Blur Detection using 2D DFT - ELE510

Ola Andrè Flotve - Håvard Moe Jacobsen
University of Stavanger, Norway

1 SUMMARY

For this project the aim was to get more familiar with the theory of the discrete Fourier transform(DFT) as well as exploring how the 2D DFT could be used for blur detection. First we introduce the DFT at the surface and then state the goal of the project. Next there is a theory section which goes more in depth about the different versions of the DFT, followed by a detailed section on the implementation. After this we present the results where we as expected saw that the fast Fourier transform (FFT) performs better than the DFT and that our blur detector is able to distinguish between blurry and sharp images. Lastly, we noticed that the blur detector would need some reference points for the threshold because the scenery of the image directly affects the scoring of the detector.

2 INTRODUCTION

The discrete Fourier transform (DFT) lays the foundation for one of the most used algorithms in the world, which is the fast Fourier transform (FFT). One can find it in everything from image storage to acceleration of heavy calculations, which normally would take a lot of time. Another application where it could be used, is in the detection of blurry images. It could then be used by photographers to quickly discard unusable images or for example in image processing before deep learning.

Before performing blur detection it would be convenient to get a better understanding of the fundamentals of the DFT. The most fundamental part of the DFT is the Fourier transform itself. When signals are processed we often want to perform an analysis of the different frequencies that exist in the signal, and this is where the Fourier transformation comes in. A common use for this is on audio signals [1, p. 272]. The Fourier transform is continuous, but in the real world, we usually gather signals on a discrete form [1, p. 277]. This is why we will focus on the DFT. The DFT is the variation of the Fourier transform that is most commonly used in practice. The reason behind this is as mentioned that most signals from the real world although often continuous are sampled a finite number of times [1, p. 277]. This is a nice start, but we are going to work with images so we need the two-dimensional DFT. The two-dimensional DFT is simply an extension of the 1D DFT where it's possible to analyze frequencies in two directions instead of one. Even though a straightforward implementation will produce the results, it will take a lot of time. A faster version of this as mentioned earlier was the FFT, which was rediscovered by John Tukey in 1963 [4] and resulted in a revolution in many fields.

DFTs can be used in many different settings and in this project we want to explore the following:

- How does the DFT compare in performance compared to the FFT?
- How can 2D DFT be used for blur detection?

Project in Image Processing and computer vision (ELE510), IDE, UiS 2018.

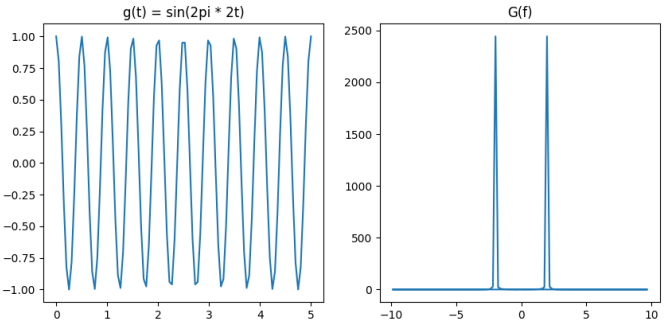

Figure 3.1: Transition from the time domain to frequency domain by the use of Fourier transform

- Is the performance of the blur detector affected by different types of images?

3 THEORY

The Fourier transform is a mathematical multi-purpose tool with a lot of different uses. It is used to convert signals (in the time domain) or images (spatial domain) into an equivalent representation in the frequency domain. In this section, we will introduce the Fourier transform, some of its different variants, properties, and some of its use cases.

3.1 Fourier Transform

Equation 1 is the standard 1D Fourier transform. if $g(t)$ is a signal (in the time domain) for which you want to analyze the frequencies, you need to multiply it with a certain complex exponential and take the integral with respect to time. The resulting $G(f)$ is the transformed signal that now takes a frequency as input (frequency domain) as opposed to time, which is used as input in g.

$$G(f) = \int_{-\infty}^{\infty} g(t) e^{-j2\pi f t}\, dt \tag{1}$$

Figure 3.1 is a textbook example: here use $g(t) = \cos(2\pi \cdot ft)$ where $f = 2$ (2 cycles per unit of time). We use the Fourier transform on the signal, and we get its frequency representation as seen on the right in the figure. $G(f)$ shows two spikes, one at 2 and the other one at -2, this is because the original signal $g(t)$ is only composed of one frequency 2 (cycles per time unit).

$$e^{j\theta} = \cos(\theta) + j\sin(\theta) \tag{2}$$

Formula 2 is the famous "Euler's formula". It lets us write complex exponentials as a real cosine term added with a complex sin term. By inserting Euler's formula into the equation 1, we get equation 3 where we divide the original Fourier transform integral into two terms. These terms are often called G_{even} and G_{odd} because the $G_{odd}(t) = 0$ for any signal with $g(t) = g(-t)$ meaning the signal

is symmetric around 0 [1, p. 273]. On the contrary, $G_{even}(t) = 0$ whenever $g(t) = -g(-t)$ meaning the signal is asymmetric around 0.

$$G(f) = \underbrace{\int_{-\infty}^{\infty} g(t)\cos(2\pi ft)\,dt\}}_{G_{even}} + \underbrace{j\int_{-\infty}^{\infty} g(t)\sin(2\pi ft)\,dt\}}_{G_{odd}} \quad (3)$$

When we have divided the Fourier transform into the two terms of odd and even, it is easy to see that the transformation from the spatial to the frequency domain will result in real values if the signal is purely composed of even frequencies, likewise a purely imaginary result if the signal is a sum of odd frequencies only.

3.1.1 Magnitude and Phase. From the figures in the appendix (A.1, A.2, A.3) one can see that even though the signal $g(t)$ is different (i.e. using a different combination of cos and sin), we still get the same **magnitude** output after taking the Fourier transform. This is because the magnitude function does not care about the direction of the complex number (i.e. how much of it is in the imaginary and real direction), it only cares about the magnitude.

A simple explanation is a coordinate system where the imaginary term is on the y-axis, and the real term is on the x-axis. If the complex number is $2 + 2j$, the complex number could be thought of as a point on coordinate (2, 2). And the magnitude would be $\sqrt{2^2 + 2^2} = 2\sqrt{2}$. This explains how the magnitude of different transformed signals can be equal even though the input function is not the same.

This also explains the need for the **phase**. If you were to calculate the inverse transform, without knowing the phase, only the magnitude. You do know which frequencies are present in the original signal, but there is no way of knowing how much of a frequency contributes to the odd and even (imaginary and real) part of the signal.

3.2 Inverse Fourier Transform

Another important aspect of the Fourier transform is that it is reversible. The inverse transform is almost identical to the forward transform with just a few differences. Since it is the reverse the output will be in the time-domain and the values are calculated by taking the integral with respect to the frequency-domain, where one multiplies frequency samples with a complex term, which in this case is without the subtraction sign. The equation described can be seen in figure 4.

$$g(t) = \int_{-\infty}^{\infty} G(f)e^{j2\pi ft}\,df \quad (4)$$

3.3 Discrete Fourier Transform

As mentioned earlier, the DFT is probably the most used variant of the Fourier transforms due to the fact that all real world signals are sampled at a given rate. If we take a look at the discrete forward transform shown in equation 5; we see that it is similar to the equation for the standard FT, except for the fact that we use a sum over a range of samples (w), and that we use the parameter $\frac{d}{w}$ instead of the frequency f, meaning $\frac{d}{w}$ works as a *discrete frequency*[1, p. 277]

$$G(k) = \sum_{x=0}^{w-1} g(x)e^{-j2\pi x\frac{k}{w}} \quad (5)$$

The DFT of the signal is the sum of all the samples from the original signal g(x) [$x = 0, x = 1, ..., x = (w-1)$] multiplied with the same complex exponential where k is the only "unknown" variable, which yields the discrete transformed signal $G(k)$.

3.3.1 Sampling and Aliasing. It is important to mention that data can be lost when you are sampling a real world continuous signal. The **Nyquist–Shannon sampling theorem** tells us that you will only preserve the correct information for frequencies up to half the sampling rate [1, p. 276]. The frequencies which are of higher rate than half the sampling rate, will get an *alias* meaning that that frequency will be disguised as another, and data from the original signal will be lost.

