Aplikacja Audio
Dokumentacja i opis projektu

Autor:
Jakub Półtorak

Data:
29 marca 2025

Spis treści

1 Wstęp

Niniejsza dokumentacja przedstawia opis aplikacji służącej do wczytywania, odtwarzania oraz podstawowej analizy sygnałów dźwiękowych w formacie `.wav`. Projekt został zrealizowany w języku `Python` z wykorzystaniem biblioteki `Tkinter` do stworzenia interfejsu graficznego oraz wielu innych pakietów zapisanych w pliku `requirements.txt`

Aplikacja umożliwia:

- Wczytanie pliku `.wav`.
- Odtworzenie dźwięku (z możliwością pauzowania).
- Wizualizację przebiegu czasowego sygnału.
- Zaznaczanie fragmentów ciszy lub części dźwięcznych/bezdźwięcznych.
- Analizę wybranych cech sygnału (Volume/RMS, STE, ZCR, itp.) w parametrach w dziedzinie czasu na poziomie ramki.

2 Opis aplikacji

2.1 Struktura projektu i główne moduły

Aplikacja została podzielona na kilka plików i modułów w następujący sposób:

- `main.py` – główny punkt startowy aplikacji uruchamiający okno `Tkinter` i tworzący instancję klasy `AudioApp`.
- `audio_app.py` – moduł zawierający główną klasę `AudioApp`, odpowiadającą za interfejs użytkownika i obsługę logiki związanej z odtwarzaniem oraz wizualizacją dźwięku.
- `audio_processing.py` – definicja klasy `VoicedAudioProcessor`, dziedziczącej po `BaseAudioProcessor`, zawierająca funkcje związane z detekcją ciszy oraz segmentacją sygnału na fragmenty dźwięczne i bezdźwięczne.
- `features.py` – moduł z funkcjami do obliczania podstawowych parametrów sygnału na poziomie ramki, takich jak RMS, ZCR, STE czy częstotliwość podstawowa.
- `features_window.py` – klasa `FeaturesWindow`, która wyświetla dodatkowe wykresy wcześniej wspominanych cech dla poszczególnych ramek.
- `design.py` – moduł zawierający klasę `ColorScheme` i funkcję `configure_style`, definiujące kolorystykę i style użyte w aplikacji.

2.2 Interfejs graficzny (GUI)

Aplikacja wykorzystuje pakiet `Tkinter` do stworzenia okna, przycisków i widżetów. Główny interfejs zawiera:

1. Przycisk do wczytania pliku `.wav`.
2. Przycisk `Odtwórz` wraz z przyciskiem `Pauza` i `Odtwórz od początku`.

3. Przycisk `Wykresy cech`, otwierający nowe okno z dodatkowymi wykresami.

4. Suwak do nawigacji po sygnale (w sekundach).

5. Wykres przebiegu czasowego narysowany za pomocą `matplotlib`.

6. Etykiety wyświetlające nazwę pliku, aktualny czas, średni RMS, itp.

Dodatkowo wyróżnione są obszary ciszy (kolor pomarańczowo-różowy) lub części dźwięczne i bezdźwięczne (odpowiednio kolor zielony i różowy) zgodnie z wybranym trybem podświetlania.

2.3 Opis kodu

2.3.1 Plik `main.py`

Zawiera funkcję `main()`, która uruchamia całą aplikację. Jej kod prezentuje się następująco:

```
def main():
    root = Tk()
    root.title("AudioApp")
    root.geometry("900x700")

    app = AudioApp(root)
    root.protocol("WM_DELETE_WINDOW", app.on_close)
    root.mainloop()
    sys.exit(0)
```

2.3.2 Plik `audio_app.py`

Składa się z klasy `AudioApp`, która zawiera wiele metod potrzebnych do niezbędnego działania głównego okna aplikacji. W pierwszej kolejności zostanie opisany konstruktor, który ma poniższą postać:

```
    def __init__(self, master):
        self.master = master

        # Konfigurujemy styl
        self.style = ttk.Style()
        configure_style(self.style)

        # Klasa do przetwarzania audio (analizy ciszy, d wi czno ci
            itd.)
        self.processor = VoicedAudioProcessor()

        self.master.title("Aplikacja␣Audio")
        self.master.geometry("900x700")

        # Zmienne audio
        self.fs = None
        self.data = None
        self.total_samples = 0
        self.time_array = None
        self.current_index = 0
        self.filename = ""
```

```
21
22          # Flagi sterowania
23          self.playing = False
24          self.paused = False
25
26          # Strumie  audio w sounddevice
27          self.stream = None
28
29          # Parametry analizy
30          self.silence_threshold = 0.001
31          self.frame_size = 256
32
33          # Zmienna do wyboru trybu pod wietlania
34          self.highlight_mode = tk.StringVar(value="silence")  #
                domy lnie "silence"
35
36          # Referencja do pionowej linii
37          self.line = None
38
39          # Bliting     statyczne t o wykresu
40          self.background = None
41
42          # G   wna ramka
43          self.main_frame = ttk.Frame(self.master, style="App.TFrame")
44          self.main_frame.pack(fill="both", expand=True)
45
46          self.create_widgets()
47
48          # Wywo ujemy p tl   do aktualizowania UI (pozycji odtwarzania
                itp.)
49          self.ui_after = None
50          self.update_ui()
51          self.master.protocol("WM_DELETE_WINDOW", self.on_close)
```

Konstruktur ustawia główny interfejs użytkownika, a także styl. Inicjalizuje kluczowe zmienne związane z przetwarzaniem dźwięku, tworzy także instancje klasy `VoicedAudioProccessor`, która w późniejszym etapie zostanie dokładniej opisana. Ustawia też flagi niezbędne do sterowania odtwarzaniem i dostosowuje parametry analizy takie jak próg ciszy czy wielkość ramki. Ustawia także domyślnie wykrywanie ciszy na wykresie w głównym oknie. Rozpoczyna pętlę do aktualizowania pozycji odtwarzania i aspektów powiązanych z wykresem. Wywołuje także metodę `create_widgets(self)`, której treść znajduje się poniżej:

```
1       def create_widgets(self):
2           # --- G rny panel z przyciskami ---
3           self.top_frame = ttk.Frame(self.main_frame, style="Controls.
                TFrame")
4           self.top_frame.pack(side="top", fill="x", padx=10, pady=10)
5
6           self.load_button = ttk.Button(
7               self.top_frame,
8               text="Wczytaj␣plik␣WAV",
9               command=self.load_file
10          )
11          self.load_button.grid(row=0, column=0, padx=5, pady=5)
12
13          self.play_button = ttk.Button(
14              self.top_frame,
15              text="Odtw rz",
```

```
            command=self.play_audio,
            state="disabled"
        )
        self.play_button.grid(row=0, column=1, padx=5, pady=5)

        # Odtwarzanie od pocz tku
        self.play_from_start_button = ttk.Button(
            self.top_frame,
            text="Odtw rz od pocz tku",
            command=self.play_from_start,
            state="disabled"
        )
        self.play_from_start_button.grid(row=0, column=2, padx=5, pady
            =5)

        self.pause_button = ttk.Button(
            self.top_frame,
            text="Pauza",
            command=self.toggle_pause,
            state="disabled"
        )
        self.pause_button.grid(row=0, column=3, padx=5, pady=5)

        self.features_button = ttk.Button(
            self.top_frame,
            text="Wykresy cech",
            command=self.open_features_window,
            state="disabled"
        )
        self.features_button.grid(row=0, column=4, padx=5, pady=5)

        self.close_button = ttk.Button(
            self.top_frame,
            text="Zamknij",
            command=self.on_close
        )
        self.close_button.grid(row=0, column=5, padx=5, pady=5)

        # --- Sekcja info: nazwa pliku, czas, tryb ---
        info_frame = ttk.Frame(self.main_frame, style="App.TFrame")
        info_frame.pack(side="top", fill="x", padx=10, pady=(0, 5))

        self.file_label = ttk.Label(
            info_frame,
            text="Brak wczytanego pliku",
            style="TitleLabel.TLabel"
        )
        self.file_label.pack(side="top", anchor="w")

        self.time_label = ttk.Label(
            info_frame,
            text="Czas: 00:00"
        )
        self.time_label.pack(side="top", anchor="w", pady=5)

        # Etykieta z aktualnym trybem
        self.highlight_label = ttk.Label(
            info_frame,
```

```
            text="Aktualnie pokazujemy: CISZ "
        )
        self.highlight_label.pack(side="top", anchor="w", pady=5)

        # Suwak (inicjalizujemy po wczytaniu pliku)
        self.slider = None

        # Tekst z parametrami
        self.frame_params_text = tk.StringVar()
        self.params_label = ttk.Label(
            info_frame,
            textvariable=self.frame_params_text,
            justify="left"
        )
        self.params_label.pack(side="top", anchor="w", pady=5)

        # --- Opcje wyboru trybu podświetlania ---
        mode_frame = ttk.Frame(self.main_frame, style="Controls.TFrame")
        mode_frame.pack(side="top", fill="x", padx=10, pady=5)

        ttk.Label(
            mode_frame,
            text="Tryb podświetlania:",
            style="TitleLabel.TLabel"
        ).pack(side="left", padx=(5, 10))

        rb_silence = ttk.Radiobutton(
            mode_frame,
            text="Cisza",
            variable=self.highlight_mode,
            value="silence",
            command=self.update_highlight_mode
        )
        rb_silence.pack(side="left", padx=5)

        rb_voiced = ttk.Radiobutton(
            mode_frame,
            text="Dźwięczne/Bezdźwięczne",
            variable=self.highlight_mode,
            value="voiced_unvoiced",
            command=self.update_highlight_mode
        )
        rb_voiced.pack(side="left", padx=5)

        # --- Ramka z wykresem audio ---
        plot_frame = ttk.LabelFrame(
            self.main_frame,
            text="Przebieg czasowy sygnału",
            style="App.TFrame"
        )
        plot_frame.pack(fill="both", expand=True, padx=10, pady=10)

        self.fig, self.ax = plt.subplots(figsize=(8, 3))
        self.canvas = FigureCanvasTkAgg(self.fig, master=plot_frame)
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True, padx
            =5, pady=5)

        # Podpięcie obsługi zdarzenia zmiany rozmiaru wykresu
```

```
130         self.canvas.mpl_connect('resize_event', self.on_resize)
```

Głowny cel tej metody to tworzenie i rozmieszczenie widgetów w interfejsie użytkownika. Na początku buduje główny panel z przyciskami: Wczytaj plik WAV, Odtwórz, Odtwórz od początku, Pauza, Wykresy cech, Zamknij. Następnie dodaje panel informacyjny wyświetlający nazwę wczytanego pliku, aktualny czas odtwarzania i możliwość wyboru detekcji ciszy lub detekcji fragmentów dźwięcznych i bezdźwięcznych. Inicjalizuje także suwak oraz tworzy ramkę z wykresem przebiegu czasowego pliku audio. Obsługuje także zdarzenie zmiany rozmiaru wykresu w przypadku, gdy zmniejszymy lub zwiększymy rozmiar okna. Następną rozważaną metodą jest `on_resize(self,event)`:

```
1     def on_resize(self, event):
2         self.background = None
3         if self.line is not None:
4             self.canvas.draw()
5             self.background = self.canvas.copy_from_bbox(self.ax.bbox)
```

