Documentation:

Patrick Ikedi Egbuchulam

Music Engine Modules

Analyzer.py

Module Classes

AnalyzedElement: Wrapper class for music21[1] note, rest and chord objects

Attributes:

self.key*:*

the key signature that note instance corresponds to

self.element*:*

the music21 note, chord, or rest object it contains

self.roman*:*

the roman numeral, or scale degree, of that element analyzed in ***self.key***

self.measureNumber*:*

the measure the element occurs in a piece, or ***None***

self.timeSignature*:*

the time signature of the piece the element occurs in, or ***None***

self.beatOffset*:*

the beat within the measure the element occurs on (i.e. beatOffset = 2.0)

Methods:

get_notes_midi(self):

returns the midi pitch number(s) of ***self.element****.*

is_note(self):

returns true if self.element is a music21 Note object, else false

is_rest(self):

returns true if self.element is a music21 Rest object, else false

[1] Music21 Module Reference: http://mit.edu/music21/doc/moduleReference/

is_chord(self):

returns true if self.element is a music21 Chord object, else false

copy(self, key, element, measureNumber, timeSignature, beatOffset):

custom copy function with optional parameters

in_new_key(self, newKey):

returns a new AnalyzedElement whose self.element is a music21 object with the same function (roman numeral) as the current note, but in the new key.

Example:

```
key1 = music21.key.Key('c major')
key2 = music21.key.Key('f major')
note = music21.note.Note('G')
el = AnalyzedElement(key1, note)

// el has the note G in the key of C major, and self.roman now
corresponds to scale degree V

elInNewKey = el.in_new_key(key2)

print(elInNewKey.element.name())
```

```
Print output is 'C'
The note C in the new key of F Major is the same scale degree (V)
as the original note G is in the original key C Major
```

Module Methods

analyze(song_file):

Takes in a song file and parses it into measures of AnalyzedElement objects

generate_rhythmic_frequency_distribution(stream):

Observes and calculates the probability of observing a note with ql = b based off of the song

generate_rhythmic_transitions_distributions(stream):

Observes and calculates the conditional probabilities of transitioning to a beat of ql = t+1, given that the current beat is ql = t

to_stream(measures_of_elements):

Converts a group of measures of AnalyzedElement objects detailing a song, back into a playable stream.

Transformer.py

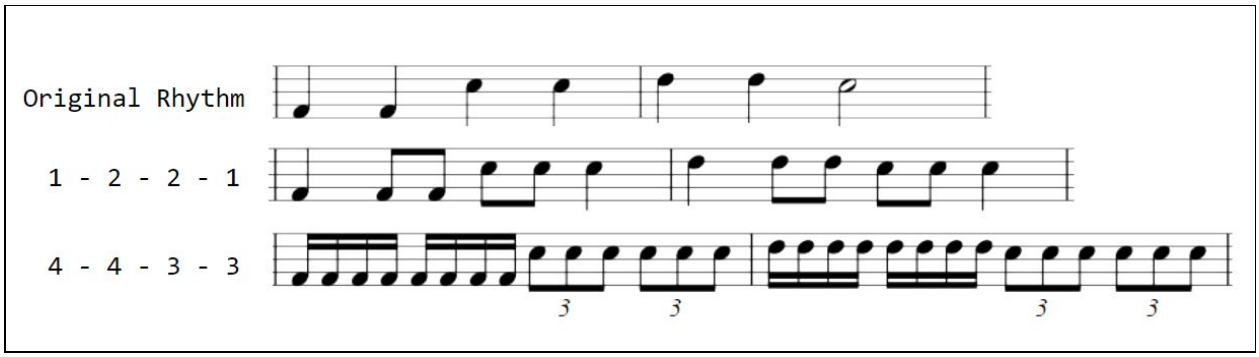
Module Methods

transpose_to_new_key(measures, key):

Translates all notes from their current key to the new key. Uses AnalyzedElement.in_new_key() method

fill_ostinato(measures, rhythm):

Takes the rhythm and applies the rhythm over the measures in the song. Forms a song structured on a single rhythmic idea. Rhythm are sequences of four integers (i.e. [1,2,2,1] or [4,4,3,3]. Example:



replace_rests(measures):

Removes all rests from measures, and replaces them with a nearby note.

Modulation.py

Module Classes

KeyNode: Node object in the "Key Signature Graph", corresponding to a key and mode combination.

Attributes:

self.tonic*:*

The tonic note, or name of that key. Must be in the set {A,B,C,D,E,F,G}

self.mode*:*

Mode of that key, currently either 'major' or 'minor'

self.edges*:*

dictionary, where keys are other KeyNote objects, and values are common chords

Methods:

insert_edge(self, other_node, chords):

inserts a common chord edge between this node and other_node, labeled with chords

get_adjacent_vertices(self):

returns all adjacent vertices in the graph (e.g. keys with common chords)

is_connected(self, other):

returns true if this node and other node are connected.

