EE 4490

Music Tech Engineering: Project Report "Live" Autotune

Anthony Nguyen

Ben Levario

Kennesaw State University

Electrical Engineering

Marietta, GA

Dr. Lance Crimm

Table of Content

Abstract

This project successfully developed an autotune effect for audio files, aligning their pitch with a specified musical scale. The core process involves three key steps. First, Pitch Detection estimates the fundamental frequency of the audio signal over time. Next, Pitch Correction precisely shifts these detected frequencies to align with the notes of the chosen musical scale. Finally, Audio Synthesis recreates the audio with the corrected pitch using techniques like Pitch Synchronous Overlap and Add (PSOLA). While the initial ambition was to achieve real-time autotuning, this proved unfeasible due to the demanding processing requirements of large audio data with low latency, coupled with the time constraints of a single summer semester. However, the project achieved similar functionality to real-time pitch correction by upgrading the processing platform from a microcontroller to a full computer.

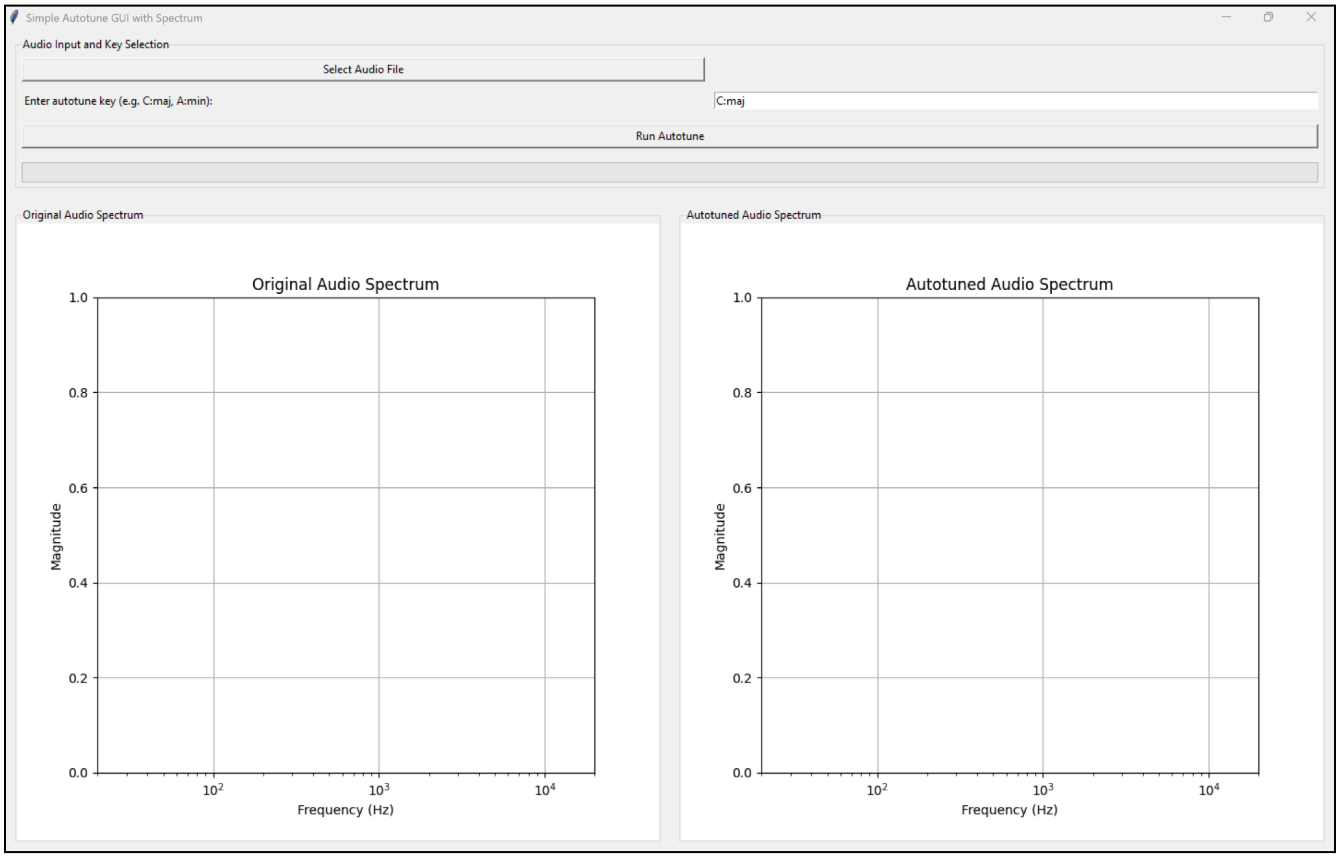
Introduction

This project set out to develop and implement a robust autotune effect for audio signals, a process critical in modern music production for pitch correction and creative vocal manipulation. The core challenge involved precisely analyzing and adjusting the fundamental frequency of an audio signal to align seamlessly with a specified musical scale. Our comprehensive design methodology was structured around three interconnected key objectives: first, to accurately and reliably detect the instantaneous pitch of the input audio; second, to effectively correct this detected pitch, snapping it to the nearest target note within the chosen musical key while preserving natural vocal characteristics; and finally, to synthesize the modified audio using advanced signal processing techniques to achieve a natural-sounding and professional output. While initially ambitious in aiming for real-time processing, our development ultimately pivoted to an offline application, a strategic decision necessitated by the substantial computational demands of high-fidelity audio manipulation and the scope of a single development semester. This report details the successful realization of this autotune system, demonstrating its core functionalities and the underlying technical principles.

