Sinusoids, noise and transients: spectral analysis, feature detection and real-time transformations of audio signals for musical applications

John Glover

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

A thesis presented in fulfilment of the requirements for the degree of Doctor of Philosophy

Supervisor: Dr. Victor Lazzarini

Head of Department: Prof. Fiona Palmer

Department of Music

National University of Ireland, Maynooth

Maynooth, Co.Kildare, Ireland

October 2012

Contents

List of Figures

List of Tables

Publications

John Glover, Victor Lazzarini, and Joseph Timoney. Simpl: A Python library for sinusoidal modelling. In *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, Como, Italy, September 2009.

John Glover, *The Audio Programming Book*, Richard Boulanger and Victor Lazzarini (ed.), chapter 19. The MIT Press, 2010.

John Glover, Victor Lazzarini, and Joseph Timoney. Python for audio signal processing. In *Proceedings of the 2011 Linux Audio Conference*, Maynooth, Ireland, May 2011.

John Glover, Victor Lazzarini, and Joseph Timoney. Real-time detection of musical onsets with linear prediction and sinusoidal modeling. *EURASIP Journal on Advances in Signal Processing*, 2011(1):68, 2011.

John Glover, Victor Lazzarini, and Joseph Timoney. Real-time segmentation of the temporal evolution of musical sounds. In *The Acoustics 2012 Hong Kong Conference*, Hong Kong, China, May 2012.

John Glover, Victor Lazzarini, and Joseph Timoney. Metamorph: Real-time high-level sound transformations based on a sinusoids plus noise plus transients model. In *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12)*, York, UK., September 2012.

Acknowledgements

There are many people who have played a major part in the completion of this thesis. Firstly, I would like to thank my supervisor, Dr. Victor Lazzarini, for his incredible guidance and seemingly limitless supplies of knowledge, time and patience. Thanks also to Dr. Joseph Timoney in the Computer Science department, who has always been available to provide extra help when needed.

I would also like to thank everyone in the Department of Music and in An Foras Feasa for making my time in NUI Maynooth so memorable and enjoyable. This research would also not have been feasible without the generous financial support provided by An Foras Feasa.

A special thanks goes to my family and friends for their help, kindness and understanding. In particular I would like to thank my mother Geraldine for her constant support and encouragement, Jessica Hayden whose assistance has been invaluable, and the rest of the Hayden family who have been very welcoming and accommodating.

Abstract

This thesis examines the possibilities for real-time transformations of musical audio signals using sinusoidal models. Four open-source software libraries were developed with the goal of providing real-time spectral synthesis by analysis tools for composers, musicians, researchers and audio signal processing developers. The first of these, called *Simpl*, provides a consistent API for interacting with established sinusoidal modelling systems from the literature, and integrates seamlessly with a powerful suite of libraries for scientific computing.

Sinusoidal models have been used to transform slowly-varying quasi-harmonic signals with much success, but they have well-documented weaknesses in dealing with transient signal regions. This problem is significant, as in monophonic musical sounds, transient locations often correspond with the attack section of notes and this region plays a large part in our perception of timbre. However in order to improve the synthesis of note attack transients, these regions must first be accurately identified. The first step in this attack transient identification process is generally musical note onset detection. Novel approaches to real-time note onset detection were developed as part of this research and are included in the *Modal* open-source software library. Modal also includes a set of reference samples that were used to evaluate the performance of the onset detection systems, with the novel methods

being shown to perform better than leading solutions from the literature.

The onset detection process was then combined with cues taken from the amplitude envelope and the spectral centroid to produce a novel method for segmenting musical tones into attack, sustain and release regions in real-time. The real-time segmentation method was shown to compare favourably with a leading non-real-time technique from the literature, and implementations of both methods are provided in the open-source *Note Segmentation* library.

The Simpl sinusoidal modelling library, Modal onset detection library and Note Segmentation library were then combined to produce a real-time sound manipulation tool called *Metamorph*. Metamorph is an open source software library for performing high-level sound transformations based on a sinusoids plus noise plus transients model. An overview of the design and implementation of the system is provided, in addition to a collection of examples that demonstrate the functionality of the library and show how it can be extended by creating new sound transformations.

Chapter 1

Introduction

The personal computer has had an undeniably profound impact on the processes of musical composition and performance. Although initially computers were only capable of non-real-time rendering of sound files to fixed storage, recent advances in both hardware and software have made the computer a powerful tool for real-time sound manipulation. For many musicians and composers the computer has become a musical instrument, and is an essential component in live performances of electronic music. The abundance, quality, and general reduction in cost of software instruments has played a key role in the computer's rise to prevalence in the musical performance space.

There are many different ways that software instruments can generate sound, although they can generally be broken down into four main categories: processed recordings, abstract algorithms, physical models and spectral models [111, 116]. These categories contain powerful sound synthesis techniques, some of which are theoretically capable of producing any possible sound. However, one of the biggest problems that still remains in sound synthesis is how to maintain the flexibility

to create a wide variety of sound timbres and yet provide an intuitive means of controlling the synthesis process. In addition, if the resulting software instrument is to be used in live performances, then ideally the sound synthesis parameters need to be controllable in real-time and be reactive with little noticeable latency.

The main goal of this research is to provide new software synthesis tools for composers, musicians and researchers, enabling them to perform flexible and intuitive transformations on live audio streams. A brief survey of some of the current sound synthesis techniques is given in Section 1.1, followed by a description of the approach to sound synthesis that is taken in this thesis in Section 1.2. The structure of the remainder of the thesis is described in Section 1.3.

1.1 Overview of sound synthesis techniques

Sampling, which can be grouped under the category of processed recordings, has become increasingly popular in recent years. The falling cost of computer memory has meant that it is quite feasible to have sample-based instruments containing several high quality recordings of each note that is playable by an acoustic instrument, with each recording representing notes played at different volumes and with different articulation. This can produce very accurate imitations of the original instrument. However, other than basic filters and amplitude envelope control, samplers rarely offer any means of manipulating the instrument timbres. Also in this category is the powerful technique of granular synthesis [99, 118]. It is a lot more flexible, but can require specifying hundreds of parameters even for short segments of sound, making simple yet precise real-time control difficult.

The abstract algorithms category consists of synthesis techniques that are not

based on any of the other three main sound synthesis paradigms. This includes popular sound generation methods such as Frequency Modulation (FM) synthesis [23, 70], Amplitude Modulation (AM), analog emulation and waveshaping synthesis. Although the relative importance of this synthesis category has been predicted to decline [111], many of the techniques have enjoyed a resurgence in recent years. In particular, the emulation of "classic" analog circuits has become popular. However as many of the synthesis techniques in this category are not based on any underlying analysis of existing sound sources, they are generally not best suited to producing accurate imitations of acoustic instruments.

Physical models are based on trying to represent the mechanical aspects of a sound generation process instead of the sound itself. Some physical models will be able to offer an intuitive set of control parameters, as they will be based on the physical properties of the mechanical system. For example, a physically modelled guitar may offer timbre controls such as string material (steel or nylon) and string thickness. However, as physical models are by their nature closely coupled with the underlying sound generation process, creating an arbitrary sound may require constructing a completely new physical model. This is not a trivial task, and may be time-consuming and unintuitive if it is not based on an existing physical instrument.

Spectral models on the other hand are based on modelling the frequency domain characteristics of the audio signal that arrives at the microphone, and so are not inherently tied to a specific sound creation process. This includes techniques such as subtractive synthesis, formant synthesis and additive synthesis. Additive models represent audio signals as a sum of sine waves with different frequencies and amplitudes. As research has shown that the perception of timbre is largely dependent on the temporal evolution of the sound spectrum [45], it seems natural to use a

sound model that is based on the frequency spectrum as a tool to manipulate timbre. Additive spectral models provide a means to process musical audio signals that can be perceptually and musically intuitive, but they do also have problems. Spectral processing tools such as the Phase Vocoder [29] for example are a well established means of time-stretching and pitch-shifting harmonic musical notes, but they have well-documented weaknesses in dealing with noisy or transient signal regions [30]. This has lead to the development of models that are based on a combination of sinusoids and noise [110, 34] and models that consist of a combination of sinusoids, noise and transients [77, 121]. Correctly identifying these noise and transient signal components in real-time is still a difficult problem however.

1.2 Our approach to sound synthesis

The existence of such a large variety of techniques for sound synthesis suggests that there is no single approach that is ideal for all musical situations.
However, the appeal of additive synthesis and the promise of a general-purpose sound model that can provide flexible and intuitive control of transformations has proven hard to resist for many researchers, as illustrated by the numerous recent developments in the field [78, 109, 121, 34, 94, 59]. These ideals also resonated with the authors and so this is the approach that is taken in this work.

Similarly to granular synthesis and some physical models, additive spectral models[1] can potentially suffer from the problem of having too many control parameters to allow the synthesised sounds to be manipulated in a meaningful manner. This problem is further exacerbated when operating under the additional constraints

[1] Additive spectral models will simply be referred to as spectral models for the remainder of this thesis.

that are imposed by real-time musical performance. One solution to this problem is to identify perceptually salient features from the spectral representation (which are also called "high-level features"), and then perform transformations on these features before synthesis. By manipulating one high-level feature many of the spectral model parameters can be changed at once while still enabling intuitive transformation of the synthesised sound. This process can also be aided by segmenting the sound into regions with homogeneous characteristics, allowing features to be selected based on the attributes of the regions [109].

This thesis presents a body of research that follows this approach. We present novel solutions to the problem of identifying the boundaries of note onset, attack, sustain and release regions during real-time analysis and synthesis. We also describe the creation of new open source software tools for composers, musicians and researchers that enable live audio streams to be manipulated using these techniques. The work culminates in the development of our software for the high-level manipulation of sound called *Metamorph*.

1.3 Thesis overview

Chapter 2 surveys the current state of the art in spectral modelling of musical instrument sounds, and provides context for the remainder of the thesis. It examines the short-time Fourier transform, then describes models of musical sounds composed purely of sinusoids, a combination of sinusoids and noise and then finally a combination of sinusoids, noise and transient signal components. It also provides a brief overview of some of the leading software tools for sound transformations based on spectral models.

Chapter 3 introduces the main software framework for sinusoidal synthesis by analysis that is used in this work, which is called *Simpl*. It starts with a high-level overview of the design of the system, then proceeds to look in detail at the inner workings of the different software modules that are contained in Simpl. The chapter concludes with some examples that explore the functionality of the software.

In Chapter 4 *Modal* is introduced, which is an open source software library for real-time musical note onset detection. Modal also comes with a free database of musical samples, each of which has accompanying metadata that includes hand-annotated locations for each note onset in the sample. The chapter concludes with an explanation of how this sample database was used to evaluate the performance of the software library.

Chapter 5 examines the problems that occur when trying to model transient signal components using sinusoidal and sinusoids plus noise models. It then presents our solution to these problems: an open source software library for real-time musical note segmentation, that separates audio streams into attack (transient), sustain and release regions. The software library is examined in detail, followed by an evaluation of its performance.

Metamorph, an open source software library for the high-level transformation of musical audio, is introduced in Chapter 6. It makes use of the software libraries that are described in Chapters 3, 4 and 5, resulting in a flexible and powerful tool for real-time sound manipulation. The chapter begins by exploring the software library. It then shows how Metamorph can be used as a standalone sound manipulation tool using the C++ or Python programming languages, and how it can be integrated with the Csound system for composition, sound design and synthesis [119].

Finally, Chapter 7 provides a summary of the contributions made by this thesis and some closing remarks. We also suggest some potential ideas for future work.

Chapter 2

Spectral models and transformations of musical sounds

Spectral models are flexible, general-purpose sound models that have been an important research topic in the field of computer music for decades. This chapter provides an overview of some of the major developments in the field, and also serves to introduce important terminology and background knowledge that is assumed in the remainder of the thesis. It begins by describing the Fourier transform and the short-time Fourier transform in Sections 2.1 and 2.2 respectively. An overview of the Phase Vocoder is given in Section 2.3, followed by an examination of some leading approaches to sinusoidal modelling in Sections 2.4, 2.5 and 2.6. Section 2.7 provides an exploration of software packages for sinusoidal modelling and manipulation of musical audio. Finally a summary of the chapter and an assessment of areas that can be improved are given in Section 2.8.

2.1 The Fourier transform

In the 19th century, Jean Baptiste Joseph Fourier showed that any periodic waveform can be represented by a sum of harmonically related sinusoids, each with a particular amplitude and phase. The Fourier transform is a function that computes the set of coefficients (amplitude and phase values) from the time domain waveform. It can be calculated according to Equation 2.1, where t is the continuous time index in seconds and ω is the continuous frequency index in radians per second.

$$X(\omega) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} x(t) e^{-j\omega t} dt \tag{2.1}$$

The Fourier transform returns a set of coefficients $X(\omega)$ that are known as the spectrum of the waveform. These coefficients are are complex numbers of the form $a + jb$. The amplitude $|X(\omega)|$ and phase $\theta(X(\omega))$ value of each sinusoidal component ω can therefore be calculated by rectangular-to-polar conversion, given by Equations 2.2 and 2.3 respectively.

$$amp(X(\omega)) = |X(\omega)| = \sqrt{\Re\{X(\omega)\}^2 + \Im\{X(\omega)\}^2} \tag{2.2}$$

$$\theta(X(\omega)) = tan^{-1} \left(\frac{\Im\{X(\omega)\}}{\Re\{X(\omega)\}} \right) \tag{2.3}$$

No information is lost during a Fourier transform, the original waveform can be unambiguously reconstructed from the Fourier coefficients by a process known as the inverse Fourier transform, given by Equation 2.4.

$$x(t) = \int_{-\infty}^{+\infty} X(\omega) e^{j\omega t} d\omega \tag{2.4}$$

2.1.1 Discrete time signals

In computer music it is less common to work with continuous-time signals, as computers operate on digital audio signals that are sampled at discrete time points. Therefore instead of the signal being represented as a function of continuous-time in $x(t)$, the digital signal is given as a set of discrete values $x(n)$. These samples are taken at a constant rate of one sample every T seconds and so the *sampling rate* is $1/T$ samples per second (Hertz). The *sampling theorem* (or *Nyquist theorem*) states that in order to digitally represent a signal that contains frequency components up to F Hz, the signal must be sampled at with a sampling rate of at least $2F$ samples per second. This is stated formerly by Equation 2.5, where f_s is the sampling rate, F is the highest frequency component in the signal and $2F$ is the *Nyquist rate*.

$$f_s > 2F \tag{2.5}$$

If the sampling rate is lower than $2F$ then a problem called *aliasing* will occur, in which signal components that have a frequency that is greater than half of the sampling rate are represented as components with a frequency that is between 0 Hz and half the sampling rate. The reason for this is illustrated in Figure 2.1; the samples taken from the higher frequency component are indistinguishable from that of the lower frequency component.

To avoid aliasing, the input signal is usually low-pass filtered before sampling to ensure that all the highest frequency component in the signal is less than $f_s/2$ Hz. This process is depicted in Figure 2.2. The sampled version of the signal is not identical to the original signal, but if aliasing has been avoided then the original signal can be recovered to within the quantisation error of the samples.

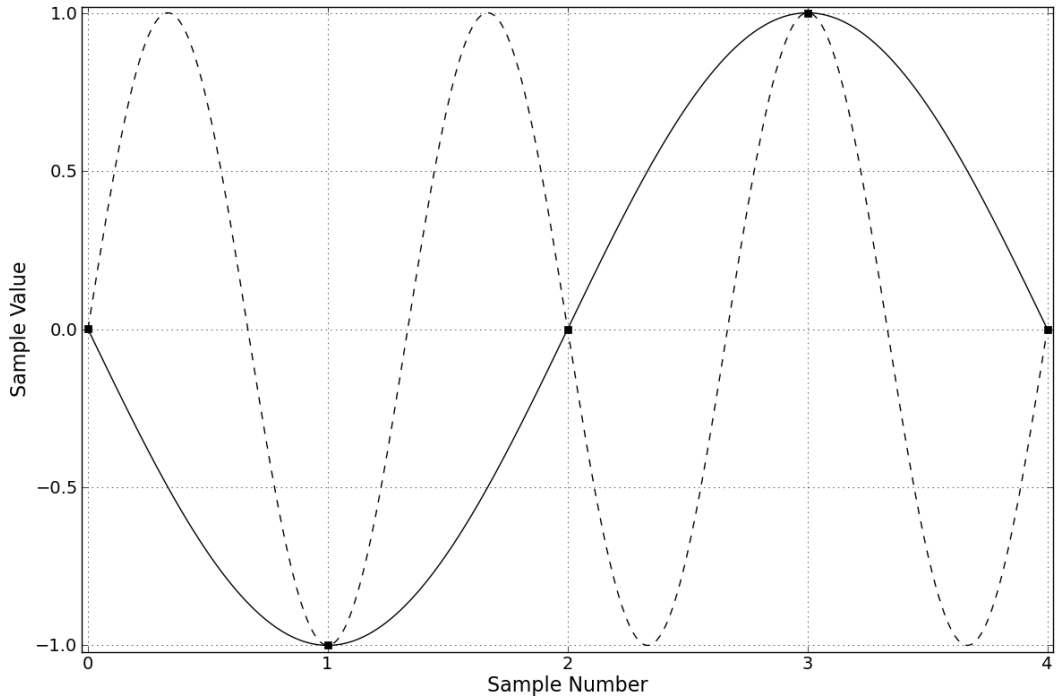

Figure 2.1: A correctly sampled signal (solid line) and a signal that produces an alias (dashed line). The samples from the second signal are indistinguishable to those from the first signal.

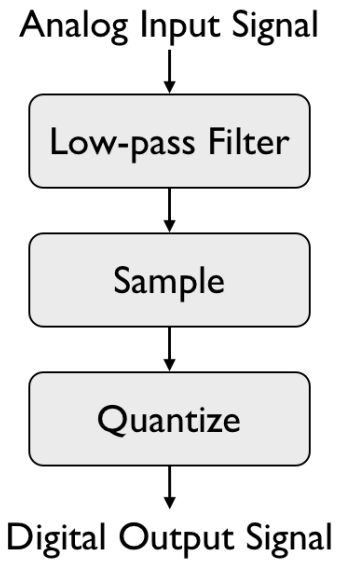

Figure 2.2: Converting a continuous-time signal to a discrete signal.

2.1.2 The discrete Fourier transform

In order to be used with sampled digital audio signals, the continuous-time versions of the Fourier transform and inverse Fourier transform must therefore be replaced with corresponding discrete-time implementations. The discrete Fourier transform (DFT) is calculated according to Equation 2.6, where n is the discrete time index in samples, $x(n)$ is a sampled waveform that is N samples in duration, and k is the discrete frequency index [114, 53, 54, 92, 16].

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \tag{2.6}$$

As with the continuous-time Fourier transform, no information is lost by performing a DFT, and the original sampled waveform can be recovered by Equation 2.7.

$$x(n) = \sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N} \tag{2.7}$$

The sampled signal $x(n)$ is bandlimited in frequency as the DFT represents $x(n)$ as the summation of a finite number of sinusoids that are evenly spaced between 0 Hz and the Nyquist frequency. The DFT can be computed efficiently using a fast Fourier transform (FFT) algorithm. The popular Cooley-Tukey algorithm [24] for example reduces the computational complexity of the DFT from being proportional to N^2 to being proportional to $N\,log\,N$, but requires that N is a power of two.

2.2 The short-time Fourier transform

The frequency and magnitude values that are returned by the DFT are easy to interpret, but the information describing the temporal evolution of the spectral content is entangled in the phase spectrum in a way that is difficult to understand. In order to analyse spectral changes over time, we can sequentially isolate blocks of samples from the time-domain signal (a process called *windowing*) and process them using the DFT. This is known as the short-time Fourier transform (STFT) [3, 17], and is the basis for sinusoidal analysis of time-varying signals in many areas of signal processing.

2.2.1 Windowing

When computing the STFT, the extracted blocks of samples (also known as *frames*) are usually multiplied by another time-domain signal called a *window*. The window is generally a smooth, non-zero function over a finite number of samples and zero everywhere else. The choice of window function is important as it determines the trade-off in time resolution versus frequency resolution in the resulting spectrum. To understand the reason for this, consider what happens when no window is applied to the audio frame. This is equivalent to applying a rectangular window which has a value of 1 throughout the duration of the frame and 0 everywhere else, as defined in Equation 2.8.

$$w(n) = \begin{cases} 1 & n = 0, \ldots, N-1 \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

As multiplication in the time domain is equivalent to convolution in the frequency domain, the waveform that is passed to the DFT function will be the convolution

of the frame spectrum with the spectrum of the rectangular window. As shown in Figure 2.3, the magnitudes of the side-lobes are quite high starting at only -13dB below the peak of the main lobe. Unless the signal that is being analysed is harmonic

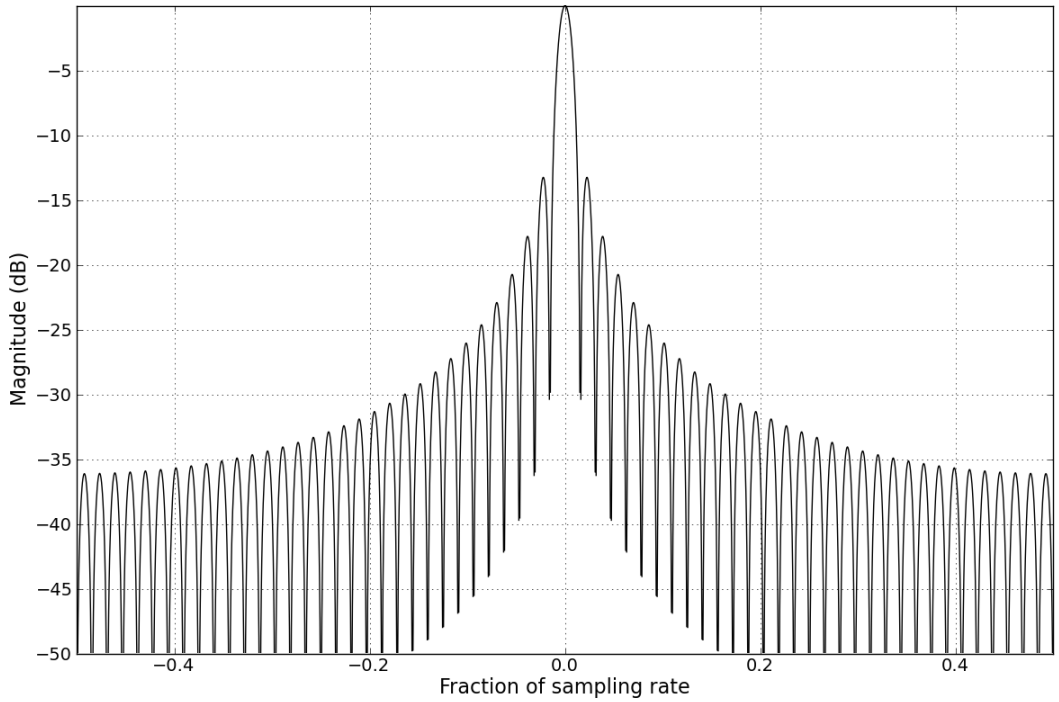

Figure 2.3: Magnitude spectrum of a rectangular window.

with period N (or an integer multiple of N that is less than the Nyquist frequency), these high-side lobes will cause energy from each sinusoidal component to leak into additional DFT channels (or *bins*). As perfectly harmonic signals are relatively rare in a musical context, this interference must be minimised by choosing a window with smaller side-lobes. However, the rectangular window has the narrowest main lobe, so using any other window will reduce (to varying degrees) the ability to resolve sinusoidal components in the resulting spectrum.

A number of different window functions have been proposed such as the Hamming,

Hanning, Blackman and Kaiser, the latter of which allows control of the trade-off between the main lobe width and the highest side-lobe level. A good overview of these windows (and others) can be found in [46]. Examples of two common windows, the Hamming window and the Hanning window, are shown in Figure 2.4, and their corresponding spectra are shown in Figure 2.5. It can be seen that although the main lobes are slightly wider than that of the rectangular window, the side lobes are largely reduced in magnitude. In this research we generally use the Hanning window as it provides a reasonable balance between main lobe and side-lobe interference. However some of the sinusoidal modelling algorithms that we have implemented require other windows, which we describe in more detail in Chapter 3.

