

**Aplikacja Audio w języku Python z interfejsem  
graficznym Tkinter**  
**Dokumentacja i opis projektu**

**Autor:**  
Jan Kowalski

**Data:**  
29 marca 2025

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
<b>2</b>	<b>Opis aplikacji</b>	<b>2</b>
2.1	Struktura projektu i główne moduły . . . . .	2
2.2	Interfejs graficzny (GUI) . . . . .	2
2.3	Opis kodu . . . . .	3
2.3.1	Plik <code>main.py</code> . . . . .	3
2.3.2	Plik <code>audio_app.py</code> . . . . .	3
2.3.3	Plik <code>features.py</code> . . . . .	14
2.3.4	Plik <code>features_window.py</code> . . . . .	17
2.3.5	Plik <code>design.py</code> . . . . .	22
2.3.6	Plik <code>audio_processing.py</code> . . . . .	22
<b>3</b>	<b>Prezentacja wyników działania</b>	<b>24</b>
3.1	Porównanie męskiego i damskiego głosu . . . . .	24
3.1.1	Analiza i porównanie wyników . . . . .	27
3.2	Porównanie nagrań z radio (mowa i muzyka) . . . . .	28
3.2.1	Analiza dodatkowych aspektów nagrań . . . . .	28
3.2.2	Podsumowanie porównania . . . . .	30
3.3	Analiza nagrania muzycznego (dźwięk dzwonów kościelnych) . . . . .	30
3.3.1	Charakterystyka przebiegu czasowego . . . . .	31
3.3.2	Analiza cech sygnału . . . . .	32
3.3.3	Wnioski końcowe . . . . .	32
<b>4</b>	<b>Wnioski i obserwacje</b>	<b>33</b>

# 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 wspomnianych cech dla poszczególnych ramek.
- `design.py` – moduł zawierający klasę `ColorScheme` i funkcję `configure_style`, definiującą 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 żółty) 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:

```

1 def main():
2     root = Tk()
3     root.title("AudioApp")
4     root.geometry("900x700")
5
6     app = AudioApp(root)
7     root.protocol("WM_DELETE_WINDOW", app.on_close)
8     root.mainloop()
9     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:

```

1 def __init__(self, master):
2     self.master = master
3
4     # Konfigurujemy styl
5     self.style = ttk.Style()
6     configure_style(self.style)
7
8     # Klasa do przetwarzania audio (analizy ciszy, dźwięczności
9     # itd.)
10    self.processor = VoicedAudioProcessor()
11
12    self.master.title("Aplikacja Audio")
13    self.master.geometry("900x700")
14
15    # Zmienne audio
16    self.fs = None
17    self.data = None
18    self.total_samples = 0
19    self.time_array = None
20    self.current_index = 0
21    self.filename = ""

```

```

22     # Flagi sterowania
23     self.playing = False
24     self.paused = False
25
26     # Strumień 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 podświetlania
34     self.highlight_mode = tk.StringVar(value="silence") #
        domyślnie "silence"
35
36     # Referencja do pionowej linii
37     self.line = None
38
39     # Blitting statyczne tło wykresu
40     self.background = None
41
42     # Główna 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     # Wywołujemy pętli 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)

```

Konstruktor ustawia główny interfejs użytkownika, a także styl. Inicjalizuje kluczowe zmienne związane z przetwarzaniem dźwięku, tworzy także instancje klasy `VoicedAudioProcessor`, 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         # --- Główny 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="Odtwórz",
16            command=self.play_audio,

```

```

17         state="disabled"
18     )
19     self.play_button.grid(row=0, column=1, padx=5, pady=5)
20
21     # Odtwarzanie od pocz tku
22     self.play_from_start_button = ttk.Button(
23         self.top_frame,
24         text="Odtw rz od pocz tku",
25         command=self.play_from_start,
26         state="disabled"
27     )
28     self.play_from_start_button.grid(row=0, column=2, padx=5, pady
    =5)
29
30     self.pause_button = ttk.Button(
31         self.top_frame,
32         text="Pauza",
33         command=self.toggle_pause,
34         state="disabled"
35     )
36     self.pause_button.grid(row=0, column=3, padx=5, pady=5)
37
38     self.features_button = ttk.Button(
39         self.top_frame,
40         text="Wykresy cech",
41         command=self.open_features_window,
42         state="disabled"
43     )
44     self.features_button.grid(row=0, column=4, padx=5, pady=5)
45
46     self.close_button = ttk.Button(
47         self.top_frame,
48         text="Zamknij",
49         command=self.on_close
50     )
51     self.close_button.grid(row=0, column=5, padx=5, pady=5)
52
53     # --- Sekcja info: nazwa pliku, czas, tryb ---
54     info_frame = ttk.Frame(self.main_frame, style="App.TFrame")
55     info_frame.pack(side="top", fill="x", padx=10, pady=(0, 5))
56
57     self.file_label = ttk.Label(
58         info_frame,
59         text="Brak wczytanego pliku",
60         style="TitleLabel.TLabel"
61     )
62     self.file_label.pack(side="top", anchor="w")
63
64     self.time_label = ttk.Label(
65         info_frame,
66         text="Czas: 00:00"
67     )
68     self.time_label.pack(side="top", anchor="w", pady=5)
69
70     # Etykieta z aktualnym trybem
71     self.highlight_label = ttk.Label(
72         info_frame,
73         text="Aktualnie pokazujemy: CISZ "

```

```

74     )
75     self.highlight_label.pack(side="top", anchor="w", pady=5)
76
77     # Suwak (inicjalizujemy po wczytaniu pliku)
78     self.slider = None
79
80     # Tekst z parametrami
81     self.frame_params_text = tk.StringVar()
82     self.params_label = ttk.Label(
83         info_frame,
84         textvariable=self.frame_params_text,
85         justify="left"
86     )
87     self.params_label.pack(side="top", anchor="w", pady=5)
88
89     # --- Opcje wyboru trybu pod wietlania ---
90     mode_frame = ttk.Frame(self.main_frame, style="Controls.TFrame")
91     mode_frame.pack(side="top", fill="x", padx=10, pady=5)
92
93     ttk.Label(
94         mode_frame,
95         text="Tryb pod wietlania:",
96         style="TitleLabel.TLabel"
97     ).pack(side="left", padx=(5, 10))
98
99     rb_silence = ttk.Radiobutton(
100         mode_frame,
101         text="Cisza",
102         variable=self.highlight_mode,
103         value="silence",
104         command=self.update_highlight_mode
105     )
106     rb_silence.pack(side="left", padx=5)
107
108     rb_voiced = ttk.Radiobutton(
109         mode_frame,
110         text="D wi czne/Bez d wi czne",
111         variable=self.highlight_mode,
112         value="voiced_unvoiced",
113         command=self.update_highlight_mode
114     )
115     rb_voiced.pack(side="left", padx=5)
116
117     # --- Ramka z wykresem audio ---
118     plot_frame = ttk.LabelFrame(
119         self.main_frame,
120         text="Przebieg czasowy sygnału",
121         style="App.TFrame"
122     )
123     plot_frame.pack(fill="both", expand=True, padx=10, pady=10)
124
125     self.fig, self.ax = plt.subplots(figsize=(8, 3))
126     self.canvas = FigureCanvasTkAgg(self.fig, master=plot_frame)
127     self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True, padx=
128         =5, pady=5)
129
130     # Podpięcie obsługi zdarzenia zmiany rozmiaru wykresu
131     self.canvas.mpl_connect('resize_event', self.on_resize)

