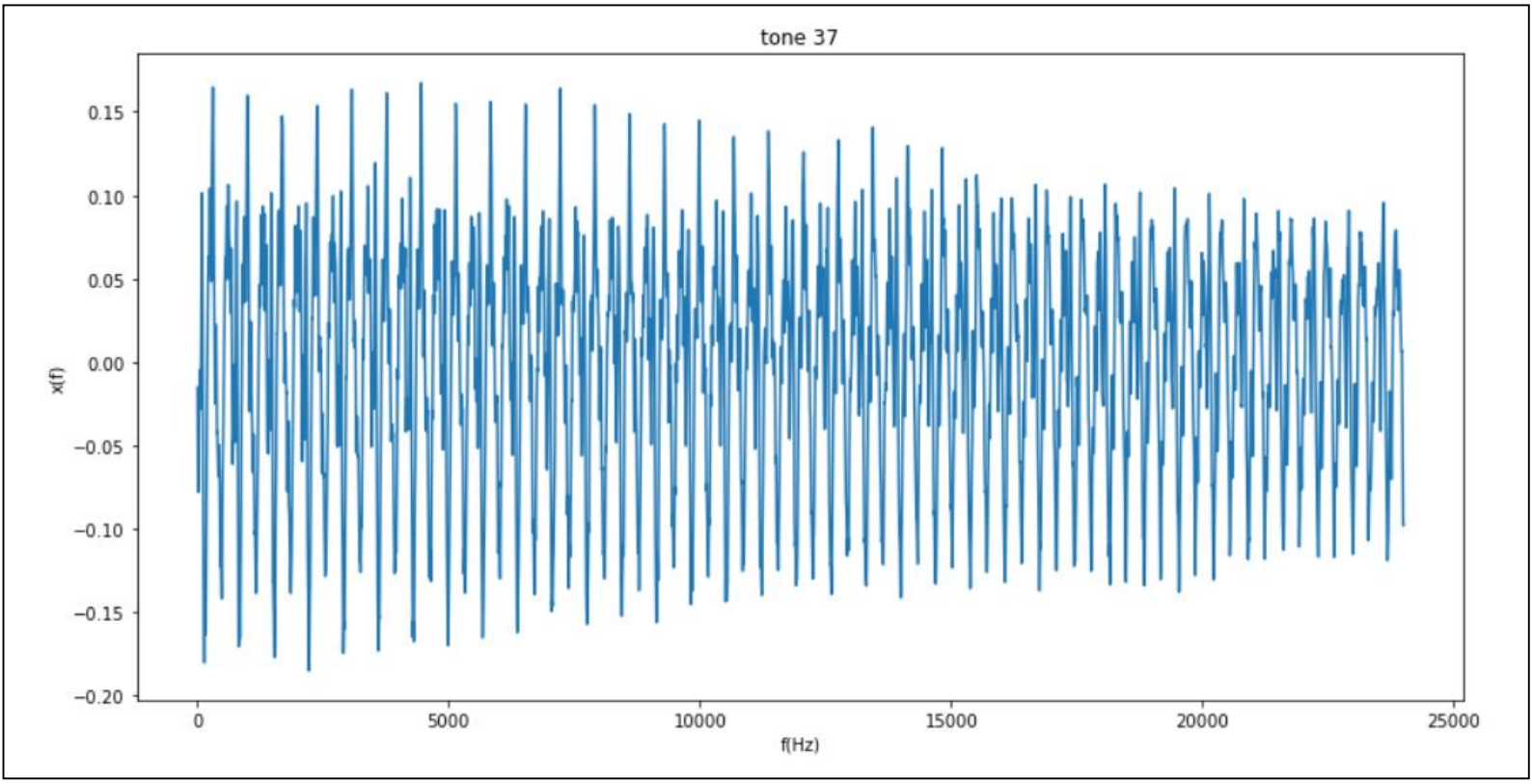
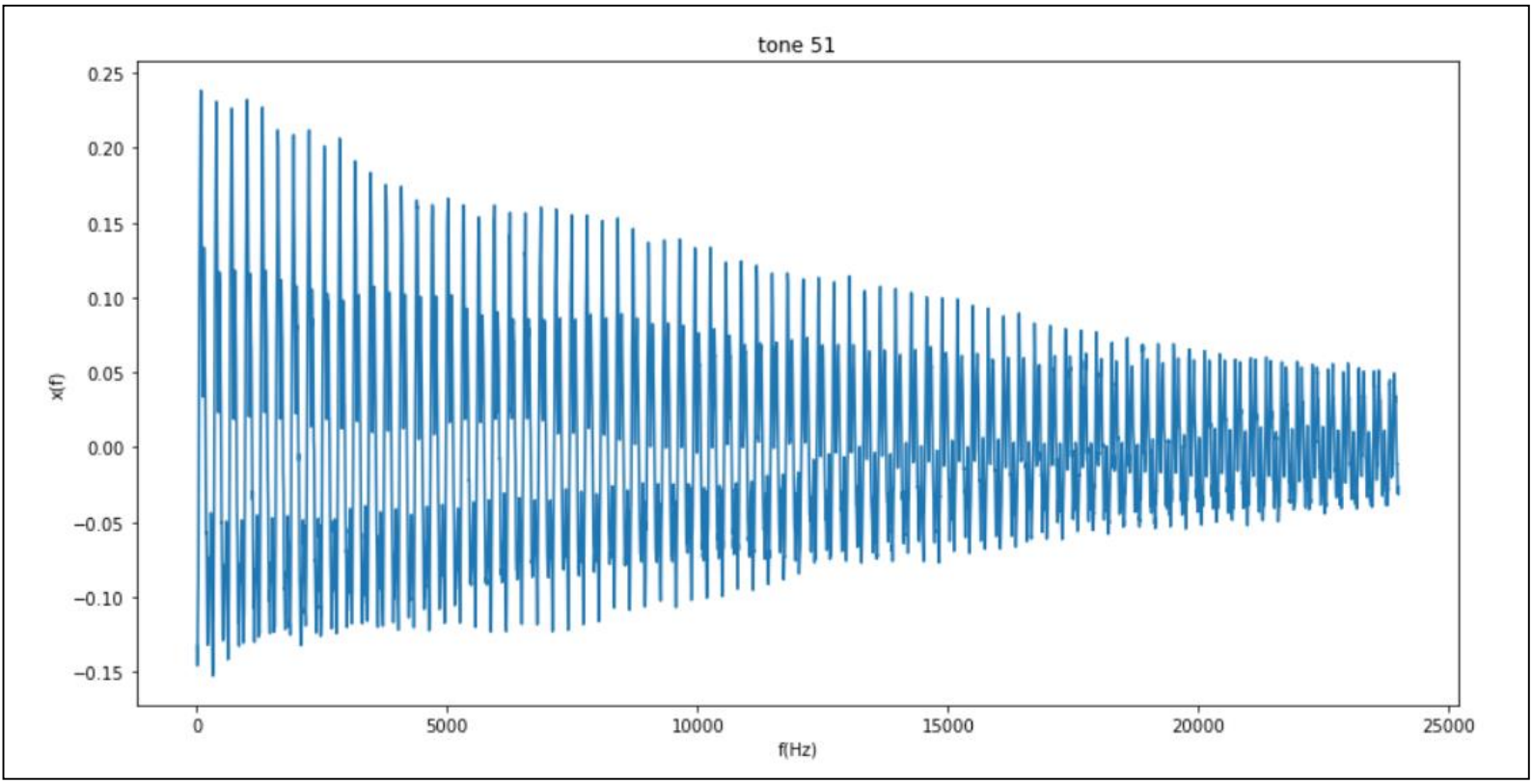
Protocol for ISS project 2022/23

