Politecnico di Torino

Applied Signal Processing Laboratory

Assignment 3: Introduction to Audio and Image Processing

Francesco Laterza, 271087

28/05/2023

Contents

1 Introduction to digital audio processing

1.1 Exercise 1

This exercise is about the pitch estimation of a guitar note and of our voice.

1.1.1 Guitar note

We read the audio file of a guitar playing the A2 note, which correspond to 110Hz.

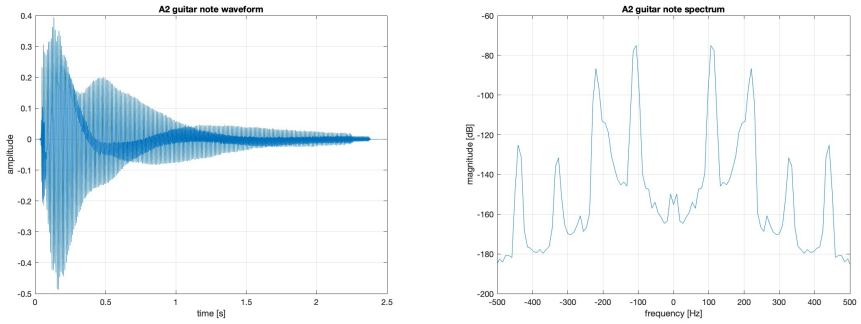

Figure 1: Waveform and spectrum of the A2 guitar note

To estimate the pitch we compute the auto-correlation of the signal. We then measure the time between the first two peaks after the zero. The inverse of the computed time difference will be the pitch of the note.

For this signal we get a value of 109.7015Hz, which is close to the expected value of 110Hz.

1.1.2 Recorded voice

Now I record my voice saying the vocal "o".

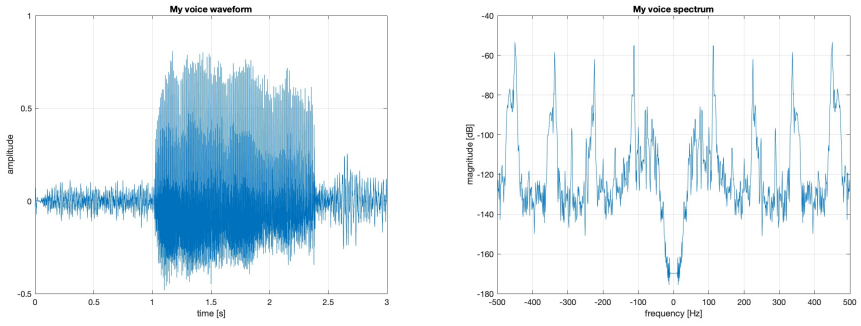

Figure 2: Waveform and spectrum of recorded voice saying the vocal "o"

The estimated pitch this time is 112.6761Hz. We now cut the signal to be sure to get exactly the part where to vocal is pronounced. We get the pitch as before equal to 112.6761Hz.

1.2 Exercise 2

In this exercise we consider the recording of the A major scale on a guitar.

