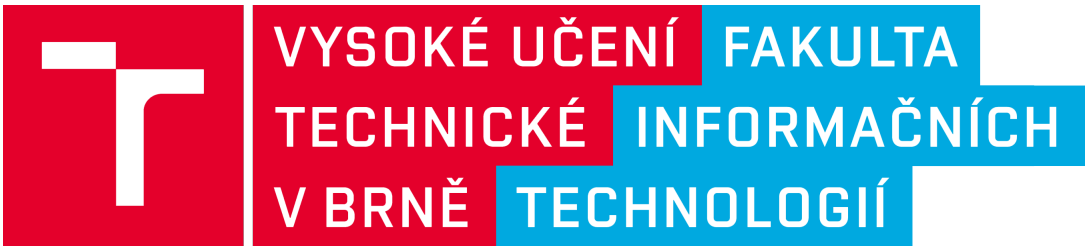


Signály a systémy

Protokol k projektu 2020

3. ledna 2021 Vojtěch Fiala (xfiala61)

1 Úvod

Tento protokol popisuje moji implementaci projektu do předmětu ISS. Všechny úkoly jsou řešeny v jazyce `Python`, zdrojové kódy lze najít v souboru `src/main.py`. Veškeré obrázky v tomto protokolu jsou vektorového typu a lze je tedy libovolně přiblížit beze ztráty kvality. Program je navržen tak, že musí být spouštěn výhradně z kořenového adresáře projektu (`python3.8 src/main.py`). Program je testován na serveru `merlin` na verzi Pythonu 3.8, která je pro spouštění preferovaná. Grafy se nevypisují sami, pro jejich vytvoření v kořenovém adresáři je nutné program spustit s přepínačem `-s`, tzn. `python3.8 src/main.py -s`. V kořenovém adresáři projektu se také nachází soubor `requirements.txt`. Ten slouží k případnému stažení použitých knihoven skrz program pip. Je nutno jej zavolat příkazem `pip install -r requirements.txt`.

2 Prvotní odhad

Pro přehlednost jsou jednotlivé úkoly vypracovány ve vlastních podsekcích.

2.1

Projekt jsem začal tím, že jsem vytvořil nahrávky "tónu"s rouškou a bez ní. Tabulka níže popisuje jejich délku a počet vzorků. Tato data jsem získal za pomoci programu `soxi`.

Název nahrávky	Délka v sekundách	Délka ve vzorcích
maskon_tone	03.54	56667
maskoff_tone	03.54	56667

