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 of the 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.

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

    # The relevant imports for this piece of code are:

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

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

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

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

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

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

    # Extra, to print the smallest and lagest bin value.
    counts, _ = np.histogram(numbers_mil, bins=20)
    print('[1.a] Max counts: ', max(counts))
    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
```
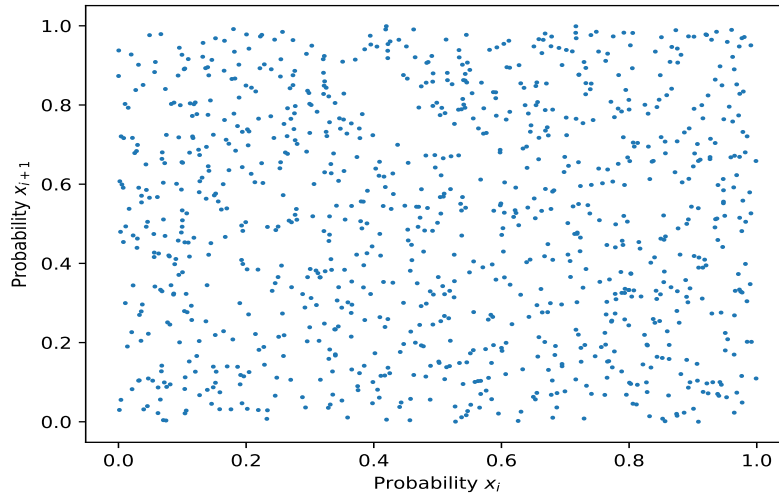
./Output/assigment1_out.txt

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.
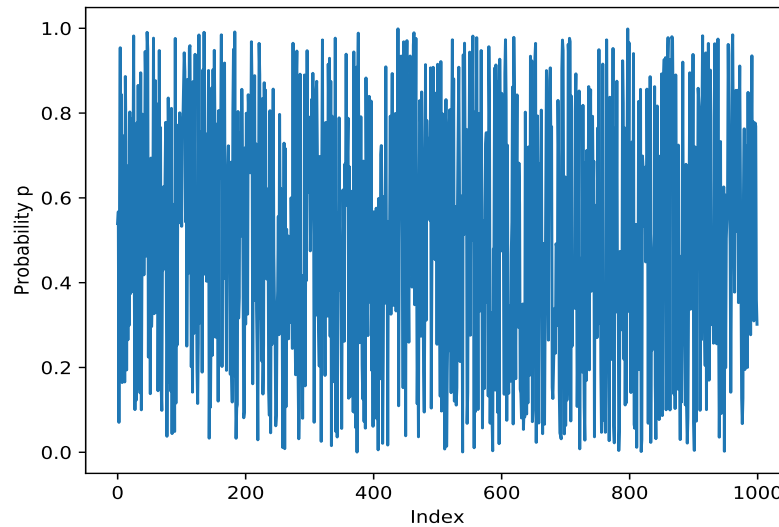

Figure 2: The first 1000 random uniform numbers produced by the random number generator (RNG) against their index. A good random number generator should not have large wide gaps (e.g when moving from index 400 to 450 it should not only produce values larger than 0.8, which would leave a wide gap). In 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 $p = 0.2$.
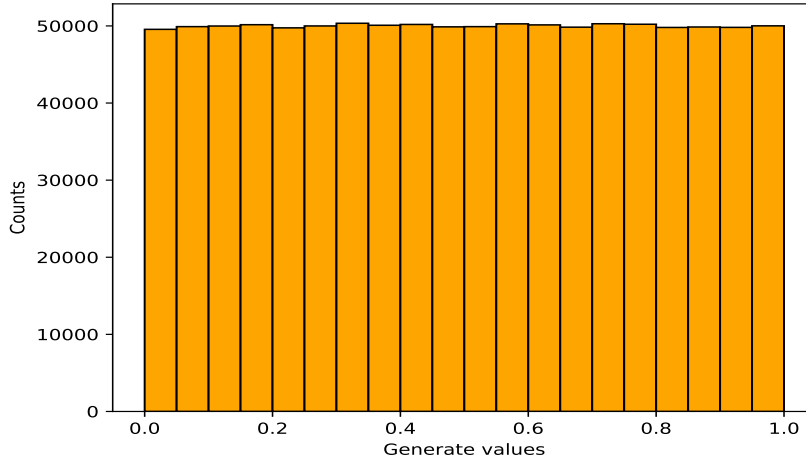
Figure 3: The uniforms of the random number generator for 1 million random values. The values are binned in 20 bins. A good random number generator should fluctuate around $50000 \pm 2\sqrt{50000} = 50000 \pm 447$ counts per bin (2 sigma). The maximum and minimum amount of counts corresponds to 50343 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,

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

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

$$P(R \leq r_1, \Theta \leq \theta_1) = \int_0^{r_1} \int_0^{\theta_1} G(r\cos(\theta)\sigma + \mu|\mu, \sigma^2)G(r\sin(\theta)\sigma + \mu|\mu, \sigma^2)rdrd\theta$$

$$= \frac{1}{2\pi\sigma^2} \int_0^{r_1} \int_0^{\theta_1} re^{-\frac{1}{2}\left[\left(\frac{r\cos(\theta)}{\sigma}\right)^2 + \left(\frac{r\sin(\theta)}{\sigma}\right)^2\right]} drd\theta$$

$$= \frac{1}{2\pi\sigma^2} \int_0^{r_1} \int_0^{\theta_1} re^{-\frac{r^2}{2\sigma^2}} drd\theta$$

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

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

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

The CDFs can be used to convert two uniform distributed variables to the polar coordinates of the Gaussian distributed variables. Let $U_1, U_2 \sim U(0,1)$ be two i.i.d uniform variables. From the transformation law of probability we then must have that,

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

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

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

$$r_1 = \sqrt{-2\sigma^2 \ln(1 - u_1)} \tag{6}$$
$$\theta_1 = 2\pi u_2 \tag{7}$$

Converting back to Cartesian coordinates 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.

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

    # The relevant imports for this piece of code are:

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

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

    # Generate 1000 random normal variables for the given mean and sigma.
```

```python
        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
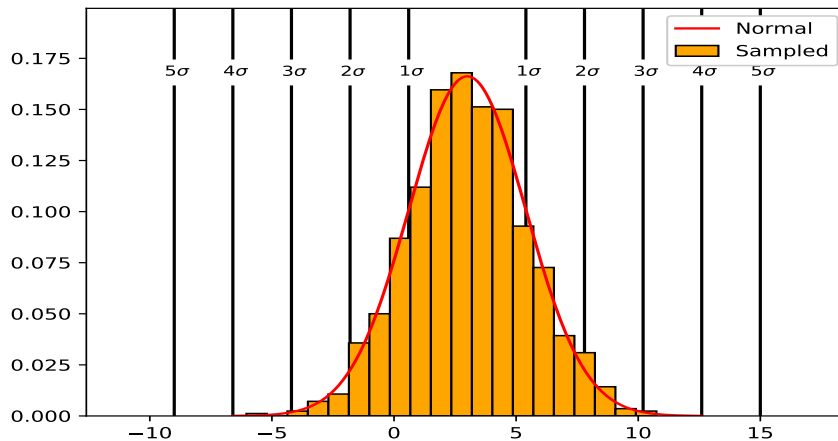
Code - Output plot(s)

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,

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

where the erf is given by,

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

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*,

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

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

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.
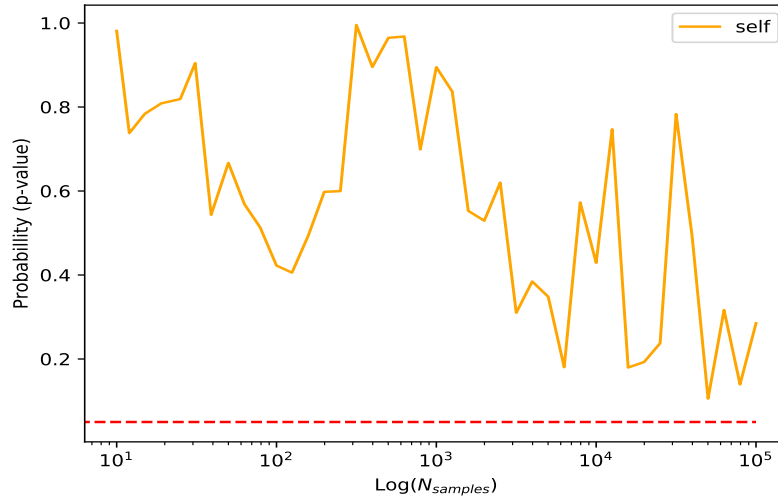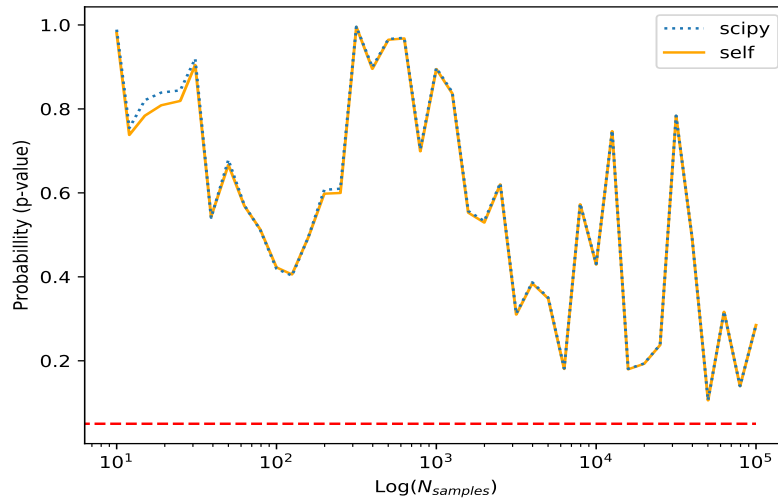

Figure 6: The P-value produced by the KS-test against the number of samples on which the KS-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. The orange line is the self written implementation of the KS-test and the blue line is the scipy version. A point **below**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

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

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

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

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

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

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

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

---

[1]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.
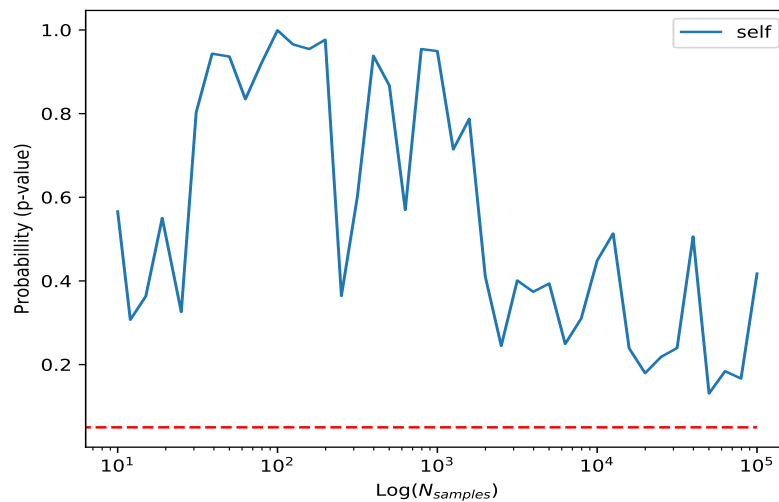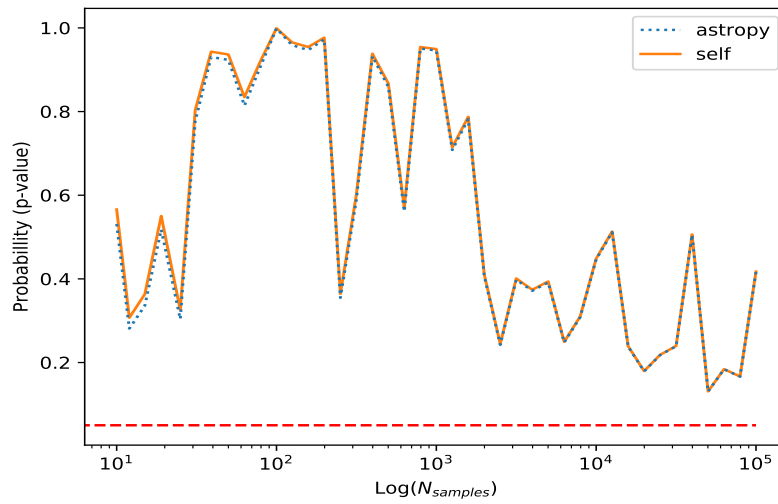
Figure 8: The P-value produced by the kuiper test against the number of samples on which the kuiper-test is performed for the self written RNG. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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. This indicates that there is enough statistical evidence to reject the hypothesis that they follow a normal distribution with $\sigma = 1$ and $\mu = 0$.

---

[2]A p-value only shows statistical evidence against the null hypothesis, it isn't a measure of how good the null hypothesis is.

Code - Plots

The code for generating the 10 plots.

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

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

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

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

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

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

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


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

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

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

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

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

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

./Code/assigment_1.py

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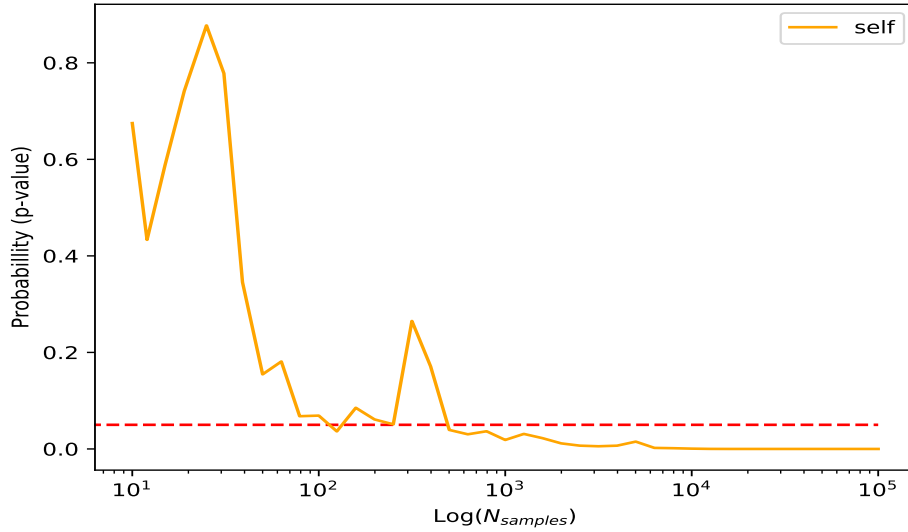
Figure 11: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **third** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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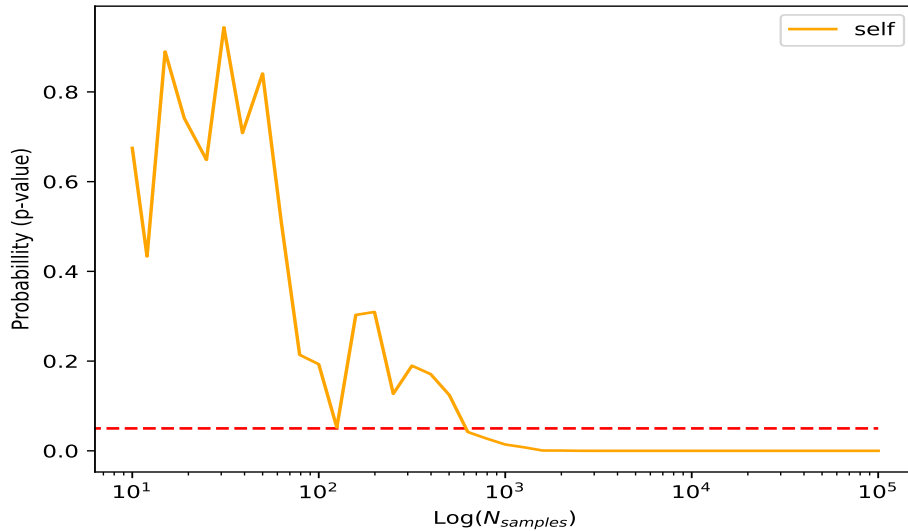

Figure 12: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **fourth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to **reject** the (null)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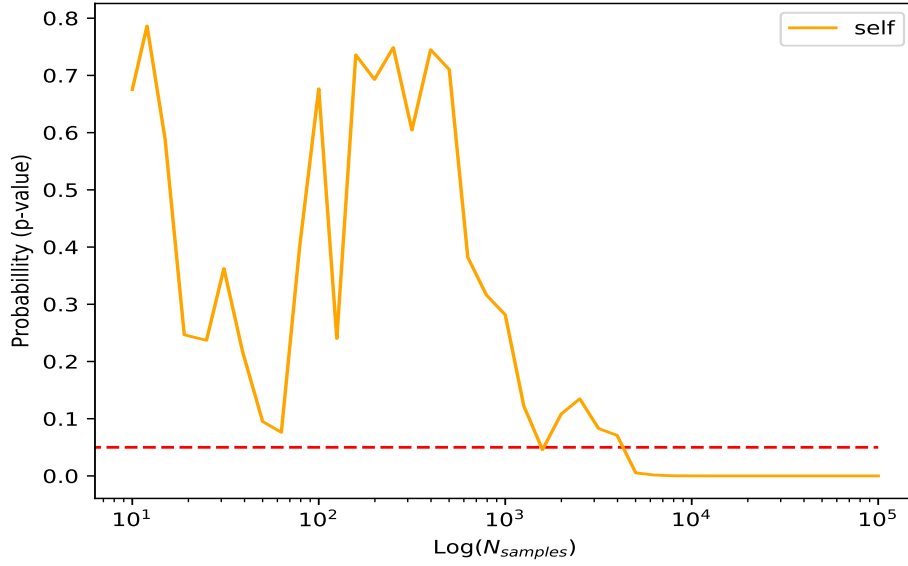

Figure 14: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **sixth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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 $\sigma = 1$.
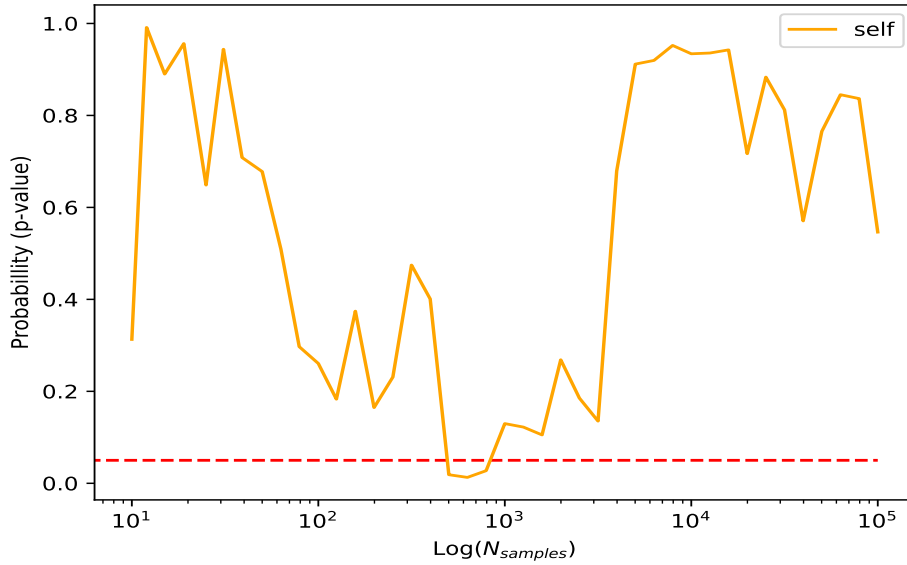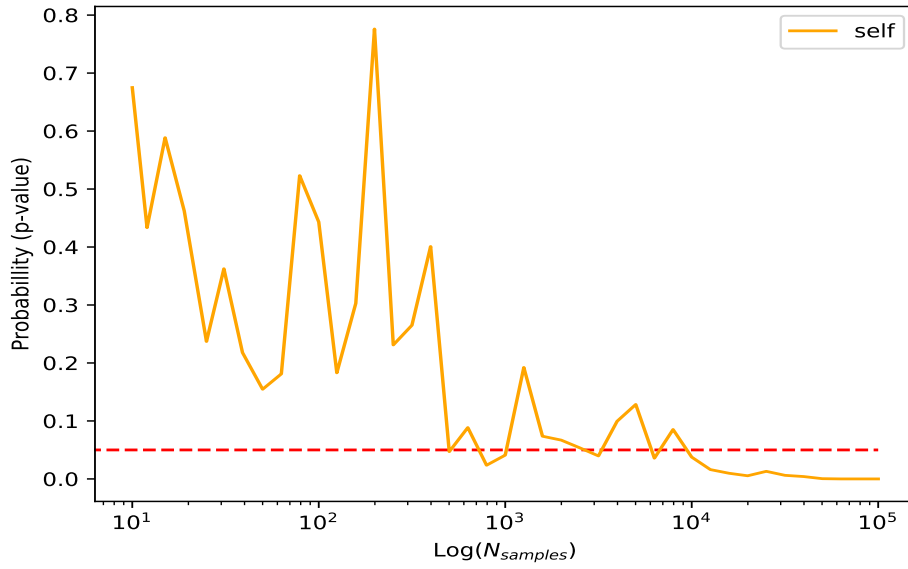
Figure 15: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **seventh** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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 $\sigma = 1$.
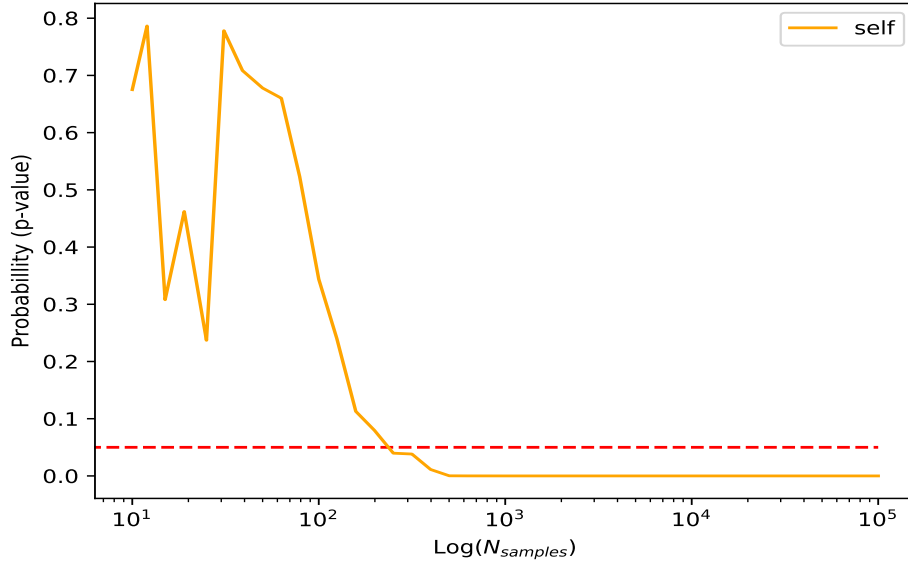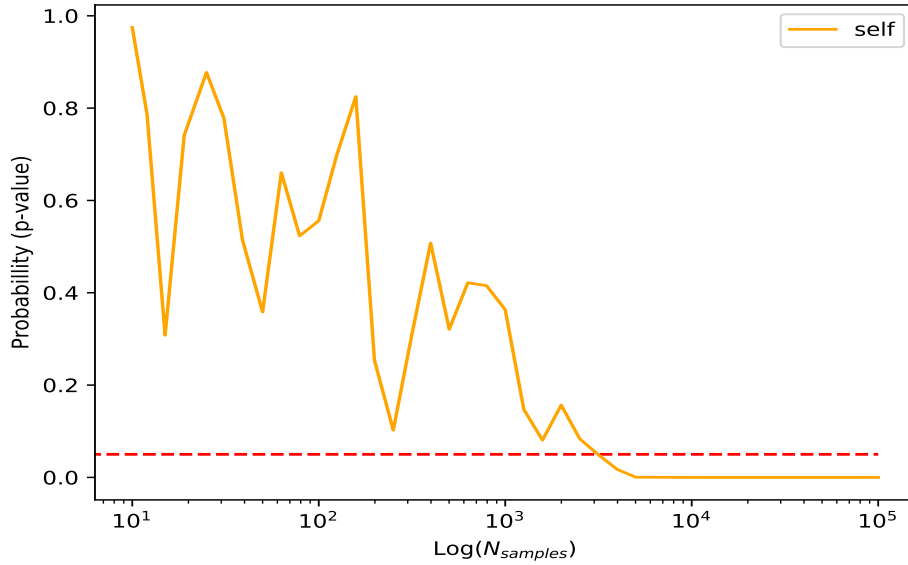

Figure 16: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **eight** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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 $\sigma = 1$.
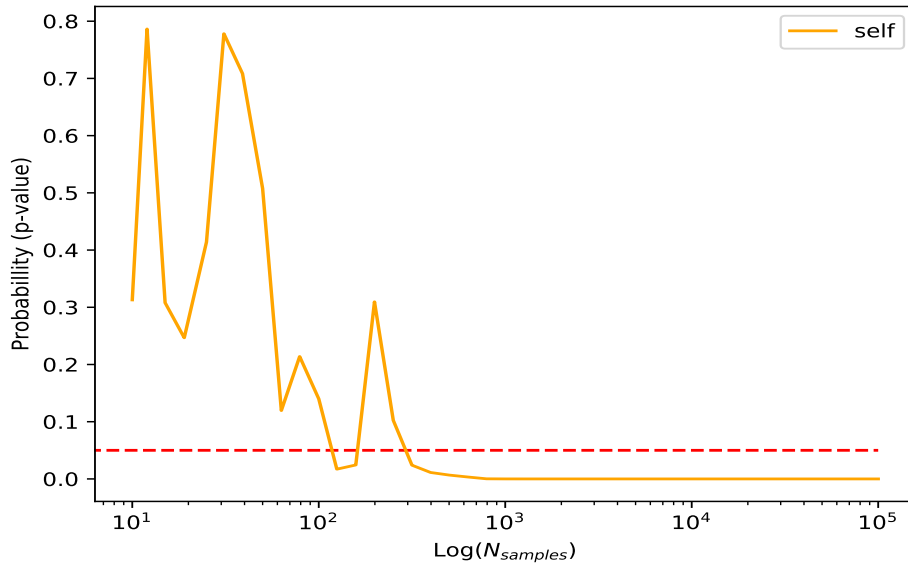
Figure 17: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **ninth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis 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 $\sigma = 1$.
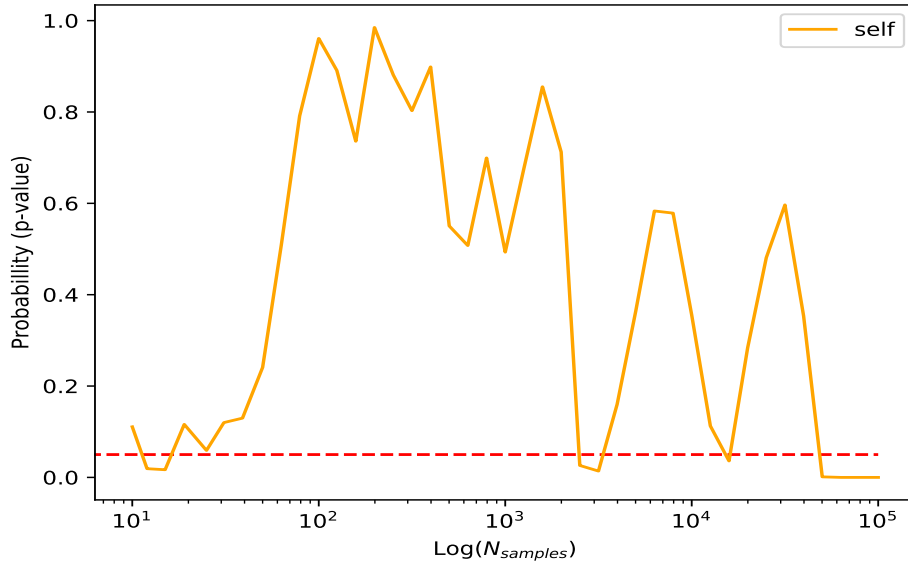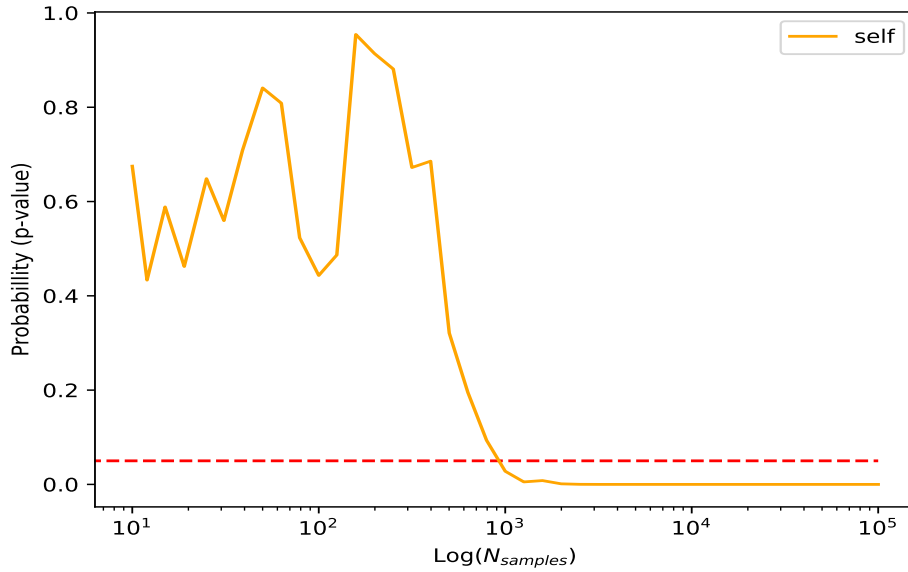

Figure 18: The P-value produced by performing the KS-test2 for a normal distribution with $\mu = 0$ and $\sigma = 1$ on the **tenth** column. The red line indicates the line of $p = 0.05$. A point **below** there line would suggests that there is enough statistical evidence to reject the (null) hypothesis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

