7MU009 Music Computing – Evaluation & Report

Antonia Sloan
MSc Audio Technology
7MU009 Music Computing 2020
For attention of: Richard Burn, Dr. Mathew Dalgleish & Matthew Bellingham

This essay is going to discuss and evaluate the algorithm that has been created for the 7MU009 module. The algorithm, (created in Pure Data) can be located in the same folder as this document under the title "*Sloan_Antonia_1511347_7MU009_Algorithm*" (*see appendix 1*).

IDENTIFYING THE PROBLEM

Broadly speaking, the term *algorithm* can be referred to as a set of rules, or steps that are undertaken to complete a take or solve a problem (Corman et al., 2009). However, this report is specifically interested in *computational algorithms*, which can be better defined as "*a sequence of computational steps that transform the input into the output*" (Corman et al., 2009). In this context an algorithm can be considered as a tool, that gives a user the ability to solve a clearly defined problem (Corman et al., 2009). It is important to note that the use of the term *problem*, within computer science, specifically refers to any task that is trying to be solved, computationally (Goldreich, 2008). However, for a problem to be resolvable, it must be "*clearly defined*", which means that both the inputs and "*desired outputs*" are feasible and can be produced (Bronson, 2009). The patch that has been created for this module does not solve a specific problem but instead aims to offer a solution to a class of problems within a wider *problem space*. The aim for this project, was to create an algorithm that could generate unique musical sequences based on numbers that are fed into a system. The initial objective had been to use this algorithm as a way of generating unique ringtones for phone contacts, based on the digits within their phone number. However, when going through the process of creating the patch, it was realised that it could potentially also be used as a way of recognising patterns within numerical data sets (*see appendix 2*). This general idea was then honed into a more well-defined problem that was capable of being solved computationally. The problem was defined as follows:

A string of numerical data is entered into a text file. This data may contain any numerical value and there is no limit to how many distinct numbers may appear within the file.
As each numerical value is read by the algorithm it will arbitrarily trigger one of three oscillators; a sine wave, sawtooth or a square wave, which will then play a frequency relative to the original value read. The length of time the sound is audible for is not fixed, as every time the initial value changes, it triggers the selection of a nondeterministic value between 500ms - 1000ms, which is then used as the decay time.
The sequence repeats until stopped by the user.

Defining the problem and understanding its potential use then made it possible to determine which coding software would be most appropriate when actualizing the algorithm.

THE CANDIDATE PROGRAMMING LANGUAGES

There are many different types of programming languages currently available (TypesnUses, 2019), however, this project shall only focus on two; visual-based programming languages (VPL) and text-based languages (Weintrop and Wilensky, 2017). Text-based languages are perhaps the most recognisable form of coding (Cash, 2018), however, there are many different types of text-based languages. For example, C++, Java and Python, are all different types of programming languages that each offer their own unique "*syntax and semantics*", whilst still being based on the fundamental idea of using text-based logic to generate algorithms or manipulate machine performance (TypesnUses, 2019).

Two terms that often come up in relation to text-based programming languages are "*high-level*" and "*low-level*"; high-level referring to programming languages that are easier to understand and use, and low-level referring to languages that use a more complicated syntax that is perhaps more difficult to understand and apply (*see appendix 3*). Most commercial coding languages tend to be high-level (though to differing degrees), as their parent companies want to encourage people to engage with their product and that is less likely to happen if the language they have created is difficult to understand and use (Teach Computer Science, 2019). Two programming languages that are regarded as being relatively high level are the text-based language, SuperCollider (McCartney, 2002) and VPL, Pure Data (Puckette, 1997). Both Pure Data and SuperCollider are open source programming languages (Magnusson, 2005), which is beneficial for individuals who choose to use them as it means there is an abundance of helpful materials and resources online that are free and readily

available to use (Pure Data Forum, 2017). This is perhaps the main reason why both Pure Data and SuperCollider were chosen as potential candidates to facilitate the building of the algorithm for this module.

In theory SuperCollider is probably better suited to solve the problem outlined earlier, primarily because the language was specifically designed to deal with audio synthesis and algorithmic composition (SuperCollider, 2018a). It is also worth noting that as a text-based language, it would potentially be better at handling larger, algorithms that feature frequently repeated commands or sections of code (Fukazawa, Saito, and Washizaki, 2017) a task that can be extremely time consuming within VPL's (Cash, 2018).

Experimenting In SuperCollider

During the early stages of this module, some experimenting was done within SuperCollider to assist in understanding how best to approach the proposed problem. Those initial experiments, in SuperCollider, can be seen in *figure 1* and found in the parent folder for this file in "*Accompanying Docs*" → "*Experimenting In SuperCollider*".

```
FileReader.read(thisProcess.nowExecutingPath.dirname +/+ "example2.txt");



(
{
    var phone = FileReader.readInterpret(thisProcess.nowExecutingPath.dirname +/+ "example2.txt");
    var first = phone[0].postln;
    var freq = 100 * first.choose;

    SinOsc.ar(freq, 0, 0.5) ! 2
}.play
)



(
SynthDef(\test, { | out, freq = 440, amp = 0.1, nharms = 10, pan = 0, gate = 1 |
    var audio = Blip.ar(freq, nharms, amp);
    var env = Linen.kr(gate, doneAction: Done.freeSelf);
    OffsetOut.ar(out, Pan2.ar(audio, pan, env) );
}).add;
)
```

Figure 1: A Screenshot displaying the code created in the preliminary stages of this project, using SuperCollider. As the reader will observe there are actually 3 sections (blocks) of code, separated using pairs of open and close brackets.

It must first be noted that within *figure 1*, there are actually three separate blocks of code that, in their current state are acting independently of each other. The initial plan had been to start experimenting with these blocks and figure out the different ways they could possibly be interconnected. This next step was not accomplished as SuperCollider was not selected for the project, for reasons that shall be disclosed further on in this report.

```
FileReader.read(thisProcess.nowExecutingPath.dirname +/+ "example2.txt");
```

Figure 2: A closer view of the first line of code visible in figure 1. This line of code will instruct SuperCollider to read the text file located in the given location.

The first block of code visible in *figure 2*, is a line of code that reads "*FileReader.read(thisProcess.nowExecutingPath.dirname +/+ "example2.txt*");". This was not included to accomplish any specific task but rather to test the syntax and that the command functioned as expected. This line of code could then be copied and pasted into the succeeding blocks of code, when required. The "*FileReader"* is an object that allows the programme to read "*space-delimited text files into 2D arrays line by line*" (SuperCollider, 2007a). What this essentially means is that SuperCollider will be able to not only read the file but also offer the user the ability to manipulate its data to some extent (Shiffman, 2009). This object is followed by the method "*.read*". In programming a method refers to a procedure that is carried out in association with a specific object, which in this example would be "*FileReader*" (Study.com, 2013). The "*.read*" simply tells the "*FileReader*" object to read whatever file it has been instructed to. This method is then followed by an open bracket and "*thisProcess.nowExecutingPath.dirname*", which has been used to provide a relative path to the file that the user wishes to access (Collins, Cottle and Wilson, 2011), which, in *figure 2*, would be "*example2.txt*". Relative paths allow a programming language, such as SuperCollider, to locate a file by looking relative to the position of the current working directory (Microsoft, 2019). Put simply, rather than having to express the entire path the computer needs to follow in order to get to the desired file (which could potentially be quite lengthy), a relative path tells the programme to look in the same folder that the current file is in (Microsoft, 2019).

```
(
{
    var phone = FileReader.readInterpret(thisProcess.nowExecutingPath.dirname +/+ "example2.txt");
    var first = phone[0].postln;
    var freq = 100 * first.choose;

    SinOsc.ar(freq, 0, 0.5) ! 2
}.play
)
```

Figure 3: A closer look at the second section of code visible in figure 1. This section of the code is effectively a synthesizer that uses variables in order to help it generate sound

The next block of code to be discussed can be seen in *figure 3*. This section of code is essentially a rudimentary version of a synthesizer, that makes use of SuperCollider's *local variables* (SuperCollider, 2011a). A local variable is delineated with the term "*var*", which is then followed by whatever title the user wishes to assign to that variable (SuperCollider, 2011a). Local variables are variable objects that, once declared, will only work within the block of code they have been defined in, they will not function outside of that block (SuperCollider, 2011a). For example, if the "*var freq*" variable were to be used in *figure 4*, it would not work as it has not been given any description or definition within that block.

In *figure 3* there are three separate local variables that can be seen:

- "*var phone*"
- "*var first*"
- "*var freq*"

Each of these variables is followed by an equal sign, which tells SuperCollider that the following information is going to be the definition of the proceeding variable title (SuperCollider, 2011a). The first variable, "*var phone*" is followed by "*FileReader.read(thisProcess.nowExecutingPath.dirname +/+ "example2.txt"*"" which is the same line of code that can be seen in *figure 2*. This means that, within this block of code, instead of having to write "*FileReader.read(thisProcess.nowExecutingPath.dirname +/+ "example2.txt"*" every time you wish to recall the text file, the word "*phone*" can be used instead, which will keep the code compact and readable, as well as help the user avoid potential syntax errors. Use of these variables can be seen in "*var first*", as instead of writing out the whole "*FileReader*" command, it has been replaced with "*phone*", so it reads "*var first = phone[0].postln;*". The square brackets are used to define a collection of items, in this case there is only one that reads "*0*", which is then followed by "*postln;*". The "*postln;*" acts as a print object and will print the specified code in the post window of the interface, which in *figure 3* would the number "*0*" (SuperCollider, 2011b).

The last variable that can be seen in *figure 3* is "*var freq = 100 * first.choose;*". The first part of the variable is the numerical value "*100*" followed by an asterisk indicating that the following section of the code, (the "*first*" variable), will be multiplied by 100. Due to the way the previous variables have been defined, when the "*first*" variable is multiplied by 100 it means that it is actually each piece of numerical data within the "*example2*" text file that is being multiplied by 100 (*see appendix 4*). The line then ends with the method "*.choose;*", which will choose an arbitrary value from the preceding data set, with all values having an equal chance of being chosen (SuperCollider, 2007b).

The last line of code in this block reads "*SinOsc.ar(freq, 0, 0.5) ! 2*". The "*SinOsc*" is one of the sine wave oscillators available within SuperCollider (SuperCollider, 2011c). The "*.ar*" method that follows "*SinOsc*" stands for '*audio rate*' and essentially tells SuperCollider that the object is an audio object and as such is going to produce an audio signal. The "*SinOsc*" object has 4 default arguments, which are "*freq*" (frequency), "*phase*", "*mul*" (multiplier), and "*add*" (SuperCollider, 2011c). In *figure 3*, the freq argument has been replaced by the variable "*freq*", so the numbers being read from the "*example2*" file, (after being multiplied by 100), will act as the frequency. After this the oscillator has been assigned a phase value of 0 and the output level will be multiplied by 0.5, making it audible when triggered. These arguments are then ended with a close bracket. Following the "*SinOsc*", there is an exclamation mark followed by the number 2, which is essentially shorthand for the duplicate command within SuperCollider (SuperCollider, 2011d). This command instructs SuperCollider to copy "*SinOsc.ar(freq, 0, 0.5)*", effectively changing the audio signal from mono to stereo.

