Real-Time Pitch Detection

Rajayogeshwar G
Course Project: EE698K

Abstract

Real-time pitch detection is crucial for applications such as vocal training, music analysis, and audio processing. By analyzing audio signals in real-time, pitch detection algorithms can capture nuances in musical performances and provide immediate feedback. This report explores the implementation of real-time pitch detection for singing using Python, integrating libraries such as PyAudio, Aubio, and Pygame.

1 Introduction

1.1 Pitch

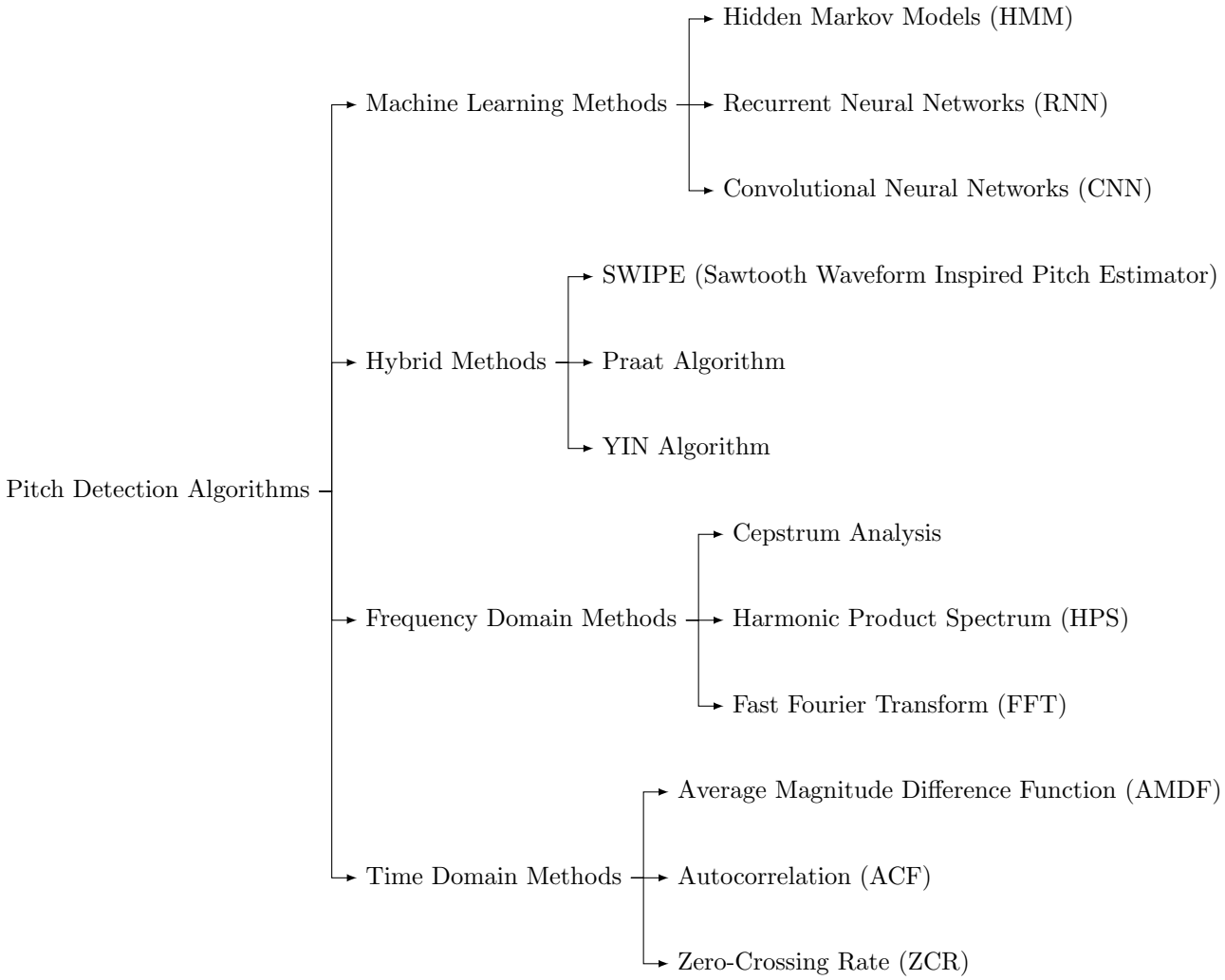
In acoustics, *pitch* is the perceptual attribute of sound that allows it to be ordered on a scale from low to high. It is primarily determined by the frequency of the sound wave, typically measured in hertz (Hz), which represents the number of oscillations or cycles per second. Sounds with higher frequencies (e.g., 1000 Hz) are perceived as having a higher pitch, while sounds with lower frequencies (e.g., 100 Hz) are perceived as having a lower pitch.
Mathematically, pitch perception is closely associated with the *fundamental frequency*, f_0, of a sound. Pure tones, such as sine waves, produce a clear pitch directly corresponding to their frequency. Complex sounds, like those produced by musical instruments or the human voice, contain multiple frequency components (harmonics), but the pitch is typically associated with the fundamental frequency, even if it is not the most prominent frequency present.

