University of Southern Queensland

Faculty of Health, Engineering & Sciences

Harmony Analysis in A'Capella Singing

A dissertation submitted by

J. Oliver

in fulfilment of the requirements of

ENG4111/ENG4112 Research Project

towards the degree of

Bachelor of Electrical & Electronic Engineering

Submitted: October, 2019

Abstract

Speech production is made by the larynx and then modified by the articulators; this speech contains large amounts of useful information. Similar to speech, singing is made by the same method; albeit with a specific acoustic difference; singing contains rhythm and is usually of a higher intensity. Singing is almost always accompanied by musical instruments which generally makes detecting and separating voice difficult (Kim Hm 2012). A' Capella singing is known for singing without musical accompaniment, making it somewhat easier to retrieve vocal information.

The methods developed to detect information from speech are not new concepts and are commonly applied to almost every item in the average household. Singing processing adapts a large portion of these techniques to detect vocal information of singers including melody, language, emotion, harmony and pitch. The techniques used in speech and singing processing are catagorised into one of three categories:

1. Time Domain
2. Frequency Domain
3. Other Algorithms

This project will utilise an algorithm from each category; In particular, Average Magnitude Difference Function (AMDF), Cepstral Analysis and Linear Predictive Coding (LPC). AMDF is the result of taking the absolute value of a sample taken a time (k) and a delayed version of itself at (k-n). Its known to provide relatively good accuracy with low computational cost, however it is prone to variation in background noise (Hui, L et al 2006).

Cepstral Analysis is known for separating the convolved version of a signal into the source and voice tract components and provides fast computational speeds from utilising the

Fourier Transform and its Inverse. LPC provides a linear estimation of past values of a signal, the resulting predictor and error coefficients are utilised to develop the spectral envelope for pitch detection.

The project tested the algorithms against 11 tracks containing different harmonic content, each method was compared on their speed, accuracy, where applicable the number of notes correctly identified. All three algorithms gave relatively good results against single note tracks, with the LPC algorithms providing the most accurate results. When tested against multi-note tracks and pre-recorder singing tracks the AMDF and Cepstral Analysis methods performed poorly in terms of the accuracy and number of correctly identified notes. LPC method performed considerably better returning an average of 66.8% of notes correctly.

University of Southern Queensland

Faculty of Health, Engineering & Sciences

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Dean

Faculty of Health, Engineering & Sciences

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

J. Oliver

Acknowledgments

First and foremost I would like to thank my partner Amanda for the support she has provided me throughout this degree. I would like to thank my supervisor Mark Phythian for not only the idea of the project, also the support and guidance provided throughout the project. Finally, I would like to thank my university peers for the support they have provided throughout the year.

J. Oliver

Contents

List of Figures

List of Tables

Nomenclature

ACF Autocorrelation Function

ADF Adaptive Decorrelation Filtering

AMDF Average Mean Difference Function

ASDF Average Squared Difference Function

ASIC Application Specific Integrated Circuit

BSS Blind Source Separation

CASA Computational Auditory Scene Analysis

CIS Continuous Interleaved Sampling

CPLD Complex Programmable Logic Device

DCT Discrete Cosine Transform

DFT Discrete Fourier Transform

FFT Fast Fourier Transform

FPGA Field Programmable Gate Array

HPS Harmonic Product Spectrum

ICA Independent Component Analysis

LPA Linear Predictive Analysis

LPC Linear Predictive Coding

PAL Programmable Array Logic

PLA Programmable Logic Array

PROM Programmable Read Only Memory

SPLD Simple Programmable Logic Device

Chapter 1

Introduction

Speech analysis has touched the lives of almost every person in some way. It has been implemented in a variety of systems from simple everyday items such as children's toys to the more sophisticated, cochlear implants. Continued advances in this field are improving the way people connect with technology. Software such as Apple's Siri use sophisticated algorithms to interact with the user. Speech signal processing still contains many disadvantages, it tends to operate poorly in multiple speaker environments or areas that contain a substantial amount of background noise. This makes most speech signal processing algorithms perform poorly when used for singing analysis. The production of speech is made by the larynx or voice box, this sound source contains a spectrum of frequencies which are then modified using the articulators (contains the teeth, tongue and lips), these modifications which are made over time produce the vowels and consonants the are contained in speech (J. Wolfe et al 2013).

While singing uses the same human organs to develop the sound, singing is more rhythmic and consists of a higher intensity than speech. In general singers have longer vocalic segments along with improved resonance quality in their speaking voice than those who don't sing (Riley and Carrol 2016). Riley and Carrol go on to state that the "benefits from training of the voice include respiratory health, psychological well being through socialization, vocal stability through longevity and maintenance of cognitive function"; these benefits gained from training go on to produce improved vocal quality as well as efficiency of vocal power. Techniques such as SOVT and breath management are used to help make these improvements in singing clarity and quality. One of the most important

traits for a singer and arguably one of the hardest traits to learn is to be able to sing in pitch all the time. The singer needs to be able to review and learn when they are either singing too flat or too sharp (Brophy 2015)

In A' Capella singing, singers don't tune their voice to musical instruments, instead they tune their voice to each other. Generally speaking, the music scale is separated into 12 different tones of an equally tempered tuning system, this is known as the chromatic scale, it is commonly used in western music. The subset of these 12 tones is known as the diatonic scale and it consists of 7 tones which are said to be in key, while the other 5 are out of key (Brattico et al, 2006). With out musical accompaniment it makes it easier for the singers to tune to each other using a musical scale as the one described above, however they still lack any feedback excluding their own auditory perception.

1.1 Aims of this research project

The singing voice contains large amount of information including melody, language, emotion and pitch, currently there has been a moderate effort of research into the area of singing analysis or signal processing of music. Most deal with music that contains musical accompaniment and tends to separate the singing segments from the non-singing segments. Techniques that are widely employed have been adapted from standard speech processing algorithms for use with moderately good results (Muller, Ellis, Klapuri & Richard 2011). Most systems developed used in singing processing utilise LPC, Cepstral Analysis, HMM or variants of either to detect traits in the singing voice. The intended aim of the research project is to develop a system that will provide singers with visual feedback on singing information to help in the development of their singing ability. The algorithm will be analysing the following specific traits: Pitch, Melody, Harmony and timbre. The main objectives of this project are

1. Design an algorithm to analyse vocal cues such as pitch and quality.
2. Analyse two or more A' Capella tracks.
3. Determine the closeness of the detected signals.
4. Provide visual feedback to the user.

5. If time permits develop and implement a Hardware solution.

1.2 Overview of the Dissertation

This dissertation covers a range of areas dealing with signal processing of A' Capella and is arranged as follows:

Chapter 2: Literature Review – Describes previous research done in the area of speech and singing signal processing

Chapter 3: Methodology – Discusses the methods used for development of the prototype

Chapter 4: Results – Discusses the results of the initial prototype along with prototype improvements made throughout the development process

Chapter 5: Conclusions – An overview of the project, results and details any further work required for this area of research

Chapter 2

Literature Review

This chapter presents a thorough review of vocal production, the techniques developed for use in speech signal processing and FPGA implementation.

2.1 Vocal Production

Speech is generated using the lungs, trachea, larynx, pharynx, mouth and naval cavity. The airstream as it passes through the larynx, the airstream is modulated from the activity of the vocal cords, this modulated speech signal is then modified as it passes through the mouth and nose (Kovacevic 2017).

The phoneme is the basic linguistic element of a language which can be divided into the following categories, vowels, fricatives, plosives, affricates. (Kovacevic 2017). Vowels: are created when phonation occurs in the larynx and passes through the vocal track before radiation at the lips (Huckvale 2019). Consonants: are an articulation of the vocal tract causing the vocal tract to narrow or close completely, they are classified into three different categories:

1. Fricative: are created when the mouth is positioned in such a way to cause a partial block (narrowing of the air passage) of the air stream, causing the airstream to generate audible friction.

2. Plosives: Is a consonant that is created from a momentary occlusion of select parts

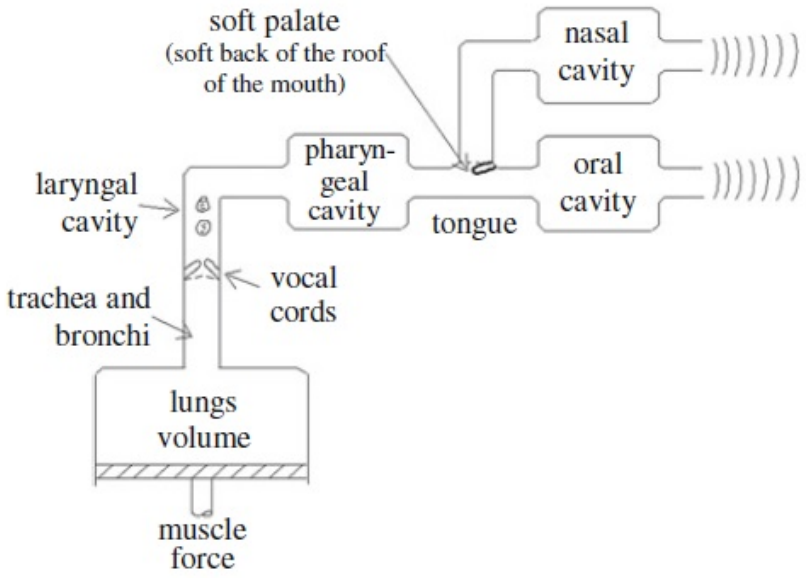

Figure 2.1: Block Diagram of the speech system (Kovacevic 2017)

of the oral cavity this differs from a fricative in that the occlusion is total rather than partial

3. Affricates: are a consonant that beings a stop or complete obstruction of air and concludes with an incomplete closure and friction sound. This obstruction of air occurs when a barrier is created within the mouth cavity causing a abrupt removal of the airstream followed by a narrowing of the mouth cavity in turn causing turbulent motion of the air which the vocal tract reacts too. (Kovacevic 2017, Britannica 2016).

When a person sings, production of the musical notes depends on the use of the lungs, larynx, head cavities and the tongue; these mechanisms act independently of each other and are coordinated to establish a vocal output. Singing distinguishes itself from speaking by the requirement of how the breath is expended to vibrate the vocal cords, it also requires more breath the louder, longer and higher a person sings (Britannica 2018).

2.2 Pitch Detection Methods

Many of the techniques used through out digital signal processing in speech have readily been applied to the field of singing process or music processing. Singing however possess specific acoustic and structural characteristics that distinguishes itself from speech or non-

musical signals. Singing is almost always accompanied by musical instruments, these are generally harmonic, broadband and coherent to the singing voice. This makes separating the singing voice from music more difficult than it would to remove speech (Kim Hm 2012). A'Capella singing however is considered as singing without instrumental accompaniment, in some case A'Capella groups use their voices to emulate instruments, while there may not be the need to remove the singers voice from accompaniment many of the techniques described during in the following sections can be adapted for use in this project.

2.2.1 Time Domain Algorithms

The time domain algorithms presented in this section process raw audio signals that have been digitized at various sampling rates. Sampling rates for most audio signals are 44100 samples per second. This section summarises commonly used time domain algorithms for pitch detection including autocorrelation (ACF), average magnitude difference function (AMDF) and Average Sqaure Difference Method (ASDF).

Autocorrelation Method

Autocorrelation method is a widely used approach in signal processing industry. It is employed in a variety of area within the signal processing realm including radar, pitch detection, and noise rejection (El-Ali, TS 2012). Autocorrelation is defined as the correlation or similarity between a signal and a delay (k) version of itself and is mathematically represented by equation 2.1

$$Rxx = \frac{1}{N} \sum_{n=-\infty}^{\infty} x(n)x(n-k) \tag{2.1}$$

Where n is the n_{th} sample of N length and k is the lag or delay of the signal being correlated. A signal of period 'p' will have a maximum, where the signal matches or has a high correlation with the delayed version of itself and k=0. In practice this is computed over a short time period of windowed data. This causes the frequencies within the window to appear stationary. This window is normally kept to a minimum of two periods of the waveform to correlate the waveform accurately (McLeod 2008). Windowing functions are generally applied to the autocorrelation to reduce leakage or edge effects as the windows transitions throughout the signal. Once autocorrelation of the window is performed, the

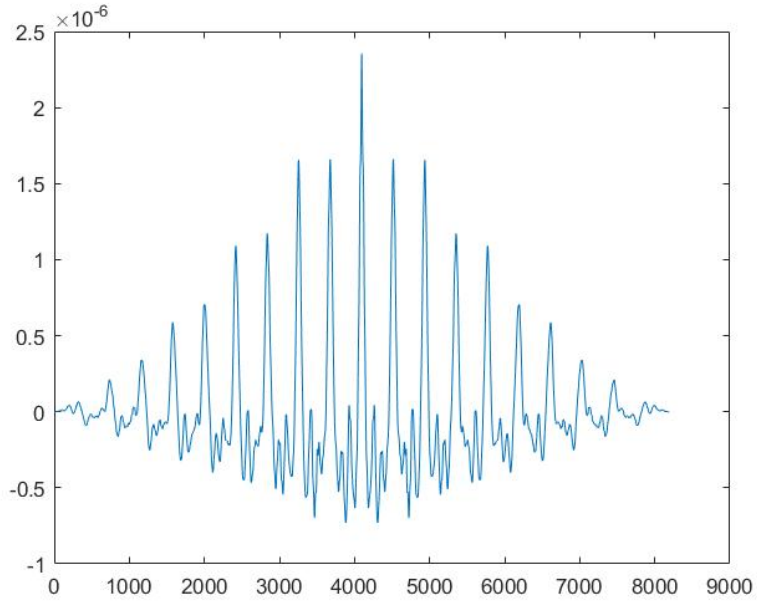

Figure 2.2: Autocorrelation of a randomly generated signal

maximum peak is found and is used as the fundamental period estimate. To determine the frequency this is value is divided by the sampling rate.

$$Frequency = \frac{P}{Fs} \tag{2.2}$$

Where Fs is the sampling frequency and P is the period to the location of the local maxima.

This algorithm performs well with periodic waveforms however it usually sees degraded performance with speech processing due to the changing periods, pitches and amplitudes of the speech waveform.

Center-clipping autocorrelation method is a method in which the autocorrelated waveform is clipped at a predetermined level. This helps to minimise the vocal tract transfer function and enhancing the vocal source (McLeod 2008). First the waveform is pre-processed where computation of the clipping level is done. The window is then center clipped and infinite peak clipped resulting in a signal of that either exceeds the clipping level or falls below the clipping level. If the waveform doesn't meet this criteria it is given a value of 0 (Rabiner, Cheng, Rosenberg & McGonegal 1976).

Average Magnitude Difference Function

Average Magnitude Difference Method of pitch detection is a variation of the autocorrelation analysis. It provides an estimate of pitch periods within voice signals based of a signal difference formed between the signal at time (k) and a delayed signal at (k-n). The absolute magnitude difference is taken, this value will be negative going at delays corresponding to the pitch periods of voiced speech. AMDF only needs subtraction, add and absolute operations for the detection of pitch. It provides accurate detection with low computational cost. AMDF is unfortunately prone to variations in background noise within the speech signal (Hui, L et al 2006).

$$X_m(m) = \frac{1}{N-m-1} \sum_{n=0}^{N-m-1} |S_w(n+m) - S_w(n)| \tag{2.3}$$

$$X_m(m) = \frac{1}{N-m-1} \sum_{n=0}^{N-m-1} |S_w(n) - S_w(n-m)| \tag{2.4}$$

Where $S_w(n)$ is the speech signal. N is the frame length of the signal and the range of m is 0 to N. AMDF will produce a notch in the output as opposed to a peak found with autocorrelation (Suma 2010). The pitch period is found by finding the minimum value from the output of AMDF and calculating the distance between the position and the origin point.

Average Squared Difference Function

The Squared difference function possess similarities to the AMDF covered earlier in this section. Instead of finding the absolute difference between the signal and a delayed version, SDF finds the squared difference between the two inputs. The squared difference results in the equation:

$$\sum_{n=1}^{t+k} (x(n) - x(n+k))^2 = 0 \tag{2.5}$$

This makes sure that the opposite values do not cancel each other when computing the SDF values. The minima occurs when k is a multiple of the period, this corresponds to the ACF at zero lag where the ACF produces a maximum (Kumaraswamy 2017, McLeod

2008). Like AMDF and ACF, the SDF or ASDF is required to have at least two cycles of the waveform present in the window segment. There is presently a modified version of the SDF function the provides a normalised output, known as YIN. YIN is defined as:

$$d'(n) = \begin{cases} 1, & \text{if } k = 0 \\ \frac{d_t(k)}{\frac{1}{k}\sum_{j=1}^{k} d_t(j)}, & \text{otherwise} \end{cases} \tag{2.6}$$

2.2.2 Frequency Domain Algorithms

Frequency domain methods of pitch detection use the property of periodic and quasi-periodic waveforms producing sharp peaks when transformed into the frequency domain. To transform a waveform from its time domain to its frequency domain a discrete Fourier transform (DFT) or Fast Fourier Transform (FFT) is used. Fourier's theorem states that any periodic function may be decomposed into an infinite series of sine and cosine functions Mathematically, this is defined as

$$x(t) = a_0 + \sum_{K=1}^{\infty} (a_k cos(K\Omega_0 t) + b_k sin(K\Omega_0 t)) \tag{2.7}$$

The coefficients are then found by integrating over one period of the waveform

$$a_0 = \frac{1}{t}\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} x(t)dt \tag{2.8}$$

$$a_0 = \frac{1}{t}\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} x(t)cos(K\Omega_0 t)dt \tag{2.9}$$

$$a_0 = \frac{1}{t}\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} x(t)sin(K\Omega_0 t)dt \tag{2.10}$$

If the requirement to have a periodic waveform of period (t) is now removed, the equation can be applied to convert any signal to the frequency domain. This is known as the Fourier Transform; it is defined as:

$$X(t) = \int_{-\infty}^{\infty} x(t)e^{-j\Omega_0 t} \tag{2.11}$$

The Fourier transform as defined above is used on continuous time systems, for discrete or sampled system the Fourier transform is known as discrete Fourier transform and can be defined as

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-jn\omega_k} \tag{2.12}$$

where

$$\omega_k = \frac{2\pi k}{N} = \text{frequency of the } k_t h \text{ sinusoid} \tag{2.13}$$

The DFT is a cyclic function and results in the mirror imaging of components sampled at time k. The DFT is computationally expensive to carry out, in most circumstances the function known as the Fast Fourier Transform is used to carry out the Fourier Transform of periodic and qausi-periodic signals. The Fast Fourier Transform reduces the amount of calculations required by reducing one large calculation in to several sequential smaller calculations (Leis, 2011).

Autocorrelation via FFT

Autocorrelation is generally considered a time domain method of pitch estimation, however using the properties of the Wiener-Khinchin equation combined with the FFT reviewed earlier, autocorrelation computation can be significantly reduced improving efficiency (McLeod, 2008).

The Wiener-Khinchin Theorem states that the power spectrum is the Fourier transform of the autocovariance function (Zbilut & Marwan 2008). The Wiener-Khinchin theorem is defined as:

$$Rxx = F[|s(t)|^2] = \int_{-\infty}^{\infty} |s(t)|^2 e^{-j2\pi\partial ft} \tag{2.14}$$

Where $F[|s(t)|^2]$ is the power spectrum and Rxx is the autocorrelation of the signal. This algorithm is relatively easy to implement in MATLAB, and very efficient to carry out. The resulting output of algorithm is the equivalent autocorrelation of the signal input.

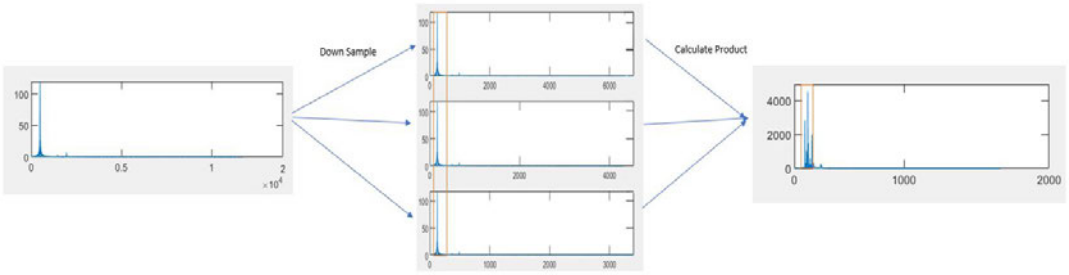

Figure 2.3: HPS Process

Harmonic Product Spectrum

Harmonic Product Spectrum compares the harmonics within a waveform to determine the fundamental frequency f_0 . To achieve this the signal is initially windowed to size N with a set hop size. The Fourier Transform is carried out on the windowed segment to transition to the frequency domain. The signal gets down sampled at multiples of the fundamental using a process known as re-sampling, once this process is complete the fundamental is found by finding the harmonics that corresponds to the peak located in the unmodified signal as seen in figure 2.3.

