Politecnico di Torino

Applied signal processing laboratory
(A.Y. 2024/25)

Assignment 3

Introduction to audio and image processing

Authors:
Castorina Giovanni (307594)
Fanton Vittoria (308625)
Giordano Andrea (310679)

Contents

1 Introduction

This report presents an analysis of various digital signal processing techniques applied to both audio and image data. The work is organized into three main parts: pitch estimation and time-frequency analysis of audio signals, digital audio synthesis, and basic operations on digital images.

In the first part, a custom pitch estimation algorithm is developed and applied to several audio recordings, including musical notes, guitar bends, and human voice. This analysis is improved by the use of spectrograms, which offer a visual representation of the frequency content over time.

The second part focuses on digital audio synthesis, where musical notes and chords are generated programmatically. The synthesis process is enhanced by the implementation of an ADSR envelope and further refined through the application of digital effects such as reverb. These techniques are then combined to create simple chord progressions.

The third section introduces basic image processing operations, including color space conversion and filtering, which demonstrate the versatility of signal processing concepts when applied beyond the audio domain.

In general, it is important to note that the audio files are uploaded to a GitHub repository and linked using the `hyperref` *package.*

2 Pitch estimation and spectrogram

The aim of this exercise is to create a function called `my_pitch_estimator()` to estimate the pitch of a signal, which is obtained as the inverse of the argument of the first maximum of the correlation function after lag zero.

2.1 Explanation `my_pitch_estimator()`

This function is designed to estimate the pitch of an audio signal, defined as the inverse of the time lag corresponding to the first significant peak of the autocorrelation function after lag 0. The inputs of the function are:

- The audio signal vector x
- The sampling frequency f_s
- A boolean variable `pl_on`

The output of the function is the estimated pitch. If `pl_on` is set to `true`, the function also displays a plot of the autocorrelation function, highlighting the selected peak with a vertical dashed line. A text label is shown near the peak, indicating the estimated pitch in Hz and the corresponding lag in seconds.

Since the autocorrelation is symmetric around zero, only the positive lags (excluding lag 0) are considered in the analysis. Then, the function `findpeaks()` is used to detect local maxima in the autocorrelation, with a minimum peak height threshold of 0.05. This threshold is introduced to avoid small spurious peaks due to noise or low-energy components.

Among the detected peaks, the one with the maximum value is selected, since usually the first one (which is the one that we expect to find as the pitch) is the maximum one after lag 0, and the corresponding lag is used to compute the estimated pitch according to the following formula:

$$f_{\text{estimated}} = \frac{f_s}{\texttt{best\_lag}}$$

Finally, if `pl_on` is enabled, the function produces a figure showing the autocorrelation curve and the selected pitch-related lag.

2.2 A2 note - guitar

In order to verify the correctness of the custom function `my_pitch_estimator()`, the audio file `"A2_guitar.wav"` was read using the `audioread()` function (the normalized audio signal can be listened here: `A2_audio.wav`). Only the first channel was selected by extracting the first column of the resulting matrix. The signal was then normalized by dividing it by its maximum absolute value.

An `audioplayer` object was subsequently created, allowing the signal to be played using `playblocking()`, in order to listen to the recorded audio.
The time-domain waveform of the normalized signal is shown in Figure 1.

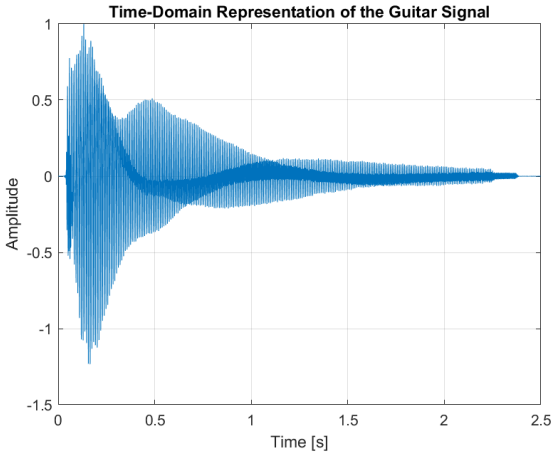


Figure 1: Time-domain representation of the normalized guitar signal from the file `A2_guitar.wav`

Analyzing the guitar signal in time, as can be observed in Figure 1, the signal duration is approximately 2.4 seconds. It is clear that the first time instants have higher power and energy and, in fact, they have high amplitude oscillations. This is due to the fact that they probably correspond to the moment in which the guitar string is plucked. Then, a fast decrease in power and energy is visible, up to a total dissolution after around 2.4 seconds, as already highlighted. These rapid, nearly exponential, decay is typical of a damped vibrating system, as it could be a guitar string.

The next step was to estimate and plot the power spectral density (PSD) of the signal, expressed in decibels, using the `pwelch()` function. To that end, the following parameters were defined:

- $N = 2000$ samples for the length
- Hamming window of length N
- 50% overlap between adjacent segments (`n_overlap` $= N/2$)

The PSD estimation was then performed using the following command:

```
[S_guitar, f_guitar] = pwelch(audio_guitar, window,...
n_overlap, N, fs, 'centered');
```

Here, the `'centered'` option ensures that the frequency axis spans from $-f_s/2$ to $f_s/2$, rather than from 0 to f_s. The estimated Power Spectral Density (PSD) is reported in Figure 2.

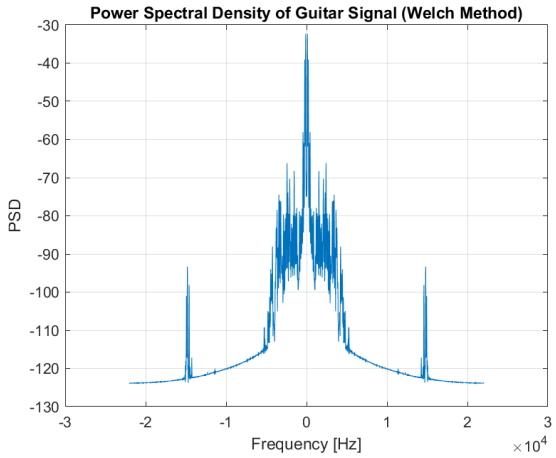


Figure 2: Power Spectral Density of the A2 guitar note estimated using the `pwelch()` method.

As can be seen from Figure 2, the spectrum appears symmetric around 0 Hz, which is a consequence of the Fourier Transform of a real signal, and it shows peaks at a frequency of 110 Hz, which correspond to the fundamental frequency of the A2 note, with some lower peaks at integer multiples of it, which correspond to the harmonics.

In addition, a spectrogram analysis was performed using the following parameters:

- Window length: $N = 4000$ samples
- Overlap: 90% of the window length ($n\_overlap = 0.9 \cdot N$)
- FFT size: 2^{12} points

The corresponding MATLAB command used is:

```
spectrogram(audio_guitar, window, n_overlap, N, fs, 'yaxis')
ylim([0 8])
```

The resulting spectrogram is shown in Figure 3.

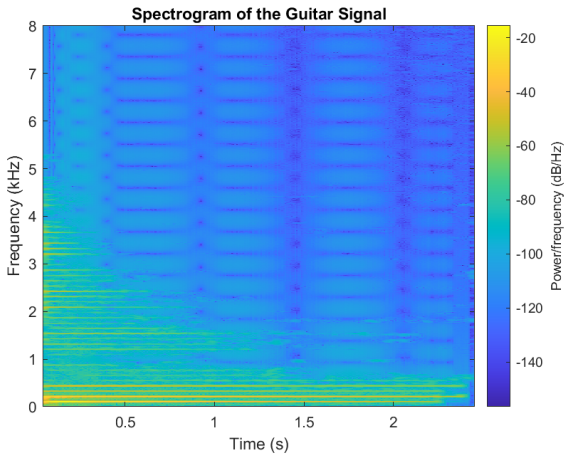


Figure 3: Spectrogram of the A2 guitar signal, limited to 8 kHz.

From Figure 3, several horizontal lines are visible across different frequencies, in particular during the initial moments of the signal. These lines correspond to the fundamental frequency and its harmonics generated by the guitar. Specifically, the first prominent frequency component appears at 110 Hz, which matches the fundamental frequency of the note A2. The subsequent lines represent its harmonics, appearing at integer multiples of the fundamental frequency.
The temporal decay of the signal is also evident: the intensity of these frequency components decreases over time, as indicated by the fading color, reflecting the expected reduction in energy. Most of the spectral energy is, in fact, concentrated within the first seconds of the signal. Furthermore, as also shown by the power spectral density, the majority of the energy is concentrated at relatively low frequencies. This is consistent with the fact that

the amplitudes of the harmonics decrease as their frequency increases.

