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

GARSO SIGNALŲ APDOROJIMAS

Penktasis laboratorinis darbas

Garso signalų apdorojimas dažnių srityje

Darbą atliko 4 kurso ISI 2 grupės studentas

Tomas Šinkūnas

Vilnius, 2023

Turinys

Įvadas

Darbo tikslas

Dažninė garso signalų analizė ir apdorojimas.

Darbo užduotis

Sukurti priemonę garso signalams ir jų spektrams grafiškai atvaizduoti. Atlikti signalo apdorojimą dažnių srityje, įvertinti matomas (ir girdimas) signalo savybes prieš ir po apdorojimo.

Darbo priemonės

Darbui atlikti buvo naudojama *Python* programavimo kalba su *matplotlib*, *numpy* ir *wave* bibliotekomis. Programinis kodas tęsiamas nuo praeito darbo.

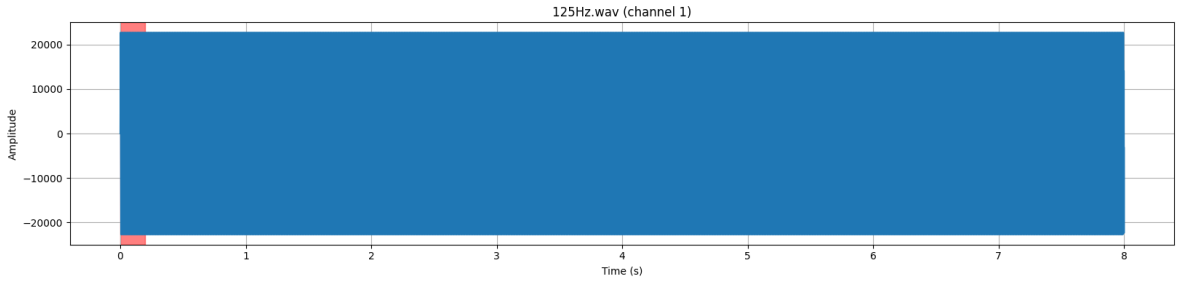
Duomenys

Laboratoriniame darbe yra apdorojamas 1 garso failas: Sinus_125Hz.wav

Darbo eiga

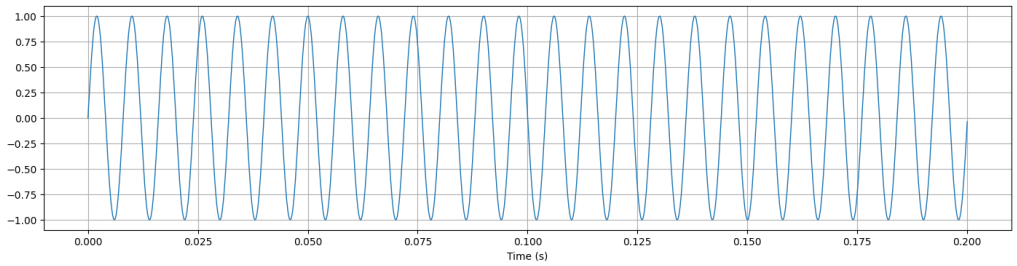
Paleidus programą iš karto pateikiama pasirinkto failo laiko diagrama, kurioje vartotojas gali nuspręsti, kurią atkarpą analizuoti/apdoroti. Uždarius diagramos langą programa prašo įvesti signalo pradžios laiką bei trukmę (sekundėmis). *Pirminė laiko diagrama bei nustatymų įvedimo langai laboratoriniame darbe nėra pateikti.*

Sekančiame žingsnyje programa pateikia viso audio signalo laiko diagramą, tačiau su pažymėta/išskirta vieta, kuri bus analizuojama (paveikslas 1).



Pav. 1. Signalo laiko diagrama su pasirinkta atkarpa

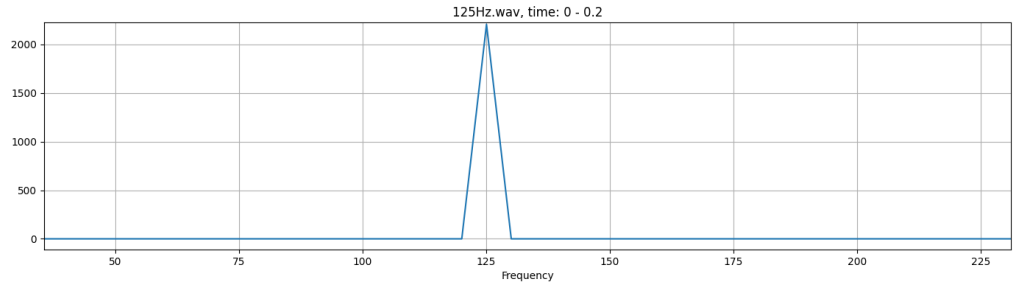
Uždarius diagramos langą yra pateikiama išskirta (vartotojo pasirinkta) 200ms trukmės atkarpa (paveikslas 2).



Pav. 2. Signalo atkarpos laiko diagrama

Nagrinėjamai signalo atkarpai *Hanning* lango funkcija nebuvo pritaikyta, nes analizuojamas sintetinis signalas, sugeneruotas kompiuterio. Dėl to iš lango funkcijos jokios naudos nebūtų.