HPS estimates the best harmonic using the following equation

$$N(w) = \prod_{s=1}^{S} |M(ws)| \tag{2.15}$$

S is the number of harmonics to be considered. The pitch estimate is found using $\hat{N} = maxN(w)$; where w is the range of possible fundamental frequencies (Kodag, Patil & Gaikwad 2016) HPS allows for higher pitches to be tracked accurately and is computationally efficient. However, it does perform poorly with increasingly smaller windows, this is due to the accuracy in the bins, causing the harmonic peaks to become combined (McLeod 2008).

Cepstral Analysis

To accurately measure and analyse the components of singing, which can be considered a convolution of the excitation source and the voice tract components, they first need to be separated. Cepstral analysis separates such a signal into its two separate components without any knowledge of the signal being analysed. The output is known as the Cepstrum. The Cepstrum will represent the transformation of the signal with two important properties:

1. The representatives of the component signals will be separated in the Cepstrum
2. The representatives of the component signals will be linearly combined in the cepstrum (Rabiner 1978).

To carry out Cepstral analysis of a signal, first the waveform is windowed to the required size, then the log of the Fourier transform is carried out on the signal, before the inverse Fourier transform is carried out. This can be seen in figure 2.4.

The coefficients of Cepstal analysis are found with the following equation

$$c(\tau) = F^{-1}\{\log(|F\{x[n]\}|^2)\} \tag{2.16}$$

The independent parameter of the Cepstrum is called the Quefrency, which is a time domain parameter. The process of sperating these cepstral elements is known as liftering or linear filtering.

2.2.3 Other Algorithims

Previously this report looked at methods that can be distinctly seperated by the domain used to find pitch information from a signal. This section of the report will describe methods that are not easily sepearated into the frequency or time domain.

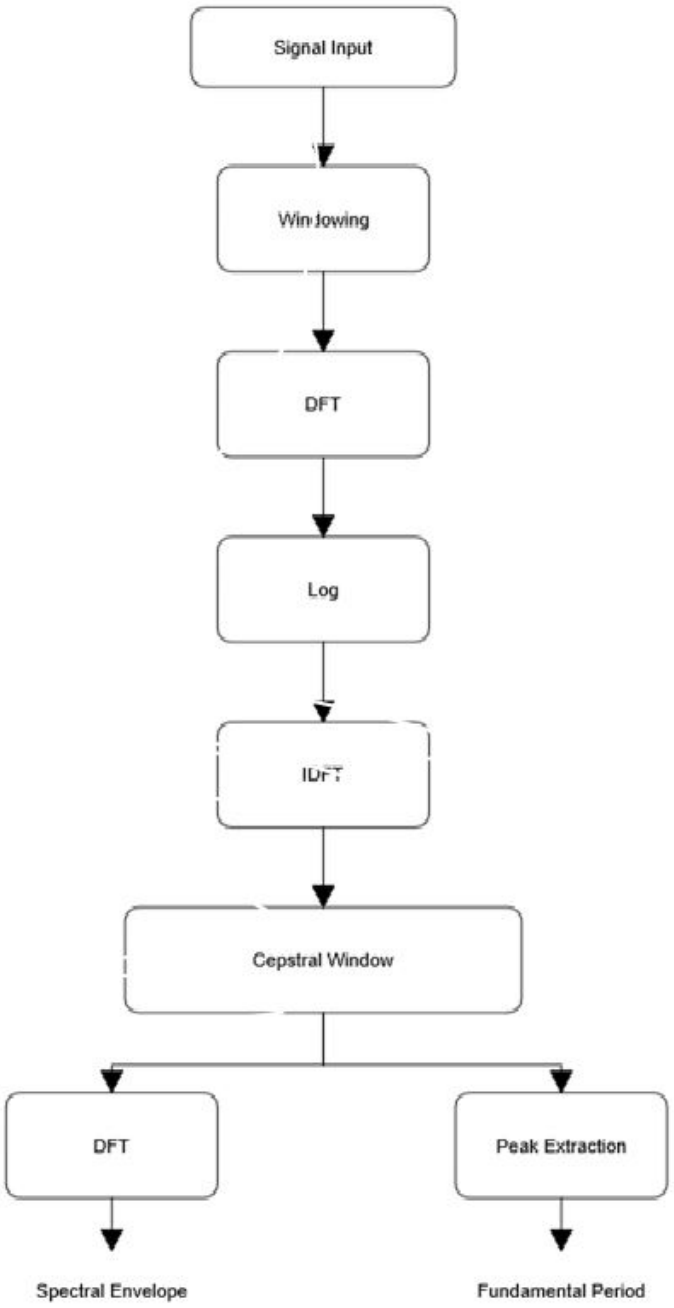

Figure 2.4: Cepstrum Analysis block diagram (Furui, 2001)

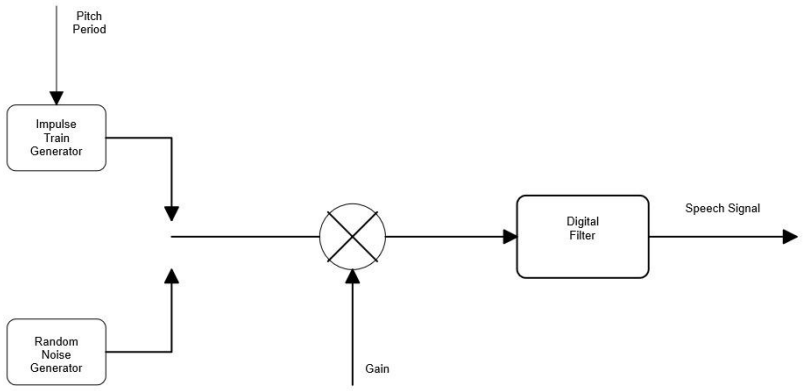

Figure 2.5: Speech Production block digram (Deller 1993)

Linear Prediction Analysis

The Linear Predication Analysis (LPA), commonly referred to as Linear Predictive Coding (LPC) technique was first described in the later years of the 1960s and has become a widely used tool in the analysis of speech parameters such as pitch, spectra and formants. It can provide accurate results with good computational performance.

In modelling a voiced system, figure 2.5 shows that voiced speech is simulated by an impulse train and unvoiced speech is simulated by a random noise generator. Noting that the parameters in the model are pitch period for voiced speech, gain, the digital filter and voiced/unvoiced classification, they are therefore represented by the equation:

$$A(z) = \frac{G}{1 - \sum_{k=1}^{n} \alpha_k Z^{-k}} \tag{2.17}$$

In LPA the speech is approximated as a linear combination of the past speech samples; predictor coefficients are developed by minimizing the sum of the squared differences between actual speech samples and linear predict samples. Due to the time varying nature of speech signals, the mean squared prediction error must be completed in a short period of time by utilising an N length hamming window or similar (Rabiner 1978).

Since:

$$e(n) = s(n) - \widetilde{s(n)} = s(n) \sum_{k=1}^{p} a_k s(n-k) = Gu(n) \tag{2.18}$$

$e(n)$ would result in small impulses for voiced speech segments and if excited by non-time varying white noise then it would result in the mean squared prediction error to be identical to the coefficients of $s(n)$

Where:

$$s(n) = \sum_{k=1}^{p} a_k s(n-k) = Gu(n) \tag{2.19}$$

This allows for the set of model parameters to be solved by a set of linear equations, which give way to an accurate representation of the speech signal.

$$E_n = \sum_{m} e - n^2(m) \tag{2.20}$$

$$= \sum_{m} (S_n(m) - S(M))^2 \tag{2.21}$$

$$= \sum_{m} [S_n(m) - \sum_{k=1}^{p} a_k S_n(m-k)]^2 \tag{2.22}$$

The limits for the linear equations are specified by one of two basic methods, Autocorrelation and Covariance methods.

The Autocorrelation Method for determining the limits

The autocorrelation method assumes that the waveform is segmented, where all values of $S_n(m)$ are zero outside the limits of the equation.

$$S_n(m) = S(m+n)w(m) \tag{2.23}$$

Where $w(m)$ is a finite length window such as a Hamming window. It can be seen by using a Hamming window of N length that the prediction error $e_n(m)$ will only be non-zero for the interval $0 \leq m \leq N - 1 + p$ for a p_{th} order system (Rabiner 1978).

Therefore E_n can be expressed as

$$E_n = \sum_{m=0}^{N+p-1} e_n^2(m) \tag{2.24}$$

The Covariance Method for determining the limits The covariance method approach to defining the limits of an LPA is to fix the interval over which the mean squared error is computed and then consider any effects on the computation of $\phi(i,k)$.

Where:

$$\phi_n(i,k) = \sum_{m=-k}^{N-k-1} S_n(m)S_n(m+k-i) \tag{2.25}$$

This function is a crosscorrelation between two similar signals of finite length segments of a speech wave (Rabiner 1978), which produces a set of equations to be solved in the form of:

$$\sum_{k=1}^{p} \alpha\phi_n(i,0) \quad i = 1,2,...p \tag{2.26}$$

Both methods produce a solution in the form of a linear vector matrix problem; Ax=b, where A is a square matrix and x is the vector which needs to be found.

Solution to the Autocorrelation method The Levison-Durbin alogrithm is a recursive solution developed to solve for the autocorrelation equations, the solution is developed from the lower order models starting at the 0th order predictor model or no predictor model (Deller 1993). This algorithm is considered one of the most efficient solutions to the autocorrelation problem. It is mathematically stated as:

$$\sum_{k=1}^{p} \alpha\phi_n(i,0) \quad i = 1,2,...p \tag{2.27}$$

This group of equations is resolved recursively for I = 1,2, to n_{th} iteration with the final solution give as

$$E^{(0)} = R(0) \tag{2.28}$$

$$k_i = \frac{(R(i) - \sum_{j=1}^{i-1} \alpha_j^{(i-1)} R(i-j))}{E^{(i-1)}} \tag{2.29}$$

$$\alpha_j^{(i)} = k_i \tag{2.30}$$

$$E^{(i)} = (1 - k_i^2)E^{(i-1)} \tag{2.31}$$

This group of equations is resolved recursively for $I = 1, 2, \cdots p_{th}$ iteration with the final solution give as

$$\alpha_j = \alpha_j^{(p)} \quad 1 \leq j \leq i-1 \tag{2.32}$$

If the autocorrelation coefficients are replaced with normalised coefficients, it can be shown that the system polynomial roots are located inside the unit circle, which guarantees the stability of the autocorrelation system (Rabiner 1978).

Solution to the Covariance Method In the covariance method the equations are in the form of equation (2.26). The covariance matrix is not a Toeplitz matrix is however still symmetrical matrix of the form $\phi\alpha = \psi$ where ϕ is a positive definite symmetric matrix and α is the column vector to be found (Rabiner 1978).

Solving the covariance equations is done by the decomposition of ϕ into lower and upper triangles L and U, therefore $\phi = LU$, once decomposition has occurred the covariance method can be sequentially solved by solving the equations:

$$\phi_s(m) = L\delta L^T \tag{2.33}$$

$$\Phi_s(m) = CC^T \tag{2.34}$$

Wavelets

Wavelets is a function that meets two criterion the first, the integral of the function of x is 0 and the second; the square of the function has an integral of 1. Mathematically these can both be represented by the following equations

$$\int_{-\infty}^{\infty} \psi(x)dx = 0 \tag{2.35}$$

$$\int_{-\infty}^{\infty} \psi(x)dx = 1 \tag{2.36}$$

This shows that a wavelet function has an equal area above and below zero, and that the function approaches zero and positive and negative infinity (*Wavelets* 2014). Wavelets

are generated from a single wavelet known as a mother wavelet by scaling and translating operations. The general wavelet function is defined by:

$$\psi_{s,t}(x) \equiv \frac{1}{\sqrt{s}\psi(\frac{(x-t)}{s})} \tag{2.37}$$

The function varies in both time and scale and may be translated and shifted by 's' and 't' where necessary. Where t is the time series $x(t)$ and s is the scale, which is similar to the frequency for the FFT function. The scaling provides the advantages of large- or small-scale transform windows over the same time series. There are variations of wavelet functions commonly used throughout signal processing including the Morlet wavelet used for short, high frequency transients, and the Mexican Hat wavelet used when temporal resolution is important. Other variations of note include Haar wavelet, Meyer Wavelet and S8 Symment wavelet (Shukla 2013) these variations are shown in figure 2.6

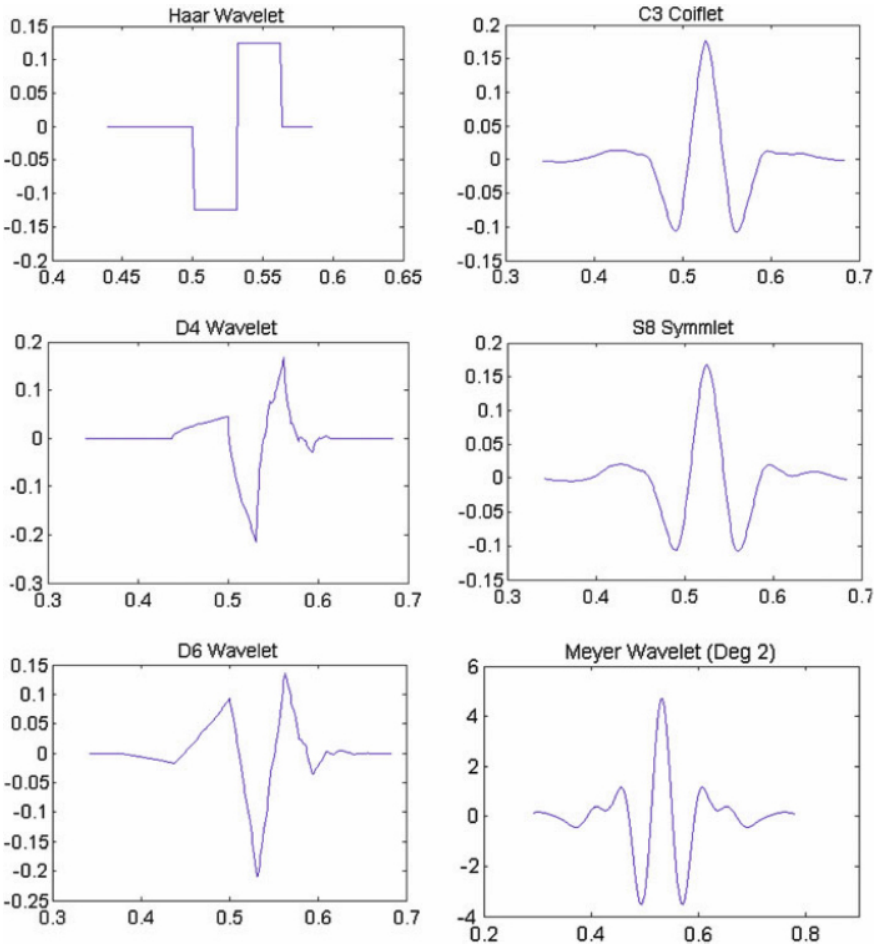

Figure 2.6: Wavelet Variations (Shukla 2013)

The Continuous Wavelet Transform (CWT) maps a one-dimensional function to a two-dimensional function of two continuous variables. The CWT can be defined by

$$CWT(s,t) = \int_{-\infty}^{\infty} x(u)\psi_{s,t}(u)du \tag{2.38}$$

Therefore the Discretised Wavelet Transform (DWT) is given by

$$DWT(j,k) = \int x(t)\psi_{j,k}(t)dt \tag{2.39}$$

Since the DWT generates a large amount of data for every scale, the scales are based on a power of two also known as the dyadic scales. Shukla, kk 2013, shows that the computation of the DWT can be performed using Finite Impulse Response (FIR) filters. It results in a recursive transform and the outputs are used to compute the wavelet coefficients at the next octave of resolution.

2.3 Signal Separation

2.3.1 Signal Separation from accompaniment

While A'Capella is generally singing without accompaniment, most research is done in the area of singing analysis with accompaniment, removing and analysing voice from broad band instrumental backgrounds. Hu and Wangs propose an algorithm, where the target pitch can be estimated using a few harmonics of the target signal, this estimate is then used separate the target speech, consider the harmonicity and temporal continuity and improve the accuracy of the results iteratively. This system shown to give better results than current CASA systems for auditory analysis (Guoning & Deliang 2010).

Chao-Ling Hsu and DeLain Wang expanded on Hu and Wangs tandem algorithm; their improved tandem algorithm uses a trend estimation algorithm that first estimates the pitch ranges of the singers. The estimation is used to limit the range in the tandem algorithm to get an accurate initial estimate of the singer's pitch. The voice is then separated using the estimated pitch and improved each iteration. Hsu and Wang, also incorporated a post processing to deal with sequential grouping problems (Chao-Ling Hsu, Deliang Wang, Jang & Ke Hu 2012). Other existing methods can be categorised into three different areas depending on the methods that are used; Model based, Pitch based, and Spectrogram factorisation based. Spectrogram factorisation methods use the redundancy of the singing voice and music component by decomposing the input signal into a pool of repetitive components. Each of which is then assigned to a sound source. Model based methods learn a set of music only sections; spectra of the voice are then learned from the sound mixture by the comparing music only sections. Pitch based methods extract vocal

pitch contours to separate the singing voice (Chao-Ling Hsu et al. 2012).

Each method has their own limitations, pitch-based methods tend to have fewer limitations than that of modelling or spectrogram factorisation methods. This is due to only requiring pitch contours as the cue for separation of the voice signal. Their main draw back is its interdependency between pitch estimation and pitch-based separation. These systems have been previously implemented in areas that have broad band background noise produced from musical instruments. Separating two speakers or singers voice promotes its own difficulties particularly when they are matched closely in pitch and harmony, currently there is limited progress in the area of separating multiple speakers.

K. Min, D. Chen, S. Li and C. Jones present a system for improving separation techniques of voices recorded on a single channel. The system (a multi-step pitch detection scheme) includes auto-correlation functions, AMDF, and look forward and backward checking schemes. The pitch detection scheme is used to determine the frame size to have sufficient length to cover a multiple number of pitch periods for each speech signal. Speech detection is applied to determine if one or more speakers exists in each frame. Then the desired spectral components are selected and reconstructed. Filtering is used to reduce any high frequency noise as part of the post processing scheme (Min et al. 1988).

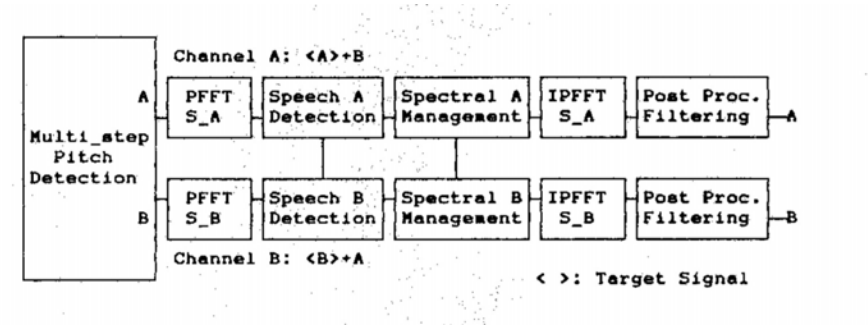

Figure 2.7: Parallel processing scheme (Min et al. 1988)

2.3.2 Multiple Speaker Signal Separation

Speech analysis software is generally utilised for single person environments, when faced with multiple speakers, performance of these systems tends to degrade to a point where the system becomes unusable. To account for such environments a speech or signal separation stage is essential for continued performance of speech analysis systems.

C. Kwan et al conducted a study to analyse some of the current methods for signal separation in noisy environments. Their study looked at three algorithms, examined their performance and formulated guidelines on which algorithm is appropriate for different applications. The three different methods examined are, blind source separation (BSS) based on independent component analysis (ICA), BSS based on Adaptive Decorrelation Filtering (ADF) and Beamforming algorithms (Kwan et al. 2008).