```

Główny 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:
5                 CISZ ")
6         else:
7             self.highlight_label.config(text="Aktualnie pokazujemy:
8                 D WI CZNE / BEZD WI CZNE")
9
10        # Przerysuj wykres w nowym trybie
11        if self.data is not None:
12            self.draw_main_plot()

```

Metoda odpowiada za aktualizację wykresów po zmianie trybu wyświetlania detekcji ciszy bądź detekcji fragmentów dźwięcznych i bezdźwięcznych. Następną metodą 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("Błąd", f"Nie udało się wczytać
16                pliku WAV: {e}")
17            return
18
19        if len(raw_data.shape) > 1:
20            raw_data = raw_data[:, 0]
21
22        raw_data = raw_data.astype(np.float32)
23        peak = np.max(np.abs(raw_data))

```



```

23     if peak > 1e-9:
24         raw_data /= peak
25
26     self.data = raw_data
27     self.total_samples = len(self.data)
28     duration = self.total_samples / self.fs if self.fs else 0.001
29     self.time_array = np.linspace(0, duration, self.total_samples)
30     self.current_index = 0
31
32     # Rysujemy g wny wykres
33     self.draw_main_plot()
34     self.calculate_and_display_frame_params()
35
36     # Tworzymy / aktualizujemy suwak
37     if not self.slider:
38         self.slider = ttk.Scale(
39             self.main_frame,
40             from_=0,
41             to=max(duration, 0.001),
42             orient="horizontal",
43             command=self.on_slider_move,
44             length=600
45         )
46         self.slider.pack(pady=5)
47     else:
48         self.slider.config(to=duration)
49         self.slider.set(0)
50
51     # W czamy przyciski
52     self.play_button.state(["!disabled"])
53     self.pause_button.state(["!disabled"])
54     self.features_button.state(["!disabled"])
55     self.play_from_start_button.state(["!disabled"])
56
57     # Zatrzymujemy poprzedni strumie (je li by )
58     self.stop_audio()
59
60     # Tworzymy nowy strumie audio
61     self.stream = sd.OutputStream(
62         samplerate=self.fs,
63         blocksize=1024,
64         channels=1,
65         dtype='float32',
66         callback=self.audio_callback
67 )

```

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)`:

```

1     def draw_main_plot(self):
2         self.ax.clear()
3         self.ax.set_title("Przebieg_czasowy_sygna u", fontsize=11,
4                             color=ColorScheme.ACCENT)
5         self.ax.set_xlabel("Czas[s]", fontsize=9)
6         self.ax.set_ylabel("Amplituda", fontsize=9)
7         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.
16               silence_threshold
17       )
18       for (start_idx, end_idx) in silence_regions:
19           start_t = start_idx / self.fs
20           end_t = end_idx / self.fs
21           self.ax.axvspan(start_t, end_t, color=ColorScheme.
22               SILENCE_COLOR, alpha=0.6)
23
24       # Patch do legendy
25       silence_patch = Patch(facecolor=ColorScheme.SILENCE_COLOR,
26           alpha=0.6, label="Cisza")
27       legend_patches.append(silence_patch)
28
29   else:
30       # Zaznaczamy d wi czne/bezd wi czne
31       vu_regions = self.processor.detect_voiced_unvoiced(self.data
32           , self.fs, self.frame_size)
33       for (start_idx, end_idx, is_voiced) in vu_regions:
34           start_t = start_idx / self.fs
35           end_t = end_idx / self.fs
36           if is_voiced:
37               self.ax.axvspan(start_t, end_t, color=ColorScheme.
38                   VOICED_COLOR, alpha=0.3)
39           else:
40               self.ax.axvspan(start_t, end_t, color=ColorScheme.
41                   UNVOICED_COLOR, alpha=0.3)
42
43       # Patche do legendy
44       voiced_patch = Patch(facecolor=ColorScheme.VOICED_COLOR,
45           alpha=0.3, label="D wi czne")
46       unvoiced_patch = Patch(facecolor=ColorScheme.UNVOICED_COLOR,
47           alpha=0.3, label="Bezd wi czne")
48       legend_patches.extend([voiced_patch, unvoiced_patch])
49
50   # Linia aktualnej pozycji
51   self.line = self.ax.axvline(x=0, color="#004D40", linewidth=2)
52
53   # Dodajemy legend (je li cokolwiek zaznaczamy)
54   if legend_patches:
55       self.ax.legend(handles=legend_patches, loc="upper_right",
56           fontsize=8)
57
58   self.canvas.draw()
59
60   # Po narysowaniu wykresu usuwamy background,
61   # kt ry zostanie zaktualizowany przy starcie odtwarzania
62   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. Kolejną metodą 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). Następną metodą 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. Kolejną metodą 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"
19         f"    redni RMS (Volume): {avg_rms:.6f}\n"
20         f"    rednie ZCR: {avg_zcr:.6f}\n"
21         f"(Pr g ciszy: {self.silence_threshold})"

```

```

22         )
23         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)`:

```

1     def audio_callback(self, outdata, frames, time_info, status):
2         if status:
3             #print("Status audio_callback:", status, flush=True)
4             pass
5
6         if (self.data is None) or (not self.playing) or (self.paused):
7             outdata.fill(0)
8             return
9
10        end_index = self.current_index + frames
11        if end_index > self.total_samples:
12            end_index = self.total_samples
13
14        chunk = self.data[self.current_index:end_index]
15        out_len = len(chunk)
16        if out_len < frames:
17            outdata[:out_len, 0] = chunk
18            outdata[out_len:, 0] = 0
19            self.playing = False
20            self.current_index = self.total_samples
21        else:
22            outdata[:, 0] = chunk
23            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)`

```

1     def play_audio(self):
2         if self.data is None or not self.stream:
3             return
4
5         # Je eli linia nie istnieje, tworzymy j
6         if self.line is None:
7             self.line = self.ax.axvline(x=0, color="#004D40", linewidth
8                 =2)
9             # Najpierw p e ne rysowanie wykresu
10            self.canvas.draw()
11            # Dopiero teraz pobieramy background
12            self.background = self.canvas.copy_from_bbox(self.ax.bbox)
13
14            # Je eli dotarli my do ko ca wr my na pocz tek
15            if self.current_index >= self.total_samples:
16                self.current_index = 0
17                if self.slider:
18                    self.slider.set(0)
19                # Reset linii do 0
20                if self.line:
21                    self.line.set_xdata([0, 0])

