Analysis of computer music using machine learning models

Damian Haziak

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2020

Analysis of computer music using machine learning models

Submitted by: Damian Haziak

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

This project aims to create a framework for analysing software synthesizers. This is done by sweeping the parameters of the synthesizer and extracting features from each sound generated by the parameters. A pipeline is designed and created that generates a sound for each permutation of parameters in the software synthesizer, and analyses this sound by extracting low and high level features from it. This feature data is then plotted on graphs to create visualisations of the capabilities of the software synthesizer that is analysed. The data is also used to train classifiers (using high level features) and regressors (using low level features). Classifiers are trained to predict high level features using the synthesizer parameters while the regressors are trained in reverse, with an aim to predict synthesizer parameters using low level features. Performance of both machine learning techniques is analysed, with classifier performance peaking at 91.76% and regressor performance peaking at 97.39% ($R^2 = 0.988$). However this is only seen for simple problems, and increasing the complexity of the synthesizer rapidly decreases regressor performance. Some further improvements to the pipeline and machine learning models is suggested.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I would like to thank Dr Julian Padget for coming up with this, and guiding me in some of the research and development done for this dissertation.

I would also like to thank my friends, family, and my girlfriend, in helping me stay focused when working on this, and helping me overall through my time at the university.

The effect of COVID-19 on this project

Overall there was minimal impact on the project by the lockdown measures put in place as a response to the COVID-19 pandemic. There was an opportunity for user research which would have been beneficial for this project. However, managing the playback of different sounds and asking questions regarding them would have been too challenging to complete in this timeframe. Logistically, it would have been impossible to keep confounding variables from affecting the study results. Nonetheless the project does not suffer from the lack of user studies, there is enough quantitative analysis available to be performed on the gathered data.

Chapter 1

Introduction

Synthesizers are very complex electronic instruments. Each synthesizer is characterized by the input parameters that are available to the user. Each possible sound made by the instrument is defined by the combination of parameters that are input into it. Considerable effort needs to be put into learning how to operate a synthesizer due to the large number of parameters that they can have. In addition to this, making a particular sound can be challenging, if an artist does not know the particular combination of parameters that would make that particular sound.

This dissertation proposes a system that analyses all possible sounds made by a software synthesizer by creating sounds for all combinations of parameters, and extracting information from those sounds using a feature extractor. The information is then visualized in graphs to give the user of the synthesizer a starting point in creating their desired sound. A secondary aim of this project is to use the extracted information in training a machine learning model capable of creating sounds given some features that the user wants the sound to have.

The remainder of this report is structured in the following way:

- Chapter 2 looks into the literature of software synthesis, and feature extraction in order to provide a grounding of understanding of each of those areas.
- Chapter 3 defines a set of requirements for each aspect of both aims of this project.
- Chapter 4 explains the design process for components of the system.
- Chapter 5 discusses the implementation of the designs created in the previous chapter.
- Chapter 6 analyses the results obtained from running the created system.

- Chapter 7 summarises the findings, concludes the main part of the report, and discusses potential future work that can be completed to improve the system.
- Following the bibliography the appendices show the full test results (Appendix B), extra graphs (Appendix C), and code listings (Appendix D).

Chapter 2

Literature Survey

This project aims to create a machine learning model that can build up an understanding of sound created by an instrument. The model will require access to a music creation tool, and a sound analysis tool. Section 2.1 examines some sound creation tools. Section 2.2 investigates methods of sound analysis.

2.1 Programmatic Sound Creation

For a sound to be analysed it has to be created first. One part of the project requires the creation of sound through the use of music programming languages, such that sounds can be created for analysis by the model itself. This section will examine music programming platforms to find the best one for the project.

2.1.1 Csound

Csound is known to be the longest-running heir to the early music programming languages (Lazzarini (2016)). It is a framework based on the C programming language, and it can serve many purposes based on the requirement of the user. Figure 2.1 shows the three main ways in which the Csound framework can be interacted with. On the highest level a music producer can use it as a sound design application. A synthesizer developer can use this as an audio engine, which is the lowest level of usage. This project is based on the middle level, using Csound as a sound generator within a piece of software.

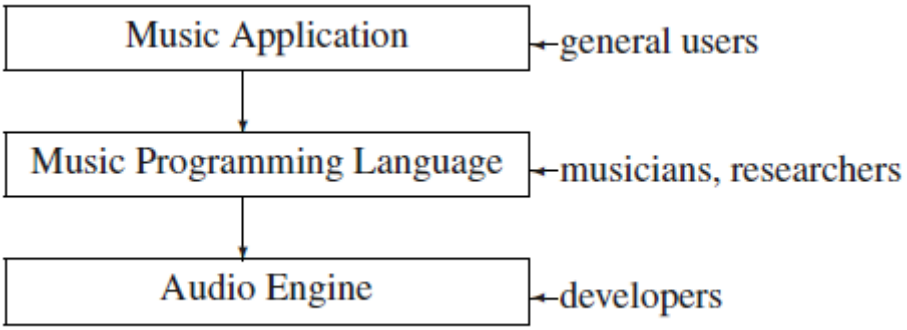

Figure 2.1: Possible levels of interaction with the Csound framework (Lazzarini (2016))

Programming in Csound is done in CSD files, which are a combination of HTML like tags and simple programming commands. Listing 2.1 shows a basic example of a CSD file. The `CsoundSynthesizer` tag defines the beginning and end of a Csound file. The next tag announces the beginning of an instrument definition and a very simple instrument is defined using the `rand` opcode. Opcodes will be covered in a later section. The `schedule` opcode then plays the instrument for one second. The Csound engine reads the file, compiles it, and runs the instrument. An alternative to the `schedule` opcode is to use a numeric score within a `CsScore` tag. This is a slightly more flexible solution as the score can be loaded in after the instruments are defined. The choice of solution solely depends on the needs of the developer.

Listing 2.1: Csound example, taken from Lazzarini (2016)

```
1 <CsoundSynthesizer>
2 <CsInstruments>
3     instr 1
4         out rand(1000)
5     endin
6     schedule(1,0,1)
7 </CsInstruments>
8 </CsoundSynthesizer>
```

Opcodes

Opcodes are considered the building blocks of instruments within Csound (Lazzarini (2016) pg.53). They can be thought of as functions like in any other programming language. Csound comes with many opcodes preprogrammed in the system, but it also allows developers to create their own. In the same fashion as instruments are defined between `instr` and `endin` keywords, opcodes can be defined using `opcode` and `endop` keywords.

Conclusion

Csound is a very powerful audio programming language capable of many things. Its flexibility and power allow it to be used in many contexts for both research and development. The availability of application programming interfaces for major programming languages (such as C, C++, Python, etc.) (Lazzarini (2016) pg.44) makes it a very accessible tool.

2.1.2 SuperCollider

SuperCollider is another major audio programming language. It is an open source blend of Smalltalk, C, and object oriented programming (Wilson, Cottle and Collins (2011)).

Listing 2.2: SuperCollider example, taken from Wilson, Cottle and Collins (2011)

```
SynthDef(
    "sine",
    {
    arg gate = 1, out = 0,
    freq = 400, amp = 0.4,
    pan = 0, ar = 1, dr = 1;

    var audio;
    audio = SinOsc.ar(freq, 0, amp);
    audio = audio * Linen.kr(gate, ar, 1, dr, 2);
    audio = Pan2.ar(audio, pan);
    OffsetOut.ar(out,audio);
    }
).add;
```

Listing 2.2 shows a simple example of a synthesizer definition in SuperCollider. The `SynthDef` class is used to define synthesis engines. In this case the `sine` class is created, and it defines a simple sound creator that is based on a sine wave. The definition of this is more complex than definitions in Csound but it gives the programmer more flexibility in programming.

To play the instrument the programmer requires to define an event. An example of an event is shown in listing 2.3. The event defines what instrument to use, and what parameters to pass to it. It is a simple key-value pair set that maps values onto arguments in the instrument definition, which can be seen in listing 2.2.

Listing 2.3: SuperCollider event example, taken from Wilson, Cottle and Collins (2011)

```
a = [
    type:           \note,
    instrument:     'sine',
    freq:           400,
    amp:            0.1,
    pan:            0,
    ar:             2,
```

```
    dr:             4,
    sustain:        2
];
```

Just-in-time programming

An interesting application of SuperCollider is live musical performance. Instruments can be defined and modified as the program is running, allowing a performer to adjust it as needed. This is an example of many audio programming languages used for live coding, another example of that would be Sonic Pi[1].

Machine Listening

SuperCollider comes with many useful functions for machine listening (Wilson, Cottle and Collins (2011) pg. 439-461). Simple features such as pitch and loudness, and more complex ones such as Mel-Frequency Cepstral Coefficients, can be retrieved from an audio signal using straightforward function calls. Refer to section 2.2 for an in depth look at music information retrieval methods and tools.

Conclusion

SuperCollider is a powerful and flexible audio programming language. It also has some APIs that allow it to be used in other programming languages such as Python[2][3], Scala[4], and even web development in JavaScript[5].

2.1.3 Other Languages

Many other audio programming languages exist, although they usually are not as well developed or powerful as the two previously mentioned. Some more popular alternative audio programming languages are: Common Music[6], Kyma[7], Nyquist[8], and the previously mentioned Sonic Pi.

2.2 Music Information Retrieval

Once the sound is created it requires analysis. Figure 2.2 shows a simplified diagram of the whole process of analysing and classifying a sound. The fol-

[1] https://sonic-pi.net/
[2] https://github.com/josiah-wolf-oberholtzer/supriya
[3] https://github.com/ideoforms/python-supercollider
[4] https://github.com/Sciss/ScalaCollider
[5] https://crucialfelix.github.io/supercolliderjs/
[6] http://commonmusic.sourceforge.net/
[7] https://kyma.symbolicsound.com/
[8] https://www.cs.cmu.edu/~music/nyquist/

lowing subsections cover pre-processing, feature extraction, and briefly discuss machine learning model training.

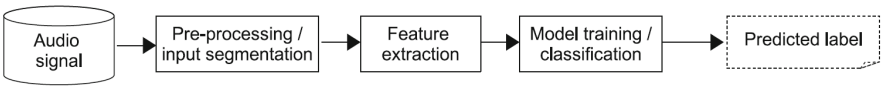

Figure 2.2: A simplified representation of extracting a label from a signal, taken from Eyben (2016)

2.2.1 Feature extraction

The way that information is retrieved from sound is through feature extraction.

Low Level Descriptors

The most basic features that can be extracted from audio signals are called Low Level Descriptors (LLDs). Different sources define LLDs in different ways. Widmer et al. (2008) calls them the basic components that make up a sound such as harmony, timbre, and rhythm, amongst others. On the other hand, Schedl, Gómez and Urbano (2014) consider them to have little meaning for humans but be easily understandable by computers. Ultimately, all studies do agree that LLDs are "simple, low complexity audio features" (Kim, Moreau and Sikora (2006)).

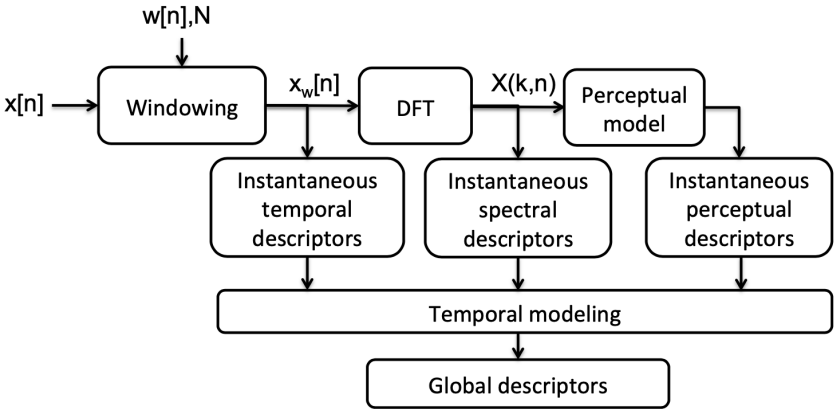

Figure 2.3: Diagram for LLD extraction, taken from Schedl, Gómez and Urbano (2014)

Figure 2.3 shows the general process of extracting a low level descriptor of any kind. LLDs only consider a short audio signal, therefore, the first step is to

apply a window function $w[n]$ that cuts out a section of the signal (with frame size N) away from the total sound $x[n]$. The process of windowing divides the total signal into several overlapping frames, typically between 20ms and 60ms in length (Eyben (2016)). Some features can be extracted straight away from this windowed signal, for example, zero crossing rate which is simply the number of times the signal crosses the 0dB mark (Chen (1998) in Eyben (2016)). An alternative to that is the mean crossing rate, where instead of 0dB the mark that is crossed is the mean signal strength. In the latter case the root mean square is used.

More often than not before any features are extracted, the windowed signal, $x_w[n]$, is preprocessed. Common ways of preprocessing the signal are:

- Down-mixing: instead of having a two channel (or any arbitrary number of channels) signal, sum it together into a single channel, creating one signal to process out of multiple ones.
- Resampling: the sample rate of a signal is the number of discrete values that represent a continuous signal, measured as a frequency in Hertz (Hz). To resample a signal is to change its sample rate, in this case to reduce it. The reduction in sample rate is done to reduce computation time.
- Filtering some of the audio signal in order to emphasize important features, or remove unnecessary components e.g. very low frequencies when aiming to extract melody features.

Low Level descriptors can be categorized into two types: ones retrieving information from the time domain and ones retrieving information from the frequency domain. Zero and mean crossing rates are examples of time domain descriptors (Eyben (2016)). Another time domain descriptor is root mean square power which can be used to model loudness (Widmer et al. (2008)).

To obtain the frequency domain descriptors the signal needs to be transformed into the frequency domain. This is displayed on Figure 2.3 with a Discrete Fourier Transform taking a windowed signal $x_w[n]$ as input and outputting the frequency domain representation $X(k, n)$ of that signal. Frequency domain descriptors include: band energy ratios, ratios between total energy in one frequency band and another frequency band; spectral centroids, centres of gravity of a frequency distribution which can model brightness of a sound; other spectral moments such as skewness and kurtosis; and so called Mel-frequency cepstral coefficients (MFCCs).

Mel-frequency cepstral coefficients are important in music information retrieval as they represent a signal spectrum in a compact way (Schedl, Gómez and Urbano (2014)). They are the most frequently used acoustic features in mood recognition (Eyben (2016)), and are used commonly in research (Soleymani et al. (2014); Lie Lu, Liu and Hong-Jiang Zhang (2006); Nalini and

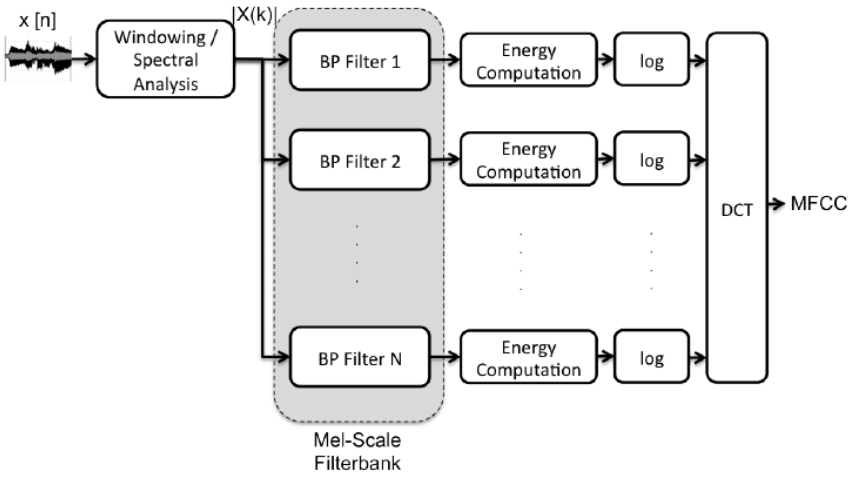

Figure 2.4: Diagram for MFCC extraction, taken from Schedl, Gómez and Urbano (2014)

Palanivel (2013); Yang and Chen (2012)). Figure 2.4 shows a diagram of how MFCCs are computed. The Mel-Scale filterbank transforms the frequency of the audio sample from the conventional frequency spectrum, $X(k)$, to the Mel-frequency spectrum. The Mel-Scale is a frequency representation that closely mimics the human auditory perception (Stevens, Volkmann and Newman (1937); Schedl, Gómez and Urbano (2014)), therefore making MFCCs important features as they represent an aspect of human hearing. The Mel-Scale processed signal is then processed by taking the logarithm of it, and applying a Discrete Cosine Transform (DCT) to all the logarithms, which results in the coefficients (Widmer et al. (2008)).