The block of code then ends with; a curly bracket, the method "*.play*" and a final close bracket to indicate the end of the block. When the section of code shown in *figure 3* is triggered, the "*.play*" will tell SuperCollider to play the code.

```
16
17 (
18 SynthDef(\test, { | out, freq = 440, amp = 0.1, nharms = 10, pan = 0, gate = 1 |
19     var audio = Blip.ar(freq, nharms, amp);
20     var env = Linen.kr(gate, doneAction: Done.freeSelf);
21     OffsetOut.ar(out, Pan2.ar(audio, pan, env) );
22 }).add;
23 )
```

Figure 4: The last section of code from figure 1. This block functions as a "SynthDef" which is effectively a set of instructions that can be executed by the creation of a subsequent Synth (SuperCollider, 2007c).

The final block of code, shown in *figure 4*, is something called a "*SynthDef*". In SuperCollider a "*SynthDef*" essentially defines a sound/function that can then be executed by the creation of a corresponding synth (SuperCollider, 2007c).

While these initial experiments served to confirm SuperCollider's suitability to the task at hand, they also highlighted the significant research and learning required to use the programming language effectively. The code seen in *figure 4* was taken from the SuperCollider help website (SuperCollider, 2011e) with the intention of effectively reverse engineering the code. However, when researching each of the arguments within the "*SynthDef*" object and understanding their function and application, it was realised how much time was being spent on essentially becoming familiar with the programme. This did not seem like an efficient use of the time given to complete the project. The "*SynthDef*" was just one element of the language and this same learning process would undoubtedly have to be repeated for most of SuperCollider's syntax. Whilst being familiar with multiple different programming languages is undoubtedly a useful skill to have, it was felt that, given the time restraints, this module was perhaps not the best occasion to develop said skill. Whilst not being fluent with SuperCollider's syntax would not have brought the creation of the algorithm to a complete stop, it would potentially cause the creation process to move a lot slower. As such, it was decided that it would be more beneficial to move forward using Pure Data instead.

The primary decision to use Pure Data was based on past experience with the language, however, there were other factors that helped inform this decision.

PURE DATA: THE RATIONALE

Pure Data is a visual-based programming language that is capable of dealing with audio synthesis and algorithmic composition, like SuperCollider (Pure Data, 2018). However, Pure Data also possess a wide array of tools that allow for broader application within multiple multimedia formats (Pure Data, 2018). These tools can be useful when faced with challenging sections of code, giving users multiple ways of addressing potential problems. For example the [spigot], [route] and [moses] objects within Pure Data all offer the same basic function (allowing data to pass through with some form of preliminary filter), however, they accomplish this task in different ways (Kreidler, 2009a). Their different capabilities offer the user different methods of addressing one problem whilst also offering them the option to extend their code in different ways (Kreidler, 2009a). As Pure Data is a VPL, the way that these options are displayed, can often inspire ideas as to how the algorithm can be furthered. VPL's can also be very beneficial to those with prior knowledge of programming, as languages such as Pure Data, can clearly show the direction in which data is flowing through a patch (Petre, 1995), (*see appendix 5*).

One of the major shortcomings of text-based languages is that it can sometimes be difficult to understand what exactly is going on inside of the code, unless the creator has included some form of labelling (McCartney, 2002). This may be why some individuals choose to use VPL's such as Pure Data, or MAX as in some cases they can be considered more high-level, in comparison to text-based languages, such as SuperCollider (McCartney, 2002).

Although it was not known exactly how large the algorithm would be, it was estimated that it would probably be relatively large, based on its intended purpose. Creating an algorithm of that size, in SuperCollider, could cause some issues regarding how '*readable*' it is when checking back through the code. It should be noted that it is possible to make the coding in SuperCollider more compact and easier to read, however, this relies upon the existing knowledge of the person writing the algorithm. As this knowledge has yet to be acquired, Pure Data was chosen instead, as it was known that even if the patch became larger than expected, it was probably going to be somewhat easier to follow and understand, for a non-expert programmer, than if the same algorithm were written in SuperCollider (*see appendix 6*).

It is also important to note that when creating or modifying large algorithms, there is an increased risk of syntax errors occurring within the code (Guah, 2008). Syntax errors can occur in a variety of ways, including incorrect spelling, missing or incorrect punctuation and even the use of incorrect case can cause errors within the code (Denny, Luxton-Reilly and Tempero, 2012) . These errors can occur within any programming language, as they are usually the result of human error rather than an issue with the language itself (Denny, Luxton-Reilly and Tempero, 2012) , however, these errors are usually easier to make within text-based languages (Des, 2015). Text-based languages are more reliant upon the focus of the user and how attentive they are to what they have written, it is not uncommon for errors in text-based languages to go unnoticed until the user eventually tries to run the code (Guah, 2008). This problem is a bit easier to avoid within VPL's because they often have measures put in place that make it hard or impossible to move on until the problem has been addressed (Des, 2015). For example, in Pure Data, if an object's name has been spelled incorrectly or with the wrong case, the object will appear to have a "*dashed line*" around it and the error window will read the object's name and "*couldn't create*" (Puckette, 1997). Pure Data will try to resolve any syntax errors as soon as they are made but it isn't always easy to identify, for example, if the correct spelling has been used but the incorrect punctuation has or has not been added (Puckette, 1997). For instance, certain object commands require a tilde (~) sign after them in order to process an audio signal, however, they can exist without that symbol for application elsewhere (e.g. [*] and [*~]) (Blažíček, 2014). Pure Data still tries to avoid these kinds of errors by making it difficult to connect two objects that do not function together, however these precautions are not infallible and it is still possible to connect incompatible objects to each other, or assign the wrong inputs and outputs to corresponding objects (Puckette, 1997). Because of this it was noted that, as the algorithm was being built, it would have to be checked regularly to see if everything were functioning correctly and make any necessary amendments as soon as possible.

Once Pure Data had been chosen as the language for this project, work could then begin on actualising the solution to the problem created for this algorithm.

EXPLAINING THE ALGORITHM

The following section of this report shall discuss how the key parts of the algorithm function, whilst also offering an explanation as to why they have been included in the final version of the algorithm. It is important to note that each of the sub-patches visible in *figure 5* contains the exact same contents, the only differences being the names that have been assigned to the [send] and [receive] objects and sub-patches. As such, this section will primarily be discussing the contents of [pd value1] but the reader should assume the explanations are relevant to all other sub-patches.

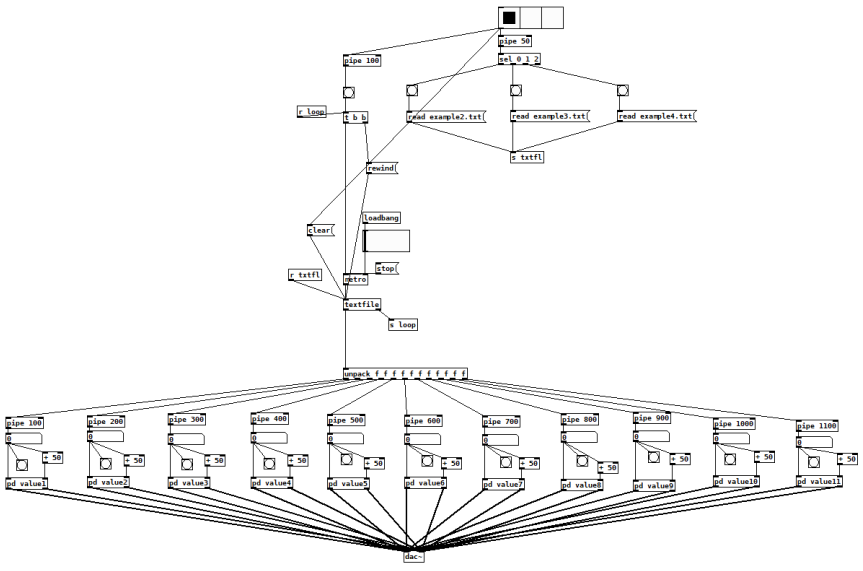

Figure 5: A print screen of the complete algorithm as it appears in Pure Data. The algorithm utilises 11 different sub-patches. Each contains the same contents however; the names of certain objects have been changed to avoid any data being sent to the wrong object and causing errors to occur.

Triggering The Algorithm

The patch begins with a radio object, which is the rectangular object separated into four segments at the top of *figure 6*. As a user clicks on each separate segment it will cause the object to output a number that is one lower than the current segment (*n-1),* e.g. if the third box is selected it will output a number 2 (FLOSS Manuals, 2006). The radio object is then connected to three separate items, one of which is a [pipe] object, that reads [pipe 100]. A full description of how the [pipe] object works will be given later in this section, however, in summary, the [pipe] object acts like a delay (Kreidler, 2009a). When a numerical value is received through its input, the object will hold it for a predetermined amount of time (in milliseconds), before the value is then passed through the outlet (Kreidler, 2009a). This means that after 100 milliseconds the [pipe 100] will output the stored value into the [bang] object that is connected to its outlet. The [bang] object will register the received value as a trigger, of sorts, which will then cause the object to let out another subsequent bang through its outlet and into the inlet of the [t b b] object. The [t b b] is something called a [trigger] object. Once again, the way this object functions shall be disclosed later within the report. However, put simply, the object releases a series of bangs from right to left, depending on how many outlets have been assigned (Puckette, 1997). The [t b b] object in *figure 6* only has two outlets and thus will only output two bangs. The first of these bangs will be sent to a message box that contains a message that reads {rewind}, which is an instruction that is to be sent to the [textfile] object it is connected to. The second bang will be sent to a [metro] object. The [metro] object is effectively a metronome that when triggered will output a series of bangs that occur at regular intervals (Kreidler, 2009a). There are two ways to determine the length of these intervals, one of which is to include a numerical value within the [metro] object that will represent the time in milliseconds between each bang, for example, [metro 500] (Kreidler, 2009a). The second way to control this interval, and the way that has been demonstrated in *figure 6*, is by sending a numerical value through the rightmost inlet of the [metro] object (Kreidler, 2009a). This allows a user to vary the length of the intervals by an arbitrary amount, effectively giving them an easy way to control the speed of the [metro] object (Kreidler, 2009a). In *figure 6* a slider object has been used to achieve this. Using the properties tab, the lower and upper limits of the slider were changed to 1000(ms) and 100(ms) respectively. It is also worth noting that a [loadbang] object has been attached to the inlet of this slider which will automatically send a bang whenever the patch is loaded (Blažíček, 2014). It has been noted, from past experience with Pure Data, that sometimes when a patch is loaded, the slider object will not initialize properly and may load with a default value of 0 rather than the assigned lower bound. The [loadbang] object is a way of resolving this problem as when the patch is loaded a bang will automatically be sent to the object, forcing it to initialize with the lower limit.

