Keyboard_Pi_Conductor/PythonPiano/key_midi.py

```
import pygame
import PythonPiano.piano_lists as pl
from pygame import mixer
from music21 import note, stream, tempo
from threading import Thread
import time


class AudioRecorder:
    def __init__(self, save_path, bpm=90):
        print('initializing new audio recorder')
        self.save_path = save_path
        self.bpm = bpm
        self.metronome_mark = tempo.MetronomeMark(number=self.bpm)
        self.left_oct = 4
        self.right_oct = 5
        self.white_notes = pl.white_notes
        self.black_notes = pl.black_notes
        self.black_labels = pl.black_labels
        # windowSurfaceObj = pygame.display.set_mode((64,48),1,16)

        self.initialize(bpm)

    def initialize(self, bpm=None):
        if bpm is not None:
            self.bpm = bpm
            self.metronome_mark = tempo.MetronomeMark(number=self.bpm)
        self.score = stream.Stream()
        self.score.append(self.metronome_mark)
        pygame.mixer.init()
        pygame.mixer.set_num_channels(50)

        self.white_sounds = []
        self.black_sounds = []
        for i in range(len(self.white_notes)):

            self.white_sounds.append(mixer.Sound(
f'/home/pi/Documents/ECE5725_final_proj/PythonPiano/assets/notes/{self.white_notes[i]}.wav'))

        for i in range(len(self.black_notes)):
            self.black_sounds.append(mixer.Sound(
f'/home/pi/Documents/ECE5725_final_proj/PythonPiano/assets/notes/{self.black_notes[i]}.wav'))

        self.note_with_time = []
        self.left_dict = {'Z': f'C{self.left_oct}', 'S': f'C#{self.left_oct}', 'X': f'D{self.left_oct}', 'D': f'D#{self.left_oct}',
                          'C': f'E{self.left_oct}', 'V': f'F{self.left_oct}', 'G': f'F#{self.left_oct}', 'B': f'G{self.left_oct}',
                          'H': f'G#{self.left_oct}', 'N': f'A{self.left_oct}', 'J': f'A#{self.left_oct}', 'M': f'B{self.left_oct}'}

        self.right_dict = {
            'R': f'C{self.right_oct}',
            '5': f'C#{self.right_oct}',
            'T': f'D{self.right_oct}',
            '6': f'D#{self.right_oct}',
            'Y': f'E{self.right_oct}',
            'U': f'F{self.right_oct}',
            '8': f'F#{self.right_oct}',
            'I': f'G{self.right_oct}',
            '9': f'G#{self.right_oct}',
            'O': f'A{self.right_oct}',
            '0': f'A#{self.right_oct}',
            'P': f'B{self.right_oct}'
        }

    def start_recording(self):
        self.recording = True
        self.record()

    def process_event(self, event):
        if event.type == pygame.QUIT:
            self.clean()
            return
        if event.type == pygame.TEXTINPUT:
            if event.text.upper() in self.left_dict:
                keynote = self.left_dict[event.text.upper()]
                self.note_with_time.append((keynote, time.time()))

                if self.left_dict[event.text.upper()][1] == '#':
                    index = self.black_labels.index(keynote)

                    self.black_sounds[index].play(0, 1000)
                else:
                    index = self.white_notes.index(keynote)
                    self.white_sounds[index].play(0, 1000)

                # self.score.append(note.Note(keynote, type='quarter'))
            if event.text.upper() in self.right_dict:
                keynote = self.right_dict[event.text.upper()]
                self.note_with_time.append((keynote, time.time()))

                if self.right_dict[event.text.upper()][1] == '#':
                    index = self.black_labels.index(keynote)
                    self.black_sounds[index].play(0, 1000)
                else:
                    index = self.white_notes.index(keynote)
                    self.white_sounds[index].play(0, 1000)

                # self.score.append(note.Note(keynote, type='quarter'))
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RETURN or event.key == pygame.K_KP_ENTER:
                print('enter:stop')
                self.save_recording()
                return

    def record(self):
        i = 0
        while self.recording:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.clean()
                    return
                if event.type == pygame.TEXTINPUT:
                    if event.text.upper() in self.left_dict:
                        keynote = self.left_dict[event.text.upper()]

                        if self.left_dict[event.text.upper()][1] == '#':
                            index = self.black_labels.index(keynote)

                            self.black_sounds[index].play(0, 1000)
                        else:
                            index = self.white_notes.index(keynote)
                            self.white_sounds[index].play(0, 1000)

                        self.note_with_time.append((keynote, time.time()))
                        self.score.append(note.Note(keynote, type='quarter'))

                    if event.text.upper() in self.right_dict:
                        keynote = self.right_dict[event.text.upper()]

                        if self.right_dict[event.text.upper()][1] == '#':
                            index = self.black_labels.index(keynote)
                            self.black_sounds[index].play(0, 1000)
                        else:
                            index = self.white_notes.index(keynote)
                            self.white_sounds[index].play(0, 1000)

                        self.note_with_time.append((keynote, time.time()))
                        # self.score.append(note.Note(keynote, type='quarter'))
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_RETURN or event.key == pygame.K_KP_ENTER:
                        print('enter:stop')
                        self.save_recording()
                        return

    def save_recording(self):
        eighth_timestep = 1 / (self.bpm * 2 / 60)
        for i in range(len(self.note_with_time) - 1):
            curr_note, curr_time = self.note_with_time[i]
            _, next_time = self.note_with_time[i+1]

            n_time_steps = round((next_time - curr_time) / eighth_timestep)
            self.score.append(
                note.Note(curr_note, quarterLength=n_time_steps/2))

        self.score.append(
            note.Note(self.note_with_time[-1][0], quarterLength=2))

        print('stop record')
        self.recording = False
        self.score.write('midi', fp=self.save_path)
        print(f"Recording stopped. MIDI file saved as {self.save_path}")
        pygame.mixer.quit()


if __name__ == "__main__":
    recorder = AudioRecorder(save_path='test_recorder.mid')
    print('Hit Enter to Start Recording')
    input()
    recorder.start_recording()
    print('Hit Enter to stop recording')
    input()
    recorder.stop_recording()

```

Keyboard_Pi_Conductor/PythonPiano/piano_lists.py

```python
left_hand = ['Z', 'S', 'X', 'D', 'C', 'V', 'G', 'B', 'H', 'N', 'J', 'M']
right_hand = ['R', '5', 'T', '6', 'Y', 'U', '8', 'I', '9', 'O', '0', 'P']

piano_notes = ['A0', 'A0#', 'B0', 'C1', 'C1#', 'D1', 'D1#', 'E1', 'F1', 'F1#', 'G1', 'G1#',
               'A1', 'A1#', 'B1', 'C2', 'C2#', 'D2', 'D2#', 'E2', 'F2', 'F2#', 'G2', 'G2#',
               'A2', 'A2#', 'B2', 'C3', 'C3#', 'D3', 'D3#', 'E3', 'F3', 'F3#', 'G3', 'G3#',
               'A3', 'A3#', 'B3', 'C4', 'C4#', 'D4', 'D4#', 'E4', 'F4', 'F4#', 'G4', 'G4#',
               'A4', 'A4#', 'B4', 'C5', 'C5#', 'D5', 'D5#', 'E5', 'F5', 'F5#', 'G5', 'G5#',
               'A5', 'A5#', 'B5', 'C6', 'C6#', 'D6', 'D6#', 'E6', 'F6', 'F6#', 'G6', 'G6#',
               'A6', 'A6#', 'B6', 'C7', 'C7#', 'D7', 'D7#', 'E7', 'F7', 'F7#', 'G7', 'G7#',
               'A7', 'A7#', 'B7', 'C8']

white_notes = ['A0', 'B0', 'C1', 'D1', 'E1', 'F1', 'G1',
               'A1', 'B1', 'C2', 'D2', 'E2', 'F2', 'G2',
               'A2', 'B2', 'C3', 'D3', 'E3', 'F3', 'G3',
               'A3', 'B3', 'C4', 'D4', 'E4', 'F4', 'G4',
               'A4', 'B4', 'C5', 'D5', 'E5', 'F5', 'G5',
               'A5', 'B5', 'C6', 'D6', 'E6', 'F6', 'G6',
               'A6', 'B6', 'C7', 'D7', 'E7', 'F7', 'G7',
               'A7', 'B7', 'C8']

black_notes = ['Bb0', 'Db1', 'Eb1', 'Gb1', 'Ab1',
               'Bb1', 'Db2', 'Eb2', 'Gb2', 'Ab2',
               'Bb2', 'Db3', 'Eb3', 'Gb3', 'Ab3',
               'Bb3', 'Db4', 'Eb4', 'Gb4', 'Ab4',
               'Bb4', 'Db5', 'Eb5', 'Gb5', 'Ab5',
               'Bb5', 'Db6', 'Eb6', 'Gb6', 'Ab6',
               'Bb6', 'Db7', 'Eb7', 'Gb7', 'Ab7',
               'Bb7']

black_labels = ['A#0', 'C#1', 'D#1', 'F#1', 'G#1',
                'A#1', 'C#2', 'D#2', 'F#2', 'G#2',
                'A#2', 'C#3', 'D#3', 'F#3', 'G#3',
                'A#3', 'C#4', 'D#4', 'F#4', 'G#4',
                'A#4', 'C#5', 'D#5', 'F#5', 'G#5',
                'A#5', 'C#6', 'D#6', 'F#6', 'G#6',
                'A#6', 'C#7', 'D#7', 'F#7', 'G#7',
                'A#7']
```

