Investigation of Digital Audio Manipulation Methods

A dissertation submitted by

B. Trevorrow

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Electrical & Electronic Engineering

Submitted: October, 2013
Abstract

This project investigates the application of signal processing techniques to three unique situations: 1) the extraction and suppression of a melody in a digital recording, 2) changing the tempo (time stretching) of a polyphonic musical recording without changing its pitch and 3) the reconstruction of audio waveforms which have been truncated (clipped) due to sample values exceeding full scale deflection.

Each of the applications under investigation requires the implementation of signal processing algorithms. The melody extraction/suppression application is performed through the implementation of digital filters which target the harmonic components of a musical note. Four filters are examined: the finite impulse response filter, the two pole resonant filter, the notch filter, and the infinite impulse response bandstop filter. The constant pitch time stretching application investigates two techniques: the phase vocoder and overlap-add. Finally the truncated waveform reconstruction application investigates the effectiveness of using low pass filtering and interpolation. All algorithms are implemented using a high level programming language and the performance of these algorithms is measured using a combination of quantitative and qualitative assessment.

The melody extraction/suppression algorithms are shown to be capable of extracting and suppressing a melody to a certain extent, their effectiveness being dependant on the type of instrument which is being filtered. The truncated waveform reconstruction techniques proposed show only limited success, capable of removing clipping in simple waveforms with only minor amounts of clipping. The three time stretching techniques all manage to successfully change the speed of a digital recording without changing its pitch, however each technique introduces their own distinct audible artifacts.
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.
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.

B. TREVORROW

0050055187

______________________________
Signature

______________________________
Date
Acknowledgments

I would like to thank my supervisor, Dr. John Leis, whose continuous support and insight helped lead me to approach these tasks in ways I would not have considered by myself.

B. Trevorrow

University of Southern Queensland
October 2013
Contents

Abstract i

Acknowledgments iv

List of Figures ix

List of Tables xii

Chapter 1 Introduction 1

1.1 Melodic Filter .............................................. 1

1.2 Truncated Waveform Reconstruction ......................... 2

1.3 Constant Pitch Time Stretching .............................. 2

Chapter 2 Melodic Filter 4

2.1 Musical Note Background Theory ........................... 4

2.2 Finite Impulse Response Filter ............................ 8

2.3 Resonant Two Pole Infinite Impulse Response Filter .......... 10

2.4 Infinite Impulse Response Notch Filter .................... 12

2.5 Infinite Impulse Response Band-Stop Filter ................. 13
Chapter 3  Truncated Waveform Reconstruction  

3.1  The Effect of Digital Audio Truncation  

2.5.1  Chebyshev Low Pass Filter Prototype  
2.5.2  Conversion to Band-stop filter  
2.5.3  Conversion to Discrete Time  
2.5.4  Limitations  

2.6  Implementation & Test Methodology  

2.6.1  Melody Specification  
2.6.2  Batch Melody Filter Implementation  
2.6.3  Limitations  
2.6.4  Test Methodology  

2.7  Block Filtering  

2.8  Audio Reconstruction  

2.9  Initial Results  

2.9.1  Finite Impulse Response Filter  
2.9.2  Resonant Two Pole Infinite Impulse Response Filter  
2.9.3  Infinite Impulse Response Notch Filter  
2.9.4  Infinite Impulse Response Band-Stop Filter  

2.10  Effect on Various Types of Instruments  

2.11  Further Work  

2.12  Conclusion
List of Figures

2.1 440Hz sawtooth wave time domain plot. ............................... 5
2.2 Frequency spectrum of a 440Hz sawtooth wave. ...................... 6
2.3 Frequency spectrum of a piano playing middle A. ...................... 6
2.4 Frequency spectrum of a piano playing middle A with background instrumentation. ......................................................... 7
2.5 Desired frequency response for extracting 440 Hz note. ............... 9
2.6 Localised desired frequency response for extracting 440 Hz note. .... 9
2.7 Localised actual frequency response of 440 Hz filter. .................. 10
2.8 Frequency response of resonant two pole filter centred at 440 Hz. ... 11
2.9 Local frequency response of two pole filter centred at 440 Hz. ....... 11
2.10 Local frequency response of notch filter centred at 440 Hz. ........... 13
2.11 Relationship between discrete time and continuous time using the bilinear transform at a 44100 Hz sampling rate ...................... 16
2.12 40 Hz wide 440Hz centred band stop digital filter frequency response. 17
2.13 Cascaded bandstop filter frequency response. .......................... 17
2.14 Melodic filter program overview. ........................................... 18
LIST OF FIGURES

2.15 Melody filter class diagram. .............................................. 20
2.16 440 Hz, $N = 1111$ FIR filter transient response. ................. 22
2.17 Filtered note envelope. .................................................. 23
2.18 Envelopes for melody suppression. ..................................... 24
2.19 Spectrogram of input audio waveform. ................................ 24
2.20 Spectrogram of FIR melody extraction filter output. ............... 25
2.21 Spectrogram of two pole resonant melody extraction filter output. 26
2.22 Spectrogram of Notch melody suppression IIR filter output. ....... 27
2.23 Spectrogram of bandstop filter output. ................................ 27
3.1 Theoretical 440Hz sinusoid amplified beyond full scale deflection 32
3.2 Truncated 440Hz sinusoid ................................................. 32
3.3 Clipping noise signal. .................................................... 33
3.4 Clipping noise spectrum. .................................................. 33
3.5 Truncated 440 Hz sinusoid. ................................................. 35
3.6 Filtered and unfiltered truncated waveforms showing mixing envelopes. 36
3.7 Reconstruction of a truncated sinusoid. ................................. 39
3.8 Frequency spectrums of reconstructed sinusoids ..................... 40
3.9 Reconstructed composite waveforms. .................................. 41
3.10 Reconstructed audio waveforms. ....................................... 42
3.11 Sinusoid with large number of samples clipped relative to period. 43
3.12 Reconstruction of composite waveform using MATLAB’s interp1 function. 45
3.13 Reconstructed audio waveform using MATLAB’s interp1 function. . . . 46

4.1 TD-PSOLA algorithm for time compression. . . . . . . . . . . . . . . . . 49

4.2 Time stretching by 0.8 and allowing for rhythm. . . . . . . . . . . . . . . 51

4.3 Time stretching using the SOLA algorithm. . . . . . . . . . . . . . . . . 52

4.4 440 Hz test pulse centred at 0.025 seconds. . . . . . . . . . . . . . . . . 53

4.5 Beat alignment of modified PSOLA algorithm. . . . . . . . . . . . . . . . 55

4.6 Beat alignment of SOLA algorithm. . . . . . . . . . . . . . . . . . . . . . 55

4.7 Beat alignment of phase vocoder algorithm. . . . . . . . . . . . . . . . . 56

4.8 Output of phase vocoder time compression. . . . . . . . . . . . . . . . . 56

B.1 Batch filter class diagram. . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

B.2 Input/Output class diagram. . . . . . . . . . . . . . . . . . . . . . . . . . . 68

B.3 Entry point and testing functions. . . . . . . . . . . . . . . . . . . . . . . 69
List of Tables

2.1  Filter summary ................................................................. 30

4.1  Summary of Audible Time Stretching Artifacts .................... 57
Chapter 1

Introduction

This project aims to investigate the application, effectiveness and limitations of signal processing techniques to three unique applications; a melodic filter, truncated waveform reconstruction and constant pitch time stretching.

1.1 Melodic Filter

When listening to a musical recording, a human is easily capable of distinguishing the different parts of a musical composition, such as vocals, solo instrumentation and background instrumentation, however from the perspective of a computer, it is much more difficult to isolate the various musical components in an audio recording. If a person wishes to extract one of these components from a musical recording, currently their options are limited. Through the development of a digital filter which is capable of targeting the specific parts in a digital audio recording corresponding to notes in a melody, it is hoped that the melody part in a musical audio recording can be separated from the background instrumentation.

The filter would work by exploiting the fact that musical notes correspond to discrete frequency values and contain harmonic frequency components at multiples of this frequency. It is assumed that the melody definition is already known, since it is very simple for a musician to transcribe a melody by listening to a recording. This definition can then be used to set the parameters of the filter, such that the unwanted
frequency components are attenuated.

This technique could see many potential uses in the field of music. Examples include, the ability of removing solo instruments from a recording, the ability of extracting a solo part (such as a violin, or vocals) for use in another composition as well as the ability to turn any arbitrary (noisy) sound recording into a melody.

1.2 Truncated Waveform Reconstruction

Truncation of a digital waveform occurs when it is amplified such that some sample values end up exceeding the full scale deflection value (0dB-FS) which results in those samples being set to that full scale deflection value (this is also commonly known as clipping). The effect of this on audio is to produce a harsh sounding distortion, the amount of distortion being proportional to the amount of clipped samples. Although it is ideal to avoid this situation from occurring in the first place, it is becoming increasingly common that digital audio music files are being sold and distributed with such distortion included. Modern digital audio manipulation software packages often include a facility for removing clipping, however details on the techniques used are usually not readily available (more so in the case of commercial software).

Two techniques are considered in this investigation. The first technique is to interpolate the value of the clipped samples using the surrounding non-clipped samples, effectively attempting to reconstruct the undistorted waveform. The second technique is to use a low pass filter on small sample frames centred on the clipping, effectively filtering out the localised noise distortion. The performance of these techniques are assessed in both the time and frequency domain and a qualitative assessment on the amount of distortion is also made.

1.3 Constant Pitch Time Stretching

In the electronic music scene, musical tracks are typically played such that the ending of one track is mixed into the beginning of the next track, which requires that the tracks be played at the same tempo. In the days of vinyl records, this was achieved
by speeding up or slowing down the rotation of one of the discs, however large enough changes in speed would result in a noticeable change in the playback pitch of the track. With digital recordings, re-sampling (i.e. changing the sample rate) achieves the same result, with a change in playback speed accompanied by a change in pitch. If the change in pitch is large enough, the resulting dissonance after mixing can be unsettling for some listeners (Zolzer & Amatriain 2002).

Methods currently exists in which the playback speed of a waveform can be altered while maintaining pitch, the two main approaches are to use either time domain or frequency domain scaling techniques. In commercial software, the technique most commonly employed is in the time domain, which relies on a form of synchronised overlap-add (SOLA) of signal excerpts (Laroche & Dolson 1999). When applied to complete musical recordings, artifacts such warbling, transient doubling or skipping\(^1\) and tempo modulation can be introduced, which can cause clashes when a time stretched track is mixed into another track.

An alternative to SOLA is to utilise a technique known as the Phase Vocoder, which is a frequency domain technique based on the Short Time Fourier Transform. The aim of this part of the project is to investigate the use of both the phase vocoder and overlap-add techniques as a means of providing constant pitch time stretching, implement the algorithms in software and measure their performance in a musical application, more specifically, in relation to (mis)alignment of beat accents.

\(^1\)Percussive instruments may occur twice in quick succession (when it shouldn’t) or disappear altogether
Chapter 2

Melodic Filter

2.1 Musical Note Background Theory

The musical note is the most basic building block in a musical composition, with a melody comprising a sequence of notes. In order to apply digital signal processing techniques to a melody, first the concept of a musical note must be understood thoroughly.

Musical notes (as seen on a musical staff or as the keys of a piano) correspond to discrete frequencies, these frequencies being the rate at which the vibrating component (such as the strings in a piano or a guitar) oscillates. In modern western music, the frequency of any given note can be calculated using the following equation:

\[ f_0 = A_0 2^{N/12} \] (2.1)

Where:

\( f_0 \) is the frequency of the note.