```python
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.

```python
import numpy as np

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

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

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

        """

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

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

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

        # The values for the multiply with carry.
        self._mwc_a = np.uint64(4294957665)
```

```python
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.
```

```python
103             """
104
105             # Generate two uniform variables.
106             u1 = self.gen_uniform()
107             u2 = self.gen_uniform()
108
109             # Use the box muller transformation.
110             return sigma*np.sqrt(-2* np.log(1-u1))*np.cos(2*np.pi*u2) + mean
111
112     def gen_normal_uniform(self, mean, sigma, u1, u2):
113             """
114                 Generate a random normal distributed float from two provided
115                 uniform variables.
116
117
118             """
119             pre_factor= sigma*np.sqrt(-2* np.log(1-u1))
120
121             return pre_factor*np.cos(2*np.pi*u2) + mean, pre_factor*np.sin(2*np.pi*u2
        ) + mean
122
123     def gen_normals(self, mean, sigma, amount):
124             """
125                 Generate multible random normal distributed float.
126
127             In:
128                 param: mean -- The mean of the gaussian distribution.
129                 param: sigma -- The squareroot of the variance of the distribution.
130                 param: amount -- The amount of floats to generate.
131             Out:
132                 return: An array with random floats drawn from the parameterized
        normal
133                         distribution.
134             """
135
136             # Pre-factors in the box muller transformation.
137             square_pre_factor = -2*sigma**2
138             angle_pre_factor = 2*np.pi
139
140             # With the Box-muller two random normals can be generated for two
141             # uniforms. If the amount of requested variables is odd then add
142             # one to it and later remove it when returning the result.
143             elements = amount if amount % 2 == 0 else amount + 1
144
145             # Array in which the drawn normal distributed variables are stored.
146             normal_dist = np.zeros(elements)
147
148             # Apply the box muller transformation to generate the samples.
149             for i in range(0, elements, 2):
150
151                 # Generate the uniforms.
152                 u1 = self.gen_uniform()
153                 u2 = self.gen_uniform()
154
155                 # Calculate common terms.
156                 pre_fact = np.sqrt(square_pre_factor*np.log(1-u1))
157
158                 # Calculate the samples.
159                 normal_dist[i] = pre_fact*np.cos(angle_pre_factor*u2) + mean
160                 normal_dist[i+1] = pre_fact*np.sin(angle_pre_factor*u2) + mean
161
162             # If amount is odd, don't return the last element.
163             return normal_dist[0:amount]
164
165     def _update_state(self):
166             """
167                 Update the state of the random number generator.
168
169             Out:
170                 return: The new state of the random number generator.
171             """
```

```
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 26.

