Signal Analysis and Processing Task 2 (Fourier Filter based on Cooley–Tukey Fast Fourier Transform (FFT) algorithm)

Justinas Lekavičius

May 27, 2022

Abstract

This is the report for Signal Analysis and Processing project number 2 – implementation of Fourier Filter based on Cooley–Tukey Fast Fourier Transform (FFT) algorithm (variant number 6). The purpose is to implement an FFT-based algorithm and demonstrate its performance.

Contents

---

1 List of sound signals analyzed

The sound signals used for the project were acquired from Sound Jay data library, as .wav type sound files [1]. Several different signals were used, some are longer, while others are shorted in length – seven signals in total. The following numbered list shows the sound signal used (the title of the file), and its source link. The signals are numbered exactly as they are loaded in the project notebook (source code). The signal illustrations are also displayed in figures 1 to 7 respectively (the signals are already preprocessed), detailing their amplitudes (y axis) for one channel, and their length in seconds (x axis).

1. bag-zipper-2.wav – sound of a bag closing. Download link: `https://www.soundjay.com/cloth/bag-zipper-2.wav`
2. empty-bullet-shell-fall-01.wav – sound of an empty bullet shell dropping on the floor. Download link: `https://www.soundjay.com/mechanical/empty-bullet-shell-fall-01.wav`
3. footsteps-4.wav – sound of four footsteps. Download link: `https://www.soundjay.com/footsteps/footsteps-4.wav`
4. creaking-door-2.wav – sound of a door opening and making a creaking sound. Download link: `https://www.soundjay.com/door/creaking-door-2.wav`
5. matches-3.wav – sound of striking a match against a matchbox, lighting it on fire. Download link: `https://www.soundjay.com/nature/matches-3.wav`
6. coin-drop-4.wav – sound of a coin dropping on top of other coins. Download link: `https://www.soundjay.com/misc/coin-drop-4.wav`
7. button-46.wav – sound of a button (used for websites, menu navigations, etc.). Download link: `https://www.soundjay.com/buttons/button-46.wav`

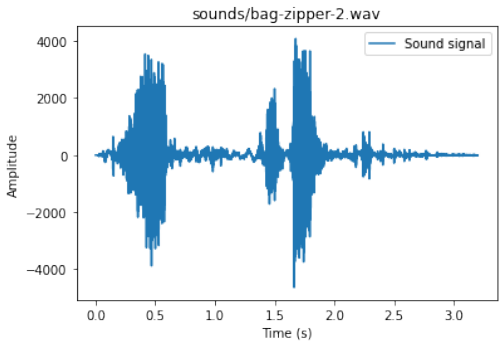

Figure 1: Visual graph of the bag-zipper-2.wav signal.

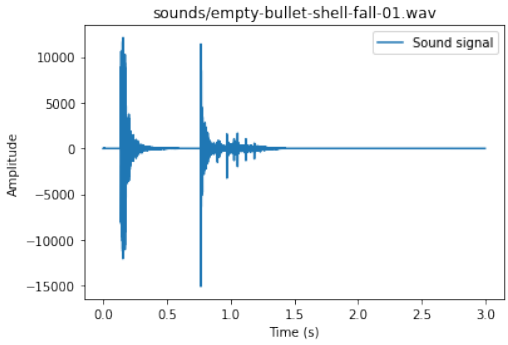

Figure 2: Visual graph of the empty-bullet-shell-fall-01.wav signal.

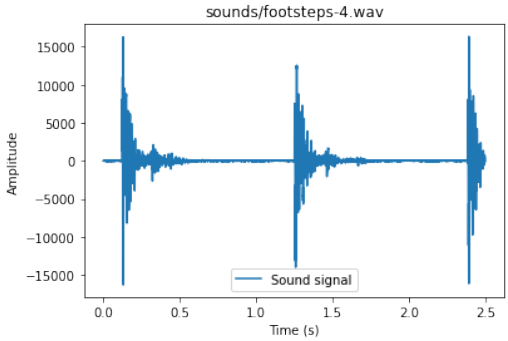

Figure 3: Visual graph of the footsteps-4.wav signal.

