

ISS PROJECT 2021/22

 **Skuratovich Aliaksandr**
2BIT student.
Brno University of Technologies
Brno, Božetěchova 1/2
xskura01@vutbr.cz

January 5, 2022

ABSTRACT

In this project there has been implemented simple signal analysis with signal filtering.

The signal was originally taken from the TIMIT (2) database, which is a famous corpus of phonemically and lexically transcribed speech. Four cosine waves imitating noise were added to the signal.

Project has been implemented in Python notebook (.ipynb extension) primarily using google colab platform¹.

¹google colab platform.

Contents

1	Project files	3
1.1	Project structure	3
1.2	Implementation details	3
2	Basic assignment	3
2.1	Basics	3
2.2	Preprocessing and framing	4
2.3	DFT	5
2.4	Spectrogram	7
2.5	Cosine frequencies	8
2.6	Signal generation	9
2.7	Filtering	11
2.8	Zero-Phase response	14
3	Bonus assignment	16
3.1	Increasing frequency resolution	16
3.2	Determining amplitudes of cosines	18
3.3	Determining the phases of cosines	19
3.4	Generating cosines and subtracting a noisy signal	20
4	Conclusion	21

1 Project files

1.1 Project structure

There are two subdirectories in the project folder. First directory is `src/` containing `xskura01.ipynb` file and the second one is `audio/` for audio files in .wav format with sampling frequency of 16kHz.

1.2 Implementation details

As it has been said, project was implemented in Python, so there are a couple of libraries used in the project. List of the libraries used in the project: [`numpy`, `scipy`, `soundfile`, `matplotlib`, `IPython`, `regex`].

Also, there will be a need to upgrade some of this libraries (See the 1.st cell in `xskura01.ipynb`)

2 Basic assignment

2.1 Basics

Signal properties

Sampling frequency of the given signal (hereinafter just F_s) is 16kHz.

The signal is 2.0864375 seconds long. By multiplying with F_s we get the length of the signal in samples, which is 33383 samples.

	Value	Index
max	0.124053955078125	14184
min	-0.12115478515625	13892

Table 1: Signal properties: min, max.

Plot of an input signal

The input signal itself. Here, we can already see that signal has noise, because there are non-zero values in the signal while the person is not speaking.

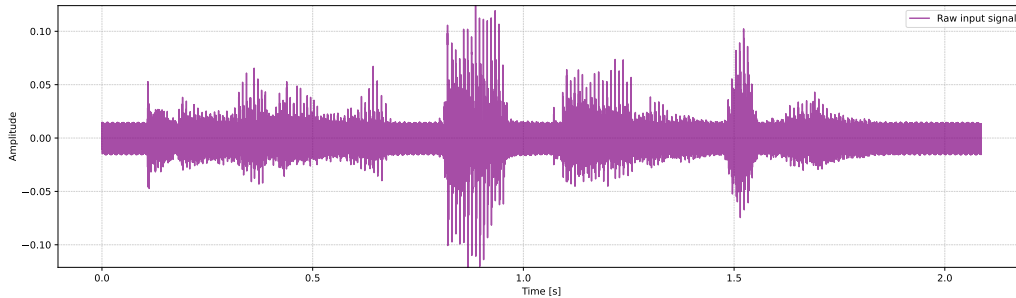


Figure 1: Raw input signal.

2.2 Preprocessing and framing

Normalization of the signal

To work with the signal, we first need to get rid of DC component by subtracting signal's mean value, then there is need to normalize the signal by dividing it by the maximum of its absolute value.

After applying the following functions we get a signal between -1...1.

```

1 def absmax(y):
2     return np.max(np.abs(y))
3
4 def mean_and_normalize(arr):
5     return (arr - np.mean(arr)) / absmax(arr)

```

Framing

Next thing we have to do is to divide the signal into an array of overlapping frames. Because given array is pretty long (there is more than 30000 elements in it), I decided to use numpy's `stride_tricks` to manipulate with array metadata, because it is faster and takes less lines of code to implement.

The algorithm itself was inspired by the article [Advanced NumPy: Master stride tricks with 25 illustrated exercises](#)². Also, the equality for N_{row} was taken from ZRE textbook (1).

Note: I store frames as rows of a matrix and Transpose them when plotting a spectrogram.

```

1 def make_overlapping_frames(arr, row_len=1024, overlap=None):
2     # custom overlapping size can be set. Otherwise, it is a half of a window size.
3     overlap = row_len // 2 if overlap == None else overlap
4     # N_row = 1 + floor((len(arr) - row_len) / overlap)
5     N_row = 1 + np.floor((len(arr) - row_len) / overlap).astype(np.int64)
6
7     strides = (overlap * arr.itemsize, arr.itemsize)
8     return np.lib.stride_tricks.as_strided(
9         arr,
10        shape=(N_row, row_len),
11        strides=strides
12    )

```

The implementation above is a very efficient one and it definitely outperforms same functions with python for cycles. This implementation takes only $8.47 \mu s$ to create an array of frames from a signal, which length is almost 34000 elements!

²[Advanced NumPy: Master stride tricks with 25 illustrated exercises](#)

Analyzed frame

The last part of this section is about choosing good frame with periodic properties.

My choice is the 27th frame, because here it is clearly seen, that waveform looks very familiar to a vowel [e:] (like in a word "father", for example).