Keyboard_Pi_Conductor/Audio_Midi/init**.py**

```

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/integrate_main.py

```
import os
import sys
import logging

from Audio_Midi.audio_to_midi import converter, progress_bar



def _convert_beat_to_time(bpm, beat):
    try:
        parts = beat.split("/")
        if len(parts) > 2:
            raise Exception()

        beat = [int(part) for part in parts]
        fraction = beat[0] / beat[1]
        bps = bpm / 60
        ms_per_beat = bps * 1000
        return fraction * ms_per_beat
    except Exception:
        raise RuntimeError("Invalid beat format: {}".format(beat))


def audio_to_midi_conv(infile, outfile, beat='1/4', single_note=True):
    try:
        logging.basicConfig(level=logging.DEBUG, format="%(message)s")

        time_window = _convert_beat_to_time(bpm, beat)

        global bpm
        process = converter.Converter(
            infile=infile,
            outfile=outfile,
            time_window=time_window,
            activation_level=0.0,
            condense=False,
            condense_max=False,
            max_note_length=0,
            note_count=1 if single_note else 0,
            transpose=0,
            pitch_set=[],
            pitch_range=None,
            progress=progress_bar.ProgressBar(),
            bpm=bpm,
        )
        process.convert()
    except KeyboardInterrupt:
        sys.exit(1)
    except Exception as e:
        logging.exception(e)
        sys.exit(1)

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/midi_writer.py

```
from collections import defaultdict

import python3_midi as midi


class NoteState:
    __slots__ = ["is_active", "event_pos", "count"]

    def __init__(self, is_active=False, event_pos=None, count=0):
        self.is_active = is_active
        self.event_pos = event_pos
        self.count = count


class MidiWriter:
    def __init__(
        self,
        outfile,
        channels,
        time_window,
        bpm=60,
        condense=False,
        condense_max=False,
        max_note_length=0,
    ):
        self.outfile = outfile
        self.condense = condense
```

```python
        self.condense_max = condense_max
        self.max_note_length = max_note_length
        self.channels = channels
        self.time_window = time_window
        self.bpm = bpm
        self.note_state = [defaultdict(lambda: NoteState()) for _ in range(channels)]

        bps = self.bpm / 60
        self.ms_per_beat = int((1.0 / bps) * 1000)
        self.tick_increment = int(time_window)
        self.skip_count = 1
        self._need_increment = False

    def __enter__(self):
        self.stream = midi.FileStream(self.outfile)
        self.stream.start_pattern(
            format=1,
            tick_relative=False,
            resolution=self.ms_per_beat,
            tracks=[],
        )
        self.stream.start_track(
            events=[
                midi.TimeSignatureEvent(
                    tick=0,
                    numerator=1,
                    denominator=4,
                    metronome=int(self.ms_per_beat / self.time_window),
                    thirtyseconds=32,
                )
            ],
            tick_relative=False,
        )
        return self

    def __exit__(self, type, value, traceback):
        self._terminate_notes()
        self.stream.add_event(midi.EndOfTrackEvent(tick=1))
        self.stream.end_track()
        self.stream.end_pattern()
        self.stream.close()

    def _skip(self):
        self.skip_count += 1

    def _reset_skip(self):
        self.skip_count = 1

    @property
    def tick(self):
        ret = 0
        if self._need_increment:
            self._need_increment = False
            ret = self.tick_increment * self.skip_count
            self._reset_skip()
        return ret

    def _note_on(self, channel, pitch, velocity):
        pos = self.stream.add_event(
            midi.NoteOnEvent(
                tick=self.tick, channel=channel, pitch=pitch, velocity=60
            )
        )
        self.note_state[channel][pitch] = NoteState(
            True,
            pos,
            1,
        )

    def _note_off(self, channel, pitch):
        self.note_state[channel][pitch] = NoteState()
        self.stream.add_event(
            midi.NoteOffEvent(
                tick=self.tick,
                channel=channel,
                pitch=pitch,
            )
        )

    def add_notes(self, notes):
        """
        notes is a list of midi notes to add at the current
            time step.

        Adds each note in the list to the current time step
            with the volume, track and channel specified.
        """
        self._need_increment = True
        if not self.condense:
            self._terminate_notes()

        for channel, notes in enumerate(notes):
            new_notes = set()
            stale_notes = []
            for note in notes:
                note_state = self.note_state[channel][note.pitch]
                new_notes.add(note.pitch)
                if (not self.condense) or (self.condense and not note_state.is_active):
                    self._note_on(channel, note.pitch, note.velocity)
                elif self.condense and note_state.is_active:
                    event = self.stream.get_event(
                        midi.NoteOnEvent, note_state.event_pos
                    )
                    old_velocity = event.data[1]
                    if self.condense_max:
                        new_velocity = max(note.velocity, old_velocity)
                    else:
                        count = note_state.count
                        note_state.count += 1
                        new_velocity = ((old_velocity * count) + note.velocity) // (
                            note_state.count
                        )
                    if old_velocity != event.data[1]:
                        event.data[1] = new_velocity
                        self.stream.set_event(event, note_state.event_pos)

            if self.condense:
                active_notes = [
                    note
                    for note in self.note_state[channel]
                    if self.note_state[channel][note].is_active
                ]
                for note in active_notes:
                    if (
                        note not in new_notes
                        or self.note_state[channel][note].count > self.max_note_length
                    ):
                        stale_notes.append(note)

                for note in stale_notes:
                    self._note_off(channel, note)

        if self._need_increment:
            self._skip()

```

```
    def _terminate_notes(self):
        for channel in range(self.channels):
            for note, note_state in self.note_state[channel].items():
                if note_state.is_active:
                    self._note_off(channel, note)

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/converter.py

```
import logging

from collections import namedtuple
from functools import lru_cache
from operator import attrgetter

import numpy
import soundfile

# import midi_writer, notes
from Audio_Midi.audio_to_midi import midi_writer, notes


class Note:
    __slots__ = ["pitch", "velocity", "count"]

    def __init__(self, pitch, velocity, count=0):
        self.pitch = pitch
        self.velocity = velocity
        self.count = count


class Converter:
    def __init__(
        self,
        infile=None,
        outfile=None,
        time_window=None,
        activation_level=None,
        condense=None,
        condense_max=False,
        max_note_length=0,
        transpose=0,
        pitch_set=None,
        pitch_range=None,
        note_count=None,
        progress=None,
        bpm=60,
    ):

        if infile:
            self.info = soundfile.info(infile)
        else:
            raise RuntimeError("No input provided.")