Finally, the custom function `my_pitch_estimator()` is applied to the audio signal with the flag `pl_on` set to `true`. The resulting normalized autocorrelation, along with the detected pitch, is shown in Figure 4.

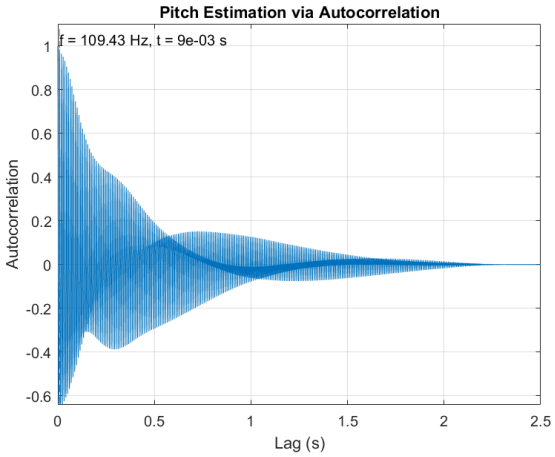


Figure 4: Normalized autocorrelation of the A2 guitar signal with pitch detection.

As shown in Figure 4, the plot shows only the positive part of the normalized autocorrelation function, and reaches its maximum at $x = 0$, as expected. The first significant peak after the origin occurs at approximately 9.13 ms, indicating a fundamental period of that duration. Given that the expected pitch for the note A2 is around 110 Hz, it is possible to compute the estimated pitch based on the autocorrelation method as:

$$f_{\text{pitch}} = \frac{1}{0.0091 \text{ s}} = 109.43 \text{ Hz} \approx 110 \text{ Hz}$$

This value is very close to the theoretical value, with a small error of $\Delta f = 0.57$ Hz, which falls within an acceptable range. The slight deviation can be attributed to factors such as noise or small inaccuracies in the detection.

2.3 Analysis `Bending.wav`

All the steps described in Section 2.2 are repeated for the audio file `Bending.wav`. The audio is read using the `audioread()` function, and once again only the

first channel is selected for analysis. The signal is then normalized by its maximum absolute value. An `audioplayer` object is created to allow playback using the `playblocking()` function (the normalized audio signal can be listened here: `A2_audio.wav`).
The time-domain waveform of the normalized signal is illustrated in Figure 5.

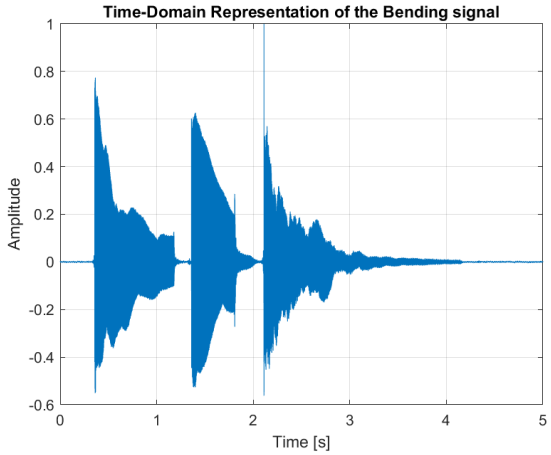


Figure 5: Time-domain representation of the normalized bending signal from the file `Bending.wav`.

As shown in Figure 5, the audio signal consists of three distinct notes played in sequence. Every note begins with a sharp increase in amplitude, corresponding to the moment the note is struck, followed by a gradual decay over time. This decay reflects the natural damping of the sound as the energy dissipates after the initial excitation.

The next step involves estimating and plotting the Power Spectral Density (PSD) of the signal, in decibels, using the `pwelch()` function. The parameters used for the estimation are the same as those adopted in Section 2.2, and the same MATLAB function is employed. The resulting PSD is presented in Figure 6.

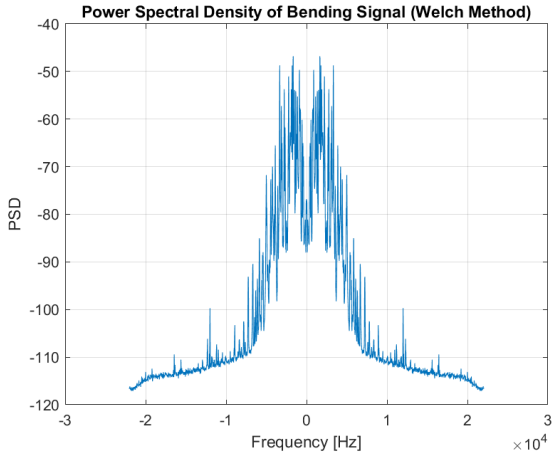


Figure 6: Power Spectral Density of the bending signal estimated using the `pwelch()` method.

As can be seen from Figure 6, the majority of the signal's power is concentrated around zero frequency, with the spectral content extending not really much from the very low values. This indicates that the bending signal primarily consists of low-frequency components, as expected. Outside this region, the PSD rapidly decreases.

In addition, a spectrogram analysis was performed using the same parameters adopted in Section 2.2. The MATLAB command used for this purpose was identical, and the resulting spectrogram is shown in Figure 7.

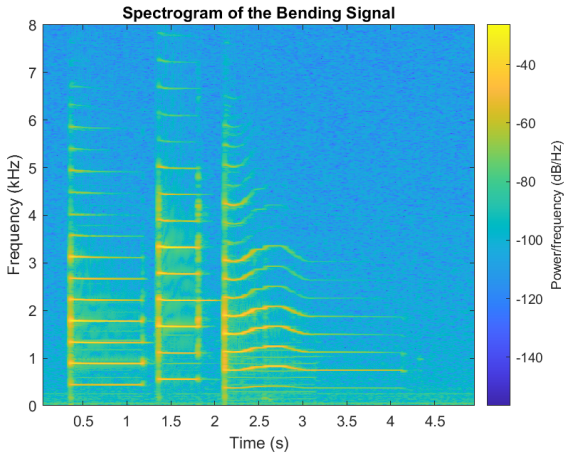


Figure 7: Spectrogram of the bending signal, limited to 8 kHz.

The spectrogram in Figure 7 shows the time-frequency representation of the bending signal, highlighting how the spectral changes evolve over time. For what concerns the first 2 notes, distinct horizontal lines are visible. These lines represent the fundamental frequency and its harmonics, appearing as evenly spaced horizontal bands.

After about 2 seconds, when the third note is played, a different behavior emerges. Instead of horizontal lines, some curves are observed, gradually rising in frequency, especially in the first few harmonics. This is characteristic of pitch bending: the pitch of the note tends to increase smoothly over time, and then go back to around the previous note again.

The next step is to isolate the three notes present in the bending signal. This was accomplished by analyzing the time-domain waveform and estimating the approximate time intervals corresponding to the start and end of each note. To extract the notes from the signal, it was necessary to operate on sample indices rather than time values. Therefore, the time instants were converted into sample indices using the relation $N = t \cdot f_s$. The notes were, therefore, isolated using the following MATLAB commands:

```
note1 = audio_bending(0.22*fs:1.26*fs, 1);
note2 = audio_bending(1.26*fs:2.1*fs, 1);
note3 = audio_bending(2.1*fs:3.6*fs, 1);
```

With the notes isolated, the pitch of the first two notes was estimated using the custom function `my_pitch_estimator()`. The estimated pitch values are the following:

$$f_{pitch_1} = 445.45 \text{ Hz}$$
$$f_{pitch_2} = 558.23 \text{ Hz}$$

According to the standard note frequency chart, these frequencies correspond approximately to the following musical notes:

$$f_{pitch_1} = 445.45 \text{ Hz} \approx 440 \text{ Hz} \rightarrow A4$$
$$f_{pitch_2} = 558.23 \text{ Hz} \approx 554.37 \text{ Hz} \rightarrow C\sharp 5$$

The results of the pitch estimation for the first two notes are shown in Figure 8, where subfigure (a) corresponds to the first note and subfigure (b) to the second one. The position of the vertical lines in both figures perfectly show the frequencies f_{pitch_1} and f_{pitch_2}.