Funkcja wywoływana przy zmianie rozmiaru okna wykresu.Resetuje tło do aktualnych rozmiarów, aby blitting działał poprawnie. Kolejna rozważana metoda to `update_highlight_mode(self)`, której treść prezentuje się następująco:

```
1     def update_highlight_mode(self):
2         mode = self.highlight_mode.get()
3         if mode == "silence":
4             self.highlight_label.config(text="Aktualnie␣pokazujemy:␣
                  CISZ ")
5         else:
6             self.highlight_label.config(text="Aktualnie␣pokazujemy:␣
                  D WI CZNE␣/␣BEZD WI CZNE")
7
8         # Przerysuj wykres w nowym trybie
9         if self.data is not None:
10            self.draw_main_plot()
```

Metoda odpowiada za aktualizację wykresów po zmianie trybu wyświetlania detekcji ciszy bądź detekcji fragmentów dźwięcznych i bezdźwięcznych. Następna metoda to `load_file(self)`:

```
1     def load_file(self):
2         filepath = filedialog.askopenfilename(
3             filetypes=[("WAV␣files", "*.wav"), ("All␣files", "*.*")]
4         )
5         if not filepath:
6             return
7
8         self.filename = filepath
9         base_name = os.path.basename(filepath)
10        self.file_label.config(text=f"Plik:␣{base_name}")
11
12        try:
13            self.fs, raw_data = wavfile.read(filepath)
14        except Exception as e:
15            messagebox.showerror("B    d", f"Nie␣uda o␣si ␣wczyta ␣
                  pliku␣WAV:\n{e}")
16            return
17
18        if len(raw_data.shape) > 1:
19            raw_data = raw_data[:, 0]
20
```

```
        raw_data = raw_data.astype(np.float32)
        peak = np.max(np.abs(raw_data))
        if peak > 1e-9:
            raw_data /= peak

        self.data = raw_data
        self.total_samples = len(self.data)
        duration = self.total_samples / self.fs if self.fs else 0.001
        self.time_array = np.linspace(0, duration, self.total_samples)
        self.current_index = 0

        # Rysujemy g   wny wykres
        self.draw_main_plot()
        self.calculate_and_display_frame_params()

        # Tworzymy / aktualizujemy suwak
        if not self.slider:
            self.slider = ttk.Scale(
                self.main_frame,
                from_=0,
                to=max(duration, 0.001),
                orient="horizontal",
                command=self.on_slider_move,
                length=600
            )
            self.slider.pack(pady=5)
        else:
            self.slider.config(to=duration)
            self.slider.set(0)

        # W   czamy przyciski
        self.play_button.state(["!disabled"])
        self.pause_button.state(["!disabled"])
        self.features_button.state(["!disabled"])
        self.play_from_start_button.state(["!disabled"])

        # Zatrzymujemy poprzedni strumie   (je li by )
        self.stop_audio()

        # Tworzymy nowy strumie   audio
        self.stream = sd.OutputStream(
            samplerate=self.fs,
            blocksize=1024,
            channels=1,
            dtype='float32',
            callback=self.audio_callback
        )
```

Odpowiada za wczytanie pliku WAV i ustawienie niezbędnych parametrów w celu owocnego odtworzenia pliku audio, a także wywołuje inne niezbędne metody w celu odpowiedniej analizy i przetwarzania pliku dźwiękowego. Dane są konwertowane i normalizowane w celu skutecznego odtworzenia pliku. Kolejna metoda to `draw_main_plot(self)`:

```
    def draw_main_plot(self):
        self.ax.clear()
        self.ax.set_title("Przebieg␣czasowy␣sygna u", fontsize=11,
            color=ColorScheme.ACCENT)
        self.ax.set_xlabel("Czas␣[s]", fontsize=9)
```

```
5           self.ax.set_ylabel("Amplituda", fontsize=9)
6           self.ax.plot(self.time_array, self.data, linewidth=0.8, color=
                ColorScheme.WAVEFORM_COLOR)
7
8           # Tworzymy legend patches     zale nie od trybu
9           legend_patches = []
10          mode = self.highlight_mode.get()
11
12          if mode == "silence":
13              # Zaznaczamy tylko cisz
14              silence_regions = self.processor.detect_silence(
15                  self.data, self.fs, self.frame_size, self.
                        silence_threshold
16              )
17              for (start_idx, end_idx) in silence_regions:
18                  start_t = start_idx / self.fs
19                  end_t = end_idx / self.fs
20                  self.ax.axvspan(start_t, end_t, color=ColorScheme.
                        SILENCE_COLOR, alpha=0.6)
21
22              # Patch do legendy
23              silence_patch = Patch(facecolor=ColorScheme.SILENCE_COLOR,
                    alpha=0.6, label="Cisza")
24              legend_patches.append(silence_patch)
25
26          else:
27              # Zaznaczamy d wi czne/bezd wi czne
28              vu_regions = self.processor.detect_voiced_unvoiced(self.data
                    , self.fs, self.frame_size)
29              for (start_idx, end_idx, is_voiced) in vu_regions:
30                  start_t = start_idx / self.fs
31                  end_t = end_idx / self.fs
32                  if is_voiced:
33                      self.ax.axvspan(start_t, end_t, color=ColorScheme.
                            VOICED_COLOR, alpha=0.3)
34                  else:
35                      self.ax.axvspan(start_t, end_t, color=ColorScheme.
                            UNVOICED_COLOR, alpha=0.3)
36
37              # Patche do legendy
38              voiced_patch = Patch(facecolor=ColorScheme.VOICED_COLOR,
                    alpha=0.3, label="D wi czne")
39              unvoiced_patch = Patch(facecolor=ColorScheme.UNVOICED_COLOR,
                     alpha=0.3, label="Bezd wi czne")
40              legend_patches.extend([voiced_patch, unvoiced_patch])
41
42          # Linia aktualnej pozycji
43          self.line = self.ax.axvline(x=0, color="#004D40", linewidth=2)
44
45          # Dodajemy legend  (je li cokolwiek zaznaczamy)
46          if legend_patches:
47              self.ax.legend(handles=legend_patches, loc="upper␣right",
                    fontsize=8)
48
49          self.canvas.draw()
50
51          # Po narysowaniu wykresu usuwamy background,
52          # kt ry zostanie zaktualizowany przy starcie odtwarzania
```

```
53          self.background = None
```

Funkcja odpowiada za rysowanie wykresu przebiegu czasowego audio. W zależności od wybranego trybu dokonuje detekcji ciszy lub zaznacza fragmenty dźwięczne i bezdźwięczne audio. Dodaje także legendę i niezbędne opisy, a także pionową linię, która pokazuje w rzeczywistym czasie przebieg nagrania. Po narysowaniu wykresu tło jest usuwane i aktualizowane dopiero przy starcie odtwarzania Kolejna metoda to metoda `on_slider_move(self,value)`

```
1     def on_slider_move(self, value):
2         if self.data is not None:
3             new_index = int(float(value) * self.fs) if self.fs else 0
4             self.current_index = np.clip(new_index, 0, self.
                  total_samples)
5             self.update_time_label(float(value))
6 
7             # Je eli linia istnieje, przesuwamy j
8             if self.line is not None and self.background is not None:
9                 self.canvas.restore_region(self.background)
10                self.line.set_xdata([float(value), float(value)])
11                self.ax.draw_artist(self.line)
12                self.canvas.blit(self.ax.bbox)
```

Odpowiada za aktualizację pozycji odtwarzania przy przesunięciu suwaka na dole. Oblicza nowy indeks danych audio na podstawie pozycji suwaka i dzięki temu znajduje odpowiednie miejsce na wykresie. Dodatkowo aktualizuje etykietę czasu. Najpierw przywracany jest stan tła (`restore_region`), potem rysowana jest linia, na koniec wywołujemy `blit` (przyspiesza renderowanie, bo nie musimy rysować całego wykresu od nowa). Następna metoda to `update_time_label(self,current_time)`

```
1     def update_time_label(self, current_time):
2         minutes = int(current_time // 60)
3         seconds = int(current_time % 60)
4         self.time_label.config(text=f"Czas:␣{minutes:02d}:{seconds:02d}"
              )
```

Aktualizuje etykietę czasu odtwarzania wyświetlając go w minutach i sekundach. Kolejna metoda to `calculate_and_display_frame_params(self)`

```
1     def calculate_and_display_frame_params(self):
2         frame_step = self.frame_size
3         rms_values = []
4         zcr_values = []
5         for i in range(0, self.total_samples, frame_step):
6             frame = self.data[i:i+frame_step]
7             if len(frame) == 0:
8                 continue
9             rms = np.sqrt(np.mean(frame**2))
10            rms_values.append(rms)
11            zero_crosses = np.count_nonzero(np.diff(np.sign(frame)))
12            zcr = zero_crosses / len(frame) if len(frame) != 0 else 0
13            zcr_values.append(zcr)
14        avg_rms = np.mean(rms_values) if rms_values else 0
15        avg_zcr = np.mean(zcr_values) if zcr_values else 0
16 
17        text = (
18            f"Parametry␣nagrania␣(ramkowe):\n"
```

```
            f"␣␣    ␣ redni ␣RMS␣(Volume):␣{avg_rms:.6f}\n"
            f"␣␣    ␣ rednie ␣ZCR:␣{avg_zcr:.6f}\n"
            f"(Pr g␣ciszy:␣{self.silence_threshold})"
        )
        self.frame_params_text.set(text)
```

Oblicza parametry sygnału audio i wyświetla je na górze w skompresowanej formie. Wyświetla między innymi średnią głośność i średnie ZCR. Następna funkcja to `audio_callback(self, outdata, frames, time_info, status)`:

```
    def audio_callback(self, outdata, frames, time_info, status):
        if status:
            #print("Status audio_callback:", status, flush=True)
            pass

        if (self.data is None) or (not self.playing) or (self.paused):
            outdata.fill(0)
            return

        end_index = self.current_index + frames
        if end_index > self.total_samples:
            end_index = self.total_samples

        chunk = self.data[self.current_index:end_index]
        out_len = len(chunk)
        if out_len < frames:
            outdata[:out_len, 0] = chunk
            outdata[out_len:, 0] = 0
            self.playing = False
            self.current_index = self.total_samples
        else:
            outdata[:, 0] = chunk
            self.current_index = end_index
```

