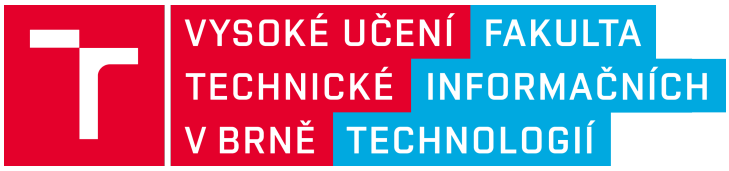


ISS – Protokol s řešením projektu

Dominik Vágner,
xvagne10

28. prosince 2021

Obsah

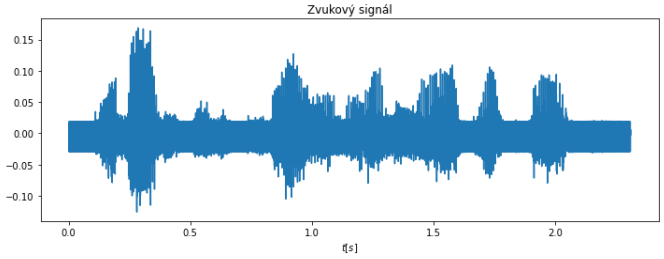
1 Základy

Pomocí `Python` knihovny `soundfile` (importovanou jako `sf`) si načteme náš zadaný osobní signál. Po jeho načtení si vytvoříme pole stejně velké jako původní signál naplněné hodnotami odpovídajícím indexům pole a celé ho vydělíme vzorkovací frekvencí naší nahrávky a získáme tak časovou osu, která se nám bude hodit v budoucích krocích. Z poslední hodnoty časové osy jsme schopni zjistit délku signálu. A pomocí atributu `.size` polí v `NumPy` získáme počet vzorků nahrávky. Následně pomocí knihovny `matplotlib` vykreslíme graf signálu.

Délka signálu je tedy **2,310375 sekund**.
Počet vzorků je **36967**.

```
In [4]: # Nacteni zvukoveho souboru, s - pole s hodnoty signalu, fs - vzorkovaci frekvence
        s, fs = sf.read('xvagne10.wav')
        # Vytvoreni pole se setejnou delkou jako signal, hodnota odpovida casu vyskytu kazdeho prvku.
        t = np.arange(s.size) / fs
        # Delka v sekundach
        print(t[-1])
        # Pocet vzorku
        print(s.size)

        2.310375
        36967
```

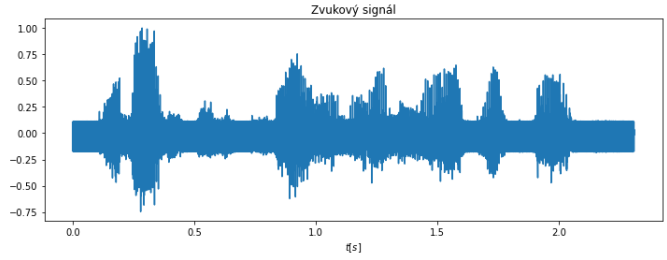
Obrázek 1: Ukázka kódu pro načtení zvukového signálu.



Obrázek 2: Graf původního signálu.

2 Předzpracování a rámce

Ustřednění a normalizování signálu s pomocí knihovny `NumPy` (importovanou jako `np`) je velice jednoduché. Stačí nám pouze tři funkce pro práci s poli a to získaní střední hodnoty, převedení do absolutní hodnoty a získání maximální hodnoty z pole. Získanými hodnotami vydělíme nebo je odečteme od původního signálu.



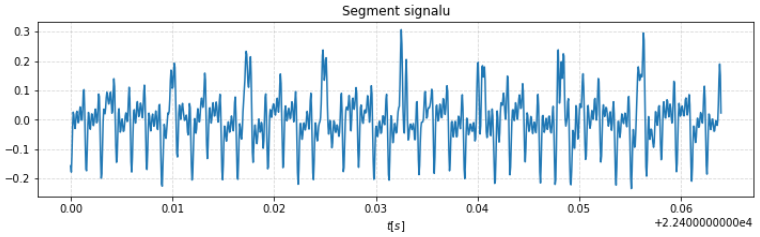
Obrázek 3: Graf ustředněného a normalizovaného signálu.