2.2.2 Performing the STFT

The STFT of an arbitrary-length signal $x(n)$ at time t is described by Equation 2.9, where $w(n)$ is a window function that is only non-zero between 0 and $N-1$. The process is summarised in Figure 2.6.

$$X(k,t) = \frac{1}{N} \sum_{n=-\infty}^{\infty} w(n-t)x(n)e^{-j2\pi kn/N}, \quad k = 1, 2, \ldots, N-1 \tag{2.9}$$

For each time point, the STFT results in a full spectral frame of N values, so the amount of data that are produced and the computational cost of the STFT are potentially very large. Therefore, instead of performing a STFT at each sample location, it is common to move forward (or *hop*) by a number of samples called the *hop size*.

As the STFT spectra are sampled in both time and frequency it is necessary

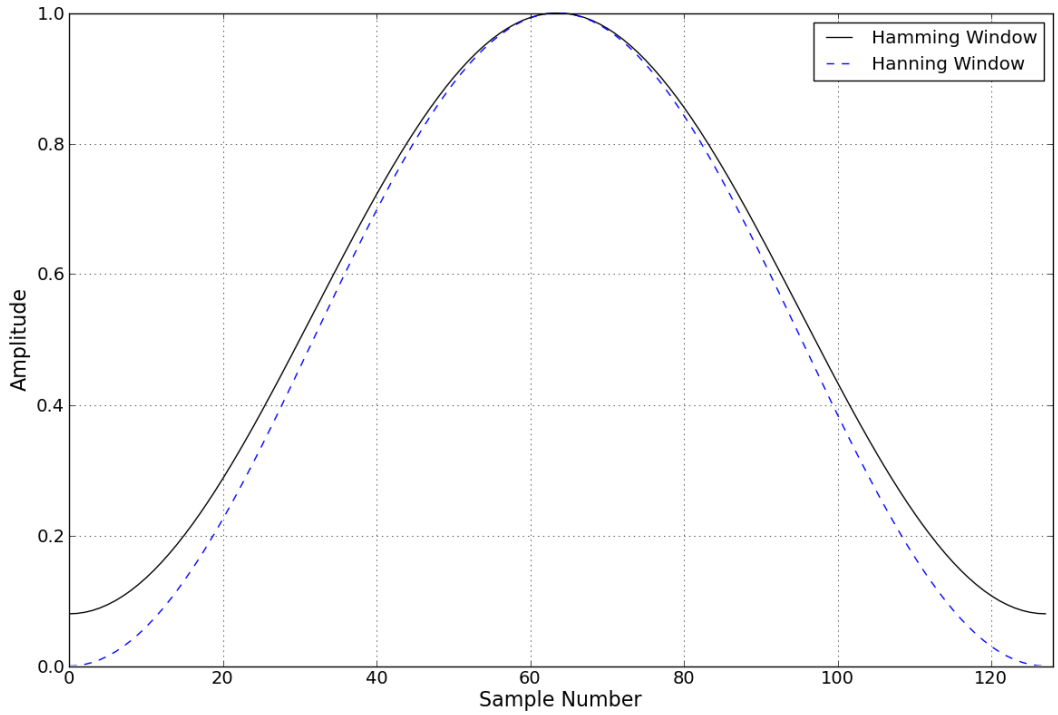

Figure 2.4: 128 point Hamming (solid line) and Hanning (dashed line) windows.

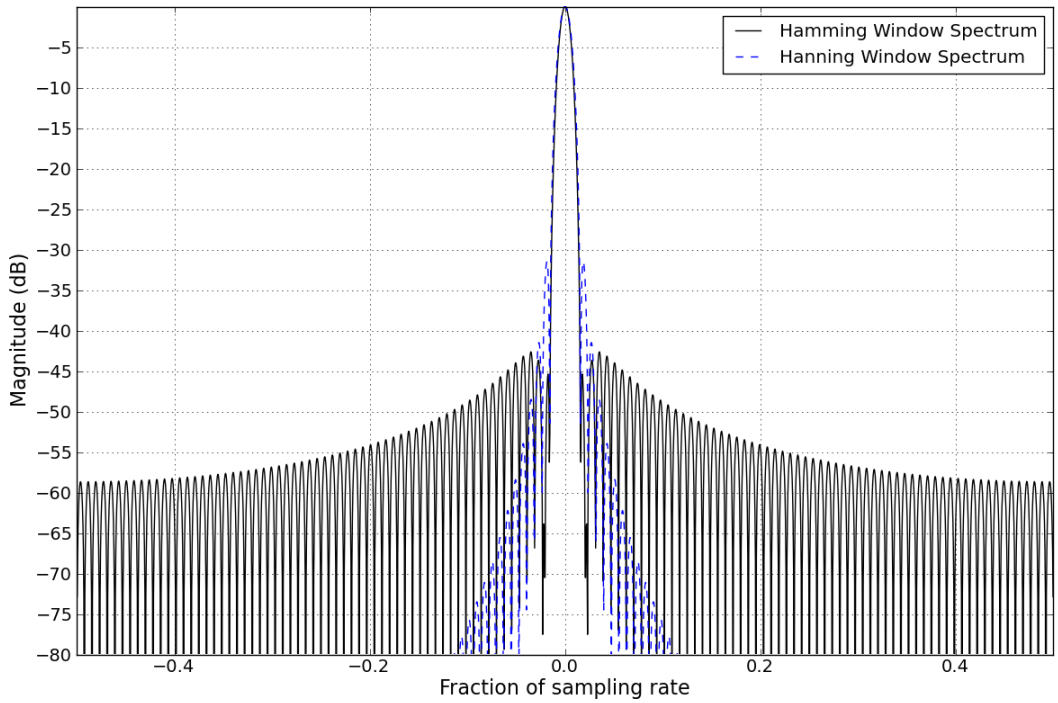

Figure 2.5: Magnitude spectrum of the Hamming (solid line) and Hanning (dashed line) windows.

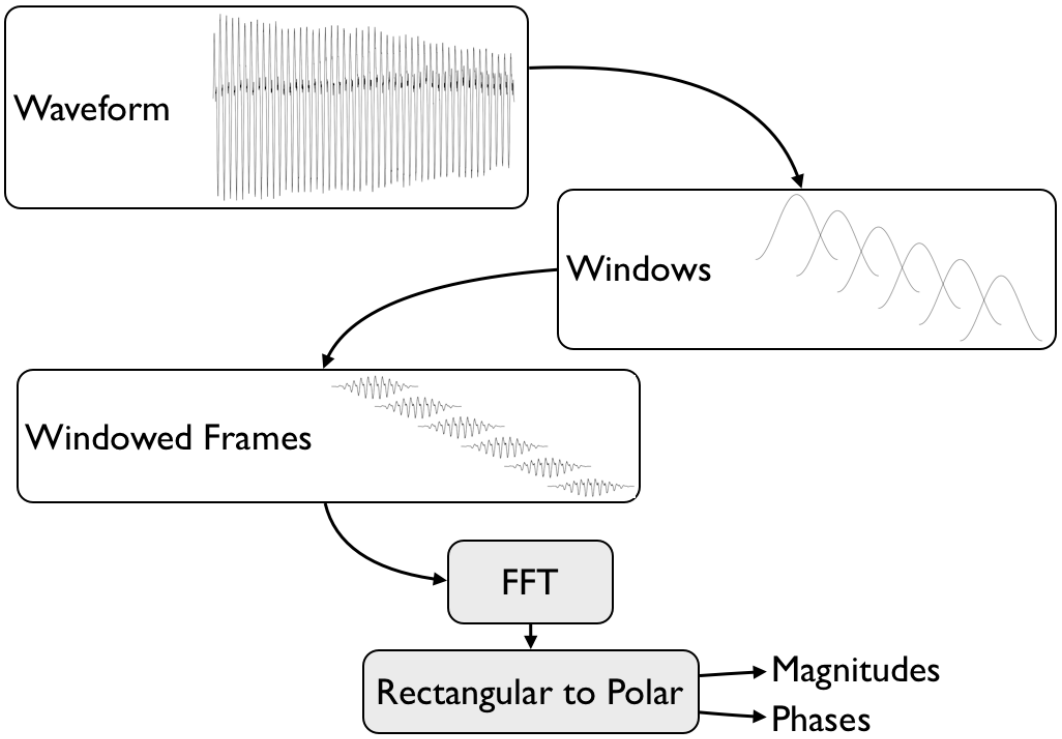

Figure 2.6: The short-time Fourier transform.

to consider the sampling rates in both domains in order create a robust spectral representation. The window function $w(n)$ has two important "lengths". The first length is in the time domain and can be defined as the time period T over which $w(n)$ is significant (non-zero). As the window is generally defined to be zero outside of the range 0 to $N-1$, T is equal to N samples. The second length is the frequency range F over which the Fourier transform of the window is significant. This value will change depending on the window type, but for the Hamming window it can be shown that $F = \frac{4}{N}$ samples [84, 2]. These lengths can then be used to define sampling rates for the time and frequency domains. Due to the Nyquist theorem, the density of the samples in the frequency domain must be greater than T, and so they

may be sampled with a spacing of $\frac{1}{T}$. Similarly, the time domain samples must have a density higher than F and so can be sampled at $\frac{1}{F}$ [2]. For a Hamming window this results in a hop size of $\frac{N}{4}$.

Another important STFT parameter is the size of the DFT frame, as it determines the frequency resolution of the analysis process. As the DFT is usually computed using an FFT algorithm, the frame size is often expected to be a power of two. While this may seem like a limitation, in practise the frame size can be chosen to meet analysis time/frequency resolution requirements, with the FFT size then set to be the next power of two that is greater than the frame size. The difference is filled in with zeros, a process that is known as *zero-padding*. Zero-padding does not add any frequency additional resolution or temporal smearing, but has the effect of interpolating the frequency samples, making the resulting spectrum easier to interpret.

The magnitude of the STFT of a note played on a clarinet is shown in Figure 2.7[1]. Darker colours indicate larger spectral component magnitudes, so relatively stable sinusoidal components are shown as horizontal lines. Lighter colours represent spectral components with lower magnitudes, noise and/or analysis artifacts.

To reconstruct the original signal, the inverse DFT is first performed on each spectral frame to produce a windowed output frame $x_l(n)$, defined in Equation 2.10.

$$x_l(n) = \sum_{k=0}^{N-1} X_l(k) e^{j2\pi kn/N} \tag{2.10}$$

The waveform is then recreated by overlapping the windowed segments and summing them (this process is called *overlap-add* [25]). This is defined by

[1]The STFT magnitude is also known as the *spectrogram*.

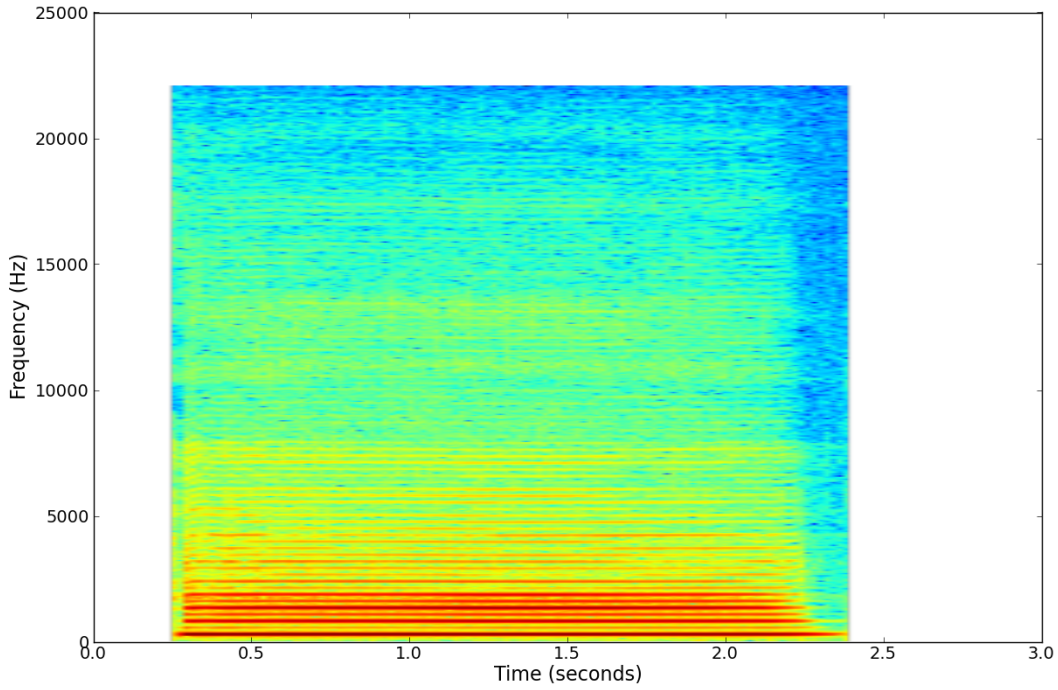

Figure 2.7: Spectrogram of a clarinet note.

Equation 2.11, where l is the frame number and H is the hop size. The original signal can be reconstructed exactly if the sum of the overlapped and added analysis windows is equal to 1 [112].

$$s(n) = \sum_{l=-\infty}^{\infty} x_l(n - lH) \tag{2.11}$$

As it is common to perform transformations on STFT data, a window is usually applied to each synthesised frame before summation in order to remove any potential discontinuities. This introduces additional constraints which are described in [96, 25].

2.3 The phase vocoder

Although it is possible to perfectly reconstruct the original sampled waveform from the STFT data, this research focuses on providing ways to transform spectral analysis information in order to create new sounds. The phase vocoder [36, 29], which is based on the channel vocoder, is a tool that is frequently used for both analysing and modifying musical sounds. Fundamentally it can be seen as an extension of the STFT, consisting of a sequence of overlapping Fourier transforms of audio frames[2]. However, the STFT does not in itself present an easy way to manipulate the frequencies of the sinusoidal components that are present in the analysed signal as it does not have enough resolution to calculate their exact values.

The phase vocoder uses the Fourier transform phase data to form a more accurate estimate of the frequency values. It assumes that the input signal can be modelled as a sum of sinusoidal components with relatively slowly changing amplitudes and frequencies. The components should be spaced far enough apart in the spectrum that only one component contributes significantly to each Fourier transform channel. For each component, it is therefore possible to estimate the frequency by measuring the rate of change of the phase in each channel. The phase deviation for frequency bin k in frame l can be defined by Equation 2.12 where $\phi(k, l)$ is the phase of bin k in frame l, H is the hop size, Δf is the STFT bin spacing (in hertz) and f_s is the sampling rate.

$$\Delta\phi(k, l) = \phi(k, l) - \left[\phi(k, l-1) + \frac{2\pi H k \Delta f}{f_s}\right] \tag{2.12}$$

[2]Like the STFT, the phase vocoder can also be described in the time domain using a bank of bandpass filters with centre frequencies that are evenly spaced between 0 Hz and the Nyquist frequency. For brevity we will only discuss the interpretation that is based on the Fourier transform here.

These phase deviations must then be re-wrapped to the half-open interval $[-\pi, \pi)$, locating the estimated frequency around the centre frequency of the analysis bin. The frequency estimate (in hertz) for bin k in frame l is given by $f(k, l)$ in Equation 2.13, where $\Theta(k, l)$ is the re-wrapped phase deviation.

$$f(k,l) = \left[\frac{\Theta(k,l)}{2\pi}\right]\left[\frac{f_s}{H}\right] + k\Delta f \tag{2.13}$$

This calculation makes it possible to perform independent time and frequency modifications on the spectral data [85, 29, 65]. For example, a sound can be time-stretched by repeating frames during STFT synthesis, but the frequency values can be kept unchanged by altering the synthesis phase values to ensure that the rate of change of phase matches the corresponding analysis phase change rate in each bin [64].

Although the phase vocoder performs well and is now widely used, it does have problems. As the DFT size is usually constant, the channel frequencies are fixed and so the frequency of each sinusoid cannot normally vary outside of the bandwidth of its channel. The phase vocoder channel bandwidth is constant throughout the analysis process, which means that there will inevitably be a trade-off between the accuracy of the estimates of low frequency components and the analysis time resolution.

The phase vocoder also represents the entire audio signal using sinusoids, even if it includes noise-like elements, such as the key noise in a piano note for example or the breath noise in a flute note. This is not a problem when synthesising unmodified Fourier transform frames, but if any transformation is performed, these noisy components are modified along with the harmonic content which often produces

audible artifacts. Modelling noisy components with sinusoids is computationally expensive as theoretically recreating noise requires sinusoids at every frequency within the band limits. Sinusoidal representations of noise are also unintuitive and do not provide a way to manipulate the sound in a musically meaningful way.

To overcome these issues more powerful spectral models were created, which are based on identifying the prominent harmonic components in a signal, then separating the harmonic and noise-like components. These developments are discussed in detail in Sections 2.4, 2.5 and 2.6.

2.4 A sinusoidal model

Two STFT-based systems were developed independently during the 1980's in order to address some of the shortcomings in the phase vocoder. The first was a system designed for speech analysis and synthesis by McAulay and Quatieri[3] [80], with the second being a system created by Smith and Serra called PARSHL [112]. This section provides an overview of the MQ model, PARSHL is not discussed here as it is very similar to MQ. A significant difference in PARSHL however is that the estimates of the sinusoidal peak frequencies are improved by using parabolic interpolation. This process is discussed in the context of Spectral Modelling Synthesis in Section 2.5.1.

The MQ method models audio signals as the sum of sinusoidal components with slowly-varying amplitudes, frequencies and phases (referred to here as *partials*). The audio signal s can be created from a sum of partials according to Equations 2.14 and 2.15, where N_p is the number of partials and A_p, f_p and θ_p are the amplitude,

[3]This is referred to as the *MQ* method for the remainder of the document.

frequency and phase of the p-th partial respectively.

$$s(t) = \sum_{p=1}^{N_p} A_p(t) cos(\theta_p(t)) \tag{2.14}$$

$$\theta_{p(t)} = \theta_p(0) + 2\pi \int_0^t f_p(u) du \tag{2.15}$$

In contrast to the phase vocoder approach, these sinusoidal components do not have to be quasi-harmonic and their frequency can change by more than the bandwidth of one analysis bin between consecutive frames. In addition, the number of sinusoidal components does not have to be constant as partials can start and stop at any time during analysis.

The sinusoidal parameters are estimated from the spectra returned by the STFT. For every frame, the amplitudes, frequencies and phases of the most prominent (in terms of magnitude) spectral peaks are selected, through a process that we refer to as *peak detection*. Corresponding spectral peaks in consecutive frames are then matched together during a process called *partial tracking*, and their values are smoothly interpolated between frames to form the sinusoidal partials. The peak detection and partial tracking processes are summarised in Figure 2.8. Finally, the output signal can be created by additive synthesis.

2.4.1 Peak detection

MQ analysis begins by performing a STFT on contiguous frames of audio from the input waveform. It is assumed that the sinusoidal parameters are approximately constant over the duration of each analysis frame. The waveform of a particular frame s is then modelled by Equation 2.16, where A_p, ω_p and θ_p are the amplitude,

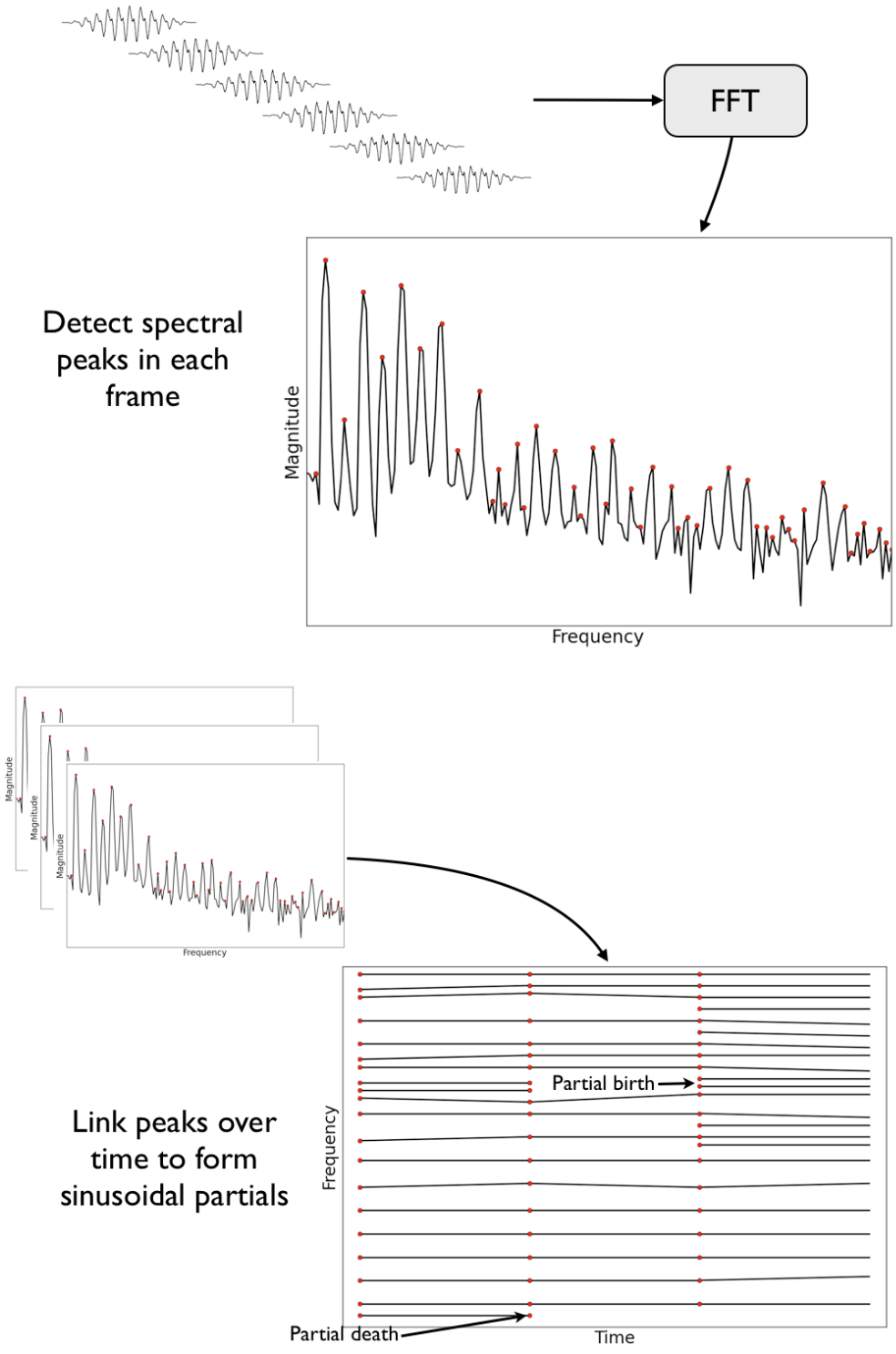

Figure 2.8: MQ peak detection and partial tracking.

frequency and phase of the p-th sinusoidal component respectively, and N_p is the number of sinusoidal components in the frame.

$$s(n) = \sum_{p=1}^{N_p} A_p(n) e^{j(n\omega_p + \theta_p)} \tag{2.16}$$

The problem is now to find a set of sinusoidal parameters that minimises the mean squared error between Equation 2.16 and the actual waveform. In order to simplify this problem slightly, assume that the waveform is perfectly harmonic, the pitch period is known and that the size of the analysis frame is a multiple of that pitch period. The sinusoidal parameters that will minimise this error can then be found at magnitude peaks in the Fourier spectrum of the frame, which are defined as spectral bins that have a magnitude value greater than or equal to that of both neighbouring bins. This is given by Equation 2.17 where k is a bin number of a spectral peak and $|X(k)|$ is the magnitude of the k-th bin.

$$|X(k-1)| \leq |X(k) \geq |X(k+1)| \tag{2.17}$$

This simplified solution can be extended so that it can be applied to other types of signals. For waveforms that are not strictly harmonic, if most of the energy is still concentrated near strong magnitude peaks, then as long as the analysis window is large enough the sinusoidal parameter estimates can still be obtained from the Fourier spectrum. McAulay and Quatieri show that in practise this means that the window size must be at least $2\frac{1}{2}$ pitch periods in order make sure that the main lobe of the DFT window (a Hamming window) is narrower than the frequency separation between spectral peaks [80]. To ensure that this criteria is met, the size of

the analysis window during voiced speech regions is tuned using an pitch detector. Average pitch is used instead of instantaneous, so that the analysis process is less sensitive to the performance of the pitch estimator. During unvoiced speech, the analysis window is kept constant. The model can also be applied to more noise-like signals, providing that the frequencies of the spectral peaks are close enough together that the power spectral density changes slowly over consecutive frames, and thus meet the requirements imposed by Karhunen-Loève expansion [117]. This should be achievable as long as the analysis window is at least 20 ms in duration, so that the peaks are no more than about 100 Hz apart.

