Abstract

This project portfolio details the progress made in the development of waveform decomposition as a technique to identify timbre. It was produced within the context of the PXT101 micro-project module and as such is a continuation of the work previously initiated in past years. A database of high quality single-note audio files was sourced, and an automatic kernel processing application was produced to automate the conversion of single-note audio files into kernel pseudo-wavelets suitable for cross correlation. A cross-correlating application was also produced that allows the cross-correlation of produced groups of single-instrument kernels with any audio track, generating a spectrogram of response values over time at different instrument frequencies.

Project Portfolio (2017-10-05 - 2017-12-14)

Pseudo-Wavelet Audio Waveform Decomposition

CARDIFF SCHOOL OF PHYSICS AND ASTRONOMY

MICHAEL NORMAN

Michael Norman, *portfolio author*. **Email:** NormanM5@cardiff.ac.uk
Jack Godfrey, *project partner*. **Email:** GodfreyJ4@cardiff.ac.uk
Dr Richard Lewis, *project supervisor.* **Email:** LewisR54@cardiff.ac.uk

Contents

1: Statement, discussion and evolution of the micro-project design specification 1
1.1: Micro Project Abstract 1
1.2: Initial Proposal 1
1.3 Evolution of the project specifications 3
2: Timeline of Activities 5
3: Software Development 5
3.1: Automatic Kernel Processor Development 5
3.1.1: Reading files to memory 5
3.1.2: Automatic frequency determination (Fourier transforms) 7
3.1.3: Automatic frequency determination (Peak Finding) 8
3.1.4 Normalisation 10
3.1.5 Windowing Function 10
3.1.6 Saving to binary file 10
3.2: Cross Correlation Development 11
3.2.1: Cross Correlation 11
4: Software Operation Instructions 11
4.1: Automatic Kernel Processing Operation 12
4.1.1: File Structure 12
4.1.2: Parameter control 12
4.1.3: Application compilation and execution 13
4.2: Cross correlating Operation 13
4.2.1: File structure 13
4.2.2: Parameter control 14
4.2.3: Compilation and Execution 14
5: Results 14
5.1: Application Research and Development 14
5.2: Kernel generation tests 14
5.3: Cross correlation spectrograms 15
5.4: Kernel Group comparison 17
6: Discussion of Results 18
6.1 Discussion of AKP results 18
6.2 Discussion of Cross Correlation results 18
7: Future work 18
7.1: AKP Future Work 18
7.2: Cross Correlator Future Work 18
8: Evaluation of the Microproject Output 19
9: Possible application of micro project output in the wider scientific context 19

10: References 21
Appendices 22
i: Risk Assessment: 22
ii: Project Updates: 25
iii: Lab Diary: 26
iv: Source code: 27
V: Note to frequency conversion table 28

Preface: A note on nomenclature:

During this report it will be necessary to make reference to a number of different ideas which are not readily distinguishable by everyday nomenclature, thus the following terms will be used throughout the report, referenced initially in full but thereafter simply by the assigned names.

- Source: The raw audio files of single-note instrument recordings which will be used as the input to the kernel processing application, these are all WAVE format files.
- Kernel: A processed source file containing the waveform of an individual pseudo-wavelet
- Kernel group: The collection of all differing-frequency kernels that share the same instrumental origin. Kernels are saved in these collections in custom binary ".kern" files.
- APK: Automatic kernel processor, the developed software to automatically convert source files into kernel files

1: Statement, discussion and evolution of the micro-project design specification

1.1: Micro Project Abstract

The original micro-project abstract produced by project supervisor Richard Lewis was as follows:

Project Title: Pseudo-Wavelet Audio Waveform Decomposition

"This micro-project follows on from a 2016/17 MSc research project, "Pseudo-Wavelet Waveform Decomposition for Automated Musical Genre Recognition", which aimed to assess the practicality of leveraging aspects of timbre, or musical "texture" into musical visualisation, and to investigate potential computational methods to achieve this.

The original research project validated the basic approach, but was far too ambitious in its scope. In this micro-project, you will address two aspects which were suggested as follow-up work to the original research project:

- *The relative magnitude of the cross-correlative response of recordings of different examples of the same instrument;*
- *The relative magnitude of the cross-correlative response of onset versus sustain kernels.*

The NI myDAQ (http://www.ni.com/en-gb/shop/engineering-education/portable-studentdevices/mydaq/what-is-mydaq.html), which is used throughout PXT101 has an audio in and a relatively fast ADC which can be used for capturing signals for analysis and/or for use as cross-correlation kernels.

Modern music visualization software typically decomposes a polyphonic performance using a fast Fourier transform (FFT), which is used as the basis for an aesthetically-pleasing visual representation of the performance in real-time.

It has been noted since at least the late 1970s that timbre, or musical "texture" could be considered a dimension or degree of freedom in a psychological "timbre space", which is in principle calculable or quantifiable and therefore available for further analysis by a computer program (Grey, 1977, doi:10.1121/1.381428; Wessel, 1979, doi:10.2307/3680283).

Melody extraction is the isolation of a component of a polyphonic performance, the methods for which have been extensively reviewed (Salamon et al., 2014, doi:10.1109/MSP.2013.2271648). It is recognised that melody extraction is a computationally difficult task (Bosch et al., 2016, doi:10.1080/09298215.2016.1182191), and many techniques concentrate on extracting pitch only (Salamon et al., 2014, doi:10.1109/MSP.2013.2271648), however several methods exist to extract characteristic features from audio recordings, which have been recently reviewed (Alías et al., 2016, doi:10.3390/app6050143) and compared (Siedenburg et al., 2016, doi:10.1080/09298215.2015.1132737).

This micro-project is explorative in nature and will form the basis of an MSc Physics research project for summer 2018 (i.e. this academic year). It will therefore be possible to continue this project through into the spring semester and summer research project period." [1]

1.2: Initial Proposal

The project would be simultaneously developed by fellow MSc student Jack Godfrey, my project partner, and myself, Michael Norman. During the course of the project the goals have shifted somewhat from what was originally proposed, focusing in on a specific avenue of development. From the original abstract presented above and discussion with the project supervisor Richard Lewis and project partner Jack Godfrey the following functional specification has been produced to represent the initial project proposal.

Project Functional Specification

Task:

It has been proposed, that by cross-correlating a given audio track with a selection of pseudo-wavelet kernels, with groups of kernels generated from differing-frequency notes of the same instrument, response values could be generated. These response values, if working as intended, should then show some quantitive information about the differing likelihoods that the instruments whose kernel groups have been correlated against, are present at any given time index of the track.

By the end of the project a working prototype of the cross-correlation program produced last year should be generated, along with optimisation improvements and ease of use enhancements. It will also be necessary to reassess the source files. In work done previously by previous project researcher and past MSc student Javiera Zamorano Osorio, each kernel group was created by stretching and compressing a single kernel to represent the different frequencies, this would seem an inaccurate method which could perhaps be improved.

It is also hoped that during the development and testing of the program some idea of the axes of 'timbre-space' may be identified. Using the relative response strengths of different instrument types and how they related to each other it might be possible to gain some understanding of how the brain interprets some timbres as sounding similar, and others as completely different.

Instructions:

1. Find new source files.
2. Create a kernel cross-correlating application to correlate the instrument kernel groups with an audio track.
3. Compare results from the correlation of different kernel groups against different audio tracks of different instrument content.

Rules:

1. The source files should be produced either via recording new sources ourselves or through another means, potentially analytically interpolating waveforms, or the location of a suitable online database.
2. The application should return response values for each kernel inputted over an array of different time indexes.
3. The application will be suitable to handle kernels of different lengths, as an optimisation to reduce program run time, it is speculated that the same information is stored in fewer numbers of samples in higher. frequency notes, seen as the period, and thus the repetition of information, is much smaller.
4. The application will also handle kernel-groups of differing numbers and frequencies of kernel. Not all instruments have the same frequency range, thus correlating outside the frequency range of any given instrument is a waste of processing time, and can be optimised.
5. The application will produce a spectrogram as method of displaying the response information to the user in a readable manner.

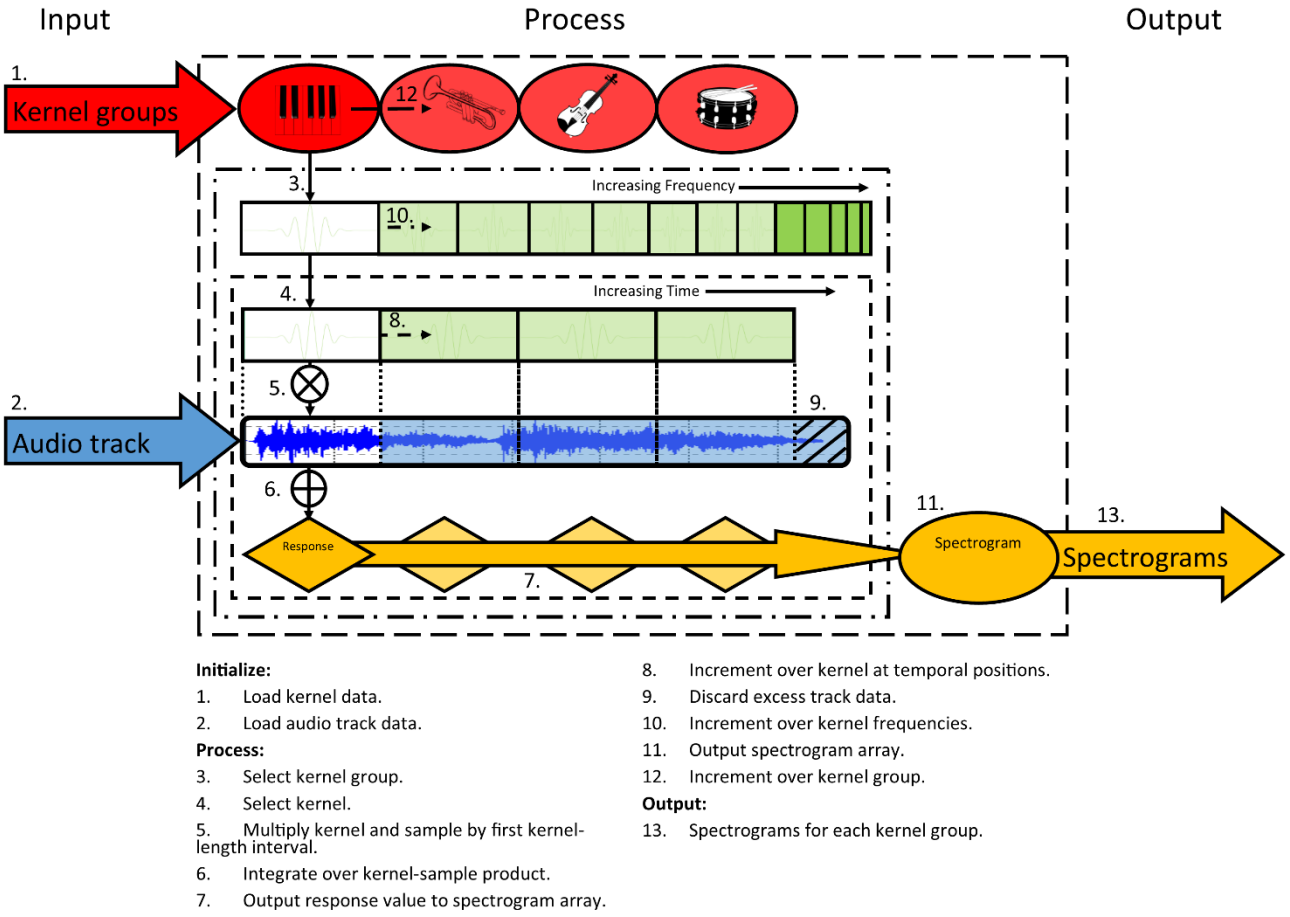


Figure 1.1: Pseudo flow-diagram of proposed cross-correlation application.

