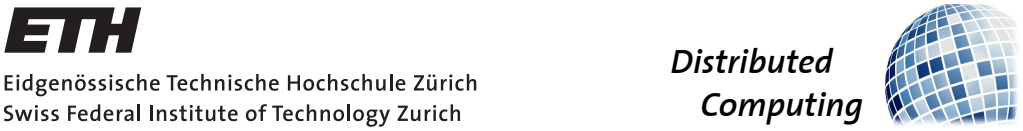



# Robot Composer Framework

Semester Thesis

Noah Studach

studachn@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

**Supervisors:**
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

January 12, 2018

# Abstract

As a successor to the project Robot Composer, this project provides a framework for knowledge-based algorithmic music generation. The knowledge-base is set by western music theory. The framework is implemented in Python and uses the *MIDIUtil* and *mido* library extensions. The generated music is saved as standard MIDI file to make it independent of audio samples, small in size and easy to modify. The framework also generates a simple musical piece as a proof of concept but the idea is that in the future the framework can be extended with additional functionality.

# Contents

Chapter 1

# Introduction

The overall goal of this project is to create music using only rules derived from music theory and discard machine learning techniques. This semester thesis is not the first one in the distributed computing group tackling this problem. Previously a robot composer semester thesis was written by Roland Schmid [1]. The scope of the thesis was to create a program that is able to act either as a backing group for a practicing soloist or to play full pieces of the Jazz and Blues genre. The approach was to solve the problem with help of Sonic Pi, a freeware music coding program implemented in Ruby [2]. Sonic Pi is built upon the principle of playing music in a loop and modifying this loop live, conveniently called a live-loop. The program can generate audio via a synthesizer or via replaying audio samples. A raspberry pi was used to run the program and there were some difficulties with installing Sonic Pi as well as performance restrictions caused by the hardware. In addition writing complex programs in the development environment of Sonic Pi proved to be rather difficult as it was designed for newcomers to playfully learn programming. Instead, ruby gem was used to send code fragments directly via command line. One needs to face these problems only once so they are not very impactful, but since Sonic Pi generates music by replaying sample, new samples must be provided when adding any new instrument. This is rather work intensive as every note ideally has its own sample. If not the samples have to be pitch bent. This impacts their audio quality and duration. Given previously faced challenges with audio samples and the unpredictable character of randomized music generation the benefits of outputting the music as a Standard MIDI file (SMF) was recognized. SMFs are smaller in size and easier to manipulate than audio files since the only store instructions but no audio. For this reason, the task of providing usable samples is passed on to the audio player. MIDI files are widely adopted and there are plenty of MIDI-capable players and digital audio workstation available. Sonic Pi is not capable of writing a SMF, thus we need a new framework to do the job. To keep the scope of this semester thesis reasonable the main task is providing this framework on which extensions can be added over time. As proof of concept of the framework, some extensions are created to the point that pieces of one music genre can be generated. To achieve our goals the framework should have

the following characteristics:

- Readability: The code must be readable for others, so they can continue the project.
- Extendability: The code must be structured with future extensions in mind to prevent future restructuring

The framework is written in Python, which provides good library extensions for MIDI capability and is overall a good language for rapid prototyping.

Chapter 2

# Theory

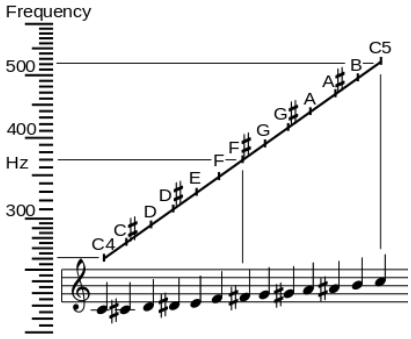
## 2.1 Notes

In western music theory, any note is represented in a musical notation defining its pitch and duration. A rest is represented as a note with no pitch and only a duration. The pitch connects the note to the frequency it produces when played as shown in figure 1a. Each pitch is represented by an uppercase letter and optionally a number and a # or b. The number defines the octave the note is played in and the letter defines which exact note in this octave is meant. The # represents an increase by a semitone while a b signals a decrease. The duration depends on the symbol and in western music theory it is quantized to either multiple or fraction by the power of two of the length of a measure as shown in figure 1b. [3]

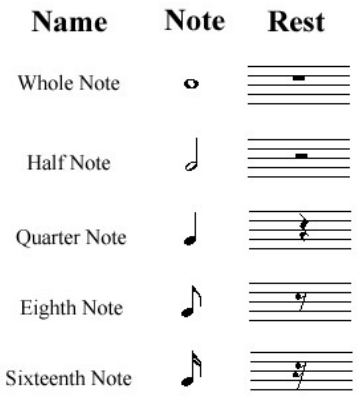
## 2.2 The MIDI Standard

Most electronic instruments and digital audio workstations (DAW) work or at least support the MIDI protocol and Standard MIDI Files (SMF). MIDI is an alternative source of music compared to audio files. Instead of representing music as a sum of signals with different frequency and amplitude, a SMF saves the instructions on how to produce this audio signals on any instrument that supports MIDI. Therefore a MIDI file is much smaller than its counterpart, but it loses information about the actual sound envisioned by the composer because the instructions can be played on any compatible instrument. A SMF is at its core just a set of MIDI events. An event is a MIDI message together with a timestamp. The timestamp, also called *Delta Time* is the time difference from the occurring event to the time the latest event occurred [6].

The file consists of *chunks* starting with the header chunk followed by one or more Track chunks. Each chunk starts with literal string identifier, either *MThd* or *MTrk*, followed by the length of the rest of the chunk. This is always 6 bytes for the header and for the track it depends on the number of MIDI events




(a) Frequency relation of notes [4]




(b) Note rhythm symbols [5]