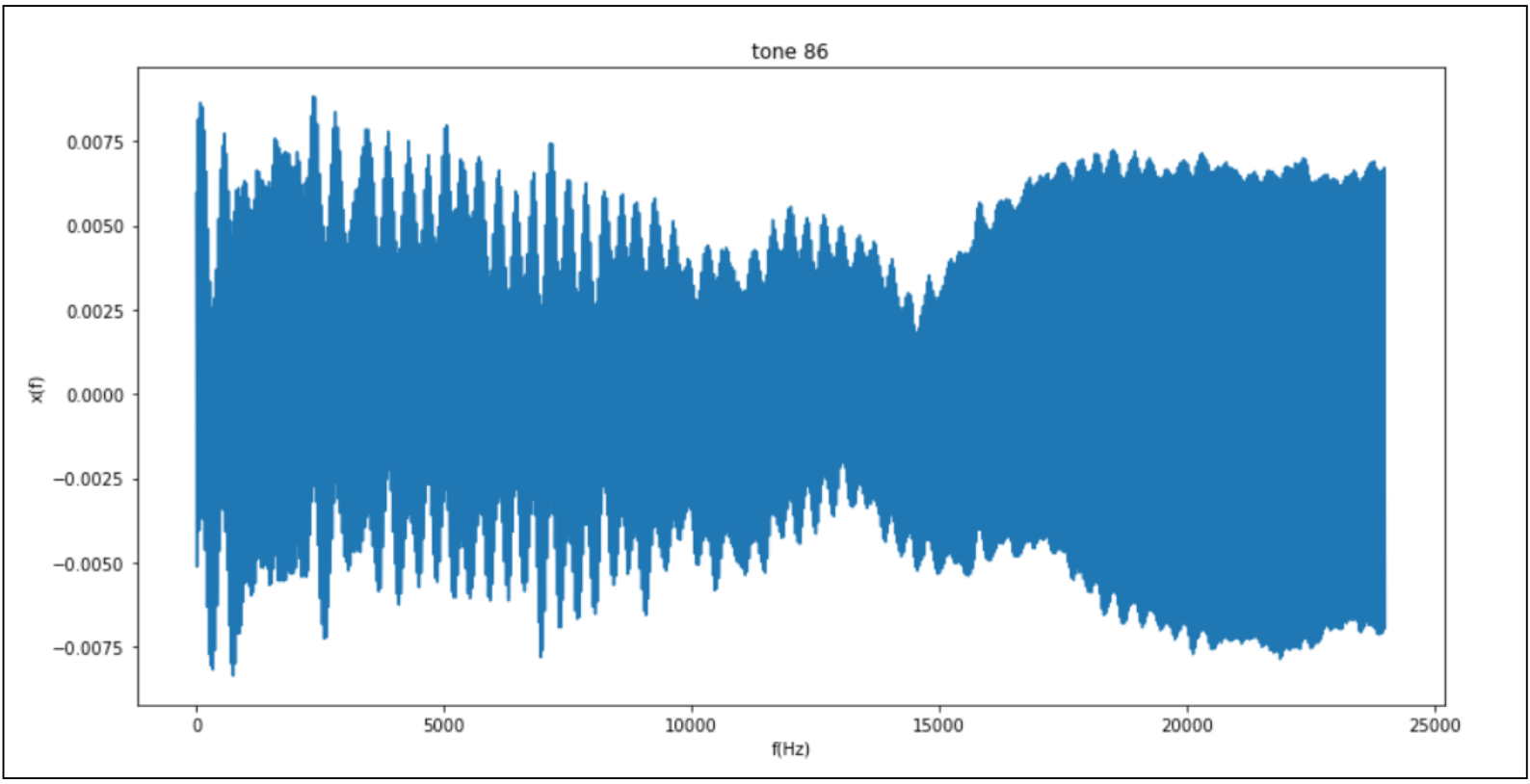
Aleksandr Shevchenko

xshevc01

1) For a good visibility I've plotted 0.5 seconds of each of my 3 tones:

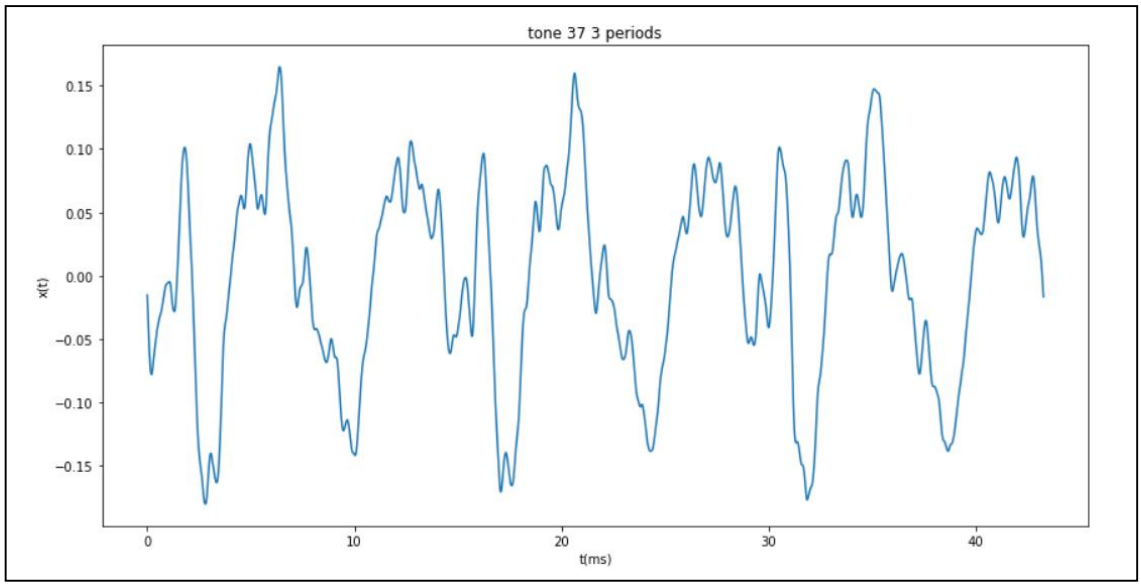


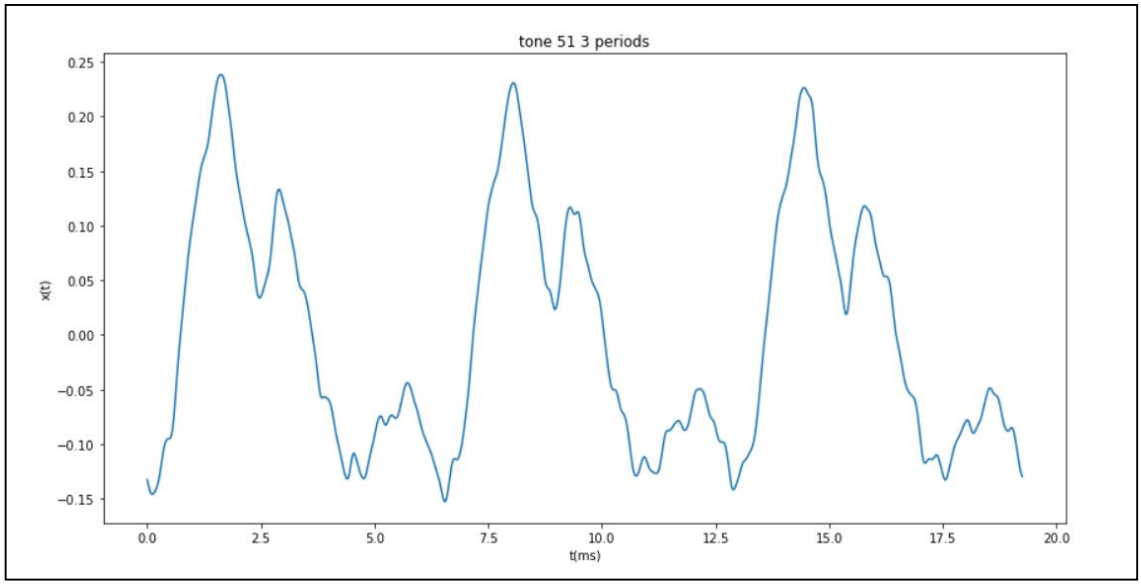


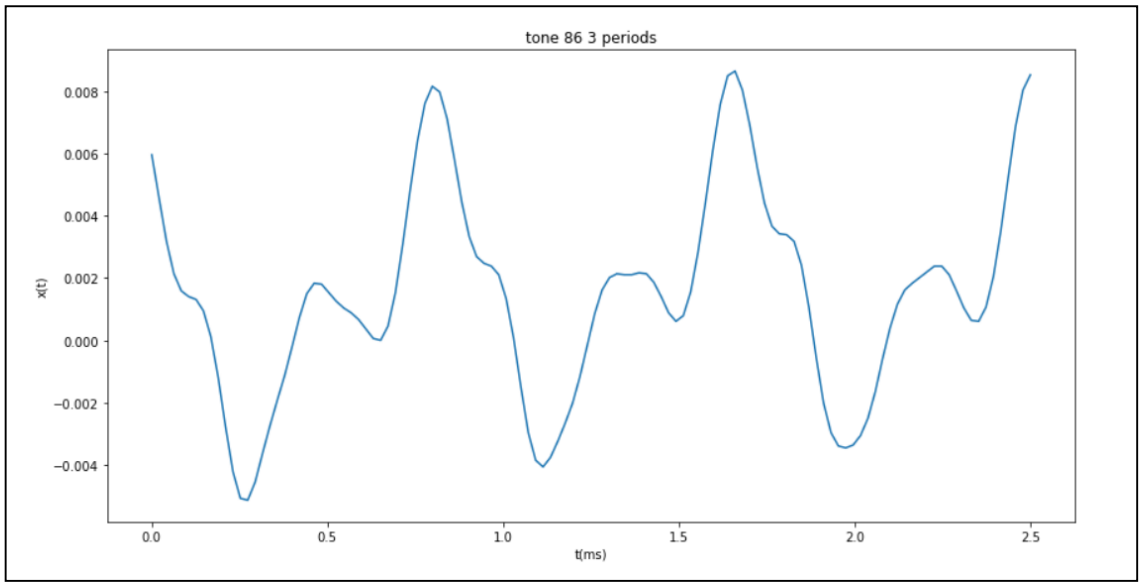


To plot 3 periods of my tones I had to find a number of samples in 1 period. I've done it by diving the sample frequency (Fs) by tone frequency. Here I have a code example for one tone:

```
# TONE 1
freq1 = 69.30
samplesInPeriod1 = int(Fs / freq1)
x = np.linspace(0, 3*samplesInPeriod1/Fs*1000, 3*samplesInPeriod1)
y = xall[mytone1][0:samplesInPeriod1*3]
plt.figure(figsize=(14,7))
plt.plot(x,y)
plt.gca().set_title("tone " + str(mytone1) + " 3 periods")
plt.gca().set_xlabel("t(ms)")
plt.gca().set_ylabel("x(t)")
plt.show()
```


tone 37 3 periods
x(t)
t(ms)
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
10
20
30
40

