Gavin Grooms
Tufts University
Capstone Project: Just Intonation Auto Accompaniment

Design Document

Introduction

This document will attempt to serve as an implementation plan for my capstone project, outlining the desired outcomes of my project while explaining in more technical detail how exactly I intend to reach those goals. As discussed in my README, my project has multiple phases which will act as separate goalposts where my project could be considered to be in a complete state and ready for submission for the purpose of this Capstone course. However, it is my intention to one day complete all phases to be described in this document, even if that occurs well after my time at Tufts is over. It's critical that each phase is in a completed working state before I can move on to the next phase due to the layered functionality I have planned, so thorough testing will need to be done at the end of the development of each phase.

Phase 1

The focus of phase 1 is to build the program that can turn MIDI input into a .wav file of all the notes in the file played together using Just Intonation, which is a purer intonation than the standard Equal Temperament (please see my Requirements Specification presentation for more information on the difference between Equal Temperament and Just Intonation). This program will be written in Python and I intend to use the following Python packages:

- NSound - https://pypi.org/project/Nsound/ - C++ framework built on top of CSound for synthesizing sound in mono/stereo tracks. This will be used to actually generate each tone in the output .wav file.
- Pydub - https://pypi.org/project/pydub/ - Python audio manipulation package. Using this to combine tracks of audio into a single .wav output.

- MIDIFile - https://pypi.org/project/MIDIFile/ - Python3 MIDI File stream parser. Using this to read the MIDI input to determine which notes to generate.

The above packages may be subject to change. My plan is to create a data structure to store one or more "tracks" which are to be read by the MIDIFile package and fed into individual NSound streams. Each stream will be part of an array of tracks so that any number of notes can be combined together to form a chord that is tuned together. Ideally, all tracks will be parsed one note at a time where an analysis will be performed on all notes occurring at the same time to determine what the chord is which is how the base note frequency will be decided with thirds, fifths, sevenths, etc. determined using Just Intonation. Once frequencies are determined, each note will be fed into its own NSound stream where they will later be combined using Pydub. Here is a quick rundown of my planned data structures:

- I'll be using A = 440 hz as the standard frequency, with a dictionary of all other notes saved based on this tuning for quick reference to be used as chord base frequencies. Note that notes such as $C^{\#}$ and D^{b} will use the same frequency in this dictionary saved under the more common name.
    - notes = {
      "c" : 261.63,
      "c#" : 277.18,
      "d" : 293.66,
      "eb" : 311.13,
      "e" : 329.63,
      "f" : 349.23,
      "f#" : 369.99,
      "g" : 392.00,
      "ab" : 415.30,
      "a" : 440.00,
      "bb" : 466.16,
      "b" : 493.88
      }
- A second array for notes will be used to help with determining intervals between notes for later chord evaluation. Using the absolute

value of the difference between any two notes in this array can tell us the exact interval between them.

```
intervals = {
  “c” : 0,
  “c#” : 1,
  “d” : 2,
  “eb” : 3,
  “e” : 4,
  “f” : 5,
  “f#” : 6,
  “g” : 7,
  “ab” : 8,
  “a” : 9,
  “bb” : 10,
  “b” : 11
}
```

- An additional array will be used to determine frequencies of notes in Just Intonation referencing a base note. The intervals will exist as translations from semitone difference.

```
intonations = {
  “1” : 16 / 15,
  “2” : 9 / 8,
  “3” : 6 / 5,
  “4” : 5 / 4,
  “5” : 4 / 3,
  “6” : 11 / 8,
  “7” : 3 / 2,
  “8” : 8 / 5,
  “9” : 5 / 3,
  “10” : 9 / 5,
  “11” : 15 / 8
}
```

- A class called MidiParser which will be used to save important variables for use in interfacing with the MIDI input file.

```
class MidiParser:
  def __init__(self, midiFile):
    self.midiFile = midiFile #MIDI File in use, see MIDIFile docs
    self.tracks = midiFile.parse() #Array of MIDI Tracks
    self.currTrack = 0 #Int for self.tracks index tracking
    self.midiEvents = self.tracks.parse() #Array of MIDI Events
                                          #from current track
```