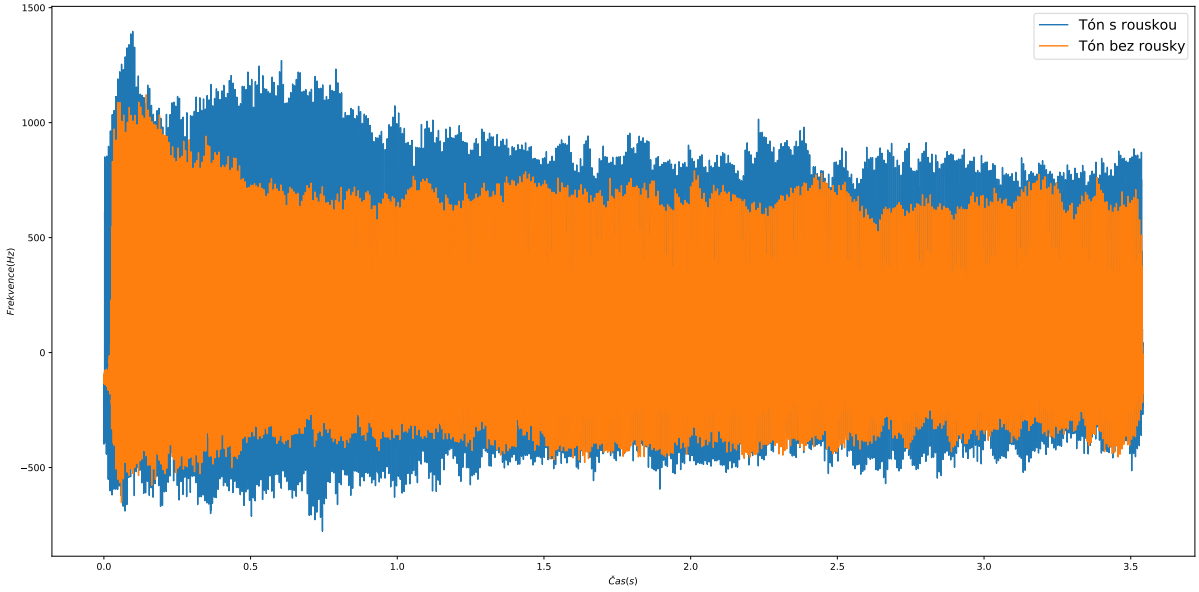
2.2

Následovalo nahrávání věty s nasazenou rouškou a bez ní. V tabulce níže je uvedena jejich délka v sekundách a počet vzorků, které nahrávka obsahuje. Data jsou získáná opět za pomoci programu `soxi`.

Název nahrávky	Délka v sekundách	Délka ve vzorcích
maskon_sentence	05.60	89536
maskoff_sentence	05.60	89536

2.3

Jednu vteřinu dlouhou část, která začíná v čase `0,1875s` a končí v čase `1,1875s`, jsem vybral na základě optického srovnání grafů obou signálů, které je zobrazeno níže.



Obrázek 1: Grafické znázornění nahraného tónu

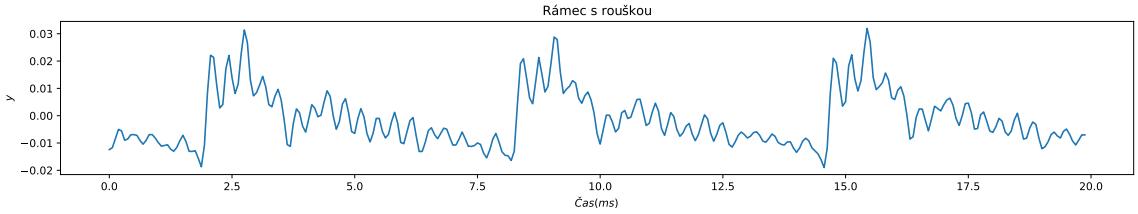
Tón jsem dále ustřednil, jeho normalizace pak nebyla potřeba, neboť je normalizován již díky způsobu čtení (Ve své implementaci nahrávky otevírám za pomoci knihovny `soundfile`, která normalizaci provádí sama).

Segment tónu je dále rozdělen na 99 rámců – poslední, neúplný 100. rámec, je zahozen. Vzorec pro výpočet velikosti rámce je následující:

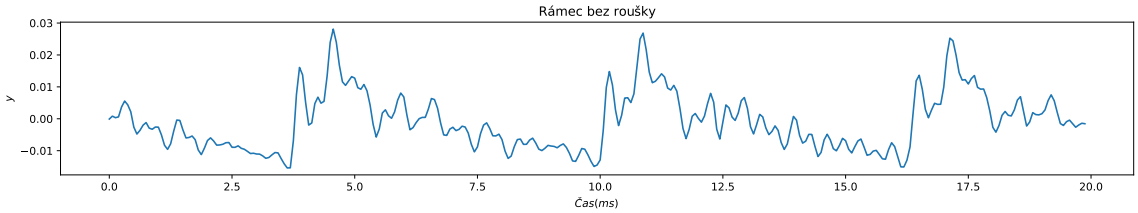
$$Velikost\_ramce = Vzorkovaci\_frekvence * Delka\_ramce$$

`Velikost_ramce` značí výslednou velikost 1 rámce ve vzorcích.
`Vzorkovaci_frekvence` značí počet vzorků v jedné vteřině signálu (16000).
`Delka_ramce` pak značí délku jednoho rámce v sekundách (0.02).

Níže je pak zobrazen graf dvou vybraných rámců.