        self.infile = infile
        self.outfile = outfile
        self.time_window = time_window
        self.condense = condense
        self.condense_max = condense_max
        self.max_note_length = max_note_length
        self.transpose = transpose
        self.pitch_set = pitch_set
        self.pitch_range = pitch_range or [0, 127]
        self.note_count = note_count
        self.progress = progress
        self.bpm = bpm

        self.activation_level = int(127 * activation_level) or 1
        self.block_size = self._time_window_to_block_size(
            self.time_window, self.info.samplerate
        )

        steps = self.info.frames // self.block_size
        self.total = steps
        self.current = 0

        self._determine_ranges()

    def _determine_ranges(self):
        self.notes = notes.generate()
        self.max_freq = min(self.notes[127][-1], self.info.samplerate / 2)
        self.min_freq = max(self.notes[0][-1], 1000 / self.time_window)
        self.bins = self.block_size // 2
        self.frequencies = numpy.fft.fftfreq(self.bins, 1 / self.info.samplerate)[
            : self.bins // 2
        ]

        for i, f in enumerate(self.frequencies):
            if f >= self.min_freq:
                self.min_bin = i
                break
        else:
            self.min_bin = 0
        for i, f in enumerate(self.frequencies):
            if f >= self.max_freq:
                self.max_bin = i
                break
        else:
            self.max_bin = len(self.frequencies)

    def _increment_progress(self):
        if self.progress:
            self.current += 1
            self.progress.update(self.current, self.total)

    @staticmethod
    def _time_window_to_block_size(time_window, rate):
        """
        time_window is the time in ms over which to compute fft's.
        rate is the audio sampling rate in samples/sec.

        Transforms the time window into an index step size and
            returns the result.
        """

        # rate/1000(samples/ms) * time_window(ms) = block_size(samples)
        rate_per_ms = rate / 1000
        block_size = rate_per_ms * time_window

        return int(block_size)

    def _freqs_to_midi(self, freqs):
        """
        freq_list is a list of frequencies with normalized amplitudes.

        Takes a list of notes and transforms the amplitude to a
            midi volume as well as adding track and channel info.
        """

        notes = [None for _ in range(128)]
        for pitch, velocity in freqs:
            if not (self.pitch_range[0] <= pitch <= self.pitch_range[1]):
                continue
            velocity = min(int(127 * (velocity / self.bins)), 127)

            if velocity > self.activation_level:
                if not notes[pitch]:
                    notes[pitch] = Note(pitch, 60)
                else:
                    notes[pitch].velocity = int(
                        ((notes[pitch].velocity * notes[pitch].count) + velocity)
                        / (notes[pitch].count + 1)
                    )
                    notes[pitch].count += 1

        notes = [note for note in notes if note]

        if self.note_count > 0:
            max_count = min(len(notes), self.note_count)
            notes = sorted(notes, key=attrgetter("velocity"))[::-1][:max_count]

        return notes

    def _snap_to_key(self, pitch):
        if self.pitch_set:
            mod = pitch % 12
            pitch = (12 * (pitch // 12)) + min(
                self.pitch_set, key=lambda x: abs(x - mod)
            )
        return pitch

    @lru_cache(None)
    def _freq_to_pitch(self, freq):
        for pitch, freq_range in self.notes.items():
            # Find the freq's equivalence class, adding the amplitudes.
            if freq_range[0] <= freq <= freq_range[2]:
                return self._snap_to_key(pitch) + self.transpose
        raise RuntimeError("Unmappable frequency: {}".format(freq[0]))

    def _reduce_freqs(self, freqs):
        """
        freqs is a list of amplitudes produced by _fft_to_frequencies().

        Reduces the list of frequencies to a list of notes and their
            respective volumes by determining what note each frequency
            is closest to. It then reduces the list of amplitudes for each
            note to a single amplitude by summing them together.
        """

        reduced_freqs = []
        for freq in freqs:
            reduced_freqs.append((self._freq_to_pitch(freq[0]), freq[1]))

```

```python
        return reduced_freqs

    def _samples_to_freqs(self, samples):
        amplitudes = numpy.fft.fft(samples)
        freqs = []

        for index in range(self.min_bin, self.max_bin):
            # frequency, amplitude
            freqs.append(
                [
                    self.frequencies[index],
                    numpy.sqrt(
                        numpy.float_power(amplitudes[index].real, 2)
                        + numpy.float_power(amplitudes[index].imag, 2)
                    ),
                ]
            )

        # Transform the frequency info into midi compatible data.
        return self._reduce_freqs(freqs)

    def _block_to_notes(self, block):
        channels = [[] for _ in range(self.info.channels)]
        notes = [None for _ in range(self.info.channels)]

        for sample in block:
            for channel in range(self.info.channels):
                channels[channel].append(sample[channel])

        for channel, samples in enumerate(channels):
            freqs = self._samples_to_freqs(samples)
            notes[channel] = self._freqs_to_midi(freqs)

        return notes

    def convert(self):
        """
        Performs the fft for each time step and transforms the result
            into midi compatible data. This data is then passed to a
            midi file writer.
        """

        logging.info(str(self.info))
        logging.info("window: {} ms".format(self.time_window))
        logging.info(
            "frequencies: min = {} Hz, max = {} Hz".format(self.min_freq, self.max_freq)
        )

        with midi_writer.MidiWriter(
            outfile=self.outfile,
            channels=self.info.channels,
            time_window=self.time_window,
            bpm=self.bpm,
            condense=self.condense,
            condense_max=self.condense_max,
            max_note_length=self.max_note_length,
        ) as writer:
            for block in soundfile.blocks(
                self.infile,
                blocksize=self.block_size,
                always_2d=True,
            ):
                if len(block) != self.block_size:
                    filler = numpy.array(
                        [
                            numpy.array([0.0 for _ in range(self.info.channels)])
                            for _ in range(self.block_size - len(block))
                        ]
                    )
                    block = numpy.append(block, filler, axis=0)
                notes = self._block_to_notes(block)
                writer.add_notes(notes)
                self._increment_progress()

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/init.py

```

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/notes.py

```
import numpy

def generate():
    """
    Generates a dict of midi note codes with their corresponding
        frequency ranges.
    """

    # C0
    base = [7.946362749, 8.1757989155, 8.4188780665]

    # 12th root of 2
    multiplier = numpy.float_power(2.0, 1.0 / 12)

    notes = {0: base}
    for i in range(1, 128):
        mid = multiplier * notes[i - 1][1]
        low = (mid + notes[i - 1][1]) / 2.0
        high = (mid + (multiplier * mid)) / 2.0
        notes.update({i: [low, mid, high]})

    return notes

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/progress_bar.py

```
import time
import threading
import progressbar


class ProgressBar:
    def __init__(self, current=0, total=0):
        self.current = current
        self.total = total
        self.bar = progressbar.ProgressBar(max_value=self.total)

    def update(self, current=0, total=0):
        current = min(current, total)
        self.bar.max_value = total
        self.bar.update(current)

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/main.py

```
#!/usr/bin/env python3

import argparse
import os
import sys
import logging

import converter, progress_bar
```

```


def _convert_beat_to_time(bpm, beat):
    try:
        parts = beat.split("/")
        if len(parts) > 2:
            raise Exception()

        beat = [int(part) for part in parts]
        fraction = beat[0] / beat[1]
        bps = bpm / 60
        ms_per_beat = bps * 1000
        return fraction * ms_per_beat
    except Exception:
        raise RuntimeError("Invalid beat format: {}".format(beat))