Figure 1: A note is defined by frequency and timing

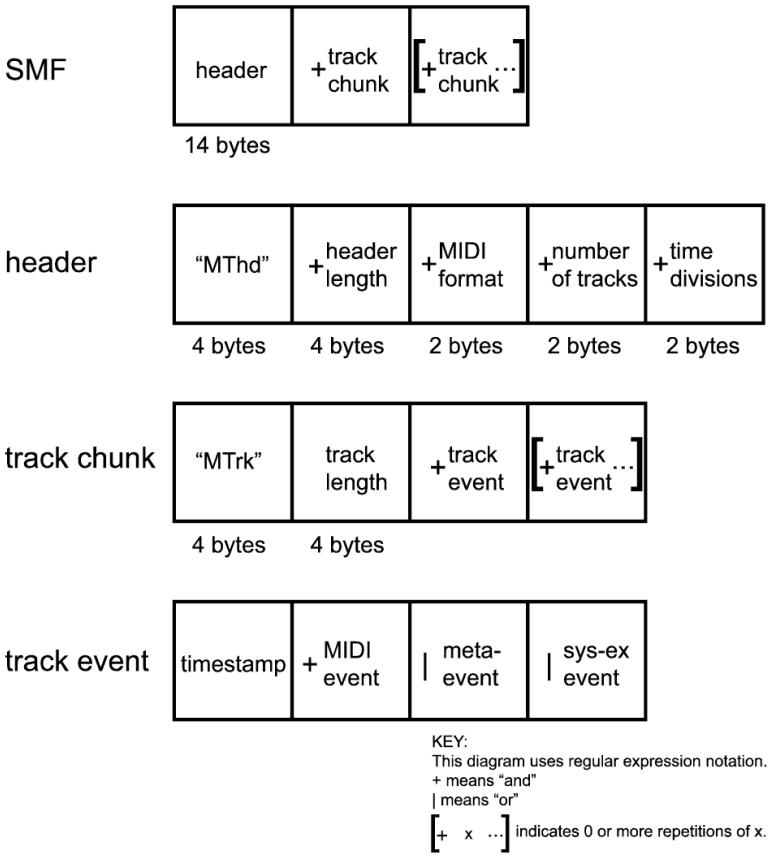



Figure 2: SMF file structure [7].

associated with it. The format field in the header identifies the file format and there are three different formats. Usually referred to as Type 0, 1 or 2. Type 0 writes all track events in one single track, while Type 1 has multiple tracks and Type 2 is a series of Type 0 files, but rarely used. The next two bytes of the Header define the number of Tracks, while the last two bytes give information about the time division. The file structure of a SMF can bee seen in Figure 2. Time in MIDI can either be encoded in metrical timing or timecode. When using metrical timing a value *pulse per quarter note* or *ppqn* is set. This value represents how many discrete steps there are between each quarter note, while the time between two quarter notes is given by the tempo of the track. While metrical timing is dependent on the tempo timecode is not. Here the first byte of the field specifies the number of frames per second (24, 25, 29 or 30) and the second byte the number of subframes per frame [8, 9].

There are three types of track events, a *MIDI*, *Meta* or *System exclusive* event. The type of event is identifiable by the first byte of the data.

### 2.2.1 MIDI Events

MIDI messages that are responsible for playing notes and shaping the sound are called MIDI events. They are *Note Off, Note On, Polyphonic Pressure, Controller, Program Change, Channel Pressure* and *Pitch Bend*. Any of these messages can be sent over one of 16 available channels. The messages are summarized in Table A.2 and a table relating MIDI notes to the classical notation is found in Appendix A.1.

### 2.2.2 Meta Events

*Meta* events are only of importance for MIDI files since they contain additional information about the song. Examples of these events are copyright information, lyrics as well as track and instrument names. Also under this category are tempo, time signature and key signature events, which can be useful when analyzing or manipulating a SMF as they provide additional information. Every *Meta* event has the following general form:

FF type length data

FF is a prefix, marking this as a *Meta* message. The type field gives the type of the message, the length field the length of the data field in bytes and the data field contains the effectively written information. Some events have restrictions on their appearance for example *tempo*, *key signature* and *cue point*. The *end of track* event deserves a special mentioning as it is mandatory after the last event of each *track chunk*.

### 2.2.3 System Exclusive Events

System Exclusive or *SysEx* events are specific to certain hardware. They exist to give hardware manufacturer more freedom when working with MIDI but are not important for this work.

### 2.2.4 General MIDI Standard

The MIDI standard only provides an instruction set on how a song is to be played, but never specifies which instrument plays it. To ensure the sound produced by a MIDI file is as close to the sound intended by the composer as possible the General MIDI Standard (GM) was invented. Electronic instruments compatible with the GM standard provide an instrument bank with 128 different instruments and a drum set. A control message can be sent to the instrument to select an instrument. The channel 9 is reserved for drums.[10]

## 2.3 Chords

A chord is an arrangement of three or more notes that together build a harmonic unit. Usually when mentioning a chord it is refers to a triton, meaning a three-note-chord. A chord's name is defined by the root note and the intervals (difference of two pitches) between each note it contains. An A# minor chord, for example, has the root A#, the next note is the minor 3rd (3 half steps higher), so C#, followed by F as the perfect 5th (7 half step higher). Other than the minor 3rd, also a diminished (2 semitones), major (4 semitones) and augmented (5 semitones) 3rd. One can combine the intervals and change the number of notes to create a huge variety of chords. The most common chords are listed in Appendix A.1.

### 2.3.1 Chord Inversion

A chord inversion is a shift from a specific note in the chord by an octave. There are two directions a chord can be inverted, upwards and downwards. When inverting a chord upwards, the note with the lowest pitch is increased by one octave and if inverting downwards the highest pitch is decreased by one octave. If a chord is inverted as many times in one direction as there are notes in the chord, the whole chord has been shifted by one octave.[11]

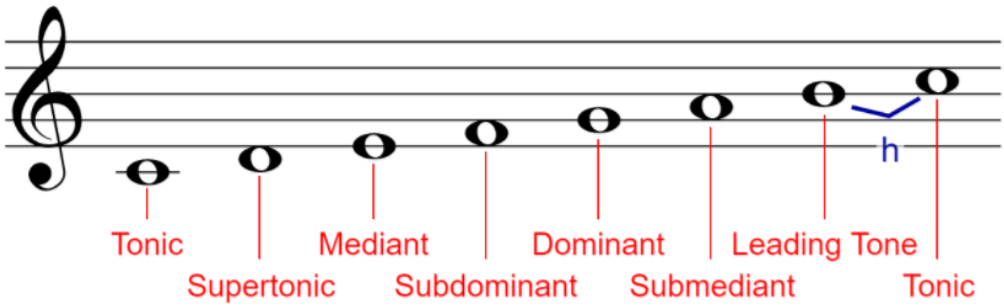



Figure 3: Function of notes [21].