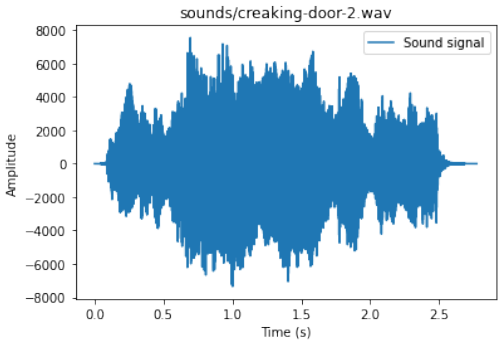

Figure 4: Visual graph of the creaking-door-2.wav signal.

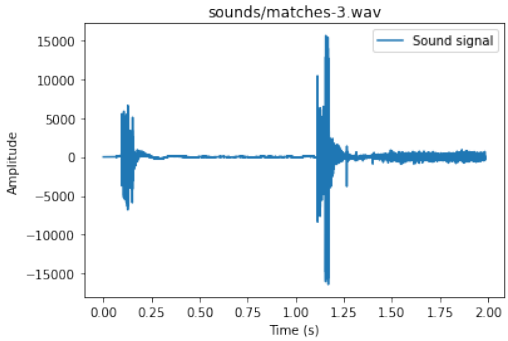

Figure 5: Visual graph of the matches-3.wav signal.

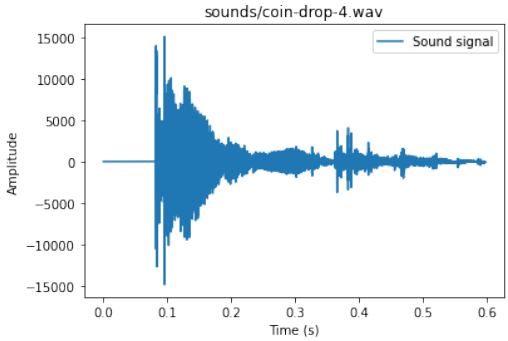

Figure 6: Visual graph of the coin-drop-4.wav signal.

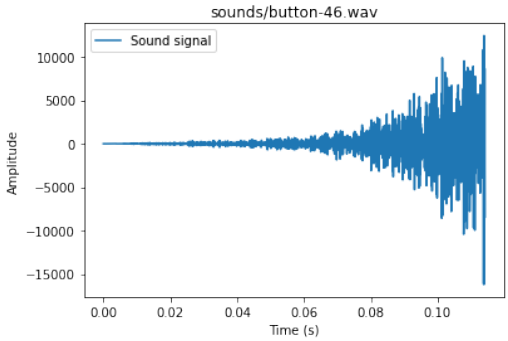

Figure 7: Visual graph of the button-46.wav signal.

2 Signal preprocessing

The signals were preprocessed via the main Python code to either simplify them or to reduce computation times.

Firstly, the original sound signals have two channels – left and right (stereo sound), therefore to simplify the signal, it has been converted to mono via Python source code. Furthermore, the Fourier Transform algorithm has a presumption that N is a power of 2 and is a whole number [2],

$$N = 2^n, \text{where } \ n > 0 \text{ is a whole number.}$$

Therefore, signals have to be preprocessed for this algorithm. In case number of N does not satisfy the requirement - the signal will be shortened. This is done by finding the nearest previous power of 2 and setting it as the length of the signal. Although the reduction (discarding data) is not recommended, in this case it was done to increase performance, as working with larger data was more technically difficult. To calculate the previous best N value that would satisfy the requirement - simply drop set bits from N except the very last set bit. Then, the difference of old N value and new N value is omitted from the end of the signal (removing last elements, shortening the signal).

3 The source signals

For reference, the source signals coin-drop-4.wav and button-46.wav are once again displayed in Figures 8 and 9 respectively for easier comparisons. It should be noted once again that the source signals are already preprocessed (shortened and converted from stereo to mono).

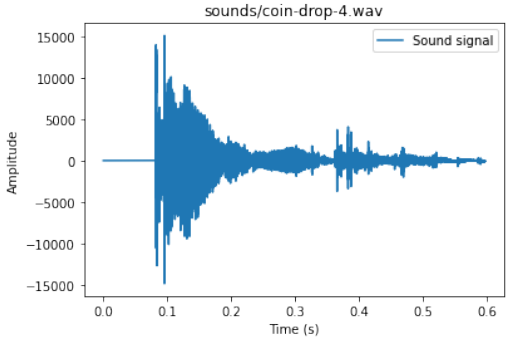

Figure 8: Visual graph of the coin-drop-4.wav source signal.