The signals being measured in a real time environment are convolved with acoustic path impulse responses, these signals by their nature are extremely difficult to separate. BSS based on ICA resulted in the signals being separated effectively, Kwan et al notes that the algorithm can be done with multiple signal sources, however it must be done pairwisely and needs an alignment algorithm to be implemented to fuse the separated signals together. The results of signal separation of BSS based on ICA is shown in figure 2.8, the voice signal is now easily recognisable once removed from the background noise.

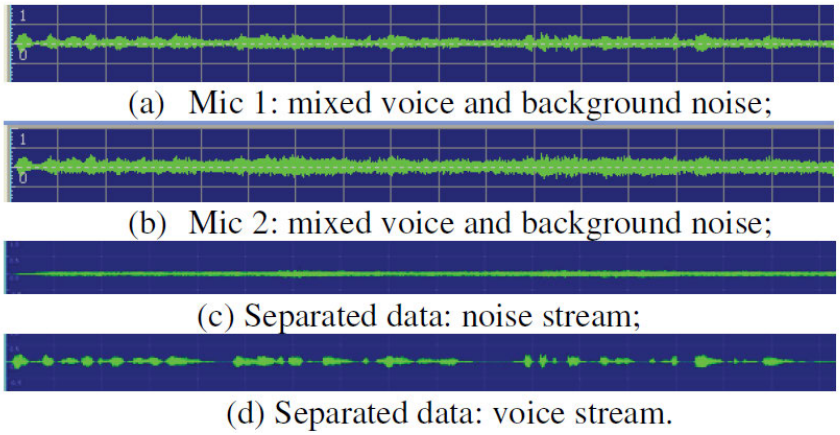

Figure 2.8: ICA Voice separation(Kwan et al. 2008)

BSS based on ICA was implanted in the C software language, running on a Windows environment. It consisted of 3 modules, the input module, processing module (In the frequency domain) and the output module which displays the results (Kwan et al. 2008). BSS based on ADF is an adaptive technique, with the convolutive nature of speech signals. BSS algorithms are required to estimate de-mixing filters, therefor the speech separation result is dependent on the reverberation level, source microphone distance and spatial separation of sources. Figures 2.9 and 2.10 shows Kwans mixing and separation block diagram and the processing steps required of the ADF algorithm. Kwan et al found similar

results to BSS based on ICA, however the computation of BSS based on ADF takes longer and the ADF based algorithm can be repeated for improved results. In similar structure to the ICA, the ADF based algorithm used three modules, input, processing and output to separate voices. Figure 2.11 shows the results from the ADF based algorithm, two speakers can clearly be identified.

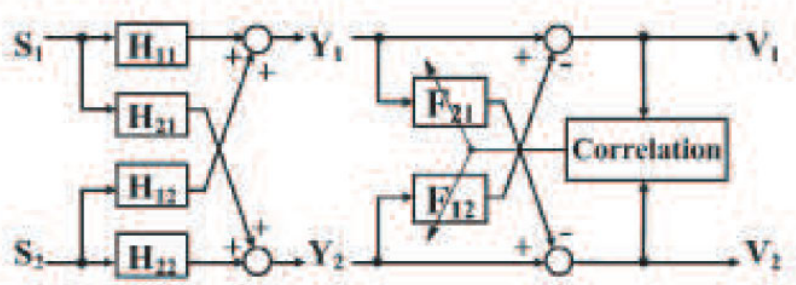

Figure 2.9: Block Diagram Source Mixing and Separation (Kwan et al. 2008).

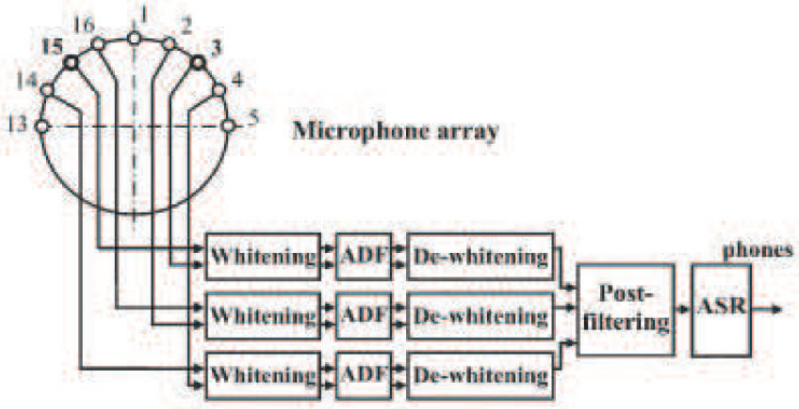

Figure 2.10: ADF Processing steps (Kwan et al. 2008)

.

The Beamforming technique trialled by C.Kwan's team implemented three beamformers. A Delay and Sum beamformer, in which they compute the delay time for a given speaker location to microphone location. Each speech signal is then divided into frequency bins and processed by a weighting function and an inverse FFT function to obtain time domain data. The DS beamformer was implemented in both MATLAB and C language. Next was a Super-directive Beamformer otherwise known as Minimum Variance Distortion-less Response Filter; this system was developed to alleviate the wide beam and lack of control of the DS beamforming technology. The Super-directive beamformer surppresses side lobe energy. In this system the speaker locations are known and parameters a pre calculated, when speech is detected, the system beamforms to each location and selects the signal

with the highest energy.

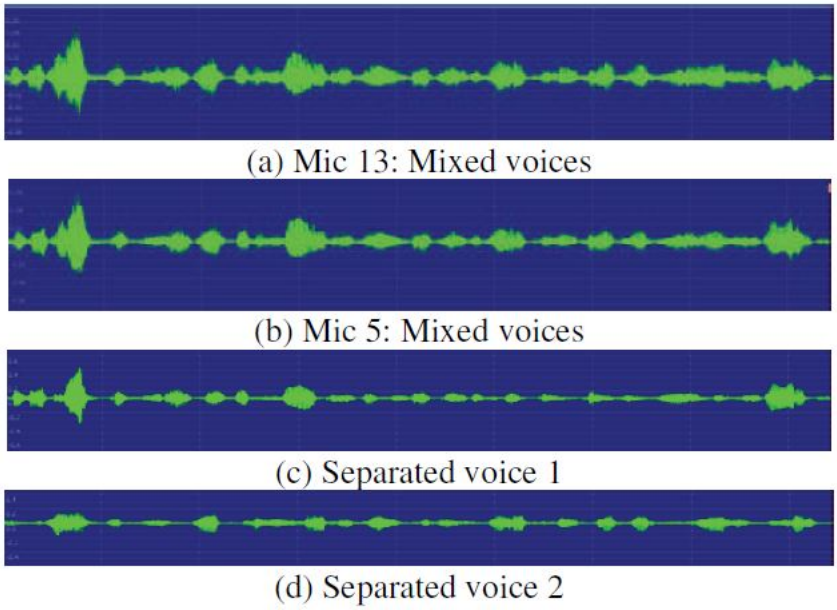

Figure 2.11: ADF Voice separation(Kwan et al. 2008)

The final beamforming implementation is known as Interference Rejection Beamformer, in this system the interference signal location needs to be known, once known, zero gain is placed in that direction eliminating any background noise. Table 2.1 shows the key features of each beamforming technology and comparisons to BSS based ICA and BSS based ADF technologies.

2.4 Programmable Logic Devices

Introduced in the late 1970s, the PLD; is a reconfigurable electronic component that has no specific function at the time of manufacture and only needs to be programmed to perform a specific task. The PLD family consists SPLDS, CPLDS and FPGAs. The PLD family has several advantages over traditional SSI/MSI IC systems, namely, less board space, faster, lower power requirement, cost and reliability (Rao 2016).

Table 2.1: Comparison of Signal separation algorithims (Guan 2017).

System	**Advantages**	**Disadvantages**
BSS based ICA	Fast, no need of speaker direction and location information	May not yield good results for many situations, dependent on speaker characteristics
BSS based ADF	Good performance, no speaker location needed	Long computation time, dependent on speaker characteristics
Beamforming	Fast, excellent performance, independent of speech characteristics	Needs speaker location and direction
Beamforming	**Type**	**Key Features**
	DS	Spatial matched filter Maximize bemformer output SNR No sidelobe control, Wide beams hence low resolving power between close sources
	Super-Directive	Minimize energy from other directions Narrow beams, Good resolving power between close by sources
	Interference Rejection	Good rejection of directional interferences, Narraow Beams

2.4.1 SPLDS

SPLDS come in several configurations, PROM, PAL and PLA. All these configurations utilise AND arrays and OR array setups. In PROM, the OR array is programmable, PAL the AND array is programmable and in PLA both the arrays are programmable. SPLDS have a costly disadvantage, being that they are only one time programmable and can only be programmed for combinational logic.

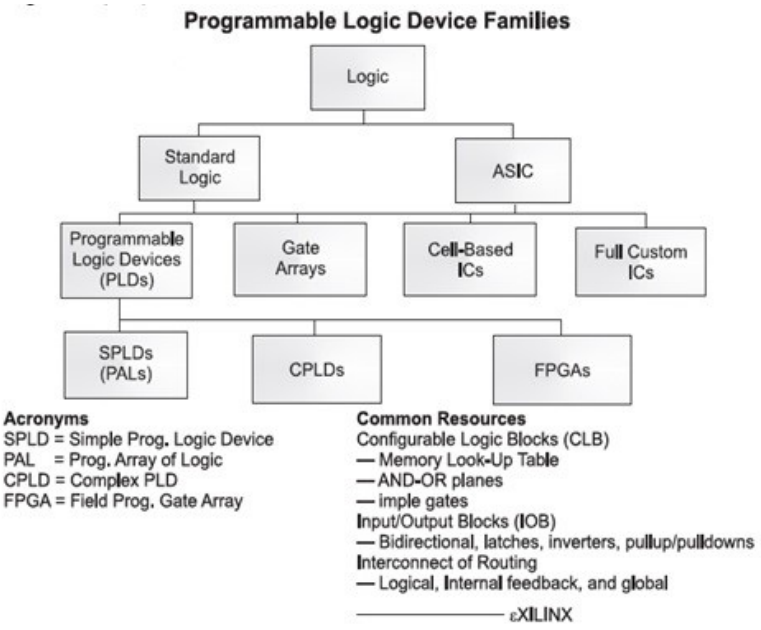

Figure 2.12: Logic Families (Rao 2016).

2.4.2 CPLDS

Complex programmable logic devices are used for more complex tasks than their SPLDS counterparts. A CPLD consist of multiple circuit blocks on a single chip with internal resources to connect each block, with more inputs and outputs than the SPLD setups. Each block is similar to a PLA or PAL these like blocks are connected to wire interconnections and to a sub circuit I/O block. They can perform a variety of functions that a SPLD cannot, however they are slow in operation compared to a SPLD. A CPLD is show below in figure 2.12.

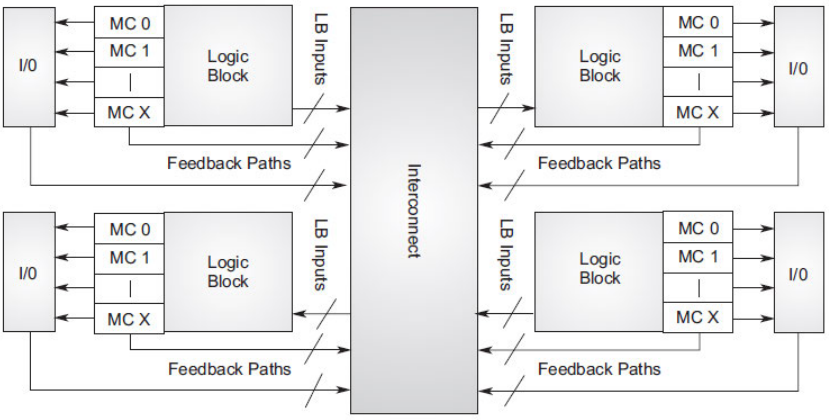

Figure 2.13: CPLD Architecture (Rao 2016).

2.4.3 FPGA

Field Programmable gate arrays have become an important logic device since their implementation in the mid 1980's, they have good performance and have the potential to use less power than other programmable logic devices if used correctly (Leong 2008).

A FPGA array is an integrated circuit configurable by the consumer or a designer after manufacturer. They are used in several various industries from communications, office automation through to defence. There are several architectures of FPGA, the two primary variants are coarse grained and fine grained. Coarse grained have fewer larger components than fine grained systems. Early and simpler versions of FPGAs generally consisted of logic blocks and registers, called configurable logic blocks (CLBs) and a ring of programmable input/output blocks providing interface to the user or outside world.

As advances were made, FPGAs began to get additional resources such as a clock managers, multipliers, RAM, Hard and Soft processors, DSPs and transceivers (Rao 2016). Blocks such as the DSP added in later model give the FPGA support for logical operations, shifting, addition and complex multiplication. This has further been improved through multiple word-length support and cascadability (Leong 2008). Implementing these commonly used blocks increases the speed of the FPGA while reducing the power usage of the system.

The decision to implement FPGA system over other systems such as an application specific integrated circuit, will be made considering many factors such as cost, speed, software, hardware and power. One of the main advantages of an FPGA based system over others is that they have a lower cost when compared to custom systems and provides the flexibility to be used in several areas. ASICs are a system that are built around a specific application, which has some obvious advantages such as size and operating speed (generally in the order of 20 times faster), they also don't include any unnecessary gates. This make ASICs an obvious choice for high speed applications.

The other major competitor to FPGA technology is the microcontroller; these systems are constructed from a CPU, memory, general and special purpose controllers and interfaces. They tend to have limited access to ROM, EPROM, Flash ROM/RAM and I/O ports. Microcontrollers also need a program loaded on to the system, which can take considerable amount of time to program and debug. Industries that utilise the microcontroller include

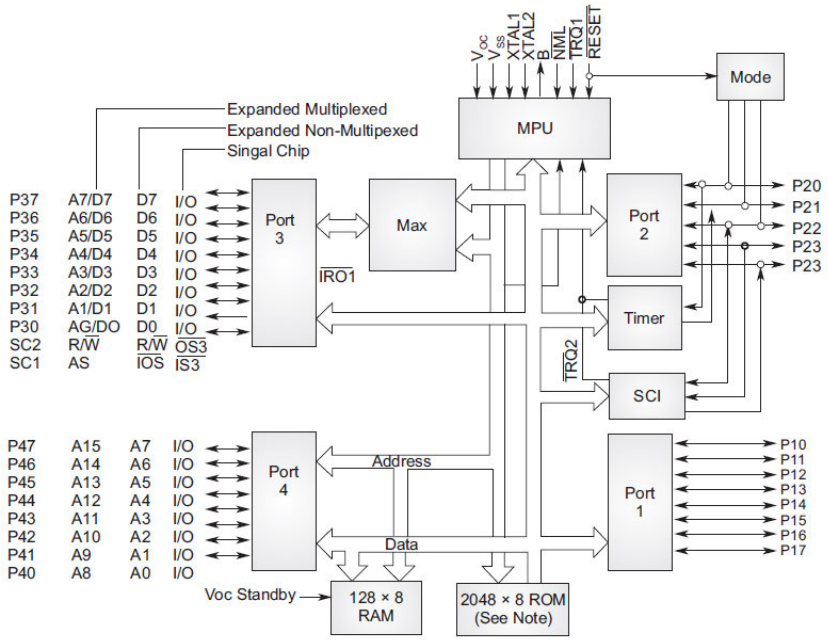

Figure 2.14: Microcontroller Architecture (Rao 2016).

the automotive, communication, military and medical for applications like automatic breaking or theft deterrent systems.

Table 2.2 provides a snapshot of the differences between FPGA, Microcontroller and ASIC systems. Each of these options are suitable for signal processing of audio or speech signals.

Table 2.2: Comparison between systems (Rao 2016).

System	MICROCONTROLLER	FPGA	ASIC
Structure	Custom Builtcomputers with limited resources	Interconnected logicblocks interconnected electrically	Full Custom System
POWER USAGE	Low	moderate	
COST	Low	Low - Moderate	Lower for large number of units
SOFTWARE	Yes	Mostly Hardwired	Yes
APPLICATIONS	Some Real time application	Most real time applications	Real time (Cars,voice recorders, mobile phones etc)
SETUP TIME	Low-Med	High	
SPEED	Low	Med- High	High

2.5 Speech Signal Processing with FPGA Architecture

As technology improves signal processing of speech is finding its way into everyday life; computer systems provide good results for implementing speech processing methods, if the user is willing to accept some latency. However, it is not always the most convenient option, economically viable or suitable for the application. FPGA architecture provides the developer with the ability to configure the system after manufacturer.

While FPGAs haven't been heavily used for processing of singing, they have found use in speech processing and speech recognition applications; Melnikoff, Quigley and Russell used FPGA architecture for such a system. Speech recognition is a computationally demanding task, requiring the system to convert and continually process real time data. Melnikoff et al implemented an FPGA on the decoder for discrete and continuous Hidden Markov Models and successfully processed data nearly 5000 times faster than real time.

Their system design uses a pre-processing stage, which takes the speech input extracts features and observation vectors required for the speech recognition. The second stage of the system is for recognition and decoding performed using the hidden Markov models(Melnikoff, Quigley & Russell 2002). Although the system uses an FPGA structure it still utilises a PC as a host system. The hardware consist of a Xilinx Virtex XCV1000 FPGA in conjunction with a Celoxica's RC1000-PP development board, an RC1000 PCI card, and 8 Mb of RAM accessible by both the FPGA and the Pentium III 450 Mhz host computer (Melnikoff et al. 2002).

Melnikoff found that an FPGA in conjunction with a Pentium III PC was highly capable of implementing signal processing for speech recognition applications far faster than the software equivalent. There was a trade off between the speed at which the system can process information and the accuracy of the processed information.

Table 2.3: Experiment Results (J.L. Gomez Cipriano et al 2002).

	Hardware	**Software**
Stage	*Time (us)*	*Time (us)*
Parameter Extractionand Pre-Processing	2085	110000
Vector Quantization	173	440000
Recognition withViterbi decoding	375	50000

A similar system proposed by Jose L Gomez Cipriano, was implemented with functional blocks for a portable speech recognition system using FPGAs. It contained a pre-processing system, a Mel-Cepstra Parameters extraction system and a Viterbi processor. The pre-processing system carries out pre-emphasis, frame separation and windowing of the signal. The Mel-Cepstra system caries out the FFT, processes it through a bandpass filter and carries out the discrete cosine transform. The output is passed through to the Viterbi processor via a vector quantization stage. Left and Right HMM are then outputted (J.L. Gomez Cipriano et al 2002).

The functions in Gomez Ciprianos design were implemented using Maxplus II in conjunction with MATLAB and the C software language. The results from J.L. Gomez Ciprianos study are listed in table 2.3. It clearly shows that implementing speech processing with hardware takes a fraction of the processing time compared to its software counterpart.

Other projects of note were developed by Amit.B and Jun Xu; Amit used a FPGA to implement an efficient speaker verification system. The system was developed around a 10_{th} order LPC algorithm. To implement the system in hardware they utilised the efficient Levison Durbin algorithm to determine the LPC coefficients (Amit S B 2018). These coefficients are then compared to the stored data vectors to verify the speaker. Implementing the algorithm in hardware required Amit's team developed five main hardware sections, segmentation logic, Autocorrelation Logic, Reflection coefficients Logic, LPC coefficients logic and LPC Vector Matching logic. The signal was sampled using a 8Khz sampling frequency, then segmented into 160 samples (Amit S B 2018).

The system was implemented on a VIRTEX-7 and a KINTEX-7 FGPA, the VIRTEX -7 implementation used a little over 6% of the on board memory and 50% of the bonded

I/O. While the KINTEX-78.71 used 10% of the on board memory and 70% of the bonded I/O(Amit S B 2018).

Jum Xu's dystem was implemented on a Virtex E V2000 FPGA. Like Amit's implementation this system used the Levison Durbin algorithm to obtain the LPC Coefficients. The system was a 10_{th} order system and used 84.66% of the CLB slices and 44.06% of the available I/O (Jun 2005).

