Projekt Sygnały Akustyczne

Temat: “Mobilna aplikacja do dekodowania kodu Morse’a w sygnale akustycznym”.

Twórcy: Michał Okoń, Marcin Oleszczuk

Prowadzący: dr inż. Mirosław Łazoryszczak, dr inż. Tomasz Mąka

1. **Cel Projektu**
   Celem projektu było wytworzenie aplikacji w środowisku android pozwalającej na odczytanie i przetłumaczenie sygnału morsa.
2. **Przeprowadzone działania**
   Aby zrealizować cele zadania, w początkowej fazie projektu, utworzono odpowiednie repozytorium na platformie github, następnie zainicjowano project i rozpoczęto wstępną implementacje programu.

   2.1 Generowanie sygnału morsa z tekstu.
   W pierwszej iteracji program wykonana została część odpowiedzialna za generowanie sygnału morsa z dostarczonego przez interfejs użytkownika tekstu.

Jak można dostrzec bazowy interfejs użytkownika jest bardzo prosty, ze względu na to iż głównym założeniem projektu była część badawczo-programistyczna a nie stylistyczna. Wprowadzone

słowa w polu tekstowym a następne kliknięcie guzika “ROZPOCZNIJ” inicjuje procedure przetłumacznia słowa na sygnał morsa a następnie odebranie odpowiednich tonacji dźwięku.

```
private void PlayGeneratedPattern() {
    Log.i( tag: "Pattern: ",pattern);
    TextToMorseConverter.SetUpEverything();
    TextToMorseConverter.GenerateSoundWave();
    TextToMorseConverter.GenerateSoundWaveLine();
    pattern=pattern.toUpperCase();
    String morsePat = TextToMorseConverter.ConvertPatternToMorsePattern(pattern);
    TextToMorseConverter.PlayPattern(morsePat);

}
private void stopRecording(){
```

Dostarczony wzorzec słowny przekazywany jest do odpowiedniego konwertera.

```
public static void PlayPattern(String morsePattern) {
    Log.i( tag: "pattern morse",morsePattern);


    audioTrack.play();
    for (int i = 0; i < morsePattern.length(); i++) {
        if (MainActivity.isThreadWorking == false) break;


        if (morsePattern.charAt(i) == '.') {

            //play znak
            PlayDot();
            //if (isStillSameCharacter(morsePattern, i) == 1) continue;
            PlaySilence();

        }
        if (morsePattern.charAt(i) == '-') {
            PlayLine();
            //if (isStillSameCharacter(morsePattern, i) == 1) continue;
            PlaySilence();

        }
        if (morsePattern.charAt(i) == ' ') {
            PlaySilenceBetweenChars();

        }
        if (morsePattern.charAt(i) == '[') {
            PlaySilenceBetweenWords();

        }


    }
    MainActivity.isThreadWorking = false;

}
```

Przekazany tekst rozbijany jest na pojedyńcze znaki, które po wcześniejszym przetłumaczeniu na sekwencje kropek oraz kresek są rozpoznawane a natstępnie dla każdego rozpoznanego znaku generowana jest odpowiednia tonacja sygnału.

```
    private static void PlayDot() {

        audioTrack.write(samples,  offsetInShorts: 0, numofSamples);

    }

    private static void PlayLine() {

        audioTrack.write(samplesLine,  offsetInShorts: 0,  sizeInShorts: numofSamples * 3);


    }

    private static void PlaySilence() {

        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
    }


    private static void PlaySilenceBetweenChars() {

        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);


    }
    private static void PlaySilenceBetweenWords(){

        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);
        audioTrack.write(silenceTab,  offsetInShorts: 0, numofSamples);


    }
```

Zgodnie z założeniami kodu morsa po wyznaczeniu czasu trwania kropki (.), kreska posiada czas trwania trzech kropek, cisza także o długości jednej kropki, przerwa pomiedzy znakami jest sekwencją trzech kropek a cisza pomiędzy słowami to siedem kropek.

2.2 Odczytywanie sygnału morsa z audio.

Kolejną i główną funkcjonalnością aplikacji jest odczytywanie sygnału audio, odnalezienie w nim sygnału morsa i odpowiednie przetłumaczenie go. Ta część projektu realizowana jest w dwojaki sposób. Domyślnie aplikacja miała działać na zasadzie utworzenia potoku (streamu) audio nagrania dźwięków sygnału morsa i ewentualnych zakłóceń a następnie odpowiednie przerobienie danych audio. Jednak ze względu na liczne problem z obsługą nagrywania streamu audio oraz przez to możliwości tłumaczenia z sygnału na tekst w czasie rzeczywistym zrezygnowano z tego pomysłu. Został on zastąpiony wczytywaniem wskazanego pliku .wav przed włączeniem aplikacji. Aby skorzystać z tłumaczenia audio na tekst użytkownik w menu głównym musi skorzystać z opcji “GŁOSOWY”.

Po kliknięciu wspomnianego wyżej guzika użytkownik przekierowany zostaje do nowego widoku odpowiedzialnego za dostarczenie opcji zajmujących się obróbką pliku audio.

Początkowo użytkownik może dostrzec puste pole oraz dwa nowe guziki. W pustym polu po wybraniu odpowiednich opcji pojawią się odpowiednie wykresy I tłumaczenie, bądź włączone zostaną inne opcje.