\( A_0 \) is the frequency of Middle A (440 Hz in modern western music).

\( N \) is the number of semitones (or number of keys on a piano) above or below the note Middle A.

Fourier theory states that any periodic waveform with a frequency of \( f_0 \) can be con-
sidered to be the superposition of sinusoids whose frequencies are multiples of $f_0$. The sum of this is what is known as a Fourier series. The Fourier series of a sawtooth wave (variations of sawtooth waves are common in electronic music) is shown in equation (2.2) below.

$$x(t) = \frac{2}{\pi} \sum_{N=1}^{\infty} (-1)^{N+1} \frac{\sin(2\pi N f_0 t)}{N}$$  \hspace{1cm} (2.2)

Figure 2.1 shows the first few cycles of a sawtooth wave synthesised using (2.2) with $f_0 = 440$ and up to $N = 50$.

![Figure 2.1: 440Hz sawtooth wave time domain plot.](image)

The frequency of each sinusoid in the Fourier series can be calculated using $Nf_0$. The case where $N = 1$ is known as the fundamental frequency, the cases where $N = 2$ or higher are known as harmonics. If the magnitude of each sinusoid in the 440 Hz sawtooth waveform is plotted onto a graph with frequency along the horizontal axis, the plot shown in Figure 2.2 is generated.

The plot shown in Figure 2.2 shows the frequency spectrum of the sawtooth waveform. As can be seen, the frequency spectrum of the 440 Hz sawtooth wave has a value of zero at all frequencies except those which are multiples of the fundamental frequency. This theory not only applies to the theoretical sawtooth waveform shown here, but also extends to all tonal musical instruments. Using a technique known as Short-Time Fourier Transform (STFT), the frequency spectrum of any waveform can be determined.
at any point in the waveform\footnote{The STFT of a waveform only gives an approximation to the frequency spectrum at a given time.}, for example, the frequency spectrum of a piano playing the note Middle A is shown in Figure 2.3. Note that the vertical axis of this plot is given in decibels with reference to full scale deflection.

As can be seen in Figure 2.3, the piano note only contains frequency components at the base frequency of 440 Hz and at multiples of this base frequency. A typical musical composition contains several notes playing simultaneously, with the frequency components of all notes played at any one time adding to the total frequency spectrum, for example,
a typical frequency spectrum of a piece of music with background instrumentation just as after a piano plays the note Middle A is shown in Figure 2.4.

![Frequency spectrum of a piano playing middle A with background instrumentation.](image)

In the field of signal processing, it is possible to design a digital filter which applies a non-uniform attenuation across a frequency spectrum. For the purposes of a melodic extraction filter, the idea is to design a filter which rejects (or otherwise greatly attenuates) all frequency components in a waveform except for those which correspond to a specific note, for example, to remove the background instrumentation from the spectrum shown in Figure 2.4, a filter would be required which rejects everything except the frequency components at 440 Hz, 880 Hz, etc. This forms the basis of the melodic filter and is similar to the process described in (Shalom, Shalev-Shwartz, Werman & Dubnov 2004) although the actual filter implementations used here are different.

There are several techniques which could be employed in designing a filter to suit these purposes. Four methods are considered and implemented, two techniques for extracting a melody from a digital audio recording (removing the background instrumentation) and two for suppressing the melody (removing the melody, leaving only background instrumentation).

The filters that were investigated for melody extraction are the Finite Impulse Response (FIR) filter and the resonant two pole Infinite Impulse Response (IIR) filter. The filters that were investigated for melody suppression include the IIR notch filter and the IIR
finite impulse response filter.

### 2.2 Finite Impulse Response Filter

A finite impulse response (FIR) filter is an implementation of a difference equation which generates an output waveform \( y(n) \) from an input waveform \( x(n) \). The general form of an FIR filter difference equation is given in equation (2.3) and the transfer function shown in equation (2.4)

\[
y(n) = \sum_{k=0}^{N-1} b_k x(n - k) \quad (2.3)
\]
\[
H(z) = \sum_{k=0}^{N-1} b_k z^{-k} \quad (2.4)
\]

The coefficients \( b_k \) of the FIR filter correspond to the impulse response of the desired frequency response, which is calculated using the inverse Fourier transform as follows:

\[
h_d(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_d(\omega) e^{jn\omega} \quad (2.5)
\]

In equation (2.5) the term \( H_d(\omega) \) is a vector which represents the desired frequency response and the vector \( h_d(n) \) represents the calculated filter coefficients. To extract a musical note, the desired frequency response is ideally rejection of all frequencies except for narrow bands centred at the fundamental and harmonic frequencies. For a 440 Hz note at a sample rate of 44100 Hz, the desired frequency response \( H_d(\omega) \) is shown in Figure 2.5.

Although difficult to see at the scale shown, the frequency response shown in Figure 2.5 is mirrored about the \( \omega = 0 \) axis and consists of narrow bands corresponding to a width of 10 Hz centred at the radian frequencies corresponding to the harmonics of the 440 Hz note. To help demonstrate this, the desired frequency response in the region from 0 to 2500 Hz is shown in Figure 2.6 with the frequency axis scaled to actual audio frequency.

The impulse response given in equation (2.5) is infinite in extent, however in order
2.2 Finite Impulse Response Filter

to implement a practical filter the impulse response $h_d(n)$ needs to be truncated to a finite number of samples $N$, to give an $N$ order FIR filter. For the note extraction application, because the pass band width of the desired response is so narrow, the required filter order will be quite high.

The frequency response of a digital filter can be calculated by substituting $z = e^{j\omega}$ into its transfer function and evaluating from $-\pi$ to $+\pi$. The frequency response of a 1111 order FIR filter with 10 Hz pass bands for extracting a 440 Hz note is shown in Figure 2.7.

What is interesting to note here is that the peak gain of the pass bands is not unity. This
occurs due to the truncation of the filter order as by definition, the Fourier transform of
the filter coefficients should equal the desired frequency response, however if only 1111
samples are specified, then the spacing of the discrete Fourier transform bins will be
\[ \frac{44100}{1111} = 39.7 \text{ Hz} \]
for a sampling frequency \( f_s = 44100 \), which is wider than the specified
pass bands of 10 Hz. Therefore, in order to correct for this either the filter order needs
to be increased or the pass band width \( f_b \) increased such that the equation \( f_b \geq \frac{f_s}{N} \) is
satisfied.

### 2.3 Resonant Two Pole Infinite Impulse Response Filter

The infinite impulse response (IIR) filter differs from the FIR filter in that the difference
equation is dependent on both the input waveform \( f(n) \) and the output waveform \( y(n) \)
and has a transfer function which contains poles other than \( z = 0 \).

The transfer function for the resonant two pole IIR filter is given below.

\[
H(z) = \frac{B(z)}{A(z)} = K \frac{1 - z^{-2}}{1 - 2R \cos(\Omega_c)z^{-1} + R^2z^{-2}} \tag{2.6}
\]

Where:

\( R \) is the radius of the poles and must satisfy \( 0 < R < 1 \).
2.3 Resonant Two Pole Infinite Impulse Response Filter

\( \Omega_c \) is the center frequency in radians per sample.

\( K \) is the normalising function, equivalent to \( \frac{1-R^2}{2} \).

The pass bandwidth of this filter can be controlled by modifying the value of \( R \), with values closer to unity giving narrower bandwidths. The frequency response of this filter with centre frequency of 440 Hz, \( R \) value of 0.999 and sample rate of 44100 Hz, is shown in Figure 2.8, with the response local to the centre frequency shown in Figure 2.9.

![Figure 2.8: Frequency response of resonant two pole filter centred at 440 Hz.](image)

![Figure 2.9: Local frequency response of two pole filter centred at 440 Hz.](image)

In contrast to the FIR filter, this filter only contains one peak at the center (fundamental) frequency, however in order to fully extract a musical note, this filter would also require peaks in the frequency response at the harmonic frequencies as well (880, 1320,
etc). To overcome this limitation, the superposition principle of waveforms can be applied where each frequency in a note is extracted from the input waveform separately and the output waveform generated by summing these filtered components.

2.4 Infinite Impulse Response Notch Filter

In order to suppress a musical note, a filter which performs the opposite of those discussed so far is required, which is a filter which passes all frequencies except those at the note’s fundamental and harmonic frequencies. The first type of filter investigated which can perform this function is the notch filter.

The transfer function of an IIR notch filter is shown in equation (2.7).

\[
H(z) = K \frac{1 - 2 \cos(\Omega_c) z^{-1}}{1 - 2R \cos(\Omega_c) z^{-1} + R^2 z^{-2}}
\]  

(2.7)

Where:

- \( R \) is the radius of the poles and must satisfy \( 0 < R < 1 \).
- \( \Omega_c \) is the center frequency in radians per sample.
- \( K \) is the normalising function, for this application \( K = 1 \) is sufficient.

Like the two pole resonant filter, the bandwidth of this filter can be controlled by modifying the parameter \( R \), with values closer to unity giving a narrower notch. It is not really possible to specify wide stopbands with this type of filter, as lower values of \( R \) result in significant attenuation in the passband. Therefore, for this application an \( R \) value close to unity is required so that only the frequencies corresponding to note harmonics are attenuated. For a notch filter with \( R = 0.999 \) and centre frequency of 440 Hz, the local frequency response is shown in Figure 2.10.

Again, like the two pole resonant filter, the notch filter as presented here is capable of only filtering a single frequency. To filter each of the harmonic frequencies, filters for each harmonic frequency should be run in cascade, such that the output of a 440 Hz notch filter becomes the input for an 880 Hz notch filter, which becomes the input for a 1320 Hz notch filter and so forth.
2.5 Infinite Impulse Response Band-Stop Filter

The notch filter described in the previous section contains a limitation in its extremely narrow bandwidth; it is best suited to filtering frequencies which are at exactly the centre frequency specified. However due to a number of reasons (such as instrument tuning, modulation of the note, or simply singing slightly off tune) the frequency of a musical note in an audio recording may not be at exactly the frequency expected. Therefore it is desirable to use a filter which will attenuate a narrow range of frequencies centred at the note frequencies to accommodate for this variation.

It is possible to design an IIR filter which is capable of rejecting a band of frequencies although the process is quite in-depth. An overview of this process is outlined as follows:

1. Design s-domain prototype low pass filter
2. Convert low pass filter to band-stop filter by substituting for s
3. Convert s-domain prototype to z-domain transfer function

Figure 2.10: Local frequency response of notch filter centred at 440 Hz.
2.5 Infinite Impulse Response Band-Stop Filter

2.5.1 Chebyshev Low Pass Filter Prototype

The Chebyshev prototype low pass filter has a cut-off frequency of 1 radian per second and is described by the following equation:

\[ |G(s)|^2 = \frac{K}{1 + \varepsilon^2 T_N^2 s} \]  (2.8)

The \( T_N \) term in (2.8) represents an \( N \)th order Chebyshev polynomial of \( s \). A Chebyshev polynomial of order \( N \) can be calculated using the following rules:

\[
\begin{align*}
T_0(s) &= 1 \\
T_1(s) &= s \\
T_N(s) &= 2sT_{N-1}(s) - T_{N-2}(s)
\end{align*}
\]  (2.9)

The equation given in (2.8) is in terms of the square of the magnitude of the transfer function \( |G(s)|^2 \). This can be transformed into the magnitude \( G(s) \) by recognising that the denominator polynomial has complex roots, which are made from complex pairs \( a \) and \( a^* \) and that \( aa^* = |a|^2 \). The magnitude transfer function is then found by cancelling out the complex poles containing positive imaginary values (which are unstable poles in the \( s \) domain). Expanding equation (2.8) with an \( N \)th order Chebyshev polynomial yields the general form of the low pass transfer function:

\[ G(s) = \frac{K}{a_0 s^N + a_1 s^{N-1} + \ldots + a_N s^0} \]  (2.10)

2.5.2 Conversion to Band-stop filter

To convert the low pass filter prototype into a bandstop filter, the following substitution is made:

\[ s = \Omega_b \frac{s}{s^2 + \Omega_c^2} \]  (2.11)

Where:
\( \Omega_b \) is the stop band width.

