

Keyboard_Pi_Conductor/PythonPiano/key_midi.py

```
1  import pygame
2  import PythonPiano.piano_lists as pl
3  from pygame import mixer
4  from music21 import note, stream, tempo
5  from threading import Thread
6  import time
7
8
9  class AudioRecorder:
10     def __init__(self, save_path, bpm=90):
11         print('initializing new audio recorder')
12         self.save_path = save_path
13         self.bpm = bpm
14         self.metronome_mark = tempo.MetronomeMark(number=self.bpm)
15         self.left_oct = 4
16         self.right_oct = 5
17         self.white_notes = pl.white_notes
18         self.black_notes = pl.black_notes
19         self.black_labels = pl.black_labels
20         # windowSurfaceObj = pygame.display.set_mode((64,48),1,16)
21
22         self.initialize(bpm)
23
24     def initialize(self, bpm=None):
25         if bpm is not None:
26             self.bpm = bpm
27             self.metronome_mark = tempo.MetronomeMark(number=self.bpm)
28         self.score = stream.Stream()
29         self.score.append(self.metronome_mark)
30         pygame.mixer.init()
31         pygame.mixer.set_num_channels(50)
32
33         self.white_sounds = []
34         self.black_sounds = []
35         for i in range(len(self.white_notes)):
36
37             self.white_sounds.append(mixer.Sound(
38
39 f'/home/pi/Documents/ECE5725_final_proj/PythonPiano/assets/notes/{self.white_notes[i
40 ]}.wav'))
41
42         for i in range(len(self.black_notes)):
43             self.black_sounds.append(mixer.Sound(
```

```

42 f'/home/pi/Documents/ECE5725_final_proj/PythonPiano/assets/notes/{self.black_notes[i
43   ]}.wav'))
44
45     self.note_with_time = []
46     self.left_dict = {'Z': f'C{self.left_oct}', 'S': f'C#{self.left_oct}', 'X':
47 f'D{self.left_oct}', 'D': f'D#{self.left_oct}',
48     'C': f'E{self.left_oct}', 'V': f'F{self.left_oct}', 'G':
49 f'F#{self.left_oct}', 'B': f'G{self.left_oct}',
50     'H': f'G#{self.left_oct}', 'N': f'A{self.left_oct}', 'J':
51 f'A#{self.left_oct}', 'M': f'B{self.left_oct}'}
52
53     self.right_dict = {
54         'R': f'C{self.right_oct}',
55         '5': f'C#{self.right_oct}',
56         'T': f'D{self.right_oct}',
57         '6': f'D#{self.right_oct}',
58         'Y': f'E{self.right_oct}',
59         'U': f'F{self.right_oct}',
60         '8': f'F#{self.right_oct}',
61         'I': f'G{self.right_oct}',
62         '9': f'G#{self.right_oct}',
63         'O': f'A{self.right_oct}',
64         '0': f'A#{self.right_oct}',
65         'P': f'B{self.right_oct}'
66     }
67
68     def start_recording(self):
69         self.recording = True
70         self.record()
71
72     def process_event(self, event):
73         if event.type == pygame.QUIT:
74             self.clean()
75             return
76         if event.type == pygame.TEXTINPUT:
77             if event.text.upper() in self.left_dict:
78                 keynote = self.left_dict[event.text.upper()]
79                 self.note_with_time.append((keynote, time.time()))
80
81                 if self.left_dict[event.text.upper()][1] == '#':
82                     index = self.black_labels.index(keynote)
83
84                     self.black_sounds[index].play(0, 1000)
85             else:
86                 index = self.white_notes.index(keynote)

```

```

83         self.white_sounds[index].play(0, 1000)
84
85         # self.score.append(note.Note(keynote, type='quarter'))
86     if event.text.upper() in self.right_dict:
87         keynote = self.right_dict[event.text.upper()]
88         self.note_with_time.append((keynote, time.time()))
89
90         if self.right_dict[event.text.upper()][1] == '#':
91             index = self.black_labels.index(keynote)
92             self.black_sounds[index].play(0, 1000)
93         else:
94             index = self.white_notes.index(keynote)
95             self.white_sounds[index].play(0, 1000)
96
97         # self.score.append(note.Note(keynote, type='quarter'))
98     if event.type == pygame.KEYDOWN:
99         if event.key == pygame.K_RETURN or event.key == pygame.K_KP_ENTER:
100             print('enter:stop')
101             self.save_recording()
102             return
103
104     def record(self):
105         i = 0
106         while self.recording:
107             for event in pygame.event.get():
108                 if event.type == pygame.QUIT:
109                     self.clean()
110                     return
111                 if event.type == pygame.TEXTINPUT:
112                     if event.text.upper() in self.left_dict:
113                         keynote = self.left_dict[event.text.upper()]
114
115                         if self.left_dict[event.text.upper()][1] == '#':
116                             index = self.black_labels.index(keynote)
117
118                             self.black_sounds[index].play(0, 1000)
119                         else:
120                             index = self.white_notes.index(keynote)
121                             self.white_sounds[index].play(0, 1000)
122
123                         self.note_with_time.append((keynote, time.time()))
124                         self.score.append(note.Note(keynote, type='quarter'))
125
126                     if event.text.upper() in self.right_dict:
127                         keynote = self.right_dict[event.text.upper()]
128

```

```

129         if self.right_dict[event.text.upper()][1] == '#':
130             index = self.black_labels.index(keynote)
131             self.black_sounds[index].play(0, 1000)
132         else:
133             index = self.white_notes.index(keynote)
134             self.white_sounds[index].play(0, 1000)
135
136         self.note_with_time.append((keynote, time.time()))
137         # self.score.append(note.Note(keynote, type='quarter'))
138     if event.type == pygame.KEYDOWN:
139         if event.key == pygame.K_RETURN or event.key ==
pygame.K_KP_ENTER:
140             print('enter:stop')
141             self.save_recording()
142             return
143
144     def save_recording(self):
145         eighth_timestep = 1 / (self.bpm * 2 / 60)
146         for i in range(len(self.note_with_time) - 1):
147             curr_note, curr_time = self.note_with_time[i]
148             _, next_time = self.note_with_time[i+1]
149
150             n_time_steps = round((next_time - curr_time) / eighth_timestep)
151             self.score.append(
152                 note.Note(curr_note, quarterLength=n_time_steps/2))
153
154             self.score.append(
155                 note.Note(self.note_with_time[-1][0], quarterLength=2))
156
157             print('stop record')
158             self.recording = False
159             self.score.write('midi', fp=self.save_path)
160             print(f"Recording stopped. MIDI file saved as {self.save_path}")
161             pygame.mixer.quit()
162
163
164 if __name__ == "__main__":
165     recorder = AudioRecorder(save_path='test_recorder.mid')
166     print('Hit Enter to Start Recording')
167     input()
168     recorder.start_recording()
169     print('Hit Enter to stop recording')
170     input()
171     recorder.stop_recording()
172

```

Keyboard_Pi_Conductor/PythonPiano/piano_lists.py

```
1 left_hand = ['Z', 'S', 'X', 'D', 'C', 'V', 'G', 'B', 'H', 'N', 'J', 'M']
2 right_hand = ['R', '5', 'T', '6', 'Y', 'U', '8', 'I', '9', 'O', '0', 'P']
3
4 piano_notes = ['A0', 'A0#', 'B0', 'C1', 'C1#', 'D1', 'D1#', 'E1', 'F1', 'F1#', 'G1',
5               'G1#',
6               'A1', 'A1#', 'B1', 'C2', 'C2#', 'D2', 'D2#', 'E2', 'F2', 'F2#', 'G2',
7               'G2#',
8               'A2', 'A2#', 'B2', 'C3', 'C3#', 'D3', 'D3#', 'E3', 'F3', 'F3#', 'G3',
9               'G3#',
10              'A3', 'A3#', 'B3', 'C4', 'C4#', 'D4', 'D4#', 'E4', 'F4', 'F4#', 'G4',
11              'G4#',
12              'A4', 'A4#', 'B4', 'C5', 'C5#', 'D5', 'D5#', 'E5', 'F5', 'F5#', 'G5',
13              'G5#',
14              'A5', 'A5#', 'B5', 'C6', 'C6#', 'D6', 'D6#', 'E6', 'F6', 'F6#', 'G6',
15              'G6#',
16              'A6', 'A6#', 'B6', 'C7', 'C7#', 'D7', 'D7#', 'E7', 'F7', 'F7#', 'G7',
17              'G7#',
18              'A7', 'A7#', 'B7', 'C8']
19
20 white_notes = ['A0', 'B0', 'C1', 'D1', 'E1', 'F1', 'G1',
21               'A1', 'B1', 'C2', 'D2', 'E2', 'F2', 'G2',
22               'A2', 'B2', 'C3', 'D3', 'E3', 'F3', 'G3',
23               'A3', 'B3', 'C4', 'D4', 'E4', 'F4', 'G4',
24               'A4', 'B4', 'C5', 'D5', 'E5', 'F5', 'G5',
25               'A5', 'B5', 'C6', 'D6', 'E6', 'F6', 'G6',
26               'A6', 'B6', 'C7', 'D7', 'E7', 'F7', 'G7',
27               'A7', 'B7', 'C8']
28
29 black_notes = ['Bb0', 'Db1', 'Eb1', 'Gb1', 'Ab1',
30               'Bb1', 'Db2', 'Eb2', 'Gb2', 'Ab2',
31               'Bb2', 'Db3', 'Eb3', 'Gb3', 'Ab3',
32               'Bb3', 'Db4', 'Eb4', 'Gb4', 'Ab4',
33               'Bb4', 'Db5', 'Eb5', 'Gb5', 'Ab5',
34               'Bb5', 'Db6', 'Eb6', 'Gb6', 'Ab6',
35               'Bb6', 'Db7', 'Eb7', 'Gb7', 'Ab7',
36               'Bb7']
37
38 black_labels = ['A#0', 'C#1', 'D#1', 'F#1', 'G#1',
39                'A#1', 'C#2', 'D#2', 'F#2', 'G#2',
40                'A#2', 'C#3', 'D#3', 'F#3', 'G#3',
41                'A#3', 'C#4', 'D#4', 'F#4', 'G#4',
42                'A#4', 'C#5', 'D#5', 'F#5', 'G#5',
43                'A#5', 'C#6', 'D#6', 'F#6', 'G#6',
```

```
37         'A#6', 'C#7', 'D#7', 'F#7', 'G#7',
38         'A#7' ]
```