2.2.1 Average demorse.

Przycisk “Average Demorse” inicjuje operacje tłumaczenia morsa z sygnału audio w tak zwanych warunkach idelanych ze względu na to, iż nie posiada on bardziej zaawansowanych opcji odfiltorwania sygnału morsa w sytuacji pojawienia się szumów.

```
InputStream inp = getAssets().open( fileName: "sos.wav");
File file = stream2file(inp);
MorseToTextConverter morseToTextConverter = new MorseToTextConverter(file.toString());
int defaultSampleRate = -1;        //-1 value implies the method to use default sample rate
int defaultAudioDuration = -1;    //-1 value implies the method to process complete audio duration
JLibrosa jLibrosa = new JLibrosa();
float[] audioFeatureValues = jLibrosa.loadAndRead(file.toString(), defaultSampleRate, defaultAudioDuration);
double[] audioValues = convertFloatsToDoubles(audioFeatureValues);
```

W pierwszym kroku wczytywany jest plik sygnału w formacie .wav, następnie przekazywany jest on do konstruktora klasy odpowiadającej za tłumaczenie sygnału morsa na tekst.

```java
public MorseToTextConverter(String filename) throws WavFileException, IOException {
    this.wavFile = WavFile.openWavFile(new File(filename));
    this.numChannels = this.wavFile.getNumChannels();
    this.checkStateInterval = 10;
    this.minSampleValue = 0.01;
    this.sampleRate = ((int) wavFile.getSampleRate() / 1000) * checkStateInterval;
    this.buffer = new double[this.sampleRate * this.numChannels];
}
```

Po przekazaniu pliku a tak właściwie ścieżki do niego do konsturktora, odpowiednia klasa wczytuje pliko jako plik .wav oraz wykorzystując możliwości oferowane przez zaimplementowaną klasę WavFile.

```java
public class WavFile {
    private enum IOState {READING, WRITING, CLOSED}

    private final static int BUFFER_SIZE = 4096;

    private final static int FMT_CHUNK_ID = 0x20746D66;
    private final static int DATA_CHUNK_ID = 0x61746164;
    private final static int RIFF_CHUNK_ID = 0x46464952;
    private final static int RIFF_TYPE_ID = 0x45564157;

    private File file;                    // File that will be read from or written to
    private IOState ioState;              // Specifies the IO State of the Wav File (used for snaity checking)
    private int bytesPerSample;         // Number of bytes required to store a single sample
    private long numFrames;               // Number of frames within the data section
    private FileOutputStream oStream;   // Output stream used for writting data
    private FileInputStream iStream;      // Input stream used for reading data
    private double floatScale;            // Scaling factor used for int <-> float conversion
    private double floatOffset;         // Offset factor used for int <-> float conversion
    private boolean wordAlignAdjust;      // Specify if an extra byte at the end of the data chunk is required for word alignment

    // Wav Header
    private int numChannels;              // 2 bytes unsigned, 0x0001 (1) to 0xFFFF (65,535)
    private long sampleRate;              // 4 bytes unsigned, 0x00000001 (1) to 0xFFFFFFFF (4,294,967,295)
    // Although a java int is 4 bytes, it is signed, so need to use a long
    private int blockAlign;               // 2 bytes unsigned, 0x0001 (1) to 0xFFFF (65,535)
    private int validBits;                // 2 bytes unsigned, 0x0002 (2) to 0xFFFF (65,535)

    // Buffering
    private byte[] buffer;                // Local buffer used for IO
    private int bufferPointer;            // Points to the current position in local buffer
    private int bytesRead;                // Bytes read after last read into local buffer
    private long frameCounter;            // Current number of frames read or written
```

Klasa ta oferuje większość operacji umożliwiających wczytywanie/zapis oraz obróbkę plików .wav. Pozwala on ana ekstrakcje z pliku .wav danych takich jak: liczba kanałów, częstotliwość próbkowania, licznik wczytanych bajtów itd.

Po wczytaniu pliku wav zgodnie z konstruktorem ustawiane są wszystkie potrzebne zmienne.

```
int defaultSampleRate = -1;        //-1 value implies the method to use default sample rate
int defaultAudioDuration = -1;    //-1 value implies the method to process complete audio duration
JLibrosa jLibrosa = new JLibrosa();
float[] audioFeatureValues = jLibrosa.loadAndRead(file.toString(), defaultSampleRate, defaultAudioDuration);
double[] audioValues = convertFloatsToDoubles(audioFeatureValues);

AnyChartView anyChartView = (AnyChartView) findViewById(R.id.any_chart_view);
Cartesian cartesian = AnyChart.line();
cartesian.title("Original signal");
cartesian.yAxis( index: 0).title("Value");
cartesian.xAxis( index: 0).title("Sample");
List<DataEntry> seriesData = new ArrayList<>();
for (int i = 0; i < audioValues.length; i++) {
    seriesData.add(new CustomDataEntry(i, audioValues[i]));
}
Set set = Set.instantiate();
set.data(seriesData);
Mapping series1Mapping = set.mapAs( mapping: "{ x: 'x', value: 'value' }");
Line series1 = cartesian.line(series1Mapping);
series1.hovered().markers().enabled(true);
series1.hovered().markers()
        .type(MarkerType.CIRCLE)
        .size(4d);
series1.tooltip()
        .position("right")
        .anchor(Anchor.LEFT_CENTER)
        .offsetX(5d)
        .offsetY(5d);
anyChartView.setChart(cartesian);

morseToTextConverter.executeTranslation();
TextView textView = findViewById(R.id.decodedTextField);
textView.setText(morseToTextConverter.result());
Log.i( tag: "File info: ", morseToTextConverter.toString());
inp.close();
```