The [metro] object is then connected to the inlet of the [textfile] object. The [textfile] object is essentially Pure Data's version of "*FileReader*" in SuperCollider. It is capable of writing text files as well as reading them and retrieving any subsequent data they might contain (Kreidler, 2009b). It is important to note that the [textfile] object will only look at the text files that are located within the same folder as the patch, unless instructed to do otherwise (Kreidler, 2009b). As such, the [textfile] object serves as the medium through which the text files located in the parent folder of this document, can be accessed. A user is able to exhibit some control over the data received by the [textfile] object, with the use of message boxes (Kreidler, 2009b). This then leads back to the {rewind} message connected to the inlet of the [textfile] object. The {rewind} message will instruct the [textfile] object to go back to the start of the text file it is currently reading (Kreidler, 2009b). This was included with the intention of creating a loop, so the user would not have to keep triggering the patch every time the [textfile] reached the end of the text file. This loop is triggered using a [send] object entitled [s loop]. The [send] object is essentially a way of wirelessly connecting two or more objects, with the intended receiving object(s) being allocated with the corresponding receive command (FLOSS Manuals, 2009a), which in this instance would be [r loop]. Once [textfile] reaches the end of the loaded text file, it will output a bang from its rightmost outlet, which will then be sent through [s loop] and received by [r loop]. The [r loop] object is then attached to the [bang] object located at the top of *figure 6*, causing the [textfile] object to rewind and start again, thus creating the loop. In order to stop this loop, a message box reading {stop} has been attached to the left inlet of the [metro] object, which, when received will instruct the object to stop outputting bangs until the user triggers the patch again (Kreidler, 2009b).

As has already been mentioned the radio object, at the top of *figure 6*, is connected to 3 items. The second of these items is a message box that reads {clear}. This message box will register any value that is output from the radio object as a '*bang*', which will then cause the message, {clear}, to be transmitted to the [textfile] object, effectively deleting everything inside of it (Kreidler, 2009b). The {clear} message has been included within this algorithm as a way of stopping the [textfile] object from accumulating all the different text files into one large subsequent file (Kreidler, 2009b). Every time a new text file is chosen in the radio object, it will cause the {clear} message to trigger, thus wiping the remnants of the last text file from the [textfile] object (Kreidler, 2009b).

Finally, the last item connected to the radio object is another [pipe] object that reads [pipe 50], meaning that it causes a 50ms delay before transmitting the value stored inside of it. This value is then passed on to a [select] object. A full explanation of how [select] objects work will be provided later in this section of the report, however, in short, the [select] object will compare incoming messages and values, against its creation arguments (Kreidler, 2009a). If the input matches, it will be output from the corresponding outlet, if the input does not match, then it is sent through the rightmost outlet on the [select] object (Kreidler, 2009a). The [select] object in *figure 6* has been given the arguments "*0*", "*1*" and "*2*", to match the values that will be output from the radio object. Each of these arguments has its own outlet, and each of those outlets has its own [bang] object attached to it. When the creation argument matches with the value received through the inlet, it will cause the corresponding [bang] object to trigger. Each [bang] object is connected to a different message box, each of which contains one of the following messages:

- {read example2.txt} - first outlet
- {read example3.txt} - second outlet
- {read example4.txt} - third outlet

The outlets of all three message boxes are connected to the inlet of the [textfile] object via a [send] object ([s txtfl]), and a [receive] object ([r txtfl]). These messages essentially tell the [textfile] object what text file to load and where that text file is (Kreidler, 2009b). The "*read*" part of the message indicates that the text file is stored locally and can be located in the same parent folder as the algorithm (Kreidler, 2009b). The "*example2.txt*", "*example3.txt*" and "*example4.txt*" are the names of each text file, and so whichever message box is triggered will cause the corresponding text file to be received by the [textfile] object (Kreidler, 2009b).

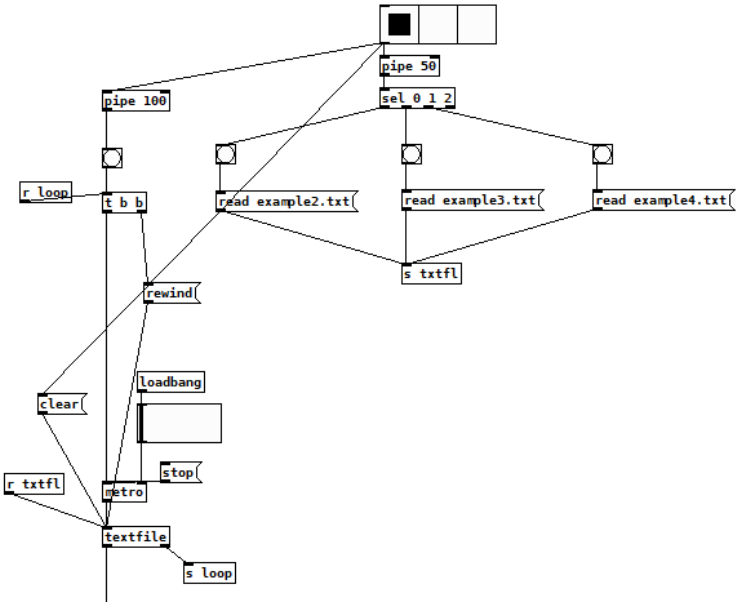

Figure 6: A closer view of the initial part of the algorithm that deals with triggering rest of the patch.

Unpacking The Information

At the top of *figure 7*, there is an object that reads [unpack f f f f f f f f f f]. The [unpack] object is capable of splitting a list into its individual elements (FLOSS Manuals, 2010). These elements will then be distributed through its outlets, the number of which can be determined by the user (FLOSS Manuals, 2010). What this means is that the [unpack] object will receive the list being output by [textfile] and separate each value it detects, sending it through one of the assigned outlets, starting from left to right (FLOSS Manuals, 2010). It must be mentioned that the [unpack] object is able to distinguish different values via the inclusion of spaces and/or commas which cause a 'break' between the atoms (FLOSS Manuals, 2010). For example, the [unpack] object would register -100.578235 as one value but would register -100.57 8235 as two separate values.
The [unpack] object then leads into 11 different [pipe] objects that are succeeded by a single number, the first of which is 100. The [pipe] object is essentially a delay object that will release any messages it receives after a specified amount of time (Kreidler, 2009a). In *figure 7* the first [pipe] object reads [pipe 100], this means that after 100 milliseconds [pipe 100] will release whatever message it has received through its inlet. The [pipe]

command was included in an attempt to separate the sounds being produced and make them more reminiscent of a musical sequence. The patch would function without the [pipe] object, however, it would result in every value being triggered at the same time, creating a massive and potentially dangerous level of sound being produced. The [pipe] objects stop this from happening whilst also creating enough space between values, to let the user process the information their ears are receiving (Yeung, 1980). The [pipe 100] is then attached to a number box that has been added so that the user can see if the correct numerical values are being passed through. This number box is then attached to three separate objects:

- A [bang] object
- An arithmetic command [+ 50]
- A sub-patch, entitled [pd value1]

The [bang] object has been added to act as a trigger for other objects inside of the [pd value1] sub-patch, that shall be discussed in a moment. However, it is worth nothing that the bang has been assigned a '*send symbol*' ("*bang1*"), to reduce the amount of connections that there would be in the patch and instead the corresponding '*receive symbol*' can be assigned where it is necessary (FLOSS Manuals, 2009a). The corresponding receive command can be found within [pd value1]. The [pd value1] object is a sub-patch that has been created within pure data, to accommodate the synthesizer section of the algorithm. A sub-patch is a separate window that can be created within Pure Data, to separate and organise sections of code, to avoid the occurrence of *spaghetti coding* (FLOSS Manuals, 2009b). It was felt that if the synthesizer section were not organised in some manner, the patch would be quite hard to navigate and understand especially from the perspective of a non-expert programmer, as such several sub-patches were used as a means of keeping the code organised.

The contents of the sub-patch shall be fully outlined in the next section. However, in summary the sub-patch contains a synthesiser that utilises a feedback loop, of sorts, to generate a more texturally interesting sound. The object that reads [+ 50] feeds into the right inlet of the [pd value1] sub-patch and was added as a way of distinguishing the sounds being produced within the left and right sides of the feedback loop in addition to making the overall sound more texturally interesting.

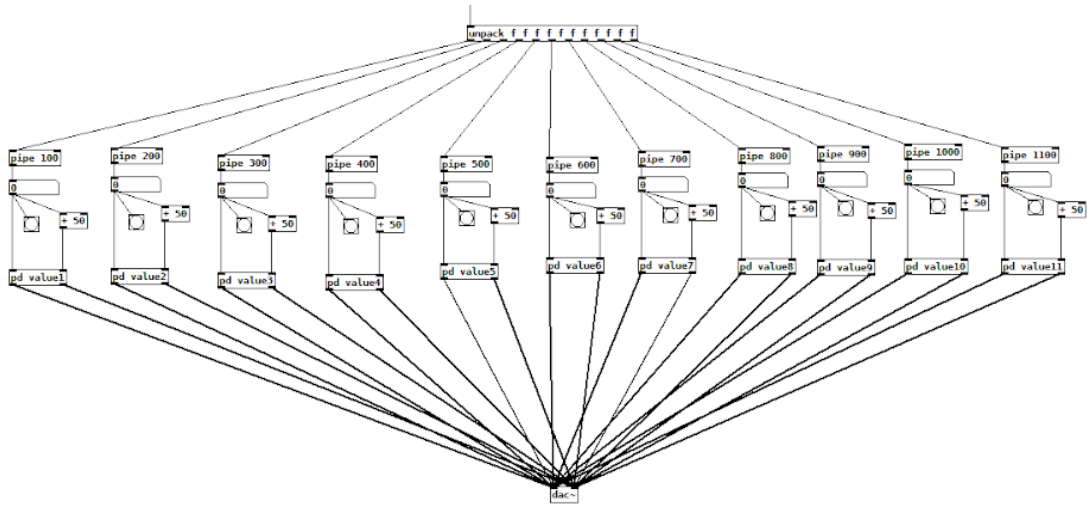

Figure 7: In this image there are actually 11 separate columns that have each been created to deal with one numeric value at a time. The [unpack] object will send these differing values through each of its outlets and into the subsequent corresponding [pd value] sub-patch.

Inside The Sub-Patch

As has been mentioned, inside the sub-patch is a synthesizer, that can be seen in *figure 8*. This synthesizer makes use of some key features that help create the characteristic sound of this algorithm. Rather than discussing the patch in full, this section shall focus on those key features and how they function.

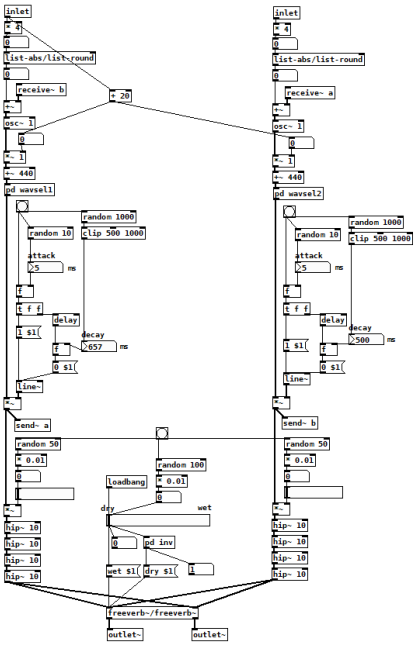

Figure 8: A general overview of what is visible inside each of the [pd value] sub-patches (this print screen was taken from [pd value1]). The image shows two columns that will separately deal with any incoming value before combining that signal via the use of [send] & [receive] objects.