Keyboard_Pi_Conductor/Audio_Midi/init.py

```
1 |
```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/integrate_main.py

```
1  import os
2  import sys
3  import logging
4
5  from Audio_Midi.audio_to_midi import converter, progress_bar
6
7
8  def _convert_beat_to_time(bpm, beat):
9      try:
10         parts = beat.split("/")
11         if len(parts) > 2:
12             raise Exception()
13
14         beat = [int(part) for part in parts]
15         fraction = beat[0] / beat[1]
16         bps = bpm / 60
17         ms_per_beat = bps * 1000
18         return fraction * ms_per_beat
19     except Exception:
20         raise RuntimeError("Invalid beat format: {}".format(beat))
21
22
23  def audio_to_midi_conv(infile, outfile, beat='1/4', single_note=True):
24      try:
25         logging.basicConfig(level=logging.DEBUG, format="%(message)s")
26
27         time_window = _convert_beat_to_time(bpm, beat)
28
29         global bpm
30         process = converter.Converter(
31             infile=infile,
32             outfile=outfile,
33             time_window=time_window,
34             activation_level=0.0,
35             condense=False,
36             condense_max=False,
```

```

37         max_note_length=0,
38         note_count=1 if single_note else 0,
39         transpose=0,
40         pitch_set=[],
41         pitch_range=None,
42         progress=progress_bar.ProgressBar(),
43         bpm=bpm,
44     )
45     process.convert()
46 except KeyboardInterrupt:
47     sys.exit(1)
48 except Exception as e:
49     logging.exception(e)
50     sys.exit(1)
51

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/midi_writer.py

```

1  from collections import defaultdict
2
3  import python3_midi as midi
4
5
6  class NoteState:
7      __slots__ = ["is_active", "event_pos", "count"]
8
9      def __init__(self, is_active=False, event_pos=None, count=0):
10         self.is_active = is_active
11         self.event_pos = event_pos
12         self.count = count
13
14
15  class MidiWriter:
16      def __init__(
17          self,
18          outfile,
19          channels,
20          time_window,
21          bpm=60,
22          condense=False,
23          condense_max=False,
24          max_note_length=0,
25      ):
26         self.outfile = outfile
27         self.condense = condense

```

```

28         self.condense_max = condense_max
29         self.max_note_length = max_note_length
30         self.channels = channels
31         self.time_window = time_window
32         self.bpm = bpm
33         self.note_state = [defaultdict(lambda: NoteState()) for _ in
range(channels)]
34
35         bps = self.bpm / 60
36         self.ms_per_beat = int((1.0 / bps) * 1000)
37         self.tick_increment = int(time_window)
38         self.skip_count = 1
39         self._need_increment = False
40
41     def __enter__(self):
42         self.stream = midi.FileStream(self.outfile)
43         self.stream.start_pattern(
44             format=1,
45             tick_relative=False,
46             resolution=self.ms_per_beat,
47             tracks=[],
48         )
49         self.stream.start_track(
50             events=[
51                 midi.TimeSignatureEvent(
52                     tick=0,
53                     numerator=1,
54                     denominator=4,
55                     metronome=int(self.ms_per_beat / self.time_window),
56                     thirtyseconds=32,
57                 )
58             ],
59             tick_relative=False,
60         )
61         return self
62
63     def __exit__(self, type, value, traceback):
64         self._terminate_notes()
65         self.stream.add_event(midi.EndOfTrackEvent(tick=1))
66         self.stream.end_track()
67         self.stream.end_pattern()
68         self.stream.close()
69
70     def _skip(self):
71         self.skip_count += 1
72

```



```

73     def _reset_skip(self):
74         self.skip_count = 1
75
76     @property
77     def tick(self):
78         ret = 0
79         if self._need_increment:
80             self._need_increment = False
81             ret = self.tick_increment * self.skip_count
82             self._reset_skip()
83         return ret
84
85     def _note_on(self, channel, pitch, velocity):
86         pos = self.stream.add_event(
87             midi.NoteOnEvent(
88                 tick=self.tick, channel=channel, pitch=pitch, velocity=60
89             )
90         )
91         self.note_state[channel][pitch] = NoteState(
92             True,
93             pos,
94             1,
95         )
96
97     def _note_off(self, channel, pitch):
98         self.note_state[channel][pitch] = NoteState()
99         self.stream.add_event(
100             midi.NoteOffEvent(
101                 tick=self.tick,
102                 channel=channel,
103                 pitch=pitch,
104             )
105         )
106
107     def add_notes(self, notes):
108         """
109         notes is a list of midi notes to add at the current
110         time step.
111
112         Adds each note in the list to the current time step
113         with the volume, track and channel specified.
114         """
115         self._need_increment = True
116         if not self.condense:
117             self._terminate_notes()
118

```

```

119         for channel, notes in enumerate(notes):
120             new_notes = set()
121             stale_notes = []
122             for note in notes:
123                 note_state = self.note_state[channel][note.pitch]
124                 new_notes.add(note.pitch)
125                 if (not self.condense) or (self.condense and not
note_state.is_active):
126                     self._note_on(channel, note.pitch, note.velocity)
127                 elif self.condense and note_state.is_active:
128                     event = self.stream.get_event(
129                         midi.NoteOnEvent, note_state.event_pos
130                     )
131                     old_velocity = event.data[1]
132                     if self.condense_max:
133                         new_velocity = max(note.velocity, old_velocity)
134                     else:
135                         count = note_state.count
136                         note_state.count += 1
137                         new_velocity = ((old_velocity * count) + note.velocity) // (
138                             note_state.count
139                         )
140                     if old_velocity != event.data[1]:
141                         event.data[1] = new_velocity
142                         self.stream.set_event(event, note_state.event_pos)
143
144             if self.condense:
145                 active_notes = [
146                     note
147                     for note in self.note_state[channel]
148                     if self.note_state[channel][note].is_active
149                 ]
150                 for note in active_notes:
151                     if (
152                         note not in new_notes
153                         or self.note_state[channel][note].count >
self.max_note_length
154                     ):
155                         stale_notes.append(note)
156
157                 for note in stale_notes:
158                     self._note_off(channel, note)
159
160             if self._need_increment:
161                 self._skip()
162

```

```

163     def _terminate_notes(self):
164         for channel in range(self.channels):
165             for note, note_state in self.note_state[channel].items():
166                 if note_state.is_active:
167                     self._note_off(channel, note)
168

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/converter.py

```

1  import logging
2
3  from collections import namedtuple
4  from functools import lru_cache
5  from operator import attrgetter
6
7  import numpy
8  import soundfile
9
10 # import midi_writer, notes
11 from Audio_Midi.audio_to_midi import midi_writer, notes
12
13
14 class Note:
15     __slots__ = ["pitch", "velocity", "count"]
16
17     def __init__(self, pitch, velocity, count=0):
18         self.pitch = pitch
19         self.velocity = velocity
20         self.count = count
21
22
23 class Converter:
24     def __init__(
25         self,
26         infile=None,
27         outfile=None,
28         time_window=None,
29         activation_level=None,
30         condense=None,
31         condense_max=False,
32         max_note_length=0,
33         transpose=0,
34         pitch_set=None,
35         pitch_range=None,
36         note_count=None,

```

```

37         progress=None,
38         bpm=60,
39     ):
40
41         if infile:
42             self.info = soundfile.info(infile)
43         else:
44             raise RuntimeError("No input provided.")
45
46         self.infile = infile
47         self.outfile = outfile
48         self.time_window = time_window
49         self.condense = condense
50         self.condense_max = condense_max
51         self.max_note_length = max_note_length
52         self.transpose = transpose
53         self.pitch_set = pitch_set
54         self.pitch_range = pitch_range or [0, 127]
55         self.note_count = note_count
56         self.progress = progress
57         self.bpm = bpm
58
59         self.activation_level = int(127 * activation_level) or 1
60         self.block_size = self._time_window_to_block_size(
61             self.time_window, self.info.samplerate
62         )
63
64         steps = self.info.frames // self.block_size
65         self.total = steps
66         self.current = 0
67
68         self._determine_ranges()
69
70     def _determine_ranges(self):
71         self.notes = notes.generate()
72         self.max_freq = min(self.notes[127][-1], self.info.samplerate / 2)
73         self.min_freq = max(self.notes[0][-1], 1000 / self.time_window)
74         self.bins = self.block_size // 2
75         self.frequencies = numpy.fft.fftfreq(self.bins, 1 / self.info.samplerate)[
76             : self.bins // 2
77         ]
78
79         for i, f in enumerate(self.frequencies):
80             if f >= self.min_freq:
81                 self.min_bin = i
82                 break