Obrázek 2: Grafické znázornění rámce s rouškou



Obrázek 3: Grafické znázornění rámce bez roušky

2.4

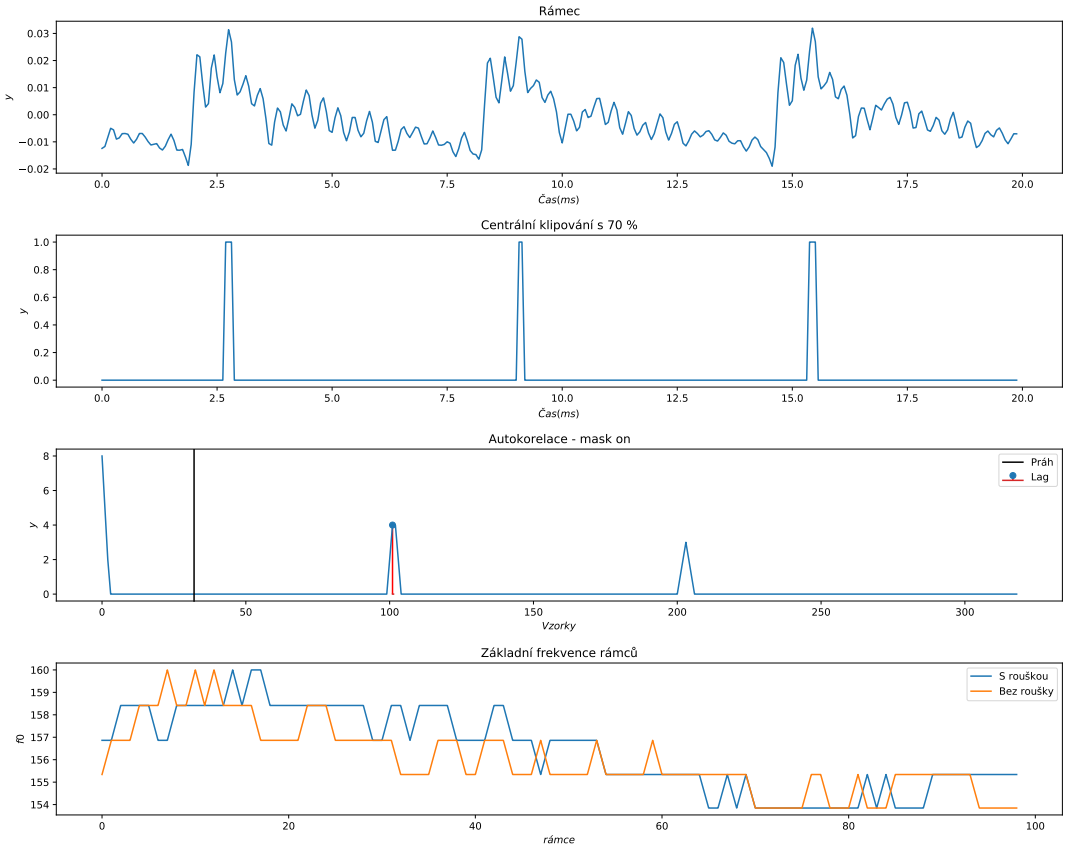
Dále je na jednotlivé rámce aplikována metoda centrálního klipování, autokorelace a je vypočítána lag. Jako práh jsem zvolil $500Hz$. Nakonec je určena základní frekvence f0.

V rámci tohoto podúkolu také používám funkci na odstranění "extrémů", které by mohli deformovat základní frekvence – funkce `remove_invalid_values`, která spočítá průměr všech frekvencí a odstraní hodnoty, které jsou o více než $80Hz$ vyšší či o $100Hz$ nižší než zmíněný průměr. Tuto funkce jsem implementoval kvůli původní nahrávce tónu, kde bylo až 40% všech hodnot extrémně zkreslených. Nahrávka tónu, kterou v rámci projektu nyní zpracovávám již těmito problémy netrpí a funkce tedy nic odstraňovat nemusí – přesto stále slouží jako "pojistka".

V následující tabulce jsou uvedeny střední hodnoty a rozptyly 1 vteřinu dlouhých segmentů obou nahrávek.

Název nahrávky	**Střední hodnota základní frekvence**	**Rozptyl základní frekvence**
maskon_tone	156.41907250368612	1.8110270384111935
maskoff_tone	155.99765942415974	1.5534419437371814

Vybraný rámec a na něm proveden clipping, autokorelace, zobrazení prahu a lagu a také základní frekvence všech rámců je zobrazena na grafech níže.



