Parameter automation for granular synthesis

author: Karol Bakunowski

supervisor: Michael Zbyszyński

BSc (Hons) Music Computing
Goldsmiths, University of London

2019

Abstract

My research is about stuff.

It begins with a study of some stuff, and then some other stuff and things.

Acknowledgements

Acknowledge all the things!

Contents

1 Introductiones **5**

1.1 Aims and Objectives 5

1.1.1 Deliverables 6

2 Literature Review **7**

2.1 Problem background 7

2.1.1 Synthesizer programming 7

2.1.2 Sound matching 8

2.2 Audio descriptors 9

2.2.1 Mel-frequency Cepstrum Coefficients 9

2.2.2 Spectral Flux 10

2.3 Synthesis 10

2.4 Predictions 11

3 Methods **13**

3.1 Audio descriptors 14

3.1.1 Definitions and Formulation 14

3.1.2 Implementation 15

3.2 Predictions 17
3.2.1 Dataset 17
3.2.2 Neural networks 18
3.3 Synthesis 20
3.3.1 Granular Synthesis (basic explenation of the granular synth concept) 20
3.3.2 The JUCE implementation (author's implementation) 20

4 Results and Analysis **26**
4.1 Methodologies 26
4.2 Sound similiarity 27
4.2.1 Quantative evaluation 27
4.2.2 Qualitative evaluation 27
4.3 User experience 27

5 Discussion **28**
5.1 Summary of Findings 28
5.2 Evaluation 28
5.3 Future Work 28

6 Conclusion **29**
.1 Title of Appendix A 30

List of Figures

3.1 Basic flowchart 13

3.2 logic of the MFCC computation 16

3.3 filling the buffer 16

3.4 computing Essentia algorithms 16

3.5 Example of one instance of the dataset 17

List of Tables

Chapter 1

Introductiones

1.1 Aims and Objectives

The initial aim of the project was to implement a machine learning solution to the task of granular synthesizer programming based on sound matching. Consequently building a tool that would assist musicians in creating interesting sounds, provided an audio input to the system.

Throughout the entire development process, the focus remained mainly on creating a well functioning, usable program, despite some shortcomings in the areas of synthesis, audio analysis and machine learning respectively. Integrating these three modules well took priority over designing a more sophisticated solutions to each individual problem. Consequently allowing for creation of a tool functioning in near real-time, with the possibility of extending the modules that contribute to the whole.

The methods here differ from other approaches in literature (cite some papers that have done this, eg. Matthew, Leon) primarily in that here, one standalone tool has been created, that is usable on it's own. Users have direct ability to program the synthesizer using input from the microphone, and the entire process is handled by two buttons. No prior experience with programming is required, and the only assumption made about the user is knowledge of basic synthesizer programming.

The main objectives therefore are:

1. To build a tool that is helpful to artists in the process of creating sounds

2. To challenge the interaction between an artist and a preset as a starting point to synthesis

3. To achieve a response that is not only stimulating to the user but also differs from simply

randomizing the parameter values

In measuring the project's success, the subjective sonic coherence and similarity of predicted outputs may be considered the best indicator (the human discriminator?). Undoubtedly, if the predictions achieved are satisfying to the user, the main goal of this project was achieved. Additionally, users' opinions about whether the tool would be usefull in their work process will be a good indicator of the project's success.

Along with that, a quantitative analysis, such as comparing the audio descriptors on target and predicted sounds should prove itself useful. It will allow to do a quite generalisable, and objective comparison of target vs. predicted sounds. This process will enable the evaluation of neural networks used for predicting the parameter values, and help in assuming whether users will find the program useful.

EDIT!!!!!!: In (future chapters) I provide some critique on established techniques of assessment of these types of problems and ultimately conclude that some stuff in the best way of doing it.

1.1.1 Deliverables

In the inerest of achieving the above decalred goals, I propose a standalone program, that allows users to set parameters of a granular synthesizer based on a one second buffer of audio, generated form the microphone input.

The program could be arbitrarily split into three different modules, in order to clarify the it's basic structure. Namely, synthesis, audio analysis and machine learning.

The synthesis is implemented in the "Juce" framework. Audio analysis tools from the "Essentia" library are used to help create training datasets, as well as help make predictions. And lastly, an implementation of a Multilayered Perceptron feedforward neural network with the "Keras" API is presented. The "Frugally Deep" library is responsible for deploying the "Keras" model into C++ code, allowing for near-real time performance, and removing the need for communication between C++ and Python.

Resulting program is a standalone "JUCE" application, capable of performing granluar synthesis, extracting audio features, and predicting parameter values of the synthesiser based on microphone input.

Chapter 2

Literature Review

2.1 Problem background

2.1.1 Synthesizer programming

Programming synthesizers can be enjoyable, fun and gratifying, yet very often presents a big challenge. The amount of adjustable parameters on such a device can be overwhelming, and consequently the range of sounds that can be achieved is usually very extensive.