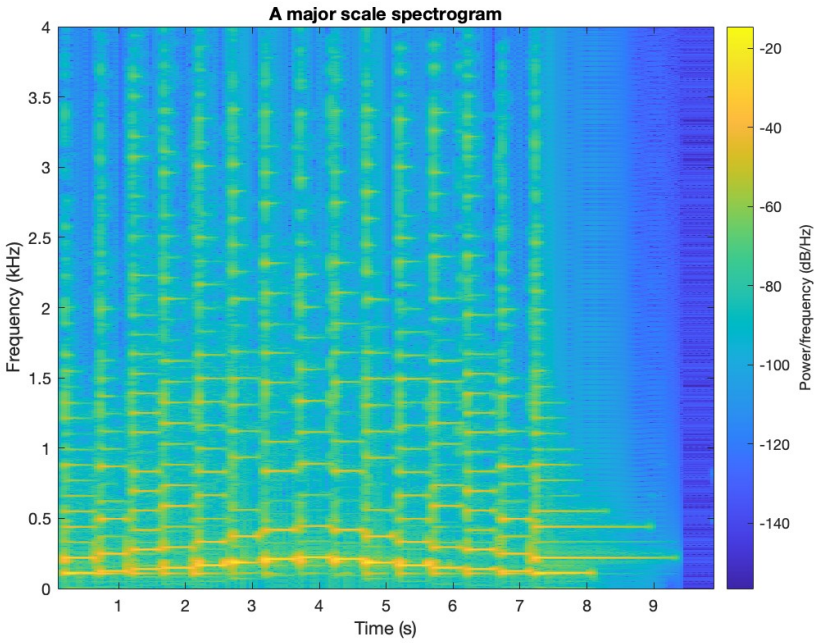

Figure 3: Spectrogram of the A major scale signal

In Figure 3 it can be observed the spectrogram of the A major scale signal. The notes are clearly visible and it is also possible to visualize the ADSR envelope.

1.2.1 Higher note pitch estimation

We isolate the part of the signal containing the highest note and we estimate the pitch. The resulting pitch is 222.7273Hz which is what we expected.

1.2.2 Custom function my_tuner

We now create a custom function *my_tuner* that takes as input the signal of a note, the sampling frequency and the number of samples and outputs the corresponding note name and the difference in Hz from the actual note.

Function implementation The function first computes the pitch of the given signal. Then, it computes the difference in Hz between the pitch and a note in order to find the note with the minimum distance.

Note that there is no need to save the frequency value of each note. Given a reference note it is possible to compute another note with the following formula

$$newNote = referenceNote \cdot 2^{i+j/12}$$

where i is the difference in octave and j is the difference in semitones.

It is enough to iterate between the values of i and j and map them to the correct note.

Testing First we test the function with the higher note used before. We get as result that the note is A3 and the difference from the actual note is 2.7273.

Then, we test the function with three notes recorded on a guitar. The results are:

- note: F6 - distance: -18.7879
- note: $(\mathrm{F}^{\#}/\mathrm{B}^{\mathrm{b}})6$ - distance: -9.9777
- note: C6 - distance: 29.1075

1.2.3 Filtered signal

Let's filter the signal with a low-pass FIR filter having cutoff frequency 5kHz, transition band 500Hz and attenuation of at least 50dB, thus we use an Hamming window.

We under-sample the resulting signal by a factor of 4.

We repeat the filter setting the cutoff frequency to 2.5kHz and under-sample by a factor of 8.

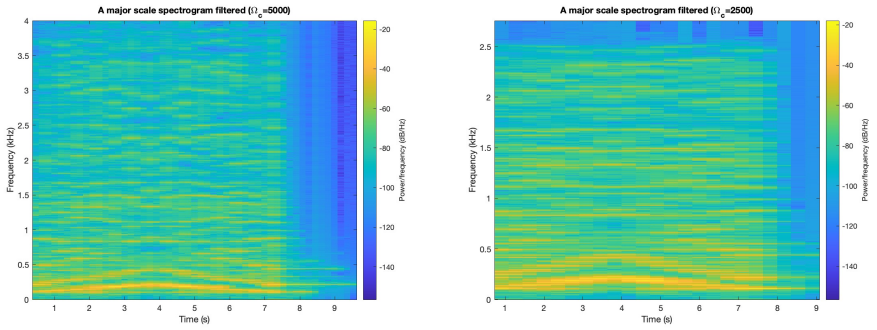

Figure 4: Spectrogram of the A major scale signal filtered

In Figure 4 we can see the spectrogram of the 2 filtered signals. It is clearly noticeable the reduction in resolution due to the under-sampling. In the second spectrogram we can see how frequencies above 2.5kHz are cut.

If we listen to the audio of the filtered signals, we can notice very easily the lack if high tones.

1.3 Exercise 3

In this exercise we write a custom function that computes the spectrogram of a given signal. The function takes as inputs a vector x, the window size M, the overlapping number of samples L, the number of samples of the FFT N and a flag that indicates if a plot as to be displayed. The output of the function will be the spectrogram matrix S, the frequency axis f and the time axis t.

1.3.1 Function implementation

First we check if the number of samples of the window is odd, otherwise we make it odd.

Then the window function is computed as a Hamming window. We compute the hop size R and the number of time frames T. Now we have all the elements to initialize S of dimensions $(N/2 \times T)$.

Now, we multiply each frame by the time window and compute the FFT for each time frame. We put each FFT to the corresponding column of the S matrix.