Zamiast tworzyć własny wątek do odtwarzania, kod korzysta z `sounddevice` i przypisanej funkcji zwrotnej `audio_callback`. Funkcja ta jest automatycznie wywoływana przez moduł `sounddevice` (wątek dźwięku) każdorazowo, gdy należy dostarczyć kolejną porcję próbek. Korzysta z `self.current_index` do pobrania wycinka danych `chunk` i wpisuje je do `outdata`. Kolejna metoda to `play_audio(self)`

```
    def play_audio(self):
        if self.data is None or not self.stream:
            return

        # Je eli linia nie istnieje, tworzymy j
        if self.line is None:
            self.line = self.ax.axvline(x=0, color="#004D40", linewidth
                =2)
            # Najpierw pe ne rysowanie wykresu
            self.canvas.draw()
            # Dopiero teraz pobieramy background
            self.background = self.canvas.copy_from_bbox(self.ax.bbox)

        # Je eli dotarli my do ko ca      wr   my na pocz tek
        if self.current_index >= self.total_samples:
            self.current_index = 0
            if self.slider:
                self.slider.set(0)
```

```
            # Reset linii do 0
            if self.line:
                self.line.set_xdata([0, 0])
                self.canvas.draw()
                self.background = self.canvas.copy_from_bbox(self.ax.
                    bbox)

        self.playing = True
        self.paused = False
        self.pause_button.config(text="Pauza")

        # Start strumienia
        if not self.stream.active:
            self.stream.start()
```

Metoda odpowiada za uruchomienie pliku audio i ustawia niezbędne flagi na odpowiednie wartości. Sprawdza także czy dane audio są dostępne, a także czy nie osiągnięto końca pliku. Gdy `self.line` jeszcze nie istnieje, tworzymy ją przez `axvline`. Następnie rysujemy cały wykres i zapisujemy w `self.background` aktualne tło. Kolejna metoda to `play_from_start(self)`

```
    def play_from_start(self):
        self.current_index = 0
        if self.slider:
            self.slider.set(0)
        if self.line:
            self.line.set_xdata([0, 0])
            self.canvas.draw()
        self.play_audio()
```

Metoda odpowiada za odtwarzanie pliku od początku. Resetuje indeks, i slider na zerowe wartości i wywołuje poprzednią metodę w celu odtworzenia nagrania. Następna metoda to `stop_audio(self)`

```
    def stop_audio(self):
        self.playing = False
        self.paused = False
        if self.stream and self.stream.active:
            self.stream.stop()

        # Usuwamy lini  z osi, je li istnieje
        if self.line:
            self.line.remove()
            self.line = None

        # Od wie  wykres (linia znika)
        self.canvas.draw()
        self.background = None
```

Po prostu zatrzymuje odtwarzanie i usuwa niepotrzebne linie, które zostały wcześniej stworzone. Kolejna metoda to `toogle_pause(self)`

```
    def toggle_pause(self):
        if not self.playing:
            return
        if self.paused:
            self.pause_button.config(text="Pauza")
            self.paused = False
```

```
            self.play_audio()
        else:
            self.pause_button.config(text="Wzn w")
            self.paused = True
            sd.stop()
```

Odpowiada za wstrzymanie nagrania i ewentualne odtworzenie z powrotem uwzględniając przy tym zmiany wyświetlanych nazw w odpowiednich miejscach. Kolejna metoda to metoda `update_ui(self)`

```
    def update_ui(self):
        if self.data is not None and self.playing and not self.paused:
            current_time = self.current_index / self.fs if self.fs else
                0

            # Ustawiamy suwak
            if self.slider:
                self.slider.set(current_time)

            # Aktualizujemy etykiet  z czasem
            self.update_time_label(current_time)

            # Przesuwamy lini  (blitowanie)
            if self.line and self.background:
                self.canvas.restore_region(self.background)
                self.line.set_xdata([current_time, current_time])
                self.ax.draw_artist(self.line)
                self.canvas.blit(self.ax.bbox)

        # Wywo anie za 50 ms ponownie
        self.ui_after = self.master.after(50, self.update_ui)
```

Metoda odpowiada za okresowe odświeżanie interfejsu, które jest realizowane co 50ms.Przywracamy stan tła i rysujemy samą linię (`self.line`). To tzw. blitting, który zapewnia płynne, wydajne przesuwanie się znacznika aktualnej pozycji audio. Kolejna metoda to `open_features_window(sel` która ma nastepującą definicję:

```
    def open_features_window(self):
        if self.data is None:
            messagebox.showwarning("Brak danych", "Najpierw wczytaj plik
                WAV!")
            return
        FeaturesWindow(self.master, self.data, self.fs, self.frame_size,
            self.silence_threshold)
```

Odpowiada za otwarcie okna z wykresami z odpowiednimi cechami sygnału audio. Zabezpiecza się także przed otwarciem okna w przypadku braku wczytanego pliku wav. Ostatnia metoda w tej klasie i tym pliku to `on_close(self)`

```
    def on_close(self):
        self.stop_audio()
        if self.ui_after is not None:
            self.master.after_cancel(self.ui_after)
            self.ui_after = None
        if self.stream:
            self.stream.close()
            self.stream = None
        self.master.destroy()
```

```
10          sys.exit(0)
```

Uruchamia ją przycisk "Zamknij". Zatrzymuje wszystkie istotne wątki i zamyka aplikację.

2.3.3 Plik `features.py`

Zawiera kilka funkcji pozwalających przeprowadzać analizę dźwięku. Pierwsza z nich to `compute_volume(frame)`, której definicja prezentuje się następująco:

```
1     def compute_volume(frame):
2         return np.sqrt(np.mean(frame**2)) if len(frame) > 0 else 0.0
```

Oblicza wartość RMS na podstawie próbki sygnału. Korzysta z następującego wzoru matematycznego

$$\mathrm{RMS} = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} x[n]^2}. \tag{1}$$

gdzie N - długość ramki, a $x[n]$ to amplituda n-tej próbki w ramce. Druga funkcja to `compute_ste(frame)`:

```
1     def compute_ste(frame):
2         return np.sum(frame**2) / len(frame) if len(frame) > 0 else 0.0
```

Na wejściu również przyjmuje ramkę i na jej podstawie oblicza wskaźnik STE określany jako krótkoczasowa energia ramki, która definiowana jest następującym wzorem:

$$\mathrm{STE} = \frac{1}{N}\sum_{n=0}^{N-1} x[n]^2. \tag{2}$$

Oznaczenia są identyczne jak w poprzednio rozważanej funkcji. Następną rozważaną funkcją jest `compute_zcr(frame)`, która ma następujący kod:

```
1     def compute_zcr(frame):
2         if len(frame) == 0:
3             return 0.0
4         zero_crossings = np.count_nonzero(np.diff(np.sign(frame)))
5         return zero_crossings / len(frame)
```

Na podstawie ramki obliczamy miarę ZCR(Zero Crossing Rate), która odpowiada za liczbę przejść sygnału przez zero. Dla ramki $x[n]$ i funkcji indykatorowej $\mathcal{I}$ obserwujemy następujący wzór:

$$\mathrm{ZCR} = \frac{1}{N-1}\sum_{n=1}^{N-1} \mathcal{I}\big(\mathrm{sgn}(x[n]) \neq \mathrm{sgn}(x[n-1])\big), \tag{3}$$

Kolejna rozważana funkcja to `compute_scr(frame,vol_threshold=0.01,zcr_threshold=0.1)`:

```
1     def compute_sr(frame, vol_threshold=0.01, zcr_threshold=0.1):
2         vol = compute_volume(frame)
3         zcr = compute_zcr(frame)
4     return 1 if (vol < vol_threshold and zcr < zcr_threshold) else 0
```

Liczy po prostu silent ratio na podstawie wzoru:

$$SR = \mathcal{I}((RMS < vol\_threshold)\&(ZCR < zcr\_threshold)) \tag{4}$$

Kolejna funkcja to `compute_autocorr_f0(frame,fs,fmin=50,fmax=500)`:

```
    def compute_autocorr_f0(frame, fs, fmin=50, fmax=500):
        if len(frame) == 0:
            return 0
        frame = frame - np.mean(frame)
        corr = np.correlate(frame, frame, mode='full')
        corr = corr[len(corr)//2:]
        d = np.diff(corr)
        start = np.nonzero(d > 0)[0]
        if len(start) == 0:
            return 0
        lag = start[0]
        if lag == 0:
            return 0
        f0 = fs / lag
        if f0 < fmin or f0 > fmax:
            return 0
        return f0
```

Na początku funkcja usuwa średnią wartość sygnału, aby wyeliminować składową stałą:

$$x'[n] = x[n] - \mu, \quad \text{gdzie} \quad \mu = \frac{1}{N}\sum_{n=0}^{N-1} x[n]. \tag{5}$$

Następnie obliczana jest autokorelacja znormalizowanej ramki dla opóźnień $\tau \geq 0$. Dla sygnału $x'[n]$ definicja autokorelacji przyjmowana jest w postaci:

$$R_{xx}(\tau) = \sum_{n=0}^{N-1-\tau} x'[n] \cdot x'[n+\tau]. \tag{6}$$

Aby wyznaczyć okres sygnału, funkcja oblicza różnicę między kolejnymi wartościami autokorelacji:

$$d(\tau) = R_{xx}(\tau+1) - R_{xx}(\tau). \tag{7}$$

Pierwszy indeks $\tau = \text{lag}$, dla którego wartość $d(\tau)$ staje się dodatnia (tj. $d(\tau) > 0$), jest traktowany jako początek okresu sygnału. Przy założeniu, że znaleziony lag odpowiada okresowi sygnału, estymowana wartość F0 jest obliczana jako:

$$f_0 = \frac{f_s}{\text{lag}}, \tag{8}$$

gdzie f_s to częstotliwość próbkowania. Jeśli uzyskana wartość f_0 nie mieści się w zadanym przedziale $[f_{\min}, f_{\max}]$, funkcja zwraca 0. Kolejna rozważana funkcja to `compute_amdf(frame)`:

```
    def compute_amdf(frame):
        length = len(frame)
        if length == 0:
            return np.array([0])
        amdf = []
        for tau in range(length):
            diff_sum = 0.0
            for n in range(length - tau):
                diff_sum += abs(frame[n] - frame[n + tau])
            amdf.append(diff_sum / (length - tau))
        return np.array(amdf)
```

Oblicza wartość funkcji AMDF dla każdego opóźnienia τ, gdzie:

$$\mathrm{AMDF}(\tau) = \frac{1}{N-\tau} \sum_{n=0}^{N-\tau-1} |x[n] - x[n+\tau]| .$$

Ostatnia funkcja w tym pliku to `compute_amdf_f0`:

```
    def compute_amdf_f0(frame, fs, fmin=50, fmax=500):
        length = len(frame)
        if length == 0:
            return 0
        frame = frame - np.mean(frame)
        amdf_values = compute_amdf(frame)

        min_lag = int(fs // fmax)
        max_lag = int(fs // fmin) if fmin != 0 else length // 2
        if max_lag > len(amdf_values):
            max_lag = len(amdf_values) - 1
        if min_lag < 1 or min_lag >= max_lag:
            return 0

        lag_range = np.arange(min_lag, max_lag)
        best_lag = lag_range[np.argmin(amdf_values[min_lag:max_lag])]
        if best_lag == 0:
            return 0
        f0 = fs / best_lag
        if f0 < fmin or f0 > fmax:
            return 0
        return f0
```

Pierwsze dwa kroki są analogiczne jak dla autokorelacji z taką różnicą, że zamiast autokorelacji obliczamy $AMDF$. Aby ograniczyć poszukiwania opóźnienia odpowiadającego częstotliwości podstawowej do interesującego nas przedziału, definiuje się:

- **Minimalny lag:**
$$\tau_{\min} = \left\lfloor \frac{f_s}{f_{\max}} \right\rfloor ,$$
- **Maksymalny lag:**
$$\tau_{\max} = \left\lfloor \frac{f_s}{f_{\min}} \right\rfloor ,$$
gdzie f_s oznacza częstotliwość próbkowania, a $f_{\min}$ oraz $f_{\max}$ to odpowiednio minimalna i maksymalna dopuszczalna wartość F0.