def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("infile", help="The sound file to process.")
    parser.add_argument(
        "--output", "-o", help="The MIDI file to output. Default: <infile>.mid"
    )
    parser.add_argument(
        "--time-window",
        "-t",
        default=5.0,
        type=float,
        help="The time span over which to compute the individual FFTs in milliseconds.",
    )
    parser.add_argument(
        "--activation-level",
        "-a",
        default=0.0,
        type=float,
        help="The amplitude threshold for notes to be added to the MIDI file. Must be between 0 and 1.",
    )
    parser.add_argument(
        "--condense",
        "-c",
        action="store_true",
        help="Combine contiguous notes at their average amplitude.",
    )
    parser.add_argument(
        "--condense-max",
        "-m",
        action="store_true",
        help="Write the maximum velocity for a condensed note segment rather than the rolling average.",
    )
    parser.add_argument(
        "--max-note-length",
        "-M",
        type=int,
        default=0,
        help="The max condensed note length in time window units.",
    )
    parser.add_argument(
        "--single-note",
        "-s",
        action="store_true",
        help="Only add the loudest note to the MIDI file for a given time window.",
    )
    parser.add_argument(
        "--note-count",
        "-C",
        type=int,
        default=0,
        help="Only add the loudest n notes to the MIDI file for a given time window.",
    )
    parser.add_argument(
        "--bpm", "-b", type=int, help="Beats per minute. Defaults: 60", default=60
    )
    parser.add_argument(
        "--beat",
        "-B",
        help="Time window in terms of beats (1/4, 1/8, etc.). Supercedes the time window parameter.",
    )
    parser.add_argument(
        "--transpose",
        "-T",
        type=int,
        default=0,
        help="Transpose the MIDI pitches by a constant offset.",
    )
    parser.add_argument(
        "--pitch-set",
        "-p",
        type=int,
        nargs="+",
        default=[],
        help="Map to a pitch set. Values must be in the range: [0, 11]. Ex: -p 0 2 4 5 7 9 11",
    )
    parser.add_argument(
        "--pitch-range",
        "-P",
        nargs=2,
        type=int,
        help="The minimum and maximum allowed MIDI notes. These may be superseded by the calculated FFT range.",
    )
    parser.add_argument(
        "--no-progress", "-n", action="store_true", help="Don't print the progress bar."
    )
    args = parser.parse_args()

    args.output = (
        "{}.mid".format(os.path.basename(args.infile))
        if not args.output
        else args.output
    )

    if args.single_note:
        args.note_count = 1

    if args.pitch_set:
        for key in args.pitch_set:
            if key not in range(12):
                raise RuntimeError("Key values must be in the range: [0, 12)")

    if args.beat:
        args.time_window = _convert_beat_to_time(args.bpm, args.beat)
        print(args.time_window)

    if args.pitch_range:
        if args.pitch_range[0] > args.pitch_range[1]:
            raise RuntimeError("Invalid pitch range: {}".format(args.pitch_range))

    if args.condense_max:
        args.condense = True

    return args
```

```python


def main():
    try:
        logging.basicConfig(level=logging.DEBUG, format="%(message)s")

        args = parse_args()

        process = converter.Converter(
            infile=args.infile,
            outfile=args.output,
            time_window=args.time_window,
            activation_level=args.activation_level,
            condense=args.condense,
            condense_max=args.condense_max,
            max_note_length=args.max_note_length,
            note_count=args.note_count,
            transpose=args.transpose,
            pitch_set=args.pitch_set,
            pitch_range=args.pitch_range,
            progress=None if args.no_progress else progress_bar.ProgressBar(),
            bpm=args.bpm,
        )
        process.convert()
    except KeyboardInterrupt:
        sys.exit(1)
    except Exception as e:
        logging.exception(e)
        sys.exit(1)


if __name__ == "__main__":
    main()

```

Keyboard_Pi_Conductor/Audio_Midi/python_midi.py

```python
# from midiutil.MidiFile import
import pyaudio
import wave

from Audio_Midi.audio_to_midi.converter import Converter
from threading import Thread

class AudioRecorder:

    def __init__(self, save_path):
        self.recording = False

        self.audio = pyaudio.PyAudio()


        self.usb_idx = 0

        # Outputs index of every audio-capable device on the Pi
        for ii in range(self.audio.get_device_count()):
            if "USB PnP Sound" in self.audio.get_device_info_by_index(ii).get('name'):
                self.usb_idx = ii
            print(f"idx: {ii}, name: {self.audio.get_device_info_by_index(ii).get('name')}")

        # Set index of usb microphone

        # Set parameters for audio recording
        self.form_1 = pyaudio.paInt16  # 16-bit resolution
        self.chans = 1  # 1 channel
        self.samp_rate = 44100  # 44.1kHz sampling rate
        self.chunk = 4096  # 2^12 samples for buffer
        self.save_path = save_path  # name of .wav file
        self.frames = []


    def start_recording(self):
        self.stream = self.audio.open(
            format=self.form_1,
            rate=self.samp_rate,
            channels=self.chans,
            input_device_index=self.usb_idx,
            input=True,
            frames_per_buffer=self.chunk
        )
        print("recording")

        self.recording = True
        self.record_thread = Thread(target=self.record)
        self.record_thread.start()

    def record(self):
        while self.recording:
            data = self.stream.read(self.chunk)
            self.frames.append(data)

    def stop_recording(self):
        self.recording = False

        self.record_thread.join()
        print('Recording Stopped, closing streams')

        self.stream.stop_stream()
        self.stream.close()
        self.audio.terminate()

        print("finished recording")

        # save the audio frames as .wav file
        wavefile = wave.open(self.save_path, 'wb')
        wavefile.setnchannels(self.chans)
        wavefile.setsampwidth(self.audio.get_sample_size(self.form_1))
        wavefile.setframerate(self.samp_rate)
        wavefile.writeframes(b''.join(self.frames))
        wavefile.close()


def start_recording(audio, form_1, chans, samp_rate, chunk, usb_idx):
    # create pyaudio stream
    stream = audio.open(format=form_1, rate=samp_rate, channels=chans,
                        input_device_index=usb_idx, input=True,
                        frames_per_buffer=chunk)
    print("recording")
    return stream


def record_loop(stream, samp_rate, chunk, record_secs):
    # loop through stream and append audio chunks to frame array
    frames = []
    for ii in range(int((samp_rate/chunk)*record_secs)):
        data = stream.read(chunk)
        frames.append(data)
    return frames


def stop_recording(audio, stream, filename, chans, samp_rate, frames, form_1):
    # stop the stream, close it, and terminate the pyaudio instantiation
    stream.stop_stream()
    stream.close()
    audio.terminate()

    print("finished recording")

    # save the audio frames as .wav file
    wavefile = wave.open(filename, 'wb')
    wavefile.setnchannels(chans)
    wavefile.setsampwidth(audio.get_sample_size(form_1))
    wavefile.setframerate(samp_rate)
    wavefile.writeframes(b''.join(frames))
    wavefile.close()


def audio_to_midi_conv(input_filename, output_filename, time_window=5.0, activ_level=0.0, note_count=0, bpm=60):
    process = Converter(
        infile=input_filename,
        outfile=output_filename,
        time_window=time_window,
        activation_level=activ_level,
        condense=None,
        condense_max=False,
        max_note_length=0,
        note_count=note_count,
        bpm=bpm,
    )
    process.convert()


def record_audio(audio_length, save_path):
    audio = pyaudio.PyAudio()

    # Outputs index of every audio-capable device on the Pi
    for ii in range(audio.get_device_count()):
        print(f"idx: {ii}, name: {audio.get_device_info_by_index(ii).get('name')}")

    # Set index of usb microphone
    usb_idx = 1

    # Set parameters for audio recording
    form_1 = pyaudio.paInt16  # 16-bit resolution
    chans = 1  # 1 channel
    samp_rate = 44100  # 44.1kHz sampling rate
    chunk = 4096  # 2^12 samples for buffer
    wav_output_filename = save_path  # name of .wav file
    frames = []

    stream = start_recording(audio, form_1, chans, samp_rate, chunk, usb_idx)
    frames = record_loop(stream, samp_rate, chunk, audio_length)
    stop_recording(audio, stream, wav_output_filename,
                   chans, samp_rate, frames, form_1)


if __name__ == "__main__":
    # record_audio(10, 'test_recorder.wav')
    recorder = AudioRecorder(save_path='test_recorder.wav')

    print('Hit Enter to Start Recording')
    input()
    recorder.start_recording()

    print('Hit Enter to Stop Recording')
    input()
    recorder.stop_recording()

```

Keyboard_Pi_Conductor/metadata.py

```

RAW_AUDIO_PATH = 'out/recorded_audio.wav'
RECORDED_VIDEO_PATH = 'out/conducting_video.mp4'
RAW_MIDI_PATH = 'out/raw_midi.mid'
MODIFIED_MIDI_PATH = 'out/modified_midi.mid'
MODIFIED_AUDIO_PATH = 'out/modified_audio.wav'
MIDI_BEAT = '1/4'
MIDI_BPM = 90



```

Keyboard_Pi_Conductor/integrate.py