Granulation, or any other corpus based synthesis technique for that matter, presents what could be called a special case in this domain. Not only do the parameters determine the final output, but also the input sample has a tremendous influence over the sound.

In a granular synthesizer, users are presented with choices ranging from determining the audio to be sampled, to the amliptude envelope for each grain. A huge array of possibilities.

Therefore the ability to make decisions about programming a granulator to achieve desired results has to come from either a place of certainty about what each parameter is responsible for combined with intuition about the instrument, or a place of experimental though, and somewhat random parameter value assignments.

Users fairly new in the realm of synthesizer programming may encounter issues creating sounds they desire. This is to be expected, as like in any other area, novices have to get through a rather steep learning curve, to achieve certain intuition, and skill. This, however, often prevents people from creating what they want, or makes them settle for less, consequently limiting their creative output.

On the other hand, experienced musicians are constantly looking for inspiration and new ways of interaction with the process of creating exciting sounds[4]. Furthermore, programming synthesizers by following a not clearly defined intuition, combined with the use of presets and a mixture of experimantal methods is a popular approach[9], especially considering the amount of tools available today, and the ease of acquiring them. The fact of having too many VST plugins in a DAW, is a common problem today.

There are of course all the people in between the complete novice, and succesfull artist. They may have some experience, and can follow their own intuition, yet are trying to find their own sound, and define their style of production. Making that process easier, consequently making the music making process more accessible would be a great contribution to the community (EDIT, BUT IDEA IS GOOD).

2.1.2 Sound matching

The problem of sound matching seems to be almost unexistent for humans. We are able to hum along a song we have heard previously, or replicate sounds of different objects, like cars, or animals, like dogs. This activity does not present any real challenge for us. It could even be said that such a behaviour is taken for granted.

It has been around since forever, and it is mostly how children learn language. The word "onomatopoeia" desribes a process of creating a word that phonetically resembles, or suggests the sound it describes (SOURCE) and has been around even before language came aobut (CHANGE) (ADD SOURCE)!

Conversly, creating a specific desired sound on an instrument of any kind takes serious practice, and sacrifices. Musicians who are able to translate what they hear in their head, directly to an instrument, be it a physical one like piano or guitar, or a software based instrument, are considered viruosic. Therefore, the ability to replicate any sound in a synthesizer, based solely on an audio input seems like a very intuitive way of interaction, and possibly an inspiring way of working for musicians.Additionally, it would blur the line between a begginer, and a virtuoso in that specific area, allowing people with less technical expertise to create music they want to hear.Primarily, however, such interaction would be an immensly helpful, and time-saving scenario for musicians, allowing them to spend more time on creative work, taking over, at least partially the task of programming a synthesiser.

In the next three sections I go over each area of the project, and talk about previous work that has been done on each aspect.

2.2 Audio descriptors

Because of the way audio is represented digitally, mimicking sounds is not as intuitive for computers as it is for humans. Algorithms that are capable of generating meaningful data about given audio signal are needed in order to describe sounds in different ways, and based on different assumptions. (add more about why these are needed, and pure audio singal can't be used?)

A great amount of these analysis tools exist, and a lot of them are easily accessible either through libraries in different programming languages, such as 'libriosa'[7] for Python, or 'Essentia'[5] for C++, or directly through different audio software such as 'Audacity'[1] or most Digital Audio Workstations today.

The decision about which were suitable for the task of automatic synthesizer programming was mainly based on previous research done in this domain. However, some amount of experimentation with different audio extractors was done, to see if certain assumptions made during early stages of the project were correct.

EXPAND: The early work done with focus on sound matching was using mainly spectral features, such as the Fast Fourier Transform (source). More recently, however, the Mel-frequency Cepstrum Coefficients seem to be the prefferable descriptor (source) [22][23]

2.2.1 Mel-frequency Cepstrum Coefficients

In order to get information about the frequencies present in an audio signal, a conversion to the frequency domain has to be done. Extracting frequency information form the time-domain can be done (reference), however a well established, and more prefferable method today exists, which is the Fourier Transform (source). The algorithm can dissect a signal to it's most basic sinusoidal components, therefore determining how much of which frequencies are present in the signal allowing for creation of a spectrogram.

The Fourier Transform, or more precisely it's less computationally expensive version the fast Fourier Transform (FFT) (source) serves as the basis to compute an algorithm which is possibly the most powerful algorithm for determining the timbre of an audio signal today - the Mel-frequency cepstrum coefficients (MFCC).

Details of this algorithm are well beyond the scope of this paper, however many studies have been done to prove the usefullness of MFCCs in sound matching tasks[14][3], as well as in monophonic instrument recognition tasks[2].