```
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)
```

```python
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
71      null hypothesis.
72      Out:
73          return: The p-value obtained by performing the KS-test
74      """
75
76      # Amount of values in the input distributions.
77      x1_size = len(x1)
78      x2_size = len(x2)
79
80      # Sort both arrays.
81      x1 = sorting.merge_sort(x1)
82      x2 = sorting.merge_sort(x2) if type(x2_sorted) is not None else x2_sorted
83
84      # The maximum distance
85      max_dist = 0
86
87      # The iteration values used to determine
88      # the emperical pdf's and the max distance.
89      x1_i, x2_j = 0,0
90
91      # Find the maximum distance by updating the emperical CDFs.
92      while x1_i < x1_size and x2_j < x2_size:
93
94          # Update the indices used for the emperical CDF's.
```

```python
 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
117        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,
147        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:
```

```python
            param: z -- The value to calculate the cdf at.
        Out:
            return: An approximation of the cdf for the given value.       print(
        max_dist_above + max_dist_below)
        """

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

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

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


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

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

    # Approximateed value of the sum by performing 100 iterations

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

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

    return 1- 2*ret




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

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


def normal(x, mean = 0, sigma = 1):
    """
        Evaluate the normal distrbution for the given
        parameters.
```

```
231    In :
232        param: x — The point to evaulte the distribution at.
233        param: mean — The mean of the distribution.
234        param: sigma — The square root of the variance for the distribution.
235    Out:
236        return: The value of the parameterized distribution evaluated at the
       given point.
237    """
238
239    return 1/((np.sqrt(2*np.pi)*sigma))*np.exp(-0.5*((x - mean)/sigma)**2)
240
241 def erf(x):
242    """
243        Evaluate the erf function for a value of x.
244    In :
245        param: x — The value to evaluate the erf function for.
246    Out:
247        return: The erf function evaluated for the given value of x.
248    """
249
250    # Numerical approximation taken from Abramowits and Stegun.
251
252    # Constants for the numerical approximation
253    p = 0.3275911
254    a1 = 0.254829592
255    a2 = -0.284496736
256    a3 = 1.421413741
257    a4 = -1.453152027
258    a5 = 1.061405429
259
260    # Array in which the result is stored
261    ret = np.zeros(len(x))
262
263    # The approximation functions
264    erf_func_t_val = lambda x: 1/(1+ p*x)
265    erf_func_approx = lambda t, x : 1 - t*(a1 + t*(a2 + t*(a3 + t*(a4 + t*a5))))
       *np.exp(-x**2)
266
267    # Evaluate for both positive and negative
268    neg_mask = x < 0
269    neg_x = x[neg_mask]
270    pos_mask = x >= 0
271    pos_x = x[pos_mask]
272
273    ret[neg_mask] = -erf_func_approx(erf_func_t_val(-neg_x), -neg_x)
274    ret[pos_mask] = erf_func_approx(erf_func_t_val(pos_x), pos_x)
275
276    return ret
```

./Code/mathlib/statistics.py

Code - Sorting

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

```
1 import numpy as np
2
3 def merge_sort(array):
4    """
5        Sort an array using merge sort
6
7    In :
8        param: array — The array to sort.
9    Out:
10        return: The sorted array.
11    """
12
13    # Get the length of the array.
14    size = len(array)
```

```python
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 $1 \le i, j < N$. 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.

```python
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:
```

```
63        sigma = 1/k
64     else:
65        sigma = np.sqrt(k**n)
66
67    # Determine the complex value
68    a,b = random.gen_normal_uniform(0,sigma,rand1,rand2)
69    return complex(a,b)
70
71
72
73 if __name__ == "__main__":
74    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.

```python
1  import numpy as np
2
3  def gen_wavenumbers(size, min_distance):
4      """
5          Generate the shifted wavenumbers
6          for the discrete fourier transform.
7      In:
8          param: size —— The size of the matrix.
9          param: min_distance —— The distance of a cell/ the sample spacing.
10     Out:
11         return: An array with shifted wave numbers.
12     """
13     # Array to return.
14     ret = np.zeros(size)
15
16     # Positive values
17     ret[0:int(size/2)+1] = np.arange(0,int(size/2)+1)
18
19     if size % 2 == 0: # even
20         ret[int(size/2):] = -np.arange(int(size/2),0,-1)
21     else: # odd
22         ret[int(size/2)+1:] = -np.arange(int(size/2),0,-1)
23
24
25     return (ret/(size*min_distance))*2*np.pi
26
27
28 def generate_matrix_2D(size, min_distance, func, random_numbers, power):
29     """
30         Generate a 2D matrix with complex numbers in
31         shifted fourier coordinates using the power spectrum
32     In:
33         param: size —— The size of the matrix (size x size).
34         param: min_distance —— The physical size of 1 cell.
35         param: func —— A functiion that takes the power and to random uniform
36                        variables to calculate the correct complex number.
37         param: random_numbers —— An array with random uniform numbers.
38                                  Must be of atleast size: size x size x 2.
39         param: power —— The power of the power spectrum to create the matrix for.
40     Out:
41         return: A 2D matrix with complex numbers assigned by the power spectrum
42                 in fourier shifted coordinates.
43     """
44
45     # Generate the shifted wavenumbers
46     wavenumber = gen_wavenumbers(size,min_distance)
47     # The matrix to return
48     ret = np.zeros((size,size),dtype=complex)
49
50     # A counter for the random uniform variables.
51     steps = 0
```

```python
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
90  point A)
          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
```

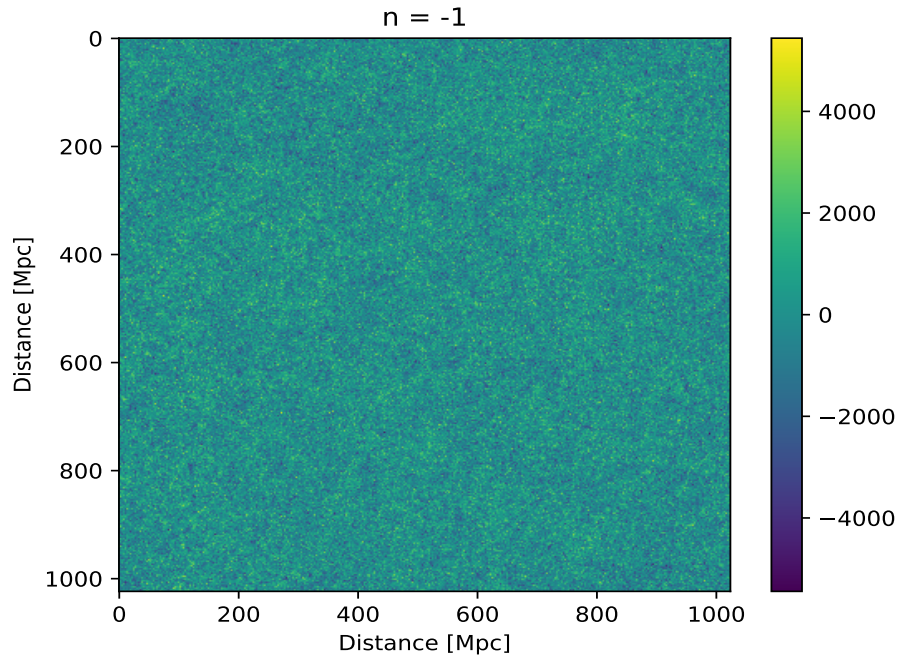./Code/mathlib/misc.py

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.
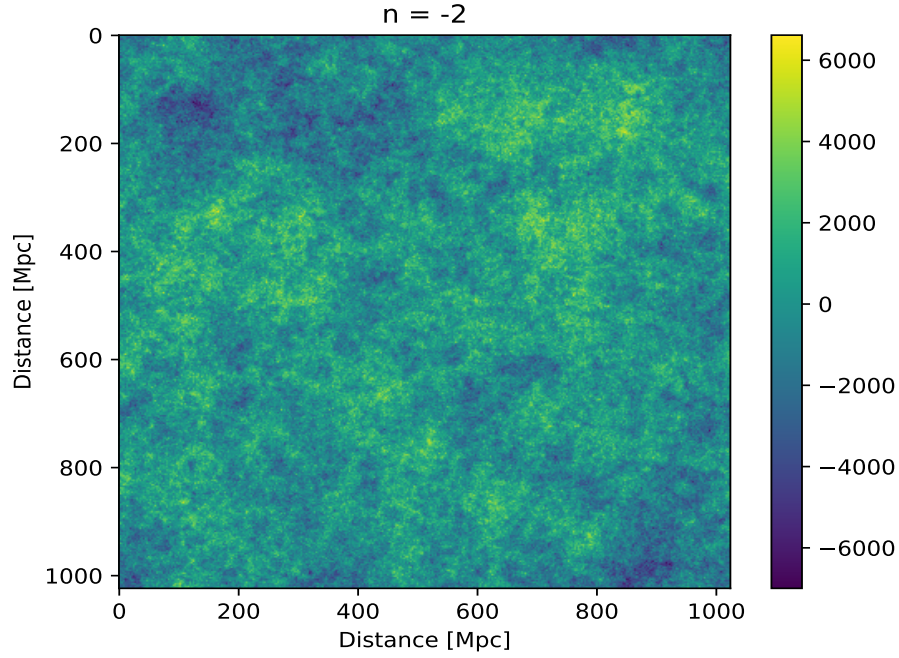

Figure 20: The Gaussian field for $n = -2$ and a minimal physical size of 1 Mpc. The plot seems to be less noisy than the previous plot, which is expected as n is now smaller.
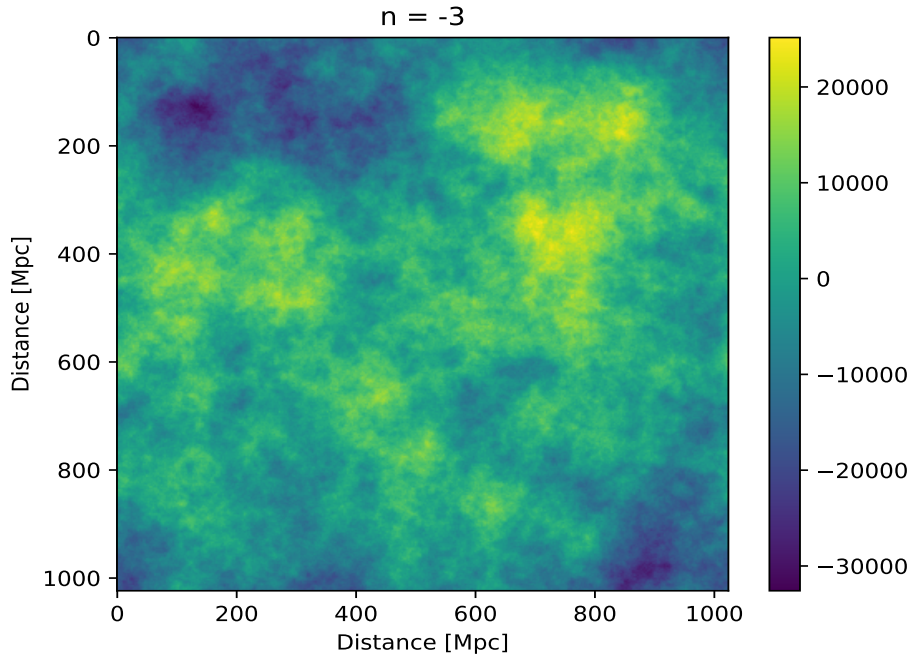
Figure 21: The Gaussian field for $n = -3$ and a minimal physical size of 1 Mpc. High and low density fluctuations are now clearly visible. Notice that the plot looks similar to the plot for $n = -2$. This is a consequence of using the same random uniform variables in the Box-Muller method.

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.

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

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 **??** 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,

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

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

(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,

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

This simplifies to

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

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,

$$D(t) = c_1 t^{2/3} + c_2 t^{-1} \tag{24}$$

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.

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

def main():

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

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

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

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

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

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

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

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

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

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

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


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

    # Current value of the linear growth function.
```

35

```
67       d = values [0]
68       # Current value of the derivative of the linear growth function.
69       u = values [1]
70
71       # The two systems of first order ODE's.
72       first = u
73       second = -(4/(3*t))*u + (2/(3*t**2))*d
74
75       return np.array([first , second])
76
77
78  if __name__ == '__main__':
79       main()
```

./Code/assigment_3.py

Code - Runge-Kutta

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

```
1  import numpy as np
2
3
4  def runge_kutta_54(func , y0, t_start , t_stop , t_step , atol=1e−6, rtol=1e−3):
5       """
6            Perform the 4the order runga_kutta method for first order ODE integration
       .
7       In:
8            param: func — The function describing the differential equation or the
       system
9                            of first order ODEs to integrate. Must return a numpy
       array.
10           param: y0   — The initial conditions.
11           param: t_start — The time to start integration at.
12           param: t_stop — The time to stop integration at.
13           param: t_step — The initial step size to use.
14           param: steps — The step size to use for integration.
15           param: order — The order of the algorithm to use.
16      """
17
18
19      # If true , solving a single ODE, else solving a system.
20      if type(y0) is not np.ndarray:
21          y0 = np.array([y0]) # convert to array
22
23
24      # Array with values to return , for both the integrated
25      # values and the time stemps. The size is increased by a
26      # factor of 2 when needed.
27
28      ret = np.zeros((int((t_stop−t_start)/t_step)+1, len(y0)))
29      time = np.zeros(int((t_stop−t_start)/t_step)+1)
30
31      # Set initial state.
32      ret[0] = y0
33      time[0] = t_start
34
35      # Solve the ODE or the system of ODEs
36      min_update_scale = 0.2
37      max_update_scale = 10
38
39      # Current time at the integration
40      t_now = t_start
41      # Total amount of executed steps
42      steps = 1  # skip zero
43
44      # Current error
45      error = 1.1
46      y_next = 0
47
48      while t_now <= t_stop :
```

