Introduction to Image Analysis

Compulsory Assignment 2

Ugnius Alekna

November 2023

1 Introduction

The second compulsory assignment concerns the practical exercises we've covered in weeks 6 to 8 of the course "Introduction to Image Analysis", taking place at the Faculty of Mathematics and Informatics, Vilnius University, Lithuania during Autumn 2023. This written report serves as a reflective summary of the material covered in the course so far as well as a practical implementation of it. It builds on top of the foundational knowledge gained within the report for the first compulsory assignment by addressing more advanced topics of Image Analysis. Adhering to the first compulsory assignment, Python is chosen as the programming language, ensuring consistency across all three assignments. The report includes a brief explanation of the theoretical aspects behind each relevant topic, details on how these theoretical concepts were put into practice, an overview of the practical challenges encountered, and an analysis of how these challenges affected the implementation. Additionally, the report is supported by illustrations that demonstrate the outcomes of the exercises, providing proof of the concepts in action without the explicit need for running the code. However, each task includes instructions on how to replicate the execution of those examples on an Ubuntu Linux system. The report, the implemented code, and the dataset used for this assignment will be submitted. This dataset includes specific data provided in `ImgSetCa2.zip` archive and additional hand-picked TIFF files from the previous weekly exercises. All of the used resources can be found under "Files" section in the "General" channel of the MIF-IA Teams Channel.

2 Preface

In the first compulsory assignment, we focused on image processing in the spatial domain, where we manipulated pixel intensity values directly in order to enhance images, apply filters, and detect edges. Transitioning to the second assignment, we now expand our toolkit by employing Fourier analysis. This powerful approach enables us to transform our images into the frequency domain, giving us insights into the various frequency components that make up an image. By understanding and applying Fourier analysis, we can perform more complex operations that are not easily handled in the spatial domain, such as frequency filtering and noise removal.

2.1 Fourier Analysis

Fourier analysis is a fundamental methodology for understanding both theory and applications of various types of Fourier transforms in digital image processing. Historically, Fourier analysis arose as a way to deal with periodic functions in order to simplify the study of heat transfer. It was *Joseph Fourier* [1] who first developed a method for representing real functions (with a period

[1] Jean-Baptiste Joseph Fourier (1768-1830) - a French mathematician and physicist.

of L) as a linear combination of fundamental L-periodic basis functions, namely sines and cosines in terms of the complex exponential. Fourier stated, that f, periodic in $[-\frac{L}{2}, \frac{L}{2}]$ (under some conditions[2]), can be expressed as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n(f) e^{i\frac{2\pi}{L}nt}, \tag{1}$$

where the coefficients are described as $c_n(f) = \frac{1}{L}\int_{-L/2}^{L/2} f(t)e^{-i\frac{2\pi}{L}nt}dt$. The representation is now called the *Fourier series* and is a cornerstone of the Fourier analysis. Fourier series can also be applied to functions that are not necessarily periodic. This can be achieved by considering them over a finite interval and extending them periodically beyond that interval. The integrals, defining the coefficients can then be calculated over that interval.

The bridge to the Fourier transform is built upon the concept of the Fourier series for non-periodic functions. As the interval of observation grows, the series adapts to represent the function over larger and larger intervals. In the limit, as the period L approaches infinity, the discrete frequencies of the Fourier series become a *continuous* variable, and we transition from a sum to an integral. This results in the *Fourier transform*, a tool, that breaks down functions into their continuous frequency components over the entire real number line. Unlike the Fourier series, which represents a function with a discrete set of sinusoidal waves, the Fourier transform represents a function with a continuous spectrum of frequencies. The Fourier transform of a function f is given by

$$F(\mu) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t)e^{-i2\pi\mu t}dt, \tag{2}$$

where μ represents the frequency. This continuous-time Fourier transform is particularly useful for analyzing non-periodic signals and has great implications in the fields of signal processing and image analysis. It allows us to analyze the frequency content of signals that are not confined to repetitive patterns, thus broadening the horizon of Fourier analysis.

The continuous Fourier transform is well-suited for analytical and theoretical work, however, we encounter a need for its computational counterpart when dealing with digital signals and images. Therefore, we use the *Discrete Fourier Transform* (DFT), which is specifically designed for the analysis of digital signals. Each two-dimensional signal can be thought of as a signal with two spatial dimensions, and the DFT extends to two dimensions as well. The DFT converts the spatial representation of an image into a frequency representation by mapping the pixel intensity values into a space where each point represents a particular frequency contained in the spatial domain.

[2]The Fourier series of a function f exist if the Dirichlet conditions are met:

- The integral of the absolute value of the function f over the period L must be finite;
- In one period, $f(t)$ must have a finite number of minima and maxima;
- In one period, $f(t)$ must have a finite number of discontinuities, and each one must be finite.

For an image of size $M \times N$, the DFT is given by

$$F(u,v) = \frac{1}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-i2\pi(u\frac{x}{M}+v\frac{y}{N})}, \tag{3}$$

where $F(u,v)$ is the DFT of the image $f(x,y)$ and a pair (u,v) correspond to the discrete frequency components. The DFT however, operates under the assumption of periodicity due to its design for discrete and finite data. When applied to digital images, the DFT interprets them as if they were periodically extended, seamlessly connecting each edge to its opposite. This characteristic rises from the DFT's roots, as it is essentially a discretized version of the Fourier series, rather than being directly based on the continuous Fourier transform. The assumption of periodicity can lead to edge-related artifacts in the frequency domain representation and techniques, like image padding, are often employed to mitigate them.

For computational efficiency, especially with large images, we use the *Fast Fourier Transform* (FFT), an algorithm that reduces the complexity of computing the DFT from $\mathcal{O}((MN)^2)$ to $\mathcal{O}(MN\log(MN)$ for an image of size $M \times N$. The FFT is the fundamental principle of digital image processing as it allows for rapid computation of the DFT, enabling real-time image analysis. This efficient and powerful tool will be used extensively throughout the course of this assignment.

3 Assignment Tasks

3.1 Fourier Transform

Now, given a brief explanation of the Fourier analysis, it is time to see it work in action. We will implement the Fourier transform on a grayscale image using Python's '`numpy.fft`' module. This process involves several critical steps, and each one of these steps will be backed by theory to explain why it's needed.

3.1.1 Reading the image

The initial step involves loading the image as a `numpy` array, with the image being the size of $M \times N$. This can be achieved using `libtiff` Python library that enables users to read and write TIFF files. The reader can refer to the first compulsory assignment, which includes a comprehensive explanation and utilization of the mentioned library. The implementation of a function used for TIFF file reading can be accessed in the `image_io.py` Python file, that is in `code/exercise_1/functions`. This step translates the image into a numerical format that can then be manipulated.

3.1.2 Padding the image

The next step is to pad the image to a size of $2M \times 2N$ using zero intensity values. This step is a preparation for the convolution in the frequency domain (we will get into what it is in just a bit). When filters are applied in the frequency domain, they are typically designed to have the same size as the image itself. Since the DFT interprets functions as if they were periodically extended, this must be taken into account. The convolution of two periodic functions results in a periodic function as well. The periods in the resulting convolution are close enough to interfere with each other, leading to a wraparound error. This can be avoided by padding the function (and the filter) with zeros by extending the original dimensions to a size $P \times Q$, where $P \geq 2M - 1$ and $Q \geq 2N - 1$ must hold. In general, DFT algorithms run faster when applied to arrays of even size, so the smallest even integers, namely $P = 2M$ and $Q = 2N$, are selected. This is implemented in Python by initializing an empty array (array filled with zeros) of size $2M \times 2N$ and then assigning the intensity values of the input image to the top left quadrant of this array:

```
def pad_image(image):
    height, width = image.shape
    padded_image = np.zeros((2 * height, 2 * width))
    padded_image[0:height, 0:width] = image
    return padded_image
```

This implementation (along with a few others that will be seen in this section) can be found in the '`image_manipulations.py`' file, located in `code/exercise_1/functions` directory.

3.1.3 Shifting the image

Before applying the DFT, each pixel's intensity value is multiplied by $(-1)^{x+y}$. This operation, known as 'shifting for periodicity', centers the low-frequency components in the frequency domain. The translation in the Fourier domain $F(u - u_0, v - v_0)$ can be obtained by multiplying the image $f(x, y)$ by a complex exponential $e^{i2\pi(u_0 x/M + v_0 y/N)}$ (which is a corollary of (3)). Thus, if we want to shift the data so that $F(0, 0)$ is at $(M/2, N/2)$, we must let $(u_0, v_0) = (M/2, N/2)$, leading to $e^{i2\pi(M/2 \cdot x/M + N/2 \cdot y/N)} = e^{i\pi(x+y)}$, which is the same as $(-1)^{x+y}$. This can be realized either by looping over each pixel (x, y) and directly multiplying its intensity value with $(-1)^{x+y}$, or by using a method `numpy.indices(dimensions)`, which returns two arrays representing the indices. The latter approach is computationally less expensive, so we will opt for it:

```
def shift_image(image):
    height, width = image.shape
    y_indices, x_indices = np.indices((height, width))
    image *= (-1) ** (y_indices + x_indices)
    return image
```

Without this step, the low-frequency components would be at the corners of the frequency spectrum, making it harder to analyze and interpret.