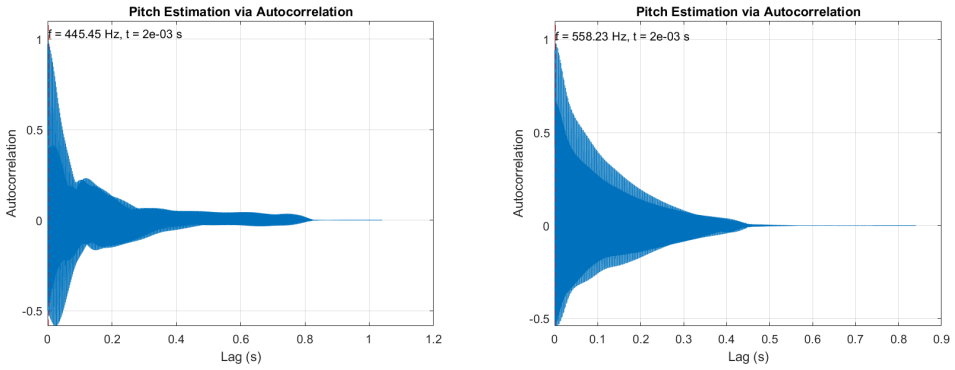


Figure 8: Pitch estimation using autocorrelation for (a) the first and (b) the second isolated notes from the `bending.wav` file.

In addition, for what concerns the third note, it is stated that it is bent, i.e., the pitch of the note is increased by a tone through string bending. For this note, exactly 1.5 seconds of duration are extracted starting at 2.1 seconds of audio. The segment is then divided into frames of 0.1 seconds using a sliding window technique with 90% overlap to estimate the pitch of each frame using the custom function. The parameters defined for this operation are:

- **N**: number of samples corresponding to 1.5 seconds. Considering that the third note has a duration of 1.5 seconds, N can be obtained as `length(note3)`;
- **M**: number of samples in 0.1 seconds, computed as `0.1*fs`;
- **L**: computed as $L = 0.9 \cdot M$;
- **S**: total number of frames, given by the formula $S = \left\lfloor \frac{N-L}{M-L} \right\rfloor$.

The goal is to plot the estimated pitch for each frame versus the time instant corresponding to the center of that frame. This is done by splitting the signal into S overlapping frames of length M, each shifted by $(M - L)$ samples. For every frame, the corresponding segment is extracted and the pitch is estimated using `my_pitch_estimator`. The result is stored in the first row of the matrix `pitches_ind`. In the second row, the central index of the frame is saved, so that the pitch can be correctly plotted in time.

The result obtained from the loop, showing the estimated pitch for each frame, is presented in Figure 9.

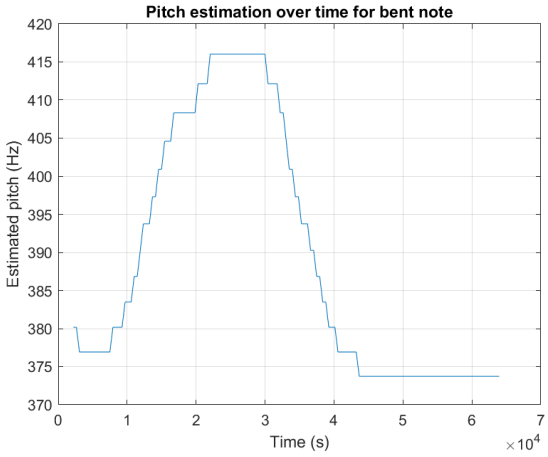


Figure 9: Estimated pitch for each frame of the bent note segment using the sliding window technique.

As shown in Figure 9, a starting note (lower frequency) and a target note (higher frequency, corresponding to the bending peak) can be observed. According to the note chart and considering the corresponding pitch frequencies, the possible notes are:

1. Estimated frequency: $f \approx 375$ Hz $\rightarrow$ Closest note: $F\sharp 4$ (370 Hz)
2. Estimated frequency: $f \approx 416$ Hz $\rightarrow$ Closest note: $G\sharp 4$ (415 Hz)

Of course, it is clear that the frequency estimation is not totally perfect, because there can be variations and noise in the signal, as well as inaccuracies in the pitch detection. These may lead to fluctuations in the estimated pitch values, especially during transitional phases such as the bending movement. Nevertheless, the overall pitch trajectory is clearly identifiable: the signal begins at a stable lower frequency (of 370 Hz), corresponding to the starting note, and then gradually increases until reaching the peak frequency, which matches the target bent note.

For the next section of the exercise, the complete signal from the file `Bending.wav` is considered. The objective is to apply a low-pass FIR filter characterized by the following specifications:

- Cutoff frequency: $f_c = 5$ kHz

- Transition band: $B_t = 500$ Hz
- Minimum stopband attenuation: $A_s > 50$ dB

Firstly, since the required minimum stopband attenuation is greater than 50 dB, a Hamming window is selected due to its adequate attenuation characteristics. Once the window type is defined, the transition bandwidth is normalized with respect to the sampling frequency f_s, and the number of coefficients required for the window is computed using the formula specific to the Hamming window:

$$N = \left\lceil \frac{6.6\pi}{2\pi \cdot \text{trans\_band\_normalized}} \right\rceil = \left\lceil \frac{3.3}{\text{trans\_band\_normalized}} \right\rceil = 292$$

Then, the cutoff frequency is normalized with respect to the sampling frequency:

$$f_c^{\text{norm}} = \frac{f_c}{f_s} = 0.11$$

The Hamming window is then created according to its definition:

$$w_{\text{Hamming}}[n] = 0.54 - 0.46 \cdot \cos\left(\frac{2\pi n}{N-1}\right)$$

Next, the delay in samples, denoted as M, is defined to ensure that the filter is causal:

$$M = \left\lceil \frac{N-1}{2} \right\rceil = \left\lceil \frac{292-1}{2} \right\rceil = 146$$

Once both the window and the delay are computed, the final FIR filter is obtained by multiplying the ideal impulse response of a low-pass filter with the window function:

$$h[n] = h_{\text{id}}[n] \cdot w_{\text{Hamming}}[n]$$

where $h_{\text{id}}[n]$ is the impulse response of the ideal low-pass filter. This impulse response is given by:

$$h_{\text{id}}[n] = 2f_c^{\text{norm}} \cdot \text{sinc}\left(2f_c^{\text{norm}}(n-M)\right)$$

where f_c^{norm} is the normalized cutoff frequency and M is the delay previously defined.

Finally, the filtering operation is performed using the following command:

```
bending_filtered = filter(h, 1, audio_bending);
```

Since the filter is FIR, the numerator is given by the filter coefficients h and the denominator is set to 1. This applies the designed filter to the audio signal.

Once the filtering process has been completed, the next step is to undersample the filtered signal by a factor of 4. This is achieved by selecting one sample every four and defining the new sampling frequency as $f_{s_2} = f_s/4$:

```
under_factor = 4;
fs_2 = fs / under_factor;

bending_filtered_under = bending_filtered(1:under_factor:end);
```

After applying the low-pass filter and undersampling the signal by a factor of four, the resulting spectrogram is shown in Figure 10. As can be observed, this operation affects the spectral content of the signal. First, the visible frequency range is restricted due to two effects: the low-pass filter with a cutoff frequency of 5 kHz, and the reduced Nyquist frequency resulting from the new sampling rate ($f_s/4$). As a result, all frequency components above 5 kHz are effectively removed, and the upper part of the original spectrum is no longer present.

These spectral changes are reflected in the perceived audio quality. The corresponding audio clip can be heard at the following link: `undersample4.wav`. Compared to the original signal, the sound appears duller, mainly because the higher frequency components have been attenuated or entirely removed. Nevertheless, the audio still resembles the original, although the overall timbre is slightly darker due to the loss of high-frequency content.

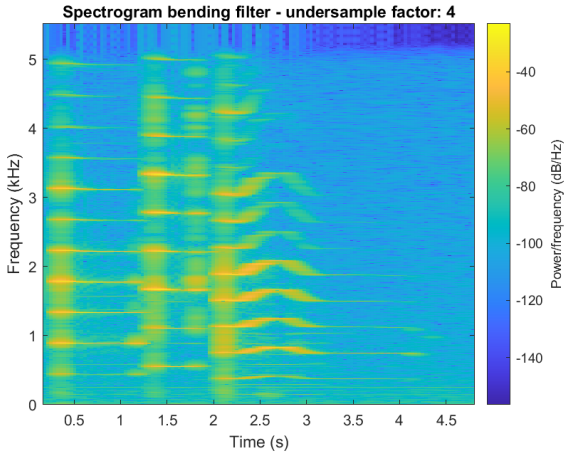


Figure 10: Spectrogram of the filtered signal after undersampling by a factor of 4

The same procedure is then repeated starting from the original signal, but with a new cutoff frequency set to 2.5 kHz. In this case, the signal is undersampled by a factor of 8, and the new sampling frequency is defined as $f_{s_3} = f_s/8$.