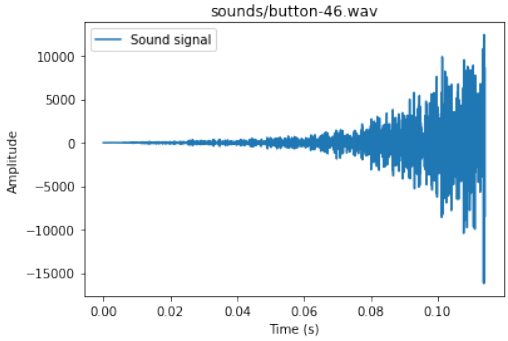

Figure 9: Visual graph of the button-46.wav source signal.

4 Fourier Transform implementation

This section covers the procedures for implementing the Fourier Transform (both DFT and FFT), the results (including visual graphs) and insights on successful and unsuccessful implementations. Because of performance issues when testing the DFT and FFT implementations with real data, shorter signals were selected – 6th and 7th, coin-drop-4.wav and button-46.wav respectively.

4.1 Discrete Fourier Transform

Firstly, Discrete Fourier Transform (DFT) was implemented (both normal and inverted versions). Although its computation time is significantly slower when compared to FFT, it still provided satisfactory results. It was implemented based on Equations 1 and 2 (based on learning material).

Regular DFT [2]

$$c_k = 1/N \sum_{j=0}^{N-1} f_j e^{-i\frac{2\pi}{N}jk}, \qquad k = 0, 1, ..., N-1 \tag{1}$$

Inverse DFT [2]

$$c_k = \sum_{j=0}^{N-1} f_j e^{i\frac{2\pi}{N}jk}, \qquad k = 0, 1, ..., N-1 \tag{2}$$

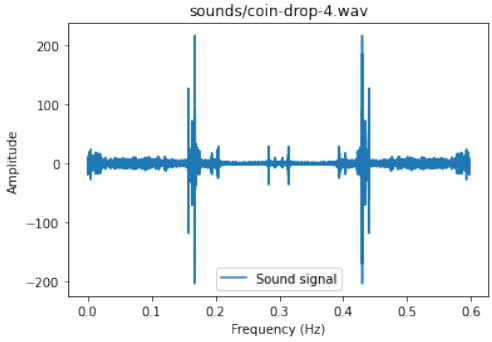

Figure 10: Visual graph of the DFT applied to coin-drop signal.

The DFT application to signal coin-drop-4.wav is displayed in Figure 10. The application appears to be successful, as the DFT displays symmetry. When applying inverse DFT to the DFT signal, it returns the original sound signal (as displayed in Figure 11, indicating that DFT implementation was successful.

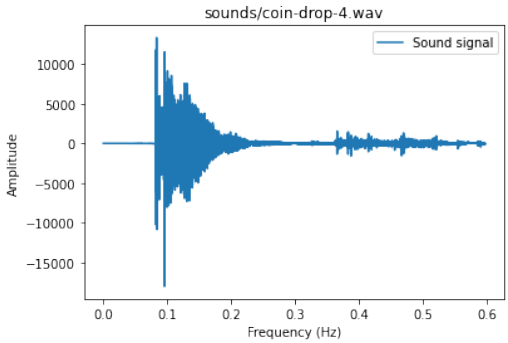

Figure 11: Visual graph of inverse DFT applied to DFT, resulting in the original coin-drop-4.wav signal.

When comparing the source signal and the returned signals, however, there can be seen subtle differences in amplitude, even though the whole form appears very similar. Therefore, the reconstructed signal is not 100 percent identical to the original sound.

Figures 12 and 13 show the same procedure of applying DFT and then using inverse DFT to reconstruct the original signal by using the button-46.wav signal as the input. Once again, it appears that the implementation was a success, although inefficient due to the nature of DFT, resulting in long computational times.

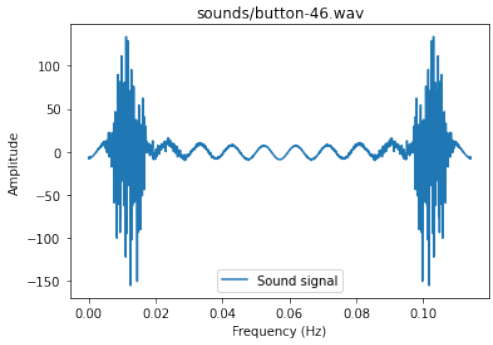

Figure 12: Visual graph of the DFT applied to button-46.wav signal.