1.3 Evolution of the project specifications

As would perhaps be expected, the specifications were refined with time as insights into the project were gained, and decisions regarding direction and focus were finalised. It was decided within the first week of the project's commencement that the first priority should be to obtain new source files, whichever method was chosen, the reasoning being that without any kernels to test the cross-correlation, work on that aspect would be extremely difficult to test.

During the first week a promising online database of source files was located, a University of Iowa Electronic Music Studios database[2] , and by week two it was decided that the database would be used throughout the project, essentially achieving one of the initial objectives. Once the files had been acquired, the extensive task of converting all, or at least a large portion of the obtained source files into a pseudo-wavelet kernel format remained. Manually, this would have been a time-consuming activity that would have to have been repeated in order to propagate any changes to the way the kernels were processed. Thus, it was decided that one of the main objectives of the project was to be the creation of an Automatic Kernel Processing (AKP) application, which was the largest shift in goals throughout the project and would come to comprise the main achievement of the semester. An outline of the method proposed for the APK can be seen below in **Figure 1.2.**

APK Functional Specification

Task:

The cross correlating application being developed requires a database of pseudo-wavelet kernels. In order to produce these kernels, a database of source files must first be processed. To do this manually would require significant time expenditure, as such an application to intake source files and automatically convert them into the required kernel format will be developed.

Instructions:

1. Application should read source files into memory.
2. Application should convert files into kernel format.
3. Kernel files should be a set, selectable integer number of wavelengths long.
4. Application should output kernel files for later use by the cross-correlating application.

Rules:

1. The application should read source files into memory from WAVE files.
2. The application should convert the data format for 16-bit unsigned integers into float 32, to allow some headroom for manipulation.
3. The application should find the index of maximum amplitude to use to locate part of the source file with high instrument activity.
4. Taking a sample from the index of maximum onward a fast Fourier transform should be performed, to produce a frequency spectrum.
5. A peak finding algorithm should then find the lowest frequency peak, which will correspond to the fundamental frequency.
6. Using this fundamental frequency, the application should then take a sample of a whole number of integer multiples of the wavelength dependant on a user defined variable.
7. This sample should then be multiplied by a windowing function, the hann window.
8. Then the kernel will be normalised.
9. The kernels of a kernel group will be outputted into a single binary file to be read by the cross correlating application.

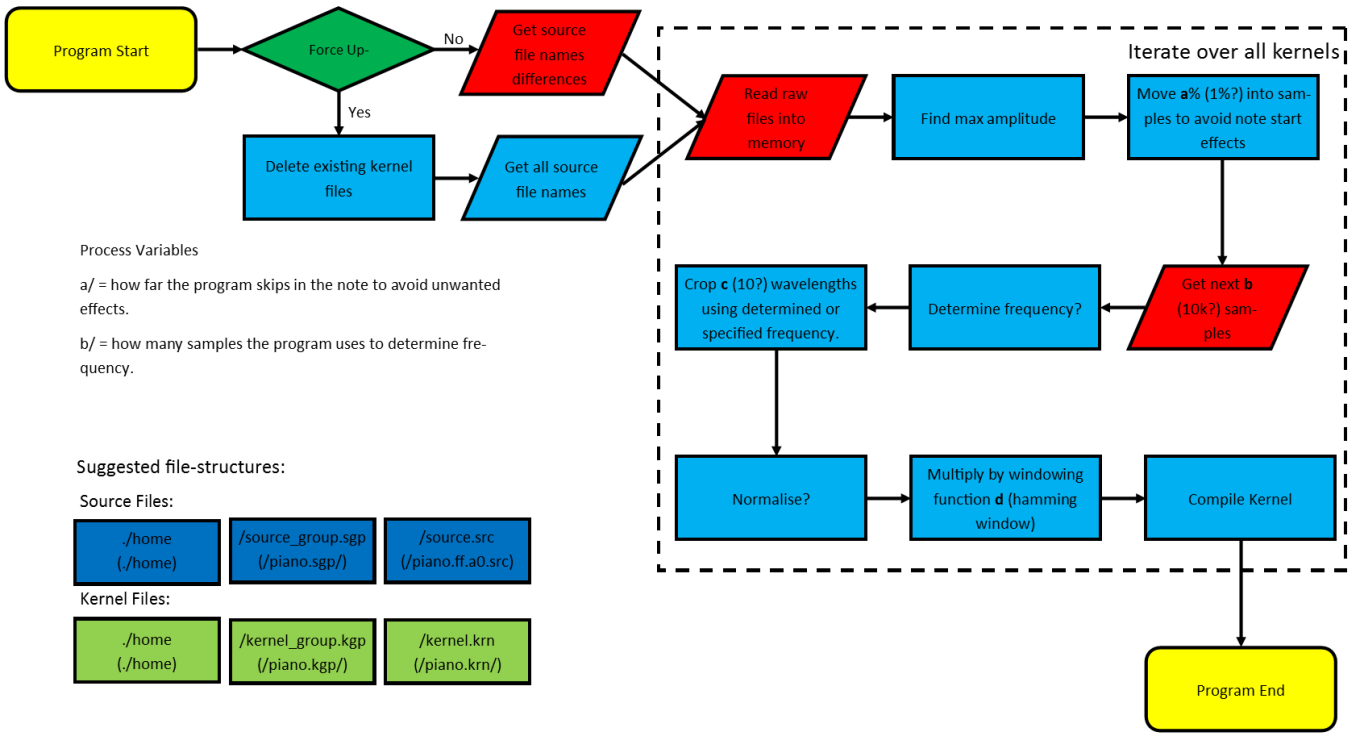


Figure 1.2: A flow diagram of the proposed AKP application.

2: Timeline of Activities

The micro-project has been carried out over a period of nine weeks, from 2017-10-05 to 2017-12-06. The timeline was not rigidly agreed on beforehand and developed dynamically over the course of the project with short term objectives refined based on current progress. The following **Figure 2.1**, gives an outline of the general activities performed each week.

Week No.	1	2	3	4	5	6	7	8	9
Date Commencing	2017-10-05	2017-10-12	2017-10-19	2017-10-26	2017-11-02	2017-11-09	2017-11-16	2017-11-23	2017-11-30
Initial research and planning									
Automatic Kernel Processor									
Planning, file loading and code structure.									
Automatic frequency determination.									
Code merge and refactor									
Wav importing and finalisation									
Cross Correlation									
Prototyping									
Plotting Spectrograms									
General Improvements									
Conclusion									

Figure 2.1: Gant diagram of activities as they actually progressed

Research, planning and coding of specific application elements were performed within individual tasks as illustrated in **Figure 2.1** and coding progress was generally continual throughout the project.

3: Software Development

3.1: Automatic Kernel Processor Development

3.1.1: Reading files to memory

Audio **sample rate** quantifies the amount of individual amplitude data points (samples) recorded per second, it is the temporal resolution, and is most nominally standardised to 44.1 kHz or 48 kHz. Audio **bit depth** refers to the resolution at which each individual audio sample is recorded, this is the amplitude resolution. The most common bit depth is 16-bit, which allows for 65,536 discrete values of amplitude, however bit depth of 24-bit are often advocated by audiophiles. In studio conditions, and where headroom is needed to perform calculations on the audio data, 32-bit depth is often used [15], and will be used during calculations in the AKP and cross correlation application.

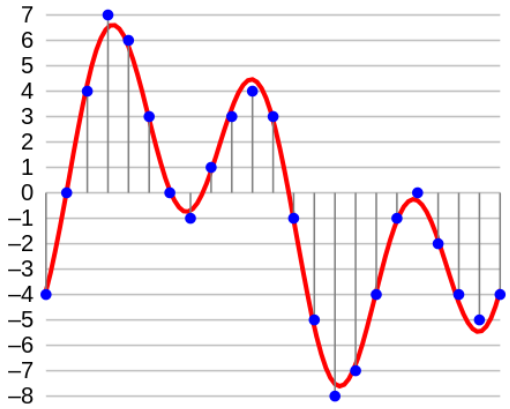
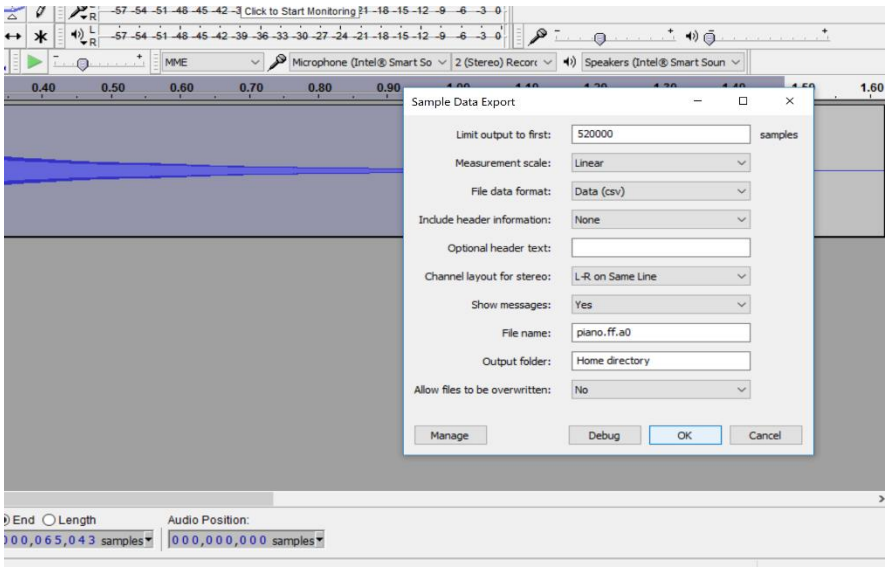


Figure 3.1 (Left)[4]**:** This figure shows an example of digital audio data storage, the x axis shows the sample number whilst the y axis shows the amplitude. This graph is showing only 4-bit data storage for illustration purposes, and thus only has 16 possible amplitude values. As can be seen the frequencies of the wave arise upon culmination of many data points.

Digital audio data is stored as a string of integers, (one for each channel), with length equivalent to the product of the recording length and the sample rate, and precision dependant on the specified bit depth. This list of integers stores all the audio data, all other information, frequency, timbre etc., emerges from the shape of this discreetly stored data. See **Figure 3.1**.

There are many different audio formats, but they can be split into two broad categories:

1. **Lossy**, (for example: *MP3*, *WMA* and *OGG* files), these formats use lossy compression to reduce file size whilst still trying to strike a compromise with quality, however, there is always some information lost, hence the name. For the purposes of this project these types of file will not be considered.[5]
2. **Lossless,** (for example, *WAVE*, *FLAC*, and *ALAC* files), these types of files do not use lossy compression. Although some formats like FLAC do use compression, this compression is lossless and does not damage the audio information. Thus lossless files are theoretically always as high quality as the recording itself, or at least as high quality as the bit rate and sample rate will allow.[6]



The AKP application uses lossless Microsoft WAVE files which do not use compression, simply storing the audio data in bit format after a standardised header. Initially the source files were converted into text documents using the Audacity audio editor, see **Figure 3.2,** and then read into the program.