Dodatkowo, funkcja weryfikuje, czy wyznaczony zakres jest prawidłowy i dostosowuje $\tau_{\max}$ do długości tablicy AMDF, jeśli jest to konieczne. W obrębie przedziału $\tau \in [\tau_{\min}, \tau_{\max})$ szukane jest opóźnienie, dla którego wartość AMDF jest minimalna:

$$\tau_{\text{best}} = \arg \min_{\tau \in [\tau_{\min}, \tau_{\max})} \mathrm{AMDF}(\tau). \tag{9}$$

Jeżeli znalezione opóźnienie wynosi 0, funkcja zwraca 0, co oznacza nieudane wyznaczenie F0. Znając najlepsze opóźnienie τ_{best}, częstotliwość podstawową wyznacza się według wzoru:

$$f_0 = \frac{f_s}{\tau_{\text{best}}}. \tag{10}$$

Dodatkowo, funkcja sprawdza, czy obliczona wartość F0 mieści się w przedziale $[f_{\min}, f_{\max}]$. Jeśli nie, zwraca 0.

2.3.4 Plik `features_window.py`

Plik zawiera klasę `FeaturesWindow`, która odpowiada za interfejs okna opisującego cechy sygnału, które wyświetla się po kliknięciu w głównym panelu. Dodatkowo zawiera dwie funkcje, które pomagają w redukcji czasu rysowania poszczególnych wykresów. Oto definicja `auto_frame_size(total_samples, max_frames=2000)`:

```
def auto_frame_size(total_samples, max_frames=2000):

    frame_size = max(256, int(np.ceil(total_samples / max_frames)))
    return frame_size
```

Zwraca wielkość ramki tak, by liczba ramek nie przekraczała max_frames. Minimalna wartość to 256. Kolejna funkcja to `downsample_block(x, y, max_points=2000)`:

```
def downsample_block(x, y, max_points=2000):
    n = len(x)
    if n <= max_points:
        return x, y
    block_size = int(np.ceil(n / max_points))
    # Podziel dane na bloki
    x_blocks = [x[i*block_size : (i+1)*block_size] for i in range(int(np
        .ceil(n / block_size)))]
    y_blocks = [y[i*block_size : (i+1)*block_size] for i in range(int(np
        .ceil(n / block_size)))]
    x_ds = np.array([np.mean(block) for block in x_blocks])
    y_ds = np.array([np.mean(block) for block in y_blocks])
    return x_ds, y_ds
```

Agreguje dane (przez średnią) w blokach tak, aby liczba punktów nie przekraczała max_points. Dzięki temu wykresy są rysowane szybciej i nadal zachowują ogólny kształt. Przejdźmy teraz do głównej klasy tego pliku. Jej konstruktor ma następującą postać

```
    def __init__(self, master, data, fs, frame_size, silence_threshold):
        self.top = tk.Toplevel(master)
        self.top.title("Wykresy␣cech␣sygna u")
        self.top.geometry("1000x800")

        # Pastelowe t o
        self.main_bg_color = ColorScheme.MAIN_BG
        self.top.configure(bg=self.main_bg_color)

        # G   wna ramka
        self.main_frame = tk.Frame(self.top, bg=self.main_bg_color)
        self.main_frame.pack(fill="both", expand=True)

        self.data = data
        self.fs = fs

        # Automatyczne dobranie rozmiaru ramki     dla d ugich nagra
            zwi kszamy j  , aby liczba ramek nie by a zbyt du a.
        candidate = auto_frame_size(len(data))
        self.frame_size = min(candidate, frame_size)  # wybieramy
            wi ksz  z tych warto ci
        self.silence_threshold = silence_threshold

        # Dzielimy sygna  na ramki i obliczamy cechy
        self.frames, self.times = self.frame_signal(data, fs, self.
            frame_size)
```

```
24          self.volume = np.array([compute_volume(f) for f in self.frames])
25          self.ste = np.array([compute_ste(f) for f in self.frames])
26          self.zcr = np.array([compute_zcr(f) for f in self.frames])
27          self.sr = np.array([compute_sr(f) for f in self.frames])
28          self.f0_autocorr = np.array([compute_autocorr_f0(f, fs) for f in
                self.frames])
29          self.f0_amdf = np.array([compute_amdf_f0(f, fs) for f in self.
               frames])
30
31          # Przechowujemy cechy
32          self.features_info = {
33              "Volume␣(RMS)": (
34                  self.volume,
35                  "Volume␣okre la␣  redni  ␣g  o  n o  ␣sygna u␣(RMS)."
                        ,
36                  "#4DB6AC"
37              ),
38              "STE": (
39                  self.ste,
40                  "Short␣Time␣Energy␣  ␣rozr  nianie␣fragment w␣
                        d wi cznych/bezd wi cznych.",
41                  "#81C784"
42              ),
43              "ZCR": (
44                  self.zcr,
45                  "Zero␣Crossing␣Rate␣  ␣liczba␣p r z e j  ␣przez␣zero.",
46                  "#FFF176"
47              ),
48              "SR␣(Silent␣Ratio)": (
49                  self.sr,
50                  "1␣oznacza␣ramk ␣sklasyfikowan ␣jako␣cisza.",
51                  "#FFD54F"
52              ),
53              "F0␣(Autocorr)": (
54                  self.f0_autocorr,
55                  "C z  s t o t l i w o  ␣podstawowa␣-␣metoda␣autokorelacji.",
56                  "#BA68C8"
57              ),
58              "F0␣(AMDF)": (
59                  self.f0_amdf,
60                  "C z  s t o t l i w o  ␣podstawowa␣-␣metoda␣AMDF.",
61                  "#FF8A65"
62              ),
63          }
64
65          # Panel wyboru cech
66          self.select_frame = tk.Frame(self.main_frame, bg=self.
               main_bg_color)
67          self.select_frame.pack(side="top", fill="x", padx=10, pady=10)
68
69          tk.Label(
70              self.select_frame,
71              text="Wybierz␣cechy␣do␣wy wietlenia:",
72              bg=self.main_bg_color,
73              fg="#1B5E20",
74              font=("Helvetica", 11, "bold")
75          ).pack(side="left", anchor="n")
76
```

```
77          self.feature_vars = {}
78          for feat_name in self.features_info.keys():
79              var = tk.BooleanVar(value=True)
80              cb = tk.Checkbutton(
81                  self.select_frame,
82                  text=feat_name,
83                  variable=var,
84                  bg=self.main_bg_color,
85                  highlightthickness=0,
86                  font=("Helvetica", 9)
87              )
88              cb.pack(side="left", padx=5)
89              self.feature_vars[feat_name] = var
90
91          self.draw_button = tk.Button(
92              self.select_frame,
93              text="Rysuj",
94              command=self.draw_selected_features,
95              bg="#B2DFDB",
96              fg="#004D40",
97              activebackground="#80CBC4",
98              font=("Helvetica", 9, "bold")
99          )
100         self.draw_button.pack(side="left", padx=10)
101
102         self.plot_frame = tk.Frame(self.main_frame, bg=self.
                main_bg_color)
103         self.plot_frame.pack(fill="both", expand=True, padx=10, pady=(0,
                10))
104
105         self.fig = None
106         self.canvas = None
107
108         # Rysujemy wykresy przy starcie
109         self.draw_selected_features()
```

Tworzy nowe okno, ustawia jego tytuł, rozmiar oraz tło (z wykorzystaniem kolorystyki zdefiniowanej w pliku `design.py`. Sygnał dzielony jest na ramki przy użyciu metody `frame_signal`. Dokonujemy tutaj automatycznego dzielenia na podstawie długości nagrania. Dla każdej ramki obliczany jest czas początkowy. Na podstawie ramek obliczane są cechy sygnału (Volume, STE, ZCR, Silent Ratio, F0 metodą autokorelacji oraz F0 metodą AMDF) z wykorzystaniem funkcji z modułu `features.py`. Klasa tworzy panel, w którym użytkownik może zaznaczyć, które cechy mają być wyświetlone. Informacje o cechach (opis, kolor, dane) są przechowywane w słowniku `features_info`. Metoda `draw_selected_features` odpowiada za rysowanie wykresów wybranych cech:

- Na podstawie zaznaczonych opcji budowany jest nowy wykres przy użyciu `matplotlib.Figure`.
- Układ subplotów określany jest przez metodę pomocniczą `calc_subplot_grid`.
- Dla każdej cechy rysowany jest wykres zależności danej cechy od czasu.
- Wykres jest osadzany w oknie przy pomocy `FigureCanvasTkAgg`.