The spectrogram of the second undersampled version is shown in Figure 11. The effects of undersampling are more pronounced compared to those in Figure 10, due to the greater reduction in sampling frequency and cutoff frequency of the filter. Consequently, the visible frequency range is further narrowed and higher frequency components are much more attenuated.

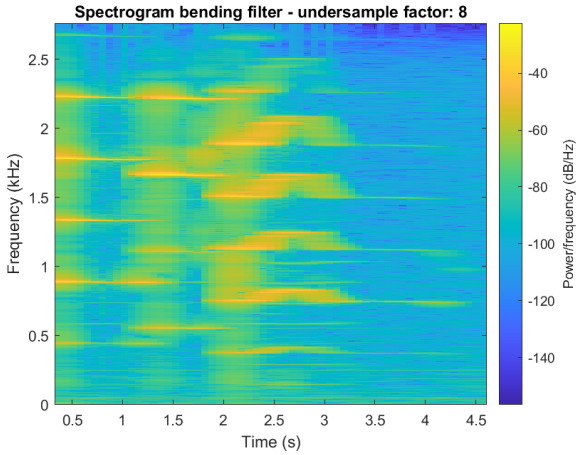


Figure 11: Spectrogram of the filtered signal after undersampling by a factor of 8

This version of the signal can also be played using the playblocking function in MATLAB, or listened to directly here: `undersample8.wav`. Compared to the previous versions, the audio sounds different. As expected from the processing steps, it appears much more muffled and dominated by low-frequency content, due to the stronger suppression of high-frequency components.

2.4 Analysis of the voice

The last section of the exercise required the estimation of the pitch of a voice signal, recorded directly from MATLAB.

First, the parameters are set as follows:

- Sampling frequency: $f_s = 8$ kHz
- Bit depth: $N_{bits} = 16$

An `if-else` statement is used to check whether the voice recording file already exists (using the `isfile` command). If the file does not exist, an `audiorecorder` object is created, and a 3-second audio clip is recorded. The recording is then saved to a WAVE file using `audiowrite`. Otherwise, if the file is already present, it is loaded using `audioread`.

Once the signal is loaded, it is normalized by dividing it by its maximum absolute value. It is then played using the `playblocking` function (Note

that the audio of the voice signal can be listened here: `voice.wav`).

Subsequently, the signal is plotted in the time domain, as shown in Figure 12. From the plot, it is evident that the segment between 0.5 and 1.3 seconds contains a strong vocalized portion, whereas the remaining parts appear to consist mainly of noise or other non-vocal components.

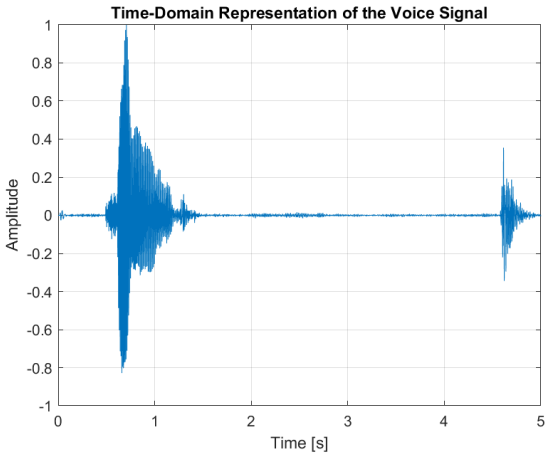


Figure 12: Time-domain representation of the normalized voice signal.

The power spectral density (PSD) of the signal is then estimated using the `pwelch` function with the following parameters:

- $N = 2000$ samples
- Hamming window
- 50% overlap

The resulting Power Spectral Density (PSD) is shown in Figure 13. The plot reveals a prominent peak around 110 Hz, which corresponds to the dominant frequency component of the voice signal. Given that the speaker is male, this value is consistent with the typical male pitch range, generally around 90-100 Hz. Additionally, the PSD appears symmetric and centered around zero frequency, as expected due to the use of the `fftshift()` function, which reorders the frequency components to place the zero frequency at the center of the spectrum.

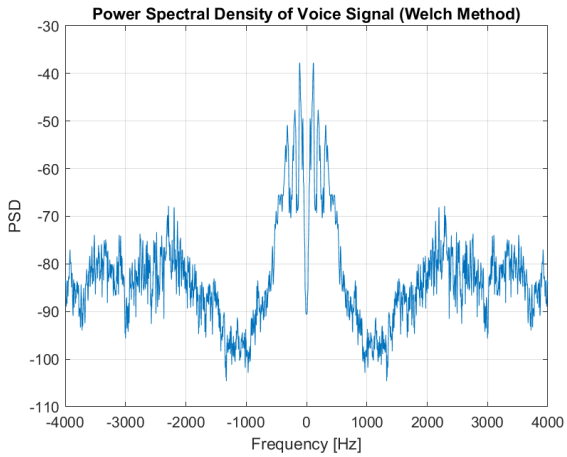


Figure 13: Power Spectral Density (PSD) of the voice signal estimated using Welch's method.

To estimate the pitch accurately, it is essential to isolate only the vocalized portion of the signal. By examining the time-domain representation of the voice signal, it can be observed that the vocal activity is primarily concentrated between 0.5 and 1.2 seconds. To ensure the selection of a fully vocalized segment, without including transitional regions, the interval from 0.5 to 1.0 seconds is chosen for the analysis. This is implemented in MATLAB as follows:

```
my_Speech_vocalized = my_Speech(round(0.5*fs_voice)...
+1 : round(1.0*fs_voice));
```

This operation extracts the appropriate range of samples by converting the time boundaries into sample indices, using the sampling frequency `fs_voice`.

Finally, the pitch of the vocalized signal is estimated using the custom function `my_pitch_estimator`. The resulting pitch is:

$$f_{\text{pitch}} = 108.11 \text{ Hz}$$

This value appears to be fully consistent with the expected pitch range for human voices, which typically spans from 80–90 Hz (for male voices) up to around 200 Hz (for female voices). Since the input audio was based on a male voice, the result is considered valid. Moreover, the autocorrelation function used to estimate the pitch is shown in Figure 14.

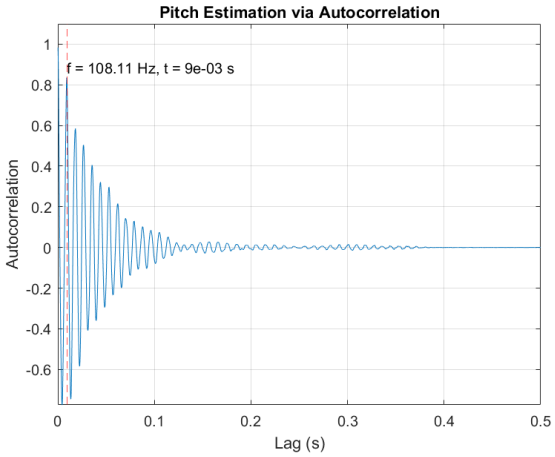


Figure 14: Autocorrelation of the vocalized segment used for pitch estimation.

Figure 14 shows a structure similar to a periodic one, characteristic of voiced speech. The first peak after the zero lag corresponds to the fundamental period of the signal and is used to compute the pitch frequency, which, as expected correspond to 108 Hz.

3 Digital audio synthesis

3.1 Note generation

The `note_generation()` function uses a simple form of subtractive synthesis to turn a raw pulse-width-modulated signal into a bowed-string-like tone. First, it fixes the sampling rate at 44.1 kHz and builds a 2.5 s time vector. Next, it creates a standard sawtooth at the target pitch f_0 and a slow sine at $lfo = f_0/200$, then builds a PWM waveform with

```
saw = sawtooth(2*pi*f0*t);
sine = sin(2*pi*lfo*t);
pulse = saw < sine;
```

so that the duty cycle changes gently over time, enriching the harmonic content. To smooth out the harsh edges of this pulse train, the code designs a low-pass FIR filter by hand: it sets the passband edge $wp = f_0$, stopband edge $ws = 16 * f_0$, and transition band $Bt = 15 * f_0$, then computes the normalized cutoff

```
fc_norm = (wp+ws)/(2*fs);
N = ceil(6.1*pi/(Bt/fs*2*pi));
M = ceil((N-1)/2);
n = -M:M;
```

It forms the ideal sinc via

```
h_id = 2*fc_norm * sinc(2*fc_norm*(n-M));
w_bartlett = 1 - 2*abs(n)/M;
h = h_id .* w_bartlett;
h_norm = h / sum(h);
```

and finally filters the PWM into the bowed-string-like note with

```
note = filter(h_norm, 1, pulse);
```

Every step, from PWM generation to Bartlett-windowed sinc filtering, combines to give each 2.5 s note its characteristic warmth and realism.