Apskaičiavę Furje transformaciją matome, kad signale vyrauja vienintelis 125Hz dažnis (paveikslas 3)

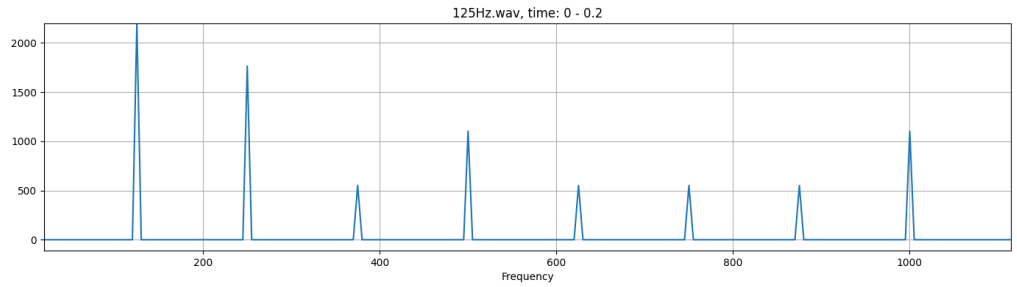


Pav. 3. Signalo spektas

Laboratorinio darbo eigoje nusprendžiau modifikuoti signalą pridedant papildomų dažnių su skirtingomis amplitudėtims:

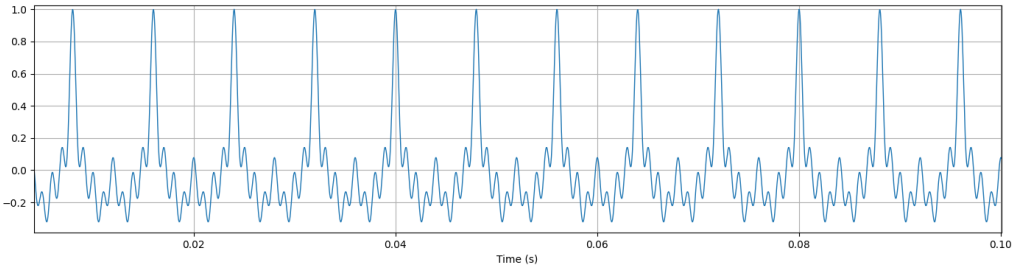
- 250Hz (pirmasis obertonas),
- 375Hz
- 500Hz (antrasis obertonas),
- 625Hz
- 750Hz
- 875Hz
- 1000Hz (trečiasis obertonas)

Gauti rezultatas pateiktas paveiksle 4.



Pav. 4. Apdorotas signalo spektas

Sekančiame etape iš apdoroto spektro buvo atstatytas signalas (paveikslas 5).



Pav. 5. Atstatytas signalas (apkarpytas)

Išvados

Laboratorinio darbo metu buvo išanalizuotas garso signalas, modifikuotos jo dažninės charakteristijos bei atstatyta apdoroto signalo laiko diagrama. Buvo įdomu sužinoti, kaip pasikeis garso signalas pridėjus 3 papildomus obertonus bei tarpus užpildžius pagrindinio signalo kartotinėmis.

Iš laiko diagramos, pateiktos paveiksle 5 matosi, jog sėkmingai pavyko modifikuoti signalo dažnių spektrą - laiko diagramoje atsirado “naujų bangelių”. Taip pat matosi, jog kiekvienos bangelės pikai yra monotoniškai nutolę vieni nuo kitų. Tai yra dėl to, jog bazinė dažnio dedamoji buvo kartojama kelis kartus, o papildomų atsitiktinių dažnių į signalą nebuvo pridėta.

Girdimas garso signalas taip pat pasikeitė - monotoniškas sinusoidinis garsas pasikeitė tiek tembru, tiek diapazonu. Rezultate gautas garsas yra “sudėtingesnis”, nes girdimos net kelios harmonikos vienu metu.

Programos kodas

Main.py

```
from FFT import FFT
from WAV import WAV

import Utils

if __name__ == "__main__":
    wav = WAV("../sounds/Sinus_125Hz.wav")

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

    start_time = Utils.promptForCTI(wav.getDuration())
    end_time = start_time + Utils.promptForDuration(200)

    # 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)

    # Plot FFT before adding new frequencies
    fft.plot()

    # Add some frequencies
    fft.transform(125)

    # Plot final FFT
    fft.plot()

    # Plot restored signal
    wav.plot(wav.normalize(fft.restore()), start_time, end_time)

    # Save signal to file
    wav.samples = fft.restoreSamples()
    wav.save("restored_audio.wav")
```

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()
        self.phase_spectrum = np.angle(self.fft)

    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.hamming(count)

        fft = np.fft.rfft(self.samples)
        fft = np.abs(fft)

        return fft

    def addFrequency(self, frequency, amp_multiplier=1):
        index = int(frequency * len(self.fft) * 2 / self.sampleRate)
        self.fft[index] = amp_multiplier * max(self.fft)

    def restore(self):
        return np.fft.irfft(self.fft * np.exp(1j * self.phase_spectrum), n=len(self.samples))

    def restoreSamples(self):
        return np.int16(self.restore() * self.sampleRate)

    def transform(self, base_freq):
        self.addFrequency(base_freq * 2, 0.8) #
        self.addFrequency(base_freq * 3, 0.25)
        self.addFrequency(base_freq * 4, 0.5) #
        self.addFrequency(base_freq * 5, 0.25)
        self.addFrequency(base_freq * 6, 0.25)
        self.addFrequency(base_freq * 7, 0.25)
        self.addFrequency(base_freq * 8, 0.5) #
```

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(duration=30):
    default_duration = duration / 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}"
    )
```