2.6 Ethical Considerations

This project focuses around analysing vocal traits of A' Capella singers; with that comes several ethical considerations that need to be accounted for when analysing tracks. Speech or singing signals contains a large amount of information including pitch, emotion, language and emotion and in some cases can be used to identify the individual. According to the Queensland Government personal information must meet two criteria

1. It must be about the individual
2. The individuals identity must be reasonably ascertainable from the information or opinion

(Government 2019)

In most states of Australia, it is illegal to record or reproduce a recording of a person without consent. Some states, such as Queensland, it is considered legal to record a conversation you're involved in without consent (Queensland 2019). Most of these acts deal with the recording conversations between people and not singing.

It is still highly important that through out this project no breaches of privacy have been made and consent must be sought from the persons on the track. There-for, the research conducted throughout this project will be done using audio tracks that have been provided by and with consent from the supervisor of this project and no recordings will be made. The legal acts that pertain to this research are:

1. The Invasion of Privacy Act 1971 – QLD

2. Information Privacy Act 2009 – QLD

Since this report will only be showing results of the analysed tracks, with limited information related to speech or singing traits of the people singing on the track; It is highly unlikely that personnel information about the people singing on the track will be attainable from this report.

Furthermore, the Institute of Engineers Australia requires Engineers, including students, working around Australia to strictly follow a Code of Ethics. This code is published freely on the Engineers Australia webpage. As such, while conducting this research the members of this project are required to demonstrate Integrity, practice competently, exercise leadership and promote sustainability.

Chapter 3

Methodology

This chapter will detail the methodology of this research project. The chapter will outline the following phases of the research project: Software design, Software Test and Evaluation and Hardware Design.

3.1 Software Design

Generally, singing analysis techniques have been derived for traditional speech analysis techniques. Singing possesses specific acoustic characteristics that separate it from speech. Typically singing consists of pitch, timbre, rhythm and harmony. This project will be developed based upon Linear Predictive coding, Average Magnitude Difference Function and Cepstral Analysis techniques.

Each algorithm has been coded using the MATLAB development environment. Three MATLAB scripts were developed with common sub-functions called by each script. The sub-functions perform the tasks that are not directly related to the algorithm being tested, they are described later in this chapter.

Figure 3.1 depicts a simplified version of the overall system. It is separated into three main components, Pre-processing, Signal Analysis and Post-processing. The Pre-processing stage contains the functions to prepare the track to be analysed by one of the algorithms. Signal Analysis section calls the applicable algorithm to analyse the window and the Post-processing section calls the functions required to extract and display pertinent information

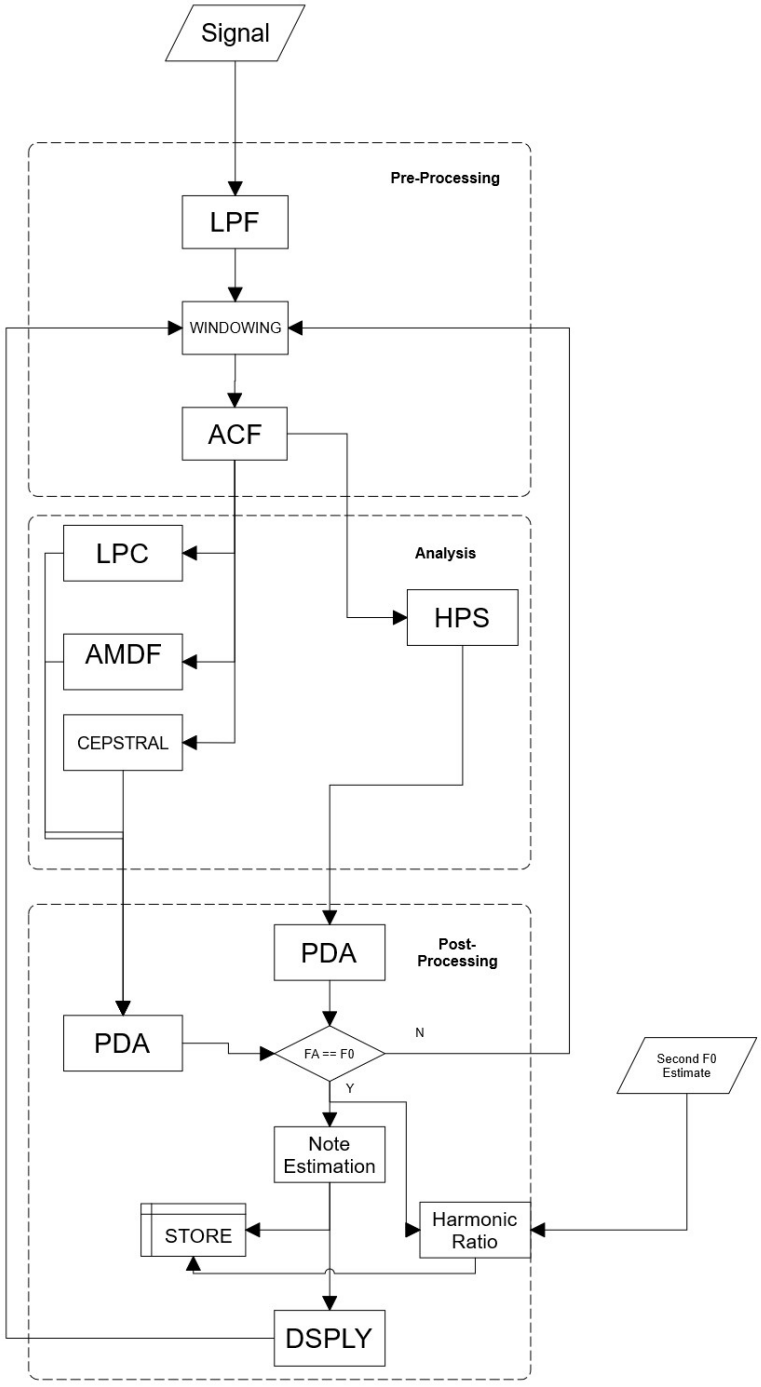

Figure 3.1: System Block Diagram

to the user.

Once a track is loaded into the MATLAB script, the audio information is passed to the pre-processing stage, where it is filtered, windowed and its periodicity emphasised. It is then passed to the Signal Analysis section where the HPS algorithm and either LPC, Cepstral Analysis or AMDF are used to analyse the track. The signal is then processed by the post processing stage where the notes are detected and displayed. Once the information has been displayed, the window is incremented to the next window and the process repeats until then end of the track is reached. All the relevant information is saved to an Excel spread once the processing of the track is completed.

3.1.1 Pre-Processing

Pre-Emphases and Low Pass Filter

The pre-emphasis and low pass filter are primarily used to balance the spectrum and remove high frequency noise from the signal. The signal is processed through the pre-emphasis filter, where the signal is combined with an altered version of itself. This follows the form of:

$$y(n) = x(n) - \alpha x(n-1) \tag{3.1}$$

Where $x(n)$ is the signal being filtered and α is a filter coefficient of approximately 0.94 (Vergin, R and O'Shaughnessy, D 1995). Since speech naturally attenuates per decade, passing the signal through the pre-emphasis filters results in the signal having a more balanced spectrum.

The low pass filter is a finite impulse response filter chosen for its inherent stability and ease of implementation within MATLAB.
The FIR filter is of the form:

$$H(n) = \sum_{n=-\infty}^{\infty} h_d(\lambda) e^{jn\lambda} \tag{3.2}$$

Where,

λ = Normalised cutoff frequency = $\frac{Wc\pi}{Fs}$ $N = 31$ $m = n - \frac{(N-1)}{2}$

Substituting our required values into the general equation for a FIR low pass filter, (Wc

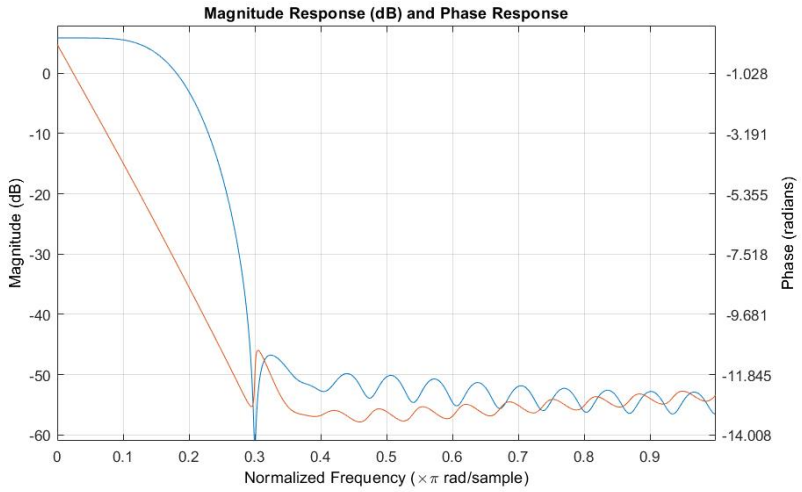

Figure 3.2: LPF: Frequency Response

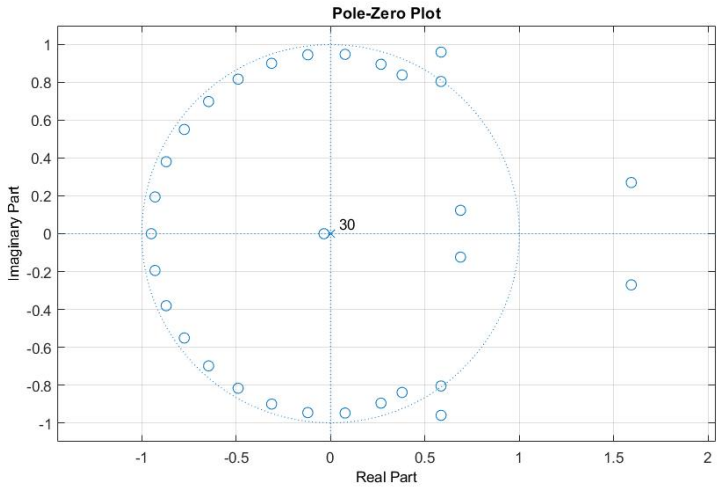

Figure 3.3: LPF: Zero Pole Plot

= 4kHz, Fs = 44100, N = 31), results in equation

$$h(n) = \frac{1}{2\pi} \int_{\frac{-4\pi}{44.1}}^{\frac{4\pi}{44.1}} H_d(\frac{4\pi}{44.1})[cos(m\frac{4\pi}{44.1}) + jsin(m\frac{4\pi}{44.1})]d\lambda \tag{3.3}$$

Solving the integration results in the equation describing the FIR low pass filter used for the singing analysis software.

$$h(n) = \frac{sin(m\frac{4\pi}{44.1})}{\pi m} \tag{3.4}$$

The filters response was improved by the addition of a hamming window to reduce the ripple through the stop band. A 4kHz cut off frequency was chosen to ensure all intelligible information from the singers can be analysed in the program.

The required information for low pass filter operation includes cut-off frequency (Wc), the signal (y), Sampling frequency (Fs) and the filter order(m). The function will then pass the filtered result back to the main equation for further analysis of the signal.

Windowing

The signal is windowed using a Hamming Function, into short segments with a 25% overlap on the previous window. The window size is a trade-off between the resolution in the time domain to that of the frequency domain, as well as processing power required for program operation. A time sample of 92.9 ms was chosen as the window size. This requires a window width of 4096 samples at a sampling frequency (Fs) of 44.1kHz.

$$H = 0.54 - 0.46cos(2\pi n/N) \tag{3.5}$$

This Hamming window allows for good frequency resolution and reduced spectral leakage of the signal.

Autocorrelation

By taking the autocorrelation of a signal, periodicity is emphasised; producing relatively large peaks at periods $k = 0, N, 2N, 3N$. As k approaches m, the amplitude of the peaks that do not correlate will reduce to zero. If low frequency noise still prevails on the signal, it can impact the performance of the pitch estimation being carried out. Using autocorrelation prior to sending it through the analysis section allows the program to emphasise any periodic or qausi-periodic waveform that may be accompanied by noise; from one or more sources. This will reduce the affect any noise has during the analysis of the signal.

$$Rxx = \frac{1}{m}\sum_{N=0}^{m-1} x(N)x(N-k) \tag{3.6}$$

3.1.2 Signal Analysis

Linear Predictive Coding Method

LPC analysis is the process of approximating speech for the linear combination of past and present values. In this program MATLABs built in LPC function is used to create the LPC coefficients and error values. LPC was chosen for the initial performance testing stages of this project as it is less computationally expensive compared to some of its counterparts. The LPC is solved using the autocorrelation method as stated in the literature review, the result is the linear prediction coefficients that allows accurate representation of a signal.

$$y(w) = \frac{\sigma_e^2}{|A(w)|^2} \tag{3.7}$$

$$y(w) = \frac{\sigma_e}{|A(w)|} \tag{3.8}$$

σ is the variance of the prediction error and A(w) is the prediction coefficients of the LPC.

The linear prediction coefficients alone are not able to represent the information required for the project. To accurately gain pitch information the output is converted to the LPC spectral envelope describe by equation 3.8 above. The LPC analysis block carries out the conversion to LPC spectral envelope using the following steps:

1. Take the Fourier Transform of the linear prediction coefficients
2. Divide the error coefficient by the output in step 1
3. Take the logarithm of step 2

The LPC envelope is shown in Figure 3.4 by the blue line. This line mirrors the peaks and troughs of the frequency spectrum.

Some error is introduced when using the spectral envelope, this error can be offset by increasing the order of the LPC analysis carried out. However, the trade is a higher computational effort to calculate the LPC coefficients. The order of LPC used determines the accuracy of the spectral envelope to the original signal frequency spectrum output. For use in this project, a 150th order Linear predictive analysis was carried out; this provided

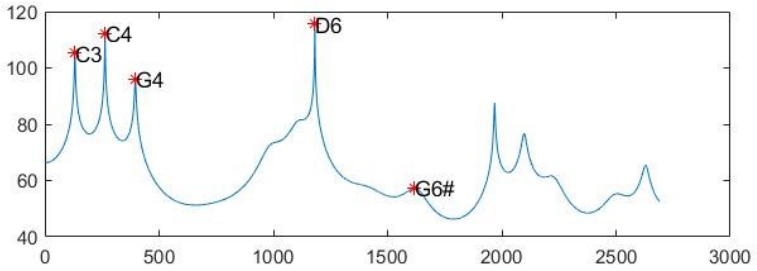

Figure 3.4: LPC spectrum envelope

the required resolution to separate notes that appear close together on the frequency spectrum.

Cepstral Analysis Method

Cepstral Analysis is used to accurately separate the convolution of speech signal into its base components. The complex Cepstrum is defined as the inverse FFT of the complex logarithm of the FFT, (Verhelst 1988). This is mathematically represented as:

$$\hat{x(n)} = F^{-1}log(F[x(n)]) \tag{3.9}$$

There are two methods used to obtain the cepstral coefficients, Cepstral Analysis as described in the Chapter 2 of this document and converting LPC Coefficients to Cepstral Coefficients for further processing. This project looks at the method described in chapter 2.

The following steps are used to calculate the Cepstral coefficients:

1. Calculate the Fourier Transform
2. Calculate the Log of step a
3. Calculate the Inverse Fourier Transform

A liftering operation is carried out on the Cepstral coefficients limiting the output to frequencies between $100Hz$ and $2205Hz$. The cepstral coefficients are passed back to the main script along with Quefrency information to plot the data.

Average Mean Difference Function Method

The Average Mean Difference Function is a pitch detection method based in the time domain. It provides accurate estimates of pitch periods based on the difference formed between time (n) and time $(n - m)$. The absolute value is calculated to remove any negative going delays that may correspond to voiced sections of the track.

AMDF is defined as:

$$x_m(m) = \frac{1}{N - m - 1} \sum_{n=0}^{N-m-1} |S_w(n + m) - S_w(n)| \tag{3.10}$$

$$x_m(m) = \frac{1}{N - m - 1} \sum_{n=0}^{N-m-1} |S_w(n) - S_w(n - m)| \tag{3.11}$$

(Hui, L, Dai, BQ and Wei, L 2006)

The AMDF coefficients are passed to the peak detection algorithm with a parity bit to tell the function to search for local minima in the signal.

Harmonic Product Spectrum

HPS was added to the program to determine the fundamental frequency of the singer. To do this it transfers the windowed data to the frequency domain using the Fast Fourier Transform. The HPS system then downsamples the original data input up to 5 times. It was limited to 5 to reduce the amount of error when detecting the fundamental frequency; if the number of downsamples is increased there is a chance the the multiple over tones of the downsampled data will fall within one frequency bin compromising the results.

Once the data has been down sampled it is then multiplied with previous results and the maximum peak in the output is found. This frequency is then used to compare the pitch detected from one of the three previous algorithms discussed. If there is a match between the two, then the data is stored in a vector array and displayed to the user.

3.1.3 Post-Processing

Pitch Detection Algorithm

The PDA MATALB function is passed the output of either one of analysis algorithm as well as a parity bit. The parity bit tells the PDA function to calculate the maxima or the minima of the passed signal.
Pitch detection is carried out by iterating through the samples of each frequency spectrum vector, as it iterates through the algorithm, it takes the differential of the wave form and looks for a change from a positive going waveform to a negative going wave form. When this change in slope is detected the algorithm stores the bin location and magnitude of where the change occurred, which is always located at $Bin = n-1$, where n is the current sample.

Pitch Period Estimation

The pitch period is found by finding the local minima and maxima of the signal, and calculating the period from time $t(0)$ to the local minima/maxima at time $t(k)$. The pitch period relates to the frequency by the following equation:

$$Freq = \frac{1}{Period} \tag{3.12}$$

Note Estimation

The Note Estimation function is used to determine the musical note designation and the difference between the note sung and the musical note value. Frequency information can be represented by the Semitone and Cents values, this project will represent the note to the nearest semitone by implementing the equation:

$$Semitone = 12log_2\frac{F_0}{440} + 69 \tag{3.13}$$

The detected frequency is compared to the required frequency values of the semitones to determine the error (%). Both the error and the calculated note are passed back to the main script.

Harmonic Ratio Estimation

The Harmonic Ratio Estimation function calculates the ratios of the fundamental note of two or more tracks. The ratio between notes is required to be within ± 0.1 of the following ratios $\frac{2}{1}, \frac{3}{2}, \frac{4}{3}, \frac{5}{4}, \frac{6}{5}, \frac{5}{3}$ and $\frac{8}{5}$ in order to be considered in harmony or a consonant chord. A vector containing the results is then passed back to the main script; a 1 bit indicating harmony, a 0 indicating the notes are out of harmony.

3.2 Software Testing and Evaluation

Testing and evaluation of the software will be carried out using a combination of pre-generated frequencies and singing tracks supplied by the supervisor of this project, Mark Phythian.
First each method was tested against a singular frequency from notes A2 through to A6. The accuracy, time, and notes detected were documented for each algorithm. These tracks are software generated tracks of 3 seconds duration with a sampling frequency (Fs) of $44.1kHz$.

Then each MATLAB script was tested against two tracks each containing three different frequencies, with a duration of 3 seconds and an Fs of $44.1kHz$.

Both tracks contain frequencies associated with musical notes commonly used throughout western music. Table 1 details the note, frequency, period and the MIDI number of the notes that were used during the second track.

Table 3.1: Musical Note Information.

Midi Note	Designation	Frequency	Period
48	C3	130.813	0.008
55	G3	195.998	0.005
64	E4	329.628	0.00303
69	A4	440	0.00227
73	C5#	554.336	0.00180
78	F5#	739.998	0.00135

The song Hallelujah by Leonard Cohen 1984 (Adam Scott rendition) was used to assess each algorithms ability to discern harmonic information from pre-recorded vocals. It was assessed against a 30 second length of track. There were four tracks used to carry out the test, Baritone, Bass, Lead and Tenor only tracks. Equation 3.13, was used to calculate the Semitone values and therefore the corresponding note. This value was rounded to the nearest whole number.

If HPS detected notes matched those from either LPC, Cepstral analysis or AMDF; the time stamp and note information are then stored and compared to sheet music for a match. These results are reported in Chapter 4 of this report.

3.3 Hardware Design

3.3.1 Hardware Requirements

There are several measures to estimate the performance of a system, the most common are time elapsed and floating point operations per second. A floating point operation is defined as the addition or multiplication of single or double precision numbers that conform to the IEEE 754 standard (Parker 2017, Karp & Flatt 1990).