In both acoustics and music, *frequency* and *pitch* are related but distinct concepts. **Frequency** is a physical property of sound, referring to the number of cycles (oscillations) a sound wave completes per second and is measured in hertz (Hz). For instance, a sound wave with a frequency of 440 Hz oscillates 440 times per second. Each note in a song has a specific frequency or set of frequencies that corresponds to its physical vibration. **Pitch**, on the other hand, is a perceptual quality, reflecting how humans perceive the highness or lowness of a sound. Although pitch is directly related to frequency, it can also be influenced by other auditory factors, such as loudness and timbre (tone quality).

Pitch perception is essential in music and voice recognition. Even when multiple instruments in a song play the same fundamental frequency, the *timbre* or harmonic structure of each instrument gives it a distinct sound, though the perceived pitch remains the same. While frequency is objective and measurable, pitch is subjective and depends on the listener's perception. In a song, an instrument playing a note at a particular frequency will generally be perceived as having a pitch corresponding to that frequency. However, factors like loudness and harmonic content can affect the exact pitch perceived, as our auditory system interprets these characteristics when assigning pitch.

1.2 Pitch Detection

Pitch detection involves identifying the fundamental frequency of an audio signal, which corresponds to the perceived pitch of a sound. Techniques like Fourier Transform and Autocorrelation are commonly used to extract pitch information. Shown below is the flow chart of such algorithms used,



Time domain and frequency domain are two distinct approaches to signal analysis, each with its strengths and limitations. **Time domain methods** focus on analyzing a signal based on its amplitude variations over time. These methods, such as Zero-Crossing Rate (ZCR), Autocorrelation, and Average Magnitude Difference Function (AMDF), are simple, computationally efficient, and work well for periodic signals like pure tones or speech with clear time-domain patterns. However, they can struggle with complex, noisy, or non-periodic signals, as they rely heavily on detecting patterns in the raw signal. On the other hand, **frequency domain methods** analyze a signal in terms of its frequency components using tools like Fast Fourier Transform (FFT), Harmonic Product Spectrum (HPS), and Cepstrum Analysis. These methods are more accurate for detecting pitch in complex signals, such as musical instruments or vocals with multiple harmonic components, as they can separate the frequency content from noise. However, frequency domain methods tend to be computationally more expensive and may not capture transient or high-time-resolution details as effectively as time domain methods. In summary, time domain methods are best for simple, periodic signals, while frequency domain methods excel at handling complex, harmonic-rich signals.

Hybrid methods combine both time domain and frequency domain techniques to leverage the strengths of both approaches for more accurate pitch detection. These methods aim to capture the best features of each domain, improving performance in complex, noisy, or transient-rich signals. **Machine learning-based methods** utilize algorithms like neural networks or support vector machines to learn patterns in pitch from labeled data. These methods excel in handling complex, noisy, and non-linear pitch variations, often outperforming traditional methods by adapting to various signal types and conditions.

1.3 Applications of Pitch Detection

Pitch detection technology is applied across a range of fields, leveraging its ability to analyze and interpret pitch variations in audio signals:

- **Music Analysis and Production**: In music software, pitch detection is used for transcription, enabling automatic notation of melodies and harmonies, and for pitch correction tools, such as autotune, allowing precise pitch adjustments in vocals and instrument tracks.
- **Vocal Training and Feedback**: Pitch detection systems provide real-time feedback in singing and instrument training applications, assisting users in refining pitch accuracy and consistency, thus enhancing performance quality.
- **Speech Processing and Linguistic Analysis**: In natural language processing (NLP), pitch detection helps analyze speech intonation, stress, and prosody—key for language learning, emotion recognition, and improving user interactions with virtual assistants.
- **Audio Signal Processing**: Used for tasks like vocal separation, noise reduction, and frequency enhancement, pitch detection enables more refined editing in audio engineering and helps extract or isolate voices or sounds within complex recordings.
- **Healthcare and Diagnostics**: Clinicians use pitch analysis to diagnose and monitor conditions affecting voice stability, such as Parkinson's disease, enabling early detection and improved patient monitoring through vocal pattern analysis.
- **Emotion Recognition**: In affective computing, pitch variation is a crucial marker for detecting emotions in speech, such as happiness, sadness, or anger, aiding in sentiment analysis and enhancing emotional intelligence in AI systems.
- **Assistive Technology for Hearing Impairment**: Pitch detection is used to provide real-time visual feedback on voice pitch, improving speech comprehension for the hearing impaired and enriching text-to-speech systems for more natural-sounding outputs.
- **Human-Computer Interaction (HCI) and Robotics**: In HCI and robotics, pitch detection enhances interaction quality by enabling devices to respond to human speech cues and emotional states, fostering more natural and responsive communication.
- **Forensic Audio Analysis**: Forensic experts utilize pitch detection for speaker identification, tampering detection, and emotional state assessment, contributing to evidence analysis and investigation.
- **Animal Communication Research**: Bioacoustic studies use pitch detection to analyze animal calls and vocalizations, aiding in behavioral research, species identification, and understanding social structures within animal groups.

Overall, pitch detection is a critical technology in audio analysis, providing insights and enhancing applications across music, speech, healthcare, security, and beyond.