Operating Instructions

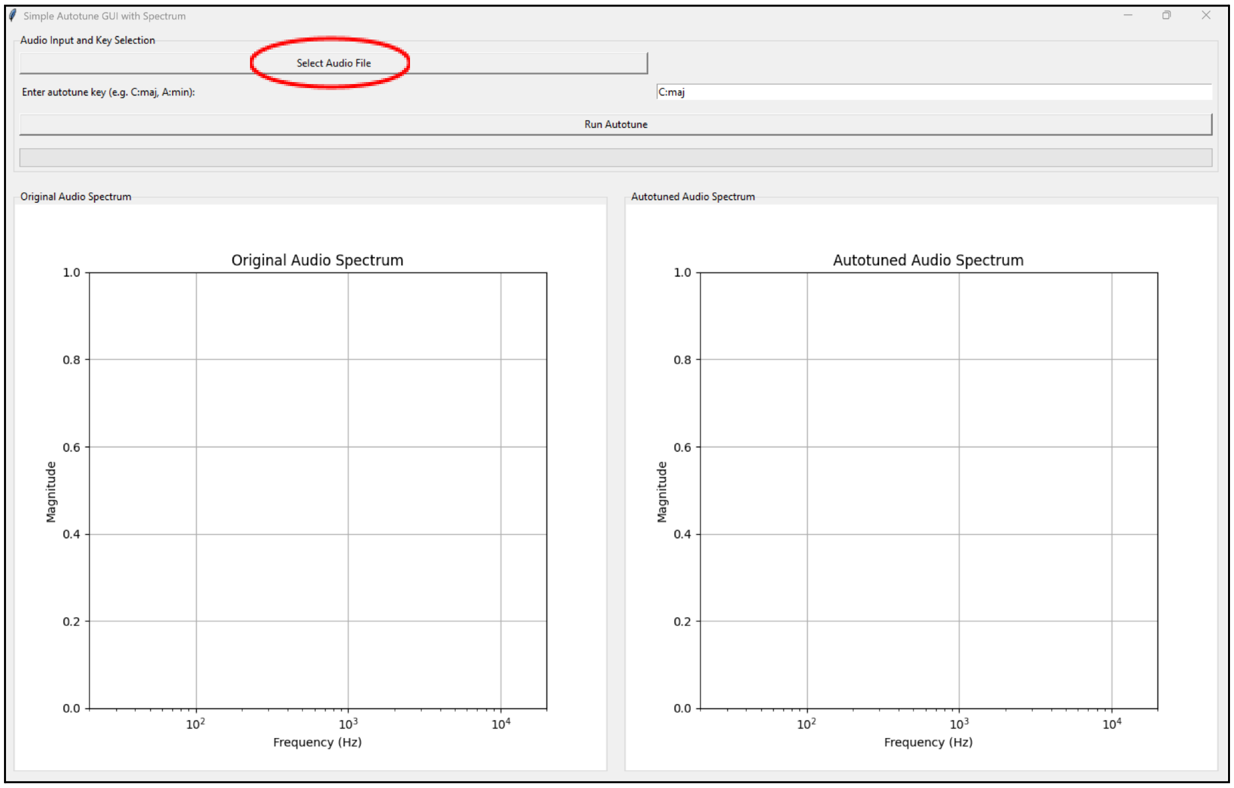
This application allows users to apply an autotune effect to audio files through a simple graphical user interface (GUI). Follow these steps to operate the system:

1. **Launch the Application:**
    - Run the Python script. A window titled "Simple Autotune GUI with Spectrum" will appear.



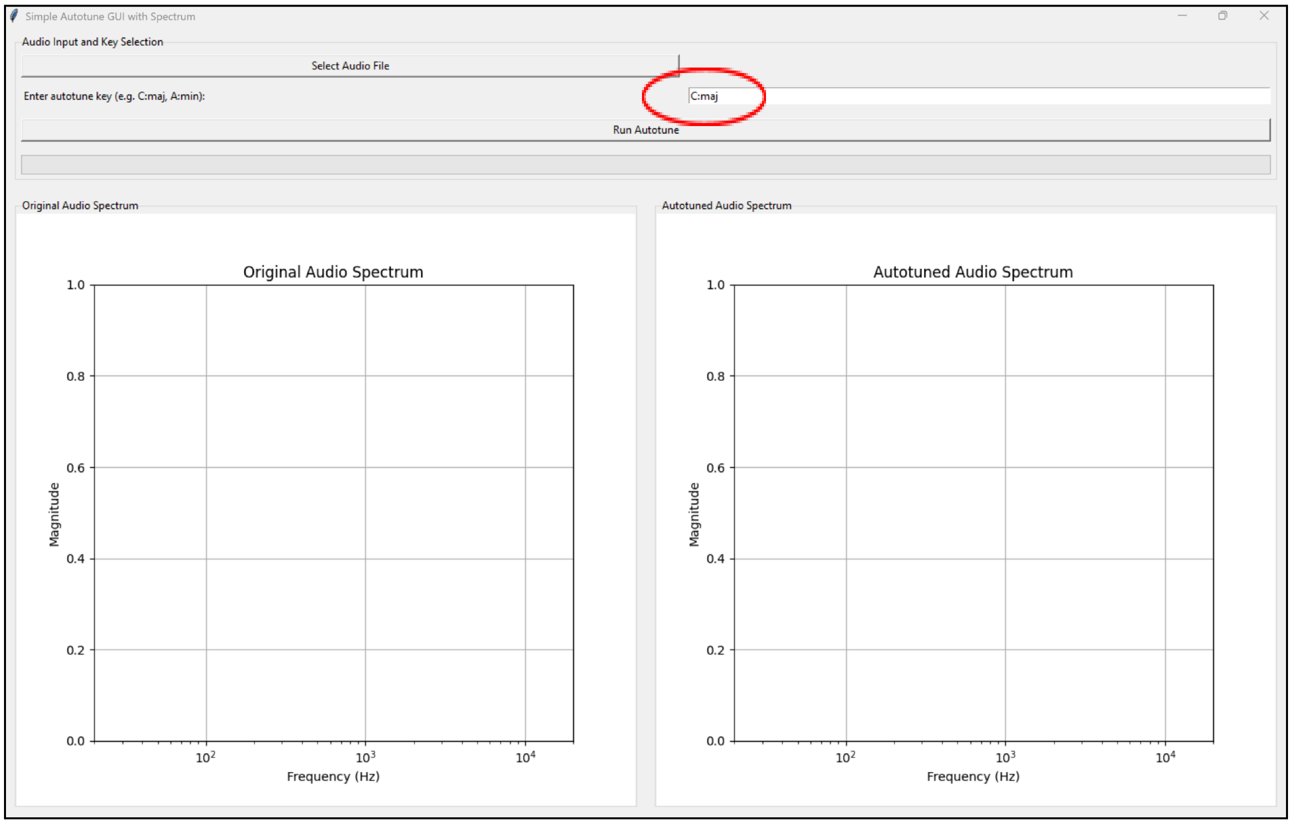
Enter autotune key (e.g. C:maj, A:min): C:maj