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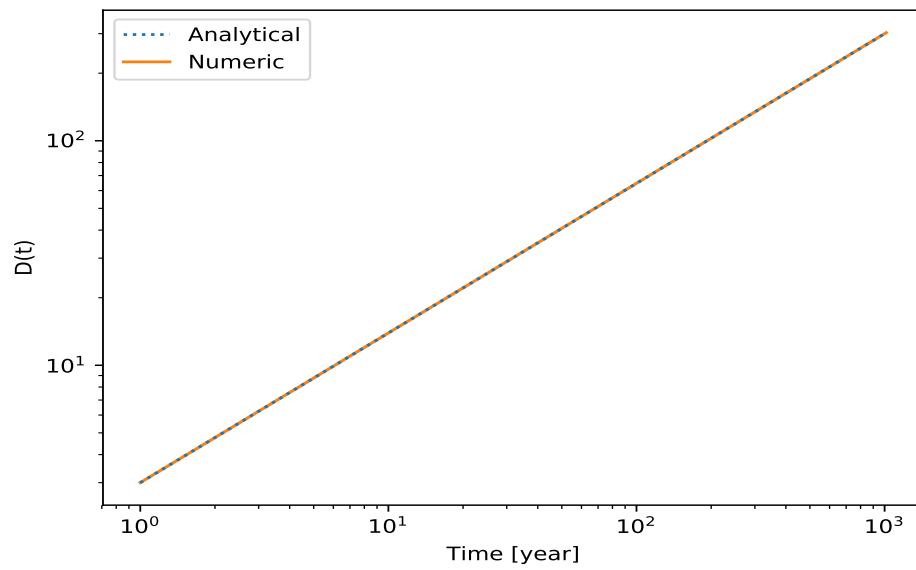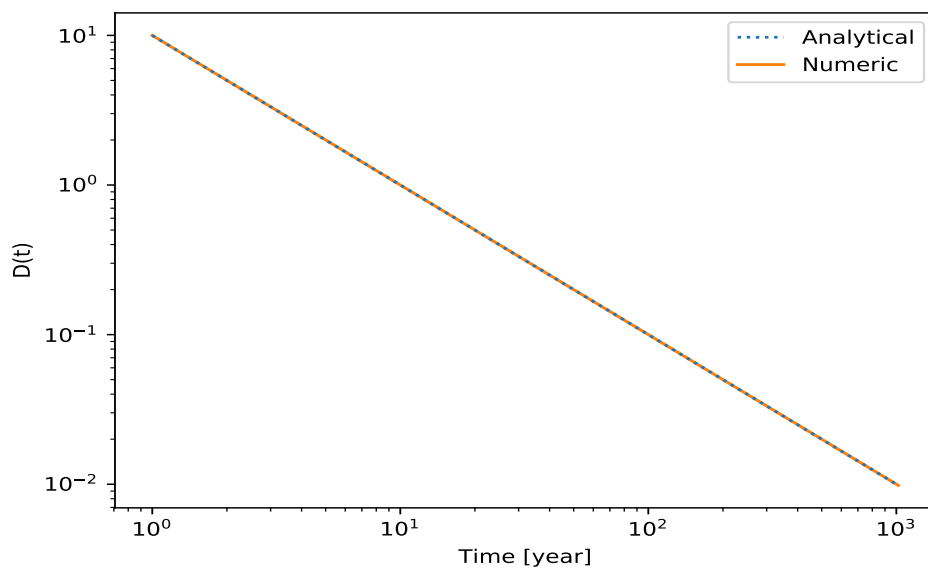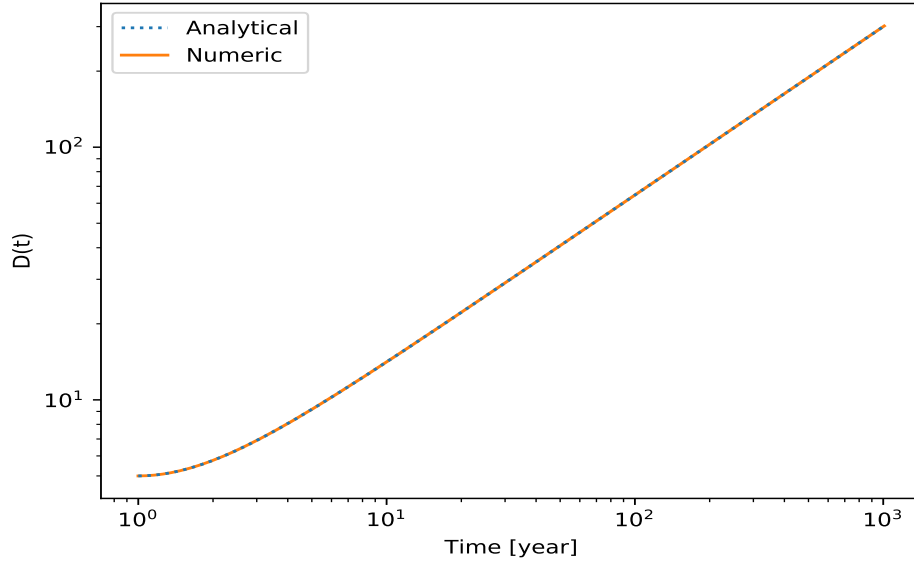
Figure 24: The analytical (blue) and numerical (orange) solution of the ODE with initial conditions $D(1) = 5$, $D'(1) = 0$. The plots show that he numerical solution does not appear to have a visible deviation from the analytical solution for the given time interval.

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,

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

39

Substituting this in equation 30 and simplifying yields,

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

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

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.

```python
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 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.

```python
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.

```python
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.
    """
```

```python
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
```

```
86                    # New precision is smaller-> update
87                    if precision_tmp < current_precision:
88                        current_precision = precision_tmp
89                        ret = results[j-1]
90
91                        # Terminate if requested precision is reached
92                        if current_precision < precision:
93                            return ret
94
95                    # Abort early if the error of the last combined result is worse
96                    # than the previous order by a large amount.
97                    if j == (approximations-combination-1) \
98                        and abs(ret - previous) > 100*current_precision:
99
100                       return ret
101         return ret
```

./Code/mathlib/derivative.py

Text - Output

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

```
1 [4b]  Analytical:    2.800638157126267e-10
2 [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 $\mathbf{S(q)}$. 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 $ic_k k_x$ and $ic_k k_y$. 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).

44

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.

```python
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 * grid_size

    # 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)
```

```python
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         print(idx)
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, alpha=0.7)
101         plt.colorbar()
102         plt.title('a=' + str(a))
103         plt.xlabel('x [Mpc]')
104         plt.ylabel('y [Mpc]')
105         plt.savefig('./Plots/4c/4c={0}.png'.format(idx))
106         plt.close()
107
108     # Plot the momentum and position
109     pos_top_10 = np.array(pos_top_10)
110     momentum_top_10 = np.array(momentum_top_10)
111
112     # Momentum
113     for i in range(0,10):
114         plt.plot(scale_factors, momentum_top_10[:,i],label='particle ' + str(i+1)
    )
115         plt.xlabel('a')
116         plt.ylabel(r'$P_y$')
117     plt.legend()
118     plt.savefig('./Plots/4c_momentum.pdf')
119     plt.close()
120
121     # Position
122     for i in range(0,10):
123         plt.plot(scale_factors, pos_top_10[:,i],label='particle ' + str(i+1))
124         plt.xlabel('a')
125         plt.ylabel(r'y')
126
127     plt.legend()
128     plt.savefig('./Plots/4c_pos.pdf')
129     plt.close()
```

./Code/assigment_4.py

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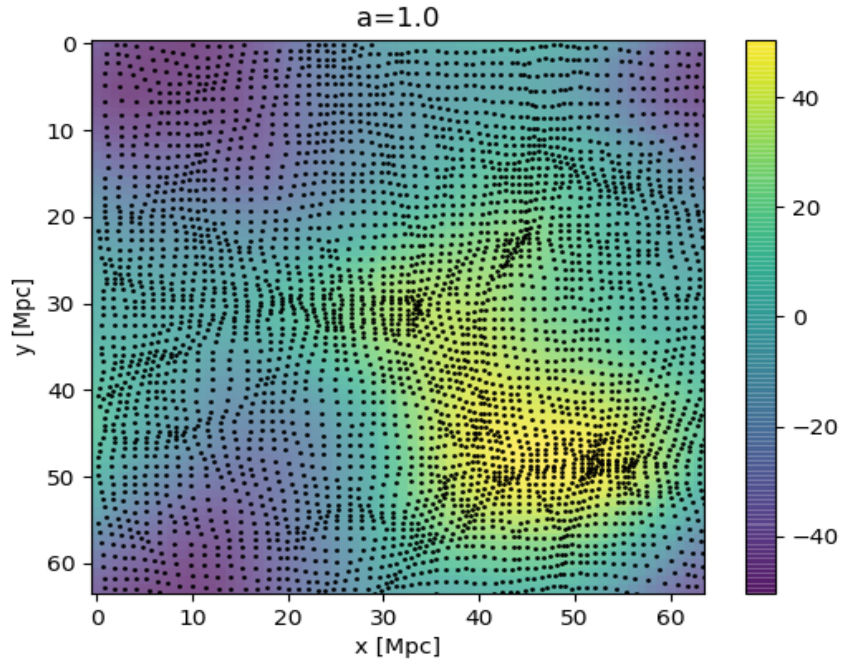
Figure 25: The final plot of the movie. It can clearly be seen that the particles (black dots) moved to the denser region.

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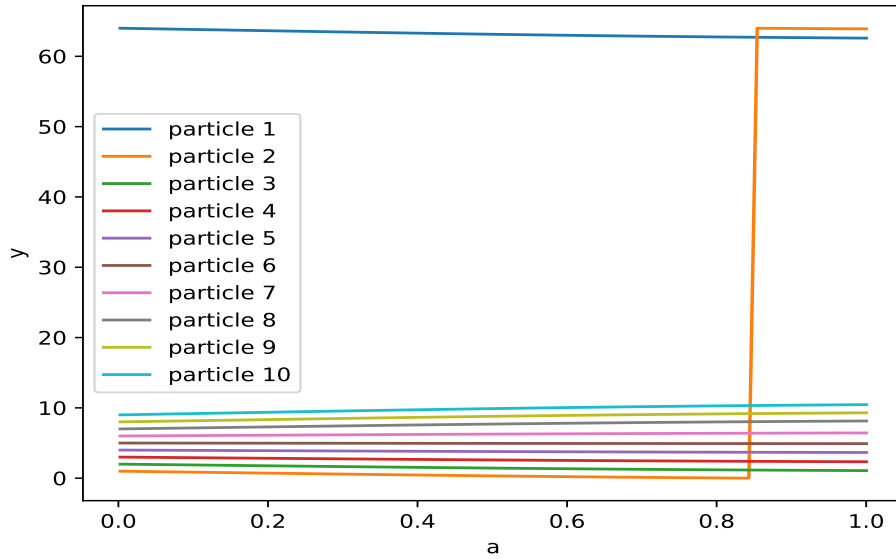
Figure 26: The y-positions of the first 10 particles as function of the scale factor. The plot shows that the particles are slowly moving to a denser region (see figure 25, where the first 10 particles are at the left top). The large jump for particle 2 (orange) at a scale factor of around $a \approx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

47

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

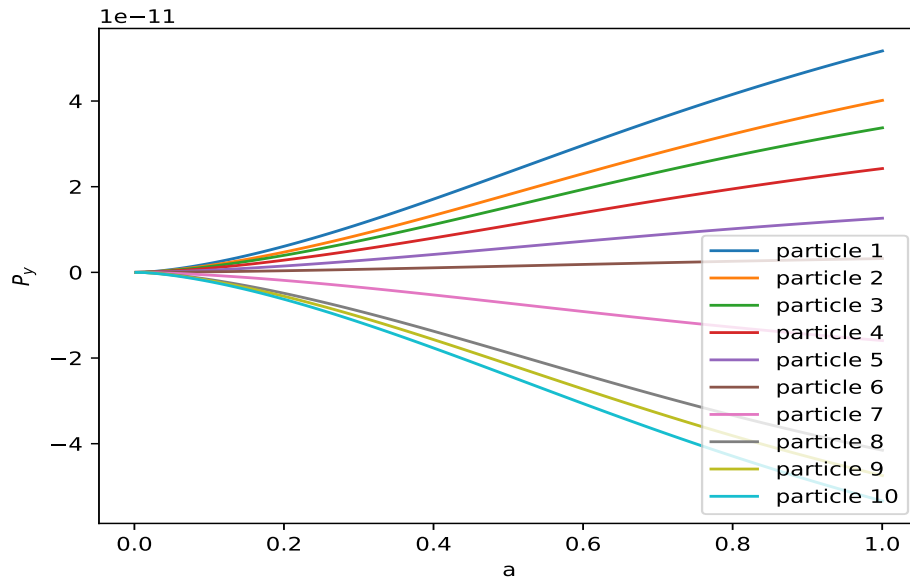Generate initial conditions for a three dimensional box to do a N-body simulation, make initial conditions for 64^3 particles starting at redshift $z = 50$. Besides this make 3 separate movies of a slice of thickness 1/64th of your box at its center, make a slice for $x - y$, $x - z$,$y - z$. Again make 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 **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.

48

Code - Plots:

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

```python
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
```

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

66

67        matrix_z [ int ( grid_size /2) , int ( grid_size /2) ,0] = matrix_z [ int ( grid_size /2) ,
          int ( grid_size /2) ,0]. imag
68        matrix_z [ int ( grid_size /2) , 0 , int ( grid_size /2)] = matrix_z [ int ( grid_size /2)
          ,0 , int ( grid_size /2) ]. imag
69        matrix_z [0 , int ( grid_size /2) , int ( grid_size /2)] = matrix_z [0 , int ( grid_size
          /2) , int ( grid_size /2) ]. imag
70        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

71

72

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

76

77                if ( j == 0 or j == int ( grid_size /2) ) and k > int ( grid_size /2):
78                    matrix_y [ int ( grid_size /2) ,j ,k] *= −1
79                elif   j > int ( grid_size /2):
80                    matrix_y [ int ( grid_size /2) ,j ,k] *= −1

81

82        # special points
83        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
84        matrix_y [ int ( grid_size /2) , int ( grid_size /2) ,0] = matrix_y [ int ( grid_size /2) , int
          ( grid_size /2) ,0]. imag + 0J
85        matrix_y [ int ( grid_size /2) ,0 , int ( grid_size /2)] = matrix_y [ int ( grid_size /2) ,0 ,
          int ( grid_size /2) ]. imag + 0J
86        matrix_y [ int ( grid_size /2) ,0 ,0] = matrix_y [ int ( grid_size /2) ,0 ,0]. imag + 0J

87

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

92

93

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

96

97        #
98        # Begin simulation

99

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

103

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

108

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

111

112           # calculate da
113           da = 0

114

115           if idx != 0:
116               da = scale_factors [ idx ] − scale_factors [ idx −1]
117           else :
118               da = scale_factors [ idx ] − a_min

119

120           # Calculate D(a)
121           # Calculate \dot (D)
122           d = helpers4 . calculate_linear_growth (a)
123           ddot = helpers4 . calculate_linear_growth_dir (a −da/2)

124

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