2 Implementation and Code Explanation

2.1 Main Page

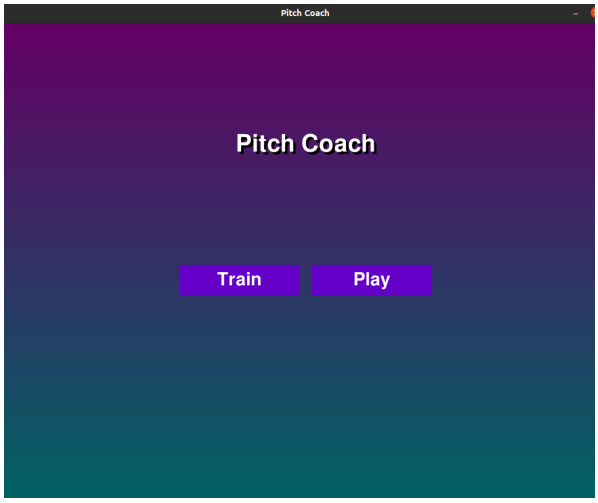


Figure 1: Main page of the tool

Explanation

The following code creates the main page for the Pitch Coach application:

- **Initialize Pygame and Display Window**: Initializes Pygame and sets up a resizable window for the display.
- **Define Colors and Fonts**: Sets up colors for the background gradient, title, button, and text, and loads custom fonts for the title and buttons.
- **Draw Gradient Background**: The `draw_gradient` function fills the window with a smooth vertical gradient from the top to the bottom.
- **Draw Title with Shadow Effect**: Adds a shadow effect to the title text using an offset to create a highlighted look.
- **Draw Button with Hover Effect**: The `draw_button` function changes the button color on hover based on the mouse position.
- **Event Handling**: Monitors for events, specifically the window quit event and window resizing.

The resulting code establishes a visually appealing main page with a title and two buttons labeled "Train" and "Play," which forms the basis for the Pitch Coach application's initial UI layout.

Code

```
import pygame
import sys

pygame.init()
pygame.mixer.init()
window = pygame.display.set_mode((1000, 800), pygame.RESIZABLE)
pygame.display.set_caption("Pitch Coach")

grad_colour1 = (100, 0, 100)
grad_colour2 = (0, 100, 100)
title_color = (255, 255, 255)
```

```
shadow_color = (0, 0, 0)
button_color = (100, 0, 200)
button_hover_color = (70, 70, 230)
text_color = (255, 255, 255)

title_font = pygame.font.SysFont("verdana", 40, bold=True)
button_font = pygame.font.SysFont("verdana", 30, bold=True)
button_width, button_height = 200, 50

def draw_gradient(window, color_top, color_bottom):
    width, height = window.get_size()
    for y in range(height):
        ratio = y / height
        r = int(color_top[0] * (1 - ratio) + color_bottom[0] * ratio)
        g = int(color_top[1] * (1 - ratio) + color_bottom[1] * ratio)
        b = int(color_top[2] * (1 - ratio) + color_bottom[2] * ratio)
        pygame.draw.line(window, (r, g, b), (0, y), (width, y))

def draw_title(window, text, font, color, shadow_color, position):
    shadow = font.render(text, True, shadow_color)
    shadow_rect = shadow.get_rect(center=position)
    window.blit(shadow, (shadow_rect.x + 4, shadow_rect.y + 4))
    title = font.render(text, True, color)
    title_rect = title.get_rect(center=position)
    window.blit(title, title_rect)

def draw_button(window, rect, text, font, color, hover_color, text_color):
    mouse_pos = pygame.mouse.get_pos()
    if rect.collidepoint(mouse_pos):
        pygame.draw.rect(window, hover_color, rect)
    else:
        pygame.draw.rect(window, color, rect)
    text_surface = font.render(text, True, text_color)
    text_rect = text_surface.get_rect(center=rect.center)
    window.blit(text_surface, text_rect)

running = True
clock = pygame.time.Clock()

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.VIDEORESIZE:
            window = pygame.display.set_mode((event.w, event.h), pygame.RESIZABLE)

    window_width, window_height = window.get_size()
    title_position = (window_width // 2, window_height // 4)
    train_button_rect = pygame.Rect(window_width // 2 - button_width - 10,
        window_height // 2, button_width, button_height)
    play_button_rect = pygame.Rect(window_width // 2 + 10, window_height // 2,
        button_width, button_height)

    draw_gradient(window, grad_colour1, grad_colour2)
    draw_title(window, "Pitch Coach", title_font, title_color, shadow_color,
        title_position)
    draw_button(window, train_button_rect, "Train", button_font, button_color,
        button_hover_color, text_color)
    draw_button(window, play_button_rect, "Play", button_font, button_color,
        button_hover_color, text_color)

    pygame.display.flip()
    clock.tick(60)

pygame.quit()
sys.exit()
```

