A
Project Report
on
PITCH TRANSPOSER – AUDIO SAMPLER

Submitted By

GUNDLURU JEEVAN [R200783]

Under the Guidance of

Dr. SK MAHAMMAD RAFI

M.Tech (JNTU Hyderabad), Ph.D (IIT Hydearabad)

Assistant Professor

Electronics And Communication Engineering

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES (APIIIT)

R K VALLEY, VEMPALLI, YSR (Dist.) – 516330

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

2024-2025

Rajiv Gandhi University of Knowledge Technologies

RK Valley, YSR (Dist.), Andhra Pradesh - 516330

CERTIFICATE

This is to certify that the project report titled **"Pitch Transposer – Audio Sampler"**, carried out by **G. Jeevan (R200783)**, has been completed under my guidance and supervision. This report is submitted to the **Department of Electronics and Communication Engineering** in fulfilment of the requirements for the **Mini Project**, as part of the curriculum for the **Bachelor of Technology in Electronics and Communication Engineering** during the **Academic Year 2024–2025**. The work embodied in this report has been reviewed and is found to be in accordance with the academic requirements of the University.

Signature of Internal Guide

Dr. SK. Mahammad Rafi

M.Tech, Ph.D.

Assistant Professor

Electronics and Communication Engineering

RGUKT RK VALLEY

Signature of HOD

Dr. Y Arun Kumar Reddy

M.Tech, Ph.D.

Head of the Department

Electronics and Communication Engineering

RGUKT RK VALLEY

Signature of External Examiner

DECLARATION

I hereby declare that the project work titled **"Pitch Transposer – Audio Sampler"**, submitted to the **Department of Electronics and Communication Engineering**, is a genuine piece of work carried out by our team under the guidance of **Dr. SK. Mahammad Rafi**. This report is submitted in fulfilment of the requirements for the **Mini Project**, as part of the curriculum for the **Bachelor of Technology in Electronics and Communication Engineering** during the **Academic Year 2024–2025**.

I further declare that this work has not been submitted elsewhere for the award of any other degree or diploma.

G. Jeevan (R200783)

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to all those who supported and guided us throughout the successful completion of this mini-project titled **"Pitch Transposer – Audio Sampler."**

We are deeply thankful to our project guide, **Dr. . SK. Mahammad Rafi** *M.Tech, Ph.D.*, for his invaluable guidance, constant encouragement, and constructive suggestions at every stage of the project. His mentorship played a crucial role in shaping the direction and outcome of our work.

We extend our heartfelt thanks to **Dr. Y Arun Kumar Reddy** *M.Tech, Ph.D.*, **Head of the Department, Electronics and Communication Engineering**, for him continuous support and for providing us with the resources and academic environment necessary for this project.

We are also grateful to **Prof. A. V. S. S. Kumara Swami Gupta** *M.Tech, Ph.D.*, **Director of RGUKT RK Valley**, for his encouragement and for fostering a culture of research and innovation.

Our sincere thanks go to the **faculty members of the Department of Electronics and Communication Engineering** for their guidance, and academic support.

Finally, we appreciate the collaborative efforts of our team members and the support extended by our peers, which made this project a rewarding learning experience.

G. Jeevan (R200783)

CONTENTS

ABSTRACT

This project focuses on developing a real-time, interactive audio pitch shifting system that processes monophonic audio signals through a time-domain overlap-add method combined with linear interpolation. The core of the system involves detecting the fundamental frequency of an input waveform using the YIN algorithm, mapping it to a MIDI representation, and then transforming the pitch by resampling individual audio windows. Each window is stretched or compressed based on the desired pitch shift and weighted using a Hanning window to ensure smooth transitions.

The project includes a user-friendly Gradio interface that allows users to interact with the system by uploading custom audio, selecting preloaded instrument samples (like guitar), and triggering pitch-shifted notes via keyboard input mapped to MIDI note numbers. The processed audio samples are dynamically generated and saved, enabling flexible real-time playback. This tool is particularly useful for educational demonstrations in signal processing, experimental music applications, and digital instrument development. It bridges fundamental audio analysis techniques with hands-on, creative interaction, providing a practical foundation for more advanced music technology projects.

CHAPTER 1
INTRODUCTION

1.1 Background

Pitch-shifting is a technique used to modify the perceived pitch of an audio signal without altering its tempo. Applications range from music production and instrument synthesis i.e creating an instrument of anything. Traditional pitch shifting uses frequency-domain methods like phase vocoders but can introduce artifacts and require complex computation. This project explores a time-domain alternative using resampling, interpolation, and overlap-add to shift pitch while maintaining natural audio quality.