Time interval of the given frame is: (0.864, ..., 0.9279375) seconds, length is 0.064 seconds.

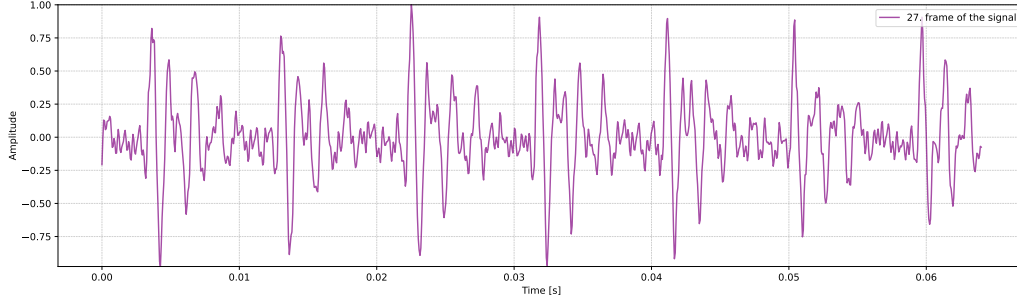


Figure 2: Beautiful frame, [e:].

2.3 DFT

Theory

We need a Discrete Fourier transform to convert a signal from time to frequency domain. In the frequency domain, we will be able to see how much individual frequencies are represented in our signal. To transform the signal from time domain all we need is a big well-known sum over all n from 0 to $N - 1$. Also, K is equal to N .

$$X_k = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\omega \frac{n}{N} k}$$

We also can represent the whole sum with as a matrix by vector multiplication:

$$\begin{pmatrix} e^{-j\omega \frac{0}{N} \cdot 0} & \dots & e^{-j\omega \frac{(N-1)}{N} \cdot 0} \\ \vdots & \ddots & \vdots \\ e^{-j\omega \frac{0}{N} \cdot (K-1)} & \dots & e^{-j\omega \frac{(N-1)}{N} \cdot (K-1)} \end{pmatrix} \cdot \vec{x} = \vec{X}$$

Implementation

The whole function works as following: firstly, we create a DFT matrix, then just apply a transformation on signal. In case, if signal is two-dimensional we basically iterate over its frames (rows).

It is worth mentioning that only a half of the DFT matrix is created, because the Discrete Fourier transform itself is symmetric and there is no need to perform more multiplications - just multiply two times less and concatenate matrix with the flipped one.

```

1 def dft(frames, N=1024):
2     DFT_mat2 = create_DFT_mat(N)
3     def _dft(signal):
4         res = np.dot(DFT_mat2, signal)
5         res = np.concatenate((res, np.flip(res)[1:len(res)-1]))
6         return res
7     return np.apply_along_axis(_dft, -1, frames)

```

DFT matrix initialization is implemented in the following way.

```

1 def create_DFT_mat(N=1024):
2     N_2 = N // 2
3     jomega = -1j * 2*np.pi / N_2
4     # Frames then are stored as rows, so there will be a need
5     # to transpose a matrix while creating a spectrogram.
6     return (np.fromfunction(
7         lambda k, n: np.exp(jomega * k * n), (N_2 + 1, N),
8         dtype="complex128")
9 )

```

Because there is no python cycles in these functions, my implementation of DFT is a pretty efficient. Transformation of an array with 1024 elements takes only 63.5 ms in average.

Moreover, we do not even need to initialize a matrix every time the `dft()` function is called, but only once. Hence, the matrix initialization can easily be moved outside the function (which makes the size of the DFT matrix constant, though, but there is no need to change N in this project).

In this case, transformation of an array with 1024 elements takes only 430 μ s, which is roughly 150 times faster than the current implementation.

Applying a transformation

Next step is to apply a transformation on a chosen frame, plot two signals (`dft()` and `np.fft.fft()` results), and compare them using `numpy.allclose()` function.

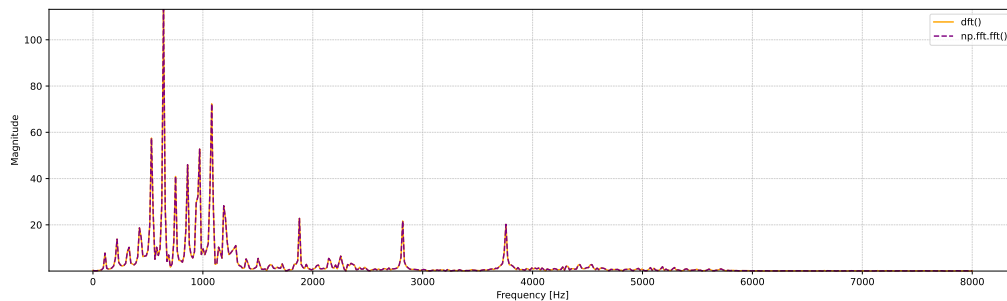


Figure 3: Two transformed signals.

As expected, results are the same (we can see it from the plot). Also, function `np.allclose()` returns `True`.

2.4 Spectrogram

Implementation

Next task is to create spectrogram. I decided to implement my own function for creating and plotting the spectrogram.

I have to mention that functions from `scipy` and `matplotlib` apply a windowing function on each frame, that is why a spectrogram from these libraries looks a little bit sharper.