2.2 Train Page

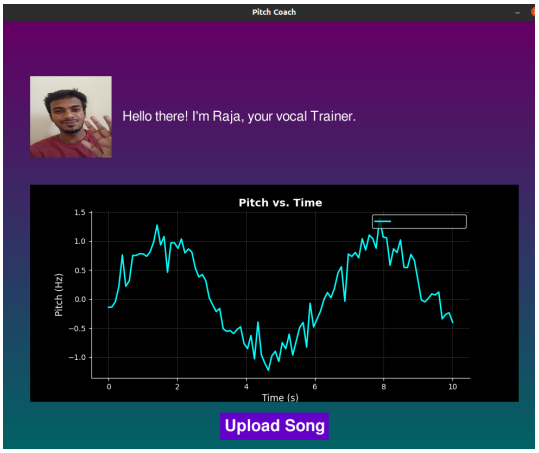


Figure 2: Train page of the tool

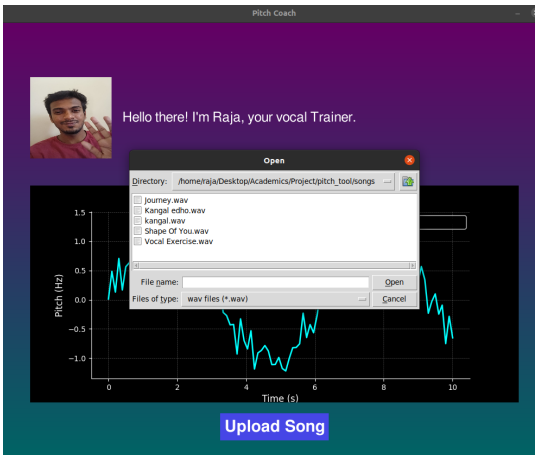


Figure 3: Song Upload UI

The following code implements the "Train" page of the Pitch Coach application after the train button is pressed.

Explanation

The following code creates the *Train Page* for the Pitch Coach application:

- **Trainer's Avatar**: The trainer's avatar is displayed on the screen inside a rectangular frame. The photo is placed at coordinates $(50, 100)$ with a size of $(150, 150)$ pixels. A border around the photo is drawn using `pygame.draw.rect`, and the photo itself is placed using `window.blit(photo, photo_rect)`.
- **Text Display**: The text "Hello there! I'm Raja, your vocal Trainer." is rendered using the `pygame.font.SysFont()` function, which sets the font style and size. The text is positioned to the right of the photo by adjusting its rectangle position (`text_rect`). It is drawn on the screen using `window.blit(text_surface, text_rect)`.
- **Professional Plot**: A plot, generated by `create_professional_plot()`, is placed at the center of the window (horizontally) and just below the photo and text area. The plot is displayed using `window.blit(plot_surface, plot_rect)`.
- **Upload Song Button**: The "Upload Song" button is rendered below the plot. The button's position is calculated dynamically based on the window size and plot position. It is drawn using the `draw_button()` function, which was previously defined. The button includes a hover effect, where its color changes when the mouse cursor is over it. The button listens for a click event, and if clicked, the `upload_song()` function is triggered.
- **Mouse Event Handling**: The mouse position is checked using `pygame.mouse.get_pos()`, and the left-click event is detected with `pygame.mouse.get_pressed()[0]`. If the mouse click occurs within the boundaries of the "Upload Song" button (`upload_button_rect.collidepoint(mouse_pos)`), the `upload_song()` function is called.
- **Song Upload Functionality**: The `upload_song()` function is responsible for opening a file dialog that allows the user to select a `.wav` file. The file dialog uses the Tkinter `filedialog.askopenfilename()` function, and the initial directory is set to a folder containing the audio files. Once a song is selected, the file path is saved in the global variable `uploaded_song`, and the application switches to the "pitch_detection" page by setting `current_page` to "pitch_detection".

The code is structured to create an interactive "Train" page that allows users to interact with the application by uploading a song for pitch detection. The layout is carefully designed with a gradient background, trainer's photo, informative text, and a button that enables users to upload their own audio files. The event handling ensures that when the user clicks the button, the song is uploaded, and the page transitions accordingly.

Code

```
# Function to draw the Train Page
def draw_train_page(window):
    draw_gradient(window, grad_colour1, grad_colour2)

    window_width, window_height = window.get_size()
    photo_rect = pygame.Rect(50, 100, 150, 150)
    pygame.draw.rect(window, border_color, photo_rect, 3)
    window.blit(photo, photo_rect)

    text = "Hello there! I'm Raja, your vocal Trainer."
    font = pygame.font.SysFont("verdana", 25)
    text_surface = font.render(text, True, text_color)
    text_rect = text_surface.get_rect(left=photo_rect.right + 20, centery=photo_rect.
        centery)
    window.blit(text_surface, text_rect)

    plot_surface = create_professional_plot()
    plot_rect = plot_surface.get_rect(center=(window_width // 2, 500))
    window.blit(plot_surface, plot_rect)

    upload_button_text = "Upload Song"
    upload_button_rect = pygame.Rect(window_width // 2 - button_width // 2,
                                     plot_rect.bottom + 20, button_width, button_height)
    draw_button(window, upload_button_rect, upload_button_text, button_font,
                button_color, button_hover_color, text_color)

    mouse_pos = pygame.mouse.get_pos()
    if pygame.mouse.get_pressed()[0]:
        if upload_button_rect.collidepoint(mouse_pos):
            upload_song()

# Function to handle song upload
def upload_song():
    global uploaded_song, current_page
    root = tk.Tk()
    root.withdraw()
    relative_path = os.path.join(os.path.dirname("/home/raja/Desktop/Academics/Project
        /pitch_tool/songs"), "songs")
    song_path = filedialog.askopenfilename(initialdir=relative_path, filetypes=[("wav
        files", "*.wav")])
    if song_path:
        uploaded_song = song_path
        current_page = "pitch_detection"

# Mouse event to trigger song upload when the button is clicked
mouse_pos = pygame.mouse.get_pos()
if pygame.mouse.get_pressed()[0]:
    if upload_button_rect.collidepoint(mouse_pos):
        upload_song()
```