Figure 3.2: Using Audacity[7] to export audio data to a character file, in this case as CSV comma delimited file. In this case the source has already been converted to mono (single channel), from stereo (two channel), although the program has so far been written to handle either.

However, after development the source files were read directly from WAVE files into memory. This was achieved via the use of a new data loading function. The header was read directly into a custom-made structure with the same memory layout as the WAVE header. After this the audio data itself was read, converted into single-precision floats and stored in memory as arrays.

```
239  void readWav(char* fn, wav_header** ret_header, float** data_ret, size_t* ret_num_samples)
240  {
241      FILE* f = fopen(fn, "rb");
242      if (!f)
243      {
244          fprintf(stderr, "Warning! Could not open file: %s\n", fn);
245          *ret_header = NULL;
246          *data_ret = NULL;
247          ret_num_samples = NULL;
248          return;
249      }
250
251      wav_header* header = malloc(sizeof(wav_header));
252      fread(header, sizeof(wav_header), 1, f);
253
254      size_t num_samples = get_num_samples(header);
255
256      if (header->bits_per_sample != 16)
257      {
258          fprintf(stderr, "Warning! Only supports 16 bits per sample\n");
259          return;
260      }
261
262      int16_t* data = (int16_t*)malloc(sizeof(int16_t)*header->subchunk2_size);
263
264      fread(data, 1, header->subchunk2_size, f);
265
266      int num_chnls = (int)header->num_chnls;
267      float* out_data = (float*)malloc(sizeof(float)*num_samples*num_chnls);
268
269      float max_val = 0.0f;
270
271      for (size_t i = 0; i < num_samples; ++i)
272      {
273          for (size_t j = 0; j < num_chnls; ++j)
274          {
275              out_data[i*num_chnls+j] = (float)data[i*num_chnls+j];
276              max_val = (fabs(out_data[i*num_chnls+j]) > max_val) ? fabs(out_data[i*num_chnls+j]) : max_val;
277          }
278      }
279
280      *data_ret = out_data;
281      *ret_num_samples = num_samples;
282      *ret_header = header;
283
284      fclose(f);
285  }
```

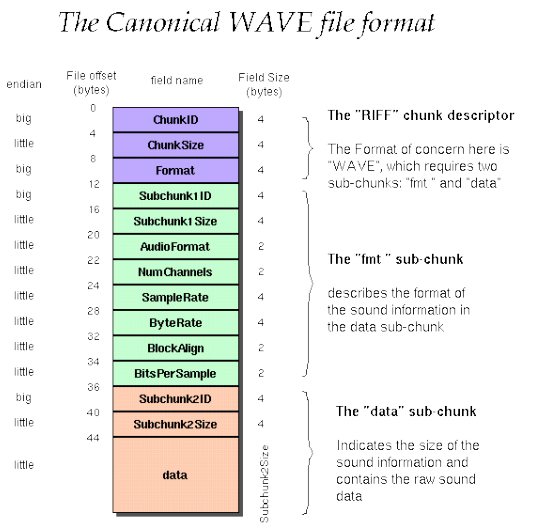


Figure 3.3 (Left): Wav reading code used in both the AKP and the cross-correlation application. NOTE: the function only works with 16-bit audio files and thus just checks for this. **Figure 3.4 (Above):** Diagram of wave file format.

```
56  typedef struct
57  {
58      size_t name_len;
59      char* name;
60
61      wav_header* header;
62
63      float freq_guess;
64
65      size_t data_len;
66      float* data;
67
68      float* max;
69      size_t* max_idx;
70
71  }src;
```

For ease of use a function was created to read the file names inside the source files directory, check their extensions then import all the source files into memory, grouped into instruments by group subfolder, see Section **4.1.1** for more information about the required AKP file structure. The read source files were initially stored within arrays of arrays, however with development this was updated to store kernel information within a structure, as seen in **Figure 3.5** left**,** which tidied code and allowed for easier sorting of sources and kernels by individual variables later on.

Figure 3.5(Left): Source structure code snippet

3.1.2: Automatic frequency determination (Fourier transforms)

In order to automatically determine the frequency of any selected sample, it was determined that using Fourier transforms was the best option.

A Fourier transform is a mathematical function that, when applied to any given waveform, can determine its sine and cos composition, turning an array of time indexed amplitudes into an array of frequency indexed amplitudes. **Figure 3.6** below illustrates how a Fourier transform splits a waveform into its component frequencies, here showing different harmonic circles to represent the different frequency elements.

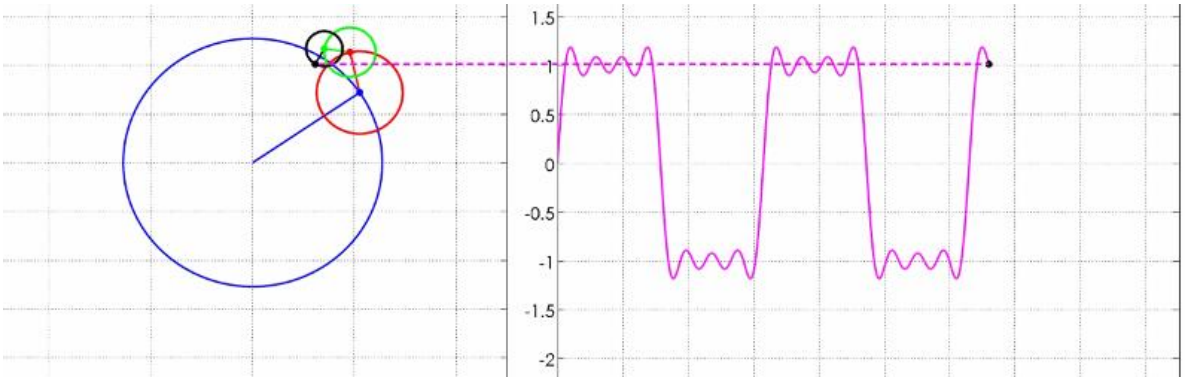


Figure 3.6[8]**:** Fourier transform visualised as the determination of harmonic circles.

A Fourier transform on any real function will return both real and imaginary components due to the manner in which the function operates. The imaginary part of the function is simple there to capture wave elements that do not start in phase with the signal.

The Fourier transform is described by the following general **Equation 3.1**, for any real value x.

$$\hat{f}(x) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \varepsilon x} dx \tag{3.1}$$

However since digital audio data is quantised we will be using the Discrete Fourier Transform (DFT) [9], which has the following form, seen in **Equation 3.2.**

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i 2\pi k n / N} \tag{3.2}$$

This equation takes a sequence of N complex numbers and turns them into another sequence of complex numbers, now indexed by frequency domain rather than time. The effect of applying this DFT to some simple composite sine

functions can be seen in **figure 3.7** right. As can be seen the function takes a sequence of values comprising a sine wave, then turns it into a signal with distinct peaks at the present frequencies.

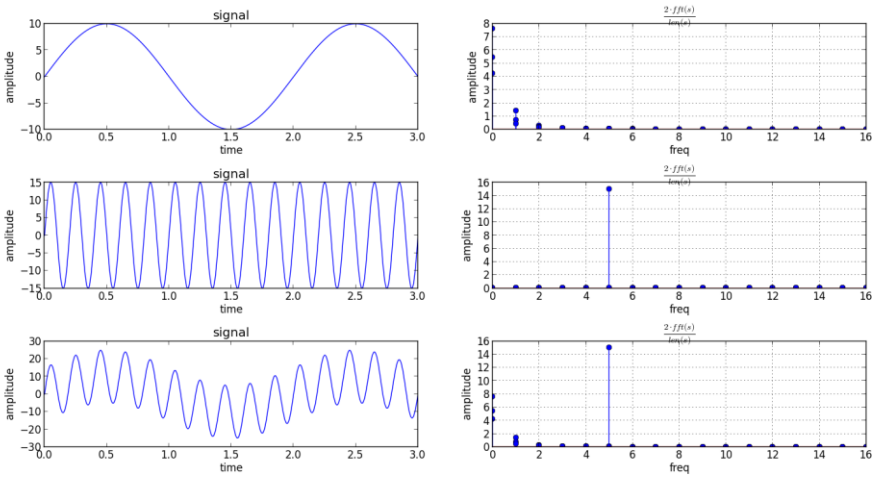
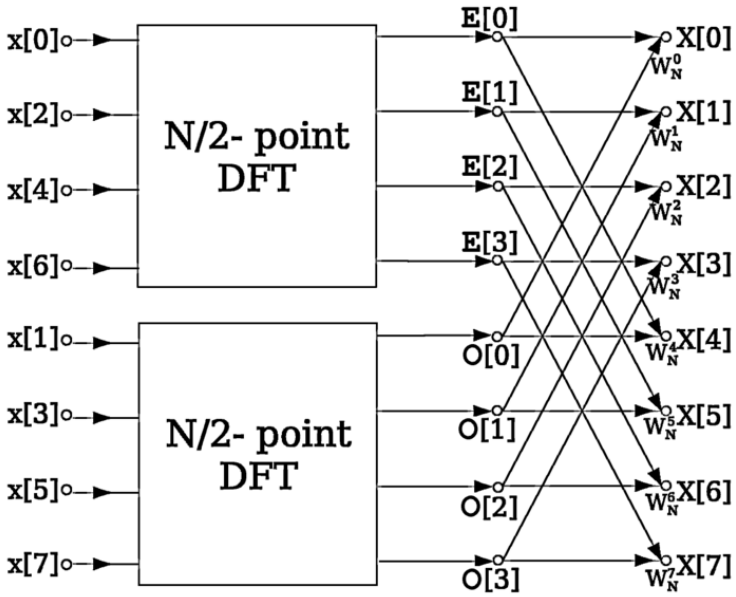


Figure 3.7 [20]: Fourier transform of various sine functions.

This DFT will be used to split our inputted source data into its component frequencies, since the input will be a natural audio signal its Fourier transform will not be nearly so clean and comprised of many component frequencies, however the aim is only to isolate the fundamental frequency, so this should still be possible.



In the cause of program speed and efficiency, using an ordinary DFT function alone would not be optimum, it would have an execution time of the order n^2, where n is the length of the sequence to be transformed. whereas using a faster method offered by a Fast Fourier Transform (FFT) algorithms this could be improved to approximately the order of n*log*(n), a significant improvement when talking about the thousands of data points that will be analysed.

Figure 3.8 (Left) [10]: FFT using Cooley-Tukey algorithm.

Figure 3.9 (Below): Code snippet showing the FFT function.