3.1.4 Performing the DFT

The Discrete Fourier Transform is then applied, resulting in a complex array. The DFT decomposes an image into its component frequencies. This step is where the transition from the spatial domain to the frequency domain occurs. The complex array can be expressed in polar form:

$$F(u,v) = \Re[F(u,v)] + i \cdot \Im[F(u,v)] = |F(u,v)| \cdot e^{i\phi(u,v)}, \tag{4}$$

where the magnitude

$$|F(u,v)| = \sqrt{\Re[F(u,v)]^2 + \Im[F(u,v)]^2} \tag{5}$$

is called the *amplitude (Fourier) spectrum*, and the angle

$$\phi(u,v) = arctan(\frac{\Re[F(u,v)]}{\Im[F(u,v)]}) \tag{6}$$

is the *phase spectrum*. The amplitude component represents the magnitude of each frequency in the image, whereas the phase component encodes the position of these frequencies in the original image. The implementation in Python is quite straightforward with the help of `numpy.fft.fft2` function, which performs the transform, and the `numpy.abs` and `numpy.angle` functions, that easily extract the amplitude and phase components of the transformed data. Please note that `norm='ortho'` argument is passed to the `numpy.fft.fft2` function, specifying the DFT scaling factor. This scales both, the DFT and the inverse DFT by a symmetrical factor of $1/\sqrt{MN}$, which seemed like a more intuitive way of scaling, rather than having a factor of $1/MN$ on one of the transformations, and none for the other.

```
def forward_fourier_transform(image):
    image_fft = np.fft.fft2(image, norm='ortho')
    amplitude_spectrum = np.abs(image_fft)
    phase_spectrum = np.angle(image_fft)
    return amplitude_spectrum, phase_spectrum
```

The code can be accessed in the file `transformations.py` which is in `code/exercise_1/functions`. In general, visualizing the phase spectrum yields little intuitive information, therefore we will only analyze the amplitude spectrum of the image. In this step, the amplitude spectrum is traditionally processed with a filter to enhance or suppress certain frequency components. However, for this particular example, we will not apply any filter, but instead, create a visualization of the Fourier spectrum for now. After the shift for periodicity, the frequency spectrum is symmetric about its center point $F(0,0)$, which is proportional to the average of $f(x,y)$ (This is another corollary of

(3)). The proportionality constant of the average value is large ($\sqrt{MN}$), so $|F(0,0)|$ is usually the largest component of the frequency spectrum, several orders of magnitude larger than others. Because of this, the log transform $log(1 + |F(u,v)|)$ is employed in order to modify the contrast and bring out the dimmer parts of the frequency spectrum image. The spectrum values are also scaled to the full range $[0, 255]$ using minmax conversion mode. The conversion modes from float to 8bit integer values are explained more carefully in the report of the first compulsory assignment.

3.1.5 Performing the inverse of DFT

The inverse DFT is the process of transforming the frequency domain data back into the spatial domain. It involves combining the frequency components back into their spatial representation. In Python implementation, `numpy.fft.ifft2` is used for this purpose. However, this function takes a complex array from the frequency domain as its input, hence it is needed to reconstruct it from the amplitude and phase spectra. This can be done with the help of a symbol `j`, which denotes the imaginary unit and lets us define the complex polar expression easily as `amplitude_spectrum*np.exp(1j*phase_spectrum)`. After computing the inverse DFT, which results in a complex array, it is important to note, that the original input image is a real-valued function. As such, the imaginary part of the output must be equal to zero, since the original image contains no imaginary component. To extract only the real part of this complex output, the method `real` of a `numpy` array object is invoked and finally, the reconstructed representation of an image can be returned. After the inverse DFT, the reconstructed image is still in the form where the low-frequency components are centered. To obtain the correct spatial representation, it is necessary to reverse the shift applied initially for periodicity. This is once again achieved by multiplying each pixel's intensity by $(-1)^{x+y}$. These computations can be accessed within the same file `transformations.py` in the directory `code/exercise_1/functions`.

3.1.6 Displaying the results

All of the prior calculations can be visualized in Figure 1 using the `imshow()` function from the `matplotlib.pyplot` module. For the correct display of the images, each one of them must be scaled back to 8bit integer values with a relevant conversion - either minmax or truncation. Figure 1(a) - Figure 1(e) displays every step of the implementation of the Fourier transform, discussed so far. The final result in the Figure 1(f) is obtained by cropping the resulting image of the shifted inverse DFT back to the original size. The $M \times N$ region from the top left quadrant of the image, displayed in Figure 1(e), is extracted. It is necessary to highlight, that the input and the final images are of size $M \times N$, whereas the rest of the images are $2M \times 2N$. However, in Figure 1, all images are displayed at the same scale for consistency, despite the differences in their actual dimensions. We choose to scale the frequency spectrum images displayed in each one of the upcoming Figures (spectrum will be double the image size due to padding but shown in half size).

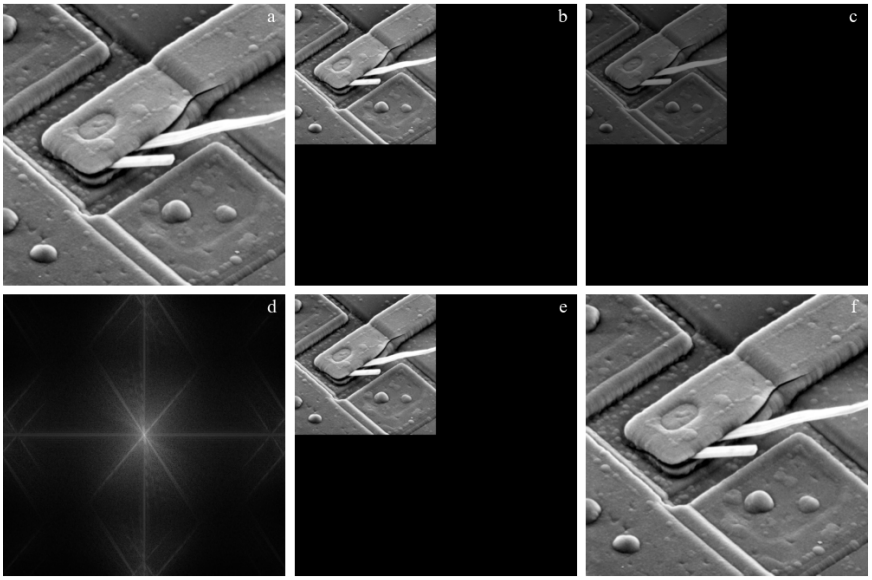

Figure 1: Implementation of the Fourier transform.

To validate the correctness of the implementation of the Fourier transform and its inverse, a practical approach is to compare the original input image with the image obtained after applying the DFT and its inverse. This comparison can be conducted through a couple of methods. One way would be achieved by subtracting the final image from the original input image and then assessing the difference visually. Figure 2 displays the difference as an image. It is clear, that all or most of the pixel values are close to zero, indicating no change made by the transformations.

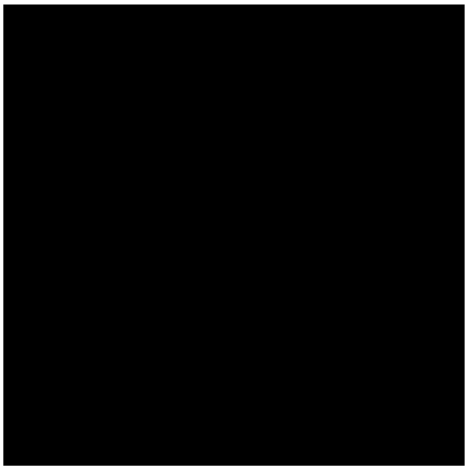

Figure 2: Difference between input and final images visualized.

Another way of making sure that the steps have been implemented correctly is to calculate the

mean of the absolute values of the differences between the original and the final images. Note that the arrays used in computing this measure of accuracy are of float type, which helps to minimize the discrepancies that could arise from rounding errors when converting to integer type. The value is obtained by executing the following commands on the Ubuntu Linux system

```
> cd code/exercise_1
> python fourier_transform.py
< The absolute mean difference between the input image and the final image:
3.424555599788157e-14
```

The value symbolizes that the two images are nearly identical, confirming that the implementation of the DFT and its inverse was executed correctly. By running these commands, not only the accuracy measure can be obtained, but also the results and visualizations shown in Figure 1 and Figure 2 can be generated. The `fourier_transform.py` file contains a full implementation of the 1st exercise, that was described in this section.

3.2 Fourier Transform of the Generated Images

Having established a foundational understanding of Fourier analysis and its practical application, this section focuses on developing a deeper knowledge of how the Fourier transform works. In the previous section, we focused on applying the Fourier transform to an existing image, analyzing the transformation from the spatial to the frequency domain and vice-versa. Now, we shift our focus to synthetically created images. By generating images with known properties and patterns, we can predict their Fourier transform outcomes, thereby building better intuition for the behavior of this transformation. This section will detail the process of image generation, outline our predictions for each image's Fourier spectrum, and then present and analyze the results, comparing our expectations with the actual outcomes.

3.2.1 Synthetic Sine Wave

The first image of our interest contains a pattern of horizontal stripes, generated using a sine wave function which creates variations in pixel intensity values along the y-axis. The code used for generating this image is as follows:

```
height, width = 256, 256
image = np.zeros((height, width))
frequency = 20 * 1/height
for y in range(height):
    for x in range(width):
        image[y, x] = 255 * (np.sin(2 * np.pi * frequency * y) + 1) / 2
```

The `frequency` variable is set so that the wave's period is 1/20th of the height of the image. The code snipped provided above for the generation of the sine wave image is a simplified illustration. The actual implementation, which follows the same concept, is found in the `generate_images.py` file, in `code/exercise_2` directory.

The generated image contains a single frequency along the y axis. This should result in distinct peaks (of intensity) in the Fourier spectrum corresponding to this frequency. Since the input image is purely real (i.e. the intensity values contain no imaginary component), the resulting Fourier spectrum exhibits a conjugate symmetry. This means that for every positive frequency component $F(u, v)$ there is a corresponding negative frequency component $F(-u, -v)$ that is its complex conjugate. In practice, this means that the frequency spectrum is symmetrical along the center point. As the sine wave varies along the y axis, the Fourier transform should also display a peak along the vertical axis. The peak's location is dependent on the frequency of the wave - the larger the frequency, the further away from the center the peak is. Also, because of the symmetry, a second peak should be positioned along the center point. Finally, the frequency spectrum must include a third peak, which is located in the center. It exhibits the average of the image's intensity values, multiplied by a large factor (as explained in 3.1.4), which means that the center point will be the brightest.