(not sure if this paragraph is needed?) Today they are mainly used in speech symthesis and speech

recognition, however can be apply to any signal, and are a very powerful descriptor. More recently MFCCs have been used in the field of music information retrieval applications (sources)

Blurring the line between timbre and rhythm detection, a sequence of MFCCs can tell a computer quite a substantial amount of information about rhythmical qualities of sound, as well as any temporal changes in both time, and frequency domains. Listener studies have shown that movement in MFCC space is associated with a similar 'sized' movement in human perceived timbre space[11] (change as this sentence is a total ripoff)

2.2.2 Spectral Flux

MORE CONTENT, CITE PAPERS (not specifically used for sound matching tasks i think)

Onset detection, which tries to estimate how many 'peaks' there are in a signal is a very useful algorithm to estimate how much rhythmical content there is present.

It can be achieved with an algorithm called "Spectral Flux". It compares consequtive spectra, determining how much change has happend, producing a float value corresponding to it. By thresholding this change, based on the mean value of the flux, floats corresponding to the onsets in a signal can be derived (source, source).

2.3 Synthesis

Research has been done previously as an investigation into automation of parameters in synthesizers based on sound matching. Taking a snippet of sound, the algorithm would try to find parameter settings to match a produced sound as closely as possible to the source[12].

This research mostly focuses on FM synthesis[6], although experiments on different synthesis techniques have been done[**dahlstedt˙creating˙nodate**], including VST plug-ins[14].

However, using this approach on corpus-based synthesis remains an untapped area of research, worth investigating[8].

Many software based granular synthesizers exist, both in standalone (source, source) and VST form (sources). The implementations, and certain parameters etc. differ in each example. Nonetheless, this type of synthesis introduces an interesting problem in the contex of parameter prediction, which applies to any corpus based synthesis engine. The sounds created are heavily based on the sample fed into the synthesizer. Each parameter changes meaning significantly, once the input sample is changed.

Two possible solutions come to mind, when trying to overcome this problem. Using one input sample for synthesis, and trying to predict samples for one perticular instance of the granulator is one. Another, would be trying to fit some universal analysis algorithms, that could decribe rhythym, pitch, density of sound etc. independently of the original sample. This way any sample could be 'molded' into what would resemble the original prediction target.

Concatenative synthesis is an alternative approach to this problem. It could be beneficial, as it would try to find 'grains' as closely resembling parts of the target as possible, and recreate it using little parts, almost like puzzles, that the algorithm thinks fit. However, that approach also has limitations, as it could only recreate the target sound out of the samples stored in it's database, therefore making the output biased. (cataRT source)

2.4 Predictions

A impressively sized bulk of work has been done around the automatic programming of synthesizers. Possibly the most complete body of work on the topic is Yee-King's thesis [13]. One of the first approaches described by Horner et al. was a Genetic Algorithms [6] aimed at predicting settings for an FM synthesizer. Another possibility would be to use the simplest algorithms available and suitable for this task, the Hill Climber algorithm (source-matthew's paper).

These approaches are all quite valid, however the main problem occuring with them is the computational time required to come up with a result. None of these approaches really offers the ability of prediction in real, or near real time, making them an unappealing choice for user facing systems.

An approach in line with the most recent reaserach would seem to be an adaptation of neural networks for the task of automatic synthesizer programming. Fedden et. al (source) has made a vst host for extraction of parameters, and creation of a dataset, as well as a framework to train neural networks in. (more?) A more in depth exploartion of possibilities in this domain was done by Yee-King, Fedden, and D'Inverno et al (source), which compared and considered GA, HC, MLP, LSTM, and LSTM++ algorithms.

Another possibility is the use of convolutional networks, however this approach would be limited to classification.

Alternatively, in order to try and generalise the predictions for different input samples for the synthesizer. The predictions could be devided between different parameters. Then linking different prediction algorithms with different parameters could possibly allow for a creation of a more direct relationship between audio descriptors and the synthesis results. No widely available research

seems to approach this task in this modular way, yet it seems like discovering certain linear relationships between parameter values and audio descriptors is certainly possible.

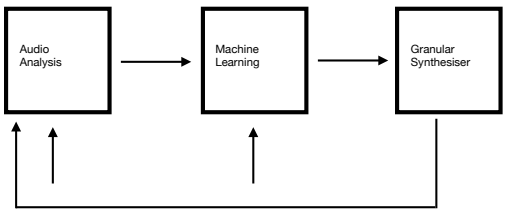
It seems that most of this research focuses on reproducing the original input[10]. Perhaps more interesting and novel sounds could arise as an effect of bad performance, but is does not seem to be the desired outcome in most cases.

Chapter 3

Methods

The method of tackling this problem will be based on previous research done in this domain. There are three essential components for this project:

Figure 3.1: Basic flowchart



The box "Granular Synthesiser" in figure 1. represents a granular synthesizer. It was implemented in C++, using the 'JUCE' framework (SOURCE). (Add about it being quasi-synchronous, and all other possible information.)

'Machine Learning' stands for the predictions module, or more precisely the Multilayer Perceptron model built in 'Keras', and later ported to C++ with the help of the 'Frugally Deep' library.

Lastly, the 'Audio Analysis' square represents the aspect of the project responsible for extracting audio features from input sounds. This was done with the 'Essentia' library, that allowed me to perform frame cutting, windowing, extracting the spectrum and finally MFCCs on any specified buffer.