Feedback Loop

Perhaps the biggest contributor to the characteristic sound of this algorithm is the feedback loop that has been included within it. This feedback loop was inspired by the MAX "*FM Surfer*" synth patch, created by John Bischoff (Hartung, 2007). The implementation of this loop can be difficult to see in *figure 8*, so a diagram of how the system operates can be found in *figure 9(ii)*. This diagram was based on another prior diagram (*see figure 9(i)*) drawn by Dr. Mathew Dalgleish (2020).

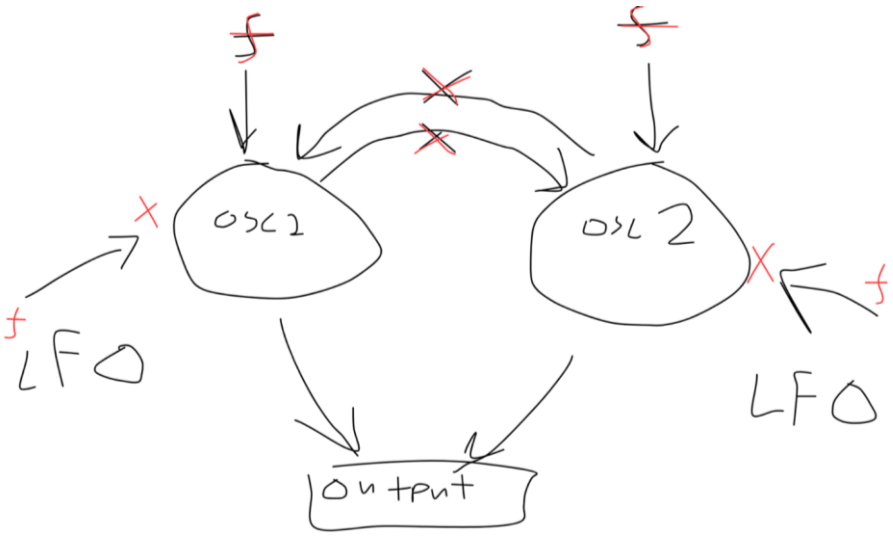

Figure 9(i): This image shows a diagram drawn by Dr. Mathew Dalgleish (2020), explaining the signal path of John Bischoff's FM Surfer algorithm

The key parts of the feedback loop are the [send] and [receive] objects that allow the signal to continuously cycle throughout the patch. In *figure 9(ii)* the two parts that read "*incoming frequency*" represent the two inlets at the top of *figure 8*. As the signal travels through the inlets it will enter the oscillators that exist on both the left- and right-hand sides of the sub-patch. As can be seen in *figure 8*, there are two distinct columns that exist within the sub-patch, that are joined by a [freeverb~/freeverb~] object at the bottom. These two columns are represented by the "*OSC 1*" and "*OSC 2*" in *figure 9(ii)*. As the signal exits the oscillator, on the left side of *figure 8*, and continues through the patch, it will arrive at a [*~] object, which effectively acts as an amplifier

within the context of this patch. In *figure 9(ii)* these amplifiers are located on the curved arrows moving between both oscillators. The outlet of the left amplifier is then connected to the inlet of a [send] object entitled [send~ a] and the corresponding [receive] object ([receive~ a]) can be located on the right hand side of the sub-patch, attached to the right inlet of the [+~] object. As the signal leaves the [receive~ a], the [+~] will add it on top of the signal that is still flowing through the system. The signal then travels down the right column in *figure 8* and will enter another amplifier and subsequent [send] object entitled [send~ b], the corresponding [receive] to this object can be found at the top of the left hand side of *figure 8* attached to the [+~] located on that side of the sub-patch.

As has already been mentioned, in *figure 9(ii)* this looping of the signal is represented by the two arrows flowing in opposite directions. As the signal is passed through the loop it gives the sound more texture, effectively '*adding on*' more layers each time it passes through, with the use of the [+~] object. With each cycle of the loop some of the signal is sent through the outlet of the [*~] object connected to the [send~ a] and [send~ b] objects and to another amplifier object located near the bottom of the sub-patch. This grants the user the ability to be able to hear the signal, however, if left unchecked, it could potentially result in an infinite loop which, in turn, could cause the algorithm to produce dangerous levels of sound. This problem was avoided by including an envelope within the sub-patch, which effectively cuts off the signal before it can reach that level.

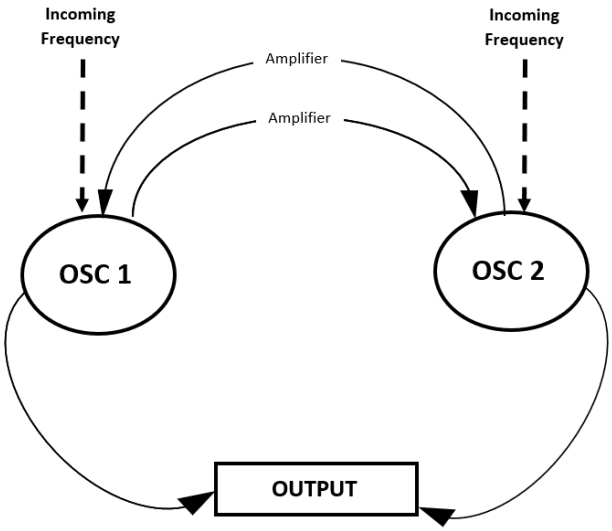

Figure 9(ii): A diagram based on figure 9(i). This diagram is a rough representation of what can be seen in figure 8. The direction of each arrow indicates the direction in which the signal is travelling whilst inside of the feedback loop.

Wave Select Section

In *figure 8* there is a sub-patch entitled [pdwavsel1]. This sub-patch contains the section of the algorithm designed to select one of three different waveforms, in a pseudorandom fashion, for use within the algorithm.

As the signal enters the sub-patch it is sent to three separate oscillators, each using a different waveform to generate sound. These oscillators are then connected to the left inlet of an amplifier object ([*~]), that also have [toggle] objects connected to their right inlets. Here, the [toggle] objects have been used as a rudimentary form of volume control. When a [toggle] object is triggered it will output either a 1 (switched on) or a 0 (switched off) (Kreidler, 2009a). As the outlets of the [toggle] objects are connected to the right inlet of the amplifier, it interprets these values as volume levels. When the [toggle] objects are off the amplifier will receive a value of 0 and no sound will be audible from the oscillators. However, when the [toggle] objects are switched on, they will output a value of 1 into the amplifier, meaning sound will be audible from the corresponding oscillator.

It is important to note that, in the [pdwavsel1] sub-patch, only one [toggle] can be switched on at any time due to the section of code that is connected to the inlets of all three [toggle] objects and is visible in *figure 10*. This section starts off with a [receive] object entitled [r bang1], which is the corresponding receive message to the [bang] object mentioned in "*Unpacking The Information*". The outlet of the [receive] object is then connected to two [random] objects; [random 100] and [random 3]. As [r bang1] receives a bang it will trigger both [random] objects to output arbitrary integer values between 0 and the value specified in the creation argument. The outlet of the [random 3] object is connected to the inlet of a [select] object (*see appendix 7*).

The [select] object is a comparison object that can compare any messages it receives against its creation arguments, which will each have their own designated outlet (Kreidler, 2009a). The [select] object will compare any incoming messages against its arguments, if the message matches any of those arguments it will be output from the corresponding outlet (Kreidler, 2009a). If the incoming message does not match any of the creation arguments then it will be output from the rightmost outlet (Kreidler, 2009a). In *figure 10* there is [select] object that contains the values "*0*", "*1*" and "*2*" ([select 0 1 2]), with each of the three values representing one of the three oscillators used within the [pd wavsel1] sub-patch. As the [random 3] object outputs a 0, 1 or 2, it will be sent out of the corresponding outlet within the [select 0 1 2] object it is connected to. Each of these outlets is connected to three message boxes, two of which will read {0} and one that will read {1}. The {1} message box has been attached to the [toggle] of the oscillator the user wishes to switch on when the corresponding value is output from the [select] object. The two {0} message boxes will be attached to the inlets of the other two [toggle] boxes that can be seen in *figure 10*. As such when those message boxes receive a bang, they will output a 0 value into the [toggle] boxes, causing them to switch off or remain off as another oscillator is triggered (*see appendix 8*).

As was mentioned earlier there is also a [random 100] object attached to the [bang] object inside of the [pd wavsel1] sub-patch. This object is intended to automate the pulse width modulation of the square wave inside of [pd wavsel1] (*see figure 10* {[phasor~] → [expr~ $v1 > $v2]}).

The output of the [random 100] object is attached to a number box that is connected to the left inlet of an arithmetic command box that reads [/ 100]. The number box has simply been included to display the resulting output of the [random 100], whilst the [/ 100] will divide all the values output from [random 100] by 100. Like a lot of other objects in Pure Data the [expr~] object takes the values of 0 and 1 to represent minimum and maximum, respectively (FLOSS Manuals, 2008b). By dividing the initial values by 100 they then fit within the 0 - 1 range that is recognised by the [expr~] object. It was hoped that varying the pulse width of the square wave would result in a more texturally interesting sound being produced, that the user would perceive as being more engaging to listen to, compared to if the same pulse width was present every time the square wave was triggered.

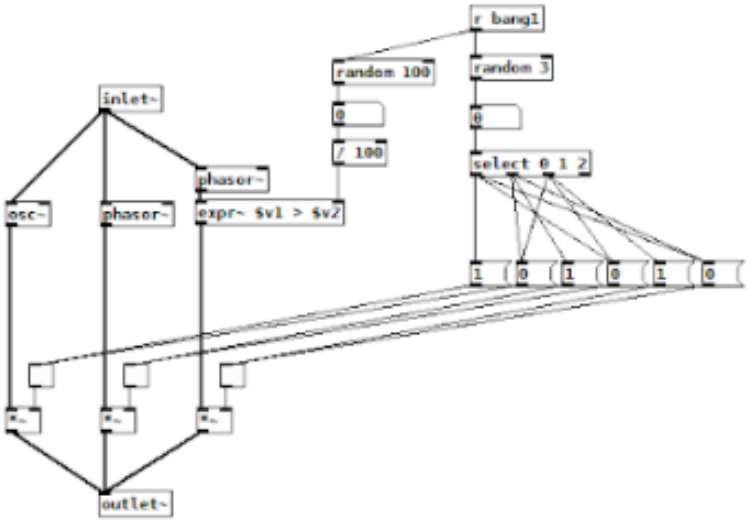

Figure 10: A print screen showing the contents of the [pd wavesel] sub-patch. On the left-hand side of the image the reader should see three separate oscillators that each possess their own [toggle] switch. Each of the [toggle] objects are connected to a section of code on the right-hand side of the image. This section of the algorithm will output numbers that will cause the arbitrary selection of one of the oscillators for every value that passes through the parent [pd value] sub-patch.