Pro rozdělení signálu na rámce si nejdříve vytvoříme matici nul (dvourozměrné pole) o požadované velikosti a tu poté naplníme daty ze signálu, který projdeme cyklem. Velikost matice získáme pomocí velikosti posunu rámců (tedy jejich překrytí). Poslední rámec doplníme nulami protože počet vzorků není dělitelný posunem beze zbytku. Po naplnění matice ji transponujeme aby jednotlivé rámce byly jako sloupce v matici. Můžeme si také vypsat atribut `.shape`, z kterého lze zjistit počet rámců našeho signálu (73).

```
In [9]: out_matrix = np.zeros((int(np.ceil(s.size / shift)), N))

In [10]: for i in range(int(np.floor(s.size / shift))):
             pos = i * shift
             if s[pos:pos + N].shape[0] < N:
                 out_matrix[i][0:s[pos:pos + N].shape[0]] = s[pos:pos + N]
             else:
                 out_matrix[i] = s[pos:pos + N]

         out_matrix = np.transpose(out_matrix)
         print(out_matrix.shape)

         (1024, 73)
```

Obrázek 4: Ukázka kódu pro rozdělení signálu na rámce.

Jako znělý rámec jsem si vybral šestnáctý rámec. Tento rámec je podle mě periodický a také po provedení DFT jsou z něj dobře čitelné rušivé frekvence jak uvidíme v následujících krocích.



Obrázek 5: Graf vybraného "pěkného" rámce.

3 Diskrétní Fourierova transformace

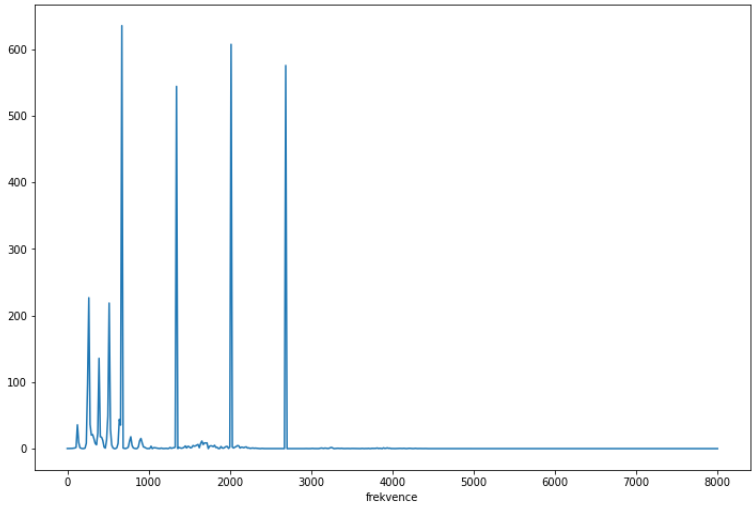
Implementaci mojí funkce pro výpočet diskrétní Fourierovy transformace jsem se snažil udělat s co nejméně cykly za pomocí násobení vektorů pomocí funkce `np.dot()`. Cyklus byl použit jen jeden pro procházení jednotlivých sloupců vstupní matice. Funkce přesně kopíruje rovnici (1) pro DFT.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-j2\pi kn/N} \tag{1}$$

```
In [3]: def mine_dft(matrix):
            N = 1024
            n = np.arange(N)
            k = n.reshape((N, 1))
            e = np.exp(-2j * np.pi * k * n / N)

            if len(matrix.shape) == 1:
                return np.dot(e, matrix)

            output = np.zeros(matrix.shape, dtype=np.cdouble)
            for i in range(matrix.shape[1]):
                output[:, i] = np.dot(e, matrix[:,i])

            return output
```

Obrázek 6: Ukázka kódu vlastní implementace DFT.

Po spuštění funkce na námi vybraném rámci, zobrazíme modul výsledku pro frekvence od nuly do půlky vzorkovací frekvence v grafu.



Obrázek 7: Graf modulu DFT na vybraném rámci.