2.3 Pitch detection Page

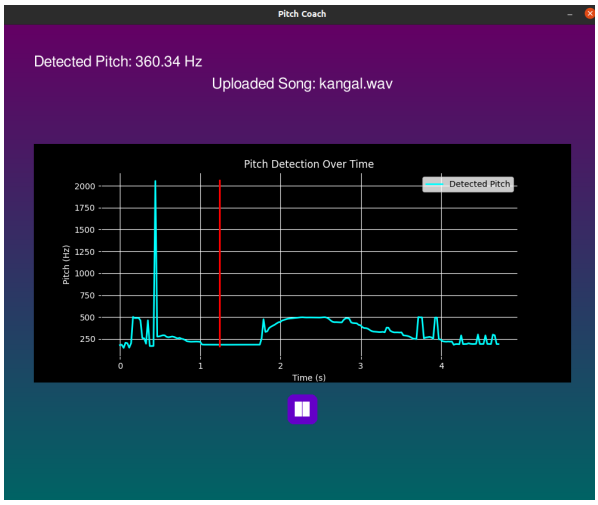


Figure 4: Main page of the tool

The following Python code implements a system for real-time pitch detection and visualization of audio data. This system leverages the Pygame and Aubio libraries for handling audio playback and pitch detection. Below is an explanation of each function and component:

Explanation

- **Initialization of Pygame Mixer:** The code begins by initializing the Pygame mixer for sound playback.
- **Aubio Settings:** Aubio is used for pitch detection, where the ‘pitch‘ function is initialized with a buffer size and sample rate.
- **Pitch Plotting Function:** This function generates a plot showing the pitch over time for the uploaded song using pitch detection function which is dicussed below. It uses Matplotlib to create the plot and saves it as a Pygame surface to display.
- **Get Song Length:** This function uses Pygame's Sound object to calculate the length of the uploaded song.
- **Drawing the Pitch Detection Page:** This function is responsible for drawing the graphical interface for the pitch detection system, which includes the song name, plot of detected pitch, and control buttons for playback.
- **Toggling Playback:** This function toggles the playback of the uploaded song, either starting, pausing, or unpausing it based on the current state.
- **Getting Real-Time Pitch:** This function captures audio data from the microphone in real-time and uses Aubio to detect the pitch.
- **Creating Play and Pause Buttons:** These functions create the play and pause buttons for the graphical interface using Pygame's drawing functions.
- **Creating Button Surfaces:** This part of the code creates the surfaces for the play and pause buttons after defining the button size.

Python Code

```
# Initialize the mixer for playing sound
pygame.mixer.init()

# Aubio settings for pitch detection
samplerate = 44100
buffer_size = 1024
p = pyaudio.PyAudio()
pitch_detector = aubio.pitch("default", buffer_size, buffer_size//2, samplerate)
pitch_detector.set_unit("Hz")
audio_buffer = deque(maxlen=int(samplerate / buffer_size))  # Store ~1 second of audio

def pitch_plot(song_path):
    time_data, pitch_data = detect_pitch(song_path)
    if time_data is None or pitch_data is None:
        return None

    fig, ax = plt.subplots(figsize=(9, 4), dpi=100, facecolor='black')
    ax.plot(time_data, pitch_data, label="Detected Pitch", color='cyan', linewidth=2)
    ax.set_xlabel("Time (s)", color='white')
    ax.set_ylabel("Pitch (Hz)", color='white')
    ax.set_title("Pitch Detection Over Time", color='white')
    ax.grid(True, color='white')
    ax.legend()
    ax.set_facecolor('black')
    ax.tick_params(axis='both', which='both', labelcolor='white', color='white')

    img_buffer = BytesIO()
    fig.savefig(img_buffer, format='png')
    img_buffer.seek(0)
    img_data = img_buffer.read()
    img_surface = pygame.image.load(BytesIO(img_data))

    plt.close(fig)
    return img_surface

def get_song_length(song_path):
    sound = pygame.mixer.Sound(song_path)
    return sound.get_length()

def draw_pitch_detection_page(window):
    draw_gradient(window, grad_colour1, grad_colour2)
    window_width, window_height = window.get_size()
    font = pygame.font.SysFont("verdana", 25)

    song_text = f"Uploaded Song: {uploaded_song.split('/')[-1]}"
    song_text_surface = font.render(song_text, True, text_color)
    song_text_rect = song_text_surface.get_rect(center=(window_width // 2, 100))
    window.blit(song_text_surface, song_text_rect)

    plot_surface = pitch_plot(uploaded_song)
    if plot_surface:
        plot_rect = plot_surface.get_rect(center=(window_width // 2, window_height //
            2))
        window.blit(plot_surface, plot_rect)

        control_button_rect = pygame.Rect(
            window_width // 2 - control_button_size // 2,
            plot_rect.bottom + 20,
            control_button_size,
            control_button_size
        )

        pygame.draw.rect(window, button_color, control_button_rect, border_radius=10)

        if is_playing:
            window.blit(pause_button_surface, control_button_rect)
        else:
            window.blit(play_button_surface, control_button_rect)