To implement these algorithms in hardware, the hardware resources first needed to be estimated. MATLAB was used to test the complexity of the software and therefore estimate the hardware required to implement these algorithms. Three parameters were used to estimate the requirements; memory utilised, computation time and FLOPS. These results were then used to estimate the latency and minimum memory resources for each algorithm to be implemented on an FPGA and an Android Mobile based system. Since the average mobile device has a significant amount of RAM and flash memory on board, only the latency of the system was calculated.

Testing was carried out on a Intel I7-7700HQ 2.8 GHz processor. Since it is difficult to estimate how often and how long MATLAB utilises multiple cores for processing, it was restricted to one core for testing the time elapsed of each algorithm. The testing took advantage of MATLAB's inbuilt profiler estimate the memory used and the *tic* and *toc* functions to accurately estimate the computation time. Each algorithm was timed for various window lengths; 512(2^9) through to 16,384(2^{14}) samples. This data is used to

provide an estimate of the latency in the hardware for different data lengths.

To test the capability of the Intel processor I7-7700HQ and Snapdragon 855 processors; Linpak Extreme and Mobile Linpak were used to estimate the amount of floating point operations per second. The timing data and GFLOPS data were used to calculate the expected latency across each system.

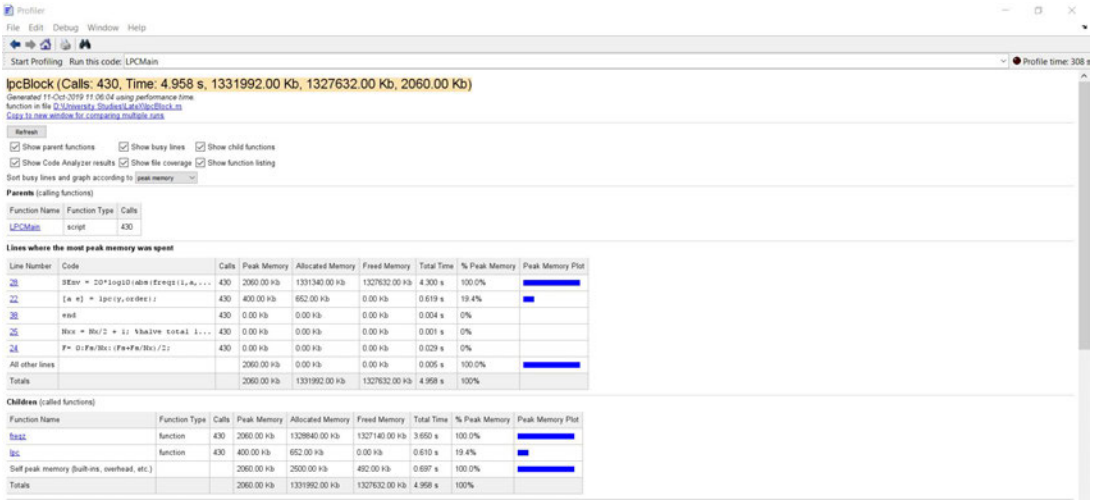

Figure 3.5: MATLAB Profiler Viewer

FPGA Calculations were based of the Altera Cyclone series. Using the allocated memory from MATLABs profiler and the M20K,M10K and M9K memory blocks available on the FPGA, memory resources to carry out. If the assumption that the memory is set up as dual port RAM and pipelined then the amount of resources required to store a audio sample can be calculated using

Memory Blocks Required

$$= \frac{Allocatedmemory}{M * 2} \tag{3.14}$$

Where M is the size of the FPGA memory block.

3.3.2 Hardware Implementation

The hardware implementation may be undertaken if there is enough time permitting before completion of this project. The design of the system will likely be based around the FPGA architecture. FPGAs are integrated circuits that are easily programmable by the

end user. More information on FPGAs and their architecture can be found in chapter 2 of this report.

If this project moves forward with the hardware design, it is likely that MATLAB and Simulink will be used to generate C or HDL languages from the code previously developed for this project. The HDL language will be simulated and optimised using the built in HDL coder function targeting the following areas Speed, Memory management and Resource sharing.

Chapter 4

Results

This chapter will review the results for the three types of processing algorithms developed in this project. The algorithms tested include:

1. Linear Predictive Coding
2. Cepstral Analysis
3. Average Mean Difference Function

Signal processing algorithms are categorised by either the time domain, frequency domain or a combination of both. The chosen algorithms represent a method from each category. These algorithms were used in conjunction with common sub function to output information related to pitch, harmony and accuracy.

The operation of these MATALB scripts are described in the Methods section of this report as well as the testing carried out on the three algorithms. Table 4.1 lists the tracks used for testing in this project.

Table 4.1: Track Information.

Track	**Description**	**Length (s)**	**Sampling Frequency**
1	110 Hz	3	44.1kHz
2	220 Hz	3	44.1kHz
3	440 Hz	3	44.1kHz
4	880 Hz	3	44.1kHz
5	1760 Hz	3	44.1kHz
6	130.813 Hz, 329.628 Hz, 554.336 Hz	3	44.1kHz
7	195.998 Hz, 440 Hz, 739.998, Hz	3	44.1kHz
8	Hallelujah: Tenor Only	30	44.1kHz
9	Hallelujah: Bass Only	30	44.1kHz
10	Hallelujah: Baritone Only	30	44.1kHz
11	Hallelujah: Lead Only	30	44.1kHz

4.1 Linear Predictive Coding

The LPC testing carried out, approximates the singing content based on the past and present values. It is converted to the spectral envelope to improve the pitch detection of the notes. The test conducted on single frequency signals provided relatively accurate results. Table 4.2 shows the results for the single note tracks using the LPC algorithm. The accuracy column indicates how close the estimation is to the actual note. A positive going percentage indicates the note was overestimated while the negative going, indicates it underestimated the note.

Table 4.2: LPC Tacks 1-5 Results.

Track	Freq Detected (Hz)	Accuracy %	Time(s)
1	110.69	0.6272	0.0047
2	220.378	0.17182	0.0052
3	440.198	0.20864	0.0022
4	884.8801	0.5545	0.0048
5	1764.7	0.26705	0.0047

The accuracy of the LPC pitch detection was affected by the Order of the Linear Prediction being used and the length of the buffer used to create the spectral envelope. This effect wasn't seen until the software was tested using multiple note tracks and resulted in the spectral envelope peak falling between to closely located notes. As a result the order of the LPC was increased in an attempt to reduce this effect.

The addition of multiple frequencies didn't affect the speed of computation dramatically; however, the accuracy of the results were affected; in some instances, up to 10% particularly at the lower end of the frequency range.

The pitch detection algorithm developed, detected several notes that were not contained within the track used for testing. This inaccuracy was also present when testing against the Hallelujah tracks as seen in the table 4.4 below.

The LPC algorithm performed relatively well when they were tested with the Hallelujah tracks; in some case returning up to 89% of all detected notes correctly. The algorithm performed poorly against the Tenor and Bass tracks returning 33% and 57% of all notes; respectively. The LPC method provided the best accuracy of the three methods chosen

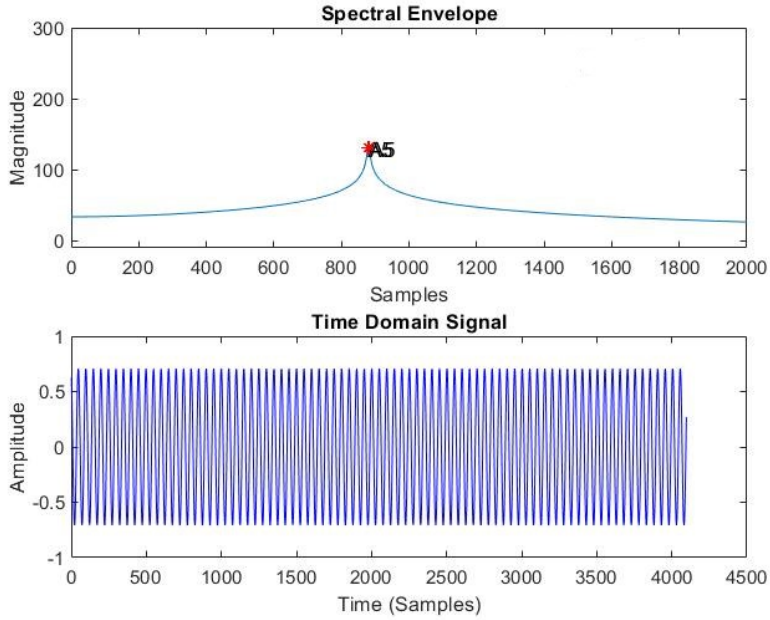

Figure 4.1: LPC Track 5

Table 4.3: LPC Tacks 1-5 Results.

Track	Freq Detected (Hz)	Accuracy (%)	Time (s)
6	144	10.8	0.0053
	338.1	2.57	
	554..8	0.0837	
7	211.6	7.96	0.0055
	444.5	1.022	
	740.5	0.07	

to test. It returned an average of 66% of all notes correctly across the test tracks and had an average accuracy of 4% on the single and multi-note test tracks.

Table 4.4: LPC Tracks 8-11 Results.

Track	**Notes Correctly Identified %**	**Average Time(s) per sample**
8	33.8	0.0044
9	57.02	0.0051
10	86.87	0.0046
11	89.95	0.0044

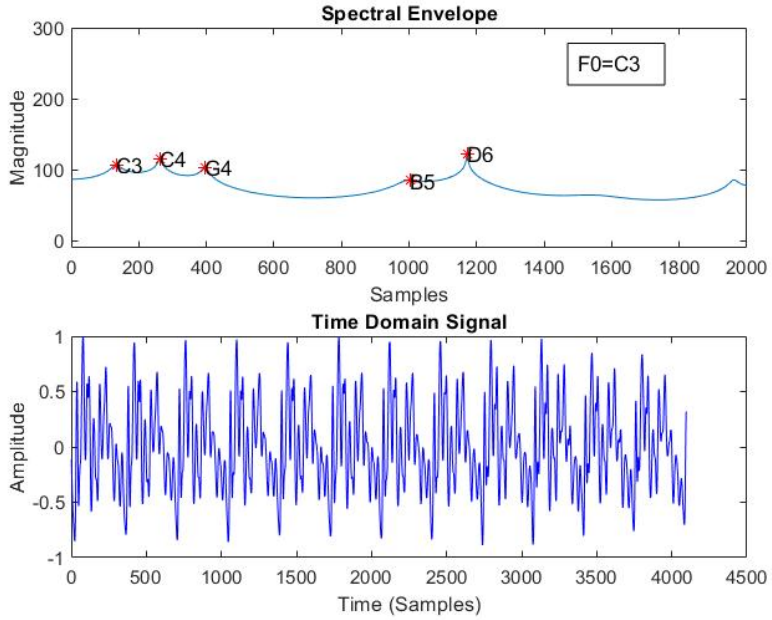

Figure 4.2: LPC Track 9

4.2 Cepstral Analysis

Defined as the inverse FFT of the complex logarithm of the FFT, the cepstral analysis algorithm provided the fastest computation compared to AMDF and LPC algorithms. Excluding the detection of the A2 note, Cepstral analysis performed adequately against the single note tracks.

Table 4.5: Cepstral Analysis Tacks 1-5 Results.

Track	Freq Detected (Hz)	Accuracy %	Time(s)
1	Could Not Detect	N/A	0.0047
2	227.32	3.32	0.0038
3	421.053	-4.306	0.004
4	882	0.2272	0.004
5	1764	.2272	0.0038

When tested against the multi-note tracks, the Cepstral analysis algorithm was relatively inaccurate, detecting a frequency 28% greater than the required frequency.

When tested against the pre-recorded tracks containing the song Hallelujah; the algorithm returned an average of 28.32% of the notes correctly and in one case returned less than 1% of all notes correctly, since the Cepstral analysis output contains a considerable amount of

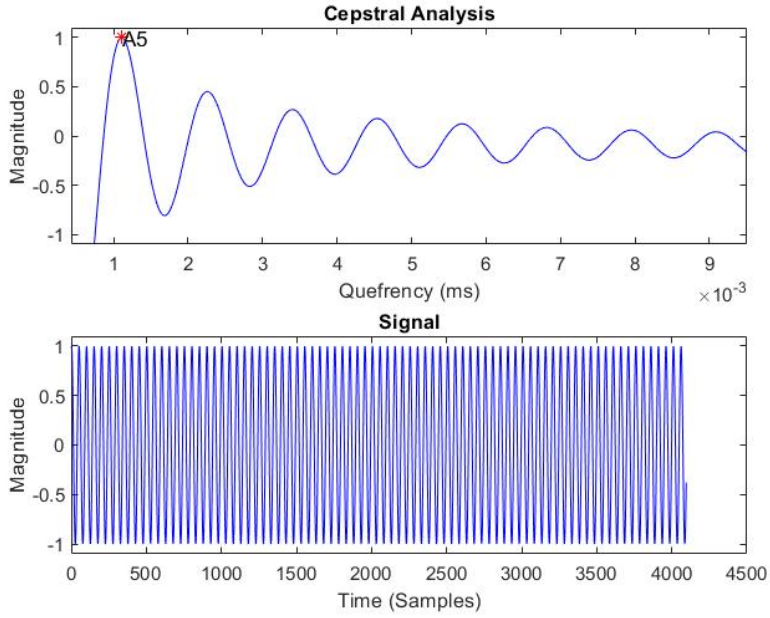

Figure 4.3: Cepstral Analysis Track 4

Table 4.6: Cepstral Analysis Tacks 6 and 7 Results.

Track	Freq Detected (Hz)	Accuracy (%)	Time (s)
6	144.18	10.17	0.0015
	288.235	12.55	
	572.727	3.31	
7	238.378	28.10	0.0019
	380.172	-13.59	
	722.95	2.30	

noise, this poor performance could be attributed to the threshold for detecting frequency at which the peak occurs; resulting in the noise being picked up instead of the pitch period of the note. The only positive performance from the Cepstral Analysis algorithm came from its computational speed; which was considerably faster than the other two algorithms.

4.3 Average Magnitude Difference Function

AMDF is a method based in the time domain; it provides an estimate of the pitch based on the difference between the current value and the previous value in the signal. The

Table 4.7: Cepstral Analysis Tacks 8-11 Results.

Track	Notes Correctly Identified %	Average Time(s) per sample
8	30.8	0.0011
9	41.56	0.0011
10	40.95	0.0012
11	< 1	0.0011

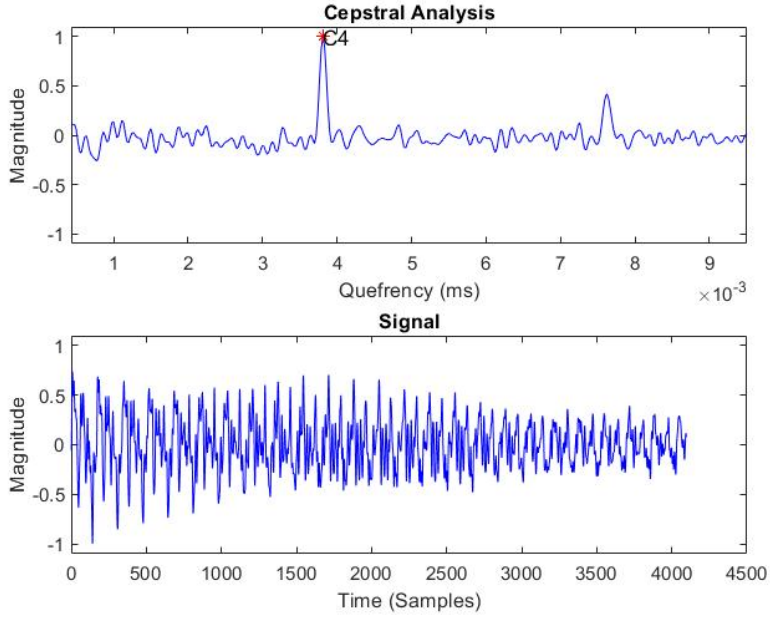

Figure 4.4: Cepstral Analysis Track 8

AMDF algorithm has its bound set at a minimum of 105 Hz to a maximum of 2205 Hz. The AMDF pitch detection provided accurate results for all single note test tracks (Note A2 and above). The AMDF algorithm was most accurate sampling the A2 note (110 Hz) underestimating the fundamental frequency by 0.232 Hz, while the most inaccurate was sampling the note A4(440 Hz); underestimating the frequency by 18.95 Hz. The Table

Table 4.8: AMDF Tacks 1-5 Results.

Track	Freq Detected (Hz)	Accuracy %	Time(s)
1	109.98	-0.01777	0.0047
2	219.40	-0.272	0.0038
3	421.05	-4.306	0.004
4	864.17	-1.7375	0.004
5	1696.15	-3.627	0.0038

Table 4.9: Cepstral Analysis Tacks 6 and 7 Results.

Track	Freq Detected (Hz)	Accuracy (%)	Time (s)
6	140	7.023	0.00114
	279.11	15.325	
	525	5.29	
7	186.075	5.06	0.00108
	370.588	15.77	
	711.29	3.879	

4.8 shows the test results against the single note test tracks.

These results weren't replicated in the multi-note tracks used for the next stage of testing. The algorithm miss detected all the notes embedded on the tracks. It underestimated the frequency of each note by a minimum of 5%. This caused each note detected to be returned incorrectly to the user by the Note Estimation script.

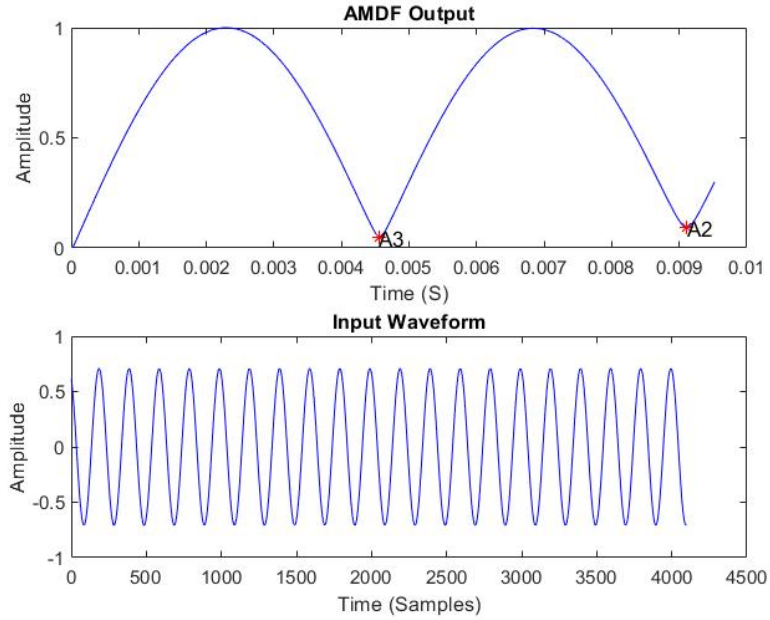

Figure 4.5: AMDF Track 2

Against the Hallelujah tracks, the AMDF algorithm performed poorly, returning an average of 37% of all notes correctly. It performed the best against the Baritone only track returning a total of 78.13% of the notes detected correctly.

The AMDF algorithm has a longer latency compared to the other two algorithms, in some

Table 4.10: AMDF Tacks 8-11 Results.

Track	Notes Correctly Identified %	Average Time(s) per sample
8	12.6	0.0099
9	25.24	0.0097
10	78.13	0.0096
11	33.61	0.0098

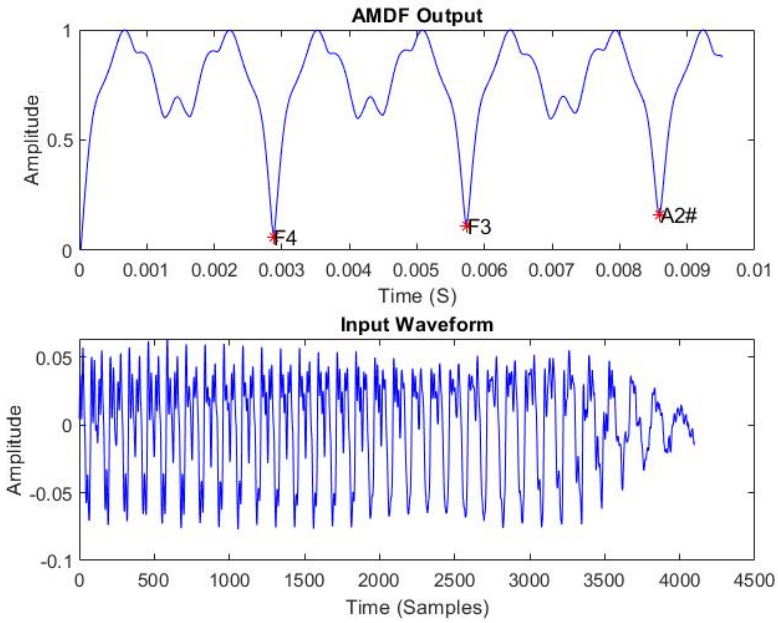

Figure 4.6: AMDF Track 8

instances it was almost double that of the LPC algorithm.

