

DRUMBOOTH

An audio separation application for percussion.

Sean Breen, R00070693

16th Nov 2015

DCOM4 Final Year Project



<u>INTRODUCTION</u>	2
<u>AUDIO</u>	3
<u>THE SPECTROGRAM</u>	5
<u>REAL-TIME AUDIO SIGNAL PROCESSING</u>	6
<u>THE AUDIO CALLBACK</u>	7
<u>MIXING SOUNDS TOGETHER</u>	8
<u>DESCRIPTION OF PROGRAM</u>	11
<u>HARMONIC/PERCUSSIVE SEPARATION</u>	11
<u>METRONOME</u>	13
<u>RUDIMENT LIBRARY</u>	16
<u>TECHNICAL DESCRIPTION</u>	17
<u>FUNCTIONAL REQUIREMENTS</u>	17
<u>NON-FUNCTIONAL REQUIREMENTS</u>	17
<u>CONSTRAINTS</u>	18
<u>PROTOTYPE</u>	19
<u>MEDIAN FILTER</u>	19
<u>PROGRAM EXECUTION</u>	21
<u>DEVELOPMENT</u>	27
<u>JUCE</u>	
<u>FFTReal</u>	
<u>Eigen</u>	
<u>BIBLIOGRAPHY</u>	30

INTRODUCTION

DrumBooth is an application that enables the user to separate an audio file into its harmonic and percussive components.

It is primarily for drummers who wish to extract the percussion from an audio track, either to listen to the drums in isolation, or to make a backing track without any drums. The method of separating the audio is derived from Derry Fitzgerald's method described in [1], which uses a median filter technique on the audio data to create two spectrograms, one with just percussive elements and one with the harmonic elements, which are then converted back to audio.

In addition to the harmonic/percussive separation, the application also features an index of the Percussive Arts Society's 40 drum rudiments, which are the basic tools that a drummer uses to practice rhythmic patterns. Each rudiment is a pattern of rhythm which should be practiced often. A metronome can be used to keep in time while practicing. DrumBooth features a standard metronome for use when practicing.

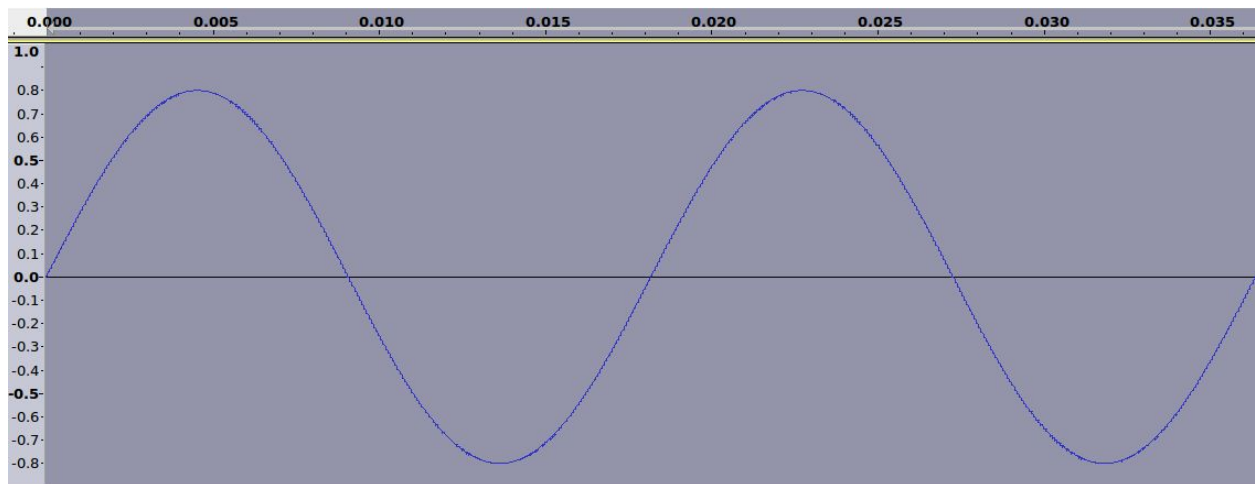
DrumBooth was developed by Sean Breen as his DCOM4 Final Year Project 2015 submission. Audio separation algorithm was developed by Derry Fitzgerald.

Source code for this project is available at:

<https://github.com/mangledjambon/drumbooth.git>

AUDIO

A sound is a pressure wave that causes a periodic disturbance in the air, or other medium. The human ear can hear vibrations that are in the 20Hz to 20kHz range. Hertz describes the frequency of a sound wave, i.e how many times the wave cycles in a second. For example, if a sound wave cycles less than 20 times in a second (or more than 20,000 times), we definitely won't be able to hear it because it is outside our hearing range.

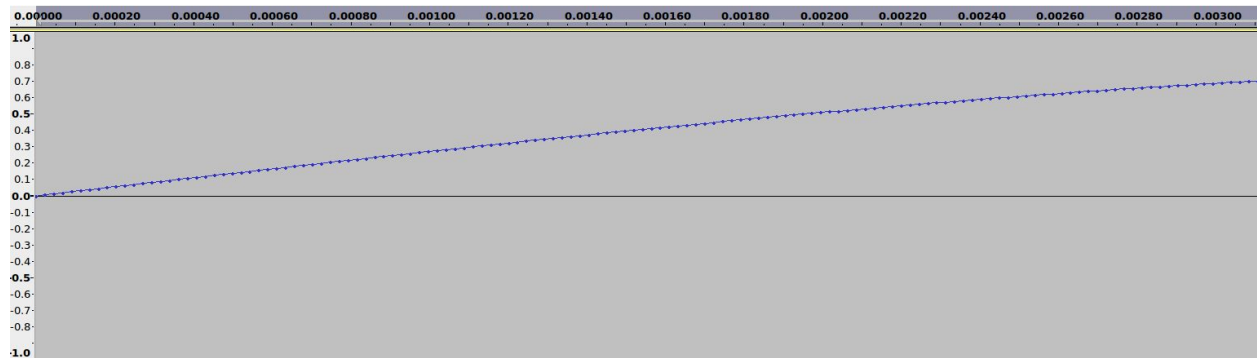


Here is a wave cycling 55 times a second, or at 55Hz. You can see the wave cycles twice, that is, it travels up from 0 to 0.8 amplitude, then down to 0, then from 0 to -0.8 amplitude and back to 0. That is one cycle. This happens twice in the image above and exactly 55 times in one second. This sound is an A note in the second octave, or A1 (A0 is first note). This is just inside the human hearing range but still may be inaudible without the appropriate speakers to accommodate such low frequencies.

A continuous audio signal is unreadable by a computer, so it must be converted to a numerical, or digital, format in order to be represented in your software, or as a file. To do this, the audio wave is sampled tens of thousands of times a second using an Analog-to-Digital converter, or ADC. The amount of samples to be taken every second is referred to as the audio's 'sample rate'. A sample is just a record of the signal's amplitude at a point in time. If we zoom-in on the waveform above, you can see

individual samples as they appear across the waveform as small ticks.

CDs use a sampling rate of 44.1kHz, or 44,100 samples every second.