2.4.2 Partial tracking

The number of spectral peaks will generally vary from frame to frame for a number of reasons: some peaks will be the result of side-lobe interference, the number and location of peaks will change as the pitch of a sound changes, and different spectral characteristics will be evident in signal frames that have mostly harmonic content versus frames that are more inharmonic or noise-like. Therefore, some system is needed to "match" peaks that are considered to be part of the same slowly-varying sinusoidal component across consecutive frames, while making sure that the partial is not "corrupted" by spurious peaks.

The MQ system manages this by allowing partials to start and stop at any frame boundary through a process that they call the "birth" and "death" of sinusoidal components. To understand how this works, suppose that all of the peaks up to frame l have been matched. The peaks in frame $l+1$ are now being considered and ω_k^l refers to a peak with frequency k in frame l. First, find the peak ω_x^{l+1} in $l+1$ that

is closest in frequency to ω_k^l and is within a specified matching interval Δ of ω_k^l, as described in Equation 2.18.

$$|\omega_k^l - \omega_x^{l+1}| \geq \Delta \tag{2.18}$$

If no such peak exists, then ω_k^l is declared "dead" in frame $l+1$ and is matched to itself with 0 amplitude. If ω_x^{l+1} exists and ω_k^l is the peak that is closest in frequency to ω_x^{l+1} in frame l, then the two peaks are matched and cannot be considered when matching the remaining peaks in $l+1$. If ω_x^{l+1} exists but has a closer match in l than ω_k^l, then an attempt is made to match ω_k^l to a peak that is lower in frequency than ω_x^{l+1} but still within the matching interval Δ. If this is not possible, then ω_k^l is "killed". Finally after all of the peaks in l have either been matched or killed, there may still be unmatched peaks in $l+1$. These are "born" in $l+1$ and matched to a new peak in l with the same frequency and 0 amplitude.

2.4.3 Synthesis

In order to avoid discontinuities at frame boundaries during synthesis it is necessary to smoothly interpolate the amplitudes, frequencies and phases that are estimated during MQ analysis between adjacent frames. One way to achieve this is to use the overlap-add process that was described in relation to the STFT in Section 2.2.2. In [80] McAulay and Quatieri describe two overlap-add synthesis implementations, both using triangular windows that are twice the size of the synthesis frame (and so have an overlap factor of 2). The first one had a window size of 11.5 ms and the second had a window size of 23 ms. They note that the first one produced high-quality synthesis that was practically indistinguishable from the original, however the second sounded "rough" and was of poor-quality. Additive synthesis with overlap-add

can therefore be used if higher frame rates are acceptable. However, as the MQ system was originally developed for the domain of speech coding, McAulay and Quatieri also devised a synthesis algorithm that could produce high-quality results at lower frame rates.

After partial tracking, sinusoidal parameters for a frame l are matched with a corresponding set of parameters for frame $l + 1$. The amplitude, frequency and phase of the p-th partial can be denoted by A_p^l, ω_p^l and θ_p^l. The amplitude values for each component can be linearly interpolated across frames, as described in Equation 2.19, where $n = 0, 1, \ldots, N - 1$ is the time index into the l-th frame.

$$A_p(n) = A_p^l + \frac{A_p^{l+1} - A_p^l}{N} n \tag{2.19}$$

The phase and frequency components cannot be linearly interpolated however, as an extra constraint is imposed by the fact that instantaneous frequency is the derivative of phase. The phase measurements are modulo 2π, and as the phase differences between consecutive frames can easily exceed 2π (particularly for higher frequencies), the phase must first be unwrapped. To solve this problem suppose that the phase interpolation function is a cubic polynomial as given in Equation 2.20, where t is the continuous time variable, α and β are coefficients that satisfy the matrix equation defined in Equation 2.21, M is an integer and N is the length of the frame.

$$\hat{\theta}_p^l(t) = \theta_p^l + \omega_p^l t + \alpha t^2 + \beta t^3 \tag{2.20}$$

$$\begin{bmatrix} \alpha(M) \\ \beta(M) \end{bmatrix} = \begin{bmatrix} \frac{3}{T^2} & \frac{-1}{T} \\ \frac{-2}{T^3} & \frac{1}{T^2} \end{bmatrix} \begin{bmatrix} \theta_p^{l+1} - \theta_p^l - \omega_p^l N + 2\pi M \\ \omega_p^{l+1} - \omega_p^l \end{bmatrix} \tag{2.21}$$

An additional constraint is imposed in that it is desirable that the interpolated sinusoidal components are "maximally smooth", or have the least possible amount of variation. It can be shown that in order to achieve this, M should be set to the closest integer to x^* in Equation 2.22 [80].

$$x^* = \frac{1}{2\pi}\left[(\theta_p^l + \omega_p^l N - \theta_p^{l+1}) + (\omega_p^{l+1} - \omega_p^l)\frac{N}{2}\right] \tag{2.22}$$

The final waveform s of frame l can then be constructed by additive synthesis using Equation 2.23.

$$s^l(n) = \sum_{p=1}^{N_p} A_p^l(n) cos(\hat{\theta}_p^l(n)) \tag{2.23}$$

2.5 Sinusoids plus noise

The sinusoidal model presented in Section 2.4 overcame one of the problems with the phase vocoder; the frequencies of the sinusoidal components could vary outside of the DFT channel bandwidth. It can also produce high-quality analysis and synthesis of a wide variety of musical sounds. However, as it models all sounds as combinations of sinusoids, it does not provide an intuitive or flexible way to manipulate sounds that contain noise components. For example, when performing a pitch modification on a piano note it is not usually desirable to also alter the key and hammer noise, but this is unavoidable in a purely sinusoidal representation of sound. This section examines two systems that were developed in order to address this issue, spectral modelling synthesis and bandwidth-enhanced sinusoidal modelling, which are described in Sections 2.5.1 and 2.5.2 respectively.

2.5.1 Spectral Modelling Synthesis

Serra and Smith [108] created a hybrid sinusoids plus noise model of musical sound called Spectral Modelling Synthesis (SMS). SMS assumes that a sound is composed of a combination of a deterministic component and a residual or stochastic component, as defined by Equation 2.24 where s is the synthesised waveform, e is the residual signal, A is the instantaneous amplitude and θ is the instantaneous phase.

$$s(n) = \sum_{p=1}^{N_p} A_p(n) cos(\theta_p(n)) + e(n) \tag{2.24}$$

Deterministic in this context means that it can be accurately modelled as a sum of slowly varying quasi-sinusoidal components. The residual component is then defined as being the difference between the deterministic component and the original sound. In order to allow for more flexible sound transformations the residual can be modelled separately and treated as a stochastic component, where the signal is assumed to be approximately noise-like and so can be represented by just the general shape (or envelope) of its spectrum and then synthesised as filtered noise. An overview of the SMS system is given in Figure 2.9.

Peak detection

The peak detection process in SMS in quite similar to the MQ method. The input signal is first analysed using the STFT, yielding a sequence of frames of magnitude and phase data. The magnitude spectrum is then searched for spectral peaks. An important addition in SMS however is the use of parabolic interpolation to form more accurate estimates of spectral peak frequency values, a technique that was

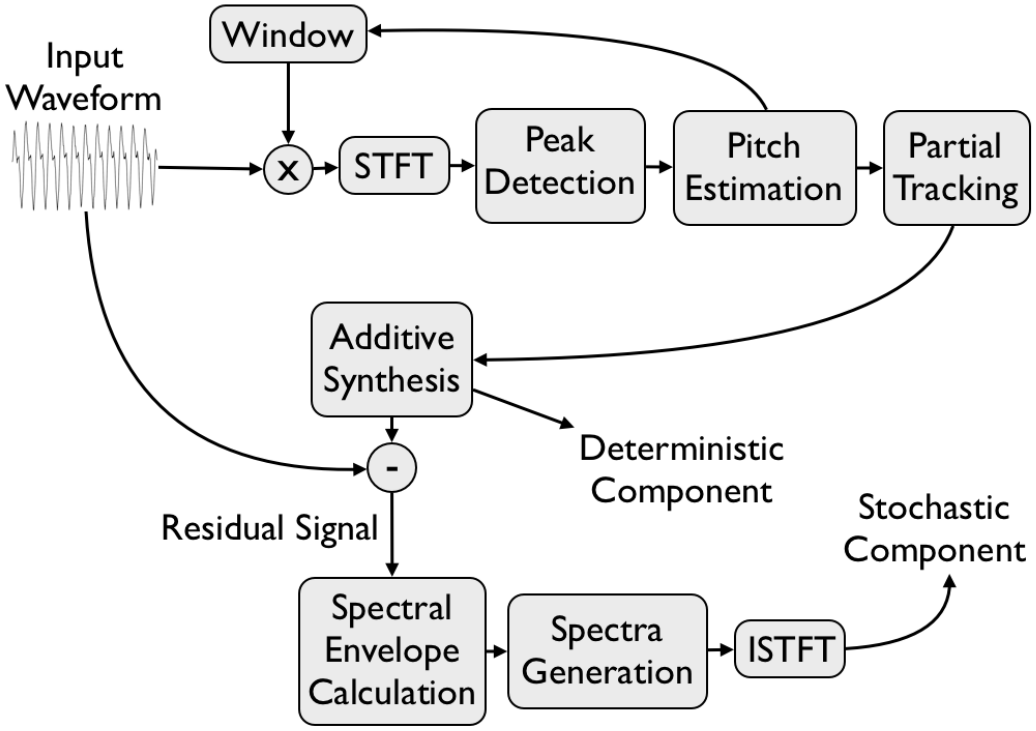

Figure 2.9: The SMS analysis/synthesis method.

first introduced by Smith and Serra in PARSHL [112]. As the STFT returns sampled spectra, each peak is only accurate to within half a sample. Zero-padding in the time domain increases the number of DFT bins (spectral samples) per frame, which in turn increases the accuracy of the peak frequency estimates, but it is computationally expensive as very large frame sizes can be required in order to achieve high accuracy in frequency estimation. A more efficient technique for obtaining accurate frequency estimates is to use a small zero-padding factor, then fit a parabola between the detected spectral peak and the two adjacent bins. The peak frequency can then be taken to be the frequency of the fractional sample location at the vertex of the parabola.

The parabolic interpolation process begins by first defining a coordinate system centred at $(k_\beta, 0)$, where k_β is the bin number of a spectral peak. The goal is to find a general parabola of the form given in Equation 2.25, where c is the centre of the parabola, a is a measure of the concavity and b is the offset.

$$y(x) = a(x - c)^2 + b \tag{2.25}$$

This can be rewritten to give the peak location (Equation 2.26). The estimated true peak location is described by Equation 2.27 and the true magnitude value then estimated according to Equation 2.28, where α, β and γ are the magnitudes (in decibels) of the bins with x coordinates at -1, 0 and 1 respectively.

$$c = \frac{1}{2}\frac{\alpha - \gamma}{\alpha - 2\beta + \gamma} \tag{2.26}$$

$$k^* = k_\beta + c \tag{2.27}$$

$$y(c) = \beta - \frac{1}{4}(\alpha - \gamma)c \tag{2.28}$$

As the accuracy of this parabolic interpolation technique depends on how closely the parabola corresponds with the shape of the magnitude spectrum at the peak, it is therefore influenced by the spectral shape of the analysis window [108].

Partial tracking

The partial tracking algorithm in SMS differs from its counterpoint in the MQ method in that SMS only attempts to match spectral peaks that are part of stable underlying sinusoidal components. MQ analysis by comparison will create partials

any time time point if necessary in order to incorporate new peaks, as the goal is to build a sinusoidal representation of the entire sound. During SMS partial tracking, a set of *frequency guides* advance through time, trying to select appropriate spectral peaks in each frame and matching them to form partials (called *trajectories* in SMS).

In each frame, guides are advanced by finding the spectral peak that is closest in frequency to each guide, as long as the frequency difference is less than a specified *maximum-peak-deviation* parameter. If a match is found and no other guide also wants to select that particular peak, the guide continues and the peak is added to the corresponding trajectory. If more than one guide wants to match with the same peak, the guide that is closest in frequency is declared the "winner" and the other guide must look for a different peak. If no match is found, the trajectory is "turned off" in frame l by being matched to itself with 0 amplitude. If the trajectory has been turned off for more than the value of the *maximum-sleeping-time* parameter, it will be "killed".

The next step is to update the guide frequency for each guide that successfully found a matching peak. The frequency is updated according to Equation 2.29. f_g is the new frequency of guide g, $\tilde{f}_g$ is the current frequency of guide g, ω is the frequency of the peak that the guide was matched to and α is the *peak-contribution-to-guide* parameter.

$$f_g = \alpha(\omega - \tilde{f}_g) + \tilde{f}_g, \quad \alpha \in [0, 1] \tag{2.29}$$

If there are unmatched peaks in a frame, new guides and corresponding trajectories will be created for each one as long as the current number of guides is less than

a specified maximum. Guides are created for the largest magnitude peaks first, and must be separated from all existing guides by a minimum frequency value, specified by the *minimum-starting-guide-separation* parameter. The guide frequency is the frequency of the spectral peak. Guides are initially buffered and only marked as "normal guides" when they have existed for a given number of frames in order to ensure that short-lived guides and trajectories are avoided.

An additional partial tracking parameter exists in SMS that allows the input sound to be labelled as being harmonic. In this case a number of changes are made to the partial tracking algorithm: there is a specific fundamental frequency for each frame, the number of guides remains constant throughout the sound and each guide tracks a specific harmonic. To estimate the fundamental frequency in each frame, the three peaks with the largest magnitudes are selected and then a search is performed for a peak which would be a suitable fundamental for all three.

SMS synthesis

The deterministic component is synthesised using the same process as MQ synthesis, described in Section 2.4.3. The sinusoidal parameters are obtained from the spectral peaks that are matched to form the SMS trajectories.

Residual component

As the analysis phase information is preserved during deterministic synthesis, it is possible to compute the residual signal by simply subtracting the deterministic component from the original waveform in the time-domain. If a different synthesis method is used and the phase information is lost, it is still possible to obtain the residual component by performing the subtraction in the frequency domain. This is

described in Equation 2.30, where l is the frame number, $|X_l(k)|$ is the magnitude spectrum of the original waveform, $|D_l(k)|$ is the magnitude of the deterministic component and $|E_l(k)|$ is the magnitude of the residual. The resulting residual waveform can then be synthesised using the inverse STFT.

$$|E_l(k)| = |X_l(k)| - |D_l(k)| \tag{2.30}$$

Ignoring phase

The objective of SMS is to create a flexible model for sound manipulation. As this only requires that synthesised sounds be perceptually similar to the original sounds, and not mathematically identical, there is strictly no need to preserve the analysis phases during deterministic synthesis. The residual component can still be calculated using frequency domain subtraction as discussed in Section 2.5.1. In this case, the synthesised waveform can still be created by Equation 2.24, but each sinusoidal component can now be described by only its amplitude and frequency parameters. The instantaneous phase is then taken to be the integral of the instantaneous frequency. SMS peak detection and partial tracking can proceed as before, but each spectral peak only needs to retain amplitude and frequency values from the STFT spectrum.

Stochastic component

The residual signal should consist mainly of inharmonic signal components. In order to recreate sounds that are perceptually similar to the original waveform, the residual can simply be stored as raw samples and added to the synthesised deterministic component. However, as it is desirable to be able to perform transformations

on the analysed sound, a more flexible sound model can be created if it is assumed that the residual signal is largely noise-like or stochastic. The magnitude spectrum of each residual frame can then be approximated by its spectral envelope, enabling the component to be modelled as filtered white noise according to Equation 2.31. $u(t)$ is white noise and $h(t, \sigma)$ is the impulse response of a slowly time-varying filter.

$$\hat{e}(n) = \int_0^n h(t, t-\tau)u(\tau)d\tau \tag{2.31}$$

The residual spectral envelope is computed by line-segment approximation. Contiguous windows of size M are extracted from the magnitude spectrum and the largest magnitude value in each window is selected, resulting in $Q = N/M$ equally spaced points, where N is the frame size. The envelope is formed by connecting straight lines between these points. The accuracy of the envelope is dictated by the number of points Q, which can be adjusted depending on the complexity of the sound. An example of a residual spectrum and the resulting SMS spectral envelope is given in Figure 2.10.

Stochastic synthesis can be interpreted as applying a time varying filter to a white noise signal. However, in SMS each stochastic frame is created using the inverse STFT overlap-add synthesis technique (as described in Section 2.2.2), with the spectra obtained from the spectral envelopes. The spectral magnitude values are generated by linear interpolation of the Q envelopes samples to make a curve of length $N/2$, where N is the FFT size. There is no phase information in the stochastic envelope, but as the synthesised signal is intended to be noise-like then the phases for each frame can be created by mapping the output of a random number generator to numbers in the range $[0, 2\pi]$.

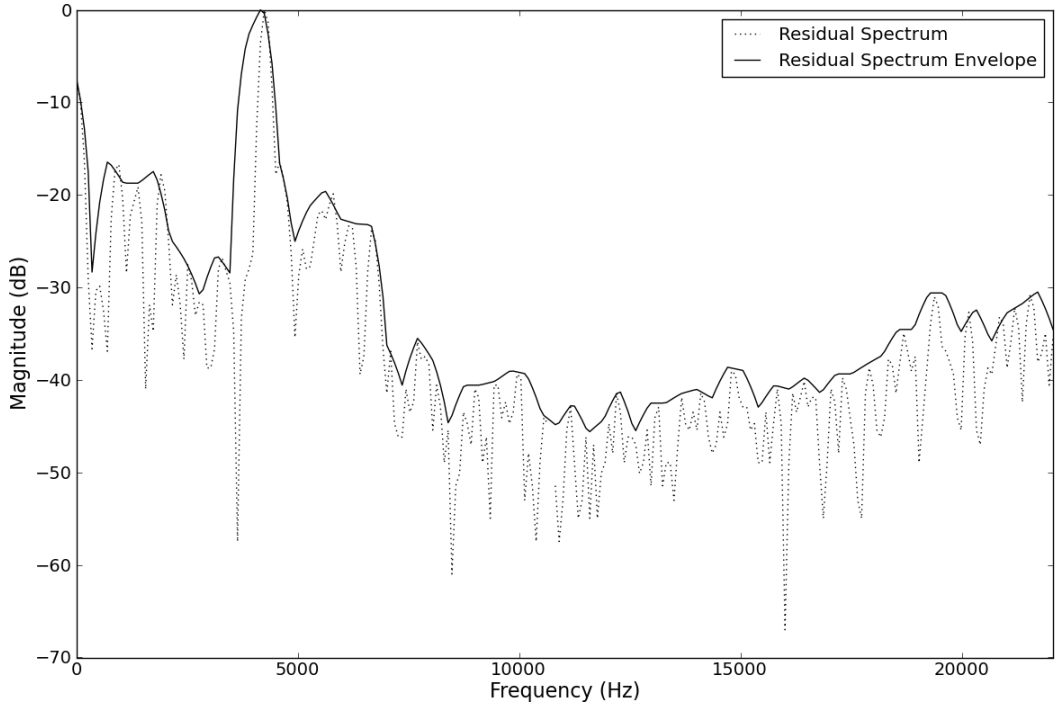

Figure 2.10: The spectrum of the residual component from a piano tone and the SMS spectral envelope.

2.5.2 Bandwidth-enhanced sinusoidal modelling

An alternative solution to the problem of adding a representation for noisy signals to the sinusoidal model was proposed by Fitz [34]. Instead of the SMS approach of having independent deterministic and stochastic components, the Fitz system uses a single type of component with a new oscillator called a *bandwidth-enhanced oscillator*. This is basically a sinusoidal oscillator that is ring-modulated by noise, which results in the noise band being centred around the oscillator frequency. The overall noise levels in different parts of the spectrum can therefore be adjusted by varying the noisiness of each bandwidth-enhanced oscillator. This approach retains the homogeneity of the purely sinusoidal model, allowing components to be edited and

transformed uniformly. Fitz notes that this is of particular interest when performing timbre morphing in [34], where he presented a synthesis by analysis system that uses these noise-modulated oscillators that is known as *bandwidth-enhanced sinusoidal modelling*.

Bandwidth-enhanced oscillators

A spectrum consisting of a combination of quasi-harmonic components and other noise-like components can be synthesised using narrow-band noise generators with variable amplitudes, centre frequencies and bandwidths. These generators can be defined by Equation 2.32, where ς_n is a white (or wideband) noise sequence that excites the filter h_n and Δ_w is the desired bandwidth of the resulting noise component. ω_c is the centre frequency of the sinusoid and β controls the amplitude of the noise.

$$y(n) = \beta[\varsigma_n * h_n(\Delta_w)] \cdot e^{j\omega_c n} \tag{2.32}$$

If many of these generators are used and their centre frequencies are close together (relative to their bandwidths), then the generators can be simplified by having fixed bandwidths and letting β control the amount of noise energy in a given spectral region. This simplified oscillator is given by Equation 2.33.

$$y(n) = \beta[\varsigma_n * h_n] \cdot e^{j\omega_c n} \tag{2.33}$$

If the bandlimited noise is thought of as being the modulating signal then this can be written according to Equation 2.34, where A is the sinusoidal amplitude. This is known as a bandwidth-enhanced oscillator. A block diagram for the bandwidth-

enhanced oscillator is given in Figure 2.11.

$$y(n) = (A + \beta[\varsigma_n * h_n]) \cdot e^{j\omega_c n} \tag{2.34}$$

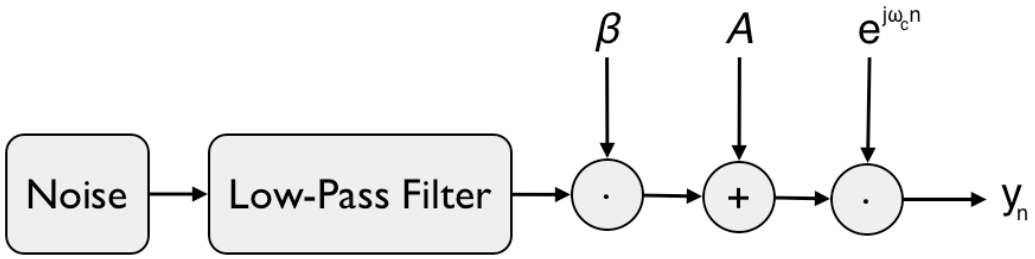

Figure 2.11: The bandwidth-enhanced oscillator.

Peak detection

The peak detection process for the bandwidth-enhanced sinusoidal model is similar to the SMS peak detection system as described in Section 2.5.1. However, instead of using parabolic interpolation to improve the accuracy of the sinusoidal component frequency parameters, the accuracy of the peak detection in the bandwidth-enhanced model is improved by a process called *reassignment*. Reassignment was initially introduced as the modified moving window method [60]. It works by localising spectral components to the centre of gravity of their spectral energy distribution, which is computed from partial derivatives of the short-time phase spectrum in STFT analysis frames. This produces more accurate time and frequency estimates than the STFT, as components in the latter are localised at the geometrical centre of the analysis window. However, it can be shown that the reassignment frequency

estimates are equivalent to performing phase vocoder frequency analysis with a hop size of 1 sample [43, 76].