Obrázek 4: Grafy k podúkolu č. 4

Velikost změny f0, kterou by způsobilo posunutí lagu o +- 1, by se dala zmenšit například zvýšením vzorkovací frekvence (Například navýšení F_s na 32000 by vedlo k poloviční změně oproti současnému stavu, kdy je F_s 16000).

2.5

V podúkolu č. 5 bych chtěl zmínit, že pro výpočet DFT nepoužívám svoji implementovanou funkci z důvodu, že její průběh je přibližně **40x** pomalejší, než průběh stejné funkce z knihovny `numpy`. Moji implementaci funkce DFT provádí funkce `my_count_dft`. V rámci ní také rozšiřuji původní pole hodnotami '0', až na velikost 1024. Tuto funkci provádím nad každým rámcem.

Kód mé implementace funkce provádějící diskrétní Fourierovu transformaci je ukázán na další stránce. Zde bych ještě dodal, že v (mojí verzi) Pythonu správně nefungují hodnoty kolem Eulerovy Identity[1]. Tyto hodnoty proto napravuji ručně sérií podmínek.

[1] https://en.wikipedia.org/wiki/Euler%27s_identity

Stejným způsobem problém řeším i v rámci IDFT.

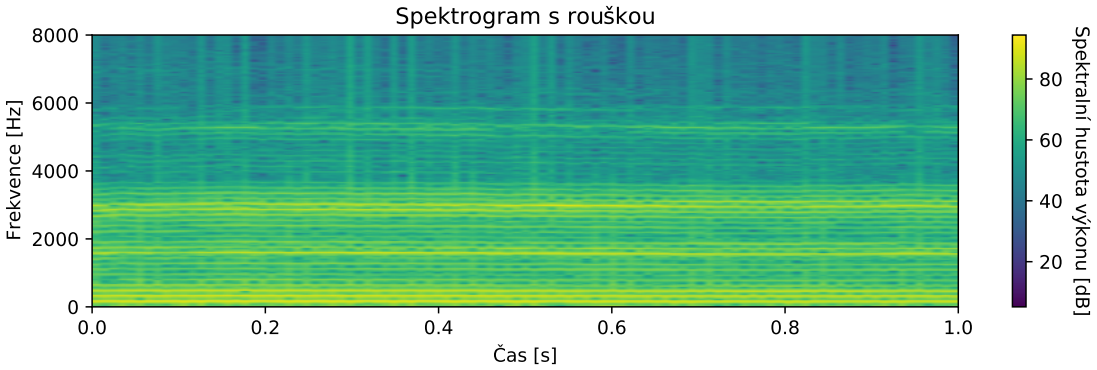
Algoritmus 1: DISKRÉTNÍ FOURIEROVA TRANSFORMACE

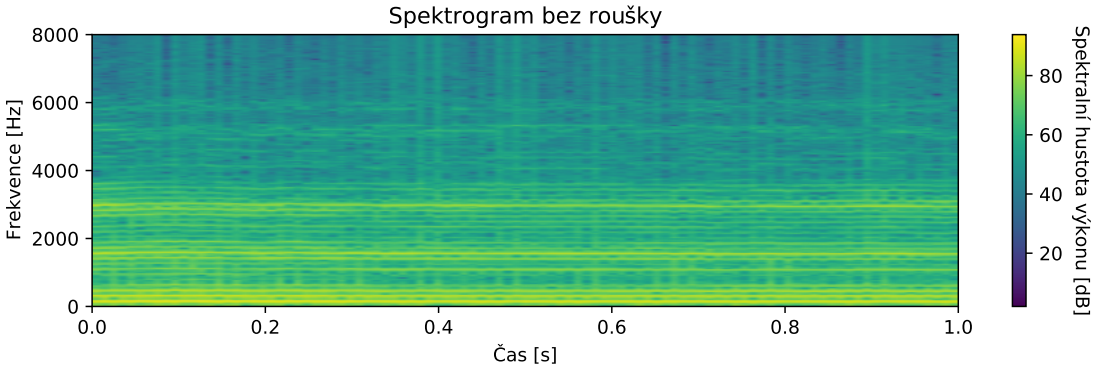
Input: $frames$
Output: $frame\_results$

```
1:  tmp = np.copy(frames)
2:  tmp = tmp * (2**15)
3:  results = []
4:  frame_results = [[]]
5:  length = len(tmp[0])
6:  length_max = len(tmp)
7:  tmp = np.pad(tmp, (0,1024-length), 'constant')
8:  length = len(tmp[0])
9:  For i in range(0, length_max):
10:     For x in range(0, length):
11:       tmp_result = 0
12:       For y in range(0, length):
13:         b = np.exp((-(1j) * np.pi * 2 * x * y) / length)
14:         if b == np.exp(-(1j) * np.pi): # Python nezvlada cisla okolo Eulerovy identity
15:           b = -1+0j
16:         elif b == np.exp(1j * np.pi):
17:           b = 1+0j
18:         elif b == (np.exp(1j * np.pi)) + 1:
19:           b = 0
20:         elif x == length/2:
21:           b = np.real(b) + 0j
22:         tmp_result += tmp[i][y] * b
23:       results.append(tmp_result)
24:     frame_results.append(results)
25:     results = []
26: frame_results.pop(0)
27: frame_results = np.array(frame_results)
28: return frame_results
```