There are many more low level descriptors defined in the documentation (for example in Peeters and Rodet (2004)). This section will not be covering all of them. Please refer to the references for more information including mathematical definitions of the mentioned LLDs.

High Level Descriptors

Whilst LLDs extract the basic components of sounds, high level descriptors (HLDs) aim to capture semantic and/or syntactic meaning (Amatriain (2005)). They are often made up of collections or combinations of LLDs that together form a meaningful description that can be easily understood by a human. HLDs aren't as well defined between researchers as LLDs, they come under many names such as supra-segmental features or derived features (Eyben (2016)), higher level musical patterns (Widmer et al. (2008)), and high level

descriptors (Amatriain (2005)). For the sake of simplicity, this document will refer to them as HLDs. Despite the naming differences, all the researchers recognize that more complex features of sound can be extracted from combining the low level features across a longer period in the signal. Due to this, there are many ways in which HLDs are defined. This section will cover some more important definitions, and later sections will show how they can be useful for further analysis.

The simplest out of all HLDs is a feature vector. Multiple LLDs are all put together into an array and used as a complex descriptor of the sound. The limitation of this approach is that it grows at an exponential rate as the number of features and/or as the the length of the sample grows (Eyben (2016)). Eyben (2016) also proposes statistical functionals, which are maps that translate a series of values to a single value. Examples of these maps are means, maxima, minima, and the standard deviation. When applied to multiple descriptors, the functionals can extract trends between them such as covariance and correlation. Another type of functional available is the modulation functional, and that aims to describe the modulations and periodicities of a signal.

Widmer et al. (2008) proposes a few other HLDs such as pitch, also known as the fundamental frequency or the key of the song, the rhythm, the beats per minute and the location of beats, and harmony. These can be used to define some high level concepts, such as genre, however, Widmer et al. (2008) argues that the only way to truly define high level descriptors such as mood is through the usage of machine learning algorithms. Widmer et al. (2008), alongside Yang et al. (2007), also argues that music is too personal to be defined objectively.

2.2.2 Music Emotion Recognition

Using all the tools described in section 2.2.1 many researchers aim to extract emotional features from music. Before emotional sentiment can be extracted from a sound it first needs to be defined. There are two main approaches to defining emotion in music emotion recognition (MER): the categorical approach and the dimensional approach (Yang and Chen (2012)). The former uses a collection of adjectives to describe emotional sentiment (Sorussa, Choksuriwong and Karnjanadecha (2017)), and the latter defines emotions by creating a numerical representation of them in some emotional axis space. The categorical approach is often criticised for lacking expression and granularity (Yang and Chen (2012)), although this approach would be difficult to make granular, as that implies introducing a lot of keywords describing very similar emotional states which would lead to confusion (Sloboda and Justin (2001) in Yang and Chen (2012)). There is also a lack of common taxonomy of adjectives describing emotion (Lie Lu, Liu and Hong-Jiang Zhang (2006)), which makes research difficult. Researchers often define their own set of adjectives,

although recent study has used Hevner's eight clusters of affective terms as the basis of research (Yang (2011)), which can be seen in figure 2.5.

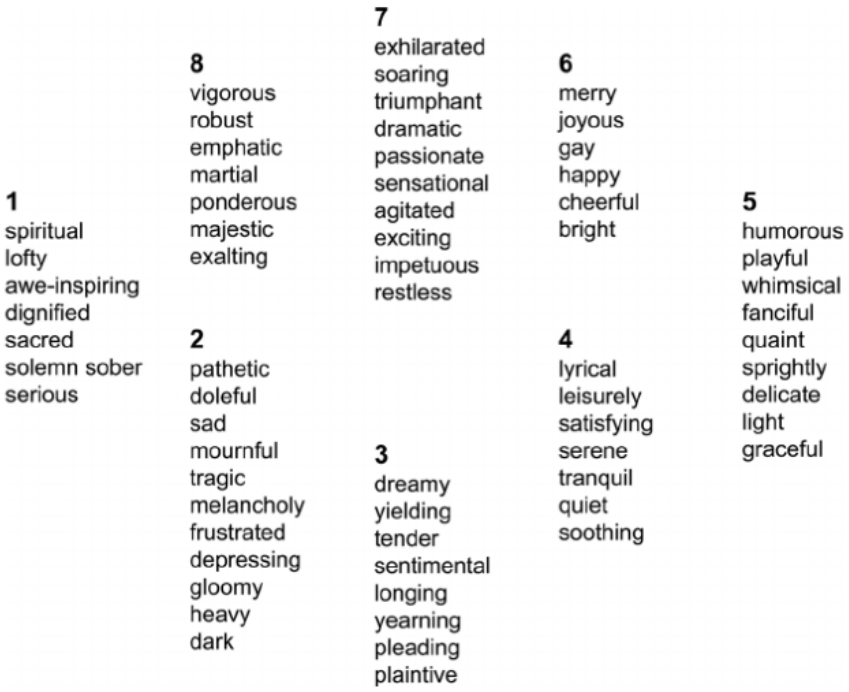

Figure 2.5: Eight clusters of affective terms, taken from Hevner (1935)

The dimensional approach does not have these issues and therefore it is often the preferred approach (Lie Lu, Liu and Hong-Jiang Zhang (2006); Yang et al. (2007); Soleymani et al. (2014); Aljanaki, Yang and Soleymani (2017); Huron (2000); Atcheson, Sethu and Epps (2017)). The most commonly used dimensional approach is the circumplex model of affect (Posner, Russell and Peterson (2005)). It defines emotional affect as two axes, valence and arousal, as shown in figure 2.6. The horizontal axis represents valence, sometimes also called pleasure, and the vertical axis represents the degree of arousal. The categorical adjectives can be defined using the dimensional model (Sorussa, Choksuriwong and Karnjanadecha (2017)), thus the dimensional model can be thought of as a generalization of the adjective clusters.

Music Emotion Recognition models are most often built upon machine learning. As already mentioned many researchers believe that machine learning is the only way that proper connections between musical features and emotional features can be made (Widmer et al. (2008); Sorussa, Choksuriwong and Karnjanadecha (2017)). Various machine learning models have been used for this purpose: decision trees (for the categorical approach), k-nearest neighbours, Gaussian mixture models, artificial neural networks, and many others.

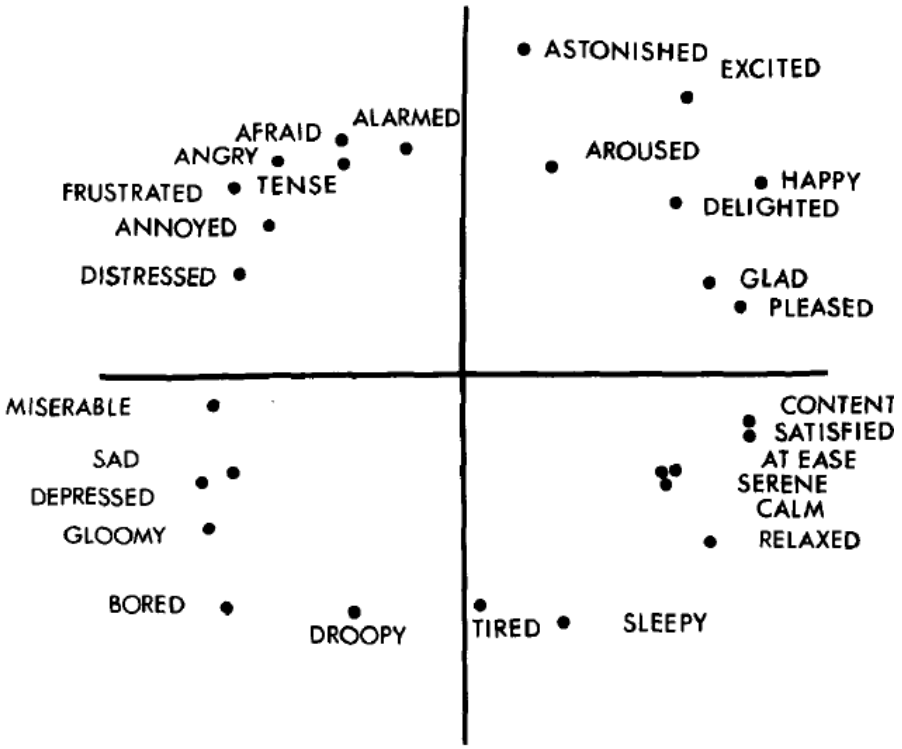

Figure 2.6: The circumplex model of affect, taken from Russell (1980)

However, a lot of research agrees that the most successful ones are support vector machines (Li and Ogihara (2003); Laurier et al. (2010); Laurier and Herrera (2007); Markov and Matsui (2014); Song, Dixon and Pearce (2012)).

2.2.3 Tools

Over the years many tools have been developed to help researchers extract features from audio signals. This subsection will look at some tools available and what they can offer.

openSMILE

openSMILE[9] is an open source audio extractor. It combines feature extraction algorithms from the MIR and speech processing communities into one package. It is also built in C++ and offers an API that can be used in other programming languages. All of the features covered in section 2.2.1 are possible to extract using openSMILE. The same company that created this toolkit also

[9] `https://www.audeering.com/opensmile/`

created a similar one primarily for affect recognition called openEAR (Eyben, Wöllmer and Schuller (2009)), which is what is necessary for this project.

Marsyas

Marsyas[10] is another tool that allows for easy Music Information Retrieval. It has been used in research completed by Soleymani et al. (2014) and in other industrial or academic projects (Tzanetakis (2009)). It has a very limited number of features that can be extracted from it, therefore it may not prove as useful for this project.

Essentia

Essentia[11] provides an open source library and tools for audio and music analysis, description, and synthesis. Similarly to openSMILE, it has a large set of algorithms for feature extraction. The Essentia model also offers some pre-trained models for HLD extraction, one of which is mood (Bogdanov et al. (2013)). New models for usage in this tool can be generated using Gaia[12], a C++ library with Python bindings. It is highly unlikely that Gaia will be used, as creating a new model from scratch is beyond the scope of this project.

SuperCollider functions

The machine listening functions found in SuperCollider can prove to be useful too, although they only extract the features. This project requires a tool that will also combine features into high level descriptors. SuperCollider does not provide that functionality.

[10] `http://marsyas.info/`
[11] `https://essentia.upf.edu/documentation/`
[12] `https://github.com/MTG/gaia`

Chapter 3

Requirements

This chapter outlines the requirements made for the system, and justifications for them. The requirements are based on the general problem definition in the introduction and the literature survey. At the time of research and writing, no systems existed that solved the problem defined in Chapter 1, therefore the requirement creation is not based on or relates to such systems.

The primary objective of the system is to establish a pipeline for sweeping through parameters of a software synthesizer and extracting features from the sounds created. The two secondary objectives are as follows: visualise the feature data obtained from the parameter sweep of the synthesizer, and use the feature data to create a machine learning model able to understand the relationship between the software synthesizer parameters and the features extracted from the sound.

Requirements are split into two sections: primary requirements for the primary objective, and secondary requirements for the secondary objectives. Primary requirements are considered essential and must be completed as part of the project. Secondary requirements are not considered essential, and have only been completed if time allowed for that.

3.1 Primary requirements

PR 1.1: The system must be able to use a software synthesizer to create a sound, given a set of parameters.

> *In order for the sound to be analysed, the system must have a way of creating sounds.*

PR 1.2: The system must have access to the parameters of the software synthesizer.

> *Information about the parameters is required so that they can be applied to the synthesizer when the sound is created. They are also used for further analysis of the sound.*

PR 1.3: The system must be able to input parameters into the software synthesizer.

> *Input parameter information must be used by the synthesizer otherwise the system will be unable to properly analyse all possible sounds that can be made by it.*

PR 2.1: The system must be able to use a feature extractor to retrieve feature information from a given sound.

> *A feature extractor must be available to the system so that it can be used for retrieving feature information from the sounds made by the synthesizer.*

PR 2.2: The feature extractor must export data in a usable format.

> *The extracted features will be used in secondary objectives of the project, therefore they must be stored and available to be used by any other part of the system.*

PR 3: The software synthesizers and feature extractors must be easily repluggable.

> *Once created, changing from one synthesizer and/or feature extractor should be straightforward. This allows for more detailed analysis of a synthesizer using multiple feature extractors.*

PR 4.1: The selected software synthesizer and feature extractor must be part of a pipeline.

> *The pipeline joins the two together in one workflow that creates sounds and extracts features from the created sound.*

PR 4.2: The pipeline must sweep through all parameter combinations given the parameter definitions in the software synthesizer.

> *Each parameter combination has to be studied i.e. the sound corresponding to those parameters has to be created and the features from that sound extracted in order to fully explore the capabilities of a synthesizer.*

3.2 Secondary requirements

SR 1.1: The system should use the data from feature extraction to train an appropriate machine learning model.

> *The machine learning model is trained so that it can predict parameter values given the features that the user wants the output sound to have. The parameter values can then be input into the software synthesizer and the desired sound is created.*

SR 1.2: The model should be appropriate to the output feature data type.

The choice of model depends on the type of data output by the feature extractor e.g. regression is more suited for continuous data, while classification is more suited for categorical data.

SR 2: The system should visualize the extracted feature data.

The visualizations can become a starting point for the user creating a sound, indicating parameters that would create certain types of sound.

Chapter 4

Design

This chapter describes the design decisions made for the system architecture and its separate parts. It starts by examining the overall structure of the pipeline and the system components. Each component is then looked into with more detail, describing the necessary data and functions within it.

4.1 Overall Pipeline

Based on the requirement specification from Chapter 3 and the problem definition from Chapter 1, the overall system function pipeline can be summarized using Figure 4.1.

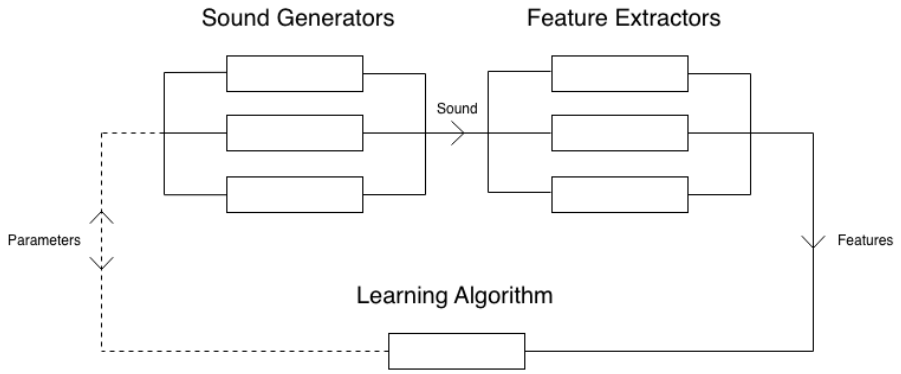

Figure 4.1: Overall structure of the software pipeline.

There are three major components: sound generators, feature extractors, and learning algorithms.

4.1.1 Sound Generators

The sound generators are any pieces of software capable of creating a sound file given some input parameters. As defined in section 3.1, the parameters of the sound generator need to be exposed to the software for it to operate correctly. This excludes most proprietary software synthesizers available to the public as they can only be operated through the use of a graphical user interface. Choice of software synthesizer is covered in Chapter 5.

Figure 4.1 displays three sound generators in parallel. This is purely for illustrative purposes highlighting the repluggability of the generators. Only one sound generator should be analysed at one time to avoid confusion. However in practice, the system should be able to support any arbitrary sound generator as long as it is defined correctly.

The sound generator gets given some parameters and creates a sound that is then passed onto the next component.