The next step is to apply the Fourier transform and analyze its frequency spectrum. The outcome is visualized in Figure 3. It contains the original generated sine wave image and its corresponding frequency spectrum. The frequency spectrum is logarithmically scaled to enhance visibility.

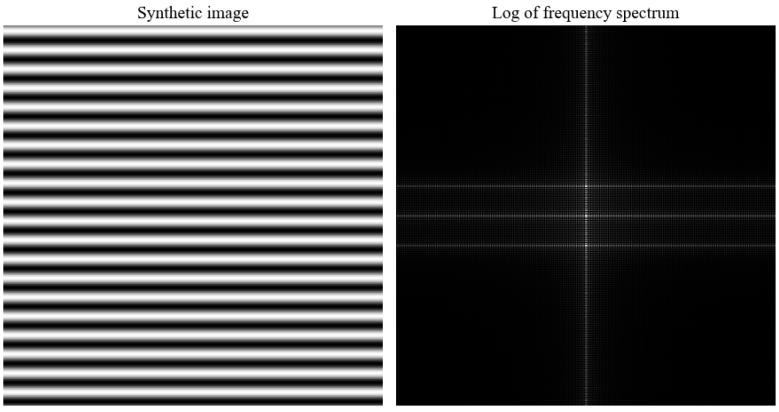

Figure 3: Generated sine wave image and its frequency spectrum.

As we can see, the Fourier spectrum, depicted in Figure 3 does seem to validate our theoretical predictions, as the 3 distinct peaks are clearly visible. However, additional lines are apparent in the plot - 3 horizontal and 1 vertical, each going through the peaks. These lines are artifacts

and are not part of the expected result. This spectral leakage is likely due to the assumption of periodicity when applying DFT. The image is interpreted as if it was periodically extended in each direction, and when it is padded with zeros, an abrupt transition from the image to the padded area is created, most probably introducing high-frequency components. The first idea to address this issue would be by applying a window function to our input image. A window function gradually reduces the intensity values of the pixels towards the edges of the image, creating a smoother transition, which may help minimize the abnormalities in the frequency domain. The effect of the window function can be seen in Figure 4.

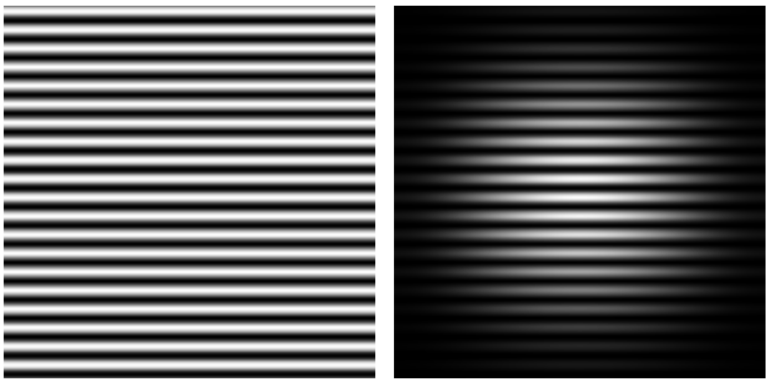

Figure 4: Generated sine wave image with and without windowing applied.

When the windowed image is then extended with padding, the smoother transition might help avoid the creation of additional lines. The application of the window function to our input image can be achieved by setting the `window` parameter to `True` (it is set to `False` by default) of a function `apply_and_visualize_fft_on_generated_images()` which is in a file `horizontal_sine_wave.py`, dedicated for this specific example only. The Python file is inside `code/exercise_2` subfolder. The frequency spectrum of an image with the window function applied is visible in Figure 5.

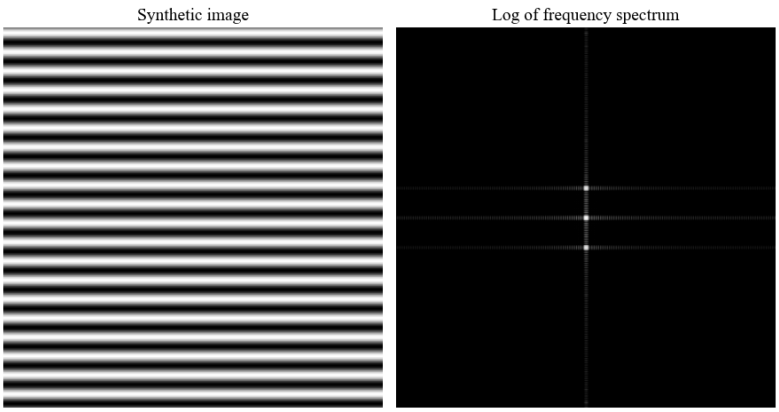

Figure 5: Generated sine wave image and its frequency spectrum with windowing applied.

The effect of the window functions is clearly visible, having improved the intensity of the three peaks and reduced the additional lines. However, a closer inspection reveals that the artifacts are not fully eliminated. It seems like the padding itself is forming additional periodic patterns when an image is extended, and those periodic patterns are being captured in the frequency spectrum. Let us try yet another approach by avoiding the padding of an image altogether. Since the image of interest is generated using a sine function, and the period is selected to match the borders of an image, the extension of such an image should not introduce any abrupt changes in intensity values. Thus, the rejection of padding may lead to promising results. The implementation of it is once again in `horizontal_sine_wave.py` file. It is accomplished by setting the argument `padding` to `False` of a function `apply_and_visualize_fft_on_generated_images()`. The final result is displayed in Figure 6, in which the frequency spectrum is calculated without the image padding (and without the windowing). These results can be generated by navigating to the desired directory and running the Python file, which, in the Ubuntu Linux environment is achieved by

```
> cd code/exercise_2
> python horizontal_sine_wave.py
```

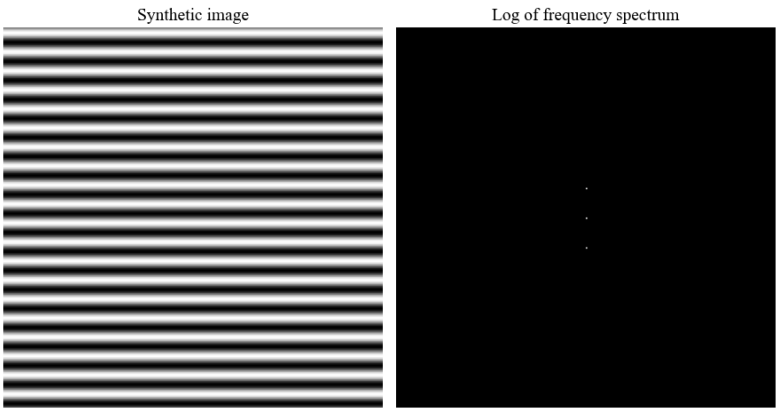

Figure 6: Generated sine wave image and its frequency spectrum without padding applied.

The frequency spectrum finally shows three distinct peaks along the vertical axis: a central peak at the origin, and two symmetrically placed peaks above and below it. The absence of horizontal and vertical lines shows that the discontinuity-related artifacts have been successfully avoided. However, such an approach of removing the padding can only be taken into account, when an image has clear periodicity. This result aligns with the expected outcome for a Fourier transform of a sinusoidal pattern. It is also worth mentioning, that since the padding was omitted, the image size matches its spectrum size representation, which was not the case in previous Figures.

3.2.2 Synthetic Checkerboard Pattern

For the second synthetic image, we are introducing a checkerboard pattern, formed by a combination of two sinusoidal waves, one in a horizontal direction, and another in a vertical direction. It was generated by looping over each pixel (`x, y`) of an empty `image` and setting its intensity value according to

```
image[y, x] = 255 * (np.cos(2 * np.pi * frequency * x) + \
np.sin(2 * np.pi * frequency * y) + np.sqrt(2))/(2 * np.sqrt(2)).
```

The `frequency` is set to `20 * 1 / 256`, which implies that each sinusoid completes 20 periods over the width and height of the image (having the width and height equal to 256).

We can expect to see four distinct peaks in the frequency domain. These peaks should be symmetrically located about both horizontal and vertical axes, reflecting the presence of two orthogonal sinusoidal patterns in the image. The intensity of these peaks should be equivalent, given that the amplitude of the sinusoids in the spatial domain is the same. The frequency domain must also include the brightest point at the center location. Also, based on our previous analysis, we can expect horizontal and vertical lines, crossing the peak points, which appear because of the discontinuities when an image is extended in both directions. In this case, there should be an equal

amount of lines in each direction, because of the equivalent sinusoidal components in x and y axes. The result of the Fourier transform applied to this synthetic image is displayed in Figure 7.

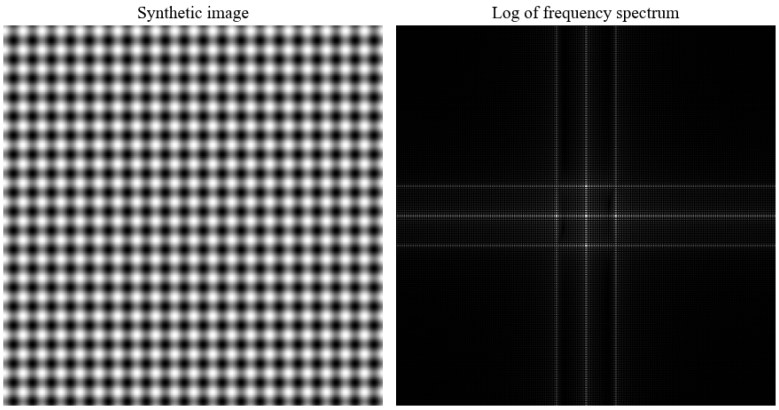

Figure 7: Generated checkerboard pattern and its frequency spectrum.