W klasie znajdują się także metody, które już częściowo zostały przytoczone. Oto treść pierwszej z nich `frame_signal(self, data, fs, frame_size)`:

```
    def frame_signal(self, data, fs, frame_size):
        total_samples = len(data)
        num_frames = int(np.ceil(total_samples / frame_size))
        frames = []
        times = []
        for i in range(num_frames):
            start = i * frame_size
            end = start + frame_size
            frame = data[start:end]
            if len(frame) < frame_size:
                frame = np.pad(frame, (0, frame_size - len(frame)), mode
                    ='constant')
            frames.append(frame)
            times.append(start / fs)
        return frames, np.array(times)
```

Metoda dzieli sygnał audio na ramki o zadanej wielkości. Dla każdej ramki określa jej czas początkowy. W przypadku, gdy ostatnia ramka jest krótsza niż `frame_size`, dopełnia ją zerami. Zwraca listę ramek oraz odpowiadający wektor czasów. Kolejna metoda to `draw_selected_features(self)`:

```
    def draw_selected_features(self):
        selected_features = [name for name, var in self.feature_vars.
            items() if var.get()]

        if self.canvas:
            self.canvas.get_tk_widget().destroy()
            self.canvas = None
        if self.fig:
            plt.close(self.fig)
            self.fig = None

        if not selected_features:
            info_label = tk.Label(
                self.plot_frame,
                text="Nie␣wybrano␣ adnych ␣cech␣do␣wy wietlenia!",
                bg=self.main_bg_color,
                fg="#B71C1C",
                font=("Helvetica", 12, "bold")
            )
            info_label.pack()
            return

        self.fig = plt.Figure(figsize=(12, 6), dpi=100)
        self.fig.set_tight_layout(True)

        n_feats = len(selected_features)
        rows, cols = self.calc_subplot_grid(n_feats)

        # Dla ka dej cechy agregujemy dane, by rysowa  mniej punkt w
        for i, feat_name in enumerate(selected_features, start=1):
            ax = self.fig.add_subplot(rows, cols, i)
            data_array, description, color_line = self.features_info[
                feat_name]
            # U ywamy funkcji downsample_block, by zredukowa  liczb
                punkt w
            x_plot, y_plot = downsample_block(self.times, data_array,
```

```
                max_points=2000)
34              ax.plot(x_plot, y_plot, linewidth=1.0, color=color_line,
                    rasterized=True)
35              ax.set_title(feat_name, fontsize=10, fontweight="bold",
                    color="#2E7D32")
36              ax.set_xlabel("Czas [s]", fontsize=9)
37              ax.set_ylabel(feat_name, fontsize=9)
38              ax.grid(True)
39
40          self.canvas = FigureCanvasTkAgg(self.fig, master=self.plot_frame
                )
41          self.canvas.draw()
42          self.canvas.get_tk_widget().pack(fill="both", expand=True)
```

Metoda ta odpowiada za odświeżenie i narysowanie wykresów wybranych cech sygnału. Jej działanie można opisać następująco:

1. **Pobranie opcji:** Tworzy listę nazw cech, dla których zaznaczono wyświetlanie (na podstawie wartości w `feature_vars`).
2. **Czyszczenie poprzednich wykresów:** Usuwa istniejący widget `canvas` oraz zamyka poprzednią figurę, aby uniknąć nakładania wykresów.
3. **Obsługa braku wyboru:** Jeśli żadna cecha nie została wybrana, wyświetla komunikat informacyjny.
4. **Tworzenie nowej figury:** Generuje nową figurę o wymiarach 12x6 cali (dpi=100) z włączonym układem `tight layout`.
5. **Wyznaczenie układu subplotów:** Na podstawie liczby wybranych cech określa optymalny układ (wiersze i kolumny) wykresów.
6. **Agregacja danych:** Dla każdej cechy wykorzystuje funkcję `downsample_block` do redukcji liczby punktów (do maksymalnie 2000), co przyspiesza rysowanie.
7. **Rysowanie wykresów:** Każda cecha jest rysowana w osobnym subplocie z odpowiednimi etykietami, tytułem i siatką.
8. **Osadzenie figury:** Nowo utworzona figura jest osadzana w oknie za pomocą `FigureCanvasTkAgg` i wyświetlana.

Ostatnia metoda to wcześniej wspomniana `calc_subplot_grid(self,n)`:

```
1       def calc_subplot_grid(self, n):
2           if n == 1:
3               return (1, 1)
4           elif n == 2:
5               return (1, 2)
6           elif n == 3:
7               return (1, 3)
8           elif n == 4:
9               return (2, 2)
10          elif n <= 6:
11              return (2, 3)
12          else:
13              return (2, 3)
```

Metoda pomocnicza, która wyznacza optymalny układ siatki subplotów w zależności od liczby wybranych cech. Na przykład, dla 2 cech zwraca układ 1x2, dla 4 cech układ 2x2 itd.

2.3.5 Plik `design.py`

Składa się z klasy `ColorScheme`, której atrybuty wyglądają następująco:

```
    MAIN_BG = "#E8F5E9"
    FRAME_BG = "#C8E6C9"
    ACCENT = "#43A047"
    WAVEFORM_COLOR = "#1B5E20"
    SILENCE_COLOR = "#FFECB3"
    VOICED_COLOR = "#B3E5FC"
    UNVOICED_COLOR = "#FFCDD2"
```

Opisują one kody poszczególnych kolorów w aplikacji. Dodatkowo mamy metodę `configure_style(styl ttk.Style)`, której definicja prezentuje się następująco:

```
    def configure_style(style: ttk.Style):
        style.theme_use("clam")
        style.configure("App.TFrame", background=ColorScheme.MAIN_BG)
        style.configure("Controls.TFrame", background=ColorScheme.
            FRAME_BG)
        style.configure("TLabel",
                        background=ColorScheme.MAIN_BG,
                        font=("Helvetica", 10))
        style.configure("TitleLabel.TLabel",
                        background=ColorScheme.MAIN_BG,
                        font=("Helvetica", 11, "bold"),
                        foreground=ColorScheme.ACCENT)

        style.configure("TButton",
                        font=("Helvetica", 9, "bold"),
                        padding=5)
        style.map("TButton",
                  foreground=[("active", "#212121"), ("disabled", "
                      #999999")],
                  background=[("active", "#A5D6A7"), ("!active", "#81
                      C784")])
```

Odpowiada ona za designowy aspekt poszczególnych komponentów aplikacji.

2.3.6 Plik `audio_processing.py`

Plik zawiera dwie klasy `BaseAudioProcessor` i `VoicedAudioProcessor`. Pierwsza z nich ma następującą definicję:

```
class BaseAudioProcessor:

    def detect_silence(self, data, fs, frame_size, silence_threshold):
        total_samples = len(data)
        frame_step = frame_size
        silence_regions = []
        in_silence = False
        start_silence = 0
        for i in range(0, total_samples, frame_step):
            frame = data[i:i + frame_size]
```

```
            if len(frame) == 0:
                continue
            # U ywamy compute_volume do obliczenia RMS
            rms = compute_volume(frame)
            if rms < silence_threshold:
                if not in_silence:
                    in_silence = True
                    start_silence = i
            else:
                if in_silence:
                    silence_regions.append((start_silence, i))
                    in_silence = False
        if in_silence:
            silence_regions.append((start_silence, total_samples))
        return silence_regions
```

Klasa `BaseAudioProcessor` definiuje podstawową metodę do detekcji ciszy w sygnale. Jej główna metoda `detect_silence(data, fs, frame_size, silence_threshold)` dzieli sygnał na ramki o zadanym rozmiarze i oblicza wartość RMS dla każdej ramki z wykorzystaniem funkcji `compute_volume`. Jeżeli RMS danej ramki jest mniejszy niż zadany próg ciszy (`silence_threshold`), ramka jest uznawana za cichą. Metoda łączy kolejne ramki ciche w jeden segment i zwraca listę krotek postaci (start_idx, end_idx) określających przedziały, w których występuje cisza. Druga klasa to `VoicedAudioProcessor`, która ma następującą definicję;

```
class VoicedAudioProcessor(BaseAudioProcessor):

def detect_voiced_unvoiced(self, data, fs, frame_size, vol_threshold
    =0.02, zcr_threshold=0.3,
                           silence_threshold=0.001):

    total_samples = len(data)
    frame_step = frame_size
    results = []
    current_state = None
    start_idx = 0

    for i in range(0, total_samples, frame_step):
        frame = data[i:i + frame_size]
        if len(frame) == 0:
            continue

        # Obliczamy RMS ramki za pomoc  compute_volume
        rms = compute_volume(frame)
        # Je li ramka jest cicha, ko czymy bie  cy segment (
            je li istnieje)
        if rms < silence_threshold:
            if current_state is not None:
                results.append((start_idx, i, current_state))
                current_state = None
            continue

        # Obliczamy ZCR dla niecichych ramek
        zcr_val = compute_zcr(frame)
        # Klasyfikacja: d wi czny , gdy RMS > vol_threshold i ZCR <
            zcr_threshold
        is_voiced = (rms > vol_threshold and zcr_val < zcr_threshold
```

```
                )
            if current_state is None:
                current_state = is_voiced
                start_idx = i
            elif is_voiced != current_state:
                results.append((start_idx, i, current_state))
                current_state = is_voiced
                start_idx = i

        if current_state is not None:
            results.append((start_idx, total_samples, current_state))
        return results
```

Klasa `VoicedAudioProcessor` rozszerza funkcjonalność klasy bazowej o detekcję fragmentów dźwięcznych i bezdźwięcznych. Jednakże, aby analiza była precyzyjna, przetwarzane są jedynie ramki nieciche (czyli te, które mają wartość RMS większą niż zadany próg ciszy). Dla każdej niecichej ramki obliczana jest wartość RMS (z pomocą `compute_volume`) oraz Zero Crossing Rate (z pomocą `compute_zcr`). Jeżeli RMS jest większe niż wartość progowa `vol_threshold` oraz ZCR jest mniejsze od `zcr_threshold`, ramka jest klasyfikowana jako dźwięczna (`is_voiced = True`). W przeciwnym przypadku ramka uznawana jest za bezdźwięczną (`is_voiced = False`). Fragmenty ramek o tej samej klasyfikacji są łączone w jeden segment. Zmiana stanu (z dźwięcznego na bezdźwięczny lub odwrotnie) powoduje zakończenie bieżącego segmentu i zapisanie przedziału (start_idx, end_idx, is_voiced).

3 Prezentacja wyników działania

Po wczytaniu pliku `.wav` użytkownik ma możliwość:

1. Odtworzenia dźwięku oraz zatrzymania go w dowolnym momencie (pauza).
2. Wyświetlenia wykresu przebiegu czasowego z zaznaczonymi:
   - **ciszami** (kolor żółty), przy selekcji *Tryb podświetlania: Cisza*,
   - **obszarami dźwięcznymi** (kolor zielony) oraz **bezdźwięcznymi** (kolor czerwony), przy wybraniu *Tryb podświetlania: Dźwięczne/Bezdźwięczne*.
3. Otworzenia okna z dodatkowymi wykresami cech (**RMS**, **ZCR**, **SR**, **F0 (Autocorr)**, **F0 (AMDF)**). Można zaznaczać, które cechy mają zostać wyświetlone.