Kontrolu správnosti mojí implementace můžu provést pomocí funkce `np.allclose()`, která rozhodne o tom jestli jsou dvě `NumPy` podobné podle určené tolerance. Porovnání provedeme proti knihovní implementaci FFT (`np.fft.fft()`).

```
In [11]: seg = out_matrix[:, 16]
         seg_spec = mine_dft(seg)
         seg_spec_ref = np.fft.fft(seg)

         'Everything OK' if np.allclose(seg_spec, seg_spec_ref) else 'Big oof'
Out[11]: 'Everything OK'
```

Obrázek 8: Ukázka kódu a výsledku porovnání mojí s referenční funkcí pro výpočet DFT.

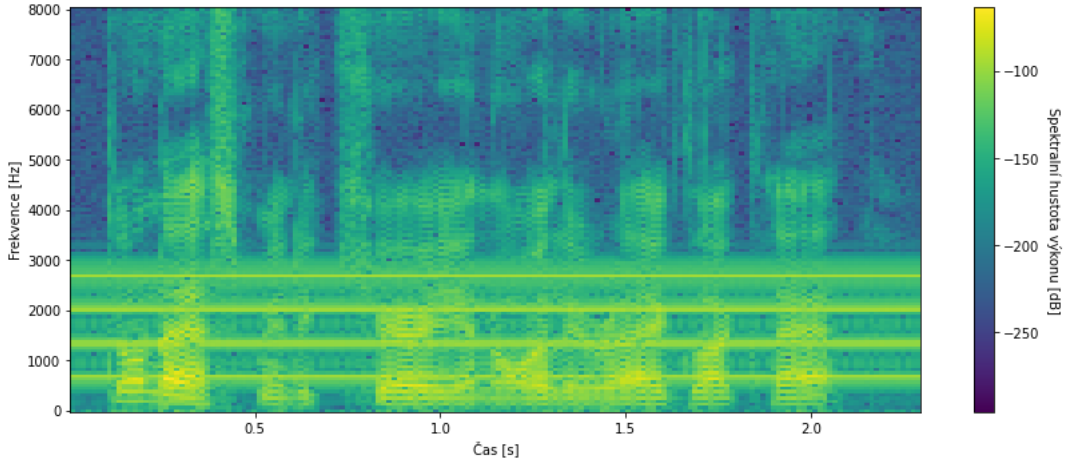
4 Spektrogram

Pro spektrogramu jsem použil funkci `spectrogram` z knihovny `SciPy`. Tato funkce automaticky zobrazí frekvence pouze do poloviny vzorkovací frekvence, takže stačí funkcí vygenerovat spektrogram a upravit ho podle zadaného vzorce $P[k] = 10 \log_{10} |X[k]|^2$.

```
In [14]: # Spectogram
         f, t, sgr = spectrogram(s, fs)
         sgr_log = 10 * np.log10(np.abs(sgr)**2)
```

Obrázek 9: Ukázka kódu pro vygenerování a úpravu spekrtrogramu.

Následně jen správně vykreslíme graf pomocí `matplotlib`.



Obrázek 10: Spektrogram pro celý signál.

5 Určení rušivých frekvencí

Z grafu modulů DFT (obrázek č. 7) můžeme vidět čtyři rušivé komponenty. Hodnoty jejich modulů jsou ty čtyři nejvyšší, proto můžeme seřadit pole s výsledky a jednoduše vybrat ty které potřebujeme. Poté také ověříme jestli jsou harmonicky vztažené.

```
In [16]: sorted_PSD = np.argsort(PSD) / 1024 * fs
         print(sorted_PSD[-4:])

         [1343.75  2687.5   2015.625  671.875]

In [17]: f_1 = np.argmax(PSD) / 1024 * fs
         f_2 = f_1 * 2
         f_3 = f_1 * 3
         f_4 = f_1 * 4

         print('f_1 OK') if f_1 == sorted_PSD[-1] else 'f_1 oof'
         print('f_2 OK') if f_2 == sorted_PSD[-4] else 'f_2 oof'
         print('f_3 OK') if f_3 == sorted_PSD[-2] else 'f_3 oof'
         print('f_4 OK') if f_4 == sorted_PSD[-3] else 'f_4 oof'

         f_1 OK
         f_2 OK
         f_3 OK
         f_4 OK
```

Obrázek 11: Ukázka kódu pro určení rušivých frekvencí a ověření jejich harmonické vztaženosti.