It can be further enhanced given the success of the analysis on the previous image. Avoiding the padding should again yield a cleaner frequency spectrum because the sinusoidal checkerboard pattern can be extended indefinitely without encountering discontinuities at the edges. The absence of artifacts in the frequency space can be seen in Figure 8.

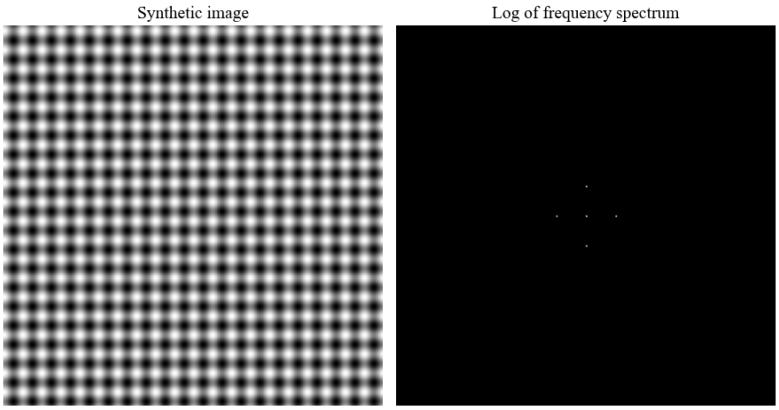

Figure 8: Generated checkerboard pattern and its frequency spectrum without padding.

The resulting frequency spectrum of the synthetic checkerboard image closely matches the expected outcome. The image clearly exhibits a pattern of two orthogonal sinusoidal waves, which, in the frequency domain, correspond to four distinct peaks on u and v axes, along with a peak in

the center. The results of the latter two figures can be obtained by running the following command lines in the Ubuntu terminal

```
> cd code/exercise_2
> python checkerboard_sine_wave.py
```

3.2.3 Synthetic Square Shape

For the last image, a square shape is generated with uniform intensity inside and zero intensity outside its boundaries. It is generated according to the following snippet of the code

```
height, width = 256, 256
image = np.zeros((height, width))
size = 80
for y in range(height):
    for x in range(width):
        square_equation = np.abs(x + y - width)/np.sqrt(2) + np.abs(y - x)/np.sqrt(2)
        if(square_equation >= 0 and square_equation < size):
            image[y, x] = 255
```

For the full implementation of the generated image, one can refer to the `generate_images.py` file, located within the `code/exercise_2` directory. The Fourier transform of a square shape with uniform interior intensity is expected to have a distinct pattern, which is easily reasoned by theoretical knowledge. A continuous Fourier transform of a square figure $f(t) = A \cdot \mathbf{1}_{[-W/2,W/2]}(t)$ produces a *sinc* function [3]

$$\begin{aligned} F(\mu) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i2\pi\mu t} dt = \frac{1}{\sqrt{2\pi}} \int_{-W/2}^{W/2} A e^{-i2\pi\mu t} dt = \\ \frac{1}{\sqrt{2\pi}} \frac{A}{i2\pi\mu} (e^{i\pi\mu W} - e^{-i\pi\mu W}) = \frac{AW}{\sqrt{2\pi}} \frac{sin(\pi\mu W)}{\pi\mu W} =: \frac{AW}{\sqrt{2\pi}} sinc(\mu W). \end{aligned} \tag{7}$$

For the sake of simplicity, a one-dimensional example is provided, however, this can be easily extended to two dimensions, resulting in two sinc functions multiplied together. The sinc function is characterized by a high central peak with ripples fading outwards. The Fourier spectrum of the square shape should share similar characteristics along the horizontal and vertical axes, creating a cross-shaped pattern. The plot should have a distinct peak in the middle of the frequency domain and symmetrical ripples, decaying away from the center. The spacing and intensity of the ripples are inversely related to the width of the square (as can be seen in (7)). In order to verify whether

[3] The indicator function $\mathbf{1}_I$ is defined as $\mathbf{1}_I(t) = \begin{cases} 1 & \text{if } t \in I, \\ 0 & \text{if } t \notin I. \end{cases}$

the results fit the expectations, DFT is applied to the generated image, resulting in a frequency spectrum, visualized in Figure 9.

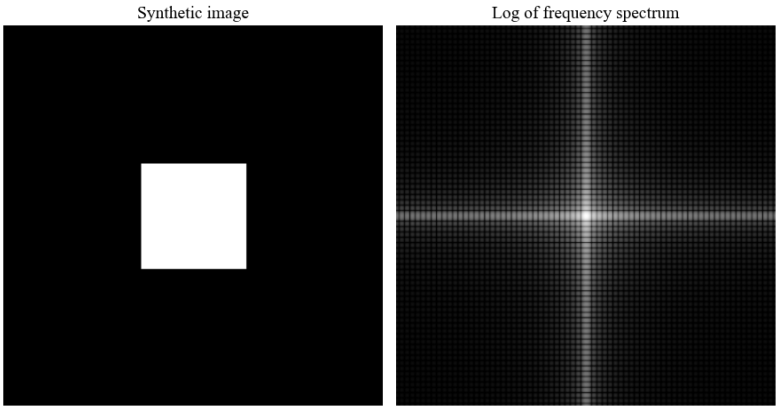

Figure 9: Generated square shape and its frequency spectrum.

The output of the Fourier transform aligns with theoretical expectations. In the frequency spectrum, we observe a central peak with horizontal and vertical rippling lines, indicating the sinc function behavior. These ripples are the result of sharp transitions at the square's boundaries, which require a broad range of frequency components to be accurately represented in the spatial domain. A general rule in Fourier analysis is that finite shapes in the spatial domain often correspond to infinite shapes in the frequency domain (i.e. infinite amount of frequency components is required to describe them), and vice-versa. The frequency spectrum, seen in Figure 9, can be replicated by running the following commands in the Ubuntu Linux terminal

```
> cd code/exercise_2
> python square_shape.py
```

Apart from the three examples discussed, several additional images were generated, encompassing a diverse range of patterns. These would include different combinations of sinusoidal waves, a circle, a cross, and a Mandelbrot fractal (which takes some time to run). These additional examples were not included in the report, however, the frequency spectra for each one of them are readily accessible by executing the `fourier_transform_all_images.py` script, which is in `code/exercise_2` directory.

3.3 Filtering in Frequency Space

This section is dedicated to understanding how filtering works in the frequency domain compared to the spatial domain. The third task involves implementing high-pass and low-pass filters, including *ideal*, *Butterworth*, and *Gaussian* types, and analyzing their effect on images. We will

explore the advantages and limitations of frequency domain filtering and compare these effects with spatial filtering, gaining a clearer perspective on the applications of both filtering approaches.

3.3.1 Implementing Frequency Space Filters

The process of filtering images in frequency space mainly relies on a convolution characteristic of the Fourier transform. It can be shown, that the Fourier transform of the convolution of two functions in the spatial domain is equal to the *product* in the frequency domain. This property is called the *Convolution Theorem* and because of it, filtering in frequency space boils down to modifying the Fourier transform of an image and then computing the inverse transform to obtain its spatial representation. Given a padded image $f(x, y)$ of size $P \times Q$ pixels, a mathematical expression of the processed image can be written as

$$g(x, y) = \Re[\mathcal{F}^{-1}\{H(u, v) \cdot F(u, v)\}], \tag{8}$$

where $\mathcal{F}^{-1}$ is the inverse DFT, $F(u, v)$ is the DFT of the input image, $H(u, v)$ is a filter and $g(x, y)$ is the filtered output image. The filter function can either be created directly or transformed from the spatial domain. For the element-wise multiplication to be valid, the filter applied must be of the same size as the padded image. Thus, the filter is usually a symmetric function of size $P \times Q$ centered at $(P/2, Q/2)$. Because of the filter size, it may be faster to perform the actual filtering in the spatial domain, however, designing the filters is usually a simpler process in the frequency domain.

Low-pass and high-pass filters are two fundamental types of filters used in image processing. A low-pass filter allows frequencies below a certain cutoff frequency to pass through, lowering the amplitude of frequencies above this threshold. This typically results in smooth, gradual changes in intensity, which gives a blurring (smoothing) effect. A high-pass filter, on the other hand, allows frequencies that are higher than a specific cutoff frequency to pass while reducing the lower frequencies. It is used to enhance or detect edges and sharp intensity transitions in an image.

Ideal Filters

1. The ideal low-pass filter creates a binary mask for the frequencies of the image. It has a sharp cutoff at a specified frequency, allowing frequencies below this cutoff to pass while blocking higher frequencies. It is defined as

$$H(u, v) = \begin{cases} 1 & \text{if } D(u, v) \leq D_0, \\ 0 & \text{if } D(u, v) \geq D_0, \end{cases}$$

where $D(u, v) = \sqrt{(u - P/2)^2 + (v - Q/2)^2}$ and D_0 the the cutoff frequency radius. The initial idea to implement the ideal low-pass filter in Python was by iterating over each pixel of the

image and calculating the inequality, given above. However, the approach was too expensive computationally, so a more efficient way using `numpy`'s `meshgrid` function was adopted.

```
filter = np.zeros((P, Q))
U, V = np.meshgrid(np.arange(Q), np.arange(P))
D = np.sqrt((U - Q//2)**2 + (V - P//2)**2)
if filter_type == 'lowpass':
    filter[D <= cutoff_radius] = 1
```

A full function is located in `code/exercise_3/frequency_filtering` directory, in the `filters.py` Python script. The effect, of an ideal low-pass filter applied to the input image is visualized in Figure 10. The `cutoff_radius` was selected to be 30.