```
128              pos_z_new = (pos_z + s_z*d) % grid_size
129
130              pos_top_10.append(pos_z_new[0:10,0,0])
131
132              # Update momentum
133              p_x = -(a-da/2)**2 * ddot *s_x
134              p_y = -(a-da/2)**2 * ddot*s_y
135              p_z = -(a-da/2)**2 * ddot*s_y
136
137              momentum_top_10.append(p_z[0:10,0,0])
138
139              # slices
140              z_mask = np.where(abs(pos_z_new - 32) < 0.5)
141              y_mask = np.where(abs(pos_y_new - 32) < 0.5)
142              x_mask = np.where(abs(pos_x_new - 32) < 0.5)
143
144              # Create the plots.
145
146              # z
147              plt.scatter(pos_x_new[z_mask], pos_y_new[z_mask],s=1,c='black')
148              plt.title('a=' + str(a))
149              plt.xlabel('x [Mpc]')
150              plt.ylabel('y [Mpc]')
151              plt.savefig('./Plots/4d/xy/4d_xy={0}.png'.format(idx))
152              plt.close()
153
154              # y
155              plt.scatter(pos_x_new[y_mask], pos_z_new[y_mask],s=1,c='black')
156              plt.title('a=' + str(a))
157              plt.xlabel('x [Mpc]')
158              plt.ylabel('z [Mpc]')
159              plt.savefig('./Plots/4d/xz/4d_xz={0}.png'.format(idx))
160              plt.close()
161
162              # x
163              plt.scatter(pos_y_new[x_mask], pos_z_new[x_mask],s=1,c='black')
164              plt.title('a=' + str(a))
165              plt.xlabel('y [Mpc]')
166              plt.ylabel('z [Mpc]')
167              plt.savefig('./Plots/4d/yz/4d_yz={0}.png'.format(idx))
168              plt.close()
169
170         # Plot the momentum and position
171         pos_top_10 = np.array(pos_top_10)
172         momentum_top_10 = np.array(momentum_top_10)
173
174         # Momentum
175         for i in range(0,10):
176              plt.plot(scale_factors, momentum_top_10[:,i], label = 'particle ' + str(i
         +1))
177              plt.xlabel('a')
178              plt.ylabel(r'$P_z$')
179
180         plt.legend()
181         plt.savefig('./Plots/4d_momentum.pdf')
182         plt.close()
183
184         # Position
185         for i in range(0,10):
186              plt.plot(scale_factors, pos_top_10[:,i], label = 'particle ' + str(i+1))
187              plt.xlabel('a')
188              plt.ylabel('z')
189
190         plt.legend()
191         plt.savefig('./Plots/4d_pos.pdf')
192         plt.close()
```
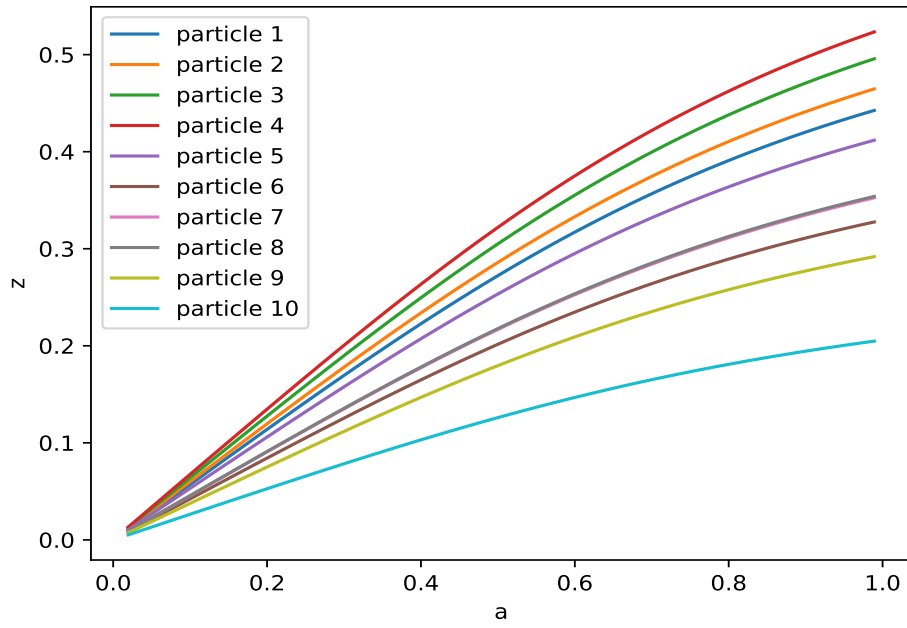
./Code/assigment_4.py

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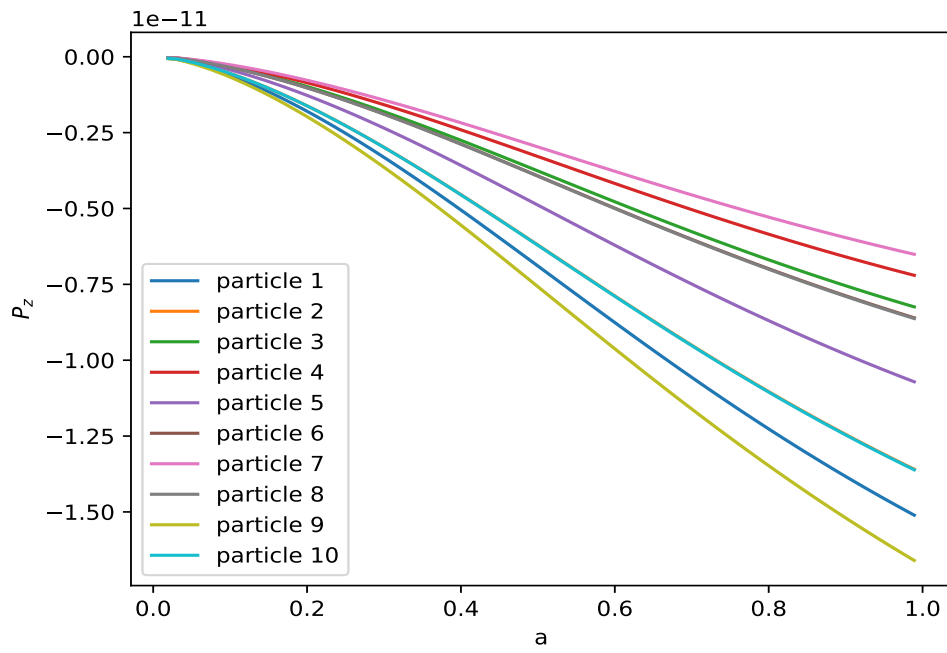

Figure 29: The z-component of the momentum of the first 10 particles against the scale factor.

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.

```python
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)
    a,b = random.gen_normal_uniform(0,1,rand1,rand2)
    return complex(a,-b)*(1/k**2) *0.5*sigma
```

./Code/assigment_4.py

Code - Linear growth

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

```python
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
```

```
16
17      # The prefactor of the integral
18      pre_factor = 0.5*(5*omega_m)*(omega_m*a_max**(-3) + omega_lambda)**(0.5)
19
20      # The function to integrate.
21      func = lambda a: a**(-3) /(omega_m * (1/(a**3))+omega_lambda)**(3/2)
22
23      return pre_factor*integrate.romberg(func, 1e-7, a_max, 15)
24
25 def calculate_linear_growth_dir(a, omega_m = 0.3, omega_lambda = 0.7):
26      """
27          Calculate the derivative of the linear growth factor
28      In:
29          param: a -- The scale factor for which to calculate it.
30          param: omega_m -- The fraction of matter in the universe.
31          param: omega_lambda -- The fraction of dark energy in the universe
32      Out:
33          return: The derivative of the scale factor for the given parameters
34      """
35      # Hubble constant
36      H0 = 7.16e-11
37
38      # The function to integrate, which appears in the second term.
39      func = lambda a: a**(-3) /(omega_m * (1/(a**3))+omega_lambda)**(3/2)
40
41
42      # The terms in the expression.
43      pre_factor = (5*omega_m*H0)/(2*a**2)
44      first_bracked_term = 1/(omega_m*a**(-3)+omega_lambda)**0.5
45      second_bracked_term = (-3*omega_m*integrate.romberg(func, 1e-7, a, 15))/(2*a)
46
47      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.

```
1  import numpy as np
2
3  def gen_wavenumbers(size, min_distance):
4      """
5          Generate the shifted wavenumbers
6          for the discrete fourier transform.
7      In:
8          param: size -- The size of the matrix.
9          param: min_distance -- The distance of a cell/ the sample spacing.
10     Out:
11          return: An array with shifted wave numbers.
12     """
13     # Array to return.
14     ret = np.zeros(size)
15
16     # Positive values
17     ret[0:int(size/2)+1] = np.arange(0,int(size/2)+1)
18
19     if size % 2 == 0: # even
20         ret[int(size/2):] = -np.arange(int(size/2),0,-1)
21     else: # odd
22         ret[int(size/2)+1:] = -np.arange(int(size/2),0,-1)
23
24
25     return (ret/(size*min_distance))*2*np.pi
26
27
28 def generate_matrix_2D(size, min_distance, func, random_numbers, power):
29     """
30          Generate a 2D matrix with complex numbers in
31          shifted fourier coordinates using the power spectrum
```

```python
     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
```

```python
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
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.
```

```python
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
             .
173
174      size = tensor.shape[0]
175
176      # Inner matrix
177      for row in range(1, size):
178          for column in range(1, size):
179              for depth in range(1, size):
180                  tensor[row, column, depth] = complex(tensor[size-row, size-column,
     size-depth].real, -tensor[size-row, size-column, size-depth].imag)
181
182      # Outside top
183      for depth in range(1, int(size/2)+1):
184
185          tensor[0,0,depth] = complex(tensor[0,0,size-depth].real, -tensor[0,0,size
     -depth].imag)
186          tensor[0,depth,0] = complex(tensor[0,size-depth,0].real, -tensor[0,size-
     depth,0].imag)
187          tensor[depth,0,0] = complex(tensor[size-depth,0,0].real, -tensor[size-
     depth,0,0].imag)
188
189
190          # outside inner
191          for column in range(1, size):
192              tensor[depth, column,0] = complex(tensor[size-depth, size-column,0].
     real, -tensor[size-depth, size-column,0].imag)
193              tensor[depth,0,column] = complex(tensor[size-depth,0, size-column].
     real, -tensor[size-depth,0, size-column].imag)
194              tensor[0, depth, column] = complex(tensor[0,size-depth,size-column].
     real, - tensor[0,size-depth,size-column].imag)
195
196
197      if size % 2 == 0:
198          # Finally fix niquist
199          for i in range(0,size):
200              for j in range(0,size):
201                  tensor[int(size/2),i,j] = tensor[int(size/2),i,j].real + 0J
202                  tensor[i,j,int(size/2)] = tensor[i,j,int(size/2)].real + 0J
203                  tensor[i,int(size/2),j] = tensor[i,int(size/2),j].real + 0J
204
205      return tensor
```

./Code/mathlib/misc.py

Code - Romberg

The code for romberg integration

```python
1  import numpy as np
2
3  def trapezoid(function, a, b, num_trap):
4      """
5          Perform trapezoid integration
6
7      In:
8          param: function -- The function to integrate.
9          param: a -- The value to start the integration at.
10          param: b -- The value to integrate the function to.
11          param: num_trap -- The number of trapezoids to use.
12
13      Out:
14          return: An approximation of the integral from a to b of
15                  the function 'function'
16      """
17
18      # The step size for the integration range.
19      dx = (b-a)/num_trap
20
21      # Find trapezoid points.
```

```python
22    x_i = np.arange(a,b+dx,dx)
23
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 to assigning the full mass of a particle to its 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 to the nearest gird point, but it might also be partially assigned to multiple grid points.

The mass is for the given particle shape assigned to the gird points by abusing the fact that a cast to an integer always result in down a cast (i.e 15.7 casted to an integer gives 15). The indices of a grid point to which a particle has to assign its mass can, by abusing the down cast, be found by adding 0.5 to the position and then down casting the result to an integer. To include circular boundary conditions the modulo with 16 (grid size) is taken.

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 73

Code - slices

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

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

    # Relevant imports are:
    # (1) import numpy as np
    # (2) import mathlib.mass as mass
    # (3) import matplotlib.pyplot as plt

    # 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]:
        slice = mass_grid[:,:,z]
        #slice[slice == 0] = np.nan

        plt.imshow(slice)
        plt.colorbar()
        plt.savefig('./Plots/5a_slice_{0}.pdf'.format(z))
        plt.close()
```

./Code/assigment_5.py

Plots - Slices

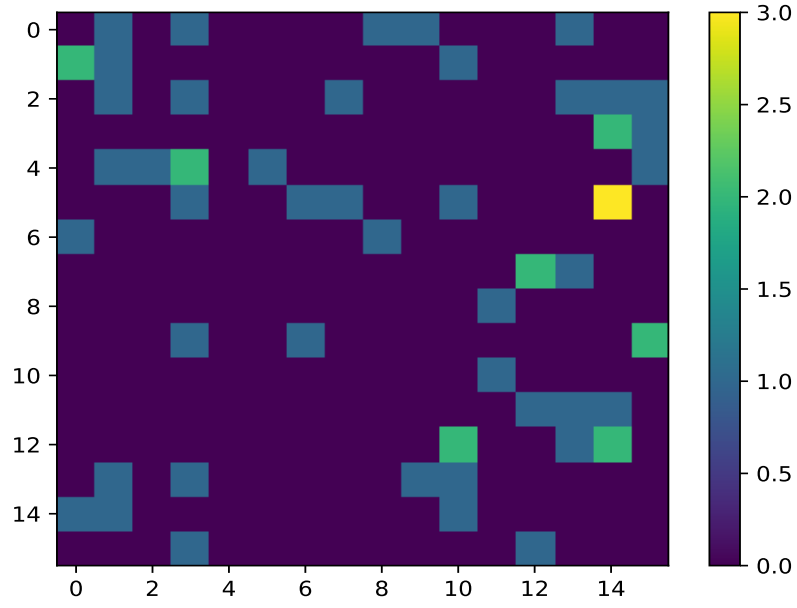Figure 30: The x-y slice of the created mass grid for $z = 4$. The color indicates the assigned mass in terms of particle mass.
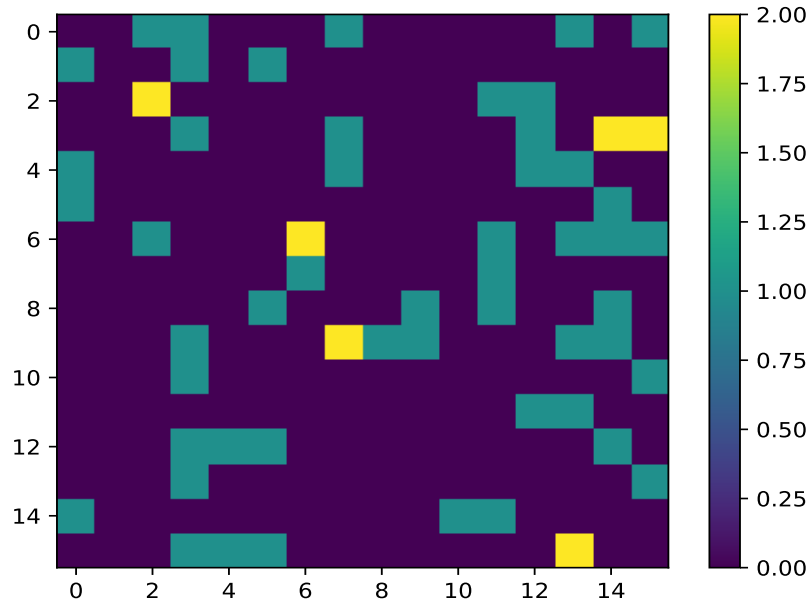

Figure 31: The x-y slice of the created mass grid for $z = 9$. The color indicates the assigned mass in terms of particle mass. Notice that the range of the colorbar is different from the first and last plot.
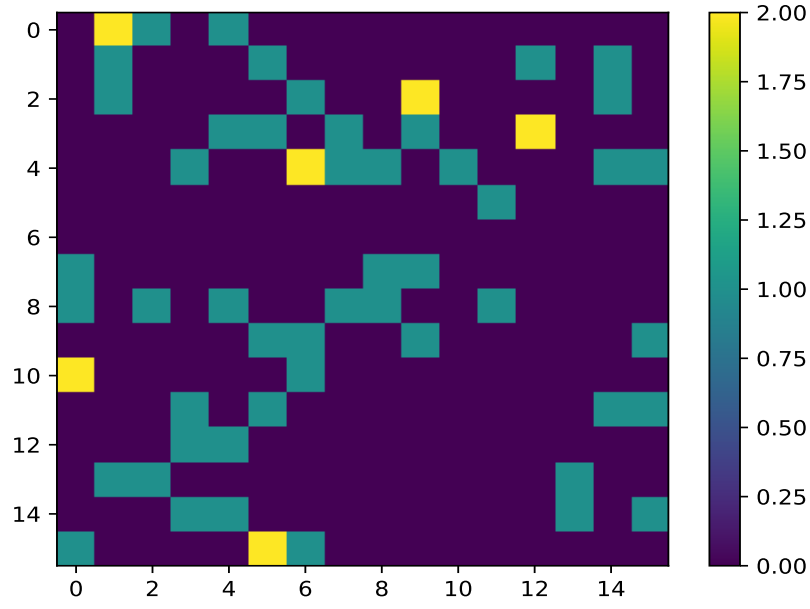
Figure 32: The x-y slice of the created mass grid for $z = 11$. The color indicates the assigned mass in terms of particle mass. Notice that the range of the colorbar is different than the first and last plot.
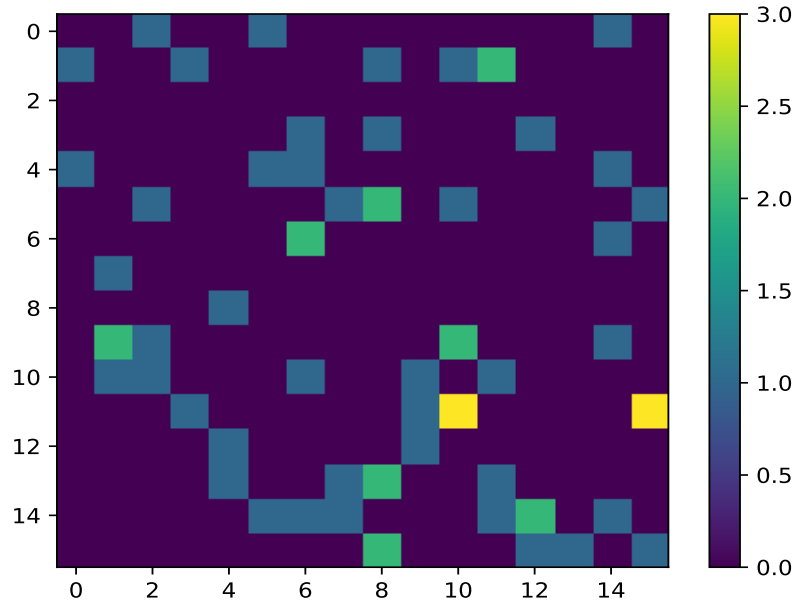

Figure 33: The x-y slice of the created mass grid for $z = 14$. The color indicates the assigned mass in terms of particle mass.