3.4 Inverse Discrete Fourier Transform

As for the standard form of the Fourier transform, the inverse DFT is almost identical to its forward transformation. The first difference to notice is that for the inverse there is used a scaling factor $1/w$. The scaling factor makes sure that the two transforms are inverses of each other [1, p. 278]. In the next part of the equation, frequency-domain values are multiplied together with the exponential part which is equal to the exponential in the DFT, except that the subtraction sign is removed. The equation for the inverse DFT can be seen in equation 6.

$$G(k) = \frac{1}{w}\sum_{k=0}^{w-1} G(k)e^{j2\pi x\frac{k}{w}} \quad (6)$$

Another important property of the inverse DFT that's worth mentioning is that if the results we obtain for g(x) are real, then the imaginary part of g(x) will be equal to 0 [1, p. 279].

3.5 2D Discrete Fourier Transform

First we focused on one dimension, in this section one will move on to the 2D DFT. Earlier the phrase time-domain was used, but when working in the 2D-domain, we refer to it as the spatial-domain. It is also the 2D DFT that is used when working with images. As with many equations mentioned, the DFT and the 2D DFT are quite similar. The main difference is that we have to go through both the width and the height of the image. This also results in different frequencies for the two directions. The formula [1, p. 289] for the 2D DFT can be found in the equation 7. For the inverse 2D DFT the differences are the same as between 2D DFT and 2D DFT inverse as we discuss in the inverse DFT section. The formula can be found in equation 8.

$$G(k_x, k_y) = \sum_{x=0}^{w-1}\sum_{y=0}^{h-1} G(x, y)e^{-j2\pi x^T f} \quad (7)$$

$$G(x, y) = \frac{1}{wh}\sum_{k_x=0}^{w-1}\sum_{K_y=0}^{h-1} G(k_x, k_y)e^{j2\pi x^T f} \quad (8)$$

where $x = [x\ y]^T$ $f = [\frac{k_x}{w}\ \frac{k_y}{h}]^T$.

There also exists an important property for the 2D DFT which can be utilized to reduce run time. The property is separability. This means that we can compute values for the rows first, and then send these results as input for the computations of the columns to get the final results. It is also important to mention that the order of the calculations does not matter.

3.6 Fast Fourier Transform

The FFT is a method for calculating the DFT more efficiently. The DFT has a runtime of $O(w^2)$ whilst the FFT has a runtime of $O(w \ \log w)$ [4] [1, p. 278]. The reduction in runtime and computations needed for the calculations is achieved by the mathematical fact that waves of sin and cosine functions will overlap in a predictable way [4]. With this prior knowledge, if you know that n-number of frequencies are to overlap in a given position, you can perform a calculation that includes all of these frequencies instead of having to do the calculation once per frequency thus reducing the number of computations [4].

Going from $O(w^2)$ to $O(w \ \log w)$ may seem like a small improvement, but for large images or signals containing a large number of samples, this has a drastic effect on computation time. This is the reason the FFT is the superior algorithm, and is what is used and has a wide variety of applications across multiple fields.

4 IMPLEMENTATION

This section gives a detailed description of the implementation of this project.

4.1 Discrete Fourier Transform

In this project, we wanted to be able to compare one of the basic algorithms for the DFT to the FFT. The algorithms implemented are found in "Image Processing and Analysis" by Stan Birchfield in chapter 6 [1, chp. 6] and will be described in detail below.

The first algorithm implemented was the 1D DFT. The function takes in an array-like object which represents the 1D signal and is defined by g. Next there are created two lists. One for the even and one for the odd output values. After this w is defined which is the number of samples. Two loops are then created. The first represents the output values, and the second the input values. When calculating the output values frequencies are derived by taking the index of the nth output value and dividing by the number of samples. For each output sample we go through all input samples. For even values the input sample is multiplied by the cosine term and added to the output, and for odd values it is the same, but with sinus and subtraction. The output is then an array the same size as the input g with real and complex values for each index. The function DFT_1D can be seen in 1.

```
def DFT_1D(g: npt.ArrayLike) -> npt.ArrayLike:
    G_even = [0 for i in range(len(g))]
    G_odd = [0 for i in range(len(g))]
    w = len(g)
    for k in range(w):
        f = k/w
        for x in range(w):
            G_even[k] = G_even[k] + g[x]*math.cos(2*np.pi
    *f*x)
            G_odd[k] = G_odd[k] - g[x]*math.sin(2*np.pi*f
    *x)
    return np.array([complex(g_ev, g_odd) for g_ev, g_odd
      in zip(G_even, G_odd)])
```

Listing 1: DFT_1D function

The second algorithm implemented was the 1D DFT inverse. As in the previous function this one takes in an array-like object G, but here it contains frequency-domain values which are both real and complex. w is again defined as the length of the samples, and a list called g_{real} is defined for the output values. Since this is the reverse the values signal values are in the outer loop, and the frequency values are in the inner loop. Other than a little restructuring the process is identical to the DFT except we now we have a negative sign for the exponential, and we use the scaling factor $\frac{1}{w}$. The function called DFTI_1D can be seen in 1.

```
def DFTI_1D(G: npt.ArrayLike) -> npt.ArrayLike:
    w = len(G)
    g_real = [0 for i in range(w)]

    for x in range(w):
        for k in range(w):
            f = k/w
            # Exponential part with eulers rule for cos
     and sin
            ex = (math.cos(2*np.pi*f*x)+1j*math.sin(2*np.
     pi*f*x))
            g_real[x] += ex*G[k]

    return np.array([(1/w)*g for g in g_real])
```

Listing 2: DFTI_1D function

Next we move on to the DFT 2D. This function takes in a ndarray which in our case is a 2D image. We first extract the width w and the height h from the ndarray. Secondly two ndarrays are created G and $temp_G$ with size (w,h). As mentioned in the theory the 2D DFT is separable, which has been taken advantage of in this implementation. First the DFTs are calculated with the 1D function for columns and are then added to the temporary ndarray $temp_G$. Next we perform the DFTs by row with the values from $temp_G$ as input. The final values are then added to the ndarray G and returned. The function DFT_2D can be seen in 3.

```
def DFT_2D(g_img: npt.NDArray) -> npt.NDArray:
    w,h = g_img.shape
    G = np.zeros((w,h), dtype = 'complex_')
    temp_G = np.zeros((w,h), dtype = 'complex_')
    for y in range(h):
        temp_G[:,y] = DFT_1D(g_img[:,y])

    for x in range(w):
        G[x,:] = DFT_1D(temp_G[x,:])
    return G
```

Listing 3: DFT_2D function

The last function which is used for inverse 2D DFT is almost identical to the 2D DFT. The only difference is that it takes in the 2D frequency domain instead of the spatial-domain values, and it uses DFTI_1D inverse transform function instead for the DFT_1D. The result is then a ndarray of the spatial-domain values of the image, and the function named DFTI_2D can be seen in 4.

```
def DFTI_2D(G: npt.NDArray) -> npt.NDArray:
    w, h = G.shape
    g = np.zeros((w,h), dtype = 'complex_')
    temp_g = np.zeros((w,h), dtype = 'complex_')
```

```
    for y in range(h):
        temp_g[:,y] = DFTI_1D(G[:,y])
    for x in range(w):
        g[x,:] = DFTI_1D(temp_g[x,:])
    return g
```

Listing 4: DFTI_2D function

4.2 Blur Detection Algorithm

```
def blur_detector(
    image: List[List],
    thresh: float = 8,
    size: int = 50,
    dft: Callable = np.fft.fft2,
    idft: Callable = np.fft.ifft2) -> Dict[str, Any]:
```

Listing 5: Input parameters for blur detection algorithm

The blur detector algorithm takes in a grayscale-image, a threshold that lets us classify an image as either blurry or not, *size* which is the size of the filter that is used to filter out the lower frequencies of the frequency band, more specifically the number of pixels from the center of the filter to either of the edges. The last input parameters are the DFT and inverse-DFT functions we want to use for the algorithm. The default threshold is set to 8, and the size parameter is set to 50, this seemed to work for most images. It is important to note that the original idea of implementing this method was found in a blog post by Adrian Rosebrock [3].

The algorithm follows all of the steps discussed in 5.2

(1) Take DFT of the image
(2) Shift DC component to the middle
(3) Remove lower frequencies from the transformed image
(4) Shift DC component back to the top left corner
(5) Reconstruct the image by using the inverse DFT
(6) Take the magnitude of the reconstructed image
(7) Take the mean from the image in the previous step
(8) Classify as blurry if mean is under the threshold, non-blurry if mean is above the threshold

```
    return {
        'is_blurry': score < thresh,
        'score': score,
        'step_images': step_images
    }
```

Listing 6: Output from blur detection algorithm

The output is a dictionary with keys 'is_blurry', 'score', and 'step_images'. **is_blurry** is a boolean for if the score (mean mentioned in point 8 above) is below the threshold, **score** is the mean, **step_images** gives you a list of images showing what is happening throughout the algorithm. Some of which are available in the appendix. If you are interested in viewing our full code including other files, you can take a look at our GitHub repository [2].