By integrating pitch detection, MIDI mapping and playback, the project's core operations involved in pitch shifting and creating an interface for instrument for easy access.

The primary objectives of this project are:

- To detect the fundamental frequency of an audio signal using the YIN algorithm.
- To convert the detected pitch into its corresponding MIDI value.
- To shift the pitch of the audio signal using time-domain interpolation and overlap-add.
- To implement an interactive Gradio interface for uploading audio, triggering note playback via keyboard, and visualizing pitch changes.

By achieving these goals, the system serves as both a learning tool and a foundation for advanced development in audio signal processing. It gives the user the first step towards music production and creation

CHAPTER 2
PILLAR ALGORITHMS

To understand the Pitch transposal, we first need to know about algorithms to detect the frequency of the given audio sample and the altercation of frequency

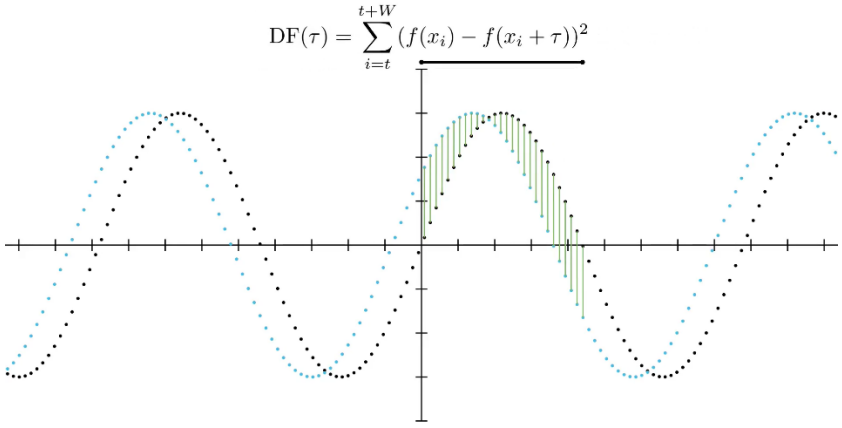
2.1 YIN algorithm

The YIN algorithm is a fundamental pitch detection method introduced by Alain de Cheveigné and Hideki Kawahara in 2002. It is designed to estimate the fundamental frequency (f_0) of monophonic audio signals with high accuracy and robustness, even in the presence of noise or harmonic interference. The algorithm works by computing a modified difference function over a range of lag values, identifying periodicity in the signal. It then normalizes this function using a cumulative mean to suppress spurious minima and emphasizes true pitch candidates. The lag corresponding to the first significant dip below a threshold in the normalized function is interpreted as the signal's pitch period, which is then converted into a frequency using the sample rate.

One of YIN's key strengths is its balance between computational efficiency and perceptual accuracy, making it suitable for both offline analysis and real-time applications. Unlike zero-crossing or autocorrelation-based methods, YIN offers improved resolution and fewer octave errors, particularly in low and mid-frequency ranges.

2.1.1 Procedure

- YIN algorithm starts with taking two waveforms, the original waveform an another time delayed waveform
- The difference function $DF(\tau)$ is calculated



The difference function is the sum of square of difference between the original waveform and the delayed waveform in the time interval *t* to *t+W*.

- The DF(τ) is represented by Auto Correlation Function ACF(τ)

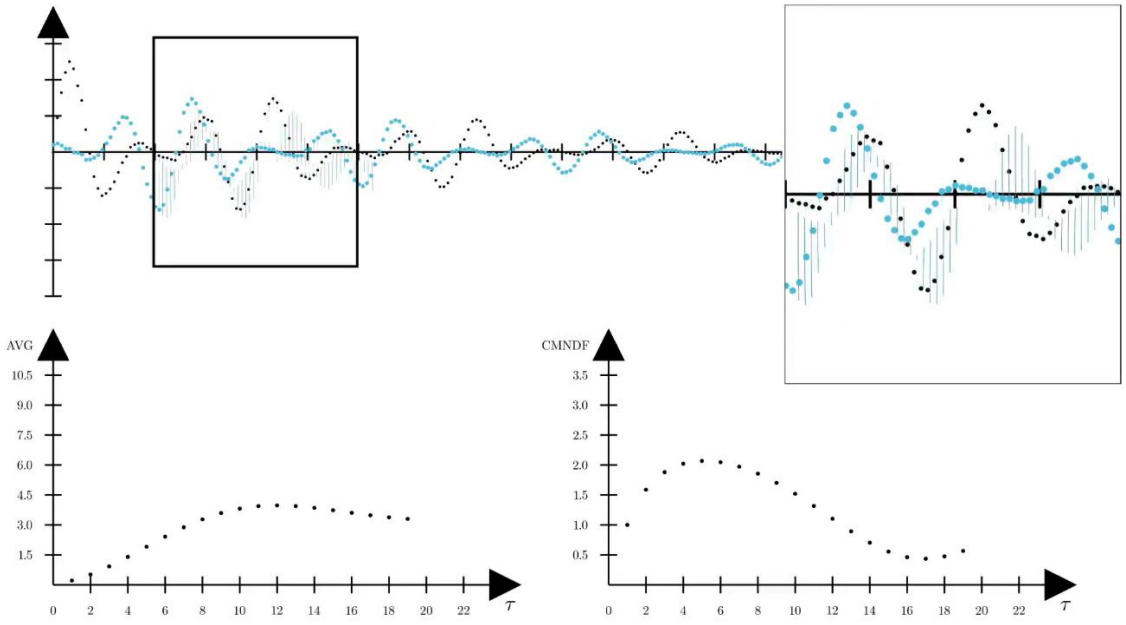
$$\mathrm{ACF}(\tau, t) = \sum_{i=t}^{t+W} f(x_i) \cdot f(x_i+\tau)$$