You can see my implementation of function `spectrogram()` below.

```

1  def create_spectrogram(signal, NFFT=1024, noverlap=None, Fs=16000):
2      def shorten(arr):
3          newlen = 1 + (len(arr) // 2)
4          return arr[:newlen]
5      overlap = NFFT // 2 if noverlap == None else noverlap
6
7      # Mean, normalize and create overlapping frames once again to imitate
8      # implementation of function from scipy library.
9      normalized = mean_and_normalize(signal)
10     frames = make_overlapping_frames(normalized, NFFT, overlap)
11
12     transformed = dft(frames, N=NFFT)
13
14     # Then, there is need to cut off half of frequencies, because
15     # there are identical.
16     freq_half = np.apply_along_axis(shorten, -1, transformed)
17
18     # Time and frequency arrays are needed to plot a spectrogram.
19     t = np.linspace(0, (len(signal)/Fs), num=freq_half.shape[0])
20     f = np.linspace(0, Fs//2, num=freq_half.shape[1])
21
22     return (f, t, (np.absolute(freq_half) ** 2).T)

```

In the next function, we compute a log-power spectrum from transformed frames by using the formula

$$P[k] = 10 \log_{10} |X[k]|^2$$

and then plot it.

```

1  def spectrogram(signal, Fs=16000, NFFT=1024, noverlap=512):
2      overlapping = NFFT//2 if noverlap == None else noverlap
3
4      f, t, sgr = create_spectrogram(signal, NFFT=NFFT, noverlap=noverlap)
5      sgr = 10 * np.log10(sgr+1e-20)
6
7      fig, ax = plt.subplots(figsize=(7,10))
8
9      ax.imshow(sgr, interpolation='nearest', aspect='auto',
10              origin='lower', extent=[0, len(signal)/Fs, 0, Fs/2])
11
12
13     fig.gca().set_xlabel('Time [s]')
14     fig.gca().set_ylabel('Frequency [Hz]')
15     fig.tight_layout()
16     savefig("spectrogram")

```

Spectrogram plot, cosine noise

Looking at the spectrogram, it is obvious that the first cosine is sitting roughly under a frequency of 1kHz and frequencies of other cosines are multiplies of the first one.

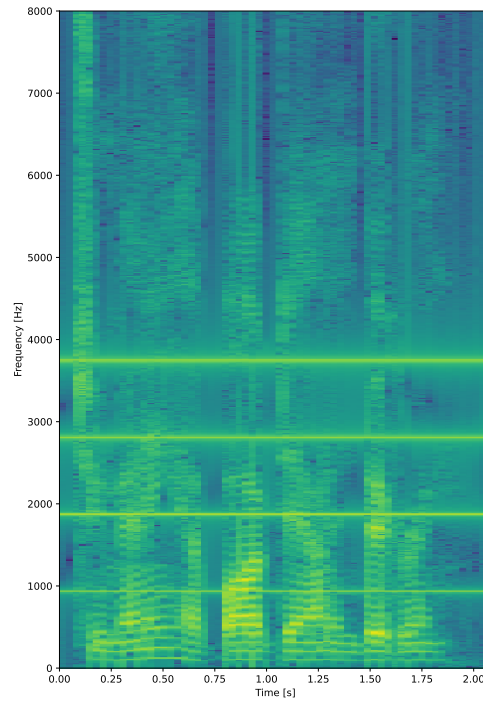


Figure 4: Spectrogram.

2.5 Cosine frequencies

Cosine noise generation in the frequency domain

Here we extract four cosines from the spectrum of the signal and then check whether they are harmonically related.

We can extract cosines from the spectrogram "by hand" or use `numpy.argmax()` function. I Decided to use the second option. The code for generating these cosines in frequency is shown below:

```

1  freq_res = Fs / N
2  def find_frequencies(frames):
3      threshold = 20
4      frame0 = np.abs(dft(frames[0]))[:N//2]
5
6      indices = np.argmax(frame0 > 20).ravel()
7      frequencies = (freq_res * np.argmax(frame0 > threshold)).ravel()
8      magnitudes = frame0[(frequencies/freq_res).astype(np.int64)]
9
10     return indices, frequencies, magnitudes
11
12 indices, frequencies, magnitudes = find_frequencies(frames)

```

The first cosine with frequency f_1 is sitting on the 60th frequency bin with an approximate frequency of 937.5Hz.

There is word 'approximately' here, because $\frac{F_s}{N} = 15.6\text{Hz}$, which means that one frequency bin contains 15.6Hz.

For more precise solution (see subsection 3.1).

Comparison of generated noise and original signal in frequency domain

In the plot we can see the spectrum of generated cosines and 0th frame. It is obvious that cosines were generate accurately, because pikes of frame0 and cosines are the same.

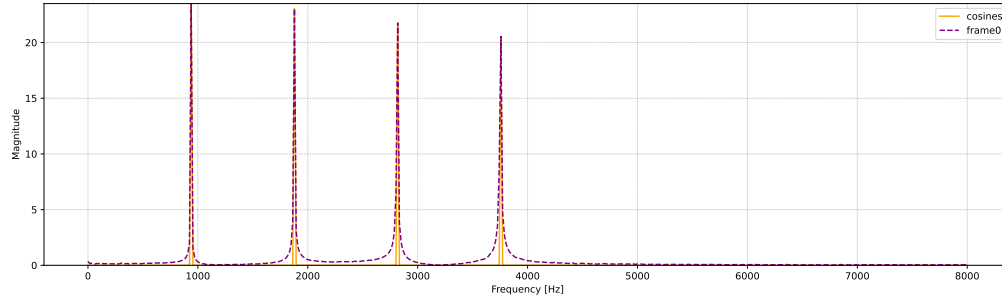


Figure 5: Cosines and `dft(frames[0])`.