1.3.2 Function testing

We test our function comparing it with the results from the built-in Matlab function *spectrogram*.

In order to do thet we use the signal from the *A major scale*.

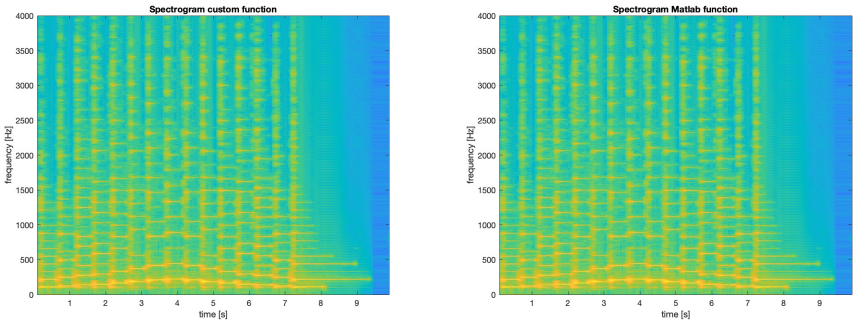

Figure 5: Comparison of the spectrogram of the A major scale signal

In Figure 5 it is possible to visualize the result of the comparison. The two spectrogram perfectly match.

2 Digital audio synthesis

2.1 Exercise 4

In this exercise we will generate sound by additive synthesis. We create a custom function *fnote* that takes as inputs a vector of amplitudes, a vector of frequencies, a vector of phases, a duration and the sampling frequency and outputs the tone and the time vector.

2.1.1 Function implementation

First we check that the amplitude and frequency vector have the same number of elements. We also check if the phases vector has the correct number of elements, if it has more we truncate it, if it has less we zero-pad it.

To generate the tone it is enough to sum for each amplitude, frequency and phase a sinusoidal function

$$x(t) = A\cos(2\pi ft + \varphi)$$

and then normalize it by its maximum value.

2.1.2 Vowels sound

We use the Helmholtz table that suggest the loudness of each harmonic to sum to generate vowels similar to human voice. We than concatenate all vowels to generate a sound resembling "AEIOU". The audio file is saved with the name *aeiou_tone.wav*.

2.1.3 Bell sound

Now we reuse the custom function to generate the bell sound. This time we modulate the resulting tone by the exponential function $e^{-t/\tau}$ with $\tau = 0.9$s. The audio file is saved with the name *bell_tone.wav*.

2.2 Exercise 5

In this exercise we generate the sound of an alarm and of a bell by the use of FM synthesis, thus by generating a sinusoidal signal with phase varying in time.

2.2.1 Alarm sound

The alarm sound is generated as

$$y(t) = A\cos[2\pi f_c t + I_0 \sin(2\pi f_m t)]$$

where $f_c = 1760$Hz, $f_m = 4.4$Hz, $I_0 = 120$ and $A = 1$.

In Figure 6 it is shown the spectrogram of the generated alarm signal. It can be visualize how the frequency varies in time with a sinusoidal behaviour as expected.

2.2.2 Bell sound

The bell sound instead is generated as

$$y(t) = Aa(t)\cos[2\pi f_c t + I_0 a(t)\sin(2\pi f_m t)]$$

where $f_c = 200$Hz, $f_m = 280$Hz, $I_0 = 10$, $A = 1$ and $a(t) = e^{-t/\tau}$ with $\tau = 2$s.

The difference with the previous signal is that also the amplitude and the intensity of the phase modulation varies in time.

In Figure 7 the spectrogram of the alarm signal is plotted.

2.3 Exercise 6

In this last exercise we are asked to generate a simple chord progression by means of subtracting synthesis.