## 2.4 Musical Scale

A musical scale is a set of notes that are in some musical relation to each other. A scale can be characterized by how many notes are in the set per octave and interval between each note. Notations for the size of the set are *Octatonic* [12] (8 notes), *Heptatonic* [13] (7 notes), and *Hexatonic* [14] (6 notes). Notations relating to the steps are *diatonic* [15] (7 notes per octave, 5 whole steps, and 2 half steps), *chromatic* [16] (12 steps per octave and equivalent to a normal piano keyboard), and *whole tone* [17] (6 notes per octave each interval is a whole step). There are many different scales but the most common scales are the two modes of the diatonic scale, called *Major* [18] and *Minor* [19].

## 2.5 Scale Degree

Each note in a diatonic scale has a name relating it to a melodic function [20] it provides. If the leading tone is a whole step below the tonic instead of a half step it changes its function to subtonic. The *C Major* scale with its notes and their function is shown in figure 3.

The same principle is applicable if we use the scale notes as root notes for tritone chords. However, now the functions are broken down to three essential functions: *Tonic*, *Dominant*, *Predominant*. A tonic chord, in particular the I, is a tonal center or the final resolution tone, a predominant chord normally resolves to a dominant chord, while a predominant chord creates instability that needs resolution provided by a tonic chord. These chords only contain notes that are also in the scale and are denoted with roman numbers. Figure 4 shows which chord provides each function. The value of the number refers to the position of the root note in the scale and a upper or lower case indicate if it is a major or minor chord respectively [21].

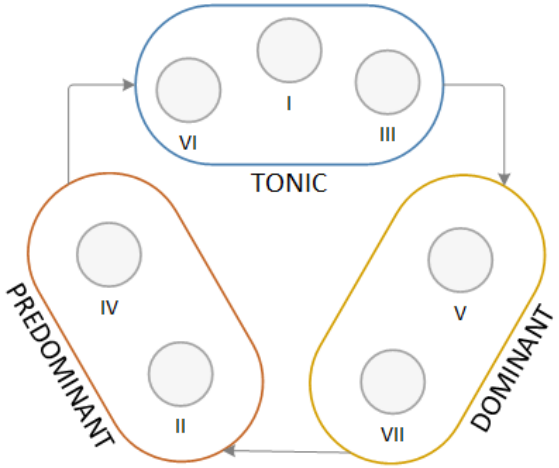



Figure 4: Chords grouped by function.

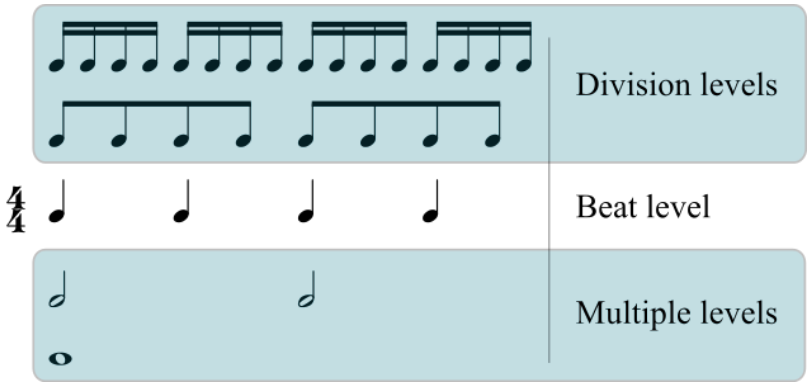



Figure 5: Metric levels of $^{4}/_{4}$ meter.

## 2.6 Meter

The meter defines how much time passes in a measure and how it is supposed to be played rhythmically. A meter is represented as a fraction and has two components, the numerator, and denominator. The denominator defines the note length of the beat level. The beat level can be seen as the base of the rhythm. When subdividing the notes from the base rhythm one enters the division level and when combining base notes it is referred to as multiple level. The figure 5 shows division and multiple levels of the 4/4 meter. The numerator indicates how many of the base notes are played per measure.

## 2.7 Metric Structure

The metric structure is important when looking at a song measure per measure. Not every beat in a measure serves the same musical functions. The more the listener expects a note at a position the higher is its metric strength measured in a metric weight. Each beat can either be an *on-* or *offbeat*. Onbeat notes mark a strong accent, while offbeat notes are weaker. All the notes in the multiple level are Onbeats, the others are Offbeats. So in figure 5 the first and third Beat are onbeat and the second and fourth are offbeat. The notes in the highest multiple level have the strongest metric position. The strength decreases the more notes are in the level. So the metric weights for the meters displayed in figure 5 are:

$$^{1}/_{1} : [0],$$
$$^{2}/_{2} : [0, -1],$$
$$^{4}/_{4} : [0, -2, -1, -2],$$
$$^{8}/_{8} : [0, -3, -2, -3, -1, -3, -2, -3],$$
$$^{16}/_{16} : [0, -4, -3, -4, -2, -4, -3, -4, -1, -4, -3, -4, -2, -4, -3, -4]$$

The first beat in a measure is called the Downbeat and is stronger than any other Onbeat, the last beat is called Upbeat and anticipates the Downbeat. Eric Thul presents an algorithm to calculate this metric structure from the meter [22]. *Numerator* is the numerator of the meter and *primefactors* are the primes from the prime factorization of the numerator.

Listing 2.1: Longuet-Higgins and Lee algorithm

```
metric_weight(numerator, primefactors):
    #weights vector w length n,level parameter k
    w = [0]*numerator
    k = 1
    for prime in primefactors:
            for i in range(1,numerator):
                if (i % (numerator/k) != 0):
                    #decrease weight vector
                    w[i]-=1
            k=k*prime
    return w
```

## 2.8 Tempo

The tempo defines how the metric time domain is mapped to the real-time and is usually defined as beats per measure *bpm* with one beat representing a one-quarter note. For example in a 60 *bpm* song each quarter note lasts for 1 second.

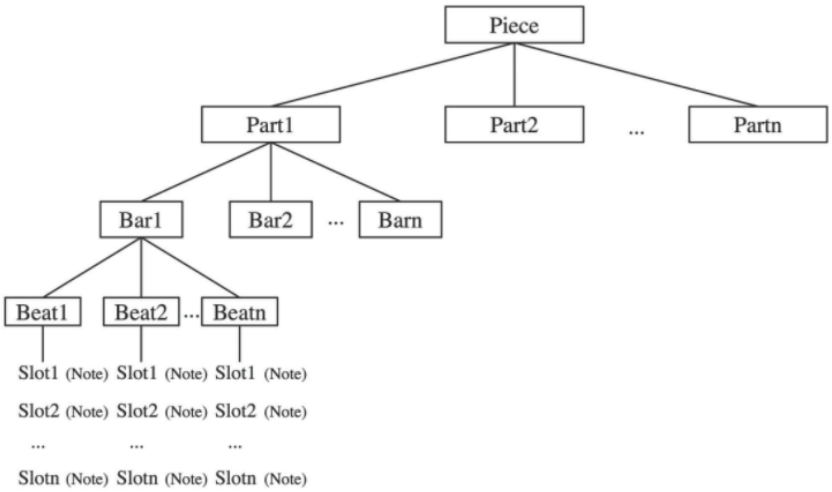



Figure 6: Song in hierarchical tree structure.