\( \Omega_c \) is the center frequency of the stop band.

Substituting equation (2.11) into (2.10) yields:

\[
G(s) = \frac{K}{a_0(s\Omega_b)^N + a_1(s\Omega_b)^{N-1} + \ldots + a_1(s\Omega_b)^0}
\]

(2.12)

\[
G(s) = \frac{K}{a_0(s\Omega_b)^N(s^2 + \Omega_c^2)^0 + a_1(s\Omega_b)^{N-1}(s^2 + \Omega_c^2)^1 + \ldots + a_N(s\Omega_b)^0(s^2 + \Omega_c^2)^N}
\]

2.5.3 Conversion to Discrete Time

The Chebyshev filter designed so far has been completely in continuous time (s-domain). In order to implement it as a digital filter, it needs to be converted to discrete time (z-domain). To do this, the bilinear transform is used. The bilinear transform is defined as the following substitution:

\[
s = \frac{2}{T} \left( \frac{z - 1}{z + 1} \right)
\]

(2.13)

To consider how this relates discrete time to continuous time, the substitutions \( s = j\Omega \) and \( z = e^{j\omega} \) into equation (2.13) are made to give:

\[
\begin{align*}
    j\Omega &= \frac{2}{T} \left( \frac{e^{j\omega} - 1}{e^{j\omega} + 1} \right) \\
    \Omega &= \frac{2}{T} \tan \frac{\omega}{2} \\
    \omega &= 2 \arctan \frac{\Omega T}{2}
\end{align*}
\]

(2.14)

(2.15)

Equation (2.15) describes how the bilinear transform relates discrete frequency and continuous frequency. Figure 2.11 is a plot of equation (2.15) with \( T \) set to 44100\(^{-1}\).

The dashed line in Figure 2.11 represents the ideal relationship between continuous and discrete time while the solid line represents the actual relationship. What is clear is
that the bilinear transform is not an exact conversion between continuous and discrete time with “warping” occurring at higher frequencies. To compensate for this frequency warping, equation (2.14) can be used where a desired z-domain centre frequency $\omega_c$ can be allowed for by calculating a compensated s-domain centre frequency $\Omega_c$ which the s-domain filter is then designed around.

2.5.4 Limitations

The process described so far is to be implemented on a digital computer, which utilises floating point numbers to represent continuous variables. However this introduces a significant cause of potential error where operations on numbers with vastly differing orders of magnitude are concerned since the mantissa of a floating point number can only contain a finite number of digits (IEEE 754 Group 2008). Consider the implementation of equation (2.12), specifically the term $a_N(s\Omega_b)^{0}(s^2+\Omega_c^2)^N$. If a third order filter was specified with a centre frequency of 440 Hz, then when this term is expanded out the first term in the resulting polynomial will be 1, while the last term (the constant) will be $4.4647 \times 10^{20}$! The implication here is that certain terms of the polynomial will be so insignificant that the accuracy of any operations on the polynomial will be affected. This also effectively puts a limit on the filter order which can be specified as this difference in magnitude greatly increases with larger filter orders $N$. 

Figure 2.11: Relationship between discrete time and continuous time using the bilinear transform at a 44100 Hz sampling rate
Due to these limitations, the bandstop filter can be no more than a 3rd order filter, however the actual filter that was implemented was a 2nd order filter. The frequency response of a 2nd order digital band stop filter, as produced following the design steps outlined in the previous sections, is shown in Figure 2.12.

![Figure 2.12: 40 Hz wide 440Hz centred band stop digital filter frequency response.](image)

This filter was designed with a stop band width of 40 Hz and a centre frequency of 440 Hz. As can be seen, the attenuation in the stop band of this filter is not very ideal, however a simple method of improving the frequency response is to cascade this filter, that is recursively filtering the output a number of times. The frequency response of this same filter, except now cascaded four times, is shown in Figure 2.13.

![Figure 2.13: Cascaded bandstop filter frequency response.](image)

As can be seen, the response is now much better with a band of about 8 Hz where the attenuation is below -96dB, which would attenuate a 16-bit sampled full scale amplitude sinusoid below the quantisation “noise floor”.
2.6 Implementation & Test Methodology

The melodic filter combines several complex processes, the first is melody specification, the second is filter design/application and the third is waveform reconstruction. During the early stages of the project, MATLAB was used to develop automatic algorithms which are capable of generating the necessary filter coefficients for any given frequency. This was recommended since MATLAB contains many built-in functions (such as filter(), roots() and conv()) which expedite the filter design process. However, while MATLAB is suitable for the filter design stage, for the actual implementation of the filters in a melody application, the C# programming language was used. This allowed easier implementation of multi-threading, class hierarchy and also of the MIDI to melody converter since MATLAB isn’t suited to the manipulation of binary files.

The developed prototype is a simple console program run from a command line interface. The prototype accepts a number of command line arguments, the minimum of which include input file, output file name, input MIDI and finally the type of filtering to perform. The operation of the program is summarised in Figure 2.14.

![Figure 2.14: Melodic filter program overview.](image)
2.6 Implementation & Test Methodology

2.6.1 Melody Specification

To define a musical note for the purposes of the melodic filter, the start time, end time and fundamental frequency need to be specified at the very least. These could be specified manually, that is by calculating the frequency of each note and finding the start and end times in an audio editor, however this would be extremely tedious and there would be large potential for error. To specify the melody for use in the melodic filter, MIDI files are utilised.

MIDI (Musical Instrument Digital Interface) is a communications protocol which is typically used to allow electronic instruments to communicate with each other. A MIDI file is a binary file which contains MIDI commands such as “note on” and “note off” as well as the times at which these instructions are to occur. By referring to the MIDI specification, it is possible to extract the instructions corresponding to the notes in a melody and use these to calculate note frequencies. MIDI has been specifically chosen for this purpose since there are many freely available programs which are capable of exporting MIDI files, and transcribing a melody in one of these programs would be far less tedious than examining an audio file in an audio editor.

2.6.2 Batch Melody Filter Implementation

To implement the four types of filters described in the theory section, Object Oriented Programming (OOP) was utilised. This allows each type of filter to be designed as a unique class which only needs to define the generation of the individual filter coefficients, while tasks which are common to all of the filters, such as the allocation of audio frames to notes, are provided by higher level classes. Figure 2.15 shows the class hierarchy that was used to implement the four types of filter.

The classes shown in Figure 2.15 which are shown with a dashed outline represent abstract classes, which exist only to provide common functionality to the lower level classes. Each of the four filters are types of BatchMelodyFilter, which accepts an input waveform, provides a method for performing the melody aligned filtering (with filter design differing by type) and provides a method for reconstructing the filtered components into a single waveform.
2.6 Implementation & Test Methodology

2.6.3 Limitations

While C# has been specified for the implementation of the melodic filter, the base libraries used with this programming language lack much of the mathematical functions needed for digital signal processing when compared to MATLAB. For the first three types of filter this isn’t much of an issue as it is straightforward enough to implement difference equations in C#, however for the bandstop filter, it is necessary to calculate the roots of a polynomial, which is a non-trivial task. Although it is possible to source a polynomial library which can be used with C#, the accuracy and efficiency of these algorithms could be called into question. To avoid this, the filter coefficients can be generated using MATLAB and exported to a binary file. The C# program will then open this binary file and extract the desired filter coefficients.

The MIDI specification defines 128 possible notes, therefore filter coefficients for each of these possible notes need to be generated for the fundamental frequency of these notes and up to 50 of the note’s harmonic frequencies (provided the harmonic frequencies are below the Nyquist frequency). This works out to be 4859 different filters, each with 14 double precision floating point coefficients (for a 3rd order Chebyshev bandstop filter). While this may seem like a large amount, the memory requirements to maintain a database of these filters is insignificant when compared to the amount required to store filtered blocks at 44100 Hz, also keeping track of which coefficients belong to which note is simple using dictionary objects provided in C#.
2.6.4 Test Methodology

To confirm whether the melody filters are working according to design, spectrograms of the output waveforms are used, which ideally should only contain (or omit) the harmonic components of the musical notes in a melody. To actually determine how effective each type of filter is though, qualitative assessment of the output waveforms was performed, which required listening to the output waveforms and making a subjective assessment.

2.7 Block Filtering

The filters which have been discussed so far have been time invariant, that is, their parameters don’t change with respect to time. However a musical note is a time localised phenomena, its duration is only a fraction of an entire recording. Therefore the filters need to be able to adapt to each note in a melody, with filter coefficients being recalculated for each note and applied for only the duration of the note. While it is possible to change the coefficients of the FIR filter in real time such that it provides the right frequency response at the right time, the same cannot be said of the IIR filters which rely on additive synthesis to extract all frequency components of a note. For this reason, block filtering has been utilised to extract each note individually.

Put simply, block filtering is extracting a group of samples corresponding to a single note, then applying a filter which suits that note to the block. It is not enough to simply extract only the samples within the note duration, it is also necessary to extract a number of samples before and after the beginning and ending of the note as well. This serves two purposes, it allows windowing of the blocks (which will be discussed in the following section) plus it mitigates the transient response of the filter. To illustrate the effect of a filter’s transient response on a sample block, the time domain plot of a 440 Hz saw wave after having an $N = 1111$ FIR note filter applied to it is shown in Figure 2.16.

The fundamental frequency of the filter used in Figure 2.16 is the same as the note so ideally the output should be the same as the input (a 440 Hz saw wave), however there is a finite amount of time required before this happens because samples before
2.8 Audio Reconstruction

For the melody extraction application, the ideal reconstructed result is a waveform with zero value at all times when a note is not playing or a filtered block corresponding to a note that is playing, however it is not enough to simply add the filtered blocks on top of a zero waveform. This is because the start and end of the blocks contain abrupt transitions to zero, which will manifest as audible clicks at these transitions. To allow for this, the filtered blocks are made to be slightly longer than the note duration and an “envelope” is used as shown in Figure 2.17.

Essentially, the note fades in from zero before the start of the note and fades out to zero after the end of the note. To achieve this, the filtered block is multiplied by the

$t = 0$ (corresponding to the start of a block) are treated as being zero value. This is the transient response of the filter and for FIR filters has a sample length equivalent to the filter order and for IIR filters, narrower bandwidth requirements produce longer transient responses. In order to allow for the transient response, a sufficient number of samples before the start of the note is included in the block, which are then discarded after the filter operation is performed.

Figure 2.16: 440 Hz, $N = 1111$ FIR filter transient response.
In Equation 2.16, the parameters $t_{\text{start}}$ and $t_{\text{end}}$ refer to the time instants where the note starts and ends respectively and the parameters $t_{\text{atk}}$ and $t_{\text{rel}}$ refer to “attack” and “release” times. The attack and release times represent the amount of time taken to completely fade in or out and can vary depending on the type of sound being extracted, for example, an instrument whose sound decays slowly (such as a piano or acoustic guitar) would require a longer release time than one whose sound decays quickly (such as a violin, woodwind instrument or vocals).