Envelope Section

During the initial stages of the building the algorithm there was an issue regarding the length of the sound being played. It had been found that once the patch was triggered it would result in one continuous sound being produced and then the other unpacked values would stack on top of it as they were triggered. This was not ideal considering the purpose of the algorithm was to be able to clearly distinguish between the sonification of each value. It was decided that the best way of eliminating these kinds of problems was by including an envelope within the patch. A closer view of the envelope used within the patch can be seen in *figure 11*, and feeds into the

right inlet of an amplifier object ([*~]). The way that the section is triggered shall be discussed in a moment, however, the first object of interest is the number box feeding into the right inlet of the [float] object at the top of the patch. This value determines the sound's attack time (in milliseconds) which is then passed through the [float] object into a [trigger] object. The function of a [trigger] object depends on the creation arguments that follow it, however, the objects basic function is to receive any form of input and then release that information in sequential order from right-to-left (Puckette, 1997). The [trigger] object is capable of processing any input, but it needs to be told how this input should be processed via creation arguments (i.e. bangs, floats, symbols, pointers, or lists) (Kreidler, 2009a). In *figure 11* the [trigger] object is followed by two floats (f) meaning that as the attack time exits the float and enters the [trigger] object, it will first exit the rightmost outlet before the same value exists the leftmost outlet. The left outlet of the [trigger] object is connected to a message box that reads {1 $1}.

The right outlet of the [trigger] object is connected to the left inlet of a [delay] object. The [delay] object will interpret any numerical object being fed into its left inlet as a new creation argument i.e. a new delay time (in milliseconds) and output a bang after the assigned period of time has passed (Kreidler, 2009a). The outlet of the [delay] object is attached to the left inlet of another [float] object which stores the decay time of the sound, that has been determined by a number box connected to its right inlet. This means that when the [delay] object outputs a bang after the attack time has passed, it will trigger the [float] object, beneath it, to release the number it has stored (decay time). That number will then be transferred into a message box that reads {0 $1}.

As can be seen in *figure 11*, the two message boxes; {1 $1} and {0 $1} have been connected to the left inlet of the [line~] object. The [line~] object takes two values and interprets them as "*a target and a time (in milliseconds)*" (FLOSS Manuals, 2008a), effectively creating a ramp. The [line~] object will then interpolate any external numbers, it receives, to the target value in the given time (FLOSS Manuals, 2008a). It is important to note that the [line~] object features a tilde symbol, which means that it is an audio object (Blažíček, 2014). As such the target values the object receives will refer to the volume of the audio signal it is manipulating. This means when the [line~] object receives the message that reads {1 $1} it will interpret that to mean that the user wants to increase the volume of the synthesizer to 1 in $1 milliseconds (*see appendix 9*). Equally when the [line~] object receives the {0 $1} message, it interprets that as the user wanting the level of the audio signal to drop to 0 in a yet to be determined number of milliseconds.

At the top of *figure 11*, there are two [random] objects and a [clip] object. These were included as a way of automating the attack and decay time of the sound, to introduce more interesting rhythmic ideas into the aural sequence that was being output. As has already been mentioned, there are two [random] objects; one that reads [random 10] and one that reads [random 1000]. The left inlet of both objects is connected to a [bang] object, which features the corresponding receive symbol to the "*bang1*" send symbol discussed in "*Unpacking The Information*". Thus, when a new value is unpacked from the text file it will trigger a bang to be received from the [bang] object seen in *figure 11*. The outlet of the first [random] object ([random 10]), is attached to the inlet of the number box that is responsible for designating the attack time of the sound. When the [random 10] receives a bang in its inlet it will output a pseudorandom integer between 0 and 10, this number then feeds into the number box it is attached to, thus making the value that was output, the new attack time. The second [random] object, ([random 1000]), has a similar function and will output a number between 0 and 1000 when it receives a bang in its inlet. However, it was decided that having a decay time of 0 did not best serve the sound of the whole patch and created a series of very abrupt and harsh sounds to be produced. To stop this from happening the outlet of the [random 1000] was fed into a [clip] object. The [clip] object is typically followed by two creation arguments that act as upper and lower limits of a predetermined range (Blažíček, 2014). The [clip] object functions by comparing any numbers it receives, through its inlet, to its creation arguments and if the number does not lie within its bounds, then the [clip] object will force it into the range by assigning it to either the upper or lower limit (whichever it is closest to) (Blažíček, 2014). In *figure 11*, the [clip] object reads [clip 500 1000], this means that any number output by the [random 1000] that is below 500, will automatically be designated with a value 500. This eliminated the occurrence of the abrupt and harsh sounds that happened before, whilst still giving a wide enough range of values to be used as the decay time.

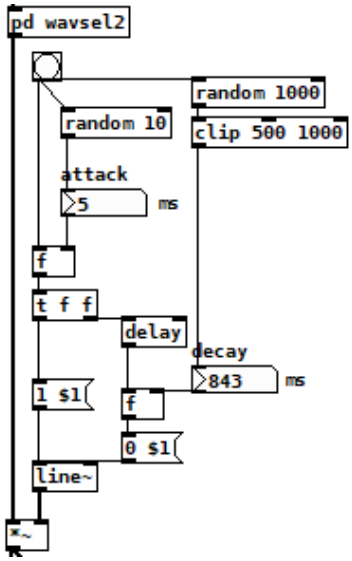

Figure 11: A print screen showing one of the envelope sections that has been included within the algorithm. This section allows the note to decay after a variable amount of time. This stops the occurrence of long continuous sounds that will eventually layer on top of each other, every time they are triggered. It should also be noted that two of these envelope sections exist in every [pd value] sub-patch, to control the signal flowing through both the left and right-hand sides of the sub-patch.

Automating The Volume

As the signal travels further through the algorithm it will encounter an amplifier ([*~]) object. As can be seen in *figure 12* this object has a horizontal slider attached to its right inlet, which has a lower limit of 0 and an upper limit of 0.5. This horizontal slider is effectively supposed to act as a volume control, which, if the user wanted to, they could manually adjust, however, manual adjustment is not necessary as the volume has been automated (to some extent). As can be seen in *figure 12*, there are a series of three objects, connected in tandem, that feed into the inlet of the horizontal slider, the first of these objects reads [random 50]. The [random] object is a "*pseudorandom integer generator*" (Blažíček, 2014), that will output arbitrary integers between 0 and a chosen value (which, in this case is 50), every time it receives a bang within its inlet. To ensure that [random 50] is outputting values at the appropriate intervals, a [bang] object has been attached to its inlet. This [bang] object has been assigned the receive symbol "*bang1*", which corresponds to the [bang] object that was mentioned in "*Unpacking The Information*". This means that every time a new value is unpacked and sent into the [pd value1] sub-patch, it will trigger [random 50] to output a new value.

The [random 50] object is then connected to an arithmetic command that reads [* 0.01]. This object multiplies all the values being output by the [random] object, by 0.01, turning them into decimal values, that lie between 0 and 0.5, which is within the range of the upper and lower limits of the slider. The slider acts as a visual aid within the algorithm, demonstrating that the volume is being automated as it should be.

It was decided that the volume should be automated in an attempt to replicate the musical dynamics that would occur, when a musician plays their instrument in real life (Lumen, 2016). Technically speaking this did work and the desired effect was achieved. However, this effect has perhaps been lost, as the sound being produced from the synthesiser is distinctively electronic and does in fact sound as though it has been generated by an algorithm. The non-deterministic dynamic expression is hard to distinguish between each pitch; however, it does still play a part in what can be heard. It was discovered that removing this section resulted in each pitch being triggered at the same volume which became very tiring to listen to after only a short while, as such, it was decided to keep this section in the final version of the algorithm.

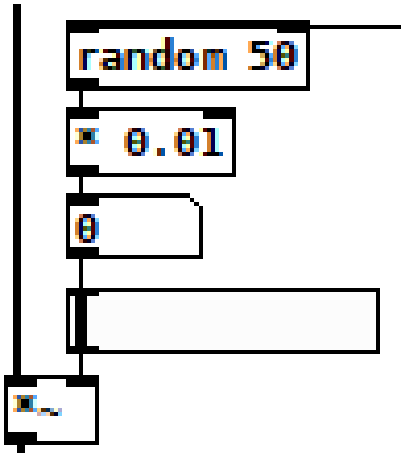

Figure 12: The image above displays an automated section of the algorithm that is able to control the level of the signal being output. The inclusion of this section was done in an effort to try and replicate the dynamics that might be observed when a musician, plays an acoustic instrument.

Adding Reverb

During the final stages of building the algorithm, it was noted that the sound being produced became very taxing to listen to after a short period of time, as it had a very harsh and grating tone. It was determined that one of the most time efficient ways of reducing this issue would be to include a reverb unit that was able to change the perceived depth of the sound, making it a less taxing listening experience (Waves Audio, 2017).

At the bottom of *figure 13* there is an object that reads [freeverb~/freeverb~]. In its simplest form the object would be [freeverb~] however, adding the extra "*/freeverb*~" turns the object into a stereo reverb unit, allowing a wider reverb effect to be employed across the entire sub-patch. The [freeverb~/freeverb~] object has two inlets and outlets that correspond to the left and right stereo inputs and outputs. However, the left inlet of the [freeverb~/freeverb~] object is also able to process several different parameters that can assist in shaping the reverb's sound. The only two parameters that have been added to the reverb unit, of this algorithm, are the {wet $1} and {dry $1} messages, that both have a horizontal slider connected to their inlet. This horizontal slider has a lower bound of 0 and an upper bound of 1, as those are the upper and lower limits of the wet and dry parameters outlined in the [freeverb~] help file within pure data (Cheetomoskeeto, 2010).

As the [bang] object at the top of *figure 13* is triggered by the corresponding [bang] object in *figure 7*, it causes the [random 100] to release an integer value between 0-100. This value is then passed into an object box that reads [* 0.01]. This is a simple arithmetic command that will multiply all the numbers output by [random 100] by 0.01, allowing them to fit within the boundaries of the horizontal slider. The outlet of the slider is attached to the message box that contains {wet $1}, meaning that the value set by the slider will replace the $1 value and indicate how much of the signal will be reverberated.

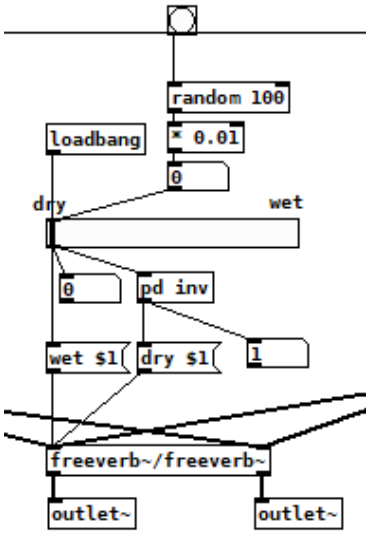

Figure 13: This is a print screen showing the reverb unit that has been added inside of each [pd value] sub-patch. The signal is first sent into the left and right inlets of the [freeverb~/freeverb~] object. Here, a nondeterministic amount of reverb is applied to it, via the objects and message boxes that lead into the left inlet of the [freeverb~/freeverb~].