Reassignment is based on the principle that for periodic signals, the variation in the Fourier phase spectrum that is not attributable to periodic oscillation is slow with respect to frequency near the frequency of oscillation and rapid in other areas. Similarly for impulsive signals, the variation in the phase spectrum is slow with respect to frequency near the time of the impulse and rapid elsewhere. During synthesis these slowly changing regions make the most significant contribution to the signal energy, as the areas with rapidly varying phase tend to destructively interfere. The time and frequency coefficients should therefore satisfy Equations 2.35 and 2.36, where $\phi(\tau, \omega)$ is the continuous phase spectrum and $\omega \cdot (t - \tau)$ is the phase travel due to periodic oscillation for an analysis frame centred at time $t = \tau$ [60, 34].

$$\frac{\partial}{\partial \omega}[\phi(\tau, \omega) + \omega \cdot (t - \tau)] = 0 \tag{2.35}$$

$$\frac{\partial}{\partial \tau}[\phi(\tau, \omega) + \omega \cdot (t - \tau)] = 0 \tag{2.36}$$

These conditions can be satisfied using the instantaneous frequency (Equation 2.37) and the group delay (Equation 2.38).

$$\hat{t} = \tau - \frac{\partial \phi(\tau, \omega)}{\partial \omega} \tag{2.37}$$

$$\hat{\omega} = \frac{\partial \phi(\tau, \omega)}{\partial \tau} \tag{2.38}$$

The reassigned frequency $\hat{\omega}$ and time $\hat{t}$ coordinates can be expressed in terms of STFTs with different window functions [6] according to Equations 2.39 and 2.40,

where X_h is the STFT of the input signal calculated using the window function h, X_{th} is the STFT windowed using a time ramped version of h and X_{dh} is the STFT windowed using the first derivative of h.

$$\hat{\omega}(\omega, t) = \omega + \Im \left\{ \frac{X_{dh}(\omega, t) X_h^*(\omega, t)}{|X_h(\omega, t)|^2} \right\} \tag{2.39}$$

$$\hat{t}(\omega, t) = t - \Re \left\{ \frac{X_{th}(\omega, t) X_h^*(\omega, t)}{|X_h(\omega, t)|^2} \right\} \tag{2.40}$$

The magnitude spectrum can now be searched for peaks using a similar process to the MQ method, with the peak frequency parameter taken to be the reassigned frequency value.

Partial tracking

Partial tracking in the bandwidth-enhanced sinusoidal model is based on the MQ partial tracking method but includes some important alterations. Peaks with large time reassignments in a given frame represent events that are far from the centre of the analysis window. They are therefore not considered for matching in that frame as they are deemed to be unreliable. This does not exclude these peaks from the final sinusoidal representation however, as the analysis windows are overlapping and therefore these components will usually be closer to the centre of either an earlier or later window.

The cubic phase interpolation introduced in the MQ system works well when performing exact reconstruction of the original signal. However when applying transformations to the sinusoidal representation it becomes mathematically complex to try to preserve the original phases at all times. Therefore, the bandwidth-

enhanced model only preserves the phase at the start of a partial and after that phase is taken to be the frequency integral.

There is also an important change relating to the concept of the "birth" and "death" of partials. In general, when a partial tracking algorithm considers only a subset of the peaks in a frame for inclusion in the spectral representation, a magnitude threshold is often applied in order to exclude insignificant peaks. When the amplitude of a partial is close to this threshold, it can vary from being just above it to just below it (and vice-versa) in consecutive frames, resulting in the partial being "killed" and "born" repeatedly. As partials that are killed are faded out over the duration of one frame, this repeated ramping of the partial amplitude can produce audible artifacts. SMS allows guides to "sleep" for a number of frames but the amplitude of the partial is still reduced to zero while the guide is inactive.

The bandwidth-enhanced model uses hysteresis in the thresholding algorithm to try to alleviate this problem. This technique that was first introduced in Lemur [35]. Effectively two separate thresholds are created: one for partial birth and another (lower) threshold for death. Partials can therefore drop below the initial birth threshold by a given amount and continue to be synthesised with the measured STFT magnitude instead of being faded out.

Bandwidth association

After the prominent spectral components have been identified and linked to form sinusoidal partials, the procedure of determining how much noise energy should be added to each bandwidth-enhanced partial can begin. This process is called *bandwidth association*.

A simple approach is to try to ensure that the spectral energy distribution in

the bandwidth-enhanced model matches that of the STFT frames. However, Fitz reported that attempts to assign bandwidth to partials based on matching energy values in different spectral regions were generally unsuccessful [34]. This method proved too sensitive to slight changes in the spectrum of the input signal over time, producing bandwidth envelopes that resulted in audible artifacts. The failure of this approach was attributed to the fact that signal energy is a physical measurement, but as the intention is to create a perceptually accurate model then ideally the bandwidth association process should take the human perception of spectral noise into account.

The solution to the bandwidth association problem was to assign noise energy to partials based on *loudness* matching. Loudness is a measure of the perceived intensity of a sound, where the intensity for a sinusoid is its root-mean-square (RMS) amplitude multiplied by the constant $\frac{1}{\sqrt{2}}$. The relationship between signal intensity and loudness is generally complex but several key insights have been established through psychoacoustic experiments. If a group of N narrowband tones have centre frequencies that are are within a bandwidth known as the *critical bandwidth*, the loudness L can be estimated according to Equation 2.41. I_n is the intensity of the n-th tone and C is a function of the centre frequency of the aggregate of the tones ω_c [103].

$$L = C(\omega_c)\sqrt[3]{I_1 + I_2 + \cdots + I_N} \tag{2.41}$$

If the frequencies of tones are separated by more than the critical bandwidth, the loudness of their combination is approximately the sum of their individual loudness levels. The *bark* frequency scale is often used in conjunction with perceptual frequency measurements as it is constructed so that critical bands all have a width of one bark. If f is frequency in hertz, then the bark frequency b is approximated by

Equation 2.42.

$$b = 13\,tan^{-1}\left(\frac{0.76f}{1000}\right) + 3.5\,tan^{-1}\left(\frac{f}{7500}\right)^2 \tag{2.42}$$

The bandwidth-enhanced model assigns noise energy to partials according to the measured loudness level in overlapping regions that are distributed evenly in bark frequency. Regions are weighted so that sinusoidal components that occupy more than one region make weighted contributions to the loudness measurement in all regions, with the largest contribution made in the region with the centre frequency closest to that of the partial. The regions are defined to be narrower in frequency than the critical bandwidth, and so loudness for each region can be calculated by Equation 2.41. To simplify the calculations it is assumed that loudness does not vary with frequency within a region.

The difference between total partial energy and measured STFT energy in a region is the cube of the difference of the loudness measurements between the two, as loudness is calculated from the cube root of energy. This can be written according to Equation 2.43 where ΔE_r is the energy difference for region r, $L_r(X)$ is the loudness computed from the STFT magnitude X in r and $L_r(A)$ is the loudness of the sinusoidal partials in r.

$$\Delta E_r = (L_r(X) - L_r(A))^3 \tag{2.43}$$

This energy difference is distributed as noise bandwidth among the partials in the region according to their relative contribution to the loudness level. The energy assigned to a partial p in r is given by $D_r(p)$ is given by Equation 2.44, where $\alpha_r(p)$

is the relative energy contribution of p to region r (and is therefore dependent on the region weightings) and N_p is the number of partials in the region.

$$D_r(p) = \frac{\alpha_r(p)}{\sum_{j=0}^{N_p} \alpha_r(j)} \Delta E_r \tag{2.44}$$

The total bandwidth assigned to a partial p is the sum of the energy assigned to it in all regions.

Synthesis

Intuitively, analysis data from the bandwidth-enhanced sinusoidal model can be synthesised via additive synthesis of each bandwidth-enhanced oscillator as described in Equation 2.45, where s is the synthesised waveform, N_p is the number of oscillators, and y_p is the output of the p-th oscillator as specified in Equation 2.34. Fitz notes in [34] that the model may also be used in conjunction with inverse STFT synthesisers, or indeed any synthesis engine that is able to generate both noise and sinusoids.

$$s(n) = \sum_{p=0}^{N_p} y_p(n) \tag{2.45}$$

2.6 Sinusoids plus noise plus transients

The additions to the basic sinusoidal model that were introduced in Section 2.5 address the problems that can result from trying to model noise-like signal components using sums of sinusoids. These models assume that these noise components are part of a series of slowly evolving spectra. There is another class of signal components however that present problems to sinusoidal representations, namely

transients. While it is possible to model transient signals using sinusoids it does not lead to meaningful transformation possibilities as transients signals are by definition very concentrated in time.

As transient signals approach being purely impulsive, they produce spectra that have energy distributed across a greater number of analysis frequency bins. This creates a similar set of problems for sinusoidal models to the difficulties that result from working with more slowly-varying noise-like components. However, they are not ideally handled by the noise models in SMS or the bandwidth-enhanced model as analysis data is interpolated on a frame-by-frame basis, which can have the effect of "smearing" transients over longer durations.

This section looks at two systems that have tried to address this problem by adding an explicit model for transients to a sinusoids plus noise representation. The first was introduced by Masri and is discussed in Section 2.6.1 while the second system, developed by Verma and Meng, is described in Section 2.6.2.

2.6.1 Improving the synthesis of attack transients

Masri introduced a model of musical sounds consisting of a combination of sinusoidal, noise and transient signal components[4] [77]. The addition of a transient model required several important changes to the analysis process. Firstly, transient events in the input signal must be identified and their duration must be estimated. Adjustments then had to be made to the sinusoidal and noise analysis processes in order to accommodate the new transient components.

[4]Here we will only examine Masri's treatment of transient components as the basic characteristics of sinusoids plus noise models are well described in Section 2.5.

Transient detection

As this system specifically aims to improve the synthesis of the attack segment of a musical note, transients are assumed to occur immediately following note onsets. The process of identifying the start of transients is therefore the process of finding note onset locations. To do this, Masri uses a technique called the *attack envelope* method, in which a separate analysis step is performed prior to sinusoidal analysis that identifies note onsets using an amplitude peak following algorithm.

The input waveform is partitioned into consecutive non-overlapping frames that are 50 samples in duration. The maximum sample value of each block is then defined to be the *onset detection function* (ODF) value for that frame[5]. The peak following algorithm is passed the ODF value for each analysis frame in sequence. It either retains the ODF value or is updated to the previous ODF value multiplied by a constant value, simulating an exponential decay. This enveloping function is defined by Equation 2.46, where $P(l)$ is the peak follower output for frame l, d is the decay factor and $ODF(l)$ is the value of the ODF at frame l.

$$P(l) = \text{MAX}\{ODF(l), P(l-1) \times k\} \tag{2.46}$$

An onset is then detected when Equation 2.47 is satisfied, where T is a static onset detection threshold.

$$\frac{P(l)}{P(l-1)} > T \tag{2.47}$$

The end of a transient region is defined to be the point at which the peak follower envelope drops below a given fraction of its peak value, or else when a predefined

[5]ODFs are usually sub-sampled versions of the original signal that vary depending on the likelihood of a given input frame containing a note onset. They are described in detail in Chapter 4.

maximum duration time has been reached, whichever happens first.

Combining transients with sinusoidal and noise components

Sinusoidal and noise analysis is performed during a second analysis pass. To ensure that the relatively unpredictable evolution of spectral components that can occur during transient regions does not effect the sinusoidal and stochastic analysis processes, the list of transient regions that is compiled during the initial analysis pass is used to "snap" the positions of the analysis windows in the second pass to the region boundaries. For each transient region, the trailing edge of the analysis window is positioned at the first sample index. Analysis for the region then proceeds as normal. When the analysis window reaches the end of the region it is adjusted so that the leading edge is positioned at the last sample index. This usually results in a reduced hop size. In order to account for the loss of the data that would ordinarily have been produced from analysis frames that crossed region boundaries, spectral data from either side of the region boundary is extrapolated (keeping the values fixed) up to the boundary index.

During synthesis, parameters are not interpolated between transient frames and non-transient frames. The frames at either side of region boundaries are extrapolated by a further 256 samples and the two areas are simply cross-faded together. This cross-fade length was chosen as it is deemed short enough to reproduce the "suddenness" of the transient region without introducing audible artifacts.

2.6.2 Transient Modelling Synthesis

Verma and Meng devised a system called Transient Modelling Synthesis (TMS) that extends SMS with a model for transients [121]. It aims to provide an explicit parametric transient representation that is flexible enough to allow for a wide range of modifications and that fits well into the existing SMS framework.

Transient detection

Sinusoidal modelling is based on the fact that a slowly varying signal which is periodic in the time domain is impulsive in the frequency domain. As transient signals are impulsive in the time domain, they must be periodic in the frequency domain, and it should therefore be possible to model them using techniques that are similar to sinusoidal modelling. However, the transient signals must first be mapped to an appropriate frequency domain. TMS uses the discrete cosine transform (DCT) [1] to provide this mapping, which can be defined by Equation 2.48. n is the sample index ranging from 0 to $N-1$ and $\beta(k)$ is defined by Equation 2.49.

$$C(k) = \beta(k) \sum_{n=0}^{N-1} s(n) cos \left[\frac{(2n+1)k\pi}{2N} \right] \tag{2.48}$$

$$\beta(k) = \begin{cases} \sqrt{\frac{1}{N}} & k = 1 \\ \sqrt{\frac{2}{N}} & \text{otherwise} \end{cases} \tag{2.49}$$

In general DCT frequencies can be mapped to the locations of time-domain impulses in the original frame; impulsive signals at the beginning of frames produce low-frequency cosines and impulses towards the end of frames produce high-frequency components.

The DCT is applied to contiguous input frames, where the frame length is chosen to ensure that a transient appears to be short in duration with a given frame. A frame size corresponding to about 1 second of audio is deemed to be sufficient in this regard. The SMS sinusoidal modelling algorithm is then applied to each DCT frame, with 30-60 SMS frames being required for each DCT frame to produce well-modelled transients. Analysis results in amplitude, frequency and phase parameters for each transient, with the frequency corresponding to the transient location within a given DCT frame.

Combining transients with sinusoidal and noise components

TMS analysis begins by using the SMS algorithm to create a model of the sinusoidal components in the input waveform, which is subtracted from the original signal resulting in a first residual signal. This residual is then processed by the transient model. The analysis parameters are used to synthesise transient signals which are subtracted from the first residual to leave a second residual component, which is subjected to SMS stochastic analysis (as described in Section 2.5.1). This process is summarised in Figure 2.12

To create the output sound, the three components are synthesised individually using the (potentially modified) analysis parameters and the results are then summed. Synthesis of the deterministic and stochastic components is performed according to the SMS algorithm. Transients can be synthesised by inverting the transient analysis process, with sinusoidal synthesis of transient parameters producing DCT frames which can be transformed back into the time domain via the inverse DCT.

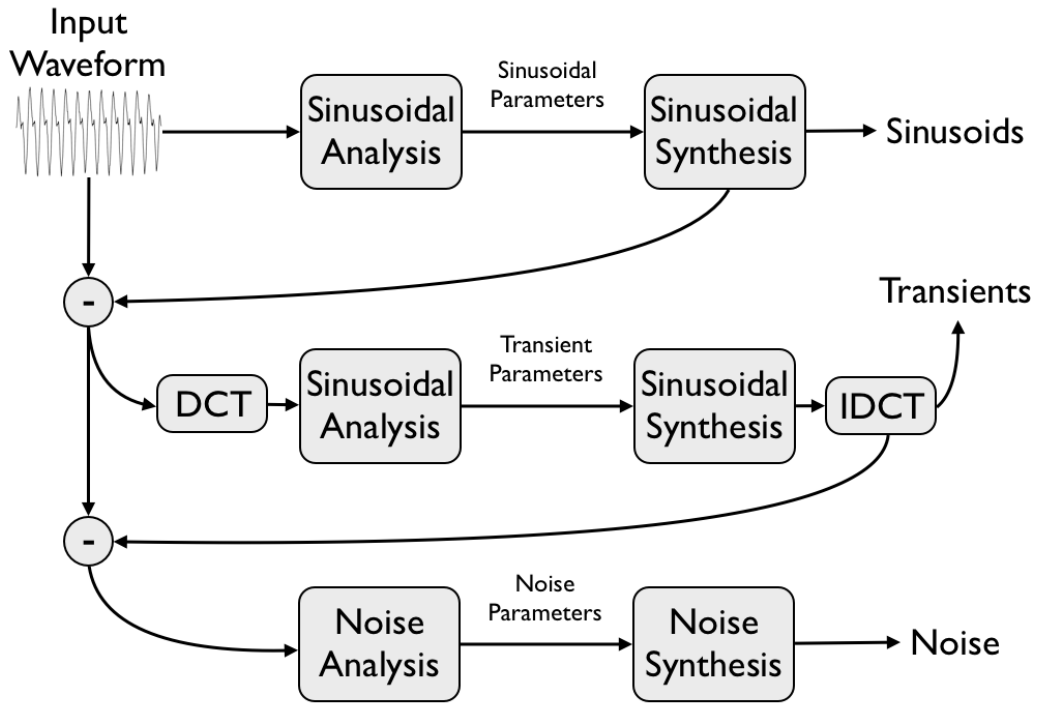

Figure 2.12: The TMS analysis process.

2.7 Software tools for spectral modelling and manipulation

Sections 2.1 - 2.6 described some of the significant developments that have influenced the current state-of-the-art in spectral modelling of musical instrument sounds. To examine the ways in which these techniques can be used by musicians, sound designers and composers, this section provides a brief overview of some of the currently available spectral modelling software packages, from specialised spectral processing tools to powerful general purpose systems for sound analysis/synthesis and composition.

2.7.1 Specialised systems

A wide variety of specialised tools for spectral processing of musical sounds can be found in the literature, generally focusing on analysis, transformation and synthesis using a particular sinusoidal modelling algorithm. Here we survey of some currently available software packages, their implementation details and the sound transformations that they provide.

SNDAN

SNDAN [8] is a free suite of C programs distributed as source code[6], allowing for the analysis, modification and synthesis of (primarily monophonic) musical sounds. Tones can be analysed using either phase vocoder analysis or by using an implementation of MQ sinusoidal modelling. Analysis data is then synthesised using additive synthesis. Programs for viewing plots of analysis data are also included in SNDAN.

A number of transformations can be applied to the analysis data, including several different sinusoidal amplitude and frequency modifications, smoothing sinusoidal amplitudes and frequencies over time, smoothing sinusoidal amplitudes versus frequency and time stretching the input waveform. Analysis data can also be saved to a SNDAN file format and then reloaded for synthesis, so it is possible for other applications to apply modifications.

ATS

ATS [94] is an open source library for spectral analysis, transformation and synthesis of sound. It is written in Common Lisp and is designed to work in conjunction

[6]Registration via email is required.

with Common Lisp Music [106]. The ATS analysis process is similar to the SMS algorithm. Spectral components are identified by peak detection and partial tracking algorithms and then used to synthesise a deterministic signal. This harmonic component then subtracted from the original sound and the residual signal is used to create a stochastic component. However, the ATS partial tracking process also allows peaks to be ignored if they are considered to be inaudible due to psychoacoustic masking effects within critical bands. Analysis data is stored in lisp abstractions called *sounds*, and can be synthesised using a variety of techniques including additive synthesis, subtractive synthesis and granular synthesis. Available transformations include scaling partial amplitudes and frequencies by constant values or dynamic envelopes, transposition (with or without maintaining formant shapes) and time stretching. A graphical user interface (GUI) is also provided that enables the real-time control of synthesis parameters.

SPEAR

SPEAR [59] is a cross-platform graphical spectral analysis, editing and synthesis tool. Sinusoidal analysis is based on an extension of the MQ method. Like SMS, peak frequency estimates are improved by using parabolic interpolation. The partial tracking algorithm selects peaks by using linear prediction to estimate the future trajectory of the partial, and peaks are then selected based on how closely their parameters match the estimates [62]. Synthesis of analysis data is performed using either the inverse FFT method or by using additive synthesis of banks of sinusoidal oscillators. As one of the design goals of SPEAR was to enable integration with other sinusoidal modelling implementations, a wide variety of analysis data file types can be imported and exported. SDIF [125] and plain text files can be both

imported and exported, and SPEAR can additionally import both SNDAN and ATS analysis files. Using the graphical editor, sinusoidal components can be cut, copied and pasted, as well as being shifted in frequency and stretched or compressed in time.

Libsms

Libsms is an open source library that provides an implementation of SMS, derived from Serra's original SMS code [32]. It is written in C, uses SWIG [9] to provide bindings for Python, and is also available as a set of external objects for Pure Data. Available SMS transformations include pitch-shifting (with and without the preservation of the original spectral envelope), time-stretching and independent control of the volumes of deterministic and stochastic components. Analysis data can also be imported from and exported to custom SMS analysis files.

Loris

Loris is an open source C++ implementation of the bandwidth-enhanced sinusoidal modelling system. Python bindings are provided using SWIG, and Loris can also be built as a Csound opcode (plugin). Time-scaling and pitch-shifting modifications can be performed on the analysis data, but sound morphing is of particular interest to the developers so a range of functions are provided for performing morphs between two sound sources. Loris also supports both importing and exporting analysis data via SDIF files.

AudioSculpt

AudioSculpt [14] is a commercial software package for spectral analysis and processing of sound files that is developed by IRCAM. It has a GUI that displays multiple representations of a source sound (a waveform view, a spectrum view and a sonagram), which are used to apply audio transformations to specific time or frequency regions. AudioSculpt relies on two different signal processing kernels in order to manipulate sounds. The first is an extended version of the phase vocoder called SuperVP [27]. It can be used to perform several different analyses including the computation of the standard and reassigned spectrograms, estimation of the spectral envelope by linear predictive coding and the true envelope, transient detection and fundamental frequency estimation. The second sound processing kernel, called Pm2, uses a proprietary sinusoidal modelling implementation. The sinusoidal partials that are estimated by the model can be exported to SDIF files.

AudioSculpt allows multiple transformations to be performed on sound files, including manipulating the gain of certain frequency regions that are selected using the GUI, transposing sounds (with or without time-correction), time-stretching, a spectral "freeze" effect, and a non-linear dynamic range stretching effect that is known as "clipping". A sequencer is also provided so that transformations can be applied at specific time points in a sound file. Sound files can be processed and played back in real-time, allowing the results to be heard before saving them to a new sound file. However, AudioSculpt does not support the real-time processing of live audio streams.

It is possible to use the SuperVP sound processing kernel to manipulate audio streams in real-time by using the set of SuperVP external objects for Max/MSP

[52]. The `supervp.trans~` object can be used to perform transposition, spectral envelope manipulation and decomposition of a signal into a combination of sinusoids, noise and transient components. Spectral envelopes and transients can also be preserved after modification. The `supervp.sourcefilter~` object enables the spectral envelope of one signal to be "imprinted" onto another signal. An alternative cross-synthesis object called `supervp.cross~` can be used to mix the amplitudes and phases of two signals in the frequency domain.

2.7.2 General purpose systems

Spectral processing tools can also be found in some general purpose languages and frameworks for sound design and composition. Describing the full feature sets of these systems is beyond the scope of this research, but this section provides a brief description of four important general purpose tools and their respective spectral processing implementations.

The SndObj Library

The SndObj (Sound Object) Library [66] is an open source object-orientated library for audio signal processing. It is written in C++, with SWIG used to create Python bindings, and runs on Linux, Mac OS X and Windows platforms. The SndObj library consists of a set of classes for signal processing and control, featuring modules for sound input and output, MIDI input and output, delays, filters, envelopes and several different oscillators. It also includes spectral analysis and synthesis modules, with implementations of phase vocoder analysis/synthesis and a custom sinusoidal modelling system. Peak detection works in a similar way to the

bandwidth-enhanced model, using reassignment to improve peak frequency estimates. Partial tracking follows a similar approach to the MQ method, and an output waveform can then be created using additive synthesis of sinusoidal oscillators.