Jednotlivé výsledky `frame_results` jsou dále před vykreslením zkráceny na v zadání zmíněných 512 prvků, které jsou pak ještě upraveny pomocí vzorce $P[k] = 10log10|X[k]|$.

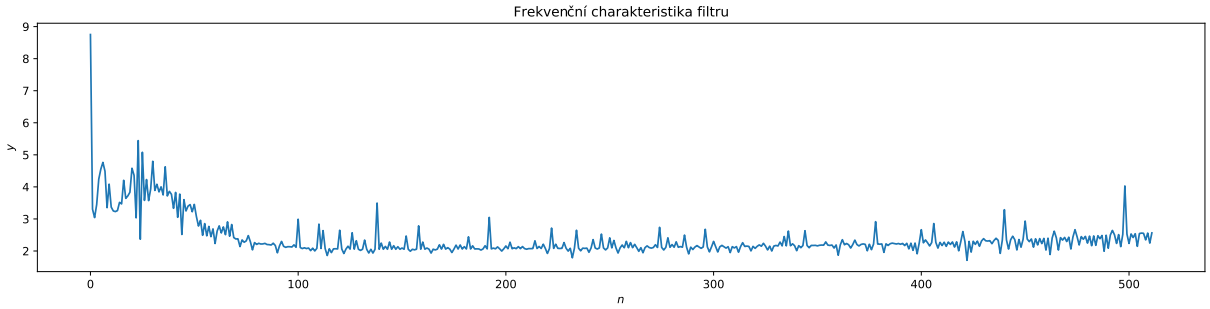
Spektrogramy jednotlivých tónů lze vidět zde:





2.6

Pro výpočet $H(e^{j\omega})$ používám vztah $H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})}$, kde B = Výsledek DFT pro mask_on; A = Výsledek DFT pro mask_off. Pro výpočet používám funkci z knihovny `numpy`, konkrétně funkci `freqz`. Jednotlivé hodnoty ze všech rámců poté převedu na jejich absolutní hodnotu, čímž se zbavím komplexní složky a poté je mezi sebou zprůměruji, čímž získám jednu frekvenční charakteristiku pro každý rámec. Její graf je poté k vidění níže.



Jelikož jsou všechny složky filtru kladné, řekl bych, že filtr bude sloužit jako zesilovač.