Następnym krokiem jest odpowiednie wczytanie danych pliku .wav I zaprezentowanie sygnału na wykresie, po przedstawieniu sygnału na wykresie, rozpoczyna się process tłumaczenia sygnału na tekst.

```java
public void executeTranslation() throws WavFileException, IOException {
    MorseCodeDict morseCodeDictionary = new MorseCodeDict();
    List<MorseSignal> allAvailableSignals = this.getSignals();
    List<String> morseMessageAssembled = this.assemble(allAvailableSignals);

    Iterator<String> wordIterator = morseMessageAssembled.iterator();
    String originalMorseMessage = "";
    String translatedMorseMessage = "";

    while (wordIterator.hasNext()) {

        String morseMessageWord = wordIterator.next();

        if (morseMessageWord.equals("LETTER_SPACE")) {

            originalMorseMessage += " ";
            translatedMorseMessage += "";

        } else if (morseMessageWord.equals("WORD_SPACE")) {

            originalMorseMessage += "   ";
            translatedMorseMessage += " ";

        } else {
            originalMorseMessage += morseMessageWord;
            translatedMorseMessage += morseCodeDictionary.translate(morseMessageWord);
        }

    }

    System.out.println("MORSE MESSAGE: " + originalMorseMessage);
    System.out.println("TRANSLATED MESSAGE: " + translatedMorseMessage);

    this.decodedMorseResult = translatedMorseMessage;

//      Close wav file
    this.close();
}
```

Funkcja execute translation jak sama nazwa wskazuje odpowiada za process tłumaczenia sygnału audio na tekst.

W pierwszej kolejności wytwarzany jest odpowiedni słownik odpowiedzialny za zamianę znaków sygnału morsa na litery alfabetu łacińskiego.

```
public MorseCodeDict() {
    this.dictionary.put(".-", "A");
    this.dictionary.put("-...", "B");
    this.dictionary.put("-.-.", "C");
    this.dictionary.put("-..", "D");
    this.dictionary.put(".", "E");
    this.dictionary.put("..-.", "F");
    this.dictionary.put("--.", "G");
    this.dictionary.put("....", "H");
    this.dictionary.put("..", "I");
    this.dictionary.put(".---", "J");
```

Kolejnym krokiem algorytmu jest odczytanie po kawałku sygnału z pliku .wav.

```
public List<MorseSignal> getSignals() throws WavFileException, IOException {
    List<MorseSignal> availableSignals = new ArrayList<>();
    boolean silenceInTheBeginning = false;

    while (this.wavFile.getFramesRemaining() > 0) {
        MorseSignal nextSignal = this.readNextSignal();
        if (!silenceInTheBeginning && nextSignal.silence && availableSignals.isEmpty()) {
            silenceInTheBeginning = true;
            nextSignal.length = 1;
            continue;
        }
        availableSignals.add(nextSignal);
    }
    return availableSignals;
}
```

Dopóki nie osiągnięto końca pliku (istnieją klatki do wczytania). Pobierz fragment pliku audio (o wielkości wyznaczonego wcześniej bufora)

```java
public MorseSignal readNextSignal() throws WavFileException, IOException {
    MorseSignal signal = new MorseSignal(this.checkStateInterval);
    signal.startFrame = this.wavFile.getFrameAlreadyRead();

    float sample = this.readBufferAudioSample();

    signal.length++; //1 = 10ms one unit of a signal

    //  this threshold let us know if it is a valid signal or silence
    if (sample > minSampleValue) {
        signal.signal = true;
    } else {
        signal.silence = true;
    }
```

```java
private float readBufferAudioSample() throws WavFileException, IOException {
    if (this.wavFile.getFramesRemaining() > 0) {
        List<Float> validSamples = new ArrayList<>();
        this.framesRead = wavFile.readFrames(this.buffer, this.sampleRate);
        if (this.framesRead != 0) {
            for (int s = 0; s < this.framesRead * this.numChannels; s++) {
                if (this.buffer[s] > 0) {
                    validSamples.add((float) this.buffer[s]);
                }
            }
            if (validSamples.isEmpty()) {
                validSamples.add((float) 0);
            }
            return this.getAverageValueInList(validSamples);
        }
    }

    return (float) 0;
}
```

Analizując pobrany fragment audio, wczytaj go I policz odpowiednią średnią z sygnału jeśli nie można policzyć to zwróć 0. Po wczytaniu średniej fragment sygnału, dokonywana jest decyzja czy wczytany sygnał jest ciszą w sygnale czy autentycznym sygnałem. Ta część algorytmu potwarzana jest do momentu wczytania wszystkich danych z pliku audio.