2. **Select an Audio File:**
    - Click the "**Select Audio File**" button.



    - A file dialog will open. Navigate to the location of your audio file (e.g., a `.wav` file) and select it.
    - A message box will confirm the selected file's name. If no file is selected, the application will attempt to use a default file named `monitoring.wav` if it exists in the script's directory.

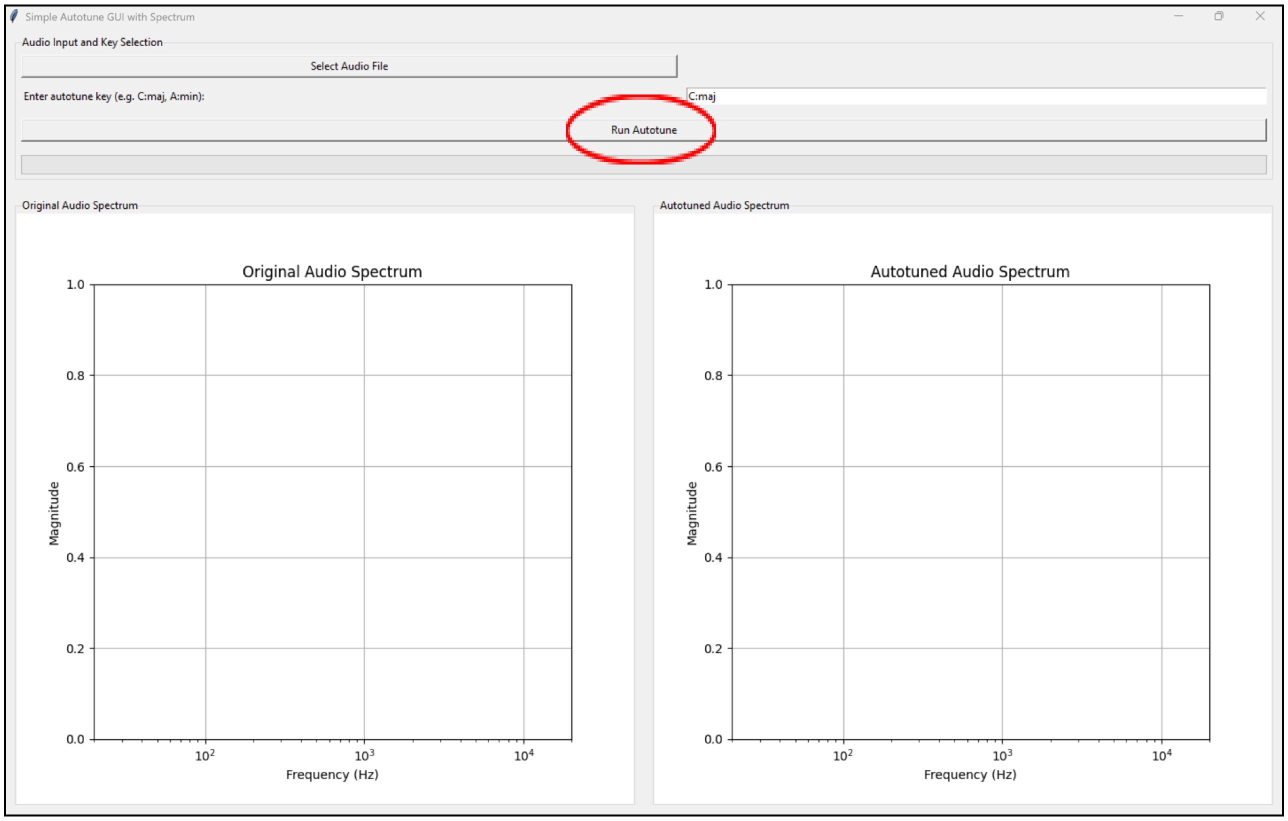
3. **Enter the Autotune Key:**
   - In the "Enter autotune key (e.g. C:maj, A:min):" field, type the desired musical key for autotuning.