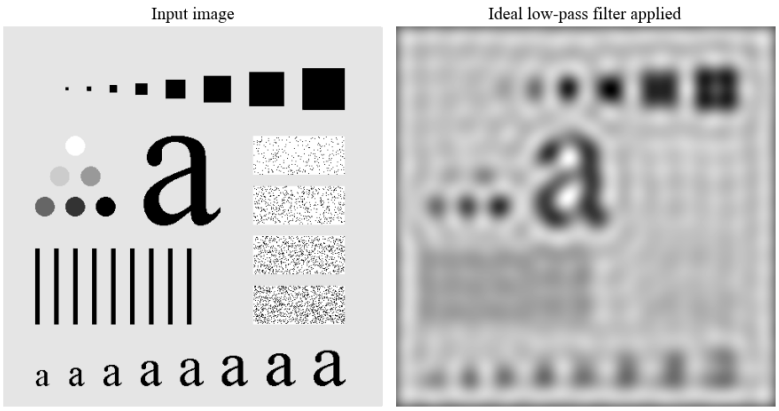

Figure 10: Ideal low-pass filter applied to an image.

The implementation of this filter involves several steps, discussed in section 3.1. The resulting image has a blurred appearance, with sharp edges and noisy sections being less pronounced. The use of an ideal filter creates ringing artifacts, which are caused by the abrupt cutoff of the filter. This is an equivalent case to one, discussed in section 3.2.3, where the Fourier transform of a uniform square pattern results in a sinc function. This function is the one, causing these artifacts. However, this time the artifacts are translated to the spatial domain.

2. The ideal high-pass filter is a complement of the ideal low-pass filter. It blocks frequencies below the threshold and allows high frequencies to pass. It is implemented in the same function, located in Python file `filters.py` as the low-pass filter, where it checks if `filter_type == 'highpass'` and sets `filter[D > cutoff_radius] = 1` if the latter holds true. The implementation is displayed in Figure 11. It shows an ideal high-pass filter, applied to the same image as before. The `cutoff_radius` is left the same as well.

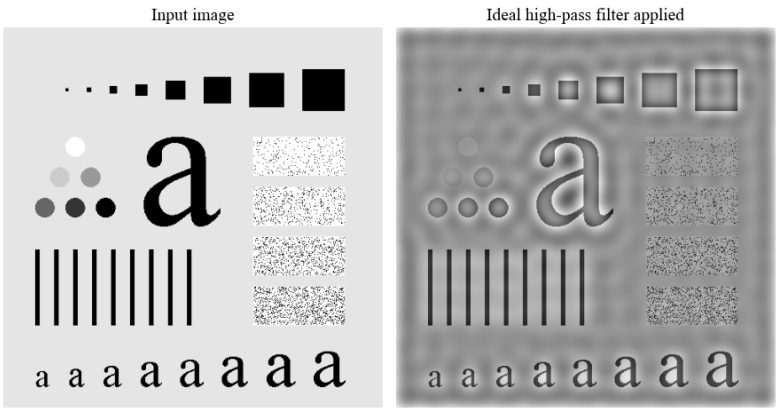

Figure 11: Ideal high-pass filter applied to an image.

The high-pass filter has amplified the edges and areas of sharp intensity transitions. There is also evidence of the ringing artifacts, just like with the low-pass filter. Note, that a high-pass filter lowers the average value of the pixel intensities, so the coloring seems off. However, a sharper image with its true colors can be obtained by adding a constant value to the high-pass filter. The two figures were acquired using a Python script from `ideal_filter_frequency.py`. One can also produce these figures by executing the following command lines in the Ubuntu Linux terminal

```
> cd code/exercise_3
> python ideal_filter_frequency.py
```

Butterworth Filters

1. The Butterworth low-pass filter allows frequencies below a certain threshold to pass, similar to the ideal low-pass filter, but does it with a smoother transition. It avoids the sharp cutoff of the ideal low-pass filter, which helps reduce ringing artifacts. The definition of the Butterworth low-pass filter is

$$H(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}}, \tag{9}$$

where $D(u,v)$ is used to measure the distance from the center again, D_0 is the cutoff frequency and n is the order of smoothness. The implementation in Python follows a similar manner of creating a meshgrid using `numpy`.

```
U, V = np.meshgrid(np.arange(Q), np.arange(P))
D = np.sqrt((U - Q//2)**2 + (V - P//2)**2)
if filter_type == 'lowpass':
    filter = 1 / ((1 + D / cutoff_radius)**(2 * order))
```

This fragment of a full Butterworth filter implementation is taken from `filters.py`, located in `code/exercise_3/frequency_filtering`. This implementation is visualized in Figure 12. The value 30 was used again for the `cutoff_radius`.

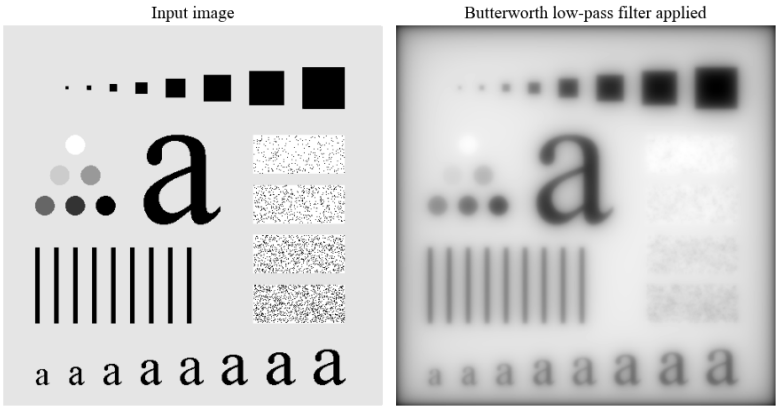

Figure 12: Butterworth low-pass filter applied to an image.

The low-pass Butterworth filter has smoothed out details and reduced the sharpness of edges. Unlike the ideal filter, the Butterworth filter is less likely to introduce ringing artifacts, due to its smooth frequency response. A dim border could also be seen, which is likely a result of zero-padding.

2. The Butterworth high-pass filter allows higher frequencies to pass while discarding the lower frequencies. The smooth transition of the Butterworth filter is maintained, providing a more gradual change than the ideal filter. It is defined as

$$H(u,v) = \frac{1}{1 + [D_0/D(u,v)]^{2n}}, \tag{10}$$

with the same $D(u,v)$ and D_0. In Python, this translates to setting `filter` values to `1 / ((1 + cutoff_radius / (D+1e-9))**(2 * order))` if the `filter_type == 'highpass'`. Note the additional `1e-9`, added to `D`, which is to prevent the division by zero. The code snippet is from the same file as a low-pass filter, being `filters.py`. The `cutoff_radius`, or D_0, was selected as 30. The high-pass filter implementation is displayed in Figure 13.

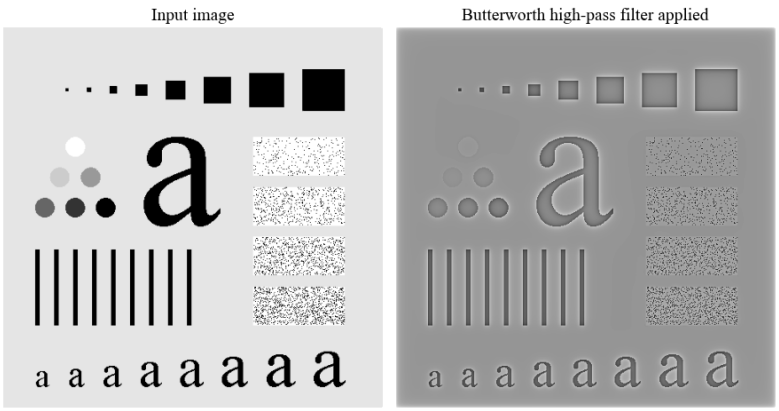

Figure 13: Butterworth high-pass filter applied to an image.

After applying the filter, we can observe an emphasis on the high-frequency components, without the pronounced ringing artifacts that have been introduced with the ideal filter. The results in Figure 12 and Figure 13 can be obtained by running the following commands in the Ubuntu command line

```
> cd code/exercise_3
> python butterworth_filter_frequency.py
```

Gaussian Filters

1. The Gaussian low-pass filter passes frequencies below a certain cutoff and diminishes others, similar to the Butterworth low-pass filter. It uses a Gaussian function for the transition, which provides a smooth decline in frequency response. The function is defined as

$$H(u, v) = e^{-D^2(u,v)/2D_0^2}, \tag{11}$$

where $D(u, v)$ and D_0 are yet the same sizes as in previous filters. The implementation in Python follows the same path

```
U, V = np.meshgrid(np.arange(Q), np.arange(P))
D = np.sqrt((U - Q//2)**2 + (V - P//2)**2)
if filter_type == 'lowpass':
    filter = np.exp( -D**2 / (2 * cutoff_radius**2))
```

The effect of the Gaussian low-pass filter on an input image can be seen in Figure 14. The standard deviation of the Gaussian function, or the `cutoff_radius` was set to 30.

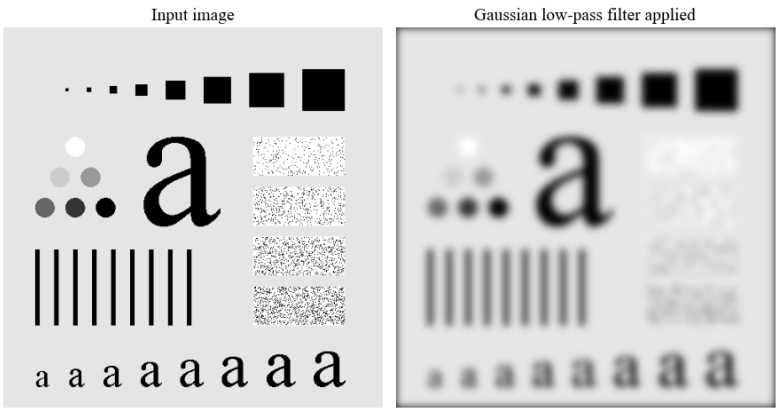

Figure 14: Gaussian low-pass filter applied to an image.

The Gaussian low-pass filter has effectively blurred the input image, smoothing out the details and reducing the overall sharpness. Personally, this seems to give the most natural smoothing effect out of all three low-pass filters.