3.1 Audio descriptors

3.1.1 Definitions and Formulation

As mentioned in chapter 2 the MFCC algorithm is proven to be one of the most reliable desriptors available, and can not only describe the temporal qualities, but also be indicative of the changes that happen in the input signal. (source again?) Therefore, it is the only algorithm used, with further justifications for this decision provided later in this chapter.

The Mel-frequency cepstrum is a representaion of the short-term power spectrum of sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency (WTF, CHANGE AND ADD SOME REFERENCE TO THE DESCRIPTION). In turn, the MFCCs, are coefficients that collectively make up the Mel Cepstrum. The biggest advantage of this algorithm is that in the MFC the frequencies are spaced in a way which approximates the human auditory system more closely than the cepstrum used in a Fourier Transform.

The MFCCs are commonly derived as follows:

1. Take the Fourier Transform of a signal
2. Map the powers of the spectrum obtained into the mel scale, using triangular overlapping windows
3. Take the logs of the powers at each of the mel frequencies
4. Take the discrete cosine transform of the list of mel log powers, as if it were a signal
5. The MFCCs are the amplitudes of the resulting spectrum

There can be variations on this process, such as differences in the shape or spacing of the windows used to map the scale, or addition of dynamics features such as delta and deltadelta coefficients.(ADD SOURCE AND CHANGE - total ripoff)

In order to achieve usable (source, what if we took mfccs on the entire song lol?) results from the algorithm, often the first element of the list above is done on a windowed frame of an input signal.

Let's assume, that we are trying to analyse an audio input which is 10 seconds long. In order to obtain usable results, we have to separate the input into multiple excerpts, and compute the algorithm on each of these frames. In order to 'smooth' out the values and achieve better results (more specific, maths, studies) the frames are often overlapped, so that certain 'history' of is taken into account. And information in one frame is not treated as independent of the rest of the signal. This is often reffered to as 'hopsize'. The effect of this behaviour is that consequtive

frames are overlapped. Lengths of both the frames, and the size of the hopsize vary, adn depend on application. However the standard values are assumed to be 2048 samples and 1024 samples consequtively. With these values our input would be split into frames of size 2048 (approximately x ms.), and would take one frame, analyse it, and move 1024 samples ahead, to start the analysis of the next frame.

For each frame, MFCC are calculated based on the extracted FFTs. The result for one frame is a vector of 13 floating point values, each corresponding to a different range of frequencies.

At a sampling rate of 44.1kHz, and a buffer size of 512 samples, this corresponds to exactly 45 vectors of 13 float values for one second of sound. Meaning that our hypothetical sound of length of 10 second would comporomise of 450 MFCCs.

Several different audio features have been experimented with, especially in the quest of finding universal ones, that could be applied to any source file in the granular synthesizer. One of such features is onset detection. It's implementation consists of the spectral flux algorithm, thresholding and peak finding, in order to estimate how many peaks, or onsets are in a specified buffer. (more description, or is this useless to talk about?)

However, during testing, which is described in chapter 4, such a connection was not found, therefore this project solely depends on MFCCs.

3.1.2 Implementation

Essentia

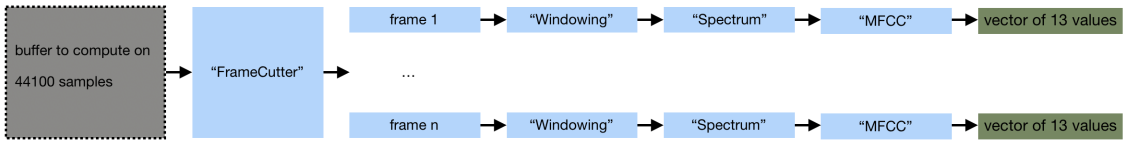
The process decribed above was achieved with algorithms provided by the 'Essentia' library[5]. It works in a modular way, that can roughly be described as spiltting the algorithms into lower, and higher level algorithms. They can be then 'stacked' to create more complex chains of computation, capable of extracting, as in this case, the MFCCs.

The process is as follows:

- Firstly, an audio buffer on which the computation will be performed has to be specified. Here, one second of sound is written into a buffer (C++ vector), that then gets passed to the first algorithm
- 'FrameCutter' is where the buffer first arrives. This function splits the buffer into multiple frames, as dictated by the parameters. Then, each frame, separately is sent further down the chain, with the help of a while loop

- Each consequtive frame of the buffer arrives at the 'Windowing' algorithm, which essentialy smoothes out the edges of each frame.
- Once the windowing is applied to the frame, the spectrum of the input signal is calculated.
- Finally, the Mel-Frequency Cepstrum Coefficients are calculated for the frame

Figure 3.2: logic of the MFCC computation



Code responsible for creation of the buffer:

Figure 3.3: filling the buffer

```
// computing the essentia algorithms
if (audioFeatureExtraction.playbackBuffer.index < audioFeatureExtraction.getLengthOfBuffer())
{
    const auto* channelData = bufferToFill.buffer->getWritePointer(0, bufferToFill.startSample);
    for (auto i = 0; i < bufferToFill.numSamples; ++i)
    {
        audioFeatureExtraction.pushNextSampleIntoEssentiaArray(channelData[i]);
    }
}
```

This process is repeated for each frame, which at the sampling rate of 44.1kHz, and a buffer size of 512 samples, with frame size of 1024, and hopsize of 512 samples equals to exactly 45 frames per second. Which in C++ code looks as follows:

Figure 3.4: computing Essentia algorithms

```
void AudioFeatureExtraction::computeEssentia()
{
    while (true)
    {
      frameCutter->compute();

      if (!frame.size())
      {
          break;
      }
      if (isSilent(frame)) continue;

      windowing->compute();
      spectrum->compute();
      mfcc->compute();

      pool.add("lowlevel.mfcc", mfccCoeffs);
    }
```

3.2 Predictions

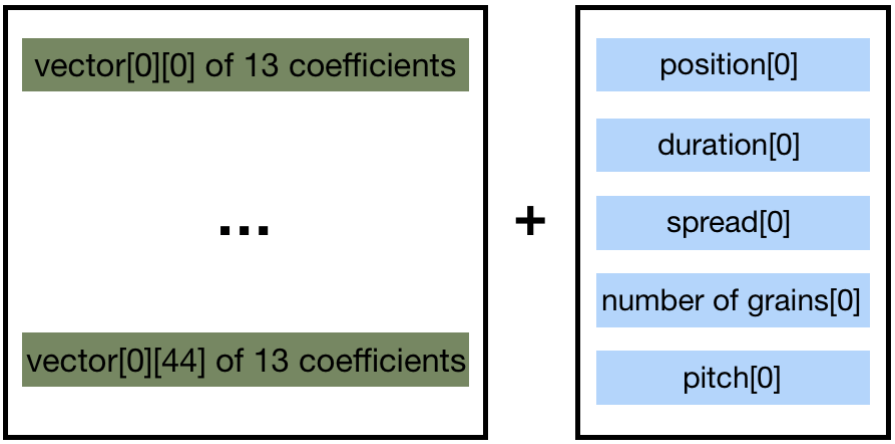
3.2.1 Dataset

Definitions and Formulations

A dataset, linking the MFCC values with the synthesis parameters had to be created, in order to create a way to use the information, and make a coorelation between parameters, and audio descriptors. Initially, I have set out to create a data set of every possible combination of parameter values, and their corresponding MFCCs. However, even though there are only 5 adjustable parameters, with a static hop size of a tenth of each parameter, it would take more than 1000 hours to complete this task, given that each parameter combination would be sampled for exactly one second.

In order to avoid this limitation, a different approach was taken. Every second, each parameter was set to a random value between their unique minimum and maximum values. Consequently, the audio features were extracted on each of these iterations, for exactly 10000 instances, which allowed for a creation of a dataset, of 10000 different parameter values, each corresponding to a 2D vector, of size 45 by 13 containing the computed MFCCs. (source here, where did i get the idea?)

Figure 3.5: Example of one instance of the dataset



Implementation

Creation of the dataset was done inside of the C++ JUCE synthesis code. A function responsible for setting a random value for the parameters was implemented using the JUCE::Random class.

After each, 1 second long iteration, the class instance is giving a random value between specified ranges, different for each parameter, which results in independent changes. This function is a pseudorandom number generator, for which seed was not specified, in order to get different values, if multiple datasets were desired.

After the parameter values are overriden by this function, the MFCC computation is initialised for a buffer, with the sound resulting from previously set parameters.

Then, two arrays are created, which contain the audio analysis results, and parameter values. In order to conveniently export these results as .JSON files, the arrays are appended to a 'Pool' data structure from the 'Essentia' library. 'Pool' includes an elegant solution for saving it's contents into a .JSON file with the help of 'YamlOutput' function, like so:

```
mergedMFCCs = factory.create("YamlOutput",
                             "filename", "MFCC.json",
                             "format", "json",
                             "writeVersion", false);
mergedMFCCs->input("pool").set(mergePool);

mergePool.merge(pool, "append");
```

The resulting dataset is imported into a Python script, in order to perform the training of different predictive models.

3.2.2 Neural networks

Multilayered Perceptron

Many different algorithms were considered for the task of predicting parameters, given the MFCC values (examples). Due to the success of neural networks for this task (source), other options were disregarded. There are many different possibilities when it comes to the architectures and types of networks, and it was shown that a Multilayered Perceptron, as well as Long-Short Term Memory network are both viable choices, with Bidirectional LSTMs having the best performance (check if true, and quote the infamous paper).

Due to time constraints, as well as the lack of computational power on machines I had access to, only a feedforward, 3 layer Multilayered Perceptron network was implemented. The popular 'Keras' (source?) library was used to create the architecture in Python.

The only activation function used was 'ReLu'. The 'Adam' optimizer, with mean absolute error served as the optimization function.