Enter autotune key (e.g. C:maj, A:min): C:maj

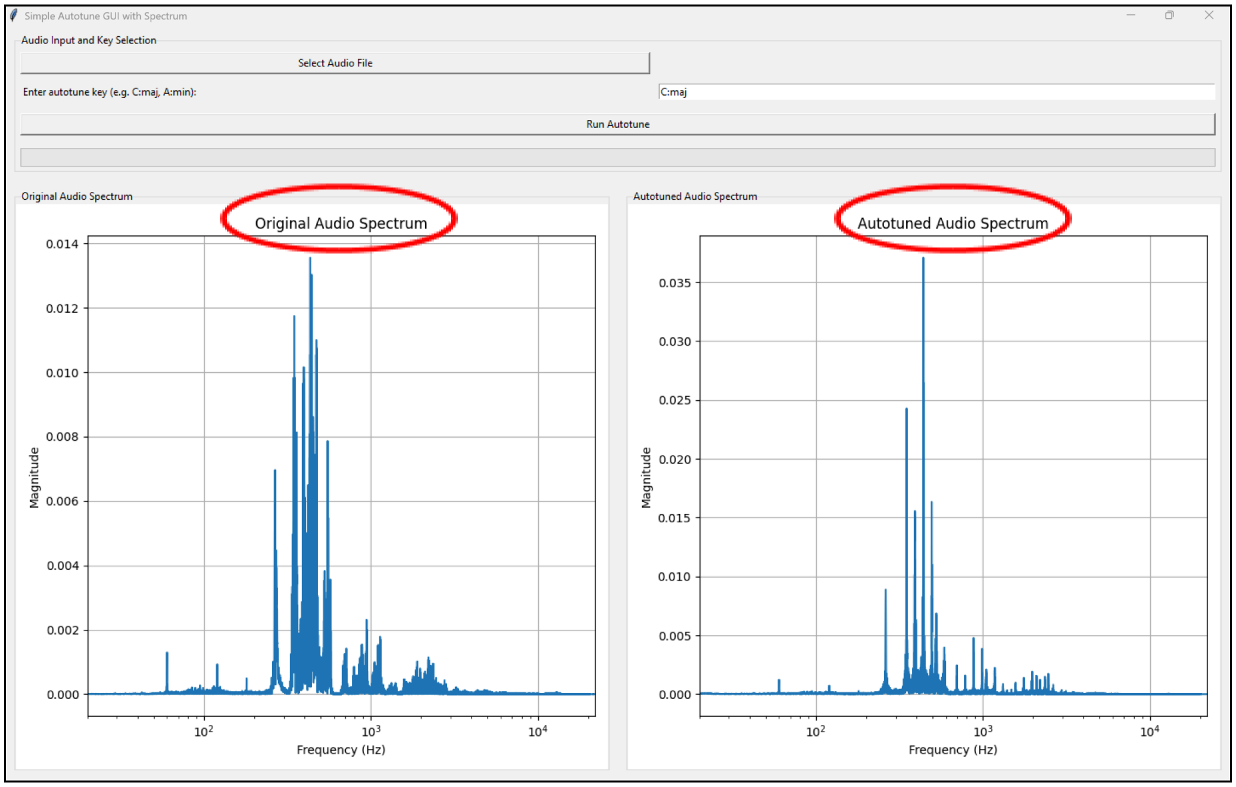
   - **Example Formats:**
     - `C:maj` (C Major)
     - `A:min` (A Minor)
     - `E:dorian` (E Dorian mode)
   - The field defaults to "C:maj".

4. **Run Autotune:**
   - Once an audio file is selected and a key is entered, click the "**Run Autotune**" button.



- The button will become disabled during processing, and a progress bar will update to show the autotuning's progress.

5. **Processing and Playback:**
   - The application will load the audio, perform pitch detection, apply pitch correction based on the selected key, and re-synthesize the audio.
   - Upon completion, a success message will appear, indicating the processing time, and the autotuned audio will automatically begin playing.
   - The application will wait for the playback to finish.
6. **View Spectrum Analysis:**
   - After the autotuning process and playback are complete, two plots will be displayed in the GUI:
     - **"Original Audio Spectrum"**: Shows the frequency spectrum of the input audio file.
     - **"Autotuned Audio Spectrum"**: Displays the frequency spectrum of the processed, autotuned audio, allowing for a visual comparison of the effect.


