VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
«DEPARTMENT OF COMPUTATIONAL AND DATA MODELING» OR
«CYBERSECURITY LABORATORY»

Course project no. 2

Fourier filter

Based on Cooley-Tuckey algorithm

Done by:
Kazimieras Vitkus

Supervisor:
Prof. Tadas Meskauskas

Vilnius
2024

Contents

1 Introduction

In the realm of digital signal processing, the Fast Fourier Transform (FFT) and its inverse (IFFT) are pivotal algorithms that enable efficient computation of the Discrete Fourier Transform (DFT) and its inverse (IDFT). These mathematical tools transform time-domain signals into their frequency-domain representations and vice versa, offering profound insights into the frequency components of signals. Implementing these transformations from scratch not only deepens the understanding of their mechanics but also highlights the computational optimizations that FFT introduces over the traditional DFT.

This report details the implementation of FFT, IFFT, DFT, and IDFT algorithms from the ground up. We will explore how these algorithms can be utilized for high-pass and low-pass filtering, allowing us to isolate and analyze specific frequency components within sound signals. By filtering and examining these signals, we gain a better understanding of the underlying structures and characteristics of different audio samples.

2 Algorithm implementation

2.1 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a mathematical operation that transforms a finite sequence of equally spaced samples of a function into a sequence of complex numbers. The DFT is defined by the formula:

$$C_k = \sum_{j=0}^{N-1} f_j \cdot e^{-\frac{2\pi i}{N} kn} \tag{2.1}$$

where f_j is the input signal, C_k is the output signal, and N is the number of samples in the input signal. The DFT algorithm computes the frequency components of the input signal by summing the product of each sample with a complex exponential function. In my implementation, I used the following Python code to calculate the DFT of an input signal:

```
1 def dft(signal):
2
3   N = len(signal)
4
5   C_k = np.zeros(N, dtype=np.complex_)
6
7   for k in range(N):
8
9       for n in range(N):
10          C_k[k] += signal[n] * np.exp(-2j * np.pi * k * n / N)
11
12      C_k[k] /= N
13  return C_k
```

for the inverse DFT, we can use the following code:

```
def idft(signal):

  N = len(signal)

  f_j = np.zeros(N, dtype=np.complex_)

  for j in range(N):

      for k in range(N):
          f_j[j] += signal[k] * np.exp(2j * np.pi * k * j / N)

  return f_j
```

2.2 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an algorithm that computes the Discrete Fourier Transform (DFT) of a sequence of complex numbers. The FFT is an optimized version of the DFT that reduces the number of arithmetic operations required to compute the DFT from $O(N^2)$ to $O(N \log N)$. The Cooley-Tukey algorithm is a popular implementation of the FFT that recursively divides the input sequence into smaller sub-sequences and combines the results to compute the DFT.

In my implementation, I used the following Python code to calculate the FFT of an input signal:

```
def fft(signal):
  N = len(signal)

  if N <= 1:
      return signal

  fft_even = fft(signal[::2])
  fft_odd = fft(signal[1::2])

  k = np.arange(N // 2)
  W_n_k = np.exp(-2j * np.pi * k / N)

  A_k = fft_even
  B_k = fft_odd * W_n_k

  C_k = np.zeros(N, dtype=np.complex_)
  C_k[:N // 2] = A_k + B_k
  C_k[N // 2:] = A_k - B_k

  return C_k
```

For the inverse FFT, we can use the following code:

```
def ifft(signal):
N = len(signal)

if N <= 1:
    return signal

ifft_even = ifft(signal[::2])
ifft_odd = ifft(signal[1::2])

k = np.arange(N // 2)
W_n_k = np.exp(2j * np.pi * k / N)

A_k = ifft_even
B_k = ifft_odd * W_n_k

f_j = np.zeros(N, dtype=np.complex_)
f_j[:N // 2] = A_k + B_k
f_j[N // 2:] = A_k - B_k

return f_j / N
```

3 Analysis of algorithms and implementation

3.1 Comparison of DFT and FFT

In theory and practice, the Fast Fourier Transform (FFT) is significantly faster than the Discrete Fourier Transform (DFT) for computing the frequency components of a signal. The FFT algorithm reduces the number of arithmetic operations required to compute the DFT from $O(N^2)$ to $O(N \log N)$, making it much more efficient for large input signals. For the example of comparison, signal was generated using numpy random function with length varying from 2^2 to 2^7.