3.2 Chord generation & ADSR envelope

The `chord_generator()` function begins by computing the three frequencies that make up a triad by moving the root note up by the required semitone intervals in the chromatic scale. Given a root frequency *root_note*, it calculates the third as

```
switch type
  case 'maj'
    f2 = 2^(4/12) * root_note;  % major third
  case 'min'
    f2 = 2^(3/12) * root_note;  % minor third
  otherwise
    error('chord_generator: invalid type, use ''maj'' or ''min''');
end
f3 = 2^(7/12)*root_note;  % perfect fifth
```

Each of these, *root_note*, f_2, and f_3, is passed to the helper `note_generation()` function, which itself builds a PWM sawtooth and then filters it with a Bartlett-windowed FIR. The three resulting waveforms (*note*1, *note*2, *note*3) are summed into a raw chord:

```
chord = note1 + note2 + note3;
```

and immediately normalized to ensure uniform level:

```
chord_norm = chord ./ max(abs(chord));
```

To impart a natural articulation, a simple ADSR envelope of duration $T = 2.5\,\mathrm{s}$ is constructed by interpolating the points

$$(0,0),\ (0.16T,1),\ (0.32T,0.7),\ (0.60T,0.7),\ (T,0)$$

with piecewise-cubic interpolation (`'pchip'`):

```
t = 0:1/fs:T-1/fs;
ADSR = interp1([0,0.16,0.32,0.6,1]*T, [0,1,0.7,0.7,0], t, 'pchip');
smooth_chord = chord_norm .* ADSR;
```

This multiplies the chord by an envelope that attacks to full amplitude, decays to 70%, sustains, then releases to zero.

The function was tested with a C5 major triad through the following code:

```
C5 = 440 * 2^(3/12);
chord1 = chord_generator(C5, 'maj');
chord1_obj = audioplayer(chord1, fs);
playblocking(chord1_obj);
```

Finally, two verification figures are produced. First, the time-domain overlay, shown in Figure 15, which confirms the envelope shape against the waveform. Second, the log-frequency spectrum, reported in Figure 16, which clearly shows three peaks at the root, third and fifth.

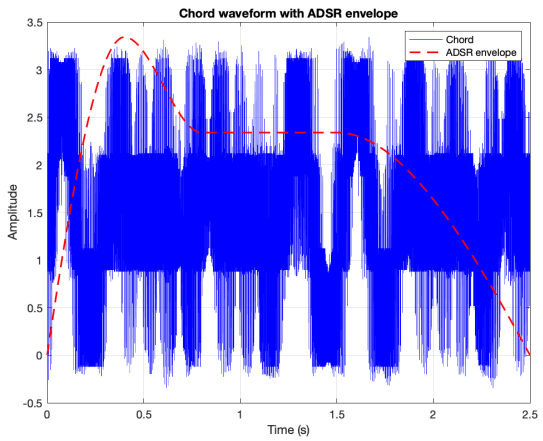


Figure 15: Chord waveform with ADSR envelope: normalized chord (blue); ADSR mask (red dashed).

3.3 Chord progression

The eight-chord sequence is built by first generating each triad via calls to `chord_generator` at the appropriate root frequencies. In code:

```
A4 = 440;
chord_A4 = chord_generator(A4, 'maj');
D5 = A4 * 2^(5/12);
chord_D5 = chord_generator(D5, 'maj');
B4 = A4 * 2^( 2/12);
chord_B4m = chord_generator(B4, 'min');
F4sharp = A4 * 2^(-3/12);
chord_F4sharpm = chord_generator(F4sharp, 'min');
```

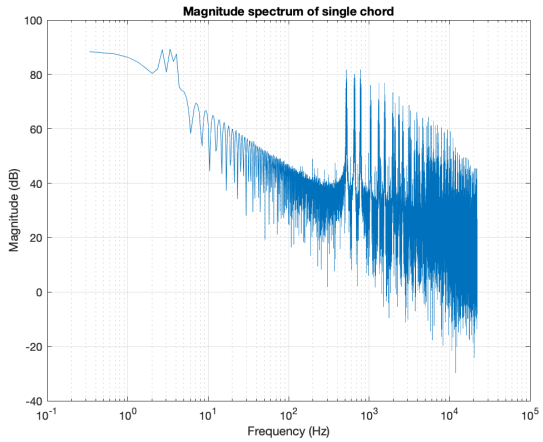


Figure 16: Magnitude spectrum of single chord: log-frequency magnitude showing peaks at root, third, and fifth.

```
G4 = A4 * 2^(-2/12);
chord_G4 = chord_generator(G4, 'maj');
```

These eight chord vectors are then concatenated and played back as follows:

```
my_progression = [chord_D5, chord_A4, ...
    chord_B4m, chord_F4sharpm, chord_G4, ...
    chord_D5, chord_G4, chord_A4];
my_progression_obj = audioplayer(my_progression, fs);
playblocking(my_progression_obj);
audiowrite("my_progression.wav", my_progression, fs);
```

Figure 17 presents the full 20 s waveform of the progression, with vertical dashed lines indicating the onset of each 2.5 s chord (D5 - A4 - Bm - F♯m - G4 - D5 - G4 - A4). This confirms both correct timing and smooth transitions between successive triads.

Furthermore, the audio rendering of this eight-chord progression, generated using `audiowrite()` built-in function, can be accessed on GitHub at `my_progression.wav`.

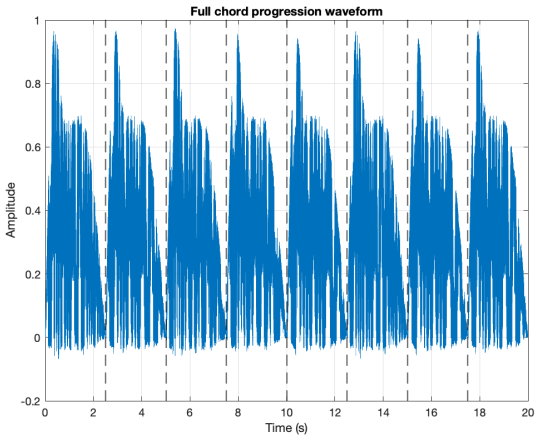


Figure 17: Waveform of the eight-chord progression (D5–A4–Bm–F♯m–G4–D5–G4–A4). Vertical dashed lines mark the onset of each 2.5 s chord segment.

3.4 Apply a digital reverb

In the final processing step, the chord progression is transformed into a more "cathedral-like" sound by convolving it with a measured impulse response. First, the impulse response file is read and resampled to match the 44.1 kHz project rate:

```
[data, fs_cathedral] = audioread("impulse_revcathedral.wav");
r = fs / fs_cathedral;  % interpolation factor
n = 4;  % upsampling order
cutoff = 1 / r;  % anti-aliasing cutoff
imp_res_cath = interp(data, r, n, cutoff);
```

Next, instead of convolving in the time domain, both the chord progression and the impulse response are transformed to the frequency domain via FFT computed over M points (which zero-pads each signal to length M), their resulting spectra are multiplied element-wise, and the inverse FFT returns the reverberated signal back to the time domain.

```
M  = 2^nextpow2(length(imp_res_cath) ...
    + length(my_progression));
fft_cath = fft(imp_res_cath, M);
fft_progr = fft(my_progression,  M);
fft_filt = fft_cath.' .* fft_progr;
```

```
filtered_progr = real(ifft(fft_filt, M));
filtered_progr_norm = filtered_progr ...
/ max(abs(filtered_progr));
```

Finally, the result is played back and written to file:

```
filtr = audioplayer(filtered_progr_norm, fs);
playblocking(filtr);
audiowrite("my_reverb_progression.wav", filtered_progr_norm, fs);
```

Figure 18 illustrates the interpolated impulse response in both the time and frequency domains. The time plot shows a long decay tail characteristic of a large reverberant space, while the magnitude response reveals the frequency coloration imparted by the cathedral.

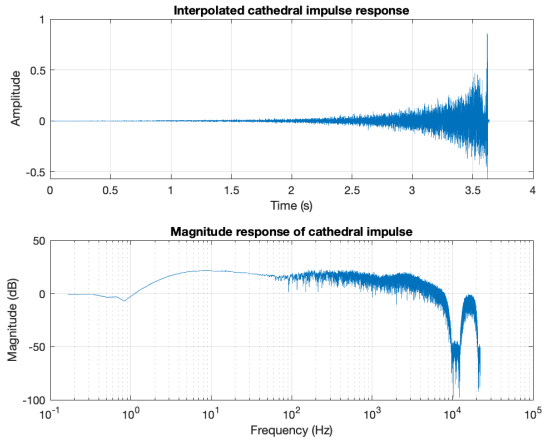


Figure 18: Interpolated cathedral impulse response: (top) time-domain decay tail; (bottom) log-frequency magnitude response.