4.4 Hardware Requirements

To determine the hardware requirements testing was carried out on each algorithm using a consumer laptop which utilises a Intel I7-7700HQ processor. It was found that on average the time taken to perform each algorithm varied heavily depending on the window sizethis can be seen in figure 4.7. Using the average times to process the audio information a GFLOPS estimate was calculated. Table 4.11 below provides an estimate of the GFLOPS capable of each piece of Hardware using a single core processing a window size of 4096 samples. If these algorithms were to be used on mobile phone, there would be a significant amount of latency in the equipment to produce the same result produced by the Intel

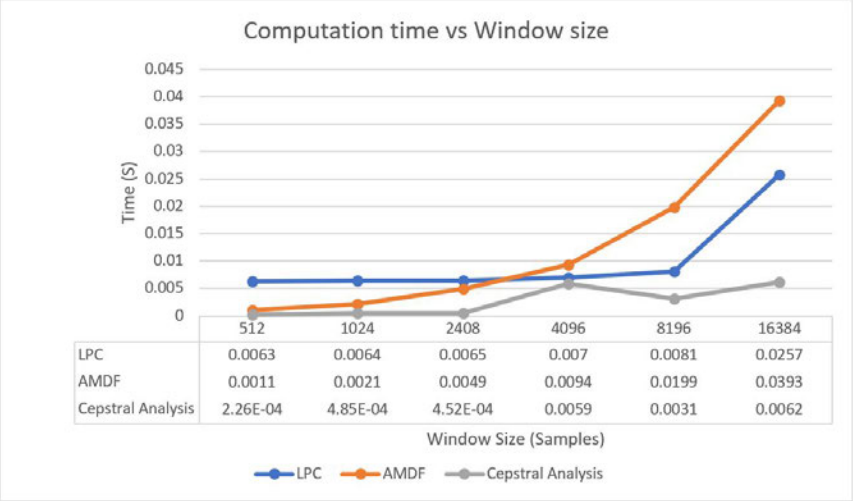

Figure 4.7: Average Time Taken Compared to Window Size

processor. An FPGA implementation, has a substantial advantage of speed with Intel estimating their mid range Altera Cyclone 10 is capable of speeds upwards of 76 GLOPS; roughly the equivalent to utilising all of the I7-700HQ cores available(79 GFLOPS). Most

Table 4.11: FLOP Estimation.

Hardware	**GLOPS**	**LPC Latency (S)**	**Cepstral Latency (S)**	**AMDF Latency(S)**
Intel I7-7700	17.25	0.007	0.0059	0.0094
Galaxy 10 - Snapdragon 855	0.13682	0.8825	0.7438	1.185
Altera Cyclone 10	76	0.00158	0.001339	0.002135

mobile phone systems have more than enough memory resources to implement these algorithms. FPGA systems on the other hand are generally limited in on board memory. Figure 4.8 provides a snap shot of the memory being allocated to the algorithm calculation.

The estimated memory requirement for computation of each algorithm is shown in table 4.12. It provides the expected number of memory blocks required to perform the calculations. Where M20K,M10K and M9K contain 20Kb, 10Kb and 9Kb of memory respectively; these memory block sizes are commonly used on Intel's FPGA range. A mid range FPGA model, such as the Cyclone V from Intel; would be recommended for these algorithms to be implemented on.

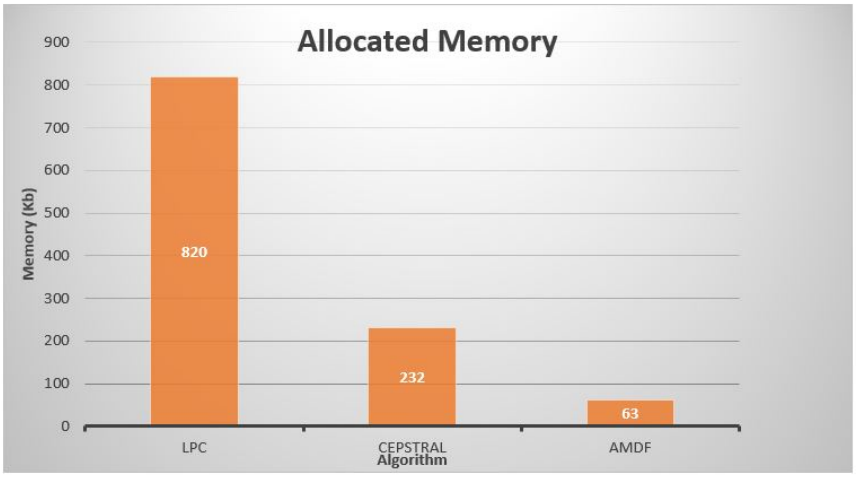

Figure 4.8: Average Memory used for computation

Table 4.12: Estimated Memory Blocks Required.

Memory Type	LPC	Cepstral	AMDF
M20K	82	4	24
M10K	164	8	48
M9K	184	7	52

Chapter 5

Conclusions and Further Work

This chapter provides a summary of the project. It addresses the main lessons learned and any possible future work that may be carried out.

5.1 Conclusions

The research conducted in this project focused around the use of various speech analysis techniques used to extract harmonic content from singers vocals. Testing was carried out using, Linear Predictive Coding, Cepstral Analysis, and Average Magnitude Difference method against software generated and pre-recoded tracks. The results from the three methods concluded with LPC providing the most accurate results across all 11 tracks used in the testing. This resulted from the linear prediction coefficients being used to generate the LPC spectral envelope, allowing the pitch detection algorithm to easily detect the correct note.

AMDF and Cepstral analysis, in most cases, the detected note, was found to be within 2% to 28% of the required note. These in-accuracies were consistent across all 11 tracks. In the provided test track containing Hallelujah, both AMDF and Cepstral Analysis returned the least amount of notes correctly, while LPC returned an average of 66.8% of notes. All methods returned incorrect notes when the singer was transitioning from one note to the next. In some cases the notes that weren't returned were due to the HPS output not being matched to the detected frequencies of the algorithms. Since the HPS algorithm is dependent on harmonic frequencies present in the signal(Sripriya & Nagarajan 2013), it

becomes difficult to estimate the fundamental frequency if the required harmonics aren't present. Better results may achieved by using a combination of LPC and Cepstral Analysis or AMDF instead.

In the initial research for the project, visual feedback to the user wasn't considered to be a bottle neck of the program, however it increased the operation time of the program significantly. General improvements to this can be easily made to improve the efficiency of plotting.

5.2 Further Work

Given the results obtained in the project thus far. There are several items that can be considered for future work. These include:

1. Investigate,design and test speech separation algorithms for multi-singer tracks.
2. Develop a rhythm and beat detection systems. Autocorrelation emphasises the periodicity of a signal and has successfully been employed for beat detection previously.
3. Hardware design and construction. Further investigation needs to be done to implement these algorithms on hardware. Specifically the feasibility into using common hardware such as a phone or microprocessor compared to a FPGA based system. Utilising FPGA Architecture would provide a suitable and relatively inexpensive option for implementing the signal processing system into hardware. Implementing this on a readily available device such as a phone or tablet, would allow people to access the system at a fraction of the cost.
4. Investigate improvements to be made to the current Linear prediction program, including methods for detecting the onset of the note, utilising a combination of LPC, Cepstral Analysis and HPS for note detection.

References

Amit S B, Rohit R, S. S. B. H. V. V. G. T. R. S. (2018), 'Power efficient speaker verification using linear predictive coding on fpga'.

Bahoura, M. & Ezzaidi, H. (2013), 'Hardware implementation of mfcc feature extraction for respiratory sounds analysis', pp. 226–229.

Britannica (2016).
URL: *https://www.britannica.com/topic/affricate*

Britannica (2018).
URL: *https://www.britannica.com/art/singing*

Chao-Ling Hsu, J.-S. R., Deliang Wang, J.-S. R., Jang, J.-S. R. & Ke Hu, J.-S. R. (2012), 'A tandem algorithm for singing pitch extraction and voice separation from music accompaniment', *Audio, Speech, and Language Processing, IEEE Transactions on* **20**(5), 1482–1491.

De Cheveigné, A. (2002), 'Yin, a fundamental frequency estimator for speech and music', *Journal of the Acoustical Society of America* **111**(4), 1917–1930.

Deller, J. R. (1993), *Discrete-time processing of speech signals*, Macmillan Pub. Co., New York.

Government, Q. (2019), 'What is personal information'.
URL: *https://www.oic.qld.gov.au/guidelines/for-community-members/Information-sheets-privacy-principles/what-is-personal-information*

Guan, L. (2017), *FPGA-based Digital Convolution for Wireless Applications*, Springer Series in Wireless Technology, Springer International Publishing, Cham.

Guoning, H. & Deliang, W. (2010), 'A tandem algorithm for pitch estimation and voiced speech segregation', *Audio, Speech, and Language Processing, IEEE Transactions on* **18**(8), 2067–2079.

Gupta, C., Li, H. & Wang, Y. (2017), 'Perceptual evaluation of singing quality'.

Huckvale, M. (2019), 'Introduction to speech science'.
URL: *https://www.phon.ucl.ac.uk/courses/spsci/iss/week5.php*

Jun, Xu Ariyaeeinia, A. R. (2005), 'Migrate levinson-durbin based linear predictive coding algorithm into fpgas'.

Karp, A. H. & Flatt, H. P. (1990), 'Measuring parallel processor performance', *Communications of the ACM* **33**(5), 539.

Kodag, R. B., Patil, M. D. & Gaikwad, C. J. (2016), 'Harmonic product spectrum based approach for tonic extraction in indian music'.

Kovacevic, B. (2017), *Robust Digital Processing of Speech Signals*, Springer International Publishing, Cham.

Kruger, H., Lotter, T. & Vary, P. (2004), 'A versatile dsp-system for student-projects on embedded real-time audio signal processing'.

Kumaraswamy, Balachandra Poonacha, P. G. (2017), 'Modified square difference function using fourier series approximation for pitch estimation'.

Kwan, C. Yin, J., Ayhan, B., Chu, S., Liu, X., Puckett, K., Zhao, Y., Ho, K. C., Kruger, M. & Sityar, I. (2008), 'Speech separation algorithms for multiple speaker environments'.

Ladefoged, P. (2001), *Vowels and consonants : an introduction to the sounds of languages*, Blackwell, Malden, Mass.

Leis, J. W. (2011), *Digital signal processing using MATLAB for students and researchers*, John Wiley & Sons, Inc., Hoboken, New Jersey.

Leong, P. H. W. (2008), 'Recent trends in fpga architectures and applications'.

Makhoul, J. (1975), 'Linear prediction: A tutorial review', *Proceedings of the IEEE* **63**(4), 561–580.

McLeod, P. G. (2008), Fast, Accurate Pitch Detection Tools for Music Analysis, Thesis.

Melnikoff, S. J., Quigley, S. F. & Russell, M. J. (2002), 'Speech recognition on an fpga using discrete and continuous hidden markov models', **2438**, 202–211.

Min, K., Chien, D., Li, S. & Jones, C. (1988), 'Automated two speaker separation system'.

Muller, M., Ellis, D. P. W., Klapuri, A. & Richard, G. (2011), 'Signal processing for music analysis', *Selected Topics in Signal Processing, IEEE Journal of* **5**(6), 1088–1110.

Parker, M. (2017), 'Learn how to calculate and compare the peak floating point capabilities of digital signal processors (dsps), graphics processing units (gpus), and fpgas.'.
URL: *https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf*

Potamitis, I. & Kokkinakis, G. (2007), 'Speech separation of multiple moving speakers using multisensor multitarget techniques', *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* **37**(1), 72–81.

Queensland, L. A. (2019), 'Privacy'.
URL: *http://www.legalaid.qld.gov.au/Find-legal-information/Personal-rights-and-safety/Privacy-and-identity/Privacy*

Rabiner, L., Cheng, M., Rosenberg, A. & McGonegal, C. (1976), 'A comparative performance study of several pitch detection algorithms', *IEEE Transactions on Acoustics, Speech, and Signal Processing* **24**(5), 399–418.

Rabiner, L. R. (1978), *Digital processing of speech signals*, Prentice-Hall signal processing series, Prentice Hall, Englewood Cliffs, N.J.

Rao, S. S. S. P. (2016), *Field programmable gate arrays and applications*, Alpha Science International Ltd, Oxford, England.

Riley, W. D. & Carrol, L. M. (2016), *The performer's voice*, second edition. edn, Plural Publishing, San Diego, California.

Shukla, K. K. (2013), *Efficient algorithms for discrete wavelet transform : with applications to denoising and fuzzy inference systems*, SpringerBriefs in computer science, Springer, London.

Sripriya, N. & Nagarajan, T. (2013), 'Pitch estimation using harmonic product spectrum derived from dct', *IEEE International Conference* pp. 1–4.

Suma, S. A. Gurumurthy, K. S. (2010), 'Novel pitch extraction methods using average magnitude difference function (amdf) for lpc speech coders in noisy environments'.

Verhelst, W. Steenhaut, O. (1988), 'On short-time cepstra of voiced speech'.

Wavelets (2014).

Zbilut, J. P. & Marwan, N. (2008), 'The wiener–khinchin theorem and recurrence quantification', *Physics Letters A* **372**(44), 6622–6626.

Appendix A

Project Specification

ENG 4111/2 Research Project

Project Specification

For:	**Jarred Oliver**
Topic:	Harmony Analysis in A 'Capella Singing
Supervisors:	M. Phythian
Sponsorship:	Faculty of Health, Engineering & Sciences
Project Aim:	To research methods for analysing two or more A 'Capella tracks and to develop a system in MATLAB to analyse these tracks for characteristics such as pitch; and indicate closeness of harmony to the user.

Program:

1. Research the background information on Speech analysis and Singing Analysis and their implementation methods.
2. Critically examine the methods used in analysis of vocal tracks without musical accompaniment.
3. Compare implementation methods.
4. Design and code a system for analyising pitch and harmonic content as an indicator of singing quality.
5. Test the designed system.
6. Analyse the results of the systems with various tracks.
7. Submit an academic disseration on the research.

As time and resources permit:

1. Design and build a practical real-time harmony measurement system.
2. Test the constructed harmony measurement system.

Agreed:

Student Name:	Jarred Oliver
Date:	3 April 2019
Supervisor Name:	Mark Phythian
Date:	3 April 2019

Appendix B

Risk Assessment

Step	Hazards What could cause injury or ill health, damage to property or damage to the environment	Risk What could go wrong and what might happen as a result	Initial Risk Rating			Potential Control Measures (Consider Hierarchy of Control - Elimination, Substitution, Isolation, Engineering Controls,	Residual Risk Rating		
			C	L	Risk Rating		C	L	RR
1	Poor Communication	No Approval/Delay of approval from USQ to start project	C	5	Very High	Communicate effectively and early with USQ and Supervisor	C	1	low
1,2,3,4,5,6	Poor ergonomic setup of workstation	lower back injuries, repative strain injuries	B	4	High	Set up work station IAW an approved ergonimics set up guide, Take regular breaks	B	3	Moderate
1,2,3,4,5,6	Long periods of computer monitor use	Possible eye strain	A	5	Moderate	Take regular breaks from screen use, setup monitor IAW ergonimcs guide.	A	1	Low
6	Data Breach	Loss of Personnel Information	D	2	High	No personnel information will be kept or stored for the purposes of this project	C	1	low

Optional 1	Design of Electrical Hardware	Poor/inaccurate electrical design could result in injury resulting from electric shock	C	3	Moderate	Design will be undertaken in accordance with applicable standards. Utilisations of correct PPE including saftey glassess, non conductive clothing, ESD safety equipment	C	2	Moderate
Optional 1	Soldering station use	Incorrect use of soldering/ air gun equipment leading to burn injuries	B	3	Moderate	Use of correct PPE, including long sleeve clothing	B	2	Low
Optional 1	use of cutters, wire-strippers & filing equipment	Incorrect use leading to cuts, lacerasions	B	3	Moderate	Use of correct PPE	B	2	Low
Optional 1	Poorly manufactured system blocks	Smoke inhalation from acring	C	1	Low	Testing carried out in well ventilated area, use of forced ventilaition system	C	1	Low
Optional 1	Testing of Electrical Hardware	Electric Shock	C	4	High	Reduce period of time equipment is live, Use correct PPE	C	2	Moderate
Optional 1	Poorly Designed Hardware	Equipment Damage	C	1	Low	Actively Carry out simulation and confirmation checks to ensure equipment functions correctly	B	1	Low

Optional 1	Lack of Rsources/ Delay in Resources	Delay in project completion or failure to complete project	C	3	**Moderate**	Identifiy equipment early during the project	C	1	**Low**

Appendix C

Resource Requirements

C.1 Consumables

1. Twisted Pair wire 28 AWG 19.90/m USD
2. Solder
3. PVC Case Size -TBA
4. FPGA Development Board
5. Electronic Display - RGB LCD 69.56 USD
6. Microphone

C.2 Lab Equipment and Software

1. Soldering Station
2. Oscilloscope
3. Wire Cutters
4. Wire Strippers
5. Soldering stand
6. Mulitmeter
7. Altium
8. MATLAB
9. Qaurtus Prime
10. Audio Tracks

Appendix D

MATLAB Code

This sections presents the code developed for the Harmony Analysis project using MATLAB. This chapter is organised as follows

1. Low Pass Filter
2. Peak Detection Algorithm
3. Pitch Period Estimation
4. Note Estimation
5. Harmonic Ratio
6. Linear Predictive Coding Function
7. Cepstral Analysis Function
8. Average Magnitude Difference Function

D.1 The Low Pass Filter MATLAB Function

The function `lpfilter.m` is passed the audio signal, sampling frequency the required order of the filter and the cut off frequency. The order and the cut off frequency are chosen by the user. Once this information is passed to the function, the signal is filtered using a filter of order `n`. The filter is a cascaded low pass filter following the equation $\frac{sin(\omega(n-m+eps)}{\pi(n-m+eps)}\frac{sin(\omega(n-m+eps)}{\pi(n-m+eps)}$.
Where `n` is the order, `m` is $\frac{n-1}{2}$ and `eps` is eqaul to the value of 2.2^{-16} this is used to prevent any divide by zero errors from occuring
Once the signal is filter it is passed back to the main equation.

Listing D.1: Low Pass Filter Function.

```
%Low Pass Filter Function
%Author: Jarred Oliver
%Date: 07/04/2019
%Function creates low pass filter coefficients by cascading the LPF
%function with a delayed version of its self
%the signal is then passed through the filter using MATLABs filter funciton
%-------------------------------------------------------------------------
%Inputs
%Wc - Chosen frequency cutoff
%y-signal input
%Fs - Sampling Frequency
%order - the order of the filter
%-------------------------------------------------------------------------
%output
%result - the filtered signal
%-------------------------------------------------------------------------

function [result] = lpfilter(Wc,y,Fs, order)
w = Wc*pi/(Fs/2);
 m = (order-1)/2;

for n = 1:order+1
 H(n) = sin((n-m+eps)*w)/(pi*(n-m+eps))*sin((n-m+eps)*w)/(pi*(n-m+eps));
end
LP = hamming(length(H))'.*H;%

result = filter(LP,1,y);
end
```

D.2 The Peak Detection Algorithim MATLAB Function

The function `axPeakDet.m` determines the location of the peaks in the spectral envelope. The matrix containing the spectral envelope information is iterated through, and each iteration is passed through to the Peak Detection Algorithim (PDA) function. The PDA function was developed using the built in fuction `diff()`, the function determines the slope at each location in the envelope. This slope is compared with the previous slope value to determine if there is a change from a positive going slope to a negative going slope. If a change is detected in the slope then the previous value is stored in matrix called `magvec`. This vector is then passed back to the main equation.