- Array of NSound tracks where individual notes from each chord will be kept separate due to NSound's limitation of one or two notes being generated at a time. This class will also handle putting the audio together at the end into .wav format.
    - class AudiOutput

```
def __init__(self, channelCount):
  self.sr = 44100.0
  self.sin = Sine(sr)
  for x in range(channelCount):
    if x = 0:
      self.tracks = [AudioStream(sr, 1)]
    else:
      self.tracks.append(AudioStream(sr, 1)]

def addNote(self, trackNum, frequency, duration, gauss_width):
  envelope = self.sin.drawFatGaussian(duration, gauss_width)
  note = self.sin.generate(duration, frequency)
  self.tracks[trackNum] << envelope * note

def exportAudio(self, name):
  for x in range(len(self.tracks)):
    fName = "nsound_track" + str(x) + ".wav"
    self.tracks[x] >> fName
    if x = 0:
      audio = [AudioSegment.from_file(fName, format="wav")
    else:
      audio.append(AudioSegment.from_file(fName, format="wav")

    os.remove(fName)

  for x in range(len(self.tracks)):
    audio[x] = audio[x] – 7   #decreasing volume of each track
    if x = 0:
      combined = audio[0]
    else:
      combined = combined.overlay(audio[x])

  file_handle = combined.export(name, format="wav")
  #at this point, a .wav file named "name" should exist of the
  #program's intended output
```

The analysis of each chord will take place by comparing intervals of the notes being played at the MIDI event timestamp. A quick summary of how some of the chord analysis will be performed is below:

- Major chords have a base note with one note four semitones higher and another note seven semitones higher than the base note (e.g. a C chord is composed of C, E, and G, where E is four semitones higher than C and G is seven semitones higher than C).
- Minor chords have a base note with one note three semitones higher and one note seven semitones higher than the base note. Optionally, a note ten semitones higher than the base note may be included. (e.g. a C minor chord is composed of C, Eb, and G, where Eb is three semitones higher than C and G is seven semitones higher than C. C minor chords may also include Bb which is ten semitones higher than C).
- This function will compare all intervals of simultaneous notes played to determine a chord. It will first check if it's a major chord, then a minor chord, among other possibilities. If a chord is determined, then frequencies of all notes besides the base note will be determined using Just Intonation intervals instead of referencing the array of standard frequencies.
    - Example: the MIDI parser determines that B^b, D, F, and G are all being played at the same timestamp. Intervals of all notes considering each note as the possible base note are considered.
        - If B^b is the base note, then we have intervals of 4, 7, and 9. This almost seems like a major chord, but no major chord has a note 9 semitones above the base note. This doesn't make sense so the program continues.
        - If D is the base note, then we have intervals of 3, 5, and 8. This doesn't match any chord, so the program continues.
        - If F is the base note, then we have intervals of 2, 5, and 9. No chord has an interval of 2, so the program continues.
        - If G is the base note, then we have intervals of 3, 7, and 10. This matches our definition of a minor chord with optional seventh, so the function decides that G is the base note and returns.
- If a chord cannot be determined, then the program will instead generate all notes directly referencing the base frequency array I have defined.

- Once all frequencies are determined, the addNote function in the AudiOutput class will be called to actually add the notes to the output audio file.

Audacity will be used as a tool to verify that everything is working correctly as it is described above. The output should be a Just Intonation audio file of the entire MIDI file.