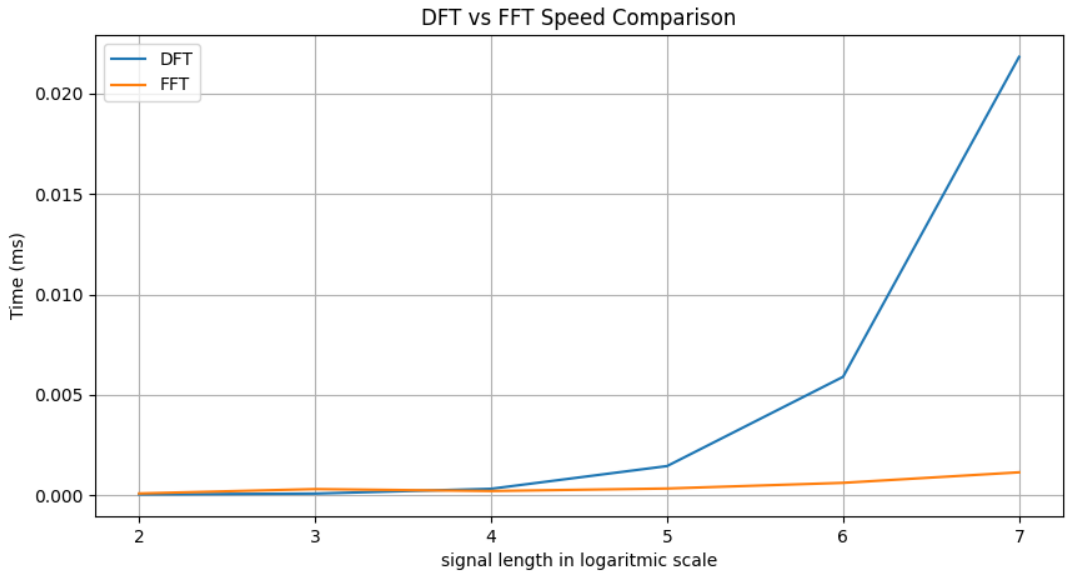

Figure 1. Comparison of computation time between DFT and FFT

As can be seen in figure 1, the FFT algorithm is significantly faster. The trend line for DFT seems to be increasing exponentially, while the FFT trend line is almost linear. This demonstrates the computational efficiency of the FFT algorithm compared to the DFT algorithm. However, the chart may vary depending on the length of the input signal used for comparison.

3.2 Comparison of FFT from scratch and numpy FFT

The numpy library provides a built-in FFT function that is highly optimized and efficient for computing the Discrete Fourier Transform (DFT) of a signal. In this section, we compare the performance of the FFT algorithm implemented from scratch with the numpy FFT function. The signal was generated using numpy random function with length varying from 2^2 to 2^{16}.