Before the data was fed to the network, it was scaled between 0 and 1, using the MaxMinScalar form the sklearn library.

The 45 by 13 MFCC vectors served as the features, and a vector of 5 different parameter values as the output of the network.

The 2D MFCC vectors, were flattened, consequently making the first layer of the MLP having 585 perceptrons. The second layer has 20 inputs, and the third 15.

The last layer has 5 ouputs - one for each parameter to be predicted, and as this is a regression problem - no activation function.

The network was trained for X epochs, making use of an early stopping function, to prevent overfitting. The loss function values for training set is about 0.9 and test set 0.12.

Different approaches were considered for handling the communication between the C++ synthesis code, and Python machine learning code. During the first few iterations of the project, starting with the first prototype, OSC messages were used to send audio analysis values to Python, predict parameters there, and send the predictions back to C++. This process however, was not optimal. It was relatively slow, not efficient, and required two separate programs to be running in order to give basic functionality.

Therefore, later in the production a search for a better solution was done. I have found the 'Frugally Deep' library that seemed to resolve all the issues present at the time. It required a Keras model to be saved into a .JSON file, and then it would convert it into XXX and allow for usage of the model directly in the C++ code, with a fairly straightforward way.

Consequently, the model in Keras was saved after training with like so:

With the toolset supplied by the 'Frugally Deep' library, it was then converted into XXX like so:

As seen above, with the file converted, only a few lines of code were needed to make use of the model directly in the C++ synthesis program, creating a standalone application, silmultaniously allowing for much quicker and simpler performance, without the need of any additional software running.

3.3 Synthesis

3.3.1 Granular Synthesis (basic explenation of the granular synth concept)

Describing granular synthesis in one paragraph is impossible without cutting some edges, and leaving out many intricacies of the technique. However, a summary of sorts, describing the main concepts behind the algorithm will be attempted here.

Granular synthesis was first imagined by Xenakis and Gabor (cite microsound and figure out more about it from there) in the late 1960s. It was a breakthrough technique, that was as controversial as it was inspiring back then. It was mostly achieved manually, cutting tape and sticking it back together. A very slow, and labour intesive process.

The most basic granular synthesiser requires some sort of an envelope over the grain, grain waveform, whether generated by an oscillator or taken from a sampled audio, and some grain spatialization. (get the graph from microsound and describe it basically.)

It is an extremely versatile method of synthesis, capable of generating anything between pitched sounds, similar to an additive, fm, or subtractive synths, to segmentation techniques, all dependent on 3 main parameters. Grain envelope, grain waveform, and grain length.

3.3.2 The JUCE implementation (author's implementation)

An original implementation of a granular synthesiser was done in order to gain the ability of creating a standalone program, with all the functionality mentioned in CHAPTER 1, and achieve all of the project's objectives. A possibility of creating only the audio analysis and machine learning modules existed, with the use of an external synthesiser, however that approach would obviously not allow for a creation of 'all in one' application, as stated in CHAPTER 1.

Separate, original implementation of the sythesizer allowed for the other modules to be included inside of it, therefore making the entire program independent.

The 'JUCE' (source) library was used as the fundament of the synthesizer. It is one of the most established libraries in the professional audio community, being used by companies such as 'Cycling 74', and 'Korg' amog others (cite website?). On top of all things audio, it provides a convenient way of creating a graphical user interface, which was an important part of this project. Thanks to 'JUCE', the implementation was kept minimal, and elegant.

The core of the implementation is contained in one class 'GrainStream'. All the functionality for

one grain, as well as the stream, otherwise vector of grains is contained in it.

Even though the concept of granular sythesis is very difficult, the implementation here is kept fairly minimal, to restiric the amount of dimensions, reducing learning time for the neural networks, and consequently minimizing the time needed to create an appropriately sized dataset.

In reponse to these requirements only 6 adjustable parameters were implemented.

Loading of samples

Loading of samples is implemented with the function setAudioSource like so:

```
void GrainStream::setAudioSource(AudioFormatReader& newAudioFile)
{
  int length = static_cast<int>(newAudioFile.lengthInSamples);

  //update grain parameters
  this->fileSize = length;
  this->samplingRate = static_cast<int>(newAudioFile.sampleRate);

  //clear the audio source and read new WAV
  this->AudioSourceBuffer.reset(new
                                AudioSampleBuffer(newAudioFile.numChannels, length));
  newAudioFile.read(this->AudioSourceBuffer.get(), 0, length, 0,
                    true, true);
}
```

It is using the JUCE::AudioFormatReader and JUCE::AudioSourceBuffer classes to handle this behaviour. The length of the sample, as well as it's sampling rate are also read here, and used to update global values.

Grain creation

Grains are created by feeding the buffer values from the input sample into the a vector of floats that makes up one grain.