As is seen in **Figure 3.8** above, the FFT utilised by the program essentially splits the sequence to be transformed in half recursively, then performs Fourier transforms on the bits using the components from its counterpart before re-"twiddling" them by multiplying by complex routes of unity [11]. The functions implemented to utilise this method are seen in **Figure 3.9** to the right.

```
497  void _calcFFT(float complex* input, float complex* buf, int n, int step)
498  {
499
500      // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
501      //
502      // Fourier transform sub function (recursive)
503      //
504      // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
505
506      if (step >= n) return;
507      _calcFFT(buf, input, n, step*2);
508      _calcFFT(buf+step, input+step, n, step*2);
509
510      for (int k = 0; k < n; k += 2*step)
511      {
512          float complex w = cexpf(-I*M_PI*k/n);
513          input[k/2]   = buf[k] + w*buf[k+step];
514          input[(k+n)/2] = buf[k] - w*buf[k+step];
515      }
516  }
517
518  void calcFFT(float complex* input, float complex* output, int n)
519  {
520
521      // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
522      //
523      // Computes the fourier transform of inputted array
524      //
525      // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
526
527      float complex* buf = malloc(sizeof(float complex)*n);
528      for (int i = 0; i < n; ++i)
529      {
530          buf[i] = input[i];
531          output[i] = input[i];
532      }
533      _calcFFT(output, buf, n, 1);
534  }
```

3.1.3: Automatic frequency determination (Peak Finding)

Once the FFT function was successfully integrated into the program, the next step was to use the generated Fourier spectrum to find the fundamental frequency of the note in question. There was an easy reference to test against seen as our source database [2] had all of the notes labeled, and a conversion from the note to its respective frequency is easy using a conversion table as seen in **Appendix 1.**

Using the method, and selected note samples used as tests, the feasibility of using this method to automatically determine frequency could be tested. The FFT takes a sample of notes mapped from the index of maximum amplitude onwards and tries to determine the frequency of the sample. As can be seen in **Figure 3.10**, a piano note and its transform, this method was often very successful, with the fundamental frequency being by far the largest peak, therefore to determine the notes frequency, simply finding the index of the maximum value, and finding the frequency from this gave an accurate guess of the wave's frequency to within 1Hz.

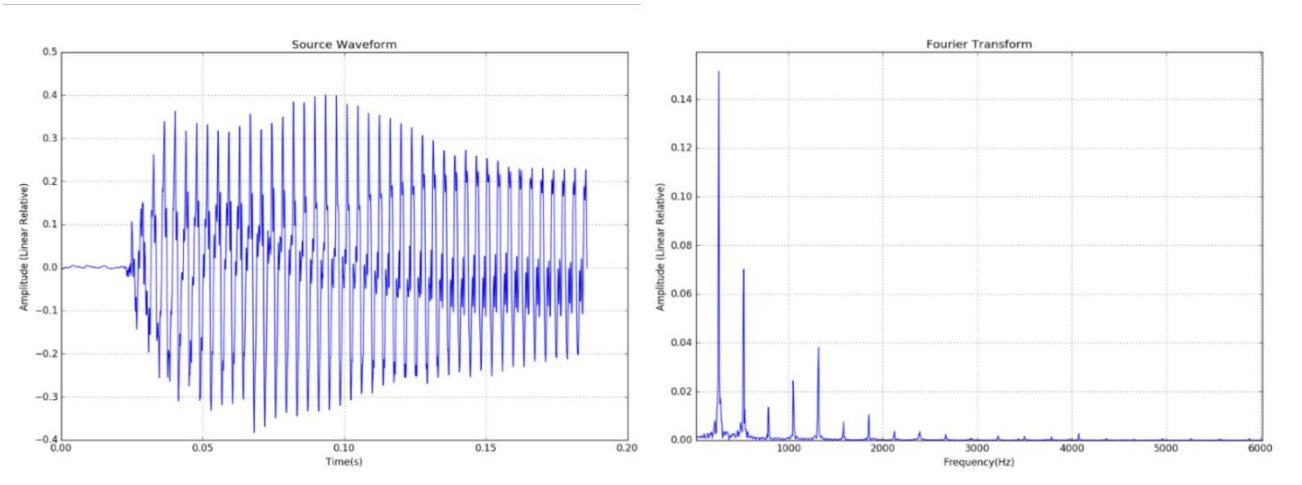


Figure 3.10: Frequency spectrum produced by my FFT function on piano, C4.

However, in an unforeseen turn of events, the maximum amplitude of the Fourier transform was not always correspondent to the fundamental frequency and could vary across the span of the note, below the violin waveform and its transform can be seen, in **Figure 3.11,** however the first peak, the fundamental frequency, lies quite substantially lower than the second harmonic.

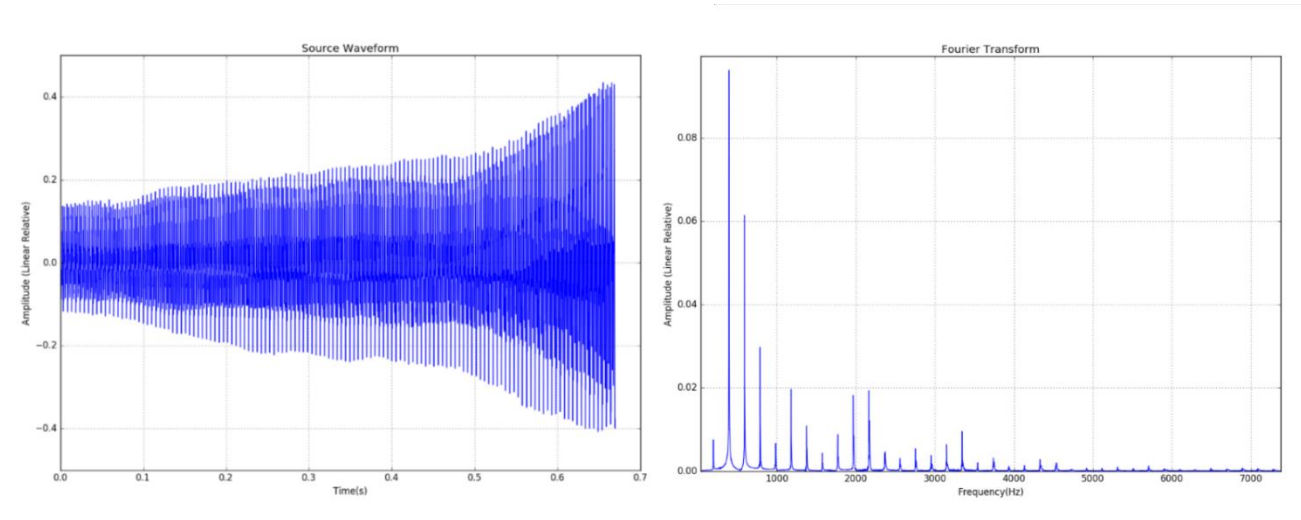


Figure 3.11: Frequency spectrum produced by my FFT function on violin, G3.

It would seem this is actually fairly common for musical instruments, thus, instead of finding the maximum amplitude of the Fourier transform, the lowest frequency peak would need to be determined in order to always find the fundamental frequency.

Programming a peak finding algorithm, proved much harder than simply finding the maximum. Since the program was being written in C, I have yet to implement a library which has an in-built function for this purpose. In the end, the program was written to find the top few percent of points, then find the lowest frequency of the points, with

```
    //Calculate FFT magnitude array:
    mag_elm* mag_arr = malloc(sizeof(mag_elm)*n);
    calcFFTMag(n, input, &mag_arr);

    //Sort by magnitude:

    qsort(mag_arr, n/2+1, sizeof(mag_arr), mag_elm_cmp);

    //Create new array of top (mag_filter) num of elements:

    mag_elm* top_mag_arr = malloc(sizeof(mag_elm)*mag_filter_int);
    for (size_t i = 0; i < mag_filter_int; ++i) { top_mag_arr[i] = mag_arr[i]; }

    free(mag_arr);

    //Apply square window and find lowest peak:

    for (size_t i = 0; i < mag_filter_int; ++i)
    {
        float freq = top_mag_arr[i].idx * config.smpl_rate/n;
        float ampl = top_mag_arr[i].mag * 2.0f/n;
        if (freq > start_skip && freq < min_freq)
        {
            if (min_freq_idx - top_mag_arr[i].idx <= consec && ampl < ampl_min_freq)
            {
                //If lowest values are consecutive, and not higher than before, skip.
                consec = min_freq_idx - top_mag_arr[i].idx + consec_o_int;
                continue;
            }
            else
            {
            min_freq = freq;
            ampl_min_freq = ampl;
            min_freq_idx = top_mag_arr[i].idx;
            consec = consec_o_int;
            }
        }
    }
```

code implemented in order to remove points with adjacent frequency index, thus hopefully finding only the lowest peaks. This has proved fairly reliable after adjustment of the filtering parameters, but does not function at extreme frequency values.

Figure 3.12(Left): Code snipped showing lowest peak finding method

After the frequency was determined, a user-controlled integer number of period intervals was stepped into the note from the index of maximum amplitude, then a sample was taken from the last step onwards by another user controlled integer number of steps to be our kernel data.

3.1.4 Normalisation

Each of these samples will have different amplitude ranges, due to differences in how the sources were recorded, and the amplitude at which the instrument can produce the note in question, thus the samples have to be normalised before use in cross correlation. Initially kernels were normalised by dividing by the maximum amplitude, however, due to the nature of cross correlation, response values depend on integral sums. They were later updated to instead normalise to the squared integer, to give them the same area between the waveforms and the 0 value.

3.1.5 Windowing Function

After the normalisation of the sample has taken place, it must be multiplied by a windowing function. Due to the way that cross correlation works, it must be padded by zeroes and yet remain a continuous function. Thus, a windowing function must be implemented.

Generally, windowing functions are implemented when there is a specific feature within a data set that is the desired focus, often, for example, when examining the results of Fourier transforms, there can be a lot of noise and other features outside of the peak that you are examining.

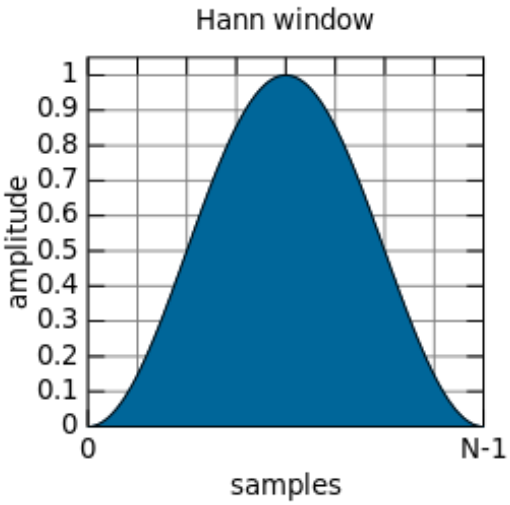


Figure 3.13(Left) [12]**:** Arbitrary hann window, tailing to 0 at boundaries.

The widow used was the hann window which has the following Equation 3.3:

$$w_n = sin^2\left(\frac{\pi n}{N-1}\right) \qquad (3.3)$$

3.1.6 Saving to binary file

After the windowing function is applied the kernel has been successfully generated and can be saved. A custom ".kern" binary file is outputted for each **kernel group** rather than from each individual kernel, it stores the data in bits in a similar way to waves but in a custom format. The binary file stores the number of kernels, the sample length of each of the kernels, the frequency guess of each kernel, and finally the full data set for each kernel. See **Figure 3.14** overleaf.

Field Name	Field Size(Bytes)	
Num_Kerns:	4	Number of kernels in kernel group
Kern_Lens:	4*num_kerns	Number of samples in kernel groups
Kern_Freq_Guess:	*4*num_kerns*	Kernel frequency guess
Kern_Data	*4*Sum(Kern_Sises)*	Kernel Audio Data

Figure 3.14: ".kern" file binary file format

3.2: Cross Correlation Development

3.2.1: Cross Correlation

The ".kern" binary files and the audio track files are read into memory in the same method as was applied in the AKP, using a custom loading function for the ".kern" files.