4.1.2 Feature Extractors

The feature extractors take the sound made by the generators and output a list of features. Just like with the sound generators, the three extractors in Figure 4.1 are purely illustrative. No knowledge of the internal programming of the extractors is required for the system to work. As long as it is known what features they extract they can be used within the system.

Once the features are extracted they are saved and can be used in the last part of the pipeline.

4.1.3 Learning Algorithms

The learning algorithms are meant to be machine learning models capable of learning the relationship between extracted features and the input parameters. The first stage of their use is training. Features from the extractors and saved parameters from the generators are input into the model for training. Once the model is trained, it can be used to generate parameters to input into the sound generator. The two way transfer of parameters between sound generators and learning algorithms is shown on Figure 4.1 by the dashed line with two arrows.

The choice of learning algorithm is highly dependent on the choice of feature extractors. As discussed in section 2.2.2, high level descriptors can be either categorical or dimensional. Both of these require different machine learning models, as one cannot handle both categorical and continuous data. The choice of model is further discussed in section 5.5.1.

4.1.4 Pipeline

Given the two stage usage of learning algorithms, the pipeline can start in two places. Initially, it starts with sound generators creating sounds given

parameter values from some parameter sweeper. These sounds are analysed using feature extractors and features are used to train the learning algorithm, as described in the previous section.

Once the machine learning model is trained, the pipeline can start with the learning algorithm, inputting predicted parameters into the sound generator. Features are extracted from the new sound and the learning algorithm is retrained using all the previous data and data from the new sound. This process could allow the learning algorithm to refine its understanding of the sound generator by filling in the gaps that might have been missed in the initial parameter sweep.

4.2 Component Architecture

Now that there is a general function of each component defined, a software architecture can be created, first for the overall system and then for each component individually. Figure 4.2 shows an architecture diagram of the entire system.

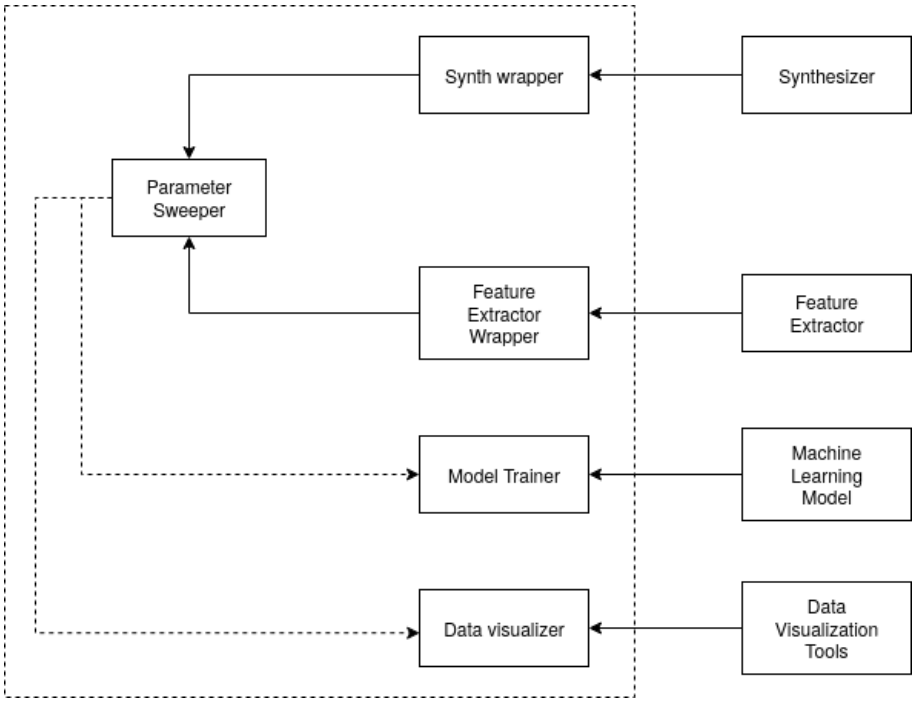

Figure 4.2: Architectural diagram of the system.

The dashed box defines the boundaries of the system. Components inside the boundaries are programmed as part of the system. Components outside are external libraries and/or software that is used to complete the objectives

of the system. Solid arrows indicate dependencies and/or input into another component. Dashed arrows denote the usage of data generated by the parameter sweeper, by the model trainers and data visualizers.

Based on the architectural diagram the obvious choice of programming paradigm is object oriented programming. Each interchangeable component can be an object, and changing between them is simply changing what object is input into a method. This yields a simple solution to **PR 3** as defined in section 3.1.

4.2.1 Sound Generators

With the software architecture created, the focus shifts now on the design of the specific components of it, starting with the synthesizer wrapper. Figure 4.3 shows a UML class diagram for the synthesizer wrapper.

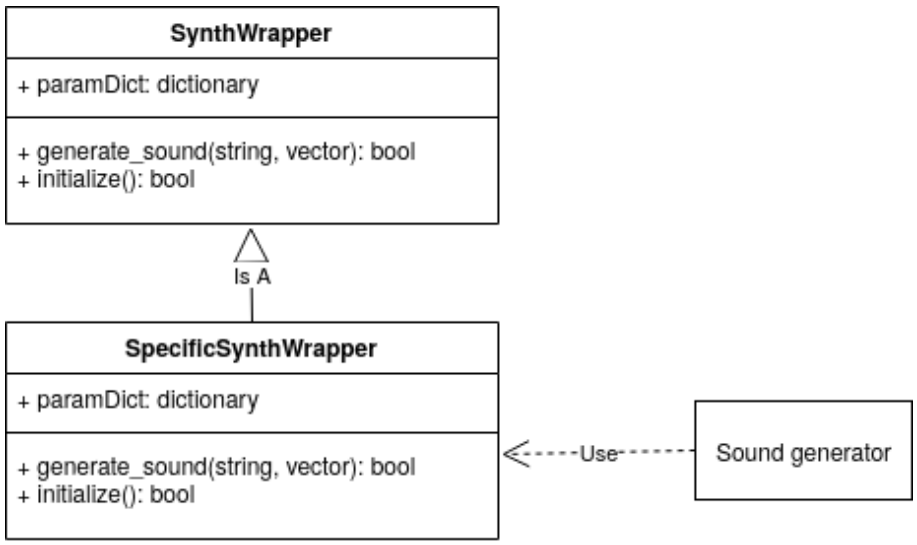

Figure 4.3: UML class diagram for the sound generator.

A wrapper is a function that is meant to abstract the execution of the synthesizer away from the parameter sweeper. The `SynthWrapper` class defines some methods and attributes that are generalized for every sound generator. They are as follows:

`paramDict`:

> A global list of all parameters that the parameter sweeper should examine. This can be a list, or ideally a dictionary of key/value pairs that correspond to the names and values of the parameters.

`generate_sound`:

A method that takes in a string with an output name, and a vector of parameters to be input into the sound generator that will define the sound. It returns a boolean denoting success or failure of creation.

`initialize`:

An initialization method, just in case the sound generator needs some initial configuration before any sound can be made.

A new class that inherits `SynthWrapper` needs to be created for each sound generator that a user wants to use in the system. This class needs to implement the general attributes and methods defined in `SynthWrapper`. This is illustrated in Figure 4.3 by the `SpecificSynthWrapper` class which uses a sound generator component. Overall this design provides sufficient solutions to requirements **PR 1.1**, **PR 1.2**, and **PR 1.3**.

4.2.2 Feature Extractors

The second component required for the functioning of the parameter sweeper, and the fulfilment of the primary objective, is the feature extractor. Figure 4.4 shows a UML class diagram for a feature extractor wrapper.

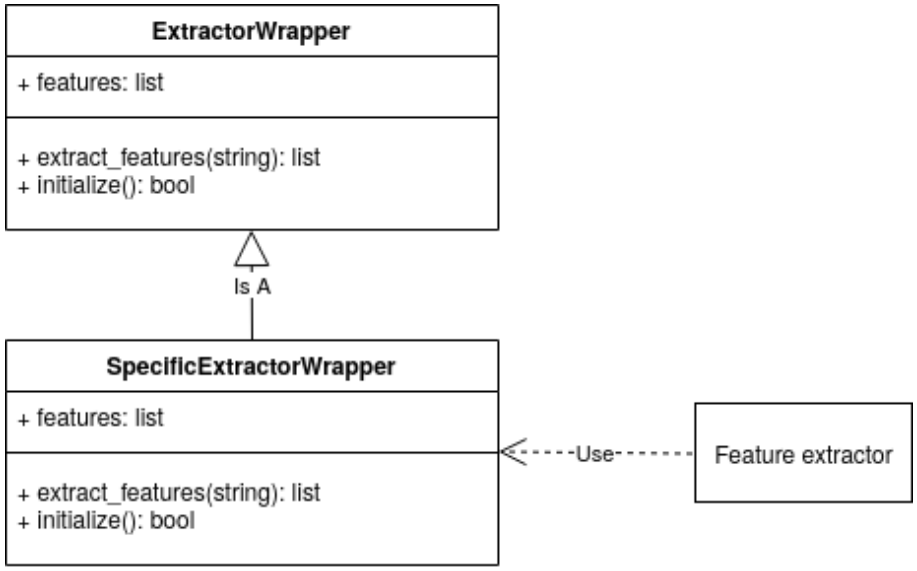

Figure 4.4: UML class diagram for a feature extractor.

The `ExtractorWrapper` and `SpecificExtractorWrapper` are analogous to the `SynthWrapper` and `SpecificSynthWrapper` classes for a sound generator. The general attributes and functions for `ExtractorWrapper` are as follows:
`features`:

A global list of all features that are extracted by the feature extractor.

`extract_features`:

> Method that extracts the features from a sound. It takes a sound filename as an input, and outputs a list of all the features.

This simple design provides solutions to requirements **PR 2.1** and **PR 2.2**. Together the designs for feature extractors and sound generators solve requirement **PR 3**.

4.2.3 Parameter Sweeper

The final component requiring design is the parameter sweeper. A simple class diagram of it is shown in Figure 4.5.

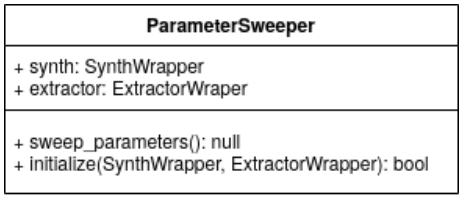

Figure 4.5: UML class diagram for the parameter sweeper.

The parameter sweeper joins a sound generator and a feature extractor in a single workflow. Each combination of parameters defined in the sound generator is used to create a sound, and each of these sounds is input into the feature extractor in the `sweep_parameters` method. All the data from each sound is put together and returned to be used in other components (as shown in Figure 4.2).

Chapter 5

Implementation and Testing

This chapter discusses the specifics of implementation of each part of the system. Similarly to Chapter 4, it firstly discusses the general aspects and follows on discussing the individual components. The priority of implementation has been decided upon based on the priority of the requirements, as described in Chapter 3.

5.1 General Choices

Initially, workflow tools have been researched that might be suitable for the system implementation, such as Apache Taverna. However, upon examining them they proved insufficient and overly complex. The system could potentially be implemented using them in the future. Python was the final choice of programming language for implementation. Many tools that are available in the market have Python wrappers, therefore machine learning, data visualisation, and some sound generators are easy to implement. It also allows for programming using object oriented programming paradigms.

5.2 Sound generators

Based on the designs laid out in Chapter 4, a synthesizer wrapper class, `SynthWrapper.py` was created. Based on this, two sound generators were also written.

5.2.1 Synthesizer Wrapper

`SynthWrapper.py` is a short file that defines certain abstract methods and global variables that need to be defined for the parameter sweeper to be able to use the sound generator. Aside from this, the file is very simple so that programmers can easily add various sound generators to the workflow. The

most notable aspect of the class is the parameter definition as shown in Listing 5.1. The entire file can be seen in Appendix D.

Listing 5.1: Sound generator parameter definition from the `SynthWrapper.py` file.

```
8     parameters = [
9         {
10            "name":          "parameter 1",
11            "min":           0,
12            "max":           10,
13            "increment":     1
14        },
15        {
16            "name":          "parameter 2",
17            "min":           2.5,
18            "max":           23.7,
19            "increment":     2.5
20        }
21    ]
```

Each parameter within the synthesizer that the user wants to be analysed by the sweeper needs to be defined. This is shown in Listing 5.1. The parameter is defined as a dictionary of key-value pairs with the following information:

- The name of the parameter, used for labelling and reference.
- The minimum value that the parameter can take.
- The maximum value that the parameter can take.
- And the increment by which to increase the parameter at every step of the parameter sweep.

Both the minimum and maximum values are inclusive. Changing the increment value changes the granularity of the data that is generated. Smaller values create more points in between the minimum and maximum, larger values create less points.

The `generate_sound` method must be defined to use the parameter values correctly. Examples of this are in the next section.

5.2.2 Synthesizers

The final choice for sound generator to be used in this project was Csound. It gives the most flexibility to be included in a programming context, and has an easy to use Python wrapper `ctcsound`. Two sound generators were created using Csound: a simple sine wave generator, and a more complex generator based on the concept of subtractive synthesis.

Simple sine wave generator

The simple generator uses two `oscili` Csound opcodes to generate a sine wave whose frequency is modulated using a low frequency oscillator (LFO).

A LFO is a wave with a frequency lower than 20Hz that is most commonly used to modify a different sound signal. Using a LFO effects such as vibrato (rhythmic frequency modulation) or tremolo (rhythmic amplitude/loudness modulation) can be achieved. In this case, the LFO is used to create a vibrato.

Listing 5.2 shows a snippet of the orchestra definition for that sound generator. The full file, `Csound_SynthWrapper.py`, can be found in Appendix D.

Listing 5.2: Simple synthesizer orchestra definition.

```
16              instr 1
17                  out(oscili(p4/2 + oscili:k(p4/2, p6), p5))
18              endin
```

Each `p` is a parameter input into the synthesizer. Parameters 1 to 3 are reserved for the Csound score, parameter 5 is the sound frequency, and parameter 6 is the LFO frequency. Listing 5.3 shows the parameter definitions for this synthesizer.

Listing 5.3: Simple synthesizer parameter definition.

```
23          self.parameters = [
24              {
25                  "name": "sound frequency",
26                  "min": 27.5,
27                  "max": 7040.0,
28                  "increment": 125.0
29              },
30              {
31                  "name": "LFO frequency",
32                  "min": 2.5,
33                  "max": 7.0,
34                  "increment": 0.125
35              }
36          ]
```

The `generate_sound` method for this synthesizer is simple. It uses the `ctcsound` library to create the Csound instrument, and apply the score that is generated from the input parameters list. The beginning of the score is set to always be the same, as shown in Listing 5.4. The `i` keyword states to play an instrument. The first parameter defines which instrument from the score to play, in this instance instrument 1. The next two parameters define the start and end time of the instrument playback respectively, this is to ensure that all sounds are of the same length.

Listing 5.4: Simple synthesizer score beginning.

```
44          sco_beg = "i 1 0 5 0.5 "
```

Subtractive Synthesizer

The subtractive synthesizer is much more complex, and resembles a real synthesizer more closely than the previous example. Subtractive synthesis starts with a complex sound e.g. a sample or a complex wave like a sawtooth wave, and removes frequencies from it to create a different sound. The removal can be completed in different ways but generally it is achieved by applying another wave to the original complex wave.

The implementation of the subtractive synthesizer follows closely the definition of the "Csound Two-Oscillator Synthesizer" from McCurdy (2015), which is in turn loosely based on the design of the original 1970 Moog Minimoog Model D synthesizer [1]. All the MIDI functionality has been stripped away from that implementation to not interfere with the software based sound generation.

The overall structure of this synthesizer is shown in a schematic format in Figure 5.1. Two oscillators are summed up together in the mixer, then passed through a low pass filter and an amplitude envelope before being output.