For the melody suppression application, the reconstruction approach needs to be modified slightly since the filtered blocks need to replace the melody in the source recording. This requires transitioning from the source recording, into the filtered block, then back into the source recording. To do this, an envelope is also applied to the source recording which is equivalent to $1 - k$ as shown in Figure 2.18. Because the sum of the two envelopes is always unity, the volume of the background should remain constant during the transition and throughout the note duration.
2.9 Initial Results

To confirm that the implemented filters were performing as expected, spectrograms were used. A spectrogram is a plot of frequency spectrum versus time, which is generated by plotting the Short Time Fourier Transform at successive short intervals. The spectrogram of the source file that was used for these initial tests is shown in Figure 2.19.

Time is shown on the horizontal axis and frequency is shown on the vertical axis. Darker shades represent strong spectral intensity. The spectrograms contained in this document were generated using the freely available Audacity audio editing software, with a window size of 4096 samples and a Hanning envelope window.
2.9 Initial Results

2.9.1 Finite Impulse Response Filter

The spectrogram of the result of the FIR melody extraction filtering is shown in Figure 2.20.

![Figure 2.20: Spectrogram of FIR melody extraction filter output.](image)

As can be seen, at any point in time the spectrogram shows islands of intensity with vertical centre spacing equal to the fundamental frequency found at the bottom. This corresponds to the frequency spectrum of the musical note playing at this point in time and shows that the output of the filter successfully rejects those frequency components which do not correspond to the note fundamental and harmonic frequencies. It should be noted that this filter took several minutes to generate this result on a relatively modern computer, which is due to the large number of terms (1111 to be precise) appearing in the difference equation.

Examining Figure 2.20 closely reveals extra spectral content at the beginning and ending of each note. This is due to the envelopes at the beginning and ending of each note causing spectral leakage. Although this results in unwanted spectral content, the audible effect of this is much less severe than if rectangular windows (with abrupt ends) were to be used, so this extra spectral content can safely be ignored.

2.9.2 Resonant Two Pole Infinite Impulse Response Filter

A spectrogram of the output of melody extraction using the resonant two pole IIR filter, is shown in Figure 2.21.

In terms of stop band rejection, the performance is better than that shown Figure 2.20,
2.9 Initial Results

in that only very narrow bands surrounding the harmonic frequencies remain in the output. The plot shown in Figure 2.21 corresponds to a resonant filter with $R = 0.999$, however, there is also more unwanted spectral content occurring during the notes, which isn’t due to the start and end envelopes. This extra spectral content is due to the long transient response time of using such a narrow bandwidth filter, where not enough samples were allowed for the transient response time of the filter, resulting in blocks whose ends do not line up properly. This could be allowed for by simply increasing the frame size of the filter blocks, however it should be mentioned that long filter transient response times are not ideal for notes which contain short transients at the beginning of a note, such as a piano. Therefore, to improve upon this situation, the final two pole resonant filter which was implemented has the bandwidth parameter set to $R = 0.995$.

It should also be mentioned that this filter is capable of processing audio several orders of magnitude quicker than the FIR filter, with this result being generated in mere seconds.

2.9.3 Infinite Impulse Response Notch Filter

A spectrogram of the output of melody suppression using the IIR notch filter is shown in Figure 2.22.

Due to the very narrow bandwidth of the notches, it is difficult to see the effect of the filter due to the vertical resolution of the spectrogram being less than the bandwidth of the filter. Playback of the audio waveform does confirm that that melody is attenuated, however some of the melody does remain in the output even though the audio tested
2.9 Initial Results

Figure 2.22: Spectrogram of Notch melody suppression IIR filter output.

was a digitally produced piece of music whose frequencies should have been in tune.

The plot shown here corresponds to a notch filter whose bandwidth parameter was set at \( R = 0.999 \), however the final implementation that was used had this parameter set to \( R = 0.995 \), which results in some attenuation of the passband, but should reject an in tune melody.

2.9.4 Infinite Impulse Response Band-Stop Filter

A spectrogram of the output of melody suppression using the IIR bandstop filter is shown in Figure 2.23.

Figure 2.23: Spectrogram of bandstop filter output.

What is immediately apparent is that the rejected frequency bands in this plot are much more visible than those shown in Figure 2.23, appearing as horizontal lines of no spectral intensity with uniform vertical spacing at a given time. The frequency bands are also well defined in that the transition between pass band and rejection band is
sufficiently sharp such that the background audio is not attenuated. Playback of the audio also confirms that most of the melody which remained in the $R = 0.999$ notch filter has been removed through the use of the bandstop filter.

### 2.10 Effect on Various Types of Instruments

Some of the subjective tests that were performed included extraction of a clarinet melody from strings and harp accompaniment, followed by extraction of a piano melody from strings accompaniment. In this situation it was found that the clarinet extraction performed considerably well, while the piano extraction was not as good. This was due to the fact that the clarinet melody consisted of notes whose sounds faded away almost immediately after each note ended, while the piano melody’s notes took time to fade away. With the current implementation, it is possible to specify longer note release times in order to accommodate for the note decay time, however this causes considerable filter overlap, which results in more unwanted frequencies appearing in the output.

When the melody extraction application was tested on extracting a vocal melody, it was found that while the melody itself does get extracted and the voice still distinctly belongs to the singer, the actual words themselves are less discernible after extraction. This is due to the fact that speech is made up of voiced and unvoiced parts and while the voiced part (which constitutes the majority of English language speech) does follow the harmonic model detailed in section 2.1, the unvoiced part of speech does not and is more white noise-like in nature (Hu & Wang 2008). This affects the melody extraction of vocals more than melody suppression of vocals, since the voiced part of a vocal melody contains the bulk of the speech energy, with the unvoiced parts typically consisting of short transients with lower energy.

### 2.11 Further Work

The four types of filter which were used in this application are only a few of many which could be utilised. From the types which were used here, the natural progression would be to implement a bandpass filter in much the same way the bandstop filter was created,
with the only difference being the prototype substitution. This could potentially give an extraction filter with better stopband gain than that given by the two pole resonant filter. Also, as an FIR filter was used for melody extraction, it would be just as simple to create an FIR filter for melody suppression, whose frequency response would consist of unity gain at all frequencies except for zero gain at narrow bands surrounding the harmonic frequencies.

With regard to the bandstop filter (and possible future bandpass filter), a major issue that was encountered concerned instability due to floating point precision errors. This was due to the fact that the polynomials in the difference equation numerator and denominator consisted of terms of differing orders of magnitude. It is possible to actually implement these high order filters as second order sections, which are essentially first and second order filters in cascade, and are built using poles and zeros, rather than numerator and denominator. It is considerably trickier to implement and handle such filters however, so they weren’t used here due to time constraints.

There are also alternative methods which have been proposed for extraction of an audio melody from a musical audio recording, one such example is given in (Raphael 2008), however due to time constraints, it was not possible to compare the algorithms implemented here with any of these methods.
2.12 Conclusion

The melody filter has been implemented and has been shown to be capable of extracting or suppressing a melody in an audio recording, although the results are dependent on the type of instrument being filtered, with notes whose sounds are continuous and fade away quickly being more suited to the filter than notes with long decay times or vocal notes.

The different types of filters used in the melody extraction application are summarised in Table 2.1.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Application</th>
<th>Process Time</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>Extraction</td>
<td>very long</td>
<td>wide</td>
</tr>
<tr>
<td>Resonant</td>
<td>Extraction</td>
<td>very short</td>
<td>very narrow</td>
</tr>
<tr>
<td>Notch</td>
<td>Suppression</td>
<td>short</td>
<td>very narrow</td>
</tr>
<tr>
<td>Bandstop</td>
<td>Suppression</td>
<td>short</td>
<td>narrow</td>
</tr>
</tbody>
</table>
Chapter 3

Truncated Waveform Reconstruction

While declipping tools are not uncommon to digital audio workstation software, there is only limited literature devoted to the subject, and the topic has attracted limited research interest (Adler, Emiya, Jafari, Elad, Gribonval & Plumbley 2011). This chapter explores some simple methods that could be utilised to perform truncated waveform reconstruction, in order to gain an understanding of the effectiveness and limitations of attempting to correct waveforms which have been damaged due to truncation.

3.1 The Effect of Digital Audio Truncation

For 16 bit signed integer sampling, which is the format used in compact disc digital audio, sample values can only take on $2^{16} = 65536$ discrete sample values, with the values -32768 and +32767 being the full scale deflection values. If a signal is amplified such that some of the amplified sample values exceed these full scale deflection values, the result is a sample value which cannot be represented with a 16 bit signed integer. When this occurs, these samples are simply set to the full scale deflection value which was exceeded, resulting in truncation of the waveform, more commonly known as clipping. To demonstrate what effect this has on a waveform, a simple example will be considered. Figure 3.1 shows a simple 440 Hz sinusoid signal which has been amplified such that its peak value exceeds the full scale deflection value by a factor of 1.2 (1.6dB-
3.1 The Effect of Digital Audio Truncation

FS). Figure 3.2 shows the signal as it would be stored in a digital audio format after truncation.

![Figure 3.1: Theoretical 440Hz sinusoid amplified beyond full scale deflection](image1)

![Figure 3.2: Truncated 440Hz sinusoid](image2)

The waveform shown in Figure 3.2 is clearly non-sinusoidal and will therefore contain harmonics of the base frequency. This is unwanted noise distortion, which has been introduced due to the signal truncation. If the sinusoid shown in Figure 3.1 is designated as signal $x_S$ and that shown in Figure 3.2 is designated $y$ then using the superposition principle of waves, the noise signal $x_N$ can be calculated using $x_N = x_S - y$. The time domain plot of this noise signal is shown in Figure 3.3. Using STFT, an approximation to the frequency spectrum of the noise signal can be found, this is shown in Figure 3.4.

From Figure 3.3 and Figure 3.4 it can be seen that clipping introduces unwanted noise
into the signal, for this simple example the noise contains odd numbered harmonics as well as a component at the base frequency of the sinusoid. It should be noted, for periodic waveforms, the noise introduced is harmonic and follows the definition of the musical note given in the melodic filter section and therefore could theoretically be used as a musical instrument. However if clipping is present in an audio recording, which would contain frequency components (and therefore noise) across most of the frequency spectrum, it is most likely an unwanted by-product of over-amplification. Therefore a means of reducing the distortion noise is desirable if it is infeasible or otherwise not possible to recreate the undistorted recording. Two methods were investigated to achieve this, the first being the use of low pass filtering and the second being polynomial
3.2 Low Pass Filter

This technique aims to reduce the amount of noise in a truncated waveform by applying a low pass filter to the regions where the noise (due to clipping) occurs. In order to understand why a low pass filter is used in this way, it is necessary to know about ideal reconstruction.

3.2.1 Sinc Interpolation

Derived from Shannon’s sampling theorem, sinc interpolation states that the exact value a waveform would be expected to take between sample instants in a band limited sampled waveform can be determined through the sum of the sample values weighted against a sinc function.

\[ x(t) = \sum_{n=-\infty}^{\infty} y(nT) \text{sinc}(f_s(t - nT)) \]  \( (3.1) \)

If applied to a simple truncated sine wave as shown in Figure 3.5 then the value of the samples in between the two marked points is what needs to be calculated, with the samples in between to not be included as part of the weighting.