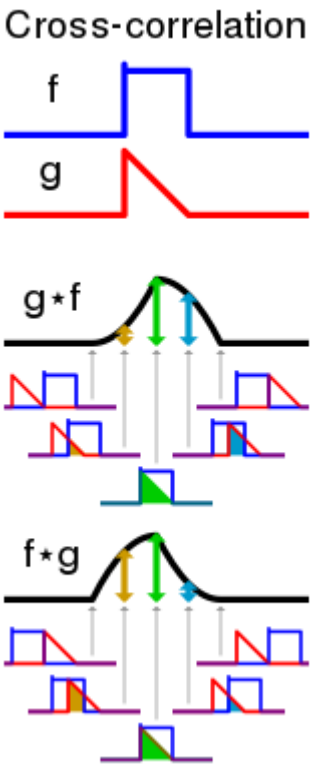


Figure 3.15 [13]**:** Diagram illustrating cross correlation

A cross correlation measures the similarity of two signals as a function of their displacement from each other. The application measures the cross-correlation of the kernel files at each integer multiple of their lengths of audio track file to produce a response value at each integer multiple. Evidently, we will have differing numbers of response values for each kernel as they are of different lengths dependant on their frequency and thus will have different divisors with the audio track length.

The cross correlation between two signals is defined as:

$$(f \star g)(\tau) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f^*(t)\, g(t+\tau)\, dt, \qquad (6.1)$$

Or in our case for discrete signals,

$$(f \star g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f^*[m]\, g[m+n]. \qquad (6.2)$$

So, for use of **equation 6.2,** with generated kernels, f is the pseudo wavelet, and g is the audio track to be cross corelated. The application multiplies each integer multiple of the kernel length by the track, and then sums across all the values to find a value for a response at that time.

```
315  void calcCrsCor(size_t num_kerns, kern* kerns, track_s track, spect_s* spects, size_t num_spects)
316  {
317      for (size_t kern_idx = 0; kern_idx < num_kerns; kern_idx++)
318      {
319          printf("Len: %zu \n", kerns[kern_idx].data_len);
320
321          kerns[kern_idx].num_devs = floor(track.data_len/kerns[kern_idx].data_len);
322          spects[kern_idx].num_devs = kerns[kern_idx].num_devs;
323          spects[kern_idx].dev_len = kerns[kern_idx].data_len;
324
325          printf("Devs: %zu \n", kerns[kern_idx].num_devs);
326          printf("Num Kerns: %zu \n", num_kerns);
327
328          spects[kern_idx].data = calloc(kerns[kern_idx].num_devs, sizeof(float)*kerns[kern_idx].num_devs);
329
330          printf("Freq %f \n", kerns[kern_idx].freq_guess);
331          printf("Devs %zu\n", kerns[kern_idx].num_devs);
332
333          #pragma omp parallel for
334          for (size_t dev_idx = 0; dev_idx < kerns[kern_idx].num_devs; dev_idx++)
335          {
336
337              for (size_t smpl_idx = 0; smpl_idx < kerns[kern_idx].data_len; smpl_idx++)
338              {
339                  spects[kern_idx].data[dev_idx] += kerns[kern_idx].data[smpl_idx]*track.data[dev_idx*kerns[kern_idx].data_len + smpl_idx];
340              }
341
342              spects[kern_idx].data[dev_idx] = fabs(spects[kern_idx].data[dev_idx]);
343
344          }
345          printf("Kern Percent: %f % \n",(float)kern_idx/(float)num_kerns);
346
347          char filename[512];
348          sprintf(filename, "./spect/spectrogram_%f.csv", kerns[kern_idx].freq_guess);
349
350          printFloats(filename, 0, kerns[kern_idx].num_devs, spects[kern_idx].data, 1);
351
352      }
353
354  }
```

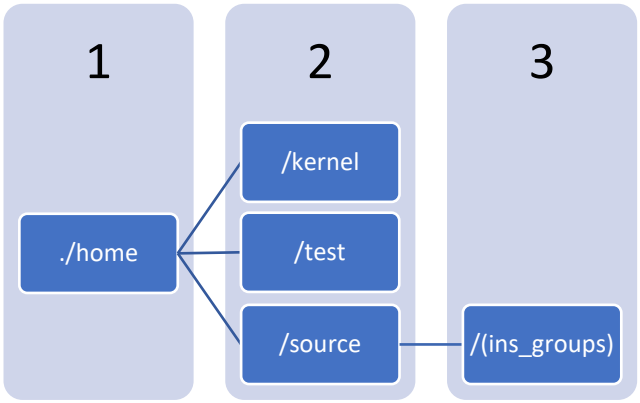
Figure 3.16 (Left): Code showing cross correlation function

4: Software Operation Instructions

4.1: Automatic Kernel Processing Operation

4.1.1: File Structure

The use of the APK requires use of a C compiler with the Gnuplot[14] library installed, the program was developed on Windows using the Cygwin[15] environment and the GGC C compiler.



The program takes a database of source files and converts these into the pseudo-wavelength kernels in a custom binary format which can be read by the cross correlator. The file structure required for the program is as described in **Figure 4.1.**

Figure 4.1 (Left): APK required folder structure.

Where './home' is the directory in which the executable is located. The kernel and test folders are output folders and thus need no initial contents. The source code must be supplied with sub folders for each desired kernel group (instrument). The sub-folders must be appended with an ".ins" extension so the application can recognise them, inside the ".ins" folders should be the desired differing-frequency source files for the specified kernel-group, i.e. the single-note audio files representing the different notes of that instrument. The source files must be of the Microsoft WAVE file format, and must be single channel and 16-bit. The sample rate can be adjusted.

4.1.2: Parameter control

Once the file structure is in place, variables within the source code should be set to desired. Ideally, a configuration-file reading function should have been generated for this purpose however due to time restrictions this was omitted.

```
952     //Expected source file parameters:
953     config.num_chnls = 1;
954     config.smpl_rate = 44100.0f;
955     config.bit_dpth = 16;
956
957     //Frequency determination parameters:
958     config.mag_filter = 0.00150f;
959     config.consec_o = 0.000150f;
960     config.start_skip = 0.000150f;
961     config.max_freq = 10000.0f;
962     config.freq_smpl_len_o = nearestPower2(1000000);
963     if (checkPower2(config.freq_smpl_len_o) == 0) { printf("Warning sample length not a powerr of 2!"); exit(1); }
964
965     //Kernel generation parameters:
966     config.kern_prds = 20;
967     config.skip_prds = 100;
968     config.wind_func = "hann";
969
970     //File extensions:
971     config.src_ext = "wav";
972     config.ins_ext = "ins";
973     config.grp_ext = "kern";
974
975     //File directory:
976     config.src_dir = "./source";
977     config.kern_dir = "./kernal";
978     config.test_dir = "./test";
```

Figure 4.2: Code sample of variable-definitions

Figure 4.2 shows the section of code in which variables should be edited to chosen values. If using the file structure as described in **Section 4.1**, then the variables under the sections files directory and file extensions should be kept at default.

The expected source file parameters should be set to the same values of the inputted source files which must all share the same parameters. It should be noted that though there are variables for bit depth and number of

channels, support for either of these values different from default is not currently functional, so these should be kept the same. The sample rate however can be changed to user desired as long as it matches source file values.

If any source file is found to have incompatible values, it will be skipped, and the program will continue. If no source files are found no kernels will be generated.

The frequency determination parameters are arbitrary values that control the filters used within the programs automatic frequency determination. These can be altered to try and get better results from the frequency determination peak finding algorithm, however it is recommended that you retain the default values as reference.

The kernel generation parameters are the parameters which you would most likely want to alter. The kernel window controls with windowing function used to window the kernel, there are two functions currently supported, “hann” and “hamming”, hann is suggested as the hamming window does not fall off to zero, more functions can be added if desired. Skip periods controls the number of determined periods away from the index of maximum amplitude that the start of the kernel sample is taken. Kern periods controls the size of the generated kernel in integer multiples of the determined period.

4.1.3: Application compilation and execution

Once the parameters have been inputted as desired, the program should be compiled. Whilst in development the application was compiled in Cygwin using the -0fast and –fopenmp flags and as such it is suggested that the same flags are used, or this might lead to errors.

```
theem@DESKTOP-MR4VOU6 ~
$ gcc akp_v19.c -Ofast -fopenmp
```

Figure 4.3: Compiling the AKP

Using the command shown above in **Figure 4.3**, whilst the apk_v19.c source code file is inside the Cygwin working directory, will compile the program into an executable with the file name “a.exe”.

Once the program has been compiled and the previous steps described in **4.1.1** and **4.1.2** have been completed the program should now be able to successfully execute and convert your source files into kernel files.

4.2: Cross correlating Operation

4.2.1: File structure

Kernel files produced from the AKP application should be outputted in a manner which can directly interface with the cross-correlation application. Several new folders will need to be created, in accordance with **Figure 4.4.** Tracks to be correlated with should be placed inside the /tracks files, and should be in the WAVE format with the same parameters as your input source files (**there is no automatic check to ensure this so be careful)**.

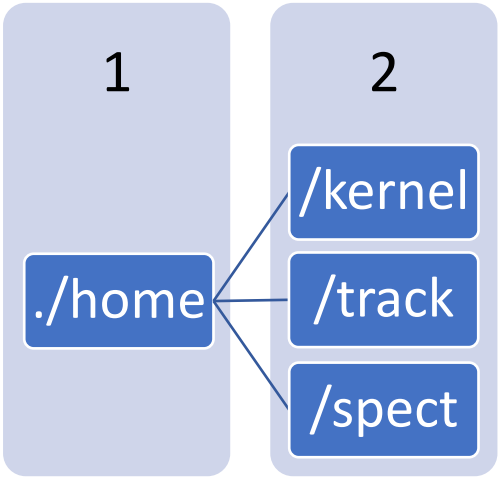


Figure 4.4 (Right): File structure required for cross correlation operation.

4.2.2: Parameter control

```
359     char* track_folder = "./tracks";
360     char* kern_folder = "./kernal";
361
362     char* kern_name = "Bassoon.ff.stereo";
363     char* track_name = "c_major_scale";
364
365     char* kern_ext = "kern";
366     char* track_ext = "wav";
367
368     size_t num_chnls = 1;
369
370     size_t num_kerns;
371     kern* kerns;
372
373     float smpl_rate = 44100.0f;
```

Figure 4.5: Code snippet of cross correlation variables

Cross correlation parameters are seen in the **Figure 4.5** above. Again, the folder and extension variables should be kept as default for the above instruction to hold. The kernel name variable, controls the kernel being used and the track name controls the audio track, stored within the track folder, to be cross correlated against.

4.2.3: Compilation and Execution

The program should be complied and executed in the same method as was described previously in **Section 4.1.3**. It should be noted that the GnuPlot library must be installed in order for the program to successfully compile. Once run the program will save the produced spectrogram into the home directory.

5: Results

5.1: Application Research and Development

The project specification quickly evolved into one focused more on software development than the direct production of scientific results. Whilst the eventual end goal of the continuing project is to investigate timbre, psychoacoustics and the axis of timbre space. Most of what has been achieved over the limited time of this micro project has been the creation of software and the advancement of software techniques to produce kernels for cross correlation.

The development of the APK application has largely been a success as will be seen in **Section 5.2** ahead, and the cross-correlation prototype has at least proved the concept to some extent, though clearly there is much work to be done to create methods that allow for rigorous analysis and comparison of results.

5.2: Kernel generation tests

In order to test the accuracy of the developed kernel processing application, it was necessary to produce waveform graphs of several produced kernels to qualitatively access their success. All kernels were produced using source files from the University of Iowa Electronic Music Studios database[2].

We have produced kernels for a large range of instruments with general success. As can be seen in **Figure 5.1** overleaf, source files with higher frequencies produce smaller kernels. This is desired behaviour as kernels were scaled depending on their period.