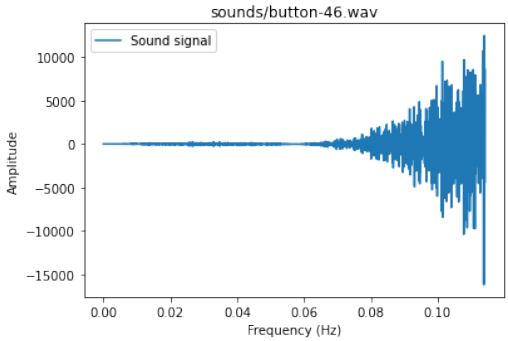

Figure 13: Visual graph of inverse DFT applied to DFT, resulting in the original button-46.wav signal.

4.2 Fast Fourier Transform

Then, the Fast Fourier Transform (FFT) was implemented (version with recursion). The algorithm can be described with these formulas (based on learning material) [2]:

$$C_k = \sum_{j=0}^{N/2-1} f_{2j} W_{N/2}^{jk} + W_N^k \sum_{j=0}^{N/2-1} f_{2j+1} W_{N/2}^{jk}, \quad k = 0, 1, ..., N-1 \tag{3}$$

$$C_{k+N/2} = \sum_{j=0}^{N/2-1} f_{2j} W_{N/2}^{jk} - W_N^k \sum_{j=0}^{N/2-1} f_{2j+1} W_{N/2}^{jk}, \quad k = 0, 1, ..., \frac{N}{2} - 1 \tag{4}$$

$$c_k = \begin{cases} A_k + W_N^k B_k, & k = 0, 1, ..., \frac{N}{2} - 1 \\ A_{k-\frac{N}{2}} - W_N^{k-\frac{N}{2}} B_{k-\frac{N}{2}}, & k = \frac{N}{2}, \frac{N}{2} + 1, ..., N-1. \end{cases} \tag{5}$$

FFT function is applied to coin-drop-4.wav signal, as displayed in Figure 14. As with DFT, the signal displays desired characteristics, such as symmetry, and the result is similar to when DFT was applied in Figure 10. However, the implementation of FFT was significantly faster in terms of computational time.

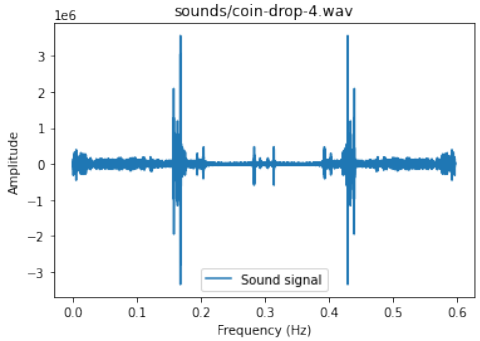

Figure 14: Visual graph of FFT applied to coin-drop-4.wav signal.

Unfortunately, reconstruction of the original signal by applying inverse FFT was unsuccessful. The result is shown in Figure 15. When compared to the original sound as shown in Figure 8, it appears that the signal was reconstructed only partially, as well as repeated two times. Furthermore, apparently the reconstruction is the mirrored version of the sound. This indicates that there may have been an issue in inverse FFT implementation.

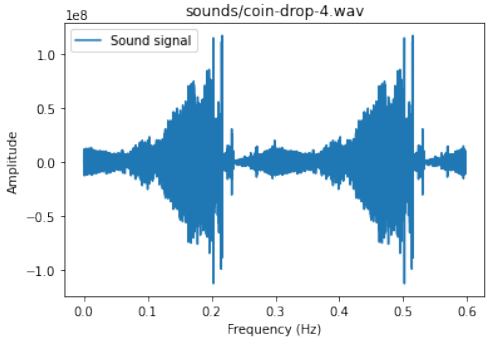

Figure 15: Visual graph of inverse FFT applied to the FFT, resulting in unsuccessfully reconstructed coin-drop-4.wav signal.

The FFT application is also successful for the button-46.wav signal as seen in Figure 16, and the same anomalies as for the previous signal can be seen in Figure 17, when attempting to reconstruct the signal.

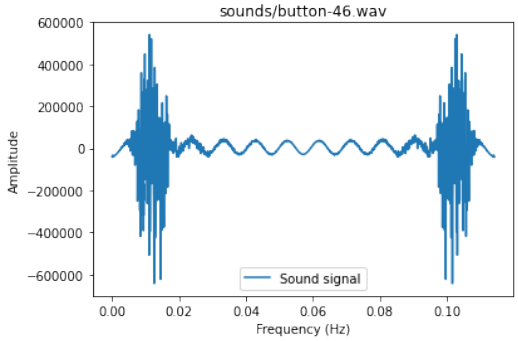

Figure 16: Visual graph of FFT applied to the button-46.wav signal.

