The plot shows that the self written implementation 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-test 2 (2 sample distributions). The KS-test 2 has to be applied to 10 columns. The random numbers that are compared with the column data are as result only generated once (i.e each column is compared with the same random numbers). This was done to save computation time. The code that contains the implementation of the ks-test 2 can be found on page 22. 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 4 (index 3, see figure 12 on page 15). 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.

```
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))
        plt.figure()
```

./Code/assigment_1.py

Code - Output plot(s)
Note: Expect very similar captions for the upcoming 10 plots.

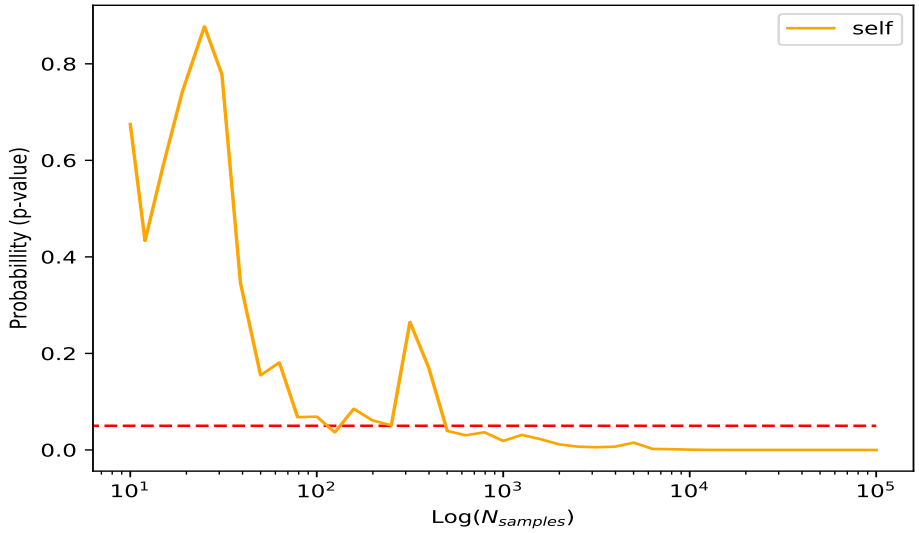

Figure 9: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **first** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

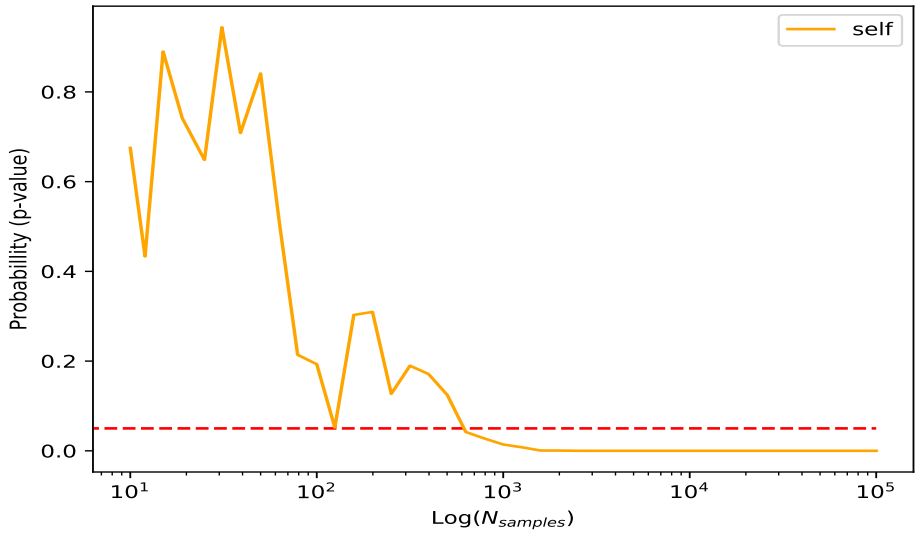

Figure 10: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **second** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

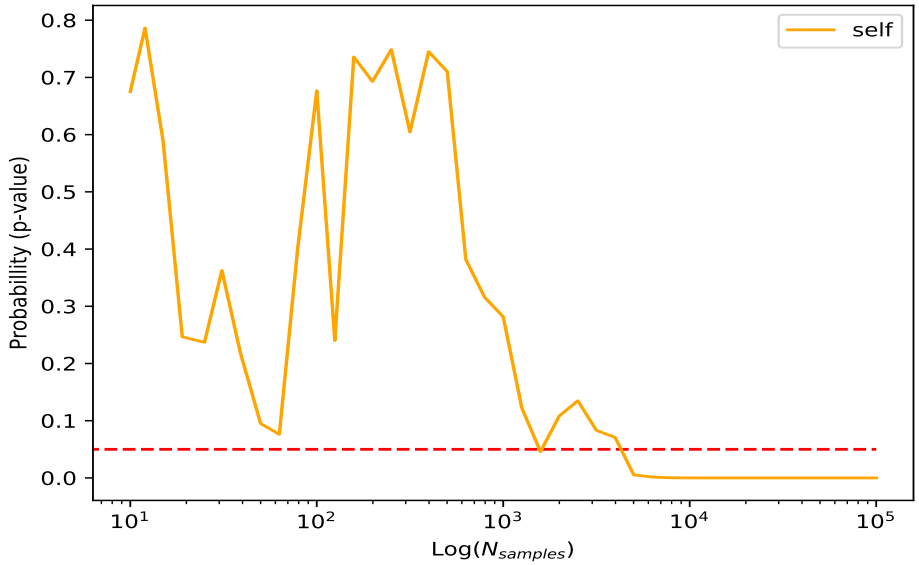

Figure 11: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **third** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

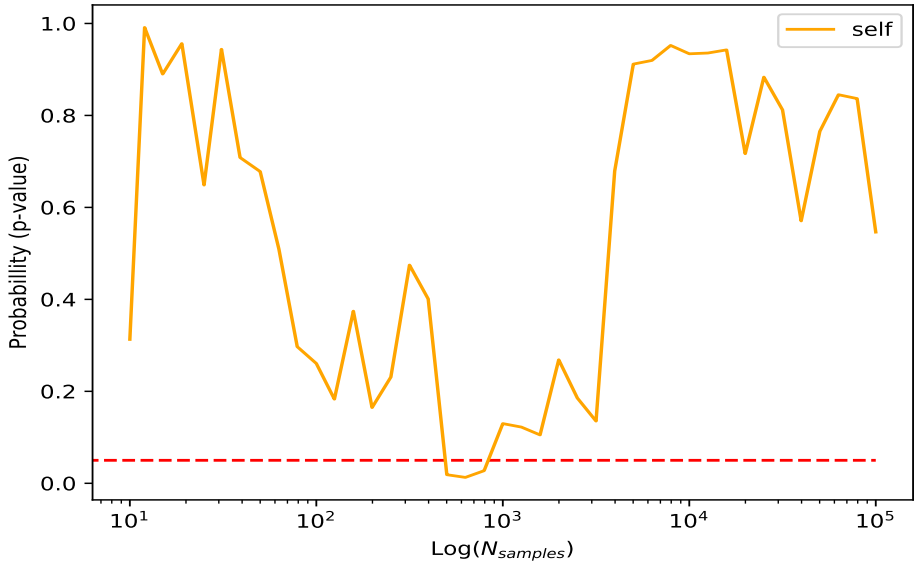

Figure 12: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **fourth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to **reject** the (null) 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. The most important point in this plot is the p-value when all samples are included. It can be clearly seen that this point isn't below the red line. The plot therefore suggests that the given c olumn might(see footnote 2 on page 12) be a Gaussian with $\mu = 0$ and $\sigma = 1$.

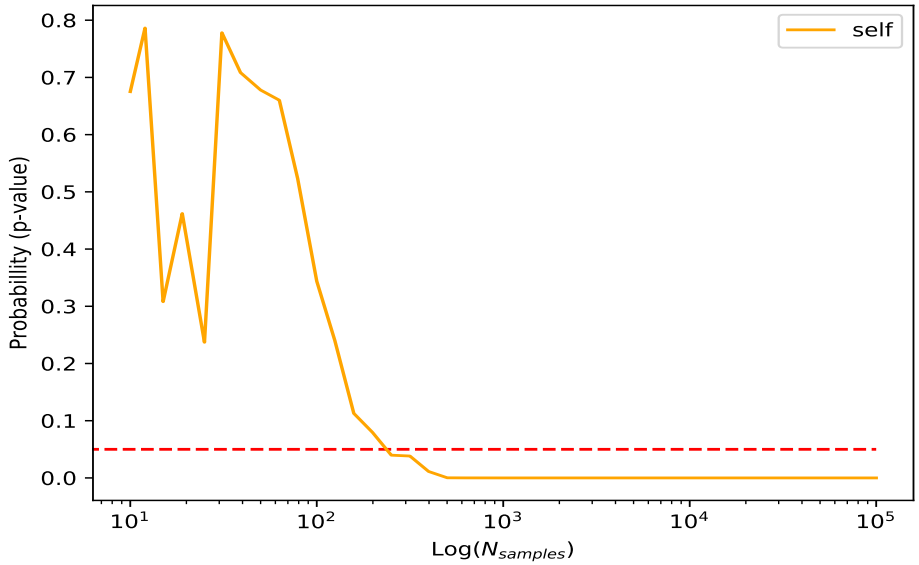

Figure 13: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **fifth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

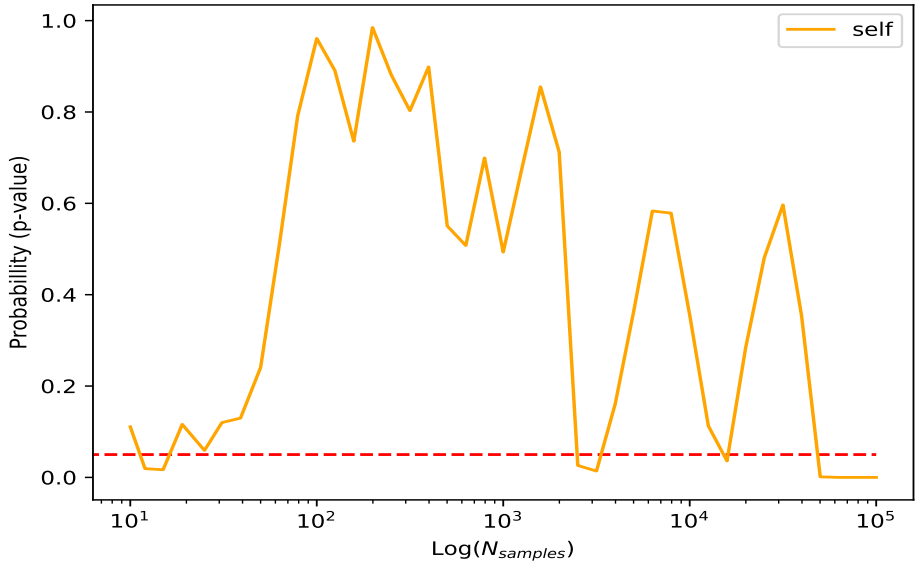

Figure 14: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **sixth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

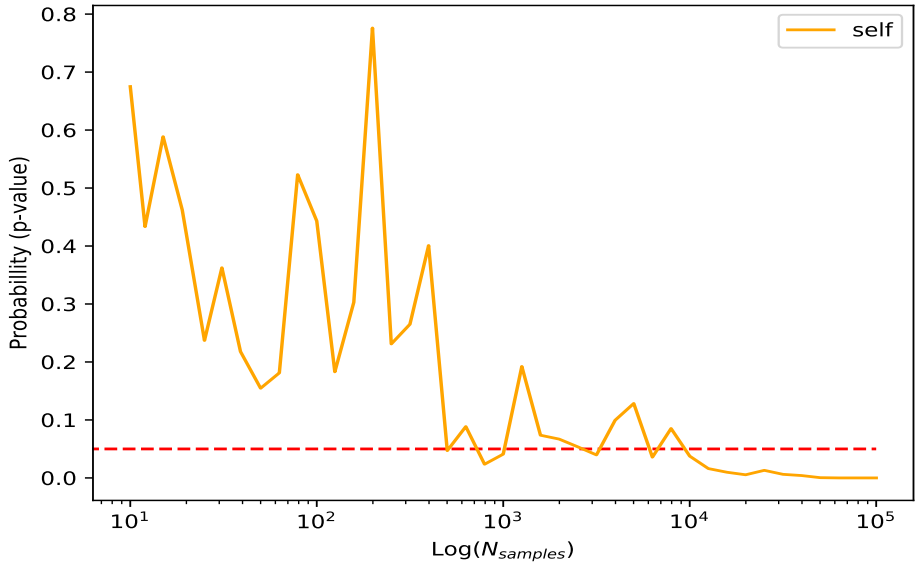

Figure 15: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **seventh** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

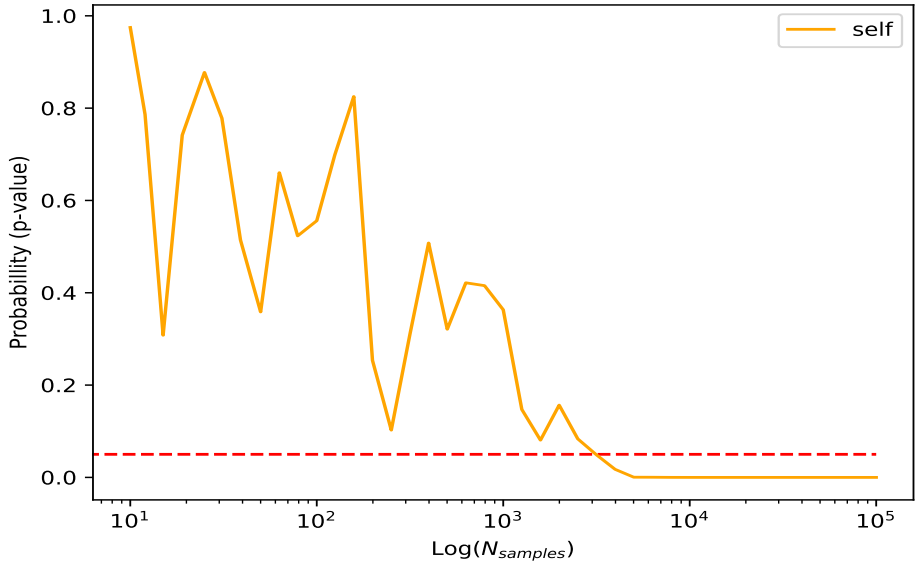

Figure 16: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **eight** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

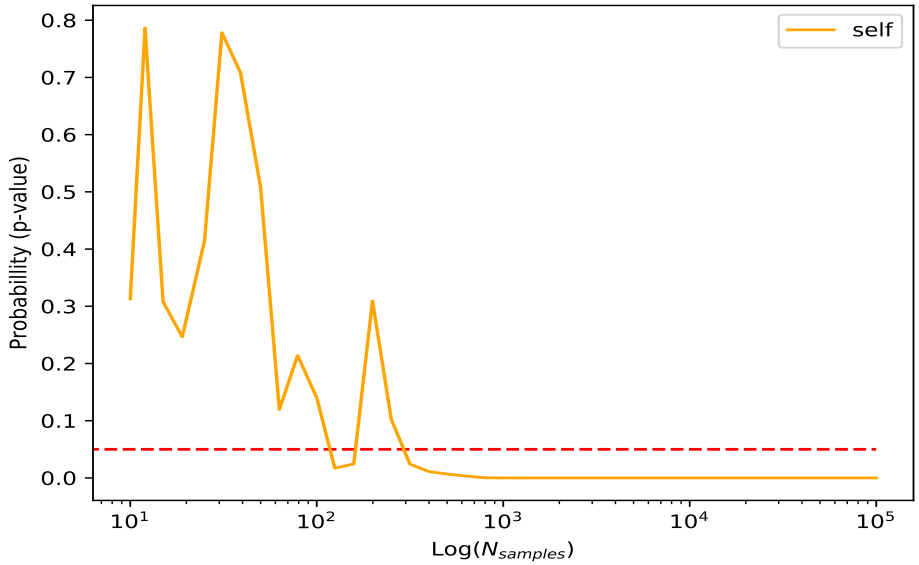

Figure 17: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **ninth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

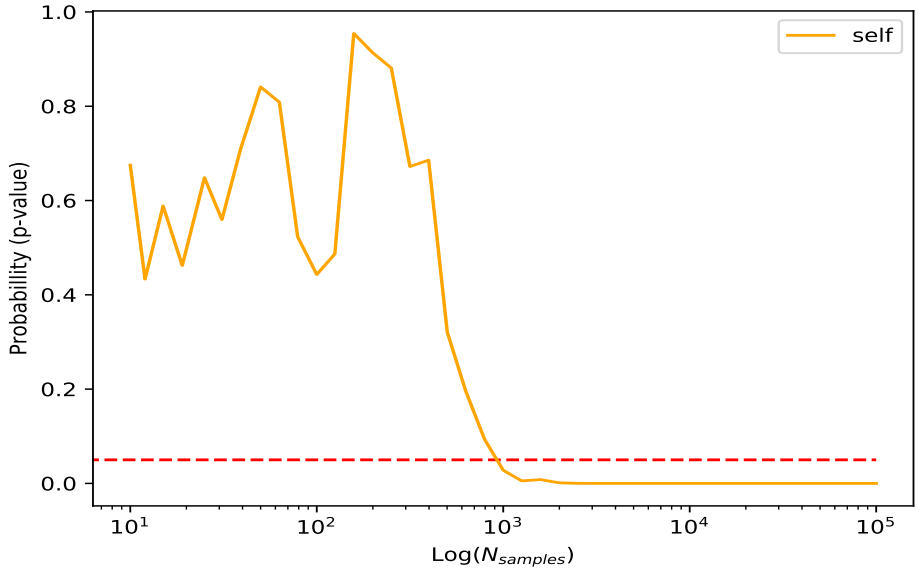

Figure 18: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **tenth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

Code - Assignment

The code of the file that executes the sub-questions. The code presented below only consists of the part that has not jet been shown in the subquestions. This part consists of the initialization of the random number generator (1.a) and the used imports.

```
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)
```