Csound

Csound [119] is a comprehensive system and programming language for audio signal processing and sound design based on the MUSIC-N model. It is open source software and written primarily in C, but due to a flexible API it can be used from many different programming languages and environments. Csound provides extensive support for a number of different spectral processing methods. The *fsig* framework[7] [67, 69] allows users to work with a special type of streaming signal that consists of spectral data. Spectral signals can be created by reading data files or by analysing audio streams using either phase vocoder analysis or by using a custom sinusoidal modelling implementation. Opcodes exist enabling a wide variety of transformations to be applied to the spectral signals such as amplitude-based transformations, frequency-based transformations, cross-synthesis effects and spectral blurring [124]. Audio signals can then be generated from the spectral data using either an inverse DFT overlap-add technique or additive synthesis. Csound can also read, manipulate, and synthesise analysis data created by both ATS and Loris.

SuperCollider

SuperCollider [81] is an open source software environment and programming language for sound design and algorithmic composition. The current version (SuperCollider

[7]The fsig type was introduced by Richard Dobson in Csound 4.13 and was further extended by Victor Lazzarini in Csound 5.

3) consists of two components: the server which is responsible for sound generation, and the language component which communicates with the server using open sound control (OSC). Similarly to Csound, SuperCollider includes a number of different modules for sound analysis and transformation based on the phase vocoder, as well as modules for importing and synthesising ATS analysis data.

Pure Data

Pure Data [97] is an open source visual programming environment for audio and video processing. It is similar in many ways to the commercial program Max/MSP, allowing users to manipulate audio, video and control information by "patching" different graphical objects together. The system can be extended by creating new objects or "externals" in C or C++, enabling spectral processing systems to be used in a Pure Data "patch". One example of this is the SMS Pure Data external, created using Libsms. However, as Pure Data includes objects for performing forward and inverse Fourier transforms, spectral processing tools such as the phase vocoder can also be created directly in Pure Data patches.

2.8 Conclusions

This chapter introduced Fourier analysis and provided an overview of the state-of-the-art in spectral modelling of musical instrument sounds. The STFT and phase vocoder analysis were described, followed by the development of the sinusoidal model and then extensions to this basic sinusoidal model to include explicit representations of noise and transient components. The chapter concluded with a survey of existing software tools for spectral modelling and sound manipulation.

The sinusoidal models and software implementations that were discussed in this chapter are generally successful, but there are a few areas that can be improved upon. In order for researchers to be able to easily compare the analysis and synthesis algorithms, it would be advantageous to have a consistent interface for interacting with the systems. Ideally, this sinusoidal modelling system could also be used in conjunction with a suite of tools for performing standard signal processing techniques and data visualisation.

An important goal of this research is to enable composers, musicians and researchers to perform flexible and intuitive transformations on live audio streams. We therefore require that the spectral models can analyse and transform audio in real-time and with low latency. The sinusoids plus noise plus transients models that are described in Section 2.6 succeed in reducing the audible synthesis artifacts that can occur as a result of the relatively poor treatment of transient signal components in earlier spectral models. However, neither system can operate in a real-time (streaming) mode. Masri's method requires an initial analysis scan of the full audio signal, while the method proposed by Verma and Meng uses DCT frames with a recommended duration of 1 second, which we consider to be too high a latency value to be useful in a live musical performance context.

Finally, we would also like to provide a tool for performing real-time sound manipulation based on the sinusoids plus noise plus transients model. Ideally this tool could be used as both a standalone application and as part of existing environments for sound design. These issues are discussed in detail in Chapters 3, 4, 5 and 6.

Chapter 3

Simpl: A software library for sinusoidal modelling and manipulation of musical sounds

A number of different approaches to creating sinusoidal models of musical sounds were introduced in Chapter 2, along with a survey of existing software tools that implement these techniques. Each different spectral representation offers composers a unique set of possibilities for sound design. For example, the MQ method creates a purely sinusoidal representation of a sound, so any manipulation of analysis data will also change any non-harmonic components in the signal. Although this is often undesirable when attempting to mimic existing acoustic instruments, this property of the model can also be used in a creative way to intentionally make waveforms that sound synthetic. It is also possible to creatively use the models without a prior analysis step, synthetically generating all of the appropriate parameters.

In order to work with sinusoidal models in a flexible way, it is desirable to

have an expressive, interactive system that facilitates rapid-prototyping of ideas, combined with a set of tools for performing standard signal processing techniques and data visualisation. Being able to interact with analysis data and parameters in a consistent manner can also significantly speed up the exploratory process. Each model shares common ideas and abstractions (such as the concepts of spectral peak detection and partial tracking), but due to their differing implementations, it can be difficult to exchange, compare and contrast analysis data. Some of the software tools support importing and exporting SDIF files which does make data exchange easier in those cases. However even these tools do not generally allow the peak detection and partial tracking analysis stages to be separated, as they only save the final partials to SDIF files.

We developed the **Si**nusoidal **M**odelling **P**ython **L**ibrary (or Simpl) to address these issues. It allows sinusoidal modelling implementations to be used in conjunction with a powerful general purpose programming language, which has a comprehensive set of libraries for audio signal processing. Simpl also facilitates the exchange of spectral analysis data between different underlying implementations, and can analyse and synthesise audio in either non-real-time or in a real-time streaming mode (depending on the capabilities of the model). The choice of Python as a tool for prototyping and rapid development of audio applications for composers and researchers is discussed in Section 3.1. This is followed by an overview of Simpl in Section 3.2, a discussion of Simpl's implementation in Section 3.3 and finally some examples in Section 3.4.

3.1 Python for audio signal processing

There are many problems that are common to a wide variety of applications in the field of audio signal processing, and so it often makes sense to rely on existing code libraries and frameworks to perform these tasks. Examples include procedures such as loading sound files or communicating between audio processes and sound cards, as well as digital signal processing (DSP) tasks such as filtering and Fourier analysis.

Due to their ubiquity, fast execution speeds and the ability to completely control memory usage, C and C++ are popular general purpose programming languages for audio signal processing. However, it can often take a lot more time to develop applications or prototypes in C/C++ than in a more lightweight scripting language. This is one of the reasons for the popularity of tools such as MATLAB [79], which allows the developer to easily manipulate matrices of numerical data, and includes implementations of many standard signal processing techniques. A major downside to MATLAB is that it is not free and not open source, which is a considerable problem for people who want to share code and collaborate. GNU Octave [33] is an open source alternative to MATLAB. It is an interpreted language with a syntax that is very similar to MATLAB and so it is possible to write scripts that will run on both systems. However, with both MATLAB and Octave this increase in short-term productivity comes at a cost. For anything other than very basic tasks, tools such as integrated development environments (IDEs), debuggers and profilers are certainly a useful resource if not a requirement. All of these tools exist in some form for MATLAB/Octave, but users must invest a considerable amount of time in learning to use a programming language and a set of development tools that have a relatively

limited application domain when compared with general purpose programming languages. It is also generally more difficult to integrate MATLAB/Octave programs with compositional tools such as Csound or Pure Data, or with other technologies such as web frameworks, cloud computing platforms and mobile applications, all of which are becoming increasingly important in the music industry.

For developing and prototyping audio signal processing applications, it would therefore be advantageous to combine the power and flexibility of a widely adopted, open source, general purpose programming language with the quick development process that is possible when using interpreted languages that include comprehensive signal processing toolkits. All of these characteristics can be found in the combination of the Python programming language [104] with the NumPy [88], SciPy [56, 89] and Matplotlib [49] libraries for scientific computing and data visualisation. Some notable features of the Python language include:

- It is a mature language and allows for programming in several different paradigms including imperative, object-oriented and functional styles.
- The clean syntax puts an emphasis on producing well structured and readable code. Python source code has often been compared to executable pseudocode.
- Python provides an interactive interpreter which allows for rapid code development, prototyping and live experimentation.
- The ability to extend Python with modules written in C/C++ means that functionality can be quickly prototyped and then optimised later.
- Python can be embedded into existing applications.
- Python bindings exist for cross-platform GUI toolkits such as Qt [86].

- Python has a large number of high-quality libraries, allowing sophisticated programs to be constructed quickly.

Numpy and SciPy extend Python by adding a comprehensive set of functions and modules for scientific computing and signal processing. NumPy [88] adds a multidimensional array object to Python, along with functions that perform efficient calculations based on this array data. SciPy builds on top of NumPy and provides modules that are dedicated to common issues in scientific computing. It can therefore be compared to MATLAB toolboxes. The SciPy modules are written in a mixture of pure Python, C and Fortran, and are designed to operate efficiently on NumPy arrays. Notable SciPy modules include:

File input/output (scipy.io): Provides functions for reading and writing files in many different data formats, including *.wav*, *.csv* and matlab data files (*.mat*).

Fourier transforms (scipy.fftpack): Contains implementations of 1-D and 2-D fast Fourier transforms, as well as Hilbert and inverse Hilbert transforms.

Signal processing (scipy.signal): Provides implementations of many useful signal processing techniques, such as waveform generation, FIR and IIR filtering and multi-dimensional convolution.

Optimisation (scipy.optimize): Contains a collection of optimisation algorithms and system solvers.

Interpolation (scipy.interpolate): Consists of linear interpolation functions and cubic splines in several dimensions.

Linear algebra (scipy.linalg): Contains functions for solving a system of linear

equations and other matrix operations such as matrix exponential and matrix square root.

Matplotlib is a library of plotting functions that enables data contained in NumPy arrays to be visualised quickly and used to produce publication-ready figures in a variety of formats. It can be used interactively from the Python command prompt, thus providing similar functionality to MATLAB or GNU Plot [123], or in conjunction with GUI toolkits (Qt for example).

3.1.1 A SciPy example

Listing 3.1 shows how SciPy can be used to perform a FFT on a windowed frame of audio samples and plot the resulting magnitude spectrum. The sample is loaded using the *wav.read* function on line 8. This function returns a tuple containing the sampling rate of the audio file as the first value and the audio samples (in a NumPy array) as the second value. The samples are stored in a variable called audio. In line 13, a 256 sample frame is selected from the centre of the audio signal and is then windowed using a Hanning window (created using the SciPy *sp.signal.hann* function). The FFT is computed on line 16, with the complex coefficients converted into polar form and the magnitude values stored in the variable *mags*. The magnitude values are converted from a linear to a decibel scale in line 19, then normalised to have a maximum value of 0 dB in line 22. In lines 25-33 the magnitude values are plotted and displayed. The resulting image is shown in Figure 3.1.

```
1 import scipy
2 import scipy.signal
3 import scipy.io.wavfile as wav
4 import numpy
```

```
5  import matplotlib.pyplot as plt
6
7  # read audio samples
8  audio = wav.read('clarinet.wav')[1]
9
10 # select a frame and apply a Hanning window
11 N = 256
12 sample_centre = len(audio) / 2
13 audio = audio[sample_centre:sample_centre + N] * scipy.signal.hann(N)
14
15 # fft
16 mags = numpy.abs(scipy.fftpack.rfft(audio))
17
18 # convert to dB
19 mags = 20 * scipy.log10(mags)
20
21 # normalise to 0 dB max
22 mags -= max(mags)
23
24 # plot
25 plt.plot(mags)
26
27 # label the axes
28 plt.ylabel('Magnitude (dB)')
29 plt.xlabel('Frequency Bin')
30
31 # set the title
32 plt.title('Magnitude spectrum of a clarinet tone')
33 plt.show()
```

Listing 3.1: Using SciPy to calculate and plot a magnitude spectrum.

3.2 An overview of Simpl

Simpl is an object-oriented library for sinusoidal modelling, that aims to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API. It is primarily intended as a tool for composers and other researchers, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms. The currently supported sinusoidal modelling implementations are the MQ method, SMS (using libsms), the bandwidth-enhanced sinusoidal model (using Loris) and The SndObj Library. Simpl is free software, released under the terms of the GNU General Public License (GPL).

The core audio analysis and synthesis algorithms are written in C++, enabling

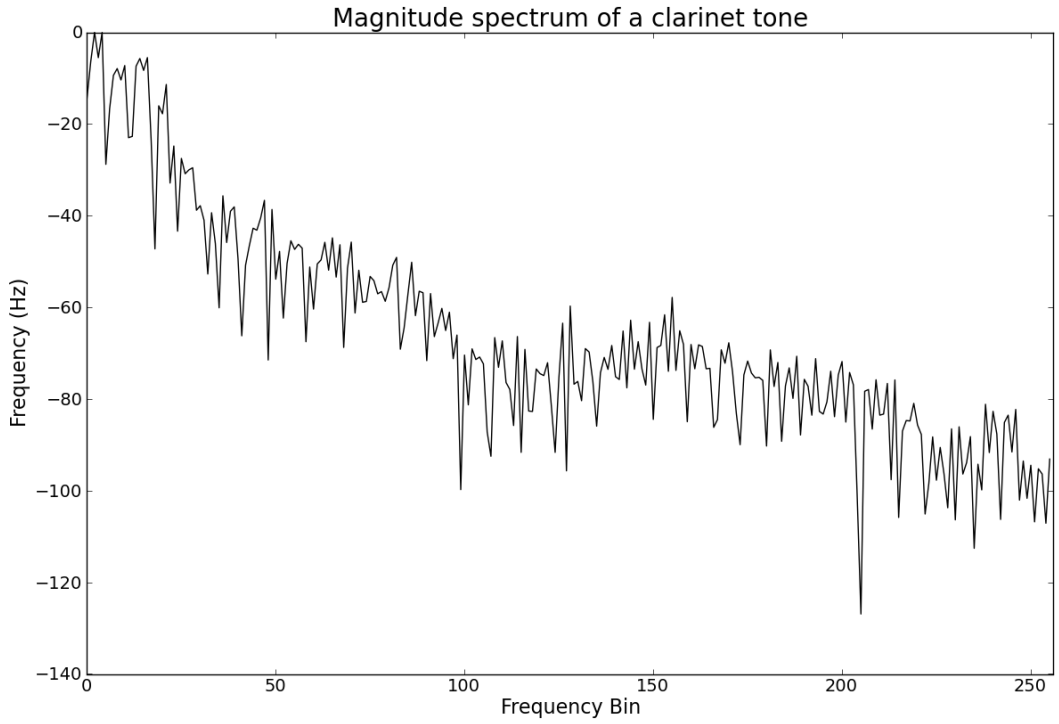

Figure 3.1: Magnitude spectrum of a 256 sample frame from a clarinet recording, produced by the code in Listing 3.1.

Simpl to be used in conjunction with other C/C++ audio software packages such as Csound. Simpl also comes with a set of Cython [10] wrapper classes which can be used in order to build it as a Python extension module. Users can therefore take advantage of the wide range of signal processing functionality that is provided by NumPy, SciPy and Matplotlib.

The sinusoidal modelling process in Simpl is broken down into three distinct steps: peak detection, partial tracking and sound synthesis. C++ classes and corresponding Python classes exist for each step, which all of the analysis and synthesis implementations derive from. For any given step, every analysis/synthesis object returns data in the same format, irrespective of its underlying implementation. This

allows analysis/synthesis networks to be created in which the algorithm that is used for a particular step can be changed without effecting the rest of the network. This process is summarised in Figure 3.2. Sections 3.2.1 - 3.2.7 provide an overview of

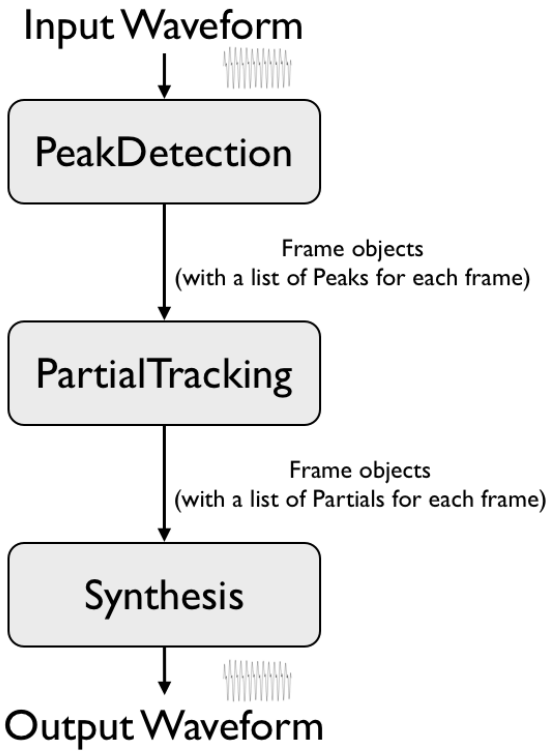

Figure 3.2: The simpl analysis and synthesis process.

the modules and classes that are available in Simpl.

3.2.1 Peaks and frames

Two classes that are used extensively in Simpl are `Peak` and `Frame`. `Peak` classes represent spectral peaks. It is a small class that primarily exists to store 4 floating-point values: the `amplitude` (or magnitude), `frequency`, `phase` and `bandwidth` of each peak. `Frame` objects contain all of the information that relates to a given frame

of audio. This includes the size of a frame (in samples), the input audio samples, any synthesised audio samples, detected spectral peaks and detected partials. For convenience, an alias for a vector of pointers to `Peak` objects called `Peaks` is defined, as is an alias for a vector of pointers to `Frame` objects called `Frames`. The key components of these objects are summarised in Tables 3.1 and 3.2[1].

Peak	
Parameter	Description
`amplitude: float`	The amplitude of the peak.
`frequency: float`	The frequency of the peak.
`phase: float`	The phase of the peak.
`bandwidth: float`	The bandwidth assigned to the peak.

Table 3.1: The Simpl Peak class.

All of the audio samples in Simpl are stored in C++ `double` arrays. However when used from Python, the audio samples are accessed using NumPy arrays. This means that SciPy functions can be used for basic tasks such as reading and writing audio files, as well as more complex procedures such as performing additional processing, analysis or visualisation of the data.

3.2.2 Peak detection

`PeakDetection` objects consist of methods that take audio samples as input and return one or more `Frame` objects[2]. These audio samples can either be contained in a single `Frame` object or be pointers to arrays of samples. In the latter case, the

[1]For simplicity the classes and parameters that are described in this section will refer to the Python implementations, the corresponding C++ classes are very similar however.

[2]When used with C++ a `std::vector` of `Frame` objects is returned. With Python a list of `Frame` objects is returned.

Frame	
Parameter	Description
`size: int`	The input frame size (in samples).
`synth_size: int`	The output frame size (in samples).
`max_peaks: int`	The max. no. peaks to detect per frame.
`num_peaks: int`	The no. peaks detected this frame.
`max_partials: int`	The max. no. partials to produce per frame.
`num_partials: int`	The no. partials in this frame.
`peaks: list of Peaks`	The detected spectral peaks.
`partials: list of Peaks`	The computed sinusoidal partials.
`audio: NumPy array`	The input audio waveform.
`synth: NumPy array`	The synthesised harmonic component.
`residual: NumPy array`	The residual component.
`synth_residual: NumPy array`	The synthesised residual component.

Table 3.2: The Simpl Frame class.

audio will be divided into multiple frames, where the exact number of frames and their sizes will depend on the peak detection hop size and frame size parameters. The frame size may change during analysis however, as some sinusoidal modelling algorithms (such as SMS for example) will vary the analysis frame size depending on the estimated fundamental frequency of the input waveform. The `Frame.peaks` variable contains a list of all spectral peaks detected in that frame by the chosen sinusoidal modelling implementation. `Peaks` may also be added to `Frames` programmatically without analysing an audio signal. `PeakDetection` classes are summarised in Table 3.3.

PeakDetection	
Parameter	Description
`sampling_rate: int`	The sampling rate of the input waveform.
`frame_size: int`	The analysis frame size.
`hop_size: int`	The analysis hop size.
`static_frame_size: bool`	Keep analysis frame size fixed during analysis.
`max_peaks: int`	The max. no. peaks to detect per frame.
`min_peak_separation: int`	The min. distance to allow between peaks (in Hz).
`frames: list of Frames`	The resulting analysis frames.
Method	Description
`find_peaks_in_frame(frame: Frame*)`	Find spectral peaks in a given frame.
`find_peaks(audio: NumPy array)`	Find spectral peaks in an audio signal. The signal will be broken up into multiple frames if necessary.

Table 3.3: The Simpl `PeakDetection` class.

3.2.3 Partial tracking

`PartialTracking` classes have methods that take either a single `Frame` or a list of `Frame` objects as input, as described by Table 3.4. Each `Frame` is expected to contain a list of peaks, calculated either during peak detection or programmatically. The spectral peaks in each frame are then used by the sinusoidal modelling implementation to form sinusoidal partials. This results in selected `Peak` values being saved to the `Frame.partials` list for each analysis frame. The order of `Peak` objects in the `Frame.partials` list is therefore of significance (unlike the list of peaks

PartialTracking	
Parameter	Description
`sampling_rate: int`	The sampling rate of the input waveform.
`max_partials: int`	The max. no. partials to detect per frame.
`min_partial_length: int`	The min. length of a partial (in frames). Shorter partials will be removed at the end of analysis. This parameter is ignored when operating in real time.
`frames: Frames`	The resulting analysis frames.
Method	Description
`update_partials(frame: Frame)`	Perform partial tracking for a given frame.
`find_partials(frames: list of Frames)`	Perform partial tracking on a sequence of frames.

Table 3.4: Simpl `PartialTracking` class.

in `Frame.peaks`). Spectral peaks at the same position in the `Frame.partials` lists in consecutive Simpl `Frame` objects are assumed to belong to the same underlying sinusoidal partial, and so their parameters will be interpolated across output frames during synthesis. Inactive partials should therefore always contain spectral peaks with an amplitude of 0.

3.2.4 Synthesis

Simpl `Synthesis` classes are described by Table 3.5. The main methods take either a single `Frame` or a list of `Frame` objects as input and use the given partial data to synthesise a frame of audio samples. `Synthesis` objects generally only calculate

Synthesis	
Parameter	Description
`sampling_rate: int`	The sampling rate of the output waveform.
`frame_size: int`	The synthesis frame size.
`hop_size: int`	The synthesis hop size.
`max_partials: int`	The max. no. partials to synthesise per frame.
`synth_frame(frame: Frames*)`	Synthesise one frame of audio from the harmonic partial data.
`synth(frames: Frames)`	Synthesise several frames of audio from the harmonic partial data.

Table 3.5: Simpl `Synthesis` class.

sinusoidal or harmonic signal components[3], calculating noise or residual components requires the use of Simpl `Residual` objects. For each frame, the synthesised signal is stored in the `Frame.synth` variable. Reconstructing an entire audio signal therefore requires concatenating these output arrays. When called from Python the concatenation is performed automatically by the Cython wrapper module, and so a NumPy array of audio samples is returned.

3.2.5 Residual

`Residual` classes are used to compute residual components or synthesise stochastic signal components. They take either an audio signal or a sequence of Simpl `Frame` objects as input. In the case of SMS, a deterministic component is then synthesised and subtracted from the original signal to obtain the residual signal, which is saved

[3]The one current exception to this is the Simpl implementation of the bandwidth-enhanced sinusoidal model, for which the `Synthesis` class can currently create both sinusoidal and noise components.

in the `Frame.residual` variable. If a synthesised stochastic component is to be created, this will be created and then saved in the `Frame.synth_residual` variable. The `Residual` class is summarised by Table 3.6.

Residual	
Parameter	Description
`sampling_rate: int`	The sampling rate of the output waveform.
`frame_size: int`	The synthesis frame size.
`hop_size: int`	The synthesis hop size.
`residual_frame(frame: Frame)`	Compute the residual signal for a given frame.
`find_residual(frames: list of Frames)`	Compute the residual signal given an original input waveform and a synthesised deterministic component.
`synth_frame(frame: Frame)`	Synthesise a stochastic component for a given frame.
`synth(frames: list of Frames)`	Synthesise a stochastic component for a sequence of frames.
`synth(original: NumPy array)`	Analyse a given waveform, compute the residual signal and synthesise a stochastic component.