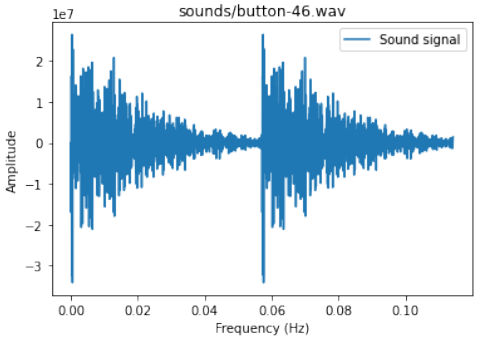

Figure 17: Visual graph of inverse FFT applied to the FFT, resulting in unsuccessfully reconstructed button-46.wav signal.

4.3 FFT and DFT Computational Time Comparison

Once FFT and DFT were implemented, their computational time was compared. The parameter for both functions "Inverse" was the default False, thus non-inverted FFT and DFT functions were tested. The computational time was determined using Python's timeit function.

The FFT and DFT functions were tested with the same signal – button-46.wav. Unfortunately, DFT function computational time was extremely long for other, longer signals, therefore a much shorter signal was chosen for the test. As displayed in the code fragment below, the FFT function performed significantly better than DFT.

```
print("FFT test with real data: ")
%timeit FFT(data[6])
print("DFT test with real data: ")
%timeit DFT(data[6])

FFT test with real data:
1 loop, best of 5: 252 ms per loop
DFT test with real data:
1 loop, best of 5: 1min 9s per loop
```

5 Conclusions

In conclusion, the implementations of DFT and FFT algorithms were successful, and was tested with several different sound signals, however, FFT inverted version of the algorithm was not successfully implemented, and produced unexpected results implementation was unsuccessful. Due to the nature of the signals, they were preprocessed for better performance and simplicity, this included converting from stereo to mono and reducing the signal length by finding the previous power of 2 for the signal sample count (although the shortening of the signal is not recommended, and was only done for the sake of better technical performance). Furthermore, there was difficulty in comparing the computational time of DFT and FFT algorithms using the same signal that is longer, thus a much shorter sound signal was used for the test. This, however, can actually be a fair point for why FFT should be used, as it is significantly faster than DFT and even suitable for longer signals.

6 Code fragment

The following code fragment contains the implementations of the FFT/FFT inverse and DFT/DFT inverse algorithms for the second task.

```
# DFT and FFT implementations

# Mokymo priemon  - 65th & 66th equations

def DFT_function(data, inverted = False):

    N = data.shape[0] # Samples
    ck = [] # Fourier coefficients
    multiplier = -1j if inverted else 1j
    divisor = 1 if inverted else N

    for j in range(0, N - 1):

        sum = 0

        for k in range(0, N - 1):
            sum += data[k] * numpy.e ** (multiplier * (2 * numpy.pi
                                                / N) * j * k)

        ck.append(sum / divisor)

    return ck

# Mokymo priemon  - 79th equation

def FFT_function(data, inverse = False):

    N = data.shape[0] # Samples
    multiplier = -1j if inverse else 1j
    divisor = N if inverse else 1

    if N == 1:
      return data

    ck = numpy.zeros(N, dtype = numpy.complex_)
    Wn = numpy.exp(multiplier * 2 * (numpy.pi / N)) * ( 1 / divisor
                                        )

    ck_even = FFT_function(data[0:N:2])
    ck_odd = FFT_function(data[1:N:2])

    for k in range(0, numpy.floor_divide(N, 2)):

        ck[k] = (ck_even[k] + numpy.power(Wn, k) * ck_odd[k])
        ck[k + numpy.floor_divide(N, 2)] = (ck_even[k] - numpy.
                                            power(Wn, k) * ck_odd[k])

    return ck
```

References

[1] Sound Jay. Sound Jay's free sound effects web site. https://www.soundjay.com/, 2022.

[2] Tadas Meškauskas. Signalu analizė ir apdorojimas. https://klevas.mif.vu.lt/~meska/SAA/T_Meskauskas_-_SAA_-_Mokymo_Priemone_LT.pdf, 2021.