3.1 Porównanie męskiego i damskiego głosu

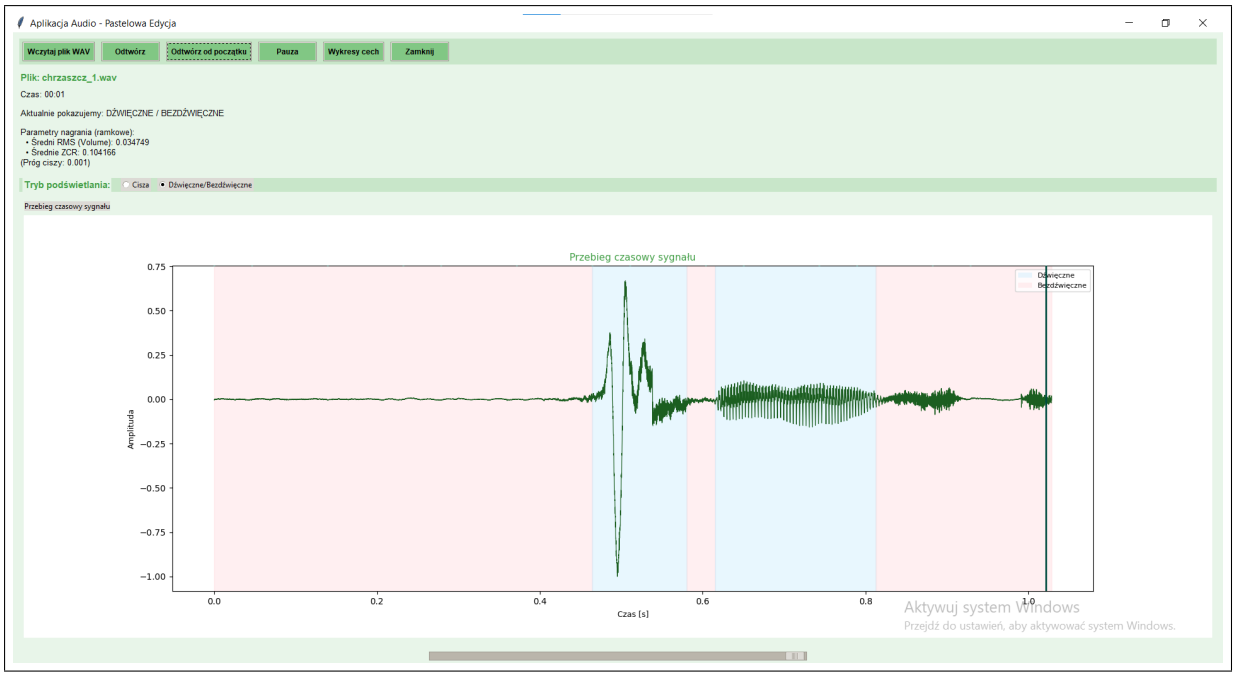
Dla przykładu, na Rysunku 1 przedstawiono fragment ekranu głównego aplikacji z wizualizacją przebiegu czasowego i zaznaczonymi obszarami dźwięcznymi i bezdźwięcznymi.

Rysunek 1: Wykres przebiegu z zaznaczonymi obszarami dźwięcznymi i bezdźwięcznymi dla słowa chrząszcz.

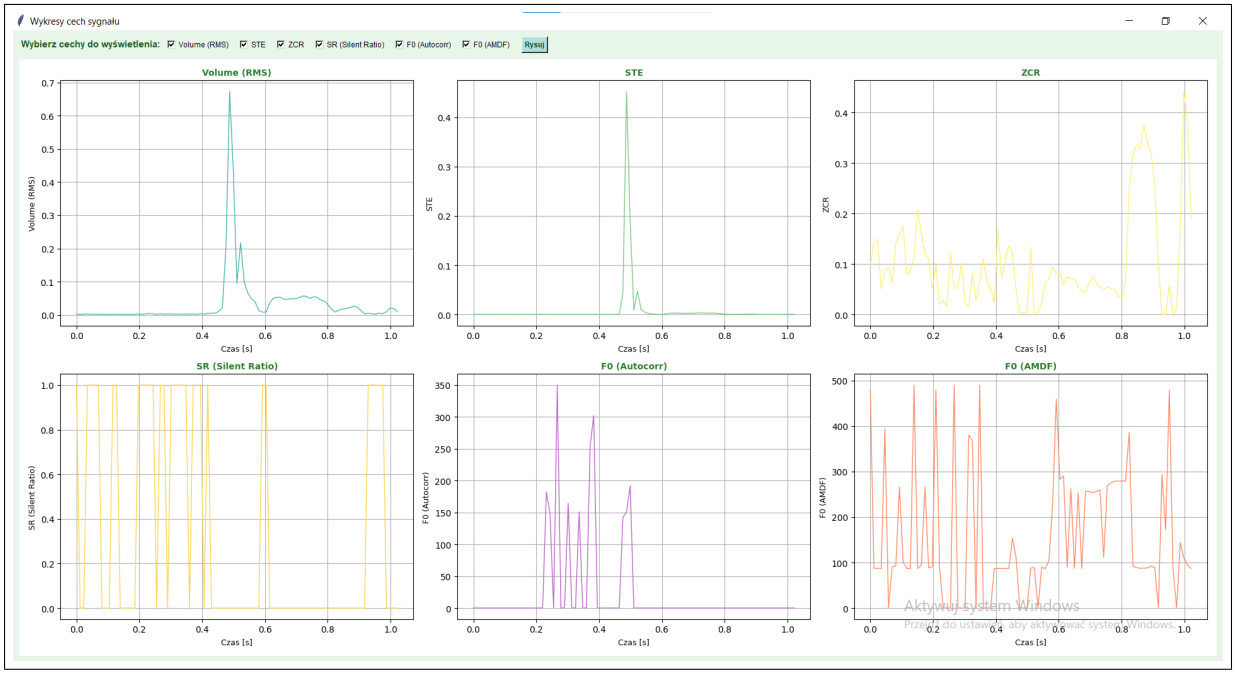
Poddano analizie słowo chrząszcz, które zostało wypowiedziane przez mężczyznę. Po kliknięciu na przycisk "Wykresy cech"otwiera nam się nowe okno, w którym możemy zaobserwować wykresy na Rysunku 2. Okno wykresy cech przedstawia wszystkie podsta-

Rysunek 2: Wykresy cech dla słowa chrząszcz, wypowiedzianego przez mężczyznę.

wowe wykresy dotyczące analizy przetwarzanego sygnału audio. Porównajmy to z tym samym słowem wypowiedzianym przez kobietę, które zostało przedstawione na Rysunku 3. Zerknijmy także na wykresy cech przedstawione na Rysunku 4



Rysunek 3: Wykres przebiegu z zaznaczonymi obszarami dźwięcznymi i bezdźwięcznymi dla słowa chrząszcz wypowiedzianego przez kobietę



Rysunek 4: Wykresy cech dla słowa chrząszcz, wypowiedzianego przez kobietę

3.1.1 Analiza i porównanie wyników

Na podstawie Rysunków 1 i 2 (słowo *chrząszcz* wypowiedziane przez mężczyznę) oraz Rysunków 3 i 4 (ten sam wyraz wypowiedziany przez kobietę) można sformułować następujące wnioski:

- **Różnice w częstotliwości podstawowej (F0):** Z wykresów cech *F0 (Autocorr)* i *F0 (AMDF)* widać, że w przypadku nagrania kobiecego wartości F0 są z reguły wyższe. Jest to zgodne z powszechną obserwacją, że kobiecy głos charakteryzuje się wyższą częstotliwością podstawową niż męski.

- **Głośność (RMS/Volume):** Wykresy *Volume (RMS)* wskazują, że w próbkach męskich amplituda (i tym samym RMS) bywa bardziej zróżnicowana i chwilowo może być większa, co może wynikać z innego sposobu artykulacji oraz siły głosu. Nagranie kobiece jest za to bardziej jednorodne pod względem poziomu głośności.

- **Czas trwania dźwięków dźwięcznych i bezdźwięcznych:** Na głównych wykresach przebiegu (Rys. 1 i 3) widoczne są różnice w długości trwania segmentów oznaczonych jako dźwięczne (kolor zielony) i bezdźwięczne (kolor różowy). U mężczyzny pewne fragmenty dźwięczne mogą być krótsze, za to silniej zarysowane amplitudowo, podczas gdy u kobiety przejścia między fazami dźwięcznymi i bezdźwięcznymi są bardziej płynne.

- **Zero Crossing Rate (ZCR):** Z wykresów ZCR wynika, że w kobiecym nagraniu występuje nieco większe zagęszczenie przejść przez zero, co jest konsekwencją wyższej częstotliwości podstawowej i innej struktury formantowej. U mężczyzny, przy niższej F0, ZCR jest zwykle mniejsze, choć przy spółgłoskach szczelinowych (*/sz/, /cz/*) może okresowo wzrastać.

- **Różnice w czasie artykulacji:** Choć samo słowo *chrząszcz* jest dość krótkie, można zauważyć, że w nagraniu kobiecym poszczególne fazy głosek (zwłaszcza */ch/* i */sz/*) mogą być artykułowane nieco dłużej, przez co całkowity czas wypowiedzi jest odrobinę większy lub bardziej rozciągnięty w czasie. U mężczyzny te przejścia wydają się być bardziej skondensowane.

- **Ogólna charakterystyka widoczna w STE i SR:** Analizując krótkoczasową energię (STE) oraz *Silent Ratio (SR)*, można zauważyć, że u kobiety pojawia się nieco więcej momentów o bardzo niskiej energii (krótkie pauzy lub wyciszenia pomiędzy elementami artykulacyjnymi), co skutkuje większym wskaźnikiem ciszy (*SR*). U mężczyzny cisza jest bardziej skumulowana na początku i końcu wypowiedzi, natomiast sam rdzeń dźwięku (*/chrzą/*) bywa głośniejszy i bardziej zwarty.

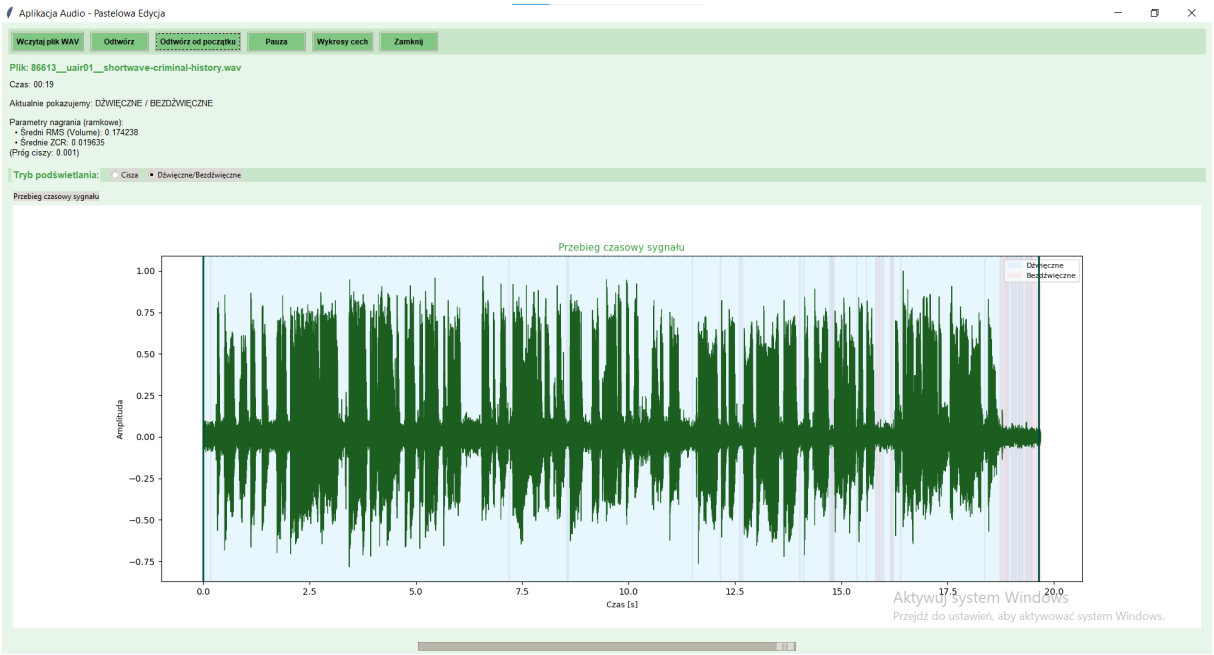
Podsumowując, oba nagrania prezentują oczywiste różnice w wartości F0, głośności i dynamice przejść między dźwiękami dźwięcznymi i bezdźwięcznymi. Jest to zgodne z typowymi cechami kobiecego i męskiego aparatu mowy: wyższa częstotliwość podstawowa u kobiet, mocniejsze zróżnicowanie amplitudy u mężczyzn oraz odmienne proporcje czasowe poszczególnych głosek. Oczywiście należy też pamiętać o sposobie artykulacji a także odległości od mikrofonu w trakcie mówienia, co na pewno wpływa na otrzymane wyniki.