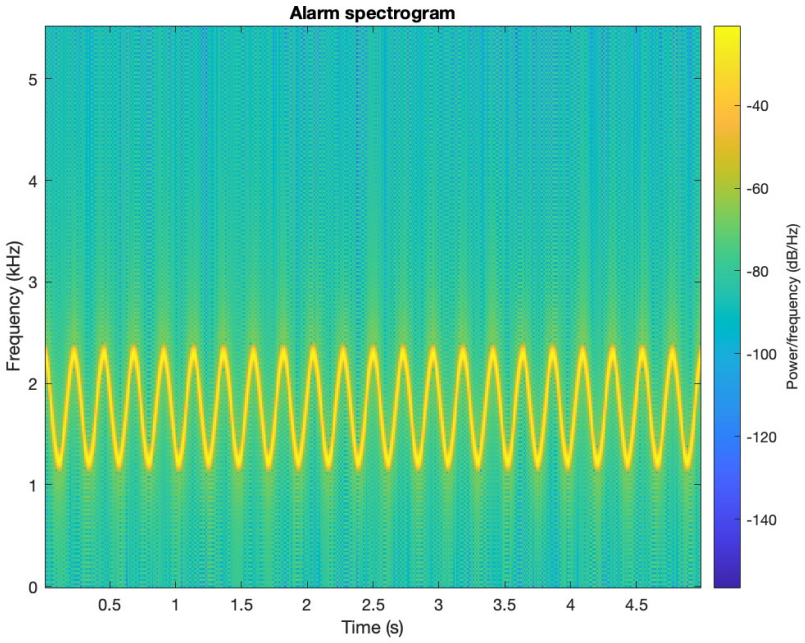

Figure 6: Spectrogram of the alarm signal

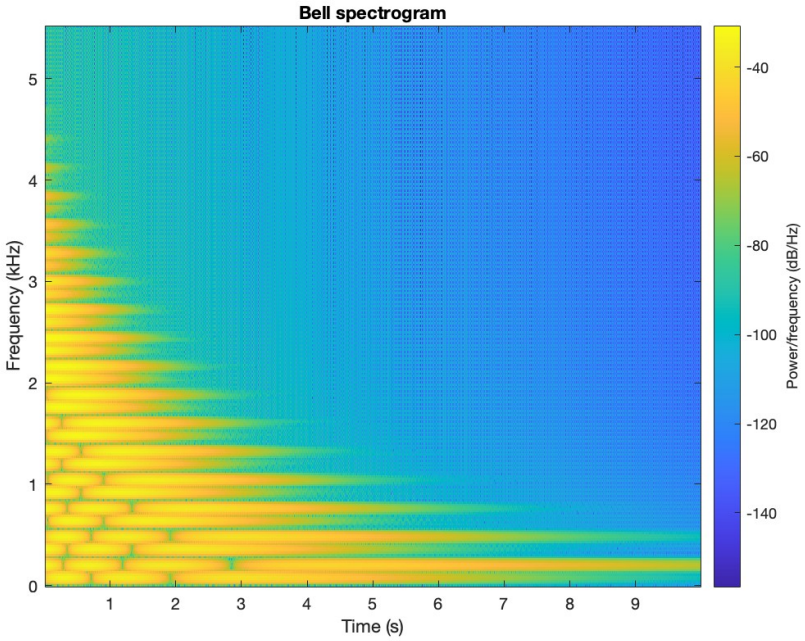

Figure 7: Spectrogram of the bell signal

2.3.1 Note generation

The first step is to generate a note. In order to do that we generate a sawtooth with frequency equal to the fundamental frequency of the note.

We modulate the signal by a PWM with lfo equal to the fundamental frequency divided by 240.

Lastly we filter the signal by a lowpass FIR filter.

2.3.2 Chord generation

Next we generate two kind of chords by adding together three notes. We get the major triad by adding the root note and two other notes at 4 and 7 semitones of distance. The minor triad instead has the other notes at 3 and 7 semitones of difference.

2.3.3 ADSR envelope

Finally, we generate an ADSR envelope typical of an instrument. To achieve a realistic result, we define the points giving the shape to the envelope and interpolate them with a cubic method to smooth the edges.

2.3.4 Progression

It is now enough to sum together the chord generated to get the progression. The audio file has been saved with the name *progression.wav*

2.4 Exercise 6 (extra)

In the Matlab file *ex_3_6_extra.m* we tried to improve Exercise 6. We compose another progression, also adding a melody on top of it.

Moreover some parameters to generate the chords have been changed. In order to achieve a more piano like sound, the signal has been generated differently with a combination of sines and sawtooths.

There are 2 lfos used. One is used to distort the frequency at the beginning of the chord, the other one to distort the volume of the note.

Also the ADSR envelope and the transition band of the filter has been modified accordingly.

3 Introduction to image processing

3.1 Exercise 7

In this exercise we will apply dithering to a black and white image which consist in mapping the original image with shades of gray to a new image which pixels are either black or white.