$$\begin{aligned}\mathrm{DF}(\tau) &= \sum_{i=t}^{t+W} (f(x_i) - f(x_i+\tau))^2 \\ &= \sum_{i=t}^{t+W} \left(f(x_i)^2 + f(x_i+\tau)^2 - 2 \cdot f(x_i) \cdot f(x_i+\tau)\right) \\ &= \mathrm{ACF}(0,t) + \mathrm{ACF}(0,t+\tau) - 2 \cdot \mathrm{ACF}(\tau,t)\end{aligned}$$

- This DF(τ) is calculated to given range of frequencies, in our case, from 27.5Hz(A0) to 4186Hz(C8) through iterations and mapped in an array.
- Now Cumulative Mean Normalised Difference Function is calculated to give better results and to Normalise the result.

$$\mathrm{CMNDF}(\tau) = \begin{cases} 1 & \tau = 0 \\ \tau \cdot \frac{\mathrm{DF}(\tau)}{\sum_{j=1}^{\tau} \mathrm{DF}(j)} & \tau > 0 \end{cases}$$

- After finding the minimum of the vector of DF(τ) the CMNDF(τ) is calculated to get the most appropriate delay.



- The delay is now converted to frequency by the formula, the sample rate of audio is commonly 44100 Hz

$$f_0 = \frac{\text{sample rate}}{\tau}$$

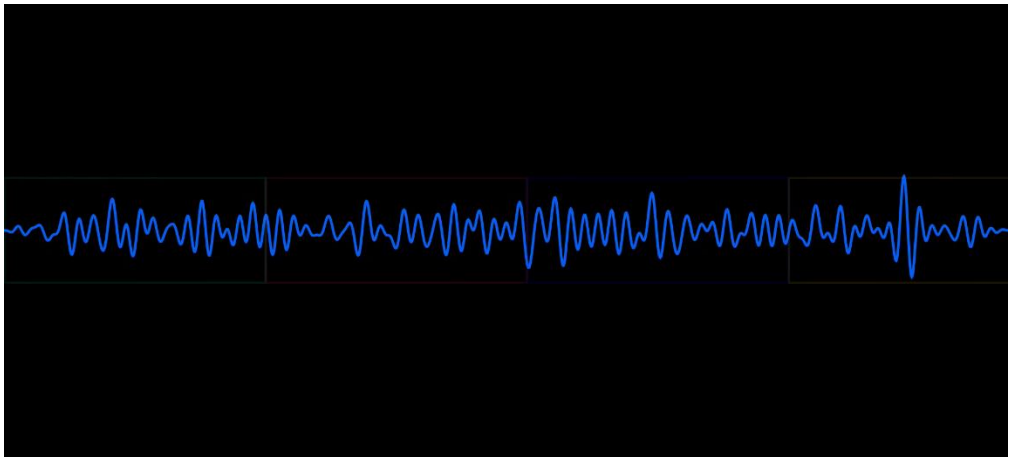
2.2 Overlapp Add

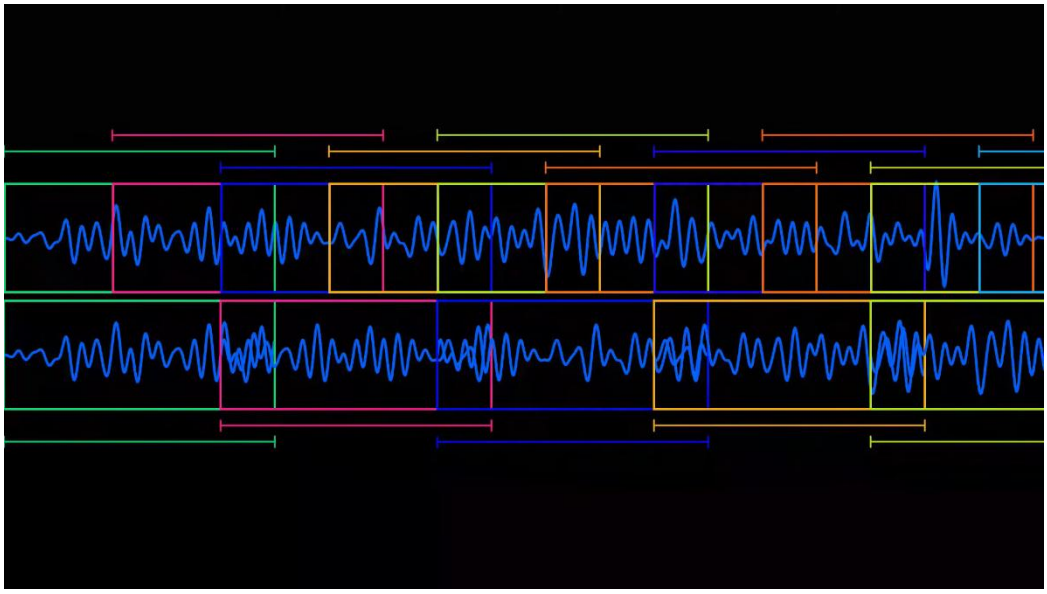
The overlap-add (OLA) method is a fundamental signal processing technique used to reconstruct or modify signals by combining overlapping, windowed segments of audio. In the context of time-domain pitch shifting, OLA works by dividing the input waveform into overlapping frames, applying a smoothing window (like the Hanning window) to each segment, and then adding these segments back together at staggered positions in the output buffer. This process ensures