VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
DUOMENŲ MOKSLO IR SKAITMENINIŲ TECHNOLOGIJŲ INSTITUTAS

GARSO SIGNALŲ APDOROJIMAS

Ketvirtasis laboratorinis darbas

Trumpalaikio spektro analizė

Darbą atliko 4 kurso ISI 2 grupės studentas

Tomas Šinkūnas

Vilnius, 2023

Turinys

Įvadas

Darbo tikslas

Trumpalaikio spektro analizė, analizės rezultatų interpretavimas.

Darbo užduotis

Sukurti priemonę signalams ir jų analizės rezultatams grafiškai atvaizduoti. Atlikti pasirinktosios signalo atkarpos spektro analizę. Įvertinti dažninę signalo sudėtį, išskirti esmines dažnines savybes.

Darbo priemonės

Darbui atlikti buvo naudojama *Python* programavimo kalba su *matplotlib*, *numpy* ir *wave* bibliotekomis.

Duomenys

Šiame laboratoriniame darbe yra analizuojami 14 garso failų:

1. Muffled-drum-kick.wav (toliau bus vadinamas kaip *drum.wav*),
2. Opera-vocal_129bpm_F_minor.wav (toliau bus vadinamas kaip *opera.wav*)
3. flute-C4.wav, flute-C5.wav, flute-C6.wav,
4. piano-C4.wav, piano-C5.wav, piano-C6.wav,
5. trumpet-C4.wav, trumpet-C5.wav, trumpet-C6.wav,
6. violin-C4.wav, violin-C5.wav, violin-C6.wav.

Darbo eiga

Visiems laboratoriniame darbe nagrinėjamiems garso signalams buvo pateikiama laiko diagrama, kurioje vartotojas gali nuspręsti, kurią atkarpą analizuoti. Uždarius diagramos langą programa prašo įvesti signalo pradžios laiką bei trukmę (sekundėmis). *Pirminė laiko diagrama bei nustatymų įvedimo langai laboratoriniame darbe nėra pateikti.*

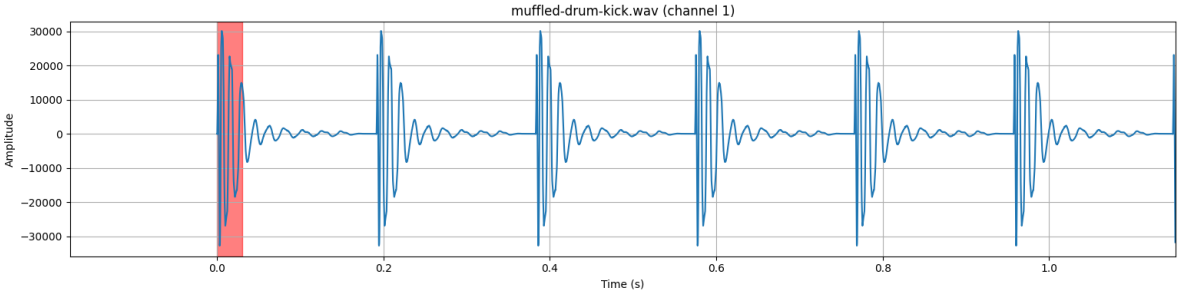
Sekančiame žingsnyje programa pateikia viso audio signalo laiko diagramą, tačiau su pažymėta/išskirta vieta, kuri bus analizuojama. Uždarius šį diagramos langą yra pateikiama išskirta (vartotojo pasirinkta) atkarpa.

Kiekvienai laiko diagramai buvo pritaikyta *Hanning* lango funkcija, kuri tolygiai sumažina signalo amplitudę ties signalo pradžia bei pabaiga. Hanning funcijos diagrama nėra pateikta darbe, tačiau yra pateiktas garso signalas su jau pritaikyta lango funkcija.

Paskutiniame žingsnyje yra pateikiama spektrinė dažnių diagrama.

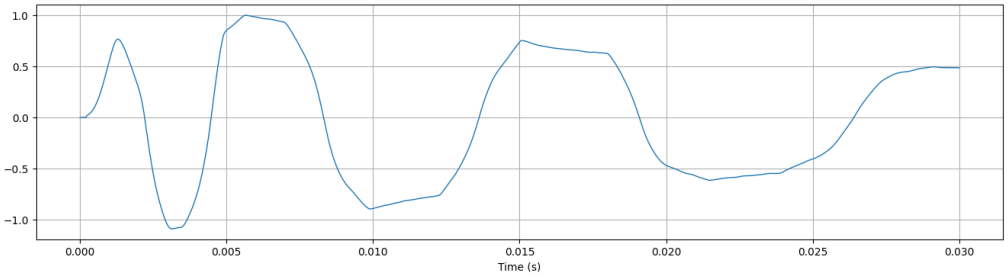
Drum.wav failo analizė

Iš laiko diagramos, pateiktos paveiksle 1 galime matyti, jog tai pasikartojantis, monotoninis garso signalas. Amplitudinės ir dažninės charakteristikos vizualiai atrodo labai panašios, kas gali simbolizuoti periodiškumą.