```
import pygame
from pygame.time import Clock
import threading
from PythonPiano.key_midi import AudioRecorder
from Gesture_IR.record_video import VideoRecorder
from Gesture_IR.gesture_ir import gesture_ir_main
from IR_Midi.modify_input import modify_volume, convert_midi_to_audio, play_audio_file

from GUI import init_system, GUI
from metadata import *
```

```
import time
from metronome import Metronome


current_frame = None

class DataFrame:
    def __init__(self):
        self.audio_recorder = AudioRecorder(save_path=RAW_MIDI_PATH)
        self.bpm = MIDI_BPM
        self.video_recorder = VideoRecorder(RECORDED_VIDEO_PATH, bpm=self.bpm)

df = DataFrame()

def start_audio_playback_recording_and_camera(frame, audio_file, video_path):
    df.video_recorder.initialize(df.bpm*2)
    df.video_recorder.spawn_camera()
    time.sleep(2.0)

    playback_thread = threading.Thread(target=play_audio_file, args=(audio_file,))

    frame.idle_components = []
    frame.add_text(
        text="START CONDUCTING TO MUSIC",
        center_frac_x=0.5,
        center_frac_y=0.25
    )
    frame.event_components = []
    frame.reset()
    frame.render()

    playback_thread.start()
    df.video_recorder.start_recording()

    playback_thread.join()
    df.video_recorder.stop_recording()

    frame.add_text(
        text="MUSIC FINISHED",
        center_frac_x=0.5,
        center_frac_y=0.5
    )
    frame.event_components = []
    frame.reset()
    frame.render()

```

```

def start_production(display_message):
    # Replace with the path to your audio file
    audio_path = 'testrun1.wav'
    video_path = 'testrun1_conduct.mp4'
    midi_path = "mid.midi"
    out_midi_path = 'testrun1_final_midi.mid'
    out_audio_path = 'testrun1_final_audio.wav'
    df.bpm = 90
    beat = '1/4'
    single_note = True


    # print("Hit enter to start recording audio: ")
    # input()
    # record_audio(20, save_path=audio_path)

    # print("Hit ENTER to convert the audio to midi file")
    # input()
    # audio_to_midi_conv(audio_path, midi_path, beat=beat, df.bpm=df.bpm)

    exit()

    print("Hit ENTER to record conducting video")
    input()
    start_audio_playback_recording_and_camera(
        audio_path, video_path
    )

    display_message("Start production")

    print('Hit ENTER to convert gesture video to IR')
    input()
    # Convert gesture to IR
    ir_data = gesture_ir_main(video_path)

    # Modify MIDI file based on IR and convert to audio
    print('Hit ENTER to convert video IR to dynamics: ')
    input()
    volume_changes = ir_data
    modify_volume(midi_path, volume_changes, out_midi_path, bpm=df.bpm)

    print('Hit ENTER to convert modified midi file to audio file')
    input()
    convert_midi_to_audio(out_midi_path, out_audio_path)

```

```python
    display_message("Production finished, let's see the result")


def show_result():
    # Visualize MIDI files and play back the modified audio
    original_midi = "converted_midi.mid"
    modified_midi = "output_file.mid"
    # visualize_notes(original_midi, modified_midi)

    play_audio_file("output_audio.wav")


def display_message(message):
    global current_message
    current_message = message

def switch_frame(switch_frame):
    print('Switching Frame')
    global current_frame
    current_frame = switch_frame

def start_recording_audio(frame, next_frame):
    if frame.get_text()[0] != '':
        df.bpm = int(frame.get_text()[0])
    metronome.reset_fps(df.bpm/30)
    print('recording with df.bpm: ', df.bpm)

    df.audio_recorder.initialize(bpm=df.bpm)
    switch_frame(next_frame)

def stop_recording_audio(next_frame):
    print('clicked stop recording audio')
    df.audio_recorder.save_recording()
    switch_frame(next_frame)

def save_recording_continue_cb(next_frame):
    convert_midi_to_audio(RAW_MIDI_PATH, RAW_AUDIO_PATH)
    switch_frame(next_frame)

def start_conducting_cb(frame, next_frame):
    frame.add_text(
        text="START CONDUCTING",
        center_frac_x=0.5,
        center_frac_y=0.25
    )

    frame.reset()
    frame.render()

    start_audio_playback_recording_and_camera(frame, RAW_AUDIO_PATH, RECORDED_VIDEO_PATH)

    frame.idle_components = []
    frame.add_text(
        text="RECORDING DONE",
        center_frac_x=0.5,
        center_frac_y=0.25
    )
    frame.reset()
    frame.render()

    time.sleep(1)
    frame.idle_components = []
    frame.add_button(
        "Start Conducting",
        0.4, 0.0625, 0.3, 0.4,
        callback=lambda: start_conducting_cb(start_record_video_frame, combine_frame)
    )

    switch_frame(next_frame)

def generate_result_cb(frame, next_frame):
    txt, button_rect, cb = frame.event_components[0]
    frame.event_components[0] = 'Converting...', button_rect, None
    frame.reset()
    frame.render()

    # Convert gesture to IR
    ir_info, ir_data = gesture_ir_main(RECORDED_VIDEO_PATH)

    # Modify MIDI file based on IR and convert to audio
    volume_changes = ir_data
    modify_volume(RAW_MIDI_PATH, volume_changes, MODIFIED_MIDI_PATH, bpm=df.bpm)

    convert_midi_to_audio(MODIFIED_MIDI_PATH, MODIFIED_AUDIO_PATH)

    frame.event_components[0] = txt, button_rect, cb
    frame.reset()
    frame.render()

    switch_frame(next_frame)
```

```python

def play_result_cb(frame, button_idx):
    txt, button_rect, cb = frame.event_components[button_idx]
    frame.event_components[button_idx] = 'Playing Audio...', button_rect, cb
    frame.reset()
    frame.render()

    play_audio_file(MODIFIED_AUDIO_PATH)

    frame.event_components[button_idx] = txt, button_rect, cb
    frame.reset()
    frame.render()

def start_over_cb():
    start_record_video_frame = GUI(window_width, window_height)
    start_record_video_frame.add_button(
        "Start Conducting",
        0.4, 0.0625, 0.3, 0.4,
        callback=lambda: start_conducting_cb(start_record_video_frame, combine_frame)
    )

    switch_frame(start_frame)

def re_conduct_cb():
    start_record_video_frame = GUI(window_width, window_height)
    start_record_video_frame.add_button(
        "Start Conducting",
        0.4, 0.0625, 0.3, 0.4,
        callback=lambda: start_conducting_cb(start_record_video_frame, combine_frame)
    )

    switch_frame(start_record_video_frame)


init_system()

window_width = 800
window_height = 600

# Combine done, playback results:
playback_frame = GUI(window_width, window_height)
play_idx = playback_frame.add_button(
    "Play Result",
    0.5, 0.0625, 0.25, 0.4,
    callback=lambda: play_result_cb(playback_frame, play_idx)
)

playback_frame.add_button(
    "Produce Another Audio",
    0.5, 0.0625, 0.25, 0.6,
    callback=lambda: start_over_cb()
)

playback_frame.add_button(
    "Re-Record Conducting Video",
    0.5, 0.0625, 0.25, 0.8,
    callback=lambda: re_conduct_cb()
)

# recording stops, click button to combine
combine_frame = GUI(window_width, window_height)
combine_frame.add_button(
    "Combine Video Dynamics with Audio",
    0.7, 0.0625, 0.15, 0.4,
    callback=lambda: generate_result_cb(combine_frame, playback_frame)
)


# frame to start recording video then convert to intermediate representation
start_record_video_frame = GUI(window_width, window_height)
start_record_video_frame.add_button(
    "Start Conducting",
    0.4, 0.0625, 0.3, 0.4,
    callback=lambda: start_conducting_cb(start_record_video_frame, combine_frame)
)

# Giving option to re-record audio or proceed:
re_record_audio_frame = GUI(window_width, window_height)
go_conducting_button = re_record_audio_frame.add_button(
    "Continue",
    0.4, 0.0625, 0.3, 0.4,
    callback=lambda: save_recording_continue_cb(start_record_video_frame)
)

re_record_button = re_record_audio_frame.add_button(
    "Re-Record Audio",
    0.4, 0.0625, 0.3, 0.6,
    callback=lambda: switch_frame(record_audio_frame)
)

```