./Code/assigment_1.py

Code - Random Number Generator

The code for the random number generator.

```
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)
```

```
32          self._mwc_base = np.uint64(2**32)
33
34      def get_seed(self):
35          """
36              Get the seed that is used to initialize this generator.
37
38          Out:
39              return: The seed used to initalize the generatorr.
40          """
41          return self._seed
42
43      def get_state(self):
44          """
45              Get the state of the generator.
46
47          Out:
48              return: The state of the generator.
49          """
50          return self._state
51
52
53      def gen_next_int(self):
54          """
55              Generate a new random 32-bit unsigned integer.
56          Out:
57              return: A random 32-bit unsigned integer.
58          """
59
60          # The state is at the end updated with mwc.
61          # We therefore shouldn't use more than 32 bits to generate
62          # the number.
63
64          return self._update_state() & self._uint32_max
65
66      def gen_uniform(self):
67          """
68              Generate a random float between 0 and 1.
69          Out:
70              return: A random float between 0 and 1.
71          """
72
73          return self.gen_next_int()*1.0 / self._uint32_max
74
75      def gen_uniforms(self, amount):
76          """
77              Generate multiple random floats
78              between 0 and 1.
79          In:
80              param: amount -- The amount of floats to generate.
81          Out:
82              return: An array with 'amount' random floats
83                      between 0 and 1.
84          """
85
86          samples = np.zeros(amount)
87
88          for i in range(amount):
89              samples[i] = self.gen_uniform()
90
91          return samples
92
93      def gen_normal(self, mean, sigma):
94          """
95              Generate a random normal distributed float.
96
97          In:
98              param: mean -- The mean of the gaussian distribution.
99              param: sigma -- The squareroot of the variance of the distribution.
100         Out:
101             return: A random float that is drawn from the parameterized normal
102                     distribution.
```

```
        """

        # 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.
        """
```

```
172
173         self._state = self._xor_shift(self._state)
174         self._state = self._mwc(self._state & self._uint32_max) ^ self._state
175
176         return self._state
177
178
179     def _xor_shift(self, number):
180         """
181             Execute the XOR-shift algorithm on the
182             input number.
183         In:
184             param: number -- The number to XOR-shift.
185         Out:
186             return: The number produced by XOR-shift.
187          """
188
189         # Shift to the right and then bitwise xor.
190         number ^= (number >> self._xor_a1)
191         # Shift to the left and then bitwise xor.
192         number ^= (number << self._xor_a2)
193         # Shift to the right and then bitwise xor.
194         number ^= (number >> self._xor_a3)
195
196         return number
197
198     def _mwc(self, number):
199         """
200            Perform multiply with carry (MWC) on
201            the given input.
202         In:
203             param: number -- The number to perform MWC on, must be an uint64.
204         Out:
205             return: The new number.
206         """
207         return self._mwc_a * (number & (self._uint32_max -np.uint64(1))) + (
    number >> np.uint64(32))
```

./Code/mathlib/random.py

Code - Statistical functions

The code that containing all statistical functions that where needed for the sub-questions. The imported sorting module used by this piece of code can be found on page 27.

```
1  import numpy as np
2  import mathlib.sorting as sorting
3
4
5  def kstest(x, cdf):
6      """
7          Perform the Kolmogorov-Smirnov test for goodness of fit
8          and return the p-value.
9      In:
10         param: x   -- An array with value's who's CDF is expected to be
11                       the same as the provided CDF. Must be atleast size 4
12
13         param: cdf -- A function that is the expected cdf under the null
       hypothesis.
14     Out:
15         return: The p-value obtained by performing the KS-test.
16     """
17
18     # Amount of values in the input array.
19     x_size = len(x)
20
21     # Sort the values and evaluate the cdf.
22     x_sorted = sorting.merge_sort(x)
23     x_sorted_cdf = cdf(x_sorted)
```

```
24
25      # Maximum distance.
26      max_dist = 0
27
28      # Value of the emperical cdf at step i-1.
29      x_cdf_emperical_previous = 0
30
31      # Find the maximum distance.
32      for idx in range(0,x_size):
33
34          # Calculate the emperical cdf.
35          x_cdf_emperical = (idx+1)/x_size
36          # The true cdf evaluation at the given point.
37          x_cdf_true = x_sorted_cdf[idx]
38
39          # Find the distance. The emperical
40          # CDF is a step function so there are two distances
41          # that need to be checked at each step.
42
43          # Calculate the two distances
44          distance_one = abs(x_cdf_emperical - x_cdf_true)
45          distance_two = abs(x_cdf_emperical_previous - x_cdf_true)
46
47
48          # Find the maximum of those two distances and
49          # check if it is larger than the current know maximum distance.
50          max_dist = max(max_dist, max(distance_one, distance_two))
51
52          # Save the current value of the emperical cdf.
53          x_cdf_emperical_previous = x_cdf_emperical
54
55      # Calculate the p-value with the help of the CDF.
56      sqrt_elemens = np.sqrt(x_size)
57      cdf = _ks_statistic_cdf((sqrt_elemens + 0.12+0.11/sqrt_elemens)*max_dist)
58      return 1 - cdf
59
60
61  def kstest2(x1, x2, x2_sorted = None):
62      """
63          Perform the Kolmogorov-Smirnov test for goodness of fit
64          and return the p-value.
65      In:
66          param: x1   -- An array with value's who's CDF is expected to be
67                         the same as the CDF of the proviced values.
68                         Must be atleast size 4.
69
70          param: x2 -- A discretized pdf of the expected distribution under the
        null hypothesis.
71      Out:
72          return: The p-value obtained by performing the KS-test
73      """
74
75      # Amount of values in the input distributions.
76      x1_size = len(x1)
77      x2_size = len(x2)
78
79      # Sort both arrays.
80      x1 = sorting.merge_sort(x1)
81      x2 = sorting.merge_sort(x2) if type(x2_sorted) is not None else x2_sorted
82
83      # The maximum distance
84      max_dist = 0
85
86      # The iteration values used to determine
87      # the emperical pdf's and the max distance.
88      x1_i, x2_j = 0,0
89
90      # Find the maximum distance by updating the emperical CDFs.
91      while x1_i < x1_size and x2_j < x2_size:
92
93          # Update the indices used for the emperical CDF's.
```

```
94
95          if x1[x1_i] < x2[x2_j]:
96              x1_i += 1
97          else:
98              x2_j += 1
99
100         # Find the max distance
101         max_dist = max(abs(x1_i/x1_size-x2_j/x2_size), max_dist)
102
103     sqrt_factor = np.sqrt((x1_size*x2_size)/(x1_size+x2_size))
104     cdf = _ks_statistic_cdf((sqrt_factor + 0.12+0.11/sqrt_factor)*max_dist)
105
106     return 1 - cdf
107
108 def kuiper_test(x, cdf):
109     """
110         Perform the Kuiper test for goodness of fit
111         and return the p-value.
112     In:
113         param: x   -- An array with value's who's CDF is expected to be
114                       the same as the provided CDF. Must be atleast size 4
115
116         param: cdf -- A function that is the expected cdf under the null
    hypothesis.
117     Out:
118         return: The p-value obtained by performing the kuiper-test
119     """
120
121     # Sort the data in ascending order, calculate the
122     # cdf and the emperical cdf for the sorted values and
123     # save the total amount of  elements we have.
124     x_sorted = sorting.merge_sort(x)
125     x_sorted_cdf = cdf(x_sorted)
126     x_elements = len(x)
127
128     # Find the maximum distance above and below
129     # the true cdf.
130     max_dist_above = 0
131     max_dist_below = 0
132
133     # Value of the cdf at step i-1.
134     x_cdf_emperical_previous = 0
135
136
137     for idx, x in enumerate(x_sorted):
138
139         # Calculate the emperical cdf.
140         x_cdf_emperical = (idx+1)/x_elements
141         # Calculate the true cdf.
142         x_cdf_true = x_sorted_cdf[idx]
143
144         # Find the maximum distance above and below
145         max_dist_above = max(x_cdf_emperical - x_cdf_true, max_dist_above)
146         max_dist_below = max(x_cdf_true - x_cdf_emperical_previous,
    max_dist_below)
147
148         # Update previous cdf
149         x_cdf_emperical_previous = x_cdf_emperical
150
151     sqrt_elem = np.sqrt(x_elements)
152     v = max_dist_above + max_dist_below
153     cdf = _kuiper_statistic_cdf((sqrt_elem + 0.155+0.24/sqrt_elem)*v)
154
155     return 1 - cdf
156
157
158 def _ks_statistic_cdf(z):
159     """
160         An approximation for the cdf of the
161         Kolmogorov-Smirnov (KS) test staistic.
162     In:
```

```
163         param: z -- The value to calculate the cdf at.
164     Out:
165         return: An approximation of the cdf for the given value.    print(
        max_dist_above + max_dist_below)
166     """
167
168     # Numerical approximation taken from:
169     # Numerical methods - The art of scientific computation.
170     # Third edition.
171
172     if z < 1.18:
173         exponent = np.exp(-np.pi**2/(8*z**2))
174         pre_factor = np.sqrt(2*np.pi)/z
175
176         return pre_factor*exponent*(1+ exponent**8)*(1+exponent**16)
177     else:
178         exponent = np.exp(-2*z**2)
179         return 1-2*exponent*(1-exponent**3)*(1-exponent**6)
180
181 def _kuiper_statistic_cdf(z):
182     """
183         An approximation for the cdf of the
184         Kuiper test statistic
185     In:
186         param: z -- The value to calculate the cdf at.
187     Out:
188         return: An approximation of the cdf for the given value.
189     """
190
191     # Value of z is to small, sum will be 1 up to 7 digits
192     if z < 0.4:
193         return 1
194
195     # Approximateed value of the sum by performing 100 iterations
196
197     # The value to return
198     ret = 0
199     # A term often needed in the sum.
200     z_squared = z**2
201
202     # Evaluate the first 100 terms in the sum.
203     for j in range(1, 100):
204         power = j**2 * z_squared
205         ret += (4 * power -1)*np.exp(-2*power)
206
207     return 1- 2*ret
208
209
210
211 def normal_cdf(x, mean = 0, sigma = 1):
212     """
213         Evaluate the cummulative normal distribution for
214         the given parameters
215     In:
216         param: x -- The point to evaluate the cdf at or an array of points to
        evaluate it for.
217         param: mean -- The mean of the normal distribution.
218         param: sigma -- The square root of the variance for the normal
        distribution.
219     Out:
220         return: The cummulative normal distribution evaluated at.
221     """
222
223     # Calculate the CDF using the erf function (defined below).
224     return 0.5 + 0.5*erf((x-mean)/(np.sqrt(2)*sigma))
225
226
227 def normal(x, mean = 0, sigma = 1):
228     """
229         Evaluate the normal distrbution for the given
230         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

Code - 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)
```

```
15
16      # If the length is 1 then  return the input array.
17      # This is an important check as this function is called
18      # recursively.
19      if size == 1:
20          return array
21
22      # Split the array in an sorted left and right segment.
23      left_sorted = merge_sort(array[:size >> 1])
24      right_sorted = merge_sort(array[size >> 1:])
25      left_sorted_len = len(left_sorted)
26      right_sorted_len = len(right_sorted)
27
28      #
29      # Merge the left and right array.
30      #
31
32      # The final sorted array.
33      result = np.zeros(size)
34
35      # Current index in the left sorted array.
36      left_idx = 0
37      # Current index in the right sorted array.
38      right_idx = 0
39      # Current index in the final sorted array.
40      result_idx = 0
41
42      # While we didn't fill the result array.
43      while result_idx < size:
44
45          # Element from left array is smaller, insert it and increase position.
46          if left_sorted[left_idx] < right_sorted[right_idx]:
47              result[result_idx] = left_sorted[left_idx]
48              left_idx += 1
49              result_idx += 1
50          # Element from right array is smaller, insert it and increase position.
51          else:
52              result[result_idx] = right_sorted[right_idx]
53              right_idx += 1
54              result_idx += 1
55
56          # Only right arrat has elements left, insert the remaining elements.
57          if left_idx == left_sorted_len:
58              result[result_idx:] = right_sorted[right_idx:]
59              break
60
61          # Only left has elements left, insert the remaining elements
62          if right_idx == right_sorted_len:
63              result[result_idx:] = left_sorted[left_idx:]
64              break
65
66      return result
```

./Code/mathlib/sorting.py

2 - Normally distributed pseudo-random numbers

Question 2

Problem

Make plots of three Gaussian random fields, using $n = -1$, $n = -2$ and $n = -3$. Give the plots a size of 1024. The axis should be in physical size. Choose a minimum physical size and explain how this impacts the maximum physical size, the minimum k and maximum k.

Solution

The Gaussian fields are initial created in k-space by using Fourier shifted coordinates (i.e the zeroth wavenumber corresponds with the left top and not the center) and are then inverse Fourier transformed. The method in which a field is created in k-space consists of two steps. One, the shifted wavenumbers are used to create a matrix with complex numbers based on the power spectrum. Two, the matrix is given the correct hermitian symmetry. The first step is briefly explained in the code. The second step, how the matrix is given the correct symmetry, is described below.

The matrix has the correct symmetry if the complex number created with wavenumbers k_{x_i}, k_{y_j} is equal to the conjugate of the complex number created with wavenumbers $-k_{x_i}, -k_{y_j}$. Let N be the size of the $N \times N$ matrix created in step one and let $c_{i,j}$ be the value in cell i, j. The matrix then has the correct hermitian symmetry if the following three points hold,

A **First row**: The value of matrix cell $c_{0,j}$ should be equal to the complex conjugate of the value in cell $c_{0,N-j}$, for $0 < j < N$. If N is even then the value in cell $c_{0,N/2}$ should be equal to its own conjugate (i.e this value should only have a real component, because it is created with k_0 and $k_{nyquist}$).

B **First column**: This point is similar to the first point. The value of matrix cell $c_{i,0}$ should be equal to the complex conjugate of the value in the cell $c_{N-i,0}$, for $0 < i < N$. If N is even then the value in cell $c_{N/2,0}$ should be equal to its own conjugate (i.e this value should only have a real component, because it is created with $k_{nyquist}$ and k_0).

C **Inner matrix**: The value in cell $c_{i,j}$ should be equal to the complex conjugate in cell $c_{N-i,N-j}$ with $1 \leq i, j < N$. If the matrix is even then cell $c_{N/2,N/2}$ must be real, as it is created with $k_{nyquist}, k_{nyquist}$.

The code that uses these properties to give the matrix the correct symmetry can be found on the next page.

The minimum physical size, the size of 1 cell, is chosen to be 1 Mpc. This immediately fixes the maximum physical size as the grid must be 1024×1024. The maximum physical size is thus 1 Mpc $\times$ 1024 = 1024 Mpc. The minimum k and maximum k are fixed by the minimum and maximum size,

$$k_{min} = \frac{2\pi}{(N \times \text{min distance})} = \frac{2\pi}{\text{max distance}} \tag{16}$$

$$k_{max} = k_{nyquist} = \frac{2\pi N}{2 \times (N \times \text{ min distance})} = \frac{\pi}{\text{min distance}} \tag{17}$$

An increase in minimum distance would thus result in a smaller value for k_{min} and a smaller value of k_{max}.

The code that creates the fields and gives them the correct symmetry, can as mentioned before, be found below. **Important** to point out is that the same random uniform variables have been Box-Muller transformed to create the fields for the different powers, $n = -1$, $n = -2$, $n = -3$. This was done to save computational time and has as consequence that the three created plots show the evolution of the fluctuations as function of the power n (i.e the second plot and third plot show the same kind of structure, only with a different intensity).

Code

The code is split over two files. The first file contains the code for the plots and the second file contains the code for the helper functions.

Code - Plots

The code for the creation of the plots.

```
import numpy as np
import matplotlib.pyplot as plt
import mathlib.random as rnd
import mathlib.misc as misc

# Constants.
grid_size = 1024
min_distance = 1 # size of a single cell in Mpc.

# Create the random number generator.
random = rnd.Random(78379522)

# The orders of the power spectrum.
powers = [-1,-2,-3]

def main():

    # Generate the random uniform numbers that
    # are later transformed to normal distributed variablaes.
    # The numbers are generated once to reduce computational time.
    random_numbers = random.gen_uniforms(grid_size*grid_size*2)

    # Create the plots for n = -1, n = -2, n = -3
    for power in powers:

        # Generate the field matrix.
        matrix = misc.generate_matrix_2D(grid_size, min_distance,
                                         gen_complex, random_numbers, power)

        #Give it the correct symmetry.
        field = misc.make_hermitian2D(matrix)

        # Plot it

        # The field is real, but it is still treated as a complex
        # value this, we have to take the r eal part. It is also multiplied
        # by grid_size^2 to correct for the normalization constant
        # in np.fft.ifft2.

        plt.imshow(np.fft.ifft2(field).real * grid_size*grid_size)
        plt.xlabel('Distance [Mpc]')
        plt.ylabel('Distance [Mpc]')
        plt.title('n = {0}'.format(power))
        plt.colorbar()
        plt.savefig('./Plots/2_field_{0}.pdf'.format(power))
        plt.figure()

def gen_complex(k, n, rand1, rand2):
    """
        Generate a complex number using the power
        spectrum.
    In:
        param:k -- The magnitude of the wavenumber.
        param:n -- The order of the power law.
        param: rand1 -- A random uniform variable between 0 and 1.
        param: rand2 -- A random uniform variables between 0 and 1.

    """

    sigma = 0

    if n == -2:
```

```
        sigma = 1/k
    else:
        sigma = np.sqrt(k**n)

    # Determine the complex value
    a,b = random.gen_normal_uniform(0,sigma,rand1,rand2)
    return complex(a,b)



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

./Code/assigment_2.py

Code - Symmetry

The code containing the functions to create the matrix and give it the correct symmetry. Only part of this file is shown, the full file is shown in assignment 4.