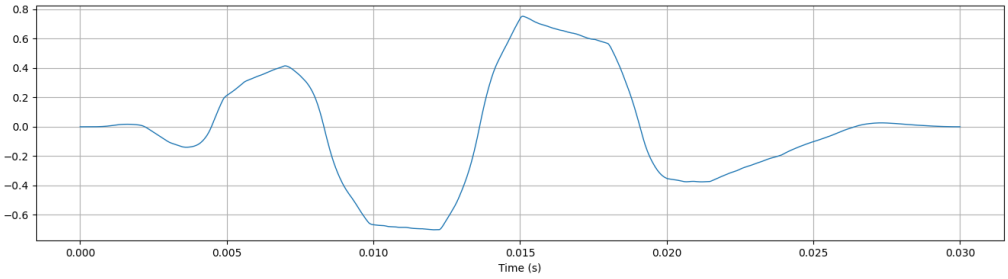


Pav 1. Failo Drum.wav laiko diagrama (apkarpyta)

Iš vartotojo išskirtos laiko diagramos, pateiktos 2 paveiksle, galime daryti prielaidą, jog tai yra žemų dažnių garso signalas, nes per 0.03 sekundžių laiko tarpą turime tik 3 sinusoidės periodus. Kreivės “kampuotumas” rodo, jog esamo 44100Hz diskretizavimo dažnio nepakanka tiksliai užfiksuoti tokios sinusoidės svyravimus.



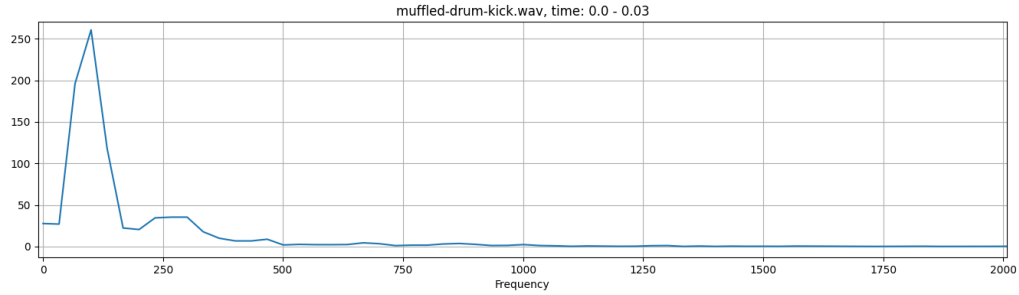
Pav 2. Failo Drum.wav laiko diagrama atkarpoje 0.00s – 0.03s



Pav 3. Failo Drum.wav laiko diagrama su pritaikyta Hanning lango funkcija

Iš dažnių spektro diagramos matome (paveikslas 4), jog vyraujantis yra ~98 Hz dažnis, kas atitinka G2 natą.

Pagal standartus, bumbo-būgnai (angl. *kick-drum*) yra derinami pagal G1 natos atskaitos tašką, kuris atitinka 49 Hz., tačiau mūsų atveju būgnas yra derinamas oktava aukščiau, t.y., apie ~98Hz (G2). Tačiau tikslių tvirtinimų, kad jis yra ar nėra suderintas G1 natoje, negalime pateikti, nes būgnas yra prislopintas



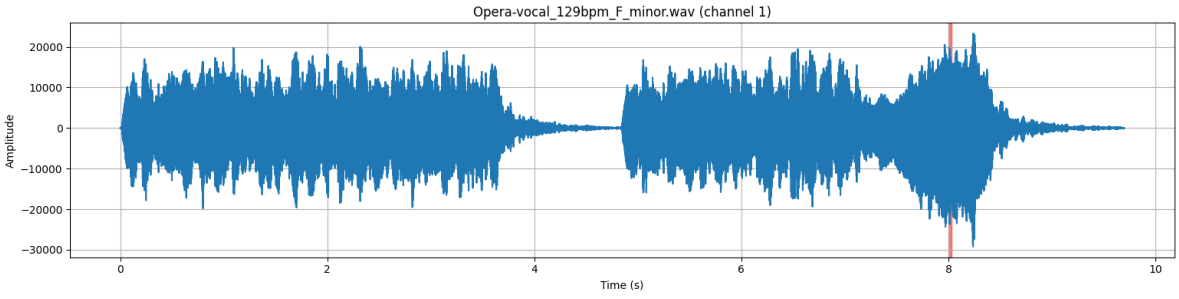
Pav 4. Failo Drum.wav dažnių spektro diagrama

Būgno slopinimui paprastai į jo vidų dedamos slopinimo medžiagas, tokios kaip pagalvės, antklodės ar putos. Šios medžiagos sugeria arba mažina būgno membranos vibracijas, kurios yra atsakingos už žemesnes dažnių sritis. Sumažindami vibracijas, sumažiname ir ilgalaikį išlaikymą (angl. *sustain*) ir rezonansą šiose žemesnėse dažnių srityse, rezultate gaudami tvirtesnį, geriau kontroliuojamą garsą. Dėl to negalime teigti, jog būgnas nėra suderintas G1 natoje.

Antraji dažnių spektro dedamoji yra labai nežymi, lyginant su pirmąja, ir svyruoja 234Hz - 300Hz atkarpoje, kas atitinka A#3 (233.08Hz) – D4 (293.66Hz) dažnių diapazoną. Tikslesnei reikšmei nustatyti reikėtų didinti audio įrašo diskretizacijos dažnį bent dvigubai, nuo 44100Hz iki 88200Hz.