smooth transitions between frames, minimizes discontinuities, and helps maintain natural-sounding audio, especially when the length of each window has been altered to shift pitch. Overlap-add is widely used in applications such as time-stretching, filtering, and synthesis because it enables efficient and flexible signal manipulation with reduced artifacts.

2.2.1 Procedure

1. Frame the Signal

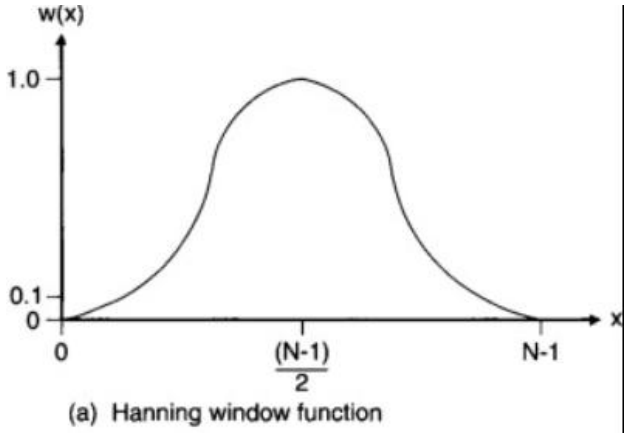
- Divide the input audio signal into **overlapping segments** (or frames).
- Use a fixed **window size** (e.g., 5000 samples) and a **hop length** (e.g., 2000 samples).
- Overlapping ensures continuity across segment boundaries.





2. Apply a Window Function

- Multiply each frame by a **smoothing window** (e.g., Hanning or Hamming).
- This reduces edge discontinuities and smoothens transitions between frames.



(a) Hanning window function

3. Modify Each Window

- If pitch shifting or time-stretching, **interpolate or resample** each frame to the new target size.
- This alters the length of each segment, changing the signal's time or pitch characteristics.

4. Add Frames to Output Buffer

- Place each windowed frame at its corresponding position in the output signal.
- **Add** the frame values to existing values in the buffer where overlaps occur.

5. Normalize the Output

- Because overlapping frames can cause certain regions to be **louder**, track the **sum of window weights**
- Divide the output by this sum to ensure **consistent amplitude**.