2.7

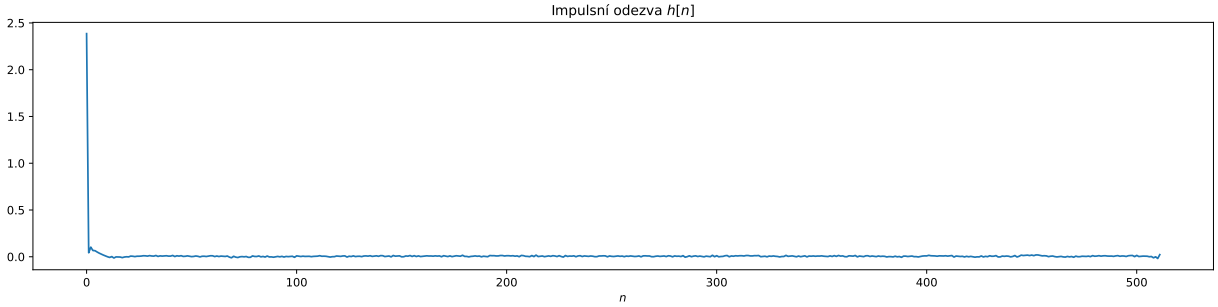
Pro výpočet IDFT používám namísto své funkce opět funkci `fft.ifft` z knihovny `numpy` z důvodu mnohonásobně vyšší efektivity. Moje implementace funkce IDFT je realizována ve funkci `my_count_idft`. Implementoval jsem ji na základě funkce `my_count_dft`, pouze s úpravami (Už probíhá ne pro každý rámec, ale jen pro jeden atp.). Její kód:

Algoritmus 2: INVERZNÍ DISKRÉTNÍ FOURIEROVA TRANSFORMACE

Input: $frame$
Output: $results$

```
tmp = np.copy(frame)
results = []
length_max = len(tmp)
For x in range(0, length_max):
    tmp_result = 0
    For y in range(0, length_max):
      b = np.exp(1j * ((np.pi * 2) / length_max) * x * y)
        if b == np.exp(-(1j) * np.pi): # Python nezvlada cisla okolo Eulerovy identity
          b = -1+0j
        elif b == np.exp(1j * np.pi):
          b = 1+0j
        elif b == (np.exp(1j * np.pi)) + 1:
          b = 0
        elif x == length/2:
          b = np.real(b) + 0j
      tmp_result += tmp[y] * b
    tmp_result = tmp_result * (1/length_max)
    results.append(tmp_result)
   results.append(tmp_result)
results = np.array(results)
return results
```

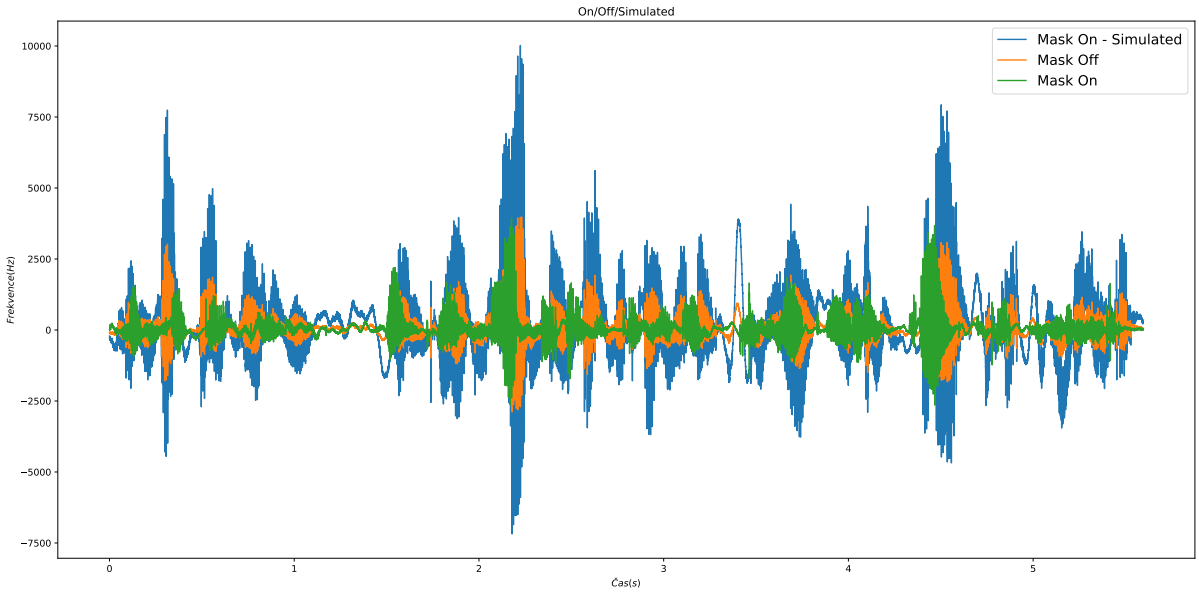
Výsledky této operace jsou dále zpracovány a slouží k vykreslení grafu impulsní odezvy roušky, který se nachází níže.