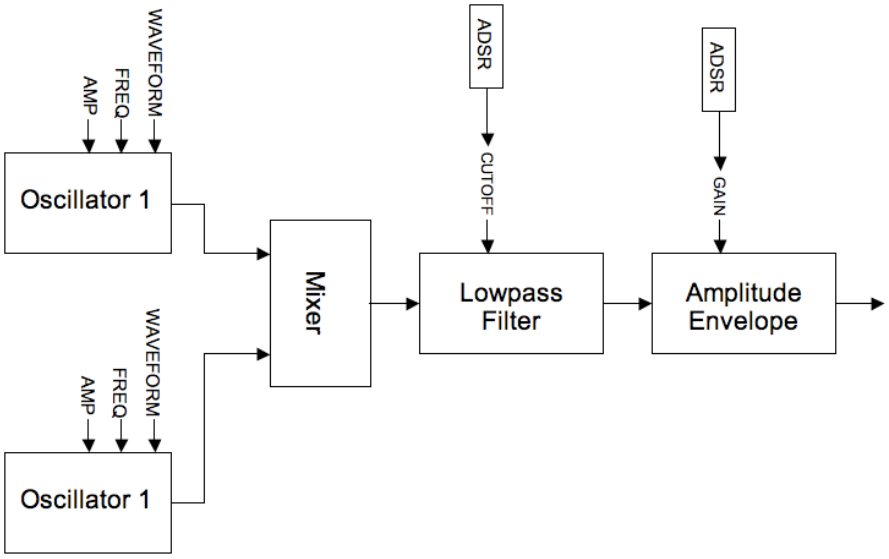

Figure 5.1: Schematic of the "Csound Two-Oscillator Synthesizer" taken from McCurdy (2015). There is an error in the schematic, one of the oscillators should say "Oscillator 2". Either one can be "Oscillator 2" as they are identical to each other.

A total of twenty-one parameters can be input into this synthesizer, alongside the three reserved for Csound scores. The main parameter is the frequency, which defines what note the synthesizer should play. It is defined

[1] `https://en.wikipedia.org/wiki/Minimoog` and `http://www.vintagesynth.com/moog/moog.php`, for the curious

as an integer above zero, where each number represents a key on a standard full-sized piano.

Each oscillator has five parameter inputs:

Amplitude:

The loudness of the generated wave as a percentage value. The synthesizer definition makes sure that it does not exceed 100%.

Waveform:

The type of wave that the oscillator can make. There are three types: a sawtooth wave (a signal that resembles consecutive triangles), a square wave (a binary signal which has only two values, on or off, that rapidly change between those values), and noise which generates white noise rather than any sound.

Pulse Width Modulation:

A parameter specific to the square wave waveform setting. It is the length of the on signal defined as a percentage of each wave cycle.

Octave Displacement:

How many octaves (up or down) should the frequency of the signal of the specific oscillator be shifted.

Tuning Displacement:

A more precise frequency displacement, defined as a value in cents. Each key is divided into 1200 cents, and the tuning displacement value is added on top of the key of the main frequency. This allows the user to obtain frequencies that are not on the standard 12 tone scale. However, it can also allow the user to span the current octave, for example the key of A displaced by 1200 cents is the key of A#.

Both the lowpass filter and the global amplitude envelope have ADSR inputs which are respectively:

Attack time:

The time it takes for the signal to reach the full amplitude. This starts from the start of the signal.

Decay time:

The time it takes for the signal to reach the sustain level. This starts from the end of the attack time.

Sustain level:

The amplitude at which the signal is meant to be held. This amplitude is constant for the remaining duration of the signal while it is still being sent (either the key is pressed or the filter is still applied).

Release time:

The time it takes for the signal to fade out fully. Starts from the moment the signal ends.

On top of which the lowpass filter has two unique parameters:
Cutoff:

The frequency at which the low pass filter starts. Any frequencies above this frequency are attenuated by the filter according to the filter definition.

Resonance:

The increase in amplitude of the cutoff frequency within the signal.

All of these put together define a complex sound generator that closely resembles a commercial synthesizer. The whole file can be seen in Appendix D as it is too long to show here.

5.3 Feature extractors

Similarly to the sound generators, a feature extractor class in the `ExtractorWrapper.py` file has been defined. Based on this, two specific extractors have been created.

5.3.1 Feature extractor wrapper

The wrapper is only a few lines of code that define the necessary method for the parameter sweeper to use. The whole file is shown in Listing 5.5.

Listing 5.5: Extractor wrapper file.

```
1  from abc import ABC, abstractmethod, abstractproperty
2
3  class ExtractorWrapper(ABC):
4      """Interface for an extractor wrapper class"""
5
6      features = ["tempo"]
7
8      @abstractmethod
9      def __init__(self):
10         pass
11
12     @abstractmethod
13     def extract_features(self, input_filename:str):
14         """Extract features from sound given by the filename"""
15         pass
```

All that is required is the `extract_features` method that uses the extractor to obtain data about the sound provided to it.

5.3.2 Essentia based extractors

The choice of feature extractor for this project is based on the ones created for the Essentia project (Bogdanov et al., 2013). This is due to its simplicity of usage and wide range of features that can be extracted. Essentia provides pretrained extractor models that can retrieve both Low Level and High Level Descriptors from the input sound.

The chosen pretrained model is the `essentia_streaming_extractor_music` command line feature extractor. A full description of it is available on the Essentia website[2].

The model itself is simply an executable file that is run using the Python `subprocess` library. It takes in a few arguments that tell it what configuration file to use, what to name the output, etc.

The model uses a YAML configuration profile to define what parameters are meant to be extracted from the sound. Amongst others, the YAML file is used to define: the sample rate of the analysis, the segment of the sound to analyse, sizes of frames for the feature extractor algorithms, etc. It is also used to define what filetype to use as the output for the features. It has been decided to use JSON as the output format for the data, therefore the `extract_features` method needs to be able to handle that output filetype.

Both of the classes described in the rest of this section use a basic four step process to extract the feature data:

1. Call the command line feature extractor as a subprocess
2. Create a temporary file with all the extracted features
3. Parse the temporary file and retrieve all the features that the class is interested in
4. Return the retrieved features

High Level Descriptors

The `Essentia_HLD_ExtractorWrapper.py` file contains the class that extracts High Level Descriptors from the input sound. It makes use of some pretrained classifiers provided by Essentia to extract mood probabilities for the 7 following mood classifications:

- “acoustic”
- “aggressive”

[2] `https://essentia.upf.edu/streaming_extractor_music.html`

- "electronic"
- "happy"
- "party"
- "relaxed"
- and "sad"

Each mood classification is given a probability value between 0 and 1, and values above 0.5 are assigned that mood, otherwise it is assigned an opposite value e.g. if a sound had an acoustic classification probability below 0.5, it would be classified as "not_acoustic". To clean up the returned data the negated values are reversed and the final probability value is simply the probability of it being the positive classification. Each sound processed with this feature extractor is given a probability value for each mood classification. 7 features per sound in total. The full file can be seen in Appendix D.

The classifiers themselves work based on `.history` files that are available to download from the Essentia website. Each mood has a `.history` file associated with it, which is added to the YAML configuration file to instruct the executable to extract that HLD. The mood configuration is applied to the `essentia_streaming_extractor_music` executable to extract the features.

Low Level Descriptors

LLDs are extracted using the class in the `Essentia_LLD_ExtractorWrapper.py` file. It works in a very similar way to the HLD extractor from the previous section, but it uses a general configuration file that extracts low level features from the input sound. A total of 313 low level features are extracted per sound. Amongst others they include: loudness, zero crossing rate (the rate at which the sound signal crosses the zero amplitude mark), and Mel Frequency Cepstral Coefficients. The full list and descriptions of the low level features can be found on the Essentia website[3].

5.4 Parameter Sweeper

Code for the parameter sweeper component is in the `ParameterSweeper.py` file, which can be seen in full in Appendix D. It is initialized with a `SynthWrapper` and a `ExtractorWrapper` object which are the sound generator that is meant to be analysed, and the feature extractor that is analysing the sound.

The `sweep_parameters` method is called to initiate the parameter sweep. To support any number of sound generator parameters this method is based on recursion. For each parameter encountered, the parameter is set and if

[3] `https://essentia.upf.edu/streaming_extractor_music.html`

more parameters exist the method is called recursively to examine the next one in the list. A sound is generated if the current parameter is the last one in that list. Algorithm 1 shows pseudocode for this method.

Algorithm 1 Parameter sweeper method algorithm.

```
function SWEEP_PARAMETERS(paramNum, paramList)
    currentParameter ← parameter[p]
    steps ← paramRange ÷ paramIncrement
    for all steps do
        p ← paramMin + (paramIncrement × step)
        inputParamList[param] ← p
        if lastParam then
            sound ← SYNTH.GENERATE_SOUND(paramList)
            features ← EXTRACTOR.EXTRACT_FEATURES(sound)
            return features
        else
            paramNum ← paramNum + 1
            SWEEP_PARAMETERS(paramNum, paramList)
        end if
    end for
end function
```

The recursion based sweeping method and repluggable object based sound generators create a framework in which any sound generator can be analysed using the parameter sweeper, as long as it can be defined according to the `SynthWrapper` class inheritance.

5.5 Testing Script

Alongside the above classes, a testing script has also been implemented to run the parameter sweeper, and examine the secondary objectives of this project. It has been split into parts specific to the testing of certain aspects of the system. Not all the parts are very significant therefore this section of the report only focuses on two of them: machine learning models, and data visualisation methods.

5.5.1 Machine Learning Models

As discussed in section 4.1, the choice of machine learning algorithm depends on the data types returned by the feature extractors. Now that the feature extractors are defined the models can be chosen.

The HLD extractor returns a list of nominal data values, truth values based on some probability confidence. Models such as logistic regression, and

Support Vector Classification, are suited best for handling this type of data. The LLD extractor returns continuous data, therefore regression models such as linear regression and Support Vector Regression will be used with this data.

The Python library `scikit-learn 0.22.2` has been chosen as it provides all of the above models in an easy to use way.

The data extracted from the parameter sweep is split into testing and training data. Both testing and training is simply a `scikit-learn` function call.

5.5.2 Data Visualisation

Data visualization is also done using visualization libraries. For this the `matplotlib` and `plotly` libraries have been chosen.

The visualisation process involves extracting the feature and parameter labels, and the data for each label, from the final parameter sweeper data and plotting them on a graph of choice. The graphs of choice and what they show are discussed in more detail in Chapter 6.

Only the HLD data has been used in the visualisation tests due to its smaller number of features. Without appropriate dimensionality reduction methods, the LLD data is simply too complex to create a meaningful representation using standard data visualisation techniques.

Chapter 6

Results and Discussion

This chapter outlines the data collection methods, and discusses the results obtained from the data.

6.1 Data Gathering

The parameter sweeper was used to extract data from both of the implemented sound generators. That data was then processed in several machine learning models, and visualized using graphs.

6.1.1 Sound generator data

The simple synthesizer was analysed using both feature extractors. Parameter increment settings produce a total of 2109 data points. As there is a limited number of parameters in this sound generator they can be analysed fully with a reasonable level of granularity of each parameter.

The subtractive synthesizer was more complex to analyse and had to be simplified due to the time restrictions of the project. Section 7.1 discusses this in more detail. The synthesizer was limited to a maximum of 7 parameters. Additionally, only meaningful parameters were chosen as the parameters to analyse. This means that the octave and tuning displacement were skipped as they do not provide ample variation in the sound, but they increase the runtime of the sweeper by a significant amount. The number of parameters was increased in several stages, with each stage returning the data it obtained in the sweep in a fashion similar to anytime algorithms. Only the low level descriptor feature extractor was used on this synthesizer due to the time limitations of this project. The parameter increase stages are as follows:

Stage 1: One parameter, the key pressed, varying between key 10 (F#1 (Goss, 2019), 46.24930Hz (Suits, 1998)) and 80 (E7, 2637.020Hz), giving a total of 71 data points.

Stage 2: The previous parameters plus the amplitude of Oscillator 1, giving a total of 350 data points.

Stage 3: The previous parameters plus the type and pulse width modulation (PWM) of Oscillator 1. These two parameters have to be modified together for meaningful analysis of them therefore both have been added in this stage. In order to decrease the runtime of the parameter sweep this stage is split into two:

- Stage 3a: The type of Oscillator 1 is limited to the sawtooth and noise signals, with no change in PWM.
- Stage 3b: The type of Oscillator 1 is set to the square wave, and PWM is set to change within a range of 0.2 to 0.8 with intervals of 0.2. Since PWM is a percentage value, both 0 and 1 create a flat off or on signal that does not make a meaningful sound, therefore these values are omitted.

In total this stage produces 2130 data points.

Stage 4: The previous parameters plus the amplitude of Oscillator 2. Up until this stage the amplitude was set to zero, therefore no signal was added from it. The same splitting method from stage 3 is applied to this stage, generating a total of 12780 data points.

Stage 5: The previous parameters plus the type and PWM of Oscillator 2. Similarly to the previous two stages, this stage is split into four sub stages:

- Stage 5a: Oscillator 1 and 2 limited to sawtooth and noise types, with no change in PWM.
- Stage 5b: Oscillator 1 limited to sawtooth and noise types, Oscillator 2 set to the square wave type with PWM.
- Stage 5c: Oscillator 1 set to the square wave, and Oscillator 2 limited to sawtooth and noise waves.
- Stage 5d: Oscillator 1 and 2 set to the square wave, with PWM change in both.

In total this stage produces 76680 data points.

The remaining parameters of the subtractive synthesizer were set to be non-intrusive with the sound that is produced. The ADSR of both the global lowpass filter and amplitude envelopes are set to `0.05, 0.95, 1, 1` respectively. The resonance of the lowpass filter is set to zero and the cutoff to 20kHz, which is the upper limit of the human range of hearing (Rosen, Howell and Bartram, 1990). Therefore, the filter should not interfere at all with the final sound.

6.1.2 Data processing

The two extracted feature data types were analysed using distinct methods.

High level descriptors were input into a logistic regression model and a Support Vector Classifier to determine the suitability of those models in classifying high level features using synthesizer parameters. Various graphs were also created to visualise the relationship between the parameters and features, and the relationships between features themselves. They include 3D scatter plots, parallel coordinate plots, and radar charts (a.k.a. spider diagrams).

Low level descriptors were processed using linear regression and Support Vector Regression. An inverse model was created, with the feature data as the independent variable and synthesizer parameters as the dependent variable. This is to test the suitability of those models in predicting parameter values given low level features from the generated sounds.

6.2 Results Analysis

This section displays and discusses the results obtained from processing the data. Different metrics were used to analyse HLDs and LLDs.

6.2.1 High Level Descriptors

Confusion Matrices

The main method of HLD results analysis was using confusion matrices. For each classification, the true positive and negative, and false positive and negative rates were calculated and used to compute precision and recall rates. These rates are displayed in Table 6.1 for logistic regression and Table 6.2 for support vector classification. Raw results for true and false classifications can be seen in Appendix B. The *acoustic* class has been omitted from the analysis as it only has *not_acoustic* classifications, therefore the model could not be trained on it. Given that the output sounds are from a synthesizer it is expected that they are all labelled as *not_acoustic*, therefore this result can be seen as positive.

Accuracy, precision, and recall are defined in Equations 6.1a, 6.1b, and 6.1c respectively, where T=True, F = False, P = Positive, and N = Negative.

$$A = \frac{TP + TN}{TP + FP + TN + FN} \tag{6.1a}$$

$$P = \frac{TP}{TP + FP} \tag{6.1b}$$

$$R = \frac{TP}{TP + FN} \tag{6.1c}$$

Class	Accuracy	MSE	Positive			Negative		
			Precision	Recall	F	Precision	Recall	F
acoustic	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
aggressive	91.44%	0.097	91.44%	100%	0.9553	0%	0%	0.000
electronic	99.05%	0.009	99.05%	100%	0.9952	0%	0%	0.000
happy	84.02%	0.168	84.02%	100%	0.9132	0%	0%	0.000
party	79.15%	0.225	79.15%	100%	0.8836	0%	0%	0.000
relaxed	98.82%	0.012	98.82%	100%	0.9941	0%	0%	0.000
sad	98.10%	0.019	98.10%	100%	0.9904	0%	0%	0.000
Overall Accuracy:			91.76%					

Table 6.1: Logistic Regressor results for each HLD class.