```

```

83         else:
84             self.min_bin = 0
85         for i, f in enumerate(self.frequencies):
86             if f >= self.max_freq:
87                 self.max_bin = i
88                 break
89         else:
90             self.max_bin = len(self.frequencies)
91
92     def _increment_progress(self):
93         if self.progress:
94             self.current += 1
95             self.progress.update(self.current, self.total)
96
97     @staticmethod
98     def _time_window_to_block_size(time_window, rate):
99         """
100         time_window is the time in ms over which to compute fft's.
101         rate is the audio sampling rate in samples/sec.
102
103         Transforms the time window into an index step size and
104         returns the result.
105         """
106
107         # rate/1000(samples/ms) * time_window(ms) = block_size(samples)
108         rate_per_ms = rate / 1000
109         block_size = rate_per_ms * time_window
110
111         return int(block_size)
112
113     def _freqs_to_midi(self, freqs):
114         """
115         freq_list is a list of frequencies with normalized amplitudes.
116
117         Takes a list of notes and transforms the amplitude to a
118         midi volume as well as adding track and channel info.
119         """
120
121         notes = [None for _ in range(128)]
122         for pitch, velocity in freqs:
123             if not (self.pitch_range[0] <= pitch <= self.pitch_range[1]):
124                 continue
125             velocity = min(int(127 * (velocity / self.bins)), 127)
126
127             if velocity > self.activation_level:
128                 if not notes[pitch]:

```

```

129         notes[pitch] = Note(pitch, 60)
130     else:
131         notes[pitch].velocity = int(
132             ((notes[pitch].velocity * notes[pitch].count) + velocity)
133             / (notes[pitch].count + 1)
134         )
135         notes[pitch].count += 1
136
137     notes = [note for note in notes if note]
138
139     if self.note_count > 0:
140         max_count = min(len(notes), self.note_count)
141         notes = sorted(notes, key=attrgetter("velocity"))[::-1][:max_count]
142
143     return notes
144
145     def _snap_to_key(self, pitch):
146         if self.pitch_set:
147             mod = pitch % 12
148             pitch = (12 * (pitch // 12)) + min(
149                 self.pitch_set, key=lambda x: abs(x - mod)
150             )
151         return pitch
152
153     @lru_cache(None)
154     def _freq_to_pitch(self, freq):
155         for pitch, freq_range in self.notes.items():
156             # Find the freq's equivalence class, adding the amplitudes.
157             if freq_range[0] <= freq <= freq_range[2]:
158                 return self._snap_to_key(pitch) + self.transpose
159         raise RuntimeError("Unmappable frequency: {}".format(freq[0]))
160
161     def _reduce_freqs(self, freqs):
162         """
163         freqs is a list of amplitudes produced by _fft_to_frequencies().
164
165         Reduces the list of frequencies to a list of notes and their
166         respective volumes by determining what note each frequency
167         is closest to. It then reduces the list of amplitudes for each
168         note to a single amplitude by summing them together.
169         """
170
171         reduced_freqs = []
172         for freq in freqs:
173             reduced_freqs.append((self._freq_to_pitch(freq[0]), freq[1]))
174

```

```

175         return reduced_freqs
176
177     def _samples_to_freqs(self, samples):
178         amplitudes = numpy.fft.fft(samples)
179         freqs = []
180
181         for index in range(self.min_bin, self.max_bin):
182             # frequency, amplitude
183             freqs.append(
184                 [
185                     self.frequencies[index],
186                     numpy.sqrt(
187                         numpy.float_power(amplitudes[index].real, 2)
188                         + numpy.float_power(amplitudes[index].imag, 2)
189                     ),
190                 ]
191             )
192
193         # Transform the frequency info into midi compatible data.
194         return self._reduce_freqs(freqs)
195
196     def _block_to_notes(self, block):
197         channels = [[] for _ in range(self.info.channels)]
198         notes = [None for _ in range(self.info.channels)]
199
200         for sample in block:
201             for channel in range(self.info.channels):
202                 channels[channel].append(sample[channel])
203
204         for channel, samples in enumerate(channels):
205             freqs = self._samples_to_freqs(samples)
206             notes[channel] = self._freqs_to_midi(freqs)
207
208         return notes
209
210     def convert(self):
211         """
212         Performs the fft for each time step and transforms the result
213         into midi compatible data. This data is then passed to a
214         midi file writer.
215         """
216
217         logging.info(str(self.info))
218         logging.info("window: {} ms".format(self.time_window))
219         logging.info(

```

```

220         "frequencies: min = {} Hz, max = {} Hz".format(self.min_freq,
self.max_freq)
221     )
222
223     with midi_writer.MidiWriter(
224         outfile=self.outfile,
225         channels=self.info.channels,
226         time_window=self.time_window,
227         bpm=self.bpm,
228         condense=self.condense,
229         condense_max=self.condense_max,
230         max_note_length=self.max_note_length,
231     ) as writer:
232         for block in soundfile.blocks(
233             self.infile,
234             blocksize=self.block_size,
235             always_2d=True,
236         ):
237             if len(block) != self.block_size:
238                 filler = numpy.array(
239                     [
240                         numpy.array([0.0 for _ in range(self.info.channels)])
241                         for _ in range(self.block_size - len(block))
242                     ]
243                 )
244                 block = numpy.append(block, filler, axis=0)
245             notes = self._block_to_notes(block)
246             writer.add_notes(notes)
247             self._increment_progress()
248

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/init.py

1 |

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/notes.py

```

1  import numpy
2
3  def generate():
4      """
5      Generates a dict of midi note codes with their corresponding
6      frequency ranges.
7      """
8

```



```

9      # C0
10     base = [7.946362749, 8.1757989155, 8.4188780665]
11
12     # 12th root of 2
13     multiplier = numpy.float_power(2.0, 1.0 / 12)
14
15     notes = {0: base}
16     for i in range(1, 128):
17         mid = multiplier * notes[i - 1][1]
18         low = (mid + notes[i - 1][1]) / 2.0
19         high = (mid + (multiplier * mid)) / 2.0
20         notes.update({i: [low, mid, high]})
21
22     return notes
23

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/progress_bar.py

```

1  import time
2  import threading
3  import progressbar
4
5
6  class ProgressBar:
7      def __init__(self, current=0, total=0):
8          self.current = current
9          self.total = total
10         self.bar = progressbar.ProgressBar(max_value=self.total)
11
12     def update(self, current=0, total=0):
13         current = min(current, total)
14         self.bar.max_value = total
15         self.bar.update(current)
16

```

Keyboard_Pi_Conductor/Audio_Midi/audio_to_midi/main.py

```

1  #!/usr/bin/env python3
2
3  import argparse
4  import os
5  import sys
6  import logging
7
8  import converter, progress_bar

```

```

9
10
11 def _convert_beat_to_time(bpm, beat):
12     try:
13         parts = beat.split("/")
14         if len(parts) > 2:
15             raise Exception()
16
17         beat = [int(part) for part in parts]
18         fraction = beat[0] / beat[1]
19         bps = bpm / 60
20         ms_per_beat = bps * 1000
21         return fraction * ms_per_beat
22     except Exception:
23         raise RuntimeError("Invalid beat format: {}".format(beat))
24
25
26 def parse_args():
27     parser = argparse.ArgumentParser()
28     parser.add_argument("infile", help="The sound file to process.")
29     parser.add_argument(
30         "--output", "-o", help="The MIDI file to output. Default: <infile>.mid"
31     )
32     parser.add_argument(
33         "--time-window",
34         "-t",
35         default=5.0,
36         type=float,
37         help="The time span over which to compute the individual FFTs in
millisecons.",
38     )
39     parser.add_argument(
40         "--activation-level",
41         "-a",
42         default=0.0,
43         type=float,
44         help="The amplitude threshold for notes to be added to the MIDI file. Must
be between 0 and 1.",
45     )
46     parser.add_argument(
47         "--condense",
48         "-c",
49         action="store_true",
50         help="Combine contiguous notes at their average amplitude.",
51     )
52     parser.add_argument(

```

```

53         "--condense-max",
54         "-m",
55         action="store_true",
56         help="Write the maximum velocity for a condensed note segment rather than
the rolling average.",
57     )
58     parser.add_argument(
59         "--max-note-length",
60         "-M",
61         type=int,
62         default=0,
63         help="The max condensed note length in time window units.",
64     )
65     parser.add_argument(
66         "--single-note",
67         "-s",
68         action="store_true",
69         help="Only add the loudest note to the MIDI file for a given time window.",
70     )
71     parser.add_argument(
72         "--note-count",
73         "-C",
74         type=int,
75         default=0,
76         help="Only add the loudest n notes to the MIDI file for a given time
window.",
77     )
78     parser.add_argument(
79         "--bpm", "-b", type=int, help="Beats per minute. Defaults: 60", default=60
80     )
81     parser.add_argument(
82         "--beat",
83         "-B",
84         help="Time window in terms of beats (1/4, 1/8, etc.). Supercedes the time
window parameter.",
85     )
86     parser.add_argument(
87         "--transpose",
88         "-T",
89         type=int,
90         default=0,
91         help="Transpose the MIDI pitches by a constant offset.",
92     )
93     parser.add_argument(
94         "--pitch-set",
95         "-p",