So, 44,100 times per second, the waveform is sampled to get its amplitude. Each blue tick along the waveform is a sample. You can also see that all this sampling occurs in just 0.003 of a second. A 3-minute piece of music on CD will contain million of samples, each containing a value for the amplitude of the signal at that point in time. In order to accurately replicate the sound in digital format, the sample rate must be over over two times faster than the highest frequency our ears can hear to avoid undesired artifacts like aliasing. So the audio data exists as a long list of floating-point numbers which measure the amplitude of the signal at each sample time. These numbers can be manipulated to make changes to the audio signal. this process is called Digital Signal Processing, or DSP. Examples of DSP uses include:

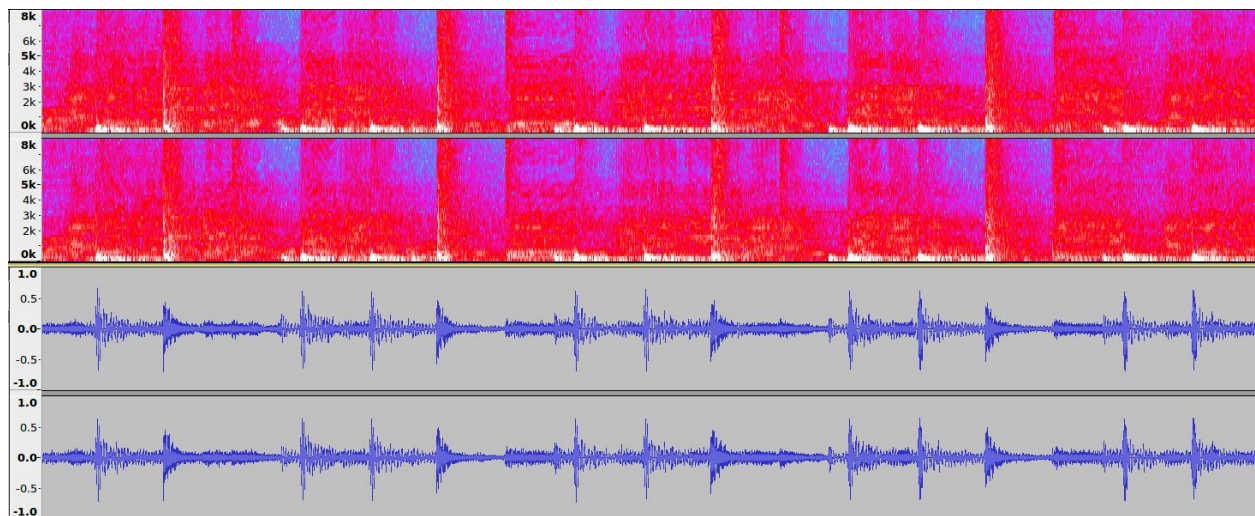
- Removing background noise from an audio signal (Communications)
- Adding echo (reverb.)
- Delaying a signal
- Adding distortion to a signal (guitar)
- many, many more.

Changing a signal to reduce or enhance aspects of the signal is know as 'filtering'.

THE SPECTROGRAM

A spectrogram is an image which provides a graphical representation of the spectrum of frequencies present in an audio signal as they vary over time.

The vertical axis of the spectrogram shows a set of frequency 'bins' which contain information for one frequency, in the image below these are running from 0 Hz to ~8000 Hz (8 kHz). The horizontal axis represents time. The different colours represent the various 'intensities' of the frequencies in the audio signal.

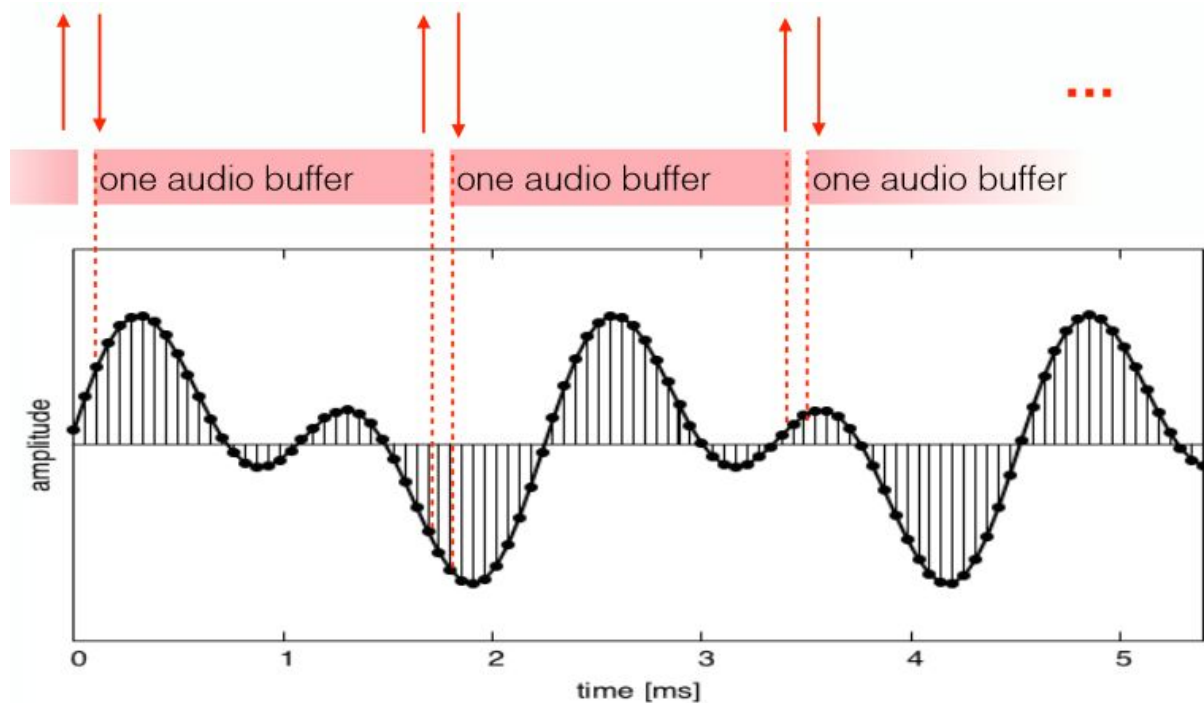


This excerpt from A Tribe Called Quest's 'Check the Rhime' shows the track's spectrogram on top, and its waveform underneath. If you look closely, you can see the scale of frequencies to the left of the spectrogram, and the amplitude range to the left of the waveform. Both of these are graphed over time to show the characteristics of the signal. A spectrogram is a signal's representation in the frequency domain and a waveform is its representation in the time domain.

A spectrogram can tell us many things about an audio signal that a waveform representation cannot. For this reason, audio is usually observed analytically in the frequency domain.

REAL-TIME AUDIO SIGNAL PROCESSING

In your computer, you have an audio card which is a piece of hardware that handles input and output of audio signals. Playing a song on your computer requires a constant stream of data so that the audio does not stop or cut-out for the duration of the audio signal, unless it is paused or stopped by the listener. Your computer could load the whole audio file into memory, but this would be terribly inefficient, so rather the audio card will request a small part of the data at a time. This is known as a 'buffer'. The audio card will request as many of these buffers as it takes until it reaches the end of the file.



Requesting the next buffer is known as a 'callback', and this callback could be called 1,000 times a second. It needs to execute every time without fail, because if the next audio buffer is not ready by the time the callback happens, the buffer will be filled with unwanted, garbage audio or it will be completely silent. Neither of these things should ever happen in an audio application.

Audio programming is 'hard real-time' programming. The audio callback waits for nothing, so you must ensure that if you make any changes to the audio signal, that those operations will always be completed by the time the callback happens.

THE AUDIO CALLBACK

The audio callback is a method that gets called by the audio thread in your application every time the next audio buffer is needed by the sounds card. Buffer size can be changed, but is typically between 32 and 1024 samples long.