Figures 19 and 20 compare the dry and reverberated chord progression. In the time domain (Figure 19), the red trace clearly exhibits the extended decay between and after chords. In the frequency domain (Figure 20), the reverberated spectrum is elevated across mid and high frequencies, illustrating the diffuse reflections that fill in harmonic content.

Finally, the audio rendering of this reverberated eight-chord progression, generated using `audiowrite()` built-in function, can be accessed on GitHub at `my_progression.wav`.

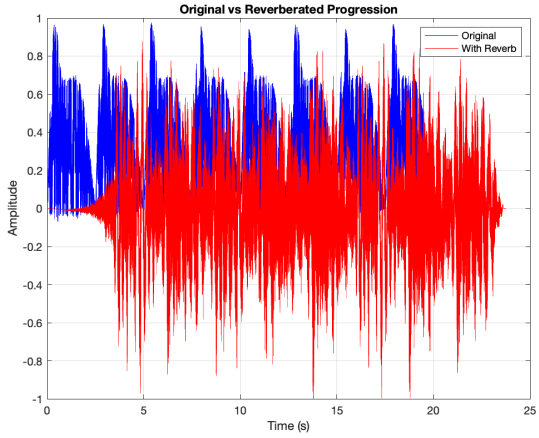


Figure 19: Time-domain comparison of the original (blue) and reverberated (red) progression, showing the added decay tail from the cathedral impulse.

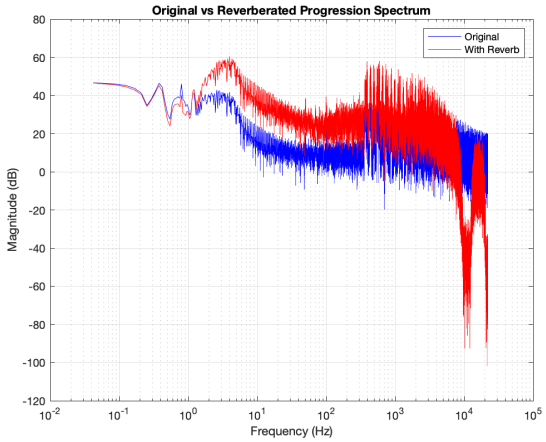


Figure 20: Log-frequency magnitude spectra of the original (blue) and reverberated (red) signals, demonstrating the spectral "lift" provided by the reverb.

3.5 *Optional part*: create our own song

Our aim in this part was to create a different sound other than the scale generated in the previous point. In particular, we wanted to recreate the beginning of *Bohemian Rhapsody* by Queen; as reference, we used the piano adaptation found *here* on YouTube. As soon as we started saving the note,

we soon realized that the parameter before to synthesize the sounds were not good enough to recreate the sounds we needed. Often we needed to play only one note at a time, without any chords, so we needed each note to be nice to hear by itself (in the previous case, if a single note was played by itself, it was possible to hear many small oscillation, following the sawtooth function). To synthesize the new note, the following code has been used:

```
init_tone = (sin(2*pi*f0*t) ...
    + sin(2*pi*f0*t + 0.2/f0) + sin(2*pi*f0*t - 0.2/f0) ...
    + sin(2*pi*f0*t + 0.4/f0) + sin(2*pi*f0*t - 0.4/f0) ...
    + sawtooth(2*pi*f0*t + 0.3/f0) + sawtooth(2*pi*f0*t - 0.3/f0) ...
    + sawtooth(2*pi*f0*t + 0.5/f0) + sawtooth(2*pi*f0*t - 0.5/f0) ...
    );
```

It is possible to see that this function, is much more complicated than the function used above to synthesize a note. Each note consists of the sum of sine and sawtooth functions with the frequency of the desired note and some phase shift. These phase shifts serve to "round" the tone, making the resulting sound richer and more natural. As before, the `init_tone` is compared with a slow varying sine wave, with frequency $f = \frac{f_0}{16}$ to generate a PWM signal. However, to further approximate the timbre of a real piano, this PWM pulse is also modulated by another slowly varying sine wave of frequency $f = \frac{f_0}{8}$, scaled by an amplitude of 0.12 and including a DC component. Again, as done before, the PWM pulse has to be filtered: this time both a low-pass and a high-pass filter have been utilized, both designed through window method, using a Bartlett window. For the low-pass filter the following parameters have been used:

```
Bt_lp = 7*f0*2*pi/fs;
Wp_lp = f0*2*pi/fs;
Ws_lp = 8*f0*2*pi/fs;
Wc_lp = (Wp_lp+Ws_lp)/2;
```

For the high-pass these were the parameters that made the sound nicer to hear:

```
Bt_hp = 100*2*pi/fs;
Wp_hp = 60*2*pi/fs;
Ws_hp = 20*2*pi/fs;
Wc_hp = (Wp_hp+Ws_hp)/2;
```

To make the note sound more like a piano, instead of a bow instrument, a different ADSR envelope was designed, again by interpolating some points using the `interp1` function with the `'pchicp'` option:

```
ADSR = interp1([0 0.01 0.9 1]*T, [0 1 0.1 0], t, 'pchip');
```

In this way, we obtain a note that reaches its peak amplitude shortly after it is being played, creating a natural and expressive attack. Following this initial rise, the amplitude gradually decreases, allowing the sound to fade out smoothly over the remaining duration, closely mimicking the behavior of many real musical instruments. In Figure 21 it is shown the envelope in the time domain.

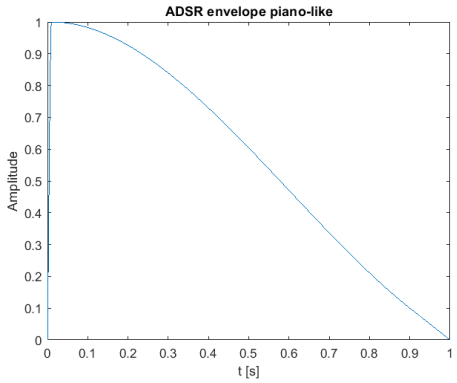


Figure 21: ADSR envelope for a piano-like note, obtained by data interpolation

Comment on the obtained symphony

If we play the music track (available on GitHub at `bohemian_rhapsody.wav`), it becomes evident that the first part differs significantly from the original, while the similarity increases noticeably when the chorus begins. Nevertheless, we are satisfied with our result: we were still able to produce a recognizable and coherent version of the song.

4 Color space conversion and image filtering

This exercises focuses on image processing. The scope of this exercise is to see what happens when images are filtered with different types of filters. Furthermore, we were interested in the design of two custom function to perform the linear mapping between RGB and YCbCr. The difference between these two functions is, as we will explain later in more detail, their time complexity.

The image `parrots.jpg`, stored in RGB format, is loaded into MATLAB using the built-in `imread()` function. The resulting array has dimensions $M \times N \times 3$, where $M = 864$ and $N = 1024$. Next, the array is first converted to 'double' and then normalized by dividing it by 255. Finally, the image is displayed with the command `imshow(parrots, [])` and is shown in figure 22.

Figure 22: Original RGB input image (`'parrots.jpg'`), as loaded in MATLAB.

An alternative to the RGB color model is the YCbCr model. While the RGB model represents each pixel by its red, green and blue components, the YCbCr model separates luminance (brightness) from chrominance (color-difference). In particular:

- Y encodes the luma component, i.e. the luminosity as perceived by the human eye.

- Cb and Cr encode the blue-difference and red-difference chroma components, respectively.