Simple Autotune GUI with Spectrum

Audio Input and Key Selection

Select Audio File

Enter autotune key (e.g. C:maj, A:min): C:maj

Run Autotune

Original Audio Spectrum

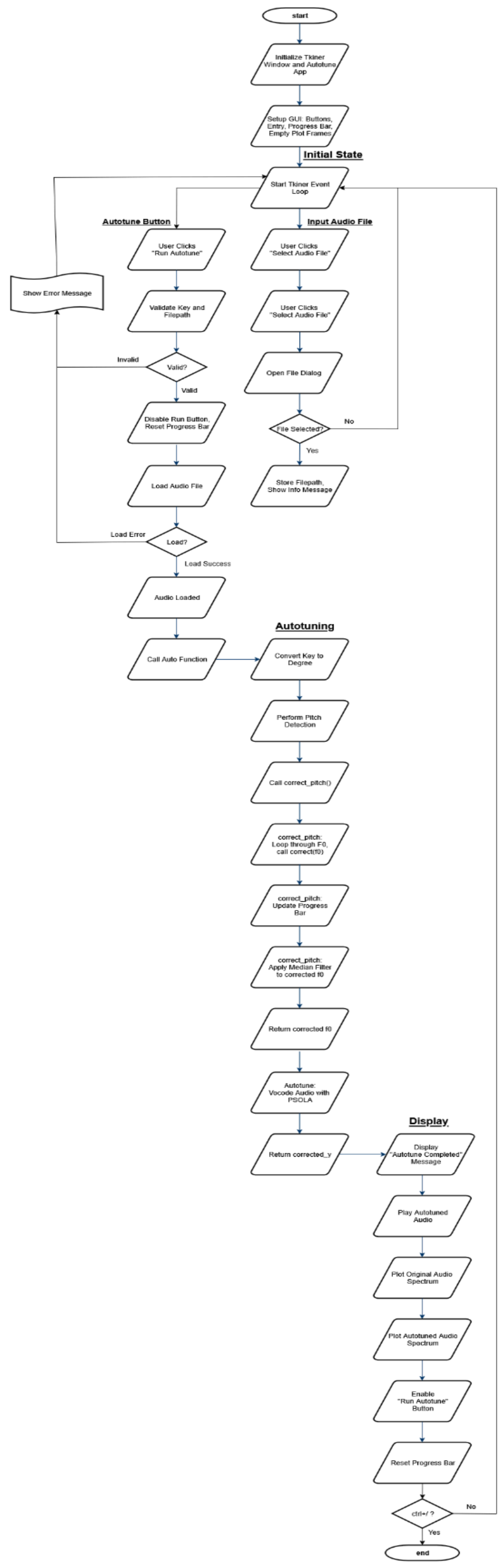
Autotuned Audio Spectrum

7. **Reset for New Operation:**
    - After the process is finished, the "Run Autotune" button will be re-enabled, and the progress bar will reset, allowing you to select a new file or key and run the autotune again.

Hardware Design Description

The project relies entirely on software processing on a full computer rather than a microcontroller. This adjustment was crucial to overcome limitations in processing large sample data with low latency, which had previously hindered the achievement of real-time autotune with a microcontroller. The system effectively leverages the computational power of a standard computer to handle the complex algorithms involved in pitch detection, correction, and audio synthesis. No custom hardware was developed beyond the use of a standard computer system to run the software.

Flow Diagram


start
Initialize Tkiner Window and Autotune App
Setup GUI: Buttons, Entry, Progress Bar, Empty Plot Frames
Initial State
Start Tkiner Event Loop
Autotune Button
User Clicks "Run Autotune"
Validate Key and Filepath
Valid?
Invalid
Valid
Show Error Message
Disable Run Button, Reset Progress Bar
Load Audio File
Load?
Load Error
Load Success
Audio Loaded
Call Auto Function
Input Audio File
User Clicks "Select Audio File"
User Clicks "Select Audio File"
Open File Dialog
File Selected?
No
Yes
Store Filepath, Show Info Message
Autotuning
Convert Key to Degree
Perform Pitch Detection
Call correct_pitch()
correct_pitch: Loop through F0, call correct(f0)
correct_pitch: Update Progress Bar
correct_pitch: Apply Median Filter to corrected f0
Return corrected f0
Autotune: Vocode Audio with PSOLA
Return corrected_y
Display
Display "Autotune Completed" Message
Play Autotuned Audio
Plot Original Audio Spectrum
Plot Autotuned Audio Spectrum
Enable "Run Autotune" Button
Reset Progress Bar
ctrl+/ ?
No
Yes
end

Code Breakdown and Mathematical Explanations

The autotune application's functionality is structured around key Python libraries and core functions for audio processing, pitch manipulation, and user interface management.

1. Core Libraries and Their Roles

The project leverages a select set of powerful libraries, each chosen for its specific capabilities crucial to the autotune pipeline:

- **librosa**: The cornerstone for audio analysis. It's used for efficient audio loading, robust fundamental frequency (pitch) estimation via the pyin algorithm, and essential musical conversions (e.g., Hertz to MIDI, musical key strings to pitch degrees). Its optimization for large audio datasets is fundamental to performance.
- **psola**: Specifically chosen for high-quality audio synthesis and pitch shifting using the Pitch Synchronous Overlap and Add (PSOLA) technique. PSOLA's ability to preserve the original audio's timbral characteristics (formants) makes it ideal for natural-sounding pitch correction.
- **numpy**: Provides the foundational numerical computing capabilities for high-performance array manipulations, which are essential for processing audio data efficiently.
- **scipy.signal**: Utilized for signal processing, primarily sig.medfilt (median filter). This non-linear filter is critical for smoothing detected pitch contours, effectively removing spurious pitch detections or "spikes" without blurring desired pitch transitions.
- **tkinter (with filedialog, messagebox, ttk)**: Python's standard library for creating the Graphical User Interface (GUI). It enables user interaction (file selection, key input), provides visual feedback (progress bar, spectrum plots), and handles informational or error messages.

2. correct(f0, selected_key_degrees) Function: The Pitch Snapping Logic

This function implements the core pitch quantization for a single fundamental frequency (f0), mapping it to the nearest note in the target musical scale.

- **Process:**
    1. **NaN Handling:** Skips processing if f0 is NaN (no confident pitch detection), propagating the NaN to prevent artifacts.