```python
# frame when audio recording is in progress and waiting to stop
audio_recording_frame = GUI(window_width, window_height)
record_audio_button = audio_recording_frame.add_button(
    "Stop Recording",
    0.4, 0.0625, 0.3, 0.8,
    callback=lambda: stop_recording_audio(re_record_audio_frame)
)

audio_recording_frame.add_text(
    text="Hit ENTER to stop and save recording",
    center_frac_x=0.5,
    center_frac_y = 0.25
)
audio_recording_frame.add_circle(radius=50, center_frac_x=0.5, center_frac_y=0.5)
metronome = Metronome(frame=audio_recording_frame, fps=2)

audio_recording_frame.add_per_frame(metronome.switch)
audio_recording_frame.add_event_processor(df.audio_recorder.process_event)


# frame for starting to record audio
record_audio_frame = GUI(window_width, window_height)
record_audio_button = record_audio_frame.add_button(
    "Start Recording",
    0.4, 0.0625, 0.3, 0.4,
    callback=lambda: start_recording_audio(record_audio_frame, audio_recording_frame)
)
record_audio_frame.add_textbox(
    0.4, 0.0625, 0.3, 0.7
)

record_audio_frame.add_text(
    text="Click button to start recording audio",
    center_frac_x=0.5,
    center_frac_y=0.25
)

# starting screen
start_frame = GUI(window_width, window_height)
start_frame.add_button(
    "Start Production",
    0.4, 0.0625, 0.3, 0.4,
    callback=lambda: switch_frame(record_audio_frame)
)

```

```
clk = Clock()
running = True
current_frame = start_frame
while running:

    for event in pygame.event.get():
        exit = current_frame.event_trigger(event)
        if exit:
            running = False

    current_frame.reset()
    current_frame.render()
    clk.tick(current_frame.fps)

pygame.quit()


```

Keyboard_Pi_Conductor/metronome.py

```
white = (255, 255, 255)
black = (0, 0, 0)
red = (255, 0, 0)

import time
from pygame import mixer

class Metronome:
    def __init__(self, frame, fps):
        self.frame = frame
        self.bit = False

        self.prev_switch = None
        self.fps = fps
        self.time_step = 1 / fps

        self.sound = mixer.Sound(f'/home/pi/Documents/ECE5725_final_proj/PythonPiano/Synth_Block_F_lo.wav')

    def reset_fps(self, fps):
        self.fps = fps
        self.time_step = 1/fps

    def switch(self):

        if self.prev_switch is not None and time.time() - self.prev_switch < self.time_step:
            return

        self.prev_switch = time.time()
        self.bit = not self.bit
        center, radius, color = self.frame.circles[0]


        if self.bit:
            self.frame.circles[0] = (center, radius, red)
            self.sound.play()
        else:
            self.frame.circles[0] = (center, radius, white)

```

Keyboard_Pi_Conductor/mid_to_wav.py

```
from IR_Midi.modify_input import convert_midi_to_audio
midi_path = 'out/raw_midi.mid'
audio_path = 'recorded_audio.wav'
convert_midi_to_audio(midi_path, audio_path)

```

Keyboard_Pi_Conductor/Gesture_IR/init.py

```

```

Keyboard_Pi_Conductor/Gesture_IR/record_video.py

```
import cv2
import numpy as np
from threading import Thread
import time
# This is a script that records a video using the camera


class VideoRecorder:
    def __init__(self, video_path, bpm):

        self.video_path = video_path
        self.initialize

    def initialize(self, bpm):
        # Create a VideoCapture object
        self.camera = cv2.VideoCapture(0)
        self.camera_open = False

        # Check if the camera is opened
        if not self.camera.isOpened():
            print("Cannot open camera")
            exit()

        self.bpm = bpm
        self.fps = self.bpm / 60
        # Define the codec and create a VideoWriter object
        fourcc = cv2.VideoWriter_fourcc(*'mp4v')
        self.video_writer = cv2.VideoWriter(self.video_path, fourcc, self.fps, (640, 480))

    def spawn_camera(self):
        self.recording = False
        self.camera_open = True
        self.record_thread = Thread(target=self.record)
        self.record_thread.start()
        self.timestamp = time.time()

    def start_recording(self):
        while not self.camera.isOpened():
            time.sleep(0.01)
        self.recording = True
        self.timestamp = time.time()

    def stop_recording(self):
        # Release everything if job is finished

        self.recording = False
        self.camera_open = False

        self.record_thread.join()
        self.camera.release()
        self.video_writer.release()
        cv2.destroyAllWindows()

    def record(self):
        frames = []

        timestep = 1 / self.fps
        while self.camera_open and self.camera.isOpened():
            ret, frame = self.camera.read()

            if not ret:
                print("Can't receive frame (stream end?). Exiting ...")
                break

            # Flip the frame horizontally
            # frame = cv2.flip(frame, 1)
            frame = cv2.flip(frame, 0)

            if self.recording and time.time() - self.timestamp > timestep:
                self.timestamp = time.time()
                print(f'Appending Frame at {self.timestamp}')
                frames.append(frame)

            # Display the resulting frame
            cv2.imshow('frame', frame)

            k = cv2.waitKey(1) & 0xFF
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

        for f in frames:
            self.video_writer.write(f)


if __name__ == "__main__":
    recorder = VideoRecorder(video_path='output.mp4', bpm=90)
    input()
    recorder.spawn_camera()
    print('camera spawned')
    input()
    recorder.start_recording()
    print('starting recording')
    input()
    recorder.stop_recording()

```

Keyboard_Pi_Conductor/Gesture_IR/gesture_ir.py

```
import cv2
import copy
import mediapipe as mp
import numpy as np
from tqdm import tqdm
```

```
import matplotlib.pyplot as plt

from argparse import ArgumentParser
def calc_bounding_rect(image, landmarks):
    image_width, image_height = image.shape[1], image.shape[0]

    landmark_array = np.empty((0, 2), int)

    for _, landmark in enumerate(landmarks.landmark):
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)

        landmark_point = [np.array((landmark_x, landmark_y))]

        landmark_array = np.append(landmark_array, landmark_point, axis=0)

    x, y, w, h = cv2.boundingRect(landmark_array)

    return [x, y, x + w, y + h]

def get_bounding_box(image, results):
    for hand_landmarks, handedness in zip(results.multi_hand_landmarks,
                                            results.multi_handedness):
        # Bounding box calculation
        brect = calc_bounding_rect(image, hand_landmarks)
        return brect


def video_to_box_info(vid_path: str) -> list:

    video = cv2.VideoCapture(vid_path)

    frame_count = video.get(cv2.CAP_PROP_FRAME_COUNT)
    fps = video.get(cv2.CAP_PROP_FPS)
    duration = frame_count/fps

    info = {'fps': fps, 'duration': duration, 'frame_count': frame_count}
    # Check if video opened successfully
    if not video.isOpened():
        print("Error opening video file")

    mp_hands = mp.solutions.hands
    hands = mp_hands.Hands(
        static_image_mode=False,
        max_num_hands=1,
        min_detection_confidence=0.7,
        min_tracking_confidence=0.5,
    )


    progress_bar = tqdm(total=frame_count, desc="Processing frames", ncols=0)

    bounding_boxes = []
    # Read until video is completed
    curr_frame = 1
    while video.isOpened():

        ret, image = video.read()
        if not ret:
            break
        image = cv2.flip(image, 1)  # Mirror display

        # Detection implementation #############################################################
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        image.flags.writeable = False
        results = hands.process(image)

        image.flags.writeable = True

        #  ####################################################################
        if results.multi_hand_landmarks is not None:
            # brect = get_bounding_box(image, results)
            thumb = results.multi_hand_landmarks[0].landmark[4]
            middle = results.multi_hand_landmarks[0].landmark[12]
            brect = (thumb.x - middle.x) ** 2 + (thumb.y - middle.y)**2 + (thumb.z - middle.z) **2
            brect = np.sqrt(brect)

            timestamp = curr_frame / fps


            bounding_boxes.append((timestamp, brect))
        curr_frame += 1
        progress_bar.update(1)

    progress_bar.close()
    # When everything done, release the video capture object
    video.release()