```

```
        if is_playing:
            song_length = get_song_length(uploaded_song)
            current_time = pygame.mixer.music.get_pos() / 1000.0

            # Calculate margins
            margin = plot_rect.width * 0.13
            usable_width = plot_rect.width - (2 * margin)

            # Calculate line position with margins
            relative_position = current_time / song_length
            line_x = plot_rect.left + margin + (relative_position * usable_width) +25

            # Calculate reduced height
            height_reduction = plot_rect.height * 0.15  # 15% reduction from top and
                bottom
            line_start_y = plot_rect.top + height_reduction
            line_end_y = plot_rect.bottom - height_reduction

            # Draw the line with the new dimensions
            pygame.draw.line(window, (255, 0, 0),
                             (line_x, line_start_y),
                             (line_x, line_end_y),
                             3)

        return control_button_rect
    else:
        error_text = "Pitch detection failed."
        error_text_surface = font.render(error_text, True, text_color)
        error_text_rect = error_text_surface.get_rect(center=(window_width // 2,
            window_height // 2))
        window.blit(error_text_surface, error_text_rect)
        return None

def toggle_playback():
    global is_playing
    if is_playing:
        pygame.mixer.music.pause()
    else:
        if pygame.mixer.music.get_pos() == -1:
            pygame.mixer.music.load(uploaded_song)
            pygame.mixer.music.play()
        else:
            pygame.mixer.music.unpause()
    is_playing = not is_playing

# Function to capture real-time microphone data
def get_realtime_pitch():
    audio_data = np.frombuffer(stream.read(buffer_size), dtype=np.float32)[:
        buffer_size // 2]  # Slice to 512 samples
    pitch = pitch_detector(audio_data)[0]
    if pitch > 0:  # Filter out zero values
        return pitch
    return None

# Function to create a play button surface
def create_play_button(size):
    surface = pygame.Surface((size, size), pygame.SRCALPHA)
    # Draw play triangle
    points = [(size//4, size//4), (size//4, size*3//4), (size*3//4, size//2)]
    pygame.draw.polygon(surface, text_color, points)
    return surface

# Function to create a pause button surface
def create_pause_button(size):
    surface = pygame.Surface((size, size), pygame.SRCALPHA)
    # Draw pause bars
    bar_width = size//4
    pygame.draw.rect(surface, text_color, (size//4, size//4, bar_width, size//2))
    pygame.draw.rect(surface, text_color, (size//2, size//4, bar_width, size//2))
    return surface

```

```
# Create button surfaces after defining the control_button_size
play_button_surface = create_play_button(control_button_size)
pause_button_surface = create_pause_button(control_button_size)
```

3 Pitch detection functions

3.1 Using librosa Library

The following Python code is used to detect the pitch of an audio file over time. It utilizes the `librosa` library, which is a popular library for music and audio analysis. The function implemented in the code extracts pitch information from an audio file (MP3, WAV, etc.) using signal processing techniques.

Explanation

`detect_pitch(song_path)` function detects the pitch of an audio file over time. The function performs the following steps:

- **Loading Audio:** The audio file is loaded using the `librosa.load` function, which converts the audio into a time-domain signal (`y`) and its corresponding sampling rate (`sr`).
- **Pitch Detection with `piptrack`:** The `librosa.core.piptrack` function is used to perform pitch tracking. This function computes the pitch and magnitude for each time frame of the audio signal.
- **Peak Selection:** For each time frame, the algorithm finds the pitch corresponding to the peak magnitude. This ensures that the most dominant pitch is selected.
- **Filtering Non-Zero Pitches:** Only valid pitch values (those greater than 0) are kept.
- **Time and Pitch Data Extraction:** The time corresponding to each pitch value is obtained using `librosa.times_like`, which generates an array of times corresponding to each frame of the pitch analysis.

The function returns two numpy arrays:

- `time_data`: The time in seconds for each pitch estimate.
- `pitch_data`: The detected pitch in Hz corresponding to each time.

If an error occurs during pitch detection, the function returns `None, None`.

Pitch Detection Algorithms

The pitch detection algorithm used in the `detect_pitch` function relies on the following key signal processing techniques:

- **Short-Time Fourier Transform (STFT):** The `librosa` library internally uses the Short-Time Fourier Transform (STFT) to analyze the frequency content of the audio signal. The STFT splits the signal into overlapping frames and computes the frequency components in each frame. This allows the algorithm to analyze the pitch at each time step.
- **Pitch Tracking with `piptrack`:** The `librosa.core.piptrack` function computes the pitch for each frame of the audio. It returns two arrays:
  - `pitches`: The estimated pitches for each time frame.
  - `magnitudes`: The magnitudes (or strengths) of the corresponding pitches.

  The algorithm then selects the pitch with the highest magnitude at each frame to get the most prominent pitch.

- **Peak Detection:** For each time frame, the pitch corresponding to the maximum magnitude is selected, ensuring that the most dominant pitch is chosen. This is performed using `argmax`, which returns the index of the maximum value in the magnitude array.
- **Filtering and Time Alignment:** Only valid (non-zero) pitches are retained. The time data corresponding to each pitch is calculated using `librosa.times_like`, which provides the time values for each frame.