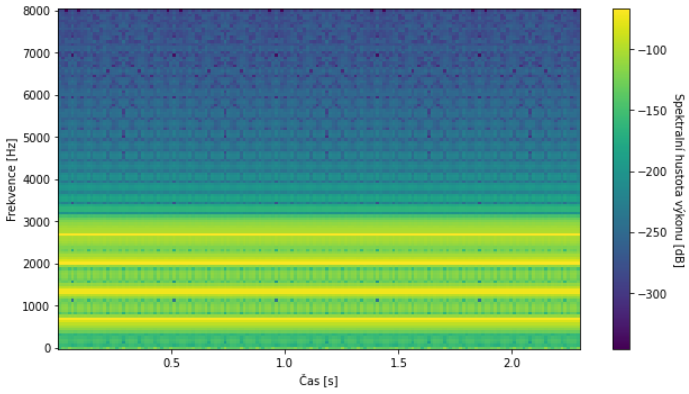
6 Generování signálu

Vzorkovací frekvence generovaného signálu bude stejná jako u původního signálu. Uděláme si pole s hodnotami od nuly do hodnoty jako délka originálního signálu a velikost kroku bude jedna děleno vzorkovací frekvence. Do daného pole pak přičítáme vygenerované cosinusovky (pomocí `np.cos()`) na rušivých frekvencích. Signál potom ustředníme, normalizujeme a uložíme do `.wav` souboru pomocí knihovny `soundfile`.

```
In [18]: # Generovani cosinus vln
         # Vzorkovaci frekvence
         gen_fs = 16000
         # Interval který vzorkovat
         gen_ts = 1.0 / gen_fs
         gen_t = np.arange(0, 2.310375, gen_ts)

         output = np.cos(2 * np.pi * f_1 * gen_t)
         output += np.cos(2 * np.pi * f_2 * gen_t)
         output += np.cos(2 * np.pi * f_3 * gen_t)
         output += np.cos(2 * np.pi * f_4 * gen_t)

         mean = np.mean(output)
         max_abs = np.amax(np.abs(output))
         # Ustredneni
         output = output - mean
         # Normalizovani, od -1 do 1
         output = output / max_abs

In [19]: sf.write('4cos.wav', output, gen_fs)
```

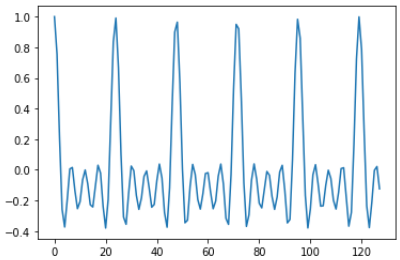
Obrázek 12: Ukázka kódu na vygenerování signál se směsí 4 cosinusovek.

Obdobně jako pro původní signál vypočítáme a zobrazíme spektrogram.



Obrázek 13: Spektrogram pro vygenerovaný signál.

Pro kontrolu se také můžeme podívat na prvních pár vzorků signálu, jestli se doopravdy skládá pouze z cosinusovek.



Obrázek 14: Prvních 128 vzorků vygenerovaného signálu.

7 Čistící filtr

Způsob návrhu čistících filtrů jsem zvolil přes čtyři pásmové zádrže se závěrnými pásmy okolo rušivých frekvencí. Pásmové zádrže jsou implementovány pomocí funkcí `buttord()` a `butter()`.

Šířku závěrného pásma jsem nastavil třeba 30 Hz a šířku přechodů do propustného 50 Hz. Maximální ztrátu v propustném pásmu (ripple) dovolíme do 3 dB a jako minimální útlum v závěrném pásmu budeme požadovat 40 dB. Tyto údaje společně se správnou vzorkovací frekvencí předáme jako parametry funkci `buttord()` a zpět dostaneme nejnižší řád filtru, který splňuje specifikace a pole s hodnotami mezi kterými se má výrazněji filtrovat. Poté zavoláme funkci `butter()`, které v parametrech pošleme výsledky `buttord()`, typ pásmové zádrže a vzorkovací frekvenci. Jako výsledek dostaneme pole koeficientů daného filtru. Toto zopakujeme čtyřikrát pro všechny rušivé frekvence.