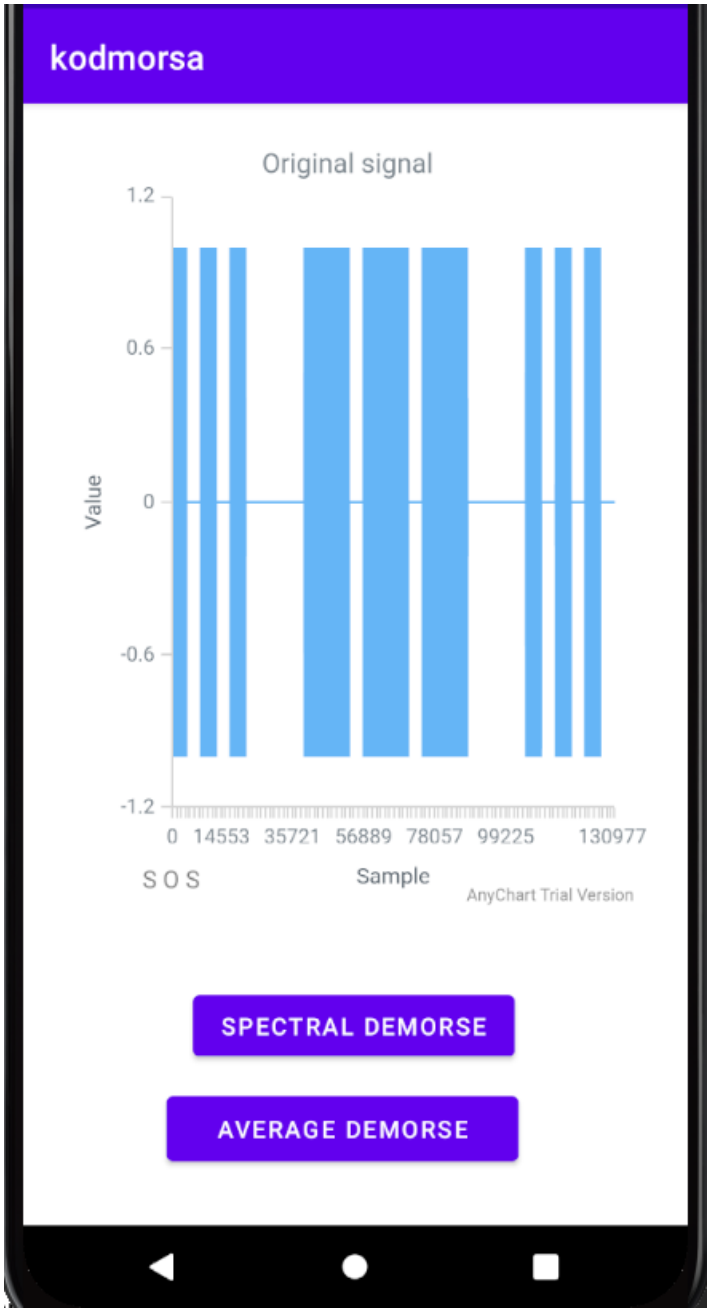
Po zakończonej operacji wczytywania fragmentów plików audio, trafiają one do funkcji “assemble” gdzie zostaną one odpowiednio przetłumaczone na zbiór kropek (.) oraz kresek (-) lub ewentualnych odstęp między literami oraz słowami.

```
            if (signal.signal) {
//              if value is signal and its average length is less than medium we know that is is a dot
                if (signal.length_ms() < this.mediumSignalValue) {
                    morseWord += ".";
//              if value is signal and its average length is higher than medium we know that is is a dash
                } else {
                    morseWord += "-";
                }

            } else {
                // check if is a silence between words
                if (signal.length_ms() >= (2 * this.mediumSilenceValue)) {
```

Ostatecznie znak po znaku sygnał jest tłumaczony na odpowiedni alfabet z zachowaniem odstępów miedzy słowami itd.

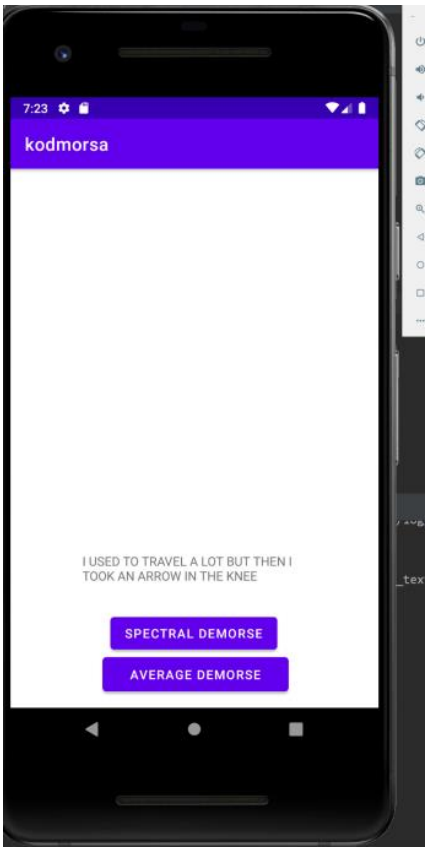
```
String originalMorseMessage = "";
String translatedMorseMessage = "";

while (wordIterator.hasNext()) {

    String morseMessageWord = wordIterator.next();

    if (morseMessageWord.equals("LETTER_SPACE")) {

        originalMorseMessage += " ";
        translatedMorseMessage += "";

    } else if (morseMessageWord.equals("WORD_SPACE")) {

        originalMorseMessage += "   ";
        translatedMorseMessage += " ";

    } else {
        originalMorseMessage += morseMessageWord;
        translatedMorseMessage += morseCodeDictionary.translate(morseMessageWord);
    }

}
```

Wynik tłumaczenia przypisywany jest do odpowiedniej zmiennej, która następnia dostępna jest z poziomu obiektu klasy za pomocą getera. Dodatkowo przetłumaczony tekst widoczny jest po stronie użytkownika zaraz pod wcześniej wspomnianym wykresie.

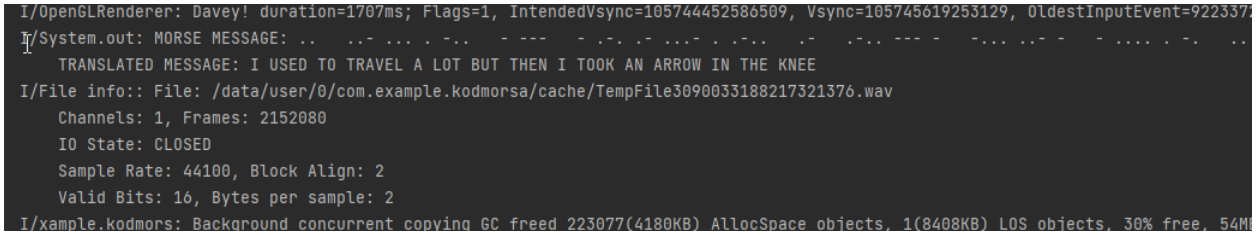


```
I/System.out: MORSE MESSAGE: ...   ---   ...
    TRANSLATED MESSAGE: S O S
I/File info:: File: /data/user/0/com.example.kodmorsa/cache/TempFile820672317030798145.wav
    Channels: 1, Frames: 132300
    IO State: CLOSED
    Sample Rate: 44100, Block Align: 2
    Valid Bits: 16, Bytes per sample: 2
```

Jak można zobaczyć na obrazach poniżej, tłumaczenie działa także dla dłuższych I złożonych zdań.