Different buffer sizes have consequences on the performance of your system, usually in the form of latency or audio artifacts like crackling, so you want to ensure that your buffer size is big or small enough to meet your speed requirements, or else drop-outs will occur.

```
void audioCallback (float** channelData,  
                    int numChannels,  
                    int numSamples)  
{  
    for (int channel = 0; channel < numChannels; ++channel)  
        for (int sample = 0; sample < numSamples; ++sample)  
            channelData[channel][sample] = ...;  
}
```



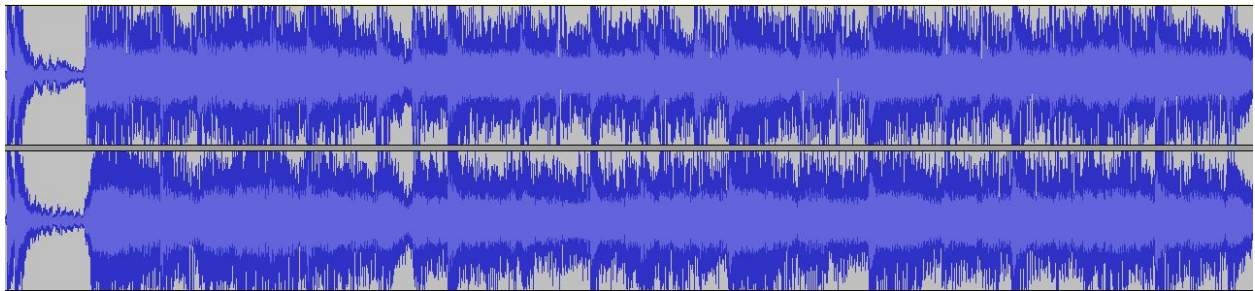
...complex DSP calculations here...

So, to avoid drop-outs, a few things need to be guaranteed by your code :

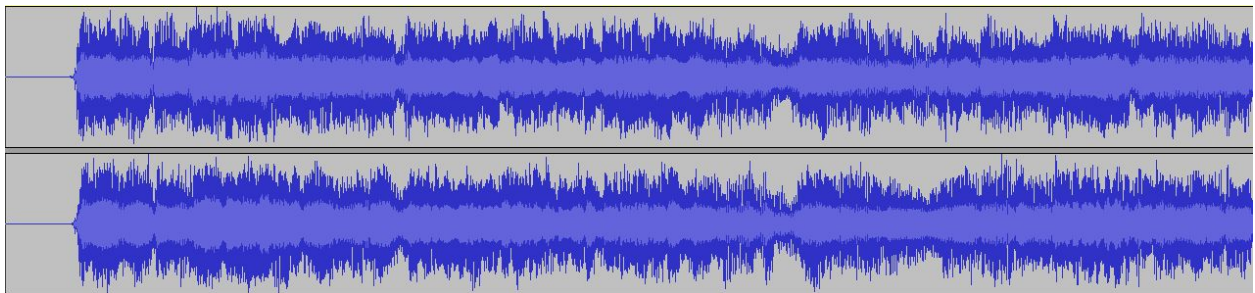
- This method will return before the next audio callback happens.
- The output buffer will contain valid audio data.
- There will be no error occur or exception thrown while executing the method.

MIXING SOUNDS TOGETHER

When recording a performance of a piece of music, each of the instruments is recorded onto a separate track and later, when the track is ready all the tracks are mixed together to give the final product. Once the final product goes out to the consumer, the mixed version is just two waveforms, one for left channel and one for right channel, containing the sounds of all the instruments mixed together. Below is an excerpt from ‘Carry on My Wayward Son’ by the Kansas as it appears on their 1976 album ‘Leftoverture’, with all the instruments playing:



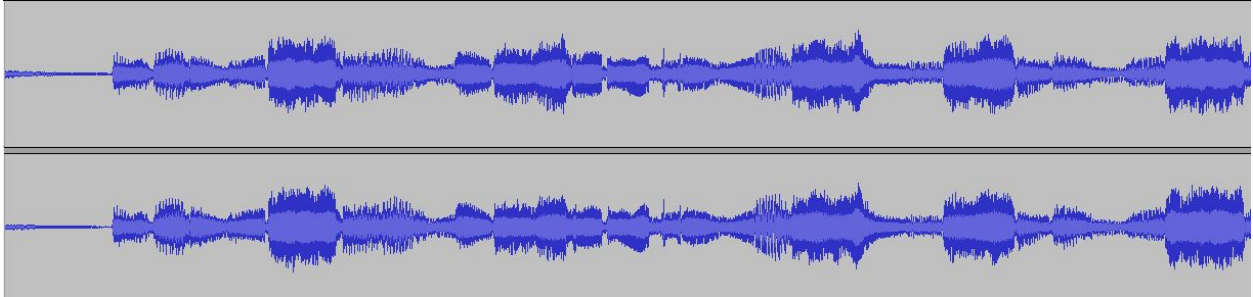
This is the product of all the individual tracks being mixed together to form one stereo track. For instance, let's see what the guitar looks like on it's own.



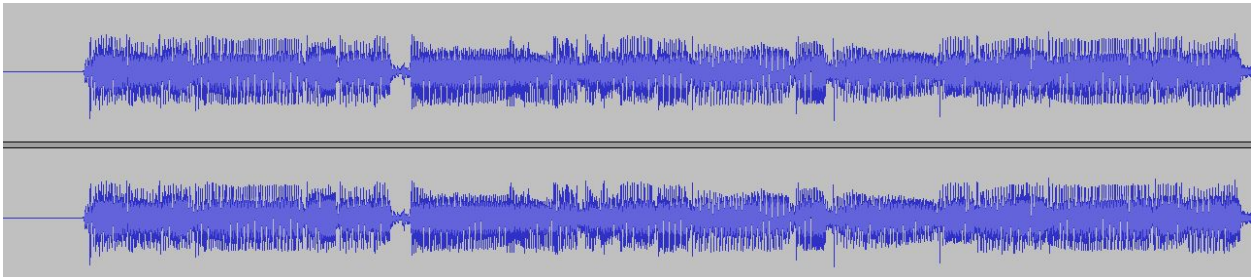
The guitar plays a big part in the overall sound of the band and is loud on the record, so the guitar's waveform resembles that of the original piece but if you listen to it, the difference is huge. This track may itself be the product of several overdubbed guitar

tracks.

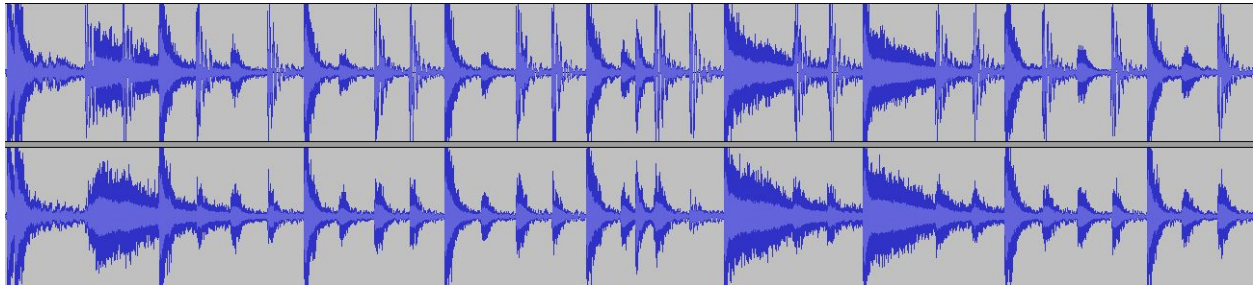
The drums, organ, vocals and bass guitar are all absent from this track. There are less spikes in the waveform when the guitar is isolated, so another instrument must be causing these spikes in the waveform. Let's look at the other tracks now.



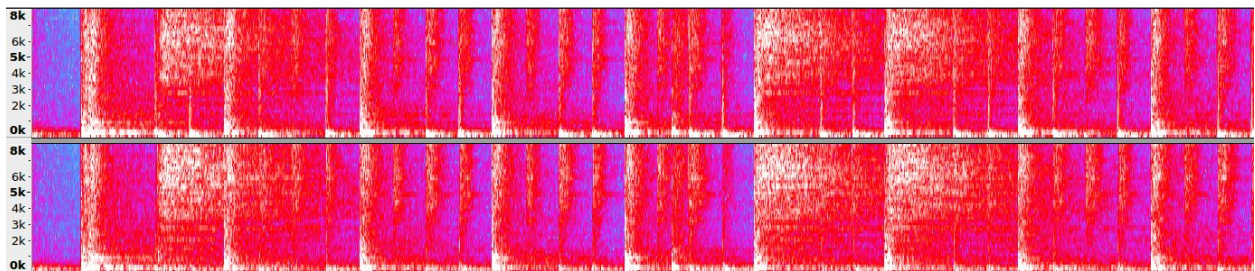
Noticeably less happening with the organ track on this song, compared to the guitar. The organ is noticeably quieter. No large spikes in amplitude exists.