2.3 MIDI calculator

MIDI (Musical Instrument Digital Interface) is a digital communication protocol that allows electronic musical instruments, computers, and other devices to exchange information about music. Instead of transmitting actual audio, MIDI sends data such as note pitch, duration, velocity (intensity), and control signals. Each musical note is assigned a specific **MIDI number**—for example, A4 (440 Hz) corresponds to MIDI note 69. This numerical system enables consistent representation of notes across software and hardware, making MIDI essential for applications in music production, synthesizers, virtual instruments, and pitch-related audio processing like in your project.

$$n = 12\, log_2 \left(\frac{f_n}{f_0} \right)$$

CHAPTER 3

SOURCE CODE AND LOGIC

3.1Technologies Used

```
import numpy as np
import librosa
import soundfile as sf
import sounddevice as sd
from pygame import mixer
from IPython.display import display, Audio
import os
import time
import shutil
import gradio as gr
```

- **NumPy**: Used for numerical operations like array creation, interpolation, windowing (np.hanning()), and mathematical functions such as np.log2, np.linspace, etc. It's the backbone for signal processing math.
- **Librosa** – A Python library for music and audio analysis.
  Used for: loading audio files (librosa.load()), pitch detection using YIN (librosa.yin()), and converting frequency to MIDI notes.
- **SoundFile** – For reading and writing audio files in WAV and FLAC formats.
  Used for: saving the pitch-shifted audio to disk using sf.write().
- **Pygame Mixer** – A module for simple sound playback in games.
  Used for: playing audio clips (mixer.music.load() and mixer.music.play()), especially the pitch-shifted samples.
- **IPython.display** – Allows inline audio playback in Jupyter notebooks.
  Used for: previewing audio in notebooks using display(Audio(...)).**OS** – Provides functions to interact with the file system.
  Used for: creating directories (os.makedirs()), splitting filenames, and path operations.

- **Time** – Provides time-related functions.
  Used optionally for adding delays (time.sleep()) between audio playback in loops (commented out in your version).
- **Gradio** – A UI framework for building web-based interfaces for ML and audio apps.
  Used for: creating the interactive interface for uploading files, playing notes, and selecting samples.

3.2Functions and Blocks used

- The frequency_finder(f0) function extracts the most prominent pitch frequency. If a stable pitch is detected, it converts the frequency to a musical note using the note_name(frequency) function, which calculates the closest MIDI value using the standard formula and converts it to a note name via Librosa's midi_to_note() function. Both the frequency and note are printed and returned. If no valid pitch is found, a warning message is printed. Together, these functions help in identifying and labeling the dominant pitch in an audio signal.

```
def frequency_finder(f0):
    # Get the most common frequency
    if f0 is not None:
        frequency = np.nanmean(f0)
        note = note_name(frequency)
        print(f"Frequency: {frequency:.2f} Hz")
        return frequency , note
    else:
        print("Could not detect a stable pitch in the audio file.")

def note_name(frequency):
    # Convert frequency to the nearest MIDI number
    midi_value = round(69 + 12*np.log2(frequency/440))
    note = librosa.midi_to_note(midi_value)

    print(f"midi: {midi_value}")
    print(f"Note: {note}")

    return note
```

- The play_clip(path) function is used to play an audio file using the Pygame mixer. It loads the audio from the specified file path and plays it back, allowing for simple and real-time audio output during pitch shifting or sample playback.

```
def play_clip(path):
    mixer.music.load(path)
    mixer.music.play()
```

- Loading and frequency block:
    - This is where the waveform is loaded using librosa.load(file_path) and returns y as the amplitude array, vector, and sr that returns the sampling rate of the audio sample. This program mainly focuses on audio input of sample rate 44100Hz.
    - og_len represents the length of the audio in terms of samples
    - display(Audio(y, rate=sr)) is used to give a audio player for the uploaded audio
    - librosa.yin(y, fmin=librosa.note_to_hz('A0'), fmax=librosa.note_to_hz('C8')) returns the vector of the frequencies in the audio sample.
    - Og_midi is the midi calculating formula that returns the midi value of the audio sample

```
file_path = 'piano.mp3'
y, sr = librosa.load(file_path)
og_len = len(y)
display(Audio(y, rate=sr))# sr variation results in pitch transposition

# Extract the fundamental frequency vector using librosa
f0= librosa.yin( y, fmin=librosa.note_to_hz('A0'), fmax=librosa.note_to_hz('C8'))

#frequency average to get the fundamental frequency
frequency , note = frequency_finder(f0)

#frequency to midi converter
og_midi = round(69 + 12*np.log2(frequency/440))

mixer.init(frequency=sr)
play_clip(file_path)
```


```
midi: 66
Note: F♯4
Frequency: 363.60 Hz
```

- Pitch Shift function:
    - The analysis window length is set to define the number of samples processed at a time from the input signal.
    - The hop length between analysis windows is specified to control how much the window shifts on each iteration.