```
import numpy as np

def gen_wavenumbers(size, min_distance):
    """
        Generate the shifted wavenumbers
        for the discrete fourier transform.
    In:
        param: size -- The size of the matrix.
        param: min_distance -- The distance of a cell/ the sample spacing.
    Out:
        return: An array with shifted wave numbers.
    """
    # Array to return.
    ret = np.zeros(size)

    # Positive values
    ret[0:int(size/2)+1] = np.arange(0,int(size/2)+1)

    if size  % 2 == 0: # even
        ret[int(size/2):] = -np.arange(int(size/2),0,-1)
    else: # odd
        ret[int(size/2)+1:] = -np.arange(int(size/2),0,-1)


    return (ret/(size*min_distance))*2*np.pi


def generate_matrix_2D(size, min_distance, func, random_numbers, power):
    """
        Generate a 2D matrix with complex numbers in
        shifted fourier coordinates using the power spectrum
    In:
        param: size -- The size of the matrix (size x size).
        param: min_distance -- The physical size of 1 cell.
        param: func -- A functiion that takes the power and to random uniform
                       variables to calculate the correct complex number.
        param: random_numbers -- An array with random uniform numbers.
                                 Must be of atleast size: size x size x 2.
        param: power -- The power of the power spectrum to create the matrix for.
    Out:
        return: A 2D matrix with complex numbers assigned by the power spectrum
                in fourier shifted coordinates.
    """

    # Generate the shifted wavenumbers
    wavenumber = gen_wavenumbers(size,min_distance)
    # The matrix to return
    ret = np.zeros((size,size),dtype=complex)

    # A counter for the random uniform variables.
    steps = 0
```

```
52
53      # Fill the matrix
54      for i in range(size):
55          for j in range(size):
56
57              # Element of k_0,k_0 is left zero.
58              if i == 0 and j == 0:
59                  continue
60
61              # Calculate the magnitude of the wavenumbers.
62              k = np.sqrt(wavenumber[i]**2 + wavenumber[j]**2)
63              # Fill the matrix.
64              ret[i][j] = func(k, power,
65                               random_numbers[steps], random_numbers[steps+1])
66              steps += 2
67
68      # Return the matrix
69      return ret
70
71
72  def make_hermitian2D(matrix):
73      """
74          Give a matrix in shifted fourier coordinates
75          the correct hermitian symmetry so that the ifft is real.
76      In:
77          param: matrix -- The matrix to give the correct symmetry.
78      Out:
79          return: A matrix with the correct hermitan symmetry so that the
80                  ifft is real.
81      """
82
83      # The size of the matrix
84      size = matrix.shape[0]
85
86      # Loop over the rows
87      for row in range(1, int(size/2) +1):
88
89          # Give the first column (index 0) has the correct symmetry (see report
        point A)
90          matrix[row,0] = complex(matrix[size-row,0].real,
91                                  - matrix[size-row,0].imag)
92          # Give the first row (index 0) the correct symmetry (see report point B)
93          matrix[0, row] = complex(matrix[0, size-row].real,
94                                   - matrix[0,size- row].imag)
95
96          # Give the inner matrix the correct symmetry (see report point C)
97          for column in range(1, size):
98              matrix[row, column] = complex(matrix[size-row, size-column].real,
99                                            -matrix[size-row, size-column].imag)
100
101     # Corrections for even matrix
102     if size % 2 == 0:
103         matrix[int(size/2), 0] = matrix[int(size/2), 0].real + 0J
104         matrix[0, int(size/2)] = matrix[0, int(size/2)].real + 0J
105         matrix[int(size/2), int(size/2)] = matrix[int(size/2), int(size/2)].real
        + 0J
106
107     # Return the matrix.
108     return matrix
```

./Code/mathlib/misc.py

Plots - Fields

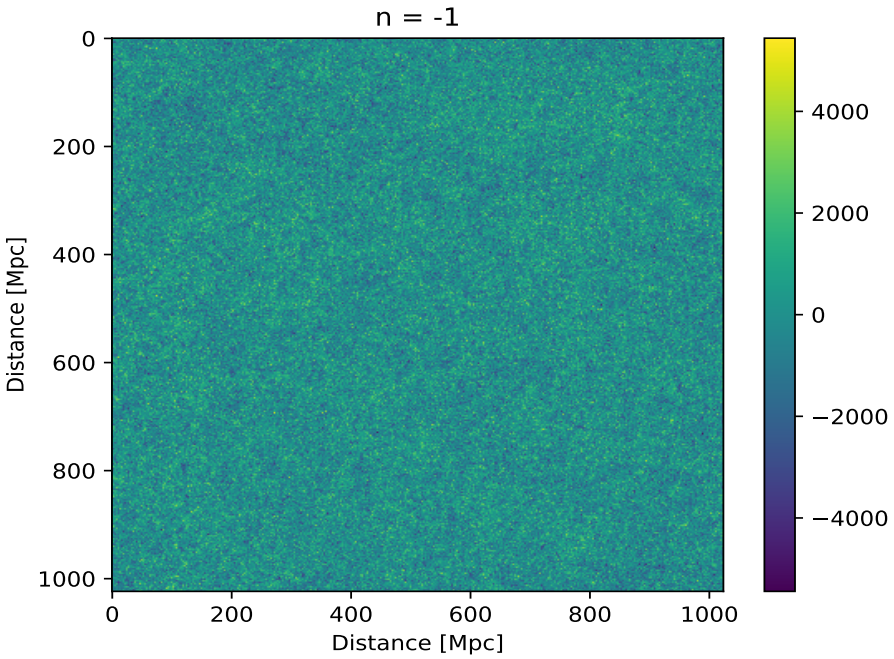

Figure 19: The Gaussian field for $n = -1$ and a minimal physical size of 1 Mpc. The power drops for $n = -1$ slowly, as result, mainly noise is expected. The plot seems to show that this is the case.

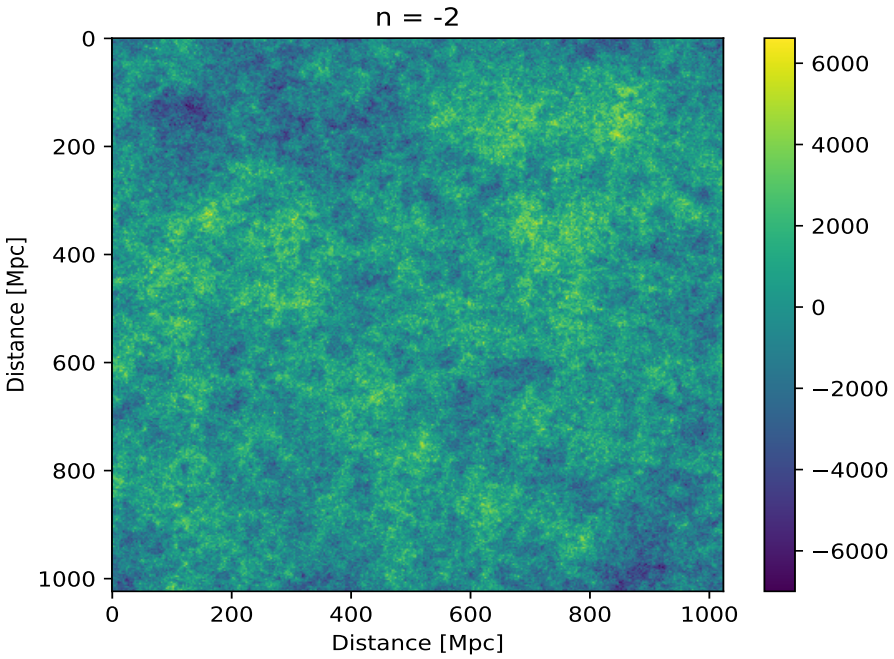

Figure 20: The Gaussian field for $n = -2$ and a minimal physical size of 1 Mpc. The plot seems to be less noisy than the previous plot, which is expected as n is now smaller.

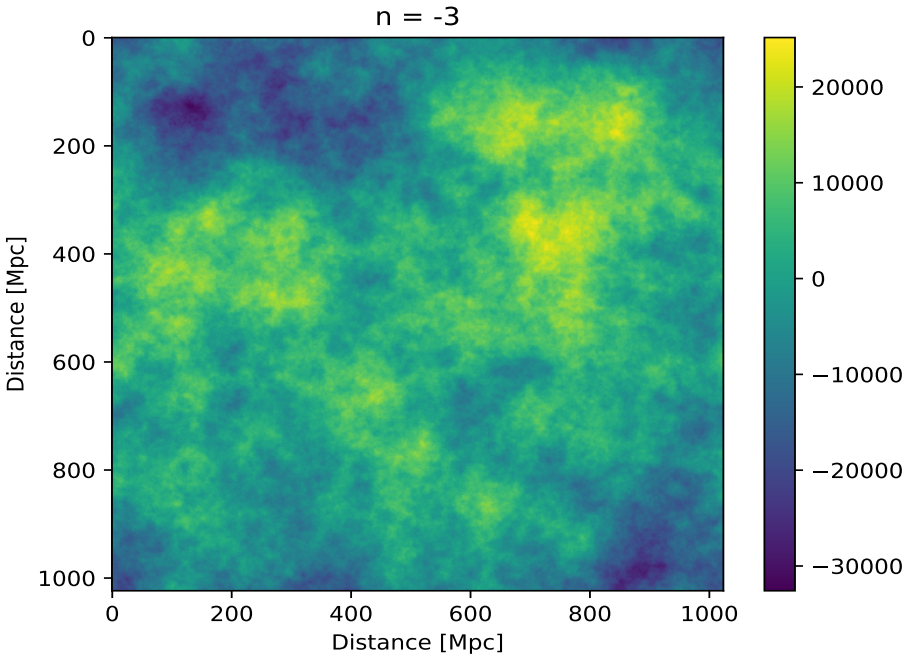

Figure 21: The Gaussian field for $n = -3$ and a minimal physical size of 1 Mpc. High and low density fluctuations are now clearly visible. Notice that the plot looks similar to the plot for $n = -2$. This is a consequence of using the same random uniform variables in the Box-Muller method.

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:

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

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 derivation of the analytical solution. Three, a (brief) explanation on how this rewritten version is used numerically.

(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 $\Omega_0 = 1$ results in the rewritten version of equation 18,

$$\frac{d^2D}{dt^2} + \frac{H_0\left(\frac{3}{2}H_0t\right)^{-1/3}}{\left(\frac{3}{2}H_0t\right)^{2/3}}\frac{dD}{dt} - \frac{3}{2}\Omega_0\frac{H_0^2}{\left(\frac{3}{2}H_0t\right)^{2/3}}D = 0 \tag{20}$$

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

(2) 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,

$$\lambda\left(\lambda - 1\right)t^{\lambda-2} + \frac{4}{3t}\lambda t^{\lambda-1} - \frac{2}{3t^2}t^\lambda = 0 \tag{22}$$

This simplifies to

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

From the above expression it can be seem that peculiar solutions of the ODE are given by,

$$D(t) = t^{-1} \qquad\qquad D(t) = t^{2/3} \tag{23}$$

The general solution is the superposition of the peculiar solutions with constants and can therefore be written as,

$$D(t) = c_1t^{2/3} + c_2t^{-1} \tag{24}$$

The constants for the three initial cases can be found by calculating the derivative of the above equation and solving the system for the derivative and the non derivative. This yields for the three cases that,

$$\text{(A) } c_1 = 3, c_2 = 0 \qquad \text{(B) } c_1 = 0, c_2 = 10 \qquad \text{(C) } c_1 = 3, c_2 = 2 \tag{25}$$

(3) 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 Runge-kutta method. The second order ODE can be written as a system of first order ODE's by substituting $dD/dt = u$. The system then becomes,

$$\begin{cases} \frac{dD}{dt} & = u \\ \frac{d^2D}{d^2} & = -\frac{4}{3t}u + \frac{2}{3t^2}D \end{cases} \tag{26}$$

The above system is as mentioned before solved with the Dormand Prince version of the Runge-Kutta method. The algorithm uses an adaptive step size that is initial set to $t_{step} = 0.01$ year for all cases. 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 Dormand Prince version of the Runge-Kutta method. The code and its output can be found below.

Code - Plots

The code that created the plots for the three given initial conditions of the ODE.

```
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

Code - Runge-Kutta

The code for the Dormand Prince version of the Runda kutta method.

```
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:
```

```
49
50          # Check if we need to expand the return arrays
51          if steps >= ret.shape[0]:
52              ret_old = ret.copy()
53              ret = np.zeros((ret_old.shape[0]*2, ret_old.shape[1]))
54              ret[0:steps] = ret_old
55
56              time_old = time.copy()
57              time = np.zeros(time_old.shape[0]*2)
58              time[0:steps] = time_old
59
60
61          # Get the value found at the previous step
62          previous = ret[steps-1]
63
64          # Calculate the constants for the Dormand Prince Runge kutta method.
65
66          # Appoligies if this looks ugly in the report.
67          k1 = t_step*func(previous, t_now)
68          k2 = t_step*func(previous + (1/5)*k1, t_now + (1/5)*t_step)
69          k3 = t_step*func(previous + (3/40)*k1 + (9/40)*k2, t_now + (3/10)*t_step)
70          k4 = t_step*func(previous + (44/45)*k1 - (56/15)*k2 + (32/9)*k3, t_now +
        (4/5)*t_step)
71          k5 = t_step*func(previous + (19372/6561)*k1 - (25360/2187)*k2 +
        (64448/6561)*k3 - (212/729)*k4, t_now + (8/9)*t_step)
72          k6 = t_step*func(previous + (9017/3168)*k1 - (355/33)*k2 + (46732/5247)*
        k3 + (49/176)*k4 - (5103/18656)*k5, t_now + t_step)
73
74          # Calculate the new value.
75          y_next = previous + ( (35/384)*k1 + (500/1113)*k3 + (125/192)*k4 -
        (2187/6784)*k5 + (11/84)*k6)
76          y_embedded = previous + ( (5179/57600)*k1 + (7571/16695)*k3 + (393/640)*
        k4-(92097/339200)*k5 + (187/2100)*k6 )
77
78          # Calculate error
79          delta = abs(y_embedded  - y_next)
80          scale = atol+np.maximum(abs(previous),abs(y_next))*rtol
81          error = np.sqrt(np.sum((delta/scale)**2)/len(k1))
82
83          # The factor used to calculate the new step size.
84          update_scale = 0.9*(error)**(-0.2)
85
86          # Make sure the factor is not to large or to small.
87          if error == 0:
88              update_scale = max_update_scale
89          elif update_scale < min_update_scale:
90              update_scale = min_update_scale
91          elif update_scale > max_update_scale:
92              update_scale = max_update_scale
93
94          # Check if the current step should be accepted.
95          if error > 1: # reject
96              t_step*= min(update_scale, 1.0)
97          else: # accept
98              t_now += t_step
99              t_step *= update_scale
100
101             ret[steps] = y_next
102             time[steps] = t_now
103             steps += 1
104
105
106     return ret[0:steps], time[0:steps]
```

./Code/mathlib/ode.py

Code - Output plot(s)

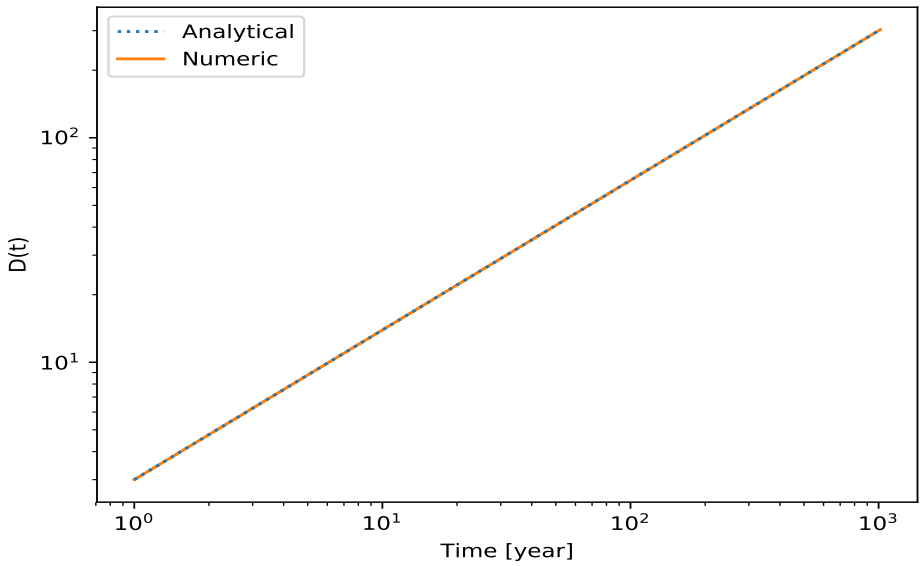

Figure 22: The analytical (blue) and numerical (orange) solution of the ODE with initial conditions $D(1) = 3, D'(1) = -10$. The plots show that he numerical solution does not appear to have a visible deviation from the analytical solution for the given time interval.

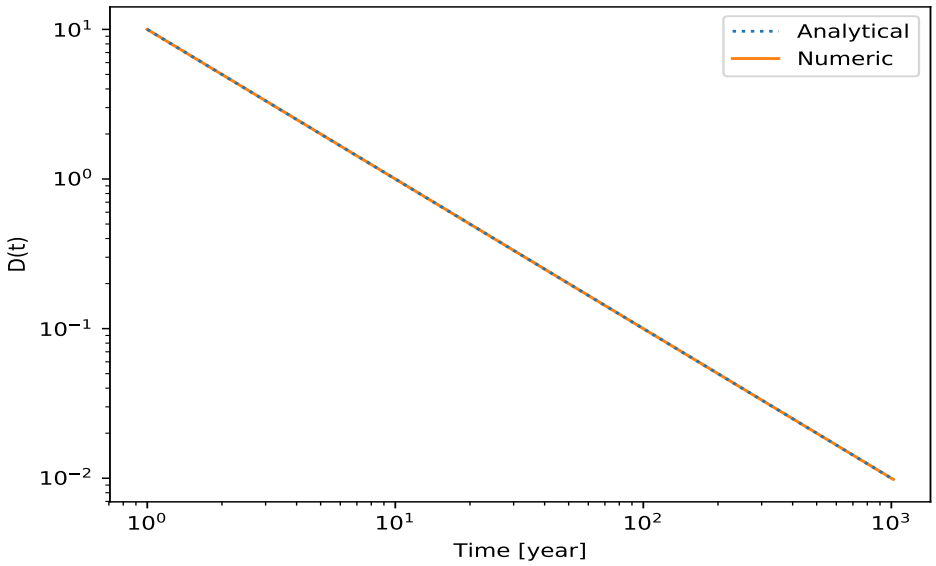

Figure 23: The analytical (blue) and numerical solution (orange) of the ODE with initial conditions $D(1) = 10, D'(1) = -10$. The plots show that he numerical solution does not appear to have a visible deviation from the analytical solution for the given time interval.