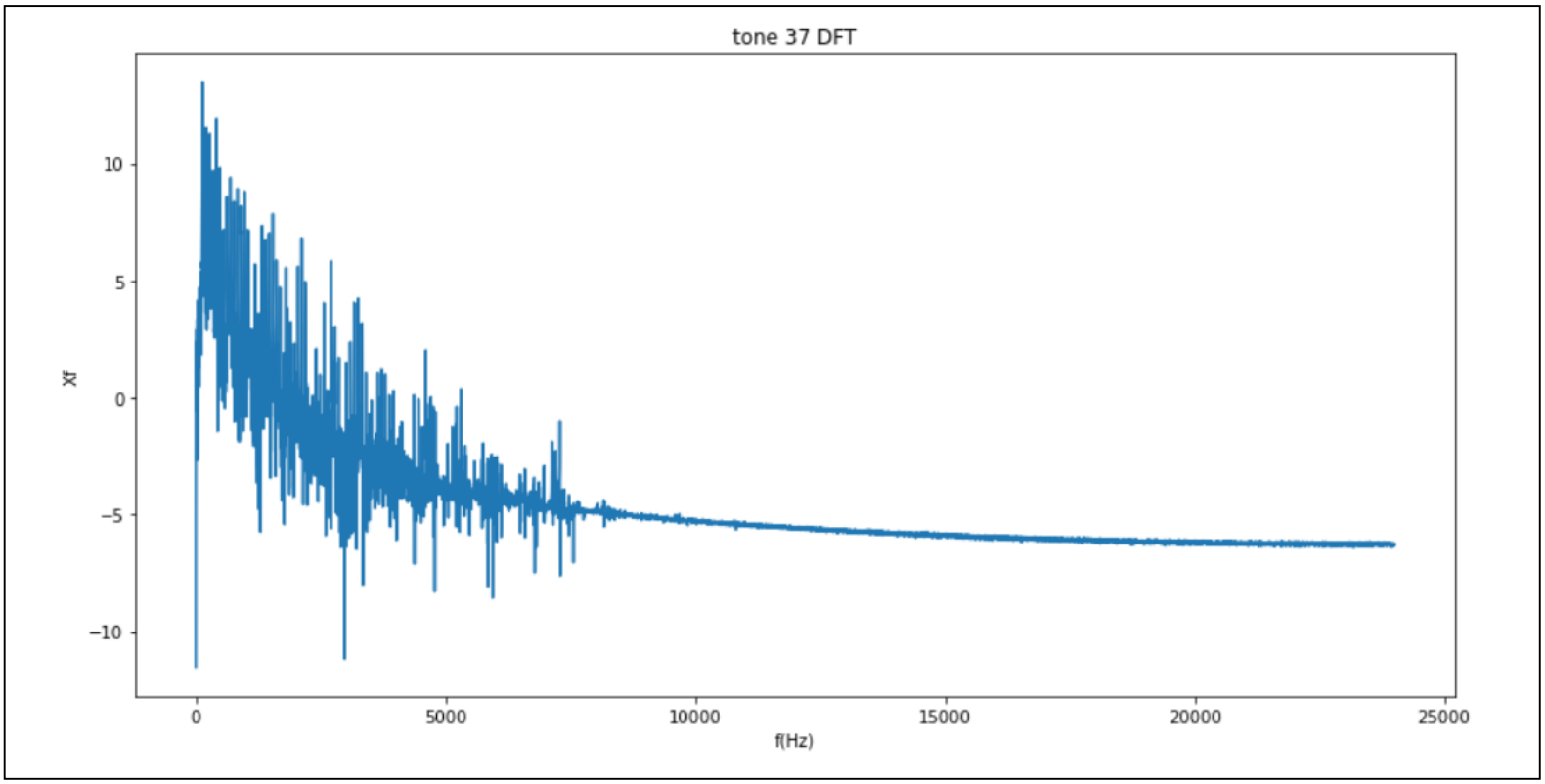

tone 51 3 periods
x(t)
t(ms)
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

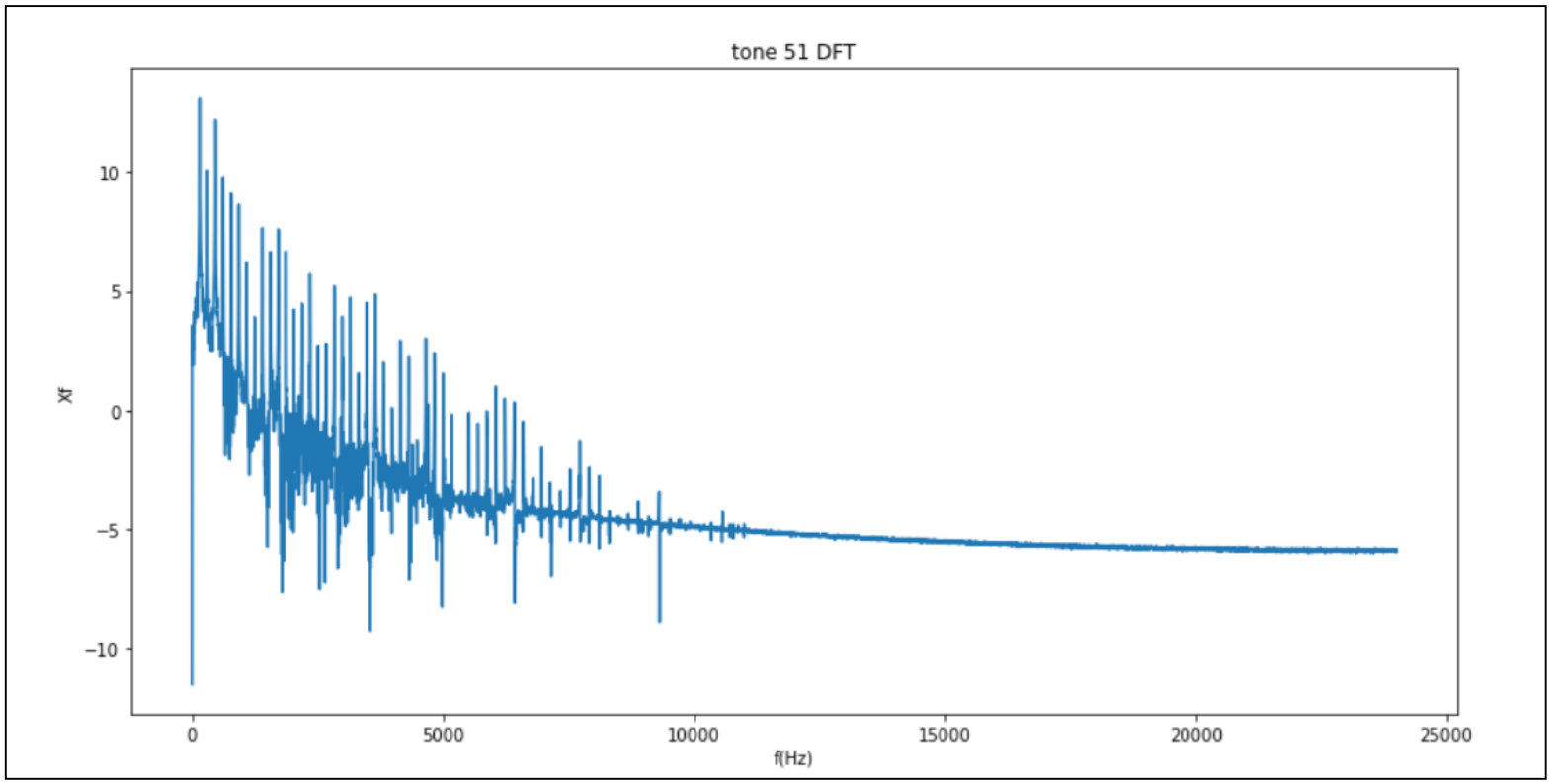

tone 86 3 periods
x(t)
t(ms)
0.008
0.006
0.004
0.002
0.000
−0.002
−0.004
0.0
0.5
1.0
1.5
2.0
2.5