Listing D.2: Peak Detection Algorithim.

```
%Peak Detection Funciton
%Author: Jarred Oliver
%Date: 01 April 2019
%Updated: 20/05/2019
%Description:
%This program calculates the differential at each points of the spectral
%envelope passed to the program. If the parity bit is set to 1, the
%function locates all positive going peaks, if set to 2 the program finds
%the location of all negative going peaks. It passes the magnitude and the
%location of each peak back to the main program
%-------------------------------------------------------------------------
%Input
%signal - signal vector peak search is to be carried out on
%F- Frequency vector
%bit- parity bit to determine mode of function
%-------------------------------------------------------------------------
%Output
%magvec - magnitude of each peak
%indexvec - location or index of each peak
%-------------------------------------------------------------------------

function [magvec, indexvec] = axPeakDet(signal,bit,F)

if nargin == 1
    bit = 1;
    F = 1:length(signal);

end
if nargin == 2
    F = 1:length(signal);

end

    tempvec = 0;
```

```
indexvec = [];
magvec = [];
abc=[];
sigdx = diff(signal);
if bit == 1
th = (max(sigdx)-min(sigdx))/2;%*mean(sigdx);
elseif bit == 2
    th = 0.3;
elseif bit == 3
    th = .1;
end
count = 1;
k = 1;

while k < length(sigdx)
   if bit == 1|bit == 3
    if sigdx(k) > th & sigdx(k) > tempvec;
        tempvec = sigdx(k);

    end


   if k>2   & sigdx(k)<0 & sigdx(k-1)>=tempvec%look for sign change

      indexvec(count) = F(k);
      magvec(count) = signal(k);
      tempvec = 0;
      abc(count) = F(k);
      count = count + 1;

   end

   k = k+1;


   if k == length(sigdx)
   %terminate program once the end of signal is reached
       break;
   end
   end
%-------------------------------------------------------------------------
     if bit == 2

    if sigdx(k) < th & sigdx(k) > tempvec;
        tempvec = sigdx(k);
    end

   if k>2 &signal(k)<signal(k+1) & signal(k-1)>signal(k) & signal(k) <= th
      %look for sign change

      indexvec(count) = F(k);
```

```
        magvec(count) = signal(k);
        tempvec = 0;
        count = count + 1;
        abc(count) = F(k);

     end
    k = k+1;
    if k == length(sigdx) %terminate program
                          %once the end of signal is reached
         break;
    end
      end
%-------------------------------------------------------------------------
end
return;
end
```

D.3 The Pitch Period Estimation MATLAB Function

The function `pitchperiod.m` calculates the pitch period in milliseconds and the Frequency in Hz for a given input. The function requires minimum and maximum samples, Sampling frequency and index to be passed from the main program to function. The default sample range is minimum of 10 to a maximum of 300 if no values are passed.

Listing D.3: Pitch Period Estimation Function.

```
%Pitch Period Calculation
%Author: Jarred Oliver
%Date: 20 May 2019
%Description:
%Calculates the pitch period of a time domain signal vector
%returns the estimated frequency & the pitch period to the main program
%-------------------------------------------------------------------------
%Input
%indexvec - Index location of peaks
%Fs - Sampling Frequency (Default 44100)
%-------------------------------------------------------------------------
%Freq - frequency estimate
%period - calculated pitch period
%-------------------------------------------------------------------------

function [Freq period] = pitchperiod(indexvec,magvec,Fs,smin,smax)
period = (indexvec/Fs); %Estimate period in seconds
Freq = 1./period; %Freq
end
```

D.4 The Note Estimation MATLAB Function

The function `NoteEstimation.m` determines the semitone value of the detected frequency according to the equation $Semitone = 12\log_2(Hz/440) + 69$

Listing D.4: Note Estimation Function.

```
%Note Estimation
%Author:Jarred Oliver
%Date: 25/06/2019
%Last Updated 15/09/2019
%-------------------------------------------------------------------------
%Program estimates the note that is likely being sung by the singer on the
%tape. Pitch values are passed to the program, and each pitch value is
%matched to the pitch of a note. Program then passes the note back to the
%main program for display
%-------------------------------------------------------------------------
%Input:
%Pitch - A vector containing the pitch values found in the audio track.

%Output:
%Note - A vector containing the string values of each note
%Accuracy - A percentage value of approximation, either positive or
%negative.
%-Positive value indicates the note is too high in pitch or sharp.
%-Negative value indicates the note is too low in pitch or flat.
%% Note Estimation
function [note, Accuracy] = NoteEstimation(Freq,scale)

note = '';
note = char(note);
m = 0;
        for k = 1: length(Freq)

Semitone = round(12*log2(Freq(k)/440)+69);

          switch Semitone

              case  36
                  Note = 'C2 ';
              m = 36;
              note(k,:) = Note;
              case 37
                  if scale == 1
                  Note = 'C2#';
                  else
                  Note = 'Db2';
                  end
              m = 37;
              note(k,:) = Note;
              case 38
```

```
        Note = 'D2 ';
    m = 38;
    note(k,:) = Note;
    case 39
        if scale == 1
        Note = 'D2#';
        else
        Note = 'Eb2';
        end
    m = 39;
    note(k,:) = Note;
    case 40
        Note = 'E2 ';
    m = 40;
    note(k,:) = Note;
    case 41
        Note = 'F2 ';
    m = 41;
    note(k,:) = Note;
    case 42
        if scale == 1
        Note = 'F2#';
        else
        Note = 'Gb2';
        end
    m = 42;
    note(k,:) = Note;
    case 43
        Note = 'G2 ';
    m = 43;
    note(k,:) = Note;
    case 44
        Note = 'G2#';
    m = 44;
    note(k,:) = Note;
    case  45
        Note = 'A2 ';
    m = 45;
    note(k,:) = Note;
    case 46
        if scale == 1
        Note = 'A2#';
        else
        Note = 'Bb2';
        end
    m = 46;
    note(k,:) = Note;
    case 47
        Note = 'B2 ';
    m = 47;
    note(k,:) = Note;
    case 48
```

```
        Note = 'C3 ';
    m = 48;
    note(k,:) = Note;
    case 49
        if scale == 1
        Note = 'C3#';
        else
        Note = 'Db3';
        end
    m = 49;
    note(k,:) = Note;
    case 50
        Note = 'D3 ';
    m = 50;
    note(k,:) = Note;
    case 51
        if scale == 1
        Note = 'D3#';
        else
        Note = 'Eb3';
        end
    m = 51;
    note(k,:) = Note;
    case 52
        Note = 'E3 ';
    m = 52;
    note(k,:) = Note;
    case 53
        Note = 'F3 ';
    m = 53;
    note(k,:) = Note;
    case 54
        if scale == 1
        Note = 'F3#';
        else
        Note = 'Gb3';
        end
    m = 54;
    note(k,:) = Note;
    case 55
        Note = 'G3 ';
    m =55;
    note(k,:) = Note;
    case 56
        if scale == 1
        Note = 'G3#';
        else
        Note = 'Ab3';
        end
    m = 56;
    note(k,:) = Note;
    case 57
```

```
        Note = 'A3 ';
    m = 57;
   note(k,:) = Note;
    case 58
        if scale == 1
        Note = 'A3#';
        else
        Note = 'Bb3';
        end
    m = 58;
    note(k,:) = Note;
    case 59
        Note = 'B3 ';
    m = 59;
    note(k,:) = Note;
    case 60
        Note = 'C4 ';
    m =60;
    note(k,:) = Note;
    case 61
        if scale == 1
        Note = 'C4#';
        else
        Note = 'Db4';
        end
    m =61;
    note(k,:) = Note;
    case 62
        Note = 'D4 ';
    m = 62;
    note(k,:) = Note;
    case 63
        if scale == 1
        Note = 'D4#';
        else
        Note = 'Eb4';
        end
    m = 63;
    note(k,:) = Note;
    case 64
        Note = 'E4 ';
    m =64;
    note(k,:) = Note;
    case 65
        Note = 'F4 ';
    m =65;
    note(k,:) = Note;
    case 66
        if scale == 1
        Note = 'F4#';
        else
        Note = 'Gb4';
```

```
        end
    m =66;
    note(k,:) = Note;
    case 67
        Note = 'G4 ';
    m = 67;
    note(k,:) = Note;
    case 68
        if scale == 1
        Note = 'G4#';
        else
        Note = 'Ab4';
        end
    m = 68;
    note(k,:) = Note;
    case 69
        Note = 'A4 ';
    m = 69;
    note(k,:) = Note;
    case 70
        if scale == 1
        Note = 'A4#';
        else
        Note = 'Bb4'
        end
    m =70;
    note(k,:) = Note;
    case 71
        Note = 'B4 ';
    m =71;
    note(k,:) = Note;
    case 72
        Note = 'C5 ';
    m =72;
    note(k,:) = Note;
    case 73
        if scale == 1
        Note = 'C5#';
        else
        Note = 'Db5';
        end
    m =73;
    note(k,:) = Note;
    case 74
        Note = 'D5 ';
    m = 74;
    note(k,:) = Note;
    case 75
        if scale == 1
        Note = 'D5#';
        else
        Note = 'Eb5';
```

```
    end
m =75;
note(k,:) = Note;
case 76
    Note = 'E5 ';
m =76;
note(k,:) = Note;
case 77
    Note = 'F5 ';
m =77;
note(k,:) = Note;
case 78
    if scale == 1
    Note = 'F5#';
    else
    Note = 'Gb5';
    end
m = 78;
note(k,:) = Note;
case 79
    Note = 'G5 ';
m =79;
note(k,:) = Note;
case 80
    if scale == 1
    Note = 'G5#';
    else
    Note = 'Ab5';
    end
m = 80;
note(k,:) = Note;
case 81
    Note = 'A5 ';
m = 81;
note(k,:) = Note;
    case 82
    if scale == 1
    Note = 'A5#';
    else
    Note = 'Bb5';
    end
m =82;
note(k,:) = Note;
case 83
    Note = 'B5 ';
m =83;
note(k,:) = Note;
case 84
    Note = 'C6 ';
m =84;
note(k,:) = Note;
case 85
```

```
        if scale == 1
        Note = 'C6#';
        else
        Note = 'Db6';
        end
    m =85;
    note(k,:) = Note;
    case 86
        Note = 'D6 ';
    m = 86;
    note(k,:) = Note;
    case 87
        if scale == 1
        Note = 'D6#';
        else
        Note = 'Eb6';
        end
    m =87;
    note(k,:) = Note;
    case 88
        Note = 'E6 ';
    m =88;
    note(k,:) = Note;
    case 89
        Note = 'F6 ';
    m =89;
    note(k,:) = Note;
    case 90
        if scale == 1
        Note = 'F6#';
        else
        Note = 'Gb6';
        end
    m = 90;
    note(k,:) = Note;
    case 91
        Note = 'G6 ';
    m =91;
    note(k,:) = Note;
    case 92.
        if scale == 1
        Note = 'G6#';
        else
        Note = 'Ab6';
        end
    m = 92;
    note(k,:) = Note;
    case 93
        Note = 'A6 ';
    m = 93;
    note(k,:) = Note;
        case 94
```

```
        if scale == 1
        Note = 'A6#';
        else
        Note = 'Bb6';
        end
    m =94;
    note(k,:) = Note;
    case 95
        Note = 'B6 ';
    m =95;
    note(k,:) = Note;
    case 96
        Note = 'C7 ';
    m =96;
    note(k,:) = Note;
    case 97
        if scale == 1
        Note = 'C7#';
        else
        Note = 'Db7';
        end
    m =97;
    note(k,:) = Note;
    case 98
        Note = 'D7 ';
    m = 94;
    note(k,:) = Note;
    case 99
        if scale == 1
        Note = 'D7#';
        else
        Note = 'Eb7';
        end

    m =99;
    note(k,:) = Note;
    case 100
        Note = 'E7 ';
    m =100;
    note(k,:) = Note;
    case 101
        Note = 'F7 ';
    m =101;
    note(k,:) = Note;
    case 102
        if scale == 1
        Note = 'F7#';
        else
        Note = 'Gb7';
        end
    m = 102;
    note(k,:) = Note;
```

```
            case 103
                Note = 'G7 ';
            m =103;
            note(k,:) = Note;
            case 104
                if scale == 1
                Note = 'G7#';
                else
                Note = 'Ab7';
                end
            m = 104;
            note(k,:) = Note;
            case 105
                Note = 'A7 ';
            m = 105;
            note(k,:) = Note;
        end

    end


ExactFreq = 2^((m-69)/12)*440;
Accuracy = Freq/ExactFreq*100;
    if Accuracy < 100
        Accuracy = -(100-Accuracy);
    elseif Accuracy > 100
        Accuracy = Accuracy - 100 ;
    end
    return;
end
```

D.5 The Harmonic Ratio MATLAB Function

The function `HarmonicRatio.m` determines the semitone value of the detected frequency according to the equation $Semitone = 12\log_2(Hz/440) + 69$, this information is then passed back to the main script.

Listing D.5: Harmonic Ratio Function.

```
%Harmonic Ratios
%Author: Jarred Oliver
%Date: 9 Aug 2019
%Description:
%This program calculates the harmonic ratio between the first detected
%frequency/notes and all other detected notes. If the ratio of the detected
%frequencies matches that of the known harmonic ratios then the singers are
%considered to be in harmony, if they dont match then the singers are
%not singing in harmony.

%-------------------------------------------------------------------------
function [result,bit] = HarmonicRatio(Freq,F)

if length(Freq) == 0
    result = 'no ';
end
for k = 1:length(Freq)
    if F == 1
    ratio = (Freq(k)/Freq(1));
    else
        ratio = (Freq(k)/Freq(end))
    end
    tol = 0.1;

        if ratio == 1
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 2/1-tol<ratio & ratio<2/1+tol
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 3/2-tol<ratio & ratio<3/2+tol
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 4/3-tol<ratio & ratio<4/3+tol
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 5/4-tol<ratio & ratio<5/4+tol
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 6/5-tol<ratio & ratio<6/5+tol
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 5/3-tol<ratio & ratio<5/3+tol
```

```
            result(k,:) = 'yes';
            bit(k,:) = 1;
        elseif 8/5-tol<ratio & ratio<8/2+tol
            result(k,:) = 'yes';
            bit(k,:) = 1;
        else
            result(k,:) = 'no ';
            bit(k,:) = 0;
        end
    if F == 1
        result = result(2:end,:);
    end
end
```

D.6 The Linear Predictive Coding MATLAB Function

The file LPCMain.m executes the script that calls the following functions:

1. Low Pass Filter
2. Peak Detection Algorithm
3. Linear Predictive Coding Function
4. Note Estimation
5. Harmony Ratio

Listing D.6: Cepstral Analysis Function.

```
%LPC Main V 4.01.1
%Author: Jarred Oliver
%Date 20 March 2019
%Last Updated 14/10/2019

%This program will take the signal input and
%calls LPCfunction to determine pitch information
%contained within the signal. It then calculates notes of the singer
%and if the notes detected are in harmony


clear;
close all
clc

%-------------------------------------------------------------------------
%read track
%
%          Filename = 'Hallelujah-Lead_Only_14975399900177.mp3';
%           Filename =  'Hallelujah-Baritone_Only_14975406008818.mp3'
%       Filename =          'Hallelujah-Tenor_Only_14975399951975.mp3';
    Filename = 'Hallelujah-Bass_14975400262231.mp3';
%      Filename = '880.wav';
          [SoundVec Fs] = audioread(Filename);
        scale = 2;
%-------------------------------------------------------------------------
%Setup values
         %single channel only for initial program
         SoundVec = SoundVec(:,1);
         bb = SoundVec;
%        SoundVec = SoundVec(1:Fs*30); %First 30 seconds for testing only
         Nx = 2^nextpow2(Fs);
         %initialise variables
```

```
        fund = []; %Fundemental Freq estimation
        F_disp = []; %Display F0 estimation
        a = []; %Annotation
        Filter_Order = 31; %Order of filter
        order = 150; %LPC Order
        Wc = 4000;

%-------------------------------------------------------------------------
%% Filter
%-------------------------------------------------------------------------
%spectral flattening
        signal_pe = SoundVec - 0.95*[0;SoundVec(1:end-1)];
%Low Pass Filter
        [signal] = lpfilter(Wc,signal_pe,Fs,Filter_Order);

%% Buffer
%-------------------------------------------------------------------------
        block = 2^12;
% Number of frames required for window
        stepsize = block*.75;
        numframe = floor(length(SoundVec)/ stepsize);
        hamming = (0.54-0.46*cos(2*pi.*(block)./(block)));

        start = 1; %Starting values for window function
%-------------------------------------------------------------------------
for x = 1: numframe
%       tic;
      if (start+block-1)>length(signal)
       break;
  end
     buffer_matrix = signal(start:start+block-1);
      out = zeros(length(buffer_matrix),1);
        %Normalise Audio for plotting
        n_buffer = buffer_matrix;
            if max(n_buffer) > abs(min(n_buffer))
                out = n_buffer*(1/max(n_buffer));
            else
                out = n_buffer*((-1)/min(n_buffer));
             end
    %Hamming Window
%-------------------------------------------------
        Windowed = hamming.*buffer_matrix';

    %% autocorrelation
%-------------------------------------------------------------------------

        acrsig = conv(Windowed,flip(Windowed));

        clear Windowed;
    %% Linear Prediction
%-------------------------------------------------------------------------
        tic
```

```
        [F, LPCR] = lpcBlock(Fs,Nx,order,acrsig);

%          T(x) = timeit(lpcBlock);
         A(1,x) = toc;
%           LPCR = LPCR/max(LPCR);
      index = find(F>2000&F<2001);
      F=F(1:index);
      LPCR=LPCR(1:index);
%-------------------------------------------------------------------------
    %Frequency locations
%-------------------------------------------------------------------------
    %Find all the peaks above 'x' threshold
        [mdv,idv] = axPeakDet(LPCR,1);

            LPC1 = abs(fft(acrsig,Nx)); %conduct fft of waveform
            LPC1 = LPC1(1:Nx/2);
%-------------------------------------------------------------------------
%Implement HPS
%-------------------------------------------------------------------------
[Freq0] = HarmonicProductSpectrum(acrsig,Nx,Fs);
%-------------------------------------------------------------------------
%Adjust information being displayed to user
%-------------------------------------------------------------------------
         M = length(idv);
           if(M==0)
               start = start + stepsize;
                continue;
           elseif M > 0
%          limit to first 5 frequencies detected
                    if M > 5
                        idv = idv(1:5);
                        mdv = mdv(1:5);
                        I = length(idv);
                                for k = 1:I
                                    if mdv(k) < 55
                                       mdv(k) = 1;
                                       idv(k) = 1;

                                    end
                                end
                     end
                    Freq = F(idv); %get freq value
%                    [result,Bit] = HarmonicRatio(Freq,1);
                   if length(Freq)<5

                       Freq(end+1:5) = 1;
                       Bit(end+1:5) = 0;
                    elseif length(Bit) == 0
                            Bit(1:5) = 0;
                            Freq(1:5) = 1;
                    end
```

```
                    [Note, Accuracy] = NoteEstimation(Freq,scale);
                    Fstore(x,:) = Freq;    %Store freq values
                    Hstore(x,:) = Bit;     %Store Harmony values

            end
%-------------------------------------------------------------------------
%Compare HPS and LPC Results
%-------------------------------------------------------------------------
[r c] = size(Freq);
for k = 1:c
    if Freq(k)> (Freq0-10)&Freq(k)<(Freq0+10)
        fund(x) = Freq0;
        F_disp = Freq0;
    elseif Freq(k)>(Freq0-10)&Freq(k)<(Freq0+10)
        fund(x) = 1;
    end
end
%-------------------------------------------------------------------------
%Plot Information
%-------------------------------------------------------------------------
        [R C] = size(Note);
        [F0_Note] = NoteEstimation(F_disp,1);
        %plot the waveform & label peak
        dim = [.7 .6 .3 .3];
        f = 'F0=';
         s = strcat(f,F0_Note);

        figure(1)
        subplot(2,1,1)
        plot(F,LPCR)
        hold on
        if length(mdv)>0
            for z = 1:R
                if Freq(z) < 2000
            plot(Freq(z),mdv(z),'r*')
            text(Freq(z), mdv(z), sprintf( Note(z,:))) %label

                end
            end
        end

        hold off
        set(gca, 'ylim', [-10 300])
        set(gca, 'xlim', [0 2000])
        title('Spectral Envelope')
        xlabel('Samples')
        ylabel('Magnitude')

        subplot(2,1,2)
        plot(out,'b')
        title('Time Domain Signal')
        xlabel('Time (Samples)')
```

```
        ylabel('Amplitude')

        delete(a);
        a = annotation('textbox',dim,'string',s,'FitBoxToText','on');

        S_start(x) = start/Fs;          %%%%%
        S_end(x) = (start+block)/Fs; %%%%%
       start = start + stepsize;

       if length(Fstore)<x & length(Hstore)<x
           Fstore(end:x,1:5)= 1;
           Hstore(end:x,1:5) = 1;

       end
end


%-------------------------------------------------------------------------
%Store Information
%-------------------------------------------------------------------------
%
%  S_start = S_start';
%  S_end = S_end';
% %Ensure Note Vector has same amount of rows as other stored vectors
% F0 = NoteEstimation(fund,scale);
% Note_1 = NoteEstimation(Fstore,scale);
% if length(F0) < length(Note_1)
%     F0(end:length(Note_1),1) = 1;
%     fund(end:length(Note_1)) = 1;
% end
% %Change Heading
% Frequency = Fstore;
% Harmony = Hstore;
% Start = S_start;
% End = S_end;
% Computation_Time = A';
% %Store in Table
% Tble = table(F0,fund',Note_1,Frequency,Harmony,Start,...
%     End,Computation_Time);
% writetable(Tble,'Halleluljah_LPC_bass.xls')
```

The function `lpcBlock.m` calculates the linear prediction coefficients of the input vector passed from the main program. It uses the built in MATLAB 'lpc' function to calculate the LPC coefficients and the prediction error. It then uses this output to transfer the coefficients to the power spectrum according to the following equation:

$$y(w) = \frac{\sigma_e^2}{|A(w)|^2}$$

It then passes the spectral envelope and frequency vector to the main program.