2.6 Signal generation

The code below generates a signal of 4 harmonically related cosines with length of the original signal.

```

1  def generate_cosines_approx(indices, samples):
2      freq_res = Fs / N
3      PIPI = np.pi + np.pi
4      omegas = PIPI * indices*freq_res
5
6      time = np.arange(samples)/Fs
7
8      cosines_sum = np.zeros(time.shape)
9      for i in range(1, 5):
10         cosines_sum += np.cos(time * omegas[i-1])
11
12     return cosines_sum

```

Spectrogram and plot of generated cosine signal are shown below.

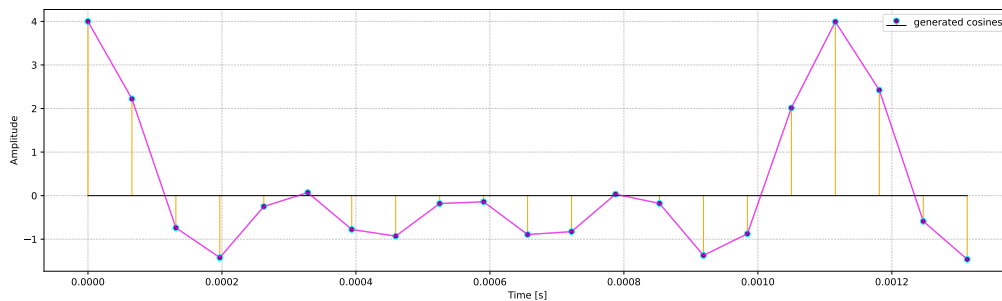


Figure 6: Cosines plot

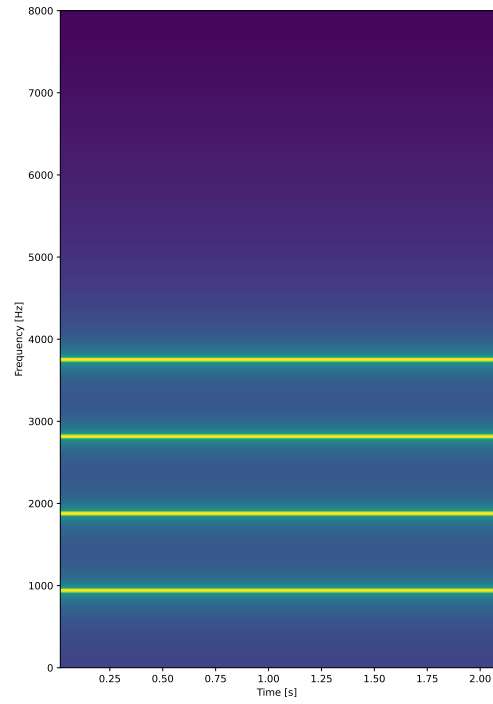


Figure 7: Cosines spectrogram

Comparison of generated noise and 1st frame of the signal in frequency domain is performed in the section 2.5. Signal itself was written to `audio/4cos.wav` file.

2.7 Filtering

Theory

My choice is a filter from 4 bandstops.

Given four frequencies I create four filters that kill the frequencies. I use `scipy.signal.butterord()` and `scipy.signal.butter()` functions to create a filter.

Then there is a need to combine these filters into one using ³ `np.convolve()` function.

I have to mention that I use my own order of the filter, which is 2, because the order returned by `scipy.signal.butterord()` (10) is too big, and hence the filter becomes unstable and kills not only the frequencies of cosines but the whole signal.

As it is specified in the assignment the maximum loss in the passband is 3dB and the minimum attenuation is 40dB.

Implementation

```
1 def ba_filter(freqs_to_filter, fs=16000, wpr=25, wsr=15): #b,a
2     def butter_bandstop_ba(freq, fs=fs, ord=2, wpr=wpr, wsr=wsr):
3         nuq = fs / 2 # Nuquist frequency
4         wp = np.array([freq-wpr, freq+wpr]) / nuq
5         ws = np.array([freq-wsr, freq+wsr]) / nuq
6
7         _, wn = scipy.signal.butterord(wp=wp, ws=ws,
8                                         gpass=3.0, gstop=40.0
9         )
10        return scipy.signal.butter(ord, Wn=wn,
11                                    btype='bandstop',
12                                    output='ba')
13
14    # create and convolve the filters.
15    b, a = butter_bandstop_ba(freqs_to_filter[0], fs=Fs)
16    for i in range(1, len(freqs_to_filter)):
17        bt, at = butter_bandstop_ba(freqs_to_filter[i], Fs)
18        b = np.convolve(b, bt)
19        a = np.convolve(a, at)
20
21    return b, a
```

³Convolution on Wikipedia

Impulse response

Because the filter is of type IIR (infinite response filter), I cannot plot the whole response.

Primarily because there is no function plotting infinitely long arrays, nor infinitely long arrays in python to show the whole response and moreover, there is no need to look at an infinite number of zeros.

That is why only 42 samples are used to represent a filter's response.