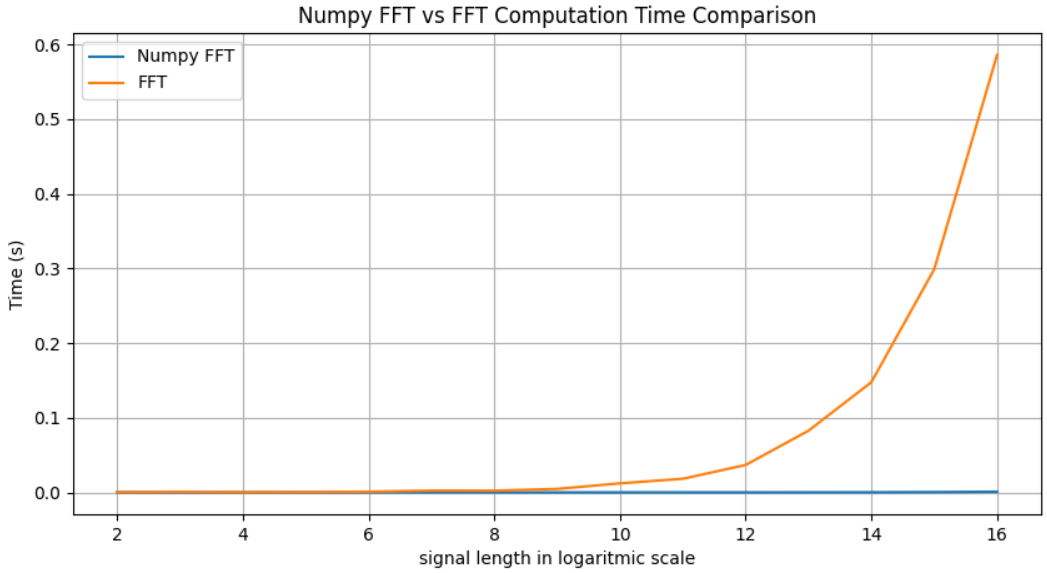

Figure 2. Comparison of computation time between FFT from scratch and numpy FFT

As can be seen in figure 2, the numpy FFT function is significantly faster than the FFT algorithm implemented from scratch. After certain length of the input, implementation from scratch seems to differ exponentially in computation times. It means, implementation quality is suboptimal and needs to be improved.

3.3 Fitting requirements for algorithm reduces performance

One of requirements for signal that needs to be processed with FFT - it's length must be a power of 2. One of the provided solutions - zero padding, which means adding zeros to the end of the signal to make it's length a power of 2. However, this solution may reduce the performance of the FFT algorithm. In this section, we compare the performance of the FFT algorithm with and without zero padding. The signal was generated using numpy random function with length varying from 2^2 to 2^{16} and for comparison, one signal has 0 appended to invoke zero padding function.

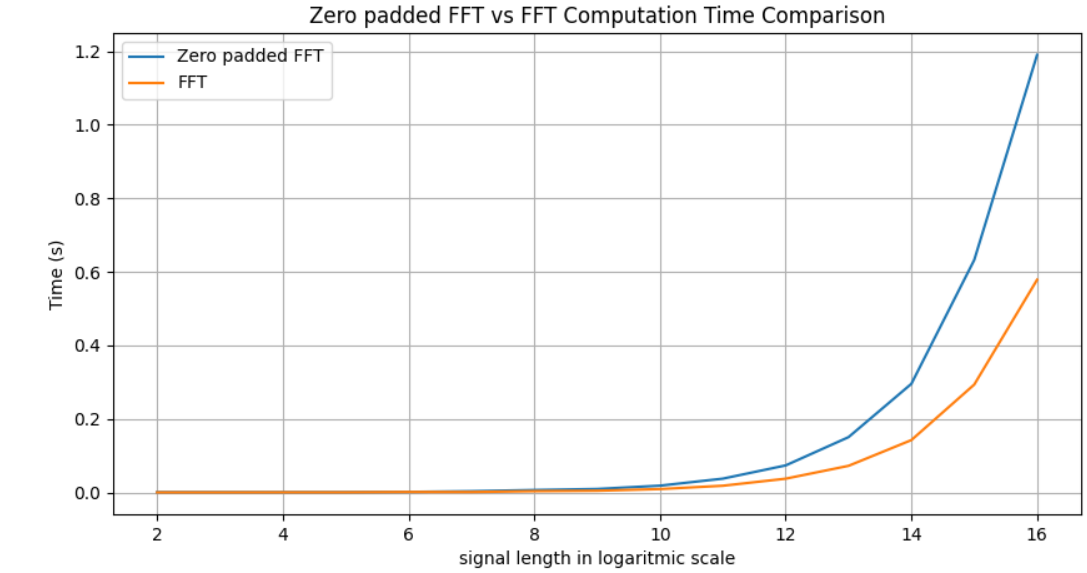

Figure 3. Comparison of computation time between FFT with and without zero padding

As can be seen in figure 3, the signal, that invokes zero padding, takes more time to compute the FFT algorithm. The relation between computation times, is that if zero padding was invoked, the computation equals to the computation time of signal with length 2^{k+1}.