Pro zobrazení impulsní odezvy (pro ukázku nám stačí třeba 32 vzorků) si vygenerujeme jednotkový impuls a ten následně vyfiltrujeme funkcí `lfilter()`, které jako parametry předáme koeficienty daného filtru.

```
In [26]: N, Wn = buttord([f_4 - 65, f_4 + 65], [f_4 - 15, f_4 + 15], 3, 40, 16000)
         b, a = butter(N, Wn, 'bandstop', fs=16000)
```

Obrázek 15: Ukázka kódu pro výpočet koeficientů čistících filtrů.

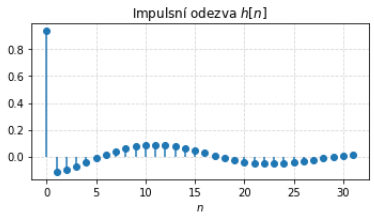
```
# impulsni odezva
N_imp = 32
imp = [1, *np.zeros(N_imp-1)]
h = lfilter(b, a, imp)
```

Obrázek 16: Ukázka kódu pro vytvoření a vyfiltrování jednotkového impulsu.

Koeficienty 1. filtru:

Čitatelé: [0.9381124, −7.24526501, 24.73628843, −48.7463241, 60.63439804, −48.7463241, 24.73628843, −7.24526501, 0.9381124]

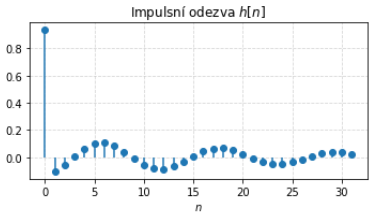
Jmenovatelé: [1.0, −7.59990727, 25.53356389, −49.51733289, 60.61629453, −47.9605276, 23.95328641, −6.90541044, 0.88005487]



Obrázek 17: Impulsní odezva prvního filtru.

Koeficienty 2. filtru:

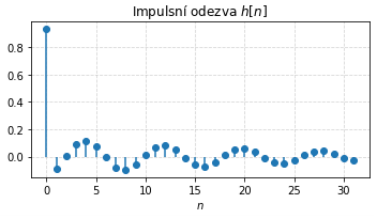
Čitatelé: [0.93686738, −6.47533738, 20.53079245, −38.75953437, 47.53955774
−38.75953437, 20.53079245, −6.47533738, 0.93686738]
Jmenovatelé: [1.0, −6.79902591, 21.2061643, −39.38441724, 47.52367684,
−38.12087996, 19.86731577, −6.1654204, 0.87772049]



Obrázek 18: Impulsní odezva druhého filtru.

Koeficienty 3. filtru:

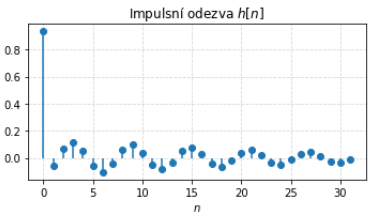
Čitatelé: [0.93642685, −5.26428022, 14.84346944, −26.19082836, 31.4674619,
−26.19082836, 14.84346944, −5.26428022, 0.93642685]
Jmenovatelé: [1.0, −5.5293702, 15.33491166, −26.61510171, 31.45544197,
−25.755197, 14.36000559, −5.01054825, 0.87689525]



Obrázek 19: Impulsní odezva třetího filtru.

Koeficienty 4. filtru:

Čitatelé: [0.93620146, −3.69093316, 9.20155909, −14.65830118, 17.41419544,
−14.65830118, 9.20155909, −3.69093316, 0.93620146]
Jmenovatelé: [1.0, −3.87749132, 9.50678676, −14.8957809, 17.40617497,
−14.41279956, 8.90028163, −3.51239691, 0.87647318]



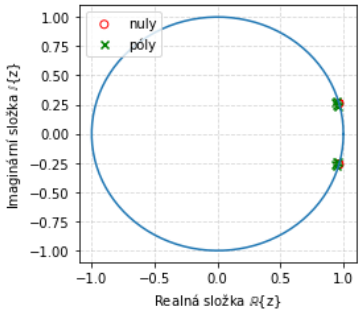
Obrázek 20: Impulsní odezva čtvrtého filtru.