5 EXPERIMENTS AND RESULTS

5.1 Runtime Experimentation with DFT vs FFT

Since our project is based on the 2D DFT, we wanted to do experimentation with some of its different algorithms. To do this we implemented versions of the DFT as mentioned in implementation, and the goal was to get a better understanding of the DFT, as well as confirming some well known aspects of the run time for the algorithms.

Algorithm Name	Mean time(s)	SD σ(s)	Runs	Loops
fft2(numpy)	0.000599	0.000055	7	1000
DFT_2D	45.1	0.272	7	1
ifft2(numpy)	0.000678s	0.000138	7	1000
DFTI_2D	13.8	0.360	7	1

Table 1: Run times for DFT algorithms.

The run time results from the experiments can be seen in table 1. As expected the FFT algorithm was faster for both the forward and backward transformation.

5.2 Blur detector

For the initial stage of the blur part of the project, we wanted to see the relationship between how blurry an image is, and its Fourier transform. We knew that a blurry image would most likely have less of the higher frequencies and that we wanted to build upon that.

We started by taking a single image, and applied average-filters of different sizes to it, in order to get the same image with different levels of blur (see A.4).

We took the Fourier transform of said images and looked at the magnitude of the frequency domain. We could clearly see that the less blurry the image was, the more of the information was stored in the higher values of the frequency domain (further away from the DC component) (figure A.5).

We then removed the higher frequencies from the frequency domain (as shown in figure with filter) and reconstructed the images (figures A.6 and A.7).

After reconstructing the images (figure A.5), we are able to see that there are more edges present in the images which were less blurry. Since the edges present in the reconstructed image are made of both lower and higher pixel-values (the lower values can be negative at this stage), we take the magnitude of all pixel values in the reconstructed image before taking the mean across the image. This mean score will give a good indication of how "edgy" the original image is. This first part of the experiment laid the foundation of our blur detecting algorithm (section 4.2).

After having implemented the algorithm, we took some images with both an edgy background and some with a more plain background and tried to recreate blurry images by moving our camera (phone) as we took the image (see A.13). We did some analysis on these photos (see "code/blur_detection_results.ipynb" in our GitHub [2] for the complete code) and noticed both the edginess of the image, combined with the resolution of the image had a great impact on the mean score which was used to determine if the image was blurry or not. The hyper parameters (size and threshold) in 4.2 were tuned based on square images ranging from 252x252 to 512x512 pixels. But for this experiment, we used images of size 2048x1536 and quickly discovered that the parameters needed some tuning which led us to use a size parameter of 100 for our experiments.

Images	Mean Scores
semi-blurry (edgy)	21.13
blurry (edgy)	16.73
sharp1 (edgy)	45.42
sharp2 (edgy)	37.22
sharp (plain)	18.93
blurry (plain)	5.34

Table 2: mean scores from detection algorithm

In table 2 you can see the mean scores for all the images, notice that the sharpest image with a relatively plain background compared to the mean score of a blurry image with an edgy background (18.9 vs 16.7).

6 CONCLUSION

In the result section, we see that the algorithm is able to differentiate between blurry and sharp images, but that the context of the image will have effect on the output. This makes us suggest the best course of action is to take reference images which are both sharp and blurry of approximately the same object and lightning, and tune the size and threshold parameters based on the output of the algorithm.

REFERENCES

[1] Stan Birchfield. 2016. *Image Processing and Analysis.* Cengage Learning.
[2] Ola Andre Flotve and Håvard Moe Jacobsen. 2022. BlurDetectionDFT. https://github.com/havardMoe/BlurDetectionDFT
[3] Adrian Rosebrock. 2020. OpenCV Fast Fourier Transform (FFT) for blur detection in images and video streams. https://pyimagesearch.com/2020/06/15/opencv-fast-fourier-transform-fft-for-blur-detection-in-images-and-video-streams/
[4] Veritasium. [n. d.]. The Most Important Algorithm Of All Time. https://youtu.be/nmgFG7PUHfo?t=886