Position

The position of each grain is determined by the 'FilePos' dial. It's value serves as a parameter to the 'setFilePosition' function, which in turn 'communicates' the starting position for each grain with the use of 'filePosition' integer. It was required to subtract 1 from the dial's value, because...XXX

The following code is responsible for this task:

```
void GrainStream::setFilePosition(int startingSample)
{
  this->filePosition = startingSample - 1;
}
```

Grain Size

The size of each grain varies between the minimum of 10ms and maximum of 1 second. It can be easily set using the Grain Size GUI dial. Value of the dial serves as a parameter to the 'setDuration' function:

```
void GrainStream::setDuration(int duration)
{
  //update duration and compute sampleDelta
  this->durationOfStream = duration;
  this->sampleDelta = static_cast<int>(this->samplingDelta *
  (duration/1000.0f));

  for (oneGrain grain : grains)
  {
    grain.grainDataEndPosition = grain.grainDataEndPosition +
    this->sampleDelta;

    if (grain.grainDataEndPosition >= this->fileSize)
        grain.grainDataEndPosition = (this->fileSize - 1);
  }
}
```

It is responsible for computing the end position of each grain (grainDataEndPosition), with the

use of a ‘sampleDelta’ variable. Which means the difference between the starting and ending position of a grain. The end position is then calculated by simply adding the delta to the start position.

Simple but effective error handling is then implemented, by ensuring that the end position of a grain will never be bigger than the length of the sample.

Spread

The so called spread functionality could be summarized as randomizing the starting position of one grain, independently of the others, in a certain range.

Let’s assume that the FilePos dial is set to 10000 samples, which would correspond to approximately 4 seconds into the file, assuming a samapling rate of 44,1kH (maybe note somewhere, that whenever I talk about sampling rate this assumption is made, sounds proffessional). With the ‘spread’ dial turned down to 0, each consequtive grain would start at exactly 10000 samples, play for it’s duration, and die, creating space for another grain to be spawned, again at the starting position of 1000 samples.

Once the dial has a value bigger than 0, a random number generator is used, to create a random integer in the range of original file position - value of the dial, and original file position + value of the dial. The random number is created for each new grain, consequently creating a new position for each new grain.

To illustrate that with an example, turning the dial to the value of 100 would create a new, random starting position for each grain in the range of 9900 - 11000, assuming our original file position was 10000, as in the example above.

This allows for each grain to have independent starting positions, creating a more interesting clusters, that do not sound so repetative and static.

Above described functionality is implemented like so:

```
if (filePositionOffset == 0 || (this->filePosition -
filePositionOffset) <=0)
grain.grainDataStartPosition = this->filePosition;
```

Where ‘filePositionOffset is the value of the dial.

Number of active grains

There can be up to 10 grains, meaning that the 2D vector of grains can contain up to 10 vectors. The main struct of one grain looks like this:

and the 2D vector like this:

- Adding grains

  Adding grains to the stream is done by simply adding a grain vector to the 2D vector containing all the grains -¿ the stream of grains. In code referenced as 'grains'.

- Removing grains

  Conversly, the removal of grains is done in exactly the same way, only using 'pop_back' rather than 'push_back', and simply removing howver many vectors (grains) indicated by the dial.

- Control over the stream of grains

  One function is implemented in order to decide whether the user wants to add, or remove grains from the main stream like so:

Because of this, the interaction is kept quite simple, and the size can be decided with the use of only one dial. If the value of the dial if bigger than current size of the stream, we remove grains to match the value. Conversly, if the dial value is smaller, we add grains to the 2D vector 'grains'.

Pitch

Each grain, as defined in the 'OneGrain' struct defined above, has a variable called grainDataPitchScalar, responsible for changing the pitch of grains independently. In the spirit of simplicity, however, in this perticular implementation, the pitch scalar is kept global, with the possibility of extension.

It is binded to the Pitch dial visible in the GUI, which changes the 'pitchOffsetForOneGrain' variable, which is then sent to the :

The grain.grainDataPitchScalar can therefore only have values between THIS and THAT, consequently either speeding up or slowing down the playback of each grain, and affecting the pitch.

Global gain

The global gain of the signal is simply controlled by the globalGain variable, which is used for scaling each individual sample sent to the audio buffer like so:

```
sample *= static_cast<float>(globalGain)
```

The globalGain variable is directly controlled with the global gain dial in the GUI.

Sound output

The sound output is mostly handled by the JUCE library, with the use of the provided 'getNextAudioBlock' function. The audio block, or buffer is being filled with samples determined in the Grain class, more specifically, in the 'createGrain' function contained within that class:

All the functionality needed to determine all the previously mentioned parameters is here. What is being returned is samples, one by one with which to fill both channels of the buffer.

Chapter 4

Results and Analysis

As previously stated, the success of this project is dependent on two things. The parameter predictions must in fact bear some resemblance to the target sound, and be more than a mere randomization of parameters. Secondly, as it is a user centered program, the users must feel like it indeed generates sounds that they find similar to the supplied input, and not less importantly, that the program is interesting to them, that an interaction it commences has some value, and colloquially that it is fun to use.