4 Audio signal filtering

For the demonstration of the Fourier filter, we will use a sample audio signal. The signal was downloaded from the following website: https://freesound.org/people/InspectorJ/sounds/412067/.

The signal in an unprocessed form is presented in figure 4. It is based on framerate of 44100 Hz and has 2 channels. The signal consists of 115327 frames and is of 2.5 seconds duration. The signal is not compressed and saved in the wave (WAV) format.

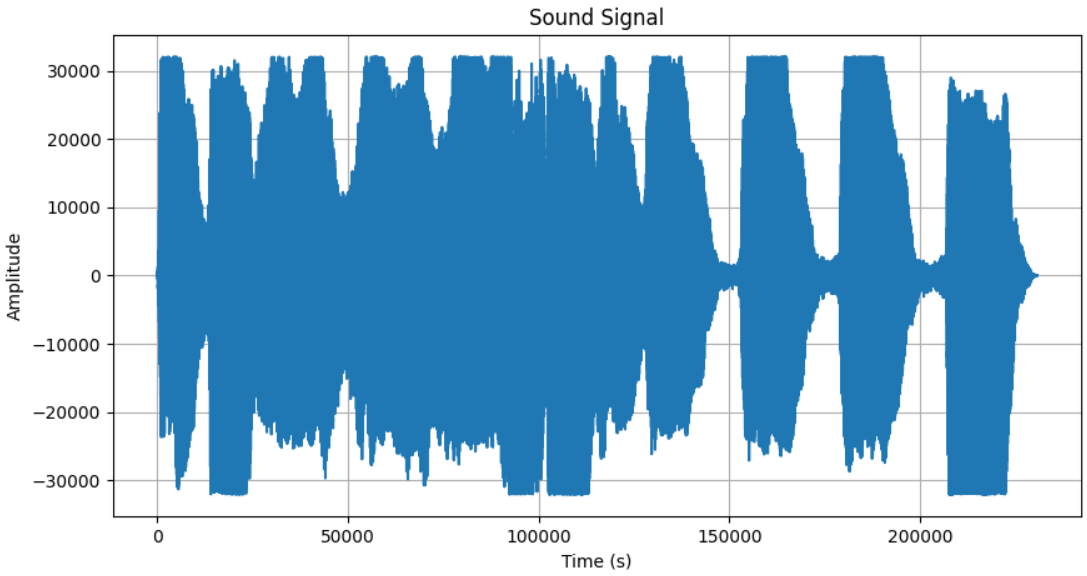

Figure 4. Unprocessed audio signal

The power spectrum of the signal is presented in figure 5. The power spectrum shows the frequency components of the signal. The x-axis represents the frequency in Hz, while the y-axis represents the power of the signal at each frequency in decibels (logarithmic scale). The power spectrum provides insights into the frequency components of the signal and can be used to identify specific frequency ranges for filtering.