Opera.wav failo analizė

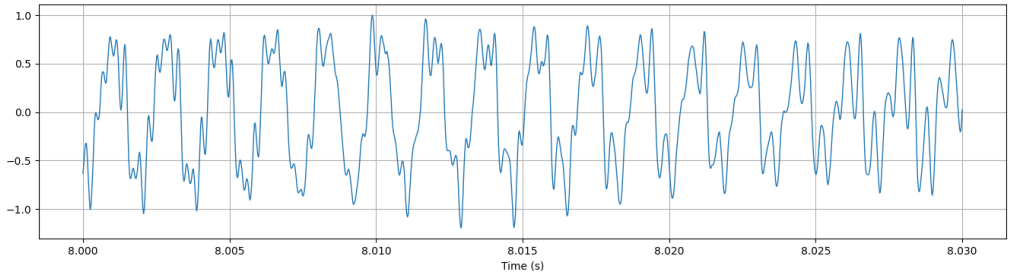
Iš laiko diagramos, pateiktos paveiksle 5 galime daryti prielaidą, jog įraše yra bent du žodžiai (ar sakiniai) su pauze tarp jų ties 4 – 5 sekunde. Šiame tarpe garso amplitudė yra nykstanti, kas gali reikšti tylą.



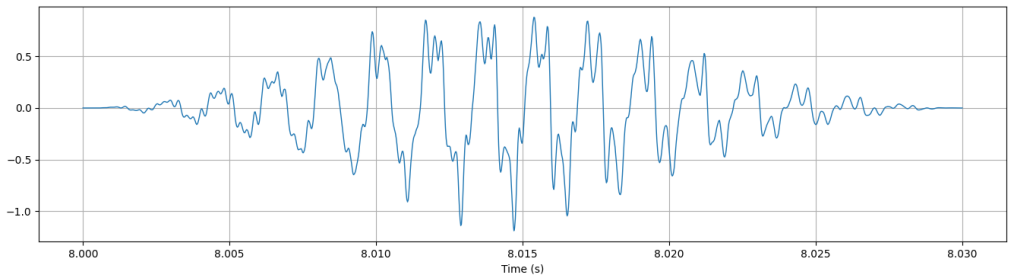
Pav 5. Failo Opera.wav laiko diagrama

Spektrinei analizei pasirinkau laiko atkarpą 8.00 – 8.03, nes buvo idomu sužinoti, kokiame registre ir kokia nata skamba šiame finaliniame akorde.

Laiko diagramoje, pateiktoje 6 paveiksme, galime įžvelgti, jog garso signale yra bent 7’ios dažnių dedamososios, signalas atrodo natūralus, nechaotiškas. Žemesnių dažnių dedamoji turi didžiausią amplitudę, o kitos, didesnių dažnių dedamosios - mažesnę amplitudę.

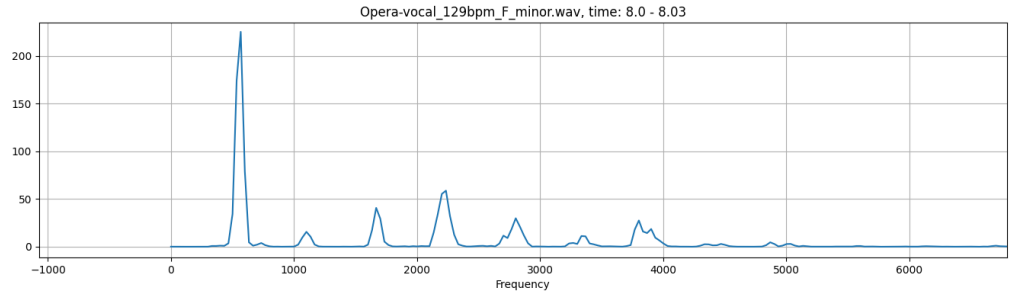


Pav 6. Failo Opera.wav laiko diagrama atkarpoje 8.00 – 8.03



Pav 7. Failo Opera.wav laiko diagrama su pritaikyta Hanning lango funkcija

Iš dažnių spektro diagramos (paveikslas 8) matome, jog spektas kinta nuo vidutinių iki aukštų dažnių. Žinant, jog tai yra operos vokalistės balsas, galime teigti, jog tai sopranas, kurios balso spektras siekia nuo 250Hz (B3) iki 1200 Hz (B6) ar daugiau.

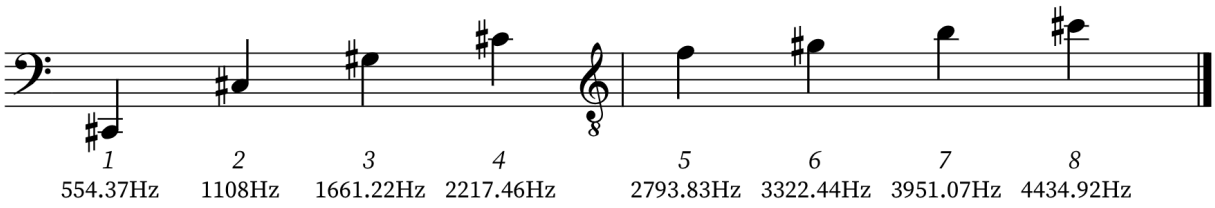


Pav 8. Failo Opera.wav dažnių spektro diagrama

Pirmoji dažnių dedamoji, dar vadinama bazine harmonika, yra stipriausia – ~560Hz, kas atitinka C#5 (554.37Hz) natą (paveikslas 9). Antroji, tačiau su mažesne amplitude – 1108Hz (C#6), kuri yra visa oktava aukščiau nuo pirmosios harmonikos. Ji yra vadinama pirmuoju obertonu.