3.2 Porównanie nagrań z radio (mowa i muzyka)

W poniższej sekcji przedstawiono kompleksową analizę nagrań radiowych, obejmującą nie tylko wizualizację przebiegów i cech sygnałów, ale także analizę dodatkowych właściwości, takich jak obecność szumu, dynamika i charakterystyka widmowa. Do porównania wykorzystano dwa przykładowe nagrania:

- **Nagranie mowy:** `radio_mowa.wav` – jest to nagranie krótkofalowej audycji, w której dominują klarowne fragmenty narracji, typowe dla programów opowiadających historie kryminalne.
- **Nagranie muzyczne:** `radio_muzyka.wav` – przykładowy fragment nagrania muzyki, charakteryzujący się obecnością transmisyjnego szumu oraz dynamiczną strukturą utworu.

Na poniższych rysunkach przedstawiono wizualizacje przebiegów czasowych oraz wykresy cech dla obu typów nagrań:

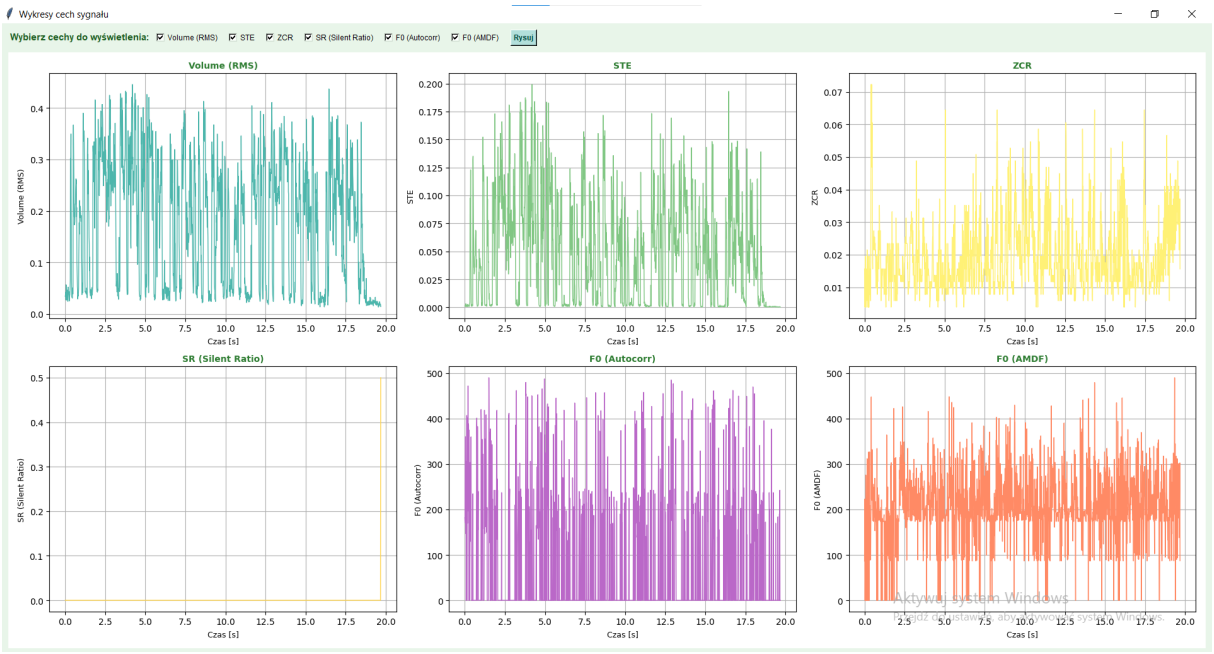


Rysunek 5: Przebieg czasowy nagrania mowy z radia, ukazujący segmentację na fragmenty dźwięczne i bezdźwięczne.

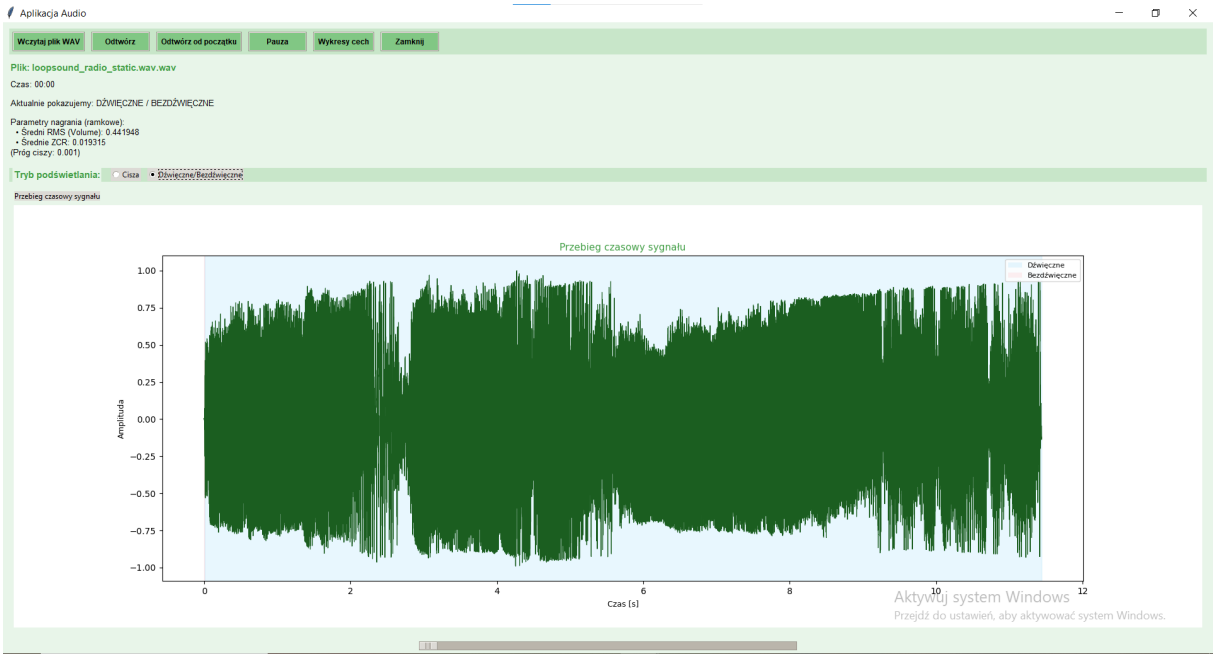
3.2.1 Analiza dodatkowych aspektów nagrań

Oprócz przedstawionych wykresów, szczegółowa analiza nagrań pozwala wysnuć następujące obserwacje:

- **Obecność szumu i charakter transmisji:** Nagranie mowy wykazuje typowy dla krótkofalowych transmisji poziom szumu oraz sporadyczne zakłócenia, które jednak nie zakłócają przejrzystości narracji. W przeciwieństwie do tego, nagranie muzyczne zawiera wyraźnie słyszalny transmisyjny szum, który nadaje mu charakter retro i autentyczny, typowy dla starszych transmisji radiowych.

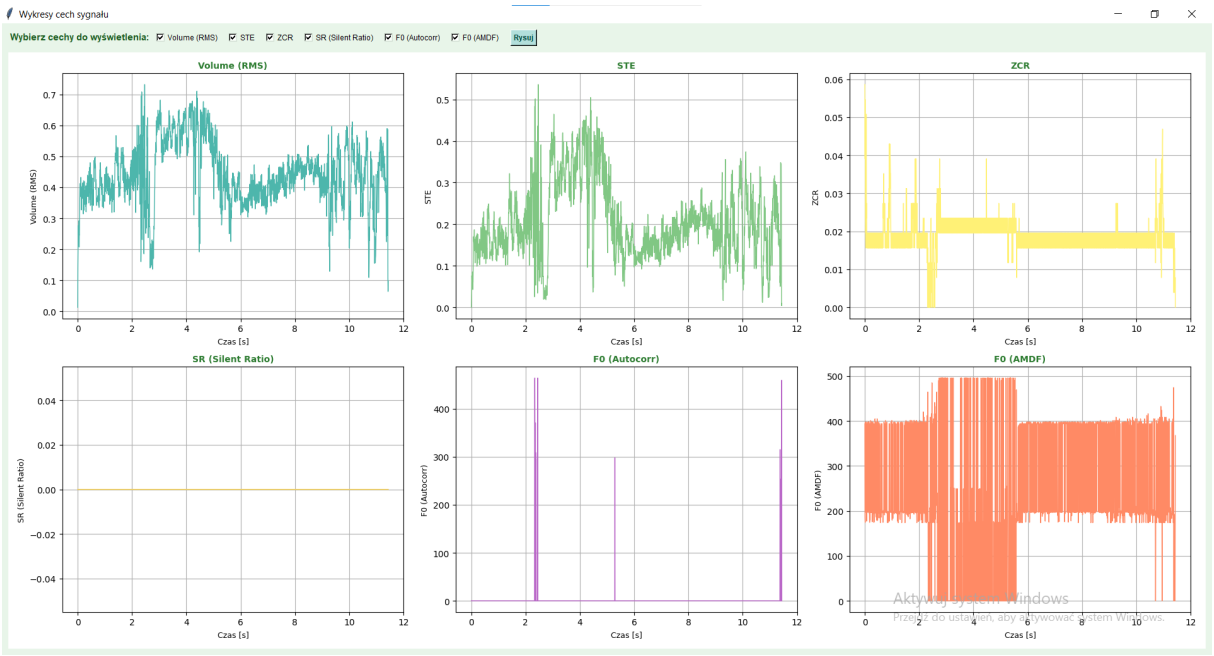


Rysunek 6: Wykresy cech (m.in. RMS, ZCR) dla nagrania mowy, ilustrujące zmienność dynamiki wypowiedzi.