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 plot and the plot can be found below.

Code - Plots

The code that creates the plots for cell 0 and cell 4.

```python
def assigment_5b():
    """
        Execute assignment 5.b
    """

    # Relevant imports are:
    # (1) import numpy as np
    # (2) import mathlib.mass as mass
    # (3) import matplotlib.pyplot as plt

    # 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_NGP(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')
    plt.ylabel('Mass in terms of particle mass')
    plt.legend()
    plt.savefig('./Plots/5b_cell.pdf')
    plt.close()
```

./Code/assigment_5.py

Plots - Cell
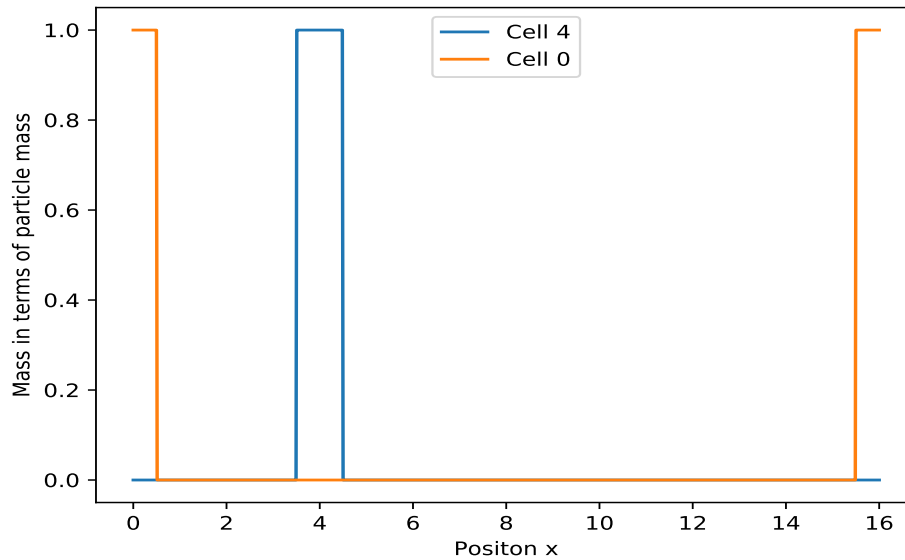

Figure 34: The mass assigned for a particle moving from x = 0 to x = 16. The orange and red line respectively indicate the mass assigned to cells 0 and 4 as function of the position of the particle. The plot is created for a mass grid of size 16 with circular boundary conditions.

Question 5.c)

Problem

Right now 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) method 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 73). The code that creates the plots and the plots them self can be found below.

Code - Plots

The code that creates the same plots as in 5.a and 5.b for a 3D grid with the CIC method.

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

    # Relevant imports are:
    # (1) import numpy as np
    # (2) import mathlib.mass as mass
    # (3) import matplotlib.pyplot as plt


    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.close()

    # 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')
    plt.ylabel('Mass in terms of particle mass')
    plt.legend()
    plt.savefig('./Plots/5c_cell.pdf')
    plt.close()
```

./Code/assigment_5.py

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. Notice that the range of the colorbar deviates from the other three figures below.
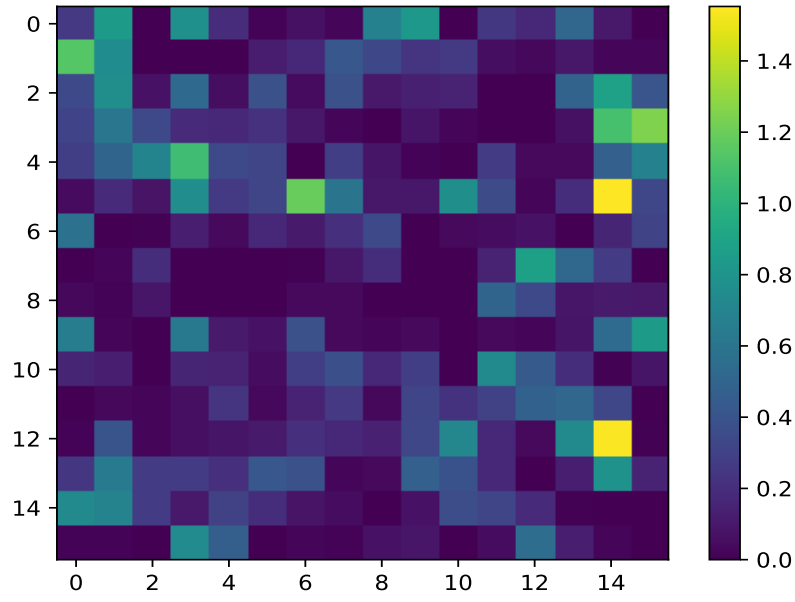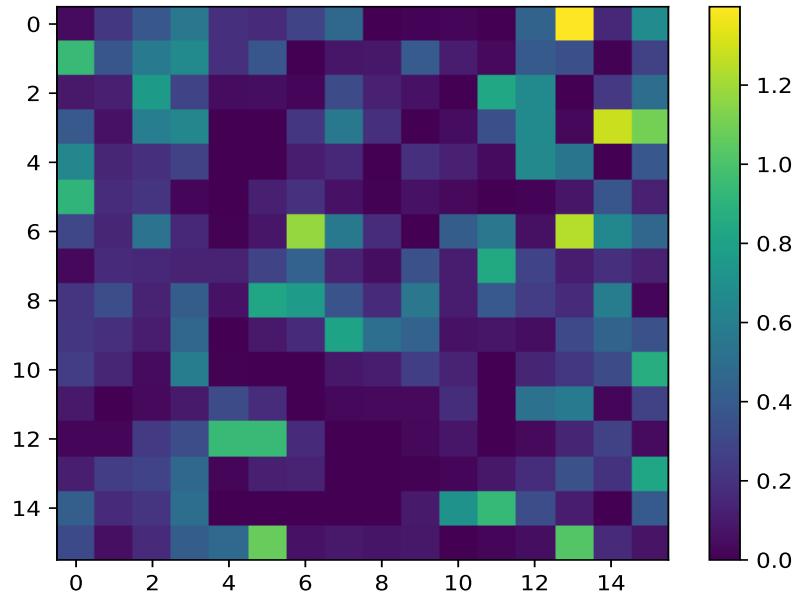

Figure 36: The x-y slice of the created mass grid with the CIC method for $z = 9$. The color indicates the assignment mass in terms of particle mass. Notice that the scalre of the colorbar deviates from the other three figures.

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. Notice that the colorbar deviates from the other three figures.

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. Notice that the scale of the colorbar deviates from the other three figures.
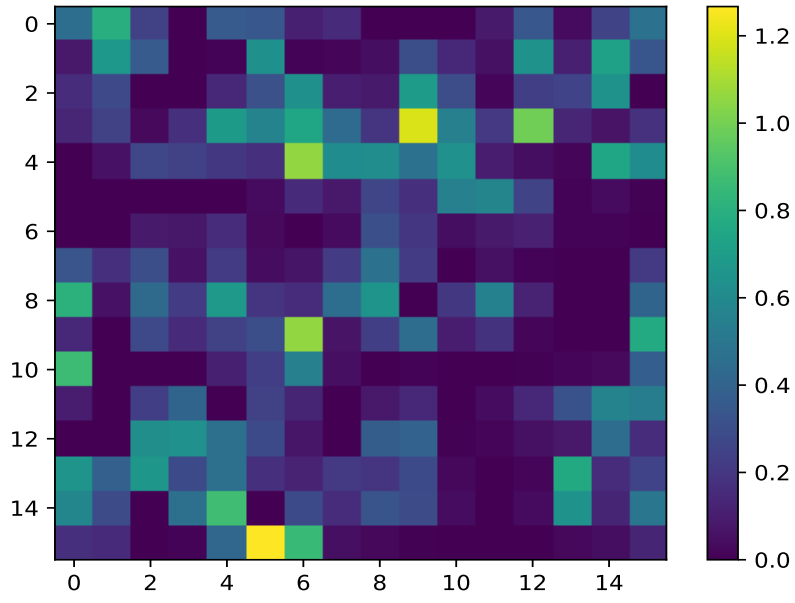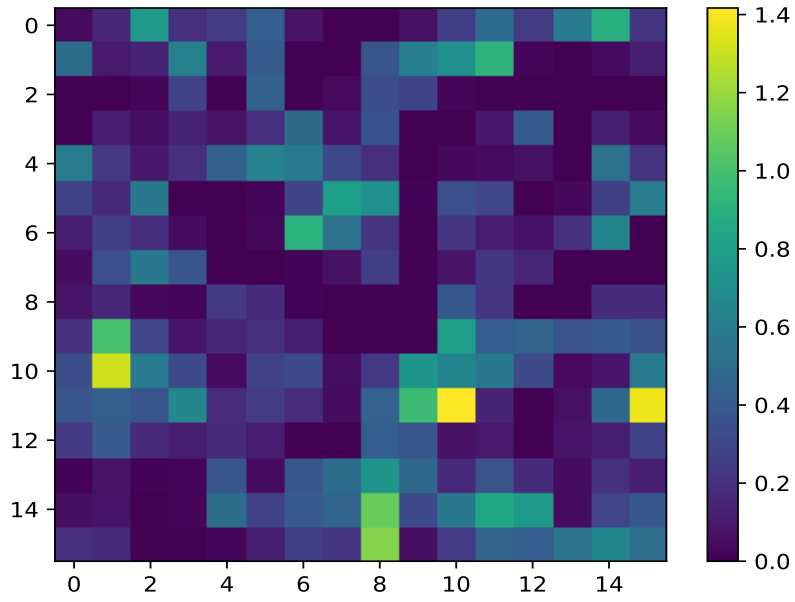
Plots - Cells

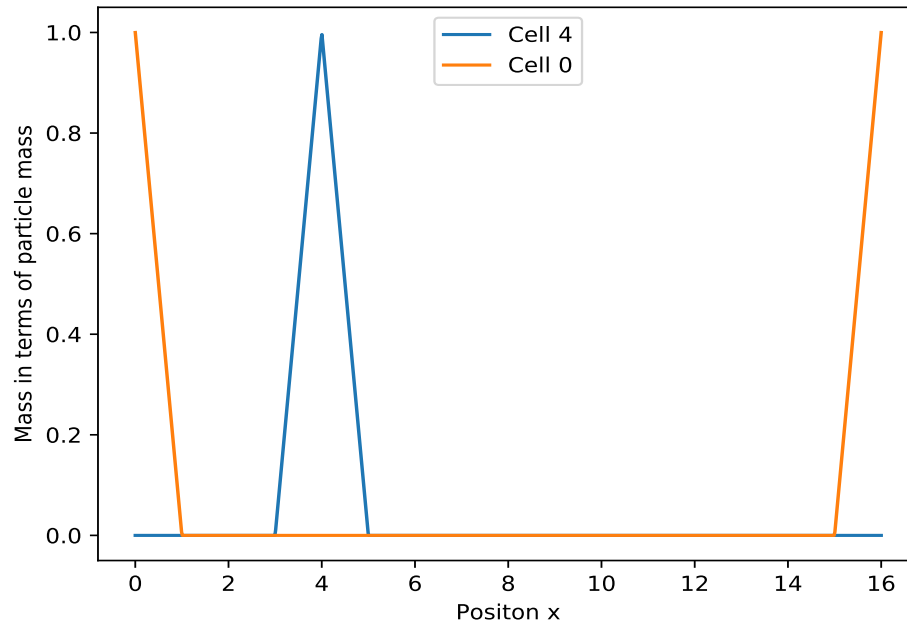Figure 39: The mass assigned for a particle moving from x = 0 to x = 16 for the cloud in the cell method. The orange and red line respectively indicate the mass assigned to cells 0 and 4 as function of the position of the particle. The plot is created for a 3D mass grid of size 16 with circular boundary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

$$\mathcal{F}(\cos(x)] \propto 0.5(\delta(f - x) + \delta(f + x)) \tag{41}$$

The code for the FFT can be found on page 75. 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 - Plot

The code that creates the plot for the comparison of the self written FFT with the numpy implementation and the analytical solution. The plot can be found below this segment of code.

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

    # Relevant imports are:
    # (1) import numpy as np
    # (2) import matplotlib.pyplot as plt
    # (3) import mathlib.misc as misc
    # (3) import mathlib.fourier as fourier

    # 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()
```
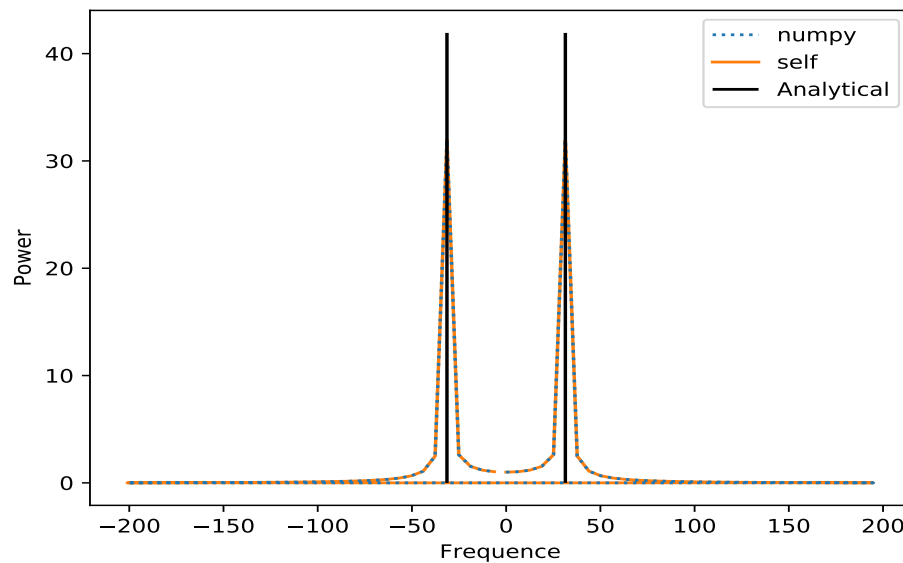
./Code/assigment_5.py

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.

Question 5.e)

Problem

Generalize your own FFT algorithm to 2 and 3 dimensions and make a plot of the FFT of a 2D function (no bivariate Gaussian) and compare it with the analytical FFT of your FFT. Also make a plot of the FFT of a 3D multivariate Gaussian function, plot 3 slices centered at the center for the 3 different slice options x-y, x-z,y-z.