Irregular kernels were seen at frequency extremes to which the current frequency determination method with current filtration parameters was insufficient, and thus sometimes produced a wildly inaccurate kernel. To combat this issue sample waveforms were plotted across the kernels to find the frequencies at which the APK was no longer functional. It was found that notes in the seventh octave or higher could not be successfully identified, and thus these source files were removed from the source database.

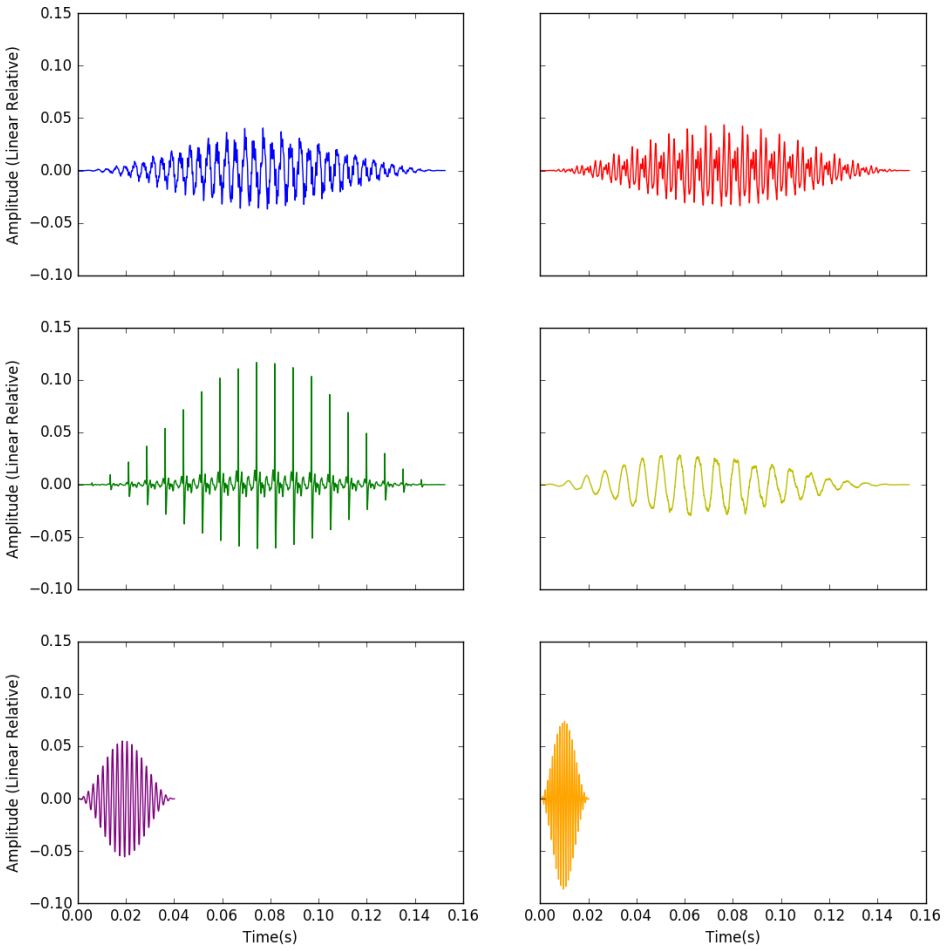
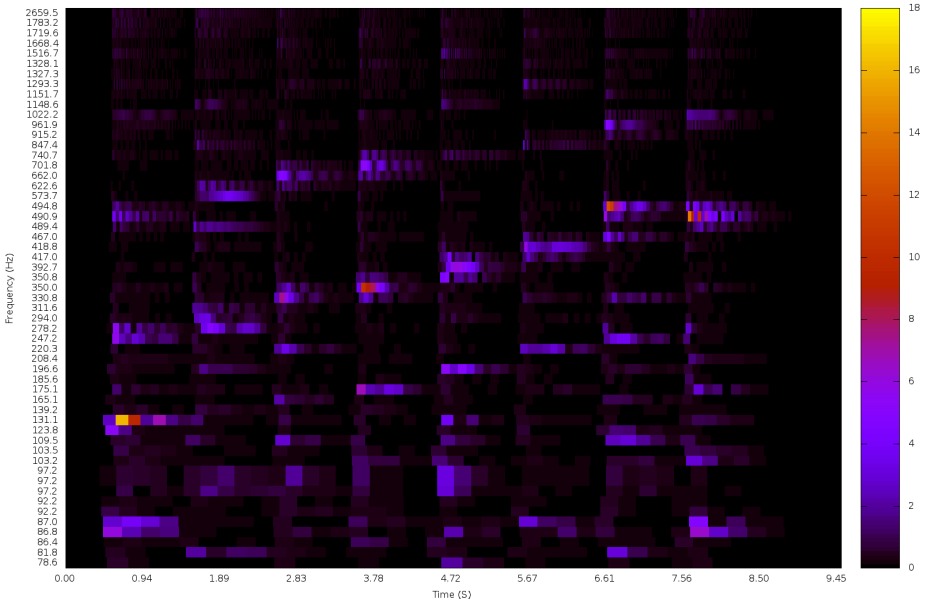


Figure 5.1 (Left): Various kernel waveforms produced using the developed AKP application, depicting differing characteristics and frequencies. The kernel size was set to twenty wavelengths. Kernels are normalised to their square integral. *Top Left:* Piano C3, *Top Right:* Bassoon C3, *Middle Left:* Bass Trombone C3, *Middle Right:* Bass Flute C3, *Bottom Left:* Xylophone B4, *Bottom Right:* Violin B5.

It is hoped that this will not have a significant effect on the data gathering powers of the cross-correlator as the use of such high frequency notes in musical compositions is relatively rare. Low notes which may produce irregular kernels, were not usually as badly wrong as the higher frequencies, with the frequency determination not failing as readily and when occurring usually only with error below 50% of their desired frequencies. Since these kernels will still correctly demonstrate the 'texture' of their respective notes, and are normalised during the process, they have not as yet been omitted.

5.3: Cross correlation spectrograms



To test the cross-correlation application and the generation of its respective spectrograms, several piano pieces were chosen. Correlations were performed with several kernel groups.

Figure 5.2 (Left): C major scale[16] correlated with piano kernel group. Response values are in arbitrary units only useful for comparison.

The piano kernel group generated by the AKP was first correlated against a C major scale, in which a note is played every second increasing in frequency whilst excluding sharps and flats, i.e., C4 (261Hz), D4 (293Hz), E4 (329Hz), F4 (349Hz), G4 (392Hz), A4 (440Hz), B4 (493Hz), C5(523Hz). The resulting spectrogram can be seen in the above **Figure 5.2,** in which high response values occur around the expected frequencies, as well as in lines of harmonics above and below the notes by octave intervals, a good though not perfect result.

Several issues can be seen in the spectrogram, the frequency determination algorithm is not always perfectly accurate nor is it assured that our source files will be pitch perfect notes, as such the frequencies determined for individual kernels are often off their supposed true values by some margin, with notes even rarely swapping in order as can perhaps be seen in the seventh note, B4, at around 6.61 seconds.

A further issue can be seen when noting the frequency axis where several kernels can be seen to have the same frequency. Again, possibly due to an error in the frequency determination process. These issues are hard to troubleshoot and are hence still unresolved.

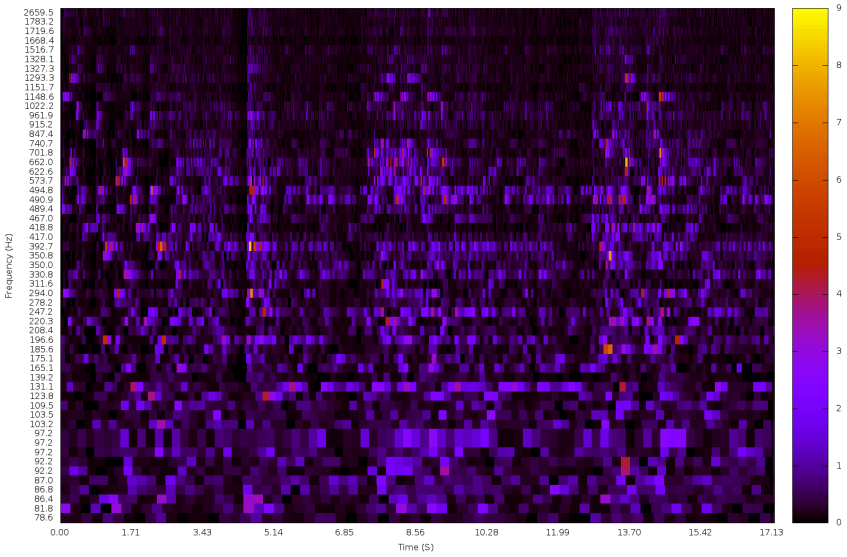
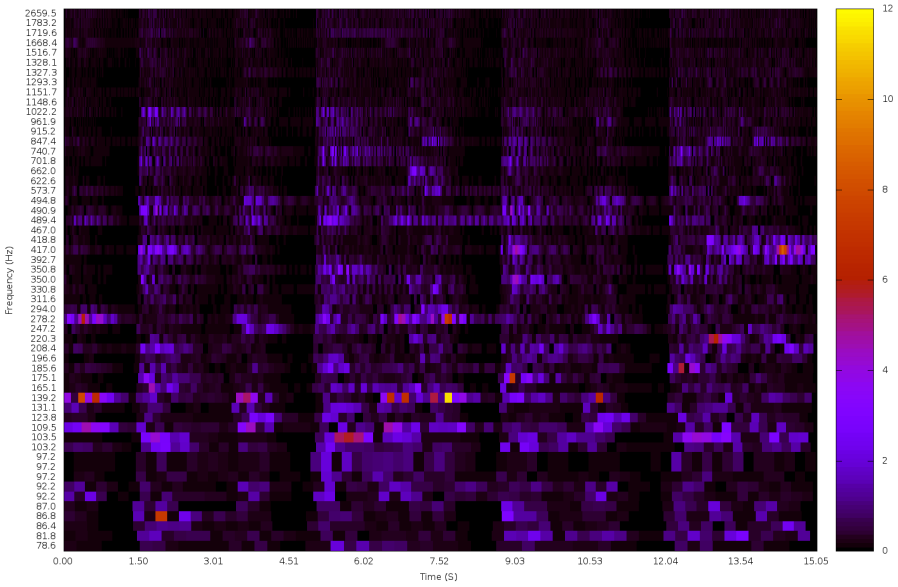


Figure 5.3 (Right): The Entertainer[17] – by Scott Joplin, correlated against the piano kernel group.

Looking at the spectrogram of a correlation with a more complex piece, the picture becomes a lot more clouded, although some recognisable structure can be seen. **Figure 5.3** illustrates a correlation of the piano kernel group with the start of a popular piano piece, the Entertainer – by Scott Joplin. A cascade of descending notes as the piece begins, followed by a district pause at around four seconds then a loud cord. Subsequent sections are also divided into distinct higher and lower frequency subdivisions and the base line can be seen running throughout the piece.