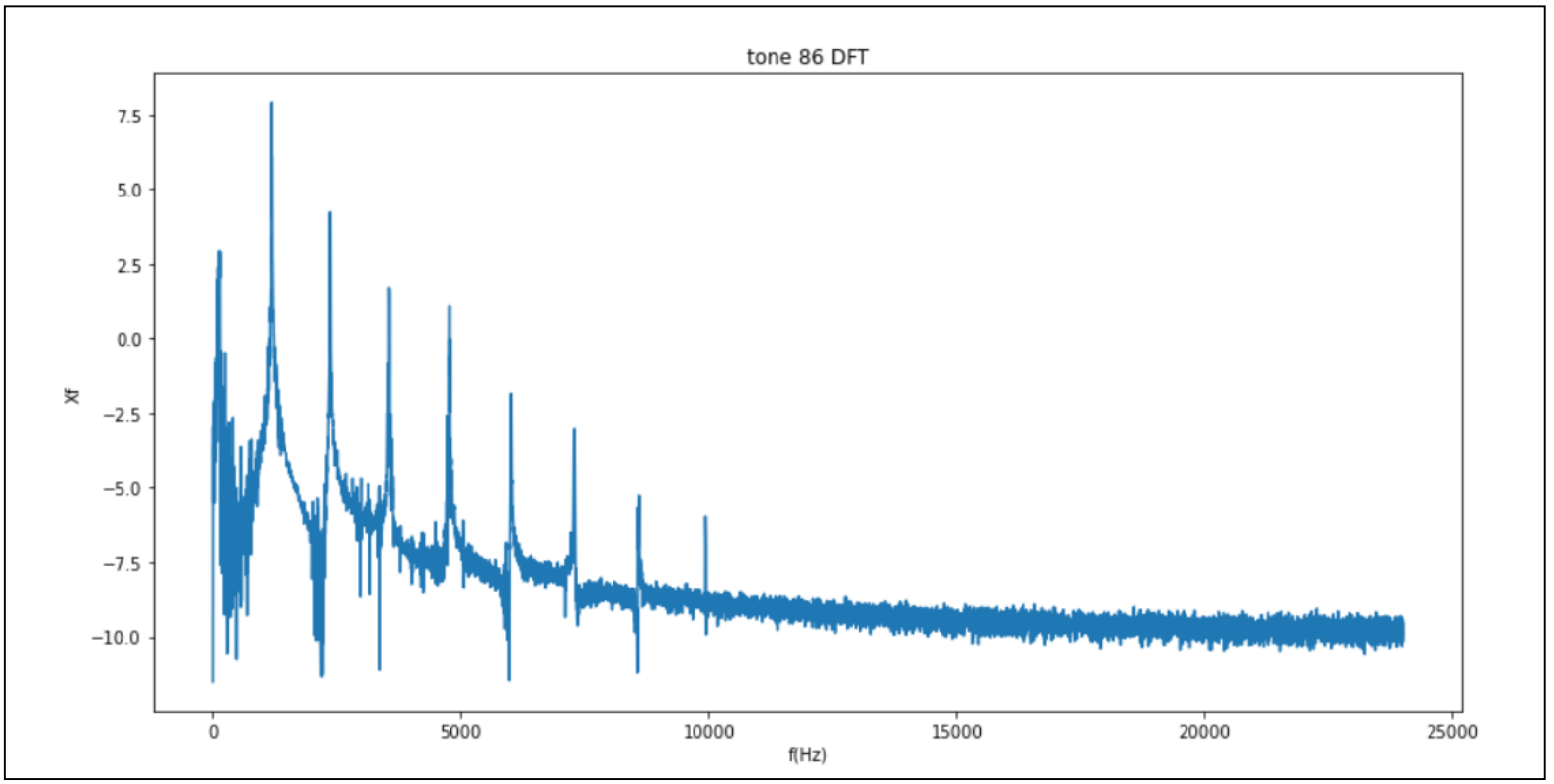
A block of code below shows my implementation for DFT, using numpy function *numpy.fft.fft*:

```
xAxesDFT = np.arange(xall[0].size/2)/HOWMUCH_SEC

# 1 tone
# his DFT
tone1DFT = np.fft.fft(xall[mytone1])
module1 = np.log(np.abs(tone1DFT)**2 + 10 ** -5)
# module1 = np.abs(tone1DFT)
module1Half = module1[:module1.size // 2]
plt.figure(figsize=(14,7))
plt.plot(xAxesDFT, module1Half)
plt.gca().set_title("tone " + str(mytone1) + " DFT")
plt.gca().set_xlabel("f(Hz)")
plt.gca().set_ylabel("Xf")
plt.show()
```




tone 51 DFT
Xf
f(Hz)
10
5
0
−5
−10
0
5000
10000
15000
20000
25000

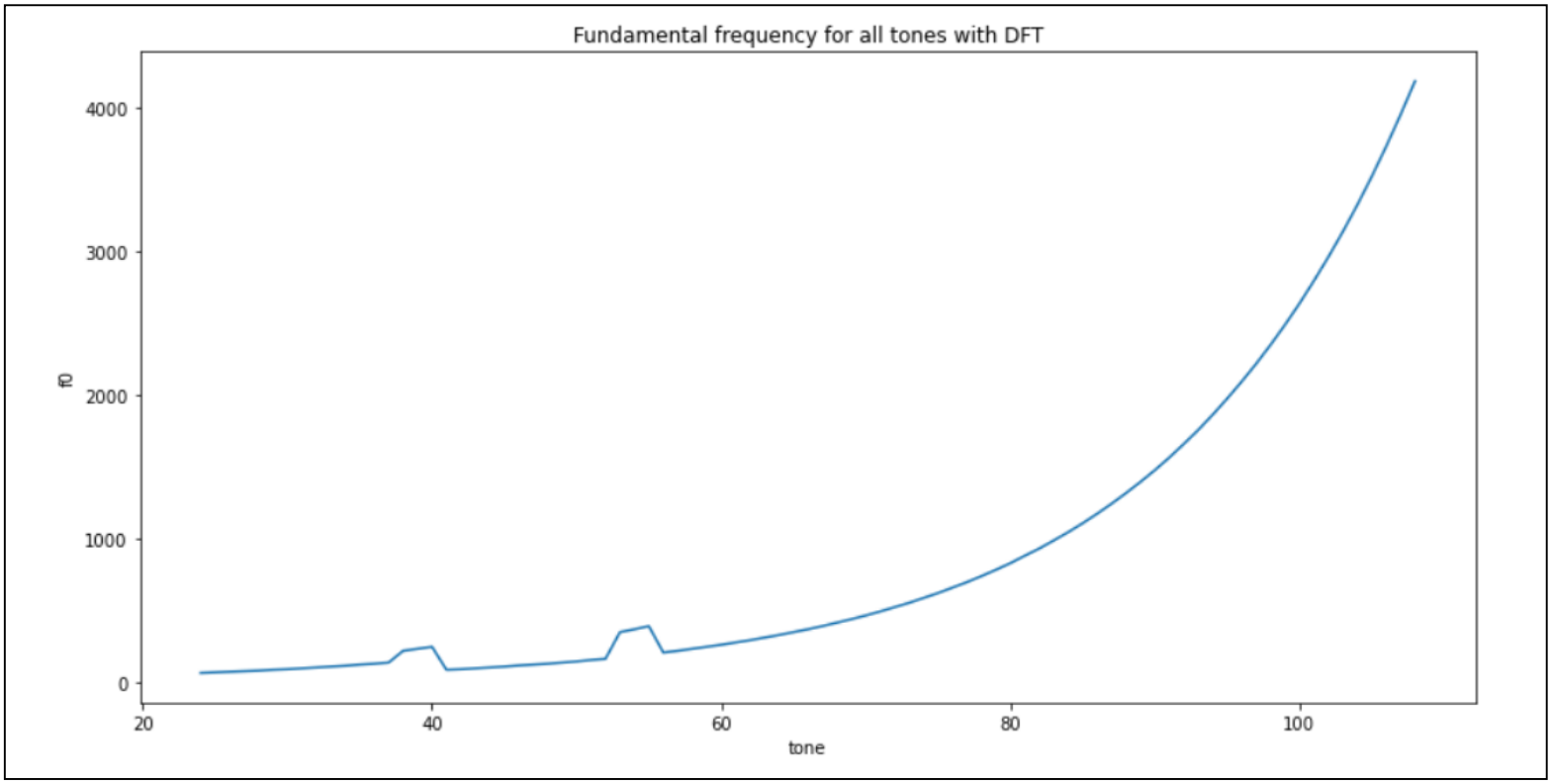

tone 86 DFT
Xf
f(Hz)
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
5000
10000
15000
20000
25000