Solution

The code for the 2D and 3D implementation makes use of the code for the 1D implementation. In 2D the FFT is first taken over the first axis of the input matrix and then over the second axis. In 3D the FFT2 is first taken over the first axis and then the FFT is taken over the third axis. The code is not generalized to N-dimensions, as the question did not ask to write your own swapaxis algorithm.

The chosen function to 2D fourier transform is $\cos(x + y)$. The analytical result consists of two 2D delta functions. The code that makes the plots for the 2D FFT and the 3D FFT can be found below, the implementation its self can be found on page 75. The plots can be found below the code.

Code - Plot

The code that creates the plots.

```python
def assigment_5e():
    """
        Execute assigment 5e
    """

    # Relevant imports are:
    # (1) import numpy as np
    # (2) import matplotlib.pyplot as plt
    # (3) import mathlib.misc as misc
    # (3) import mathlib.fourier as fourier

    # The function to 2D fourier transform
    func_2d = lambda x,y: np.cos(x+y)
    x,y = np.meshgrid(range(0, 64), range(0,64))
    x = np.array(x, dtype=complex)
    y = np.array(y, dtype=complex)

    _, ax = plt.subplots(nrows=1,ncols=2)

    # Self written FFT2
    ax[0].imshow(abs(fourier.fft2(func_2d(x,y))))
    ax[0].set_title("Self written FFT2")
    ax[0].set_xlabel('index')
    ax[0].set_ylabel('index')

    # Numpy FFT
    ax[1].imshow(abs(np.fft.fft2(func_2d(x,y))))
    ax[1].set_title("Numpy version FFT2")
    ax[1].set_xlabel('index')
    ax[1].set_ylabel('index')

    plt.savefig('./Plots/5e_2d_fft.pdf',bbox_inches='tight')
    plt.close()

    # 3D
    # The multivariate gaussian.
    # I am fully aware that this line looks ugly in the report.
    multivariate_sigma = lambda x,y,z, sigma:  (1.0/(sigma**2*2*np.pi)**(3/2)) * np.exp(-(((x**2)/2) + ((y**2)/2) + ((z**2)/2))/sigma**2)

    # The multivariate to plot, sigma = 0.5
    multivariate = lambda x,y,z: multivariate_sigma(x,y,z, 0.5)

    # Plot the multivariate gaussian
    N = 64
    values = np.arange(0,N)
    values = np.array(values,dtype=complex)
    x,y,z = np.meshgrid(values,values,values)

    matrix = multivariate(x,y,z)
    matrix_fft = fourier.fft3(matrix)

    # y-z slice
    plt.imshow(abs(matrix_fft[int(N/2)]))
    plt.xlabel('y')
    plt.ylabel('z')
    plt.savefig('./Plots/5e_gaussian_yz.pdf')
    plt.close()

    # x-z slice
    plt.imshow(abs(matrix_fft[:,int(N/2),:]))
    plt.xlabel('x')
    plt.ylabel('z')
    plt.savefig('./Plots/5e_gaussian_xz.pdf')
    plt.close()

    # y-x slice
    plt.imshow(abs(matrix_fft[:,:,int(N/2)]))
```

```
68        plt.xlabel('x')
69        plt.ylabel('y')
70        plt.savefig('./Plots/5e_gaussian_xy.pdf')
```

./Code/assigment_5.py

Plot - 2D
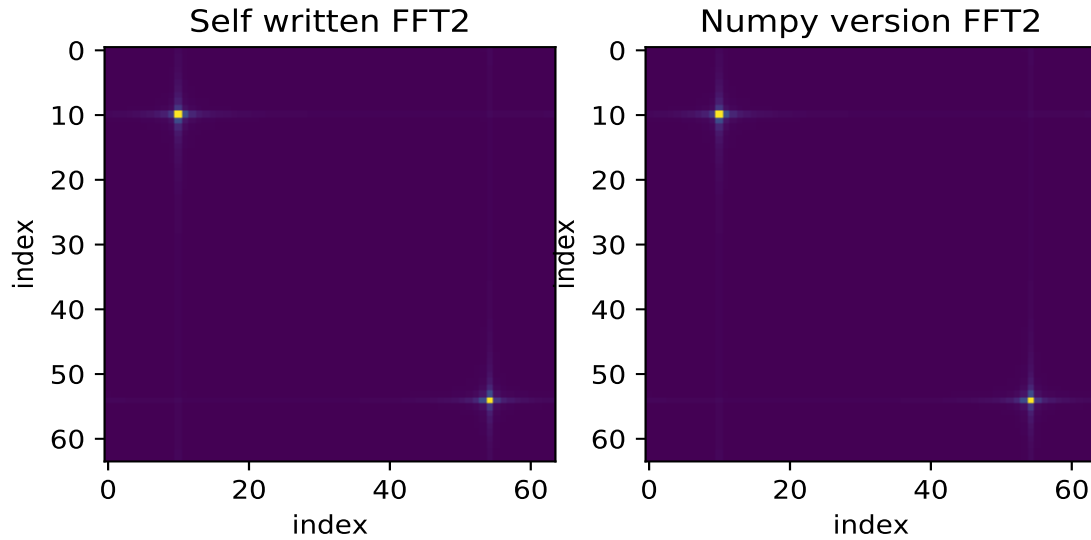
Figure 41: The 2D fourier transformation for the function $\cos(x + y)$. Left, the own implementation of the FFT2 (left) and right the numpy implementation. The results show that the self written version appears to be equal to the numpy version. For the cosine two delta peaks are expected to arise which can also be seen. The peaks are not infinite sharp as that would require an infinite amount of samples.

Plot - 3D
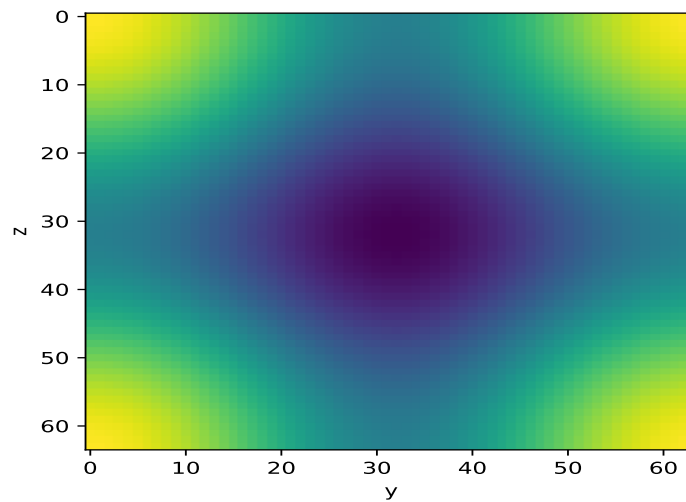
Figure 42: The yz slice around the center of the 3D Fourier transform of a multivariate gaussian with $\mu = 0$ and $\sigma = 0.5$. The slice should be perfectly symmetric and the result shows that this is indeed the case.
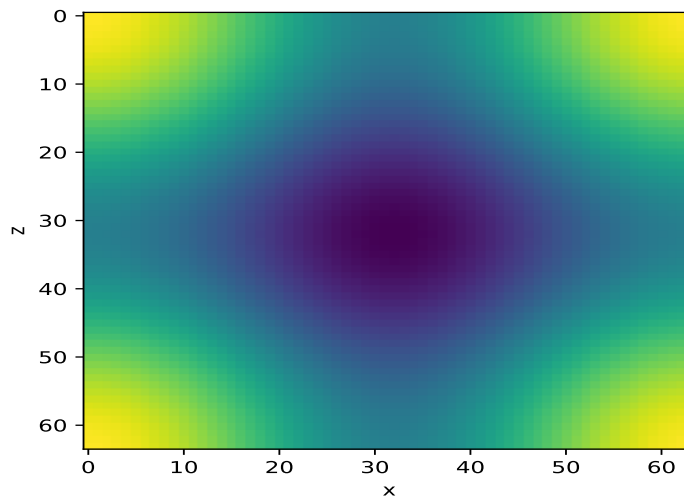
Figure 43: The xz slice around the center of the 3D Fourier transform of a multivariate gaussian with $\mu = 0$ and $\sigma = 0.5$. The slice should be perfectly symmetric and the result shows that this is indeed the case.
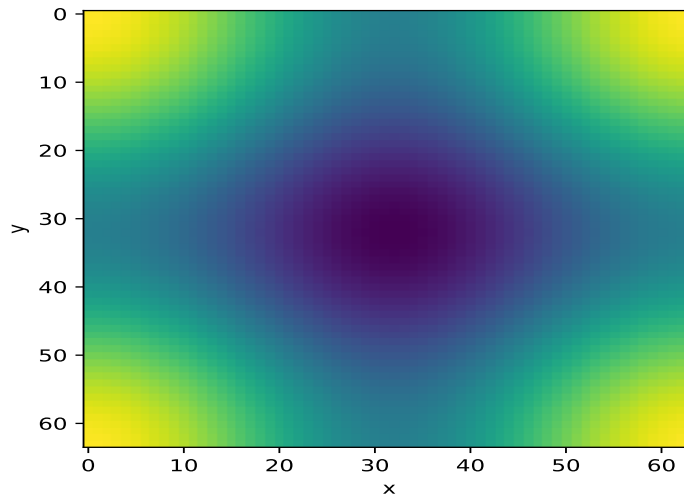
Figure 44: The xy slice around the center of the 3D Fourier transform of a multivariate gaussian with $\mu = 0$ and $\sigma = 0.5$. The slice should be perfectly symmetric and the result shows that this is indeed the case.

Question 5 - Summary

Summary

The current sub-section contains the summary of the code used for assignment 5. This includes the part of the main file that executes the subquestions, the relevant imports, the functions to assign the mass for the NGP and CIC method and, the code for the Fourier transformations.

Code - Assignment

The code with the main function that executes the sub questions and the relevant imports.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import mathlib.mass as mass
4  import mathlib.fourier as fourier
5  import mathlib.misc as misc
6
```

```python
7   def main ( ) :
8       assigment_5a ( )
9       assigment_5b ( )
10      assigment_5c ( )
11      assigment_5d ( )
12      assigment_5e ( )
```

<div align="center">./Code/assigment_5.py</div>

Code - Mass

The code that assigns the masses for the NGP and CIC method

```python
1   import numpy as np
2
3
4   def _assign_mass_NGP ( positions ) :
5       """
6           Create a mass grid of size 16x16 with
7           the Nearest Grid Point shape .
8       In :
9           param : positions — The positions of the N particles as
10                              an matrix of size 2xN.
11      Out :
12          return : A matrix representing the mass grid .
13      """
14
15      # Create the empty grid
16      mass_grid = np.zeros ((16,16,16))
17
18      # Add 0.5 and cast to an integer . The positions are now
19      # the indices in the grid .
20      positions = np.array ( positions + 0.5 , dtype=np.int32 )
21
22      # Assign the masses by looping over the positions
23      for i in range (0 , len ( positions [ 0 ] ) ) :
24          # Mod 16 for cirulair boundaries .
25          mass_grid [ positions [ 0 ] [ i ] % 16 , positions [ 1 ] [ i ] % 16 , positions [ 2 ] [ i ] %
        16] += 1
26
27      # Return the mass .
28      return mass_grid
29
30
31  def _assign_mass_CIC ( positions ) :
32      """
33          Create a mass grid of size 16x16x16 with
34          the Nearest Grid Point shape .
35      In :
36          param : positions — The positions of the N particles as
37                              an matrix of size 3xN.
38      Out :
39          return : A matrix representing the mass grid .
40      """
41
42      mass_grid = np.zeros ((16,16,16))
43
44      for i in range (0 , len ( positions [ 0 ] ) ) :
45
46          # The cube below represents the cube made by the 8 grid points that
47          # enclose a particle . Next we assume that the sides of are side
48          #     LTU————————RTY
49          #     /|           /|
50          #    / |          / |
51          # LTD————————RTD|
52          # |   |         |   |
53          # |  LBU--------|RBU
54          # |  /          |  /
55          # |/            |/
56          # LBD ————————RBD
```

<div align="center">73</div>

```python
57
58          # Let $x_{cell},y_{cell},z_{cell}$ be the position of
59          # the left top bottom (LTB) edge of the Grid Cube that
60          # encloses the particle. Let x,y,z be the position of the particle.
61          # The difference between the particle positon and the LTB edge
62          # is then given by,
63          # dx = x - x_{cell}
64          # dy = y - y_{cell}
65          # dz = z - z_{cell}
66          # The above quenties can be used to calculate the volumne fractions
67          # that needs to be assigned to the 8 nearest grid points of the cube
68          # that the particle represents.
69          # Working this out (by for example a drawing) gives.
70          #
71          # LTD = (1 - dx)*(1 - dy)*(1-dz)
72          # RTD = (1 - dx)*dy*(1 - dz)
73          # LTU = (1 - dx)*(1 - dy)*dz
74          # RTU = (1 - dx)*dy*dz
75          # LBD = dx*(1-dy)*dz
76          # RBD = dx*dy*(1-dz)
77          # LBU = dx*(1-dy)*dz
78          # RBU = dx*dy*dz
79
80          # Get the indices of the grid point at the left bottom of
81          # the cel in which the particle is
82          center_x = np.int32(positions[0][i]) % 16
83          center_y = np.int32(positions[1][i]) % 16
84          center_z = np.int32(positions[2][i]) % 16
85
86          # Helper variables to calculate the
87          # volume fractions. Mode 16 for circulair boundary conditions
88          dx = positions[0][i] % 16 - center_x
89          dy = positions[1][i] % 16 - center_y
90          dz = positions[2][i] % 16 - center_z
91
92          frac_dx = 1 - dx
93          frac_dy = 1 - dy
94          frac_dz = 1 - dz
95
96          # Calculate the volume fractions, assume unit particle mass
97
98          # Left top down
99          mass_grid[center_x][center_y][center_z] += frac_dx*frac_dy*frac_dz
100         # Right top down
101         mass_grid[center_x][(center_y+1) % 16][center_z] += frac_dx*dy*frac_dz
102         # Left top up
103         mass_grid[center_x][center_y][(center_z+1) % 16] += frac_dx*frac_dy*dz
104         # Right top up
105         mass_grid[center_x][(center_y+1) % 16][(center_z+1) % 16] += frac_dx*dy*
     dz
106
107         #Left bottom down
108         mass_grid[(center_x+1) % 16][center_y][center_z] += dx*frac_dy*frac_dz
109         # Right bottom down
110         mass_grid[(center_x+1) % 16][(center_y+1) % 16][center_z] += dx*dy*
     frac_dz
111         # Left bottom up
112         mass_grid[(center_x+1) % 16][center_y][(center_z+1) % 16] += dx*frac_dy*
     dz
113         # Right bottom up
114         mass_grid[(center_x+1) % 16][(center_y+1) % 16][(center_z+1) % 16] += dx*
     dy*dz
115
116
117     return mass_grid
```

./Code/mathlib/mass.py

Code - Fourier

The code containing the fourier transformations.