Trečioji dedamoji – ~1660Hz, atitinkanti G#6 natą (1661.22Hz) yra trečioji bazinio dažnio kartotinė, C#6-C#7 oktavą dalinanti pusiau. Ketvirtoji harmonika – per dvi oktavas aukščiau nuo bazinio dažnio – ~2220Hz, atitinkanti C#7 natą (2217.46Hz), vadinama antruoju obertonu.

Likusios trys dedamosios yra išsidėsčiusios prelimenariai ties F7 (2793.83Hz), G#7 (3322.44Hz), B7 (3951.07Hz) dažniais, kurios oktavą nuo C#7 iki C#8 dalina į 4 lygias dalis, sudarydamos dominanto septintinių akordų seką (C#7, F7, G#7, B7).



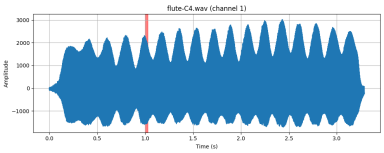
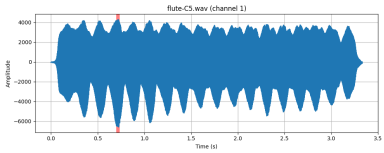
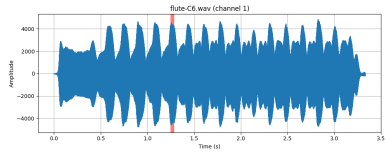
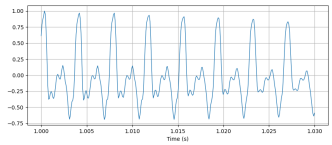
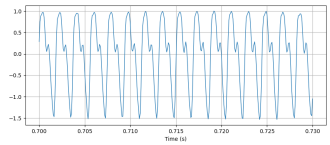
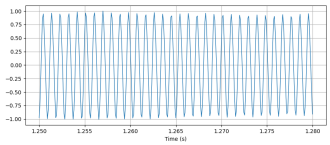
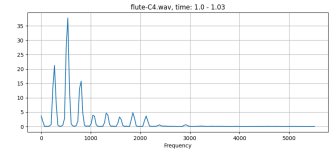
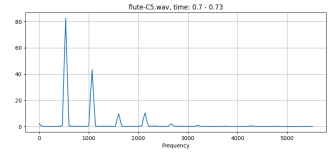
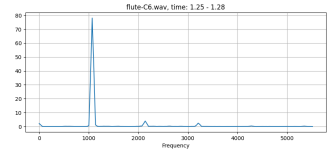
Pav 9. Failo Opera.wav dažnių spektro diagramos atitikmuo penklinėje. (natos yra pažemintos 3'imis oktavomis skaitomumo pagerinimui)

Skambant pagrindinei natai, šiuo atveju C#5, iš tikro girdime ir kitas, aukštesnes bei tylesnes natas, kurios yra vadinamos harmonikomis arba obertonais (angl. *overtones*). Girdime 2, 3, 4 ir t.t. pagrindinės natos slopstančias kartotines, dėl to dažnių diagramoje matome net kelias reikšmes, ne tik bazinę.

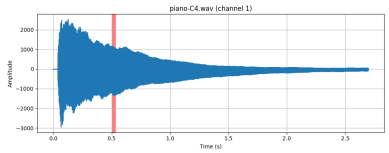
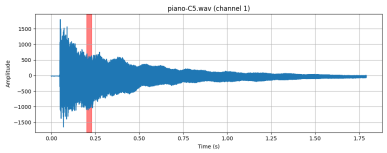
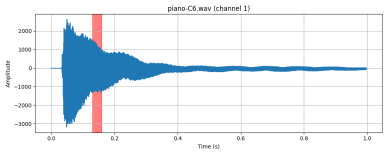
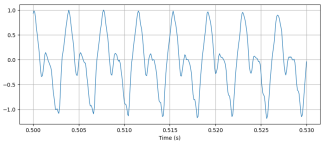
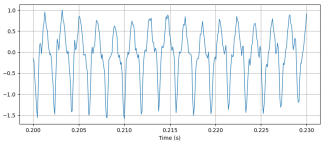
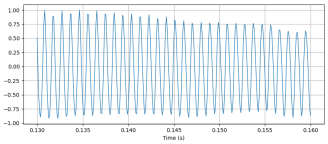
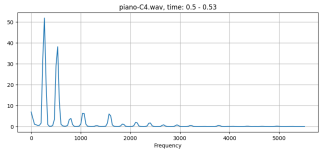
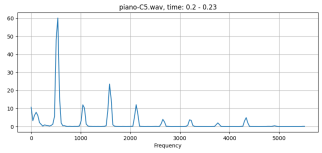
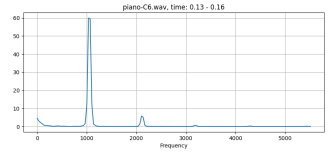
Muzikos instrumentų analizė

Šiame žingsnyje atlikau 4 skirtingų instrumentų spektinę analizę skirtingose registruose: žemame (C4), vidutiniame (C5) ir aukštame (C6). Buvo įdomu sužinoti, kuo panašios ir kuo skiriasi kiekvieno muzikos instrumento spektrinės charakteristikos grojant tas pačias natas. Šie rezultatai pateikti 1, 2, 3 ir 4 lentelėse.