Some small spikes exist in the bass guitar track's waveform. When a bass guitar string is plucked with a pick it creates a moderate amount of 'attack' which will sound quite percussive to human ears. It also may appear as a spike in a signal.



Finally, the drums. Here, you can see the drums leave many large spikes in the signal. This is because percussive sounds are very noisy, as in, they do not have a set frequency, but rather are an assortment of loud, random frequencies at high intensity. We can illustrate this by making a spectrogram of the drum track.



You can see the spikes in amplitude correspond to the large, broadband noise that is present in the spectrogram.

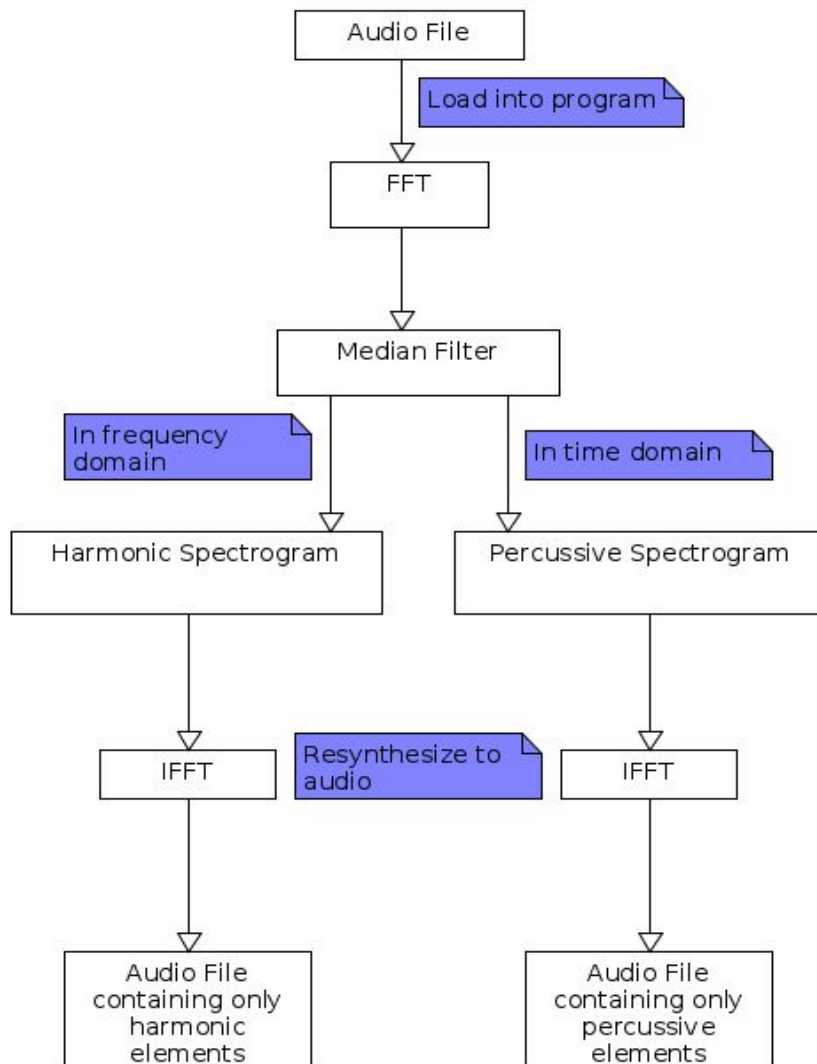
When compared with the other instruments, it is very obvious that the drums' sound characteristics are quite different from that of the other instruments. We can use the differences between these types of sounds to isolate the drums and extract them from the original waveform.

DESCRIPTION OF PROGRAM

HARMONIC/PERCUSSIVE SEPARATION

The application will take an audio file as input from the disk. This audio file can be in wave (.wav), MPEG-Layer 3 (.mp3) or Free Lossless Audio Codec (.flac) formats.

It uses digital signal processing techniques to separate the harmonic and percussive elements of the audio file. A median-filtering technique is used, based on Derry Fitzgerald's algorithm explained in [\[1\]](#). This will be implemented as both a real-time process and as a standalone process to process a whole file at once and spit out the resulting audio in their own, separate audio files.



In the non-realtime mode, the whole file will be processed by an FFT, sent to the median filter to separate the harmonics and percussion, then an inverse FFT will resynthesize the audio before it gets written to disk. Two files will be present after resynthesis, one containing harmonic characteristics and one containing the percussive. These files will be written to disk in .wav format, so they will be playable by all standard audio players.

For the real-time mode, the processing must take place while the file is being read from the disk. To perform this task, a small portion of the file must be read, processed and resynthesized before the audio card requests the next block of audio. This is very tricky and timing is a huge issue as audio 'drop-outs' will be heard if the next block of audio from the file is not ready by the time it's needed. A block of audio could be anywhere from 128 samples to 4096 samples depending on the audio driver being used.

METRONOME



DrumBooth has a built-in metronome that is useable anywhere in the application.

Music is counted in bars, and each bar, or ‘measure’, is composed of a number of beats. The amount of beats to be counted in each bar is one half of the music’s ‘time signature’. The other half is the kind of note to count. A time signature is notated by putting the amount of beats in the bar above the line, and the kind note is put below the line, much like a fraction. Here is a two bars of a piece in a 2/4 time signature.



That means that for every bar, there is two beats (above the line), each beat being a quarter-note (below the line). Note values in music are divided into whole notes (semibreve), half-notes (minim), quarter-notes (crotchet), eighth-notes (quaver), sixteenth-notes (semi-quaver), etc... These can be divided into smaller subdivisions if needed.

Note Value Chart with Rests			
British		American	Rests
semibreve		whole-note	 (also used for a one-bar rest whatever the metre)
minim		half-note	
crotchet		quarter-note	
quaver		eighth-note	
semiquaver		sixteenth-note	
demisemiquaver		thirty-second-note	
<p><i>ties</i> are used to join notes together; <i>dots</i> increase a note by half its value</p>			

Illustration of note values in music from

<http://www.paulchengviolin.com/paul%20note%20values.jpg>.

So you give the length of the bar, by giving it the amount of beats and the type of note, and the metronome will count this time signature indefinitely, until you tell it to stop. This is extremely important for musicians, where you need to keep counting time always. Even when a musician is not playing for a part of a piece, the time is still there and needs to be counted, so as not to come in too early or late for the next time that they are playing.

DrumBooth contains a metronome which can be programmed to count a selection of typical time signatures, such as

- 2 / 4
- 3 / 4
- 4 / 4
- 5 / 4
- 7 / 4
- 6 / 8
- 7 / 8

at a tempo between 50 beats-per-minute and 200 beats-per-minute.

In the future, I would like to add a 'drum-machine', which is a programmable device which uses sampled drum sounds to emulate the sound of a drum kit. This can be used to create beats or rhythms spontaneously inside the application and save them for later use.

RUDIMENT LIBRARY

An index of the 40 drum rudiments viewable in the application, including pictures for each rudiments and suggestions for applying the rudiment to music.



The 'Paradiddle' drum rudiment. Rudiments often have names which are mnemonics. Can be split to 'par-a-did-dle', each syllable being a stroke.

Rudiments are practiced to develop an understanding of the foundations of rhythm itself. They build the muscle-memory and rhythmic knowledge that a drummer needs to play efficiently. So as part of the DrumBooth application, I decided to include a section where these rudiments could be browsed and observed by the user. This can be used in conjunction with the metronome to provide an effective way to practice the rudiments.