8 Nulové body a póly

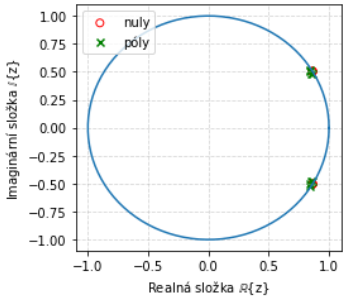
Nulové body a póly jdou lehce získat z koeficientů navrženého filtru pomocí funkce `tf2zpk()` z knihovny `SciPy`. Poté získané body a póly pomocí funkce `scatter()` z knihovny `matplotlib` promítneme do jednotkové kružnice. Musíme také zvlášť předat jednotlivě jejich reálné a imaginární části.

```
# nuly, poly
z, p, k = tf2zpk(b, a)
# jednotkova kruznice
ang = np.linspace(0, 2*np.pi,100)
plt.plot(np.cos(ang), np.sin(ang))
# nuly, poly
plt.scatter(np.real(z), np.imag(z), marker='o', facecolors='none', edgecolors='r', label='nuly')
plt.scatter(np.real(p), np.imag(p), marker='x', color='g', label='póly')
```

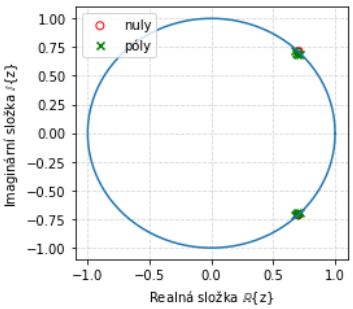
Obrázek 21: Ukázka kódu pro vygenerování a zobrazení nulových bodů a pólů.



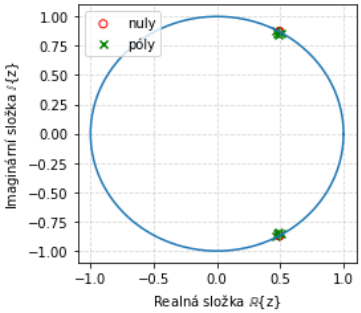
(a) První filtr



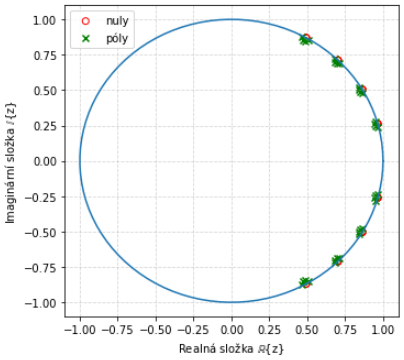
(b) Druhý filtr



(c) Třetí filtr



(d) Čtvrtý filtr



(e) Všechny

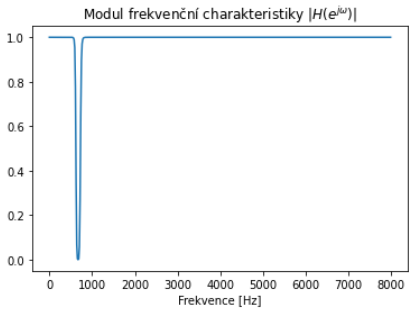
Obrázek 22: Zobrazené nulové body a póly pro jednotlivé filtry a všechny najednou.

9 Frekvenční charakteristika

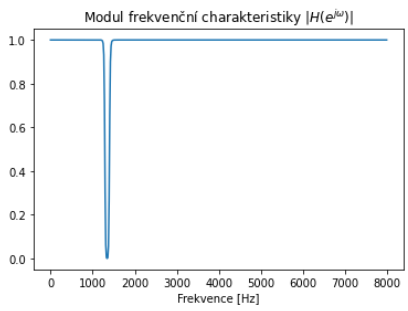
Frekvenční charakteristiky se dají získat obdobně jako nulové body a póly. Stačí použít funkci `freqz()` z knihovny `SciPy` a následně vykreslit pomocí `matplotlib`. Jako parametry funkci předáme koeficienty filtru a vzorkovací frekvenci pro zaručení správných jednotek. Poté na ose „x“ zobrazíme první návratovou hodnotu a na ose „y“ druhou návratovou hodnotu v absolutní hodnotě.

```
w, h = freqz(b, a, fs=16000)
plt.plot(w, np.abs(h))
```