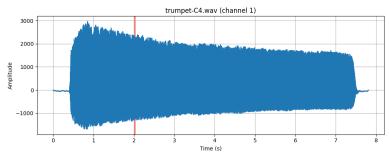
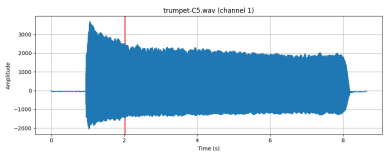
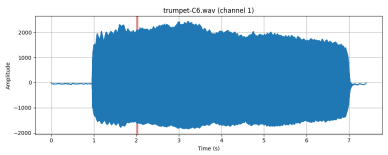
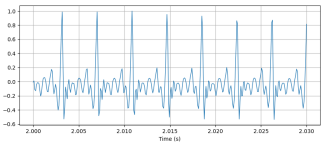
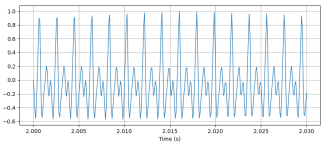
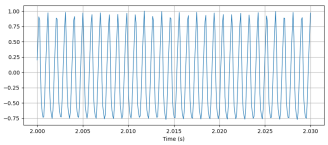
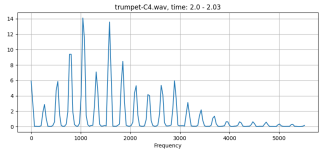
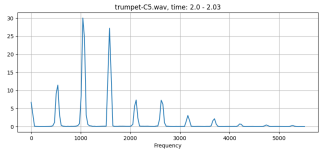
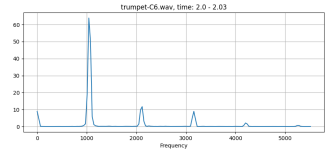
Lentelė nr. 1. Flute.wav spektrinė analizė

C4	C5	C6
		
		
		

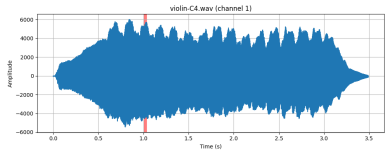
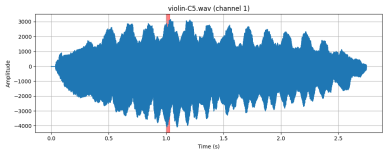
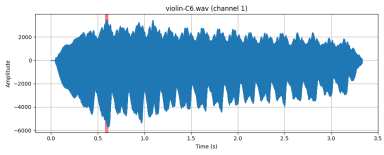
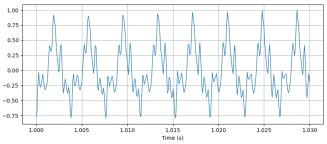
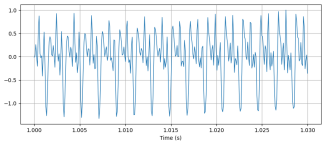
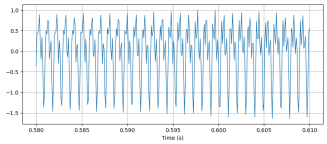
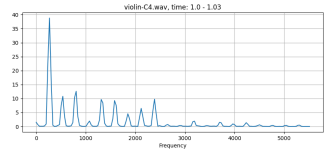
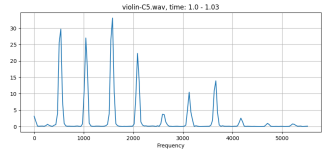
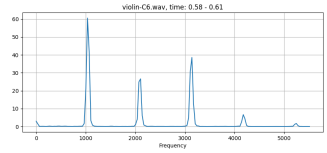
Lentelė nr. 2. Piano.wav spektrinė analizė

C4	C5	C6
		
		
		

Lentelė nr. 3. Trumpet.wav spektrinė analizė

C4	C5	C6
		
		
		

Lentelė nr. 4. Piano.wav spektrinė analizė

C4	C5	C6
		
		
		

Visi skirtingi instrumentai, grojantys tą pačią natą, generuoja tokias pačias harmonikas, skiriasi tik harmonikų ir obertonų amplitudės bei jų kiekis (lentelė 5, obertonai pažymėti tamsesne spalva, kad juos būtų lengviau atsekti, ir jie kartojasi kaip 2, 4, 8, 16 dedamoji sekoje). Kiekvieno obertono dažnį galima paskaičiuoti pagal formulę: $F_{overtone} = F_{base} \cdot 2^n$, kur $F_{overtone}$ yra obertono dažnis, F_{base} - bazinis natos dažnis, n - obertono indeksas $[0, \infty]$

Iš gautų resultatų matome, kad kuo žemesnė nata, tuo didesnis harmonikų dažnis ir atvirkšiai - aukštų natų dažninės charakteristikos yra retesnės.