    # Closes all the frames
    cv2.destroyAllWindows()

    return info, bounding_boxes


def converter(timestamp, open_scale):
    assert len(timestamp) == len(open_scale)

    output = []
    for i in range(len(open_scale) - 1):
        output.append((timestamp[i], timestamp[i+1], (open_scale[i+1] - open_scale[i])*100))

    return output

def gesture_ir_main(video_path):
    info, bounding_boxes = video_to_box_info(video_path)
    diag_len = bounding_boxes
    # diag_len = [ (time, np.sqrt((box[2] - box[0])**2 + (box[3] - box[1])**2)) for time, box in bounding_boxes]


    timestamps, diag_len = zip(*diag_len)
    plt.plot(timestamps, diag_len)
    plt.savefig('plot.png')

    box_sizes = np.array(diag_len)
    max_expand, min_expand = box_sizes.max(), box_sizes.min()

    open_scale = (box_sizes - min_expand) / (max_expand - min_expand)
    ir = converter(timestamps, open_scale)
    return info, ir

if __name__ == "__main__":

    parser = ArgumentParser()
    parser.add_argument('--vid_path', type=str)
    args = parser.parse_args()

    print(gesture_ir_main(args.vid_path))

```

Keyboard_Pi_Conductor/GUI.py

```
import pygame
```

```python

white = (255, 255, 255)
black = (0, 0, 0)
red = (255, 0, 0)

def init_system():
    pygame.init()

class GUI:
    def __init__(self, width, height, fps=30):

        self.window_width = width
        self.window_height = height
        self.window = pygame.display.set_mode((self.window_width, self.window_height))
        pygame.display.set_caption("Music Production")

        self.event_components = []
        self.event_processors = []
        self.idle_components = []
        self.circles = []
        self.per_frame_call = []
        self.text_boxes = []
        self.active_text_box = -1
        self.running = False

        self.font = pygame.font.Font(None, 36)

        self.fps = fps

    def add_button(self, text, width_frac, height_frac, center_frac_x, center_frac_y, callback=None):
        rect = pygame.Rect(
            center_frac_x * self.window_width,
            center_frac_y * self.window_height,
            width_frac * self.window_width,
            height_frac * self.window_height
        )

        self.event_components.append((text, rect, callback))
        return len(self.event_components) - 1


    def event_trigger(self, event):
        if event.type == pygame.QUIT:
            self.running = False
            return True

        for text, component, cb in self.event_components:
            if not event.type == pygame.MOUSEBUTTONDOWN:
                continue
            if component.collidepoint(event.pos) and cb is not None:
                cb()

        self.active_text_box = -1
        for i in range(len(self.text_boxes)):
            text, component, active = self.text_boxes[i]
            if not event.type == pygame.MOUSEBUTTONDOWN:
                continue

            self.text_boxes[i] = text, component, component.collidepoint(event.pos)
            if component.collidepoint(event.pos):
                self.active_text_box = i

        for i in range(len(self.text_boxes)):
            text, component, active = self.text_boxes[i]
            if not active:
                continue
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN:
                    active = False
                    self.active_text_box = -1
                elif event.key == pygame.K_BACKSPACE:
                    text = text[:-1]
                else:
                    text += event.unicode
                self.text_boxes[i] = text, component, active


        for processor in self.event_processors:
            processor(event)

        return False


    def add_text(self, text, center_frac_x, center_frac_y):
        text = self.font.render(text, True, black)
        text_rect = text.get_rect(center=(center_frac_x * self.window_width, center_frac_y * self.window_height))

        self.idle_components.append((text, text_rect))

```

```python
    def add_circle(self, radius, center_frac_x, center_frac_y):
        self.circles.append(((center_frac_x * self.window_width, center_frac_y * self.window_height), radius, black))

    def add_textbox(self, width_frac, height_frac, center_frac_x, center_frac_y):
        rect = pygame.Rect(
            center_frac_x * self.window_width,
            center_frac_y * self.window_height,
            width_frac * self.window_width,
            height_frac * self.window_height
        )

        self.text_boxes.append(('', rect, False))

    def get_text(self):
        ret = []
        for text, _, _ in self.text_boxes:
            ret.append(text)

        return ret

    def reset(self):
        self.window.fill(white)


    def start(self):
        self.running = True

    def add_per_frame(self, func):
        self.per_frame_call.append(func)

    def add_event_processor(self, func):
        self.event_processors.append(func)

    def render(self):

        for func in self.per_frame_call:
            func()

        for text, component, cb in self.event_components:
            pygame.draw.rect(self.window, black, component, 2)
            render_text = self.font.render(text, True, black)
            text_rect = render_text.get_rect(center=component.center)
            self.window.blit(render_text, text_rect)


        for text, component, active in self.text_boxes:
            color = (red if active else black)
            pygame.draw.rect(self.window, color, component, 3)
            if text == '' and not active:
                text = 'Enter Text Here'
            render_text = self.font.render(text, True, black)
            text_rect = render_text.get_rect(center=component.center)
            self.window.blit(render_text, text_rect)

        for text, text_rect in self.idle_components:
            self.window.blit(text, text_rect)

        for center, radius, color in self.circles:
            pygame.draw.circle(self.window, color, center, radius, width=0)


        pygame.display.flip()


# pygame.init()
# gui = GUI(width=800, height=600)

# gui.add_button(
#     "Start Production",
#     0.4, 0.0625, 0.3, 0.4,
#     callback=lambda: print("START PRODUCTION")
# )

# gui.add_button(
#     "See Result",
#     0.4, 0.0625, 0.3, 0.65,
#     callback=lambda: print("See Results")
# )

# gui.add_text(
#     text="Are you ready to produce your music?",
#     center_frac_x=0.5,
#     center_frac_y = 0.25
# )
# running = True


# gui2 = GUI(width=800, height=600)

# gui2.add_button(
#     "Start Production 2",
#     0.4, 0.0625, 0.3, 0.4,
#     callback=lambda: print("START PRODUCTION 2")
# )

# gui2.add_button(
#     "See Result 2",
#     0.4, 0.0625, 0.3, 0.65,
#     callback=lambda: print("See Results 2")
# )

# gui2.add_text(
#     text="Are you ready to produce your music? 2",
#     center_frac_x=0.5,
#     center_frac_y = 0.25
# )
# running = True


# i = 0
# while running:
#     if i < 100:
#         g = gui
#     else:
#         g = gui2

#     for event in pygame.event.get():
#         g.event_trigger(event)

#     g.reset()
#     g.render()
#     i += 1








```

Keyboard_Pi_Conductor/get_code.py

```
import os
from pathlib import Path

```

````python

def write_code(file):
    with open('code.md', 'a') as f:
        # remove root path from file string
        f.write(f'__{file}__\n\n')
        f.write('```\n')
        with open(file, 'r')  as code_file:
            lines = code_file.readlines()
            f.writelines(lines)
        f.write('\n```')
        f.write('\n\n')

def main(path):
    for f in os.listdir(path):
        f = os.path.join(path, f)
        if os.path.isdir(f):
            main(f)
        elif f.endswith('.py'):
            write_code(f)

if __name__ == "__main__":
    root_path = Path(__file__).parent
    print(f'{root_path=}')
    main(root_path)



````

Keyboard_Pi_Conductor/IR_Midi/visualize_note.py

```python
from music21 import converter, stream, tempo, meter, chord, note

import midi2audio
import pygame
from IR_Midi.modify_input import modify_volume
from visual_midi import Plotter, Preset
from pretty_midi import PrettyMIDI


def get_note_info(midi_file):
    score = converter.parse(midi_file)
    notes = []
    for element in score.flat.notesAndRests:
        if isinstance(element, stream.Voice):
            element = element.flat.notesAndRests
```