If we double the *bpm* to 120 the length of each quarter note is therefore halved to 0.5 seconds.

## 2.9 Song Structure

An article about the generation of tonal melodies based on music theoretical insights proposes to structure a melody in a hierarchical fashion as a tree [23]. Each part can be generated with different parameters. They handle the piece the same way as a part. So in practice, their melody consists of multiple pieces that each contain subparts. Each part is structured in bars and each bar consists of beats that can have multiple different notes playing. Such a tree is displayed in figure 6. This structure is also applicable to a whole song. A traditional song format is a selection from a few building blocks like an introduction, a verse, a pre-chorus, a chorus or refrain, a bridge, a conclusion or outro, an elision or an instrumental solo. The rules for which block to select and how to order them depends on the preferred genre. The most common format is: introduction, verse, pre-chorus, chorus, verse, pre-chorus, chorus, bridge, verse, chorus, outro. Each of these blocks can be seen as a piece of the song and can, therefore, be brought into the structure described above.

## 2.10 Melodies

Put simply, a melody is the combination of a rhythm with a succession of pitches. So a melody can be generated in at least two parts, pitch, and rhythm. Looking

at characteristics of a song such as instrument range, tempo, meter and scale, one can impose some restrictions on the choice of pitch and rhythm. On one hand, the range sets a minimal and maximal value for the pitch while the scale elects the harmonic notes. Those are a stable component of the melody and preferably appear at metrically strong positions. On the other hand, the meter and tempo presume a certain rhythm. As it is most common in music, there is no correct way of shaping a melody, but some techniques are taught more often than others. Although sometimes presented as algorithms, those techniques are far from it as they act more as a guideline and are generally vague with their instructions. One guideline [24] for pitch assignment is to first select a chord progression. In pop and rock, for example, I - V - vi - IV is one of the most common progressions and there are plenty of lists available online [25, 26]. When choosing or creating a chord progression the number of chords may vary depending on the meter, and the complexity of the rhythm. After the chord progression is selected, one has to come up with a contour curve, a mix of pitch increase, decrease and no change over time. This will set all melodies using the same chord progression apart. Then one searches for the nearest chord note to the contour curve for each onbeat note and assigns the pitch. Finally, it fills the space between each onbeat. Instead of choosing a chord progression as the base of the melody another guideline [27] first selects a scale, but then continues the same way as previously mentioned except that it selects the nearest note to the contour curve for each place in the rhythm and not only for onbeat notes. The pseudo code algorithm 1 is a mix of both guidelines, where the parameters $\alpha, \beta, \gamma, \zeta, \eta$ are *boolean* values and *range* is the range of notes of the melody,

Not every instrument voice or track can play the same melody. The requirements for a suitable melody heavily depend on the purpose the track has in the assembly. In most cases, the lead track plays a more complex melody than the others do. To achieve the desired melody there are again different techniques . Bass and ambient melodies, for example, can also be derived from the same chord progression as the lead melody did. This gives the melodies a common ground and leads to a more interconnected sound. One possible recipe for a bass and ambient melodies is shown in algorithm 2. The first step is to use the root notes of the chord progression as a simple melody to build upon. In the second step, all perfect 4ths and 5ths are eliminated as their pitch sound very similar and are therefore rather uninteresting in a musical sense. Next, a musical motif, a short musical idea or recurring figure, has to be established to give a sense of familiarity when listening. Single notes can be brought to the listener's attention if they are placed in the offbeat position. Also during longer notes, switching it out for another note in the chord connects the melody with the chord progression. If the melody should get even more complex, some note or parts of it can be shifted by an octave. The last step is adding rests to accents notes in the melody or notes from a simultaneously playing melody [28].

**Data:** range, $\alpha, \beta, \gamma, \zeta, \eta$
**Result:** Melody
choose a rhythm
choose a scale
**if** $\alpha$ **:**
    **if** $\beta$ **:**
        choose predefined chord progression
        chord = True
    **else:**
        generate chord progression
**if** $\gamma$ **:**
    start note = root note of scale
**else:**
    start note = random note
**if** $\zeta$ **:**
    end note = root note of scale
**else:**
    end note = random note
generate a contour curve from start note, end note, range
**for** *note in rhythm* **:**
    **if** *chord* **:**
        assign the nearest pitch to the contour curve that is in the chord
    **else:**
        assign the nearest pitch to the contour curve that is in the scale
    **for** *each pair of consecutive notes* **:**
        **if** $\eta$ **:**
            fill in adequate notes
add rests

**Algorithm 1:** Melody generation algorithm from Section 2.10. $\alpha, \beta, \gamma, \zeta, \eta$ represent a *boolean* value that enables a certain property

**Data:** Chord progression
**Result:** Bass/ambient melody
1. Use root notes of chord progression
2. Eliminate perfect 4th and 5ths (5/7 semitones intervals)
3. Create motif or reuse an existing motive
4. Accent offbeat (ex. split long note in 2 small ones)
5. Switch out the root note to another in-chord note
6. Move notes an octave (when nothing else is going on)
7. Add rests

**Algorithm 2:** Bass and Ambient melody algorithm [28]

## 2.11 Theme, Form and Variation

A theme is a concept where a musical idea, normally a melody and some accompaniment, is presented at the beginning of a piece and followed up by either variations or copies of it. A variation is a modified version of the original, that still shares recognizable features. A form acts in similar fashion, but with a distinct difference. While a theme focuses on the connection between two parts the form aims for contrast but keeping both parts still cohesive. Those parts are called A, the main focus of the track, and B, the contrast to A, and they can be built up from a theme and variations. A is typically 8 to 16 bars long and B is often shorter or longer with an uneven length to accentuate its weaker role. In its simplest form, A and B are arranged as ABA or better known as the simple ternary form with the key element of a return from B to A. The sequence, however, can be extended as desired. Variations are crucial when creating forms and themes and can be applied to different components such as the melody, rhythm or tempo. A list is provided in Appendix A.2. [29, 30, 31, 32]