Lentelė nr. 5. Skirtingų natų skleidžiamos harmonikos ir obertonai

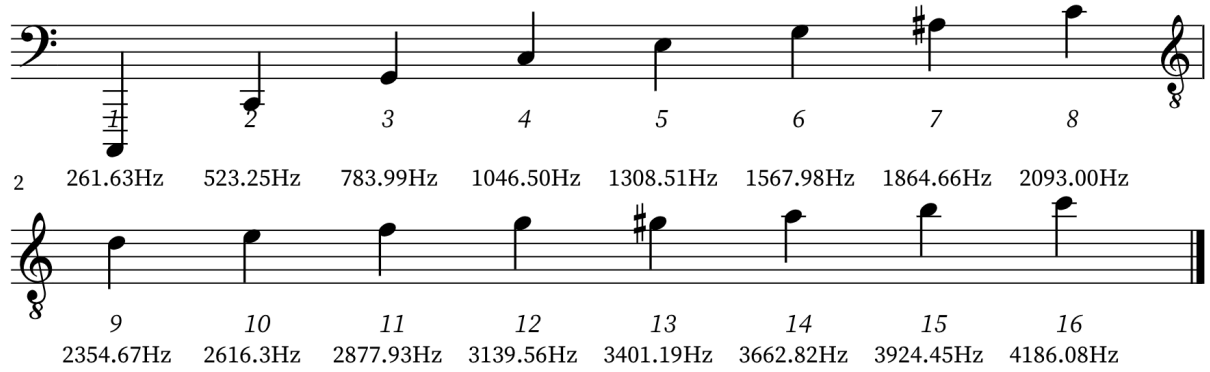
Nata	Harmonikos ir obertonai
C4	**261.63 (C4), 523.25 (C5),** 783.99 (G5), **1046.50 (C6),** 1308.51 (E6), 1567.98 (G6), 1864.66 (A#6), **2093.00 (C7),** 2354.67 (D7), 2616.3 (E7), 2877.93 (F7), 3139.56 (G7), 3401.19 (G#7), 3662.82 (A7), 3924.45 (B7), **4186.08 (C8),** 4447.71 (C#8), 4709.34 (D8), 4970.97 (D#8), 5232.6 (E8)
C5	**523.25 (C5), 1046.5 (C6),** 1569.75 (G6), **2093 (C7),** 2616.25 (E7), 3139.5 (G7), 3662.75 (A#7), **4186 (C8),** 4709.25 (D8), 5232.5 (E8)
C6	**1046.50 (C6), 2093 (C7),** 3139.5 (G7), **4186 (C8),** 5232.5 (E8), 6279 (G8)

Kiekvieno instrumento, įskaitant ir balsą, harmonikų garsumas yra skirtingas, ir tai priklauso nuo kiekvieno individualaus instrumento kokybinių charakteristikų (pagaminimo medžiagos,

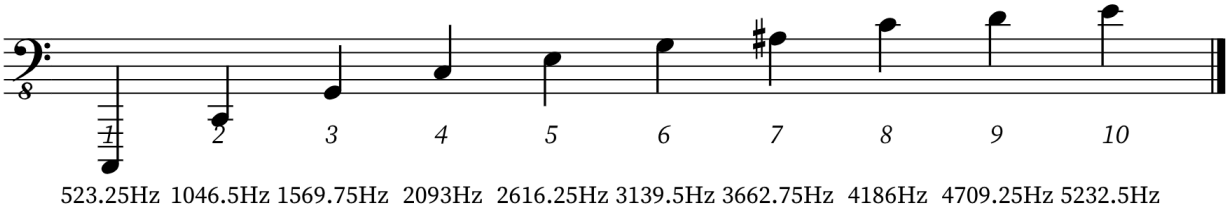
surinkimo kokybės, grojimo stiliaus, temperatūros ir panašiai). Dėl to net jei imtume du identiškus muzikos instrumentus, kurie groja tą pačia natą, jų harmonikos nebūtinai būtų vienodos. Bazinė nata bus vienoda, tačiau harmonikos - ne. Būtent harmonikos ir duoda kiekvienam muzikos instrumentui jam būdingas unikalias charakteristikas.

Jei iš viso dažnių spektro išmestume visas harmonikas ir obertonus ir paliktume tik pirmąją natą (harmoniką), gautume labai sintetinį/kompiuterinį garsą - visi instrumentai skambėtų beveik identiškai.

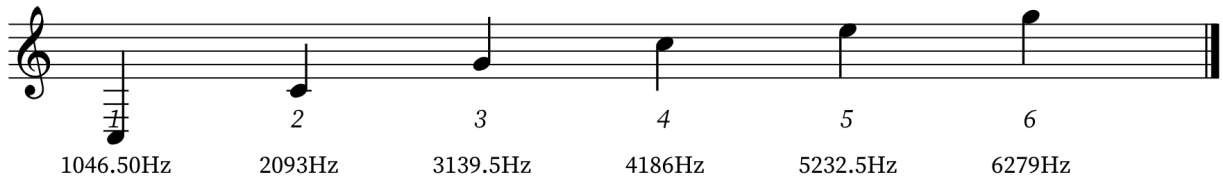
Įdomumo dėlei gautas harmonikas sudėliojau į penklinę, kad vizualiai matytųsi, kokios natos suskamba, grojant bazinę natą. Rezultatai pateikti 10, 11 ir 12 paveiksluose.