Figure 8: Original image in black and white

Figure 9: Image after dithering (4×4)

3.1.1 Dithering 4x4

We define a dithering matrix as

$$D_4 = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

Since one pixel of the original matrix will be represented by a 4×4 matrix, we replicate each pixel 4 times for both dimensions in order to get a matrix 4 times as large.

We also expand the D_4 matrix copying it as many times as how many pixels are in the image.

This way we get the image and the dithering matrix of the same dimension and we can compare them element wise.

Since the dithering matrix has values from 0 to 15, we need to normalize the image matrix to get values in the same range. Then comparing the two matrices if the value of the image is higher than D_4 we set the value to 1, otherwise 0.

In Figure 9 we can see the result of the dithering obtained by a matrix 4×4.

Figure 10: Image after dithering (8×8)

3.1.2 Dithering 8x8

Now we repeat the procedure but this time with a matrix 8×8. The matrix is obtained as follows

$$D_8 = \begin{pmatrix} 4D_4 & 4D_4 + 2 \\ 4D_4 + 3 & 4D_4 + 1 \end{pmatrix}$$

Notice also that you have to normalize the enlarged image matrix by a factor of 4.

In Figure 10 is shown the result of the dithering with a 8×8 matrix.

3.2 Exercise 8

For this exercise we have to build a custom function that transform an image from the RGB color space to YCbCr.

In Figure 11 we can visualize the image used as example. The intensities of each RGB component are represented in gray scale.

3.2.1 YCbCr fast

We implement the custom function exploiting Matlab matrix multiplication features to make it efficient.

First, we need to reshape the 3-D array to a matrix. We simply linearize the image matrix, obtaining a vector where each element is the RGB vector.

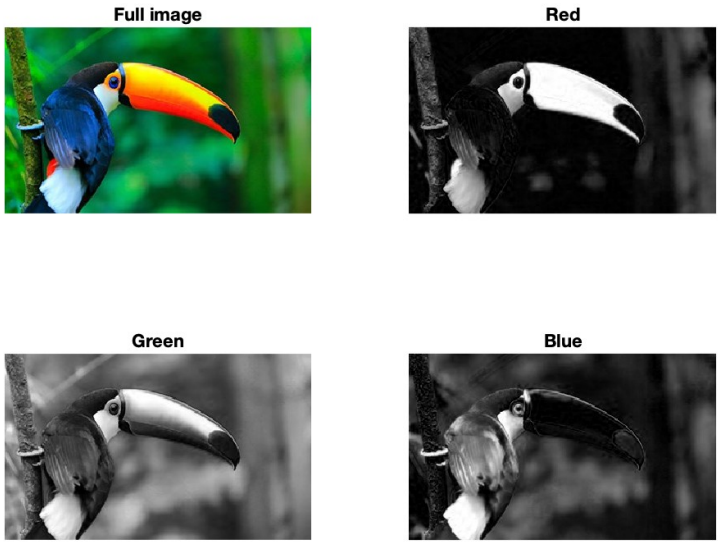

Figure 11: Toucan image showing the RGB individual components

Now we can perform the matrix multiplication

$$YC_BC_R = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{pmatrix} \times RGB$$

The last step is to reshape the matrix back as a 3-D array.

3.2.2 YCbCr slow

We also implement the same function without exploiting matrix multiplication. In this case we'll make use of 2 for loops, accessing each RGB vector individually and applying the transform.

3.2.3 Results

The can clearly see the difference in efficiency. The fast function takes only 0.002874s while the slow one 0.701855s. (the values are not consistent and varies based on the hardware the script is executed on, though the ratio between the two times should stay almost constant)

To notice also that the image is saved as a *uint8* array. To correctly develop the code is necessary to cast it to *double* (the Matlab files related to this solution of the exercise are saved with the name ending with "_double"). An alternative solution to keep the format in *uint8* is also presented.