```

```

21         self.canvas.draw()
22         self.background = self.canvas.copy_from_bbox(self.ax.
           bbox)
23
24     self.playing = True
25     self.paused = False
26     self.pause_button.config(text="Pauza")
27
28     # Start strumienia
29     if not self.stream.active:
30         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)`

```

1     def play_from_start(self):
2         self.current_index = 0
3         if self.slider:
4             self.slider.set(0)
5         if self.line:
6             self.line.set_xdata([0, 0])
7             self.canvas.draw()
8         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)`

```

1     def stop_audio(self):
2         self.playing = False
3         self.paused = False
4         if self.stream and self.stream.active:
5             self.stream.stop()
6
7         # Usuwamy linię z osi, je li istnieje
8         if self.line:
9             self.line.remove()
10            self.line = None
11
12        # Od wie wykres (linia znika)
13        self.canvas.draw()
14        self.background = None

```

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

```

1     def toggle_pause(self):
2         if not self.playing:
3             return
4         if self.paused:
5             self.pause_button.config(text="Pauza")
6             self.paused = False
7             self.play_audio()
8         else:
9             self.pause_button.config(text="Wzn w")

```

```

10         self.paused = True
11         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)`

```

1     def update_ui(self):
2         if self.data is not None and self.playing and not self.paused:
3             current_time = self.current_index / self.fs if self.fs else
4                 0
5
6             # Ustawiamy suwak
7             if self.slider:
8                 self.slider.set(current_time)
9
10            # Aktualizujemy etykiet z czasem
11            self.update_time_label(current_time)
12
13            # Przesuwamy lini (blitowanie)
14            if self.line and self.background:
15                self.canvas.restore_region(self.background)
16                self.line.set_xdata([current_time, current_time])
17                self.ax.draw_artist(self.line)
18                self.canvas.blit(self.ax.bbox)
19
20            # Wywołanie za 50 ms ponownie
21            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(self)` która ma następującą definicję:

```

1     def open_features_window(self):
2         if self.data is None:
3             messagebox.showwarning("Brak danych", "Najpierw wczytaj plik
4                 WAV!")
5             return
6         FeaturesWindow(self.master, self.data, self.fs, self.frame_size,
7             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)`

```

1     def on_close(self):
2         self.stop_audio()
3         if self.ui_after is not None:
4             self.master.after_cancel(self.ui_after)
5             self.ui_after = None
6         if self.stream:
7             self.stream.close()
8             self.stream = None
9         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

$$\text{RMS} = \sqrt{\frac{1}{N} \sum_{n=0}^{N-1} x[n]^2}. \quad (1)$$

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:

$$\text{STE} = \frac{1}{N} \sum_{n=0}^{N-1} x[n]^2. \quad (2)$$

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:

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

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:

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

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

```
1 def compute_autocorr_f0(frame, fs, fmin=50, fmax=500):  
2     if len(frame) == 0:  
3         return 0
```

```

4     frame = frame - np.mean(frame)
5     corr = np.correlate(frame, frame, mode='full')
6     corr = corr[len(corr)//2:]
7     d = np.diff(corr)
8     start = np.nonzero(d > 0)[0]
9     if len(start) == 0:
10        return 0
11    lag = start[0]
12    if lag == 0:
13        return 0
14    f0 = fs / lag
15    if f0 < fmin or f0 > fmax:
16        return 0
17    return f0

```

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

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

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

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

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

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

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:

$$f_0 = \frac{f_s}{\text{lag}}, \quad (8)$$

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)`:

```

1     def compute_amdf(frame):
2         length = len(frame)
3         if length == 0:
4             return np.array([0])
5         amdf = []
6         for tau in range(length):
7             diff_sum = 0.0
8             for n in range(length - tau):
9                 diff_sum += abs(frame[n] - frame[n + tau])
10            amdf.append(diff_sum / (length - tau))
11        return np.array(amdf)

```

Oblicza wartość funkcji AMDF dla każdego opóźnienia  $\tau$ , gdzie:

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

Ostatnia funkcja w tym pliku to `compute_amdf_f0`:



```

1  def compute_amdf_f0(frame, fs, fmin=50, fmax=500):
2      length = len(frame)
3      if length == 0:
4          return 0
5      frame = frame - np.mean(frame)
6      amdf_values = compute_amdf(frame)
7
8      min_lag = int(fs // fmax)
9      max_lag = int(fs // fmin) if fmin != 0 else length // 2
10     if max_lag > len(amdf_values):
11         max_lag = len(amdf_values) - 1
12     if min_lag < 1 or min_lag >= max_lag:
13         return 0
14
15     lag_range = np.arange(min_lag, max_lag)
16     best_lag = lag_range[np.argmin(amdf_values[min_lag:max_lag])]
17     if best_lag == 0:
18         return 0
19     f0 = fs / best_lag
20     if f0 < fmin or f0 > fmax:
21         return 0
22     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:

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

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

$$f_0 = \frac{f_s}{\tau_{\text{best}}}. \quad (10)$$

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)`:

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