The outlet of the slider is also connected to the sub-patch [pd inv] the contents of which can be seen in *figure 14*. As the value from the slider enters the sub-patch it enters a [trigger] object that features a float and a bang as its creation arguments. The [trigger] object will first release the initial value from its right outlet into the right inlet of an object box that contains the arithmetic command [-]. The trigger object will then output a bang from its left outlet, which is connected to a message box containing {1}. As the message box receives the bang it will cause the value inside to be passed through to the left inlet of the [-] object. What then happens is the [-] object will compare the numerical values it has received from its left and right inlets respectively and then perform the arithmetic operation indicated within the object box, which in this case is a subtraction. As the number chosen by the slider is effectively the value for the wet signal, it can be reasoned that the value left after subtracting this number from the maximum potential value (1), will be the value for the dry signal. This value is then passed out of the sub-patch and into the message box containing {dry $1}. The value will replace the dollar sign variable within the message box and let the [freeverb~/freeverb~] object know how much of the signal should be unprocessed.

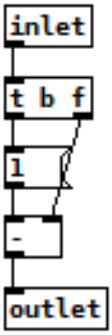

Figure 14: This is image shows the contents of the [pd inv] sub-patch that can be seen in figure 13. This section essentially subtracts the value of the wet signal from 1, in order to figure out the level of the dry signal, which is that passed through the outlet and into the {dry $1} message box.

EVALUATION

Overall, the final algorithm produced was a success and did turn out as expected, for the most part. However, the journey to achieving that success was not without its obstacles.

One of the biggest issues to be had with the final design, is how '*listenable*' the sound is. The current version of the algorithm has a very *harsh*, *nasal* tone and it is suspected that the root cause of this is the feedback loop,

outlined within the "*Feedback Loop*" section of the report. This suspicion is based on the fact that the sound from the algorithm possesses the harsh edge and tone that is typically associated with a high gain effect. Whilst the feedback loop did function correctly and was not the direct cause of any issues within the patch, its presence may have been the cause of this less desirable tone. As the signal enters the loop, and begins each 'rotation', it causes the gain level to increase (Gibson, 2011). As each rotation takes less than a second to occur, the signal can potentially go through hundreds of rotations before being cut off by the envelope, causing the occurrence of the harsh tone. Whilst the presence of high gain levels does not directly affect the functionality of the algorithm, it does arguably affect its ability to achieve its intended purpose. It is going to be harder to continuously analyse and then detect patterns and anomalies within a sound that could be deemed as unpleasant to listen to (Pullum and Rogers, 2009). It must be said that, although it is not ideal, the sound as it currently stands, is a lot better than it was initially. This can be attributed to the addition of the [freeverb~] object as well as the automation on parameters such as attack time, decay time and the pulse width of the square wave oscillator.

In hindsight, it is evident that perhaps the time management surrounding the creation of the algorithm was not as efficient as it could have been. This can partially be attributed to external factors that caused significant changes regarding the working situation of the individual building the algorithm. Although these changes were unavoidable, perhaps more initial preparation and planning could have been undertaken to make better use of the time that was available. Better time management may also have meant that more time could have been designated to figuring out how to include some sections of code that had to be omitted from the final submission, such as the section seen in *figure 15*. This omitted section of code is able to determine the minimum and maximum numeric values within whatever text files are loaded into the [textfile] object. This was achieved using the [list-abs/list-minmax] object which is an external object that must be added to Pure Data's library (Blažíček, 2014). The [list-abs/list-minmax] is able to "*find minimum and maximum in a float-list*" (Blažíček, 2014) and output those values from the left (minimum) and right (maximum) outlets. The plan had been to use these minimum and maximum values as creation arguments within the [expr] objects inside the [pd wavsel1] sub-patch. However, this did not end up happening due to time constraints and some confusion concerning the correct way to incorporate these values into an [expr] object. Whilst the omission of this section of code was not of any great detriment to the algorithm, overall, its inclusion could have changed the direction of the sound, tonally, producing something that is less machine-like and more akin to conventional music.

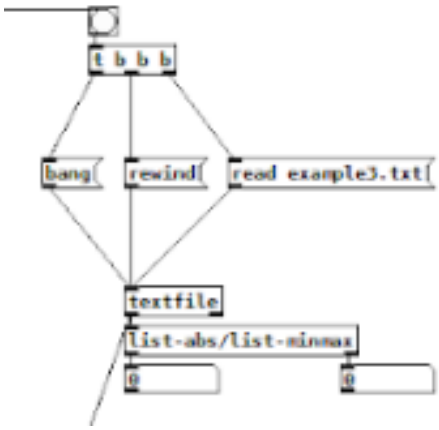

Figure 15: This image shows a section of code that had to be omitted from the final design. This section of code was able to determine the minimum and maximum values of a set of numerical data and was intended to be used as a way of controlling the pulse width of the square wave oscillator seen in figure 10.

Better time management may also have made it possible to make a more thorough attempt at building the algorithm within SuperCollider, as opposed to Pure Data. For the most part Pure Data was an efficient, functional programming language that facilitated the construction of the algorithm outlined in "*EXPLAINING THE ALGORITHM* ". It must also be acknowledged that the visual aspect to Pure Data's design was extremely useful. Using a VPL was especially helpful when it came to understanding the correct way to arrange each section of the algorithm to best suit the flow of data. It would also be remiss not to mention the abundance of information and resources, about Pure Data, that were available online. These resources were extremely useful when building the algorithm, offering explanations on how best to apply certain objects within a piece of code, as well as giving in-depth explanations on how those objects function. For example, when working on the envelope for the synthesizer, the Pure Data community forums were exceptionally helpful, with one user uploading several different examples of envelopes that could potentially be used (mod, 2011). One of these

examples did go on to provide the basis for the envelope that was used in the final version of the algorithm (*see figure 11*). However, despite the efficiency and functionality of Pure Data, it cannot help but be felt that the programme did have some issues and maybe SuperCollider would have been a better programming language to use for this project.

One of the main issues to be had with Pure Data were the several instances where the programme seemed to *crash*, as the algorithm was being tested and built. Whilst some of these crashes could have potentially been caused by some problematic coding, it is suspected, the majority of them were caused by the fact that there were other audio based programmes running on the desktop at the same time. As progress was being saved at regular intervals, this did not cause any major issues. However, there is the potential that one of those crashes could have caused a significant loss of information, which could have proven to be highly detrimental to the quality of the final algorithm.
It must also be acknowledged that some of the crashes that occurred within Pure Data were the result of incorrect or incompatible coding, especially during the very late stages of the project. The algorithm had been running without issue up to a point and then a last-minute decision was made to include a [select] object at the beginning of the patch (*see figure 16*).

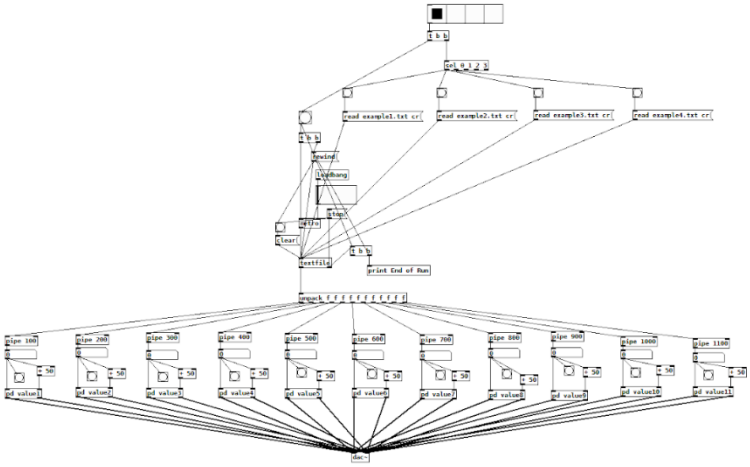

Figure 16: This is a screenshot of how the algorithm initially looked before corrections were made. The reader will note that the trigger section at the top of the algorithm is somewhat different to the one that can be seen in figure 5. The design of this version caused a lot of errors to occur within the patch. To fix this some of the connections and objects had to be removed.

This last-minute addition caused the patch to stop functioning completely. It was unknown what the cause of the issue was and so lecturer Dr. Mathew Dalgleish was contacted and asked for his opinion on what could be causing the error. He pointed out that the error was being caused by the presence of a "*recursive operation*", that was causing the programme to run out of memory (Pure Data, 2008). He then pointed out that these errors were probably being produced by the objects indicated in *figure 17*. In this instance the cause of the error, within the algorithm, could be attributed to absent-mindedness, rather than any fault within the programme itself.

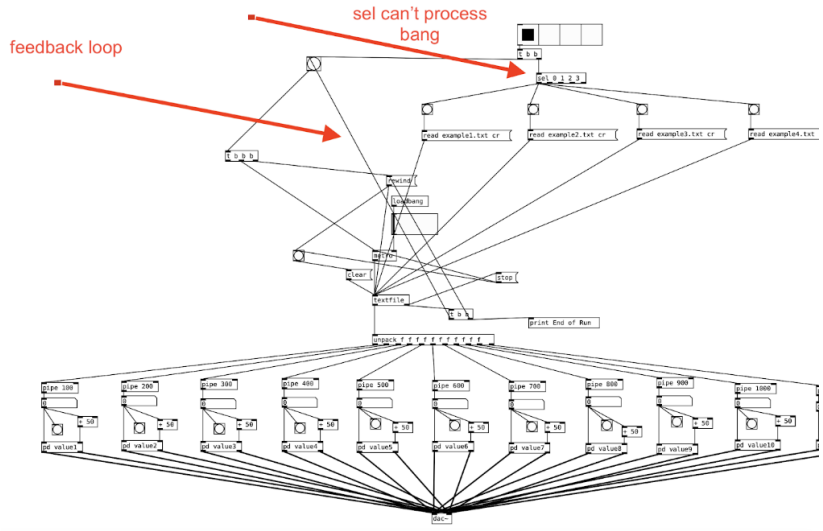

Figure 17: This is an image received from Dr. Mathew Dalgleish, concerning the version of the algorithm shown in figure 16. Here he has identified the location of the errors, as well as what those errors are.

CONCLUSION

It would be interesting to re-create the algorithm again but using SuperCollider instead. It is likely that as SuperCollider uses a different syntax to Pure Data; this would result in a notably different end product. It might then be beneficial to compare the algorithms built in both languages, to see which was actually more effective at accommodating this type of algorithm. The conclusion of this comparison can then help inform future decisions, regarding coding, and what language might best serve the project overall.

There are obviously improvements that could have been made to the final version of the algorithm, however, in its current state it is still somewhat impressive and despite its issues it is still able to fulfil its intended purpose. As it currently stands, the algorithm provides a great basis for potential future endeavours in the field of sonification and it would be interesting to continue working on the algorithm and observe if/how any future changes impact the perceived musical quality of the sound.