If the number of samples between where the clipping starts and ends is \( M \) samples then in order to use (3.1) the samples which are multiples of \( M \) are the only ones which can be used in the interpolation. This essentially means interpolation between the samples in a waveform with a lower sampling rate, however the values as they appear in the original waveform cannot be used directly because the sampling rate of the interpolated waveform is now \( \frac{f_s}{M} \) which will contain aliasing from the original waveform in most cases.

The interpolation formula (3.1) is derived from a frequency response which is flat with a value of unity from zero to the Nyquist Frequency and zero for frequencies above this. This essentially gives what is known as a “brick-wall” type low pass filter and means interpolation.
that sinc interpolation can be provided by a low pass filter with cut off frequency at the Nyquist Frequency.

If this is applied to the waveform shown in figure 3.5, which is a 440 Hz clipped sinusoid with a sampling rate of 44100 Hz and $M = 25$, then this corresponds to a low pass filter with cut off frequency of 1102.5 Hz. While this works well in the simple case of a clipped sinusoid, in practice the waveform is also most likely to contain spectral content at greater than $\frac{f_s}{2M}$.

### 3.2.2 Implementation Details

The major downside of applying a low pass filter to remove clipping noise is that it will also attenuate desired frequency components which are also above the filter cut-off frequency. To minimise this loss, an algorithm is required which applies a low pass filter to only the regions where clipping occurs. This filtered region is then mixed into the unfiltered waveform using a raised cosine envelope, as shown in Figure 3.6. The top waveform shows the unfiltered, truncated waveform and the bottom shows the waveform after the low pass filter is applied. The red dashed line represents the envelope which is applied to each waveform before they are mixed together.

The low pass filter that is used is a 2nd order Chebyshev filter. This can be created using the process described in section 2.5, with the prototype to bandstop conversion
being made by making the substitution \( s = \frac{i}{\Omega_c} \). With the filters discussed so far, no mention has been made of filter group delay, which has the effect of introducing a phase shift into the filtered waveforms. Obviously if the filtered waveform ends up being phase shifted, then the approach described so far will not work as the peaks in the resulting filtered waveform won’t line up with the clipping regions in the unfiltered waveform. The phase shift can be allowed for by reversing the filtered waveform, then applying the filter again (inducing reverse phase shift), then reversing the final waveform. This causes the phase shift to be cancelled out, but also doubles the response of the filter, although this isn’t an issue as faster stop band roll-off is actually desirable in this application.

The final consideration that needs to be made is that the amplitude of the filtered waveform will not match the amplitude of the desired waveform. This can be seen in Figure 3.6 where the filtered waveform has an amplitude which is lower than the clipping value. To get around this, the filtered waveform needs to be multiplied by a gain, such that the filtered waveform aligns better with the unfiltered waveform. To calculate this filtered waveform gain, the mean sample value of the filtered and unfiltered waveforms in the envelope region is calculated. The mean is used in this manner since even though the unfiltered waveform may contain high frequency components in this region which

![Figure 3.6: Filtered and unfiltered truncated waveforms showing mixing envelopes.](image-url)
won’t correspond to the filtered waveform, they will be cancelled out by taking the average, so the two waveforms can be compared directly.

3.3 Polynomial Interpolation

For interpolation of $N$ known samples, the value at any arbitrary time $t$ can be calculated using the Lagrange interpolation shown in equation (3.2).

$$x(t) = \sum_{k=0}^{N} L_k(t)x(t_k)$$ (3.2)

where

$x(t_k)$ is the value at sample $k$

$L_k(t)$ is the Lagrange polynomial corresponding to sample $k$

The Lagrange polynomial $L_k(t)$ is defined as

$$L_k(t) = \frac{(t - t_0) \ldots (t - t_{k-1})(t - t_{k+1}) \ldots (t - t_N)}{(t_k - t_0) \ldots (t_k - t_{k-1})(t_k - t_{k+1}) \ldots (t_k - t_N)}$$ (3.3)

The Lagrange interpolation approaches the sinc interpolation as $N \to \infty$ (Leis 2011), however unlike the sinc interpolation discussed earlier, the Lagrange interpolation does not require the known samples to be uniformly spaced, which allows the interpolation to be applied directly to the clipped waveform.

To apply this to the waveform shown in Figure 3.5, an equal number of samples to the left of the plateau and right of the plateau is needed, with more samples (up to at most the next clipping point) used giving a better result at the cost of calculation time.

When implementing polynomial interpolation, especially when larger values of $N$ are used, consideration must be made to the limitations of floating point numbers. Similar to what was discussed in section 2.5.4, high values of $N$ result in polynomials whose coefficients can vary greatly in magnitude, which can affect the accuracy of the interpolation, and also limits the value of $N$ which can be used. Also if large values are
used for the sample times \((t_0, t_1, \text{etc.})\), then this will only exacerbate this issue as time increases. To help limit this effect, an offset should be subtracted from the time values used in the interpolation so that large time values don’t end up being used in \(t_0, t_1, \text{etc.}\), then the offset added back in once the interpolation has been performed.

### 3.4 Test Methodology

The algorithms were tested using a number of test waveforms. The first of these is a simple sinusoid, which is described by equation (3.4). The second is a composite waveform consisting of two sinusoids, one at 50 Hz and the other at 5000 Hz and is described by equation (3.5). The point of testing this waveform is to gain an idea how the algorithms perform against typical audio waveforms, which often contain higher frequencies of lesser magnitude superimposed onto a lower frequency of higher magnitude. The final waveform to be tested is an audio waveform.

\[
x_0(t) = \sin (2\pi f_0 t)
\]  \hspace{1cm} (3.4)

\[
x_0(t) = 0.7 \sin (2\pi f_0 t) + 0.3 \sin (2\pi f_1 t)
\]  \hspace{1cm} (3.5)

These two test waveforms can be converted to discrete time \(x_0(n)\) by making the substitution \(t = \frac{n}{f_s}\) where \(f_s\) is the sample rate. The test waveforms \(x_0(n)\) have clipping introduced into them artificially to produce \(x(n)\). Clipping can be introduced into any waveform using digital audio manipulation software, by amplifying the waveform so that the peak sample value exceeds the full scale deflection value. When this audio is saved, those values which exceed the full scale deflection value will be truncated to the full scale deflection value. In order to compare the truncated waveform to the original waveform however, it will be necessary to attenuate the truncated waveform by whatever decibel value was used to amplify it. Since the waveforms will be tested within a programming environment, this process can be simplified by simply looping through each sample and truncating sampled values above and below a desired clipping point, which will be a sample value which is a percentage of the peak value.

For the simple waveform tests, the effectiveness of the reconstruction algorithms can
be tested by comparing the original non-truncated waveform $x_0(n)$ with the output of the reconstruction algorithm $y(n)$.

### 3.5 Experimental Results

The two reconstruction techniques were tested on a 440 Hz sinusoid waveform which was truncated at 0.8. The time domain plots of the truncated and reconstructed waveforms are shown in Figure 3.7.

![figure](image.png)

**Figure 3.7: Reconstruction of a truncated sinusoid.**

From Figure 3.7 it can be seen that the low pass filtering method produces a waveform whose amplitude is not quite correct, however the abrupt plateau has been removed.
3.5 Experimental Results

In order to gain a better idea of how well these algorithms are reducing the truncation noise, the frequency spectrums of the waveform are shown in Figure 3.8.

![Frequency spectrums of reconstructed sinusoids](image)

Figure 3.8: Frequency spectrums of reconstructed sinusoids

The first thing that stands out in Figure 3.8 is that the sinusoid which was reconstructed using the Lagrange interpolation method contains only one component at 440 Hz. As this corresponds to the frequency spectrum of a 440 Hz sinusoid, this implies that for this particular application, the reconstruction is perfect (or close enough). Comparing the low pass result with the truncated spectrum shows that there is a reduction in the noise components, however the component at 1320 Hz remains significant, with the noise dropping off at increasing frequency, which is expected with a low pass filter.

The next waveform tested was the composite waveform described by (3.5). The time
3.5 Experimental Results

domain plots of the truncated and reconstructed waveforms are shown in Figure 3.9.

![Truncated Composite Wave](image)

![Low Pass Reconstruction](image)

![Lagrange Interpolation](image)

Figure 3.9: Reconstructed composite waveforms.

Again, the Lagrange interpolation appears to perform well here, with the amplitudes of the low pass filtering method falling short of the desired result.

The final and perhaps most important test, was to see how well the algorithms perform at removing clipping from an audio waveform. The waveform that was tested was from an electronic dance music track, the exact timing of the waveform was during a percussive kick, when the audio is most likely to be truncated due to a large low frequency transient. For this particular test, the clipping point was set at 0.6, to give an idea of what happens when there is a large amount of clipping. The results of the reconstructed waveforms are shown in Figure 3.10.
What is interesting to note here is that both methods deviate from the desired result. Firstly examining the low pass filter method result, it can be seen that while the low frequency component of the waveform has remained intact, the high frequency components in the clipping regions have been filtered out. When listening to this audio waveform, the lack of high frequency components in this region is far more noticeable than the noise that is added due to truncation.

Examining the result of the Lagrange interpolation in Figure 3.10 shows that the interpolated waveform goes beyond the full scale deflection value in several places. This manifests itself as very loud popping noises when the audio is played back. This occurs when the algorithm attempts to correct a number of samples which happens to be
greater than the number of samples in the period of the highest frequency component in the desired waveform. For example, consider the waveform shown in Figure 3.11.

![Figure 3.11: Sinusoid with large number of samples clipped relative to period.](image)

This is a simple sinusoid described by the equation $y = 0.8 \sin(0.2\pi n)$ which has had samples 103 through to 143 truncated. If two samples are used to specify the known points, then the resulting interpolated waveform will be a linear interpolation between the two truncation end points. If four samples are used then the result will be an arc which travels beyond full scale deflection. Theoretically, if a large enough number of samples either side of the truncation are specified, then the interpolation will match the desired sinusoid, however more samples imply greater polynomial orders, which brings with it greater floating point precision errors (interestingly, if a large enough order is specified, then the numerator and denominator of the interpolating equation will simply overflow). Therefore, Lagrange polynomial interpolation for truncated waveform reconstruction is not suitable (at least with double precision floating point numbers), if the number of samples being corrected exceeds the period of the highest frequency component in the audio.

### 3.6 Further Work

There are a number of possible ways in which the algorithms described so far could be improved, to better the results than those shown in Figure 3.10, but were not explored
due to time constraints. Examining the low pass filter reconstruction in Figure 3.10 shows that the underlying low frequency component has remained intact, with the resulting waveform containing islands of high frequencies. If the low frequency is filtered out (to be added back in later), then the resulting waveform would consist of intermittent audio. The silent areas in between the audio regions could possibly then be reconstructed using an analysis/synthesis technique, such as the phase vocoder, or TD-PSOLA (these techniques are explored in Chapter 4) and the low frequency component added back in afterwards, in a similar approach to the estimation of audio between regions of reliable data method proposed by (Adler et al. 2011).

For the Lagrange interpolation method, the easiest solution would be to simply not attempt to interpolate regions where the number of samples between the truncation end points and the next peak or valley is equal to or greater than the number of clipped samples. The resulting waveform would still contain some clipping, which could then be run through the low pass filter reconstruction method, to give a better result than that given by just using the low pass reconstruction filter by itself.

Another option for improving the Lagrange interpolation method would be to utilise floating point numbers of arbitrary precision (as opposed to double precision which was used in the tests here). This would require either finding a suitable library, or require implementation, but would enable more samples to be used in the interpolating equation, giving more accurate results.