TECHNICAL DESCRIPTION

FUNCTIONAL REQUIREMENTS

The application must

- effectively separate harmonic and percussive components from a monophonic or stereophonic audio signal.
- perform this extraction in real-time with no audio drop-outs.
- be compatible with major audio file formats (wav, mp3 and flac).
- have a metronome which can be started/stopped at any time when using the application by clicking a start/stop button.
- have a browsable list of images representing the 40 drum rudiments.

NON-FUNCTIONAL REQUIREMENTS

The application must

- run on Windows and Linux operating systems.
- have a Graphical User Interface.
- have a progress bar for separation progress so the user knows how far along it is (could be quite long for large files).

CONSTRAINTS

The main constraints of this project are

→ Time

- ◆ the project research phase and implementation phase are both twelve weeks long.
- ◆ Research phase due date is December 3rd 2015.
- ◆ Implementation phase due date is next summer.

→ Personnel

- ◆ I am the sole developer of this project.

→ Skills

- ◆ My knowledge of C++ is not very broad.
- ◆ My knowledge of DSP techniques is also not broad, but both my C++ and DSP knowledge is growing day-by-day.

PROTOTYPE

MEDIAN FILTER

Median filtering is a technique used in image processing. The median filter has a few uses, such as removing noise from an image, detecting edges, or sharpening contrast. It is widely used in image processing.

The filter works by moving through the image and replacing a pixel with the median value of neighbouring pixels. The amount of neighbouring pixels to use is called the 'window'. for every pixel, the windowed pixels (neighbours + original pixel) are sorted and the original pixel is given the value of the middle pixel.

```

1  /*
2     Pseudo-code for median filtering
3     1-Dimensional signal x[]
4  */
5
6  int x[4]; // input signal to be filtered
7  int y[4]; // output signal
8
9  x = {2, 80, 6, 3};
10
11 // extend the first and last values beyond the edges
12 y[0] = median(2, 2, 80) = 2
13
14 y[1] = median(2, 80, 6) = median(2, 6, 80) = 6
15
16 y[2] = median(80, 6, 3) = median(3, 6, 80) = 6
17
18 y[3] = median(6, 3, 3) = median(3, 3, 6) = 3
19
20 y = {2, 6, 6, 3}

```

This median filtering technique can be used on a spectrogram data to remove certain 'noise' from the audio signal.

To extract the percussion from the spectrogram, you loop through each column of the spectrogram, or 'frame'. In one frame of the spectrogram, there will be frequency data for every frequency bin in the signal.

As mentioned earlier, a percussive sound appears as a broad-band sound composed of seemingly random frequencies, and harmonic sounds appear as horizontal lines at a given frequency bin. With this knowledge, we can use the median filter to remove any unwanted 'noise' from the frame. In this case, harmonic components will appear as spikes across the frequency bins, so a median filter can be used to smooth out these spikes and leave just the broad-band, random frequencies i.e the drums.

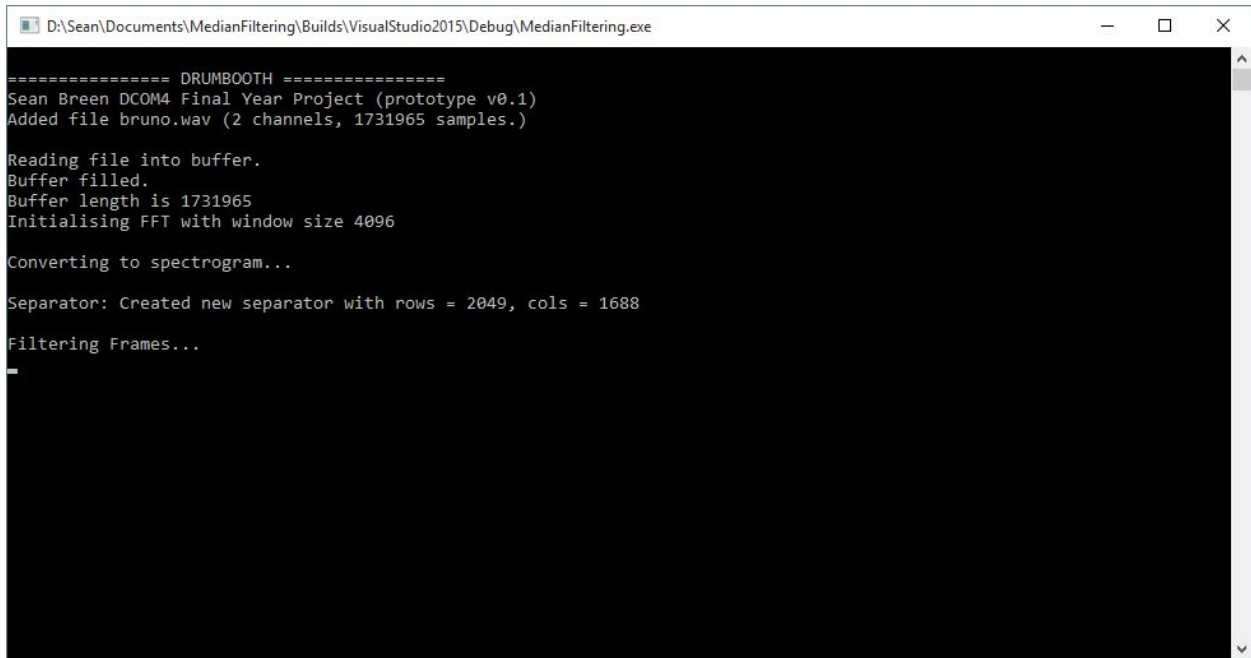
This process can also be used across each frequency bin, to smooth out any noise vertical lines in the spectrogram and leave the harmonic components.

PROGRAM EXECUTION

The program begins by taking an audio file from the disk. This can be in any the following formats:

- Wave audio file (.wav)
- MPEG-Layer 3 (.mp3)
- Free Lossless Audio Codec (.flac)

For this example, I'm using a wave audio format file.

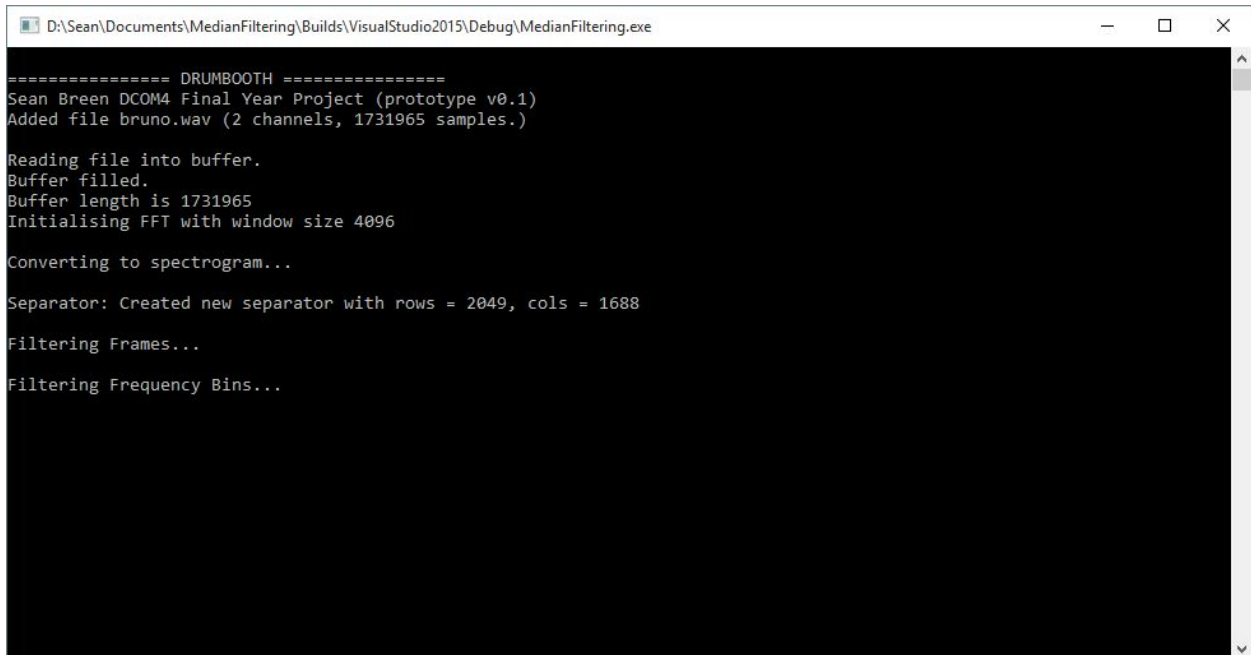


```
D:\Sean\Documents\MedianFiltering\Builds\VisualStudio2015\Debug\MedianFiltering.exe

===== DRUMBOOTH =====
Sean Breen DCOM4 Final Year Project (prototype v0.1)
Added file bruno.wav (2 channels, 1731965 samples.)

Reading file into buffer.
Buffer filled.
Buffer length is 1731965
Initialising FFT with window size 4096
Converting to spectrogram...
Separator: Created new separator with rows = 2049, cols = 1688
Filtering Frames...
-
```