2. **Hz to MIDI:** Converts the input frequency (f0 in Hz) to a MIDI note number (midi_note). This is crucial because musical scales are defined by semitone intervals, which directly correspond to MIDI increments.
    - **Mathematics:** The conversion uses a logarithmic relationship: M = 12 * log2(f/440) + 69.
3. **Pitch Class Extraction:** Calculates the pitch class (degree) of the MIDI note using the modulo 12 operation (midi_note % 12). This normalizes the note to a value between 0 and 11, representing its position within a single octave (e.g., C=0, C#=1).
4. **Closest Note Determination:** Identifies the closest target note within the selected_key_degrees array (which contains the pitch classes of the chosen musical scale, intelligently extended to handle octave wrap-around). This is achieved by finding the minimum absolute difference between the detected degree and the scale degrees.
5. **Pitch Correction:** Calculates the degree_difference (the semitone offset from the detected pitch to the closest scale note) and subtracts it from the midi_note. This effectively shifts the midi_note to align with the scale.
6. **MIDI to Hz:** Converts the corrected MIDI note back to its corresponding frequency in Hertz.
    - **Mathematics:** The inverse conversion: f = 440 * 2^((M-69)/12).

3. autotune(y, sr, selected_key, progress_callback=None) Function: The Orchestrator

This function manages the entire autotune pipeline, from initial audio processing to final synthesis.

- **Framing Parameters:** Defines how the audio is segmented for analysis:
    - frame_length = 2048: Number of samples per analysis window (approx. 46ms at 44.1kHz).
    - hop_length = frame_length // 4: Number of samples between frame starts, creating a 75% overlap for smooth analysis and synthesis.
- **Frequency Bounds (fmin, fmax):** Sets the search range for pitch detection (e.g., C2 to C7). This improves pyin's accuracy by focusing on musically relevant frequencies.
- **Key Conversion:** Transforms the user-selected musical key string (e.g., "C:maj") into an array of pitch degrees, including an octave extension (+12) for robust snapping across octave boundaries.

KENNESAW STATE UNIVERSITY

- **Pitch Detection (librosa.pyin):**
    - This is a sophisticated algorithm that estimates the fundamental frequency (f0) contour of the audio over time. It is based on the YIN algorithm, enhanced with a probabilistic model for improved accuracy in noisy conditions.
    - It outputs an array of f0 values, one for each analysis frame.
- **Pitch Correction (correct_pitch call):** Passes the detected f0 contour to the correct_pitch function (which iteratively applies the correct function and then smooths the result with a median filter), returning the scale-corrected f0 contour.
- **Audio Synthesis (psola.vocode):**
    - This is the final step, re-synthesizing the audio with the corrected pitch. PSOLA works by analyzing pitch periods in the original audio, extracting short "grains," and then repositioning/overlapping these grains according to the target_pitch (our corrected_f0).
    - It excels at preserving natural vocal characteristics (formants) compared to simpler frequency-domain methods, leading to higher quality autotuned audio.

4. AutotuneApp Class: User Interface and Workflow Management

This class encapsulates the entire application logic, providing a user-friendly Graphical User Interface (GUI) for interaction and visualization.

- **GUI Setup:** Initializes the Tkinter window and arranges widgets (buttons, input fields, progress bar, plot frames) using a responsive grid layout.
- **File Selection:** Manages opening a file dialog (filedialog.askopenfilename) for users to select audio files, including fallback to a default file for convenience.
- **Spectrum Plotting (plot_spectrum):** A helper method that calculates the magnitude spectrum of audio data using the Fast Fourier Transform (np.fft.fft) and displays it within embedded Matplotlib figures. This provides visual feedback on the autotune effect.
- **run_autotune Method:**
    - This is the main event handler, triggered by the "Run Autotune" button.
    - It validates user input (musical key), loads audio, and disables the button during processing.
    - Crucially, it defines an update_progress callback, which is passed down to the core autotune functions to update the GUI's progress bar in real-time, preventing the interface from freezing.

- It calls the autotune function, measures processing time, and then plays the corrected audio (sounddevice.play).
- Comprehensive try-except-finally blocks ensure robust error handling during file loading and processing, and guarantee that the GUI returns to an interactive state even if errors occur.