Chapter 3

# Implementation

## 3.1 Programming Language and Libraries

MIDI is the technology that allows us to write and modify music on computers and most electronic instruments. Automated music generation is not easy and any bad sounding part can ruin a whole song. Being able to change those sections in the output with relative ease is an integral part of this project. The output is not regarded as the final product but more as a source of inspiration or automation of tedious task such as extracting the root notes from each chord or arpeggiate a melody. The making of this project also spanned over a relatively short time period. With these properties in mind, python [33] was selected as the programming language of choice as it is simple to use and allows for rapid prototyping. In addition, there are several public libraries available that can interact with MIDI. Two of those were selected, *MIDIUtil* [34] and *mido* [35]. While *MIDIUtil* is good for writing standard MIDI files (SMF) it does not support reading these. On the other hand, *mido* is good for reading MIDI commands but when writing MIDI files it does not provide the same level of abstraction as *MIDIUtil* does. For example, playing a note in a SMF requires two messages. A *Note On* message followed by a *Note Off* message. In *MIDIUtil* this is executed with a single function call

$$addNote(track, channel, pitch, start, duration, volume)$$

while with *mido* the same function is invoked twice with different parameters.

$$track.append(Message('note\_on', note = 64, velocity = 64, time = 32))$$
$$track.append(Message('note\_off', note = 64, velocity = 127, time = 32))$$

This perfectly showcases the tradeoff that is made when using *MIDIUtil* over *mido*. The single function call is better to read and understand, but it gives less control over the actual messages. For emaple, using *MIDIUtil* the release velocity of a MIDI note cannot be set. This trend continues throughout the libraries. In the beginning of this project *MIDIUtil* was selected but in retrospect, a complete

switch to *mido* will be beneficial and the lost abstraction can be provided by the framework.

## 3.2 Program Structure

The next step was to find a suitable structure to enable future code extensions. Since the program logic of future extensions will very likely be oriented towards the concepts of music theory the same approach was taken for the framework. The core of the framework are a few classes (Appendix A.3) that represent the building blocks of classical music theory : *Note, Chord, Scale, Meter*, and *Key*. All the classes are initialized with enough information to clearly define the object as predetermined by the theory. The purpose of those classes is to represent the music-theoretical construct and provide functions to manipulate themself. A song or piece of music is structured in the same hierarchical fashion as presented in Section 2.9. The piece is defined as a *Piece* that holds objects of the *Block* as subparts which in return can also hold subparts. If a *Block* has no subparts it must hold *Melody* and *Instrument* objects that will eventually be played. With this framework, music is generated by the means of variation. For this reason, the generation process is split into two kinds of functions. Generation and variation functions. As seen in Section 2.11, a variation modifies a given melody. So, a variation function modifies the *melody* object given as input depending on some parameter values. Whereas a generation function creates a new *melody* object from a set of input parameters. A generation function might call variation functions to modify the generated melody as desired.

## 3.3 Program Flow

The first task the program executes is tp read the *m_config.ini* file and write all the parameters in a dictionary. From those parameters a simple melody is generated. This melody is a simple chord progression either selected from predefined progression or generated according to Section 2.5. This melody is saved in the *Piece* object as the piece's main melody from which new ones are derived when needed. This is followed up by creating the song's format from Section 2.11. As for now only the genre *Electronica* is supported and the deterministic finite automata from figure 7 selects the format. For each track, an instrument is selected. Then each block in the song format is generated with its own generation function from the main melody by applying variation. Every *Block* object has a list of tuples and each tuple holds a *Instrument* and *Melody* object. Next, all blocks are appended to the subparts list in the *Piece* object. To finally create the SMF the play function of the *Piece* class is invoked, which writes the SMF. The program flow chart is shown in figure 8.

$$
\begin{aligned}
S &\Rightarrow ABC \\
A &\Rightarrow \alpha \,|\, \epsilon \\
B &\Rightarrow BB \,|\, \beta D \,|\, \epsilon \\
C &\Rightarrow \zeta \\
D &\Rightarrow \gamma \,|\, \delta \,|\, \epsilon
\end{aligned}
$$

Figure 7: *Electronica* DFA in Chomsky Normal Form

| Form Element | intro | verse | chorus | groove | outro |
|---|---|---|---|---|---|
| Variable | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ | $\zeta$ |

Table 3.1: Variables for DFA in figure 7

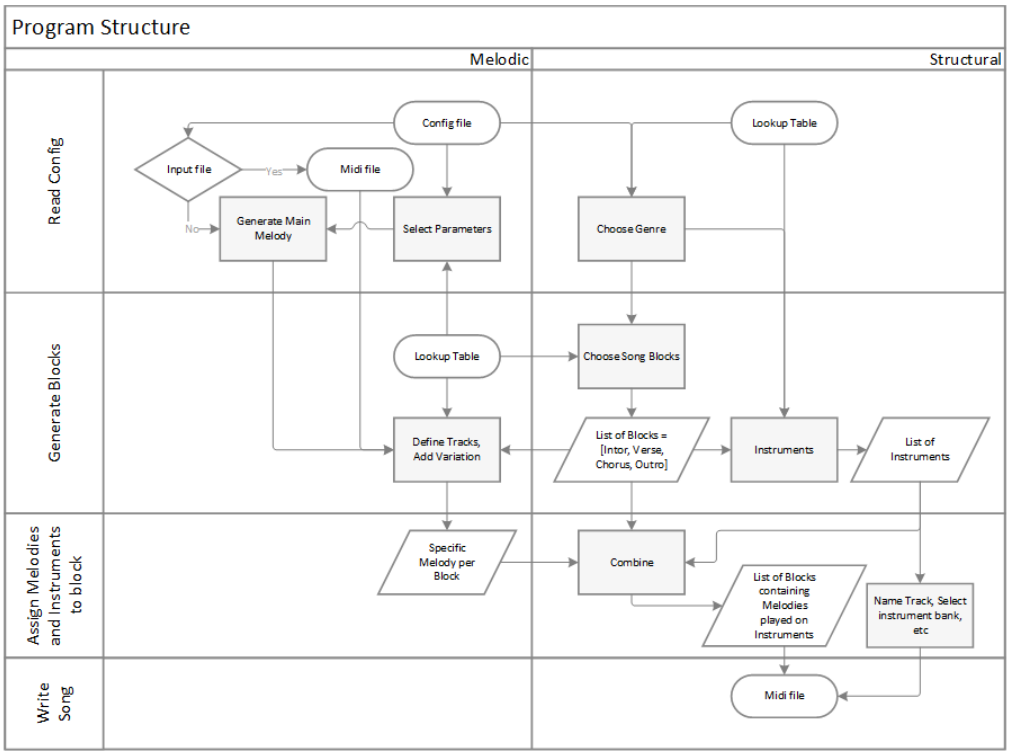



Figure 8: Program flow chart.