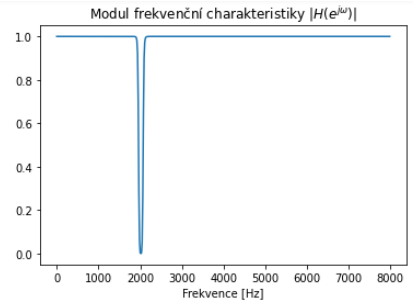
Obrázek 23: Ukázka kódu pro získání frekvenční charakteristiky.



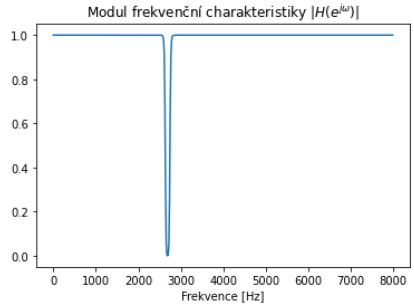
(a) První filtr



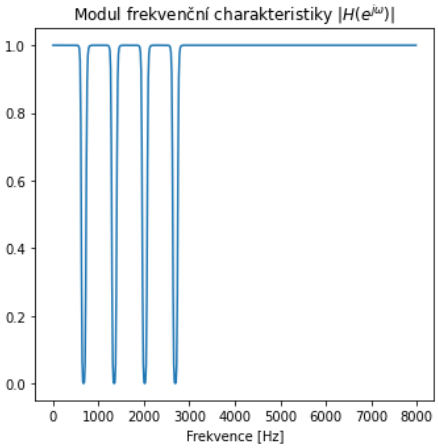
(b) Druhý filtr



(c) Třetí filtr



(d) Čtvrtý filtr

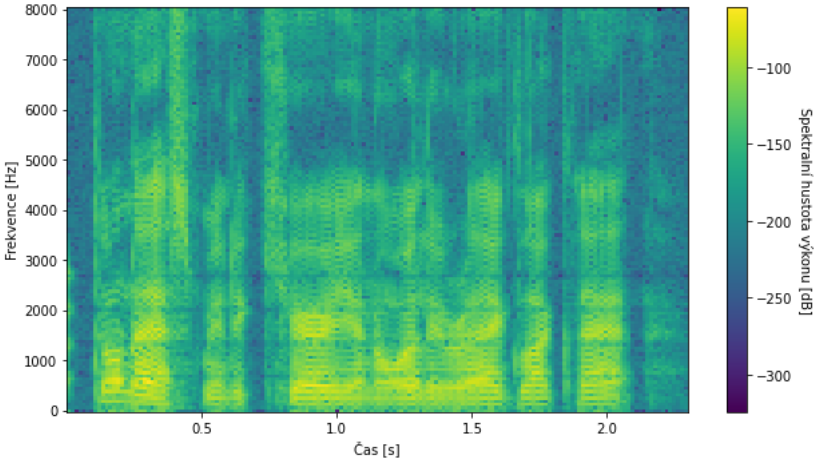


(e) Všechny

Obrázek 24: Zobrazené moduly frekvenční charakteristiky pro jednotlivé filtry a všechny najednou.

10 Filtrace

Filtraci signálu provedeme pomocí funkce `lfilter()`, která slouží k filtrování jednodimenzionálních dat pomocí FIR nebo IIR filtrů. Původní signál tedy vyfiltrujeme a normalizujeme od -1 do +1. Poté použijeme `soundfile` knihovnu k uložení výsledku do souboru `clean_bandstop.wav`. Po poslechu vyfiltrovaného zvuku není slyšet žádné pískaní, ale pořád zůstal původní hlas. Můžeme se také podívat na spektrogram výsledného signálu a neměli by být vidět „žluté čáry“ na rušivých frekvencích.



Obrázek 25: Spektrogram pro vyčištěný signál.

Literatura

[1] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[2] Kong, Q., Siauw, T., and Bayen, A. *Python Programming and Numerical Methods: A Guide for Engineers and Scientists*, paperback ed. Academic Press, 12 2020.

[3] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272.