Class	Accuracy	MSE	Positive			Negative		
			Precision	Recall	F	Precision	Recall	F
acoustic	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
aggressive	90.05%	0.100	90.05%	100%	0.9476	0%	0%	0.000
electronic	99.05%	0.009	99.05%	100%	0.9952	0%	0%	0.000
happy	84.94%	0.156	84.94%	100%	0.9186	0%	0%	0.000
party	65.10%	0.306	65.10%	100%	0.7886	0%	0%	0.000
relaxed	98.82%	0.012	98.82%	100%	0.9941	0%	0%	0.000
sad	98.10%	0.019	98.10%	100%	0.9904	0%	0%	0.000
Overall Accuracy:			89.34%					

Table 6.2: Support Vector Classifier results for each HLD class.

The accuracy for each class for both types of classifiers is high, except for the *party* class which scores significantly lower than the other classes. The *happy* class also has a lower accuracy rate. The above data is consistent with model accuracy data provided by Essentia (Anon, 2020), which shows lower accuracy for those two classes (*party*=88.38%, *happy*=83.27%) compared with the other classes in this mood set (average=91.58%). It is also consistent with the positive F value as defined by Chinchor (1992), which combines the recall and precision values into one measure. Values of F closer to 1 signify an accuracy and recall closer to 100%.

This might imply a good performance of the classifiers on the data, however looking at the precision and recall rates themselves, and the negative classification F value, suggests otherwise. This data implies that the classifiers label everything as the positive class, which obtains a high accuracy rate simply because of the data mostly being that positive class. The 0% negative recall in all classes shows this to be true. It is therefore unclear whether or not the classifiers have performed well. They might simply be overtrained on the positive classes which lead it to class positive for all sounds. A solution to this might include class balancing, which is discussed in greater detail in Section 7.1.

Visualisation

A secondary objective of the project was to visualize the feature data obtained from the sounds. This has been done for the HLD data obtained from the simple synthesizer in the following few figures.

Figure 6.1 shows a parallel coordinate graph for features extracted from sounds created by the simple synthesizer. Only one parameter was analysed for this figure. Figure 6.2 shows the same graph with both parameters of the simple synthesizer being analysed. These graphs can display the extent of the synthesizer's capabilities in creating sounds of a certain mood. Having less sounds displayed on it creates a clearer graph. It is difficult to infer any meaning from Figure 6.2, but it shows some links between parameters that can be followed up with further analysis.

A better way of displaying the extent of the synthesizer capabilities is using radar charts, as shown in Figure 6.3. The blue and red lines show the highest and lowest probability values for the features respectively. The shaded blue space above the red line shows the possible probability values that this feature can take. This type of graph profiles a synthesizer according to the features which could be valuable to a potential user. However, in the current state, this graph might be misleading. Displaying the continuous probability values when the classifier models are trained on binary values implies that the machine learning model takes into account the continuous values, which it does not. However, only showing the binary values makes the graph redundant as it does not show the range that each feature can take. This might be only a criticism

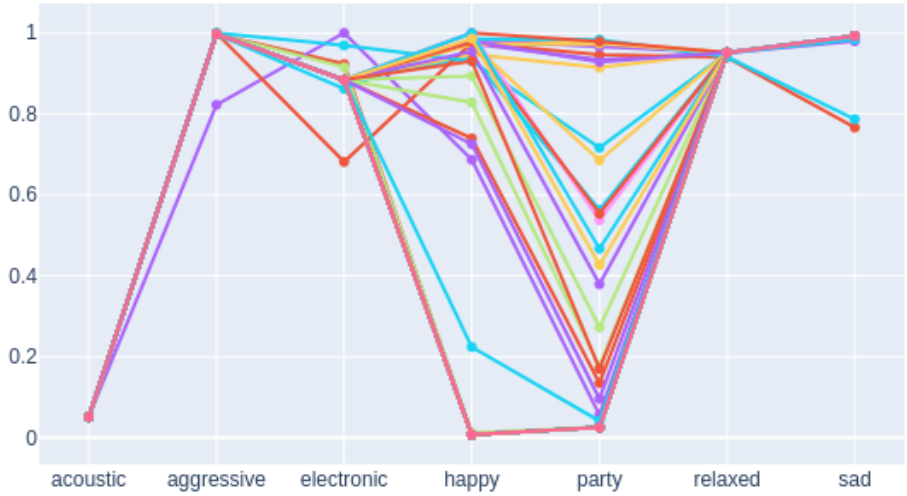

Figure 6.1: A parallel coordinate graph displaying data from the simple synthesizer. Each line is a different sound.

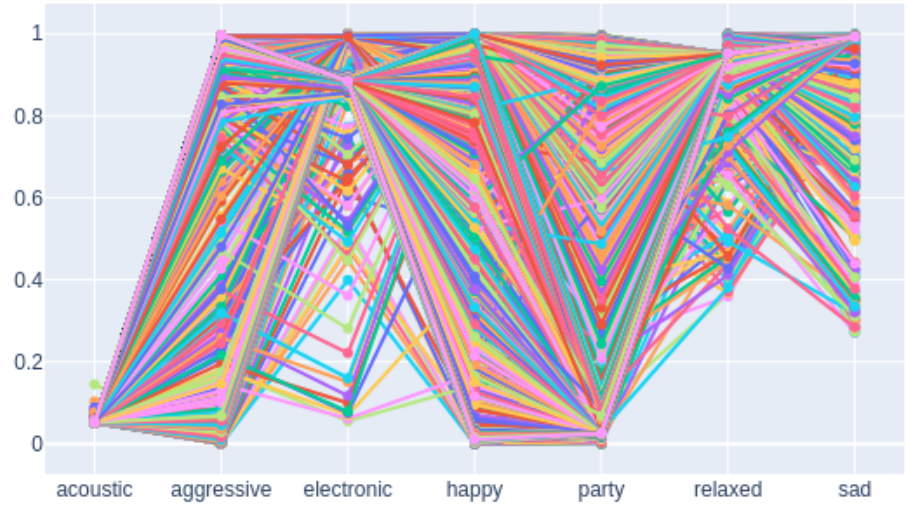

Figure 6.2: A parallel coordinate graph displaying data from the simple synthesizer. Each line is a different sound.

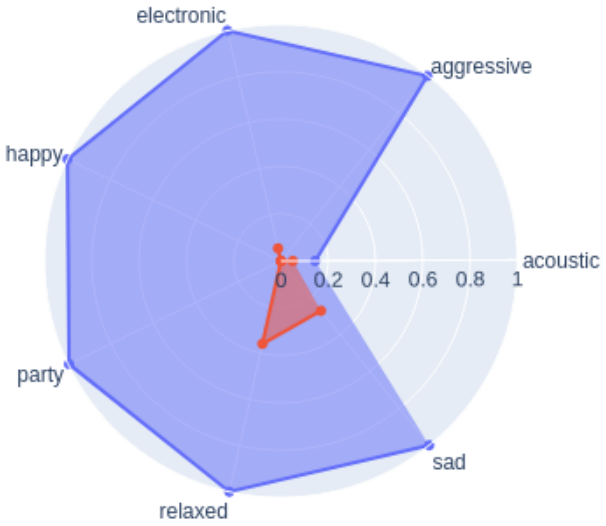

Figure 6.3: A radar chart (spider diagram) displaying the highest and lowest class probabilities for each class extracted from the simple synthesizer.

of the current features being extracted using the HLD extractor. Using a LLD extractor does not create this problem. However, the features then become unnormalized by default (as the probability values only fall between 0 and 1 therefore they do not require normalization) and might be difficult to visualize properly. Also given the current LLD extractor this would not be possible without parameter reduction, as this extractor has too many parameters to effectively display all of them on a single graph.

The final graph that can show useful data to a synthesizer user is displayed in Figure 6.4. It is a 3D plot of the two parameters of the simple synthesizer against a mood classification, in this case of the *party* class. For this class, this graph is effective, as it can show the user what parameters to choose in order to create a sound that is likely to score a high *party* probability. However, this does not work for all classes, simply as some show no discernible trend between the features and the class. This might imply that there is no actual trend between them, which makes further analysis redundant and might indicate low performance of the classifier. Low performance might also be indicated by seeing the overwhelming majority of the data points be skewed towards the positive classification for the classes. This can be seen in 3D plots for the other classes, shown in Appendix C.

An interesting qualitative study would look into whether the mood classifications are concurrent with listeners' opinions of a sound. However, as described in the "Impact of COVID-19" preamble section this was not possible. In the extra files submitted alongside this report, there are two sounds within the "Submission Sounds" folder: the "party.wav" sound created using

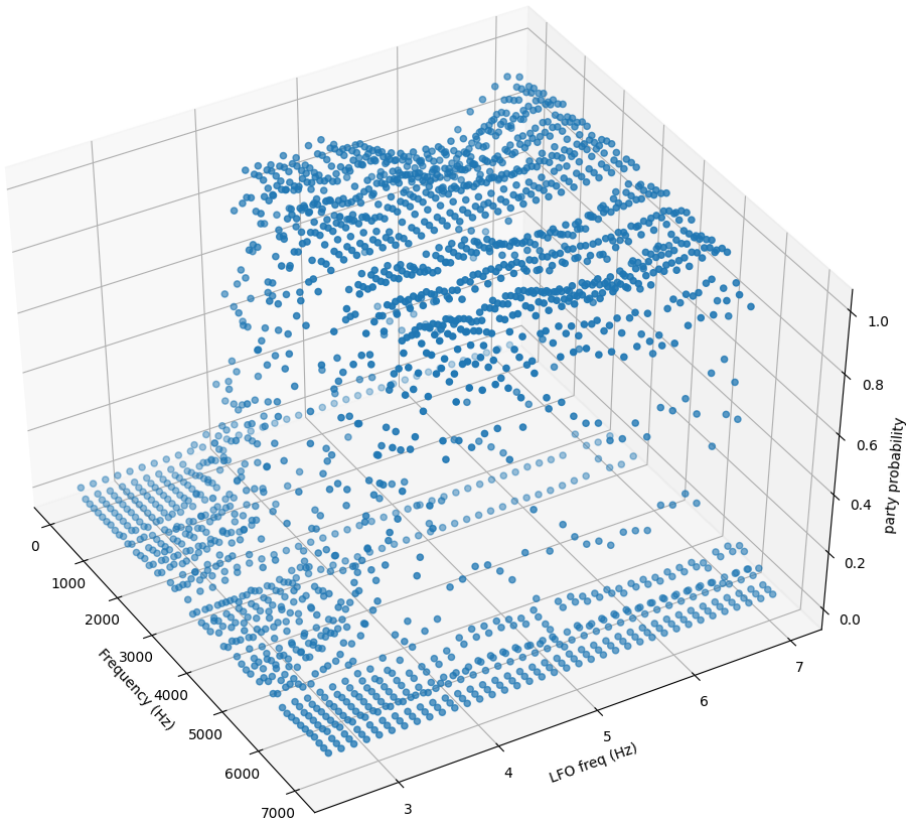

Figure 6.4: A 3D plot of the two parameters in the simple synthesizer and the *party* class.

Regressor	Accuracy	R^2	MSE
Linear	97.39%	0.988	0.366
Support Vector	4.03%	0.116	1.890×10^6

Table 6.3: Regression results for different regressors analysing simple synthesizer data.

the simple synthesizer with a LFO frequency of 3Hz, and a "non-party.wav" sound with a LFO frequency of 6Hz. Both have a sound frequency of 4500Hz.

6.2.2 Low Level Descriptors

LLDs were analysed by training Linear Regression and Support Vector Regression models on the extracted data. Investigating the secondary objective of this project, the synthesizer parameters and LLD features are reversed in order to train the model to predict the synthesizer parameters. The model features are the extracted LLDs and the model labels are the synthesizer parameters. Three measures of performance are used to assess the quality of the models: accuracy, the coefficient of determination (R^2), and the mean squared error (MSE). Accuracy is measured by comparing predicted labels with actual test labels, with a $\pm 10\%$ threshold. The coefficient of determination indicates the goodness of fit by indicating the proportion of the variation in the labels that is explained by the variation in the features (Rawlings, Pantula and Dickey, 1998). The mean squared error is the average squared difference between the predicted labels and the actual test labels.

Simple Synthesizer

Table 6.3 shows the three measures of performance evaluating Linear and Support Vector Regression models when trained on the features extracted from the simple synthesizer. The linear regression model shows much higher performance compared to the Support Vector model in all three measures. Therefore in this, instance the former model would be the best choice for sound generation.

Despite the low performance, the SVR might be promising as it has a positive R^2 value. Experimenting more with Support Vector Regression might yield much better results but some SVR kernels might require data preprocessing, such as dimensionality reduction. This is because the complexity of testing and training them can be $O(m)$ and $O(nm + m^3)$ (Keerthi, Chapelle and DeCoste, 2006) respectively, where n is the number of training/testing samples and m is the number of support vectors within the SVR.

	Accuracy	R^2	MSE
Stage 1	93.3%	0.974	11.945
Stage 2	0.0%	-2.288	0.018
Stage 3	4.5%	0.790	12.946
Stage 4	1.3%	0.767	2.701
Stage 5	0.0%	0.421	5.761

Table 6.4: Linear Regressor results for 5 subtractive synthesizer stages.

	Accuracy	R^2	MSE
Stage 1	13.3%	-0.007	470.012
Stage 2	0.0%	-1.622×10^{30}	188.600
Stage 3	0.0%	0.002	104.521
Stage 4	0.0%	0.143	77.105
Stage 5	0.0%	0.054	57.679

Table 6.5: Support Vector Regression results for 5 subtractive synthesizer stages.

Subtractive Synthesizer

The subtractive synthesizer has been studied for each stage of data extraction separately, both with a linear regression model and a Support Vector Regression Model. Table 6.4 and Table 6.5 show results for linear regression and SVR respectively.

The difference between linear and support vector models is the same as for the simple synthesizer, the overall performance for the linear model is much greater than for the SVR. It is interesting to see what happens as the number of synthesizer parameters increases. In both of the models, the accuracy and R^2 value drops but the mean squared error decreases. A possible explanation for this is that the increase in label space causes the training data to be more sparse. The label space increases to $\mathbb{R}^7$ while the feature space, although large, remains the same size at $\mathbb{R}^{313}$. The large feature space could also have caused the models to become overtrained, they explain the training data perfectly, but fail to create a usable model that extrapolates this explanation to unknown data. It is a typical example of the curse of dimensionality, the decrease of model performance with the increase of dimensions that the model is required to span.

In both cases the oscillator amplitude was challenging to model. There is a large drop of performance between Stages 1 and 2, and then the performance improves in Stage 3. This suggests no LLD accurately describes the amplitude

of the sound. Essentia documentation[1] describes some loudness features that are extracted. Examining Stage 2 data shows that the amplitude of oscillator 1 does correlate with both loudness features present in the extracted data, `loudness_ebu128_momentary` and `loudness_ebu128_short_term` (correlation coefficients of 0.8727 and 0.8568 respectively). This suggests that other features most likely act as noise that disrupts this correlation, and it would benefit to remove them from the model if only using two synthesizer parameters.

6.2.3 Sound generation from Low Level Descriptor training

As a final test of the second secondary objective, a short script was written to generate random features within the bounds of the ones extracted. This was done using the Python `random` library seeded with the number 0. The random features are then input into the trained model to obtain parameter predictions. Models used for this are the simple synthesizer linear regression model, and the Stage 1 subtractive synthesizer linear regression model.

This has been found to have mixed results. The subtractive synthesizer model generally gives reasonable predictions. 60% of the predictions are within bounds of the 71 possibilities that are part of the parameter that is being analysed. Simple synthesizer analysis does not give good data. All of the predicted parameters are out of bounds of the training parameters, therefore none of the predicted values would create meaningful sounds. The prediction data for both models can be found in Appendix B.

These results imply that random assignment of features does not generate meaningful parameters, which can imply relationships, or correlations between the features themselves are broken by random assignment of those features. This would make dimensionality reduction for this set of features a promising avenue for further research, as this would collate the features with relationships between them together and potentially allow random assignment of them to generate meaningful sounds. Unfortunately this was out of scope for this project.