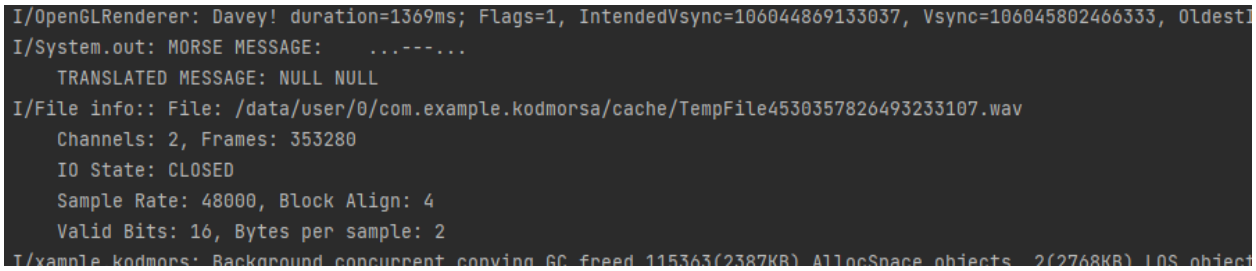



```
D/EGL_emulation: eglMakeCurrent: 0xeb64adc0: ver 3 0 (tinfo 0xbb9491d0) (first time)
I/System.out: MORSE MESSAGE: .--. .-.. . .- ... .   --. .. ...- .   -- .   ...--
    TRANSLATED MESSAGE: PLEASE GIVE ME 3
I/File info:: File: /data/user/0/com.example.kodmorsa/cache/TempFile85322742241792209O2.wav
    Channels: 1, Frames: 599759
    IO State: CLOSED
    Sample Rate: 44100, Block Align: 2
    Valid Bits: 16, Bytes per sample: 2
```

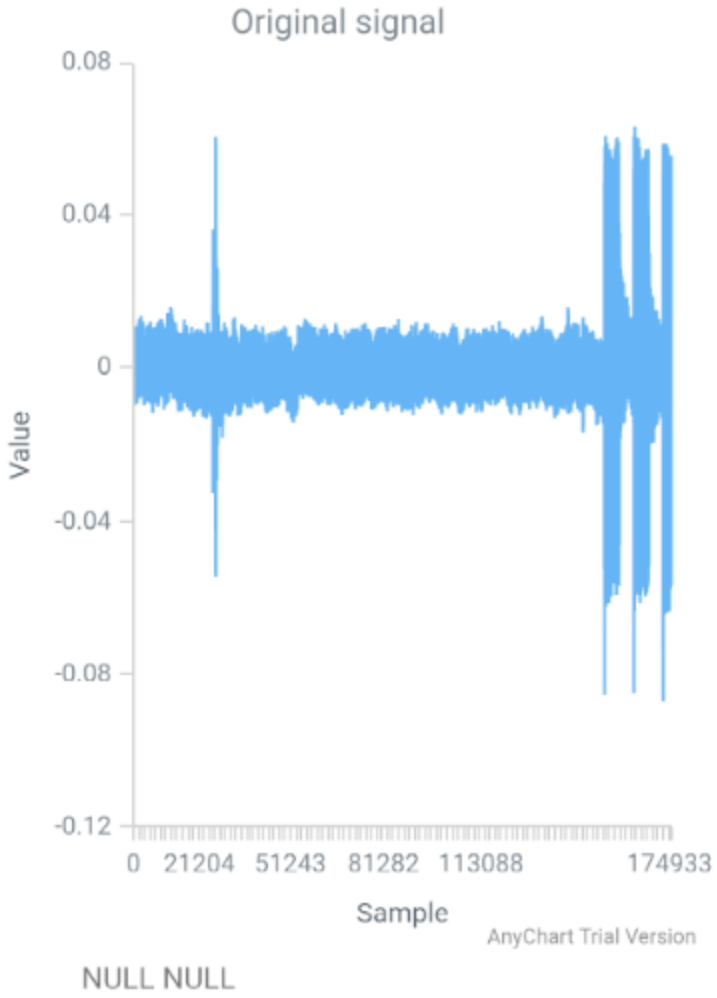


Ze względu na ilość danych do wykresów, android studio nie jest czasem w stanie odpowiednio zoptymalizować pamięci I nie pozwala na ich generacje przy dłuższych wiadomościach (zależne jest to też od dostępnych zasobów na komputerze).

Prawdziwy problem pojawia się jednak w plikach, w których obecne są szumy, zgodnie ze spostrzeżeniami prowadzącego, przy sygnałach z szumem algorytm będzie nie sprawny.