- A scaling factor is computed using the MIDI pitch shift value, determining how much to stretch or compress the signal.
- The new hop length is calculated by scaling the original hop length using the computed factor.
- The new window length is similarly scaled to match the target pitch-shifted size.
- A Hanning window is generated to smooth each audio frame and minimize discontinuities at frame edges.
- The total length of the output signal is estimated based on how many windows will be processed and their scaled sizes.
- Two zero arrays are initialized: one to hold the final pitch-shifted audio and one to accumulate overlapping window weights for normalization.
- A loop is started to process the signal in overlapping windows based on the original hop and window length.
- The actual length of the current window is adjusted if it's near the end of the signal to avoid index overflow.
- Fractional indices are generated to resample the window to the new scaled length.
- The integer part of each index is extracted to locate the base samples for interpolation.
- The fractional part of each index is calculated to use in the interpolation step.
- The start indices are clipped to ensure they stay within valid sample range and do not exceed the signal bounds.
- Linear interpolation is performed between consecutive samples using the fractional offsets to generate the resampled window.
- The resampled window is multiplied by the Hanning window and added to the appropriate position in the output buffer.
- The same window weights are added to the normalization buffer at the corresponding positions.
- Finally, the output signal is normalized by dividing it by the accumulated weights to preserve natural amplitude.

```python
def pitch_shift(waveform, midi_shift):
# single channel
    # Parameters
    anls_win_len = 5000  # Analysis window length
    anls_hop_len = 2000 # Analysis hop length
    scaling_fac = 2 ** (-midi_shift/ 12)
    shifted_hop_len = int(anls_hop_len * scaling_fac)
    shifted_win_len = int(anls_win_len * scaling_fac)

    # Hanning window for smoothing
    win_f = np.hanning(shifted_win_len)

    # Compute new waveform length
    new_y_len = int(np.ceil(og_len / anls_hop_len) * shifted_hop_len) + shifted_win_len
    new_y = np.zeros(new_y_len)
    new_scales = np.zeros(new_y_len)

    # Loop through windows
    for i in range(0, og_len - anls_win_len, anls_hop_len):
        clipped_len = min(anls_win_len, og_len - i)

        # Stretch to synth_win_len using linear interpolation
        idxs = np.linspace( i, i + clipped_len, shifted_win_len )
        start = np.floor(idxs).astype(int)
        frac = idxs - start

        # Ensure indices stay in range
        start = np.clip( start, 0, og_len - 2)

        # Interpolate
        window = y[start] * (1 - frac) + y[start + 1] * frac

        # Overlap-add method
        new_y[i:i + shifted_win_len] += window * win_f
        new_scales[i:i + shifted_win_len] += win_f

    # Normalize to avoid amplitude artifacts
    new_y = new_y / np.where(new_scales == 0, 1, new_scales)

    return new_y
```

- `anls_win_len`: Number of samples in each analysis window from the original signal.
- `anls_hop_len`: Number of samples to shift for the next window in the analysis loop.
- `scaling_fac`: Factor that scales time duration based on the number of semitones to shift.
- `shifted_hop_len`: Hop length for the output signal after applying pitch scaling.

- `shifted_win_len`: Length of each resampled window segment after scaling.
- `win_f`: Hanning window applied to smooth each frame before adding to the output.
- `new_y_len`: Estimated total length of the pitch-shifted output signal.
- `new_y`: Output buffer that accumulates the resampled, windowed signal.
- `new_scales`: Buffer that keeps track of added window weights for normalization.
- `i`: Current index of the start position for the window in the original signal.
- `clipped_len`: Actual number of valid samples in the current window to prevent overflow.
- `idxs`: Array of resampled, fractional indices used to stretch or compress each window.
- `start`: Integer part of the fractional indices used to locate base samples for interpolation.
- `frac`: Decimal part of the fractional indices used for linear interpolation.
- `window`: Final resampled and smoothed window segment ready to be added to the output.

- Pitch_shift_a(y) is a function that gives us a sample output of pitch shift to the A3 note and gives playback by taking y as waveform.

```
def pitch_shift_a(y):
    midi_shift = 45 - og_midi
    print(midi_shift)
    y_a = pitch_shift(y, midi_shift)

    display(Audio(y_a, rate=sr))
    y = np.asarray(y_a, dtype=np.float32)

# Save as WAV with explicit format
    sf.write(f'{folder_name}/a.wav', y, sr, format='WAV')

# Play with pygame
    play_clip(f'{folder_name}/a.wav')
```

- Pitch_shift_octave(y) is the function where each and every note from E to B of higher octave is created and saved in the folder created by the above function in the samples folder, this is now the data used by the gradio for realtime playback. We can shift the octave by multiplying the shift factor with 12 and add it midi_shift

```
def pitch_shift_octave(y):
    for i in range(40,61):
        midi_shift = i - og_midi +12*(1)# add or subtract multiples of 12 to change the octave
        y_shift = pitch_shift(y, midi_shift)
        sf.write(f'{folder_name}/{i}.wav', y_shift, samplerate=sr)
        play_clip(f'{folder_name}/{i}.wav')
        time.sleep(0.3)
        print(midi_shift, end = "  ")
pitch_shift_octave(y)
```

```
-14  -13  -12  -11  -10  -9  -8  -7  -6  -5  -4  -3  -2  -1  0  1  2  3  4  5  6
```