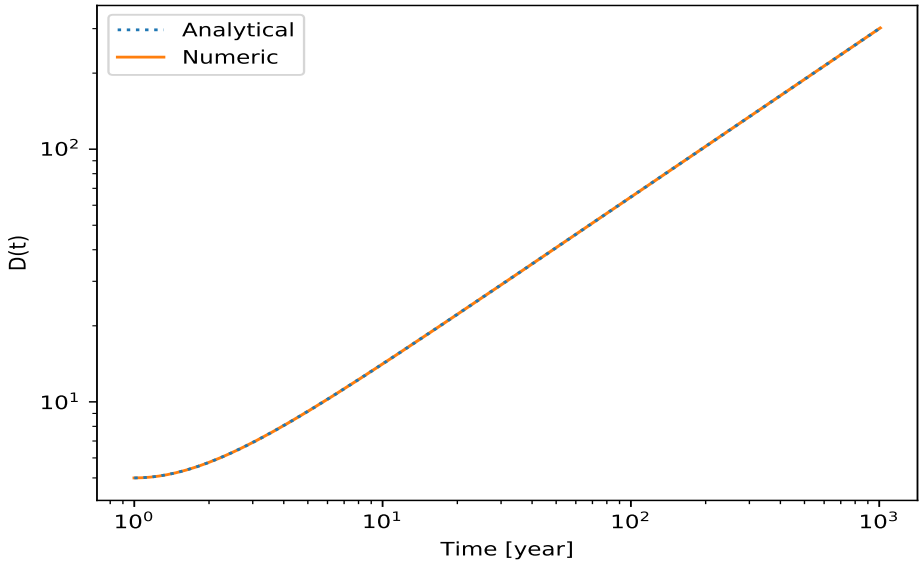

Figure 24: The analytical (blue) and numerical (orange) solution of the ODE with initial conditions $D(1) = 5, D'(1) = 0$. The plots show that he numerical solution does not appear to have a visible deviation from the analytical solution for the given time interval.

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,

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

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

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

Substituting this in equation 30 and simplifying yields,

$$D(a) = \frac{5\Omega_m H_0^3}{2}(\Omega_m a^{-3} + \Omega_\Lambda)^{0.5} \int_0^a \frac{a'^{-3}}{(H_0^2(\Omega_m a^{-3} + \Omega_\Lambda))^{3/2}} da' \tag{32}$$

$$= \frac{5\Omega_m}{2}(\Omega_m a^{-3} + \Omega_\Lambda)^{0.5} \int_0^{a'} \frac{a'^{-3}}{(\Omega_m a^{-3} + \Omega_\lambda)^{3/2}} da' \tag{33}$$

The above integral is with the help of Romberg integration solved for $\Omega_m = 0.3$ and $\Omega_\lambda = 0.7$. The code for Romberg integration can be found at page 57. The code that prints the output and the printed output can be found below. The shared modules used by the code start at page 53.

Code - Output

The code that prints the value of the linear growth factor.The code for the called helper function, `helpers4.calculate_linear_growth` can be found on page 53.

```
def assigment4_a():
    """
        Execute assigment 4a
    """

    # Relevant imports are.
    # (1) import mathlib.helpers4 as helpers4


    # The value of the scale to integrate to
    a_max = 1/51

    # The values of the density parameters
    omega_m = 0.3
    omega_lambda = 0.7

    # Print the result
    print('[4a] D(a = 1/51) = ', helpers4.calculate_linear_growth(a_max,
                                omega_m, omega_lambda))
```

./Code/assigment_4.py

Text - Output

The output produced by the above code.

```
[4a] D(a = 1/51) =  0.019607780428272343
```

./Output/assigment4_out.txt

Question 4.b)

Problem

Use your result from the previous question to analytically calculate for $a = 1/51$,

$$\dot{D(t)} = \frac{dD(a)}{da}\dot{da} \tag{34}$$

Bonus points if you calculate it numerically and match the analytical result within 10^{-8}.

Solution

In this question the derivative is not only calculated analytically, but also numerically. The numerical calculation is performed with ridders method. The analytical solution is derived below and evaluated. The code for the numerical solution and the code that prints the analytical solution[3] can be found after this derivation. This includes the code for ridders method.

To simplify the derivation we define,

$$C = \int_0^{a'} \frac{a'^{-3}}{\left(\Omega_m a'^{-3} + \Omega_\Lambda\right)^{3/2}} da' \tag{35}$$

The derivative can then by substituting 33 in equation 34 be written as,

$$\dot{D}(t) = \frac{5\Omega_m}{2}\left(\frac{-3\Omega_m a^{-4}}{2}\left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{-0.5} C + \left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{0.5} \frac{a^{-3}}{\left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{3/2}}\right)\dot{a} \tag{36}$$

$$= \frac{5\Omega_m}{2}\left(\frac{-3\Omega_m a^{-4}}{2\left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{0.5}} C + \frac{a^{-3}}{\left(\Omega_m a^{-3} + \Omega_\Lambda\right)}\right)\dot{a} \tag{37}$$

Substituting $\dot{a} = H(a)a(t) = H_0\left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{0.5} a(t)$ results in,

$$\dot{D} = \frac{5\Omega_m H_0}{2}\left(\frac{-3\Omega_m a^{-3}}{2} C + \frac{a^{-2}}{\left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{0.5}}\right) \tag{38}$$

$$= \frac{5\Omega_m H_0}{2a^2}\left(\frac{1}{\left(\Omega_m a^{-3} + \Omega_\Lambda\right)^{0.5}} - \frac{3\Omega_m}{2a} C\right) \tag{39}$$

The above expression is evaluated in the code. The value of C is numerically calculated for the analytical expression. This value is just the result of the previous question without scale factor and was allowed to be used for this question. The analytical output (with the numerical calculation for C) and the numerical output (with ridders method) can be found below after the code section.

[3]The analytical solution is printed to make it easier to compare.

Code - Print

The code that prints the analytical and numerical result. The code for ridders method follows after this.

```
def assigment4_b():
    """
        Execute assigment 4b
    """

    # Relevant imports include:
    # (1) import mathlib.derivative as ml_dir
    # (2) import mathlib.helpers4 as helpers4
    # (3) import mathlib.integrate as integrate

    # Start with numerical

    # The linear growth factor as function of a
    D = lambda a: helpers4.calculate_linear_growth(a, 0.3, 0.7)

    # Calculate the derivative with ridder for a = 1/51
    dirivative = ml_dir.ridder(D,1/51,1e-15)

    # Calculate \dot{a} = H(a)a
    H_0 = 7.16e-11
    H = H_0 * (0.3*(1/51)**(-3) + 0.7)**0.5

    # Final result
    final_numerical = dirivative*H*(1/51)

    # Analytical

    # The function to integrate.
    func = lambda a: a**(-3) /(0.3 * (1/(a**3))+0.7)**(3/2)
    C = integrate.romberg(func, 1e-7, (1/51), 15)

    # The right term in brackets
    right = -(3*0.3*C)/(2*(1/51))
    # The left term in brackets
    left = 1/(0.3*(1/51)**(-3)+0.7)**0.5

    # Prefactor
    pre = (5*0.3*H_0)/(2*(1/51)**2)

    # Print the results
    print("[4b] Analytical: ", pre*(left+right))
    print("[4b] Numerically: ", final_numerical)
```

./Code/assigment_4.py

Code - Ridder

The code for ridders method.

```
import numpy as np

def central_diff(function, x, h):
    """
        Use the central difference method to approximate the derivative
        at a point x.
    In:
        param: function -- The function to calculate the derivative of.
        param: x -- The point to approximate the derivative of the function at.
        param: h -- A small value h that in the limit would go to zero.
    Out:
        return: An approximation of the derivative of the provided
                function at x.
    """
```

```
15
16      #f'(x) approx (f(x+h)-f(x-h))/2h
17      return (function(x+h) - function(x-h))/(2*h)
18
19
20  def ridder(function, x , precision):
21      """
22          Perform ridders differential method to estimate
23          the derivative at a point x
24
25      In:
26          param: function -- The function to estimate the derivative for.
27          param: x -- The point to estimate the derivative at.
28          param: precision -- The precision that must be obtained
29                              in the estimation.
30      Out:
31          return: An approximation of the derivative.
32      """
33
34      # The number of initial approximations to use.
35      approximations = 0
36      # The return value.
37      ret = 0
38      # The current precision.
39      current_precision = 0xFFFFFFFFFF
40
41      # While we didn't reach the request precision with
42      # the current amount of initial approximations, try again
43      # but with more approximations.
44      while current_precision > precision:
45          # Increase the amount of initial approximations.
46          approximations += 5
47          # Reset the error.
48          current_precision = 0xFFFFFFFFFF
49
50
51          # The array to store the combined results in.
52          results = np.zeros(approximations)
53
54          # The current combination:
55          # 0 = combine initial central_difference evaluations,
56          # 1 = combine the combined central difference evaluations
57          # 2 = combine the combined combined central difference evaluations etc.
58          for combination in range(0, approximations-1):
59
60              # Combine for the current 'combination'.
61               for j in range(1, approximations-combination):
62
63                  # Create the initial central difference to combine.
64                  if combination == 0:
65                      # We need two central difference's to
66                      # combine the very first time.
67                       if j == 1:
68                           results[j-1] = central_diff(function, x,1)
69
70                      # Decrease h by a factor of 2 for each next approximation.
71                       results[j] = central_diff(function, x, (1/2)**j)
72
73                  # Keep the evaluation of the previous combination that
74                  # is getting overwritten temporary in memory to update
75                  # the current precision.
76                  previous = results[j-1]
77
78                  # Combine
79                  power = 4**(combination+1)
80                  results[j-1] = (power*results[j] - results[j-1])/(power-1)
81
82                  # Determine the new precision
83                  precision_tmp = max(abs(results[j-1] - previous), \
84                                      abs(results[j-1] - results[j]))
85
```

```
                # New precision is smaller -> update
                if precision_tmp < current_precision:
                    current_precision = precision_tmp
                    ret = results[j-1]

                    # Terminate if requested precision is reached
                    if current_precision < precision:
                        return ret

                # Abort early if the error of the last combined result is worse
                # than the previous order by a large amount.
                if j == (approximations-combination-1) \
                    and abs(ret - previous) > 100*current_precision:

                    return ret
    return ret
```

./Code/mathlib/derivative.py

Text - Output

The analytical and numerical result. As can be seen is the numerical approximation within 10^{-8}.

```
[4a] D(a = 1/51) =  0.019607780428272343
[4b] Analytical:  2.800638157126267e-10
[4b] Numerically:  2.8006381571292944e-10
```

./Output/assigment4_out.txt

Question 4.c)

Problem

Use the Zeldovich approximation to generate a movie of the evolution of a volume in two dimensions from a scale factor of 0.0025 until a scale factor of 1.0. To do this, start with 64x64 particles arranged in a square grid with a grid spacing of 1 Mpc. Use $P(k) = k^{-2}$ and the given equation to generate c_k, then use the FFT on your grid of particles to calculate **S(q)**. As a increases, update the positions and momenta of the particles and save each step as a frame for your movie.

Solution

The difficulty of this exercise lays by the calculation the components of the displacement vector. A brief description of how this is done can be found below.

The displacement vector is calculated by first creating a matrix in k -space with complex values based on the given power law (similar to exercise 2). The matrix is next given the correct hermitian symmetry (see again exercise 2). The components of the displacement vector, s_x and s_y, are then calculated independent with the help of this matrix. This calculation is done as follows. The hemitian matrix is copied and the components in the matrices are multiplied with the wavenumbers and the complex number i. The wavenumbers by which the components are multiplied depends on whether we want s_x or s_y. The result is two matrices where the cells have respectively values of ic_kk_x and ic_kk_y. After this multiplication the matrices cannot be directly fourier transformed to obtain s_x and s_y, as the multiplication with the wavenumbers breaks the symmetry in the columns with nyquest wavenumbers. The symmetry is only broken by a minus sign and this is corrected before doing the IFFT.
The code that creates the movie and the plots for the first 10 particles can be found below. The final plot of the movie is also included and shows that the particles clearly move to the denser region (see figure 25 on page 47).

Code - Plots:

The code that creates the movie and the plots of the first 10 particles. The code does make use of an object called 'random' and a function called `gen_ complex`. This object and the function can be found on page 53. The code for the imported shared modules can also be found at page 53. The movie can be found in the movie folder.

```
def assigment4_c():
    """
        Execute assigment 4c
    """

    # The relevant imports include:
    # (1) import numpy as np
    # (2) import scipy.fftpack
    # (3) import mathlib.misc as misc
    # (4) random, defined outside the main function.



    # Constants and random number generator
    grid_size = 64
    min_distance = 1
    power = -2

    # Create the hermitan matrix and the ifft.
    # The ifft is used to plot the backgound
    matrix = misc.generate_matrix_2D(grid_size, min_distance,
                                     gen_complex,
                                     random.gen_uniforms(grid_size**2*2),power)
    matrix = misc.make_hermitian2D(matrix)

    # Calculate the ifft, used for the background of the movie.
    matrix_ifft = scipy.fftpack.ifft2(matrix).real

    # wavenumbers
    wavenumbers = misc.gen_wavenumbers(grid_size, min_distance)

    matrix_y = matrix * wavenumbers*1J
    matrix_x = matrix * wavenumbers[:, np.newaxis]*1J

    # Fix the symmetry that is broken

    size = matrix_x.shape[0]

    matrix_x[int(size/2),0] = matrix_x[int(size/2),0].real
    matrix_x[int(size/2),int(size/2)] = matrix_x[int(size/2),int(size/2)].real

    matrix_y[0,int(size/2)] = matrix_y[0,int(size/2)].real
    matrix_y[int(size/2),int(size/2)] = matrix_y[int(size/2),int(size/2)].real


    for i in range(int(size/2)+1, size):
        matrix_x[int(size/2),i] *= -1
        matrix_y[i,int(size/2)] *= -1

    # Generate the componentes of the displacemnet vectors
    # We have to transpose/ switch to get the x and y.
    # Multiplication with grid_size is to correct for the
    # normalization factor of the scipy implementation of the ifft.
    s_x = scipy.fftpack.ifft2(matrix_y).real *grid_size
    s_y = scipy.fftpack.ifft2(matrix_x).real *grid_size

    # Positons of particles
    pos_x, pos_y = np.meshgrid(range(0,grid_size),range(0,grid_size))

    # Begin simulation

    # Constants for integration
    a_min = 0.0025
    scale_factors = np.linspace(a_min, 1, 90)
```

```
65
66      # List in which the momentum and position
67      # of the first 10 particles is saved.
68      pos_top_10 = list()
69      momentum_top_10 = list()
70
71      # Create the plot
72      for idx,a in enumerate(scale_factors):
73
74          # Calculate da
75          da = 0
76
77          if idx != 0:
78              da = scale_factors[idx] - scale_factors[idx-1]
79          else:
80              da = scale_factors[idx] - a_min
81
82          # Calculate D(a) and \dot(D)
83          d = helpers4.calculate_linear_growth(a)
84          ddot = helpers4.calculate_linear_growth_dir(a -da/2)
85
86          # Update position.
87          pos_x_new = (pos_x+ s_x*d) % grid_size
88          pos_y_new = (pos_y+ s_y*d) % grid_size
89
90          pos_top_10.append(pos_y_new[0:10,0])
91
92          # Update momentum.
93          p_x = -(a-da/2)**2 * ddot *s_x
94          p_y = -(a-da/2)**2 * ddot*s_y
95
96          momentum_top_10.append(p_y[0:10,0])
97
98          # Create the plot.
99          plt.scatter(pos_x_new, pos_y_new,s=1,c='black')
100         plt.imshow(matrix_ifft*grid_size,alpha=0.7)
101
102         plt.colorbar()
103         plt.title('a=' + str(a))
104         plt.xlabel('x [Mpc]')
105         plt.ylabel('y [Mpc]')
106         plt.savefig('./Plots/4c/4c={0}.png'.format(idx))
107         plt.close()
108
109     # Plot the momentum and position
110     pos_top_10 = np.array(pos_top_10)
111     momentum_top_10 = np.array(momentum_top_10)
112
113     # Momentum
114     for i in range(0,10):
115         plt.plot(scale_factors, momentum_top_10[:,i],label='particle ' + str(i+1)
    )
116         plt.xlabel('a')
117         plt.ylabel(r'$P_y$')
118     plt.legend()
119     plt.savefig('./Plots/4c_momentum.pdf')
120     plt.close()
121
122     # Position
123     for i in range(0,10):
124         plt.plot(scale_factors, pos_top_10[:,i],label='particle ' + str(i+1))
125         plt.xlabel('a')
126         plt.ylabel(r'y')
127
128     plt.legend()
129     plt.savefig('./Plots/4c_pos.pdf')
130     plt.close()
```

./Code/assigment_4.py

Plots - Field

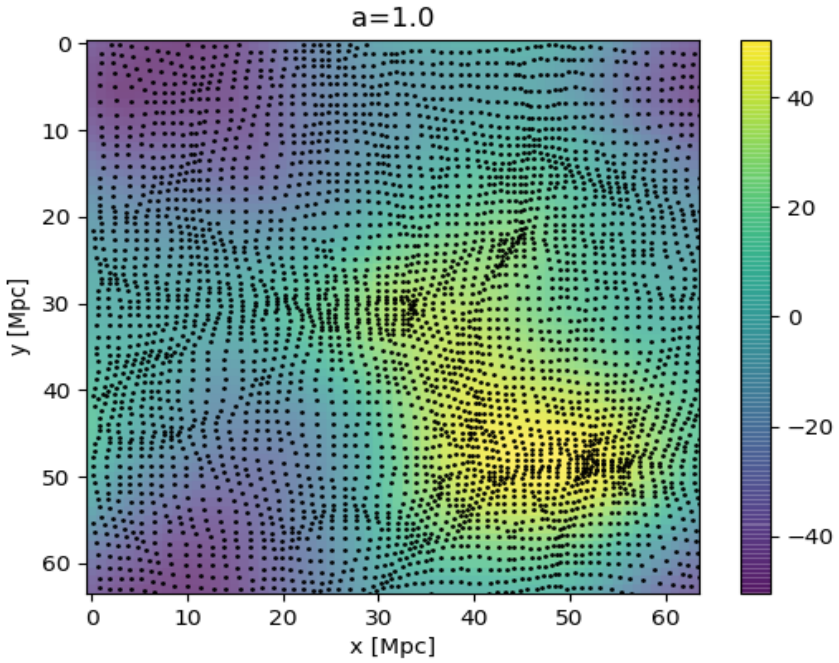

Figure 25: The final plot of the movie. It can clearly be seen that the particles (black dots) moved to the denser region.