[1] `https://essentia.upf.edu/streaming_extractor_music.html`

Chapter 7

Conclusions and Further Work

This project has set out with one primary and two secondary objectives. The primary objective was to develop a framework that would be able to sweep through parameters of a software synthesizer and extract features from it for further analysis. This has been successful, as a framework is designed in Chapter 4 and implemented in Chapter 5, although some improvements are required which are discussed in more detail in the next section.

The first secondary objective set out to visualise the extracted features to provide some information to a potential synthesizer user about the properties of the synthesizer. The potential of this is shown in Chapter 6, where three different types of graphs display the High Level Descriptor data extracted from a simple sound generator. The results are not perfect, but they are promising and further research into this area might prove to create useful representations of synthesizers.

The second secondary objective aimed to use machine learning models to learn the relationship between parameters input into software synthesizers and the features that are extracted from sounds made by them. This has yielded mixed results, with some models showing greater performance than others, and some parameters being more explainable by the features extracted from the sounds. This problem has been found to be more complex than initially assumed, and requires further research to show good results. However, initial results appear promising and more detailed research into features and models could create a good solution to this problem.

7.1 Further work

This final section of this report lays out some possible avenues of further work that could improve this system. They are ranked in order of importance, with the first being direct improvements to the performance and efficiency of the

system, and the latter provide some extra ideas that were out of scope of this project due to the resource limitations.

Parameter Reduction

A point that came up often when discussing the results in this report is that the Low Level Descriptors had too many features, either to be visualised or to describe the synthesizer parameters without introducing noise into the model. Reducing the number of parameters in LLD contexts would be a good avenue of research that would improve upon both secondary objectives of the project. This could either be done through the development of simpler LLD feature extractors that would use less features, or if all the features are required, using dimensionality reduction methods such as Principle Component Analysis (PCA) would shrink the dimensions of the feature space. Research done by Howley et al. (2006) shows an increase in machine learning model performance after reducing high dimensional data using PCA and thus might be a good starting point in further research in this area.

Algorithm Optimisation

The parameter sweeper algorithm described in Chapter 5 is sequential, therefore the runtime complexity of it grows linearly with the increase in size of the synthesizer parameter space. However, the size of the synthesizer parameter space increases exponentially with the introduction of new parameters into it. Therefore, the runtime of the sweeper algorithm also increases exponentially with the increase of the parameters within the sound synthesizer that is currently being analysed. The sweeper algorithm needs to be made more efficient, possibly through the usage of parallelization, so the runtime can be kept low when synthesizers become more complex. All the parameters of the subtractive synthesizer described in Chapter 5 could not be analysed simply because the parameter space of that synthesizer becomes too large, and runtime would become enormous. As an example, if Stage 5 was to be analysed fully it would create a total of 302400 data points. Assuming that each point is created in one second (in practice they took less than that, closer to 2.5 points per second), it would take a total of 84 hours (3.5 days) to complete the parameter sweep. The sweeping problem lends itself well to parallelization, in fact, it could be considered an "embarrassingly parallel" problem of parameter study as defined by Foster (1995).

The machine learning models used in the secondary objective would also benefit a lot from optimisation. Support Vector Machines are known to take a long time to train and test, as already mentioned in Section 6.2.2. Models that require less time to be trained and tested would be much more suited, as they would allow more realtime analysis of synthesizers. Even special types of SVMs such as Reduced SVMs (Lee and Mangasarian, 2001) could be used for

this purpose.

Improvement and usage of classifiers

The performance of logistic regression and Support Vector Classifiers for HLD was found to be good, however no solution to the inverse problem using HLDs was found. This project did not develop a way for a machine learning model to use HLDs to produce sound generator parameters. This could be possible through the use of generative machine learning models, or by breaking down the HLD into quantifiable units rather than positive or negative classes, perhaps by utilizing the probability values themselves rather than the classes.

In addition to the above, despite the classifier performance being good it was not perfect. The issues of imbalanced classes could be solved through class balancing methods e.g. by generating samples of data in the imbalanced class using methods such as SMOTE (Chawla et al., 2011).

Further studies into Data Visualization

More experimentation with data visualisation might yield better results and give potential synthesizer users more information about their tool of choice.

Creative tool creation

Combining the study into data visualisation, and a well-trained machine learning model, could create a graph-based sound generation tool somewhat similar to the UPIC[1] created in 1977 by Iannis Xenakis (Lohner and Xenakis, 1986). A user would manipulate a graph, possibly a parallel coordinate graph similar to the one shown in Figure 6.1, that is connected to a machine learning model which is trained on sound features and parameters of a sound generator. The model would generate parameters in real time and apply them to the sound generator to create real time compositions. This could potentially be possible by translating the parameters into a MIDI signal which can be sent in real time to the sound generator. Csound would be a good candidate for a sound generator in this project as it can handle realtime input MIDI signals.

[1] `https://en.wikipedia.org/wiki/UPIC`

Bibliography

Aljanaki, A., Yang, Y.H. and Soleymani, M., 2017. Developing a benchmark for emotional analysis of music. *Plos one*, 12(3), pp.1–22. Available from: https://doi.org/10.1371/journal.pone.0173392.

Amatriain, X., 2005. *An object-oriented metamodel for digital signal processing with a focus on audio and music*. Ph.D. thesis. Universitat Pompeu Fabra. Available from: https://doi.org/http://hdl.handle.net/10803/667051.

Anon, 2020. *Essentia High Level Model Accuracies*. Available from: https://essentia.upf.edu/svm_models/accuracies_v2.1_beta1.html [Accessed 30th April 2020].

Atcheson, M., Sethu, V. and Epps, J., 2017. Gaussian Process Regression for Continuous Emotion Recognition with Global Temporal Invariance. In: N. Lawrence and M. Reid, eds. *Proceedings of IJCAI 2017 Workshop on Artificial Intelligence in Affective Computing*. PMLR, *Proceedings of Machine Learning Research*, vol. 66, pp.34–44. Available from: http://proceedings.mlr.press/v66/atcheson17a.html.

Bogdanov, D., Wack, N., Gómez, E., Gulati, S., Herrera, P., Mayor, O., Roma, G., Salamon, J., Zapata, J.R. and Serra, X., 2013. ESSENTIA: an Audio Analysis Library for Music Information Retrieval. *International Society for Music Information Retrieval Conference (ISMIR'13)*. Curitiba, Brazil, pp.493–498. Available from: http://hdl.handle.net/10230/32252.

Chawla, N., Bowyer, K., Hall, L. and Kegelmeyer, W., 2011. SMOTE: Synthetic Minority Over-sampling Technique. *arxiv.org*, 16. Available from: http://search.proquest.com/docview/2086828282/.

Chinchor, N., 1992. Muc-4 evaluation metrics. *Proceedings of the 4th conference on message understanding*. USA: Association for Computational Linguistics, MUC4 '92, p.22–29. Available from: https://doi.org/10.3115/1072064.1072067.

Eyben, F., 2016. *Real-time Speech and Music Classification by Large Audio Feature Space Extraction*, Springer Theses, Recognizing Outstanding Ph.D. Research. 1st ed. Cham: Springer International Publishing.

Eyben, F., Wöllmer, M. and Schuller, B., 2009. OpenEAR - Introducing the Munich open-source emotion and affect recognition toolkit. *Proceedings - 2009 3rd international conference on affective computing and intelligent interaction and workshops, acii 2009.*

Foster, I., 1995. *Designing and building parallel programs : concepts and tools for parallel software engineering.* Reading, Mass. ; Wokingham: Addison-Wesley.

Goss, C., 2019. *Octave Notation.* Available from: `http://www.flutopedia.com/octave_notation.htm` [Accessed 27th April 2020].

Hevner, K., 1935. Expression in music: a discussion of experimental studies and theories. *Psychological review*, 42(2), pp.186–204.

Howley, T., Madden, M.G., O'Connell, M.L. and Ryder, A.G., 2006. The Effect of Principal Component Analysis on Machine Learning Accuracy with High Dimensional Spectral Data. In: A. Macintosh, R. Ellis and T. Allen, eds. *Applications and Innovations in Intelligent Systems XIII.* London: Springer London, pp.209–222.

Huron, D., 2000. Perceptual and Cognitive Applications in Music Information Retrieval. *ISMIR.*

Keerthi, S.S., Chapelle, O. and DeCoste, D., 2006. Building Support Vector Machines with Reduced Classifier Complexity. *J. mach. learn. res.*, 7, p.1493–1515.

Kim, H., Moreau, N. and Sikora, T., 2006. Low-level descriptors. *MPEG-7 Audio and Beyond.* Chichester, UK: John Wiley & Sons, Ltd, pp.13–57.

Laurier, C. and Herrera, P., 2007. Audio music mood classification using support vector machine. *International Society for Music Information Research Conference (ISMIR).* Available from: `files/publications/b6c067-ISMIR-MIREX-2007-Laurier-Herrera.pdf`.

Laurier, C., Meyers, O., Serrà, J., Blech, M., Herrera, P. and Serra, X., 2010. Indexing music by mood: Design and integration of an automatic content-based annotator. *Multimedia tools appl.*, 48, pp.161–184. Available from: `https://doi.org/10.1007/s11042-009-0360-2`.

Lazzarini, V., 2016. *Csound : A Sound and Music Computing System.* 1st ed. Cham: Springer International Publishing : Imprint: Springer.

Lee, Y.J. and Mangasarian, O., 2001. RSVM: Reduced Support Vector Machines. *Society for Industrial and Applied Mathematics. Proceedings of the SIAM International Conference on Data Mining*. Philadelphia: Society for Industrial and Applied Mathematics, pp.1–17. Available from: `http://search.proquest.com/docview/940856661/`.

Li, T. and Ogihara, M., 2003. Detecting Emotion in Music. *Proc. ismir 2003; 4th int. symp. music information retrieval*, 2003.

Lie Lu, Liu, D. and Hong-Jiang Zhang, 2006. Automatic mood detection and tracking of music audio signals. *IEEE Transactions on Audio, Speech, and Language Processing*, 14(1), pp.5–18. Available from: `https://doi.org/10.1109/TSA.2005.860344`.

Lohner, H. and Xenakis, I., 1986. Interview with iannis xenakis. *Computer music journal*, 10(4), pp.50–55.

Markov, K. and Matsui, T., 2014. Music Genre and Emotion Recognition Using Gaussian Processes. *Ieee access*, 2, pp.688–697. Available from: `https://doi.org/10.1109/ACCESS.2014.2333095`.

McCurdy, I., 2015. *Subtractive synthesis*. Available from: `https://write.flossmanuals.net/csound/b-subtractive-synthesis/` [Accessed 20th April 2020].

Nalini, N. and Palanivel, S., 2013. Emotion recognition in music signal using aann and svm. *International journal of computer applications*, 77, pp.7–14. Available from: `https://doi.org/10.5120/13364-0958`.

Peeters, G. and Rodet, X., 2004. *A large set of audio feature for sound description (similarity and classification) in the CUIDADO project*. Ircam, Analysis/Synthesis Team, 1 pl. Igor Stravinsky, 75004 Paris, France.

Posner, J., Russell, J.A. and Peterson, B.S., 2005. The circumplex model of affect: An integrative approach to affective neuroscience, cognitive development, and psychopathology. *Development and psychopathology*, 17(3), p.715–734. Available from: `https://doi.org/10.1017/S0954579405050340`.

Rawlings, J., Pantula, S. and Dickey, D., 1998. *Applied regression analysis: A research tool*, Springer Texts in Statistics. New York, NY: Springer New York.

Rosen, S., Howell, P. and Bartram, J.F., 1990. *Signals and Systems for Speech and Hearing*. Leiden, The Netherlands: Brill.

Russell, J.A., 1980. A circumplex model of affect. *Journal of Personality and Social Psychology*, 39(6), pp.1161–1178.

Schedl, M., Gómez, E. and Urbano, J., 2014. Music information retrieval: Recent developments and applications. *Foundations and trends in information retrieval*, 8, pp.127–261. Available from: `https://doi.org/10.1561/1500000042`.

Soleymani, M., Aljanaki, A., Yang, Y.H., Caro, M.N., Eyben, F., Markov, K., Schuller, B.W., Veltkamp, R., Weninger, F. and Wiering, F., 2014. Emotional Analysis of Music: A Comparison of Methods. *Proceedings of the 22Nd ACM International Conference on Multimedia*. New York, NY, USA: ACM, MM '14, pp.1161–1164. Available from: `https://doi.org/10.1145/2647868.2655019`.

Song, Y., Dixon, S. and Pearce, M.T., 2012. Evaluation of Musical Features for Emotion Classification. *Ismir*.

Sorussa, K., Choksuriwong, A. and Karnjanadecha, M., 2017. Acoustic Features for Music Emotion Recognition and System Building. *Proceedings of the 2017 International Conference on Information Technology*. New York, NY, USA: ACM, ICIT 2017, pp.413–417. Available from: `https://doi.org/10.1145/3176653.3176709`.

Stevens, S., Volkmann, J. and Newman, E., 1937. A scale for the measurement of the psychological magnitude pitch. *Journal of the acoustical society of america*, 8(3), pp.185–190.

Suits, B., 1998. *Frequencies of Musical Notes, A4 = 440Hz*. Available from: `https://pages.mtu.edu/~suits/notefreqs.html` [Accessed 27th April 2020].

Tzanetakis, G., 2009. Music Analysis, Retrieval and Synthesis of Audio Signals MARSYAS. *Proceedings of the 17th ACM International Conference on Multimedia*. New York, NY, USA: ACM, MM '09, pp.931–932. Available from: `https://doi.org/10.1145/1631272.1631459`.

Widmer, G., Dixon, S., Knees, P., Pampalk, E. and Pohle, T., 2008. From Sound to Sense via Machine Learning. In: R. Bresin, M. Karjalainen, A. Kanerva, T. Mäki-Patola, A. Huovilainen, S. Jordà, M. Kaltenbrunner, G. Geiger, R. Bencina, A. Götzen, P. Polotti and D. Rocchesso, eds. *Sound to Sense, Sense to Sound. A State of the Art in Sound and Music Computing*. Logos Verlag Berlin GmbH, chap. 4, pp.161 – 194.

Wilson, S., Cottle, D. and Collins, N., 2011. *The SuperCollider Book*. The MIT Press.

Yang, Y., Lin, Y., Su, Y. and Chen, H.H., 2007. Music Emotion Classification: A Regression Approach. *2007 IEEE International Conference on*

Multimedia and Expo. pp.208–211. Available from: https://doi.org/10.1109/ICME.2007.4284623.

Yang, Y.H., 2011. *Music Emotion Recognition*, Multimedia computing, communication and intelligence. Boca Raton, Fla. : London: CRC ; Taylor & Francis [distributor].

Yang, Y.H. and Chen, H.H., 2012. Machine Recognition of Music Emotion: A Review. *ACM Trans. Intell. Syst. Technol.*, 3(3), pp.40:1–40:30. Available from: https://doi.org/10.1145/2168752.2168754.

Appendix A

12 Point Ethics Checklist

Department of Computer Science
12-Point Ethics Checklist for UG and MSc Projects

Damian Haziak

Student

2019/2020

Academic Year or Project Title

Dr Julian Padget

Supervisor

This form must be attached to the dissertation as an appendix.

Does your project involve people for the collection of data other than you and your supervisor(s)?
~~YES~~ / **NO**

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Have you prepared a briefing script for volunteers?*
YES / NO

Briefing means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?*
YES / NO

All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. Is there any intentional deception of the participants? YES / NO

Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. Will participants be de-briefed?
YES / NO

The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. Will participants voluntarily give informed consent? YES / NO

Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete.

6. Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)? YES / NO

Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. Are you offering any incentive to the participants? YES / NO

The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. Are you in a position of authority or influence over any of your participants? YES / NO

A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. Are any of your participants under the age of 16?
YES / NO

Parental consent is required for participants under the age of 16.

10. Do any of your participants have an impairment that will limit

Their understanding or communication?
YES / NO

Additional consent is required for participants with impairments.

11. Will the participants be informed of your contact details?
YES / NO

All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. Do you have a data management plan for all recorded data? YES / NO

All participant data (hard copy and soft copy) should be stored securely, and in anonymous form, on university servers (not the cloud). If the study is part of a larger study, there should be a data management plan.