```
1 def downsample_block(x, y, max_points=2000):
2     n = len(x)
3     if n <= max_points:
4         return x, y
5     block_size = int(np.ceil(n / max_points))
6     # Podziel dane na bloki
7     x_blocks = [x[i*block_size : (i+1)*block_size] for i in range(int(np
8         .ceil(n / block_size)))]
9     y_blocks = [y[i*block_size : (i+1)*block_size] for i in range(int(np
10        .ceil(n / block_size)))]
11     x_ds = np.array([np.mean(block) for block in x_blocks])
12     y_ds = np.array([np.mean(block) for block in y_blocks])
13     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

```
1 def __init__(self, master, data, fs, frame_size, silence_threshold):
2     self.top = tk.Toplevel(master)
3     self.top.title("Wykresy cech sygnału")
4     self.top.geometry("1000x800")
5
6     # Pastelowe tło
7     self.main_bg_color = ColorScheme.MAIN_BG
8     self.top.configure(bg=self.main_bg_color)
9
10    # Główna ramka
11    self.main_frame = tk.Frame(self.top, bg=self.main_bg_color)
12    self.main_frame.pack(fill="both", expand=True)
13
14    self.data = data
15    self.fs = fs
16
17    # Automatyczne dobranie rozmiaru ramki dla długich nagrań
18    # zwróćmy j, aby liczba ramek nie była zbyt duża.
19    candidate = auto_frame_size(len(data))
20    self.frame_size = min(candidate, frame_size) # wybieramy
21    # większą z tych wartości
22    self.silence_threshold = silence_threshold
23
24    # Dzielimy sygnał na ramki i obliczamy cechy
25    self.frames, self.times = self.frame_signal(data, fs, self.
26        frame_size)
27    self.volume = np.array([compute_volume(f) for f in self.frames])
28    self.ste = np.array([compute_ste(f) for f in self.frames])
29    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 określa redni poziom sygnału (RMS).",
36         "#4DB6AC"
37     ),
38     "STE": (
39         self.ste,
40         "Short Time Energy rozróżnienie fragmentów
41         dwuczynnych/bezdwuczynnych.",
42         "#81C784"
43     ),
44     "ZCR": (
45         self.zcr,
46         "Zero Crossing Rate liczba przejść przez zero.",
47         "#FFF176"
48     ),
49     "SR_(Silent_Ratio)": (
50         self.sr,
51         "1 oznacza ramkę sklasyfikowaną jako cisza.",
52         "#FFD54F"
53     ),
54     "F0_(Autocorr)": (
55         self.f0_autocorr,
56         "Częstotliwość podstawowa - metoda autokorelacji.",
57         "#BA68C8"
58     ),
59     "F0_(AMDF)": (
60         self.f0_amdf,
61         "Częstotliwość podstawowa - metoda AMDF.",
62         "#FF8A65"
63     ),
64 }
65
66 # Panel wyboru cech
67 self.select_frame = tk.Frame(self.main_frame, bg=self.
    main_bg_color)
68 self.select_frame.pack(side="top", fill="x", padx=10, pady=10)
69
70 tk.Label(
71     self.select_frame,
72     text="Wybierz cechy do wyświetlenia:",
73     bg=self.main_bg_color,
74     fg="#1B5E20",
75     font=("Helvetica", 11, "bold")
76 ).pack(side="left", anchor="n")
77
78 self.feature_vars = {}
79 for feat_name in self.features_info.keys():
    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)`:

```

1     def frame_signal(self, data, fs, frame_size):
2         total_samples = len(data)
3         num_frames = int(np.ceil(total_samples / frame_size))

```

```

4     frames = []
5     times = []
6     for i in range(num_frames):
7         start = i * frame_size
8         end = start + frame_size
9         frame = data[start:end]
10        if len(frame) < frame_size:
11            frame = np.pad(frame, (0, frame_size - len(frame)), mode
12                           = 'constant')
13        frames.append(frame)
14        times.append(start / fs)
15    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)`:

```

1     def draw_selected_features(self):
2         selected_features = [name for name, var in self.feature_vars.
3                               items() if var.get()]
4
5         if self.canvas:
6             self.canvas.get_tk_widget().destroy()
7             self.canvas = None
8         if self.fig:
9             plt.close(self.fig)
10            self.fig = None
11
12        if not selected_features:
13            info_label = tk.Label(
14                self.plot_frame,
15                text="Nie wybrano żadnych cech do wywietlenia!",
16                bg=self.main_bg_color,
17                fg="#B71C1C",
18                font=("Helvetica", 12, "bold")
19            )
20            info_label.pack()
21            return
22
23        self.fig = plt.Figure(figsize=(12, 6), dpi=100)
24        self.fig.set_tight_layout(True)
25
26        n_feats = len(selected_features)
27        rows, cols = self.calc_subplot_grid(n_feats)
28
29        # Dla każdej cechy agregujemy dane, by rysować mniej punktów
30        for i, feat_name in enumerate(selected_features, start=1):
31            ax = self.fig.add_subplot(rows, cols, i)
32            data_array, description, color_line = self.features_info[
33                feat_name]
34            # Używamy funkcji downsample_block, by zredukować liczbę
35            # punktów
36            x_plot, y_plot = downsample_block(self.times, data_array,
37                                              max_points=2000)
38            ax.plot(x_plot, y_plot, linewidth=1.0, color=color_line,
39                  rasterized=True)
40            ax.set_title(feat_name, fontsize=10, fontweight="bold",

```

```

36         color="#2E7D32")
37     ax.set_xlabel("Czas[s]", fontsize=9)
38     ax.set_ylabel(feats_name, fontsize=9)
39     ax.grid(True)
40
41     self.canvas = FigureCanvasTkAgg(self.fig, master=self.plot_frame)
42     self.canvas.draw()
43     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 subplotcie 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:

```
1  MAIN_BG = "#E8F5E9"
2  FRAME_BG = "#C8E6C9"
3  ACCENT = "#43A047"
4  WAVEFORM_COLOR = "#1B5E20"
5  SILENCE_COLOR = "#FFECB3"
6  VOICED_COLOR = "#B3E5FC"
7  UNVOICED_COLOR = "#FFCDD2"
```

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

```
1  def configure_style(style: ttk.Style):
2      style.theme_use("clam")
3      style.configure("App.TFrame", background=ColorScheme.MAIN_BG)
4      style.configure("Controls.TFrame", background=ColorScheme.
5          FRAME_BG)
6      style.configure("TLabel",
7          background=ColorScheme.MAIN_BG,
8          font=("Helvetica", 10))
9      style.configure("titleLabel.TLabel",
10         background=ColorScheme.MAIN_BG,
11         font=("Helvetica", 11, "bold"),
12         foreground=ColorScheme.ACCENT)
13
14     style.configure("TButton",
15         font=("Helvetica", 9, "bold"),
16         padding=5)
17     style.map("TButton",
18         foreground=[("active", "#212121"), ("disabled", "#999999")],
19         background=[("active", "#A5D6A7"), ("!active", "#81C784")])
```

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:

```
1  class BaseAudioProcessor:
2
3      def detect_silence(self, data, fs, frame_size, silence_threshold):
4          total_samples = len(data)
5          frame_step = frame_size
6          silence_regions = []
7          in_silence = False
8          start_silence = 0
9          for i in range(0, total_samples, frame_step):
10             frame = data[i:i + frame_size]
11             if len(frame) == 0:
12                 continue
13             # Używamy compute_volume do obliczenia RMS
14             rms = compute_volume(frame)
15             if rms < silence_threshold:
```