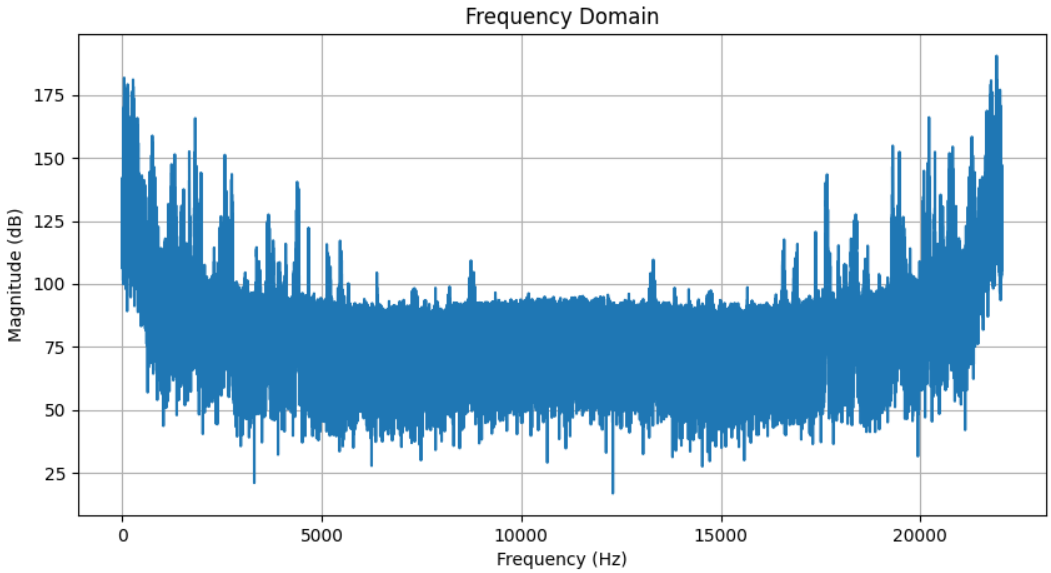

Figure 5. Power spectrum of the audio signal

4.1 High-pass filtering

High-pass filtering is a signal processing technique that allows high-frequency components of a signal to pass through while attenuating low-frequency components. In this section, we will implement a high-pass filter using the Fourier transform and apply it to the audio signal. In the scope of this task, prebuilt function was made, that takes in signal and frequency that is The threshold frequency for the high-pass filter. The function returns the filtered signal.

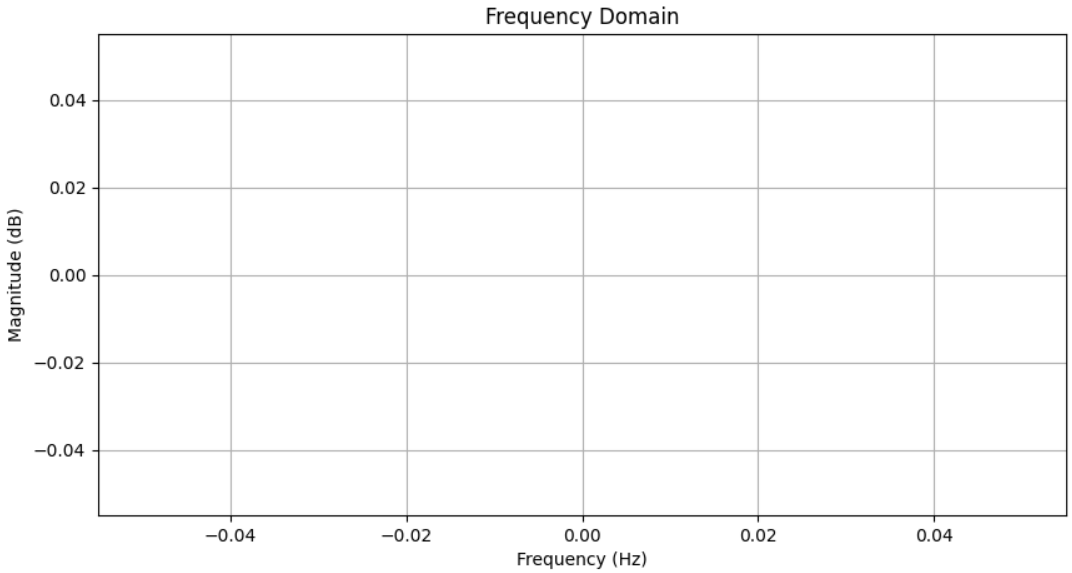

Figure 6. High-pass filtered audio signal