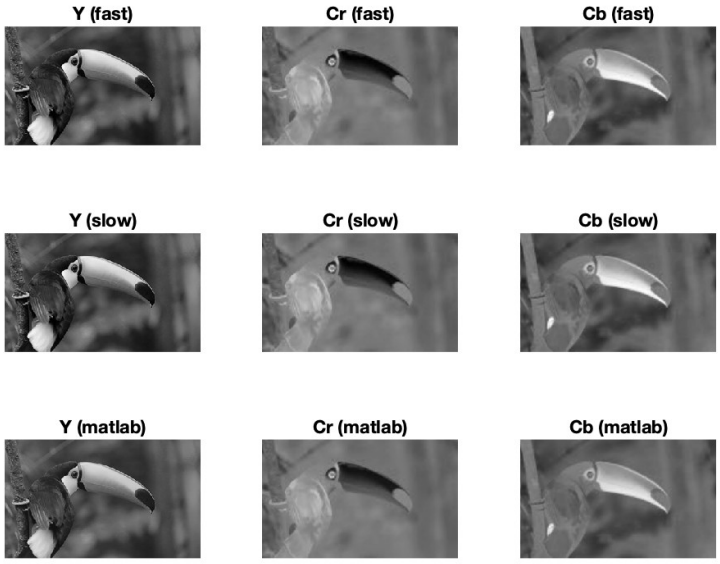

Figure 12: Toucan image showing the RGB individual components

In Figure 12 are shown the result of the custom functions implemented compared with the built-in Matlab function *rgb2ycbcr*. The results match with each others.

3.3 Exercise 9

In this exercise we apply a lowpass filter to an image to get a blur effect.

In Figure 13 the starting image in Black and White is shown.

3.3.1 Lowpass filter

We define the lowpass filter transfer function as

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

where $D_0 = P/16$ and P is the double of the next power of 2 of the biggest dimension in pixels between horizontal and vertical. $D(u,v)$ is the distance of the point (u,v) from the center.

In Figure 14 the frequency response of the computed filter can be visualized.

3.3.2 Filtering

We compute the 2-D Fourier transform of the image matrix. In order to filter the image is enough to multiply the transfer function of the filter computed before by the Fourier transform of the image.