Obrázek 5: Graf impulsní odezvy roušky

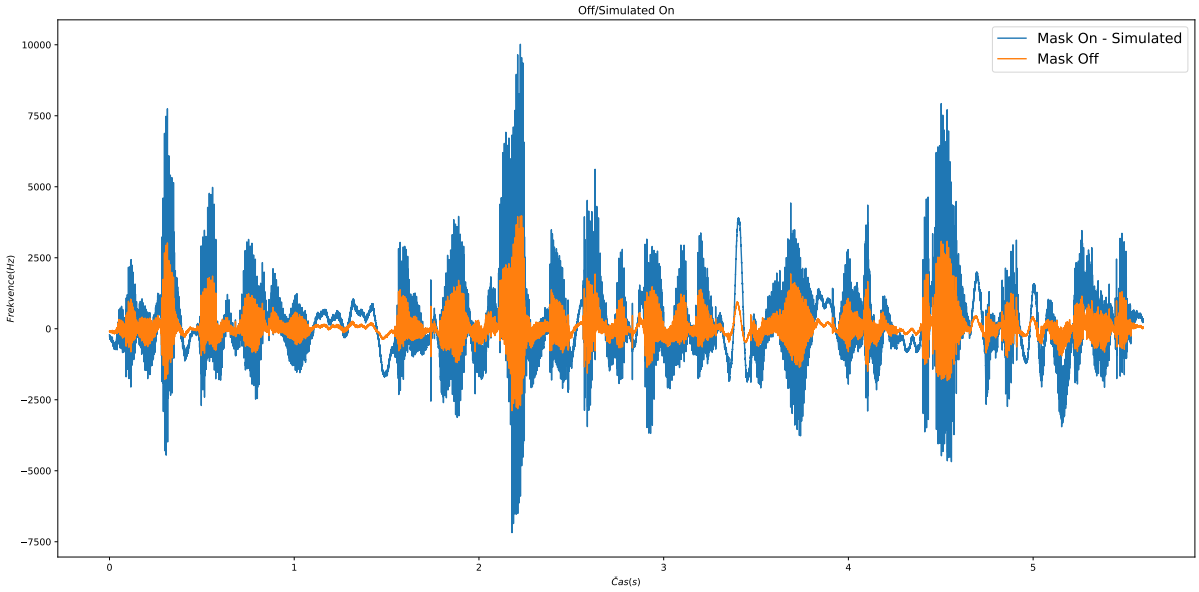
2.8

Na grafu níže je zobrazen průběh věty bez roušky, s rouškou a se simulovanou rouškou.



Obrázek 6: Simulovaná rouška

Níže je pro zpřehlednění zobrazen graf pouze věty bez roušky a se simulovanou rouškou.



Obrázek 7: Simulovaná rouška

Simulovaná rouška dosahuje výrazně vyšších frekvencí oproti oběma ostatním signálům, což je přesný opak, než bych od efektu roušky čekal. Vyšší frekvence jsou zvýšeny, nižší jsou sníženy. Tvarově si jsou signály podobné, nejyšší podobnost je u frekvencí blíže k nule. Nejvíce se naopak liší u vysokých a nízkých hodnot.

2.9 Závěr

Řešení nefunguje úplně dle očekávání. Původní nahrávka je po aplikaci filtru znatelně zesílena a jak lze vidět na grafu 7, tvarově je sice podobná, nicméně hodnotami je výrazně odlišná. Hodnoty jsou mnohonásobně vyšší (U nižších frekvencí zase nižší), což je přesný opak, než bych od filtru roušky čekal. Problém může být například v nevhodně zvoleném tónu, v příliš tenké roušce, či přímo v implementaci. Jelikož je toto moje první zkušenost se zpracováváním signálu, chyba v implementaci se zdá být nejpravděpodobnější. Mnohokrát jsem si při vypracovávání vyzkoušel, že i sebemenší změna ve filtru může vést k velmi odlišným výsledkům. V projektu jsem také neaplikoval korigující bonusové funkce, což může být dalším z faktorů.