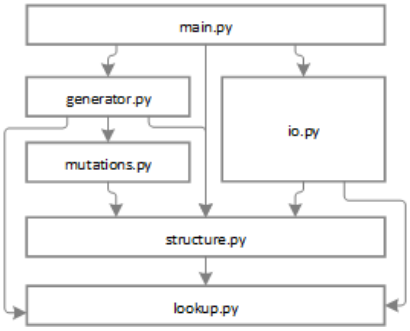



Figure 9: File dependencies.

## 3.4 Python Files

Readability is the second most important goal the program has. To make the program as readable as possible the code is split into six files according to the functionality it provides. Figure 9 shows the dependencies for each file. *Lookup.py* serves as a storage and holds all the definitions found in music theory, the instruments and their parameters as well as the definitions of the music genres, thus depends on no other file. All the class definitions can be found in *structure.py*. Therefore, it provides the basic operations in the framework and only depends on *lookup.py* for music theoretical information such as intervals in a specific chord or MIDI instrument bank numbers. All the variation functions are grouped in the *mutations.py* file. As these modify the object of the core classes, it only depends on *structure.py*. *Generator.py* on the other hand, stores all the generation functions and relies on the functions provided by *structure.py* and *mutations.py* as well as chord progressions stored in *lookup.py*. The *io.py* handles reading and writing any external files and needs information from *lookup.py* to save objects from *structure.py* in a dictionary. All those files are tied together by the *main.py* file, which is responsible to call each subpart of the program. It relies on *generator.py* for crating the melodies, *structure.py* for providing the *Piece*, and *io.py* for reading *config.ini*.

## 3.5 Time

In the framework, time is always defined in beats and therefore only dependent on the chosen tempo. The *bpm* is defined in quarter notes per minute as it is most common. So a time of 2.5 at 60 *bpm* is equal to 2.5 seconds.

Chapter 4

# Results

Using the framework, it is possible to generate MIDI files that work and play music when opened in a MIDI capable player. The biggest drawback with the generated files is their lack of diversity as they all have a similar format and all use the same small set of variation functions. As for now all parameters to influence the generation are listed in Table A.3, but not all of them are accessible through the configuration file. The program rarely crashes due to programming mistakes. One of those crashes was caused by invoking a function returning an object but never assigning the object to a variable.

In my opinion, the code with its hierarchical structure and the segments grouped by function is easy to understand. It also makes it clear where any function should be added in the future. My opinion is deduced from the experience of implementing new generation and variation functions. The more framework progressed and the amount of functionality provided by the framework increased, the less new code was required.

Chapter 5

# Discussion

The results show that the framework is not only capable of generating music, but also provides a useful set of components for future extensions. In its current state, the task of generating music perceived as good has clearly failed. However, the generated pieces can be a source of inspiration and improved manually, using a digital audio workstation. The implementation of the framework took longer than expected, leading to less music generation logic than planned. Every new addition to the framework had first to be researched and then, with possible extensions in mind, implemented and tested. Even though the testing was done in all conscience, the program occasionally did not work as intended and had to be fixed. At an early stage of the framework, the *play along* mode from the Robot Composer project [1] was implemented to test writing midi files and also to compare the outputs. It was further planned to reimplement the *play along* mode once the framework is finished and then compare the codes in regards to size and complexity, but there was not enough time left to do so.

## 5.1 Personal Insights

There are a few decisions that could have been made differently to save time and ultimately deliver a better framework. At first, an integrated development environment (IDE) should have been used instead of *notepad++* [36]. An IDE helps the programmer by providing *Docstrings* [37] and pointing out obvious programming mistakes. Secondly, too much time was spent in understanding the previous program on a very detailed level, even though a rough understanding would have been sufficient. Lastly, the knowledge about music theory was lacking, which resulted in restructuring the code twice. Then again it is equally probable that learning more about music theory would have been a waste of time as even for basic concepts it was hard to distinguish important and often used concepts and rules from those only used occasionally. Overall I am confident that with additional variation functions and more complex generation functions the output will improve immensely. To give one example, in the current state rests are completely ignored, although they are an important part of any song.

## 5.2 Future Work

The options for future extensions to this work are limitless. Apart from adding new variation and generation functions there are ways to improve the framework. To name one, more MIDI events from Appendix A.2 must be supported such as *Polyphonic Pressure* and *Controller* events. This unlocks the full potential of instruments that are able to play a continuous change in pitch such as guitar or a trombone. Other than that the option to play a note not perfectly on time will be a valuable addition. This imperfection gives the impression of a manually composed piece. Another idea that was never fully utilized, is that one or more SMFs can be selected as input, are then read by the program and finally incorporated into the output. It exists a function that reads a SMF and returns a *Melody* object consisting of *Note* objects, but it was never integrated into the generation logic. Apart from that, many comments suggesting improvements can be found in the source code.

# Bibliography


[1] Schmid, R.: Robot Composer.
https://git.ee.ethz.ch/manuelei/robot_composer_roland_schmid
(2017)

[2] Website: Ruby.
http://ruby-lang.org/en/
Accessed: 12.01.2017.

[3] Wikipedia entry: Musical Note.
http://en.wikipedia.org/wiki/Musical_note
Accessed: 09.01.2017.

[4] Image: Note frequencies.
http://en.wikipedia.org/wiki/Musical_note#/media/File:
Frequency_vs_name.svg

[5] Image: Note values.
http://musicreadingsavant.com/wp-content/uploads/2011/07/
notevalues.gif

[6] Website: About MIDI Part 4: MIDI Files.
http://midi.org/articles/about-midi-part-4-midi-files/
Accessed: 09.01.2017.

[7] Image: SMF Structure.
http://digitalsoundandmusic.com/wp-content/uploads/2014/05/
Figure-6.47-SMF-file-structure.png

[8] Website: MIDI Files Specification.
http://somascape.org/midi/tech/mfile.html
Accessed: 09.01.2017.