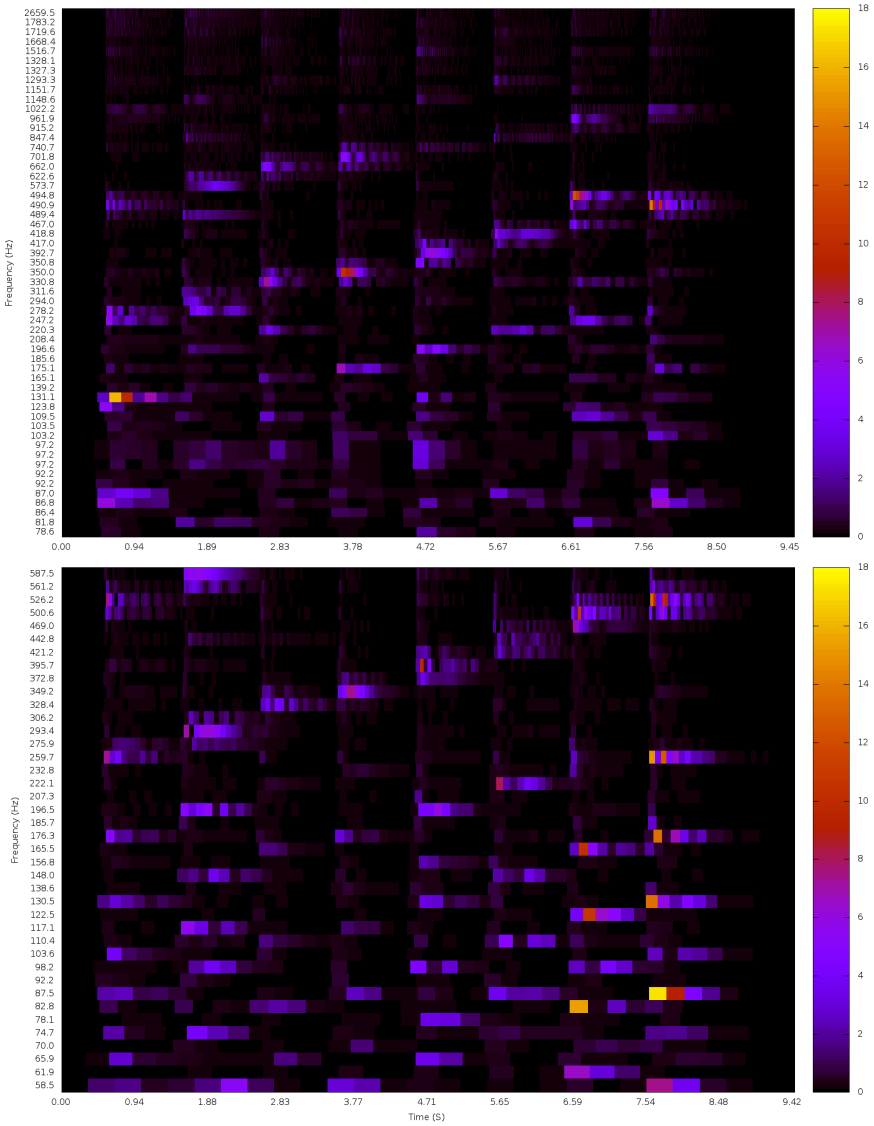


It should be noted that the creative commons track used for the correlation was not of particularly high quality, which may be an indication of the level of noise seen in the spectrogram. The piano was also significantly different timbre to that of the piano used to record, showing that even within one instrument timbre can vary greatly.

Figure 5.4 (Left): Hungarian Rhapsody No. 2[18] – by Franz Liszt, correlated against the piano kernel group.

Figure 5.4 shows the beginning a second piano piece, Hungarian Rhapsody No. 2 – by Franz Liszt, correlated again with the piano kernel group. The distinctive pauses and chords can easily be identified, though again identifying which chords are being played might be an interesting challenge.

5.4: Kernel Group comparison



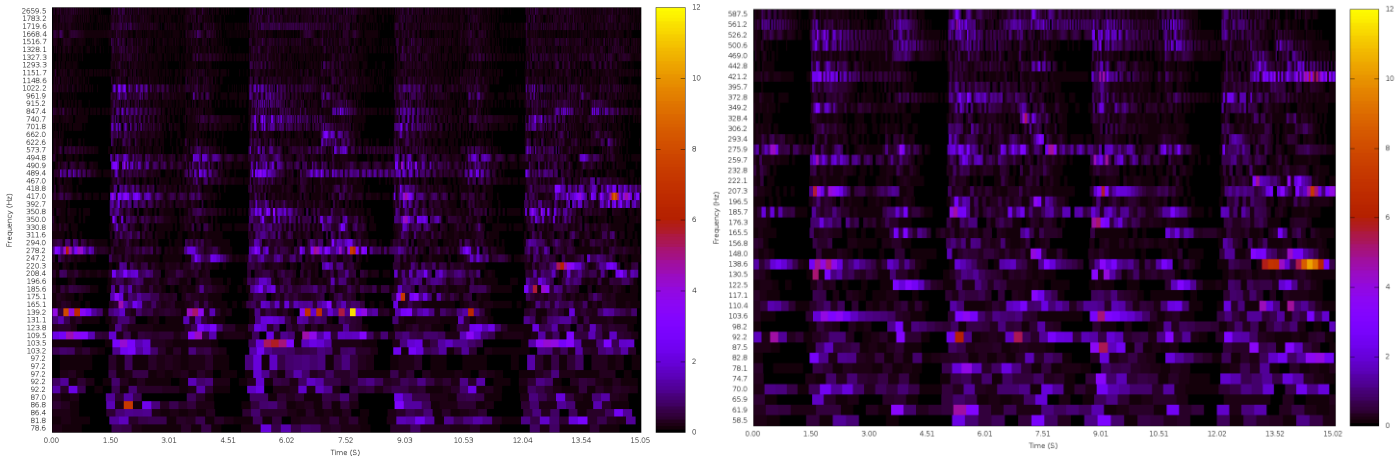
To analyse the instrument differentiation capability of the cross-correlation application, two kernel groups of different instruments, Piano and Bassoon, were correlated against the same music track. The piano C major scale previously mentioned [14]. As can be seen from **Figure 5.5**, the differences in peak response times are very minimal, with similar patterns visible in both, though the bassoon has distinctive harmonics which seem to peak erroneously at certain low frequencies which may be causing a spike in peak response.

In **Figure 5.6**, the bassoon and piano kernel groups are compared as correlations of the piano piece previously mentioned, Hungarian Rhapsody No. 2, showing again similar response values, although with less of the detail seemingly preserved.

A good way to compare the response values between different kernel groups has yet to be implemented, though there are some suggestions on ways that this might be implemented in **Section 7.**

Figure 5.5(above): Spectrograms of C major scale[16] correlated with: *top*: piano, *bottom:* bassoon.

Figure 5.6(below): Spectrograms of Hungarian Rhapsody No. 2 – by Franz Liszt correlated with: *left*: piano, *right:* bassoon.



6: Discussion of Results

Few results of scientific value have been generated as part of this project since it has evolved more into an exercise of software development that scientific research, though it should be noted that the software being developed does have the end goal of scientific interest.

As yet there has been no error propagation throughout the project as we have yet to obtain any specific results from data.

6.1 Discussion of AKP results

On visual inspection, the kernels generated via the AKP seem to look as expected. Other than visual means, there has been no other attempt to authenticate their accuracy, though it could be said that the cross-correlation results and the production of the spectrograms themselves do somewhat validate the kernel generation method. Careful work is still needed to find the operational frequency range of the APK, and perhaps adjust the peak finding filtration methods to improve results.

The APK is not functional when applied to instruments without a distinct fundamental frequency, for example many types of percussion.

6.2 Discussion of Cross Correlation results

The reliability of the spectrograms is questionable and currently unquantifiable. Looking at the differences between various spectrograms produced using different kernel groups acting on the same audio track, it is difficult to determine any kind of pattern as to what causes higher frequency responses. A survey in much greater detail into spectrograms across a range of instruments with some quantifiable way of assessing the response values of the entire spectrogram is suggested for future work.

7: Future work

Much progress has been made toward the final goals of identifying the axis of timbre space and creating an application to automatically determine genre, though I think there may now be distinctly different paths dependant on which of those you wanted to achieve.

7.1: AKP Future Work

To develop the AKP further it suggested that a more thorough investigation is made to find scenarios in which it fails, and adjust the program accordingly, perhaps by adjusting the fit parameters or introducing a new superior peak finding algorithm or overall frequency determination method. It is suggested however, that the APK should function well enough for most future cross-correlation applications, so long as the erroneous data is removed from the dataset before use. It is the most complete part of the project. There are also a few small ease-of-life improvements that were to be added were more time available. The addition of a configuration file would make it much more user friendly and remove the need for continual recompilation. In addition, a function that when activated would print all, or a least a sample of the kernels into image files to be visually inspected would be a good addition to identify failure cases.

7.2: Cross Correlator Future Work

As has been stated there is much room for continuation with the work carried out on the cross correlation, as it was only briefly commenced toward the end of the project. Analysis of the produced spectrogram data has, as of yet, been minimal. And any quantitive way of comparing the data is yet lacking. A suggested method to try and gauge the overall response differences between kernel groups, is to calculate the response values as ratio of each other. Doing this in practice poses significant difficulty as the grids in which the response values lie will not line up. Differing kernel lengths, differing frequencies for each note, even slightly differing orders of notes when frequency determination goes awry can occur. A Virtual grid of data of standardised dimensions would need to be created from each spectrogram as was done when they were plotted, though this time they would need to be aligned in two axis not just one, so may prove difficult.

The cross-correlation application itself is also somewhat lacking, and could do with significant tidying, implementation of error handling and many of the features present in the AKP and not in the cross correlator due to time investment.

A more long-term step, working toward a genre recognition application, would be the application of machine learning techniques to the response data found by the cross-correlator. The suggested avenue of attack it to use neural networks, trained on a soup of genres fed through the cross-correlator. It is hoped that the cross-correlation step would significantly reduce the work of the neural network over the use of the entire track data.

8: Evaluation of the Microproject Output

The project has achieved well compared against the original specifications. A functioning cross correlation application has been produced, comparable to that produced in past work as well as implementing numerous optimisations. As well as that, a new high-quality database of source files has been located, and a new application developed to automatically convert these source files into kernels.

This was probably the largest success of the project and will hopefully be of much use for future work, I feel confident that for any kind of cross correlation using the suggested pseudo-wavelet method the APK will be of use.

Optimisations were made in the suggested areas, discussed with Richard Lewis near the commencement of the project:

1. The kernels are no longer of equal size, but rather dependant on their frequency and a controllable parameter, and the cross correlator utilises this fact to improve application run time.
2. The quality of the kernels has been improved as they are no longer simply squashed and stretched from a single kernel but rather each is independently produced from corresponding source files.
3. The integration no longer occurs over unnecessary space, as all kernels are generated from real source files which are recorded from real instruments, thus no integration across frequency space occurs in areas inaccessible by any given instrument.

Little progress has been made on real time implementation of the cross correlation, though the spectrograms do show the response of the differing instruments at different times, this was already achieved in previous work. There has also been little progress with regards to determining instruments by timbre and investigation into the axis of timbre space, although that was always stated as a long-term goal and never has been within the reach of the micro-project.

Time has been managed relatively well over the course of the project and good progress has been made fairly consistently across the weeks. Some weeks especially toward the end of the project did see some dips in progress made however this was primarily due to other time commitments and could not be readily alleviated. Perhaps a little more time could have been invested in a few of the weeks during the middle of term.

Across the project, my project partner Jack Godfrey and I, whilst working together on ideas and concepts shared a separate codebase as we both preferred differing programming languages. More work could certainly have been achieved if we both worked together on a single codebase, although working in tandem did prove to have some benefits when one thought of a solution the other hadn't. It also helped to confirm results when both codebases produced similar outputs.