Appendix B

Raw results output

Class	Positive		Negative	
	True	False	True	False
acoustic	N/A	N/A	N/A	N/A
aggressive	374	35	0	0
electronic	418	4	0	0
happy	326	62	0	0
party	186	49	0	0
relaxed	417	5	0	0
sad	414	8	0	0

Table B.1: Logistic Regressor raw results for each HLD class.

Class	Positive		Negative	
	True	False	True	False
acoustic	N/A	N/A	N/A	N/A
aggressive	380	42	0	0
electronic	418	4	0	0
happy	327	58	0	0
party	222	119	0	0
relaxed	417	5	0	0
sad	414	8	0	0

Table B.2: Support Vector Classifier raw results for each HLD class.

72.52615112	39.58366465	76.46022493	87.9012116	72.90822494	12.73596852
3.23081691	47.67041004	22.12169145	18.93950416	55.76784081	42.11798378
21.08636391	86.94388213	22.69115892	45.20284465	47.06048366	102.77910831
31.41808627	42.03370932	46.18410852	66.94688335	58.74275679	-6.38669703
3.38314264	53.66579085	41.84399205	112.52162343	35.36018412	50.12738415
45.57636556	46.80685361	57.08651331	50.09317911	33.28404644	80.6951281
28.97751109	33.71973074	49.80035247	55.12335446	11.82892135	1.1203189
85.90428183	19.87681113	78.34626223	41.93222758	134.50548094	45.94282957
92.37830027	11.72125252	24.98196098	33.5076317	24.01105575	123.6465669
109.5619599	129.86360727	41.18491308	99.24852895	44.58970579	23.18405488
90.94359542	49.22093035	105.84295241	76.00908065	82.4470165	85.88180971
21.4263476	50.53291785	49.26450968	11.95044671	54.82768588	68.38257481
33.24906331	99.64158668	52.40820652	91.15556571	40.61849073	32.99095134
78.15663455	85.37962469	40.94710198	18.9729948	87.18256335	42.35861411
76.6131538	30.69685304	80.01397694	29.04063713	61.84784024	-4.18153518
88.06403968	122.24985646	68.60147523	7.43110604	48.43359683	98.92928082
75.92117503	84.0862203	-8.58703888	47.95087504		

Table B.3: Stage 1 Subtractive Synthesizer parameters generated from random assignment of low level features. Each cell in the table is a feature.

Sound freq (Hz)	LFO freq (Hz)	Sound freq (Hz)	LFO freq (Hz)
-1528200.84102268	-183011.75506308	-3880593.9664033	-885919.21556791
-1067172.67521818	543589.21765607	1225321.85132111	1162243.94059937
3871778.64030831	174499.84692748	-1430938.9223657	-1043475.73391064
-636519.48556188	-123649.68052419	-541914.83459745	-982663.37064541
-3142849.68207171	-810977.27194556	-140711.32011257	-3110.95103738
339325.92859574	898512.50526939	-3063603.67808067	799132.14510483
1122405.11247325	-553945.32868218	4798980.31249743	112192.11769756
3395565.88562833	-350489.74227663	-1302479.16677722	604116.76364154
-782965.6706752	-1339907.25019007	-3410777.75199676	656365.68780498
-2133550.96393699	574623.72269242	3612073.93803291	-463255.86989434
-344761.08993936	-693499.33776373	-1307099.66553972	770971.76156194
1355040.86797097	-1681247.94472574	4737428.76629155	-749632.82043709
-1964802.78766937	309245.68898426	3345045.91305304	-245768.40168285
407286.82287674	667772.13852563	-4472399.1170611	-365062.3917811
-1854577.74717071	-487918.5428655	4631537.92285176	325951.80837456
-2244853.74251556	-457553.64071818	50262.78265311	614993.23771311
2806143.45705234	-1247530.2012682	826568.07001343	498421.88249221
126060.59806641	-833949.86293844	3102446.71494133	997535.27504438
2281162.11447261	1150943.04600122	-827775.89181148	677324.29592656
-2551405.20382976	-955112.13449058	2281593.44493495	179265.39031846
2353124.02330276	-302910.37910091	387223.99037289	-424273.69715826
-256520.28488527	71860.83005336	-1583164.71282888	1126764.40891749
767101.96357682	974545.42828695	6639587.06017609	-935820.55486109
3495648.74359537	-1210300.25430109	-3466124.32663778	1525374.95690653
-1826982.49003355	164737.59135061	-631049.5394181	323558.42578199
1744467.47322355	-153796.44003411	2256134.54157578	972197.62471634
-2017208.14398154	1004797.87284125	177123.65171101	-68170.86662671
317249.21912696	-1040483.50790013	631000.54129603	-1447538.78758468
-3145541.95188265	1290561.88573301	-1163078.3509122	254778.1756923
1819823.65588271	-467522.68479589	-1399013.91891798	-723855.53742025
-4496752.20101033	1168009.25422222	190091.07472801	886212.03610691
2223128.03979656	-363339.80619753	-3945041.77275775	150391.41575753
-2037794.99028074	664278.73199413	1747816.68229323	-1336336.5290313
1561109.53174886	-862711.08642515	1563757.43251863	-306310.3809282
676664.42042966	594933.04993551	-2360141.22173418	674162.88727608
-635572.63228837	215685.24248983	-274335.74252976	-1065265.56705362
1814018.74309959	-356708.79153355	1185103.41844263	-968342.39444818
1933769.08957051	400164.09466467	-1928642.59136922	-31212.98105213
2807119.87750253	-1367325.1859044	-3601011.97825332	1489255.1720756
-101619.41390044	1086600.10941138	-2183234.24698047	1225600.20126824
-2689333.78491121	577063.52553009	1899610.74955134	-341663.12812332
-1725327.25208775	705683.71107839	5605247.10614642	205051.74343034
-2448044.64612613	-185164.23682514	-1899245.10023214	-396810.87310884
-1266908.12501579	830701.93565241	-1885030.46001325	160147.43306382
-1.54427591e+06	-9.07881070e+02	847783.93612577	133161.37274499
-2479549.92156772	-51539.74139854	-751140.70298458	-1055307.17289176
2255754.38551814	-1087412.02025608	971327.07735918	77088.55506394
4730336.64938746	-914344.99618323	-4504434.817732	1372672.03450013
-987655.02904293	1399876.44405401	1297651.55488009	-925999.14186283
1487229.37650129	421535.09011536	-3512598.26008598	-85650.16464307

Table B.4: Simple synthesizer parameters generated from random assignment of low level features.

Appendix C

Extra graphs

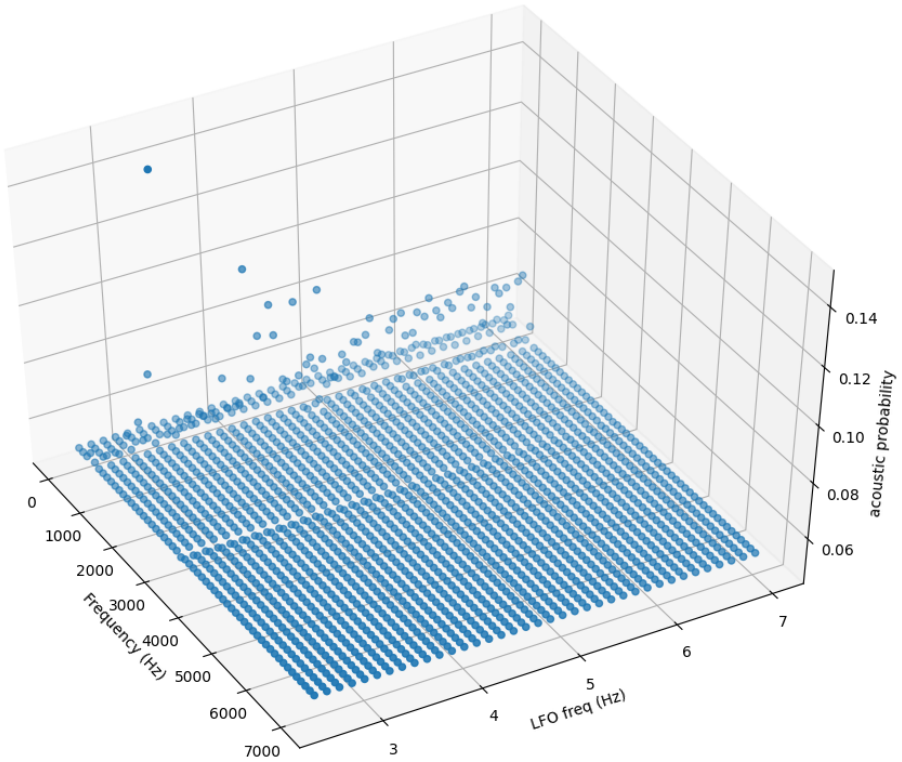

Figure C.1: A 3D plot of the two parameters in the simple synthesizer and the *acoustic* class. Note the highest value division for the probability is only 0.14.

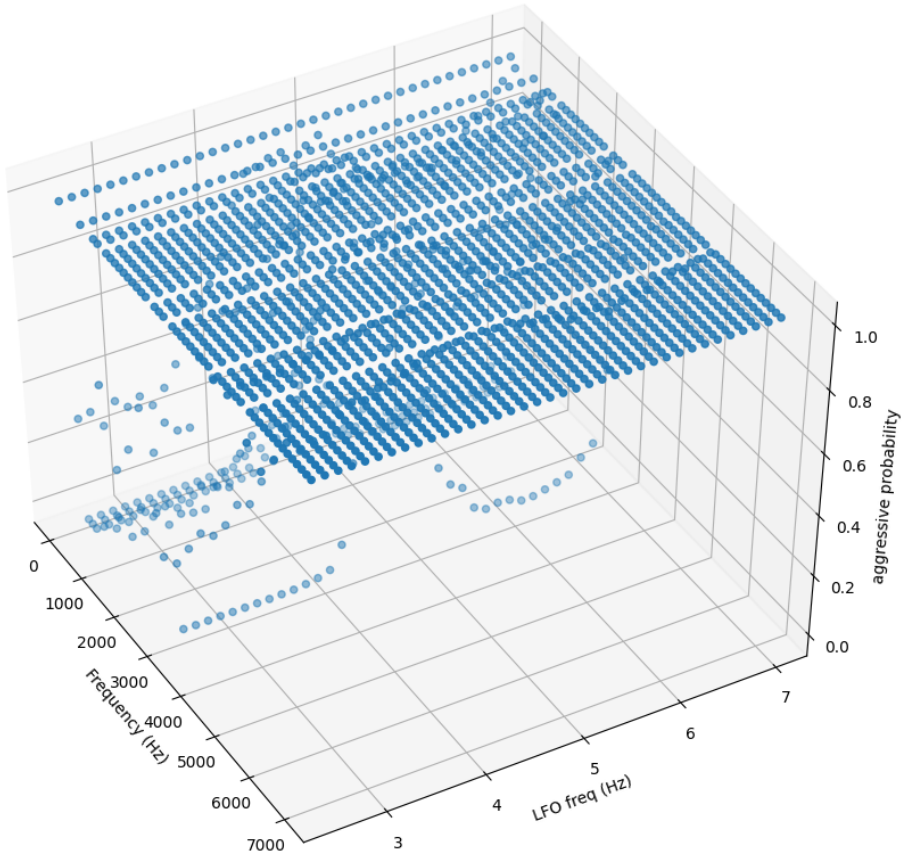

Figure C.2: A 3D plot of the two parameters in the simple synthesizer and the *aggressive* class.

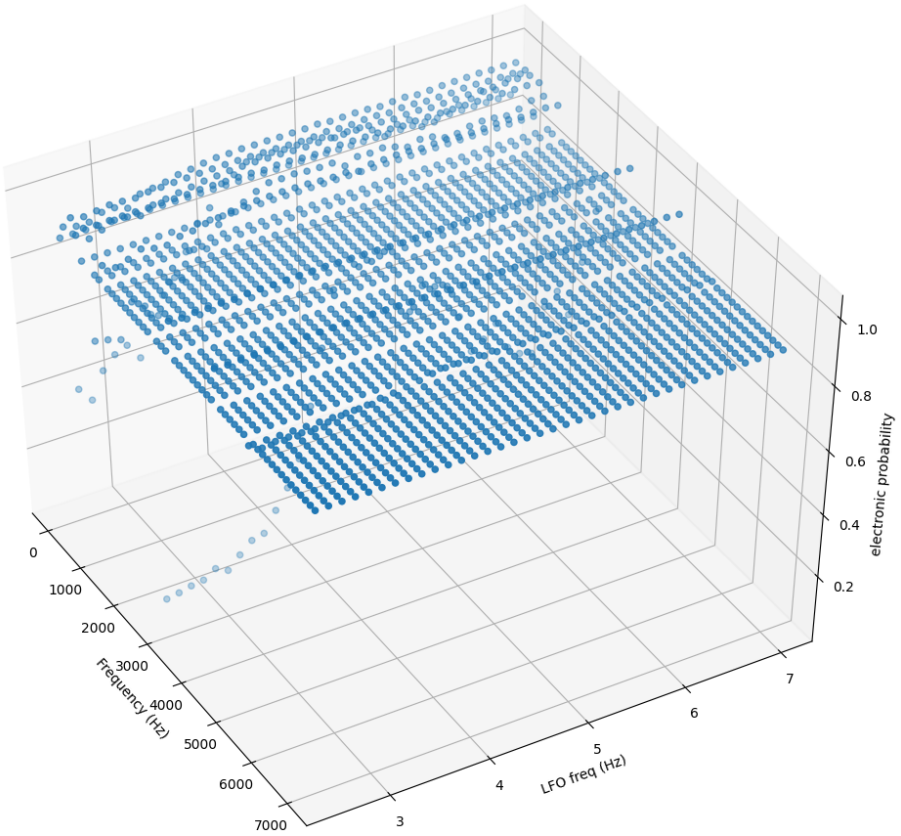

Figure C.3: A 3D plot of the two parameters in the simple synthesizer and the *electronic* class.

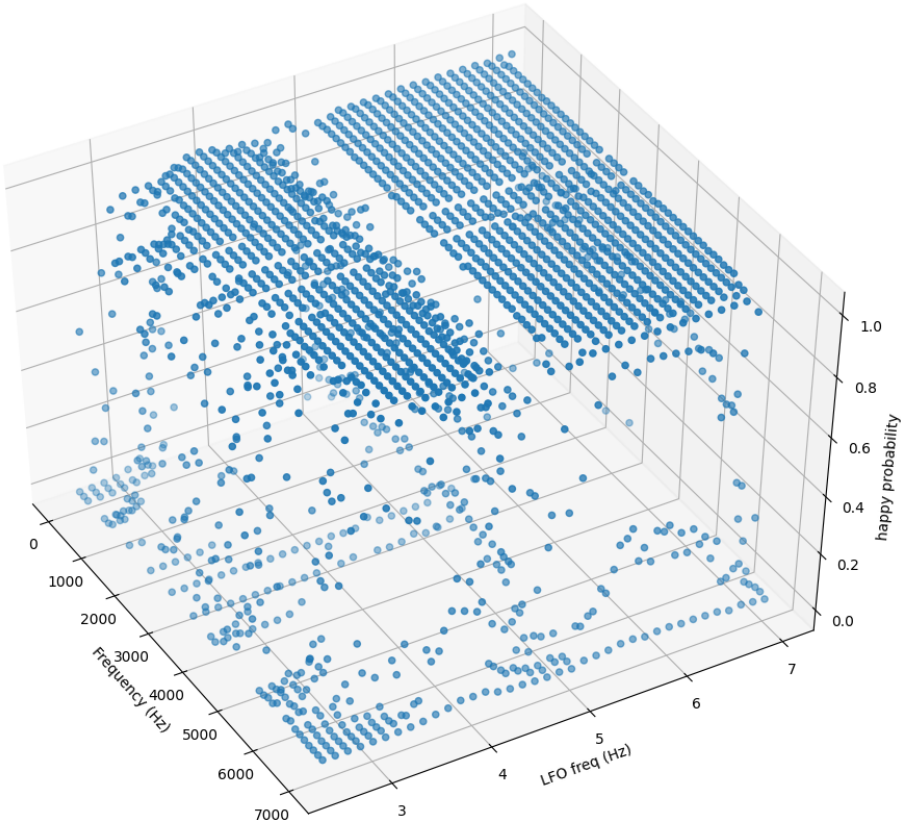

Figure C.4: A 3D plot of the two parameters in the simple synthesizer and the *happy* class.