Plots - particles

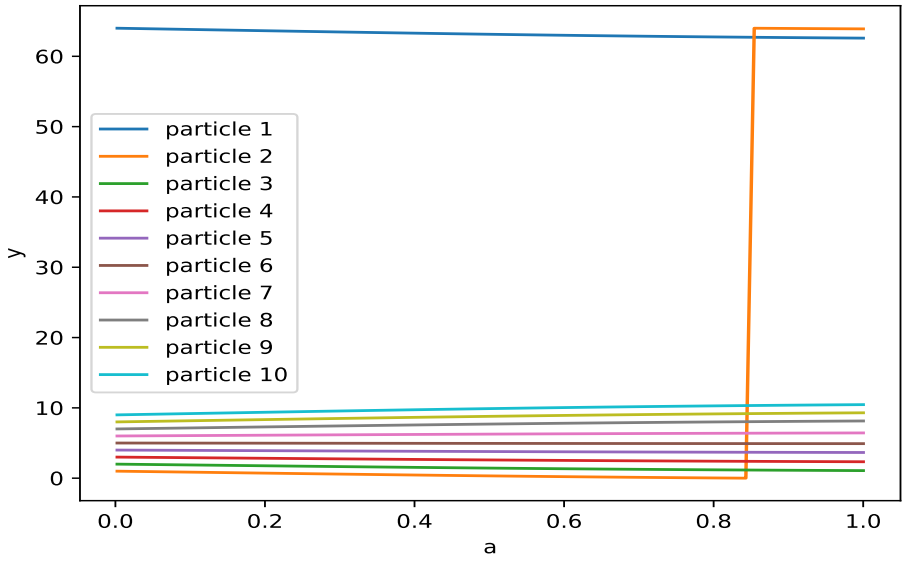

Figure 26: The y-positions of the first 10 particles as function of the scale factor. The plot shows that the particles are slowly moving to a denser region (see figure 25, where the first 10 particles are at the left top). The large jump for particle 2 (orange) at a scale factor of around $a \approx 0.8$ is the result of the circular boundary conditions.

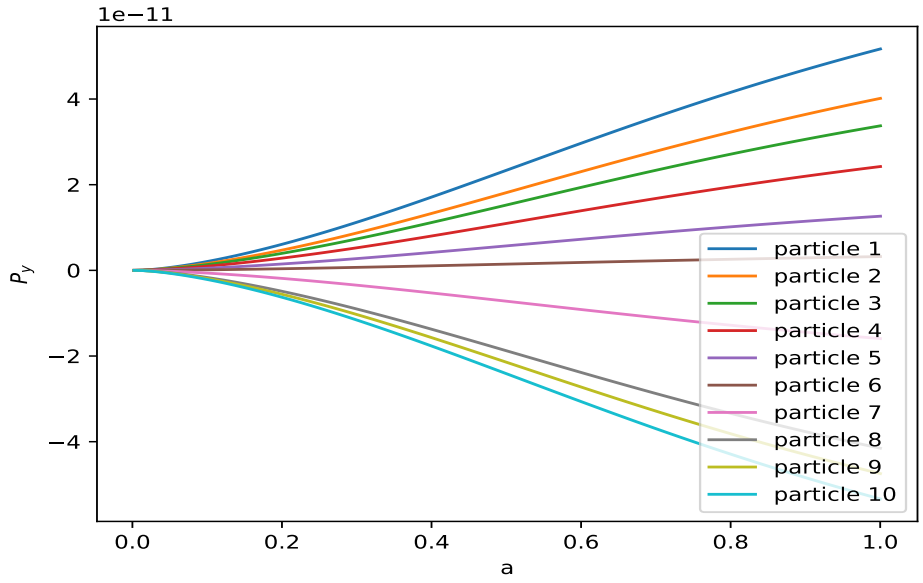

Figure 27: The momentum of the first 10 particles as function of the scale factor. The fact that about halve of the particles have a positive momentum and that the other halve have a negative momentum is the result of the circular boundary conditions and the created field.

Question 4.d)

Problem

Generate initial conditions for a three dimensional box to do a N-body simulation, make initial conditions for 64^3 particles starting at redshift $z = 50$. Besides this make 3 separate movies of a slice of thickness 1/64th of your box at its center, make a slice for $x - y$, $x - z$,$y - z$. Again make a movie of at least 3 seconds with at least 30 frames per second. Finally plot the position and momentum of the first 10 particles along the z-direction vs a.

Solution

The method in which the displacement vector is similar to the method in 4c. This thus means that first a 3D matrix (tensor) is created in k-space with complex values based on the given power law. The matrix is next given the correct hermitian symmetry, which is done by an extended version of the algorithm explained in question 2. The symmetric matrix (tensor) is now used to calculate the components s_x, s_y and s_z. This is identical to 4c: The matrix is first copied and the copies multiplied with the correct wavenumbers and the complex value i. In the end this results in three matrices that need to be inverse Fourier transformed to obtain the components of $\mathbf{S}$. The three matrices can again not be directly inverse Fourier transformed as the multiplication with the wavenumbers breaks the symmetry in the nyquest planes. The symmetry is again only broken by a minus sign and is first corrected before doing the IFFT. The corrected matrices are then used to calculate s_x, s_y and s_z.

The code that uses the displacement vector to creates the 3 simulations and the plots for the first 10 particles can found below. The code make use of an object called 'random' and a function called `gen_ complex`. This object and the function can be found on page 53. The code for the imported modules can also be found at page 53. The movie can be found in the movie folder.

Code - Plots:

The code that creates the movie and the plots of the first 10 particles in 3D.

```
def assigment4_d():
    """
        Execute assigment 4d
    """

    # The relevant imports include:
    # (1) import numpy as np
    # (2) import scipy.fftpack
    # (3) import mathlib.misc as misc
    # (4) random, defined outside the main function.

    # Create the matrix
    grid_size = 64
    min_size = 1
    power = -2

    # Create the matrix and give it the correct symmetry
    matrix = misc.generate_matrix_3D(grid_size, min_size,
                                     gen_complex,
                                    random.gen_uniforms(grid_size**3 *2),
                                    power)
    matrix = misc.make_hermitian3D(matrix)

    # Wavenumbers and matrices for positions
    wavenumbers = misc.gen_wavenumbers(grid_size,1)

    kx,ky,kz = np.meshgrid(wavenumbers,wavenumbers,wavenumbers)
    matrix_x = matrix*kx*1J
    matrix_y = matrix*ky*1J
    matrix_z = matrix*kz*1J

    # Fix the broken symmetry

    # Fix matrix_x
    for i in range(matrix.shape[0]):
        for j in range(int(grid_size/2)+1, grid_size):
            matrix_x[i,int(grid_size/2),j] *= -1

        if i > int(grid_size/2):
            matrix_x[i,int(grid_size/2),int(grid_size/2)] *=-1
            matrix_x[i,int(grid_size/2),0] *= -1

    # Special points
    matrix_x[int(grid_size/2), 0, 0] = matrix_x[int(grid_size/2), 0, 0].imag
    matrix_x[0,int(grid_size/2),0] = matrix_x[0,int(grid_size/2),0].imag
    matrix_x[0,0,int(grid_size/2)] = matrix_x[0,0,int(grid_size/2)].imag

    matrix_x[int(grid_size/2), int(grid_size/2),0] = matrix_x[int(grid_size/2),
    int(grid_size/2),0].imag
    matrix_x[int(grid_size/2), 0, int(grid_size/2)] = matrix_x[int(grid_size/2)
    ,0, int(grid_size/2)].imag
    matrix_x[0, int(grid_size/2), int(grid_size/2)] = matrix_x[0, int(grid_size
    /2), int(grid_size/2)].imag
    matrix_x[int(grid_size/2), int(grid_size/2), int(grid_size/2)] = matrix_x[int
    (grid_size/2), int(grid_size/2), int(grid_size/2)].imag

    # Fix matrix_z
    for i in range(matrix.shape[0]):
        for j in range(int(grid_size/2)+1, grid_size):
            matrix_z[i,j,int(grid_size/2)] *= -1

        if i > int(grid_size/2):
            matrix_z[i,int(grid_size/2),int(grid_size/2)] *=-1
            matrix_z[i,0,int(grid_size/2)] *= -1

    # special points
    matrix_z[int(grid_size/2), 0, 0] = matrix_z[int(grid_size/2), 0, 0].imag
    matrix_z[0,int(grid_size/2),0] = matrix_z[0,int(grid_size/2),0].imag
```

```
    matrix_z[0,0,int(grid_size/2)] = matrix_z[0,0,int(grid_size/2)].imag

    matrix_z[int(grid_size/2), int(grid_size/2),0] = matrix_z[int(grid_size/2),
    int(grid_size/2),0].imag
    matrix_z[int(grid_size/2), 0, int(grid_size/2)] = matrix_z[int(grid_size/2)
    ,0, int(grid_size/2)].imag
    matrix_z[0, int(grid_size/2), int(grid_size/2)] = matrix_z[0, int(grid_size
    /2), int(grid_size/2)].imag
    matrix_z[int(grid_size/2), int(grid_size/2), int(grid_size/2)] = matrix_z[int
    (grid_size/2), int(grid_size/2), int(grid_size/2)].imag


    # Fix matrix y
    for j in range(grid_size):
        for k in range(grid_size):

            if (j == 0 or j == int(grid_size/2)) and k > int(grid_size/2):
                matrix_y[int(grid_size/2),j,k] *= -1
            elif  j > int(grid_size/2):
                matrix_y[int(grid_size/2),j,k] *= -1

    # special points
    matrix_y[int(grid_size/2),int(grid_size/2),int(grid_size/2)] = matrix_y[int(
    grid_size/2),int(grid_size/2),int(grid_size/2)].imag + 0J
    matrix_y[int(grid_size/2),int(grid_size/2),0] = matrix_y[int(grid_size/2),int
    (grid_size/2),0].imag + 0J
    matrix_y[int(grid_size/2),0,int(grid_size/2)] = matrix_y[int(grid_size/2),0,
    int(grid_size/2)].imag + 0J
    matrix_y[int(grid_size/2),0,0] = matrix_y[int(grid_size/2),0,0].imag + 0J

    # Generate the componentes of the displacemnet vectors
    s_y = scipy.fftpack.ifftn(matrix_y).real * grid_size**(3/2)
    s_x = scipy.fftpack.ifftn(matrix_x).real * grid_size**(3/2)
    s_z = scipy.fftpack.ifftn(matrix_z).real * grid_size**(3/2)


    # Positions
    pos_x, pos_y, pos_z = np.meshgrid(range(0,grid_size),range(0,grid_size),
    range(0, grid_size))

    #
    # Begin simulation

    # Constants for integration
    a_min = 1/51
    scale_factors = np.linspace(a_min,1,90,endpoint=False)

    # List in which the momentum and position
    # of the first 10 particles is saved.
    pos_top_10 = list()#.zeros((10,10))
    momentum_top_10 = list() #np.zeros((90,10))

    # Create the plot
    for idx,a in enumerate(scale_factors):

        # calculate da
        da = 0

        if idx != 0:
            da = scale_factors[idx] - scale_factors[idx-1]
        else:
            da = scale_factors[idx] - a_min

        # Calculate D(a)
        # Calculate \dot(D)
        d = helpers4.calculate_linear_growth(a)
        ddot = helpers4.calculate_linear_growth_dir(a -da/2)

        # Update position
        pos_x_new = (pos_x + s_x*d) % grid_size
        pos_y_new = (pos_y + s_y*d) % grid_size
```

```
        pos_z_new = (pos_z + s_z*d) % grid_size

        pos_top_10.append(pos_z_new[0:10,0,0])

        # Update momentum
        p_x = -(a-da/2)**2 * ddot *s_x
        p_y = -(a-da/2)**2 * ddot*s_y
        p_z = -(a-da/2)**2 * ddot*s_y

        momentum_top_10.append(p_z[0:10,0,0])

        # slices
        z_mask = np.where(abs(pos_z_new - 32) < 0.5)
        y_mask = np.where(abs(pos_y_new - 32) < 0.5)
        x_mask = np.where(abs(pos_x_new - 32) < 0.5)

        # Create the plots.

        # z
        plt.scatter(pos_x_new[z_mask], pos_y_new[z_mask],s=1,c='black')
        plt.title('a=' + str(a))
        plt.xlabel('x [Mpc]')
        plt.ylabel('y [Mpc]')
        plt.savefig('./Plots/4d/xy/4d_xy={0}.png'.format(idx))
        plt.close()

        # y
        plt.scatter(pos_x_new[y_mask], pos_z_new[y_mask],s=1,c='black')
        plt.title('a=' + str(a))
        plt.xlabel('x [Mpc]')
        plt.ylabel('z [Mpc]')
        plt.savefig('./Plots/4d/xz/4d_xz={0}.png'.format(idx))
        plt.close()

        # x
        plt.scatter(pos_y_new[x_mask], pos_z_new[x_mask],s=1,c='black')
        plt.title('a=' + str(a))
        plt.xlabel('y [Mpc]')
        plt.ylabel('z [Mpc]')
        plt.savefig('./Plots/4d/yz/4d_yz={0}.png'.format(idx))
        plt.close()

    # Plot the momentum and position
    pos_top_10 = np.array(pos_top_10)
    momentum_top_10 = np.array(momentum_top_10)

    # Momentum
    for i in range(0,10):
        plt.plot(scale_factors, momentum_top_10[:,i], label = 'particle ' + str(i
    +1))
        plt.xlabel('a')
        plt.ylabel(r'$P_z$')

    plt.legend()
    plt.savefig('./Plots/4d_momentum.pdf')
    plt.close()

    # Position
    for i in range(0,10):
        plt.plot(scale_factors, pos_top_10[:,i], label = 'particle ' + str(i+1))
        plt.xlabel('a')
        plt.ylabel('z')

    plt.legend()
    plt.savefig('./Plots/4d_pos.pdf')
    plt.close()
```

./Code/assigment_4.py

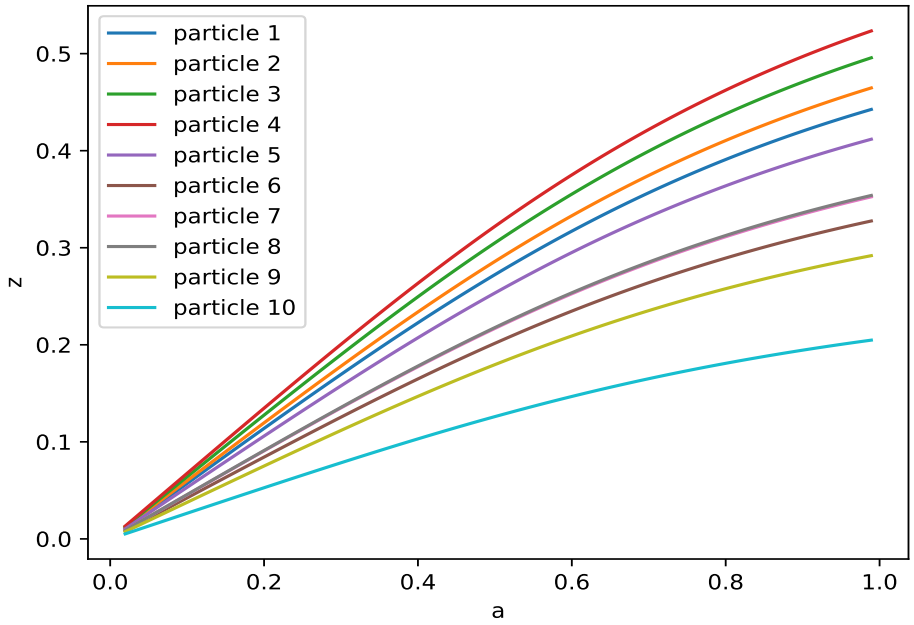

Figure 28: The z-positions of the first 10 particles against the scale factor.

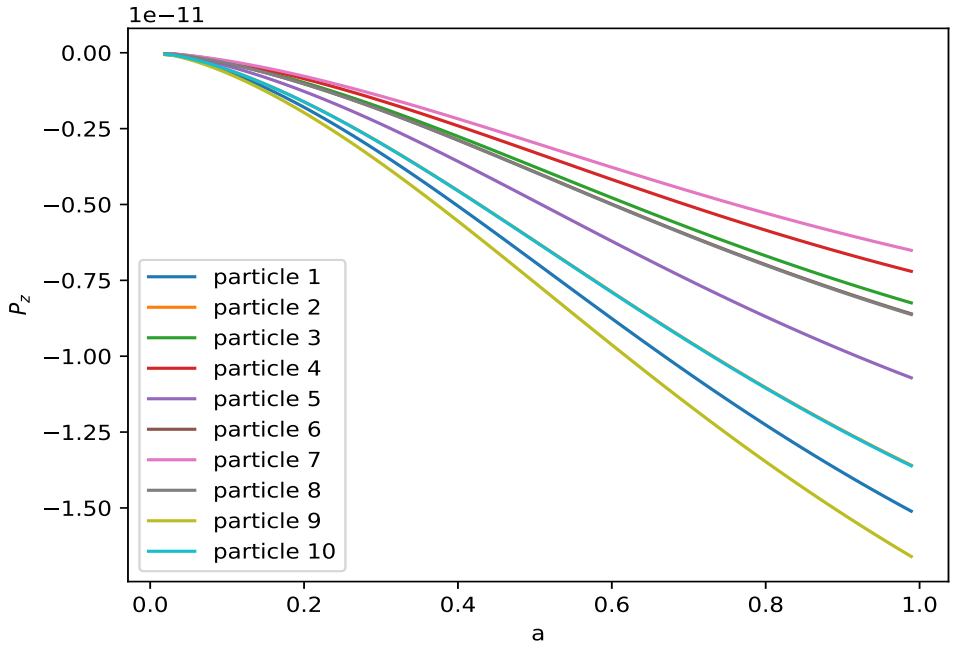

Figure 29: The z-component of the momentum of the first 10 particles against the scale factor.