2) For the second task I used both methods described in the assignment:

In **DFT** method, I found indices with maximum values and plotted these values:

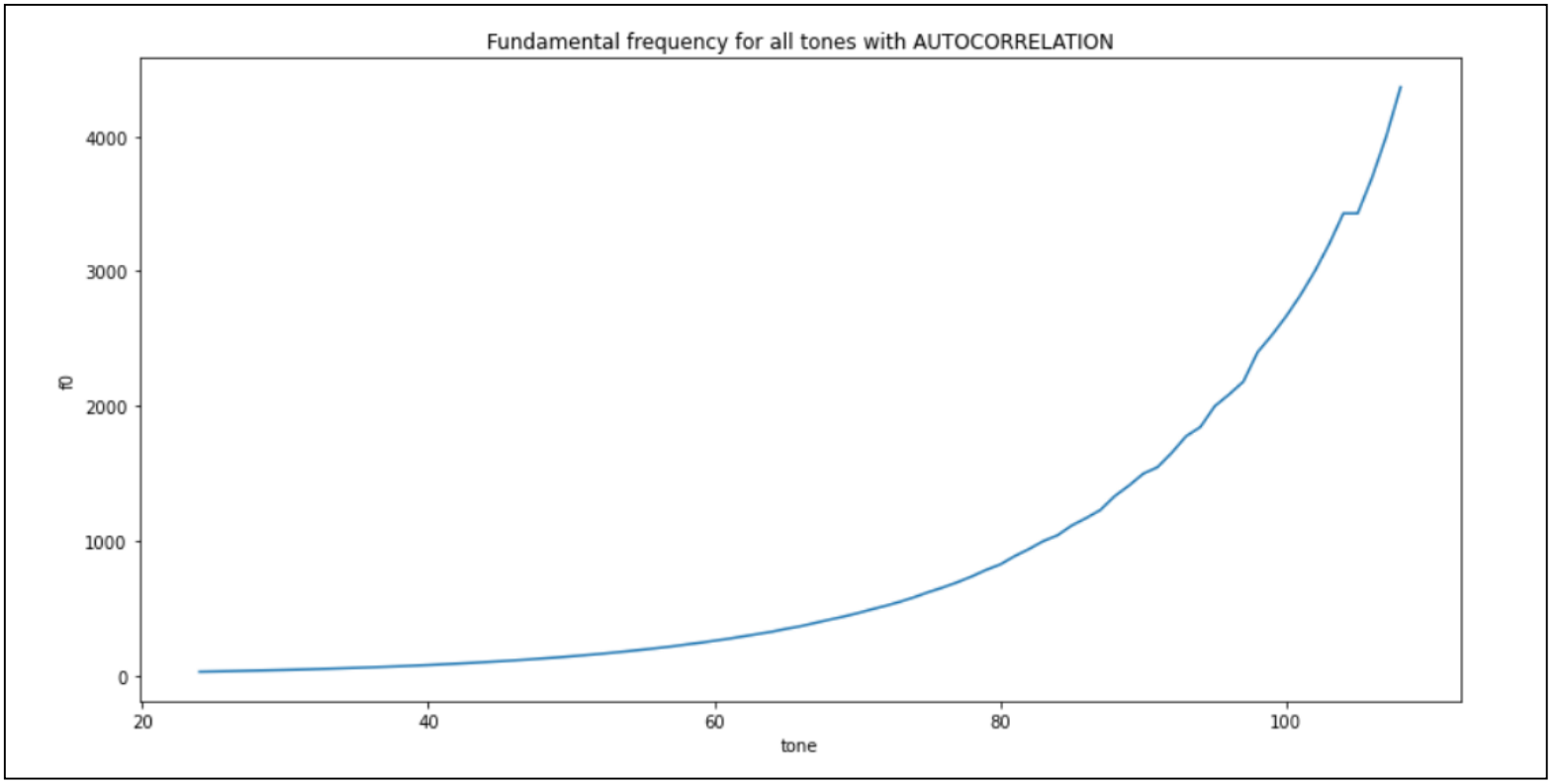
```
xAxesDFT = np.arange(xall[0].size/2)/HOWMUCH_SEC

# 1 tone
# his DFT
tone1DFT = np.fft.fft(xall[mytone1])
module1 = np.log(np.abs(tone1DFT)**2 + 10 ** -5)
# module1 = np.abs(tone1DFT)
module1Half = module1[:module1.size // 2]
plt.figure(figsize=(14,7))
plt.plot(xAxesDFT, module1Half)
plt.gca().set_title("tone " + str(mytone1) + " DFT")
plt.gca().set_xlabel("f(Hz)")
plt.gca().set_ylabel("Xf")
plt.show()
```