```

```

96         type=int,
97         nargs="+",
98         default=[],
99         help="Map to a pitch set. Values must be in the range: [0, 11]. Ex: -p 0 2 4
100 5 7 9 11",
101     )
102     parser.add_argument(
103         "--pitch-range",
104         "-P",
105         nargs=2,
106         type=int,
107         help="The minimum and maximum allowed MIDI notes. These may be superseded by
108 the calculated FFT range.",
109     )
110     parser.add_argument(
111         "--no-progress", "-n", action="store_true", help="Don't print the progress
112 bar."
113     )
114     args = parser.parse_args()
115
116     args.output = (
117         "{}.mid".format(os.path.basename(args.infile))
118         if not args.output
119         else args.output
120     )
121
122     if args.single_note:
123         args.note_count = 1
124
125     if args.pitch_set:
126         for key in args.pitch_set:
127             if key not in range(12):
128                 raise RuntimeError("Key values must be in the range: [0, 12)")
129
130     if args.beat:
131         args.time_window = _convert_beat_to_time(args.bpm, args.beat)
132         print(args.time_window)
133
134     if args.pitch_range:
135         if args.pitch_range[0] > args.pitch_range[1]:
136             raise RuntimeError("Invalid pitch range: {}".format(args.pitch_range))
137
138     if args.condense_max:
139         args.condense = True
140
141     return args

```

```

139
140
141 def main():
142     try:
143         logging.basicConfig(level=logging.DEBUG, format="%(message)s")
144
145         args = parse_args()
146
147         process = converter.Converter(
148             infile=args.infile,
149             outfile=args.output,
150             time_window=args.time_window,
151             activation_level=args.activation_level,
152             condense=args.condense,
153             condense_max=args.condense_max,
154             max_note_length=args.max_note_length,
155             note_count=args.note_count,
156             transpose=args.transpose,
157             pitch_set=args.pitch_set,
158             pitch_range=args.pitch_range,
159             progress=None if args.no_progress else progress_bar.ProgressBar(),
160             bpm=args.bpm,
161         )
162         process.convert()
163     except KeyboardInterrupt:
164         sys.exit(1)
165     except Exception as e:
166         logging.exception(e)
167         sys.exit(1)
168
169
170 if __name__ == "__main__":
171     main()
172

```

Keyboard_Pi_Conductor/Audio_Midi/python_midi.py

```

1  # from midiutil.MidiFile import
2  import pyaudio
3  import wave
4
5  from Audio_Midi.audio_to_midi.converter import Converter
6  from threading import Thread
7
8  class AudioRecorder:

```

```

9     def __init__(self, save_path):
10         self.recording = False
11
12         self.audio = pyaudio.PyAudio()
13
14
15         self.usb_idx = 0
16
17         # Outputs index of every audio-capable device on the Pi
18         for ii in range(self.audio.get_device_count()):
19             if "USB PnP Sound" in
self.audio.get_device_info_by_index(ii).get('name'):
20                 self.usb_idx = ii
21                 print(f"idx: {ii}, name:
{self.audio.get_device_info_by_index(ii).get('name')}")
22
23         # Set index of usb microphone
24
25         # Set parameters for audio recording
26         self.form_1 = pyaudio.paInt16 # 16-bit resolution
27         self.chans = 1 # 1 channel
28         self.samp_rate = 44100 # 44.1kHz sampling rate
29         self.chunk = 4096 # 2^12 samples for buffer
30         self.save_path = save_path # name of .wav file
31         self.frames = []
32
33
34     def start_recording(self):
35         self.stream = self.audio.open(
36             format=self.form_1,
37             rate=self.samp_rate,
38             channels=self.chans,
39             input_device_index=self.usb_idx,
40             input=True,
41             frames_per_buffer=self.chunk
42         )
43         print("recording")
44
45         self.recording = True
46         self.record_thread = Thread(target=self.record)
47         self.record_thread.start()
48
49     def record(self):
50         while self.recording:
51             data = self.stream.read(self.chunk)
52             self.frames.append(data)

```

```

53
54     def stop_recording(self):
55         self.recording = False
56
57         self.record_thread.join()
58         print('Recording Stopped, closing streams')
59
60         self.stream.stop_stream()
61         self.stream.close()
62         self.audio.terminate()
63
64         print("finished recording")
65
66         # save the audio frames as .wav file
67         wavefile = wave.open(self.save_path, 'wb')
68         wavefile.setnchannels(self.chans)
69         wavefile.setsampwidth(self.audio.get_sample_size(self.form_1))
70         wavefile.setframerate(self.samp_rate)
71         wavefile.writeframes(b''.join(self.frames))
72         wavefile.close()
73
74
75     def start_recording(audio, form_1, chans, samp_rate, chunk, usb_idx):
76         # create pyaudio stream
77         stream = audio.open(format=form_1, rate=samp_rate, channels=chans,
78                             input_device_index=usb_idx, input=True,
79                             frames_per_buffer=chunk)
80         print("recording")
81         return stream
82
83
84     def record_loop(stream, samp_rate, chunk, record_secs):
85         # loop through stream and append audio chunks to frame array
86         frames = []
87         for ii in range(int((samp_rate/chunk)*record_secs)):
88             data = stream.read(chunk)
89             frames.append(data)
90         return frames
91
92
93     def stop_recording(audio, stream, filename, chans, samp_rate, frames, form_1):
94         # stop the stream, close it, and terminate the pyaudio instantiation
95         stream.stop_stream()
96         stream.close()
97         audio.terminate()
98

```

```

99     print("finished recording")
100
101     # save the audio frames as .wav file
102     wavefile = wave.open(filename, 'wb')
103     wavefile.setnchannels(chans)
104     wavefile.setsampwidth(audio.get_sample_size(form_1))
105     wavefile.setframerate(samp_rate)
106     wavefile.writeframes(b''.join(frames))
107     wavefile.close()
108
109
110 def audio_to_midi_conv(input_filename, output_filename, time_window=5.0,
111     activ_level=0.0, note_count=0, bpm=60):
112     process = Converter(
113         infile=input_filename,
114         outfile=output_filename,
115         time_window=time_window,
116         activation_level=activ_level,
117         condense=None,
118         condense_max=False,
119         max_note_length=0,
120         note_count=note_count,
121         bpm=bpm,
122     )
123     process.convert()
124
125 def record_audio(audio_length, save_path):
126     audio = pyaudio.PyAudio()
127
128     # Outputs index of every audio-capable device on the Pi
129     for ii in range(audio.get_device_count()):
130         print(f"idx: {ii}, name: {audio.get_device_info_by_index(ii).get('name')}")
131
132     # Set index of usb microphone
133     usb_idx = 1
134
135     # Set parameters for audio recording
136     form_1 = pyaudio.paInt16 # 16-bit resolution
137     chans = 1 # 1 channel
138     samp_rate = 44100 # 44.1kHz sampling rate
139     chunk = 4096 # 2^12 samples for buffer
140     wav_output_filename = save_path # name of .wav file
141     frames = []
142
143     stream = start_recording(audio, form_1, chans, samp_rate, chunk, usb_idx)

```



```

144     frames = record_loop(stream, samp_rate, chunk, audio_length)
145     stop_recording(audio, stream, wav_output_filename,
146                     chans, samp_rate, frames, form_1)
147
148
149 if __name__ == "__main__":
150     # record_audio(10, 'test_recorder.wav')
151     recorder = AudioRecorder(save_path='test_recorder.wav')
152
153     print('Hit Enter to Start Recording')
154     input()
155     recorder.start_recording()
156
157     print('Hit Enter to Stop Recording')
158     input()
159     recorder.stop_recording()
160

```

Keyboard_Pi_Conductor/metadata.py

```

1
2 RAW_AUDIO_PATH = 'out/recorded_audio.wav'
3 RECORDED_VIDEO_PATH = 'out/conducting_video.mp4'
4 RAW_MIDI_PATH = 'out/raw_midi.mid'
5 MODIFIED_MIDI_PATH = 'out/modified_midi.mid'
6 MODIFIED_AUDIO_PATH = 'out/modified_audio.wav'
7 MIDI_BEAT = '1/4'
8 MIDI_BPM = 90
9
10
11

```

Keyboard_Pi_Conductor/integrate.py

```

1 import pygame
2 from pygame.time import Clock
3 import threading
4 from PythonPiano.key_midi import AudioRecorder
5 from Gesture_IR.record_video import VideoRecorder
6 from Gesture_IR.gesture_ir import gesture_ir_main
7 from IR_Midi.modify_input import modify_volume, convert_midi_to_audio,
  play_audio_file
8
9 from GUI import init_system, GUI
10 from metadata import *