Question 4 - Summary

Summary

The current sub-section contains the summary of the code used for assignment 4. This includes the part of the main file that executes the subquestions, the relevant imports, the functions to calculate the linear growth factor and the functions used to generate and sort the matrices (tensor).

Code - Assignment

The code with the main function and the function `gen_complex`, which is called in 4c and 4d.

```
import mathlib.integrate as integrate
import mathlib.misc as misc
import mathlib.random as rnd
import mathlib.helpers4 as helpers4
import matplotlib.pyplot as plt
import mathlib.derivative as ml_dir
import numpy as np
import scipy.fftpack

random = rnd.Random(783341)

def main():
    assigment4_a()
    assigment4_b()
    assigment4_c()
    assigment4_d()

def gen_complex(k, power, rand1, rand2):
    """
        Generate a complex number using the power
        spectrum.
    In:
        param: k -- The magnitude of the wavenumber.
        param: n -- The order of the power law.
        param: rand 1 -- A random uniform variable.
        param: rand 2 -- A random uniform variable.
    Out:
        return: A complex number in the fourier plane for the
                given power law.
    """

    sigma = np.sqrt(k**power)
```

./Code/assigment_4.py

Code - Linear growth

The code which calculates the linear growth factor and the derivative of the liner growth factor.

```
import numpy as np
import mathlib.integrate as integrate

def calculate_linear_growth(a_max, omega_m = 0.3, omega_lambda = 0.7):
    """
        Calculate the linear growth factor.
    In:
        param: a_max -- The value of a to evaluate the linear growth function at.
        param: omega_m -- The mass fraction in the universe.
        param: omega_lambda -- The dark energy fraction in the universe.
    Out:
        return: The value of the linear growth factor for the given parameters.
    """
    if a_max <= 0 :
        return 0

    # The prefactor of the integral
```

```
    pre_factor = 0.5*(5*omega_m)*(omega_m*a_max**(-3) + omega_lambda)**(0.5)

    # The function to integrate.
    func = lambda a: a**(-3) /(omega_m * (1/(a**3))+omega_lambda)**(3/2)

    return pre_factor*integrate.romberg(func, 1e-7, a_max, 15)

def calculate_linear_growth_dir(a, omega_m = 0.3, omega_lambda = 0.7):
    """
        Calculate the derivative of the linear growth factor
    In:
        param: a -- The scale factor for which to calculate it.
        param: omega_m -- The fraction of matter in the universe.
        param: omega_lambda -- The fraction of dark energy in the universe
    Out:
        return: The derivative of the scale factor for the given parameters
    """
    # Hubble constant
    H0 = 7.16e-11

    # The function to integrate, which appears in the second term.
    func = lambda a: a**(-3) /(omega_m * (1/(a**3))+omega_lambda)**(3/2)


    # The terms in the expression.
    pre_factor = (5*omega_m*H0)/(2*a**2)
    first_bracked_term = 1/(omega_m*a**(-3)+omega_lambda)**0.5
    second_bracked_term = (-3*omega_m*integrate.romberg(func, 1e-7, a, 15))/(2*a)

    return pre_factor*(first_bracked_term + second_bracked_term)
```

./Code/mathlib/helpers4.py

Code - Tensor/matrix

The code containing all the function sued to create the matrices (tensors) and make then symmetric.

```
import numpy as np

def gen_wavenumbers(size, min_distance):
    """
        Generate the shifted wavenumbers
        for the discrete fourier transform.
    In:
        param: size -- The size of the matrix.
        param: min_distance -- The distance of a cell/ the sample spacing.
    Out:
        return: An array with shifted wave numbers.
    """
    # Array to return.
    ret = np.zeros(size)

    # Positive values
    ret[0:int(size/2)+1] = np.arange(0,int(size/2)+1)

    if size  % 2 == 0: # even
        ret[int(size/2):] = -np.arange(int(size/2),0,-1)
    else: # odd
        ret[int(size/2)+1:] = -np.arange(int(size/2),0,-1)


    return (ret/(size*min_distance))*2*np.pi


def generate_matrix_2D(size, min_distance, func, random_numbers, power):
    """
        Generate a 2D matrix with complex numbers in
        shifted fourier coordinates using the power spectrum
    In:
        param: size -- The size of the matrix (size x size).
```

```
        param: min_distance -- The physical size of 1 cell.
        param: func -- A functiion that takes the power and to random uniform
                       variables to calculate the correct complex number.
        param: random_numbers -- An array with random uniform numbers.
                                 Must be of atleast size: size x size x 2.
        param: power -- The power of the power spectrum to create the matrix for.
    Out:
        return: A 2D matrix with complex numbers assigned by the power spectrum
                in fourier shifted coordinates.
    """

    # Generate the shifted wavenumbers
    wavenumber = gen_wavenumbers(size,min_distance)
    # The matrix to return
    ret = np.zeros((size,size),dtype=complex)

    # A counter for the random uniform variables.
    steps = 0

    # Fill the matrix
    for i in range(size):
        for j in range(size):

            # Element of k_0,k_0 is left zero.
            if i == 0 and j == 0:
                continue

            # Calculate the magnitude of the wavenumbers.
            k = np.sqrt(wavenumber[i]**2 + wavenumber[j]**2)
            # Fill the matrix.
            ret[i][j] = func(k, power,
                             random_numbers[steps], random_numbers[steps+1])
            steps += 2

    # Return the matrix
    return ret


def make_hermitian2D(matrix):
    """
        Give a matrix in shifted fourier coordinates
        the correct hermitian symmetry so that the ifft is real.
    In:
        param: matrix -- The matrix to give the correct symmetry.
    Out:
        return: A matrix with the correct hermitan symmetry so that the
                ifft is real.
    """

    # The size of the matrix
    size = matrix.shape[0]

    # Loop over the rows
    for row in range(1, int(size/2) +1):

        # Give the first column (index 0) has the correct symmetry (see report
    point A)
        matrix[row,0] = complex(matrix[size-row,0].real,
                                - matrix[size-row,0].imag)
        # Give the first row (index 0) the correct symmetry (see report point B)
        matrix[0, row] = complex(matrix[0, size-row].real,
                                 - matrix[0,size- row].imag)

        # Give the inner matrix the correct symmetry (see report point C)
        for column in range(1, size):
            matrix[row, column] = complex(matrix[size-row, size-column].real,
                                          -matrix[size-row, size-column].imag)

    # Corrections for even matrix
    if size % 2 == 0:
        matrix[int(size/2), 0] = matrix[int(size/2), 0].real + 0J
```

```
104         matrix[0, int(size/2)] = matrix[0, int(size/2)].real + 0J
105         matrix[int(size/2), int(size/2)] = matrix[int(size/2), int(size/2)].real
    + 0J
106
107     # Return the matrix.
108     return matrix
109
110
111
112
113 def generate_matrix_3D(size, min_distance, func, random_numbers, power):
114     """
115         Generate a 3D matrix with complex numbers in
116         shifted fourier coordinates using the power spectrum
117     In:
118         param: size -- The size of the matrix (size x size).
119         param: min_distance -- The physical size of 1 cell.
120         param: func -- A functiion that takes the power and to random uniform
121                        variables to calculate the correct complex number.
122         param: random_numbers -- An array with random uniform numbers.
123                                  Must be of atleast size: size x size x 2.
124         param: power -- The power of the power spectrum to create the matrix for.
125     Out:
126         return: A 3D matrix with complex numbers assigned by the power spectrum
127                 in fourier shifted coordinates.
128     """
129
130     # Generate the shifted wavenumbers
131     wavenumber = gen_wavenumbers(size,min_distance)
132     # The matrix to return
133     ret = np.zeros((size,size,size),dtype=complex)
134
135     # A counter for the random uniform variables.
136     steps = 0
137
138     # Fill the matrix
139     for u in range(size):
140         for i in range(size):
141             for j in range(size):
142
143                 # Element of k_0,k_0,k_0 is left zero.
144                 if i == 0 and j == 0 and u ==0:
145                     continue
146
147                 # Calculate the magnitude of the wavenumbers.
148                 k = np.sqrt(wavenumber[i]**2 + wavenumber[j]**2 + wavenumber[u
    ]**2)
149                 # Fill the matrix.
150                 ret[u][i][j] = func(k, power,
151                                     random_numbers[steps], random_numbers[steps+1])
152                 steps += 2
153
154     # Return the matrix
155     return ret
156
157
158
159 def make_hermitian3D(tensor):
160     """
161         Give a 3D matrix (tensor) in shifted fourier coordinates
162         the correct hermitian symmetry so that the ifft is real.
163     In:
164         param: tensor -- The tensor to give the correct symmetry.
165     Out:
166         return: A tensor with the correct hermitan symmetry so that the
167                 ifft is real.
168     """
169
170     # Note: I am fully aware that the loops below can be combined.
171     #       The way it is now is however the first way in which I managed
172     #       to get it correct and is also the way in which it is easyer to follow
```

```
    .

    size = tensor.shape[0]

    # Inner matrix
    for row in range(1, size):
        for column in range(1, size):
            for depth in range(1,size):
                tensor[row, column, depth] = complex(tensor[size-row,size-column,
    size-depth].real, -tensor[size-row, size-column, size-depth].imag)

    # Outside top
    for depth in range(1,int(size/2)+1):

        tensor[0,0,depth] = complex(tensor[0,0,size-depth].real, -tensor[0,0,size
    -depth].imag)
        tensor[0,depth,0] = complex(tensor[0,size-depth,0].real, -tensor[0,size-
    depth,0].imag)
        tensor[depth,0,0] = complex(tensor[size-depth,0,0].real, -tensor[size-
    depth,0,0].imag)


         # outside inner
        for column in range(1, size):
            tensor[depth, column,0] = complex(tensor[size-depth, size-column,0].
    real, -tensor[size-depth, size-column,0].imag)
            tensor[depth,0,column] = complex(tensor[size-depth,0, size-column].
    real, -tensor[size-depth,0, size-column].imag)
            tensor[0, depth, column] = complex(tensor[0,size-depth,size-column].
    real, - tensor[0,size-depth,size-column].imag)


    if size % 2 == 0:
        # Finally fix niquist
        for i in range(0,size):
            for j in range(0,size):
                tensor[int(size/2),i,j] = tensor[int(size/2),i,j].real + 0J
                tensor[i,j,int(size/2)] = tensor[i,j,int(size/2)].real + 0J
                tensor[i,int(size/2),j] = tensor[i,int(size/2),j].real + 0J

    return tensor
```

./Code/mathlib/misc.py

Code - Romberg

The code for romberg integration

```
import numpy as np

def trapezoid(function, a, b, num_trap):
    """
        Perform trapezoid integration

    In:
        param: function -- The function to integrate.
        param: a -- The value to start the integration at.
        param: b -- The value to integrate the function to.
        param: num_trap -- The number of trapezoids to use.

    Out:
        return: An approximation of the integral from a to b of
                the function 'function'
    """

    # The step size for the integration range.
    dx = (b-a)/num_trap

    # Find trapezoid points.
    x_i = np.arange(a,b+dx,dx)

```

```
24      # Determine the area of all trapezoids.
25      area = dx*(function(a) + function(b))/2
26      area += np.sum(function(x_i[1:num_trap])) *dx
27
28      return area
29
30
31
32  def romberg(function, a, b, num_trap):
33      """
34          Perform romberg integration
35
36      In:
37          param: function -- The function to integrate.
38          param: a -- The value to start the integration at.
39          param: b -- The value to stop the integrate at.
40          param: num_trap -- The maximum number of initial trapezoid
41                             to use to approximate the area.
42
43      Out:
44          return: An approximation of the integral from a to b of
45                  the function 'function'
46      """
47
48      # The array to store the combined results in.
49      results = np.zeros(num_trap)
50
51      # The current combination:
52      # 0 = combine trapezoids, 1 = combine combined trapezoids,
53      # 2 = combine the combined combined trapezoids etc.
54      for combination in range(0, num_trap-1):
55
56          # Iterate and combine.
57           for j in range(1, num_trap-combination):
58              # Create the initial trapezoids to combine.
59              if combination == 0:
60                   # We need two trapezoids to combine the very first time.
61                  if j == 1:
62                      results[j-1] = trapezoid(function, a, b, 1)
63
64                  results[j] = trapezoid(function,a,b,2**j)
65
66              # Combine.
67              power = 4**(combination+1)
68              results[j-1] = (power*results[j] - results[j-1])/(power-1)
69
70      return results[0]
```

./Code/mathlib/integrate.py

5 - Mass assignment schemes

Question 5.a

Problem

The most simple choice for the particle shape is a point like shape given by:

$$S(x) = \frac{1}{\Delta x}\delta\left(\frac{x}{\Delta x}\right) \tag{40}$$

Explain how we need to assign mass to the grid in this scheme and explain why this method is called the Nearest Grid Point (NGP) method. Code up your own implementation of the mass assignment scheme NGP, using a grid of 16^3. Display x-y slices of the grid with z values of 4,9,11 and 14.

Solution

The easiest way of assigning the mass of the particles to a grid is by assigning it to the grid points. For the given particle shape this would correspond of assigning the full mass of a particle to the nearest gird point, from which the name follows. For other particle shapes, such as the cloud in a cell shape, the mass might be fully assignment the nearest gird point, but might also be partially assigned to multiple grid points, depending on the position of the particle.
The mass is for the given particle shape is assigned to the gird points by abusing the fact that a cast to integers always result in down a cast (i.e 15.7 casted to an integer gives 15). The indices of the grid points to which the particles have to assign their mass can by abusing this be found by adding 0.5 to the position of a particle and then casting it to an integer. To include circular boundary conditions the modulo with 16 (grid size) is taken of the result.
The code that creates the plots with the slices and the plots can be found below. The code that assigns the mass can be found in the shared module `./Code/mathlib/mass.py` on page ...

Code - slices

The code that creates the plots with the slices of the mass grid.

```
    #assigment_5a()
    #assigment_5b()
    #assigment_5c()
    assigment_5d()
    #Passigment_5e()

def assigment_5a():
    """
        Execute assignment 5.a
    """

    # Create the positions of the particles.
    np.random.seed(121)
    positions = np.random.uniform(low =0, high=16, size=(3,1024))

    # Create the mass grid.
    mass_grid = mass._assign_mass_NGP(positions)

    # Plot the slices
    for z in [4,9,11,14]:
```

./Code/assigment_5.py

Plots - Slices

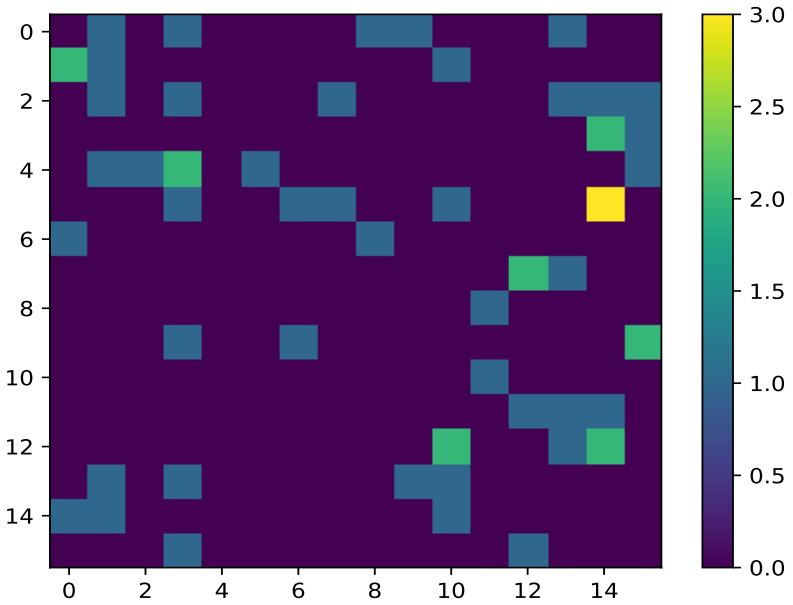

Figure 30: The x-y slice of the created mass grid for $z = 4$. The color indicates the assignment mass in terms of particle mass.

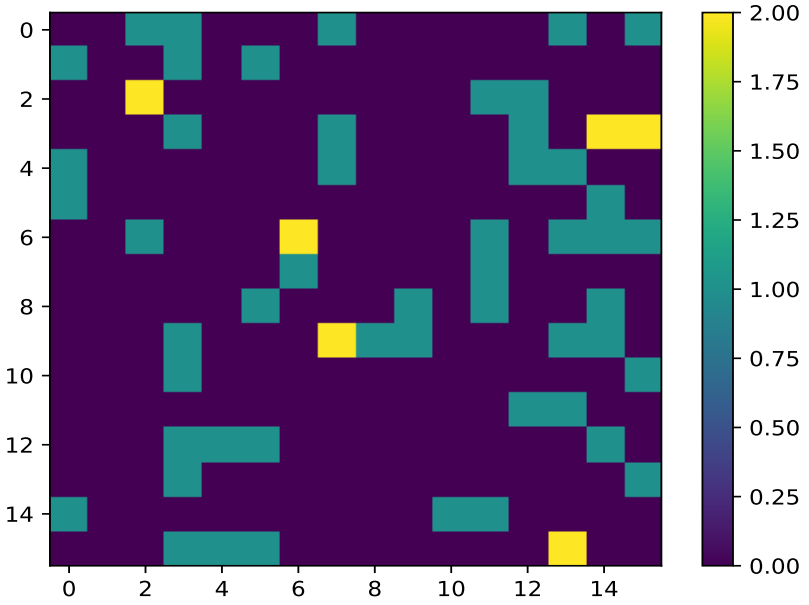

Figure 31: The x-y slice of the created mass grid for $z = 9$. The color indicates the assignment mass in terms of particle mass.