It reads the whole file into a sample buffer and performs an FFT on the audio data to convert it to two spectrograms, one for the left channel and one for the right channel, since this is a stereo file. It then begins filtering the time frames to get the percussive elements.



```
==== DRUMBOOTH =====
Sean Breen DCOM4 Final Year Project (prototype v0.1)
Added file bruno.wav (2 channels, 1731965 samples.)

Reading file into buffer.
Buffer filled.
Buffer length is 1731965
Initialising FFT with window size 4096

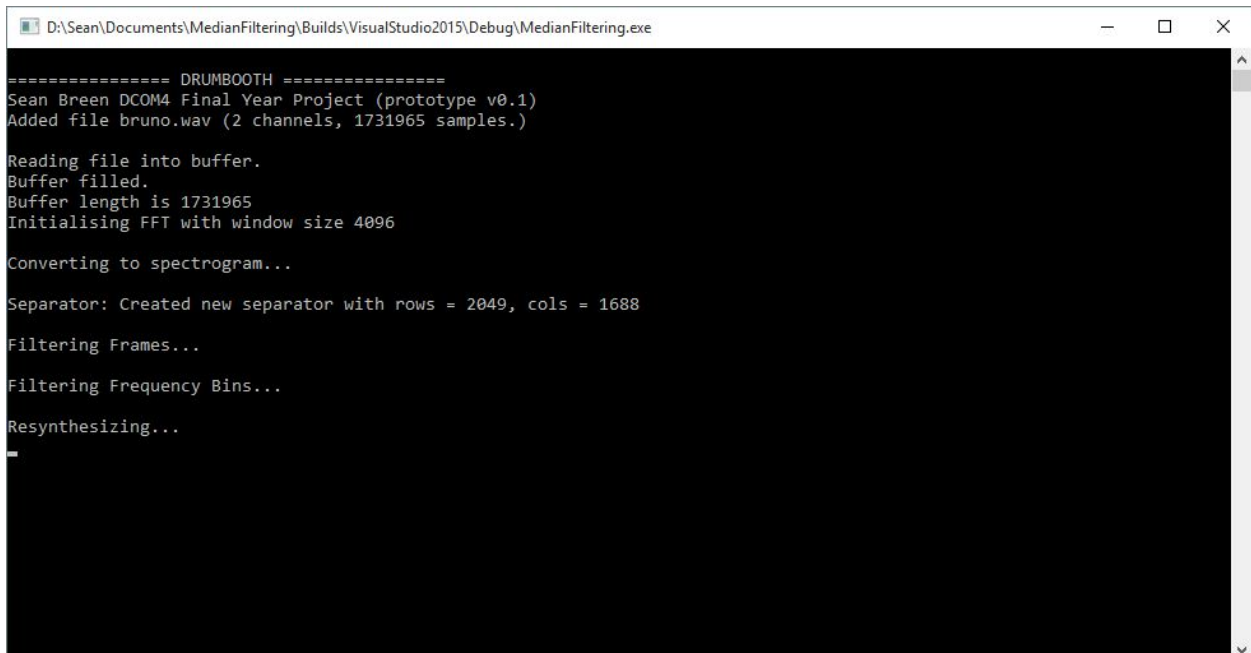
Converting to spectrogram...

Separator: Created new separator with rows = 2049, cols = 1688

Filtering Frames...

Filtering Frequency Bins...
```

When filtering of the time frames is complete, it begins filtering the frequency bins to get the harmonic components.



```
==== DRUMBOOTH =====
Sean Breen DCOM4 Final Year Project (prototype v0.1)
Added file bruno.wav (2 channels, 1731965 samples.)

Reading file into buffer.
Buffer filled.
Buffer length is 1731965
Initialising FFT with window size 4096

Converting to spectrogram...

Separator: Created new separator with rows = 2049, cols = 1688

Filtering Frames...

Filtering Frequency Bins...

Resynthesizing...
-
```


The filtered spectrogram data is then resynthesized by applying a soft mask to the original audio data. This soft mask is in the form:

$$P = \text{Original Spectrogram} * Pest^2 / (Pest^2 + Hest^2 + eps)$$

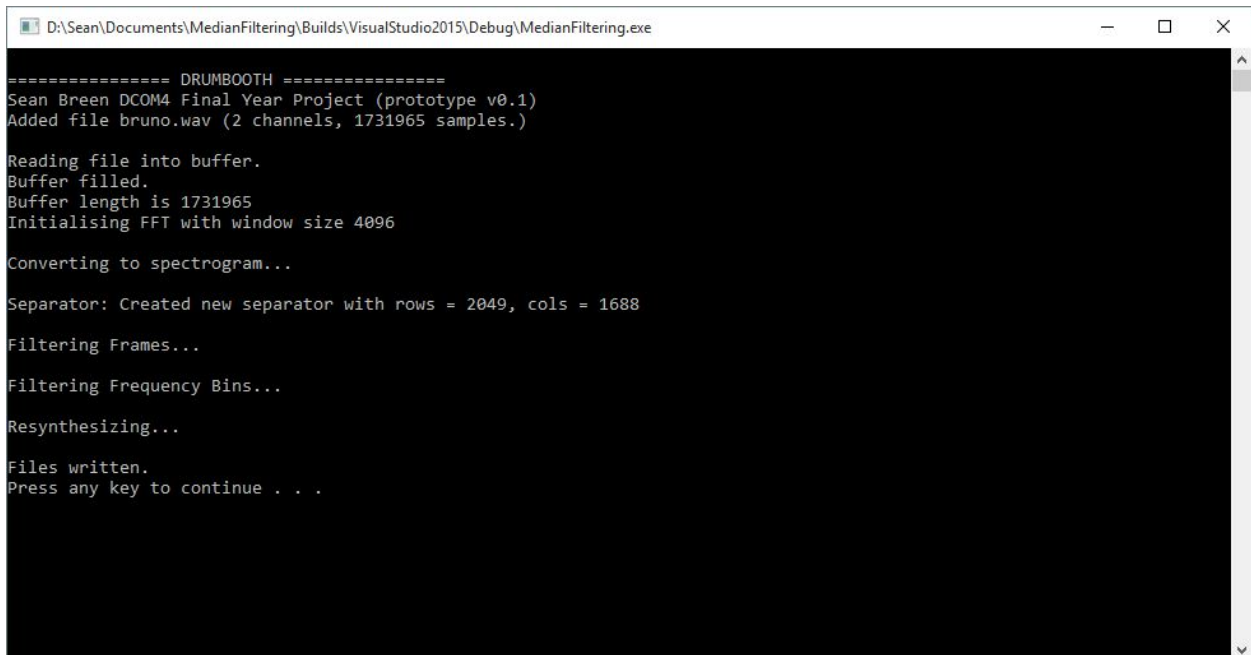
$$H = \text{Original Spectrogram} * Hest^2 / (Pest^2 + Hest^2 + eps)$$