Pav 10. C4 natos generuojamos harmonikos
(natos yra pažemintos 3'imis oktavomis skaitomumo pagerinimui)



Pav 11. C5 natos generuojamos harmonikos
(natos yra pažemintos 4'imis oktavomis skaitomumo pagerinimui)



Pav 12. C6 natos generuojamos harmonikos
(natos yra pažemintos 3'imis oktavomis skaitomumo pagerinimui)

Išvados

Laboratorinio darbo metu išanalizuota 12 garso signalų ir pateikta jų dažnių spekrinės charakteristikos. Buvo labai įdomu sužinoti, kad skambant vienai natai iš tikro girdime net kelias aukštesnes bei tylesnes natas kartu. Šito fenomeno be spektrinės analizės aptikti būtų neįmanoma.

Taip pat iš analizės matome, kad bazinės natos dažnių kartotinės sudaro muzikos instrumento harmoniką ir kiekviena iš jų yra nutolusios tuo pačiu atstumu. Dedamosios, nutolusios 2^n atsrumu nuo bazinės natos yra vadinamos obertonais, ir yra oktava aukščiau nuo prieš tai buvusio obertono.

Kiekvieno instrumento harmonikų amplitudės bei jų kiekis yra skirtingas. Būtent tai ir priduoda kiekvienam muzikos instrumentui jam būdingas garso bruožus.

Programos kodas

Main.py

```
from FFT import FFT
from WAV import WAV

import Utils

if __name__ == "__main__":
    wav = WAV(Utils.selectFile())

    # Draw waveform so the user can select time
    wav.plotSelf()

    start_time = Utils.promptForCTI(wav.getDuration())
    end_time = start_time + Utils.promptForDuration()

    # Draw waveform with highlighted user range
    wav.plotSelf(segments=[(start_time, end_time)])

    fft = FFT(
        wav.sampleRate,
        wav.normalize(wav.getSamplesInTimeRange(start_time, end_time)),
        wav.name + ', time: {0} - {1}'.format(start_time, end_time)
    )

    # Draw samples that were fed into FFT
    wav.plot(fft.samples, start_time, end_time)

    # Draw final FFT data
    fft.plot()
```

FFT.py

```
import matplotlib.pyplot as plt
import numpy as np

class FFT:
    def __init__(self, sampleRate, samples, title):
        self.sampleRate = sampleRate
        self.samples = samples
        self.title = title
        self.fft = self._process()

    def plot(self):
      time_axis = np.linspace(0, self.sampleRate / 2, len(self.fft))

      plt.figure(figsize=(10, 4))
      plt.plot(time_axis, self.fft)
      plt.grid()
      plt.title(self.title)
      plt.xlabel('Frequency')
      plt.show()

    def _process(self):
        count = len(self.samples)
        samples = self.samples * np.hanning(count)

        fft = np.fft.fft(samples)
        fft = np.abs(fft)
```

```
        return fft[0:round(count / 2)]
```

Utils.py

```
from tkinter import filedialog
import datetime
import sys

def generateMMSSms():
    time = datetime.datetime.now()
    minutes = time.minute
    seconds = time.second
    milliseconds = time.microsecond // 1000  # Convert microseconds to milliseconds
    return f"{minutes}{seconds}{milliseconds}"

def promptForCTI(maxDuration: float):
    cti = maxDuration + 1
    while float(cti) > maxDuration:
        cti = input('> Enter start position in seconds (max: {0}): '.format(maxDuration))
    return float(cti)

def promptForDuration():
    default_duration = 30 / 1000
    user_input = input('> Enter duration in seconds (defaults to {0} ms):
'.format(default_duration))

    try:
        duration = float(user_input)
    except ValueError:
        duration = default_duration

    return duration

def selectFile():
    filePath = filedialog.askopenfilename(filetypes=[('Audio file', '*.wav')])
    if not filePath:
        print("No file selected. Aborting.")
        sys.exit(1)
    return filePath
```