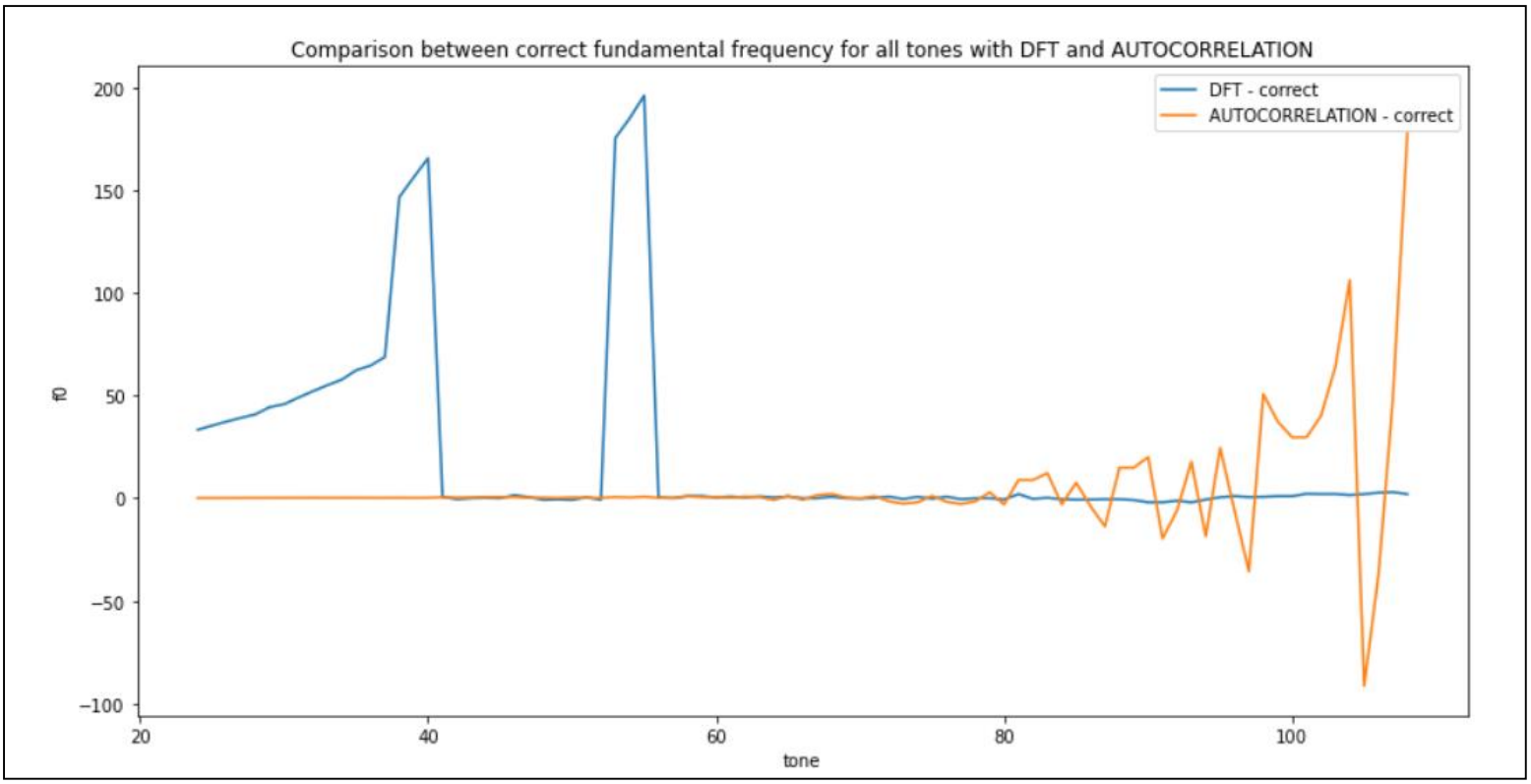


As for **autocorrelation**, the hardest thing was to find the distance between two peaks. To solve this problem I used *scipy.signal.find_peaks* and comparing values with the zeroth peak:

```
freqWithAUTOCOR = np.zeros((MIDITO-MIDIFROM+1))
# using peak0 like thrashhold and * 0.8
for tone in tones:
    toneAUTOCOR = np.correlate(xall[tone], xall[tone], "full")
    toneAUTOCOR = toneAUTOCOR[toneAUTOCOR.size // 2 - 2:]
    peaks, _ = find_peaks(toneAUTOCOR)
    maxIndValue = peaks[0]
    secMaxIndValue = 0
    for peak in peaks:
        if toneAUTOCOR[peak] > toneAUTOCOR[maxIndValue]*0.8 and toneAUTOCOR[peak] != toneAUTOCOR[maxIndValue]:
            secMaxIndValue = peak
            break
    freqWithAUTOCOR[tone-24] = Fs / (secMaxIndValue - maxIndValue)
```



Finally, the comparison between both methods by subtraction given f0 values from them:

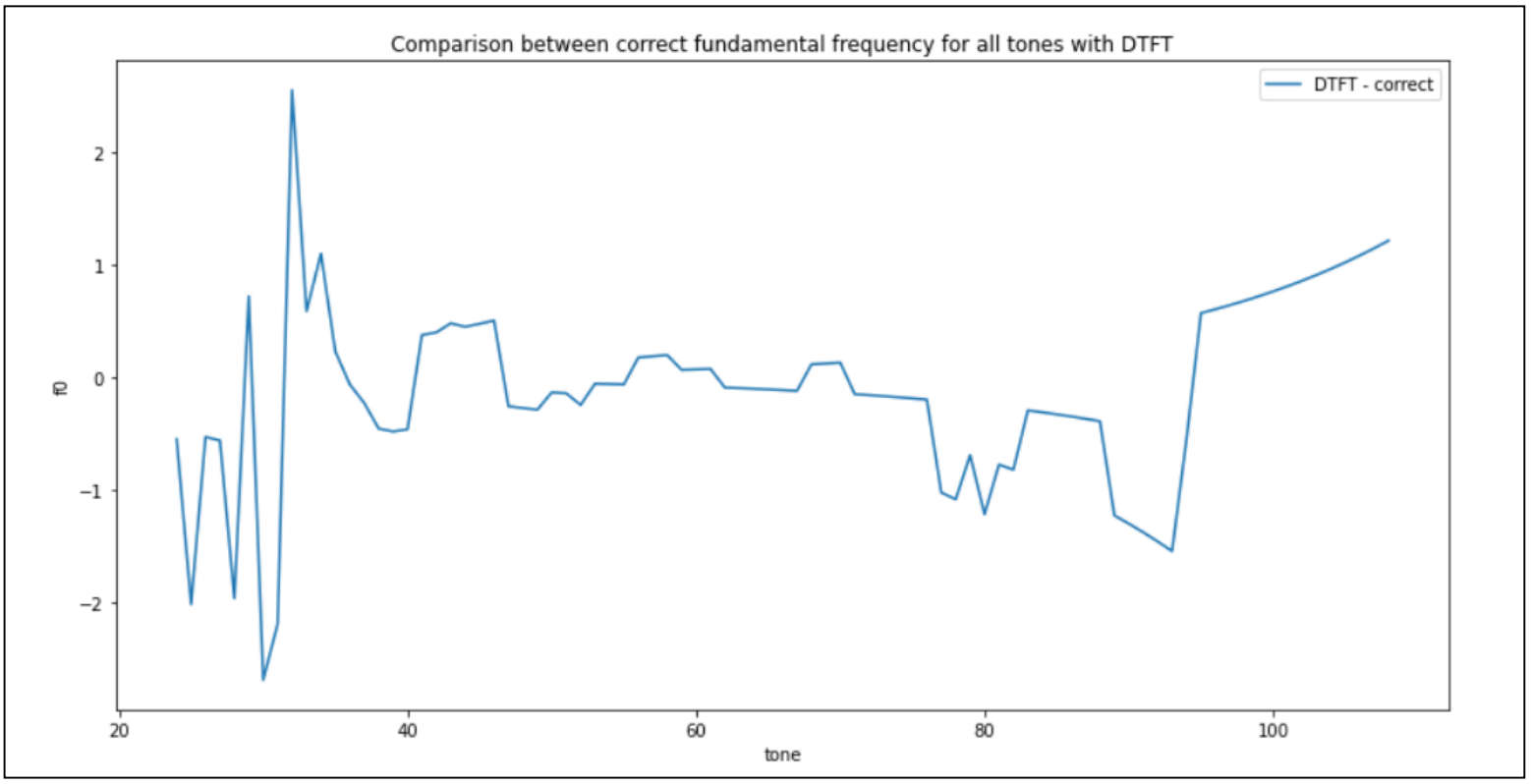


The difference for DFT in low frequencies and for autocorrelation in high frequencies corresponds to the hint, given in the task: both methods are not ideal.