This process effectively tracks the pitch of an audio signal over time, making it useful for various applications such as music analysis, voice detection, and singing training systems.

Python Code

```python
import librosa
import numpy as np

def detect_pitch(song_path):
    """
    Detects the pitch of an audio file over time.

    Parameters:
    - song_path (str): Path to the audio file (MP3, WAV, etc.)

    Returns:
    - time_data (numpy array): Time in seconds for each pitch estimate.
    - pitch_data (numpy array): Detected pitch in Hz corresponding to each time.
    """
    try:
        # Load the audio file using librosa
        y, sr = librosa.load(song_path)

        # Use librosa's pitch detection function
        pitches, magnitudes = librosa.core.piptrack(y=y, sr=sr)

        # Get the index of the peak pitch at each frame
        pitch_data = []
        time_data = librosa.times_like(pitches)

        for t in range(len(time_data)):
            # Get the pitch with the maximum magnitude at the current time
            index = magnitudes[:, t].argmax()
            pitch = pitches[index, t]
            if pitch > 0:  # Only consider valid pitches (non-zero)
                pitch_data.append(pitch)

        # Convert the time and pitch data to numpy arrays
        time_data = np.array(time_data[:len(pitch_data)])
        pitch_data = np.array(pitch_data)

        return time_data, pitch_data

    except Exception as e:
        print(f"Error in pitch detection: {e}")
        return None, None
```

3.2 Using Aubio Library

The following Python code snippet performs real-time pitch detection using the `aubio` library, combined with `pyaudio` for capturing microphone data. The main algorithm used here is *pitch tracking* based on the autocorrelation or YIN algorithm. This is explained in detail below.

Explanation

This code performs real-time pitch detection using the `aubio.pitch` function, which is a popular method for pitch detection in audio signals. Below is an explanation of the key components:

- **Aubio Settings:**
  - The sample rate is set to 44.1kHz (`samplerate = 44100`), which is a standard rate for audio processing.
  - The buffer size is set to 1024 samples (`buffer_size = 1024`), meaning that the pitch is detected over windows of 1024 samples at a time. This buffer size influences the resolution of the pitch detection, with larger buffers providing more accurate results but with lower temporal resolution.
  - The `aubio.pitch` function is initialized with the "default" pitch detection method (which typically uses the YIN algorithm or a similar method). The pitch detection function is also provided with the buffer size and sample rate.
  - The `audio_buffer` stores audio data for approximately one second (`maxlen=int(samplerate / buffer_size)`).
- **Real-Time Microphone Data Capture:**
  - The function `get_realtime_pitch()` reads audio data from the microphone in chunks of `buffer_size` (1024 samples).
  - The audio data is converted to a `numpy` array of type `np.float32` using `np.frombuffer()`.
  - Only the first half of the buffer (512 samples) is used for pitch detection, as indicated by the slicing operation `[:buffer_size // 2]`.
  - The pitch is estimated using `pitch_detector(audio_data)[0]`, and if the detected pitch is greater than 0 (i.e., valid), it is returned. Otherwise, `None` is returned.

Pitch Detection Algorithm

The pitch detection algorithm employed in the code is based on the following principles:

- **Autocorrelation and YIN Algorithm:** The core of the `aubio.pitch` function is the YIN algorithm or a similar autocorrelation-based method.
  - **Autocorrelation**: Autocorrelation is a mathematical tool used to measure the similarity between a signal and a delayed version of itself. For periodic signals (such as musical tones), the maximum similarity corresponds to the period of the signal. The inverse of the period gives the frequency (pitch).
  - **YIN Algorithm**: The YIN algorithm is an improved version of autocorrelation that minimizes harmonic errors and provides more accurate pitch estimates. It works by calculating the difference between the signal and its delayed version, and then iteratively searching for the best match (the period) based on minimizing this difference. This approach is highly effective for monophonic signals (e.g., a single musical note).
- **Frame-Based Processing:** The audio is processed in overlapping frames of `buffer_size` (1024 samples). The pitch is estimated for each frame, allowing the algorithm to track the pitch over time. This method is particularly useful for real-time processing, as it enables continuous pitch tracking without the need to process the entire audio signal at once.
- **Pitch Estimation:** The pitch is extracted from each frame by applying the YIN algorithm. If the detected pitch is greater than 0 Hz, it is considered a valid pitch and returned. Otherwise, `None` is returned. This step ensures that only valid pitch values are retained, eliminating noise or silence regions.
- **Real-Time Processing:** The algorithm works in real-time by capturing microphone input and processing the data in small chunks. This allows for continuous pitch tracking, making it ideal for applications such as live performance analysis or speech processing.

The pitch detection process involves iterating through frames of audio data, applying the YIN-based autocorrelation method, and extracting pitch estimates in real-time. This method is highly effective in detecting monophonic sounds, such as single notes in music or individual spoken words.