### 3.7 Comparison With Other Interpolation Techniques

The Lagrange interpolation function which was developed during this project has a similar function signature to the built in MATLAB interpolating function interp1. For the sake of completeness, the interpolation based algorithm was tested utilising the interp1 function in order to compare how well these techniques perform in the truncated waveform reconstruction application. The version of MATLAB which was used contained 5 methods of interpolation in the interp1 function, these were: nearest, linear, spline, cubic and v5cubic. Only the spline and cubic methods have been tested as the v5cubic method requires equally spaced sample points, and the nearest and linear methods would only give values between the truncation sample values, not above as is
required. Details of the implementation of these interpolating functions can be found in the MATLAB documentation.

Figure 3.12: Reconstruction of composite waveform using MATLAB’s interp1 function.

Figure 3.12 shows the result of the reconstruction algorithm, modified to use the interp1 function and tested on the same composite waveform that was used previously. As can be seen, the spline reconstruction is not as accurate as the polynomial interpolation used before. The cubic interpolation on the other hand shows peaks which have been attenuated below the truncation value, which is due to the samples at the ends of the truncated regions being included in the list of samples being interpolated.

Figure 3.13 shows the result of the MATLAB interpolated functions applied to the same audio waveform tested before. Like the polynomial interpolation, the spline in-
Figure 3.13: Reconstructed audio waveform using MATLAB’s interp1 function.

terpolation contains excursions beyond the full scale deflection value, although this occurs much less frequently. This is due to the interpolating function being a 3rd order polynomial and follows similar reasoning to that given at the end of section 3.5. In regard to the cubic interpolation, the result is only slightly different to the truncated waveform itself.
3.8 Conclusion

The low pass filter and Lagrange interpolation methods of reconstructing truncated waveforms presented here show only a limited amount of success. They are capable of reducing noise in simple waveforms such as compound sinusoids, but cause further unwanted distortion when applied to an audio waveform with large amounts of truncation. The developed algorithm was also tested against MATLAB’s spline interpolation, which has been shown to produce less distortion than the Lagrange interpolation technique.
Chapter 4

Constant Pitch Time Stretching

There have been many techniques proposed which use signal processing to perform constant pitch time stretching on an audio waveform. The techniques which are of most interest here are the ones which can be applied to a polyphonic musical recording.

4.1 The Phase Vocoder

The phase vocoder is an analysis/synthesis technique, the exact details of the typical phase vocoder in a time stretching application can be found under “The Basic Phase Vocoder Time Scaling Algorithm” in (Laroche & Dolson 1999) however the basics will be summarised here. The analysis stage essentially divides an input waveform \( x(t) \) into frames, each centred at a time \( t^n_a \) with a centre spacing of \( R_a \). Each frame is multiplied by a windowing function \( h(n) \) and the Fourier Transform is calculated to give the Short Time Fourier Transform (STFT) representation \( X(t^n_a, \omega) \). The synthesis stage involves modifying \( X(t^n_a, \omega) \) to give \( Y(t^n_s, \omega) \) which is a STFT representation of a desired waveform divided into frames centred at \( t^n_s \) with centre spacing \( R_s \). The inverse Fourier transform is then taken for each frame and summed to yield the synthesised waveform \( y(t) \).

In the time stretching application, for a desired time scale factor \( \alpha \), the synthesised spacing \( R_s \) is given by \( R_s = \alpha R_a \) and the phase and magnitude of each component in \( X(t^n_a, \omega) \) is modified to accommodate for this change in the centre spacing. The key
4.2 Time Domain Pitch Synchronous Overlap and Add

to the phase vocoder is that the phases in $Y(t_u, \omega)$ are modified such that when the inverse Fourier transform is taken, the phases of the overlapping regions are properly aligned.

4.2 Time Domain Pitch Synchronous Overlap and Add

Time Domain Pitch Synchronous Overlap and Add (TD-PSOLA), first proposed in (Hamon, Mouline & Charpentier 1989), is a time stretching technique which is commonly used in speech synthesis. This algorithm works by detecting the period $T$ of a periodic waveform and splitting the waveform into overlapping frames of length $2T$, with each frame centred on an epoch (usually the peak) of the waveform and each frame multiplied by a Hanning window. To perform time expansion, a number of frames are duplicated to provide the required time ratio, conversely to perform time compression, frames are discarded in accordance to the required time ratio. Figure 4.1 illustrates how the TD-PSOLA algorithm can be used to perform constant pitch time compression through the use of frame omission.

Figure 4.1: TD-PSOLA algorithm for time compression.
4.2.1 Modification of TD-PSOLA to Suit Rhythmic Time Stretching

The TD-PSOLA algorithm as described here cannot be applied to typical polyphonic musical recordings as it’s use applies to a periodic waveform (which a polyphonic recording is not), however the concepts of overlapping, duplicating and discarding frames can be used in a similar fashion. The approach that was investigated is to use frame lengths which divide evenly into the beat interval, and are at least as long as the period of the lowest audible frequency. This approach will be referred to from here on as “modified TD-PSOLA” although it should be mentioned that this is a bit of a misnomer as the algorithm is not pitch synchronous at all, rather it is based on tempo, and isn’t synchronous.

To perform time expansion, every $n$th frame is required to be duplicated. For a particular frame $k$, if the next frame added gives the required ratio $R$, then that frame is duplicated. Mathematically this is:

$$R = \frac{k + 1}{k}$$

$$k = \frac{1}{R - 1}$$  \hspace{1cm} (4.1)

The value $k$ represents how often a duplicate frame is required, i.e. a duplicate frame is required every $k$ frames, however it is very likely that this number will not be an integer. In order to ensure that the average ratio of the final stretched waveform is correct, it is necessary to maintain a variable which counts the number of frames which haven’t been skipped, when this variable exceeds $k$, then $k$ is subtracted from it and the frame is duplicated.

In order to perform time compression, Equation (4.1) needs to be modified:

$$k = \frac{1}{\frac{1}{R} - 1} - 1$$  \hspace{1cm} (4.2)

So far, the algorithm described does not make any allowances for any short transients which may exist only in a single frame (such a percussive hit). If a frame containing such a transient is duplicated or discarded, then it is possible that percussive hits in a
musical recording will be doubled or skipped, which will be perceived as an abnormality to the listener. A simple allowance for the rhythm can be made if the tempo is known beforehand, by assuming that these transients will only occur at the start of every eighth, that is, the start of every beat and off-beat for a 4/4 time signature. If a frame is within a specified tolerance from one of these beat accents, then that frame is not eligible for duplication or disposal. Figure 4.2 illustrates this situation, where every 5th frame is discarded (time scale factor of 0.8 and $k = 8$).

The algorithm as described has been implemented using MATLAB’s scripting language and can be found under Appendix D.

### 4.3 Synchronous Overlap and Add

The basis for the TD-PSOLA algorithm is the Synchronous Overlap and Add algorithm which was originally proposed in (Roucos & Wilgus 1985). The algorithm is an analysis/synthesis technique (similar to the phase vocoder) which splits the source waveform into overlapping frames with centre spacing $R_a$ and assembles an output waveform by overlapping these frames with a new centre spacing of $R_s = \alpha R_a$ where $\alpha$ is the required time scale ratio. Figure 4.3 shows diagrammatically how the algorithm can be used to achieve time expansion, with a factor of expansion, $\alpha = 1.333$ and factor of overlap, $\beta = 0.4$.

To utilise this algorithm for time stretching of rhythmic audio, the analysis centre spacing $R_a$ is chosen such that there are an integer number of frames between beat
acents. The synthesis centre spacing is calculated using $R_s = \alpha R_a$ and from this, a frame size $N$ is determined such that each synthesis frame overlaps by a specified amount.

To improve the phase coherence between overlapping frames in the output waveform, each frame in the synthesis stage is shifted by a small amount $\Delta_k$ such that the frames overlap at a point of maximum similarity. To determine the offset $\Delta_k$, the cross correlation between the overlapping region in the previous frame $y_{u-1}$ and the next frame $y_u$ is used:

$$R_{y_uy_{u-1}}(k) = \frac{1}{N} \sum_{n=0}^{N-1} y_u(n)y_{u-1}(n-k)$$

The value of $k$ which is maximum gives the point of maximum similarity $\Delta_k$, however because the previous frame only contains a finite number of samples before abruptly dropping to zero value, the maximum value of $k$ that is used must reside in the left hand side of the cross correlation, to ensure that when the frames are overlapped, the envelope applied to the previous frame reaches zero before the end of the previous frame is reached, to prevent any sharp transitions from appearing in the output waveform.
4.4 Beat Alignment

Of particular interest when gauging the performance of the time stretching algorithms is the effect of beat misalignment and beat doubling/skipping. Equation (4.4) describes a continuous time sinusoid which is multiplied by a Gaussian window which will provide a pulse that can be used to represent a beat accent in the following tests.

\[ w(t) = e^{-\frac{(t-b)^2}{2c^2}} \cos(2\pi f_0(t - b)) \]  

(4.4)

Where:

- \( t \) is continuous time
- \( f_0 \) is the carrier frequency of the pulse
- \( b \) shifts the pulse in time
- \( c \) specifies the width of the Gaussian window

A pulse with a carrier frequency of 440 Hz and window width of 0.025 seconds is shown in Figure 4.4.

![Figure 4.4: 440 Hz test pulse centred at 0.025 seconds.](image)

The equation given in (4.4) can be converted to discrete time by making the substitution \( t = \frac{n}{f_s} \) where \( f_s \) is the sample rate. If a beat interval of \( T_b \) is required, and a test rhythm
4.5 Experimental Results

waveform to be generated which contains an $M$ number of beats, then this waveform can be generated using:

$$r(n) = \sum_{b=0}^{M-1} e^{-\frac{(n-bT_b)^2}{2\sigma^2}} \cos(2\pi f_0 \frac{n}{f_s} - bT_b)$$  \hspace{1cm} (4.5)

With this set up, it is now possible to specify a rhythm of pulses with any beat interval. If a waveform $x(n)$ is generated and is time stretched by a ratio of $\alpha$ to produce waveform $y(n)$, then this can be compared with the actual desired waveform $y_r(n)$ by generating $y_r(n)$ using (4.5) with the beat interval multiplied by $\alpha$.

To determine how well the beat accents have aligned with their correct positions, the envelope of $y(n)$ is multiplied by the envelope of $y_r(n)$, with the resulting waveform indicating how well the beats have aligned, for example, the absence of a peak at a time where a beat would be expected would indicate that particular beat in $y(n)$ did not align with the beat in $y_r(n)$. The envelopes of the pulse waveforms are compared instead of the pulse waveforms themselves to side-step the issues that would occur if $y(n)$ were out of phase with $y_r(n)$. The envelope detection algorithm used here consists of full wave rectification (by finding the absolute value) followed by low pass filtering.

4.5 Experimental Results

The three algorithms described so far have been implemented in MATLAB and can be found in the appendices of this dissertation, with the exception of the phase vocoder$^1$.

4.5.1 Beat Alignment Analysis

The test performed here was time stretching from 140 beats per minute to 160 beats per minute (time compression factor of 0.875), which is a considerably large change in tempo that would result in frequent beat skipping. Using the test methodology described in the previous section, the beat alignment of the two overlap methods are shown in Figures 4.5 & 4.6.