```
        if isinstance(element, chord.Chord):
            pitches = '.'.join(n.nameWithOctave for n in element.pitches)
            volume = element.volume.velocity
        elif isinstance(element, note.Note):
            pitches = element.pitch.nameWithOctave
            volume = element.volume.velocity
        else:
            pitches = ''
            volume = 0

        notes.append({
            "pitch": pitches,
            "start": element.offset,
            "end": element.offset + element.duration.quarterLength,
            "volume": volume
        })

    return notes


def draw_note_volumes(screen, notes, y_offset, color):
    for note in notes:
        start_x = int(note["start"] * 50)
        end_x = int(note["end"] * 50)
        volume = int(note["volume"] / 127 * screen.get_height() / 2)
        pygame.draw.rect(screen, color, (start_x, y_offset -
                         volume, end_x - start_x, volume))
        pygame.draw.rect(screen, (0, 0, 0), (start_x,
                         y_offset - volume, end_x - start_x, volume), 1)

        font = pygame.font.Font(None, 24)
        pitch_text = font.render(note["pitch"], True, (0, 0, 0))
        volume_text = font.render(str(note["volume"]), True, (0, 0, 0))
        screen.blit(pitch_text, (start_x, y_offset - volume - 20))
        screen.blit(volume_text, (start_x, y_offset - volume - 40))


def draw_modified_notes(screen, original_notes, modified_notes):
    for i in range(len(original_notes)):
        if original_notes[i]["volume"] != modified_notes[i]["volume"]:
            start_x = int(modified_notes[i]["start"] * 50)
            end_x = int(modified_notes[i]["end"] * 50)
            volume = int(modified_notes[i]["volume"] /
                         127 * screen.get_height() / 2)
            pygame.draw.rect(screen, (255, 0, 0), (start_x,
                             screen.get_height() - volume, end_x - start_x, volume))
            pygame.draw.rect(screen, (0, 0, 0), (start_x, screen.get_height(
            ) - volume, end_x - start_x, volume), 1)

            font = pygame.font.Font(None, 24)
            pitch_text = font.render(
                modified_notes[i]["pitch"], True, (0, 0, 0))
            volume_text = font.render(
                str(modified_notes[i]["volume"]), True, (0, 0, 0))
            screen.blit(
                pitch_text, (start_x, screen.get_height() - volume - 20))
            screen.blit(
                volume_text, (start_x, screen.get_height() - volume - 40))


def visualize_notes(original_file, modified_file):
    original_notes = get_note_info(original_file)
    modified_notes = get_note_info(modified_file)

    pygame.init()
    screen_width, screen_height = 800, 600
    screen = pygame.display.set_mode((screen_width, screen_height))
    pygame.display.set_caption("MIDI Note Volume Visualization")

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        screen.fill((255, 255, 255))

        draw_note_volumes(screen, original_notes,
                          screen_height // 2, (0, 0, 255))
        draw_note_volumes(screen, modified_notes, screen_height, (0, 0, 255))
        draw_modified_notes(screen, original_notes, modified_notes)

        pygame.display.flip()

    pygame.quit()


def visualize_midi(original_file, modified_file):
    original_pm = PrettyMIDI(original_file)
    modified_pm = PrettyMIDI(modified_file)

    preset = Preset(plot_width=800, plot_height=400, row_height=50)
```

```python
    plotter = Plotter(preset, plot_max_length_bar=16)

    plotter.show(original_pm, "/tmp/original_midi.html")
    plotter.show(modified_pm, "/tmp/modified_midi.html")


def test_visual():
    volume_changes = [
        (3.0, 8.0, 130),  # Louder at 7s, lasting for 3s
        (10.0, 380.0, -100)  # Quieter at 15s, lasting for 3s
    ]

    modified_score = modify_volume('input.mid', volume_changes)
    modified_score.write('midi', 'output_file.mid')

    visualize_notes('input.mid', 'output_file.mid')
    visualize_midi('input.mid', 'output_file.mid')


if __name__ == "__main__":
    test_visual()

```

Keyboard_Pi_Conductor/IR_Midi/merge_midi.py

```python
from music21 import converter, instrument, note, chord, stream


def merge_midi(path_1, path2, out_path):
    """
    Merge two tracks into 1.

    Args:
        path_1, paht_2 (str): Path to the MIDI file.
        out_path (str): Path to the output MIDI file

    Returns:
        music21.stream.Stream: Modified MIDI score.
    """
    midi_file1 = converter.parse(path_1)
    midi_file2 = converter.parse(path2)

    merged_stream = stream.Stream()

    for element in midi_file1.flat:
        merged_stream.append(element)

    for element in midi_file2.flat:
        merged_stream.append(element)

    merged_stream.write("midi", fp=out_path)


if __name__ == "__main__":
    path_1 = "/home/pi/Documents/ECE5725_final_proj/testrun1_final_midi.mid"
    path_2 = "/home/pi/Documents/ECE5725_final_proj/IR_Midi/input.mid"
    out = "/home/pi/Documents/ECE5725_final_proj/IR_Midi/merged_output.mid"
    merge_midi(path_1, path_2, out)

```

Keyboard_Pi_Conductor/IR_Midi/modify_input.py

```
from music21 import converter, stream, tempo, meter

import midi2audio
import pygame
import numpy as np


def convert_midi_to_audio(midi_file, audio_file):
    """
    Convert a MIDI file to an audio file using midi2audio.

    Args:
        midi_file (str): Path to the MIDI file.
        audio_file (str): Path to save the audio file.
    """
    fs = midi2audio.FluidSynth(
        sound_font='/usr/share/sounds/sf2/FluidR3_GM.sf2')
    fs.midi_to_audio(midi_file, audio_file)


def play_audio_file(audio_file):
    """
    Play an audio file using pygame.

    Args:
        audio_file (str): Path to the audio file.
    """
    print("start play audio")
```

```python
    pygame.mixer.init()
    pygame.mixer.music.load(audio_file)
    pygame.mixer.music.play()
    while pygame.mixer.music.get_busy():
        pygame.time.delay(100)
    pygame.mixer.quit()


def modify_volume(midi_file, volume_changes, out_midi_file, bpm=60):
    """
    Modify the volume of a MIDI file based on user input.

    Args:
        midi_file (str): Path to the MIDI file.
        volume_changes (list): List of tuples containing (start time, end time) and volume change.

    Returns:
        music21.stream.Stream: Modified MIDI score.
    """
    # Load the MIDI file
    score = converter.parse(midi_file)

    flat_score = score.flat
    # bpm=score.flat.getElementsByClass(tempo.MetronomeMark)[0].number

    for note in flat_score.notes:
        print(f'note offset: {note.offset}')

    volume = 0
    low, high = 0, 0
    for _, _, volume_change in volume_changes:
        volume += volume_change
        low = min(low, volume)
        high = max(high, volume)

    for i in range(len(volume_changes)):
        t1, t2, volume_change = volume_changes[i]
        volume_changes[i] = (t1, t2, volume_change / (high - low) * 87)

    i = 0
    volume = 0 - low + 40

    note_volume_list = []
    while flat_score.notes[i].offset < volume_changes[i][0]:
        note_volume_list.append(volume)
        i += 1

    buffer = []

    for start_time_seconds, end_time_seconds, volume_change in volume_changes:
        end_offset = bpm * end_time_seconds / 60

        while i < len(flat_score.notes) and flat_score.notes[i].offset < end_offset:
            buffer.append(flat_score.notes[i])
            i += 1

        for note in buffer:
            volume += volume_change / len(buffer)
            note_volume_list.append(volume)

        buffer = []

    while i < len(flat_score.notes):
        note_volume_list.append(volume)
        i += 1

    note_volume_list = np.array(note_volume_list)
    note_volume_list = (note_volume_list - note_volume_list.min()) / \
        (note_volume_list.max() - note_volume_list.min()) * 100 + 20

    print('note_volume_list: ', note_volume_list)
    for i in range(len(note_volume_list)):
        flat_score.notes[i].volume.velocity = note_volume_list[i]

    score.write('midi', out_midi_file)


def test_volume_modification():
    volume_changes = [
        (3.0, 8.0, 130),  # Louder at 7s, lasting for 3s
        (10.0, 380.0, -100)  # Quieter at 15s, lasting for 3s
    ]

    modified_score = modify_volume(
        '/home/pi/Documents/ECE5725_final_proj/output_file.mid', volume_changes, 'output_file.mid')

    # Convert the modified MIDI file to an audio file
    audio_file = 'output_audio.wav'
    convert_midi_to_audio('output_file.mid', audio_file)

    # Play the audio file
    play_audio_file(audio_file)


if __name__ == "__main__":
    test_volume_modification()
    # import sys
    # play_audio_file(sys.argv[1])

```

Keyboard_Pi_Conductor/IR_Midi/init**.py**

```

```