A APPENDIX

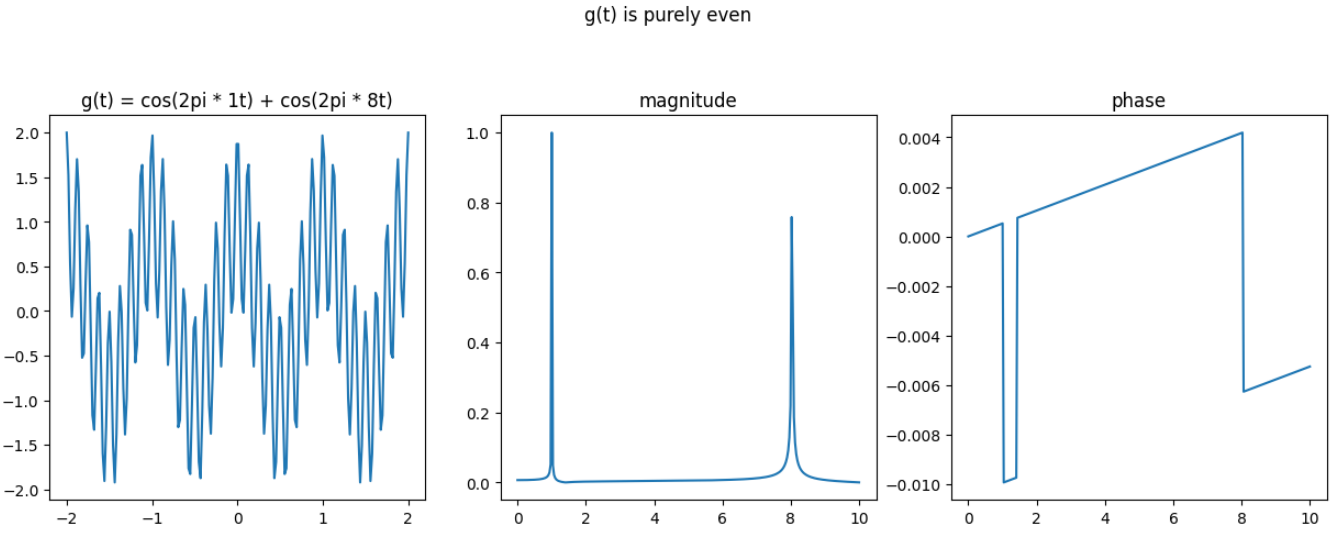

Figure A.1: Signal composed of only even frequencies

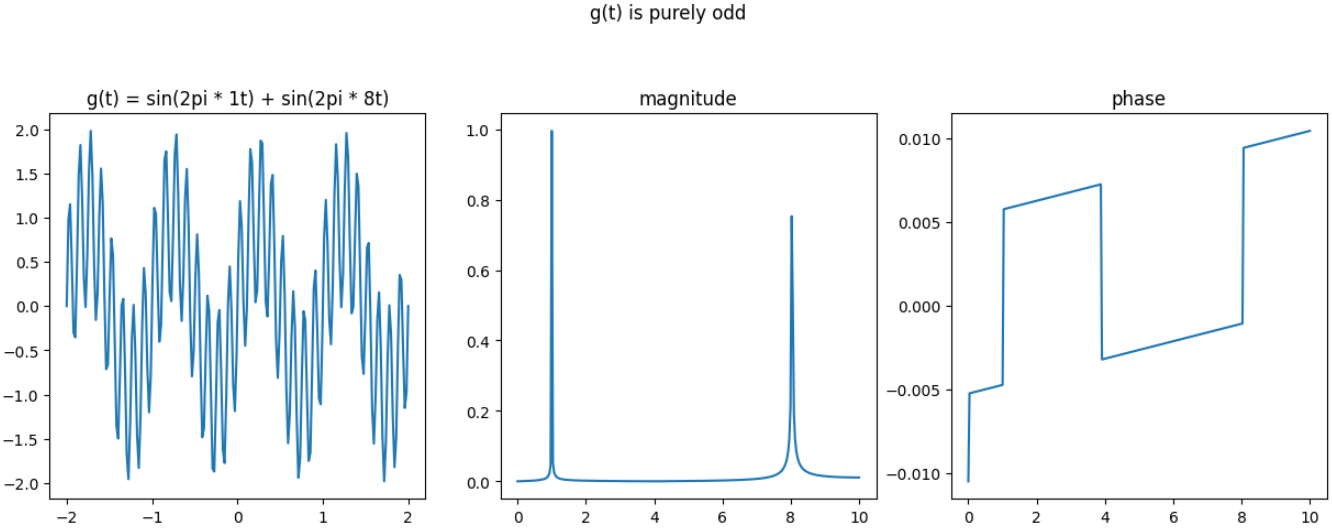

Figure A.2: Signal composed of only odd frequencies

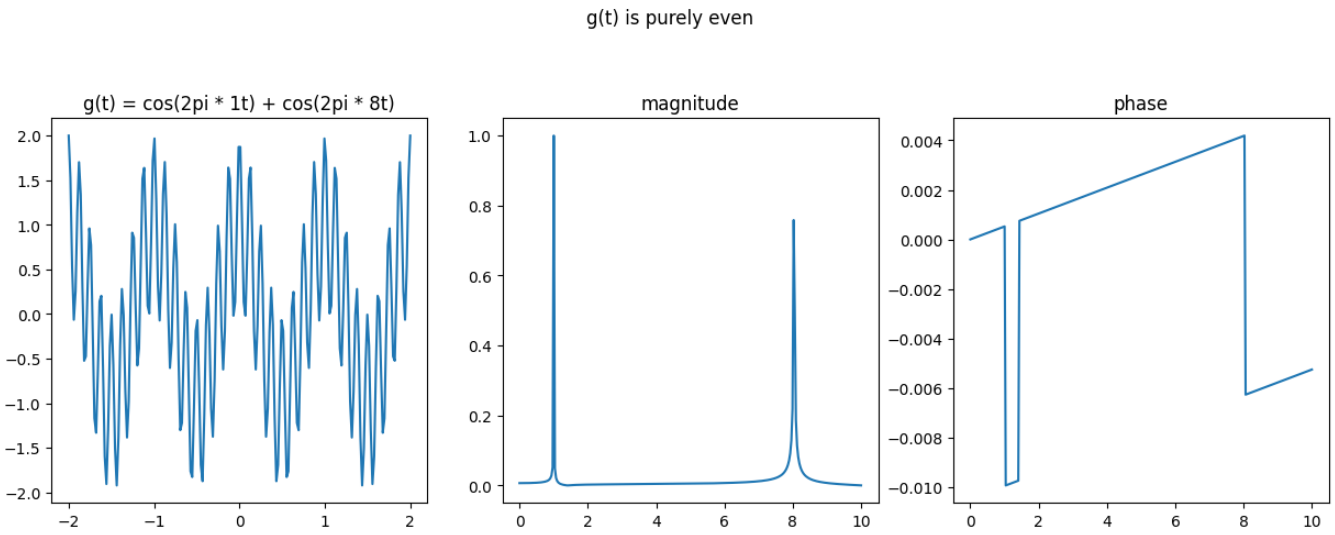