```

```

11 import time
12 from metronome import Metronome
13
14
15 current_frame = None
16
17 class DataFrame:
18     def __init__(self):
19         self.audio_recorder = AudioRecorder(save_path=RAW_MIDI_PATH)
20         self.bpm = MIDI_BPM
21         self.video_recorder = VideoRecorder(RECORDED_VIDEO_PATH, bpm=self.bpm)
22
23 df = DataFrame()
24
25 def start_audio_playback_recording_and_camera(frame, audio_file, video_path):
26     df.video_recorder.initialize(df.bpm*2)
27     df.video_recorder.spawn_camera()
28     time.sleep(2.0)
29
30     playback_thread = threading.Thread(target=play_audio_file, args=(audio_file,))
31
32     frame.idle_components = []
33     frame.add_text(
34         text="START CONDUCTING TO MUSIC",
35         center_frac_x=0.5,
36         center_frac_y=0.25
37     )
38     frame.event_components = []
39     frame.reset()
40     frame.render()
41
42     playback_thread.start()
43     df.video_recorder.start_recording()
44
45     playback_thread.join()
46     df.video_recorder.stop_recording()
47
48     frame.add_text(
49         text="MUSIC FINISHED",
50         center_frac_x=0.5,
51         center_frac_y=0.5
52     )
53     frame.event_components = []
54     frame.reset()
55     frame.render()
56

```

```

57
58 def start_production(display_message):
59     # Replace with the path to your audio file
60     audio_path = 'testrun1.wav'
61     video_path = 'testrun1_conduct.mp4'
62     midi_path = "mid.midi"
63     out_midi_path = 'testrun1_final_midi.mid'
64     out_audio_path = 'testrun1_final_audio.wav'
65     df.bpm = 90
66     beat = '1/4'
67     single_note = True
68
69
70     # print("Hit enter to start recording audio: ")
71     # input()
72     # record_audio(20, save_path=audio_path)
73
74     # print("Hit ENTER to convert the audio to midi file")
75     # input()
76     # audio_to_midi_conv(audio_path, midi_path, beat=beat, df.bpm=df.bpm)
77
78     exit()
79
80     print("Hit ENTER to record conducting video")
81     input()
82     start_audio_playback_recording_and_camera(
83         audio_path, video_path
84     )
85
86     display_message("Start production")
87
88     print('Hit ENTER to convert gesture video to IR')
89     input()
90     # Convert gesture to IR
91     ir_data = gesture_ir_main(video_path)
92
93     # Modify MIDI file based on IR and convert to audio
94     print('Hit ENTER to convert video IR to dynamics: ')
95     input()
96     volume_changes = ir_data
97     modify_volume(midi_path, volume_changes, out_midi_path, bpm=df.bpm)
98
99     print('Hit ENTER to convert modified midi file to audio file')
100    input()
101    convert_midi_to_audio(out_midi_path, out_audio_path)
102

```

```

103     display_message("Production finished, let's see the result")
104
105
106 def show_result():
107     # Visualize MIDI files and play back the modified audio
108     original_midi = "converted_midi.mid"
109     modified_midi = "output_file.mid"
110     # visualize_notes(original_midi, modified_midi)
111
112     play_audio_file("output_audio.wav")
113
114
115 def display_message(message):
116     global current_message
117     current_message = message
118
119 def switch_frame(switch_frame):
120     print('Switching Frame')
121     global current_frame
122     current_frame = switch_frame
123
124 def start_recording_audio(frame, next_frame):
125     if frame.get_text()[0] != '':
126         df.bpm = int(frame.get_text()[0])
127         metronome.reset_fps(df.bpm/30)
128         print('recording with df.bpm: ', df.bpm)
129
130         df.audio_recorder.initialize(bpm=df.bpm)
131         switch_frame(next_frame)
132
133 def stop_recording_audio(next_frame):
134     print('clicked stop recording audio')
135     df.audio_recorder.save_recording()
136     switch_frame(next_frame)
137
138 def save_recording_continue_cb(next_frame):
139     convert_midi_to_audio(RAW_MIDI_PATH, RAW_AUDIO_PATH)
140     switch_frame(next_frame)
141
142 def start_conducting_cb(frame, next_frame):
143     frame.add_text(
144         text="START CONDUCTING",
145         center_frac_x=0.5,
146         center_frac_y=0.25
147     )
148

```

```
149     frame.reset()
150     frame.render()
151
152     start_audio_playback_recording_and_camera(frame, RAW_AUDIO_PATH,
RECORDED_VIDEO_PATH)
153
154     frame.idle_components = []
155     frame.add_text(
156         text="RECORDING DONE",
157         center_frac_x=0.5,
158         center_frac_y=0.25
159     )
160     frame.reset()
161     frame.render()
162
163     time.sleep(1)
164     frame.idle_components = []
165     frame.add_button(
166         "Start Conducting",
167         0.4, 0.0625, 0.3, 0.4,
168         callback=lambda: start_conducting_cb(start_record_video_frame,
combine_frame)
169     )
170
171     switch_frame(next_frame)
172
173 def generate_result_cb(frame, next_frame):
174     txt, button_rect, cb = frame.event_components[0]
175     frame.event_components[0] = 'Converting...', button_rect, None
176     frame.reset()
177     frame.render()
178
179     # Convert gesture to IR
180     ir_info, ir_data = gesture_ir_main(RECORDED_VIDEO_PATH)
181
182     # Modify MIDI file based on IR and convert to audio
183     volume_changes = ir_data
184     modify_volume(RAW_MIDI_PATH, volume_changes, MODIFIED_MIDI_PATH, bpm=df.bpm)
185
186     convert_midi_to_audio(MODIFIED_MIDI_PATH, MODIFIED_AUDIO_PATH)
187
188     frame.event_components[0] = txt, button_rect, cb
189     frame.reset()
190     frame.render()
191
192     switch_frame(next_frame)
```

```
193
194 def play_result_cb(frame, button_idx):
195     txt, button_rect, cb = frame.event_components[button_idx]
196     frame.event_components[button_idx] = 'Playing Audio...', button_rect, cb
197     frame.reset()
198     frame.render()
199
200     play_audio_file(MODIFIED_AUDIO_PATH)
201
202     frame.event_components[button_idx] = txt, button_rect, cb
203     frame.reset()
204     frame.render()
205
206 def start_over_cb():
207     start_record_video_frame = GUI(window_width, window_height)
208     start_record_video_frame.add_button(
209         "Start Conducting",
210         0.4, 0.0625, 0.3, 0.4,
211         callback=lambda: start_conducting_cb(start_record_video_frame,
combine_frame)
212     )
213
214     switch_frame(start_frame)
215
216 def re_conduct_cb():
217     start_record_video_frame = GUI(window_width, window_height)
218     start_record_video_frame.add_button(
219         "Start Conducting",
220         0.4, 0.0625, 0.3, 0.4,
221         callback=lambda: start_conducting_cb(start_record_video_frame,
combine_frame)
222     )
223
224     switch_frame(start_record_video_frame)
225
226
227 init_system()
228
229 window_width = 800
230 window_height = 600
231
232 # Combine done, playback results:
233 playback_frame = GUI(window_width, window_height)
234 play_idx = playback_frame.add_button(
235     "Play Result",
236     0.5, 0.0625, 0.25, 0.4,
```

```

237     callback=lambda: play_result_cb(playback_frame, play_idx)
238 )
239
240 playback_frame.add_button(
241     "Produce Another Audio",
242     0.5, 0.0625, 0.25, 0.6,
243     callback=lambda: start_over_cb()
244 )
245
246 playback_frame.add_button(
247     "Re-Record Conducting Video",
248     0.5, 0.0625, 0.25, 0.8,
249     callback=lambda: re_conduct_cb()
250 )
251
252 # recording stops, click button to combine
253 combine_frame = GUI(window_width, window_height)
254 combine_frame.add_button(
255     "Combine Video Dynamics with Audio",
256     0.7, 0.0625, 0.15, 0.4,
257     callback=lambda: generate_result_cb(combine_frame, playback_frame)
258 )
259
260
261 # frame to start recording video then convert to intermediate representation
262 start_record_video_frame = GUI(window_width, window_height)
263 start_record_video_frame.add_button(
264     "Start Conducting",
265     0.4, 0.0625, 0.3, 0.4,
266     callback=lambda: start_conducting_cb(start_record_video_frame, combine_frame)
267 )
268
269 # Giving option to re-record audio or proceed:
270 re_record_audio_frame = GUI(window_width, window_height)
271 go_conducting_button = re_record_audio_frame.add_button(
272     "Continue",
273     0.4, 0.0625, 0.3, 0.4,
274     callback=lambda: save_recording_continue_cb(start_record_video_frame)
275 )
276
277 re_record_button = re_record_audio_frame.add_button(
278     "Re-Record Audio",
279     0.4, 0.0625, 0.3, 0.6,
280     callback=lambda: switch_frame(record_audio_frame)
281 )
282