REFERENCES

Birnir, A. and Browning, R. (2015) *You never learned to code? Start here.* [online]. [Accessed 14th May 2020]. Available at: <https://mashable.com/2015/12/05/learning-to-code/?europe=true>

Blažíček, M. (2014) *List of Pure Data objects* [online]. [Accessed 13th May 2020]. Available at: <http://blazicek.net/list_of_pure_data_objects.html>

Bronson, G.J. (2009) *C++ for Engineers and Scientists* [online]. [Accessed 13th May 2020]. Available at: <https://books.google.co.uk/books?id=5-WDYaUQV1UC&pg=PA16&lpg=PA16&dq=Problems+typically+specify+an+input+and+a+desired+output&source=bl&ots=HqvPAZjmzS&sig=ACfU3U1jyO_jB1Ym8pwpAkBz-B3sg4M4VA&hl=en&sa=X&ved=2ahUKEwjPrZW4lJbpAhWEYMAKHcmAB6IQ6AEwCnoECAYQAQ#v=onepage&q=Problems%20typically%20specify%20an%20input%20and%20a%20desired%20output&f=false>

Cash, J. (2018) *Text-based vs. Block-based coding* [online]. [Accessed 10th May 2020]. Available at: <https://makelearn.org/2018/06/06/text-based-vs-block-based-coding/>

CDM (2012) *Sharing Music's Source Code: Event Pairs Performances with Code, Patches, Schematics* [online]. [Accessed 15th May 2020]. Available at: <https://cdm.link/2012/10/open-source-music-berlin-retune/>

Cheetomoskeeto (2010) *PURE DATA: 33 Reverb and Delay* [online]. [Accessed 13th May 2020]. Available at: <https://www.youtube.com/watch?v=NzYzxY1WaQU>

Collins, N., Cottle, D. and Wilson, S. (2011) *The SuperCollider Book* [online]. Cambridge, Mass: The MIT Press. [Accessed 15th May 2020] Available at: <http://search.ebscohost.com/login.aspx?direct=true&db=e000tww&AN=550651&site=ehost-live>

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) *Introduction to Algorithms* [online]. [Accessed 19th May 2020]. Available at: <http://ebookcentral.proquest.com/lib/wol/detail.action?docID=3339142>

Dalgleish, M. (2020) Skype correspondence with A. Sloan, 23 April 2020

Denny, P., Luxton-Reilly, A and Tempero, E. (2012) All Syntax Errors Are Not Equal. *ITiCSE '12: Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* [online]. Israel Institute of Technology, Haifa, Israel, 3-5 July, pp. 75-80. [Accessed 19th May 2020]. Available at: <https://dl.acm.org/doi/abs/10.1145/2325296.2325318>

Des, R. (2015) *The maturity of visual programming* [online]. [Accessed 16th May 2020]. Available at: <https://www.craft.ai/blog/the-maturity-of-visual-programming>

Foglia, L. and Wilson, R. (2011) *Embodied Cognition* [online]. [Accessed 22nd May 2020]. Available at: <https://plato.stanford.edu/archives/spr2017/entries/embodied-cognition/>

FLOSS Manuals (2006) *The Interface* [online]. [Accessed 16th May 2020]. Available at: <https://booki.flossmanuals.net/pure-data/the-interface/the-interface.html>

FLOSS Manuals (2008a) *The Envelope Generator* [online]. [Accessed 10th May 2020]. Available at: <https://booki.flossmanuals.net/pure-data/audio-tutorials/envelope-generator>

FLOSS Manuals (2008b) *Square Waves and Logic* [online]. [Accessed 10th May 2020]. Available at: <http://write.flossmanuals.net/pure-data/square-waves/>

FLOSS Manuals (2008c) *Dollar Sign Arguments* [online]. [Accessed 15th May 2020]. Available at: <http://write.flossmanuals.net/pure-data/dollar-signs/>

FLOSS Manuals (2009a) *Wireless Connections* [online]. [Accessed 10th May 2020]. Available at: <http://write.flossmanuals.net/pure-data/wireless-connections/>

FLOSS Manuals (2009b) *Subpatches* [online]. [Accessed 10th May 2020]. Available at: <http://write.flossmanuals.net/pure-data/subpatches/>

FLOSS Manuals (2010) *Messages* [online]. [Accessed 16th May 2020]. Available at: <https://booki.flossmanuals.net/pure-data/dataflow-tutorials/messages.html>

Fukazawa, Y., Saito, D. and Washizaki, H. (2017) *Comparison of Text-Based and Visual-Based Programming Input Methods for First-Time Learners* [online]. [Accessed 13th May 2020]. Available at: <https://www.informingscience.org/Publications/3775>

Gibson, G. N. (2011) *6.1 Gain and Feedback* [online]. [Accessed 13th May 2020]. Available at: <https://www.phys.uconn.edu/~gibson/Notes/Section6_1/Sec6_1.htm>

Goldreich, O. (2008) *Computational Complexity: A Conceptual Perspective* [online]. [Accessed 16th May 2020]. Available at: <http://ebooks.cambridge.org/ref/id/CBO9780511804106>

Guah, M. (2008) *Managing Very Large IT Projects in Businesses and Organizations* [online]. [Accessed 13th May 2020]. Available at: <https://books.google.co.uk/books/about/Managing_Very_Large_IT_Projects_in_Busin.html?id=IQNMK69i4IkC&printsec=frontcover&source=kp_read_button&redir_esc=y#v=onepage&q&f=false>

Hartung, K. (2007) *FM Surfer* [online]. [Accessed 11th May 2020]. Available at: <https://cycling74.com/forums/fm-surfer>

Jones, M. (2018) *Spaghetti Code* [online]. [Accessed 14th May 2020]. Available at: <https://exceptionnotfound.net/spaghetti-code-the-daily-software-anti-pattern/>

Kreidler, J. (2009a) *2.2 The control level* [online]. [Accessed 13th May 2020]. Available at: <http://www.pd-tutorial.com/english/ch02s02.html>

Kreidler, J. (2009b) *4.2 Sequencer* [online]. [Accessed 16th May 2020]. Available at: <http://www.pd-tutorial.com/english/ch04s02.html>

Lumen (2016) *Dynamics and Dynamic Changes* [online]. [Accessed 11th May 2020]. Available at: <https://courses.lumenlearning.com/musicappreciation_with_theory/chapter/dynamics-and-dynamics-changes/>

Magnusson, T. (2005) *ixi software: Open Controllers for Open Source Audio Software* [online]. [Accessed 15th May 2020]. Available at: <http://sro.sussex.ac.uk/id/eprint/46909/1/bbp2372.2005.057.pdf>

McCartney, J. (2002) Rethinking the Computer Music Language: SuperCollider, *Computer Music Journal* [online]. **26**(4), pp. 61–68. [Accessed 13th May 2020]. Available at: <https://www.mitpressjournals.org/doi/abs/10.1162/014892602320991383?journalCode=comj>

Microsoft (2019) *File path formats on Windows systems* [online]. [Accessed 15th May 2020]. Available at: <https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats>

mod (2011) *Line, line~, vline~ and ADSR envelopes* [online]. [Accessed 13th May 2020]. Available at: <https://forum.pdpatchrepo.info/topic/5315/line-line-vline-and-adsr-envelopes>

Nabi (2018) *Hacker News* [online]. [Accessed 14th May 2020]. Available at: <https://news.ycombinator.com/item?id=16069309>

Petre, M. (1995) *Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming, Communications of the ACM*, [online]. [Accessed 21st May 2020]. Available at: <http://dx.doi.org/10.1145/203241.203251>

Puckette, M. (1997) Pure Data, *ICMC*, [online] Available at: <https://www.researchgate.net/profile/Miller_Puckette/publication/230554908_Pure_Data/links/577c1cca08aec3b743366f5c/Pure-Data.pdf>

Pullum, G. K. and Rogers, J. (2009) *Aural Pattern Recognition Experiments and the Subregular Hierarchy* [online]. [Accessed 7th May 2020]. Available at: <http://www.lel.ed.ac.uk/~gpullum/MoL10final.pdf>

Pure Data (2008) *What's A "Stack Overflow"?* [online]. [Accessed 13th May 2020]. Available at: <https://puredata.info/docs/faq/stack-overflow>

Pure Data (2018) *Pure Data* [online]. [Accessed 16th May 2020]. Available at: <https://puredata.info/>

Pure Data Forum (2017) *Home* [online]. [Accessed 16th May 2020]. Available at: <https://forum.pdpatchrepo.info/>

Schules, B. (2017) *High Level and Low Level Languages* [online]. [Accessed 14th May 2020]. Available at: <https://medium.com/@brettschules/high-level-and-low-level-languages-62776d0b89f0>

Shiffman, D. (2009) *Two-Dimensional Arrays* [online]. [Accessed 15th May 2020]. Available at: <https://processing.org/tutorials/2darray/>

Study.com (2013) *Object-Oriented Programming: Objects, Classes & Methods* [online]. [Accessed 15th May 2020]. Available at: <https://study.com/academy/lesson/oop-object-oriented-programming-objects-classes-interfaces.html>

SuperCollider (2007a) *FileReader* [online]. [Accessed 15th May 2020]. Available at: <http://doc.sccode.org/Classes/FileReader.html>

SuperCollider (2007b) *Randomness* [online]. [Accessed 15th May 2020]. Available at: <https://doc.sccode.org/Guides/Randomness.html>

SuperCollider (2007c) *SynthDef* [online]. [Accessed 15th May 2020]. Available at: <https://doc.sccode.org/Classes/SynthDef.html>

SuperCollider (2011a) *Environment* [online]. [Accessed 15th May 2020]. Available at: <https://doc.sccode.org/Classes/Environment.html>

SuperCollider (2011b) *Object* [online]. [Accessed 15th May 2020]. Available at: <http://doc.sccode.org/Classes/Object.html#-postln>

SuperCollider (2011c) *SinOsc* [online]. [Accessed 15th May 2020]. Available at: <http://doc.sccode.org/Classes/SinOsc.html>

SuperCollider (2011d) *Function* [online]. [Accessed 15th May 2020]. Available at: <http://doc.sccode.org/Classes/Function.html#-!>

SuperCollider (2011e) *Pbind* [online]. [Accessed 15th May 2020]. Available at: <http://doc.sccode.org/Classes/Pbind.html>

SuperCollider (2018a) *SuperCollider* [online]. [Accessed 10th May 2020]. Available at: <https://SuperCollider.github.io/>

SuperCollider (2018b) *Window* [online]. [Accessed 11th May 2020]. Available at: <http://depts.washington.edu/dxscdoc/Help/Classes/Window.html>

Teach Computer Science (2019) *High and Low Level Languages* [online]. [Accessed 10th May 2020]. Available at: <https://teachcomputerscience.com/high-low-level-languages/>

TypesnUses (2019) *What is a Programming Language and Different Types* [online]. [Accessed 11th May 2020]. Available at: <https://www.typesnuses.com/types-of-programming-languages-with-differences/>

Waves Audio (2017) *Reverb and Delay Explained – Sound Basics with Stella Episode 4* [online]. [Accessed 11th May 2020]. Available at: <https://www.youtube.com/watch?v=-jPPJEHMepA>

Weintrop, D. and Wilensky, U. (2017) *Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms* [online]. [Accessed 11th May 2020]. Available at: <https://dl.acm.org/doi/10.1145/3089799>

Wirth, M. (2013) *Spaghetti code* [online]. [Accessed 15th May 2020]. Available at: <https://craftofcoding.files.wordpress.com/2013/10/lore_spaghetti.pdf>

Xiong, X. (2013) *How does the brain locate sound sources?* [online]. [Accessed 15th May 2020]. Available at: <http://knowingneurons.com/2013/03/15/how-does-the-brain-locate-sound-sources/>

Yeung, E.S. (1980) *Pattern recognition by audio representation of multivariate analytical data* [online]. [Accessed 16th May 2020]. Available at: <https://pubs.acs.org/doi/pdf/10.1021/ac50057a028>

APPENDECIES

Appendix 1

Another version of the algorithm can be found in the folder called "*Accompanying Docs*" under the title "*User Friendly Version*". This version of the algorithm has an interface that is easier to understand and use than the previously mentioned Pure Data file.