Table 3.6: Simpl `Residual` class.

3.2.6 Visualisation

When used from Python, Simpl includes a module with two functions that use Matplotlib to plot analysis data. These functions, `simpl.plot_peaks` and `simpl.plot_partials`, plot data from the peak detection and partial tracking analysis stages respectively. Generating additional plots is trivial using Matplotlib's

MATLAB-like interface however. The plots of peak and partial data in Section 3.4 are both generated using the `simpl.plot` module.

3.2.7 Complete list of Simpl modules, classes and functions

Tables 3.7 and 3.8 list the modules, classes and functions that are currently available in Simpl. Functions that are marked with an asterisk (*) depend on SciPy and/or Matplotlib functionality and so have no corresponding C++ implementations. Classes begin with capital letters while functions start with lowercase letters. The full source code to Simpl can be found on the accompanying CD.

Python Module	**Classes and Functions in Module**
`simpl.base`	`Peak` `Frame`

`simpl.peak_detection`	`PeakDetection`
	`SMSPeakDetection`
	`LorisPeakDetection`
	`SndObjPeakDetection`

`simpl.partial_tracking`	`PartialTracking`
	`SMSPartialTracking`
	`LorisPartialTracking`
	`SndObjPartialTracking`

`simpl.synthesis`	`Synthesis`
	`SMSSynthesis`
	`LorisSynthesis`
	`SndObjSynthesis`

`simpl.residual`	`Synthesis` `SMSSynthesis`

simpl.mq	MQPeakDetection* MQPartialTracking* MQSynthesis* twm*
simpl.audio	read_wav*
simpl.plot	plot_peaks* plot_partials*

Table 3.7: Simpl Python modules.

Class or Function	Description
`Peak`	Contains spectral peak data.
`Frame`	Contains all analysis data for a frame of audio.
`PeakDetection`	Base peak detection class, primarily containing general data types and accessor methods.
`MQPeakDetection`	Performs peak detection using the MQ method.
`SMSPeakDetection`	Performs peak detection using the SMS algorithm.
`LorisPeakDetection`	Performs peak detection using the algorithm from the bandwidth-enhanced sinusoidal model.
`SndObjPeakDetection`	Performs peak detection using the SndObj library.
`PartialTracking`	Base partial tracking class, primarily containing general data types and accessor methods.
`MQPartialTracking`	Performs partial tracking using the MQ method.
`SMSPartialTracking`	Performs partial tracking using SMS (libsms).
`LorisPartialTracking`	Performs partial tracking according to the method described in the bandwidth-enhanced sinusoidal model (loris).
`SndObjPartialTracking`	Performs partial tracking using the SndObj library.
`Synthesis`	Base synthesis class, primarily consists of general data types and accessor methods.

`MQSynthesis`	Performs synthesis using the MQ method.
`SMSSynthesis`	Performs synthesis using SMS (libsms), using either the inverse FFT overlap-add technique or additive synthesis.
`LorisSynthesis`	Performs additive synthesis using loris.
`SndObjSynthesis`	Performs additive synthesis using the SndObj library.
`Residual`	Base residual component class, primarily consists of general data types and accessor methods.
`SMSResidual`	Creates a residual signal or synthesised stochastic component using SMS (libsms).
`twm`	An implementation of the two-way mismatch algorithm [74] for detecting the fundamental frequency of a collection of spectral peaks.
`read_wav*`	Reads an audio file (in the *wav* format), and stores it in a NumPy array, making sure that the type of the array (`float` or `double`) matches the array type that is used by the Simpl C++ code. Sample values are also converted to the range $[-1, 1]$.
`plot_peaks*`	Plots spectral peaks from a sequence of `Frame` objects (using Matplotlib).
`plot_partials*`	Plots sinusoidal partials from a sequence of `Frame` objects (using Matplotlib).

Table 3.8: Simpl classes and functions (available from both C++ and Python).

3.3 Implementation

The creation of open source software tools is an important component in this thesis. In providing these tools, we allow musicians and composers to freely experiment with the concepts that are described here, but also enable other researchers to reproduce, critique and potentially improve on the results of this work. Open source software can also serve as valuable teaching material, bridging the gap between abstract ideas and concrete implementations.

To provide an insight into the implementation of Simpl and to illustrate how new sinusoidal modelling implementations may be added to the library in future, this section examines the Simpl SMS peak detection module in detail. The C++ code will be described first, followed by an explanation of the Cython wrapper code that is used to generate the corresponding Python modules. We will only show the code that is necessary in order to understand the implementation of the Simpl SMS module here.

3.3.1 Simpl SMS peak detection C++ module

The C++ declaration of the Simpl `PeakDetection` class is shown in Listing 3.2.

```
39 class PeakDetection {
40     protected:
41         int _sampling_rate;
42         int _frame_size;
43         bool _static_frame_size;
44         int _hop_size;
45         int _max_peaks;
46         std::string _window_type;
47         int _window_size;
48         sample _min_peak_separation;
49         Frames _frames;
50 
51     public:
52         PeakDetection();
```

```
53          virtual ~PeakDetection();
54          void clear();
55
56          virtual int sampling_rate();
57          virtual void sampling_rate(int new_sampling_rate);
58          virtual int frame_size();
59          virtual void frame_size(int new_frame_size);
60          virtual bool static_frame_size();
61          virtual void static_frame_size(bool new_static_frame_size);
62          virtual int next_frame_size();
63          virtual int hop_size();
64          virtual void hop_size(int new_hop_size);
65          virtual int max_peaks();
66          virtual void max_peaks(int new_max_peaks);
67          virtual std::string window_type();
68          virtual void window_type(std::string new_window_type);
69          virtual int window_size();
70          virtual void window_size(int new_window_size);
71          virtual sample min_peak_separation();
72          virtual void min_peak_separation(sample new_min_peak_separation);
73          int num_frames();
74          Frame* frame(int frame_number);
75          Frames frames();
76          void frames(Frames new_frames);
77
78          // Find and return all spectral peaks in a given frame of audio
79          virtual void find_peaks_in_frame(Frame* frame);
80
81          // Find and return all spectral peaks in a given audio signal.
82          // If the signal contains more than 1 frame worth of audio, it will be
83          // broken up into separate frames, with an array of peaks returned for
84          // each frame
85          virtual Frames find_peaks(int audio_size, sample* audio);
86  };
```

Listing 3.2: `PeakDetection` declaration in *peak_detection.h*.

The `Peak` and `Frame` classes are defined in the file *base.h*, as well as the following three `typedef` definitions that are used throughout the C++ code:

```
1  typedef double sample;
2  typedef std::vector<Peak*> Peaks;
3  typedef std::vector<Frame*> Frames;
```

Listing 3.3: `typedef` statements in *Base.h*.

The `PeakDetection` class declaration begins with a set of member variables and methods (described in Table 3.3). The majority of these methods are basic getters and setters that are used to encapsulate the member variables, and so they will not

be discussed in detail. It is important to note however that most of them are declared with the `virtual` keyword so that they can be overridden in derived classes. Two important methods are declared on lines 79 and 85, `find_peaks_in_frame` and `find_peaks`. These are the main methods that classes that inherit from `PeakDetection` should override in order to provide an alternate peak detection algorithm.

The implementation of the `PeakDetection` class can be found in the file *peak_detection.cpp* (Listing 3.4). The constructor (line 5) sets default values for the member variables, with the destructor (line 16) deleting any frames that were created during analysis.

```
1  #include "peak_detection.h"
2
3  using namespace simpl;
4
5  PeakDetection::PeakDetection() {
6      _sampling_rate = 44100;
7      _frame_size = 2048;
8      _static_frame_size = true;
9      _hop_size = 512;
10     _max_peaks = 100;
11     _window_type = "hanning";
12     _window_size = 2048;
13     _min_peak_separation = 1.0;  // in Hz
14 }
15
16 PeakDetection::~PeakDetection() {
17     clear();
18 }
19
20 void PeakDetection::clear() {
21     for(int i = 0; i < _frames.size(); i++) {
22         if(_frames[i]) {
23             delete _frames[i];
24             _frames[i] = NULL;
25         }
26     }
27
28     _frames.clear();
29 }
30
31 // Getter and setter definitions omitted
32
33 void PeakDetection::find_peaks_in_frame(Frame* frame) {
34 }
35
36 Frames PeakDetection::find_peaks(int audio_size, sample* audio) {
37     clear();
38     unsigned int pos = 0;
```

```
39      bool alloc_memory_in_frame = true;
40
41      while(pos <= audio_size - _hop_size) {
42          if(!_static_frame_size) {
43              _frame_size = next_frame_size();
44          }
45
46          Frame* f = new Frame(_frame_size, alloc_memory_in_frame);
47          f->max_peaks(_max_peaks);
48
49          if((int)pos <= (audio_size - _frame_size)) {
50              f->audio(&(audio[pos]), _frame_size);
51          }
52          else {
53              f->audio(&(audio[pos]), audio_size - pos);
54          }
55
56          find_peaks_in_frame(f);
57          _frames.push_back(f);
58          pos += _hop_size;
59      }
60
61      return _frames;
62  }
```

Listing 3.4: `PeakDetection` implementation in *peak_detection.cpp*.

The `find_peaks` method (line 36) steps through the input audio file in `_hop_size` increments. At each time index, the next frame size is calculated if the variable `_static_frame_size` is set to `false`, then a new Simpl `Frame` object is created (line 46). The `Frame` `audio` variable is set to point to the input array at the current time index, and the value of the `_max_peaks` variable is passed to the `Frame`. The `find_peaks_in_frame` method is then called, before the `Frame` is appended to the current list of frames and the current time index is advanced.

`SMSPeakDetection` inherits from `PeakDetection`, as shown in Listing 3.5. It includes two additional member variables called `_analysis_params` and `_peaks`, that are of types `SMSAnalysisParams` and `SMSSpectralPeaks` respectively. These are two C `structs` that are defined in *sms.h*. It overrides the `next_frame_size`, `hop_size`, `find_peaks_in_frame` and `find_peaks` methods, while providing an additional method called `realtime`. The latter is required in order to set an additional libsms parameter that is needed when performing analysis in real-time.

```
110 // ------------------------------------------------------------------------
111 // SMSPeakDetection
112 // ------------------------------------------------------------------------
113 class SMSPeakDetection : public PeakDetection {
114     private:
115         SMSAnalysisParams _analysis_params;
116         SMSSpectralPeaks _peaks;
117 
118     public:
119         SMSPeakDetection();
120         ~SMSPeakDetection();
121         int next_frame_size();
122         using PeakDetection::frame_size;
123         void frame_size(int new_frame_size);
124         using PeakDetection::hop_size;
125         void hop_size(int new_hop_size);
126         using PeakDetection::max_peaks;
127         void max_peaks(int new_max_peaks);
128         int realtime();
129         void realtime(int new_realtime);
130         void find_peaks_in_frame(Frame* frame);
131         Frames find_peaks(int audio_size, sample* audio);
132 };
```

Listing 3.5: `SMSPeakDetection` declaration in *peak_detection.h*.

The `SMSPeakDetection` methods are defined in *peak_detection.cpp*. The constructor (shown in Listing 3.6) sets default parameters in the `SMSAnalysisParams` structure that is used by libsms, and then calls `sms_initAnalysis` on line 237, which allocates memory that is used in the structure. It also allocates memory for the SMS spectral peak arrays by calling `sms_initSpectralPeaks` on line 240. The destructor deallocates all of this memory.

```
219 SMSPeakDetection::SMSPeakDetection() {
220     sms_init();
221 
222     sms_initAnalParams(&_analysis_params);
223     _analysis_params.iSamplingRate = _sampling_rate;
224     _analysis_params.iFrameRate = _sampling_rate / _hop_size;
225     _analysis_params.iWindowType = SMS_WIN_HAMMING;
226     _analysis_params.fHighestFreq = 20000;
227     _analysis_params.iMaxDelayFrames = 4;
228     _analysis_params.analDelay = 0;
229     _analysis_params.minGoodFrames = 1;
230     _analysis_params.iCleanTracks = 0;
231     _analysis_params.iFormat = SMS_FORMAT_HP;
232     _analysis_params.nTracks = _max_peaks;
233     _analysis_params.maxPeaks = _max_peaks;
234     _analysis_params.nGuides = _max_peaks;
```

```
235     _analysis_params.preEmphasis = 0;
236     _analysis_params.realtime = 0;
237     sms_initAnalysis(&_analysis_params);
238     _analysis_params.iSizeSound = _frame_size;
239 
240     sms_initSpectralPeaks(&_peaks, _max_peaks);
241 
242     // By default, SMS will change the size of the frames being read
243     // depending on the detected fundamental frequency (if any) of the
244     // input sound. To prevent this behaviour (useful when comparing
245     // different analysis algorithms), set the
246     // _static_frame_size variable to True
247     _static_frame_size = false;
248 }
249 
250 SMSPeakDetection::~SMSPeakDetection() {
251     sms_freeAnalysis(&_analysis_params);
252     sms_freeSpectralPeaks(&_peaks);
253     sms_free();
254 }
```

Listing 3.6: `SMSPeakDetection` constructor and destructor from *peak_detection.cpp*.

The overridden `PeakDetection` methods (Listing 3.7) make sure that peak detection parameter changes are also copied to the `SMSAnalysisParams` structure, and that if the `max_peaks` method is called then the SMS spectral peaks structure is resized accordingly. The new `realtime` method simply sets a variable in the analysis parameter structure.

```
256 int SMSPeakDetection::next_frame_size() {
257     return _analysis_params.sizeNextRead;
258 }
259 
260 void SMSPeakDetection::frame_size(int new_frame_size) {
261     _frame_size = new_frame_size;
262     _analysis_params.iSizeSound = _hop_size;
263 }
264 
265 void SMSPeakDetection::hop_size(int new_hop_size) {
266     _hop_size = new_hop_size;
267     sms_freeAnalysis(&_analysis_params);
268     _analysis_params.iFrameRate = _sampling_rate / _hop_size;
269     sms_initAnalysis(&_analysis_params);
270 }
271 
272 void SMSPeakDetection::max_peaks(int new_max_peaks) {
273     _max_peaks = new_max_peaks;
274     if(_max_peaks > SMS_MAX_NPEAKS) {
275         _max_peaks = SMS_MAX_NPEAKS;
276     }
277 
278     sms_freeAnalysis(&_analysis_params);
```

```
279     sms_freeSpectralPeaks(&_peaks);
280
281     _analysis_params.nTracks = _max_peaks;
282     _analysis_params.maxPeaks = _max_peaks;
283     _analysis_params.nGuides = _max_peaks;
284
285     sms_initAnalysis(&_analysis_params);
286     sms_initSpectralPeaks(&_peaks, _max_peaks);
287 }
288
289 int SMSPeakDetection::realtime() {
290     return _analysis_params.realtime;
291 }
292
293 void SMSPeakDetection::realtime(int new_realtime) {
294     _analysis_params.realtime = new_realtime;
295 }
```

Listing 3.7: Methods overridden by `SMSPeakDetection` and `realtime` method from *peak_detection.cpp*.

The most important methods in the `SMSPeakDetection` class are `find_peaks_in_frame` and `find_peaks`, shown in Listing 3.8. As in the parent class, the `find_peaks_in_frame` method begins by creating a variable of type `Peaks`. A call is then made to the `sms_findPeaks` function in libsms, which takes the audio samples from the Simpl `Frame` object and locates spectral peaks using the SMS algorithm. Libsms saves the peak amplitudes, frequencies and phases into arrays in the `SMSSpectralPeaks` member variable `_peaks`. For each peak detected by libsms, a new Simpl `Peak` object is added to the current `Frame` (line 303), with amplitude, frequency and phase values taken from the `SMSSpectralPeaks` structure. The `find_peaks` function in `SMSPeakDetection` is very similar to the method that it overrides. The differences are that the `_analysis_params.iSizeSound` variable is set to the length of the input file (which libsms uses during non-real-time analysis), and that the frame size may vary.

```
297 // Find and return all spectral peaks in a given frame of audio
298 void SMSPeakDetection::find_peaks_in_frame(Frame* frame) {
299     int num_peaks = sms_findPeaks(frame->size(), frame->audio(),
300                                   &_analysis_params, &_peaks);
301
302     for(int i = 0; i < num_peaks; i++) {
303         frame->add_peak(_peaks.pSpectralPeaks[i].fMag,
304                         _peaks.pSpectralPeaks[i].fFreq,
305                         _peaks.pSpectralPeaks[i].fPhase,
306                         0.0);
307     }
308 }
309
310 // Find and return all spectral peaks in a given audio signal.
311 // If the signal contains more than 1 frame worth of audio,
312 // it will be broken up into separate frames, with a list of
313 // peaks returned for each frame.
314 Frames SMSPeakDetection::find_peaks(int audio_size, sample* audio) {
315     clear();
316     unsigned int pos = 0;
317     bool alloc_memory_in_frame = true;
318
319     _analysis_params.iSizeSound = audio_size;
320
321     while(pos <= audio_size - _hop_size) {
322         if(!_static_frame_size) {
323             _frame_size = next_frame_size();
324         }
325
326         Frame* f = new Frame(_frame_size, alloc_memory_in_frame);
327         f->max_peaks(_max_peaks);
328
329         if((int)pos <= (audio_size - _frame_size)) {
330             f->audio(&(audio[pos]), _frame_size);
331         }
332         else {
333             f->audio(&(audio[pos]), audio_size - pos);
334         }
335
336         find_peaks_in_frame(f);
337         _frames.push_back(f);
338
339         if(!_static_frame_size) {
340             pos += _frame_size;
341         }
342         else {
343             pos += _hop_size;
344         }
345     }
346
347     return _frames;
348 }
```

Listing 3.8: `SMSPeakDetection find_peaks_in_frame and find_peaks` methods from *peak_detection.cpp*.

3.3.2 Simpl SMS peak detection Python module

Simpl uses Cython [10] to create Python extension modules from the C++ code. Cython is a superset of the Python language that adds the ability to call C functions and declare C types on variables and class attributes. Unlike Python, Cython code must therefore be compiled. Simpl uses the standard Python `distutils` module to compile the Cython code, so when Simpl is installed the Cython modules are automatically compiled and installed together with the pure Python code. Cython is used instead of SWIG [9] as it allows for more control over the resulting Python objects and so Python modules can be created that follow standard Python coding style guides. SWIG modules often produce an API that more closely resembles C/C++ and so an additional Python wrapper would be required around the SWIG module in order to produce the same Python API. It is also generally faster to write Python extension modules in Cython than to write the wrapper by hand using the Python C API.

The Cython code that declares the C++ class interface for the Simpl `PeakDetection` and `SMSPeakDetection` classes is given in Listing 3.9. The Cython interfaces to the Simpl `Peak` and `Frame` objects (given in *base.pxd*) are `c_Peak` and `c_Frame`, which are imported on lines 7 and 8 respectively. A block that will contain code from the file *peak_detection.h* in the `simpl` namespace is started on line 14. The Cython interface to the `PeakDetection` class then begins on line 17, which is called `c_PeakDetection` in order to avoid confusion with the Python `PeakDetection` class (see Listing 3.10). The `PeakDetection` methods are added to the interface on lines 18-40. Similarly, the Cython interface to the C++ `SMSPeakDetection` class and its methods is given on lines 42 to 48.

c_SMSPeakDetection is declared as a base class of c_PeakDetection.

```
import numpy as np
cimport numpy as np
np.import_array()
from libcpp.vector cimport vector
from libcpp cimport bool

from base cimport c_Peak
from base cimport c_Frame
from base cimport string
from base cimport dtype_t
from base import dtype


cdef extern from "../src/simpl/peak_detection.h" \
    namespace "simpl":

    cdef cppclass c_PeakDetection "simpl::PeakDetection":
        c_PeakDetection()
        int sampling_rate()
        void sampling_rate(int new_sampling_rate)
        int frame_size()
        void frame_size(int new_frame_size)
        int static_frame_size()
        void static_frame_size(bool new_static_frame_size)
        int next_frame_size()
        int hop_size()
        void hop_size(int new_hop_size)
        int max_peaks()
        void max_peaks(int new_max_peaks)
        string window_type()
        void window_type(string new_window_type)
        int window_size()
        void window_size(int new_window_size)
        double min_peak_separation()
        void min_peak_separation(double new_min_peak_separation)
        int num_frames()
        c_Frame* frame(int frame_number)
        void frames(vector[c_Frame*] new_frames)
        void find_peaks_in_frame(c_Frame* frame)
        vector[c_Frame*] find_peaks(int audio_size, double* audio)

    cdef cppclass c_SMSPeakDetection \
        "simpl::SMSPeakDetection"(c_PeakDetection):
        c_SMSPeakDetection()
        void hop_size(int new_hop_size)
        void max_peaks(int new_max_peaks)
        void find_peaks_in_frame(c_Frame* frame)
        vector[c_Frame*] find_peaks(int audio_size, double* audio)
```

Listing 3.9: Cython C++ class interface for PeakDetection and SMSPeakDetection from *peak_detection.pxd*.

The PeakDetection and SMSPeakDetection Cython wrapper classes are defined in *peak_detection.pyx*, with the start of the PeakDetection definition given

in Listing 3.10.

```
13 cdef class PeakDetection:
14     cdef c_PeakDetection* thisptr
15     cdef public list frames
16
17     def __cinit__(self):
18         self.thisptr = new c_PeakDetection()
19
20     def __dealloc__(self):
21         if self.thisptr:
22             del self.thisptr
23
24     def __init__(self):
25         self.frames = []
```

Listing 3.10: Cython `PeakDetection` wrapper class from *peak_detection.pyx*.

The class has a pointer to `c_PeakDetection` called `thisptr` which is defined on line 14. It also has a member variable called `frames`, which is a publicly accessible Python list. The class constructor creates a new `c_PeakDetection` instance on line 18 which is deleted in the destructor on line 22. The frames list is then initialised to an empty Python list on line 15.

Next, Python properties are created from the C++ getter and setter methods, as shown in Listing 3.11. The `frame` method creates a new Python `Frame` object from the existing C++ `c_Frame` instance.

```
27     property sampling_rate:
28         def __get__(self): return self.thisptr.sampling_rate()
29         def __set__(self, int i): self.thisptr.sampling_rate(i)
30
31     property frame_size:
32         def __get__(self): return self.thisptr.frame_size()
33         def __set__(self, int i): self.thisptr.frame_size(i)
34
35     property static_frame_size:
36         def __get__(self): return self.thisptr.static_frame_size()
37         def __set__(self, bool b): self.thisptr.static_frame_size(b)
38
39     def next_frame_size(self):
40         return self.thisptr.next_frame_size()
41
```

```
42      property hop_size:
43          def __get__(self): return self.thisptr.hop_size()
44          def __set__(self, int i): self.thisptr.hop_size(i)
45
46      property max_peaks:
47          def __get__(self): return self.thisptr.max_peaks()
48          def __set__(self, int i): self.thisptr.max_peaks(i)
49
50      property window_type:
51          def __get__(self): return self.thisptr.window_type().c_str()
52          def __set__(self, char* s): self.thisptr.window_type(string(s))
53
54      property window_size:
55          def __get__(self): return self.thisptr.window_size()
56          def __set__(self, int i): self.thisptr.window_size(i)
57
58      property min_peak_separation:
59          def __get__(self): return self.thisptr.min_peak_separation()
60          def __set__(self, double d): self.thisptr.min_peak_separation(d)
61
62      def frame(self, int i):
63          cdef c_Frame* c_f = self.thisptr.frame(i)
64          f = Frame(None, False)
65          f.set_frame(c_f)
66          return f
```

Listing 3.11: Getters and setters from the Cython `PeakDetection` wrapper class in *peak_detection.pyx*.