```

16         if not in_silence:
17             in_silence = True
18             start_silence = i
19         else:
20             if in_silence:
21                 silence_regions.append((start_silence, i))
22                 in_silence = False
23     if in_silence:
24         silence_regions.append((start_silence, total_samples))
25     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;

```

1     class VoicedAudioProcessor(BaseAudioProcessor):
2
3     def detect_voiced_unvoiced(self, data, fs, frame_size, vol_threshold
4         =0.02, zcr_threshold=0.3,
5         silence_threshold=0.001):
6
7         total_samples = len(data)
8         frame_step = frame_size
9         results = []
10        current_state = None
11        start_idx = 0
12
13        for i in range(0, total_samples, frame_step):
14            frame = data[i:i + frame_size]
15            if len(frame) == 0:
16                continue
17
18            # Obliczamy RMS ramki za pomoc compute_volume
19            rms = compute_volume(frame)
20            # Je li ramka jest cicha, ko czymy bie cy segment (
21                je li istnieje)
22            if rms < silence_threshold:
23                if current_state is not None:
24                    results.append((start_idx, i, current_state))
25                    current_state = None
26                continue
27
28            # Obliczamy ZCR dla niecichych ramek
29            zcr_val = compute_zcr(frame)
30            # Klasyfikacja: d wi czny , gdy RMS > vol_threshold i ZCR <
31                zcr_threshold
32            is_voiced = (rms > vol_threshold and zcr_val < zcr_threshold
33                )
34            if current_state is None:
35                current_state = is_voiced
36                start_idx = i
37            elif is_voiced != current_state:

```



```

34         results.append((start_idx, i, current_state))
35         current_state = is_voiced
36         start_idx = i
37
38     if current_state is not None:
39         results.append((start_idx, total_samples, current_state))
40     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 cisy). 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`).

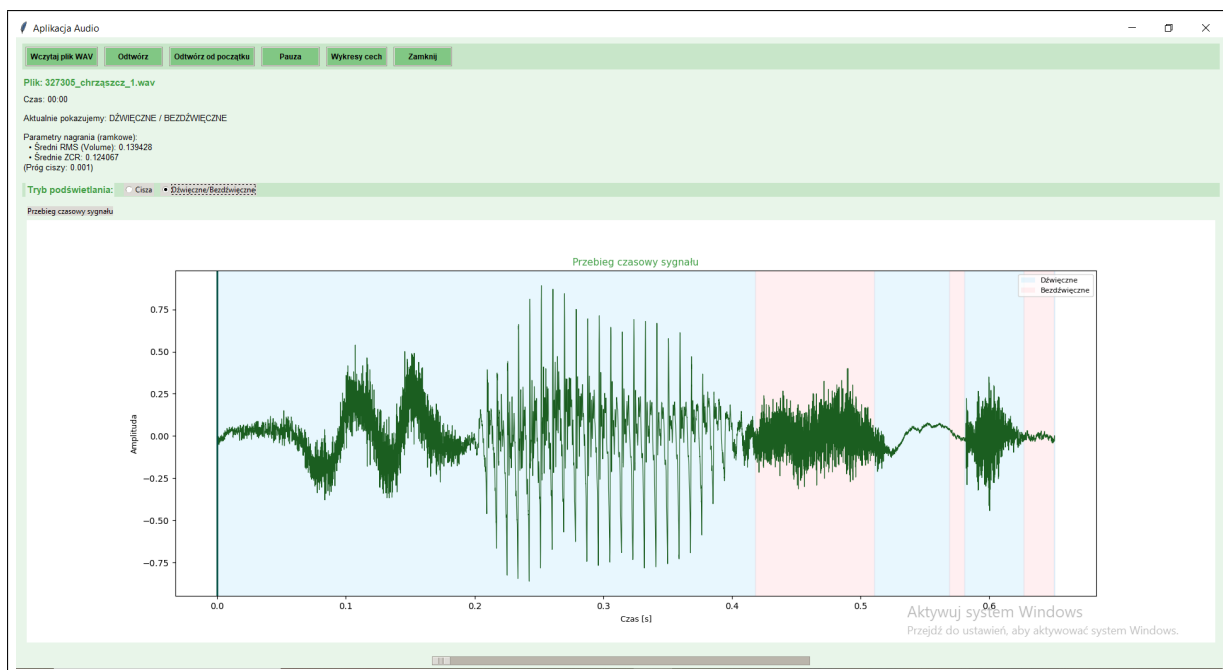
### 3 Prezentacja wyników działania

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 (Auto-corr)**, **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.



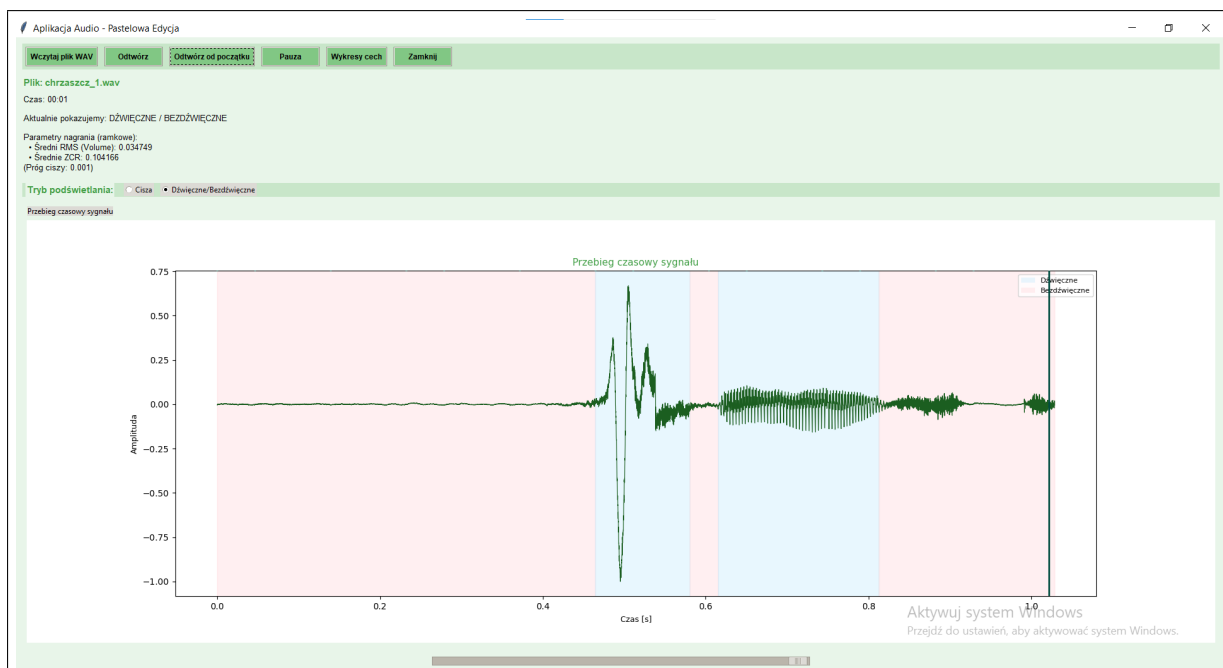
Rysunek 1: Wykres przebiegu z zaznaczonymi obszarami dźwięcznymi i bezdźwięcznymi dla słowa chrząszcz.

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-

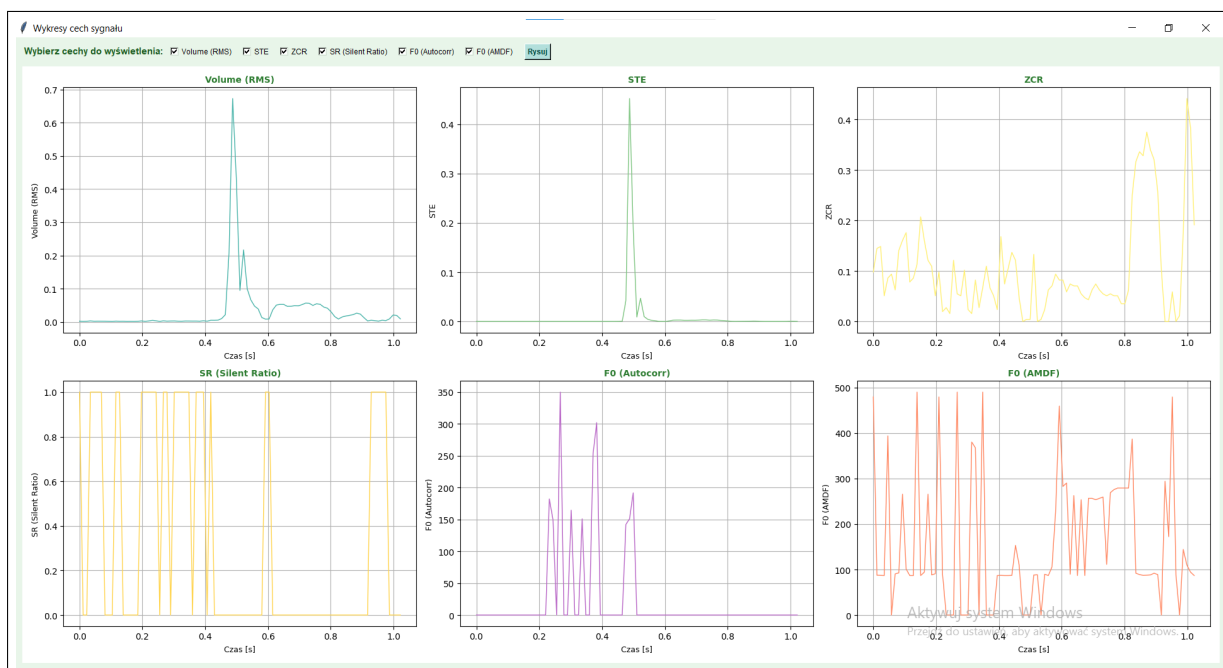


Rysunek 2: Wykresy cech dla słowa chrząszcz, wypowiedzianego przez mężczyznę.

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 (/chrząq/) bywa głośniejszy i bardziej zwarty.

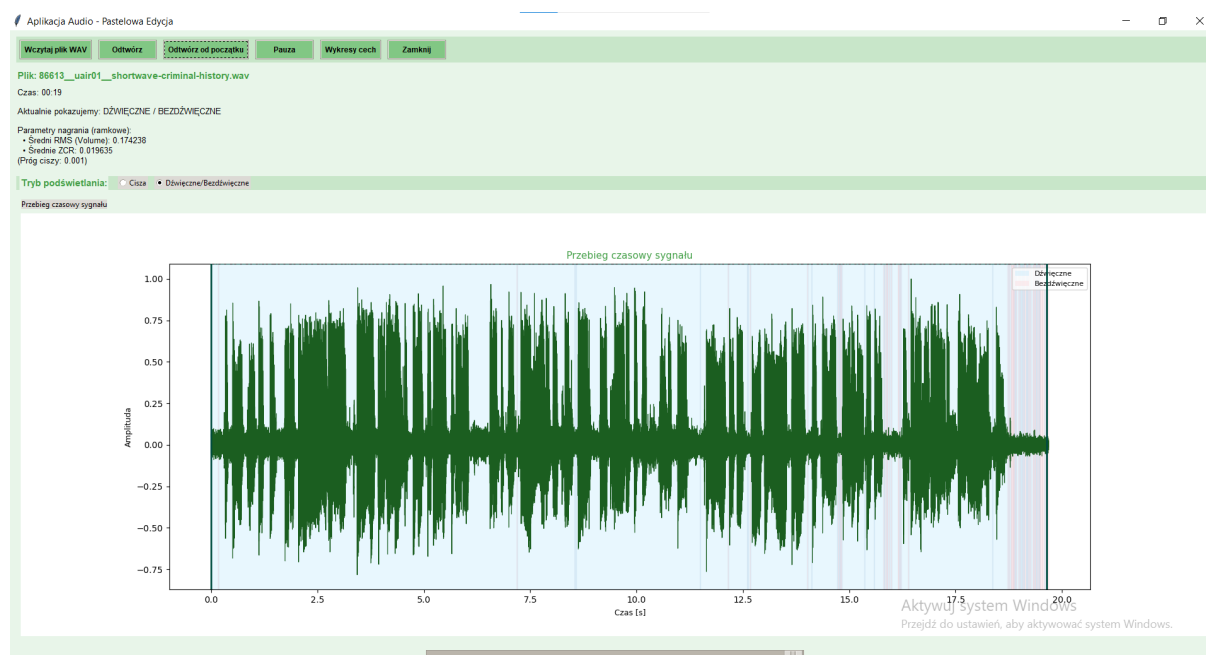