4.1 Methodologies

With these goals in mind, both qualitative, and quantative analysis were conducted. The qualitative part was made up of a questionnaire, as well as interviews during, and after using the program. This has allowed for immediate feedback, as well as a possibility to reflect upon the experience, and the ability to share those reflections later. The quantative evaluation was conducted by comparing sepctra of target, and predicted sound. Both generated with the instrument itself, as well as with users' voices, to provide a more realistic scenario.

4.2 Sound similiarity

4.2.1 Quantative evaluation

In order to determine if the predictions made are viable, and how well do they work, a form of white box testing had to be made. 10 different sets of parameter values were determined, and for each one a spectrogram, together with MFCC values were made. Then, predicions on these MFCCs values were done, following by extracting spectrograms of the results and comparing them agains the target values. Moreover, euclidean distance between the MFCC values was calculated. The reasoning behind that was the fact that as a matter of fact the parameter values predicted can differ quite largely from the target values, as certain sounds are likely possible to achieve using more than one parameter setting.

Spectra

MFCCs distance

4.2.2 Qualitative evaluation

Equally important as the technical results, are the impressions of users, and their enjoyment of the experience, as well as their assessment of how similar the predictions are to their input.

(actually, maybe only about similiarity above, and enjoyment below, as user experiences?)

4.3 User experience

- statistics, tables and graphs

- qualitative as well as quantitative data

- comparison between random button, and the algorithm

- user testing

- questionnaires for users

- no interpretation of results

Chapter 5

Discussion

5.1 Summary of Findings

5.2 Evaluation

5.3 Future Work

Chapter 6

Conclusion

.1 Title of Appendix A

Bibliography

[1] *Audacity ® — Free, open source, cross-platform audio software for multi-track recording and editing.* en-US. URL: https://www.audacityteam.org/ (visited on 05/03/2019).

[2] A. Eronen. "Comparison of features for musical instrument recognition". en. In: *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No.01TH8575)*. New Platz, NY, USA: IEEE, 2001, pp. 19–22. ISBN: 978-0-7803-7126-2. DOI: 10.1109/ASPAA.2001.969532. URL: http://ieeexplore.ieee.org/document/969532/ (visited on 05/03/2019).

[3] Sebastian Heise, Michael Hlatky, and Jörn Loviscach. "Automated Cloning of Recorded Sounds by Software Synthesizers". en. In: *New York* (2009), p. 8.

[4] Matthew Herbert. *manifesto — Matthew Herbert*. en-US. 2011. URL: https://matthewherbert.com/about-contact/manifesto/ (visited on 11/18/2018).

[5] *Homepage — Essentia 2.1-beta5-dev documentation*. URL: https://essentia.upf.edu/documentation/ (visited on 05/03/2019).

[6] Andrew Horner, James Beauchamp, and Lippold Haken. "Machine Tongues XVI: Genetic Algorithms and Their Application to FM Matching Synthesis". en. In: *Computer Music Journal* 17.4 (1993), p. 17. ISSN: 01489267. DOI: 10.2307/3680541. URL: https://www.jstor.org/stable/3680541?origin=crossref (visited on 05/03/2019).

[7] *Librosa*. URL: https://librosa.github.io/ (visited on 05/03/2019).

[8] Kyle McDonald. *Neural Nets for Generating Music*. Aug. 2017. URL: https://medium.com/artists-and-machine-intelligence/neural-nets-for-generating-music-f46dffac21c0 (visited on 11/18/2018).

[9] *Oneohtrix Point Never on the process behind his latest leap into the unknown, Garden of Delete*. EN_GB. Feb. 2016. URL: https://www.musicradar.com/news/tech/oneohtrix-point-never-on-the-process-behind-his-latest-leap-into-the-unknown-garden-of-delete-633280 (visited on 11/18/2018).

[10] Kıvanç Tatar, Matthieu Macret, and Philippe Pasquier. “Automatic Synthesizer Preset Generation with *PresetGen*”. en. In: *Journal of New Music Research* 45.2 (Apr. 2016), pp. 124–144. ISSN: 0929-8215, 1744-5027. DOI: 10.1080/09298215.2016.1175481. URL: http://www.tandfonline.com/doi/full/10.1080/09298215.2016.1175481 (visited on 11/19/2018).

[11] Hiroko Terasawa, Malcolm Slaney, and Jonathan Berger. “Center for Computer Research in Music and Acoustics (CCRMA) Department of Music, Stanford University Stanford, California”. en. In: (2005), p. 8.

[12] M. J. Yee-King, L. Fedden, and M. d’Inverno. “Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.2 (Apr. 2018), pp. 150–159. ISSN: 2471-285X. DOI: 10.1109/TETCI.2017.2783885.

[13] Matthew John Yee-King. “Automatic Sound Synthesizer Programming: Techniques and Applications”. en. In: (), p. 180.

[14] Matthew Yee-King and Martin Roth. “SYNTHBOT: AN UNSUPERVISED SOFTWARE SYNTHESIZER PROGRAMMER”. en. In: (), p. 4.