Figure A.3: Signal composed of only even and odd frequencies

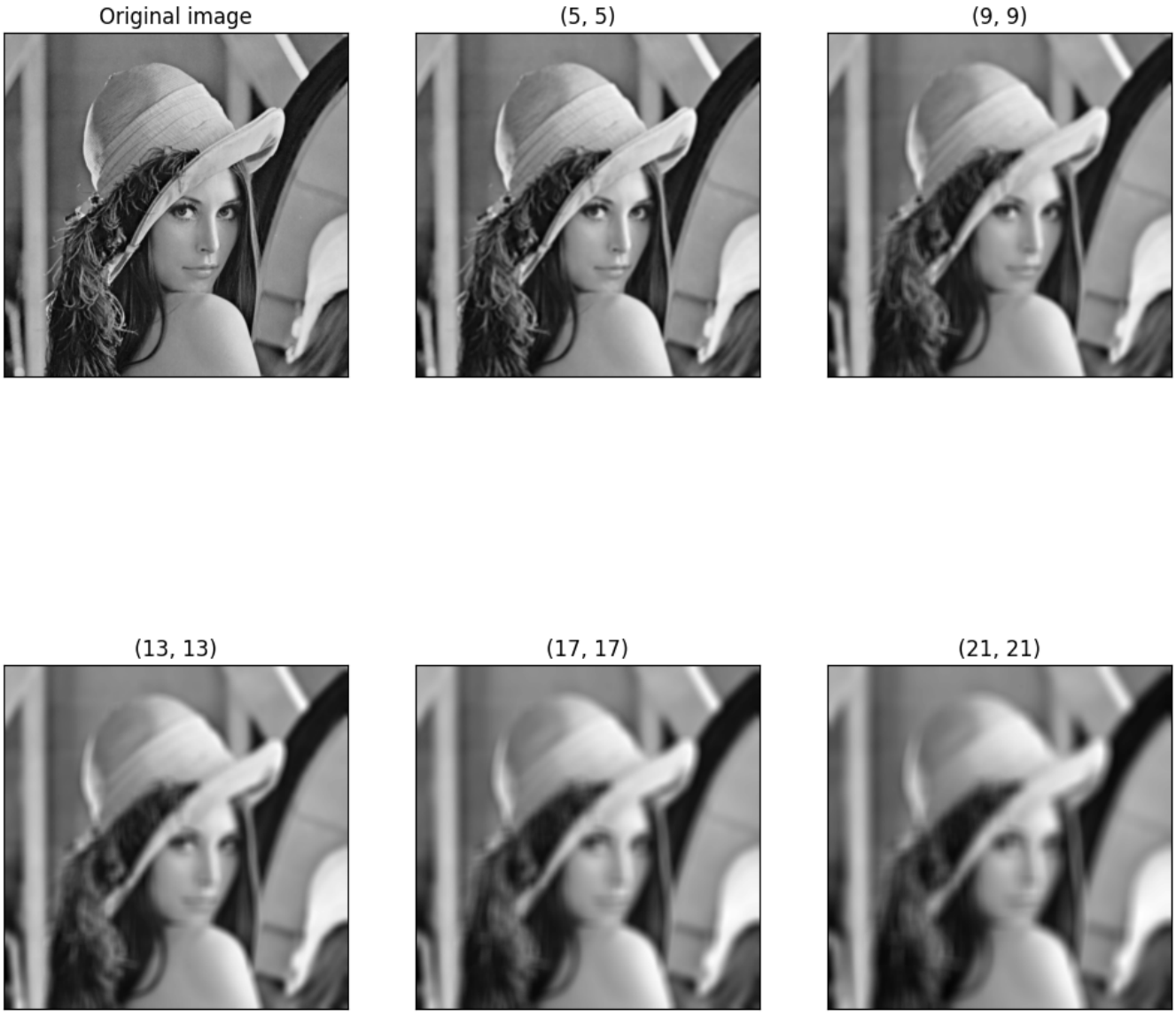

Figure A.4: Original image with different levels of blur

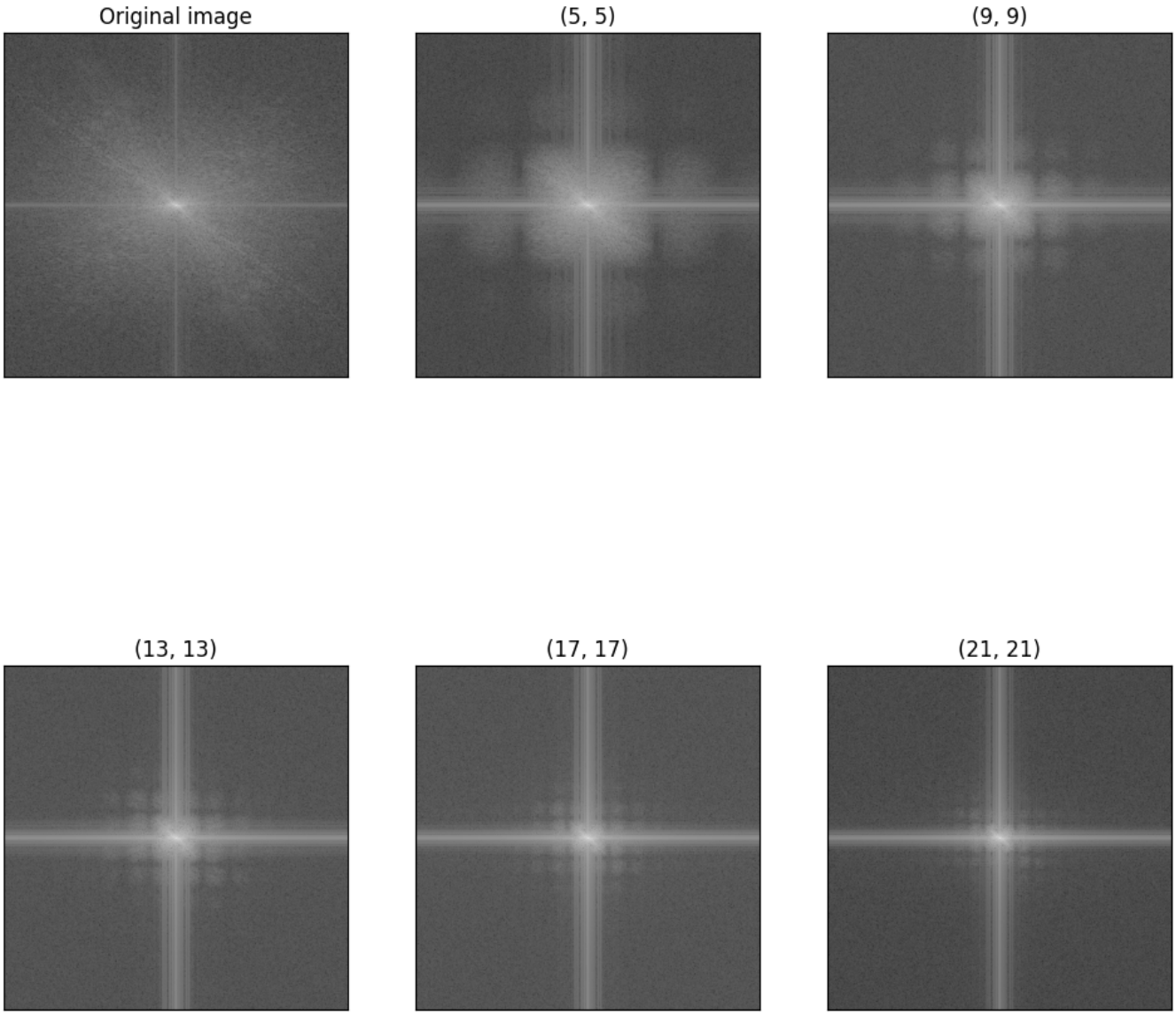

Figure A.5: Magnitudes of blurred images