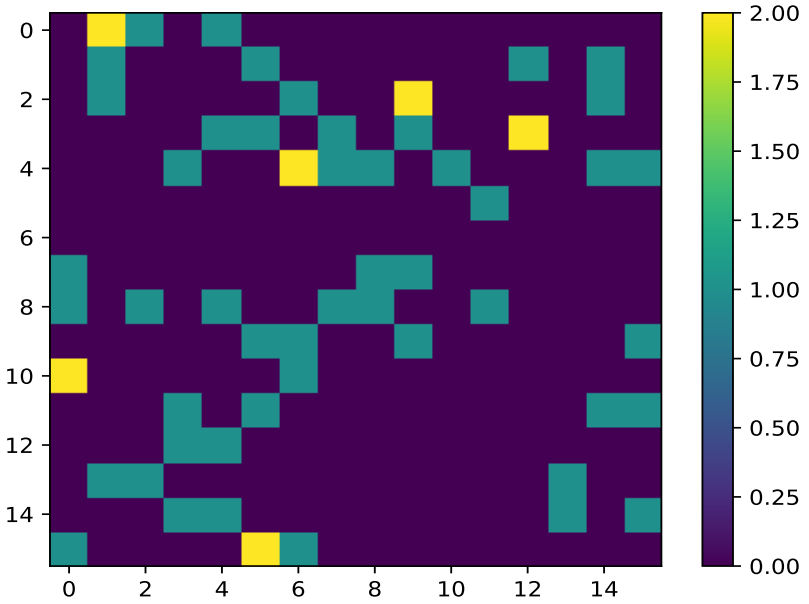

Figure 32: The x-y slice of the created mass grid for $z = 11$. The color indicates the assignment mass in terms of particle mass.

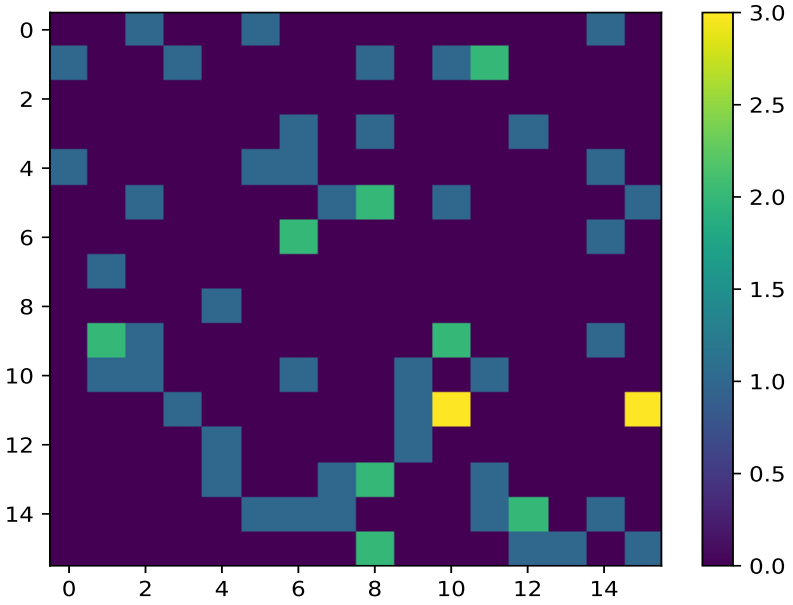

Figure 33: The x-y slice of the created mass grid for $z = 14$. The color indicates the assignment mass in terms of particle mass.

Question 5.b

Problem

To check the robustness of your implementation make a plot of the x position of an individual particle and the value in cell 4 in 1 dimension and let x vary from the lowest value to the highest possible value in x. Repeat for cell 0.

Solution

There is not much to explain here, besides that the results for cell 4 and cell 0 are shown in the same plot. The code that creates the plots and the plots can be found below.

Code - slices

The code that creates the plots for cell 0 and cell 1.

```
1
2           plt.imshow(slice)
3           plt.colorbar()
4           plt.savefig('./Plots/5a_slice_{0}.pdf'.format(z))
5           plt.figure()
6
7   def assigment_5b():
8       """
9           Execute assignment 5.b
10      """
11
12      # The x positions of the moving particle
13      x_values = np.linspace(0, 16, 1000)
```

./Code/assigment_5.py

Plots - Cells

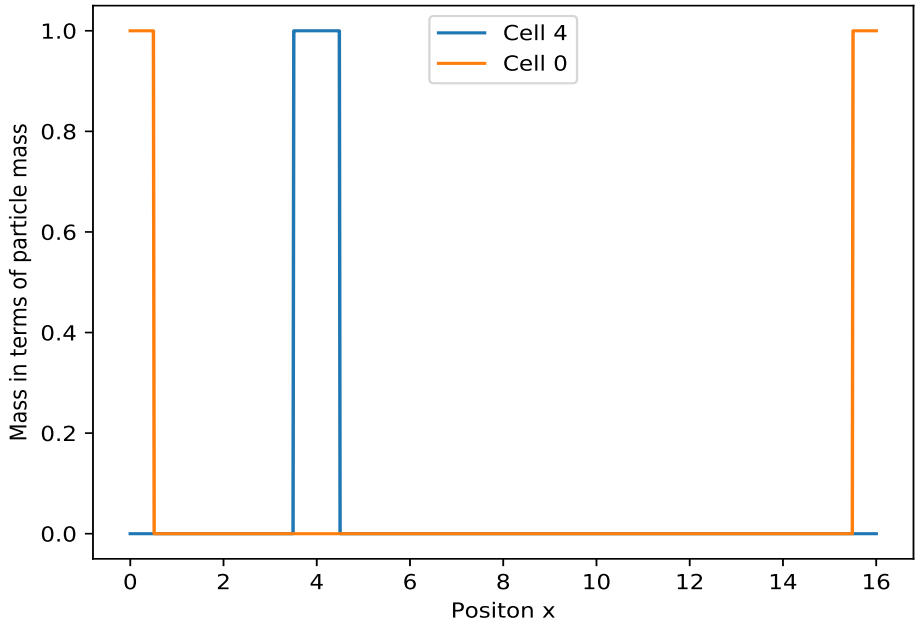

Figure 34: The mass assigned for a particle moving from x = 0 to x = 16. The orange and red line respectively indicate the mass assigned to cells 0 and 4 as function of the position of the particle. The plot is created for a mass grid of size 16 with circular boundary conditions.

Question 5.c

Problem

Right noew we want to improve this method, because NGP has several disadvantages. For this we are going to use a method which assumes that particles are cubes in 3D of uniform density and have the size of a grid cell. Calculatue how mass needs to be assigned in the case of this other method called the Cloud In Cell (CIC) methoed and implement it in code. To check the robustness of your implementation again make the same plots as before.

Solution

The mass needs to be assigned over the nearest 8 grid points. Each grid point gets thereby a fraction of the cube around the particle. The fraction assigned to a grid point here depends on the position of the particle and might be zero for some of the nearest 8 grid points. The way in which the mass is assigned to the nearest gird points is explained in the comment of the code that assigns the mass(page ...). The code that creates the plots and the plots them self can be found below.

Code - slices

The code that creates the slices, cell 0 and cell 4 in a 3D grid of size 16.

```
    plt.legend()
    plt.savefig('./Plots/5b_cell.pdf')
    plt.figure()

def assigment_5c():
    """
        Execute assigment 5.c
    """

    np.random.seed(121)
    positions = np.random.uniform(low =0, high=16, size=(3,1024))

    # Create the mass grid.
    mass_grid = mass._assign_mass_CIC(positions)

    # Plot the slices.
    for z in [4,9,11,14]:
        slice = mass_grid[:,:,z]
        #slice[slice == 0] = np.nan

        plt.imshow(slice)
        plt.colorbar()
        plt.savefig('./Plots/5c_slice_{0}.pdf'.format(z))
        plt.figure()

    # The x positions of the moving particle
    x_values = np.linspace(0, 16, 1000)

    # The y positions of the moving particle for
    # cell 0 and 4
    y_values_4 = list()
    y_values_0 = list()

    # For each position create the mass grid and get the value in cell 4 and 0
    for x in x_values:
        grid = mass._assign_mass_CIC(np.array([[x],[0],[0]]))
        y_values_0.append(grid[0,0,0])
        y_values_4.append(grid[4,0,0])

    # plot the positions
    plt.plot(x_values, y_values_4, label='Cell 4')
    plt.plot(x_values, y_values_0, label='Cell 0')
    plt.xlabel('Positon x')
```

./Code/assigment_5.py

Plots - Slices

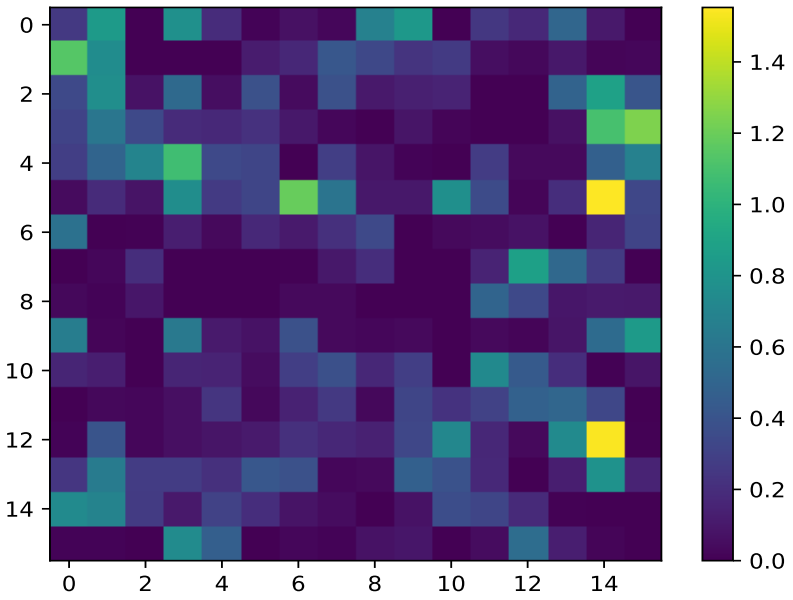

Figure 35: The x-y slice of the created mass grid with the CIC method for $z = 4$. The color indicates the assignment mass in terms of particle mass.

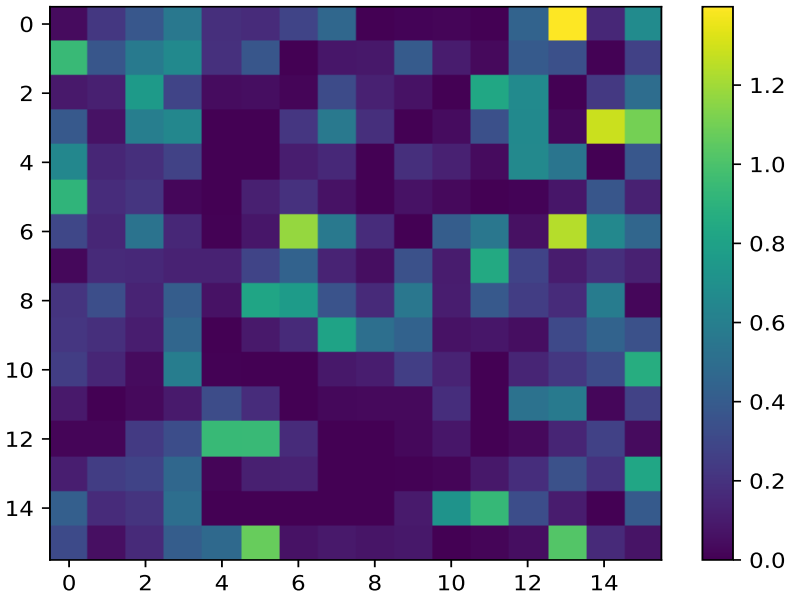

Figure 36: The x-y slice of the created mass grid with the CIC method for $z = 9$. The color indicates the assignment mass in terms of particle mass.

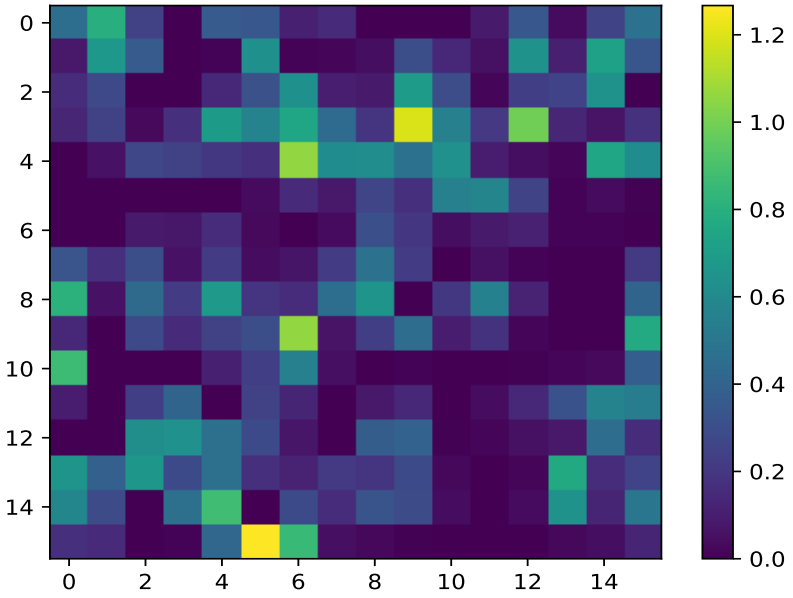

Figure 37: The x-y slice of the created mass grid with the CIC method for $z = 11$. The color indicates the assignment mass in terms of particle mass.

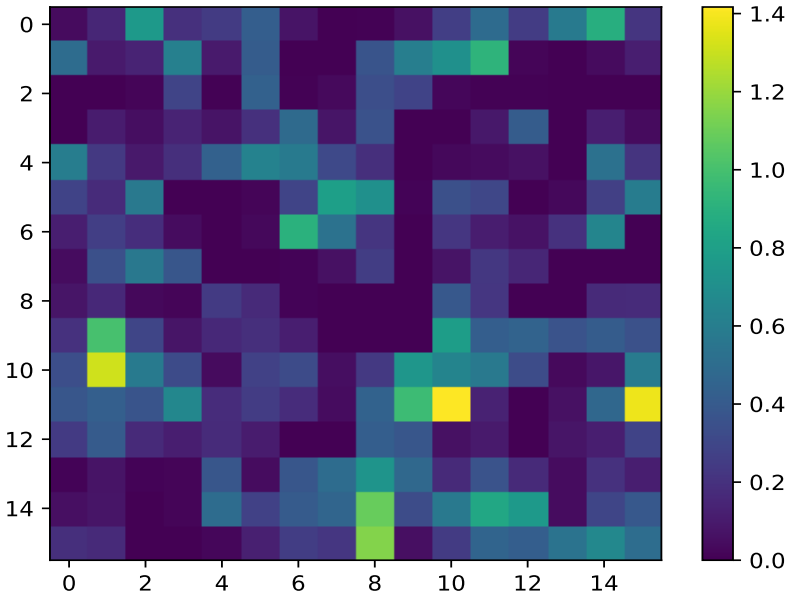

Figure 38: The x-y slice of the created mass grid with the CIC for $z = 14$. The color indicates the assignment mass in terms of particle mass.

Plots - Cells

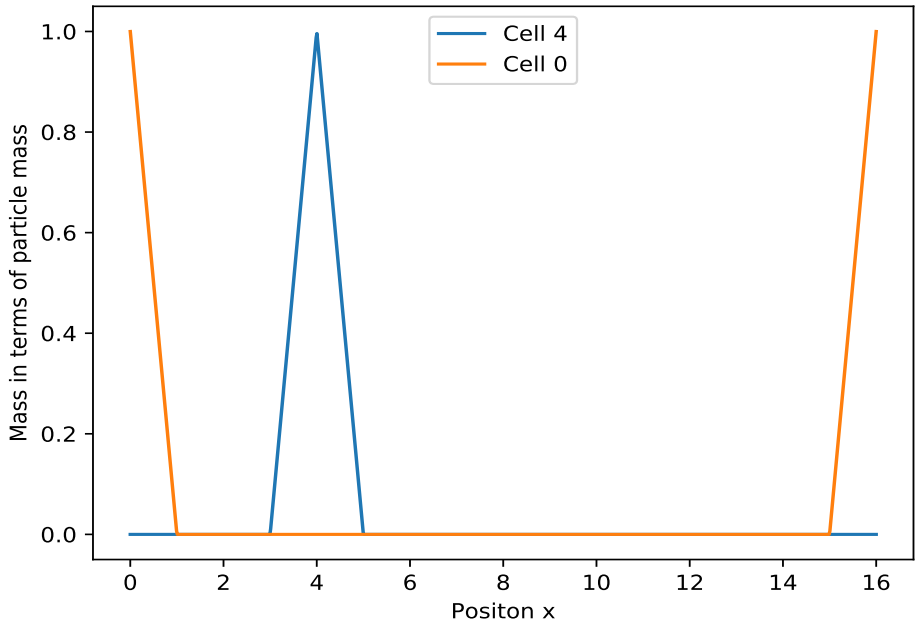

Figure 39: The mass assigned for a particle moving from x = 0 to x = 16 for the cloud in the cell method. The orange and red line respectively indicate the mass assigned to cells 0 and 4 as function of the position of the particle. The plot is created for a 3D mass grid of size 16 with circular boundary conditions.

Question 5.d

Problem

Write your own FFT algorithm; check that your code works with a 1D function (no Gaussian) by making a plot of the FFt and compare your result with a python package and the analytical FFT of your function. In the rest of this exercise you need to use your own FFT.