2. The Gaussian high-pass filter does the opposite effect of the low-pass filter, allowing higher frequencies to pass. It is defined as a complement of the low-pass filter, being

$$H(u, v) = 1 - e^{-D^2(u,v)/2D_0^2}, \tag{12}$$

where $D(u, v)$ and D_0 are defined as in the previous formulas. The implementation in Python is straightforward and does not require an explicit showcase. It can be found in `filters.py` file, which is in the directory `code/exercise_3/frequency_filtering`. Figure 15 shows its application to an image. The value of D_0 was set to 30. The resulting image has enhanced textures and sharpened features, without any harsh transitions or artifacts introduced.

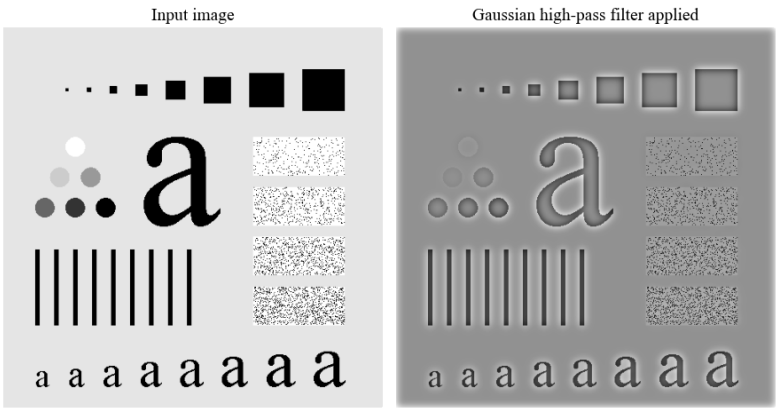

Figure 15: Gaussian high-pass filter applied to an image.

The following command lines were used to obtain figures 14 and 15

```
> cd code/exercise_3
> python gaussian_filter_frequency.py
```

3.3.2 Application of Linear Filters in Image and Frequency Space

In the report of the first compulsory assignment, we have explored the application of linear filters in the spatial domain. Regarding the convolution theorem, any linear filter, when zero-padded to match the size of the input image, applied in the frequency domain will yield the same result if convolved with an image in the spatial domain. This section will investigate the application of linear filters in both the image and frequency domains. Specifically, we will analyze the effects of blurring with averaging and Gaussian filters, as well as edge detection through differentiation in both domains. The theoretical aspects were addressed in the report of the first compulsory assignment, and thus will not be repeated here.

Image Space

In the image space, the application of a filter is achieved by convolving an image with that corresponding filter. The implementation of this is located in `filters.py`, which is in `code/exercise_3/spatial_filtering`. The file includes calculations for all three cases - averaging and Gaussian filters as well as Sobel differentiation.

Frequency Space

Filtering in the frequency space is achieved by padding the filter with zeros, calculating its Fourier transform, and multiplying it with the Fourier transform of the input image. The zero-

padding is achieved by generating an empty array and then assigning its center values to the filter values

```
padded_filter = np.zeros((image_height, image_width))
pad_height = (image_height - filter_size) // 2
pad_width = (image_width - filter_size) // 2
padded_filter[pad_height:pad_height + filter_size, \
pad_width:pad_width + filter_size] = filter_kernel
```

The padded filter is then taken care of in the same way as the input image (just like it was explained in 3.1). Their Fourier transforms are then multiplied, resulting in a complex array, from which the reconstructed image is obtained via the inverse Fourier transform. The implementation can be found in `spatial_filters_for_frequency.py` file, located in `code/exercise_3/frequency_filtering` directory. The same procedure will be applied for the three examples, provided below.

Blurring with an Averaging Filter

Blurring was applied to an image using an average block filter with a kernel size of 9×9 pixels. The process was executed in both image and frequency domains, and the results are visible in Figure 16.

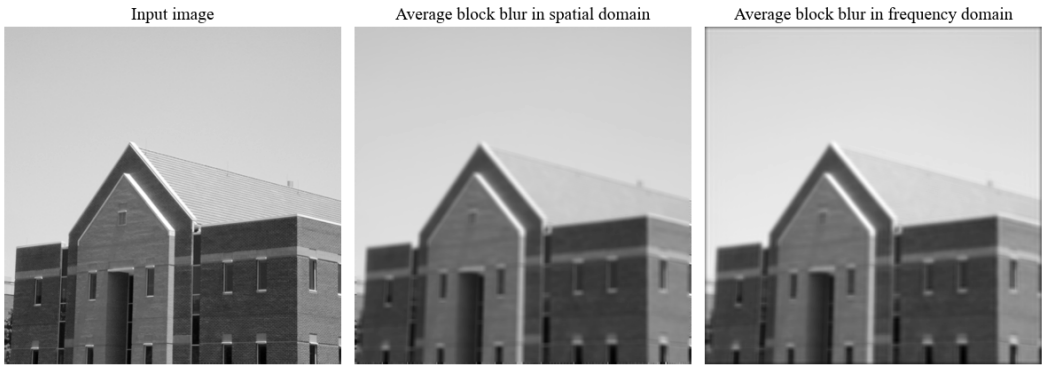

Figure 16: Application of an averaging filter in image and frequency domains.

Blurring with a Gaussian Filter

Blurring was implemented using a Gaussian filter with a kernel size of 9×9 and a standard deviation of 5. The resulting images can be seen in Figure 17.

Figure 17: Application of a Gaussian filter in image and frequency domains.

Differentiation with Sobel Filters

Finally, the image was differentiated using Sobel operators. In image space, two Sobel kernels are being convolved with an image, resulting in gradients in x and y axes, whose magnitude is then computed. In a similar manner, both gradients are computed separately using the Fourier transform, and then their magnitude is calculated. The results are visualized in Figure 18.

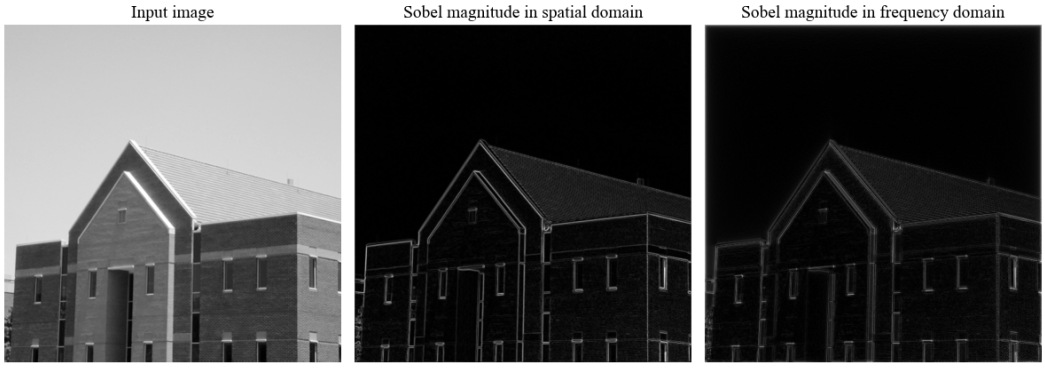

Figure 18: Application of Sobel filters in image and frequency domains.

Results

The main benefit of using the frequency domain for these calculations is the computational efficiency. The FFT reduces the complexity of convolutions, particularly for large filters or images. For comparison, computation time was calculated for differentiation with Sobel filters in both domains. The results are as such:

```
Convolution execution time: 9.690249681472778 seconds
Fourier transform execution time: 1.822422742843628 seconds
```

which confirms the efficiency of the FFT algorithm. However, applying filters in the image space is conceptually straightforward and can be more intuitive. All of these figures were generated using the same Python script `filter_comparison.py`. One can replicate these figures by running the following commands in the Ubuntu Linux terminal

```
> cd code/exercise_3
> python filter_comparison.py
```

These commands will also print the execution time which was used to compare both methods.

3.4 Image Restoration - Noise Removal

The last section focuses on gaining practical experience with various techniques of noise removal, both in the image domain and in the frequency domain. This involves understanding and applying concepts of image restoration methods, identifying different types of noise, and employing appropriate filters for noise reduction. The primary types of noise encountered in images include:

- Additive Noise: Often Gaussian, this noise is independent of the image and is added to its intensity (or frequency) values. The noise terms are unknown (defined by probability mass function) so restoration by subtraction is not an option.
- Periodic Noise: Introduces regular, structured patterns into the image. Usually appear as concentrated peaks in the Fourier spectrum and, thus, can be analyzed and filtered quite effectively.
- Salt and Pepper Noise: Randomly occurring black and white pixels. Can be effectively reduced with the use of a median filter.

3.4.1 Noise Removal in Spatial Domain

Adaptive Filters

Unlike standard linear filters, adaptive filters vary their response according to the local image characteristics. For instance, an adaptive median filter can vary its size to deal with 'salt-and-pepper' noise effectively, preserving edges while removing noise. The example effect of a non-linear filter can be seen in Figure 19.