Rysunek 7: Przebieg czasowy nagrania muzycznego z radia, z widocznym wpływem transmisyjnego szumu.

- **Dynamika sygnału:** W przypadku nagrania mowy wykresy cech, takie jak RMS, wskazują na stabilny poziom sygnału z naturalnymi pauzami między fragmentami wypowiedzi. Natomiast w nagraniu muzycznym obserwujemy większe zmiany dynamiki – fragmenty o wysokiej energii przeplatają się z obszarami wyraźnie obciążonymi szumem, co odzwierciedla złożoność struktury muzycznej oraz warunki transmisji.



Rysunek 8: Wykresy cech dla nagrania muzycznego, ukazujące dynamiczne zmiany poziomu energii oraz zmienność harmoniczną utworu.

- **Charakterystyka widmowa i artykulacja:** Analiza wykresów cech takich jak Zero Crossing Rate (ZCR) sugeruje, że nagranie mowy cechuje się bardziej regularnym wzorcem przejść przez zero, co jest zgodne z naturalnym tempem mówienia i wyraźną artykulacją. W nagraniu muzycznym natomiast, nieregularne wzorce ZCR świadczą o obecności wielu harmonicznych oraz o zmienności instrumentów.
- **Wizualna segmentacja sygnałów:** Przebiegi czasowe (Rys. 5 oraz 7) jasno pokazują, że nagranie mowy jest bardziej jednorodne i dobrze segmentowane – fragmenty dźwięczne oraz bezdźwięczne są wyraźnie odseparowane. W przypadku nagrania muzycznego segmentacja jest mniej regularna, co odzwierciedla dynamiczne przejścia między różnymi partiami utworu oraz wpływ szumu transmisyjnego.

3.2.2 Podsumowanie porównania

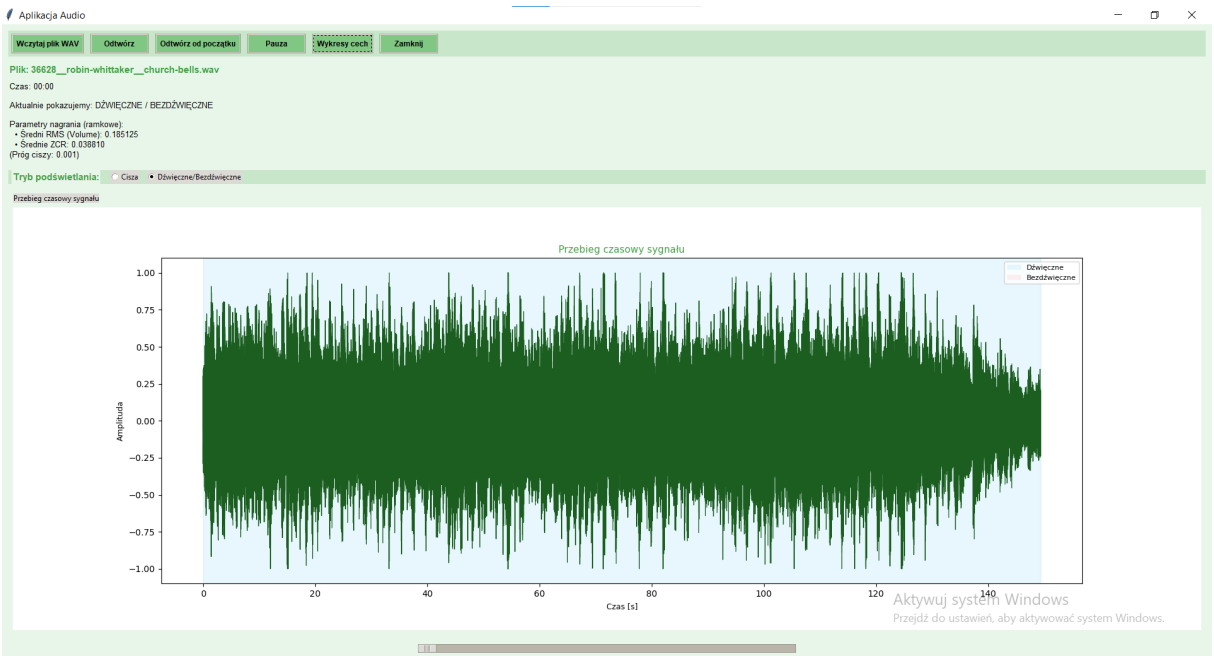
Na podstawie analizy zarówno wykresów, jak i charakterystyki samych nagrań można wyciągnąć następujące wnioski:

1. **Mowa radiowa** cechuje się wyraźną artykulacją, stabilnym poziomem energii oraz regularnymi segmentami, co umożliwia łatwą analizę cech takich jak RMS i ZCR.
2. **Muzyka radiowa** prezentuje większą dynamikę, złożoność harmoniczną oraz charakterystyczny transmisyjny szum, co wpływa na nieregularność wykresów cech i wymaga zastosowania metod redukcji punktów (downsampling) dla uzyskania czytelnych wizualizacji.

3.3 Analiza nagrania muzycznego (dźwięk dzwonów kościelnych)

W tej części przedstawiono analizę nagrania `church-bells.wav`, które zawiera dźwięk dzwonów kościelnych. Na Rysunku 9 zaprezentowano przebieg czasowy sygnału, a na

Rysunku 10 – wykresy wybranych cech (Volume (RMS), STE, ZCR, SR, F0 Autocorr oraz F0 AMDF).



Rysunek 9: Przebieg czasowy nagrania muzycznego (dźwięk dzwonów).

Rysunek 10: Wykresy cech (RMS, STE, ZCR, SR, F0 Autocorr, F0 AMDF) dla nagrania dzwonów.

3.3.1 Charakterystyka przebiegu czasowego

Z Rysunku 9 widać, że sygnał utrzymuje się na względnie wysokim poziomie amplitudy przez cały czas trwania nagrania – nie występują dłuższe fragmenty ciszy. Dźwięk dzwonów charakteryzuje się krótkim, intensywnym atakiem, po którym następuje stopniowe

wybrzmiewanie (rezonans metalowych elementów). Kolejne uderzenia dzwonów nakładają się, powodując warstwowy efekt o bogatej strukturze harmonicznej.

3.3.2 Analiza cech sygnału

Na Rysunku 10 przedstawiono wykresy wybranych parametrów sygnału:

- **Volume (RMS):** Wartość RMS jest dość wysoka i – choć ulega fluktuacjom – nie spada w okolice zera. Świadczy to o ciągłej obecności dźwięku w nagraniu (brak dłuższych pauz).
- **STE (Short Time Energy):** Krótkoczasowa energia sygnału utrzymuje się na podwyższonym poziomie, z okresowymi skokami odpowiadającymi uderzeniom dzwonów. Czas wybrzmiewania dźwięku sprawia, że STE nie wraca szybko do wartości niskich, co wskazuje na rezonans i nakładanie się harmonicznych.
- **ZCR (Zero Crossing Rate):** Dźwięk dzwonów, będący w dużej mierze nieharmonicznym sygnałem z licznymi składowymi częstotliwościowymi, skutkuje umiarkowanym poziomem ZCR. Nie jest on jednak tak niski jak w typowo niskotonowych sygnałach (np. basowych), ale też nie tak wysoki jak w przypadku głośnych głosek szczelinowych w mowie.
- **SR (Silent Ratio):** Wartość *Silent Ratio* nie wzrasta, co potwierdza brak odcinków ciszy. Sygnał jest stale aktywny, a wybrzmiewanie metalicznego dźwięku zapewnia utrzymanie się energii w sygnale.
- **F0 (Autocorr) oraz F0 (AMDF):** Wykresy częstotliwości podstawowej nie wykazują stabilnego, pojedynczego tonu, co wynika z natury dzwonów. Dzwony mają wiele składowych o różnych częstotliwościach (metaliczne alikwoty), więc metody autokorelacji i AMDF nie wyznaczają spójnego *F0*. Zamiast tego pojawiają się nieregularne piki, odzwierciedlające bogate spektrum drgań.

3.3.3 Wnioski końcowe

- **Bogactwo harmoniczne:** Dźwięk dzwonów ma charakter silnie rezonansowy, złożony z wielu składowych częstotliwościowych, co utrudnia detekcję stabilnej częstotliwości podstawowej.
- **Brak ciszy:** Analiza parametrów RMS i SR pokazuje, że w nagraniu nie występują dłuższe fragmenty ciszy – sygnał pozostaje stale aktywny.
- **Wysoka i stabilna energia:** Parametr STE utrzymuje się na względnie wysokim poziomie, co wskazuje na nakładanie się dźwięków i długie wybrzmiewanie dzwonów.

Podsumowując, nagranie dzwonów kościelnych wyróżnia się ciągłą obecnością sygnału o wysokiej amplitudzie, niestabilną (ale bogatą) strukturą harmoniczną oraz brakiem wyraźnych momentów ciszy. Wyniki analizy parametrów (RMS, STE, ZCR, SR, F0) są zgodne z charakterem dźwięków metalicznych, w których wielość nakładających się składowych częstotliwościowych utrudnia wyznaczenie pojedynczej częstotliwości podstawowej.

4 Wnioski i obserwacje

- Aplikacja poprawnie wykrywa ciszę i dzieli sygnał na obszary dźwięczne i bezdźwięczne. Dokładność detekcji zależy jednak od parametrów, takich jak próg `silence_threshold` i wielkość ramki (`frame_size`).
- Metoda autokorelacji i AMDF umożliwiają estymację częstotliwości podstawowej, lecz nie zawsze działają dobrze dla sygnałów z wieloma harmonicznymi czy dla sygnałów mowy o silnym zakłóceniu.
- Parametry takie jak f_{min}, f_{max} w funkcjach do wyznaczania F0 powinny być dostosowane do rodzaju sygnału (mowa, śpiew, instrumenty muzyczne itp.).
- Podczas pracy nad projektem napotkano problemy z odtwarzaniem nagrań. Pierwotna wersja zakładała bardzo dynamiczną aktualizację wykresu, przez co odtwarzanie się zacinało i było bardzo niespójne. W celu rozwiązania tego problemu zastosowano między innymi strumieniowanie, a także ustawienie tła jako stałego, niezmiennego, dzięki czemu odtwarzanie stało się bardziej dynamiczne.
- Napotkano także na problemy z generowaniem wykresów cech dla dłuższych nagrań. Czasem program po prostu się zawieszał, a nawet jeśli udało się wygenerować wykresy to trwało to bardzo długo. Udało się znacznie przyspieszyć ten proces stosując odpowiednie metody w pliku `features_window.py`.
- W ramach rozbudowy można uwzględnić bardziej zaawansowane metody estymacji F0 (np. YIN) czy dodać filtrację sygnału przed detekcją. Oczywiście można też zastosować parametryzację na poziomie klipu, wykrywanie muzyki lub mowy albo możliwość eksportowania pliku audio do innego formatu.