Entire Code

```
import librosa
from pathlib import Path
import psola
import numpy as np
import scipy.signal as sig
import sounddevice as sd
import time
import sys
import tkinter as tk
from tkinter import filedialog, messagebox
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg,
NavigationToolbar2Tk

# --- Pitch correction functions (same as original) ---

def correct(f0, selected_key_degrees):
    """
    Corrects a single fundamental frequency (f0) to the nearest degree in
the selected key.
    """
    if np.isnan(f0):
        return np.nan
    midi_note = librosa.hz_to_midi(f0)
    degree = midi_note % 12  # Get the degree within an octave (0-11)

    # Find the closest degree in the selected key
    closest_degree_id = np.argmin(np.abs(selected_key_degrees - degree))
    degree_difference = degree - selected_key_degrees[closest_degree_id]

    midi_note -= degree_difference # Shift the MIDI note to align with the
closest key degree
    return librosa.midi_to_hz(midi_note)

def correct_pitch(f0, selected_key_degrees, progress_callback=None):
    """
    Applies pitch correction to an array of fundamental frequencies (f0).
    Includes a progress callback for GUI updates.
    """
    corrected_f0 = np.zeros_like(f0)
    total = f0.shape[0]

    for i in range(total):
        corrected_f0[i] = correct(f0[i], selected_key_degrees)
        # Update progress periodically to avoid slowing down the loop too
much
        if progress_callback and i % 50 == 0: # Update every 50 frames
            progress_callback(i / total * 100)

    # Ensure final progress update
    if progress_callback:
        progress_callback(100)

    # Apply median filtering for smoothing, handling NaNs
    smoothed_corrected_f0 = sig.medfilt(corrected_f0, kernel_size=11)
    # Fill back NaNs where median filter might have introduced them or
original NaNs
    smoothed_corrected_f0[np.isnan(smoothed_corrected_f0)] =
f0[np.isnan(smoothed_corrected_f0)]
    return smoothed_corrected_f0

def autotune(y, sr, selected_key, progress_callback=None):
    """
    Performs the full autotune process: pitch detection, correction, and
vocoding.
    """
    frame_length = 2048
    hop_length = frame_length // 4
    fmin = librosa.note_to_hz('C2') # Minimum frequency for pitch
detection
    fmax = librosa.note_to_hz('C7') # Maximum frequency for pitch
detection

    try:
        # Convert the selected key (e.g., 'C:maj') to an array of MIDI
degrees
        selected_key_degrees = librosa.key_to_degrees(selected_key)
        # Add an octave higher for wrapping around (e.g., C major includes
C, D, E, F, G, A, B, C+12)
        selected_key_degrees = np.concatenate((selected_key_degrees,
[selected_key_degrees[0] + 12]))
    except Exception as e:
        raise ValueError(f"Invalid key '{selected_key}'. Please use format
like 'C:maj' or 'A:min'.")

    # Perform pitch detection using pYIN
    # f0: fundamental frequency contour
    # voiced_flag: boolean array indicating voiced segments
    # voiced_prob: probability of being voiced
    f0, voiced_flag, voiced_prob = librosa.pyin(
        y, frame_length=frame_length, hop_length=hop_length,
        sr=sr, fmin=fmin, fmax=fmax
    )

    # Correct the detected pitches
    corrected_f0 = correct_pitch(f0, selected_key_degrees,
progress_callback)

    # Use PSOLA (Pitch Synchronous Overlap and Add) to vocode the audio
    # This resynthesizes the audio with the corrected pitch contour
    return psola.vocode(y, sample_rate=int(sr), target_pitch=corrected_f0,
fmin=fmin, fmax=fmax)

# --- GUI ---

class AutotuneApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Simple Autotune GUI with Spectrum")
        self.filepath = None

        # Configure grid layout for better organization
        self.root.columnconfigure(0, weight=1)
        self.root.columnconfigure(1, weight=1)
        self.root.rowconfigure(3, weight=1) # Row for plots

        # Input Frame
        input_frame = ttk.LabelFrame(root, text="Audio Input and Key
Selection")
        input_frame.grid(row=0, column=0, columnspan=2, padx=10, pady=10,
sticky="ew")
        input_frame.columnconfigure(0, weight=1)
        input_frame.columnconfigure(1, weight=1)

        self.file_btn = tk.Button(input_frame, text="Select Audio File",
command=self.select_file)
        self.file_btn.grid(row=0, column=0, padx=5, pady=5, sticky="ew")

        self.key_label = tk.Label(input_frame, text="Enter autotune key
(e.g. C:maj, A:min):")
        self.key_label.grid(row=1, column=0, padx=5, pady=5, sticky="w")

        self.key_entry = tk.Entry(input_frame)
        self.key_entry.grid(row=1, column=1, padx=5, pady=5, sticky="ew")
        self.key_entry.insert(0, "C:maj") # Default key

        self.run_btn = tk.Button(input_frame, text="Run Autotune",
command=self.run_autotune)
        self.run_btn.grid(row=2, column=0, columnspan=2, padx=5, pady=10,
sticky="ew")

        self.progress = ttk.Progressbar(input_frame, length=300,
mode="determinate")
        self.progress.grid(row=3, column=0, columnspan=2, padx=5, pady=5,
sticky="ew")

        # Plotting Setup
        # Frame for original audio spectrum
        original_plot_frame = ttk.LabelFrame(root, text="Original Audio
Spectrum")
        original_plot_frame.grid(row=3, column=0, padx=10, pady=10,
sticky="nsew")
        original_plot_frame.columnconfigure(0, weight=1)
        original_plot_frame.rowconfigure(0, weight=1)

        self.figure_original = plt.Figure(figsize=(5, 4), dpi=100)
        self.ax_original = self.figure_original.add_subplot(111)
        self.canvas_original = FigureCanvasTkAgg(self.figure_original,
master=original_plot_frame)
        self.canvas_original_widget = self.canvas_original.get_tk_widget()
        self.canvas_original_widget.pack(side=tk.TOP, fill=tk.BOTH,
expand=1)

        # Frame for autotuned audio spectrum
        autotuned_plot_frame = ttk.LabelFrame(root, text="Autotuned Audio
Spectrum")
        autotuned_plot_frame.grid(row=3, column=1, padx=10, pady=10,
sticky="nsew")
        autotuned_plot_frame.columnconfigure(0, weight=1)
        autotuned_plot_frame.rowconfigure(0, weight=1)

        self.figure_autotuned = plt.Figure(figsize=(5, 4), dpi=100)
        self.ax_autotuned = self.figure_autotuned.add_subplot(111)
        self.canvas_autotuned = FigureCanvasTkAgg(self.figure_autotuned,
master=autotuned_plot_frame)
        self.canvas_autotuned_widget =
self.canvas_autotuned.get_tk_widget()
        self.canvas_autotuned_widget.pack(side=tk.TOP, fill=tk.BOTH,
expand=1)

        # Initial plot setup (empty plots)
        self.ax_original.set_title("Original Audio Spectrum")
        self.ax_original.set_xlabel("Frequency (Hz)")
        self.ax_original.set_ylabel("Magnitude")
        self.ax_original.set_xscale('log')
        self.ax_original.set_xlim([20, 20000])
        self.ax_original.grid(True)
        self.canvas_original.draw()

        self.ax_autotuned.set_title("Autotuned Audio Spectrum")
        self.ax_autotuned.set_xlabel("Frequency (Hz)")
        self.ax_autotuned.set_ylabel("Magnitude")
        self.ax_autotuned.set_xscale('log')
        self.ax_autotuned.set_xlim([20, 20000])
        self.ax_autotuned.grid(True)
        self.canvas_autotuned.draw()
```

```
    def select_file(self):
        """
        Opens a file dialog for the user to select an audio file.
        """
        file = filedialog.askopenfilename(filetypes=[("WAV files",
"*.wav"), ("All files", "*.*")])
        if file:
            self.filepath = file
            messagebox.showinfo("File Selected", f"Selected
file:\n{Path(file).name}")
        else:
            self.filepath = None # Clear filepath if selection is
cancelled

    def plot_spectrum(self, ax, audio_data, sample_rate, title):
        """
        Calculates and plots the magnitude spectrum of the given audio
data.
        """
        N = len(audio_data)
        # Compute the Fast Fourier Transform
        yf = np.fft.fft(audio_data)
        # Compute the corresponding frequencies
        xf = np.fft.fftfreq(N, 1 / sample_rate)[:N//2] # Only positive
frequencies

        ax.clear() # Clear previous plot
        ax.plot(xf, 2.0/N * np.abs(yf[0:N//2])) # Plot magnitude spectrum
        ax.set_title(title)
        ax.set_xlabel("Frequency (Hz)")
        ax.set_ylabel("Magnitude")
        ax.set_xscale('log') # Use logarithmic scale for frequency
        ax.set_xlim([20, sample_rate / 2]) # Limit x-axis to relevant
frequencies (20Hz to Nyquist)
        ax.grid(True)
        self.figure_original.tight_layout()
        self.figure_autotuned.tight_layout()


    def run_autotune(self):
        """
        Executes the autotune process, loads audio, applies autotune,
        plays the result, and plots the spectra.
        """
        key = self.key_entry.get().strip()
        if not key:
            messagebox.showerror("Input Error", "Please enter a musical
key (e.g., C:maj).")
            return

        if not self.filepath:
            # Fallback to a default file if no file is selected
            default_path = Path(__file__).parent / "monitoring.wav"
            if default_path.exists():
                self.filepath = str(default_path)
                messagebox.showinfo("Fallback", "No file selected. Using
default: monitoring.wav")
            else:
                messagebox.showerror("File Error", "No file selected and
'monitoring.wav' not found in the script directory.")
                return

        try:
            # Load audio file
            y, sr = librosa.load(self.filepath, sr=None, mono=False)
            if y.ndim > 1:
                y = y[0, :]  # Use left channel only for processing
        except Exception as e:
            messagebox.showerror("Load Error", f"Failed to load audio:
{e}")
            return

        def update_progress(p):
            """Callback function to update the progress bar."""
            self.progress["value"] = p
            self.root.update_idletasks() # Force GUI update

        try:
            self.run_btn.config(state="disabled") # Disable button during
processing
```

```
            self.progress["value"] = 0 # Reset progress bar

            start_time = time.time()
            # Run the autotune algorithm
            corrected_y = autotune(y, sr, key, update_progress)
            elapsed = time.time() - start_time

            # Display success message and play audio
            messagebox.showinfo("Success", f"Autotune completed in
{elapsed:.2f} seconds.\nPlaying result...")
            sd.play(corrected_y, samplerate=sr)
            sd.wait() # Wait for playback to finish

            # Plot spectra
            self.plot_spectrum(self.ax_original, y, sr, "Original Audio
Spectrum")
            self.canvas_original.draw()

            self.plot_spectrum(self.ax_autotuned, corrected_y, sr,
"Autotuned Audio Spectrum")
            self.canvas_autotuned.draw()

        except Exception as e:
            messagebox.showerror("Processing Error", str(e))
        finally:
            self.run_btn.config(state="normal") # Re-enable button
            self.progress["value"] = 0 # Reset progress bar

if __name__ == "__main__":
    root = tk.Tk()
    app = AutotuneApp(root)
    root.mainloop()
```