- Play_note(key, folder_name) function is used to play the sample's transposed audio of the mapped key, this is dynamically programmed for reducing redundancy. Keys is the dictionary that is used to map the keyboard keys with the midi values. 'x' char return the total notes played by user.

```
#here we use the keys in keyboard from q: E to ]: B
keys = {
    'q' : 40, 'w' : 41, '3' : 42, 'e' : 43, '4' : 44, 'r' : 45, '5' : 46, 't' : 47, 'y' : 48, '7' :49 , 'u' : 50, '8' : 51 ,
    'i' : 52, 'o' : 53, '0' : 54, 'p' : 55, '-' : 56, '[' : 57, '=' : 58, ']' : 59, "\\" : 60
}


def play_note(key, folder_name):
    if key:
        last_char = key[-1]
        if last_char != 'x':
            midi_note = keys.get(last_char)
            if midi_note:
                note=librosa.midi_to_note(midi_note)
                notes.append(note)
                print("note played: ", note)
                path = f'samples/{folder_name}/{midi_note}.wav'
                play_clip(path)
                return note
            else:
                return "unknown key"
        else:
            return notes
    else:
        return ""
```

- This is the gradio block which creates the interface for the user. Functionality allows the user to use the predefined instruments guitar and piano and allows the user to create their own instrument by giving the upload audio block. The interface allows the user to see which note he/she is playing.

```
with gr.Blocks() as demo:
    gr.Markdown("## 🎶 Audio Sampler: Pitch Transposer")
    folder_state = gr.State("guitar")  # default

    gr.Image(value="notes.png", label="Keys Reference", height = 260, width = 730)

    with gr.Row():
        gr.Button("🎸 Guitar").click(fn=guitar, outputs=folder_state)
        gr.Button("🎹 Piano").click(fn=piano, outputs=folder_state)
        gr.Button("🕶️ your Sample").click(fn=your_sample, outputs=folder_state)

    with gr.Row():
        key_input = gr.Textbox(label="Press a Key ")
        output_text = gr.Textbox(label="Note :")
    with gr.Row():
        gr.Markdown("## Upload Audio to Save in Your Project Folder")
        audio_input = gr.File(label="Upload .wav or .mp3", file_types=[".wav", ".mp3"])
        Audio_status = gr.Textbox(label="Save Status")

        audio_input.change(fn=save_uploaded_audio, inputs=audio_input, outputs=Audio_status)

    key_input.change(fn=play_note, inputs=[key_input, folder_state], outputs=output_text)

demo.launch()
print(notes)
```

- These functions are used for dynamic access of the folders with samples

```
notes=[]
def guitar():
    return "guitar"

def piano():
    return "Piano"

def your_sample():
    return "your_sample"
```

- save_uploaded_audio(audio_upload) function handles the uploaded audio file from the Gradio interface. It first checks if a file is provided, and if so, retrieves the file path and saves it locally as `your_sample.wav`. The audio is loaded using Librosa to standardize its format before saving. After saving, it calls `process_audio()` to extract and process pitch-related information and returns a confirmation message along with the detected musical note.
- process_audio(audio_upload) function processes the saved audio (`your_sample.wav`) to extract its fundamental frequency using the YIN algorithm. The detected frequency is converted into a MIDI note. Based on that, the function generates a series of pitch-shifted versions of the input audio across a MIDI range (40 to 60), shifted one octave down. These modified audio samples are saved in a new directory and played back one by one using the mixer. It returns the original note detected in the input audio.