Back To Section

Back To Top

Appendix 2

There is a scientific theory that humans are more adept at recognising patterns, and subsequent anomalies within those patterns, when they are presented aurally rather than visually (Pullum and Rogers, 2009). It is believed that this ability stems from the human capacity to perceive and recognise complex languages (Pullum and Rogers, 2009). However, other academics have theorised that the true origin of this ability is an inadvertent offshoot of a human's ability to localise sound (Yeung, 1980).

Localisation refers to a subconscious process carried out by the brain and the ears, after the emission of a sound. When a source emits a signal, it is sent in all conceivable directions and if a human is within hearing distance, their ears will inevitably detect that signal. As the signal is received by the ears, the brain will make a subconscious comparison between each ear (Xiong, X, 2013). The level of the signal being detected by each individual ear will differ based on the position of the head and how that then positions the ears in relation to the source of the signal (Xiong, X, 2013). Thus, granting the human the ability to determine the location of the signal, with relatively high accuracy, using just their ears (Xiong, X, 2013). It has also been proposed that localisation might be one of the factors that help humans determine differences between pitch (Yeung, 1980), though only some vague research has been found to support this proposition. Regardless of whether that is true or not it is clear that the human ear is capable of detecting, processing, and interpreting complex aural information. However, it must be mentioned that this ability is dependent on the attention level of the human in question (Pullum and Rogers, 2009). If they are not focused or paying attention to the information their ears are receiving, then it is possible for that information to be missed or misinterpreted (e.g. mishearing what someone says) (Pullum and Rogers, 2009).

Back To Section

Back To Top

Appendix 3

In coding, programming languages are often classified as being high-level or low-level depending on their perceived difficulty to navigate (Schules, 2017). A programme, that is relatively easy to understand, and use would be regarded as being a high-level language, whereas something that is very difficult to understand would be considered low-level (Schules, 2017). Machine code (e.g. binary code) would be an example of an extremely low-level programming language as it is not made to be "*human readable*", it is just the most efficient way of communicating data between machines (Schules, 2017).

Text-based code is typically considered to be low-level, especially in comparison to VPL's (Cash, 2018), however, that is not necessarily true as there are some text-based languages, such as Python, that are viewed as being high-level languages because their syntax is very straightforward and easy to use (Birnir and Browning, 2015). VPL's are generally considered to be high-level programming languages because, in place of text, they utilise graphical symbols or visual elements as their primary syntax (Des, 2015), making it easier for users to visually see the flow of data within the code (Des, 2015). However, as with text-based code, not every VPL is actually high-level and some can be quite difficult to navigate, for example, although Pure Data is considered a high-level language, it does have some features that a non-expert programmer may regard as being low-level (Nabi, 2018).

Back To Section

Back To Top

Appendix 4

At this point, in the project, the goal had been to make an algorithm that was able to generate ringtones based on the phone number of the individual ringing. As phone numbers, in the UK, contain a series of eleven unique single-digit numbers, it was assumed that the values would have to multiplied by 100 to make them audible, which is why the 100 value was included.

Back To Section

Back To Top

Appendix 5

VPL's such as Pure Data have been specifically designed to make programming accessible to users with little to no prior experience in the field (Petre, 1995). This does not necessarily make them high-level languages, as they all still possess their own unique syntax that must be learned.

Arguably, the primary appeal of VPL's stems from the skeuomorphic properties they possess (Petre, 1995). Whilst a programming language such as Pure Data, at first glance, may not look like it has any real-world counterparts, its wire-like connectors can be likened to patch-cables. By introducing this relatively small element into a programming language it is possible to influence a user's comprehension of how it all works. This leads into the idea of '*embedded cognition*', which is the idea that human behaviour is shaped and influenced by their surrounding environment (Foglia and Wilson, 2011). The learned behaviours they acquire, from these environments, may then go on to influence the way they react and behave in other subsequent environments (Foglia and Wilson, 2011). The individuals who created Pure Data may have been aware of this to some extent, and so by using connectors that function in the same way patch-cables do, they allow the user to make an unconscious connection between the two (Petre, 1995). That being said, embedded cognition and skeuomorphism are heavily dependent upon a user's prior experience (Foglia and Wilson, 2011), both real world and virtual. Whilst using virtual patch cables may be extremely useful to someone who understands what they are, if a user who has no concept of patch cables were to use Pure Data, they may find it just as difficult as using a text-based language like SuperCollider. Therefore, it could be argued that the terms high-level and low-level are entirely relative to the person using the programming language.

Back To Section

Back To Top

Appendix 6

It must be understood that the word "*easier*" is being used relatively to an algorithm of equal size written in SuperCollider or another text-based programming language. VPL's, especially ones with an interface that is similar to Pure data's, are prone to something called "*spaghetti coding*" which, in summary, is when a piece of code becomes very unstructured and messy, regardless of functionality (Jones, 2018). Spaghetti code can occur in any programming language but is highly visible in VPL's like pure data (*see figure 18*), which can make it difficult to follow and understand one's code. However, from the perspective of a non-expert programmer, it is still probably marginally easier to follow and comprehend than spaghetti code written in text (*see figure 19*).

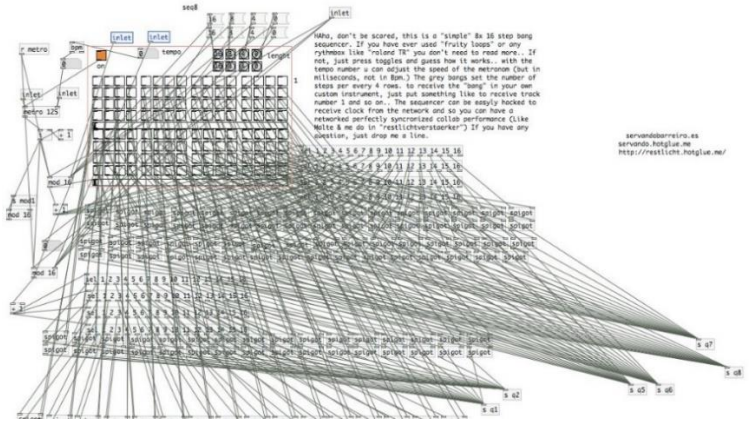

Figure 18: This is an image displaying what "spaghetti code" would look like in a VPL such as Pure Data (CDM, 2012). As can be seen the mass of connections between objects could be considered unreadable, potentially making it extremely difficult for any subsequent users to understand exactly what is going on inside of this patch (CDM, 2012)

```
int spaghetti_code(int year, int date)
{

    int day, month;
    int L;

    if (date < 1 || date > 366 || year < 0)
        return 6;

    if (date <= 31)
        goto JAN;

    L = 1;
    if (year % 400 == 0)
        goto LEAP;
    if (year % 100 == 0)
        goto NOLEAP;
    if (year % 4 == 0)
        goto LEAP;

    NOLEAP: L = 0;
            if (date > 365)
                return 0;
    LEAP:   if (date > (181 + L))
                goto G181;
            if (date > (90 + L))
                goto G90;
            if (date > (59 + L))
                goto G59;
            month = 2;
            day = date - 31;
            goto OUT;
    G59:    month = 3;
            day = date - (59 + L);
            goto OUT;
    G90:    if (date > 120 + L)
                goto G120;
            month = 4;
            day = date - (90 + L);
            goto OUT;
    G120:   if (date > 151 + L)
                goto G151;
            month = 5;
            day = date - (120 + L);
            goto OUT;
    G151:   month = 6;
            day = date - (151 + L);
            goto OUT;
    G181:   if (date > 273 + L)
                goto G273;
            if (date > 243 + L)
                goto G243;
            if (date > 212 + L)
                goto G212;
            month = 7;
            day = date - (181 + L);
            goto OUT;
    G212:   month = 8;
            day = date - (212 + L);
            goto OUT;
    G243:   month = 9;
            day = date - (243 + L);
            goto OUT;
    G273:   if (date > 334 + L)
                goto G334;
            if (date > 304 + L)
                goto G304;
            month = 10;
            day = date - (273 + L);
            goto OUT;
    G304:   month = 11;
            day = date - (304 + L);
            goto OUT;
    G334:   month = 12;
            day = date - (334 + L);
            goto OUT;
    JAN:    month = 1;
            day = date;
            goto OUT;

    OUT:    printf("day=%d/month=%d",day,month);

    return 6;
}
```

Figure 19: This is an image displaying what "spaghetti code" would potentially look like in a text-based programming language such as SuperCollider (Wirth, 2013). Whilst this image probably is not as severe as figure 18, what makes this (and text-based code in general) more difficult to understand is the syntax. In VPL's you can often just follow the connections between objects to vaguely understand their function, but this is not possible in text-based languages.

Back To Section

Back To Top

Appendix 7

In *figure 10* it can be seen that there is a number box between these two objects. However, this number box does not have a specific role in how this section of the patch functions, it is merely there to allow the user to see if the [random] object is outputting the correct values.

Back To Section

Back To Top

Appendix 8

For example. If the [random 3] object outputs a value of 1, a bang will be sent out of the first outlet on the [select 0 1 2] object. This bang will then trigger the {1} message box attached to the outlet to switch on the [toggle] object of the first oscillator present within the sub-patch (the sine wave [osc~] oscillator). As the bang triggers the {1} message box it simultaneously sends a bang to two separate message boxes that both read {0} and are each attached to the [toggle] object of the other two oscillators (*see figure 10*). When the {0} message boxes receive a bang, they will output a value of 0 thus switching off those [toggle] boxes or causing them to remain off.

Back To Section

Back To Top

Appendix 9

The $1 is a variable that acts as a placeholder for any expected values that will be entering the messages box (FLOSS Manuals, 2008c). These expected messages are being output from the [trigger] object (attack time) and the secondary [float] object (decay time).

Back To Section

Back To Top