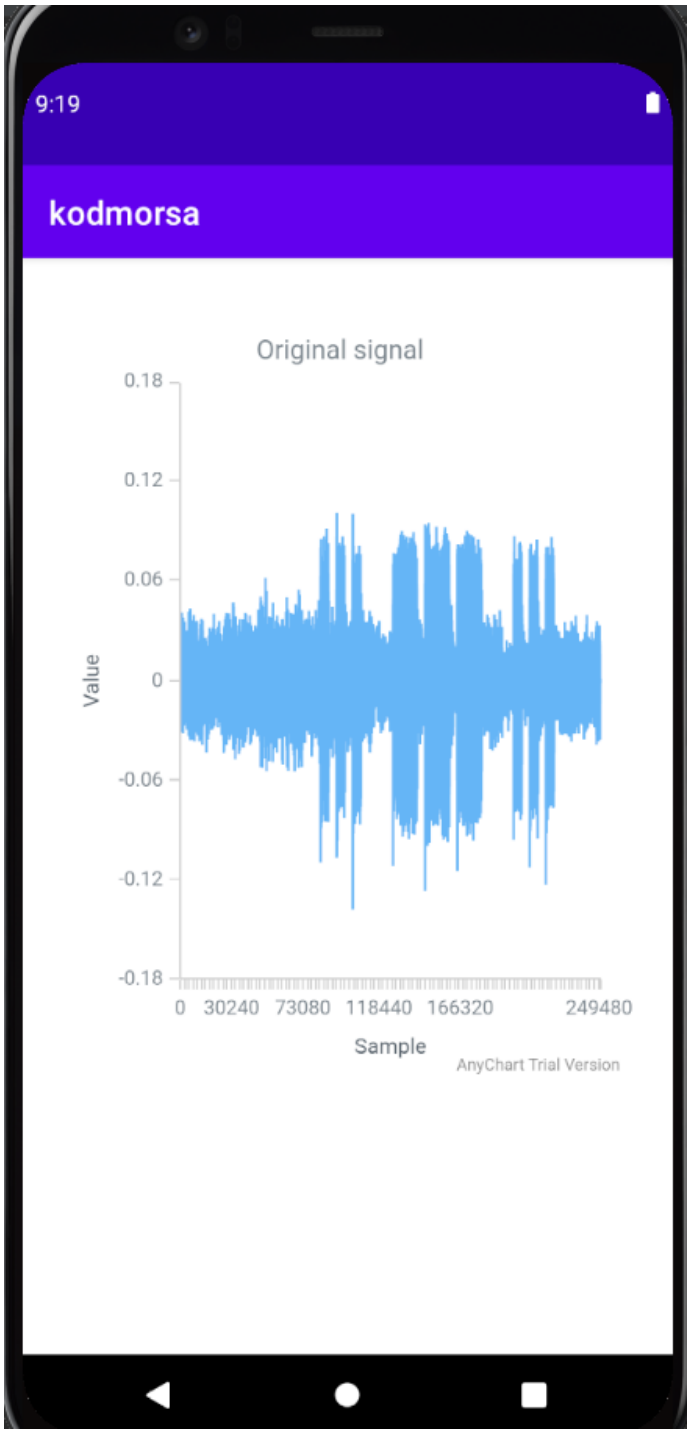


Przy plikach z drobnymi szumami można jednak dostrzec, że wiadomość w postaci kodu morsa została odseparowana prawie poprawnie (widoczne są kropki I kreski reprezentujące sygnał o tekście SOS, jednak ze względu na zaburzenie przed znakami I między nimi wkradły się puste element kompletnie zaburzające możliwość poprawnego przetłumaczenia danego tekstu.



2.2.2 Spectral demorse

Przycisk "Spectral Demorse" inicjuje rozpoczęcie procedury odczytania sygnału morsa nawet z zaszumionego audio. Proces ten podzielony jest na kilka wątków, które początkowo mają za zadanie odfiltrować częstotliwość sygnału morsa z zaszumionego audio.

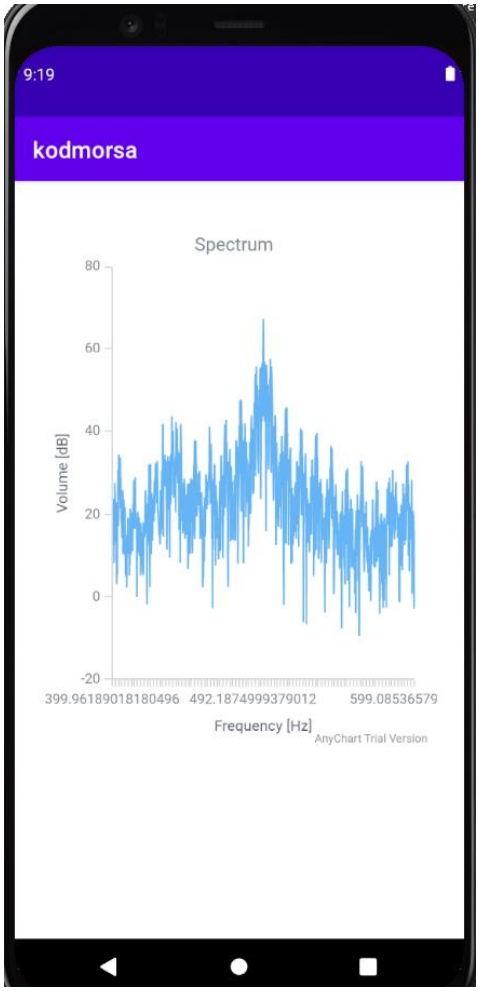


(W tym przykładzie na nagraniu obecny jest sygnał morsa nadający SOS, zmieszany z muzyką grającą z głośników).

```
InputStream inp = getAssets().open( fileName: "sos_noise_high.wav");
File file = stream2file(inp);
MorseToTextConverter morseToTextConverter = new MorseToTextConverter(file.toString());
inp.close();


int defaultSampleRate = -1;        //-1 value implies the method to use default sample rate
int defaultAudioDuration = -1;    //-1 value implies the method to process complete audio duration
JLibrosa jLibrosa = new JLibrosa();
float[] audioFeatureValues = jLibrosa.loadAndRead(file.toString(), defaultSampleRate, defaultAudioDuration);
double[] audioValues = convertFloatsToDoubles(audioFeatureValues);
```