The linear mapping from RGB to YCbCr that was used is:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

4.1 Fast RGB to YCbCr filtering

In the fast implementation custom function `my_rgb2ycbcr_fast`, this 3×3 matrix is applied in a fully vectorized way to each pixel. First, the three color planes are extracted and converted to 'double'. Then each Y, Cb and Cr component is computed as a weighted sum of R, G and B according to the linear mapping presented in the matrix above. Finally, the three resulting matrices are recombined into a single $M \times N \times 3$ output. This implementation was carried out according to the following code:

```
function YCbCrimg = my_rgb2ycbcr_fast(RGBimg)
    R = double(RGBimg(:, :, 1));
    G = double(RGBimg(:, :, 2));
    B = double(RGBimg(:, :, 3));

    Y  = 0.299 * R + 0.587 * G + 0.114 * B;
    Cb = -0.169 * R - 0.331 * G + 0.5   * B;
    Cr = 0.5   * R - 0.419 * G - 0.081 * B;

    YCbCrimg(:,:,1) = Y;
    YCbCrimg(:,:,2) = Cb;
    YCbCrimg(:,:,3) = Cr;
end
```

The execution time measured with `tic`/`toc` was

```
Elapsed time fast: 6.901098e-02 seconds.
```

Figure 23 shows the result of converting the original RGB image (Figure 22) into YCbCr with the fast implementation custom function, displayed with `imshow(parrots_fast, [])`. Since `imshow` treats the three channels as if

they were RGB, the output appears in false color, which nonetheless clearly highlights differences in luminance and chrominance.

Figure 23: False-color display of the YCbCr image from `my_rgb2ycbcr_fast`.

As expected, the luminance variations (mostly encoding lightness of the parrots' feathers) and the chrominance variations (capturing color differences) become visually distinct in this representation.

4.2 Slow RGB to YCbCr filtering

As opposed to the function defined above, the function `my_rgb2ycbcr_slow` contains two nested `for loops`. The `for loops` cycle respectively over the M and N dimensions of the photo, where M is the number of rows and N the number of columns. In this case all the `YCbCr` were computed simultaneously, employing matrix multiplication.

Due to the impossibility to multiply 2-D matrix (the conversion matrix in this case) with 3-D matrices (RGB pixel information), the command `squeeze(RGBimag(i, j, :)` was used to remove all the dimension of length one. In our case, the input was a $1 \times 1 \times 3$ (RGB components for each single pixel) and the output was a 3×1 vector, containing only the RGB information. The result was saved in `YCbCrimg(i, j, :)`, thus again in a 3-D matrix mapping each pixel to its colour YCbCr component.

The output of this function is equivalent to what obtained before, however,

the time required to compute the final image is higher. The time complexity of the function `my_rgb2ycbcr_fast` is $O(1)$, while for `my_rgb2ycbcr_slow` is $O(N^2)$. Indeed, the execution time of this function, measured using the `tic...toc` function, is:

```
Elapsed time slow: 8.047603e+00.
```

In Figure 24 are displayed all the three components of the images obtained converting the RGB image to YCbCr. Each component is shown as greyscale. The output of the custom designed functions is then compared also to the output of the build-in MATLAB function `rgb2ycbcr` (Image Processing Toolbox). We also measured the time required by this function to produce an output, obtaining:

```
Elapsed time build-in function: 7.669950e-02.
```

The build-in function is, by some micro-seconds, higher than the elapsed time for our custom fast function. This little time difference between our fast function and the build-in function can be explained considering error checks not included in our function, but present in the build-in version. The elapsed time determined, are different depending on the device where the simulation is running; the ratio between the time obtained, instead, is almost equal every time the simulation is run.
The result is as expected, the images are equal to one another, with the images representing the luminance (Y) being the one where the image is the more recognizable, while the in the others the image appears less clear.

4.3 Blur the image using Gaussian low-pass filter

Our objective in this part of the exercise was to filter the image to obtain a blurred effect: that was achieved, in a first moment, by applying a 2-D Gaussian filter to the Y component obtained above. The general shape of a Gaussian 2-D low-pass filter is reported in Equation 1. In our design the standard deviation $\sigma = 5$ and $D(x, y)$ is the distance from the point (x, y) to the centre of the filter, defined as

$$D(x, y) = \sqrt{x^2 + y^2}$$

$$h(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{D^2(x,y)}{2\sigma^2}} \tag{1}$$

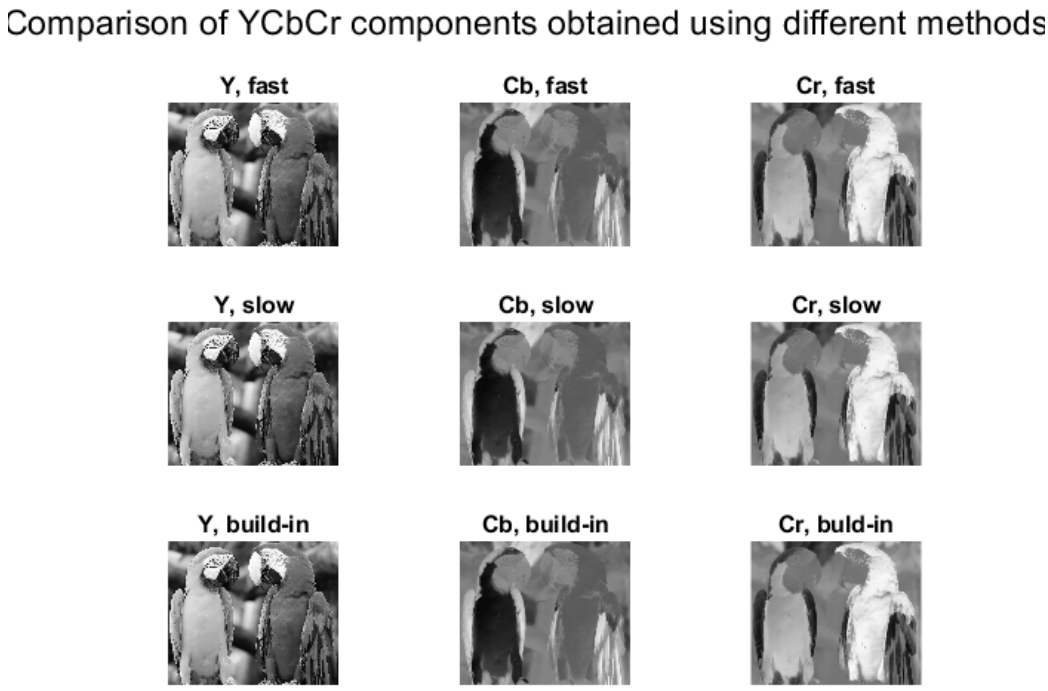


Figure 24: 3×3 subplot comparing all the Y, Cb, and Cr components as greyscale images of the outputs of the user-defined functions `my_rgb2ycbcr_slow` `my_rgb2ycbcr_fast` and build-in function `rgb2ycbcr`

To define all the points, thus to be able to compute the distance, two meshgrid matrices X, Y were defined in the range $[-\frac{Q}{2}, \frac{Q}{2}]$, $Q = 51$, with step 1. In MATLAB to generate meshgrid matrices the command `[X, Y] = meshgrid(`$-\frac{Q}{2}$`:`$\frac{Q}{2}$`)` was employed.

In Figure 25 we reported the centered spectrum of the 2-D Fourier transform of the filter, obtained using the `fft2` MATLAB command. The transform was computed with 128×128 points. The filter has the classical shape of the Gaussian function (bell distribution) over two dimensions.

The obtained filter was used to blur the image in the time domain, thus employing the convolution. To obtain an output matrix of the same dimension as the input, the option *'same'* was used. Since the convolution must be applied in both the directions of the matrix, `conv2` was used to compute the convolution.

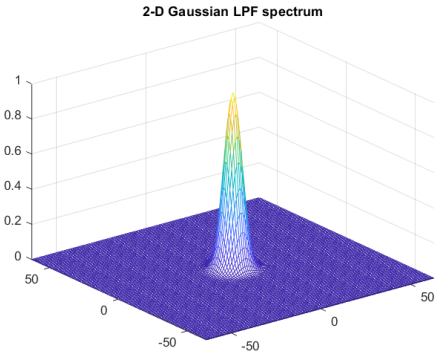


Figure 25: 2D Gaussian low-pass filter obtained with the `mesh` command

Due to the separable property of the 2-D Gaussian filter, the same result obtained here, was achievable using a 1-D Gaussian function, general shape reported in Equation 2, filtering firstly among one direction, and only after, along the other.

$$h(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$

To filter in the two directions, firstly we convolved each column with the 1-D filter, then we used the result obtained and convolved it along all the rows. Again all the convolution, this time done in 1 dimension, thus using `conv`, were done with the option *'same'*, to obtain an output with the right length. Figure 26 shows both the images obtained when blurring the original image with Gaussian low-pass filter. There are no noticeable differences between the Figure 26a, obtained by 1-D Gaussian and Figure 26b, obtained by 2-D Gaussian, thus proving the separable property of the Gaussian filter. As expected, the images resulting have smoother edges, thus achieving the desired blurring effect.