Podsumowując, oba nagrania prezentują oczywiste różnice w wartości F0, głośności i dynamice przejść między dźwiękami dźwięcznymi i bezdźwięcznymi. Jest to zgodne z typowymi cechami kobiecego i męskiego aparatu mowy: wyższa częstotliwość podstawowa u kobiet, mocniejsze zróżnicowanie amplitudy u mężczyzn 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 fra-

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

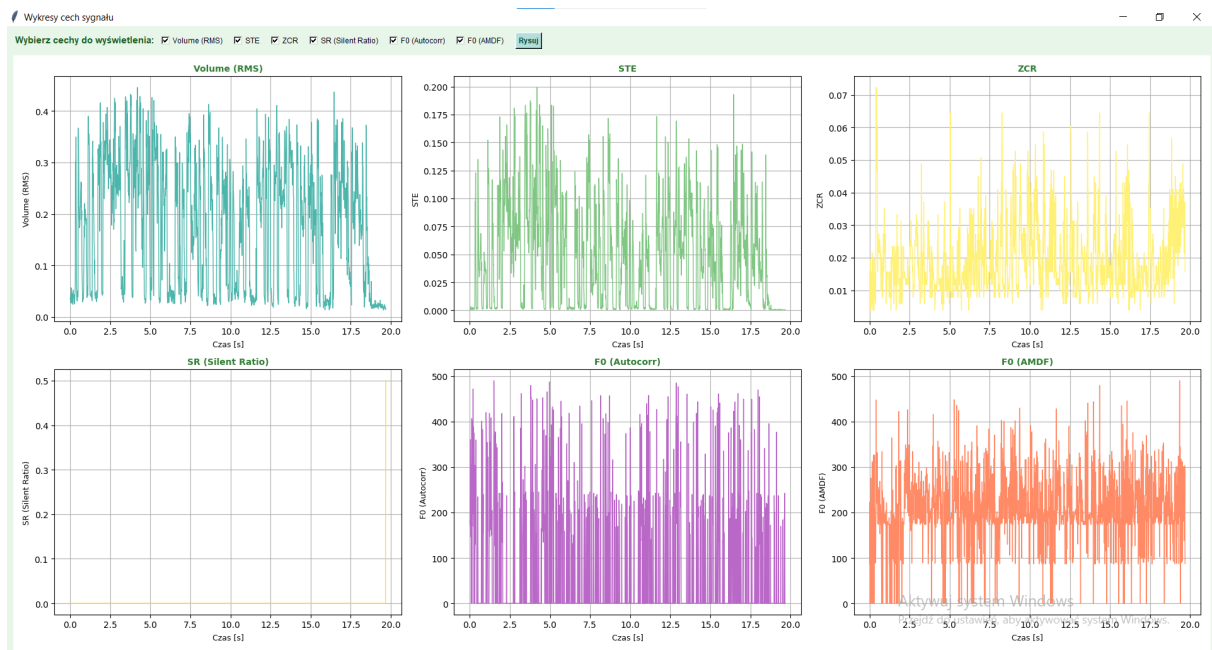


Rysunek 5: Przebieg czasowy nagrania mowy z radio, ukazujący segmentację na fragmenty dźwięczne i bezdźwięczne.

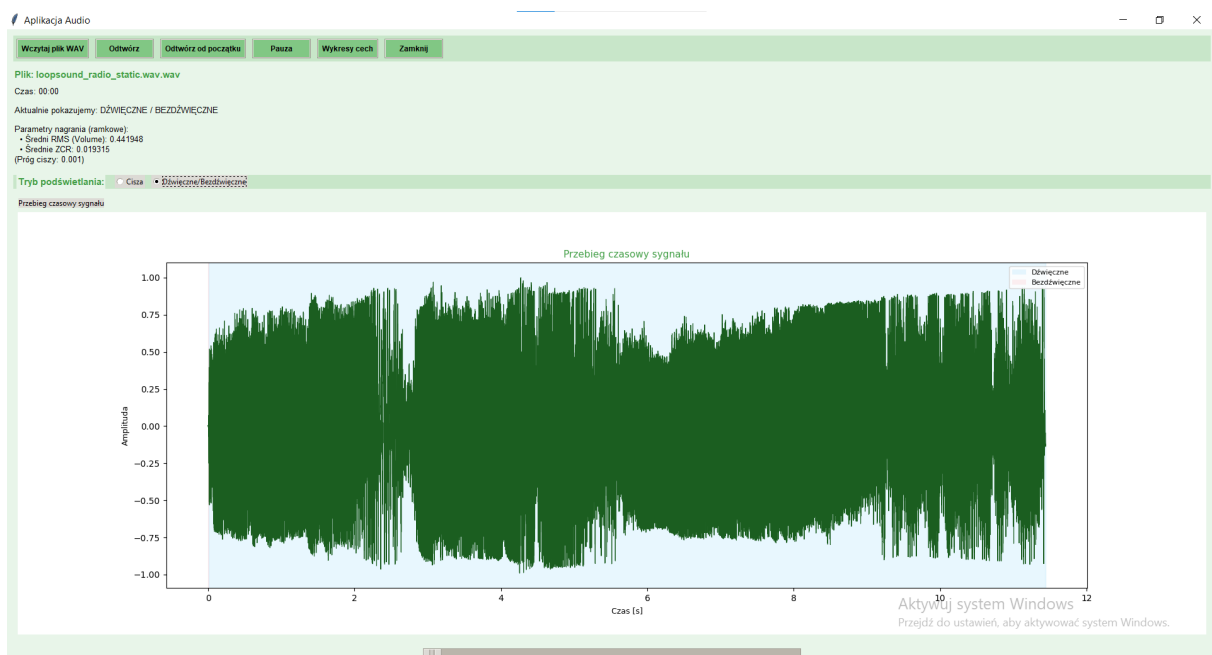
### 3.2.1 Analiza dodatkowych aspektów nagrań

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.
- **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.

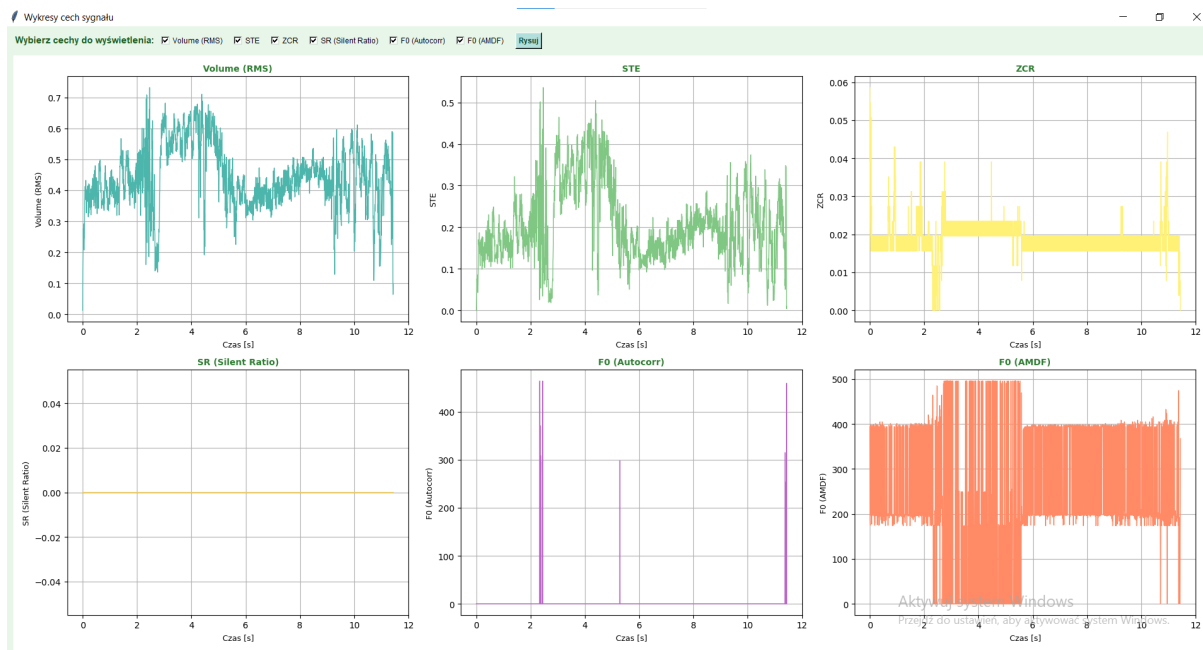


Rysunek 6: Wykresy cech (m.in. RMS, ZCR) dla nagrania mowy, ilustrujące zmienność dynamiki wypowiedzi.



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

- **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 harmoniczných oraz o zmienności instrumentów.
- **Wizualna segmentacja sygnałów:** Przebiegi czasowe (Rys. 5 oraz 7) jasno poka-



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

zują, 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.