```

```

283 # frame when audio recording is in progress and waiting to stop
284 audio_recording_frame = GUI(window_width, window_height)
285 record_audio_button = audio_recording_frame.add_button(
286     "Stop Recording",
287     0.4, 0.0625, 0.3, 0.8,
288     callback=lambda: stop_recording_audio(re_record_audio_frame)
289 )
290
291 audio_recording_frame.add_text(
292     text="Hit ENTER to stop and save recording",
293     center_frac_x=0.5,
294     center_frac_y = 0.25
295 )
296 audio_recording_frame.add_circle(radius=50, center_frac_x=0.5, center_frac_y=0.5)
297 metronome = Metronome(frame=audio_recording_frame, fps=2)
298
299 audio_recording_frame.add_per_frame(metronome.switch)
300 audio_recording_frame.add_event_processor(df.audio_recorder.process_event)
301
302
303 # frame for starting to record audio
304 record_audio_frame = GUI(window_width, window_height)
305 record_audio_button = record_audio_frame.add_button(
306     "Start Recording",
307     0.4, 0.0625, 0.3, 0.4,
308     callback=lambda: start_recording_audio(record_audio_frame,
309     audio_recording_frame)
310 )
311 record_audio_frame.add_textbox(
312     0.4, 0.0625, 0.3, 0.7
313 )
314
315 record_audio_frame.add_text(
316     text="Click button to start recording audio",
317     center_frac_x=0.5,
318     center_frac_y=0.25
319 )
320
321 # starting screen
322 start_frame = GUI(window_width, window_height)
323 start_frame.add_button(
324     "Start Production",
325     0.4, 0.0625, 0.3, 0.4,
326     callback=lambda: switch_frame(record_audio_frame)
327 )

```



```

328 clk = Clock()
329 running = True
330 current_frame = start_frame
331 while running:
332
333     for event in pygame.event.get():
334         exit = current_frame.event_trigger(event)
335         if exit:
336             running = False
337
338     current_frame.reset()
339     current_frame.render()
340     clk.tick(current_frame.fps)
341
342 pygame.quit()
343
344

```

Keyboard_Pi_Conductor/metronome.py

```

1  white = (255, 255, 255)
2  black = (0, 0, 0)
3  red = (255, 0, 0)
4
5  import time
6  from pygame import mixer
7
8  class Metronome:
9      def __init__(self, frame, fps):
10         self.frame = frame
11         self.bit = False
12
13         self.prev_switch = None
14         self.fps = fps
15         self.time_step = 1 / fps
16
17         self.sound =
18         mixer.Sound(f'/home/pi/Documents/ECE5725_final_proj/PythonPiano/Synth_Block_F_lo.wav'
19         )
20
21         def reset_fps(self, fps):
22             self.fps = fps
23             self.time_step = 1/fps
24
25         def switch(self):

```

```

24
25         if self.prev_switch is not None and time.time() - self.prev_switch <
self.time_step:
26             return
27
28         self.prev_switch = time.time()
29         self.bit = not self.bit
30         center, radius, color = self.frame.circles[0]
31
32
33         if self.bit:
34             self.frame.circles[0] = (center, radius, red)
35             self.sound.play()
36         else:
37             self.frame.circles[0] = (center, radius, white)
38

```

Keyboard_Pi_Conductor/mid_to_wav.py

```

1  from IR_Midi.modify_input import convert_midi_to_audio
2  midi_path = 'out/raw_midi.mid'
3  audio_path = 'recorded_audio.wav'
4  convert_midi_to_audio(midi_path, audio_path)
5

```

Keyboard_Pi_Conductor/Gesture_IR/init.py

```

1

```

Keyboard_Pi_Conductor/Gesture_IR/record_video.py

```

1  import cv2
2  import numpy as np
3  from threading import Thread
4  import time
5  # This is a script that records a video using the camera
6
7
8  class VideoRecorder:
9      def __init__(self, video_path, bpm):
10
11         self.video_path = video_path
12         self.initialize
13

```

```

14     def initialize(self, bpm):
15         # Create a VideoCapture object
16         self.camera = cv2.VideoCapture(0)
17         self.camera_open = False
18
19         # Check if the camera is opened
20         if not self.camera.isOpened():
21             print("Cannot open camera")
22             exit()
23
24         self.bpm = bpm
25         self.fps = self.bpm / 60
26         # Define the codec and create a VideoWriter object
27         fourcc = cv2.VideoWriter_fourcc(*'mp4v')
28         self.video_writer = cv2.VideoWriter(self.video_path, fourcc, self.fps, (640,
480))
29
30     def spawn_camera(self):
31         self.recording = False
32         self.camera_open = True
33         self.record_thread = Thread(target=self.record)
34         self.record_thread.start()
35         self.timestamp = time.time()
36
37     def start_recording(self):
38         while not self.camera.isOpened():
39             time.sleep(0.01)
40         self.recording = True
41         self.timestamp = time.time()
42
43     def stop_recording(self):
44         # Release everything if job is finished
45
46         self.recording = False
47         self.camera_open = False
48
49         self.record_thread.join()
50         self.camera.release()
51         self.video_writer.release()
52         cv2.destroyAllWindows()
53
54     def record(self):
55         frames = []
56
57         timestep = 1 / self.fps
58         while self.camera_open and self.camera.isOpened():

```

```

59         ret, frame = self.camera.read()
60
61         if not ret:
62             print("Can't receive frame (stream end?). Exiting ...")
63             break
64
65         # Flip the frame horizontally
66         # frame = cv2.flip(frame, 1)
67         frame = cv2.flip(frame, 0)
68
69         if self.recording and time.time() - self.timestamp > timestep:
70             self.timestamp = time.time()
71             print(f'Appending Frame at {self.timestamp}')
72             frames.append(frame)
73
74         # Display the resulting frame
75         cv2.imshow('frame', frame)
76
77         k = cv2.waitKey(1) & 0xFF
78         if cv2.waitKey(1) & 0xFF == ord('q'):
79             break
80
81         for f in frames:
82             self.video_writer.write(f)
83
84
85 if __name__ == "__main__":
86     recorder = VideoRecorder(video_path='output.mp4', bpm=90)
87     input()
88     recorder.spawn_camera()
89     print('camera spawned')
90     input()
91     recorder.start_recording()
92     print('starting recording')
93     input()
94     recorder.stop_recording()
95

```

Keyboard_Pi_Conductor/Gesture_IR/gesture_ir.py

```

1  import cv2
2  import copy
3  import mediapipe as mp
4  import numpy as np
5  from tqdm import tqdm

```

```

6 import matplotlib.pyplot as plt
7
8 from argparse import ArgumentParser
9 def calc_bounding_rect(image, landmarks):
10     image_width, image_height = image.shape[1], image.shape[0]
11
12     landmark_array = np.empty((0, 2), int)
13
14     for _, landmark in enumerate(landmarks.landmark):
15         landmark_x = min(int(landmark.x * image_width), image_width - 1)
16         landmark_y = min(int(landmark.y * image_height), image_height - 1)
17
18         landmark_point = [np.array((landmark_x, landmark_y))]
19
20         landmark_array = np.append(landmark_array, landmark_point, axis=0)
21
22     x, y, w, h = cv2.boundingRect(landmark_array)
23
24     return [x, y, x + w, y + h]
25
26 def get_bounding_box(image, results):
27     for hand_landmarks, handedness in zip(results.multi_hand_landmarks,
28                                           results.multi_handedness):
29         # Bounding box calculation
30         brect = calc_bounding_rect(image, hand_landmarks)
31         return brect
32
33
34 def video_to_box_info(vid_path: str) -> list:
35
36     video = cv2.VideoCapture(vid_path)
37
38     frame_count = video.get(cv2.CAP_PROP_FRAME_COUNT)
39     fps = video.get(cv2.CAP_PROP_FPS)
40     duration = frame_count/fps
41
42     info = {'fps': fps, 'duration': duration, 'frame_count': frame_count}
43     # Check if video opened successfully
44     if not video.isOpened():
45         print("Error opening video file")
46
47     mp_hands = mp.solutions.hands
48     hands = mp_hands.Hands(
49         static_image_mode=False,
50         max_num_hands=1,
51         min_detection_confidence=0.7,

```

```

52         min_tracking_confidence=0.5,
53     )
54
55
56     progress_bar = tqdm(total=frame_count, desc="Processing frames", ncols=0)
57
58     bounding_boxes = []
59     # Read until video is completed
60     curr_frame = 1
61     while video.isOpened():
62
63         ret, image = video.read()
64         if not ret:
65             break
66         image = cv2.flip(image, 1) # Mirror display
67
68         # Detection implementation
69         #####
70         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
71
72         image.flags.writeable = False
73         results = hands.process(image)
74
75         image.flags.writeable = True
76
77         # #####
78         if results.multi_hand_landmarks is not None:
79             # brect = get_bounding_box(image, results)
80             thumb = results.multi_hand_landmarks[0].landmark[4]
81             middle = results.multi_hand_landmarks[0].landmark[12]
82             brect = (thumb.x - middle.x) ** 2 + (thumb.y - middle.y)**2 + (thumb.z -
83             middle.z) **2
84             brect = np.sqrt(brect)
85
86             timestamp = curr_frame / fps
87
88             bounding_boxes.append((timestamp, brect))
89             curr_frame += 1
90             progress_bar.update(1)
91
92     progress_bar.close()
93     # When everything done, release the video capture object
94     video.release()
95
96     # Closes all the frames