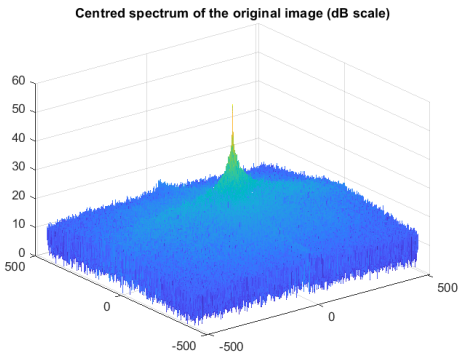
To graphically see the effect of the filter on the image, it is possible to look at the spectrum of the original signal and of the result after the filtering. Both of them are reported in Figure 27, using logarithmic scale, defined as $10log_{10}(1 + |A|)$, with $A = fftshift(fft2(image))$. Figure 27a shows the spectrum of the original image: it is characterized by a main peak in the centre, and lower peaks all around; even though not easily recognizable from this plot, the image shows a cross-shaped pattern, where there are higher frequency components on this cross. Figure 27b, on the other hand,

(a) Y component of the image blurred using the 1-D Gaussian low-pass filter

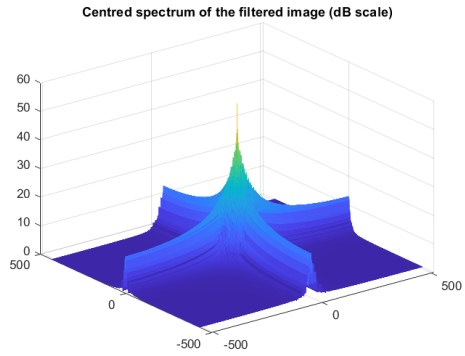
(b) Y component of the image blurred using the 2-D Gaussian low-pass filter

Figure 26

seems much cleaner, presenting only the main peak and the components along the cross-shaped pattern, now clearly visible. This is the result we were expecting: looking at the filter response, (Figure 25), it should be clear that only the component at the centre of the meshgrid are kept, while all the other are lost.



(a) Centred spectrum of Y component of the original image



(b) Centred spectrum of the Y component of the blurred image

Figure 27

4.4 High-pass filtering

For this part of the exercise, a different image than the one used up to now was used. As before, the whole analysis will be conducted on the Y component of the image; in order to obtain that, the custom function `my_rgb2ycbcr_fast` was employed. Figure 28 shows both the original image as read from the file and, on the right, the Y component, as a scale of grey, of the converted image.

(a) Original image

(b) Y component

Figure 28

To generate the high-pass filter we firstly design a low-pass filter, then converted in high-pass using the relation $H_{hp} = 1 - H_{lp}$. We implemented a 2-D Butterworth filter, defined with Equation 3, where $D(u, v)$ is again the distance between the point (u, v) and the origin, considering meshgrid matrices U, V ranging from $[-\frac{P}{2}, \frac{P}{2} - 1]$, with $P \geq 2^{2 \cdot max(M,N)}$, where M, N are the dimension of the image. In our case we set $P = 4096$. D_0 is the filter cut-off frequency, defined as $D_0 = \frac{P}{16} = 256$ and $n = 6$ is the order of the filter. Figure 29 show the centred spectrum of the obtained filter. The filter is flat almost everywhere but at the centre, where it attenuates the components.

$$H_{lp}(u, v) = \frac{1}{1 + (\frac{D(u,v)}{D_0})^{2n}} \tag{3}$$

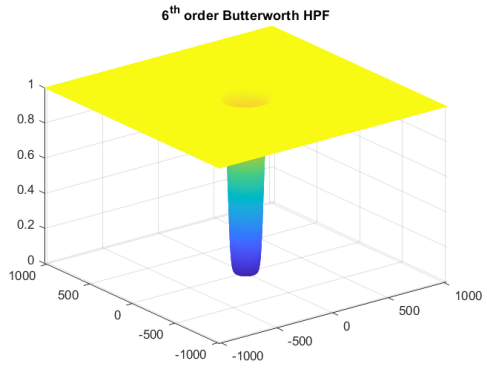


Figure 29: Centred spectrum of the high-pass Butterworth filter of 6th order plotted with `mesh`

This time, we wanted to perform the filtering operation in the frequency domain (the filter response we have calculated is already in the frequency domain), hence we need to calculate the Fourier transform of the image. To achieve that, the command `fft2` was employed, specifying as number of points the dimension of the filter ($P \times P$). The Fourier transform for the original image is reported in Figure 31a. Filtering in the frequency domain consists of simply multiplicating the matrix to be filtered with the filter response. To be able to show the image and save it in a file, the inverse Fourier transform must be applied to the filtered image spectrum, using the `ifft2` command (2-D). The filtered image is shown in Figure 30. The image result being very dark, with only the outline of all the building visible. As

a matter of fact, indeed, the technique of using high-pass filter is employed in professional graphic tools, to obtained sharpening effects (summing the result of the filtering with the starting image).

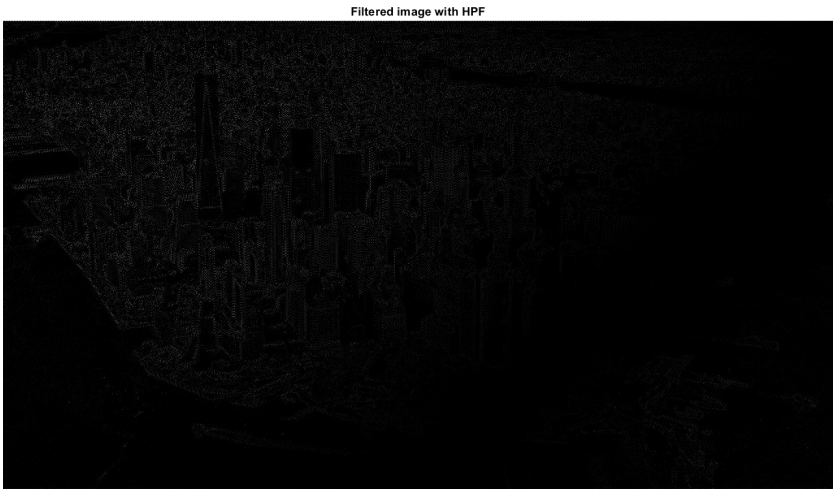
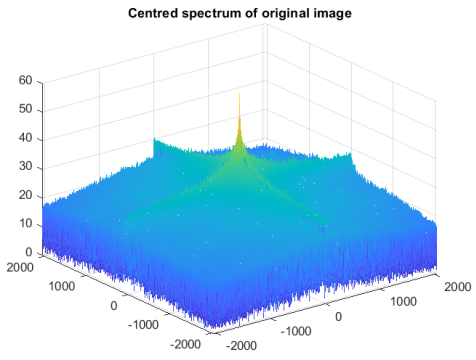
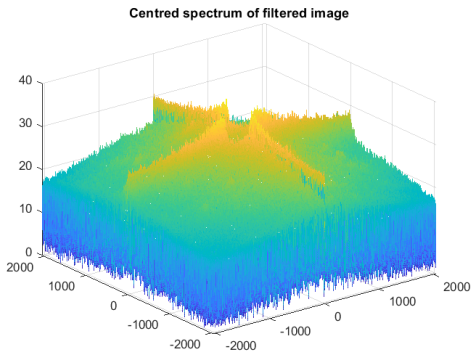


Figure 30: New York view filtered using a high pass filter

Comparing the spectrum before and after the filtering shown respectively in Figure 31a and Figure 31b one may find out that in the Fourier transform of the filtered image the central peak is removed, as expected when using a high pass filter. All the other components stay almost unchanged, keeping their characteristic.



(a) Centred of the original image (log scale)



(b) Centred spectrum of the filtered image (log scale)

Figure 31

5 Conclusion

In this assignment, we had the opportunity to apply various signal processing techniques to both audio and image data. In the first part, we focused on pitch estimation and time-frequency analysis using spectrograms. The custom pitch estimation function we implemented produced results that were consistent with the expected theoretical frequencies, even in more complex cases like bending or voice signals.

Then, we moved on to digital audio synthesis, where we generated notes and chords programmatically, applied ADSR envelopes, and added effects like reverb. This part allowed us to experiment with creating short musical sequences, and, moreover, we developed a simplified version of the intro of Bohemian Rhapsody. This helped us better understand both the potential and the limitations of our synthesis approach.

Finally, we explored image processing by performing color-space conversion from RGB to YCbCr and applying both low-pass (Gaussian) and high-pass (Butterworth) filters. We observed their effects in both the time and frequency domains, confirming how signal processing techniques can be effectively adapted to visual data.

Overall, this assignment helped us understanding the theoretical concepts covered in class and experience practical tools for processing real-world signals.