KeyModulator: Class that handles the computation of modulation paths from one key to another

Attributes:

self.triads_by_major_key*:*

dictionary, where keys are major key signatures, values are all of its triads

self.triads_by_minor_key*:*

dictionary, where keys are minor key signatures, values are all of its triads

self.dominant_7th_by_key*:*

dictionary, where keys are major and minor key signatures, values are the key's dominant 7th chord

self.diminished_7th_by_key*:*

dictionary, where keys are major and minor key signatures, values are the key's diminished 7th chord

self.tonic_by_major_key*:*

dictionary, where keys are major key signatures, values are the key's tonic

self.tonic_by_minor_key*:*

dictionary, where keys are minor key signatures, values are the key's tonic

self.common_chord_graph*:*

entire modulation graph, represented as a list of all vertices in it

Methods:

find_chord_path(self, start, end):

returns a chord path between the start key signature and the end key signature. the chord path is complete with intermittent tonicizations using the dominant or diminished 7th chords followed by tonic chords. The path takes the form:

... -> common_chord -> 7th_chord -> tonic_chord -> ...

get_modulation_measures(self, num_beats_per_measure, analyzed_chords):

returns measures (lists of AnalyzedElement objects) of the modulation path

Looper.py

Module Classes

SongLooper: Container for song's musical data, stored and looped through measure by measure.

Notable Attributes:

self.song_file*:*

the original song data

self.tempo:

the initial tempo of the song

self.original_parts:

the song data analyzed into measures of AnalyzedElement objects, split into all parts of the song

self.length:

length of song in number of measures

self.parts:

the current parts of the song (as transformations occur, this changes)

self.last_measure_beat:

keeps track of the start-beat number of the last measure played through

self.transformation_cache:

Cache that stores and provides previously computed transformations in the case that they are triggered again.

self.time_signature:

time signature of the song

self.initial_key:

initial key of the song

self.current_key:

current key of the song (as song looping and transformations occur)

self.key_modulator:

KeyModulator object from modulation.py

self.modulation_progression:

separate set of measures representing the chord modulation from old key to new key

Methods:

initialize(self):

sets up the looper with the original song parts to begin initial play through

set_tempo(self, tempo):

*sets **self.tempo** to tempo value*

reset(self):

sets the looper back to the beginning of the song

step(self, beat):

performs one step on the looper, queuing up the next measure of the music. If the current measure is the last measure of the song, step proceeds back to the beginning.

transform(self, part_indexes, key, rhythm):

transforms all part_indexes (if None, transforms all parts) and measures of music to the inputted key and rhythm. initiates modulation measures if key change occurs. sets self.parts to resulting measures after the transformation

get_current_measure(self):

returns the current measure data

get_all_parts(self):

returns all parts of all measures of music

get_measure_index(self):

returns the current measure index within the length of the song

get_last_measure_beat(self):

returns the last measure's start beat

set_modulation_progression(self, start, end, rhythm):

in the occurrence of a key change transformation, gets the playable measures representing the chord path between the previous key and the new key, setting self.modulation_progression equal to the result

AV_Grid.py

Module Classes

ParameterPoint: A single point on the AVGrid, whose location is (arousal, valence) and contains a parameter value for either rhythm, tempo, key, or instrument.

Attributes:

self.parameter_value*:*

the value the point is labeled with (rhythm, tempo, key, or grid depending)

self.arousal:

arousal value (x-axis) for that point

self.valence:

valence value (y-axis) for that point

Methods:

get_value(self):

returns the parameter value

distance_between(arousal, valence):

returns the Euclidean distance between (self.arousal, self.valence) and (arousal, valence)

AVGrid: Grid where the x-axis maps to arousal, and the y-axis maps to valence. ParameterPoint objects are placed accordingly on the grid.

Attributes:

self.min_valence:

y-axis lower bound (-1.0)

self.max_valence:

y-axis upper bound (1.0)

self.min_arousal:

x-axis lower bound (-1.0)

self.max_arousal:

X-axis upper bound (1.0)

self.points:

list of all ParameterPoint objects on the grid

Methods:

insert(self, value, arousal, valence):

creates and places a new ParameterPoint(value, arousal, valence) onto the grid

sample_parameter_point(self, arousal, valence):

creates a probability distribution of all points based off of distance from (arousal, valence), and selects a point by randomly sampling said distribution.

Synth_Runner.py

Module Classes (Relatively complicated widgets. General descriptions of each provided)

MainWidget(BaseWidget):

This is the very foundation of the audio-playback system. The majority of the code used to run this was provided by Professor Eran Egozy[2] as part of MIT's Interactive Music System course[3]. This widget holds the synthesizer that programs note_on and note_off events we hear in the audio, as well as various core objects that help map notes to rigid tempos and time signatures. This widget was augmented to host its own SongLooper object, and thus handles programming each measure to play in time, producing looped audio of the inputted song file.

TransformationWidget(MainWidget):

This widget is built on top of the MainWidget, and handles all possible transformation logic that might be triggered during looped playback of the song file. Some of the transformation functions native to this widget include:

- tempoUp and tempoDown (increases/decreases tempo by 8)
- switchInstruments (changes the synth patches currently being used)
- setVolume (changes the volume of current playback)
- keyChanged (changes the key and triggers key transformation in the SongLooper)
- rhythmChanged (changes the rhythm and triggers transformation in SongLooper)

KeyboardWidget(TransformationWidget):

This widget controls all the various transformations that are defined in the TransformationWidget via a keyboard interface.

ArousalValenceWidget(TransformationWidget):

This widget is similarly built on top of the TransformationWidget, but controls and triggers its transformations via values of arousal and valence being sent from external sources. This is the widget that provides our intended function within the scope of the game-music system. It currently listens in on a file where values of arousal and valence are being written to (for example, if say a game engine writes values to this file, then the music system can sync transformations to that gameplay). The AV Widget utilizes AVGrid objects for key, rhythm, tempo, and instrument, and samples each grid whenever a new arousal/valence value is detected.

---

[2] Eran Egozy's Profile https://mta.mit.edu/person/eran-egozy
[3] 21M.385 Course https://musictech.mit.edu/ims