$^1$Due to difficulties in creating a working implementation and due to time constraints, all further references to the implementation of the phase vocoder are in reference to that written by (Ellis 2002)
The vertical axes of these graphs represent a measure of how well the beat pulses have aligned, that is, a beat with a peak value of unity can be considered to have perfectly aligned with where it should be, while an absence of a peak would mean complete misalignment. Looking at Figure 4.5, there appears to be a repeating pattern in the alignment, with the amount of misalignment increasing for a few beats before resetting. This is a side effect of using frame omission to achieve time stretching, which implies that up to one frame width of misalignment can be expected as frames are omitted. No such pattern is observed in Figure 4.6, which aligns each frame precisely where they should appear, minus a small time delay for positioning at a point of maximum similarity, which would account for the slight misalignments occurring toward
the beginning of the waveform.

The beat alignment of the two overlap-add algorithms shown so far have performed reasonably well. In contrast to this, the beat alignment of the phase vocoder implementation is shown in Figure 4.7.

![Figure 4.7: Beat alignment of phase vocoder algorithm.](image1)

The absence of peaks in the 1 to 2 seconds region and 3 to 5 seconds region suggests that this algorithm has failed to align the beats in these positions correctly. To confirm whether this is actually true, a time domain plot of the output of the phase vocoder is shown in Figure 4.8.

![Figure 4.8: Output of phase vocoder time compression.](image2)

Examining Figure 4.8 reveals that there is an amplitude modulation occurring as well
as beat doubling. This could be explained by the fact that this particular phase vocoder implementation works on frame sizes which are powers of 2 (in this situation 1024 was used), which do not divide into the number of samples per beat evenly, resulting in beat accents which are sometimes split across frames.

4.5.2 Subjective Analysis

In addition to the quantitative analysis provided by the beat alignment test, a subjective test was performed, where each algorithm was used to time stretch electronic dance tracks. These tracks were chosen as they contain strong and distinctive percussive rhythms (which would reveal issues with short transients and beat alignment) and contain notes of differing lengths across most of the frequency spectrum (to test the performance of frame overlap).

The most prominent artifacts encountered in each of the algorithms, as well as their severity, are summarised in Table 4.1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Artifact</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>phase vocoder</td>
<td>transient smearing</td>
<td>moderate at 1024 samples</td>
</tr>
<tr>
<td>modified PSOLA</td>
<td>amplitude modulation</td>
<td>slight</td>
</tr>
<tr>
<td>SOLA</td>
<td>percussive flanger effect</td>
<td>moderate</td>
</tr>
</tbody>
</table>

With the phase vocoder, the most prominent artifact was distortion of the percussive rhythm. This would be caused by the fact that the underlying model behind the phase vocoder assumes that the waveform can be modelled by discrete sinusoids which are coherent across multiple frames, which is not true in the case of these percussive instruments, whose duration is comparable to the frame length. The result is that these short transients appear to be “smeared” across multiple frames. This effect can be reduced by using shorter STFT frames, however this reduces the frequency resolution of the algorithm.

The most prominent artifact in the proposed modifications to the PSOLA algorithm is that of modulation of background harmonising instruments. This occurs in the regions where a frame is neighbouring a duplicated or omitted frame and is caused by
destructive interference as a result of the phases of the overlapping waveforms not being properly aligned. The severity of this type of artifact is only slight, but once noticed is difficult to ignore.

The standard SOLA algorithm does not contain the above mentioned modulation artifact, however this algorithm tends to distort the percussive instruments, which end up sounding as if they have had a “flanger” effect applied to them. The severity of this artifact can be controlled by varying the parameters of the algorithm, with more overlap and smaller frame sizes producing less of this distortion.

4.6 Further Work

The topic of constant pitch time stretching is one in which has been the subject of considerable research effort, both in the past and the present. This project was mostly interested in algorithms suitable for musical recordings, and there are a number of proposals which have been made by others which could be utilised in this application, but haven’t been investigated here. At this point in time, perhaps the most promising approach would be to use an analysis/synthesis technique based on the wavelet transform (as opposed to the phase vocoder’s short time Fourier Transform), which does not suffer from the time/frequency resolution tradeoff encountered with the STFT. This may be similar to the approach used by Prosoniq’s Dirac technology, although the implementation of such technology remains a trade secret (Bernsee 1999).

Another aspect which could be looked at is synchronisation with tempo. At the moment, the algorithms are reliant on the beginning of the waveform being aligned with the beginning of a beat and make no allowance for changes in tempo occurring within the recording. These time stretching algorithms could be combined with a beat detection algorithm in order to remove these restrictions, although beat detection algorithms themselves have their own set of limitations, which are outside the scope of this project.
4.7 Conclusion

The phase vocoder does not perform well when time stretching audio which contains short transients due to the inherent assumption that each frame contains sinusoids which are coherent across multiple frames and the required trade off to increase the time resolution to counter this, is a reduction in frequency resolution. It also has issues with beat alignment, which suggests that further work is required perfecting the phase unwrapping part of the phase vocoder. On the other hand, the two overlap-add methods were capable of aligning the beat accents within the specified tolerance, however the two overlap-add methods proposed here each contain their own unique audible artifacts, the severity of which can be controlled by varying parameters.
Chapter 5

Conclusions and Further Work

The melodic filter was moderately successful, being capable of both extracting an audio melody, or suppressing a melody in an audio waveform. The effectiveness of this algorithm has shown to be dependant on the type of instrument that is to be filtered. It is most effective when used with instruments such as string and woodwind instruments (whose sounds decay very quickly at the end of each note) and is less effective with vocal melodies and sounds which have long fade times (such as a piano with the sustain pedal depressed, or a guitar).

The investigation into truncated waveform reconstruction managed to achieve only limited success, being able to reconstruct simple waveforms with minor amounts of clipping, but failing when applied to an audio recording with large amounts of clipping. A number of proposals have been made for improving the performance of the techniques explored, but at the time of writing, haven’t been followed through due to time constraints.

The constant pitch time stretch algorithms which were investigated were moderately successful, with each algorithm containing its own particular drawbacks, the severity of which could be controlled by varying the algorithm’s parameters. The two overlap-add methods investigated were capable of correctly aligning the beat accents in the time stretched waveform, although the phase vocoder struggled with doing this. Finally, the major audible encountered were; transient smearing in the phase vocoder, modulation of background harmony in the modified PSOLA algorithm and lastly, a percussive flanger like effect with the SOLA algorithm.
5.1 Further Work and Recommendations Summary

A number of suggestions for building upon the results of each of the applications have been put forward in the previous chapters which will be summarised here.

For the melodic filter described in Chapter 2, it has been suggested that an IIR bandpass filter and FIR filter could be developed for melody extraction and suppression respectively. It has also been suggested that higher order IIR bandstop/bandpass filters could be developed by utilising second order sections. Finally an alternative method of extracting a melody from a recording has been put forward by (Raphael 2008), which could be implemented as an alternative to filter based melody extraction.

For the truncated waveform reconstruction application described in Chapter 3, the low pass filter technique could be improved by separating the reconstructed low frequency from the output audio, then using an analysis/synthesis technique to fill in the regions of missing audio. For the Lagrange interpolation method, it has been suggested that the algorithm could be improved by not attempting to reconstruct regions which are likely to result in the reconstructed samples exceeding the full scale deflection value. It has also been suggested that floating point numbers of arbitrary precision be utilised in order to allow higher order polynomials to be used.

For the constant pitch time stretching application described in Chapter 4, it has been suggested that an analysis/synthesis technique based upon the wavelet transform be investigated, in order to improve upon the time/frequency resolution trade-off limitation of the phase vocoder. Another suggestion is better synchronisation with tempo, through the use of a beat detection algorithm.
References


Appendix A

Project Specification
ENG 4111/2 (or ENG8002) Research Project

Project Specification

For: **Brendan Trevorrow**
Topic: Investigation of Digital Audio Manipulation Methods
Supervisors: Dr. John Leis
Sponsorship: School of Mechanical & Electrical Engineering

Project Aim: To implement and assess the performance of several specific digital audio manipulation methods which are either not common in digital audio workstation software or are otherwise the subject of ongoing research.

Program:

1. Research, implement and test a filter based method of extracting a melody audio waveform from an existing digital audio recording.
2. Research, implement and test methods of reducing noise in digital audio recordings which have been damaged due to envelope truncation.
3. Research existing methods of constant pitch time stretching and implement a design which is able to align beat accents according to tempo.
4. Produce an academic dissertation detailing the findings of the project.

As time and resources permit:

1. Implement and test a classifier based method of extracting a melody audio waveform from an existing digital audio recording.
2. Combine the developed melody extraction software with other existing methods of voice extraction to improve performance.

Agreed:

Student Name: Brendan Trevorrow
Date: 22 February 2013

Supervisor Name: John Leis
Date: 21 March 2013

Examiner/Co-Examiner: Chris Snook
Date: 16 April 2013
Appendix B

Melodic Filter Class Diagrams

The melodic filter application developed during this project ended up at a length of just under 2000 lines of code. Many of the lines were also considerably longer than 80 characters, due to following typical C# variable naming conventions. Therefore, this appendix only includes class diagrams of the developed code, as well as a few MATLAB functions which were used also used as part of the truncated waveform reconstruction and time stretching applications.
Figure B.1: Batch filter class diagram.
Figure B.2: Input/Output class diagram.
Figure B.3: Entry point and testing functions.
B.1 Chebyshev Low Pass Prototype

This generates the numerator and denominator polynomial of a chebyshev low pass filter prototype whose cutoff frequency is 1 rad/sample, as described in section 2.5.1. It has been included here as it is also used in the truncated waveform and time stretching applications.

```matlab
function [bs as] = chebyshev_prototype(N, epsilon)
% Generate a Chebyshev low pass filter prototype with corner 1 rad/sample
% Based on Leis 2011, page 337

tc = chebyshev_matrix(N);
% Polynomial coefficients for order n

tc = tc(N+1,:);
% square, multiply by epsilon squared, add one (will be 2N+1 coefficients)
d = conv(tc, tc)';
d = d*epsilon^2;
d(2*N+1) = d(2*N+1) + 1;
% powers for s
spow = (2*N:-1:0)';
% (s)(-s) terms
s = s^2;
s = -(s)*(-s);
s = s^4;
s = -(s)*(-s)*(-s);
s = s^6;
% spow/2 will always be an integer
snew = (-1)*ones(2*N+1, 1);
svals = snew.^(spow./2);
% finally, the denominator coefficients

d2 = svals.*d;
% find the roots of chebyshev |G|^2 then stable

d = roots(d2);
i = find(real(d) < 0);
% expand to polynomial using only stable roots
poles = d(i);
% Get denominator poly of low pass filter
as = poly(poles);
% Calculate gain K

if mod(N,2)
    % Odd number of poles
    bs = -prod(poles);
else
    % Even number of poles
    bs = 1/sqrt(1+epsilon^2)*prod(poles);
end
```
B.2 Chebyshev Polynomial Generator

This generates Chebyshev polynomials and is used in the previous function.

```matlab
function T = chebyshev_matrix(M)
% Generate a matrix whose rows represent the coefficients of Chebyshev
% polynomials up to order M.
% Based on Leis 2011, page 334

T = zeros(M+1, M+1);
T(1, M+1) = 1;
T(2, M) = 1;
for n=2:M
    t = 2*conv([1 0], T(n, :));
    t = t(2:M+1);
    T(n+1,:) = t - T(n-1, :);
end
```
Appendix C

Truncated Waveform

Reconstruction Code Listings
C.1 Lagrange Polynomial Interpolation Reconstruction