[9] Website: Outline of the Standard MIDI File Structure.
http://ccarh.org/courses/253/handout/smf
Accessed: 09.01.2017.

[10] Wikipedia entry: General MIDI.
http://en.wikipedia.org/wiki/General_MIDI
Accessed: 09.01.2017.

[11] Wikipedia entry: Inversion.
http://en.wikipedia.org/wiki/Inversion_(music)
Accessed: 12.01.2017.

[12] Wikipedia entry: Octatonic scale.
http://en.wikipedia.org/wiki/Octatonic_scale
Accessed: 09.01.2017.

[13] Wikipedia entry: Heptatonic scale.
http://en.wikipedia.org/wiki/Heptatonic_scale
Accessed: 09.01.2017.

[14] Wikipedia entry: Hexatonic scale.
http://en.wikipedia.org/wiki/Hexatonic_scale
Accessed: 09.01.2017.

[15] Wikipedia entry: Diatonic scale.
http://en.wikipedia.org/wiki/Diatonic_scale
Accessed: 09.01.2017.

[16] Wikipedia entry: Chromatic scale.
http://en.wikipedia.org/wiki/Chromatic_scale
Accessed: 09.01.2017.

[17] Wikipedia entry: Wholw tone scale.
http://en.wikipedia.org/wiki/Whole_tone_scale
Accessed: 09.01.2017.

[18] Wikipedia entry: Major scale.
http://en.wikipedia.org/wiki/Major_scale
Accessed: 09.01.2017.

[19] Wikipedia entry: Minor scale.
http://en.wikipedia.org/wiki/Minor_scale
Accessed: 09.01.2017.

[20] Wikipedia entry: Diatonic function.
http://en.wikipedia.org/wiki/Diatonic_function
Accessed: 09.01.2017.

[21] Website: Music Theory Lesson 23.
http://musictheory.net/lessons/23
Accessed: 09.01.2017.

[22] Thul, E.: Measuring the Complexity of Musical Rhythm. Master's thesis, McGill University, Montreal (2008)

[23] Povel, D.J., et al.: Melody generator: A device for algorithmic music construction. Journal of Software Engineering and Applications **3**(07) (2010)

[24] Website: An Algorithm to Compose Melody Lines.
http://lenmus.org/en/paginas/composer-algorithm-pitch#contour
Accessed: 09.01.2017.

[25] Website: The 10 most used chord progressions in pop and rock.
http://thornepalmer.wordpress.com/2011/12/29/
the-10-most-used-chord-progressions-in-pop-and-rock-and-roll/
Accessed: 09.01.2017.

[26] Wikipedia entry: List of chord progressions.
http://en.wikipedia.org/wiki/List_of_chord_progressions
Accessed: 09.01.2017.

[27] Website: The Ultimate Guide to Writing Better and More Memorable Melodies.
http://edmprod.com/ultimate-melody-guide
Accessed: 09.01.2017.

[28] Video: 6 Hacks for Better Bass Lines — Hack Music Theory.
https://www.youtube.com/watch?v=ujS0Viuv5kg

[29] Website: An Introduction to Form in Instrumental Music.
http://music.tutsplus.com/tutorials/an-introduction-to-form-in-instrumental-mus
Accessed: 10.01.2017.

[30] Website: How to Write a Theme.
http://music.tutsplus.com/tutorials/how-to-write-a-theme--audio-22412
Accessed: 10.01.2017.

[31] Website: How to Write Theme & Variations.
http://music.tutsplus.com/tutorials/how-to-write-theme-variations--cms-20558
Accessed: 10.01.2017.

[32] Website: Variation Techniques for Composers and Improvisors.
http://solomonsmusic.net/vartech.htm
Accessed: 10.01.2017.

[33] Website: Python.
https://www.python.org/
Accessed: 11.01.2017.

[34] Library documentation: MIDIUtil documentation.
http://midiutil.readthedocs.io/en/1.1.3//
Accessed: 09.01.2017.

[35] Library documentation: mido documentation.
http://mido.readthedocs.io/en/latest//
Accessed: 09.01.2017.

[36] Website: Notepad++.
http://notepad-plus-plus.org
Accessed: 12.01.2017.

[37] Website: Python Docstring Conventions.
http://python.org/dev/peps/pep-0257/
Accessed: 12.01.2017.

[38] Sam Aaron: Sonic Pi – The Live Coding Synth for Everyone.
https://github.com/samaaron/sonic-pi/
Accessed: 10.01.2017.

[39] Website: MIDI Note Numbers for Different Octaves.
http://electronics.dit.ie/staff/tscarff/Music_technology/midi/midi_note_numbers_for_octaves.htm
Accessed: 09.01.2017.

Appendix A

# Appendix Chapter

## A.1 List of Chords

A list of all chords supported by the framework. The values are the number of half tone steps from the root note [38].

- M:[0,4,7]
- m:[0,3,7]
- aug:[0,4,8]
- dim:[0,3,6]
- m+5:[0,3,8]
- sus2:[0,2,7]
- sus4:[0,5,7]
- 9+5:[0,10,13]
- m9+5:[0,10,14]
- M7:[0,4,7,11]
- dom7:[0,4,7,10]
- m7:[0,3,7,10]
- dim7:[0,3,6,9]
- halfdim:[0,3,6,10]
- 6:[0,4,7,9]
- m6:[0,3,7,9]
- 7sus2:[0,2,7,10]
- 7sus4:[0,5,7,10]
- 7-5:[0,4,6,10]
- 7+5:[0,4,8,10]
- m7+5:[0,3,8,10]
- add2:[0,2,4,7]
- add4:[0,4,5,7]
- add9:[0,4,7,14]
- add11:[0,4,7,17]
- add13:[0,4,7,21]
- madd2:[0,2,3,7]
- madd4:[0,3,5,7]
- madd9:[0,3,7,14]
- madd11:[0,3,7,17]
- 9:[0,4,7,10,14]
- m9:[0,3,7,10,14]
- M9:[0,4,7,11,14]
- 9sus4:[0,5,7,10,14]
- 6*9:[0,4,7,9,14]
- m6*9:[0,3,7,9,14]
- 7-9:[0,4,7,10,13]
- m7-9:[0,3,7,10,13]
- 7-10:[0,4,7,10,15]
- 7-11:[0,4,7,10,16]
- 7-13:[0,4,7,10,20]
- 7+5-9:[0,4,8,10,13]
- m7+5-9:[0,3,8,10,13]
- 11:[0,4,7,10,14,17]
- m11:[0,3,7,10,14,17]
- M11:[0,4,7,11,14,17]
- 11+:[0,4,7,10,14,18]
- m11+:[0,3,7,10,14,18]
- 13:[0,4,7,10,14,17,21]
- m13:[0,3,7,10,14,17,21]

| Octave | Note Numbers | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | C | C# | D | D# | E | F | F# | G | G# | A | A# | B |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 1 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 2 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
| 3 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 4 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 5 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| 6 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 |
| 7 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 8 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
| 9 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |
| 10 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | | | | |

Table A.1: Table with MIDI notes [39]

| Name | Size | Nr | Structure | Parameters | Function |
|---|---|---|---|---|---|
| Note Off | 3B | 8 | Nr+ch+n+val | val = velocity of note press (def. 64) | Turns note n on channel ch off |
| Note On | 3B | 9 | Nr+ch+n+val | val = velocity of note release (def. 64) | Turns note n on channel ch on |
| Polyphonic Pressure | 3B | A | Nr+ch+n+val | val = after-touch pressure | Applies After-touch to note n on channel ch |
| Controller | 3B | B | Nr+ch+c+val | c = controller nr, val = value | Activates a controller or channel mode c with given value |
| Program Change | 2B | C | Nr+ch+val | val = program nr | Switches channel instrument to val |
| Channel Pressure | 2B | D | Nr+ch+val | val = pressure | Applies Nr A to all active notes on channel ch |
| Pitch Bend | 3B | E | Nr+ch+lsb+msb | lsb+msb = amount of bend (00 40 = no bend) | Applies pitch bend to all active notes on channel ch |

Table A.2: Table with MIDI messages: $ch \in [0, 15]$ and $n, c, val, lsb, msb \in [0, 127]$

## A.2 Variations

### A.2.1 Melody

#### Contour Curve

Generating a melody according to Section 2.10 requires a variation function that can fit any existing melody along a given contour curve. *Set to contour* and *follow contour* provide this function. While *set to contour* set the pitch of each note exactly to the MIDI note closest to the contour curve, *follow contour* finds the closest MIDI note that is in the scale for notes or the inversion with the smallest interval from any chord note to the MIDI note closest to the contour curve.

### A.2.2 Harmony & Accompaniment

#### Arpeggio

The *arpeggio* function plays each not of a chord consecutively instead of simultaneously. Optionally a *rate* argument can be given and each of the consecutive notes will have duration *rate*. The sequence derived from a chord will repeat for the chords duration.

#### Floating Chords

The idea being *floating chords* is that the transition between chords is smoother when each note that is part of both chords is in the same place. For example two tritones have a C and in the first chord the C is the second highest pitch, then in the second chord C must also be the second highest pitch.

### A.2.3 Rhythm

The variation function *syncopation* emphasizes offbeat notes by placing a strong harmonic note on a weak beat. The function applies this by starting a onbeat note one division level note duration earlier than expected e.g. in a $^4/_4$ meter stop playing the second note one eighth note earlier, so from 1 to 1.5 time units, and starting the third note one eighth note earlier so from 1.5 to 3 time units.

### A.2.4 Tonality

The program has a variation function named *switch scale*, that changes the melody from scale A to scale B. This is done in two steps. First each note

is shifted by the interval of the two root notes with the subfunction *switch key*. In the second step the subfunction *switch quality* switches the mode of the scale by checking for each note it membership to the new scale mode. If any note is not in the scale the corresponding note is calculated from the values stored in *lookup.py*. For example a change from C *minor* to C *major* mode will change all the E's to D#'s, all A's to G#'s and all B's to A#'s.

### A.2.5 Others

#### Fading

*Fading* is when the volume of the melody is continuously changing over time. The function *fade* fades the melody from a time *start* to *stop* from the value *high* to *low*.

#### Muting

The *mute out* and *mute in* functions mute respectively unmute certain pitches each bar by deleting resp. not deleting them until all pitches are muted resp. unmuted.

## A.3 Core Classes

### A.3.1 Note Class

The Note class is the most basic object and very closely related to the *MIDIUtil* function *addNote*. It is generated with a starting time, a pitch representing its MIDI note value, a duration, and a volume representing the velocity of the MIDI note press e.g. the force a piano key struck. This class provides functions to read and change its parameters and to split the note into two separate notes.

### A.3.2 Chord Class

The Chord class holds a list of notes, the quality of the chord (Appendix A.1), and information about the inversion state of the chord. It provides the same functions as the Note class, but in addition can also invert the chord, find out if the notes form a chord and which, and fit a chord as close as possible to a given note.

### A.3.3 Key Class

The name of this class is derived from the piano key and not to be mistaken with scales often also called keys. The Key class is very simple and its purpose is to easily access information about a pitch. It not only holds the MIDI note but also in which octave it is and the interval to the next lower C.

### A.3.4 Meter Class

The Meter class holds the numerator, denominator and the metric structure (Section 2.6). This is important since the metric structure need to be preserved during the run time. The reason is that the output of the algorithm for generating the weights is dependent on the order of the prime numbers. $2 * 3$ will produce different weights than $3 * 2$ and if both are simultaneously used, timing mismatches are inevitable. The Meter class additionally provides the timings of the on- and offbeat notes.

### A.3.5 Scale Class

The Scale class provides information about the notes in a scale and is able to return chord with any scale degree as root note.

### A.3.6 Melody Class

A melody is saved as a object of the Melody class. A melody object consist of a mix of Note and Chord objects, but also includes a scale and meter used. The class provides the same functions as the Note class does or rather provides access to them indirectly.

| Parameter | Purpose |
|---|---|
| seed | random seed for random variables |
| channel | the MIDI channel the instruments play on |
| length | number of format blocks of the piece |
| nominator | numerator of the meter |
| denominator | denominator of the meter |
| m range | musical range of lead melody |
| *progression length | length of the main melody in bars |
| bpm | tempo of track in bpm |
| key | root note of the scale of main melody |
| quality | mode of the scale of main melody |
| *bar per chord | duration the same chord is played in measures |
| *tempo | modifies the meter for the drum rhythm |
| start from root | if set *True* the first chord of the chord progression will be the I |
| end at root | if set *True* the last chord of the chord progression will be the I |
| predefined progression | probability to choose a predefined chord progression, instead of generating one |
| *prob | probability of switching the key of the backing track |

Table A.3: Table with program parameters. Parameters with a * are not accessible trough *m_config.ini*.