3) In this task, I am using **100 cents method**. A code below shows my implementation of DTFT:

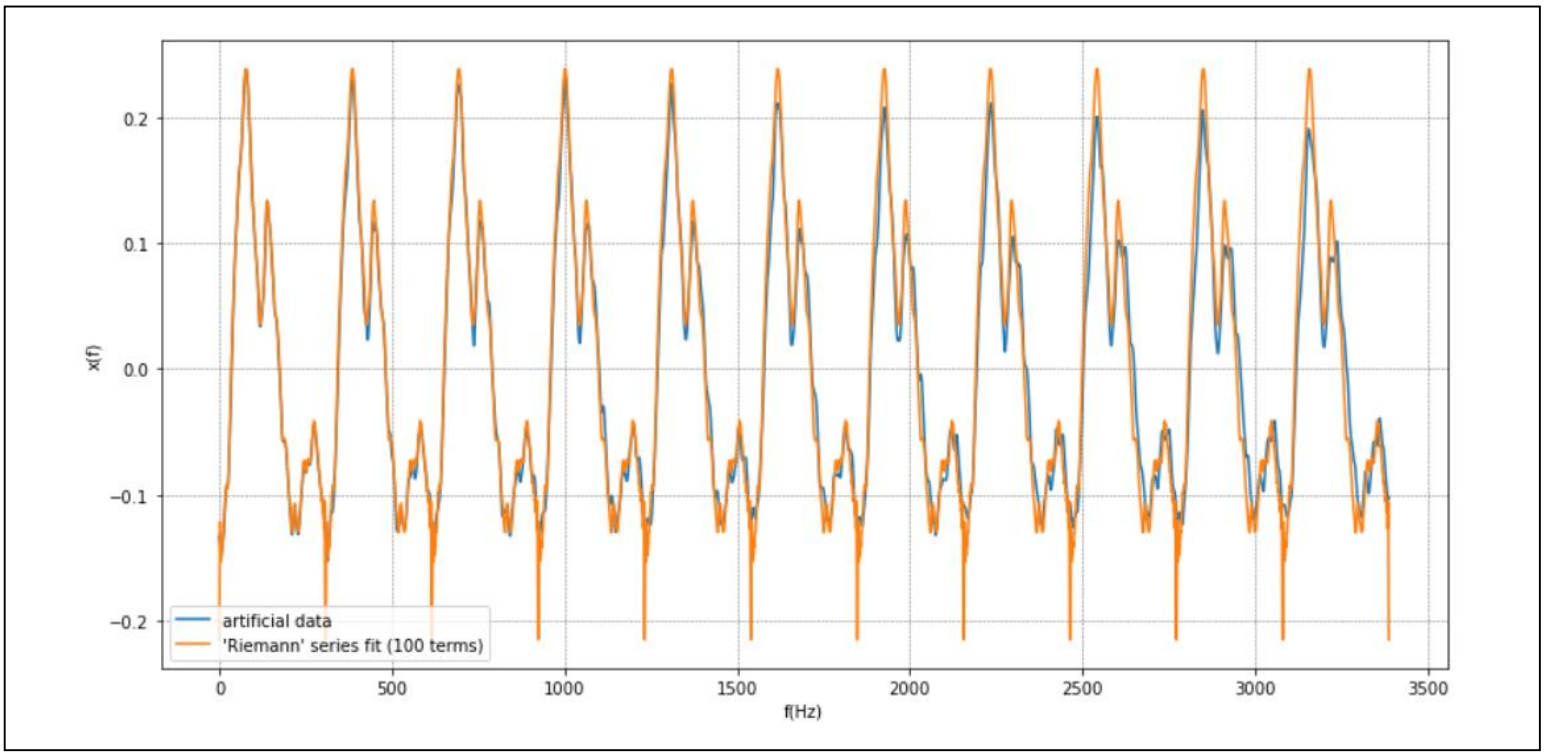
```
freqWithAUTOCOR = np.zeros((MIDITO-MIDIFROM+1))
# using peak0 like thrashhold and * 0.8
for tone in tones:
    toneAUTOCOR = np.correlate(xall[tone], xall[tone], "full")
    toneAUTOCOR = toneAUTOCOR[toneAUTOCOR.size // 2 - 2:]
    peaks, _ = find_peaks(toneAUTOCOR)
    maxIndValue = peaks[0]
    secMaxIndValue = 0
    for peak in peaks:
        if toneAUTOCOR[peak] > toneAUTOCOR[maxIndValue]*0.8 and toneAUTOCOR[peak] != toneAUTOCOR[maxIndValue]:
            secMaxIndValue = peak
            break
    freqWithAUTOCOR[tone-24] = Fs / (secMaxIndValue - maxIndValue)
```

Again, comparison between DTFT and given f0 values:



With no doubt this method is more precise than DFT and autocorrelation, the range of values is only ~2.

4,5) For these tasks I tried using “Riemann” series (link), the results were visibly fine, for example here, for tone 51:



However, for some reason synthesized signals were horrible to listen to, so I did not even attach them to the solution. I will leave the code for tasks 4 and 5 in my python notebook with obtained audios.

6) Despite the fact, that two previous tasks were failed, I created the song from MIDI file and played with my own songs using the instrument from the assignment.

Instead of Fourier series, I used the initial piano samples, stored in the variable *xall*. To do this, I saved four columns from MIDI file to the numpy arrays, found the biggest “DO” value, divided it by 1000 and got the length of song. Then I created two arrays for songs in 48k and 8k, and from row to row added the values of particular tones:

```
# CREATE AN ARRAY
freq1 = 48000
freq2 = 8000
song48 = np.zeros(int(maxSEC * freq1))
song8 = np.zeros(int(maxSEC * freq1))
```

```
# ADDING TONES
for i in range (numberOfRows):
    odSamp48 = int(od[i] * freq1 // 1000)
    doSamp48 = int(do[i] * freq1 // 1000)
    odSamp8 = int(od[i] * freq2 // 1000)
    doSamp8 = int(do[i] * freq2 // 1000)
    midiSamp = int(MIDI[i])
    hlasSamp = hlasitost[i] / 100
    for j in range(odSamp48, doSamp48):
        song48[j] += xall[midiSamp][(j-odSamp48)%len(xall[0]-1)] * hlasSamp
        song8[j] += xall[midiSamp][((j-odSamp48)-(j-odSamp48)%(freq1//freq2))%len(xall[0]-1)] * hlasSamp
```

In the result, the song is quite pleasant to listen to.

7) I didn’t make a spectrogram, because it requires successful implementation of all previous parts.