Solution

The created implementation of the FFT consist of an recursive implementation with the Cooley-Tukey algorithm. The implementation doesn't store the data inplace, but in a new array. This choice was made to prevent he input array from being modified. One consequence of this choice is that the input array does not have to be reverse bit shifted.
The implementation is compared with the numpy implementation. The function that is chosen to fourier transform is the cosine[4],

$$\mathcal{F}(\cos(2\pi At)] \propto 0.5(\delta(f - A) + \delta(f + A)) \tag{41}$$

The code for the FFT can be found on page .. in the file .... The code that compares the self written implementation with the numpy implementation and the analytical solution can be found below.

[4]The given expression contains a proportional sign as the pre-factor depends on the definition of the Fourier transformation.

Code - slices

The code that creates the plots for the comparison of the self written FFT with the numpy implementation and the analytical solution.

```
def assigment_5d():
    """
        Execute assignment 5.d
    """

    # Create the data to fourier transform.
    size = 64

    t = np.linspace(0, size, size)
    f = 5
    y = np.cos(2*np.pi*f*t)


    # Execute the FFT
    fft_np = np.fft.fft(y)
    fft_self = fourier.fft(y)

    # Frequencies to plot for.
    freq = misc.gen_wavenumbers(size,1)*size

    plt.plot(freq, abs(fft_np), label='numpy',linestyle=':', zorder=1.1)
    plt.plot(freq,abs(fft_self), label='self',zorder=1.0)
    plt.vlines(-2*np.pi*f,0,max(abs(fft_self))+10,label='Analytical')
    plt.vlines(2*np.pi*f,0,max(abs(fft_self))+10)

    plt.xlabel('Frequence')
    plt.ylabel('Power')
    plt.legend()
    plt.savefig('./Plots/5d_fourier.pdf')
    plt.close()
```

./Code/assigment_5.py

Plot - Comparison

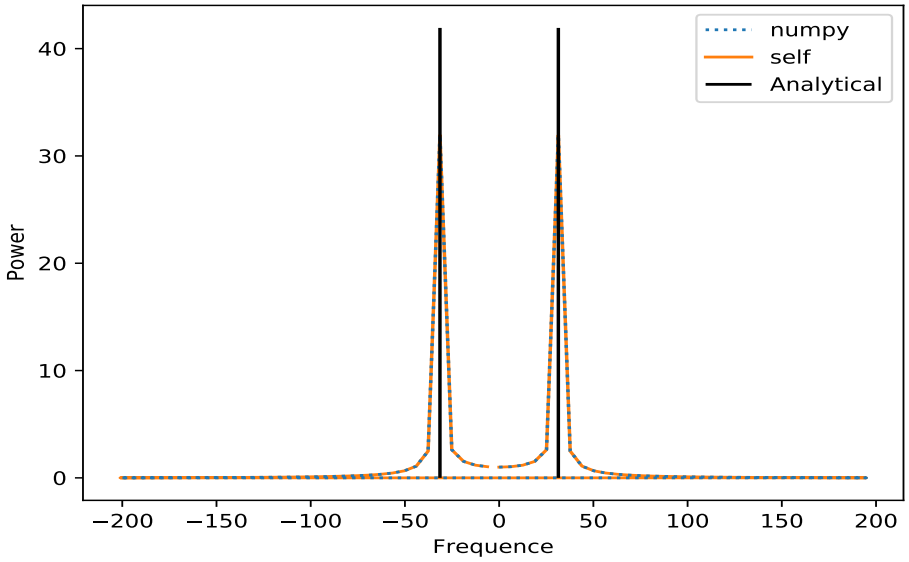

Figure 40: The own implementation of the FFT (orange), the numpy implementation (blue) and the analytical fft (black). The plot shows that there is now visible deviation from the numpy version. It can also be seen that both the numpy and the self written implementation do not correctly represent the peak of the delta function. This is expected as it would require an infinite amount of samples to obtain the exact same result.

7 - Building a quadtree

Question 7

Problem

Download the file containing 1200 particle masses and positions. Considering only the x and y coordinates of these particles (between 0 and 150), construct a Barnes-Hut quadtree with at most 12 particles per leaf node. You can treat the masses and positions as dimensionless and use $G = 1$. Plot the particles and indicate which particles are in which node. Calculate the $n = 0$ multipole moment of each leaf and then recursively for each node in the tree. Print the $n = 0$ mulitpole moment for the leaf node containing the particle with index $i = 100$ and that of all its parent nodes up to and including the root node.

Solution

The zeroth order multipole moment for a leaf node corresponds to the sum of the masses of the particles in that leaf node. The multipole moment of a parent node (can be a parent of a leaf, or of a other non leaf) is the sum of the multipole moment of its children. With this knowledge a quadtree has been constructed that calculates the multipole moment for each of its nodes. The origin of the tree is chosen to be x = 0, y = 0 and corresponds with the left bottom coordinate of the root quad. The size of the root node is chosen to be 150. The root node is thus a quad with edge points: (0,0) (origin), (150,0) right bottom, (0,150) left top, (150,150) right top.
The code for the tree, a plot of the particles in the tree and the multipole moments can be found below. The code is split over two files, the first file contains the code that constructs the quad tree and adds the particles to the tree. The second file contains the code that contains the quad tree its self.

Code - Output

The code that constructs the quad tree and adds the particles to it.

```
import h5py
import matplotlib.pyplot as plt
import numpy as np
import mathlib.quadtree as qt

# Load the data
particles_data = h5py.File('colliding.hdf5')['PartType4']
particles_pos = np.array(particles_data['Coordinates'])
particles_masses = np.array(particles_data['Masses'])

# Create a combined array with mass and positions
# (I assumend that np.concatenate was not allowed)
particle_info = np.zeros((len(particles_masses),4))
particle_info[:,3] = particles_masses
particle_info[:,2] = particles_pos[:,2]
particle_info[:,1] = particles_pos[:,1]
particle_info[:,0] = particles_pos[:,0]

# Create an instance of the quad tree with origin (0,0)
# size 150 and max 12 particles per node.
tree = qt.QuadTree((0,0), 150, 12)

# Add the particles to the tree.
for particle in particle_info:
    tree.add_boddy(particle)

# Before creating the plot, plot the particle with
# index 100.
plt.scatter(particle_info[100,0],particle_info[100,1],c='red',s=100)
# Create the plot with final particles.
tree.plot()

# Print the moments of the leaf containing the particle with
# index 100 and the moment of its parent nodes.
tree.print_moments(particle_info[100,0],particle_info[100,1])
```

./Code/assigment_7.py

Code - Tree

The code of the quadtree.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.collections as collections

class QuadTree(object):
    """
        An object representing a quad tree
    """

    def __init__(self, left_bottom, size, max_boddies):
        """
            Create an instance of the quadtree.
        In:
            param: left_bottom -- The coordinates of the leftbottom point of
                                  the root quad.
            param: size -- The size of the root quad.
            param: max_boddies -- The maximum amount of boddies to add before
                                  splitting a node.
        """

        # Create the root quad.
        self._root = Quad(left_bottom, size, max_boddies, None)

    def add_boddy(self, boddy):
```

```
26          """
27              Add a boddy to the tree.
28          In:
29              param: boddy -- The boddy to add.
30          """
31
32          # Add the boddy to the root node
33          self._root.add_boddy(boddy)
34
35      def find_leaf(self, pos_x, pos_y):
36          """
37              Find the leaf node that contains the specific positon.
38          In:
39              param: pos_x -- The x coordinate of the position.
40              param: pos_y -- The y coordinate of the position.
41          Out:
42              return: The leaf node that contains the specific positon.
43          """
44
45          # Start by assuming that the root node is the node.
46          leaf = self._root
47
48          # If it has children find the child containing the position.
49          while not leaf._leaf:
50              leaf = leaf._find_quad(pos_x, pos_y)
51
52          # return the quad that contains the current position.
53          return leaf
54
55      def print_moments(self, pos_x, pos_y):
56          """
57              Print the multipole moment for the leaf node
58              and al its parents that contain the given positon.
59          In:
60              param: pos_x -- The x coordinate of the position.
61              param: pos_y -- The y coordinate of the position.
62          """
63
64          # Find the leaf that contains the given position.
65          leaf = self.find_leaf(pos_x, pos_y)
66
67          # Print the mulitpole moment of the leaf
68          print(leaf._moment)
69
70          # Iterate through its parents and print the
71          # multipole moment of the parents
72          parent = leaf._parent
73
74          # Only root doesn't have a parent, thus abort
75          # when parent is root.
76          while parent._parent != None:
77              print(parent._moment)
78              parent = parent._parent
79
80          # Don't forget printing the moment of the root node.
81          print(parent._moment)
82
83      def plot(self):
84          """
85              Create a visual representation of the quad tree
86              and the boddies added to the tree.
87          """
88
89          # The axis used for plotting
90          axis = plt.gca()
91          # An list that contains rectangle objects (matplotlib.patches.Rectangle)
92          # for each of the quads in the tree.
93          rectangles = list()
94
95          # Fill the list with rectangles by recursively calling the
96          # children of the root. If a quad is a leaf it will furtheremore
```

```
97          # axis.scatter. This is to add the boddies of that leaf
98          # to the plot. The reason that a quad not directly adds its own
99          # rectangle to the axis (axis.add_patches) is to save time.
100         self._root._plot(axis, rectangles)
101
102         # Add all the rectangles at once to save time.
103         axis.add_collection(collections.PatchCollection(rectangles,
104                             match_original=True))
105
106         # Create and save the plot
107         plt.xlim(self._root._left_bottom[0],
108                  self._root._left_bottom[1] + self._root._size)
109         plt.ylim(self._root._left_bottom[1],
110                  self._root._left_bottom[1] + self._root._size)
111         plt.xlabel('x')
112         plt.ylabel('y')
113         plt.savefig('./Plots/7_quadtree.pdf')
114         plt.figure()
115
116 class Quad(object):
117
118     def __init__(self, left_bottom, size, max_boddies, parent = None):
119         """
120             Create an instance of a quad in a quadtree.
121         In:
122             param: left_bottom -- The coordinates of the leftbottom point of
123                                   the  quad.
124             param: size -- The size of the  quad.
125             param: max_boddies -- The maximum amount of boddies to add before
126                                   splitting this quad.
127             param: parent -- The parent quad if any.
128         """
129
130         # Geometric properties of the current quad
131         self._left_bottom = left_bottom
132         self._size = size
133         self._halve_size = size/2
134
135         # 'Facts' about the quad at the moment of initalization
136         self._max_boddies = max_boddies
137         self._leaf = True
138         self._parent = parent
139
140         # An array containing child quads (if any) and
141         # the boddies in this quad (if it is a leaf).
142
143         # The child quads are named:
144         # Left Top (index 0), Left Bottom (index 1)
145         # Right top (index 2), Right bottom (index 3)
146         self._child_quads = None
147         self._boddies = list()
148
149         # The multipole moment of the current quad.
150         self._moment = 0
151
152
153     def add_boddy(self, boddy):
154         """
155             Add a boddy to the current quad or to one of its
156             child quads.
157         In:
158             param: boddy -- The boddy to add. Should be an array
159                             in which the first for elemnts respectivelty
160                             correspond with the x-position, y-position,
161                             z-positon and mass.
162         """
163
164
165         # If current quad is not a leaf find
166         # the child quad that should hold the boddy.
167         if not self._leaf:
```

```
168             self._find_quad(boddy[0], boddy[1]).add_boddy(boddy)
169             return
170 
171         # Current quad is a leaf and can still add a boddy.
172         if len(self._boddies) < self._max_boddies:
173             # Add the boddy.
174             self._boddies.append(boddy)
175             # Update the multipole moment for this quad.
176             self._update_moment(boddy[3]) #index 3 = mass.
177 
178         # Current quad is a leaf, but must split when adding a new boddy.
179         else:
180             # The boddy is before splitting first added
181             # to the current quad. When it splits the boddies
182             # of this quad will be assigned to the child quads created
183             # in the split.
184             self._boddies.append(boddy)
185             # Split the current quad and make it a leaf.
186             self._split()
187 
188 
189 
190     def _find_quad(self, pos_x, pos_y):
191         """
192             Find a child quad that contains the specific
193             position.
194         In:
195             param: pos_x -- The x coordinate of the position.
196             param: pos_y -- The y coordinates of the position.
197         Out:
198             return: The child quad containing the position. If this
199                     quad is a leaf, then the quad its self is returned.
200         """
201 
202         # Current quad is a leaf, return its self.
203         if self._leaf:
204             return self
205 
206         # Positon is in the right top or right bottom quad.
207         if pos_x > self._halve_size+self._left_bottom[0]:
208             # Position is in the right top quad.
209             if pos_y >= self._halve_size + self._left_bottom[1]:
210                 return self._child_quads[2]
211             # Position is in the right bottom quad.
212             else:
213                 return self._child_quads[3]
214         # Position is in the left top quad.
215         elif pos_y >= self._halve_size + self._left_bottom[1]:
216             return self._child_quads[0]
217         # Position is in the left bottom quad.
218         else:
219             return self._child_quads[1]
220 
221     def _split(self):
222         """
223             Split the current quad in 4 child quads.
224         """
225 
226         # Initialize the array that holds the child wuads.
227         self._child_quads = list()
228 
229         # Left bottom coordinates of child quads
230         # lt = left top, lb = left bottom, rt = right top, rb = right bottom.
231         lt = [self._left_bottom[0],
232               self._left_bottom[1] + self._halve_size]
233         lb = self._left_bottom
234         rt = [self._left_bottom[0] + self._halve_size,
235               self._left_bottom[1] + self._halve_size]
236         rb = [self._left_bottom[0] + self._halve_size,
237               self._left_bottom[1]]
238 
```

```
239             # Add the quads
240             self._child_quads.append(Quad(lt, self._halve_size,
241                                           self._max_boddies,self))
242             self._child_quads.append(Quad(lb, self._halve_size,
243                                           self._max_boddies,self))
244             self._child_quads.append(Quad(rt, self._halve_size,
245                                           self._max_boddies,self))
246             self._child_quads.append(Quad(rb, self._halve_size,
247                                           self._max_boddies,self))
248
249             # Current quad is not a leaf anymore
250             self._leaf = False
251
252             # Add the boddies of the current quad to the new child quads.
253             for boddy in self._boddies:
254                 self.add_boddy(boddy)
255
256             # Empty the boddies that where hold by t his quad.
257             self._boddies = None
258
259         def _update_moment(self, mass = 0):
260             """
261                 Update the multipole moment of the current quad
262             In:
263                 praram: mass -- The mass to update the multipole moment with if the
264                                 current quad is a leaf.
265             """
266
267             # Current quad is a leaf, thus
268             # add the mass (zeroth order multipole moment)
269             if self._leaf:
270                 self._moment += mass
271             # Current quad is not a leaf.
272             else:
273                 # Reset the multipole moment and recalculate
274                 # it by looping of the child quads.
275                 self._moment = 0
276
277                 for quad in self._child_quads:
278                     self._moment += quad._moment
279
280             # Make sure the parents of the current quad update the multipole moment.
281             if self._parent != None:
282                 self._parent._update_moment()
283
284
285         def _plot(self, axis, quads):
286             """
287                 Plot the current quad.
288             In:
289                 param: axis -- The axis object to plot this quad at.
290                 param: qaudss -- A list of quads at which the current
291                                  quad should add a rectangle which represents
292                                  the current quad.
293             """
294
295             # Create the rectangle to plot of the current quad and add it to the list
296             rect = patches.Rectangle(self._left_bottom, self._size,
297                                           self._size, fill=False)
298             quads.append(rect)
299
300             # If the current quad is a leaf plot the boddies.
301             if self._leaf:
302                 if len(self._boddies) == 0:
303                     return
304                 a = np.array(self._boddies)
305                 axis.scatter(a[:,0],a[:,1],c='blue',s=1)
306             # If the current quad is not a leaf recursively call this
307             # method for all its 4 child quads.
308             else:
309                 for quad in self._child_quads:
```

```
310                 quad._plot(axis, quads)
```

./Code/mathlib/quadtree.py

Output - Text

The text output produced by the code. This consists of the multipole moments of the leaf containing the particle with index 100 and its parents. The first line is thus the multipole moment of the leaf containing the particle with index 100 and the last line is the multipole moment of the root.

```
0.12500000186264515
0.21250000316649675
0.28750000428408384
7.162500106729567
7.162500106729567
15.000000223517418
```

./Output/assigment7_out.txt

Output - Plot

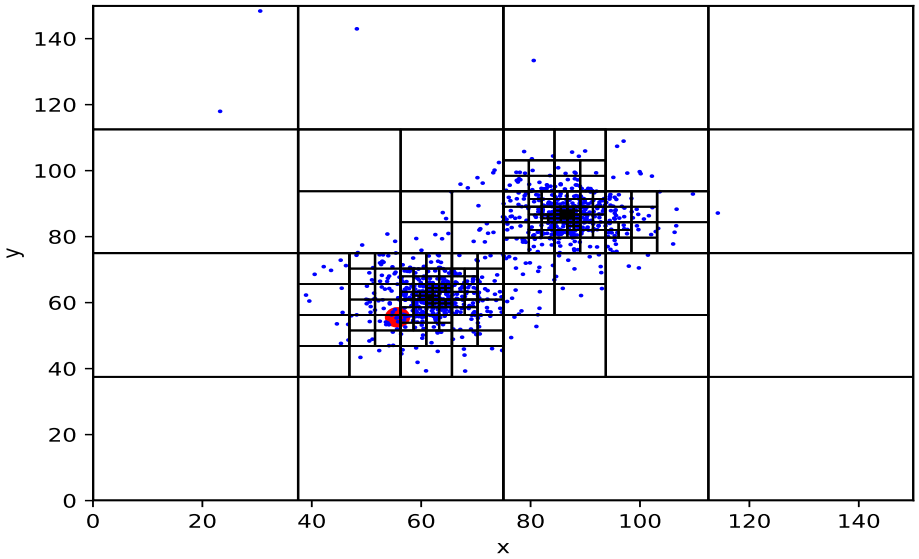

Figure 41: A visual representation of the constructed quadtree. The blue points indicate the positions of the bodies added to the tree. The red point indicates the position of the body with index 100.