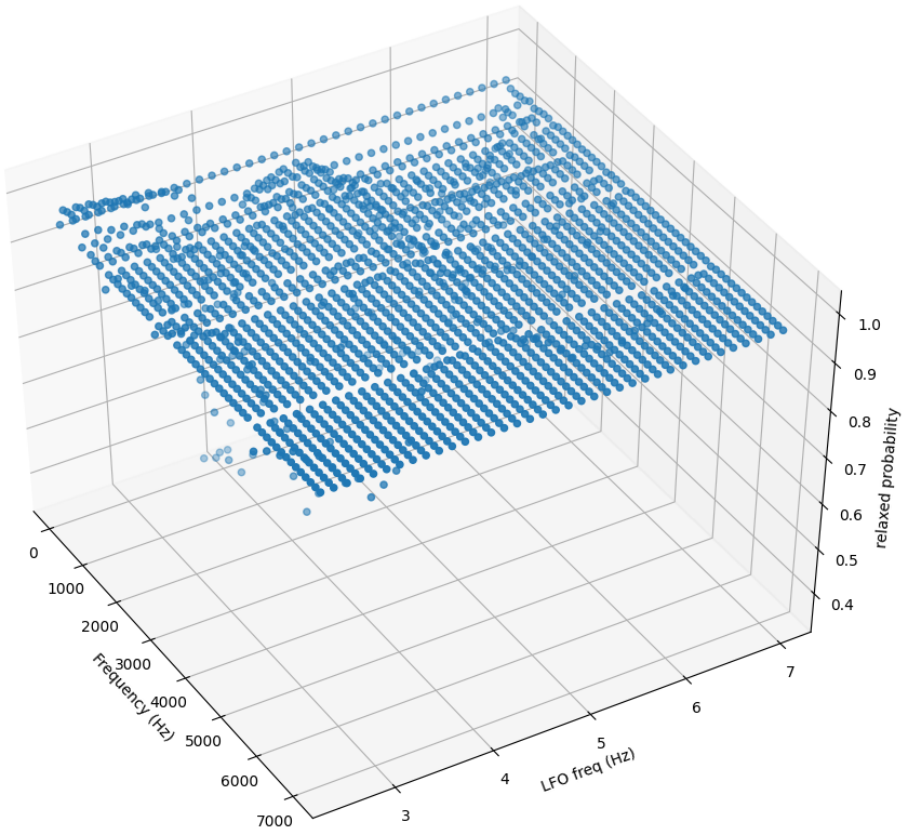

Figure C.5: A 3D plot of the two parameters in the simple synthesizer and the *relaxed* class.

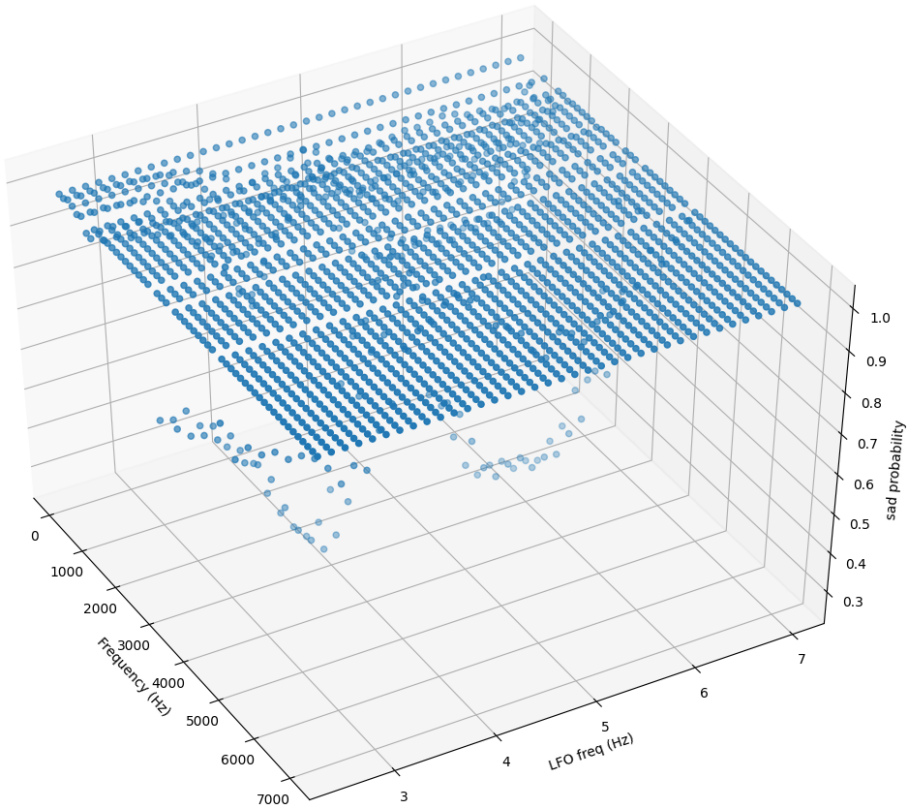

Figure C.6: A 3D plot of the two parameters in the simple synthesizer and the *sad* class.

Appendix D

Code

Listing D.1: The `SynthWrapper.py` file.

```python
from abc import ABC, abstractmethod

class SynthWrapper(ABC):
    """Interface for a synthesizer wrapper class"""

    # Example parameter list of dictionaries
    # Has to be implemented for all parameters that are required to be swept
    parameters = [
        {
            "name":         "parameter 1",
            "min":          0,
            "max":          10,
            "increment":    1
        },
        {
            "name":         "parameter 2",
            "min":          2.5,
            "max":          23.7,
            "increment":    2.5
        }
    ]

    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def initialize(self) -> bool:
        """Initializes the synthesizer, returns true if successful"""
        pass

    @abstractmethod
    def generate_sound(self, file_name: str, parameters: list) -> bool:
        """Generates the sound made by the synth given a list of parameters

        file_name: name of the output file
        parameters: list of parameters to apply to the synthesizer
        """
        pass
```

Listing D.2: The `Csound_SynthWrapper.py` file.

```python
from src import SynthWrapper as SW
from include import ctcsound as cs

class Csound_SynthWrapper(SW.SynthWrapper):

    def __init__(self):

        # Defines an orchestra of instruments that are run
        self.orc = """

            sr = 44100
```

```
            ksmps = 32
            nchnls = 1
            0dbfs  = 1

            instr 1
                out(oscili(p4/2 + oscili:k(p4/2, p6), p5))
            endin

        """

        # Setup dictionaries of parameters
        self.parameters = [
            {
                "name": "sound frequency",
                "min": 27.5,
                "max": 7040.0,
                "increment": 125.0
            },
            {
                "name": "LFO frequency",
                "min": 2.5,
                "max": 7.0,
                "increment": 0.125
            }
        ]

    def initialize(self) -> bool:
        pass

    def generate_sound(self, file_name:str, parameters:list) -> bool:
        self.synth = cs.Csound()

        sco_beg = "i 1 0 5 0.5 "
        sco_end = ""

        params = ""
        for p in parameters:
            params = params + str(p) + " "

        sco = sco_beg + params + sco_end

        self.synth.compileOrc(self.orc)
        self.synth.readScore(sco)
        self.synth.setOption('-o' + file_name + '.wav')
        self.synth.start()
        self.synth.perform()
        self.synth.stop()
        self.synth.reset()

        return True
```

Listing D.3: The `SubSynth_SynthWrapper.py` file.

```
from src import SynthWrapper as SW
from include import ctcsound as cs

class SubSynth_SynthWrapper(SW.SynthWrapper):
    # based on https://write.flossmanuals.net/csound/b-subtractive-synthesis/

    def __init__(self):

        self.orc = """

            sr = 44100
            ksmps = 4
            nchnls = 2
            0dbfs = 1

            initc7 1,1,0.8                ;set initial controller position

            prealloc 1, 10

            instr 1

                ; asign p-fields to variables
                iCPS    =           cpsmidinn(p4) ;convert from note number to cps
                kAmp1   =           p5
                iType1  =           p6
                kPW1    =           p7
                kOct1   =           octave(p8) ;convert from octave displacement to multiplier
                kTune1  =           cent(p9)   ;convert from cents displacement to multiplier
                kAmp2   =           p10
```

```
                iType2 =            p11
                kPW2   =            p12
                kOct2  =            octave(p13)
                kTune2 =            cent(p14)
                iCF    =            p15
                iFAtt  =            p16
                iFDec  =            p17
                iFSus  =            p18
                iFRel  =            p19
                kRes   =            p20
                iAAtt  =            p21
                iADec  =            p22
                iASus  =            p23
                iARel  =            p24

                ;oscillator 1
                ;if type is sawtooth or square...
                if iType1==1||iType1==2 then
                ;...derive vco2 'mode' from waveform type
                    iMode1 = (iType1=1?0:2)
                    aSig1  vco2   kAmp1,iCPS*kOct1*kTune1,iMode1,kPW1;VCO audio oscillator
                else                                        ;otherwise...
                    aSig1  noise  kAmp1, 0.5                ;...generate white noise
                endif

                ;oscillator 2 (identical in design to oscillator 1)
                if iType2==1||iType2==2 then
                    iMode2  =  (iType2=1?0:2)
                    aSig2  vco2   kAmp2,iCPS*kOct2*kTune2,iMode2,kPW2
                else
                    aSig2 noise  kAmp2,0.5
                endif

                ;mix oscillators
                aMix       sum          aSig1,aSig2
                ;lowpass filter
                kFiltEnv   expsegr      0.0001,iFAtt,iCPS*iCF,iFDec,iCPS*iCF*iFSus,iFRel,0.0001
                aOut       moogladder   aMix, kFiltEnv, kRes

                ;amplitude envelope
                aAmpEnv    expsegr      0.0001,iAAtt,1,iADec,iASus,iARel,0.0001
                aOut       =            aOut*aAmpEnv
                        outs         aOut,aOut
            endin

        """

        self.parameters = [
            {
                # Defined as a numbered note
                # that gets translated by the instrument definition into a Hz value
                "name": "input key",
                "min": 10,
                "max": 80,
                "increment": 1
            },
            # ============================================================
            # Oscillator 1
            {
                "name": "o1 amplitude",
                "min": 0.2,
                "max": 1,
                "increment": 0.2
            },
            {
                "name": "o1 type",
                "min": 2,
                "max": 2,
                "increment": 2
            },
            {
                # Values of 0 and 1 mean nothing as they create no sound
                "name": "o1 pulse width modulation",
                "min": 0.2,
                "max": 0.8,
                "increment": 0.2
            },
            # ============================================================
            # Oscillator 2
            {
                "name": "o2 amplitude",
                "min": 0,
                "max": 1,
```

```
                "increment": 0.2
            },
            {
                "name": "o2 type",
                "min": 2,
                "max": 2,
                "increment": 2
            },
            {
                # Values of 0 and 1 mean nothing as they create no sound
                "name": "o2 pulse width modulation",
                "min": 0.2,
                "max": 0.8,
                "increment": 0.2
            }
        ]


    def initialize(self) -> bool:
        pass

    def generate_sound(self, file_name:str, parameters:list) -> bool:
        self.synth = cs.Csound()

        sco_beg = "i 1 0 5 "

        sco_inter_1 = "0 0 "
        sco_inter_2 = " 0 0 "

        # defines global low pass filter envelope
        # set to be non intrusive, at 20kHz
        # also defines the global amplitude envelope,
        # set to be static for all sounds to make sure the sound is 5s long
        # attack decay sustain release
        sco_end = "20000 0.05 0.95 1 1 0 0.05 0.95 1 1"

        params = ""
        for p in parameters[:-3]:
            params = params + str(p) + " "

        params_2 = ""
        for p in parameters[-3:]:
            params_2 = params_2 + str(p) + " "

        sco = sco_beg + params + sco_inter_1 + params_2 + sco_inter_2 + sco_end

        self.synth.compileOrc(self.orc)
        self.synth.readScore(sco)
        self.synth.setOption('-o' + file_name + '.wav')
        self.synth.start()
        self.synth.perform()
        self.synth.stop()
        self.synth.reset()

        return True
```

Listing D.4: The `ExtractorWrapper.py` file.

```
from abc import ABC, abstractmethod, abstractproperty

class ExtractorWrapper(ABC):
    """Interface for an extractor wrapper class"""

    features = ["tempo"]

    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def extract_features(self, input_filename:str):
        """Extract features from sound given by the filename"""
        pass
```

Listing D.5: The `Essentia_HLD_ExtractorWrapper.py` file.

```
from src import ExtractorWrapper as EW
import subprocess as sp, json, os

class EssentiaWrapper(EW.ExtractorWrapper):
```

```
5
6       def __init__(self):
7
8           self.features = [
9               "acoustic",
10              "aggressive",
11              "electronic",
12              "happy",
13              "party",
14              "relaxed",
15              "sad"
16          ]
17
18          ## Two different executables as I work on two different machines
19          # self.executable_name = "./essentia_streaming_extractor_music (MAC)"
20          self.executable_name = "./essentia_streaming_extractor_music (Ubuntu)"
21          self.profile_name = "./profiles/config.yaml"
22
23      def extract_features(self, input_filename:str):
24
25          # call subprocess, create temp file
26          # parse temp file and extract necessary data
27
28          sp.call([self.executable_name, input_filename, "temp.txt", self.profile_name])
29
30          feature_list = {}
31
32          with open("temp.txt") as feature_file:
33              data = json.load(feature_file)
34              for mood in data['highlevel']:
35
36                  label = data['highlevel'][mood]['value']
37                  probability = data['highlevel'][mood]['probability']
38
39                  if "not_" in label:
40                      label = label[4:]
41                      probability = 1 - probability
42
43                  feature_list[label] = probability
44
45          os.remove("temp.txt")
46
47          return feature_list
```

Listing D.6: The `ParameterSweeper.py` file.

```
1   from src import SynthWrapper as SW
2   from src import ExtractorWrapper as EW
3   import math, os, json
4
5   class ParameterSweeper():
6
7       def __init__(self, synthIn:SW.SynthWrapper, extractorIn:EW.ExtractorWrapper):
8           self.synth = synthIn
9           self.extractor = extractorIn
10
11          self.filename_no = 0
12
13          self.export_data = {}
14          self.export_data['data'] = []
15
16
17      def sweep_parameters(self, parameter_num = 0, parameter_list = []):
18          # set current parameters
19          # if next one exists, pass list onto that one
20          # otherwise generate sound
21
22          curr_param = self.synth.parameters[parameter_num]
23
24          # Find number of steps based on range and increment
25          steps = math.floor((curr_param["max"] - curr_param["min"]) / curr_param["increment"])
26
27          for p in range(steps + 1):
28
29              param = curr_param["min"] + p * curr_param["increment"]
30              if len(parameter_list) < len(self.synth.parameters):
31                  parameter_list.insert(parameter_num, param)
32              else:
33                  parameter_list[parameter_num] = param
34
35              if parameter_num < len(self.synth.parameters) - 1:
36                  self.sweep_parameters(parameter_num + 1, parameter_list)
```

```
37              else:
38                  self.filename_no += 1
39                  filename = 'sound' + str(self.filename_no)
40                  if self.synth.generate_sound(filename, parameter_list):
41
42                      data = {}
43                      data['Parameters'] = {}
44
45                      for param_no in range(len(parameter_list)):
46                          data['Parameters'][self.synth.parameters[param_no]['name']] = parameter_list[param_no]
47
48                      features = self.extractor.extract_features(filename + '.wav')
49                      os.remove(filename + '.wav')
50                      data['Features'] = features
51
52                      self.export_data['data'].append(data)
53
54          # with open('data.json', 'w') as outfile:
55          #     json.dump(self.export_data, outfile, indent=2)
56
57      def dump_data(self):
58          with open('data.json', 'w') as outfile:
59              json.dump(self.export_data, outfile, indent=2)
```