Figure 13: Black and white image before lowpass filter

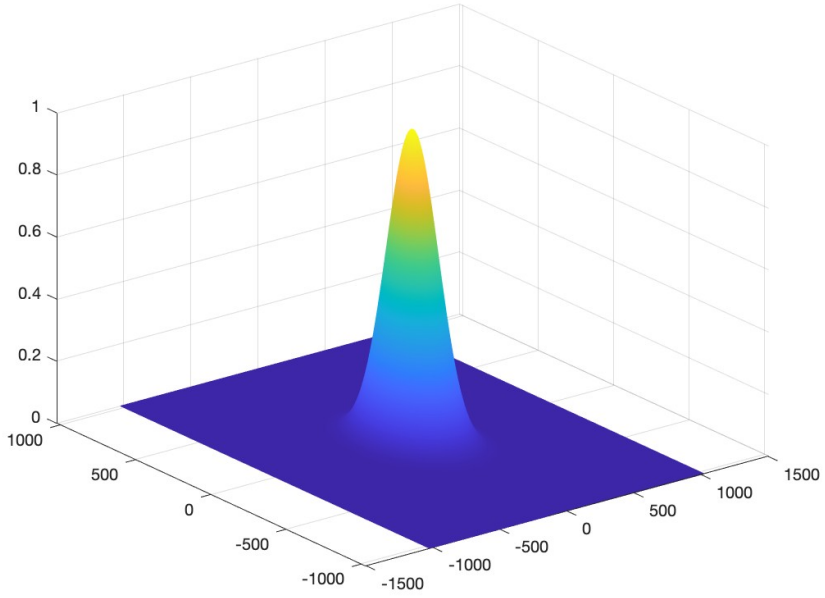

Figure 14: 2-D lowpass filter frequency response

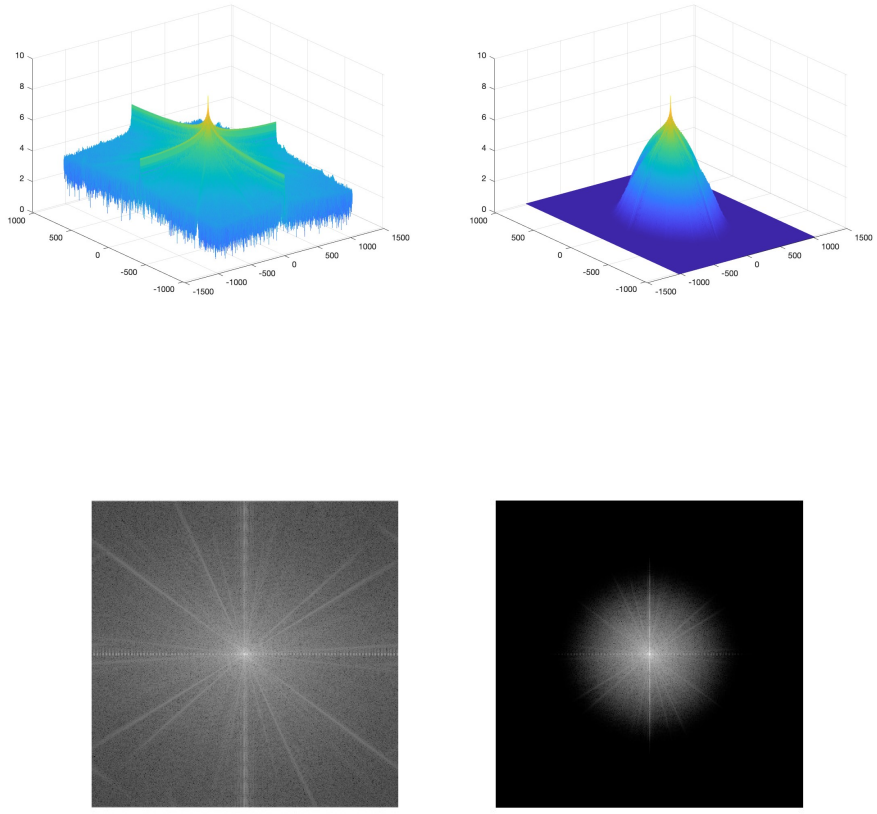

Figure 15: Comparison of the image Fourier transform before and after the filtering

Figure 16: Black and White image after filtering

In Figure 15 it is possible to visualize the image 2-D Fourier transform before and after the filtering. It can be noticed how all the high frequencies are cut off.

In Figure 16 instead is shown the image after filtering. It is clearly noticeable how the edges are smoothed achieving a blur effect.

3.4 Exercise 10

In this exercise we apply an highpass filter to an image to highlight the edges.

In Figure 17 the starting image in Black and White is shown.

3.4.1 Highpass filter

We define the highpass filter transfer function as

$$H(u,v) = 1 - H_{lp}(u,v), \qquad H_{lp}(u,v) = \frac{1}{1 + \left(\frac{D(u,v)}{D_0}\right)^{2n}}$$

where $n = 6$, $D_0 = P/16$ and P is the double of the next power of 2 of the biggest dimension in pixels between horizontal and vertical. $D(u,v)$ is the distance of the point (u,v) from the center.

In Figure 18 the frequency response of the computed filter can be visualized.

Figure 17: Black and white image before highpass filter

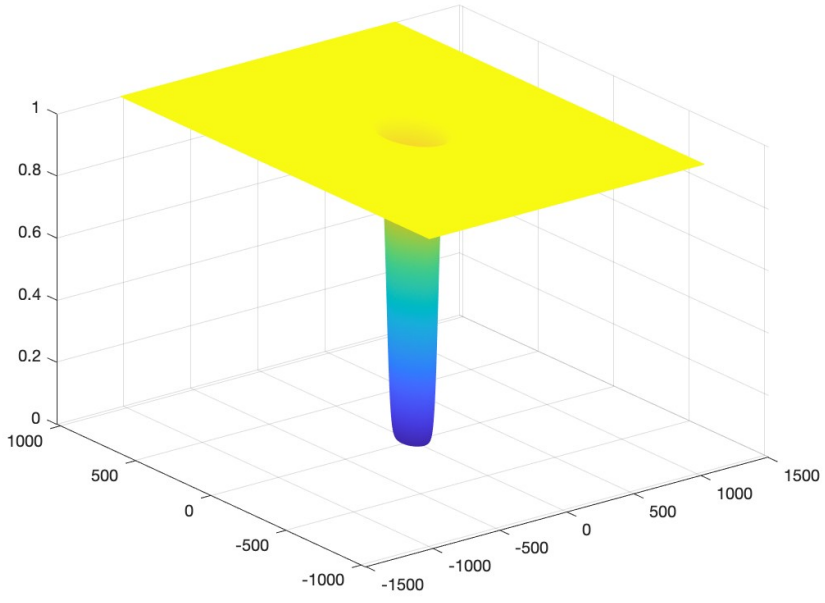

Figure 18: 2-D highpass filter frequency response

3.4.2 Filtering

We compute the 2-D Fourier transform of the image matrix. In order to filter the image we multiply the transfer function of the filter by the Fourier transform of the image.