Phase 2

The purpose of phase 2 is to introduce the ability to retune each chord identified in phase 1 individually. This means that during the parsing of all the chords in the MIDI file before each one is added to the NSound stream, I will introduce a step where a user will be able to adjust the base frequency of each chord. I will adjust the program to pause and seek out input from the user in the form of an integer value of tuning adjustment in "cents" (relevant Wikipedia page on musical cents). I suspect that this phase of submitting this input may be tedious, as a single MIDI file may contain a large number of chords, so if time allows, I would like to develop a way for a user to submit all integer values at once. This could be accomplished by having the program parse through the entire MIDI file first to find all the chords before asking the user for an array of integers, or this might be accomplished by opening a window where these values could be entered in individually if desired, with 0 as the default case.
The intention behind this phase is to provide an interface with which later phases may automate this process. Once a chord has received a value for a positive or negative change in cents for tuning, the base note of the chord will be adjusted to the new tuning while all other notes in the chord will then be built on top of the adjusted base frequency as normal. This will output well-tuned chords that can be built off of any frequency. On the backend, not much will change besides changing the hz frequency of the base note, as mathematical intervals will already be used in phase 1 to determine the other notes of each chord. All other features of phase 1 will remain the same. While this phase sounds relatively simple, I expect to spend more time

adjusting the input interface of the cents for each chord to feel more intuitive, as I want to avoid a scenario where a user is manually typing in 100+ values into the command line before the program completes.

Phase 3

The purpose of phase 3 is to begin to automate the process of retuning each chord in the output audio file. Instead of asking for integer values in cents of how each chord should be retuned, I will instead write a separate process that will analyze a user input audio file of the solo part of a solo-with-accompaniment piece of music that will decide how tuning should be adjusted on each chord in the accompaniment.

Full Disclaimer: I do not expect to reach Phase 3 as part of my Capstone Project at Tufts University. Everything described hereafter is subject to change and is planned to be completed long after I leave Tufts.

I will use a Python library such as pyAudioAnalysis to analyze the user input and determine the adjustments in frequencies to be provided as input in phase 2. To ensure that the chords stay relatively true to the actual notes, this process will only allow up to 20-25 cents positively or negatively from the base frequency.

Phase 3 is intended only to be the development of the tuning analysis program. Feeding the input from this process to the original program in phase 2 is intended for phase 4.

Phase 4

Feed the output from phase 3 into the program defined at phase 2. Most of the work involved at this phase will be in testing my code and making minor adjustments as needed to handle exceptions, edge cases, and other unexpected behavior. Ideally, I should have an array of integer values as output from phase 3 to use as input for the program defined in phase 2. As these processes will technically be separate, this phase will focus on linking them together while keeping them separate in the case that a user would prefer to use my program in its state at the end of phase 2 only.

Phase 5

The focus of Phase 5 is to take phase 4 further and actually combine the input audio file that is analyzed in phase 3 with the output of phase 4 to create one final audio file that is a human soloist accompanied by computer generated tones that are tuned to the human recording. This phase will involve the use of further analysis of the human recording to adjust tempo, note lengths, and other styling in the computer-generated accompaniment. Essentially, I want to mix the solo and accompaniment parts into one cohesive audio file. This is a very complex phase that I'm not 100% sure is a solvable task without a great deal of machine learning, so I'm saving plans for its exact implementation to depend on the actual result of phase 4.

Other Implementation Details

At the end of any phase of my project, I intend to create a separate branch on my GitHub repository that is dedicated to hosting a release version of that phase. This way, as long as even one phase is completed, I can start working on any other phase I want and always thereafter have a completed project to deliver at the end of the second semester of my Capstone Project course. As an example, if I am still working on phase 2 or 3 in April 2024, I can still submit the results of phase 1 or 2 respectively in my final presentation without having to disrupt my progress on my current phase. To ensure that the end of each phase is a completed product, I will be performing extensive testing at the end of any phase before moving on to the development of the next phase. As each phase layers additional functionality on the results of the previous phase, this strategy should allow greater chances at success in later phases. I also plan to look into using Jira to track my progress in each individual phase by dividing the phase goals into smaller tasks that I can focus on individually. If I use Jira, I will provide view access to anyone who would like to be able to track my progress.

Conclusion

I expect to have phase 1 fully completed and working correctly by the end of Spring 2024, and I intend to make as strong an effort as possible to also complete phase 2 by the end of Spring 2024. If my progress is faster than expected, I will also make an attempt to work on phase 3, however I have no expectation at this point to have phases 3-5 completed as part of my final Capstone project submission. I’m very excited to start working on this project and am looking forward to working on it long after I finish my time at Tufts University.