W początkowej fazie, wczytywany jest wskazany plik .wav oraz jego dane. Kolejnym krokiem algorytmu jest wyznaczenie spektrum sygnału, w celu określenia charakterystyki częstotliwości.

```
/**
 * @param data    - Signal data
 * @param Fs      - Sampling frequency
 * @param start_f - start frequency to plot
 * @param stop_f  - stop frequency to plot
 * @param delta   - A point is considered a maximum peak if it has the maximal value, and was preceded (to the left) by a value lower by delta
 */
private void spectri(double[] data, double Fs, int start_f, int stop_f, double delta) {
      get data length
    int N = data.length;
      cast data to Complex (will be needed in FFT)
    Complex[] dataComplex = new Complex[data.length];
    for (int i = 0; i < data.length; i++) {
        dataComplex[i] = new Complex(data[i],  dImaginary: 0);
    }
      Calculate FFT
    Complex[] fftNew = FFT.fft1D(dataComplex);

    double df = Fs / N; // Frequency bin size
    double minf = -Fs / 2;
    double maxf = Fs / 2 - df;
    int i = (int) Math.round(N / 2 + (start_f * N / 2) / (Fs / 2)); // start index of frequency range
    int j = (int) Math.round(N / 2 + (stop_f * N / 2) / (Fs / 2)); // stop index of frequency range

    // Frequency axis
    int howMany = (int) Math.ceil((maxf - minf) / df);
    List<Double> f = new ArrayList<>(howMany);
    double k = minf;
    while (k <= maxf) {
        f.add(k);
        k += df;
    }
    //
      Calculate FFTShift
    Complex[] fftShift = FFT.fftShift1D(fftNew);
    List<Double> y = new ArrayList<Double>(fftNew.length);
    for (int l = 0; l < fftNew.length; l++) {
        double test = Math.abs(fftShift[l].Abs());
          dB magnitude
        y.add(20 * Math.log10(test));
    }
```



W prezentowanym spektrum można dostrzec dominującą częstotliwość, która jest poszukiwaną przez nas częstotliwością sygnału morsa. Oczywistym zatem jest, że kolejnym krokiem jest odszukanie odpowiednich pików częstotliwości. W tym celu wykorzystany został algorytm to detekcji pików w sygnale.

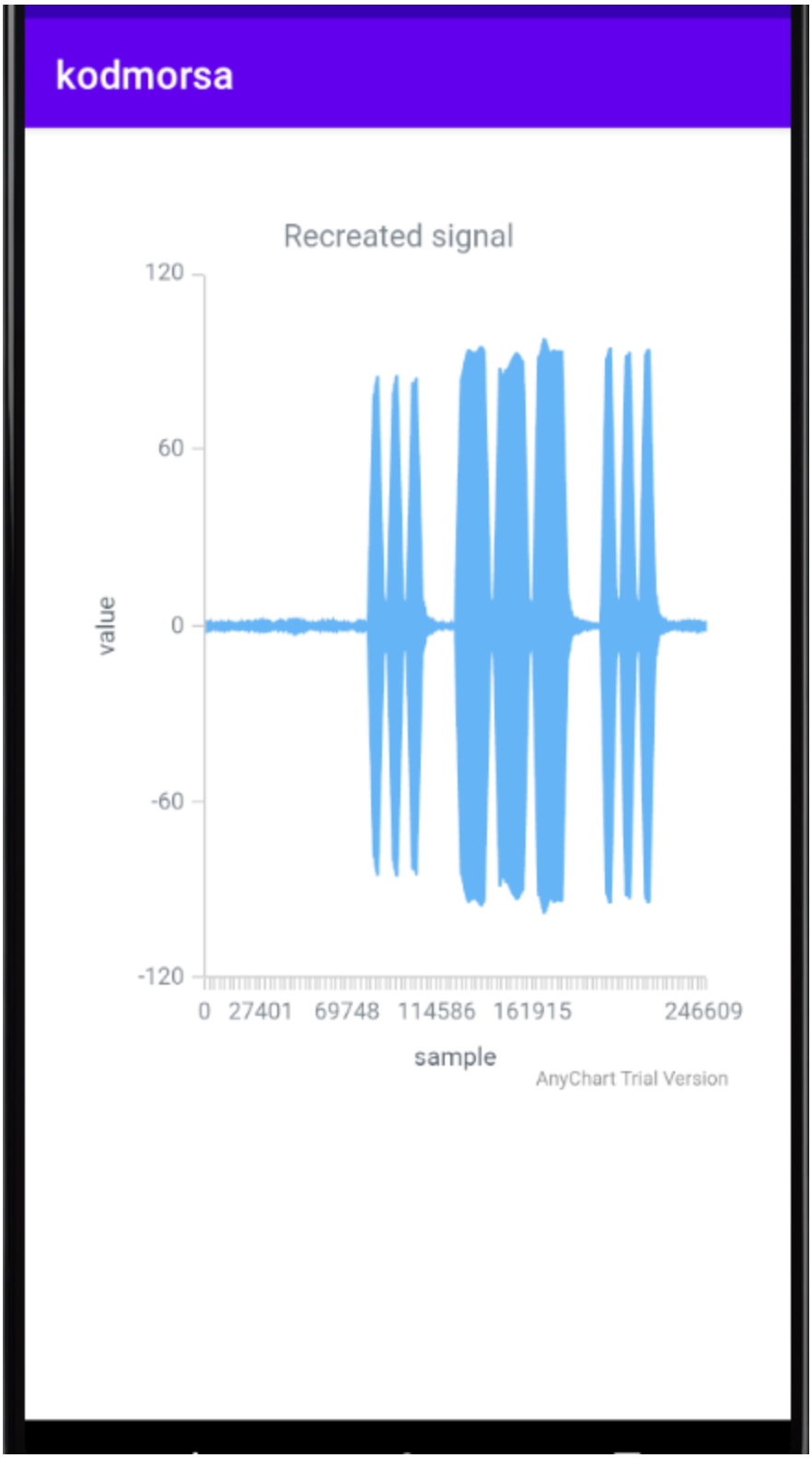
```java
        Map<U, Double> maxima = new HashMap<~>();
        Map<U, Double> minima = new HashMap<~>();
        List<Map<U, Double>> peaks = new ArrayList<~>();
        peaks.add(maxima);
        peaks.add(minima);

        Double maximum = null;
        Double minimum = null;
        U maximumPos = null;
        U minimumPos = null;

        boolean lookForMax = true;

//        int pos = 0;
        for (int i = 0; i < values.size(); i ++) {
            Double value = values.get(i);
            if (maximum == null || value > maximum) {
                maximum = value;
                maximumPos = indices.get(i);
            }

            if (minimum == null || value < minimum) {
                minimum = value;
                minimumPos = indices.get(i);
            }

            if (lookForMax) {
                if (value < maximum - delta) {
                    maxima.put(maximumPos, value);
                    minimum = value;
                    minimumPos = indices.get(i);
                    lookForMax = false;
                }
            } else {
                if (value > minimum + delta) {
                    minima.put(minimumPos, value);
                    maximum = value;
                    maximumPos = indices.get(i);
                    lookForMax = true;
                }
            }
        }
```

```
> {...} maxPeaks = {HashMap@17405} size = 1
v {...} peaks = {ArrayList@17406} size = 2
  v {...} 0 = {HashMap@17405} size = 1
    > {...} {Integer@17498} 525 -> {Double@17499} 15.893942880684065
    {...} 1 = {HashMap@17494} size = 0
v {...} peaksFrequencies = {ArrayList@17407} size = 1
  > {...} 0 = {Double@17492} 499.99999993790107
> {...} y = {ArrayList@17408} size = 251904
```