where

Pest = filtered percussive spectrogram

Hest = filtered harmonic spectrogram

eps = very small constant (0.00001f)




```

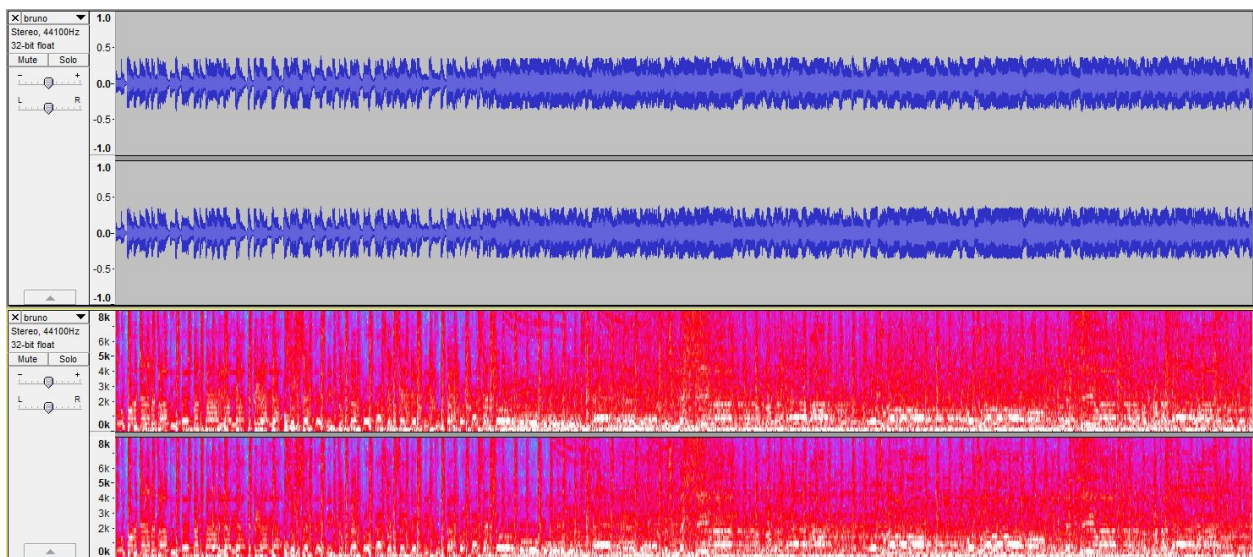
D:\Sean\Documents\MedianFiltering\Builds\VisualStudio2015\Debug\MedianFiltering.exe
==== DRUMBOOTH =====
Sean Breen DCOM4 Final Year Project (prototype v0.1)
Added file bruno.wav (2 channels, 1731965 samples.)
Reading file into buffer.
Buffer filled.
Buffer length is 1731965
Initialising FFT with window size 4096
Converting to spectrogram...
Separator: Created new separator with rows = 2049, cols = 1688
Filtering Frames...
Filtering Frequency Bins...
Resynthesizing...
Files written.
Press any key to continue . . .

```

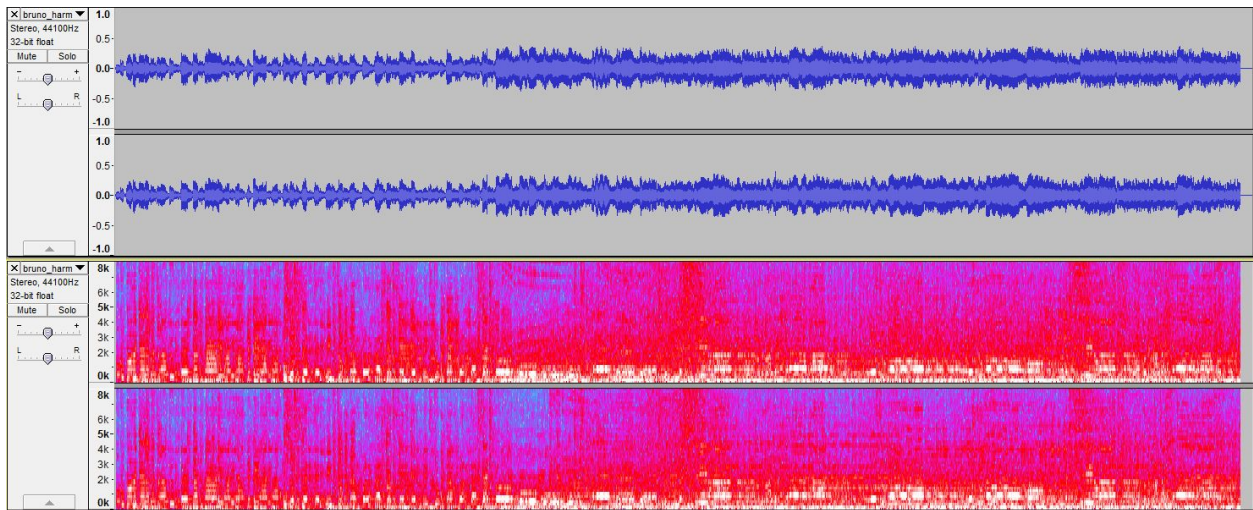
Once resynthesis is complete, the program outputs two audio files, one for percussive elements and one for harmonic elements. These files are placed in the folder where the application was executed, but the original audio file can be located anywhere on the disk when it is loaded into the application.

 bruno_harm.wav	30/11/2015 09:51	VLC media file (.w...	6,766 KB
 bruno_perc.wav	30/11/2015 09:51	VLC media file (.w...	6,766 KB

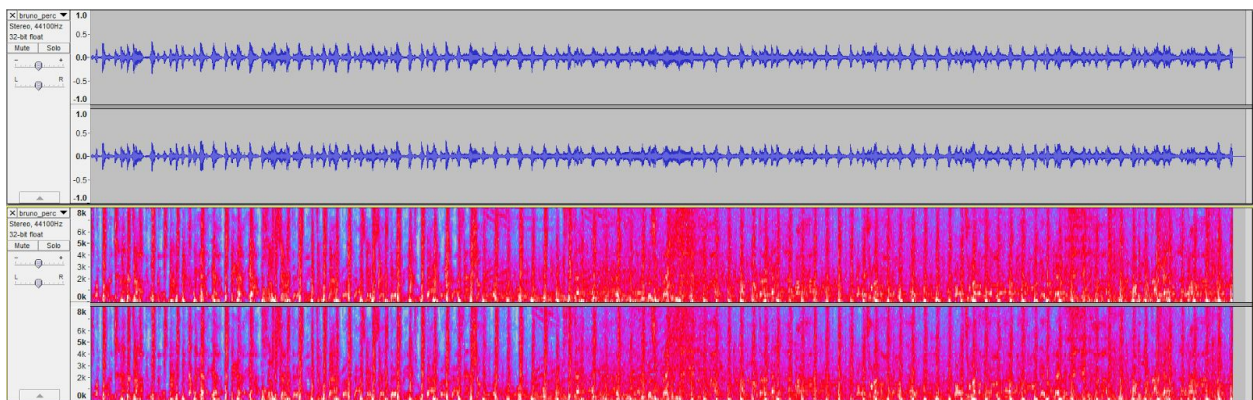
Here the two files are written to the disk in .wav format. Now let's compare the resulting audio data with that of the original file, bruno.wav, which is an excerpt from Bruno Mars' 2012 single, Locked Out of Heaven.