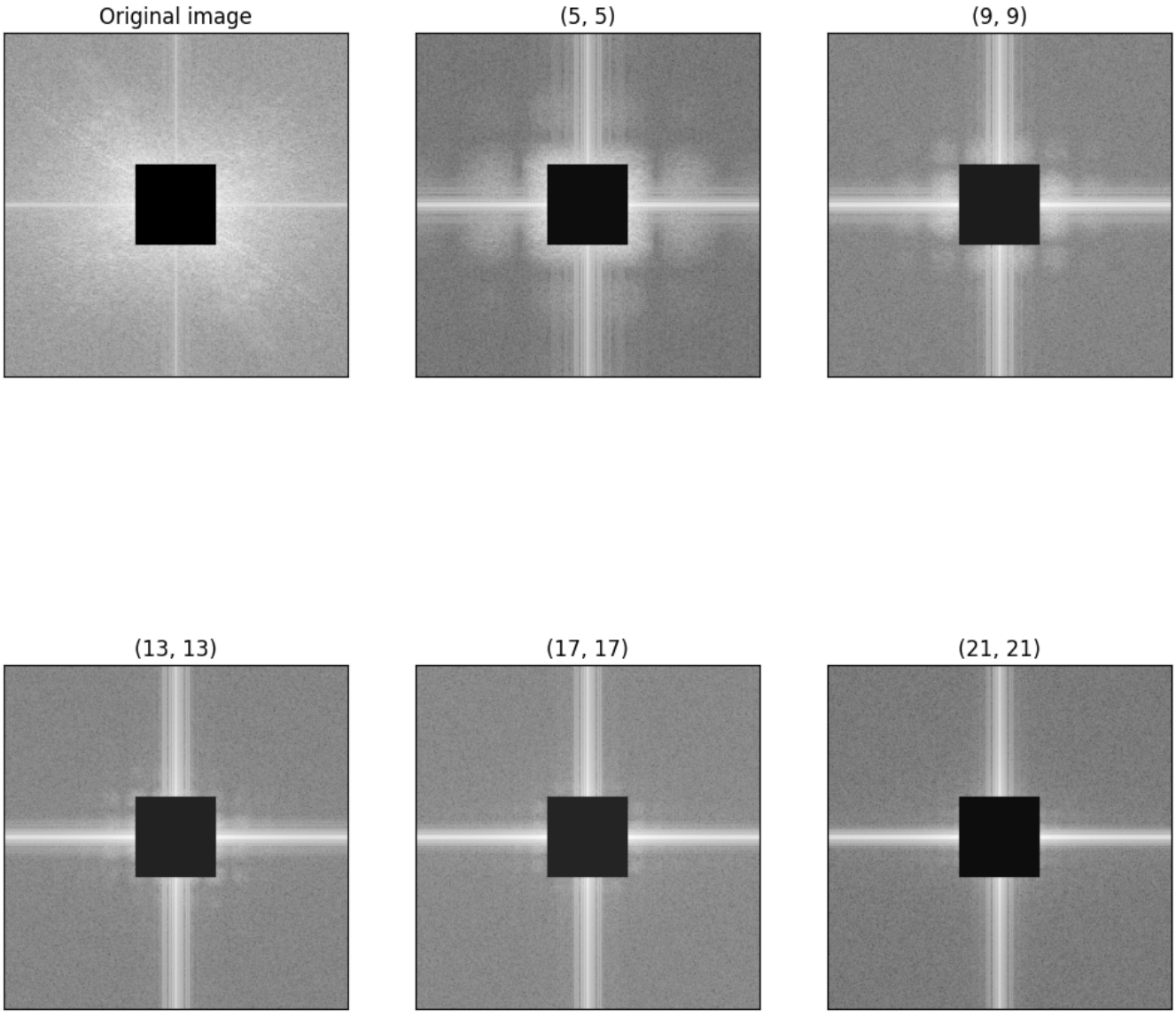

Figure A.6: Filtering the lower frequencies

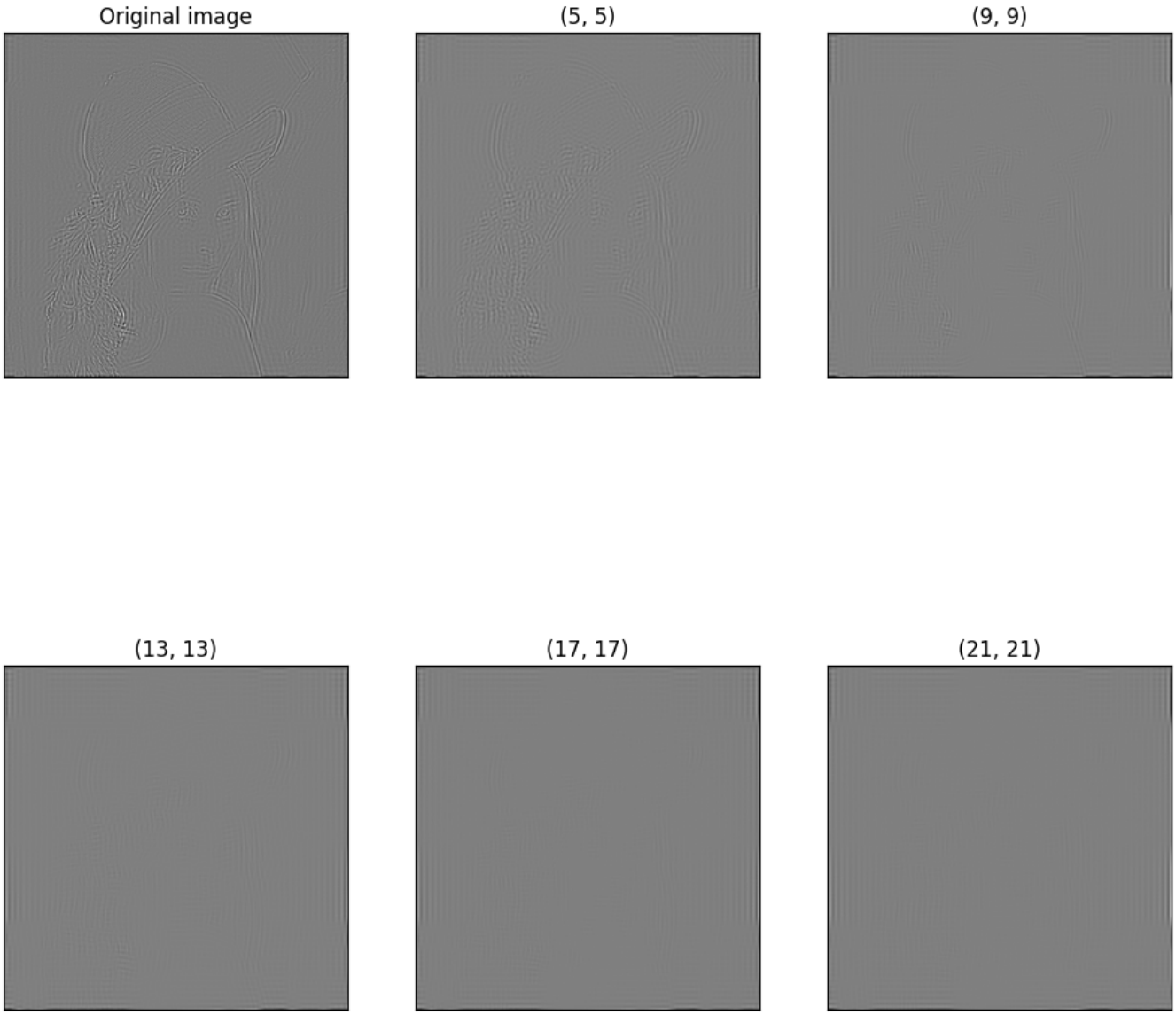

Figure A.7: Images reconstructed after filtering their magnitude

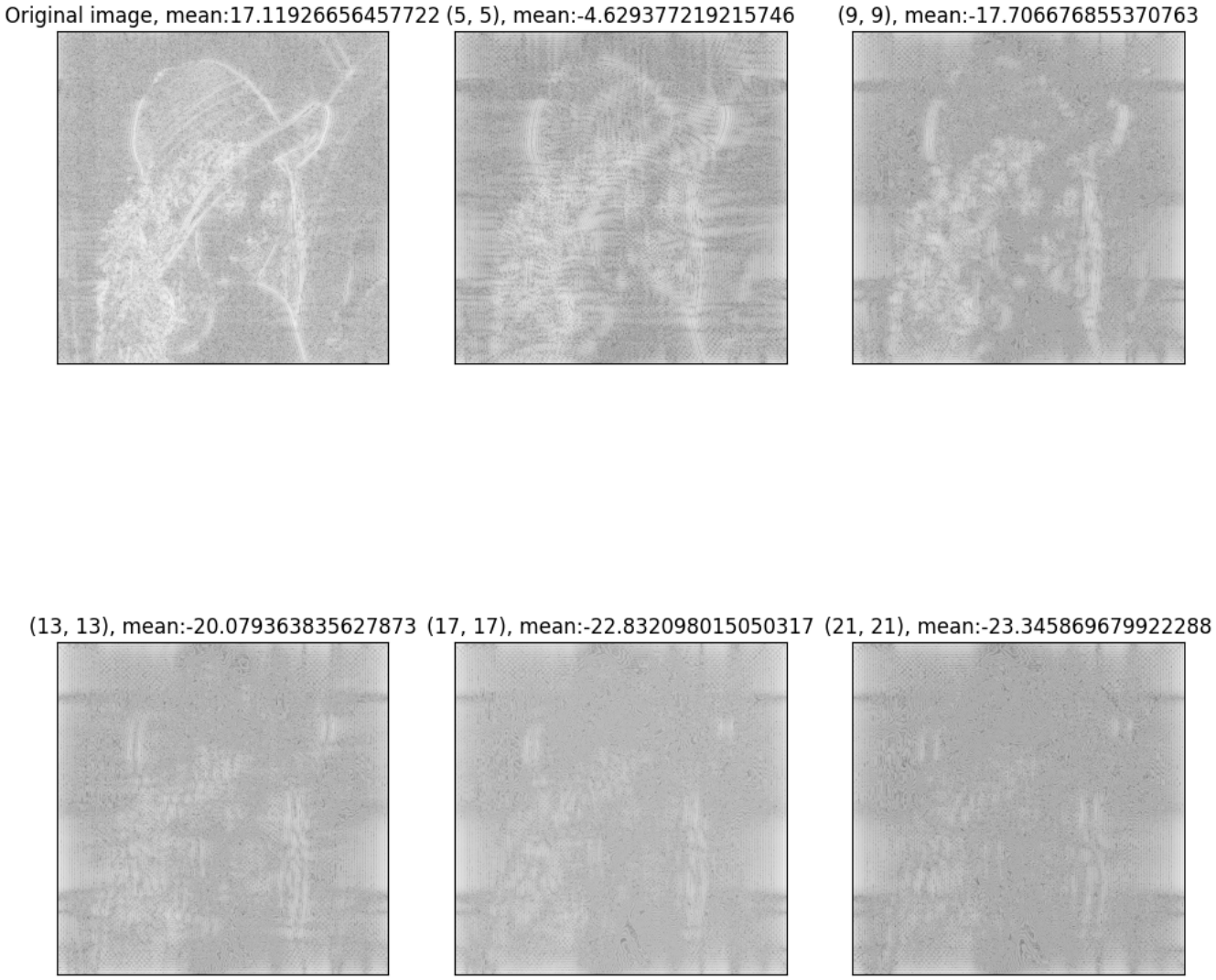