Conclusion

This project aimed to implement an autotune effect on audio, shifting pitches to align with a specified musical scale. While the initial goal was real-time processing, it transitioned to an **offline software-centric solution** due to high computational demands and time constraints.

The system features a **Graphical User Interface (GUI)** for user interaction. Its core involves:

1. **Pitch Detection:** Identifying the fundamental frequency of audio over time using `librosa.pyin`.
2. **Pitch Correction:** Snapping detected pitches to the closest note in the chosen musical key, smoothed with a median filter.
3. **Audio Synthesis:** Re-creating the audio with corrected pitches using the **PSOLA** technique.

The project is entirely **software-based**, running on a standard computer, which allowed for the necessary processing power that a microcontroller couldn't provide. Key Python libraries include `librosa`, `psola`, `numpy`, `scipy.signal`, `sounddevice`, `tkinter`, and `matplotlib`.

Limitations included the inability to achieve true real-time autotune within the project's timeframe and the tight summer semester schedule. Future enhancements could focus on achieving real-time capability through performance optimization, hardware acceleration, and adding advanced features like formant correction and vibrato control for more natural-sounding results.

Reference(s):

➢ **Librosa: https://librosa.org/doc/latest/index.html**

[1] McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. “librosa: Audio and music signal analysis in python.” In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

➢ **PSOLA: https://pypi.org/project/psola/**

[1] Y. Jadoul, B. Thompson, and B. De Boer, "Introducing parselmouth: A python interface to praat," Journal of Phonetics, vol. 71, pp. 1â€“15, 2018.

[2] P. Boersma, "Praat: doing phonetics by computer", http://www.praat.org/, 2006.

[3] E. Moulines and F. Charpentier, "Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones," Speech communication, 1990.