Listing D.7: Linear Predictive Coding Function.

```
%Linear Predictive Coding Block
%Author: Jarred Oliver
%Date: 10/05/2019
%Description:
%This function calculates the linear predictive coeffiecients
%supplied from the MainV3.m file. Once calculated the coefficients
%transposed to the LPC spectral envelope and the spectral envelope is
%passed back to the main program
%-------------------------------------------------------------------
%Input
%Fs = Sampling Frequency
%y = autocorrelated vector
%-------------------------------------------------------------------
%Output
%SEnv = Spectral Envelope
%F = Normalised Freq
%Speech = Frequency spectrum of singing signal
%-------------------------------------------------------------------
function [F SEnv, Product,e] = lpcBlock(Fs,Nx,order,y)
%for k = 1:Row

    [a e] = lpc(y,order);

F= 0:Fs/Nx:(Fs+Fs/Nx)/2;
Nxx = Nx/2 + 1; %halve total length


SEnv = 20*log10(abs(freqz(1,a,Nxx)));

%Create spectrum envelope
    %steps to transfer to spectral envelope
%    1 LPC1 = abs(fft(a,Nx));
%    2 LPC_1 = (e)./ LPC1(1:Nx/2+1);
%    3 LPC_1 = - 20*log10((LPC1));
%    4 LPC_1 = LPC_1(1:length(LPC_1)/2);
%end

end
```

D.7 The Cepstral Analysis MATLAB Function

The file `CepstralMain.m` executes the script that calls the following functions:

1. Low Pass Filter
2. Peak Detection Algorithm
3. Cepstral Analysis Function
4. Note Estimation
5. Harmony Ratio

Listing D.8: Cepstral Analysis Function.

```
%Cepstral Main Version 3.02.1
%Author: Jarred Oliver
%Date 20 March 2019
%Last Updated 3/10/2019

%This program will take the signal input and
%calls Cepstral function to determine pitch information
%contained within the signal. It then calculates notes of the singer
%and if the notes detected are in harmony
clear
close all
clc
%-------------------------------------------------------------------------
%read track

%  Filename =  'Hallelujah-Baritone_Only_14975406008818.mp3';
%      Filename ='Hallelujah-Tenor_Only_14975399951975.mp3';
    Filename = 'Hallelujah-Bass_14975400262231.mp3';
% Filename = 'Hallelujah-Lead_Only_14975399900177.mp3';
    [SoundVec Fs] = audioread(Filename);

%-------------------------------------------------------------------------
%Setup values
% %single channel only for initial program
    SoundVec = SoundVec(:,1);
    SoundVec = SoundVec(1:Fs*30);
    Filter_Order = 31;
%-------------------------------------------------------------------------

%FFT Sampling Rate adjustment
 Nx = 2^nextpow2(Fs);
```

```
%-------------------------------------------------------------------------
%spectral flattening
%-------------------------------------------------------------------------
        signal_pe = SoundVec - 0.95*[0;SoundVec(1:end-1)];

%-------------------------------------------------------------------------
%Low Pass Filter
%-------------------------------------------------------------------------
order = 30;
Wc = 4000;
[signal] = lpfilter(Wc,signal_pe,Fs,Filter_Order);
%-------------------------------------------------------------------------
% Buffer
%-------------------------------------------------------------------------
        block = 2^12;
% Number of frames required for window
        stepsize = block*.75;
        numframe = floor(length(SoundVec)/ stepsize);
        hamming = (0.54-0.46*cos(2*pi.*(block)./(block)));


        start = 1; %Starting values for window function
        count = 1;
        cnt = 1;
        F0 = [];
        fund = [];
        ff = [];
         F_0 = [];
         a = [];
         idc = [];
       m=[];
%-------------------------------------------------------------------------
for x = 1: numframe
%     tic;
        buffer_matrix = signal(start:start+block-1);
        out = zeros(length(buffer_matrix),1);
        %Normalise Audio for plotting
        n_buffer = buffer_matrix;
            if max(n_buffer) > abs(min(n_buffer))
                out = n_buffer*(1/max(n_buffer));
            else
                out = n_buffer*((-1)/min(n_buffer));
             end


%-------------------------------------------------------------------------
    %Hamming Window
%-------------------------------------------------------------------------
        Windowed = hamming.*buffer_matrix';
%-------------------------------------------------------------------------
% autocorrelation
%-------------------------------------------------------------------------
```

```
        acrsig = conv(Windowed, flip(Windowed));
%-------------------------------------------------------------------------
%Cepstral Analysis
%-------------------------------------------------------------------------
    tic;

    [FcHz,cc] = CepAn(acrsig,Fs,Nx);
 A(1,x)=toc;

    %normalise output
    cc = cc/max(cc);

%-------------------------------------------------------------------------
%Harmonic Product Spectrum
%-------------------------------------------------------------------------
     [Freq0] = HarmonicProductSpectrum(acrsig,Nx,Fs);
     F0(x) = Freq0;
%-------------------------------------------------------------------------
%Find Locations of Frequencies
%-------------------------------------------------------------------------
[mag,idx] = findpeaks(cc);

        %check peaks are above the average level
        count = 1;
        av = 0.8;
       for k=1:length(mag)
           if mag(k) > av

               idc(count) = idx(k);
               m(count) = mag(k);
               count = count +1;

           end
       end
%         disp(m)
%         pause
%-------------------------------------------------------------------------
%Adjust information being displayed to user
%-------------------------------------------------------------------------
                if length(idc) <= 0
                   idc(1:5) = 1;
                   m(1:5) = 0;
                    elseif length(idc)<5
%                     Bit(end+1:5) = 0;
                   idc(end+1:5) = 1;
                   m(end+1:5) = 0;
               elseif length(idc) > 5

                  idc = idc(1:5);
                m = m(1:5)
          end
```

```
        [Freq period] = pitchperiod(idc,m,Fs);
%-------------------------------------------------------------------------
%Compare Results betwen HPS and Cepstral Analysis
%-------------------------------------------------------------------------
        [r c] = size(Freq);
for k = 1:c
    if Freq(k)> (Freq0-10)&Freq(k)<(Freq0+10)
        ff(x) = Freq(k);
        fund(x) = Freq0;
        F_0 = Freq0;
    elseif Freq(k)>(Freq0-10)&Freq(k)<(Freq0+10)

        fund(x) = 1;
    end
end


%-------------------------------------------------------------------------
%Calculate X axis values and F0 Estimation
%-------------------------------------------------------------------------
[mm ii]= max(m); %find maximum magnitude out of remaining values
Q_freq = FcHz(idc(ii));
F_estimation = 1./Q_freq;
 [note, Accuracy] = NoteEstimation(F_estimation,1);
 [Fundemental] = NoteEstimation(F_0,1);
%-------------------------------------------------------------------------
%Plot information
%-------------------------------------------------------------------------
        Fstore(x,:) = Freq;    %Store freq values
%         [r c] = size(note);

        dim = [.7 .6 .3 .3];
        f = 'F0=';
        s = strcat(f,Fundemental);
%
        figure(1)
        subplot(2,1,1)
        plot(FcHz,cc,'b')
    if length(note)>0&length(m)>0&length(Q_freq)>0
        hold on
    for z = 1:length(Q_freq)
        plot(Q_freq(z),m(z),'r*')
        text(Q_freq(z), m((z)), sprintf(note(z,:)));%label
    end
        hold off
    end
        set(gca, 'xlim',[FcHz(1),FcHz(end)]) %sample 73 Hz to 8kHz
        set(gca,'ylim',[-1.1,1.1]);
        title('Cepstral Analysis')
        xlabel('Quefrency (ms)')
        ylabel('Magnitude')
```

```
        subplot(2,1,2)
        plot(out,'b')
        title('Signal')
        xlabel('Time (Samples)')
        ylabel('Magnitude')
        set(gca,'ylim',[-1.1,1.1]);
      delete(a);
      a = annotation('textbox',dim,'string',s,'FitBoxToText','on');

        S_start(x) = start/Fs;
        S_end(x) = (start+block)/Fs;


       if length(Fstore)<x & length(Hstore)<x
           Fstore(end:x,1:5)= 1;

       end
%-------------------------------------------------------------------------

start = start + stepsize;
   if (start+block-1)>length(SoundVec)
       break;
   end
%           clear m;
%           clear idc;

end
%-------------------------------------------------------------------------
%Store information
%-------------------------------------------------------------------------
 S_start = S_start';
 S_end = S_end';
%Ensure Note Vector has same amount of rows as other stored vectors
        Note_1 = NoteEstimation(Fstore,1);
        Fundemental_Note = NoteEstimation(fund,1);
        F = NoteEstimation(F0,1)

        if length(Note_1)<length(S_end)
            for k = length(Note_1)+1:length(S_end)
                    Note_1(k,:) = '   ';
            end
        end
        if length(Fundemental_Note)<length(Note_1)
            F(end+1:length(Note_1),1) = 1;
        end

%Change Heading
Frequency = Fstore;
% Harmony = Hstore;
Start = S_start;
End = S_end;
Computation_Time = A';
```

```
%Store in Table
Tble = table(F,F0',Note_1,Frequency,Start,...
    End,Computation_Time);
writetable(Tble,'Cep_bass.xls')
```

The function `CepAn.m` calculates the Cepstral coefficients for a given time domain signal. It takes the signal vector, sampling frequency and the desired number of Fourier Transform points and calculates the FFT, the log and inverse FFT of the signal vector before passing the coefficients and time vector back to the main program.

Listing D.9: Cepstral Analysis Function.

```
%Cepstral Analysis Function
%Author: Jarred Oliver
%Date: 26/04/2019
%Description:
%This function calculates the cepstral coefficients of the waveform
%supplied from MainV3.m, once calculated the cepstral coefficients are then
%returned to the main program.
%-------------------------------------------------------------------------
%Input
%Fs-Sampling Frequency
%Signal: The input signal vector
%-------------------------------------------------------------------------
%Output
%cc - cepstral coefficents

function [F,cc] = CepAn(Signal,Fs,Nx)
smin = 20;
smax = 420;
Signal = Signal - mean(Signal);
%xfft = abs(fft(Signal));
%Lft = log(xfft);
cep = real(ifft(log(abs(fft(Signal)))));

L = ceil((length(cep)+1)/2);
F1=(0:L-1)/Fs; %Quefrequency
F(1:(smax-smin+1))=F1(smin:smax);;
cc(1:(smax-smin+1)) = cep(smin:smax);

end
```

D.8 The AMDF MATLAB Function

The file AMDFMain.m executes the script that calls the following functions:

1. Low Pass Filter
2. Peak Detection Algorithm
3. Average Magnitude Difference Function
4. Pitch Period Estimation
5. Note Estimation

Listing D.10: Average Magnitude Difference Function Function.

```
%AMDF Main V 2.03.1
%Author: Jarred Oliver
%Date 20 March 2019
%Last Updated 4/10/2019


%This program will take the signal input and
%calls Average Magnitude Difference Funciton to determine pitch information
%contained within the signal. It then calculates the fundemental note of
%the singer.

clear;
close all
clc
%-------------------------------------------------------------------------
%read track

%           Filename = 'Hallelujah-Lead_Only_14975399900177.mp3';
%            Filename =  'Hallelujah-Baritone_Only_14975406008818.mp3'
      Filename ='Hallelujah-Tenor_Only_14975399951975.mp3';
%      Filename = 'Track_2.wav';
[SoundVec Fs] = audioread(Filename);

%-------------------------------------------------------------------------
%Setup values
%single channel only for initial program
SoundVec = SoundVec(:,1);
% SoundVec = SoundVec(1:Fs*30); %only used for testing
Filter_Order = 31;
%-------------------------------------------------------------------------
%FFT Sampling Rate adjustment
 Nx = 2^nextpow2(Fs);
%-------------------------------------------------------------------------
```

```
%spectral flattening
%------------------------------------------------------------------------
signal_pe = SoundVec - 0.95*[0;SoundVec(1:end-1)];
%------------------------------------------------------------------------
%Low Pass Filter
%------------------------------------------------------------------------
order = 30;
Wc = 4000;

[signal] = lpfilter(Wc,signal_pe,Fs,Filter_Order);

%------------------------------------------------------------------------
% Buffer
%------------------------------------------------------------------------
block = 2^12;
% Number of frames required for window
stepsize = block*.75;
numframe = floor(length(SoundVec)/ stepsize);


hamming = (0.54-0.46*cos(2*pi.*(block)./(block)));

start = 1; %Starting values for window function

avAc = [];
 ff = [];
        fund = [];
        F_disp = [];
        a=[];
%------------------------------------------------------------------------
for x = 1: numframe
%     tic;
    if start+block-1>length(SoundVec)
        break;
    end
     buffer_matrix = signal(start:start+block-1);
%------------------------------------------------------------------------
%Hamming Window
%------------------------------------------------------------------------
     Windowed = hamming.*buffer_matrix';

%------------------------------------------------------------------------
% autocorrelation
%------------------------------------------------------------------------
     acrsig = conv(Windowed,flip(Windowed));
%------------------------------------------------------------------------
%Average Magnitude Difference Function
%------------------------------------------------------------------------
     tic;
     xamdf = amdfV1(acrsig);
     A(1,x) = toc;
```

```
    [c d] = size(xamdf);
    Fa = (1:d)/Fs; %Samples to seconds
    ti = [];
    F_disp = [];

%-------------------------------------------------------------------------
%Harmonic Product Spectrum
%-------------------------------------------------------------------------
[Freq0] = HarmonicProductSpectrum(acrsig,Nx,Fs);
%-------------------------------------------------------------------------
%Find pitch period
%-------------------------------------------------------------------------
        [m, i] = axPeakDet(xamdf,2);

        count = 1;
        av = (max(diff(xamdf))-min(diff(xamdf)))-mean(m);
       for k=1:length(m)
           if m(k) < av
               i(count) = i(k);
              m(count) = m(k);
               count = count +1;
           end

       end

 [Freq period] = pitchperiod(i,m,Fs);



%-------------------------------------------------------------------------
%Compare results between HPS and AMDF
%-------------------------------------------------------------------------
[r c] = size(Freq);
for k = 1:c
    if Freq(k)> (Freq0-10)&Freq(k)<(Freq0+10)
        ff(x) = Freq(k);
        fund(x) = Freq0;
        F_disp = Freq0;
    else
        F_disp = 1;
        fund(x) = 1;

    end
end

%-------------------------------------------------------------------------
%adjust information being displayed to user
%-------------------------------------------------------------------------
          if length(Freq) > 0
           [result,Bit] = HarmonicRatio(Freq,1);
              if length(Freq)<5
```

```
                    Freq(end+1:5) = 1;
                    Bit(end+1:5) = 0;
                elseif length(Bit) == 0
                        Bit(1:5) = 0;
                        Freq(1:5) = 1;
                elseif length(Freq) > 5
                    Freq = Freq(1:5);
                    Bit = Bit(1:5);

                end

                 [Note,AA] = NoteEstimation(Freq,1);

                 Fstore(x,:) = Freq;    %%%%
%                  Hstore(x,:) = Bit;     %%%%

            end
%-------------------------------------------------------------------------
%Plot Information
%-------------------------------------------------------------------------
[F0_Note] = NoteEstimation(F_disp,1);
%           dim = [.7 .62 .3 .3];
%           f = 'F0=';
%            s = strcat(f,F0_Note);
%            figure(1)
%           subplot(2,1,1)
%       plot(Fa,xamdf,'b')
%
%       if period > 0
%           hold on
%
%
%           [R C] = size(Note);
%           for z = 1:R
%               plot(period(z),m(z),'r*')
%               text(period(z), m(z), sprintf(Note(z,:))); %label
%           end
%           hold off
%       end
%       title('AMDF Output')
%       xlabel('Time (S)')
%       ylabel('Amplitude')
%       set(gca,'ylim',[0 1.3]);
%        subplot(2,1,2)
%       plot(SoundVec(start:start+block),'b')
%       title('Input Waveform')
%       xlabel('Time (Samples)')
%       ylabel('Amplitude')
          delete(a);
%           a = annotation('textbox',dim,'string',s,'FitBoxToText','on');
          S_start(x) = start/Fs;        %%%%%
```

```
        S_end(x) = (start+block)/Fs; %%%%%
    start = start + stepsize;
% pause(.1)
% A(1,x) = toc;

end
%-------------------------------------------------------------------------
%Store and export to excel
%-------------------------------------------------------------------------
%  S_start = S_start';
%  S_end = S_end';
% % Ensure Note Vector has same amount of rows as other stored vectors
% Note_1 = NoteEstimation(Fstore,1);
% F0 = NoteEstimation(fund,1);
%         if length(fund)<length(S_end)
%             for k = length(fund)+1:length(S_end)
%                     F0(k,:) = '    ';
%             end
%         end
% %         buffer additional storage values to match length
%         if  length(F0)<length(Note_1)
%
%             F0(end+1:length(Note_1),1)=1;
%         end
% % Change Heading
% Frequency = Fstore;
%
% Start = S_start;
% End = S_end;
% Computation_Time = A';
% %Store in Table
% Tble = table(F0,Note_1,Start,End,Computation_Time);
% writetable(Tble,'AMDF_H_B_2.xls')
```

The function `amdfV1.m` calculates the average magnitude difference for a given input vector. It requires and input vector, sample minimum and sample maximum to be passed from the main program. Sample minimum and maximum are set to 30 and 300 samples respectively. This equates to a frequency range of 147 Hz to 1470 Hz.

Listing D.11: Average Magnitude Difference Function Function.

```
%Average mean difference Function
%Author: Jarred Oliver
%Date: 13/05/2019
%Description: This function calculates the amdf values for a paticular
%vector that is passed to it from the main program. if only the vector is
%passed to the funciton then the max and min samples are set to 420 and 20
%-------------------------------------------------------------------------
%Input:
%vector - Input singing signal
%smin  - minimum number of samples
%smax - maximum nnumber of samples
%-------------------------------------------------------------------------
%output
%amdfm - amdf
function [amdfm]= amdft(vector, smin, smax)
if nargin ==1
smin = 20; %2205Hz at Fs 44100
smax = 420; %105Hz at Fs 44100
end
L = length(vector);
    for x = smin:smax
amdf(x) = 1/L * sum(abs(vector(1:L-x+1)-vector(x:L)));
    end

    amdfm = amdf/max(amdf);
end
```

Appendix E

Project Timeline