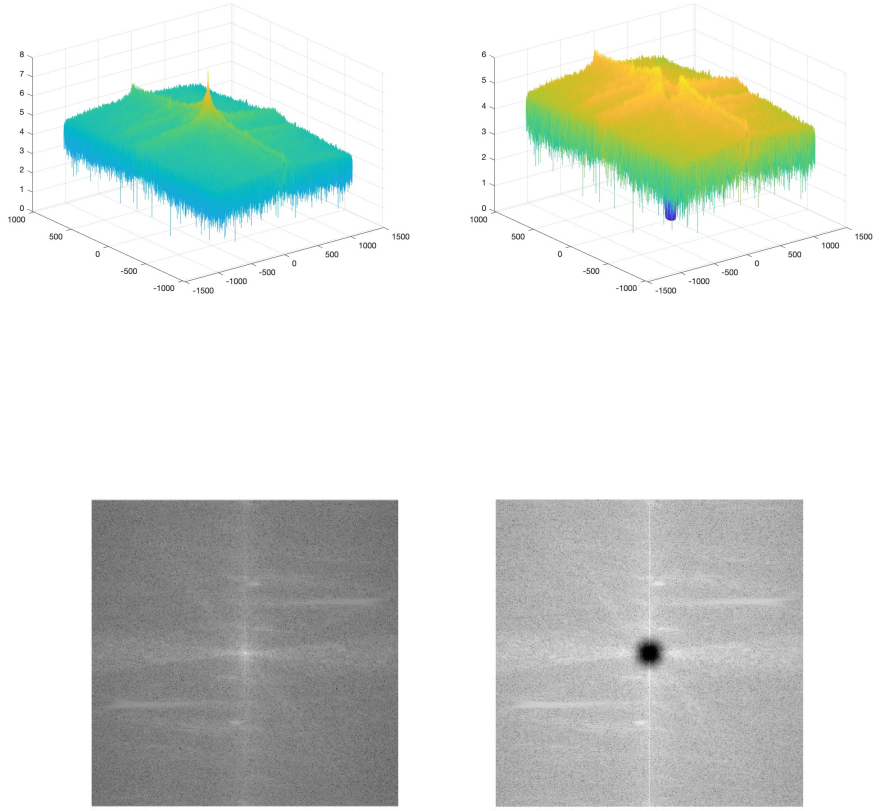

Figure 19: Comparison of the image Fourier transform before and after the filtering

In Figure 19 it is possible to visualize the image 2-D Fourier transform before and after the filtering. It can be noticed how all the low frequencies are cut off.

In Figure 20 instead is shown the image after filtering, to better see the results the colors have been inverted. It is clearly noticeable how almost only the edges are present. This image could be added to the started image to get a sharpening effect.

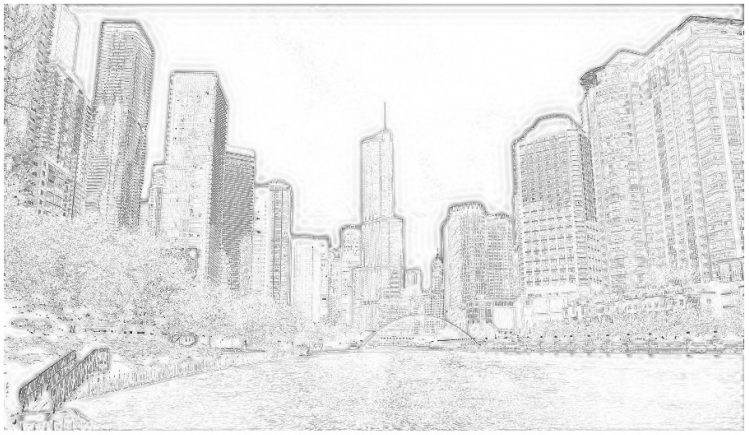

Figure 20: Image after filtering