To get an Impulse Response itself, we need to pass a signal which has 1 at the first sample (index 0) and zeros on the other samples or just use function `scipy.signal.dimpulse()`. I'd chosen the second option.

```
1 b, a = ba_filter(frequencies, fs=Fs)
2
3 system = scipy.signal.dlti(b, a)
4 t, y = scipy.signal.dimpulse(system, n=42)
```

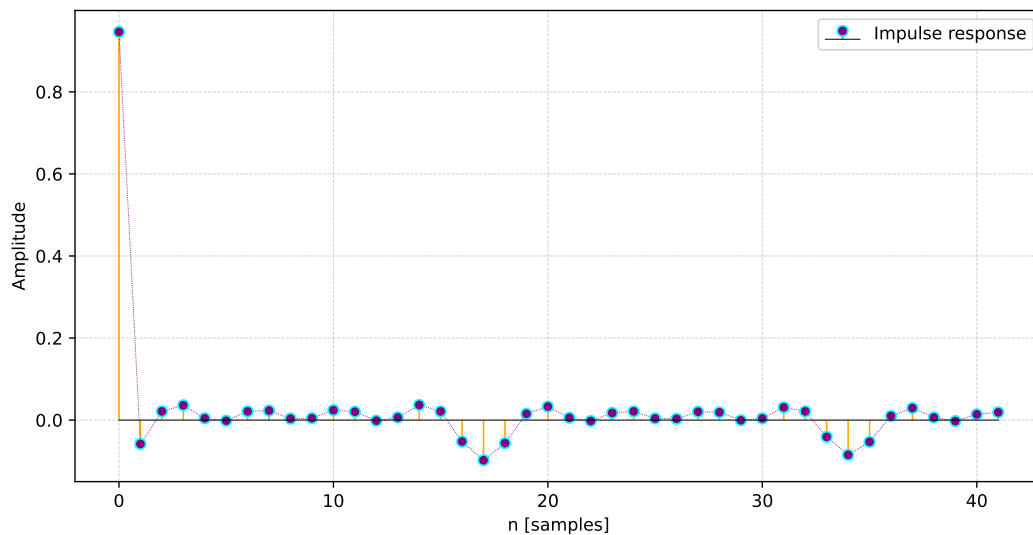


Figure 8: Impulse response

Zeros and poles, frequency characteristics

Transfer function $H(z)$ looks like.

$$H(z) = \frac{\sum_{k=0}^K b_k z^{-k}}{1 + \sum_{m=1}^M a_m z^{-m}}$$

1. Zeros are values z from D_H s.t $H(z) = 0$. In other words, z such that the numerator of the function equals 0.
2. Poles are values z from D_H s.t $H(z) = 0$. In other words, z such that the denominator of the function equals 0.

If we imagine a complex sweatshirts, then in zeros sweatshirts will pass the real-imaginary plane, so its value will be 0. On the other side, in poles sweatshirts will have infinitely big value.

All zeros and poles lie inside a unit circle, so the filter is stable.

```
1 z = np.roots(b)
2 p = np.roots(a)
3 w, h = scipy.signal.freqz(b, a)
```

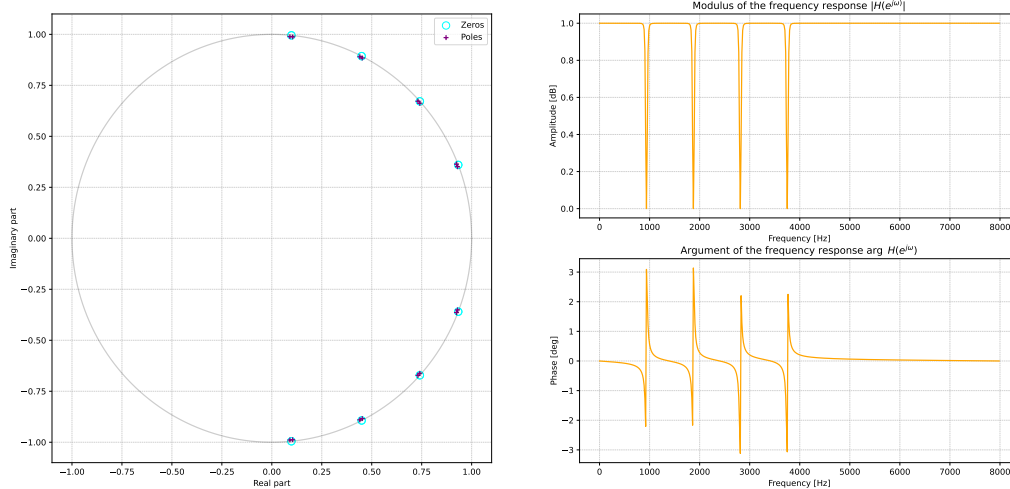


Figure 9: zeros, poles, frequency characteristic

2.8 Zero-Phase response

There is no thing such as a Zero-Phase response in the assignment, but when I was googling about it, I saw this fascinating plot in Matlab documentation⁴ function, so I decided to implement it by myself.

$$H(e^{j\omega}) = H(z)|_{z=e^{j\omega}}$$

, where H is a Transfer Function of the filter (see section 2.7).