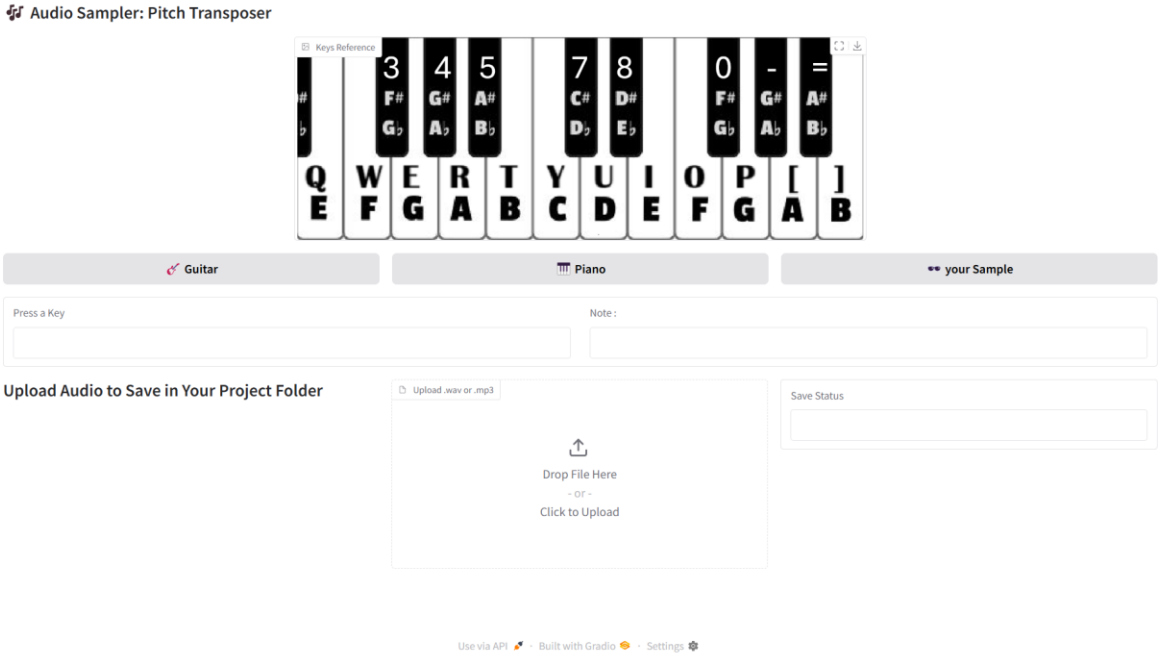
```python
def save_uploaded_audio(audio_upload):
    if audio_upload is None:
        return "No file uploaded."

    try:
        # Get the file path from the Gradio upload object
        file_path = audio_upload.name
        # Define the new path where we want to save it
        new_path = "your_sample.wav"
        y, sr = librosa.load(file_path, sr=None)
        sf.write(new_path, y, sr)
        note = process_audio(audio_upload)
        return f"Saved as {new_path} and \n note : {note}"
    except Exception as e:
        return f"Error processing audio: {e}"

def process_audio(audio_upload):
    file_path ="your_sample.wav"
    y, sr = librosa.load(file_path)
    og_len = len(y)
    f0 = librosa.yin( y, fmin=librosa.note_to_hz('A0'), fmax=librosa.note_to_hz('C8'))
    frequency, note = frequency_finder(f0)
    og_midi = round(69 + 12*np.log2(frequency/440))
    folder_name=f'samples/{file_path.split(".")[0]}'
    os.makedirs(folder_name, exist_ok = True)
    #pitch_shift_octave(y)
    for i in range(40,61):
        midi_shift = i - og_midi +12*(-1)# add or subtract multiples of 12 to change the octave
        y_shift = pitch_shift(y, midi_shift)
        sf.write(f'{folder_name}/{i}.wav', y_shift, samplerate=sr)
        play_clip(f'{folder_name}/{i}.wav')
        #ime.sleep(0.3)
        print(midi_shift, end = "  ")
    return note
```

CHAPTER 4

OUTPUT INTERFACE

4.1 INTERFACE

This is the final Front end that is visible to the user.



- **Keys Reference Image**: Provides a visual guide mapping keyboard keys to piano notes for reference during playback.
- **Instrument Selection Buttons**: Lets users choose between Guitar, Piano, or a custom uploaded sample to assign the sound folder.
- **Key Input Field**: Allows users to type a key (e.g., 'Q', 'W', 'E') to trigger a specific note from the selected instrument.
- **Note Display Field**: Shows the name of the note played when a key is pressed.
- **Audio Upload Section**: Enables users to upload a `.wav` or `.mp3` file, which gets processed and pitch-shifted.
- **Save Status Field**: Displays feedback after uploading and processing audio, including confirmation and detected note.

○

REFERENCES

1. De Cheveigné, A., & Kawahara, H. (2002). YIN, a fundamental frequency estimator for speech and music. The Journal of the Acoustical Society of America, 111(4), 1917-1930

2. 'Making a Pitch Shifter' by JentGent
   Link: https://www.youtube.com/watch?v=PjKlMXhxtTM

3. Pygame Documentation
   Link : https://www.pygame.org/docs/

4. Gradio Documentation
   Link: https://gradio.app/docs/

5. Librosa Library Documentation,
   "Python package for music and audio analysis"
   Link: https://librosa.org/doc/latest/index.html

Git link: https://github.com/gundlurujeevan2107

Mail Id: gundlurujeevan@gmail.com