```

```

96     cv2.destroyAllWindows()
97
98     return info, bounding_boxes
99
100
101 def converter(timestamp, open_scale):
102     assert len(timestamp) == len(open_scale)
103
104     output = []
105     for i in range(len(open_scale) - 1):
106         output.append((timestamp[i], timestamp[i+1], (open_scale[i+1] -
open_scale[i])*100))
107
108     return output
109
110 def gesture_ir_main(video_path):
111     info, bounding_boxes = video_to_box_info(video_path)
112     diag_len = bounding_boxes
113     # diag_len = [ (time, np.sqrt((box[2] - box[0])**2 + (box[3] - box[1])**2)) for
time, box in bounding_boxes]
114
115
116     timestamps, diag_len = zip(*diag_len)
117     plt.plot(timestamps, diag_len)
118     plt.savefig('plot.png')
119
120     box_sizes = np.array(diag_len)
121     max_expand, min_expand = box_sizes.max(), box_sizes.min()
122
123     open_scale = (box_sizes - min_expand) / (max_expand - min_expand)
124     ir = converter(timestamps, open_scale)
125     return info, ir
126
127 if __name__ == "__main__":
128
129     parser = ArgumentParser()
130     parser.add_argument('--vid_path', type=str)
131     args = parser.parse_args()
132
133     print(gesture_ir_main(args.vid_path))
134

```

Keyboard_Pi_Conductor/GUI.py

```

1 import pygame

```

```

2
3 white = (255, 255, 255)
4 black = (0, 0, 0)
5 red = (255, 0, 0)
6
7 def init_system():
8     pygame.init()
9
10 class GUI:
11     def __init__(self, width, height, fps=30):
12
13         self.window_width = width
14         self.window_height = height
15         self.window = pygame.display.set_mode((self.window_width,
self.window_height))
16         pygame.display.set_caption("Music Production")
17
18         self.event_components = []
19         self.event_processors = []
20         self.idle_components = []
21         self.circles = []
22         self.per_frame_call = []
23         self.text_boxes = []
24         self.active_text_box = -1
25         self.running = False
26
27         self.font = pygame.font.Font(None, 36)
28
29         self.fps = fps
30
31     def add_button(self, text, width_frac, height_frac, center_frac_x,
center_frac_y, callback=None):
32         rect = pygame.Rect(
33             center_frac_x * self.window_width,
34             center_frac_y * self.window_height,
35             width_frac * self.window_width,
36             height_frac * self.window_height
37         )
38
39         self.event_components.append((text, rect, callback))
40         return len(self.event_components) - 1
41
42
43     def event_trigger(self, event):
44         if event.type == pygame.QUIT:
45             self.running = False

```



```

46         return True
47
48     for text, component, cb in self.event_components:
49         if not event.type == pygame.MOUSEBUTTONDOWN:
50             continue
51         if component.collidepoint(event.pos) and cb is not None:
52             cb()
53
54     self.active_text_box = -1
55     for i in range(len(self.text_boxes)):
56         text, component, active = self.text_boxes[i]
57         if not event.type == pygame.MOUSEBUTTONDOWN:
58             continue
59
60         self.text_boxes[i] = text, component, component.collidepoint(event.pos)
61         if component.collidepoint(event.pos):
62             self.active_text_box = i
63
64     for i in range(len(self.text_boxes)):
65         text, component, active = self.text_boxes[i]
66         if not active:
67             continue
68         if event.type == pygame.KEYDOWN:
69             if event.key == pygame.K_RETURN:
70                 active = False
71                 self.active_text_box = -1
72             elif event.key == pygame.K_BACKSPACE:
73                 text = text[:-1]
74             else:
75                 text += event.unicode
76             self.text_boxes[i] = text, component, active
77
78
79     for processor in self.event_processors:
80         processor(event)
81
82     return False
83
84
85     def add_text(self, text, center_frac_x, center_frac_y):
86         text = self.font.render(text, True, black)
87         text_rect = text.get_rect(center=(center_frac_x * self.window_width,
88 center_frac_y * self.window_height))
89
90         self.idle_components.append((text, text_rect))

```

```

91     def add_circle(self, radius, center_frac_x, center_frac_y):
92         self.circles.append(((center_frac_x * self.window_width, center_frac_y *
self.window_height), radius, black))
93
94     def add_textbox(self, width_frac, height_frac, center_frac_x, center_frac_y):
95         rect = pygame.Rect(
96             center_frac_x * self.window_width,
97             center_frac_y * self.window_height,
98             width_frac * self.window_width,
99             height_frac * self.window_height
100         )
101
102         self.text_boxes.append(('', rect, False))
103
104     def get_text(self):
105         ret = []
106         for text, _, _ in self.text_boxes:
107             ret.append(text)
108
109         return ret
110
111     def reset(self):
112         self.window.fill(white)
113
114
115     def start(self):
116         self.running = True
117
118     def add_per_frame(self, func):
119         self.per_frame_call.append(func)
120
121     def add_event_processor(self, func):
122         self.event_processors.append(func)
123
124     def render(self):
125
126         for func in self.per_frame_call:
127             func()
128
129         for text, component, cb in self.event_components:
130             pygame.draw.rect(self.window, black, component, 2)
131             render_text = self.font.render(text, True, black)
132             text_rect = render_text.get_rect(center=component.center)
133             self.window.blit(render_text, text_rect)
134
135

```

```

136         for text, component, active in self.text_boxes:
137             color = (red if active else black)
138             pygame.draw.rect(self.window, color, component, 3)
139             if text == '' and not active:
140                 text = 'Enter Text Here'
141             render_text = self.font.render(text, True, black)
142             text_rect = render_text.get_rect(center=component.center)
143             self.window.blit(render_text, text_rect)
144
145         for text, text_rect in self.idle_components:
146             self.window.blit(text, text_rect)
147
148         for center, radius, color in self.circles:
149             pygame.draw.circle(self.window, color, center, radius, width=0)
150
151
152         pygame.display.flip()
153
154
155     # pygame.init()
156     # gui = GUI(width=800, height=600)
157
158     # gui.add_button(
159     #     "Start Production",
160     #     0.4, 0.0625, 0.3, 0.4,
161     #     callback=lambda: print("START PRODUCTION")
162     # )
163
164     # gui.add_button(
165     #     "See Result",
166     #     0.4, 0.0625, 0.3, 0.65,
167     #     callback=lambda: print("See Results")
168     # )
169
170     # gui.add_text(
171     #     text="Are you ready to produce your music?",
172     #     center_frac_x=0.5,
173     #     center_frac_y = 0.25
174     # )
175     # running = True
176
177
178     # gui2 = GUI(width=800, height=600)
179
180     # gui2.add_button(
181     #     "Start Production 2",

```

```

182 #      0.4, 0.0625, 0.3, 0.4,
183 #      callback=lambda: print("START PRODUCTION 2")
184 # )
185
186 # gui2.add_button(
187 #      "See Result 2",
188 #      0.4, 0.0625, 0.3, 0.65,
189 #      callback=lambda: print("See Results 2")
190 # )
191
192 # gui2.add_text(
193 #      text="Are you ready to produce your music? 2",
194 #      center_frac_x=0.5,
195 #      center_frac_y = 0.25
196 # )
197 # running = True
198
199
200 # i = 0
201 # while running:
202 #     if i < 100:
203 #         g = gui
204 #     else:
205 #         g = gui2
206
207 #     for event in pygame.event.get():
208 #         g.event_trigger(event)
209
210 #     g.reset()
211 #     g.render()
212 #     i += 1
213
214
215
216
217
218
219
220

```

Keyboard_Pi_Conductor/get_code.py

```

1 import os
2 from pathlib import Path
3

```

```

4
5 def write_code(file):
6     with open('code.md', 'a') as f:
7         # remove root path from file string
8         f.write(f'__{file}__\n\n')
9         f.write('```\n')
10        with open(file, 'r') as code_file:
11            lines = code_file.readlines()
12            f.writelines(lines)
13            f.write('\n```')
14            f.write('\n\n')
15
16 def main(path):
17     for f in os.listdir(path):
18         f = os.path.join(path, f)
19         if os.path.isdir(f):
20             main(f)
21         elif f.endswith('.py'):
22             write_code(f)
23
24 if __name__ == "__main__":
25     root_path = Path(__file__).parent
26     print(f'{root_path=}')
27     main(root_path)
28
29
30

```

Keyboard_Pi_Conductor/IR_Midi/visualize_note.py

```

1  from music21 import converter, stream, tempo, meter, chord, note
2
3  import midi2audio
4  import pygame
5  from IR_Midi.modify_input import modify_volume
6  from visual_midi import Plotter, Preset
7  from pretty_midi import PrettyMIDI
8
9
10 def get_note_info(midi_file):
11     score = converter.parse(midi_file)
12     notes = []
13     for element in score.flat.notesAndRests:
14         if isinstance(element, stream.Voice):
15             element = element.flat.notesAndRests