WAV.py

```
from playsound import playsound

import matplotlib.pyplot as plt
import numpy as np
import os
import wave

import Utils

class WAV:
    def __init__(self, filePath):
        file = wave.open(filePath, 'rb')

        self.bitsPerSample = file.getsampwidth() * 8
        self.compressionName = file.getcompname()
        self.compressionType = file.getcomptype()
        self.filePath = filePath
        self.name = os.path.basename(filePath)
```

```
        self.numChannels = file.getnchannels()
        self.numSamples = file.getnframes()
        self.sampleRate = file.getframerate()
        self.samples = np.frombuffer(
            file.readframes(file.getnframes()),
            np.int16
        )

        self.frameDuration = 25 # in milliseconds

        file.close()

    def getAmplitudeMax(self) -> float:
        return self.samples.max()

    def getAmplitudeMin(self) -> float:
        return self.samples.min()

    def getBitrate(self) -> float:
        return self.sampleRate * self.numChannels * self.bitsPerSample

    def getChannelWidth(self) -> float:
        return 2 ** self.bitsPerSample

    def getChannelWidthMax(self) -> float:
        return self.getChannelWidth() / 2 - 1

    def getChannelWidthMin(self) -> float:
        return - self.getChannelWidth() / 2

    def getDuration(self) -> float:
        return self.numSamples / self.sampleRate

    def getEnergy(self):
        return self.getFrameFeatures(self.signalToEnergy)

    def getFileSizeComputed(self) -> float:
        return self.numSamples * self.numChannels * self.bitsPerSample / 8

    def getFrameFeatures(self, feature_extractor):
        window_size, hop_size = self.parseFrameDuration()

        samples = self.normalize(self.toMono())

        features = []
        for i in range(0, len(samples) - window_size + 1, hop_size):
            signal = samples[i:i + window_size]
            result = feature_extractor(signal)
            features.append(result)

        return self.normalize(np.array(features))

    def getSamplesForChannel(self, channelIndex):
        return self.samples[channelIndex::self.numChannels]

    def getSamplesInTimeRange(self, start_time, end_time):
        start_index = int(self.sampleRate * start_time)
        end_index = int(self.sampleRate * end_time)
        return self.samples[start_index:end_index]

    def getSegments(self, data, threshold):
        window_size, hop_size = self.parseFrameDuration()
        segments = []
        segment_start = None
```

```
    step = (window_size - hop_size) / self.sampleRate
    for i, e in enumerate(data):
        if e > threshold:
            if segment_start is None:
                segment_start = i * step
        elif segment_start is not None:
            segment_end = i * step
            segments.append((segment_start, segment_end))
            segment_start = None

    return segments

def getZCR(self):
    return self.getFrameFeatures(self.signalToZCR)

def normalize(self, np_array):
    return np_array / np_array.max()

def parseFrameDuration(self):
    window_size = int(self.frameDuration * self.sampleRate / 1000)
    hop_size = window_size // 2

    return window_size, hop_size

def play(self):
    playsound(self.filePath)

def plot(self, data, start_time=0, end_time=None, label = "", title = "", segments=[]):
    if end_time is None:
        end_time = self.getDuration()

    time_axis = np.linspace(start_time, end_time, len(data))

    plt.figure(figsize=(10, 4))
    plt.title(title)
    plt.plot(time_axis, data, linewidth=1)
    plt.xlabel('Time (s)')
    plt.ylabel(label)
    plt.grid()
    for start, end in segments:
        plt.axvspan(start, end, color='red', alpha=0.5)
    plt.show()

def plotSelf(self, cti: float = -1, segments = []):
    timeAxis = np.linspace(0, self.getDuration(), self.numSamples)

    plt.figure(figsize=(10, 4))

    for channelIndex in range(self.numChannels):
        plt.subplot(self.numChannels, 1, channelIndex + 1)

        plt.grid(True)
        plt.plot(timeAxis, self.getSamplesForChannel(channelIndex))
        plt.title(f'{self.name} (channel {channelIndex + 1})')
        plt.xlabel('Time (s)')
        plt.ylabel('Amplitude')
        # plt.ylim(self.getChannelWidthMin(), self.getChannelWidthMax())

        if cti > -1:
            plt.axvline(x=cti, color='r')

        for start, end in segments:
            plt.axvspan(start, end, color='red', alpha=0.5)
```

```
    plt.tight_layout()
    plt.show()

def save(self, name=None):
    if name is None:
        name = self.name + "-" + Utils.generateMMSSms()

    file = wave.open(name, 'wb')
    file.setnchannels(self.numChannels)
    file.setsampwidth(int(self.bitsPerSample / 8))
    file.setframerate(self.sampleRate)
    file.setnframes(int(self.numSamples))
    file.writeframes(self.samples.tobytes())
    file.close()

    return name

def setFrameDuration(self, frameDuration):
    self.frameDuration = frameDuration

def signalToEnergy(self, signal):
    return np.sum(np.square(signal))

def signalToZCR(self, signal):
    return np.sum(np.abs(np.diff(np.sign(signal)))) / (2 * len(signal))

def toMono(self):
    samples = self.samples
    if self.numChannels == 2:
        left = self.getSamplesForChannel(0)
        right = self.getSamplesForChannel(1)
        samples = (left + right) // 2

    return samples

def toStereo(self, offset_in_ms: int = 10):
    if self.numChannels != 1:
        print("Cannot convert non-mono track into stereo")
        exit(1)

    offset_in_frames = int(self.sampleRate * offset_in_ms / 1000)

    print(self.samples.dtype)
    self.numChannels = 2
    interleaved = np.empty(self.numSamples * self.numChannels, dtype=np.int16)
    interleaved[0::2] = self.samples
    interleaved[1::2] = np.roll(self.samples, offset_in_frames)
    self.samples = interleaved

def toString(self) -> str:
    return (
        f"bitRate: {self.getBitrate()} bits per second\n"
        f"bitsPerSample: {self.bitsPerSample} bits\n"
        f"compressionName: {self.compressionName}\n"
        f"compressionType: {self.compressionType}\n"
        f"duration: {self.getDuration()} seconds\n"
        f"filePath: {self.filePath}\n"
        f"fileSizeComputed: {self.getFileSizeComputed()} bytes\n"
        f"name: {self.name}\n"
        f"numChannels: {self.numChannels}\n"
        f"numSamples: {self.numSamples}\n"
        f"sampleRate: {self.sampleRate} Hz\n"
        f"samples: {self.samples}"
    )
```