Figure A.8: Magnitude of reconstructed images

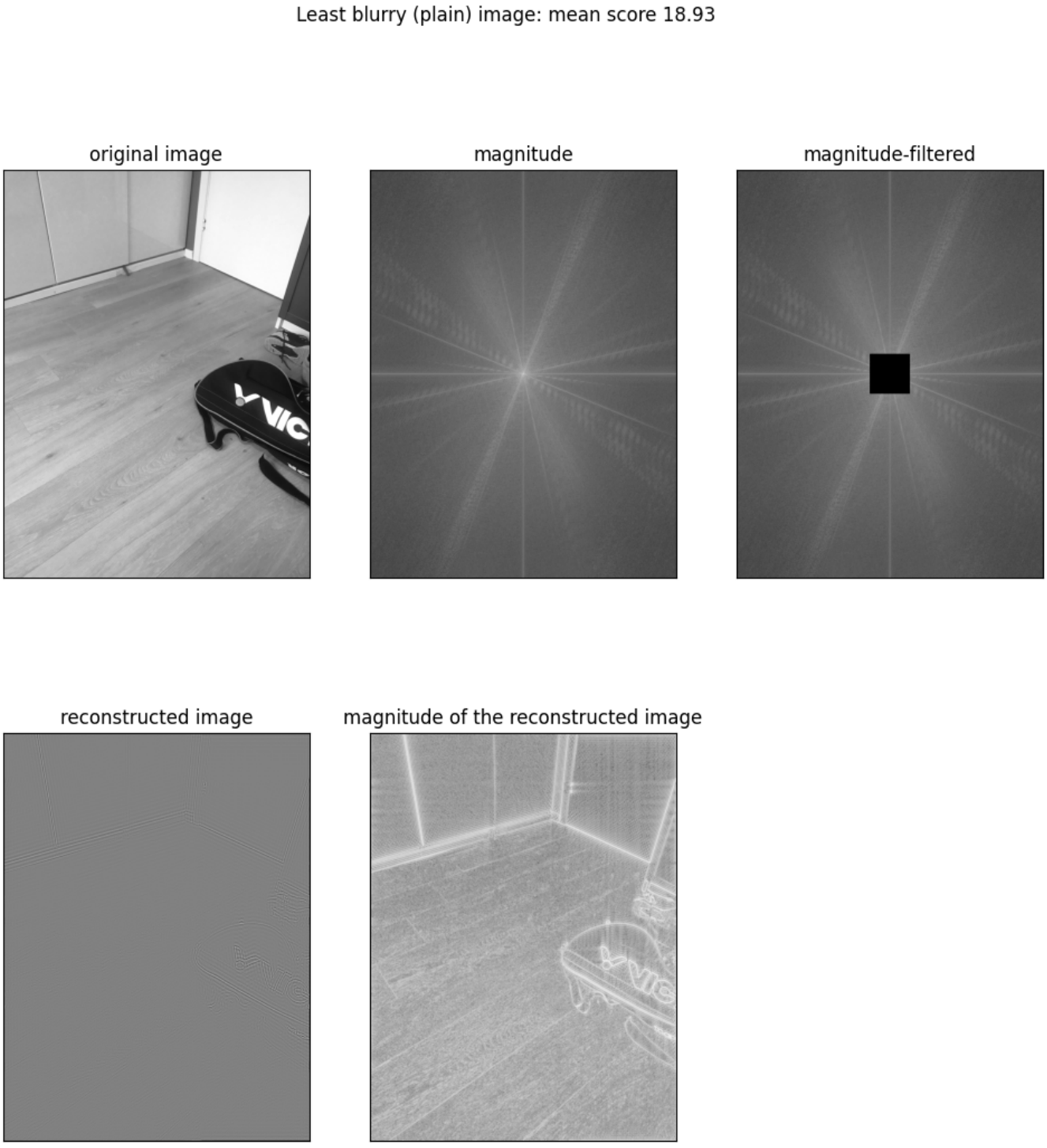

Figure A.9: The least blurry of the plain images

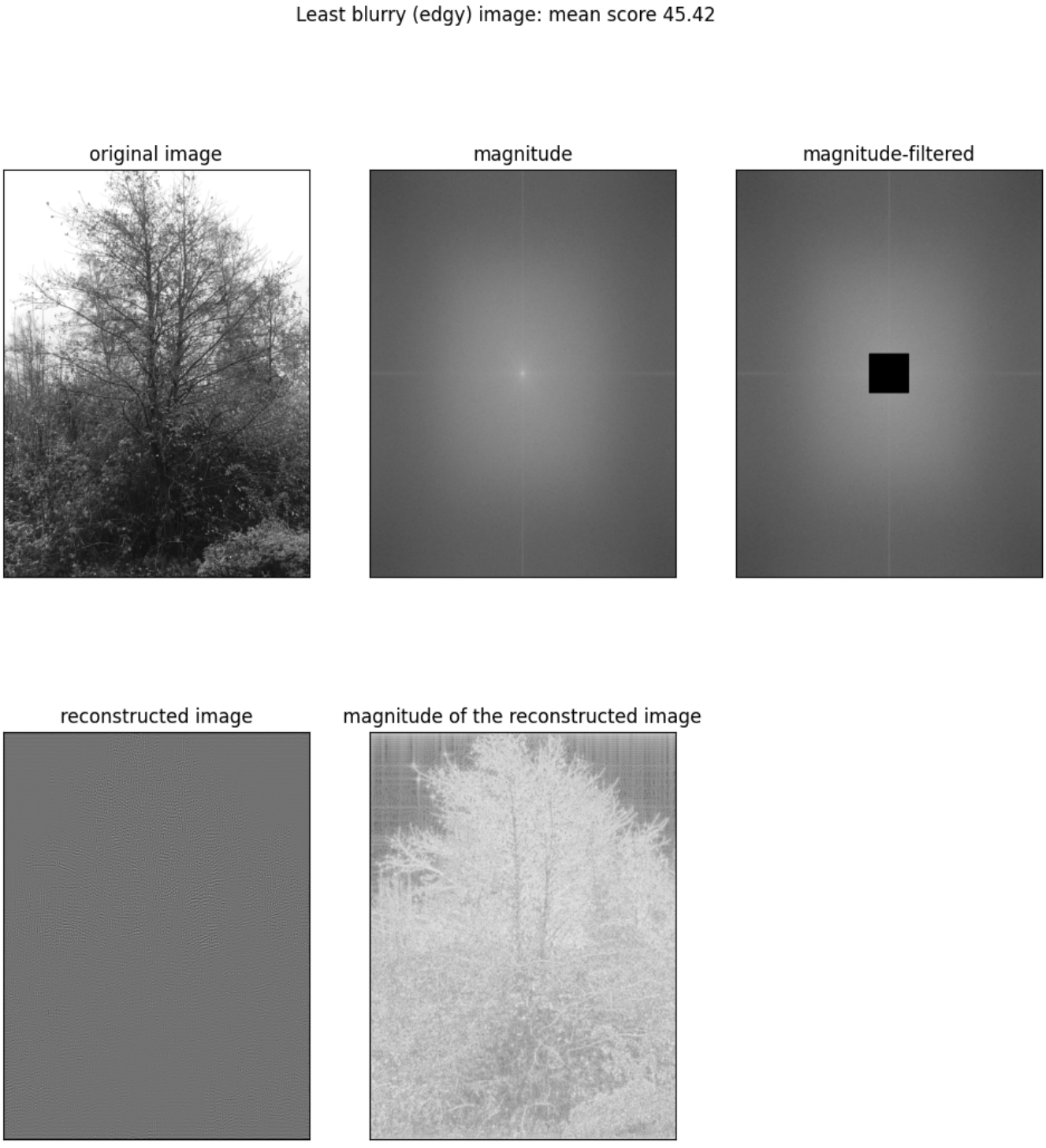

Figure A.10: The least blurry of the edgy images

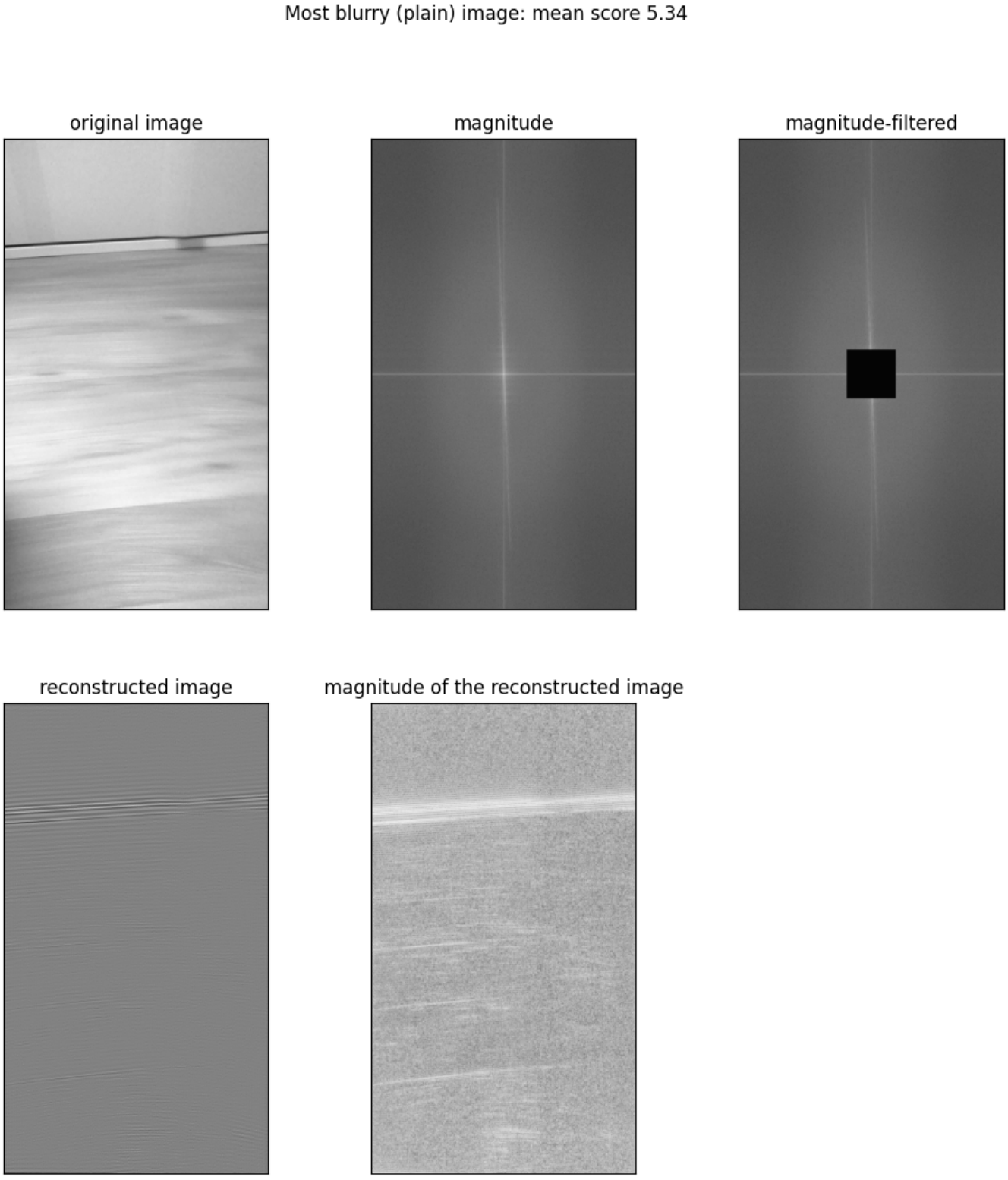