The Python wrappers for the `find_peaks_in_frame` and the `find_peaks` methods are given in Listing 3.12. The former simply calls the C++ `find_peaks_in_frame` method on line 69 and returns the peaks in the resulting `Frame` object. The Python `find_peaks` method the same as the corresponding C++ method. It steps through the input audio array in `hop_size` steps, updating the frame size at each step if necessary and then creating a new `Frame` object. The `find_peaks_in_frame` method is then called with this `Frame` as an input parameter.

```
68      def find_peaks_in_frame(self, Frame frame not None):
69          self.thisptr.find_peaks_in_frame(frame.thisptr)
70          return frame.peaks
71
72      def find_peaks(self, np.ndarray[dtype_t, ndim=1] audio):
73          self.frames = []
74
75          cdef int pos = 0
76          while pos <= len(audio) - self.hop_size:
```

```
77              if not self.static_frame_size:
78                  self.frame_size = self.next_frame_size()
79
80              frame = Frame(self.frame_size)
81
82              if pos < len(audio) - self.frame_size:
83                  frame.audio = audio[pos:pos + self.frame_size]
84              else:
85                  frame.audio = np.hstack((
86                      audio[pos:len(audio)],
87                      np.zeros(self.frame_size - (len(audio) - pos))
88                  ))
89
90              frame.max_peaks = self.max_peaks
91              self.find_peaks_in_frame(frame)
92              self.frames.append(frame)
93              pos += self.hop_size
94
95          return self.frames
```

Listing 3.12: `find_peaks_in_frame` and `find_peaks` methods from the Cython `PeakDetection` wrapper class in *peak_detection.pyx*.

The Python wrapper for `SMSPeakDetection` (Listing 3.13) inherits almost everything from the Python `PeakDetection` class. As the underlying C++ methods are called in each member function, a derived Python class will generally only have to override the constructor and destructor to set the `thisptr` variable to an instance of the desired C++ class. Here the `find_peaks` method is also overridden in order to call the corresponding C++ `SMSPeakDetection` method. This is necessary as the `SMSPeakDetection` implementation of `find_peaks` is slightly different to the `PeakDetection` version (this difference is discussed in Section 3.3.1).

```
110 cdef class SMSPeakDetection(PeakDetection):
111     def __cinit__(self):
112         if self.thisptr:
113             del self.thisptr
114         self.thisptr = new c_SMSPeakDetection()
115
116     def __dealloc__(self):
117         if self.thisptr:
118             del self.thisptr
119             self.thisptr = <c_PeakDetection*>0
120
121     def find_peaks(self, np.ndarray[dtype_t, ndim=1] audio):
122         self.frames = []
123         cdef vector[c_Frame*] output_frames = \
```

```
124             self.thisptr.find_peaks(len(audio), <double*> audio.data)
125         for i in range(output_frames.size()):
126             f = Frame(output_frames[i].size(), False)
127             f.set_frame(output_frames[i])
128             self.frames.append(f)
129         return self.frames
```

Listing 3.13: Cython `SMSPeakDetection` wrapper class from *peak_detection.pyx*.

3.4 Examples

This section presents five examples that demonstrate some of the functionality of Simpl. Sections 3.4.1 and 3.4.2 show how to detect and plot spectral peaks and sinusoidal partials respectively. The remaining examples all produce synthesised audio files. Section 3.4.3 shows how to analyse an audio sample and synthesise the detected harmonic and noise components. The examples in Sections 3.4.4 and 3.4.5 illustrate how Simpl can be used to transform existing audio files, performing pitch-shifting and time-scaling respectively.

3.4.1 Plotting spectral peaks

The first example (Listing 3.14) shows how Simpl and Matplotlib can be used to plot the spectral peaks that are found in 4096 samples of a clarinet tone using SMS. The samples are extracted starting at the centre of the waveform on line 6. An `SMSPeakDetection` object is created on line 9, with the `max_peaks` parameter set to 20 on line 10. The remaining parameters keep their default values. The default hop size is 512, which should therefore produce 8 frames of audio for the 4096 samples. The `find_peaks` method is then called on line 11, with the resulting list of `Frame` objects saved to a variable called `frames`. The Simpl `plot_peaks` function is then

called with this list of frames as a parameter on line 14. The plot axes are labelled on lines 15 and 16 and finally the figure is displayed on line 17. The resulting plot is shown in Figure 3.3.

```
1  import simpl
2  import matplotlib.pyplot as plt
3
4  # take 4096 samples frames starting at the waveform centre
5  audio = simpl.read_wav('clarinet.wav')[0]
6  audio = audio[len(audio) / 2:(len(audio) / 2) + 4096]
7
8  # peak detection using SMS
9  pd = simpl.SMSPeakDetection()
10 pd.max_peaks = 20
11 frames = pd.find_peaks(audio)
12
13 # plot peaks using matplotlib
14 simpl.plot_peaks(frames)
15 plt.xlabel('Frame Number')
16 plt.ylabel('Frequency (Hz)')
17 plt.show()
```

Listing 3.14: Using a Simpl `SMSPeakDetection` object to plot spectral peaks.

3.4.2 Plotting sinusoidal partials

The second example demonstrates the way that Simpl can be used to detect and plot sinusoidal partials. It also shows the ease with which different sinusoidal modelling algorithms can be combined to perform peak detection and partial tracking. The code is given in Listing 3.15.

This time the full audio sample is used. A list of frames containing spectral peaks detected by the SMS algorithm is saved to the `frames` variable on line 9. The maximum number of peaks is not changed this time and so has the default value of 100. The list of frames is then passed to the `find_partials` method of a `MQPartialTracking` object on line 13, which performs partial tracking using the MQ method. The partials are plotted on line 15 by calling the Simpl

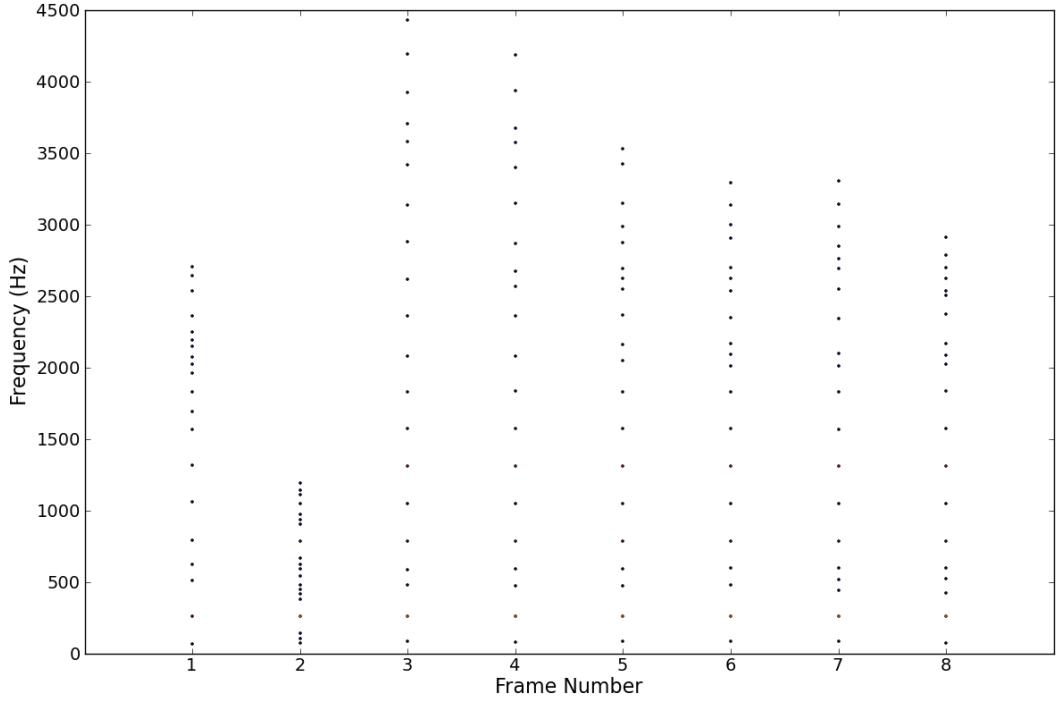

Figure 3.3: Spectral peaks identified from 8 frames of a clarinet sample using the `SMSPeakDetection` class.

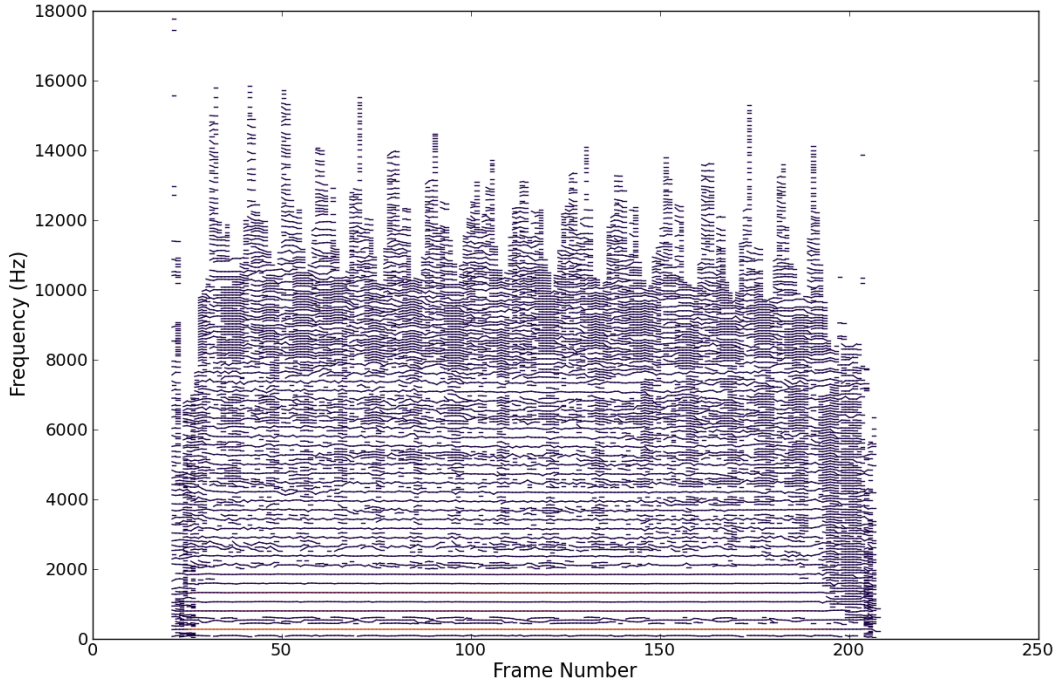

Figure 3.4: Detecting and plotting all sinusoidal partials in a clarinet sample.

`plot_partials` function, and finally the figure is displayed by calling `plt.show` on line 18. The plot of the sinusoidal partials that is produced is shown in Figure 3.4.

```
1  import simpl
2  import matplotlib.pyplot as plt
3
4  # read an audio sample
5  audio = simpl.read_wav('clarinet.wav')[0]
6
7  # peak detection using SMS
8  pd = simpl.SMSPeakDetection()
9  frames = pd.find_peaks(audio)
10
11 # partial tracking using the bandwidth-enhanced model
12 pt = simpl.LorisPartialTracking()
13 frames = pt.find_partials(frames)
14
15 simpl.plot_partials(frames)
16 plt.xlabel('Frame Number')
17 plt.ylabel('Frequency (Hz)')
18 plt.show()
```

Listing 3.15: Detecting and plotting sinusoidal partials.

3.4.3 Synthesis

The synthesis example (Listing 3.16) demonstrates the synthesis of deterministic (harmonic) and stochastic signal components using Simpl. This time spectral peaks are detected using the bandwidth-enhanced model (lines 7 and 8), and partial tracking using SMS (lines 11 and 12). A SndObj Library harmonic synthesis object is created on line 15, and then used to synthesise the deterministic component from the frames of sinusoidal partials on line 16. An SMS stochastic synthesis object is created on line 19, which performs SMS analysis on the original audio waveform and creates a stochastic component (line 20). On lines 23-46 Matplotlib is used to create a plot with 3 subplots, displaying the original waveform, synthesised harmonic component and the synthesised stochastic component respectively. This plot

is shown in Figure 3.5.

```
import simpl
import matplotlib.pyplot as plt

audio = simpl.read_wav('clarinet.wav')[0]

# peak detection using the bandwidth-enhanced model
pd = simpl.LorisPeakDetection()
frames = pd.find_peaks(audio)

# partial tracking using SMS
pt = simpl.SMSPartialTracking()
frames = pt.find_partials(frames)

# synthesis of harmonic component using The SndObj Library
harm_synth = simpl.SndObjSynthesis()
harmonic_component = harm_synth.synth(frames)

# synthesis of stochastic component using SMS
stocastic_synth = simpl.SMSResidual()
stochastic_component = stocastic_synth.synth(audio)

# plot the 3 waveforms
fig = plt.figure()

plt.subplot(3, 1, 1)
plt.ylim([-1, 1])
plt.xlabel('Sample Number')
plt.ylabel('Sample Value')
plt.title('Original Waveform')
plt.plot(audio)

plt.subplot(3, 1, 2)
plt.ylim([-1, 1])
plt.xlabel('Sample Number')
plt.ylabel('Sample Value')
plt.title('Synthesised Harmonic Component')
plt.plot(harmonic_component)

plt.subplot(3, 1, 3)
plt.ylim([-1, 1])
plt.xlabel('Sample Number')
plt.ylabel('Sample Value')
plt.title('Synthesised Stochastic Component')
plt.plot(stochastic_component)

plt.show()
```

Listing 3.16: Synthesising deterministic and stochastic signal components.

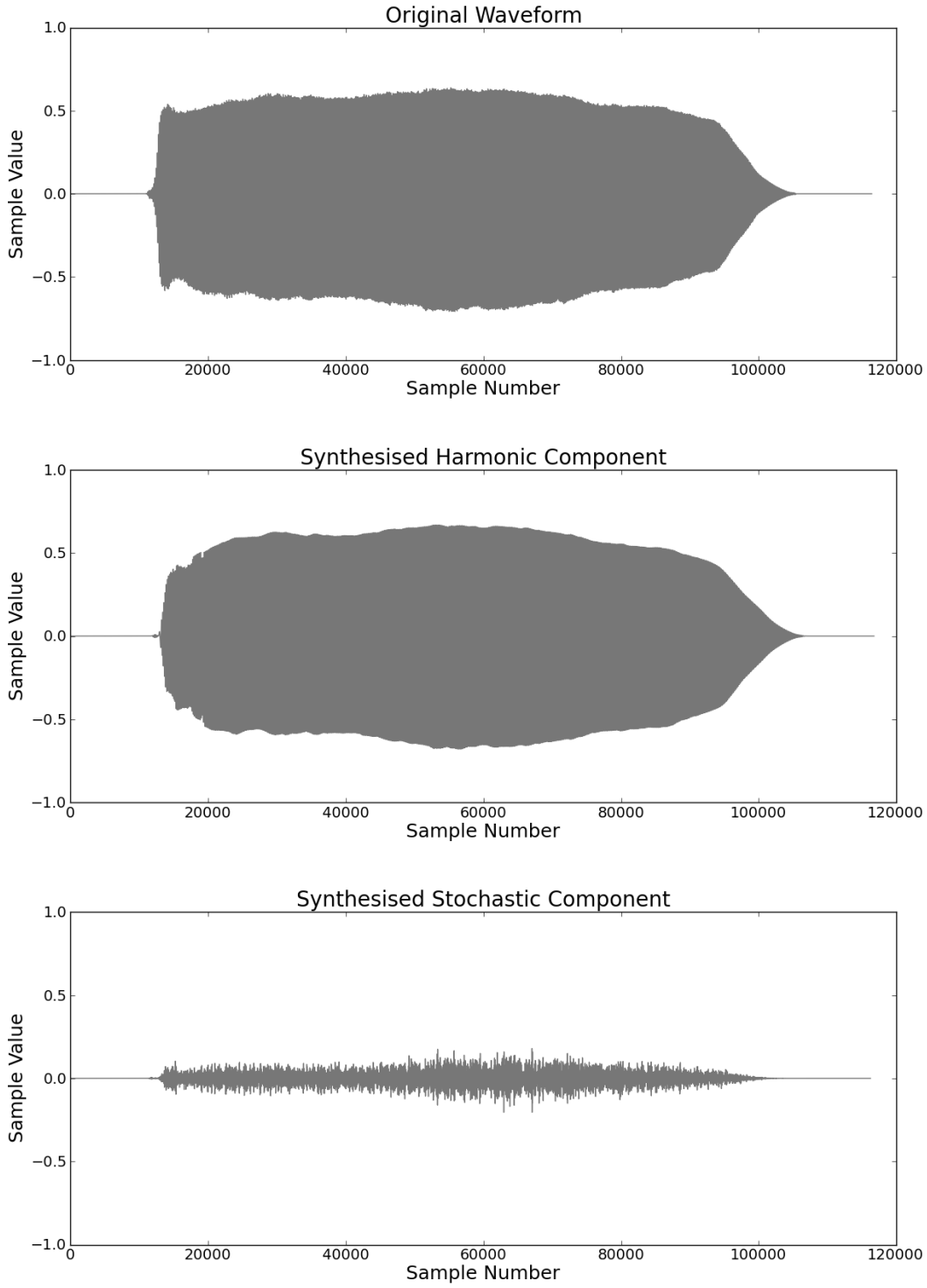

Figure 3.5: A clarinet sample (top), the synthesised deterministic component (middle) and the synthesised stochastic component (bottom) produced by the code in Listing 3.16.

3.4.4 Pitch-shifting

Listing 3.17 shows how Simpl can be used to analyse an audio sample, change the pitch of the harmonic component and then synthesise harmonic and noise components. The input audio sample and its sampling rate are read on line 6. Two more variables are then created on lines 7 and 8, defining the pitch shift amount (in semitones) and the name of the output file respectively. A pitch shift amount of 4 is specified, so the synthesised harmonic component will be 4 semitones (a major third) higher. A negative `pitch_shift_amount` can be specified in order to lower the resulting pitch.

Sinusoidal peaks are identified using the bandwidth-enhanced model on line 11, followed by SMS partial tracking on line 13. Lines 15 and 16 calculate the scaling factor that must be applied to the frequency of each sinusoidal partial in order to achieve the desired pitch shift amount, with the result saved to the `freq_scale` variable on line 16. The pitch shift is performed on lines 18-22 by looping through each sinusoidal partial in each frame and multiplying its frequency by the `freq_scale` value.

The deterministic and stochastic components are then synthesised on lines 25 and 27 respectively. The SndObj library is used to synthesise the deterministic component (using additive synthesis) and SMS is used to create the stochastic signal. The two synthesised components are then combined on line 29 and saved to a new file called *clarinet_pitch_shifted.wav* on line 31.

```
1 import math
2 import numpy as np
3 import scipy.io.wavfile as wav
4 import simpl
```

```
5
6  audio, sampling_rate = simpl.read_wav('clarinet.wav')
7  pitch_shift_amount = 4
8  output_file = 'clarinet_pitch_shifted.wav'
9
10 pd = simpl.LorisPeakDetection()
11 frames = pd.find_peaks(audio)
12 pt = simpl.SMSPartialTracking()
13 frames = pt.find_partials(frames)
14
15 twelfth_root_2 = math.pow(2.0, 1.0 / 12)
16 freq_scale = math.pow(twelfth_root_2, pitch_shift_amount)
17
18 for frame in frames:
19     partials = frame.partials
20     for p in partials:
21         p.frequency *= freq_scale
22     frame.partials = partials
23
24 synth = simpl.SndObjSynthesis()
25 harm_synth = synth.synth(frames)
26 r = simpl.SMSResidual()
27 res_synth = r.synth(audio)
28
29 audio_out = harm_synth + res_synth
30 audio_out = np.asarray(audio_out * 32768, np.int16)
31 wav.write(output_file, sampling_rate, audio_out)
```

Listing 3.17: Pitch-shifting the deterministic component.

3.4.5 Time-scaling

An example that uses Simpl to time-scale the harmonic component from a piano sample is given in Listing 3.18. The sample is read on line 5, with the desired time-scale factor and output file name specified on lines 6 and 7 respectively. A time-scale factor of 3 is chosen here, which will produce a synthesised signal that is 3 times the duration of the original. Spectral peaks are identified on line 10 and used to form sinusoidal partials on line 12, followed by the creation of the audio output array (initially empty) on line 15.

On line 16 a variable called `step_size` is defined with a value equal to the inverse of the time-scale factor. This will be used to advance the value of the current frame number, which is initialised to 0 on line 17. The time-scaling procedure

works by passing frames to the synthesis object at a different rate. To achieve a time-scale factor of 2, each frame would be passed to the synthesis object twice. If the time-scale factor was 0.5 then every second frame would be skipped, and so on. The synthesis procedure interpolates smoothly between input frames so no discontinuities occur in the output signal.

This time-scaling procedure is performed on lines 19-23. The synthesis frame index is taken to be the closest integer that is less than the value of `current_frame`. The frame at this index position is passed to the `synth_frame` method, producing one output frame which is appended to the rest of the output audio signal on line 22. This process continues as long as the `current_frame` variable is less than the number of frames in the original signal. The final output array is then saved to a new file called *piano_time_scaled.wav* on line 26.

```
1  import numpy as np
2  import scipy.io.wavfile as wav
3  import simpl
4
5  audio, sampling_rate = simpl.read_wav('piano.wav')
6  time_scale_factor = 3
7  output_file = 'piano_time_scaled.wav'
8
9  pd = simpl.LorisPeakDetection()
10 peaks = pd.find_peaks(audio)
11 pt = simpl.SMSPartialTracking()
12 partials = pt.find_partials(peaks)
13
14 synth = simpl.SndObjSynthesis()
15 audio_out = np.array([])
16 step_size = 1.0 / time_scale_factor
17 current_frame = 0
18
19 while current_frame < len(partials):
20     i = int(current_frame)
21     frame = synth.synth_frame(partials[i])
22     audio_out = np.hstack((audio_out, frame))
23     current_frame += step_size
24
25 audio_out = np.asarray(audio_out * 32768, np.int16)
26 wav.write(output_file, sampling_rate, audio_out)
```

Listing 3.18: Time-scaling the deterministic component.

3.5 Conclusions

This chapter introduced Simpl, a library for sinusoidal modelling that is written in C++ and Python. Simpl aims to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API. It is primarily intended as a tool for composers and other researchers, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms. The chapter began with an overview of the motivation for creating Simpl and then explained the choice of Python as an interactive environment for musical experimentation and rapid-prototyping. An overview of the Simpl library was then provided, followed by an in-depth look at the implementation of the SMS peak detection module. The chapter concluded with some examples that demonstrated the functionality of the library.

Chapter 4

Real-time onset detection

The Simpl sinusoidal modelling library that was introduced in Chapter 3 is a comprehensive framework for the analysis and synthesis of deterministic and stochastic sound components. However, the underlying sinusoidal models assume that both components vary slowly with time. During synthesis, analysis parameters are smoothly interpolated between frames. This process results in high-quality synthesis of the main sustained portion of a note[1] but can result in transient components being diffused or "smeared" between frames. Some of these transient components will correspond to note *attack* sections, which have been shown to be very important to the perception of musical timbre[2]. It is therefore desirable to firstly be able to accurately identify these regions, and secondly to ensure that the model is able to reproduce them with a high degree of fidelity.

This deficiency in the synthesis of attack transients by sinusoidal models was noted by Serra in [108], where he also suggested a solution: store the original sam-

[1] Also referred to as the *steady-state*.

[2] A good overview of some of the research into the perceptual importance of the temporal evolution of musical tones is given in [45]

ples for the note attack region and then splice them together with the remainder of the synthesised sound. Serra notes that even if the phases are not maintained during sinusoidal synthesis, the splice can still be performed successfully by using a small cross-fade between the two sections. No method was proposed for automatically identifying these attack regions however.