The function is pretty simple and it looks like this:

```

1  def ba2zphr(b, a, N=1024):
2      def H(ejw, b, a):
3          K, M = len(b), len(a)
4          b_s = np.dot(b, ejw **(-1*np.linspace(0, K-1, K)))
5          a_s = np.dot(a, ejw **(-1*np.linspace(0, M-1, M)))
6          return b_s / a_s
7
8      ejws = np.exp(1j * np.linspace(0, 2*np.pi, N))
9      zphrs = np.zeros(N, dtype='complex128')
10     for i, ejw in enumerate(ejws):
11         zphrs[i] = H(ejw, b, a)
12     return zphrs, ejws

```

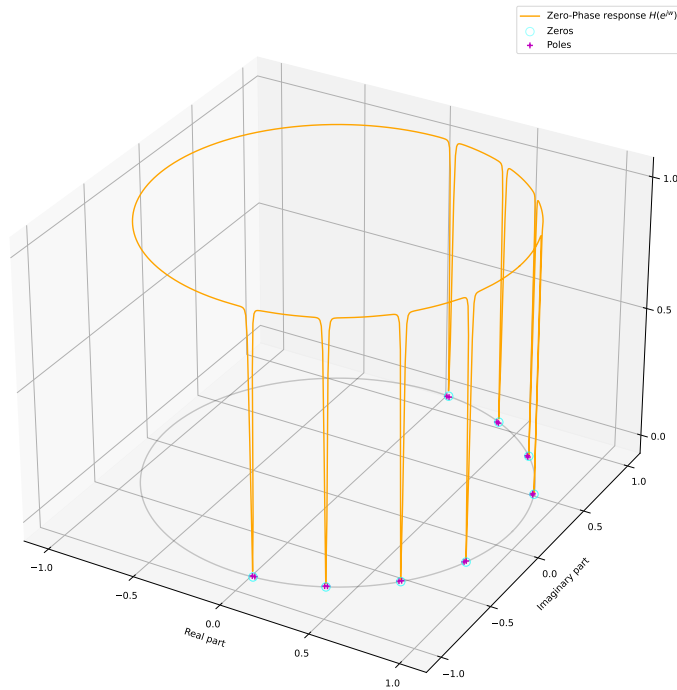


Figure 10: Zero-Phase response

⁴Matlab documentation

Filtering

Signal was filtered using `scipy.signal.lfilter` function. In the section 2.8, it is clearly seen that the filtered signal is thinner compared to a dirty one. Also, you can not see 4 horizontal lines on the spectrogram anymore, which means that they were killed by the filter.

However, there were 4 small cosines look like a beak in the beginning of the spectrogram. It is because my filter needs time to get ready and start killing frequencies. There is even a special name for this phenomenon - edge effect.

The solution of this problem is the following heuristic.

```

1  def filter_signal(signal, b, a):
2      y = signal
3      for i in range(2):
4          y = scipy.signal.lfilter(b, a, y)
5          # Here, we copy filtered values from the end of the signal
6          # to the beginning of the signal.
7      for i in range(850):
8          y[i] = y[len(y)-i-1]
9
10     y = (y-y.mean()) / absmax(y)
11
12     return y

```

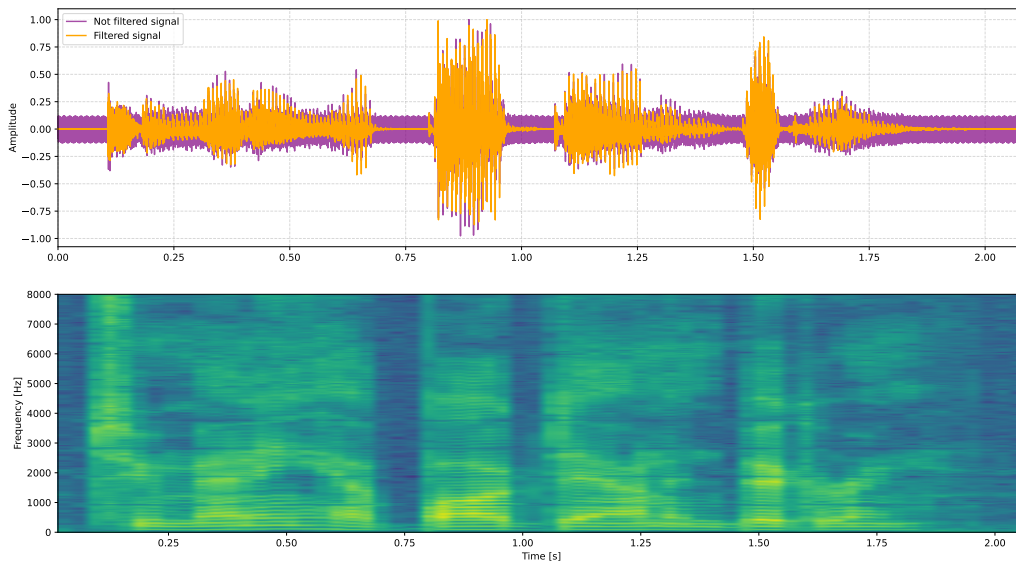


Figure 11: Two signals and a spectrum of the filtered one

3 Bonus assignment

3.1 Increasing frequency resolution

Introduction

The first bonus assignment is to find the frequencies of cosines with the resolution of 1Hz.

Due to the fact that DFT has only 1024 frequency bins and the sampling frequency is 16000Hz, one frequency bin's resolution $\frac{F_s}{N}$, which is 15.6Hz. That means that there are 15.625Hz in one bin and it is slightly (15.625 times) greater than one Hertz.