```

```

16         if isinstance(element, chord.Chord):
17             pitches = '.'.join(n.nameWithOctave for n in element.pitches)
18             volume = element.volume.velocity
19         elif isinstance(element, note.Note):
20             pitches = element.pitch.nameWithOctave
21             volume = element.volume.velocity
22         else:
23             pitches = ''
24             volume = 0
25
26         notes.append({
27             "pitch": pitches,
28             "start": element.offset,
29             "end": element.offset + element.duration.quarterLength,
30             "volume": volume
31         })
32
33     return notes
34
35
36 def draw_note_volumes(screen, notes, y_offset, color):
37     for note in notes:
38         start_x = int(note["start"] * 50)
39         end_x = int(note["end"] * 50)
40         volume = int(note["volume"] / 127 * screen.get_height() / 2)
41         pygame.draw.rect(screen, color, (start_x, y_offset -
42             volume, end_x - start_x, volume))
43         pygame.draw.rect(screen, (0, 0, 0), (start_x,
44             y_offset - volume, end_x - start_x, volume), 1)
45
46         font = pygame.font.Font(None, 24)
47         pitch_text = font.render(note["pitch"], True, (0, 0, 0))
48         volume_text = font.render(str(note["volume"]), True, (0, 0, 0))
49         screen.blit(pitch_text, (start_x, y_offset - volume - 20))
50         screen.blit(volume_text, (start_x, y_offset - volume - 40))
51
52
53 def draw_modified_notes(screen, original_notes, modified_notes):
54     for i in range(len(original_notes)):
55         if original_notes[i]["volume"] != modified_notes[i]["volume"]:
56             start_x = int(modified_notes[i]["start"] * 50)
57             end_x = int(modified_notes[i]["end"] * 50)
58             volume = int(modified_notes[i]["volume"] /
59                 127 * screen.get_height() / 2)
60             pygame.draw.rect(screen, (255, 0, 0), (start_x,
61                 screen.get_height() - volume, end_x - start_x, volume))

```

```

62         pygame.draw.rect(screen, (0, 0, 0), (start_x, screen.get_height(
63             ) - volume, end_x - start_x, volume), 1)
64
65         font = pygame.font.Font(None, 24)
66         pitch_text = font.render(
67             modified_notes[i]["pitch"], True, (0, 0, 0))
68         volume_text = font.render(
69             str(modified_notes[i]["volume"]), True, (0, 0, 0))
70         screen.blit(
71             pitch_text, (start_x, screen.get_height() - volume - 20))
72         screen.blit(
73             volume_text, (start_x, screen.get_height() - volume - 40))
74
75
76 def visualize_notes(original_file, modified_file):
77     original_notes = get_note_info(original_file)
78     modified_notes = get_note_info(modified_file)
79
80     pygame.init()
81     screen_width, screen_height = 800, 600
82     screen = pygame.display.set_mode((screen_width, screen_height))
83     pygame.display.set_caption("MIDI Note Volume Visualization")
84
85     running = True
86     while running:
87         for event in pygame.event.get():
88             if event.type == pygame.QUIT:
89                 running = False
90
91         screen.fill((255, 255, 255))
92
93         draw_note_volumes(screen, original_notes,
94             screen_height // 2, (0, 0, 255))
95         draw_note_volumes(screen, modified_notes, screen_height, (0, 0, 255))
96         draw_modified_notes(screen, original_notes, modified_notes)
97
98         pygame.display.flip()
99
100     pygame.quit()
101
102
103 def visualize_midi(original_file, modified_file):
104     original_pm = PrettyMIDI(original_file)
105     modified_pm = PrettyMIDI(modified_file)
106
107     preset = Preset(plot_width=800, plot_height=400, row_height=50)

```

```

108     plotter = Plotter(preset, plot_max_length_bar=16)
109
110     plotter.show(original_pm, "/tmp/original_midi.html")
111     plotter.show(modified_pm, "/tmp/modified_midi.html")
112
113
114 def test_visual():
115     volume_changes = [
116         (3.0, 8.0, 130), # Louder at 7s, lasting for 3s
117         (10.0, 380.0, -100) # Quieter at 15s, lasting for 3s
118     ]
119
120     modified_score = modify_volume('input.mid', volume_changes)
121     modified_score.write('midi', 'output_file.mid')
122
123     visualize_notes('input.mid', 'output_file.mid')
124     visualize_midi('input.mid', 'output_file.mid')
125
126
127 if __name__ == "__main__":
128     test_visual()
129

```

Keyboard_Pi_Conductor/IR_Midi/merge_midi.py

```

1  from music21 import converter, instrument, note, chord, stream
2
3
4  def merge_midi(path_1, path2, out_path):
5      """
6      Merge two tracks into 1.
7
8      Args:
9          path_1, path_2 (str): Path to the MIDI file.
10         out_path (str): Path to the output MIDI file
11
12     Returns:
13         music21.stream.Stream: Modified MIDI score.
14     """
15     midi_file1 = converter.parse(path_1)
16     midi_file2 = converter.parse(path2)
17
18     merged_stream = stream.Stream()
19
20     for element in midi_file1.flat:

```



```

21         merged_stream.append(element)
22
23     for element in midi_file2.flat:
24         merged_stream.append(element)
25
26     merged_stream.write("midi", fp=out_path)
27
28
29 if __name__ == "__main__":
30     path_1 = "/home/pi/Documents/ECE5725_final_proj/testrun1_final_midi.mid"
31     path_2 = "/home/pi/Documents/ECE5725_final_proj/IR_Midi/input.mid"
32     out = "/home/pi/Documents/ECE5725_final_proj/IR_Midi/merged_output.mid"
33     merge_midi(path_1, path_2, out)
34

```

Keyboard_Pi_Conductor/IR_Midi/modify_input.py

```

1  from music21 import converter, stream, tempo, meter
2
3  import midi2audio
4  import pygame
5  import numpy as np
6
7
8  def convert_midi_to_audio(midi_file, audio_file):
9      """
10     Convert a MIDI file to an audio file using midi2audio.
11
12     Args:
13         midi_file (str): Path to the MIDI file.
14         audio_file (str): Path to save the audio file.
15     """
16     fs = midi2audio.FluidSynth(
17         sound_font='/usr/share/sounds/sf2/FluidR3_GM.sf2')
18     fs.midi_to_audio(midi_file, audio_file)
19
20
21 def play_audio_file(audio_file):
22     """
23     Play an audio file using pygame.
24
25     Args:
26         audio_file (str): Path to the audio file.
27     """
28     print("start play audio")

```

```

29     pygame.mixer.init()
30     pygame.mixer.music.load(audio_file)
31     pygame.mixer.music.play()
32     while pygame.mixer.music.get_busy():
33         pygame.time.delay(100)
34     pygame.mixer.quit()
35
36
37 def modify_volume(midi_file, volume_changes, out_midi_file, bpm=60):
38     """
39     Modify the volume of a MIDI file based on user input.
40
41     Args:
42         midi_file (str): Path to the MIDI file.
43         volume_changes (list): List of tuples containing (start time, end time) and
44         volume change.
45
46     Returns:
47         music21.stream.Stream: Modified MIDI score.
48     """
49     # Load the MIDI file
50     score = converter.parse(midi_file)
51
52     flat_score = score.flat
53     # bpm=score.flat.getElementsByClass(tempo.MetronomeMark)[0].number
54
55     for note in flat_score.notes:
56         print(f'note offset: {note.offset}')
57
58     volume = 0
59     low, high = 0, 0
60     for _, _, volume_change in volume_changes:
61         volume += volume_change
62         low = min(low, volume)
63         high = max(high, volume)
64
65     for i in range(len(volume_changes)):
66         t1, t2, volume_change = volume_changes[i]
67         volume_changes[i] = (t1, t2, volume_change / (high - low) * 87)
68
69     i = 0
70     volume = 0 - low + 40
71
72     note_volume_list = []
73     while flat_score.notes[i].offset < volume_changes[i][0]:
74         note_volume_list.append(volume)

```

```

74         i += 1
75
76     buffer = []
77
78     for start_time_seconds, end_time_seconds, volume_change in volume_changes:
79         end_offset = bpm * end_time_seconds / 60
80
81         while i < len(flat_score.notes) and flat_score.notes[i].offset < end_offset:
82             buffer.append(flat_score.notes[i])
83             i += 1
84
85         for note in buffer:
86             volume += volume_change / len(buffer)
87             note_volume_list.append(volume)
88
89         buffer = []
90
91     while i < len(flat_score.notes):
92         note_volume_list.append(volume)
93         i += 1
94
95     note_volume_list = np.array(note_volume_list)
96     note_volume_list = (note_volume_list - note_volume_list.min()) / \
97         (note_volume_list.max() - note_volume_list.min()) * 100 + 20
98
99     print('note_volume_list: ', note_volume_list)
100    for i in range(len(note_volume_list)):
101        flat_score.notes[i].volume.velocity = note_volume_list[i]
102
103    score.write('midi', out_midi_file)
104
105
106    def test_volume_modification():
107        volume_changes = [
108            (3.0, 8.0, 130), # Louder at 7s, lasting for 3s
109            (10.0, 380.0, -100) # Quieter at 15s, lasting for 3s
110        ]
111
112        modified_score = modify_volume(
113            '/home/pi/Documents/ECE5725_final_proj/output_file.mid', volume_changes,
114            'output_file.mid')
115
116        # Convert the modified MIDI file to an audio file
117        audio_file = 'output_audio.wav'
118        convert_midi_to_audio('output_file.mid', audio_file)

```

```
119     # Play the audio file
120     play_audio_file(audio_file)
121
122
123 if __name__ == "__main__":
124     test_volume_modification()
125     # import sys
126     # play_audio_file(sys.argv[1])
127
```

Keyboard_Pi_Conductor/IR_Midi/init.py

```
1 |
```