Here is the waveform and spectrogram for the original audio file. The first 15 seconds contain a very prominent snare drum and for the rest of the track, the snare is less used and more bass drum is present. There is guitar, bass guitar, keyboards, vocals and some other percussion present in the track.



Here is the harmonic spectrogram with the percussive components reduced. The vertical lines in the spectrogram are still present, but the harmonics are enhanced and the drums greatly reduced. Some artifacts can be heard and evidence of cymbals and some drums are still present, but percussion is fairly suppressed.



Here is the percussion-enhanced spectrogram for the track. The vertical lines are much more prominent in this spectrogram and while some vocals and the percussive sounds from the guitar are still present, they are quite suppressed. The waveform is also full of small spikes, showing increases in amplitude consistent with the sound of the drums.

My conclusion on this implementation of the algorithm is that additional filtering will

be needed to achieve better separation. This could be in the form of several band-pass filters to isolate individual frequency ranges (such as snare drum/bass drum) or more iterations could be performed when filtering the spectrogram.

DEVELOPMENT

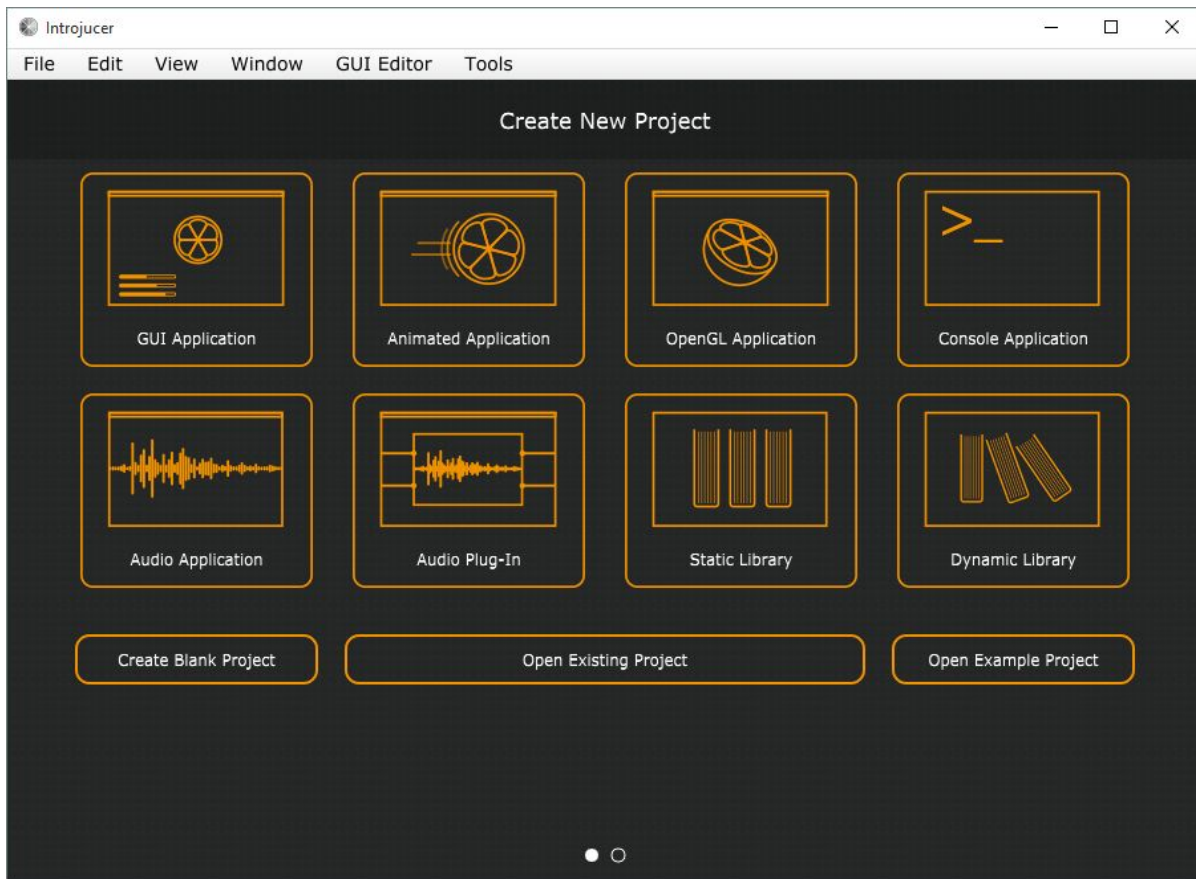
JUCE

I'm using the JUCE framework (<http://www.juce.com>) to develop the application as it is quickly becoming a standard for developing audio applications in C++.

JUCE (Jules' Utility Class Extensions) is an all-encompassing framework for C++, developed over a decade by Julian Storer, which provides excellent support for multimedia programming, such as audio and video. It also provides many wrapper classes for the C++ standard library's data-types such as String, float, etc... I used this to develop the prototype version of the application to demonstrate the use of the median filtering algorithm for percussion extraction and I will use it for the GUI version which I will develop in Semester 2.

As my Integrated Development Environment, I'm using Microsoft Visual Studio 2015 and the Introjucer, JUCE's own project management software. The Introjucer handles managing the locations of the JUCE library files, and you can generate boilerplate applications as a starting point.

The Introjucer will become the Projucer soon, with the release of JUCE 4.0 Grapefruit. The Projucer offers all the functionality of the Introjucer and also adds the ability to edit JUCE Components without having to re-build the whole project. I'm hoping to have a chance to use the Projucer in the second phase of this project, next year.



JUCE has support for developing almost any type of application. The reason it is so popular for Multimedia application development is its large set of classes for handling multimedia objects like audio files and video. Some of the Audio-specific JUCE classes that I need for my application are :

- **AudioFormatManager** - manages support for different audio formats.
- **AudioFormatReader** - manages reading of audio data from a file.
- **AudioFormatWriter** - handles writing of audio data to a file.
- **AudioSampleBuffer** - A multi-channel buffer of 32-bit floating point audio samples.
 - I use this class to hold the audio data that I from the file on disk, and again when writing the data out as a new file.

FFTReal

I make use of a free library for performing Fast Fourier Transforms called FFTReal, developed by Laurent de Soras (<http://ldesoras.free.fr/prod.html>). The library is used to perform FFTs on vectors of real numbers. I use this when transforming the time domain audio data to frequency domain data by using a forward transform, and again when I wish to transform the frequency data back to time-domain by using an inverse transform.

Eigen

Eigen (<http://eigen.tuxfamily.org/>) is a C++ template library for linear algebra. I use it's Matrix classes to store the complex spectrogram data both before and after the filtering operation. The classes I use are:

- MatrixXf - this is a dynamically-sized matrix for real, floating-point values. I use this to store the real matrix values for the resynthesis.
- MatrixXcf - this is a dynamically-sized matrix of complex numbers with real and imaginary parts. Spectrogram data is contained in this matrix of `std::complex<float>` types.

BIBLIOGRAPHY

1. Harmonic/Percussive Separation using Median Filtering, D. Fitzgerald
https://www.researchgate.net/publication/254583990_HarmonicPercussive_Separation_using_Median_Filtering
2. Spectrally Matched Click Synthesis (Metronome)
http://dafx09.como.polimi.it/proceedings/papers/paper_56.pdf
3. Improving Time-Scale Modification of Music Signals Using Harmonic-Percussive Separation. Jonathan Driedger, Meinard Müller and Sebastian Ewert -
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6678724&tag=1>
4. Percussive Beat tracking using real-time median filtering.
Andrew Robertson, Adam Stark, and Matthew E. P. Davies
<http://telecom.inescporto.pt/~mdavies/pdfs/RobertsonEtAl13-mml.pdf>
5. Percussive Arts Society's 40 International Drum Rudiments
<http://www.pas.org/resources/education/Rudiments1/RudimentsOnline.aspx>