This is the implementation of truncated waveform reconstruction using the Lagrange polynomial interpolation method as described in section 3.3.

```matlab
function x = declip(xx, tol, Nmax)
% Lagrange polynomial interpolation based declipping

x = xx;
clipval = max([max(x) abs(min(x))]);
ed = 0;
clip_end = 0;
skip = 0;
while ~ed
    left_count = 0;
    right_count = 0;
    % Read rightwards from last fix until clipping is detected
    for n = clip_end+1:length(x)
        if skip > 0
            skip = skip - 1;
            continue;
        end
        % This sample and previous sample is clipped
        if abs(clipval-abs(x(n))) < tol && abs(clipval-abs(x(n-1))) < tol
            % Previous sample is where clipping started
            clip_start = n-1;
            clip_count = 2;
            % Correct left count
            left_count = left_count - 1;
            % Read right, number of clipped samples until clipping ends
            for m = n+1:length(x)
                % If sample not clipped
                if abs(clipval-abs(x(m))) > tol
                    clip_end = m-1;
                    break;
                end
                % This sample was also clipped, increment count
                clip_count = clip_count + 1;
                % Check if actually at end
                if m == length(x)
                    return;
                end
            end
            % Determine number of usable samples to right of clipping
            for m = clip_end+1:length(x)
                % If sample clipped
                if abs(clipval-abs(x(m))) < tol
                    break;
                end
            end
        end
    end
end
```

% End of declip function
right_count = right_count + 1;
end
count = min([left_count right_count N\_max/2]);
if clip\_count > left\_count || clip\_count > right\_count
  \% Skip this clipping region
  skip = clip\_count;
  continue;
end
N = 2*count;
\% Create short time and sample vectors
xx = zeros(N, 1);
t = zeros(N, 1);
for m=1:count
  \% Left side of clipping
  mm = clip\_start-count+m-1;
t(m) = mm - clip\_start;
xx(m) = x(mm);
  \% Right side of clipping
  mm = clip\_end+m;
t(m+count) = mm - clip\_start;
xx(m+count) = x(mm);
end
\% Interpolate values between clipping points
for m=clip\_start:clip\_end
  x(m) = lgrngint(t, xx, m-clip\_start);
end
fprintf (1, ‘%d of %d\n’, n, length(x));
\% Break to start next clip detection
break;
end
\% Algorithm is complete if reached end of waveform
if n == length(x)
ed = 1;
end
left\_count = left\_count + 1;
end
C.2 Low Pass Filter Reconstruction

This is the implementation of truncated waveform reconstruction using the low pass filter method as described in section 3.2.

function x = declip2(x, tol, fs)
% Low pass filter based declipping

clipval = max([max(x) abs(min(x))]);
clip_end = 0;
count = 0;
% Read rightwards until clipping is detected
for n=1:length(x)
    % If still in a region that was already declipped
    if n < clip_end
        continue;
    end
    % This sample and previous sample is clipped
    if abs(clipval−abs(x(n))) < tol & & abs(clipval−abs(x(n−1))) < tol
        % Previous sample is where clipping started
        clip_start = n−1;
        clip_count = 2;
        % Read right, number of clipped samples until clipping ends
        for m = n+1:length(x)
            % If sample not clipped
            if abs(clipval−abs(x(m))) > tol
                % Ensure there are enough unclipped samples to right
                ed = m−min([clip_count 24]);
                if ed < length(x)
                    if clipval−max(abs(x(m:ed))) < tol
                        % Continue if not enough unclipped samples
                        clip_count = clip_count + 1;
                        continue;
                    end
                end
            end
            clip_end = m−1;
            break;
        end
        % If clipping runs off end of waveform
        if m == length(x)
            return;
        end
    end
    % This sample was also clipped, increment count
    clip_count = clip_count + 1;
end
% Determine cutoff frequency
cut_off = fs / (2*(clip_end − clip_start));
% Ensure a maximum cutoff frequency
if cut_off > fs/2.1
\[ \text{cut\_off} = \frac{\text{fs}}{2.1}; \]

\% Create even numbered window size
N = 2*(clip\_count + \text{mod}(\text{clip\_count}, 2));
h = \text{tukeywin}(N, 0.5);
\% Start of window in original waveform
off = clip\_start - N/4 - 1;
\% If window goes outside extents of waveform
if off < 1 || off+N > length(x)
    \% Ignore this clipped region
    continue;
end
\% Filter waveform
xx = \text{zero\_phase\_low\_pass\_filter}(x, \text{cut\_off}, \text{fs});
\% Calculate gain using mean in windows
meanx = mean(x(off:off+N));
meanxx = mean(xx(off:off+N));
\% Apply gain to filtered waveform so they align better vertically
xx = meanx / meanxx * xx;
\% Do mixing
for m=1:N
    x(off+m) = (1-h(m))*x(off+m) + h(m)*xx(off+m);
end
count = count + 1;
fprintf(1, \%d of \%d\n', n, length(x));
end
C.3 Lagrange Interpolation Implementation

This is an implementation of Lagrange polynomial interpolation as described by equation (3.2).

```matlab
function val = lgrngint(x, y, t)
% Lagrange interpolation algorithm, x is known sample times, y is known
% sample values and t is the time for which we want to calculate a sample
% value.

if (length(y) ~= length(x))
    error('Input vectors must be same length');
end
N = length(x);

val = 0;
for k=1:N
    % start with numerator
    num = 1;
    for n=1:N
        if n == k
            continue;
        end
        num = num * (t - x(n));
    end
    % calculate denominator (a constant)
    dem = 1;
    for n=1:N
        if n == k
            continue;
        end
        dem = dem * (x(k) - x(n));
    end
    % Do interpolation
    val = val + num / dem * y(k);
end
```

C.4 Zero Phase Shift Low Pass Filter

This applies a 4th order low pass filter to a waveform with zero output phase shift, as described in section 3.2.2. It is used as part of the low pass filter reconstruction algorithm shown in section C.2.

function y = zero_phase_low_pass_filter(x, fc, fs)
% Apply a 4th order low pass filter to waveform x, with cutoff frequency at % fc Hz and sampling rate of fs Hz with zero phase shift in output.
if fc >= fs/2
    error(’Cutoff_frequency_too_high’)
end

% Create low pass filter prototype
[bs, as] = chebyshev_prototype(2, 0.5);
% Scale cutoff
[bs, as] = lp2lp(bs, as, 2*pi*fc);
% Transform to discrete time filter
[bz, az] = bilinear(bs, as, fs);

% Apply forward filter
N = length(x);
y1 = filter(bz, az, x);
y2 = zeros(N, 1);
for n=1:N
    y2(n) = y1(N-n+1);
end

% Apply reverse filter
y1 = filter(bz, az, y2);
y = zeros(N,1);
for n=1:N
    y(n) = y1(N-n+1);
end
Appendix D

Constant Pitch Time Stretching
Code Listings
This is the implementation of constant pitch time stretching using the process described in section 4.2.1.

```matlab
function y = time_scale(x, fs, current_tempo, new_tempo, div, tol)
% Perform time stretching using modified PSOLA

scale = current_tempo / new_tempo;
% Calculate beat sample period
T_b = fs * 60 / current_tempo;
% Calculate frame spacing
T_spacing = T_b / (div);
N = floor(T_spacing);
% Calculate frame skip/duplication interval
if scale > 1
    T_f = 1/(scale - 1);
else if scale < 1
    T_f = 1/(1/scale - 1) - 1;
end
% Calculate accent frame spacing
afs = div * T_spacing / (2*N);
% Create frame window
h = hanning(2*N);
% pre-allocate result
y = zeros(floor(length(x)*scale), 1);
% Determine number of frames required
frames_required = floor(length(y) / T_spacing);

adjacent = 0; % Used to ensure adjacent frames aren’t duplicated/skipped
k = 0;
interval = 1;
skipped = 0; % Frames skipped/duplicated (for debugging)
naf = afs; % Next accent frame
for u = 0:frames_required-2
    % Extract current frame
    stx = 1 + k*N;
    edx = stx + 2*N - 1;
    if stx > length(x) || edx > length(x)
        error('Could not adhere to accent tolerance. ');
    end
    frame = h .* x(stx:edx);
    % If next frame should be skipped/duplicated and not too close to beat
    if interval >= T_f && abs(naf - k) > tol && ~adjacent
        if scale > 1 % Time expansion
            interval = interval - T_f;
            skipped = skipped + 1;
        elseif scale < 1 % Time compression
```
\begin{verbatim}
% Skip next frame by incrementing k by 2
interval = interval - T_f;
skipped = skipped + 1;
k = k + 2;
else
  k = k + 1;
  interval = interval + 1;
end
adjacent = 1;
else
  % Increment to next frame
  k = k + 1;
  interval = interval + 1;
  adjacent = 0;
end
% Overlap add into result
sty = 1 + u*N;
edy = sty + 2*N - 1;
y(sty:edy) = y(sty:edy) + frame;
% Check if next accent frame should be incremented
if k > naf + tol
  naf = naf + afs;
end
end
\end{verbatim}
D.2 Tempo Based SOLA Time Stretching

This is the implementation of constant pitch time stretching using the SOLA algorithm described in section 4.3.

```matlab
function y = sola_time_scale(x, fs, current_tempo, new_tempo, div, beta)
% Stretches an audio vector x while maintaining pitch

scale = current_tempo / new_tempo;
R_a = floor(fs * 60 / (current_tempo * div)); % Analysis hop size
R_s = floor(R_a * scale);
N = floor(2*R_s /(2-beta));

% Calculate number of analysis hops needed
U = floor(length(x)/R_a) - 1;

N_overlap = floor(N*beta/2);

% Pre-allocate result
y = zeros(R_s*U + N, 1);

offset = 0;
% Synthesis and reconstruction
for u=1:U-1
    % Get current frame from source
    stx = 1 + (u-1)*R_a;
    edx = N + (u-1)*R_a;
    frame = x(stx:edx);
    % Current frame indices in destination
    sty = 1 + (u-1)*R_s;
    edy = N + (u-1)*R_s;
    if u > 1
        % Get previous frame
        prev = y(sty-R_s:edy-R_s);
        % Get overlapping (start) part from current frame
        stf = frame(1:N_overlap);
        % Get overlapping (end) part from previous frame
        edf = prev(N-N_overlap:N-1);
        % Correlate
        cc = xcorr(edf, stf);
        % Get correlation offset from left side
        [m, offset] = max(cc(1:N_overlap));
        offset = offset - N_overlap;
        offsets(u) = offset;
    end
    % Set source index
    idx = stx;
    i = 0;
    % For each sample in destination frame
```
for idy=sty+offset : edy
    if i < N_overlap
        % Overlap region
        h = i / N_overlap;
        y(idy) = (1−h)*y(idy) + h*x(idx);
    else
        % Non-overlap region
        y(idy) = x(idx);
    end
    % Next sample
    idx = idx + 1;
    i = i + 1;
end
D.3 Beat Waveform Generator

This is an implementation of the beat waveform generator described by equation (4.5).

```matlab
function y = generate_beat_waveform(tempo, c, cf, fs, M)
% Generate M number of pulses of c width at a center frequency of cf Hz
% according to a tempo and sampling frequency fs.

T_b = 60 / tempo;

y = zeros(M*fs*T_b, 1);

n = (0:length(y)-1)';

e = exp(1);

% For each beat
for b=1:M-1
  % Envelope
  env = exp(-((n./fs - b*T_b).^2/(2*c^2)));
  % Carrier
  carr = cos(2*pi*cf*(n./fs - b*T_b));
  y = y + env .* carr;
end
```