9: Possible application of micro project output in the wider scientific context

The projects main application to the wider scientific context might be considered the development of a software toolset that can be used to further investigate the fields of psychoacoustics and genre recognition. Though little actual scientific progress has been made the paving slabs have been laid for a future investigation with some significant challenges solved as well as several usable pieces of software.

As far as the greater scope of the project. There is still much unknown about the way humans comprehend and are affected by sound, especially music. As such any research into these areas is clearly warranted.

10: References

[1] R. Lewis, “PXT101 Advanced Experimental Techniques in Physics Micro-Project Abstracts 2017/18.” 2017.
[2] “University of Iowa Electronic Music Studios.” [Online]. Available: http://theremin.music.uiowa.edu/MISpiano.html. [Accessed: 18-Oct-2017].
[3] “Sample Rates - Audacity Wiki.” [Online]. Available: http://wiki.audacityteam.org/wiki/Sample_Rates. [Accessed: 18-Oct-2017].
[4] Aquegg, *English: Encoding an analogue signal to 4-bit linear PCM*. 2013.
[5] “Lossy compression,” *Wikipedia*. 18-Oct-2017.
[6] “Lossless compression,” *Wikipedia*. 17-Oct-2017.
[7] “Audacity® | Free, open source, cross-platform audio software for multi-track recording and editing.” [Online]. Available: http://www.audacityteam.org/. [Accessed: 18-Oct-2017].
[8] By, “Visualizing the Fourier Transform,” *Hackaday*, 17-Sep-2015. .
[9] “Discrete Fourier transform,” *Wikipedia*. 25-Oct-2017.
[10] “File:DIT-FFT-butterfly.png,” *Wikipedia*. .
[11] “Cooley–Tukey FFT algorithm,” *Wikipedia*. 22-Oct-2017.
[12] O. Niemitalo, *English: Window function and its Fourier transform: Hann window*. 2013.
[13] E. W. Weisstein, “Cross-Correlation.” [Online]. Available: http://mathworld.wolfram.com/Cross-Correlation.html. [Accessed: 18-Oct-2017].
[14] “gnuplot homepage.” [Online]. Available: http://www.gnuplot.info/. [Accessed: 14-Dec-2017].
[15] “Cygwin.” [Online]. Available: https://www.cygwin.com/. [Accessed: 14-Dec-2017].
[16] “C4_Major_Scale_Piano.wav by digifishmusic,” *Freesound*. [Online]. Available: https://freesound.org/people/digifishmusic/sounds/94812/. [Accessed: 13-Dec-2017].
[17] C. S. J. M. A. Cuerden, *English: This is a recording of Scott Joplin’s rag which was composed in 1902. This performance was created by Adam Cuerden, a Wikipedian (User:Adam Cuerden) and found here: http://adamcuerden.deviantart.com/gallery/ . The MP3 file was converted to OGG by User:Major Bloodnok) and then uploaded to Commons.* 2011.
[18] F. Liszt, *Second Hungarian Rhapsody*. [object HTMLTableCellElement].

Appendices

i: Risk Assessment:

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

Risk Assessment Form

IMPORTANT: before carrying out the assessment, read the Guidance Notes

Department	School of physic and astronomy	Building	Queens building North	Room No	N/5.03(A)
Name of Assessor	Michael Norman	Date of Original Assessment	15/12/2017	Assessment No	1

Status of Assessor: Supervisor ☐ Postgraduate ☑ Undergraduate ☐ Technician ☐ Other: (Specify) .....

Brief Description of Procedure/Activity including its Location and Duration

Activity: Coding in Room N/5.03(A) using own laptop
Duration: 2017-10-05 - 2017-12-14

Persons at Risk Are they... Notes:

Staff	☑	Trained	☑	Only risk to others via being in the vicinity of electronic equipment
Students	☑	Competent	☑	
Visitors	☑	Inexperienced	☐	
Contractor	☐	Disabled	☐	

Level of Supervision Notes

None ☐ Constant ☐ Periodic ☑ Training Required ☐	Supervisor Richard Lewis's office is next door

Will Protective Equipment Be Used? Please give specific details of PPE

Head ☐ Eye ☐ Ear ☐ Body ☐ Hand ☐ Foot ☐	

Is the Environment at Risk? Notes

Yes ☐ No ☑	

Will Waste be generated? If 'Yes' please give details of disposal

Yes ☑ No ☑	Some food waste, disposed of in general waste bin

COSHH Assessment

Substance (name or formula)	Quantity (g or cm^3)	Exposure Time	Hazard	Likelihood (0 to 5)	Severity (0 to 5)	Disposal Method
N/A	N/A	N/A	N/A	N/A	N/A	N/A

Other Hazards involved in the Procedure/Activity

Hazard and how it can cause harm.	Control Measures in place and Consequence of Failure	Level of Risk (Lik x Sev=Risk)
Computer screen: can cause eye strain	Regular breaks, sit recommended distance from screen. Could make your eyes a bit sore.	2*0 = 0
Typing can case RSI	Regular breaks, ensure keyboard is at correct angle. Could get RSI.	3*1 = 3
Electric shock from electronic equipment, Faulty or damaged laptop or cable could cause health risks	Keep fluids contained, ensure wires are in good condition Could cause mild to large electric shocks	0*5 = 0
Cables: trip hazard	Keep workspace tidy	1*2 = 2

Scoring Criteria for Likelihood (chance of the hazard causing a problem)
0 – Zero to extremely unlikely, 1 – Very Unlikely, 2 – Unlikely, 3 – Likely, 4 – Very Likely, 5 – Almost certain to happen

Scoring Criteria for Severity of injury (or illness) resulting from the hazard
0 – No injury, 1 – First Aid is adequate, 2 – Minor injury, 3 – "Three day" injury, 4 – Major injury, 5 – Fatality or disabling injury

Source(s) of Information used to complete the above

Personal

Department Safety presentation

Training. Please give details of any specific training requirements for persons carrying out this procedure/activity.

Status of Trainer	Training Requirement

Further Action

Highest Level of Risk Score	Action to be taken
0 to 5	No further action needed
6 to 11	Appropriate additional control measures should be implemented
12 to 25	Additional control measures **MUST** be implemented. Work **MUST NOT** commence until such measures are in place. If work has already started it must **STOP** until adequate control measures are in place.

Additional Control Measures Required

Hazard and how it can cause harm.	Additional Controls needed to Reduce Risk	How is Likelihood or Severity reduced?	Risk Now

After the implementation of new control measures the procedure/activity should be re-assessed to ensure that the level of risk has been reduced as required.

Action in the Event of an Accident or Emergency

Depending on the severity of the accident appropriate school safety procedure will be indicated.

Arrangements for Monitoring the Effectiveness of Control

Review risk assessment on project completion

Review this assessment must be reviewed by (insert date less than 3 years from original assessment):

Name of Reviewer	Review Date	
Have the Control measures been effective in controlling the risk?		
Have there been any changes in the procedure or in information available which affect the estimated level of risk?		
What changes to the Control Measures are required?		

Signatures for printed copies:

Assessor: Date:

Approved by: Date:

Reviewed by: Date:

This copy issued to: Date:
(print name and sign)

ii: Project Updates:

Update slides available on request from Richard Lewis, LewisR54@cardiff.ac.uk

iii: Lab Diary:

Laboratory diary available on request from Richard Lewis, LewisR54@cardiff.ac.uk

iv: Source code:

Source code available on request from Richard Lewis, LewisR54@cardiff.ac.uk

V: Note to frequency conversion table

Note	Frequency (Hz)	Note	Frequency (Hz)	Note	Frequency (Hz)
C_0	16.35	C_3	130.81	C_6	1046.5
$C^{\#}_0/D^{b}_0$	17.32	$C^{\#}_3/D^{b}_3$	138.59	$C^{\#}_6/D^{b}_6$	1108.73
D_0	18.35	D_3	146.83	D_6	1174.66
$D^{\#}_0/E^{b}_0$	19.45	$D^{\#}_3/E^{b}_3$	155.56	$D^{\#}_6/E^{b}_6$	1244.51
E_0	20.6	E_3	164.81	E_6	1318.51
F_0	21.83	F_3	174.61	F_6	1396.91
$F^{\#}_0/G^{b}_0$	23.12	$F^{\#}_3/G^{b}_3$	185	$F^{\#}_6/G^{b}_6$	1479.98
G_0	24.5	G_3	196	G_6	1567.98
$G^{\#}_0/A^{b}_0$	25.96	$G^{\#}_3/A^{b}_3$	207.65	$G^{\#}_6/A^{b}_6$	1661.22
A_0	27.5	A_3	220	A_6	1760
$A^{\#}_0/B^{b}_0$	29.14	$A^{\#}_3/B^{b}_3$	233.08	$A^{\#}_6/B^{b}_6$	1864.66
B_0	30.87	B_3	246.94	B_6	1975.53
C_1	32.7	C_4	261.63	C_7	2093
$C^{\#}_1/D^{b}_1$	34.65	$C^{\#}_4/D^{b}_4$	277.18	$C^{\#}_7/D^{b}_7$	2217.46
D_1	36.71	D_4	293.66	D_7	2349.32
$D^{\#}_1/E^{b}_1$	38.89	$D^{\#}_4/E^{b}_4$	311.13	$D^{\#}_7/E^{b}_7$	2489.02
E_1	41.2	E_4	329.63	E_7	2637.02
F_1	43.65	F_4	349.23	F_7	2793.83
$F^{\#}_1/G^{b}_1$	46.25	$F^{\#}_4/G^{b}_4$	369.99	$F^{\#}_7/G^{b}_7$	2959.96
G_1	49	G_4	392	G_7	3135.96
$G^{\#}_1/A^{b}_1$	51.91	$G^{\#}_4/A^{b}_4$	415.3	$G^{\#}_7/A^{b}_7$	3322.44
A_1	55	A_4	440	A_7	3520
$A^{\#}_1/B^{b}_1$	58.27	$A^{\#}_4/B^{b}_4$	466.16	$A^{\#}_7/B^{b}_7$	3729.31
B_1	61.74	B_4	493.88	B_7	3951.07
C_2	65.41	C_5	523.25	C_8	4186.01
$C^{\#}_2/D^{b}_2$	69.3	$C^{\#}_5/D^{b}_5$	554.37	$C^{\#}_8/D^{b}_8$	4434.92
D_2	73.42	D_5	587.33	D_8	4698.63
$D^{\#}_2/E^{b}_2$	77.78	$D^{\#}_5/E^{b}_5$	622.25	$D^{\#}_8/E^{b}_8$	4978.03
E_2	82.41	E_5	659.25	E_8	5274.04
F_2	87.31	F_5	698.46	F_8	5587.65
$F^{\#}_2/G^{b}_2$	92.5	$F^{\#}_5/G^{b}_5$	739.99	$F^{\#}_8/G^{b}_8$	5919.91
G_2	98	G_5	783.99	G_8	6271.93
$G^{\#}_2/A^{b}_2$	103.83	$G^{\#}_5/A^{b}_5$	830.61	$G^{\#}_8/A^{b}_8$	6644.88
A_2	110	A_5	880	A_8	7040
$A^{\#}_2/B^{b}_2$	116.54	$A^{\#}_5/B^{b}_5$	932.33	$A^{\#}_8/B^{b}_8$	7458.62
B_2	123.47	B_5	987.77	B_8	7902.13

Table I.1: Note to frequency mapping table