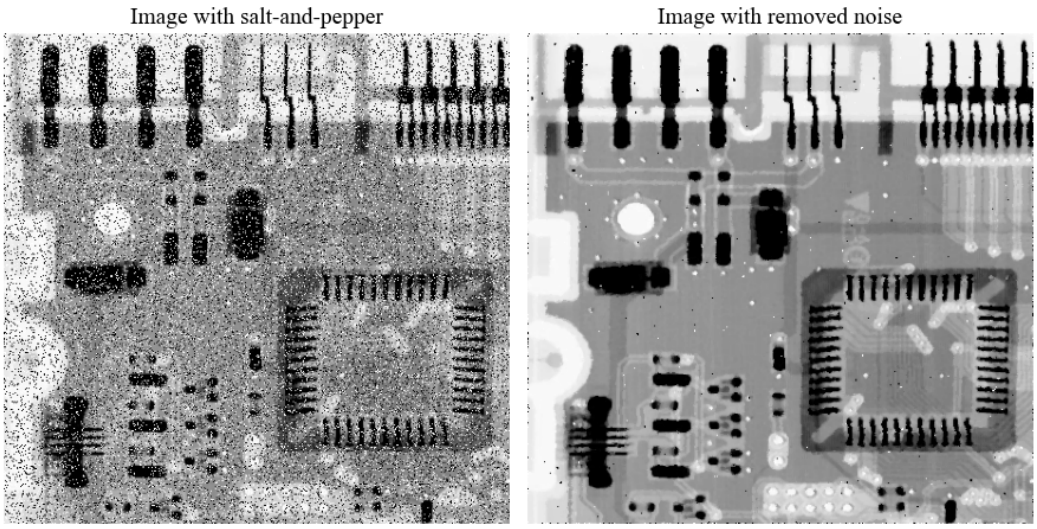

Figure 19: Application of median filter for salt-and-pepper removal.

This figure can be obtained by running the following commands

```
> cd code/exercise_4
> python salt_pepper_removal_median.py
```

in the Ubuntu Linux environment.

Error Measures

Fidelity metrics like Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR) provide quantitative assessments of noise reduction effectiveness. These measures can be expressed as functions of the original and the filtered images. They are often used for comparing the performance of different noise reduction techniques, however, the measured error is objective, and sometimes not necessarily correlated to the human visual system.

3.4.2 Noise Removal in Frequency Domain

Band-Pass and Reject Filters

These filters are instrumental in attenuating specific frequency ranges. Band-pass filters retain frequencies within a certain range, whereas reject filters do the opposite. They are particularly effective against periodic noise, which manifests as distinct peaks in the frequency spectrum. Since a band-pass filter is constructed from a combination of low-pass and high-pass filters, these filters can be designed in various forms, including Ideal, Gaussian, and Butterworth variations.

1. Ideal band-reject filter is formed by combining an ideal high-pass and an ideal low-pass filter. The formula is

$$H(u,v) = \begin{cases} 0 & \text{if } D_0 - W/2 \leq D(u,v) \leq D_0 + W/2, \\ 1 & \text{otherwise.} \end{cases}$$

There $D(u,v)$ is the distance from the center of the frequency spectrum, D_0 is the cutoff frequency, and W is the bandwidth.

2. Gaussian band-reject filter combines Gaussian high-pass and low-pass filters, providing a smoother transition than the Ideal filter. The formula is

$$H(u,v) = 1 - e^{-[\frac{D^2(u,v)-D_0^2}{DW}]^2}, \tag{13}$$

where $D(u,v)$, D_0 and W correspond to the same variables.

3. Butterworth band-reject filter is also characterized by a smoother response than the Ideal filter. Its formula is

$$H(u,v) = \frac{1}{1 + [\frac{DW}{D^2(u,v)-D_0^2)}]^{2n}}, \tag{14}$$

with the same $D(u,v)$, D_0, W as above, and n being the order of the smoothness.

Band-pass filters can be obtained by calculating the complement of band-reject filters: $H_{BP}(u,v) = 1 - H_{BR}(u,v)$. Band filters can be used to effectively remove periodic noise. Figure 20 visualizes the results of an ideal band-reject filter. A pattern of the periodic noise is visible in the frequency domain by symmetrical peaks. These peaks can be discarded from the frequency domain by using a band of width 20 and a radius of 380.

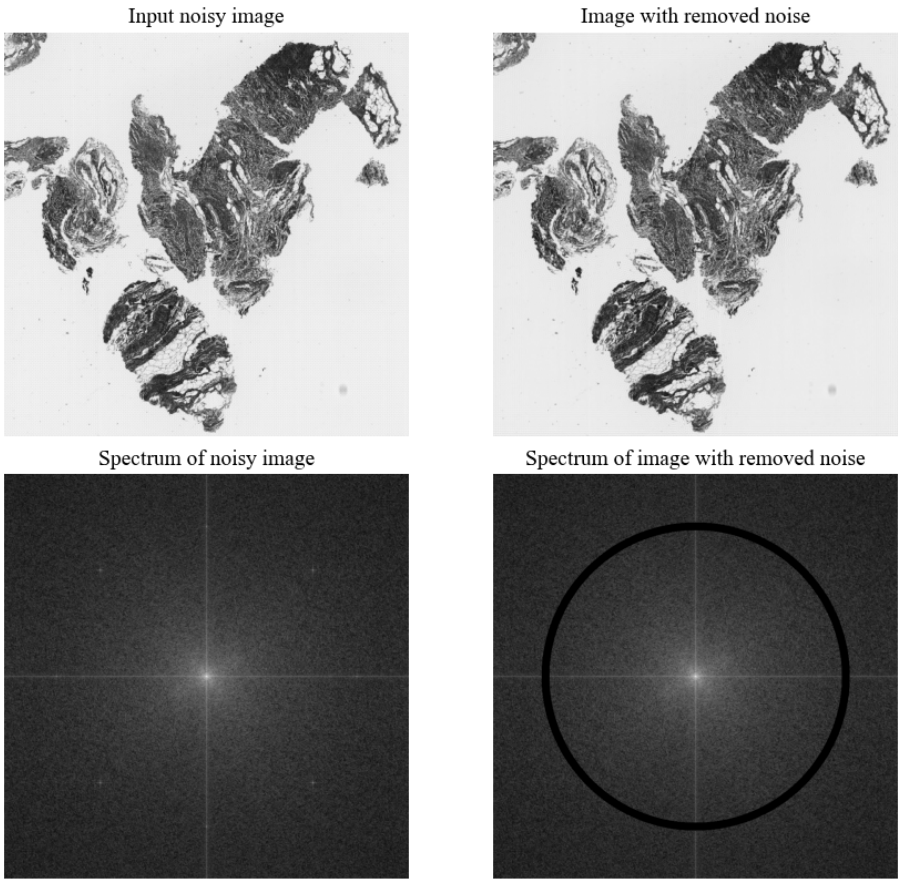

Figure 20: The results of a band-reject filter on a noisy image.

The band-reject filter is implemented by creating a meshgrid of ones and then assigning 0 to those values, which obey the formula of ideal band-reject filter

```
center_P, center_Q = P // 2, Q // 2
U, V = np.meshgrid(np.arange(Q), np.arange(P))
D = np.sqrt((U - center_Q) ** 2 + (V - center_P) ** 2)
band_filter = np.ones((P, Q), dtype=int)
band_filter[(D >= low_cutoff) & (D <= high_cutoff)] = 0
```

The full implementation of the filter for this specific image is in `noise_removal_band.py` file, located in `code/exercise_4` directory. The results of the figure can be achieved by running the following commands in the Ubuntu terminal

```
> cd code/exercise_4
> python band_reject_example_1.py
```

The band-pass method offers a flexible way of choosing the filter's characteristics and allows precise control over which frequencies are passed or rejected. However, filtering an entire band of frequencies may sometimes remove too much detail.

Notch Filters

Notch filters are designed to target and discard specific frequency components. Notch filters are effective in eliminating localized, frequency-specific noise. A notch filter rejects (or passes) frequencies in a predefined neighborhood of the frequency. The filters must be symmetric about the origin (center of the frequency rectangle), so a notch filter transfer function with center at (u_0, v_0) must have a corresponding notch at location $(-u_0, -v_0)$. Notch reject filter transfer functions are constructed as products of highpass filter transfer functions whose centers have been translated to the centers of the notches. The general form is

$$H_{NR}(u,v) = \prod_{k=1}^{Q} H_k(u,v) H_{-k}(u,v), \tag{15}$$

where $H_k(u,v)$ and $H_{-k}(u,v)$ are highpass filter functions whose centers are at (u_k, v_k) and (u_{-k}, v_{-k}) respectively. Similar to band-pass/ band-reject filters, notch pass filters can be obtained by calculating a compliment: $H_{NP}(u,v) = 1 - H_{NR}(u,v)$. A simple implementation of a notch-reject filter in Python is in the `noise_filters.py` file, located in `code/exercise_4/frequency_filtering` folder. The effect of such a filter is displayed in Figure 21.

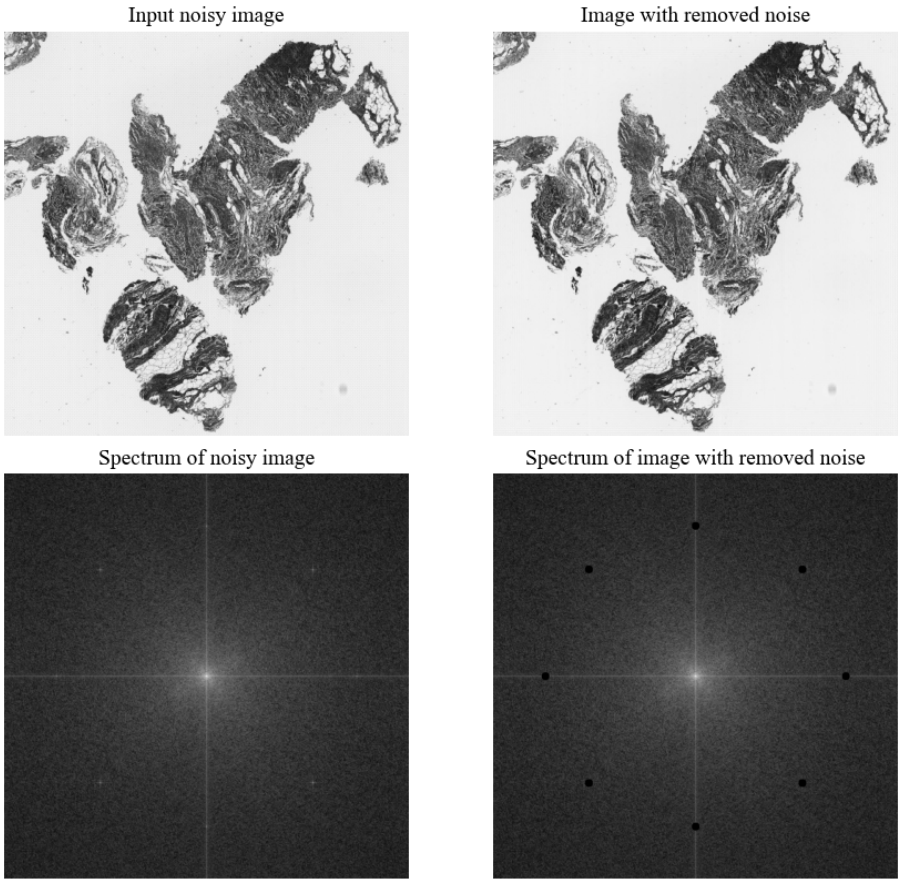

Figure 21: The results of a notch-reject filter on a noisy image.