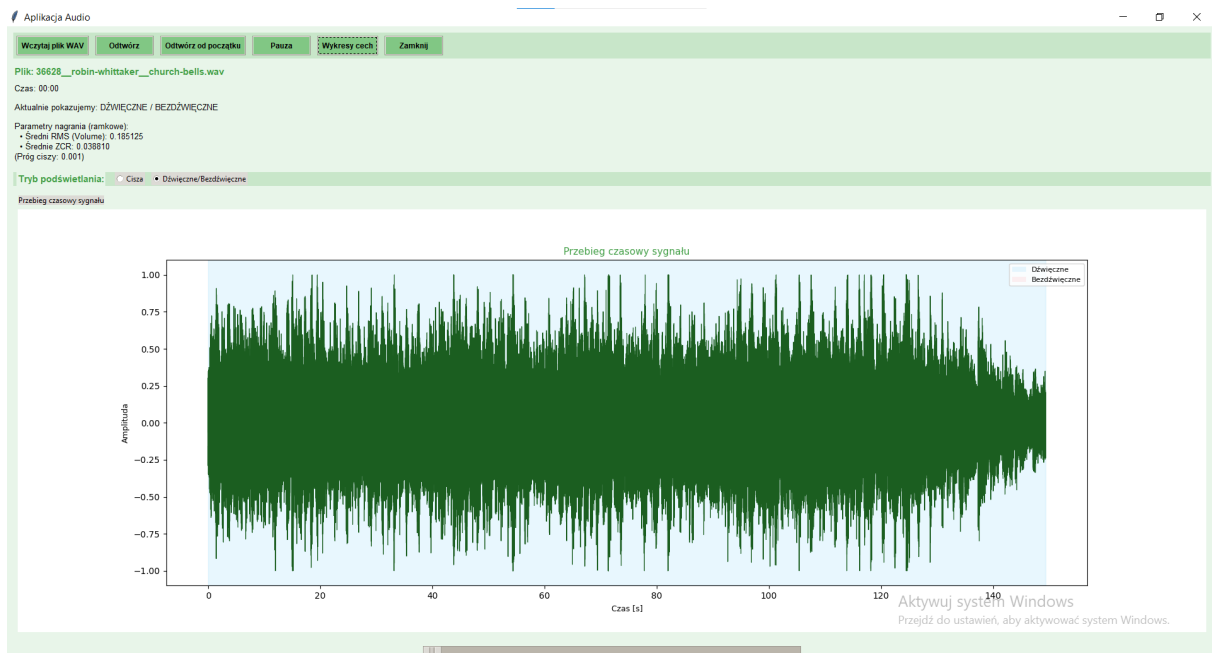
### 3.2.2 Podsumowanie porównania

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

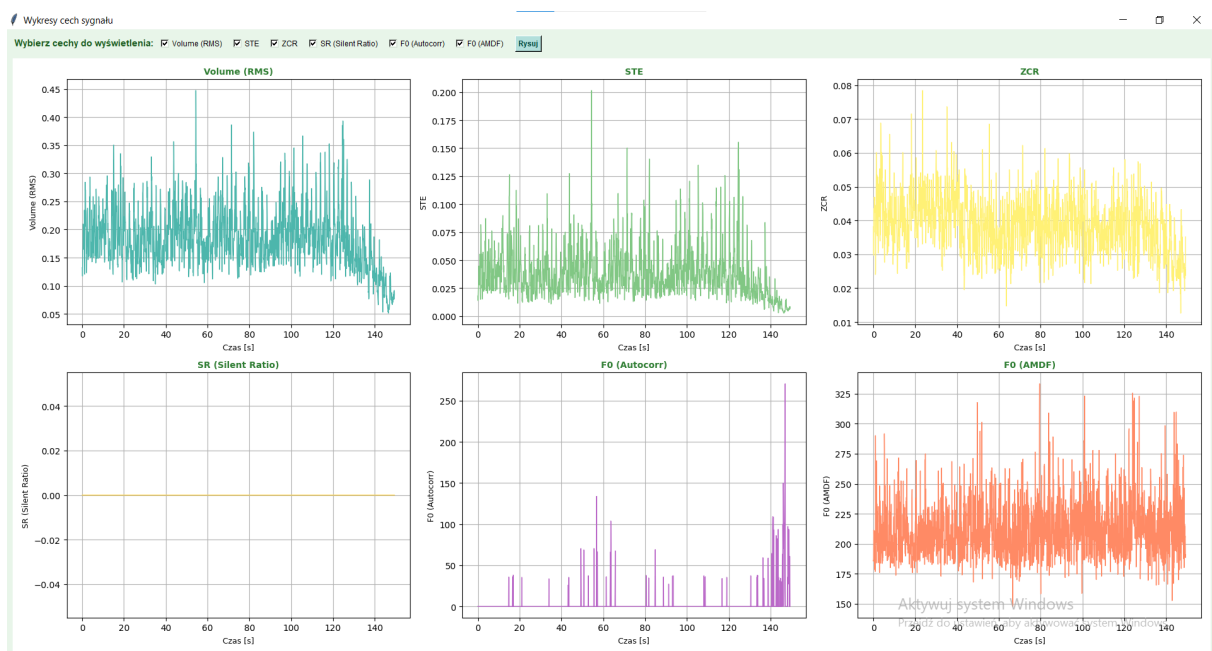
1. **Mowa radiowa** cechuje się wyraźną artykulacją, stabilnym poziomem energii oraz regularnymi segmentami, co umożliwia łatwą analizę cech takich jak RMS i ZCR.
2. **Muzyka radiowa** prezentuje większą dynamikę, złożoność harmoniczną oraz charakterystyczny transmisyjny szum, co wpływa na nieregularność wykresów cech i wymaga zastosowania metod redukcji punktów (downsampling) dla uzyskania czytelnych wizualizacji.
3. Różnice te są kluczowe przy projektowaniu systemów automatycznej klasyfikacji i przetwarzania sygnałów audio, gdzie precyzyjne dopasowanie algorytmów do specyfiki sygnału (mowa vs. muzyka) może znacząco wpłynąć na efektywność rozpoznawania i analizy.

## 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 harmoniczej.



### 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* praktycznie nie wzrasta, co potwierdza brak wyraźnych 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 zaczynało 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.