I've chosen the second given method consists of increasing resolution for determining the frequency of f_n , which uses DFT with a larger number of bins to achieve a frequency resolution of 1Hz.

Theory

First, there is a need to apply a usual ('usual' stands for 1024 frequency bins) DFT on the whole signal to determine the estimated frequency of the first cosine, which has been done in subsection 2.5.

The frequency of the first cosine is approximately 937.5 hertz, which is the sixtieth. bin with an error of 15.625 hertz. The sixtieth bin contains frequencies from the interval [937.5Hz, 953.125Hz].

In order to understand exactly what frequency the cosine is "sitting" at, I take the frequencies from 59th bin to 61th bin which are the frequencies in range 920Hz...950Hz, and analyze them using a DFT with has 16000 frequency bins.

The resulting analyzing matrix had the shape of (30, 16000), which is even smaller, than the original matrix for the DFT matrix with the shape of (512, 1024).

Due to the fact that the other cosines are just multiplies of the first one, we can find them by simply multiplying f_1 . However, to get more precise result, the function `peaks_mean()` was implemented.

The function works as follows:

$$f_1 = \sum_{i=1}^4 \frac{f_i}{i}$$

Also, it is good to compute the mean frequencies over all frames. However, you can see that my implementation calculates the frequencies over only frames with no speech signal to get clearer results.

Implementation

Function which returns one frequency 'peak' for the given index.

```

1  def get_precise_frequencies(index, freq_res, frame):
2      jomega = (1j * 2 * np.pi)
3      N = 16000
4      k_from = np.floor((index-1)*freq_res).astype(np.int64)
5      k_to = np.floor((index+1)*freq_res).astype(np.int64)
6      n_from = 0
7      n_to = N
8
9      n = np.arange(n_from, n_to)
10
11     k = np.linspace(k_from, k_to, num=k_to-k_from+1)
12     nk = (n * k[:, None])
13     freqmat = np.exp(-jomega * nk / N)
14
15     padded_vector = pad_with_0(frame, N)
16     precise_amplitudes = freqmat @ padded_vector
17     precise_magnitudes = np.abs(precise_amplitudes)
18
19     # offset in k_from
20     offset = (np.where(precise_magnitudes == precise_magnitudes.max()))[0][0]
21     z_amplitude = precise_amplitudes[offset]
22
23     precise_freq = k_from + offset
24     return precise_freq, z_amplitude

```

```

1  def main(frames):
2      N = 1024
3      freq_res = Fs / N # a frequency resolution.
4      precise_z = np.zeros(4, dtype='complex128')
5      frame = frames[0]
6
7      complex_acc = np.zeros(4, dtype=np.complex128)
8      freq_acc = np.zeros(4, dtype=np.float64)
9      _frqs = np.zeros(4, dtype=np.float64)
10     for j, frame in enumerate(get_novoice_frames(frames)):
11         for i in range(0, 4):
12             _frqs[i], pz = get_precise_frequencies(indices[i], freq_res, frame)
13             if (i + 1) % 4 == 0:
14                 freq_acc += _frqs
15
16     precise_frequencies = peaks_mean(freq_acc/len(get_novoice_frames(frames)))
17     return precise_frequencies

```

Precise frequencies f_i , $i \in 1 \dots 4$ are then [936.3 1872.6 2808.9 3745.2] Hz.

3.2 Determining amplitudes of cosines

Firstly, we need to get frames without voice, then compute four amplitudes using the following formula:

$$A_i = \frac{2}{N} |X[f_i]|, \quad i \in 1 \dots 4$$

Firstly, there is a need to find frames without a voice signal, because its frequencies can change our spectrum.

```

1  def determine_amplitudes(novoice_frames, N=1024):
2      freq_res = Fs / N
3      def per_frame_amplitudes(novoice_frame, N):
4          z_amplitudes = np.zeros(4, dtype='complex256')
5          for i in range(4):
6              _, zx = get_precise_frequencies(precise_indices[i], freq_res, novoice_frame)
7              z_amplitudes[i] = zx
8          magnitudes = (np.abs(z_amplitudes)*2.0) / N
9          return magnitudes
10
11     sum = np.zeros(4)
12     for frame in novoice_frames:
13         sum += per_frame_amplitudes(frame, N).ravel()
14
15     mean = sum / len(novoice_frames)
16     return mean

```

Precise amplitudes A_i , $i \in 1 \dots 4$ are: [0.04643105, 0.04643358, 0.04643304, 0.04645943].

3.3 Determining the phases of cosines

We can calculate phase shift of a cosine from the spectrum in the following way:

$$\phi = \tan^{-1}\left(\frac{\mathbb{I}(X[f_i])}{\mathbb{R}(X[f_i])}\right), \quad i \in 1 \dots 4$$

where f_i is precisely determined frequency of i -th cosine and X is discrete spectrum produced by DFT.

I use function `np.angle()` to determine angle. Also, there is a need to calculate a phase through multiple frames to get a mean value, which will be more precise. It can be done by adding $(-1)^r r \omega$ to the angle, where r is a number of the frame and used frames are ones without voice signal from the previous section subsection 3.2.