The outcomes displayed in this figure can be reproduced by executing the following commands in the Ubuntu terminal

```
> cd code/exercise_4
> python notch_reject_example_1.py
```

Notch filters are highly effective for targeting specific frequencies, such as removing periodic noise. Different variations (Ideal, Gaussian, Butterworth) offer options for different applications, with varying levels of smoothness. They provide control over the exact frequencies to be affected, however, they are often designed for specific types of noise or interference and may not be suitable for general-purpose filtering.

For the specific example of TIFF file `Kidney1-Crop-Noise1.tif`, the notch filter would be a more appropriate way of noise removal, since the periodic noise appears as distinct spikes in the frequency domain. However, the band filter does not seem to remove the essential frequency components, resulting in an effective way of noise removal as well.

Using the same approach on a second provided example in `Kidney1-Crop-Noise2.tif` does not yield good results. Figures 22 and 23 show the application of band-reject and notch-reject filters, applied on this TIFF image with the same parameters.

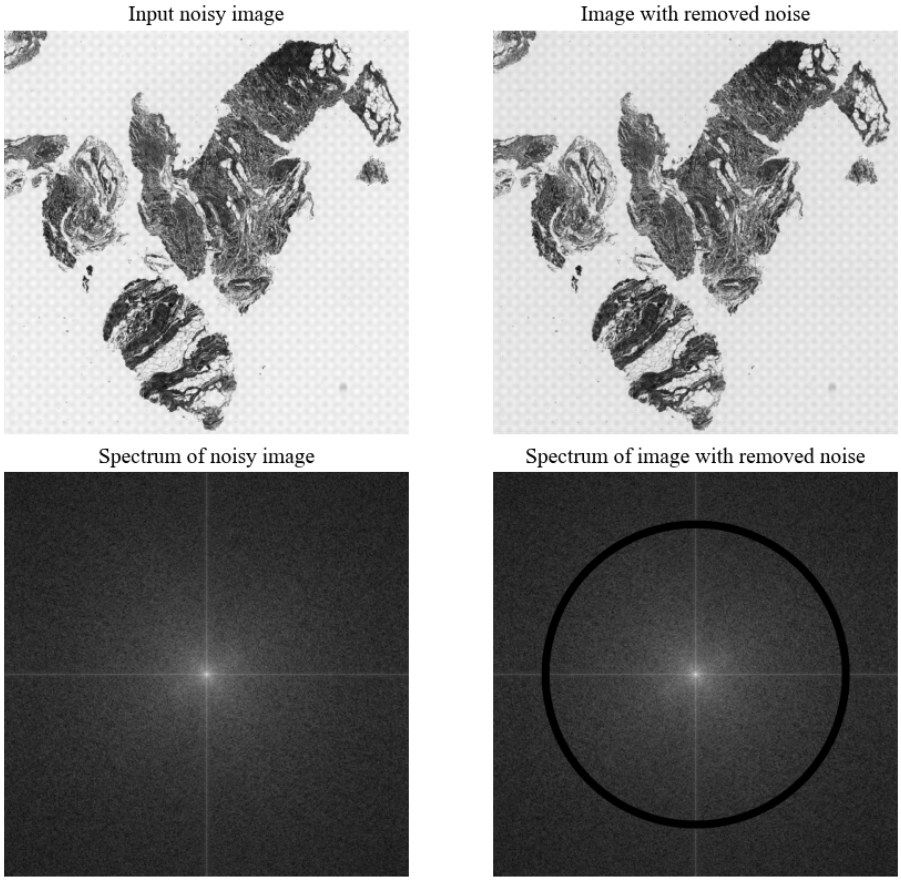

Figure 22: Unsuccessful results of a band-reject filter on a different noisy image.

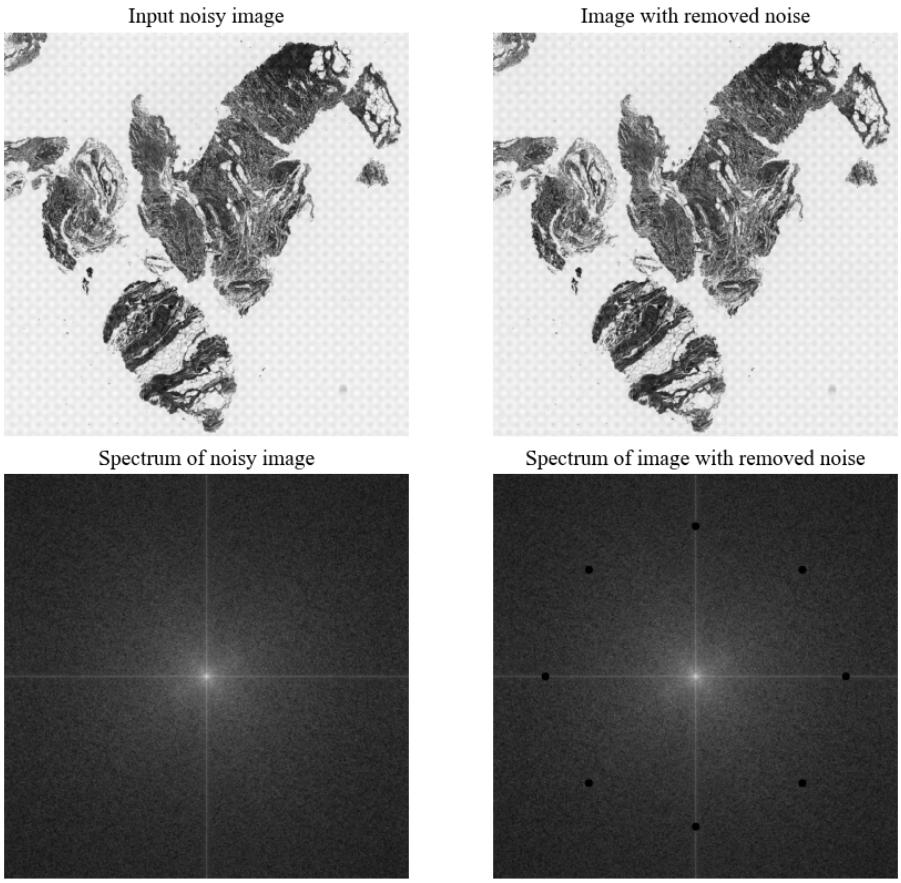

Figure 23: Unsuccessful results of a notch-reject filter on a different noisy image.

The noise pattern is periodic, however, it seems to be more complex, than the previous ones. The frequency domain reveals, that a noisy pattern is composed of a vast amount of sinusoidal components added together in x and y axes. Therefore, a different approach should be employed (perhaps, a custom rectangular reject filter) in order to reduce the noisy pattern, apparent in the given image.

3.4.3 Noise Removal of a Thorax Image

The provided noisy image of a thorax exhibits a salt-and-pepper pattern, therefore, a median filter is employed for its known effectiveness against such noise. The efficiency of the selected filter can be validated with Figure 24, which visualizes the outcome.

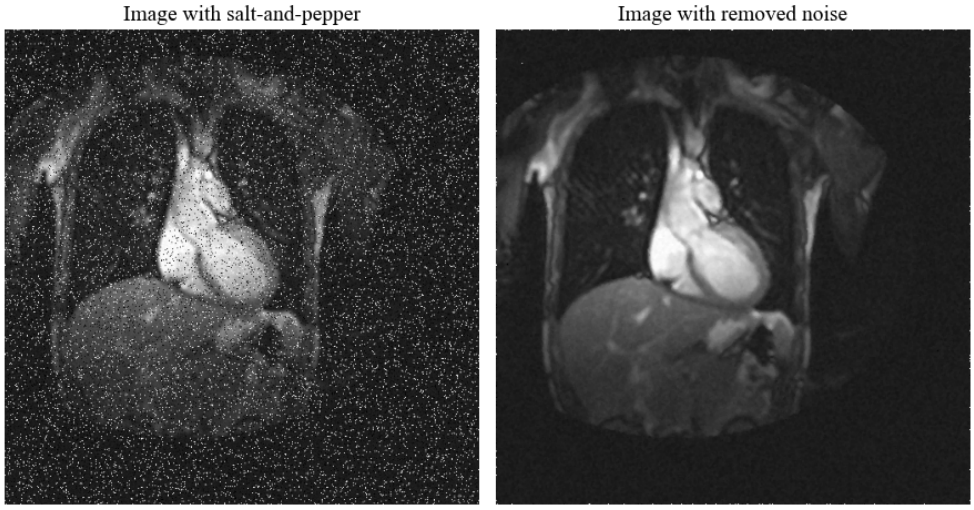

Figure 24: Noise removal of a thorax image.

The effectiveness of the filter could be evaluated through objective metrics like Mean Squared Error (MSE) or Peak Signal-to-Noise Ratio (PSNR), but due to time constraints, it was not implemented. Therefore, the evaluation is only visual. The results, displayed in Figure 24 can be obtained by running

```
> cd code/exercise_4
> python noise_removal_thorax.py
```

command lines in the Ubuntu Linux terminal.