Jak można dostrzec odnaleziony pik pokrywa się ze spektrum i spełnia oczekiwania odnalezionej częstotliwości sygnału morsa. Kolejno po poznaniu porządanej częstotliwości należało zastosować odpowiedni algorytm, pozwalający odtworzyć wybraną częstotliwość. W swoich obliczeniach wykorzystaliśmy tak zwany "matched filter" https://en.wikipedia.org/wiki/Matched_filter.

```java
public List<Double> mfilter(double[] x, int speed, double Fs, double morseCodeFrequency) {
    double dot_time = 1.2 / speed;
    int x_len = x.length;
    double[] t = new double[(int) (dot_time * Fs)];
    double sum = 0;
    for (int i = 0; i < t.length; i++) {
        t[i] = sum;
        sum += 1 / Fs;
    }

    double[] burst = new double[t.length];
    for (int i = 0; i < burst.length; i++) {
        burst[i] = Math.sin(2 * Math.PI * morseCodeFrequency * t[i]);
    }
    int N = burst.length;

    List<Double> x_f1 = new ArrayList<>( initialCapacity: x_len - N);
    for (int i = 0; i < x_len - N; i++) {
        double[] xk = new double[N];
        System.arraycopy(x, i, xk,  destPos: 0, N);
        double dotProduct = 0;
        for (int l = 0; l < N; l++) {
            dotProduct += burst[l] * xk[l];
        }
        x_f1.add(dotProduct);
    }
    return x_f1;
}
```

Tak zaimplementowany "matched filter" pozwala na odtworzenie sygnału morsa.



Tak prezentuje się sygnał otrzymany po odfiltrowaniu za pomocą wyżej wspomnianego algorytmu. Obserwując otrzymane wyniki można stwierdzić,

że algorytm zadziałał prawidłowo I pozwolił na oczyszczenie sygnału z większości zanieczyszczeń I wydobyć pożądany sygnał.

Kolejnym ostatecznym krokiem danego algorytmu byłoby sporządzenia kolejnego podalgorytmu wykorzystującego na przykład oktawy do odczytania sygnałów morsa, przypisania odpowiednich wartości I odkodowania sygnału, jednakże nie udało nam się w danym czasie zaprojektować takiego rozwiązania.

3. Wnioski

Prezentowane w aplikacji algorytmu pozwalają w pewnych warunkach na odczytanie sygnałów morsa, jednak najbardziej uniwersalne rozwiązanie niestety nie zostało w pełni zrealizowane. W przyszłości można byłoby dokończyć wspomniane rozwiązanie związane ze spektrum, odpowiednie zaimplementowanie danego rozwiązania pozwoliło by na zastąpienie algorytmu ze średnią I stałoby się rozwiązaniem w miarę uniwersalnym.

4. Podział pracy

Michał Okoń: Implementacja algorytmu generowania sygnału Morsa z dostarczonego tekstu, Przystosowanie biblioteki do FFT oraz klasy Complex na potrzeby algorytmów, Przygotowanie funkcji wczytującej części pliku .wav (w algorytmie ze średnią), część interfejsu użytkownika, generowanie wykresów

Marcin Oleszczuk: Dostosowanie biblioteki JLibrosa oraz klasy WavFile do obsługi plików dźwiękowych w projekcie, element algorytmu z wyznaczaniem morsa ze średniej takich jak : tłumaczenie ze znaków na tekst, obliczanie średniej wartości, wydobycie odpowiednich parametrów z pliku .wav, translacja otrzymanych wyników na tekst, część interfejsu użytkownika, wykonanie algorytmu przekształcenia sygnału z wartości do dziedziny częstotliwości oraz decybeli (dB), dostosowanie algorytmu wyznaczania pików, dostosowanie algorytmu matched filter.