Python Code

```python
# Aubio settings for pitch detection
samplerate = 44100
buffer_size = 1024
p = pyaudio.PyAudio()
pitch_detector = aubio.pitch("default", buffer_size, buffer_size//2, samplerate)
pitch_detector.set_unit("Hz")
audio_buffer = deque(maxlen=int(samplerate / buffer_size))  # Store ~1 second of audio

# Function to capture real-time microphone data
def get_realtime_pitch():
    audio_data = np.frombuffer(stream.read(buffer_size), dtype=np.float32)[:
        buffer_size // 2]  # Slice to 512 samples
    pitch = pitch_detector(audio_data)[0]
    if pitch > 0:  # Filter out zero values
        return pitch
    return None
```

4 Programming

4.1 Data structures used

Below is a list of all data structures used in the code along with the functions where they are utilized:

- **'deque'** (from `collections` module)
  - Used in the global scope to create `audio_buffer`.
  - Stores audio samples for pitch detection in real-time, allowing efficient FIFO access.
  - Defined as: `audio_buffer = deque(maxlen=int(samplerate / buffer_size))`
- **'pygame.Surface'**
  - Used in `create_play_button` and `create_pause_button` functions to create button surfaces for play/pause.
  - Stores surfaces for custom graphics like play/pause buttons.
  - Defined as `surface = pygame.Surface((size, size), pygame.SRCALPHA)`.
- **'numpy.array'**
  - Used in `get_realtime_pitch` function.
  - Stores real-time audio samples read from the microphone for pitch analysis.
  - Defined as: `audio_data = np.frombuffer(stream.read(buffer_size), dtype=np.float32)`
- **'pygame.Rect'**
  - Used throughout different functions like `draw_train_page`, `draw_button`, `draw_pitch_detection_page`, etc.
  - Defines rectangular areas for buttons and elements on the screen.
  - Example usage: `control_button_rect = pygame.Rect(window_width // 2 - control_button_size // 2, ...)`
- **'BytesIO'** (from `io` module)
  - Used in `pitch_plot` function.
  - Temporarily holds the plot image data in memory for displaying in `pygame`.
  - Defined as: `img_buffer = BytesIO()`
- **'string'** and **'int'** variables
  - Used throughout for storing general data like file paths, window dimensions, font sizes, etc.
  - Examples include `uploaded_song`, `is_playing`, and `current_page`.

4.2 Programming Tecchniques

The following programming techniques are utilized in the code:

- **Event-Driven Programming**
  - The code uses an event loop (`while running`) to continuously listen for and respond to user actions such as mouse clicks, window resizing, and quitting.
  - Event-driven programming is implemented through `pygame.event.get()` and handles specific actions based on the event type.
- **Object-Oriented Programming (OOP) with Libraries**
  - The code leverages external libraries (e.g., `pygame` and `matplotlib`), treating various functionalities as objects.
  - For example, `pygame.Surface` and `pygame.Rect` are objects used to create surfaces and rectangles in the GUI.
  - Functions like `pyaudio.PyAudio()` also exemplify OOP by treating the audio system as an object with properties and methods.
- **Modular Programming**
  - The code is organized into modular functions, each with a specific task, making the code more readable and maintainable.
  - Examples include `draw_button`, `create_professional_plot`, and `toggle_playback`.
  - This approach reduces redundancy and enhances reusability.
- **Real-Time Data Processing**
  - Real-time audio input and processing are handled with the PyAudio library, using the `get_realtime_pitch` function to capture and analyze pitch data as audio is streamed.
  - Techniques like deque-based buffering in `audio_buffer` store recent audio samples, allowing quick data updates.
- **Graphical User Interface (GUI) Design**
  - Pygame is used to create and display an interactive GUI, including buttons, images, gradients, and text.
  - Functions like `draw_gradient` and `draw_title` apply various visual styles to improve user experience.
- **File Handling and Path Management**
  - The code utilizes `os.path.join` and `tkinter.filedialog.askopenfilename` for file path management and to allow users to upload songs from the file system.
  - This technique ensures flexibility across operating systems and allows easy access to the audio files.
- **Data Visualization**
  - The code uses `matplotlib` to create plots, displaying pitch vs. time and allowing visual feedback of pitch detection.
  - Techniques like custom colors, grid styling, and legends are used to enhance visualization quality.
- **Conditionals and Control Flow**
  - Control flow is managed through conditionals (`if-else` statements), determining which page or state (e.g., play/pause) to display based on user actions.

  - Conditionals are used in functions like `toggle_playback` to switch between play and pause states based on the current `is_playing` status.

- **Resource Management and Cleanup**
  - Proper resource cleanup is handled by closing audio streams and terminating PyAudio and Pygame instances at the end of the program.
  - Techniques like `stream.stop_stream()` and `pygame.quit()` ensure that resources are released properly.

5 Conclusion

In conclusion, this project has demonstrated the feasibility of real-time pitch detection in a singing trainer application, utilizing Python and libraries such as PyAudio, Aubio, and Pygame. The application enables users to monitor pitch accuracy visually and interact with playback controls in a responsive GUI environment. Implementing gradient designs and dynamic pitch plots has added a user-friendly aesthetic, enhancing the experience for singing practice.

However, a key limitation encountered was Python's slower processing speed for this application. Given the demand for real-time audio processing, Python's performance, though versatile and accessible, introduced delays that affected response time. For future iterations, optimizing computational efficiency or considering faster, lower-level programming languages might improve performance, especially in applications where real-time feedback is critical.