Figure A.11: The most blurry of the images with plain background

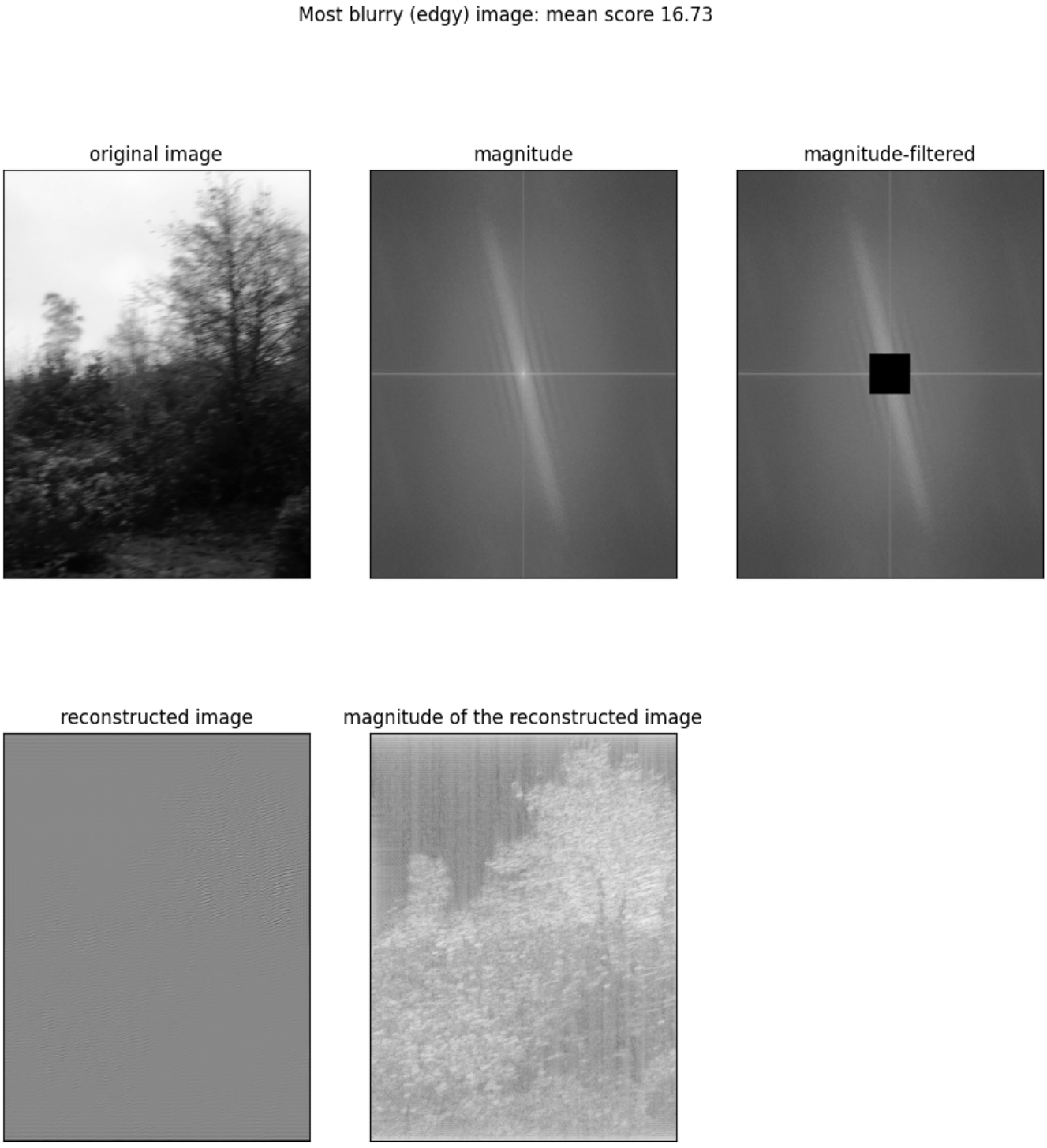

Figure A.12: The most blurry of the edgy images

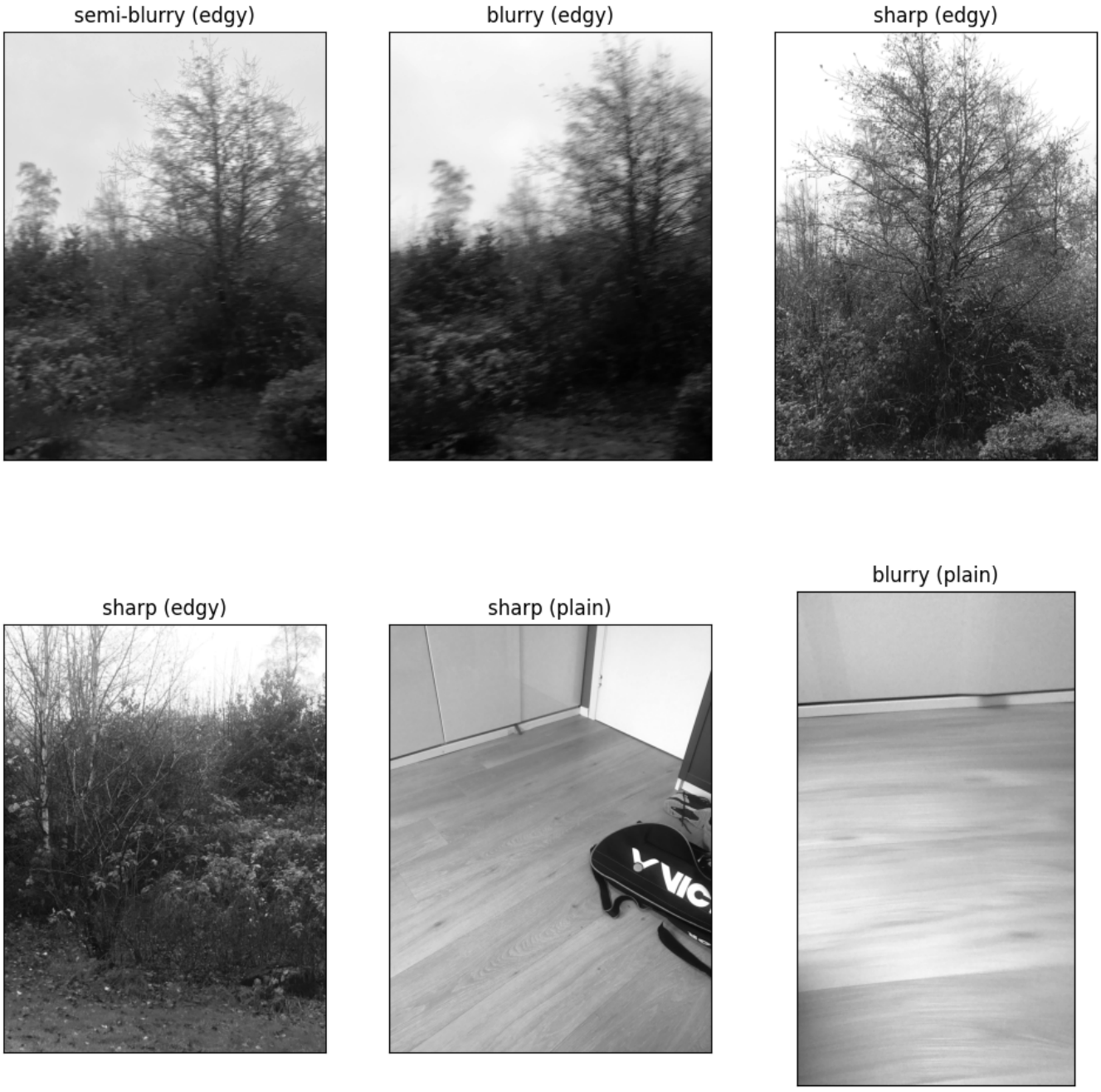

Figure A.13: The images used for experiment