Unfortunately, when I was trying to compute precise phases on multiple frames, I get even worse results, so I decided to get imprecise values from only the first frame.

```

1  def determine_phases(frames, frequencies, indices):
2      PIPI = np.pi + np.pi
3      fs = 16000
4      freq_res = fs/1024 # resolution
5      omegas = ((frequencies * PIPI) / fs)
6
7      phases = np.zeros(4, dtype=np.float128)
8      dft_Xs = np.zeros(4, dtype='complex128') # dft coefficients - accumulator
9
10     frame = frames[0]
11
12     omega512 = omegas * 512
13     for i in range(4):
14         dft_Xs[i] = get_precise_frequencies(indices[i], freq_res, frame)[1]
15
16     angles = np.angle(dft_Xs)
17     phases += angles
18
19     precise_phases = (phases + np.pi) % (PIPI) - np.pi
20     return precise_phases

```

Precise (the best I had been able to get) phases ϕ_i , $i \in 1 \dots 4$ were computed by this function are [-2.24926133 -1.06322977 -2.773656 -2.22532775] rad.

3.4 Generating cosines and subtracting a noisy signal

We are given precise parameters of cosines. The noisy cosine signal Φ is generated in the following way:

$$\Phi = \sum_{i=1}^4 A_i \cos(\omega_i x + \phi_i)$$

Function generating sum of cosines is shown below.

```

1  def generate_4_cosines(amplitudes, frequencies, phis, samples, fs=16000):
2      time = np.linspace(0, samples/fs, num=samples)
3      cosines = np.array([time, time, time, time])
4      PIPI = np.pi * 2
5      omegas = frequencies * PIPI
6
7      for i in range(4):
8          cosines[i] = cosines[i] * omegas[i] #  $\omega_i x$ 
9          cosines[i] = cosines[i] + phis[i] #  $i x + \phi_i$ 
10         cosines[i] = np.cos(cosines[i]) #  $\cos(\omega_i x + \phi_i)$ 
11         cosines[i] = cosines[i] * amplitudes[i] #  $A_i \cos(\omega_i x + \phi_i)$ 
12
13     return time, cosines

```

We can compare generated and the original noisy signals, and see... That they differ! A tiny bit, but differ.

You can see on the graph, that some samples differ while the shape of generated signal repeats the original one. Also, Mean Square Error for the first 1024 samples is not zero ($1.3613880489349789e-05$).

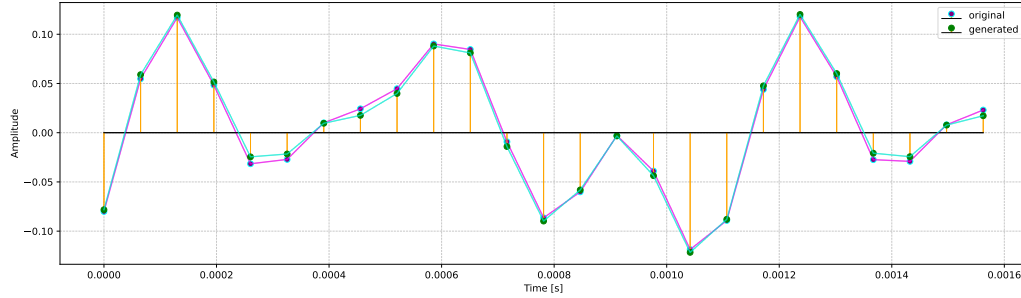


Figure 12: Generated and the original signal plotted together.

4 Conclusion

Basic assignment

The Discrete Fourier Transform was successful. First of all, I was aimed at speeding up this function, which, as it seems to me, I also managed to accomplish.

Then there were certain problems with the spectrogram, as my implementation was different from the results returned by functions from libraries like `scipy`'s (`scipy.signal.spectrogram()`) and `matplotlib`'s (`matplotlib.pyplot.specgram()`). After looking at the source code it turned out that these libraries use window functions.

The signal generated at the frequencies found sounded like noise in the original speech signal. However, since neither the amplitudes nor the phases of the cosines were known, this signal could not be subtracted from the original signal. To subtract noise, it was necessary to apply a filter (??) or take into account the parameters of the cosine signal (section 3).

The implementation of the filter was straightforward with some additional extensions (subsection 2.8) that I found very interesting.

The filtering itself has been performed successfully.

Bonus assignment

I had successfully found the amplitudes and phases of the noise.

Unfortunately, due to the impreciseness of computations, I was not able to find parameters of discrete cosines to fit original ones. However, it is obvious from the raft that generated cosines are very similar to the original ones. Even though the MCE was very small, it was significantly important not to have it at all for good filtering. I am sure that they differ because of imprecise determining of phases ϕ_{hi} of cosine signals.

The whole project

As a result, I can say that I am satisfied with my result because first of all, I have practiced a digital signal processing in python and applied the knowledge learned in the lectures in practice.

Many thanks and respect to prof. Jan "Honza" Černocký, ing. Jan Švec and Ing. Jan Brukner for this fascinating project and for the opportunity to practice in digital signal processing.

References

- [1] ČERNOCKÝ, J. *Zpracování řečových signálů – studijní opora*. 2006.
Available at: https://www.fit.vutbr.cz/study/courses/ZRE/public/opora/zre_opora.pdf.
- [2] WIKIPEDIA CONTRIBUTORS. *TIMIT — Wikipedia, The Free Encyclopedia*. 2021.
Available at: <https://en.wikipedia.org/w/index.php?title=TIMIT&oldid=1039861531>.