```python
import numpy as np

def fft(data):
    """
        Perform the DFT with the cooley-tukey algorithm.
    In:
        param: data -- The array to fourier transform, must be of length 2**N
    Out:
        return: The fft of the data with the zeroth frequence as the first elment
        .
    """

    # Get the length of the data
    n = len(data)

    # The array to return. By using this
    # array the input array wont't require
    # a reverse bitshift. This was mainly done
    # to prevent modifying the input array.
    ret = np.zeros(n, dtype=complex)

    # The FFT of 1 element is the element self.
    if n == 1:
        ret[0] = data[0]
        return ret

    # Perform the FFT on the even indices and
    # odd indices.
    even = fft(data[::2])
    odd = fft(data[1::2])

    # Calculate the recurrent split for the
    # values of k. Notice here that
    # there is a minus sign, which the result
    # of symmetry that is exploited.
    for k in range(int(n/2)):

        # Multiply with the complex exponent
        odd[k] *= np.exp(-1J*2*np.pi*k/n)

        # Calculate the FFT for the discritized k (or f) values
        ret[k] = even[k] + odd[k]
        # symmetry exploid
        ret[k+int(n/2)] = even[k] - odd[k]

    # return the fourier transform, don't use a normalization
    return ret


def fft2(data):
    """
        Calculate the 2D fourier transformation.
    In:
        param: data -- The data to fourier transform
    Out:

    """

    for i in range(data.shape[0]):
        #kx_0,ky_0,  kx_0,ky_1,  kx_0,ky_2,  kx_0,ky_3
        data[i] = fft(data[i])

    # now perform it over the rows (x direction)
    for i in range(data.shape[0]):
        #kx_0,ky_0,  kx_1,ky_0,  kx_2,ky_0,  (store in column)
        data[:,i] = fft(data[:,i])

    return data
```

```python
def fft3(data):
    """
        Perform the 3 dimensionale fourier transform
    In:
        param: -- The data to fourier transform
    Out:
        return: The 3D fourier transform of the data.
    """

    # First fourier transform the planes
    for i in range(data.shape[0]):
        data[i,:,:] = fft2(data[i,:,:])

    # Finally fourier transform the last axis.
    for i in range(data.shape[2]):
        for j in range(data.shape[1]):
            data[:,i,j] = fft(data[:,i,j])

    return data
```

./Code/mathlib/fourier.py

6 - Classifying x-ray bursts

Question 6

Problem

Use logistic regression to make a model of the data, suing a binary classification for short or long GRBs. Explain which properties your model uses to prediced whether a GRB is short or long, and how you handle missing data. Plot a histogram of the class (0 or 1) of GRB and overplot the actual class based on the value of T_{90}.

Solution

The data is before training the model pre-processed as follows. The first stap consists of removing all rows with missing T_{90} (this only consists of two rows). Next, the columns with logarithmic values are taken to the power of an exponent. Finally the values of -1 in the non logarithmic columns and the values of e^{-1} in the logarithmic columns are replaced with 0. The reason that the columns with logarithmic values where first taken to the exponent is because directly replacing -1.0 in such a column with 0 is equivalent to creating your own data. (i.e if $\ln(M_*/M_\odot) = 0$, then we would have created data that implies that $M_* = M_\odot$).

The columns that where chosen for training consist of the redshift, the star formation rate and the log of the mass ratio. These columns where selected based on a scatter plot made with seaborn in which all columns where plotted against each other. The code for this is not included[5].

With these columns an accuracy of 78 % was obtained (see page 80). This value is unfortunately the base line that needs to be broken, as in the final dataset 78% of all entries are long GRBS. The model is thus returning '1' for each entry. Attempts to use different combinations of columns always resulted in the same accuracy. The code that creates the histogram and the code for the logistic regression can be found below.

Code - Plot

The code that creates the plots.

```
import numpy as np
import mathlib.LogisticRegression as lg
import matplotlib.pyplot as plt

# Load the data
# Column 2 = red_shift, 3 = T90
data = np.genfromtxt("GRBs.txt")

# Remove the first 2 columns. These contain 'GRB/XRF' and the number.
data = data[:,2:]
data = np.array(data)
# The columns are now:
# 0 = red_shift, 1 = T90, 2 = log(M^*/M_{sun})
# 3 = SFR, 4 = log(Z/Z_{sun}), 5 = SSFR, 6 = AV

# Remove all rows without T90
data = data[data[:,1] != -1]

# Replace all missing data with zero's, but be carefull about
# the logarithms. To correct the logarithms we take the
# column of a logirhtm to the power of  e and then put e^{-1} to zero.

data[:,2] = np.exp(data[:,2])
data[:,4] = np.exp(data[:,4])

# Correct for missing data
for i in range(0,7):
    if i == 2 or i == 4:
        mask = (data[:,i] == np.exp(-1))
        data[:,i][mask] = 0
    else:
        mask = (data[:,i] == -1)
```

---

[5]I did not had enough time to recreate the plot without seaborn before the deadline.

```
33          data [: , i ] [ mask ] = 0
34
35
36 # The chosen columns are:
37 #   − redshift
38 #   − log (M^*/M_{sun})
39 #   − SFR
40 #
41 # These are used to create the final data for training the model.
42 # The final train data consits of the chosen columns plus
43 # one additional column that is all 1 (used for bias)
44
45 # The columns in the dataset to select.
46 columns = [0 ,2 ,3]
47
48 train_data = np.ones((data.shape[0], len(columns)+1))
49 train_data [: ,[0 ,1 ,2]] = data [: , columns]
50 train_labels = data [: ,1] > 10
51
52 # Train the model.
53 logistic_reg = lg.LogisticRegression(len(columns),1e−3,1e−5)
54 logistic_reg.train(train_data, train_labels)
55
56 # Predictions
57 predictions = logistic_reg.predict(train_data)
58 predictions = np.array(predictions, dtype=int)
59
60 #Create the histogram
61 bins = np.arange(0 ,1.1 ,0.05)
62
63 plt.hist(predictions, alpha=0.9, bins=bins, label='Model')
64 plt.hist(np.array(train_labels, dtype=int), bins=bins, label='True')
65 plt.legend()
66 plt.ylabel('Counts')
67 plt.savefig('./Plots/6a_hist.pdf')
68
69 # Print the accuracy
70 print('[6] Accuracy: ', 1 − np.sum(predictions − train_labels)/len(train_labels))
```

./Code/assigment_6.py

Code - Logistic regression

The code for logistic regression.

```
1 import numpy as np
2
3 class LogisticRegression(object):
4
5     def __init__(self, variables, learn_rate = 0.01, error = 1e−2):
6         """
7             Create a new instance of a LigisticRegressor
8         """
9         #weights, also add bias
10        self._weights = np.zeros(variables + 1)
11        self._learn_rate = learn_rate
12        self._error = error
13
14
15     def train(self, data, labels):
16         """
17             Train the current logistic regresion model
18         """
19
20         # Current error
21         error = 1e4
22
23         # Max iterations
24         max_iter = 1e4
25         # Current iteration
26         curr_iter = 0
```

```python
27
28          # Cost of the previous train iteration
29          old_cost = 1e4
30
31          # While we haven't accuired the required error
32          # or haven't reahced hte maximum number of iterations
33          # keep training.
34
35          while error > self._error and curr_iter < max_iter:
36
37              # Cost and gradient
38              cost = 0
39              gradient = np.zeros(self._weights.shape[0])
40
41              # Calculate the cost and the gradient
42              for idx in range(0, data.shape[0]):
43                  y_est = self._sigmoid(np.dot(self._weights, data[idx]))
44                  y = labels[idx]
45                  term = -(y*np.log(y_est) + (1-y)*np.log(1-y_est))
46
47                  cost += term
48                  gradient += term*data[idx]
49
50              cost = cost/data.shape[0]
51              gradient = gradient/data.shape[0]
52
53              # Update the weights
54              self._weights += self._learn_rate*gradient
55
56              # Update the error
57              error = abs(cost - old_cost)
58              old_cost = cost
59              curr_iter += 1
60
61
62      def predict(self, data):
63          """
64              Predict the labels for the given data.
65          In:
66              param: -- A matrix where each row represent a row in the data
67                        set for which a label needs to be predicted. The last
68                        value of each row must be a 1.
69          Out:
70              return: An array with predicted labels.
71          """
72          labels = list()
73          for row in data:
74              labels.append(self._sigmoid(np.dot(self._weights, row))> 0.5)
75
76          return np.array(labels)
77
78
79      def _sigmoid(self, x):
80          """
81              Evaluate the sigmoid activation function.
82          In:
83              param: x -- The point(s) to evaluate it for.
84          Out:
85              return: The sigmoid activation function evaluated at the
86                        given point(s).
87          """
88          # 1e-7 is to correct for exponents that are zero.
89          # This was needed as some values of columns where taken to the exponent.
90
91          return 1/(1+np.exp(-x) + 1e-7)
```

./Code/mathlib/LogisticRegression.py

Text - Output

The accuracy obtained by the model.

```
1  [6]  Accuracy:    0.7811158798283262
```
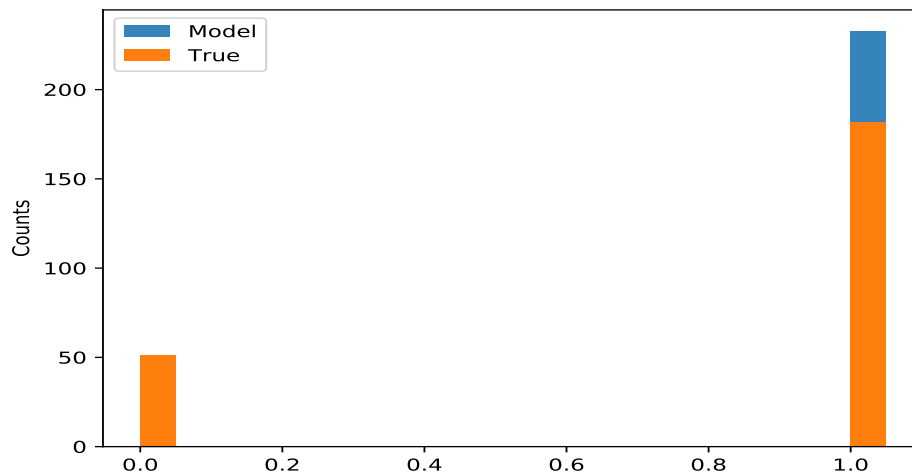
./Output/assigment6_out.txt

Plots- Histogram

Figure 45: The created histogram for the prediction made by the model and the true labels. It can be seen here that the model always returns '1'.

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 (this can be a parent of a leaf, or a parent of a 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 quadtree and adds the particles to the tree. The second file contains the code that contains the quadtree 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
```

81

```python
19                                  splitting a node.
20          """
21
22          # Create the root quad.
23          self._root = Quad(left_bottom, size, max_boddies, None)
24
25      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
```

```python
             axis = plt.gca()
             # An list that contains rectangle objects (matplotlib.patches.Rectangle)
             # for each of the quads in the tree.
             rectangles = list()

             # Fill the list with rectangles by recursively calling the
             # children of the root. If a quad is a leaf it will furtheremore
             # axis.scatter. This is to add the boddies of that leaf
             # to the plot. The reason that a quad not directly adds its own
             # rectangle to the axis (axis.add_patches) is to save time.
             self._root._plot(axis, rectangles)

             # Add all the rectangles at once to save time.
             axis.add_collection(collections.PatchCollection(rectangles,
                                  match_original=True))

             # Create and save the plot
             plt.xlim(self._root._left_bottom[0],
                      self._root._left_bottom[1] + self._root._size)
             plt.ylim(self._root._left_bottom[1],
                      self._root._left_bottom[1] + self._root._size)
             plt.xlabel('x')
             plt.ylabel('y')
             plt.savefig('./Plots/7_quadtree.pdf')
             plt.figure()

class Quad(object):

    def __init__(self, left_bottom, size, max_boddies, parent = None):
        """
            Create an instance of a quad in a quadtree.
        In:
            param: left_bottom -- The coordinates of the leftbottom point of
                                  the quad.
            param: size -- The size of the quad.
            param: max_boddies -- The maximum amount of boddies to add before
                                  splitting this quad.
            param: parent -- The parent quad if any.
        """

        # Geometric properties of the current quad
        self._left_bottom = left_bottom
        self._size = size
        self._halve_size = size/2

        # 'Facts' about the quad at the moment of initalization
        self._max_boddies = max_boddies
        self._leaf = True
        self._parent = parent

        # An array containing child quads (if any) and
        # the boddies in this quad (if it is a leaf).

        # The child quads are named:
        # Left Top (index 0), Left Bottom (index 1)
        # Right top (index 2), Right bottom (index 3)
        self._child_quads = None
        self._boddies = list()

        # The multipole moment of the current quad.
        self._moment = 0


    def add_boddy(self, boddy):
        """
            Add a boddy to the current quad or to one of its
            child quads.
        In:
            param: boddy -- The boddy to add. Should be an array
                            in which the first for elemnts respectivelty
                            correspond with the x-position, y-position,
```

```python
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],
```

```python
                    self._left_bottom[1] + self._halve_size]
        lb = self._left_bottom
        rt = [self._left_bottom[0] + self._halve_size,
                    self._left_bottom[1] + self._halve_size]
        rb = [self._left_bottom[0] + self._halve_size,
                    self._left_bottom[1]]

        # Add the quads
        self._child_quads.append(Quad(lt, self._halve_size,
                                    self._max_boddies, self))
        self._child_quads.append(Quad(lb, self._halve_size,
                                    self._max_boddies, self))
        self._child_quads.append(Quad(rt, self._halve_size,
                                    self._max_boddies, self))
        self._child_quads.append(Quad(rb, self._halve_size,
                                    self._max_boddies, self))

        # Current quad is not a leaf anymore
        self._leaf = False

        # Add the boddies of the current quad to the new child quads.
        for boddy in self._boddies:
            self.add_boddy(boddy)

        # Empty the boddies that where hold by t his quad.
        self._boddies = None

    def _update_moment(self, mass = 0):
        """
            Update the multipole moment of the current quad
        In:
            praram: mass -- The mass to update the multipole moment with if the
                            current quad is a leaf.
        """

        # Current quad is a leaf, thus
        # add the mass (zeroth order multipole moment)
        if self._leaf:
            self._moment += mass
        # Current quad is not a leaf.
        else:
            # Reset the multipole moment and recalculate
            # it by looping of the child quads.
            self._moment = 0

            for quad in self._child_quads:
                self._moment += quad._moment

        # Make sure the parents of the current quad update the multipole moment.
        if self._parent != None:
            self._parent._update_moment()


    def _plot(self, axis, quads):
        """
            Plot the current quad.
        In:
            param: axis -- The axis object to plot this quad at.
            param: qaudss -- A list of quads at which the current
                            quad should add a rectangle which represents
                            the current quad.
        """

        # Create the rectangle to plot of the current quad and add it to the list
        rect = patches.Rectangle(self._left_bottom, self._size,
                                    self._size, fill=False)
        quads.append(rect)

        # If the current quad is a leaf plot the boddies.
        if self._leaf:
            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.

```
1  0.12500000186264515
2  0.21250000316649675
3  0.28750000428408384
4  7.162500106729567
5  7.162500106729567
6  15.000000223517418
```

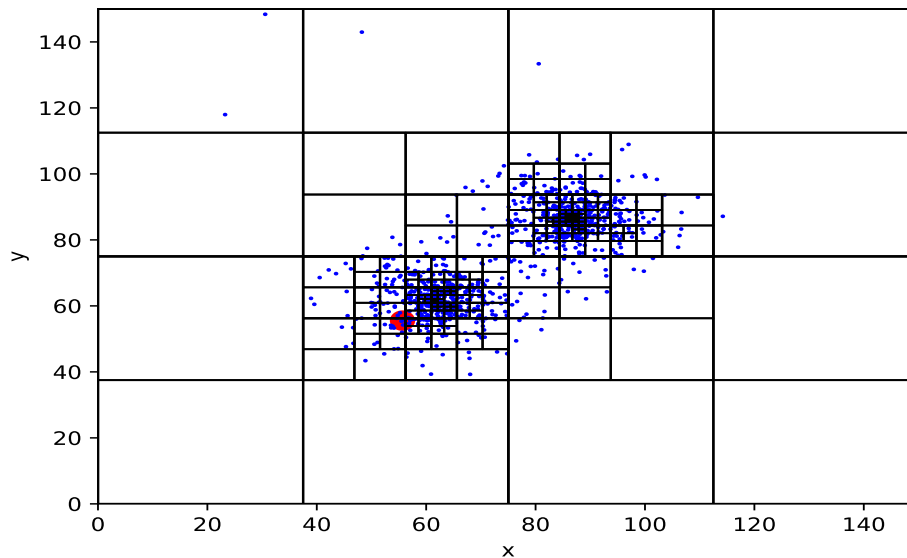./Output/assigment7_out.txt

Output - Plot

Figure 46: 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.