As neither Simpl nor any of its underlying sinusoidal modelling implementations provide a means to automatically identify transient signal components, this functionality must be added to our real-time sinusoidal modelling framework in order to improve the quality of the synthesis of transient regions. The attack portion of a musical note is usually taken to be a region (possibly of varying duration) that immediately follows a note onset [11, 44, 77, 73]. The automatic identification of note attack regions can therefore be broken up into two distinct steps:

1. Find note onset locations.

2. Calculate the duration of the transient region following a given note onset.

This chapter deals with step 1, detailing the creation of a new algorithm and software framework for real-time automatic note onset detection. Section 4.1 defines some terms that are used in the remainder of the Chapter. This is followed by an overview of onset detection techniques in general in Section 4.2, and then a description of several onset detection techniques from the literature in Section 4.3. Section 4.4 introduces *Modal*, our new open source library for musical onset detection. In Section 4.6, we suggest a way to improve on these techniques by incorporating linear prediction [75]. A novel onset-detection method that uses sinusoidal modelling is presented in Section 4.8. Modal is used to evaluate all of the previously described algorithms, with the results being given in Sections 4.5, 4.7 and 4.9. This

evaluation includes details of the performance of all of the algorithms in terms of both accuracy and computational requirements.

4.1 Definitions

The terms *audio buffer* and *audio frame* are distinguished here as follows:

Audio buffer: A group of consecutive audio samples taken from the input signal. The algorithms in this chapter all use a fixed buffer size of 512 samples.

Audio frame: A group of consecutive audio buffers. All the algorithms described here operate on overlapping, fixed-sized frames of audio. These frames are four audio buffers (2,048 samples) in duration, consisting of the most recent audio buffer which is passed directly to the algorithm, combined with the previous three buffers which are saved in memory. The start of each frame is separated by a fixed number of samples, which is equal to the buffer size.

As one of the research goals is to create software that can be used in a live musical performance context, our definition of *real-time* software has a slightly different meaning to its use in some other areas of Computer Science and Digital Signal Processing. To say that a system runs in real-time, we require two characteristics:

1. Low latency: The latency experienced by the performer should be low enough that the system is still useful in a live context. While there is no absolute rule that specifies how much latency is acceptable in this situation, the default latency of the sound transformation systems that are described in this thesis is 512 samples (or about 11.6 ms when sampled at 44.1 kHz), which should meet these requirements in many cases.

For the evaluation of the real-time onset detection process however, the time between an onset occurring in the input audio stream and the system correctly registering an onset occurrence must be no more than 50 ms. This value was chosen to allow for the difficulty in specifying reference onsets, which is described in more detail in Section 4.4. All of the onset-detection schemes that are described in this chapter have latency of 1,024 samples (the size of two audio buffers), except for the peak amplitude difference method (Section 4.8.2) which has an additional latency of 512 samples, or 1,536 samples of latency in total. This corresponds to latency times of 23.2 and 34.8 ms respectively, at a sampling rate of 44.1 kHz. The reason for the 1,024 sample delay on all the onset-detection systems is explained in Section 4.2.2, while the cause of the additional latency for the peak amplitude difference method is given in Section 4.8.2.

2. **Low processing time:** The time taken by the algorithm to process one frame of audio must be less than the duration of audio that is held in each buffer. As the buffer size is generally fixed at 512 samples, the algorithm must be able to process a frame in 11.6 ms or less when operating at a sampling rate of 44.1 kHz.

4.2 The general form of onset detection systems

As onset locations are typically defined as being the start of a transient, the problem of finding their position is linked to the problem of detecting transient intervals in the signal. Another way to phrase this is to say that onset detection is the process of identifying which parts of a signal are relatively unpredictable. The majority of

the onset detection techniques that are described in the literature[3] involve an initial data reduction step, transforming the audio signal into an *onset detection function*. Onsets are then found by looking for local maxima in the onset detection function.

4.2.1 Onset detection functions

An onset detection function (ODF) is a representation of the audio signal at a much lower sampling rate. The ODF usually consists of one value for every frame of audio and should give a good indication as to the measure of the unpredictability of that frame. Higher values correspond to greater unpredictability. An example of a sampled drum phrase and the resulting ODF, which in this case is calculated using the spectral difference method[4], is provided in Figure 4.1.

4.2.2 Peak detection

The next stage in the onset-detection process is to identify local maxima, also called *peaks*, in the ODF. The location of each peak is recorded as an onset location if the peak value is above a certain threshold. While peak picking and thresholding are described elsewhere in the literature [57], both require special treatment to operate within the constraints imposed by real-time musical signal processing applications. Our real-time peak detection process is described by Figure 4.2[5].

When processing a real-time stream of ODF values, the first stage in the peak-detection algorithm is to see if the current values are local maxima. In order to make

[3] A good overview of onset detection systems can be found in [11].
[4] The spectral difference method for creating ODFs is described in Section 4.3.2
[5] As our comparison of onset detection systems focuses on evaluating the performance of different real-time ODFs, the same peak detection and thresholding algorithms are applied to each ODF in our final evaluation (Section 4.9).

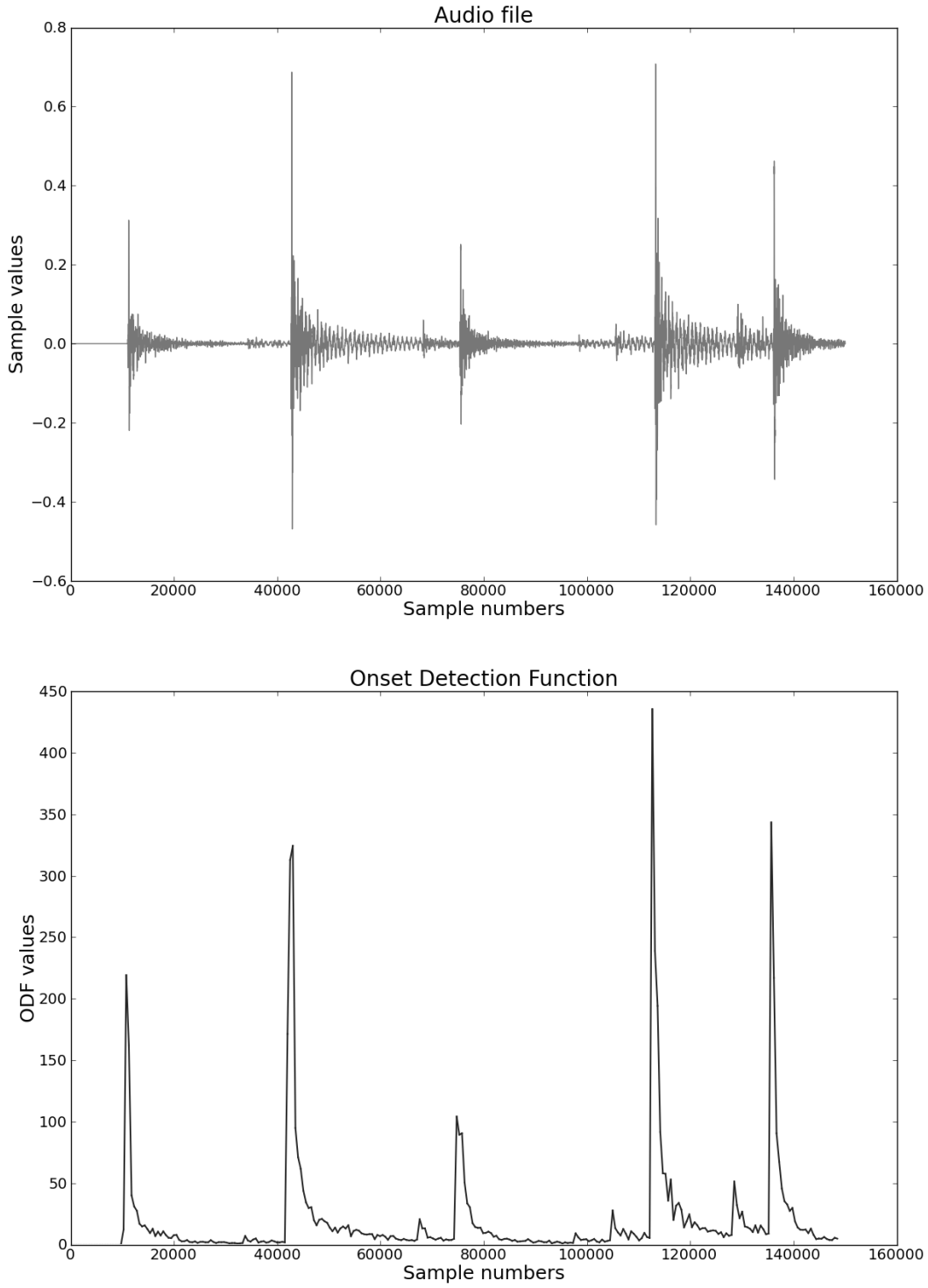

Figure 4.1: Sample of a drum phrase and the ODF generated from the sample using the spectral difference method.

this assessment the current ODF value must be compared to the two neighbouring values. As we cannot look ahead to get the next ODF value, it is necessary to save both the previous and the current ODF values and wait until the next value has been computed to make the comparison. This means that there must always be some additional latency in the peak-picking process, in this case equal to the buffer size which is fixed at 512 samples. When working with a sampling rate of 44.1 kHz, this results in a total system latency of two buffer sizes or approximately 23.2 ms.

Input: ODF value
Output: Whether or not previous ODF value represents a peak (Boolean)
$IsOnset \longleftarrow False$;
if *PreviousValue* $>$ *CurrentValue* **and** *PreviousValue* $>$ *TwoValuesAgo* **then**
  if *PreviousValue* $>$ *CalculateThreshold()* **then**
    $IsOnset \longleftarrow True$;
$UpdatePreviousValues()$;
return *IsOnset*

Figure 4.2: Real-time ODF peak detection (one buffer delay).

4.2.3 Dynamic threshold calculation

Dynamic thresholds are calculated according to Equation 4.1, where σ_n is the threshold value at frame n, $O[n_m]$ is the previous m values of the ODF at frame n, λ is a positive median weighting value, and α is a positive mean weighting value.

$$\sigma_n = \lambda \times \text{median}(O[n_m]) + \alpha \times \text{mean}(O[n_m]) + N. \tag{4.1}$$

This process is similar to the median/mean function that is described in [18]. The only difference is the addition of the term N, which is defined in Equation 4.2,

where v is the value of the largest peak detected so far and w is a weighting value. For indefinite real-time use it is advisable to either set $w = 0$ or to update w at regular intervals to account for changes in dynamic level.

$$N = w \times v, \tag{4.2}$$

An example of the dynamic threshold (dashed line) generated from the ODF of a clarinet sample is shown in Figure 4.3. The threshold was computed according to Equations 4.1 and 4.2 with $m = 7$, $\lambda = 1.0$, $\alpha = 2.0$ and $w = 0.05$. Red circles indicate ODF peaks that are above the threshold and are therefore assumed to indicate note onset locations.

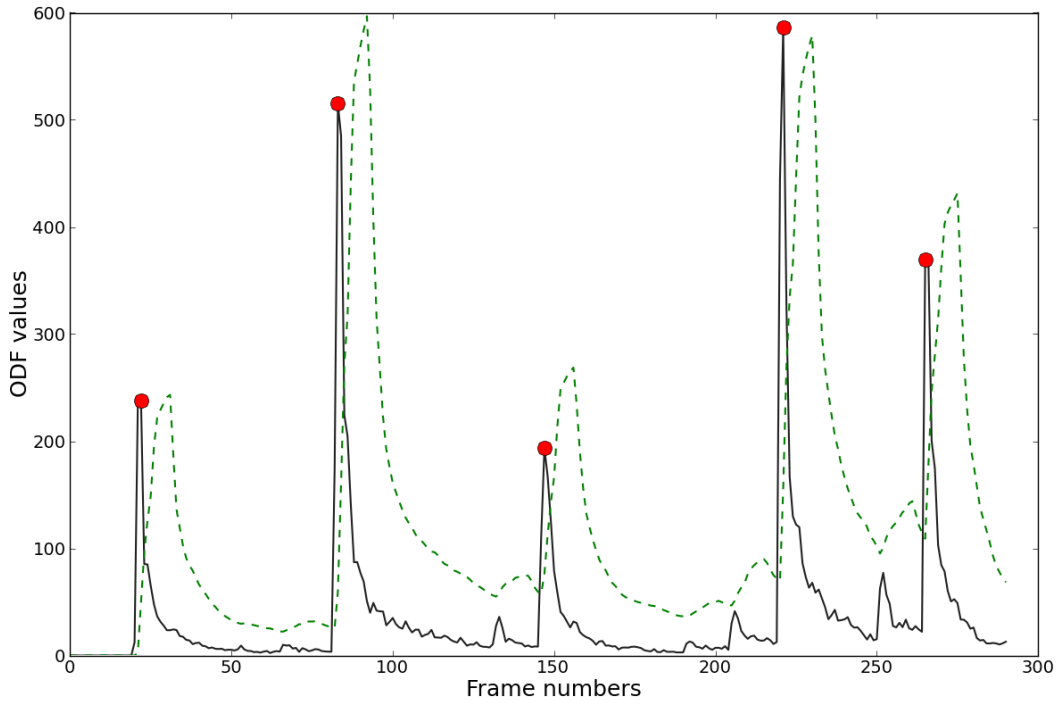

Figure 4.3: A clarinet sample, an ODF calculated using the complex domain method and the resulting dynamic threshold values (dashed line). Circles indicate ODF peaks that are above the threshold and are therefore assumed to indicate note onset locations.

4.3 Onset detection functions

This section reviews several existing approaches to creating ODFs that can be used in a real-time situation. Each technique operates on frames of N samples, with the start of each frame being separated by a fixed buffer size of h samples. The ODFs return one value for every frame, corresponding to the likelihood of that frame containing a note onset. A full analysis of the detection accuracy and computational efficiency of each algorithm is given in Section 4.9.

4.3.1 Energy ODF

The energy ODF [30] is the most conceptually simple of the surveyed ODF methods, and is also the most computationally efficient. It is based on the premise that musical note onsets often have more energy than the steady-state component of the note. This is because for many acoustic instruments, this corresponds to the time at which the excitation is applied. Larger changes in the amplitude envelope of the signal should therefore coincide with onset locations. For each frame, the energy is given by Equation 4.3 where $E(l)$ is the energy of frame l, and $x(n)$ is the value of the n-th sample in the frame.

$$E(l) = \sum_{n=0}^{N} x(n)^2 \tag{4.3}$$

The value of the energy ODF (ODF_E) for a frame l is the absolute value of the difference in energy values between consecutive frames $l-1$ and l, as defined by Equation 4.4.

$$\text{ODF}_\text{E}(l) = |E(l) - E(l-1)| \tag{4.4}$$

4.3.2 Spectral difference ODF

Many recent techniques for creating ODFs have tended towards identifying time-varying changes in a frequency domain representation of an audio signal. These approaches have proven to be successful in a number of areas, such as in detecting onsets in polyphonic signals [82] and in detecting "soft" onsets created by instruments such as the bowed violin which do not have a percussive attack [31]. The spectral difference ODF (ODF_{SD}) is calculated by examining frame-to-frame changes in the STFT spectra of an audio signal and so falls into this category.

The spectral difference [31] is the absolute value of the change in magnitude between corresponding bins in consecutive frames. As a new musical onset will often result in a sudden change in the frequency content in an audio signal, large changes in the average spectral difference of a frame will often correspond with note onsets. The spectral difference ODF is thus created by summing the spectral difference across all bins in a frame. This is defined by Equation 4.5 where $|X_l(k)|$ is the magnitude of the k-th frequency bin in the l-th frame.

$$\text{ODF}_{\text{SD}}(l) = \sum_{k=0}^{N/2} ||X_l(k)| - |X_{l-1}(k)|| \tag{4.5}$$

4.3.3 Complex domain ODF

Another way to view the construction of an ODF is in terms of *predictions* and *deviations* from predicted values. For every spectral bin in the Fourier transform of a frame of audio samples, the spectral difference ODF predicts that the next magnitude value will be the same as the current one. In the steady state of a musical note, changes in the magnitude of a given bin between consecutive frames should

be relatively low, and so this prediction should be accurate. In transient regions, these variations should be more pronounced and so the average deviation from the predicted value should be higher, resulting in peaks in the ODF.

Instead of making predictions using only the bin magnitudes, the complex domain ODF [12] attempts to improve the prediction for the next value of a given bin using combined magnitude and phase information. The magnitude prediction is the magnitude value from the corresponding bin in the previous frame. In polar form, we can write this predicted value as

$$\hat{R}_l(k) = |X_{l-1}(k)| \tag{4.6}$$

The phase prediction is formed by assuming a constant rate of phase change between frames. This is given by Equation 4.7, where $princarg$ maps the phase to the range $[-\pi, \pi]$ and $\varphi_l(k)$ is the phase of the k-th bin in the l-th frame.

$$\hat{\phi}_l(k) = princarg[2\varphi_{l-1}(k) - \varphi_{l-2}(k)] \tag{4.7}$$

If $R_l(k)$ and $\phi_l(k)$ are the actual values of the magnitude and phase of bin k in frame l respectively, then the deviation between the prediction and the actual measurement is the Euclidean distance between the two complex phasors, defined in Equation 4.8.

$$\Gamma_l(k) = \sqrt{R_l(k)^2 + \hat{R}_l(k)^2 - 2R_l(k)\hat{R}_l(k)\cos(\phi_l(k) - \hat{\phi}_l(k))} \tag{4.8}$$

The complex domain ODF (ODF_{CD}) is the sum of these deviations across all the

bins in a frame, as given in Equation 4.9.

$$\mathrm{ODF}_{\mathrm{CD}}(n) = \sum_{k=0}^{N/2} \Gamma_l(k) \tag{4.9}$$

4.4 Modal

A common way to evaluate the accuracy of an onset detection system is to use it to analyse a set of audio samples and compare the computed note onset locations with the "true" onset locations (also referred to as *reference onsets*) for each sample [83]. An onset could then be said to be correctly detected if it lies within a chosen time interval around the reference onset. This time interval is referred to as the *detection window*.

Obtaining the reference onset locations is not a straight-forward process however. In reality it is difficult to give exact values for reference onsets from natural sounds, particularly in the case of instruments with a soft attack such as the flute or bowed violin. Finding these reference onsets generally involves human annotation of audio samples, a process which inevitably leads to inconsistencies and inaccuracies. Leveau et al. found that the annotation process is dependent on the listener, the software used to label the onsets and the type of music being labelled [72]. Vos and Rasch [122] make a distinction between the *Physical Onset Time* and the *Perceptual Onset Time* of a musical note, which again can lead to differences between the values selected as reference onsets, particularly if there is a mixture of natural and synthetic sounds. To compensate for these limitations of the annotation process, we follow the decision made in a number of recent studies [11, 115, 28] to use a detection window that is 50 ms in duration when evaluating the performance

of onset detection algorithms.

The evaluation process requires two components:

1. A set of audio samples with accurate reference onset values. Preferably these samples will have a license that allows them to be freely distributed so that other researchers can both verify the results of our evaluation and also the samples in their own work.

2. Software implementations of the general onset detection system, implementations of each ODF, and a program that compares the computed onset locations with the reference onset values.

These components are discussed in Sections 4.4.1, and 4.4.2. The results of the evaluation are given in Section 4.5.

4.4.1 Reference samples

When seeking to evaluate the ODFs that are described in Section 4.3, to the best of our knowledge the Sound Onset Labellizer (SOL) [72] was the only collection of reference samples that had been freely released to the public. Unfortunately it was not available at the time of evaluation. The SOL reference set also makes use of files from the RWC database [41], which although publicly available is not free and does not allow free redistribution. This lead us to create our own set of reference samples, which together with our onset detection software constitutes the **M**usical **O**nset **D**atabase **A**nd **L**ibrary (Modal).

The Modal reference set is a collection of samples which all have creative commons licensing allowing for free reuse and redistribution. It currently consists of 71

samples which are quite diverse in nature, covering percussive and non-percussive sounds from a mixture of western orchestral instruments, contemporary western instruments and vocal samples. This sample set was selected to ensure that a wide variety of sound sources with different onset types (both "soft" and "hard" onsets) were considered in the evaluation process. Table 4.1 provides a list of the samples in the modal database together with a description of the sound sources that comprise each sample[6].

The modal reference set includes hand-annotated onset locations for each audio sample. There are 501 reference onset locations in total, with all onset locations annotated by a single listener. The samples and all associated metadata are stored and transmitted as a HDF5 database [47]. Python scripts are provided for adding samples (in *wav* format) and metadata to the database and for extracting samples and metadata.

Reference onset locations can be found using any audio playback software[7] and then added to the database. However to simplify this process a small GUI application is included with Modal that can playback audio files and mark specific sample values as being onset locations. This application is written in Python using the Qt framework [86]. A screenshot of the application is given in Figure 4.4.

[6]The majority of the sounds are taken from the OLPC free sample library [90]. The filenames have not been changed.

[7]Software that provides visualisations of the waveform and that allows for sample accurate adjustment of the playback position is desirable however.

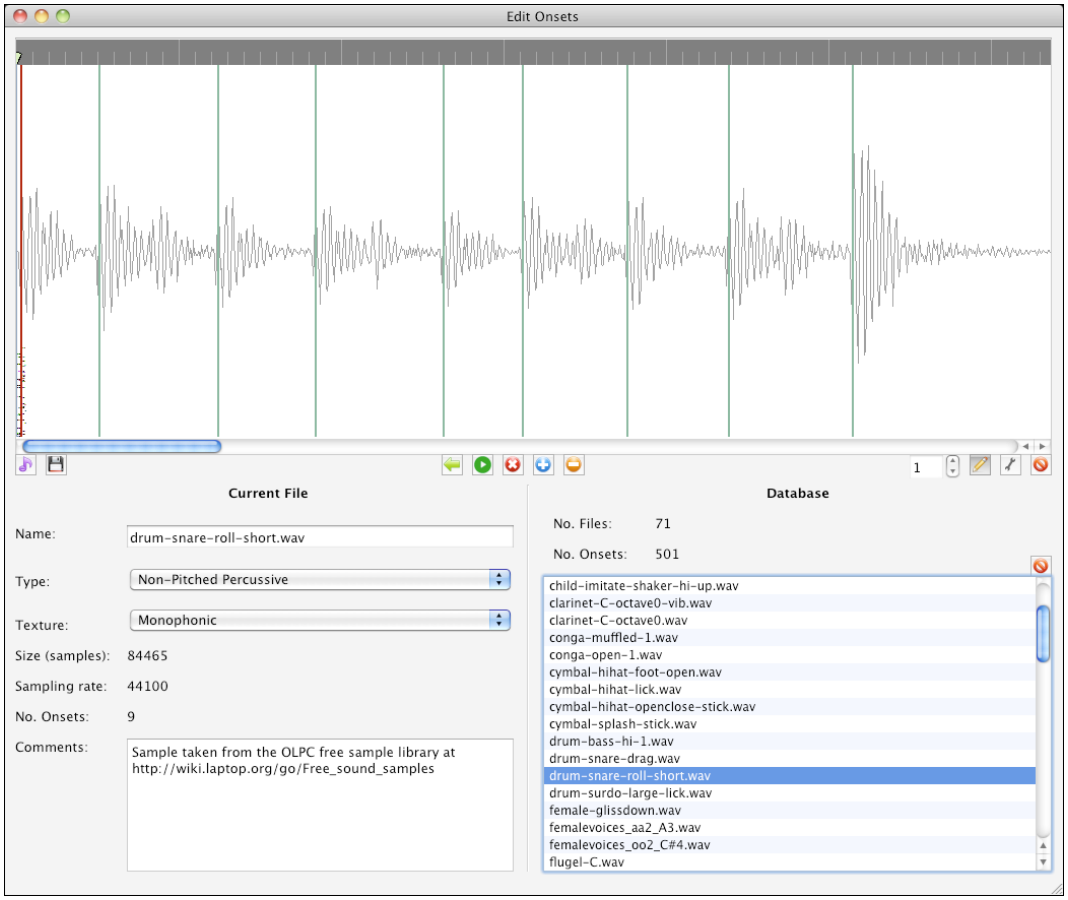

Figure 4.4: Modal's Onset Editor.

Sample Name	Description
bass-elec-C-lo.wav	A single note played on an electric bass guitar.
bass-elec2-E-lick.wav	A sequence of 16 notes played on an electric bass guitar.
bell-octave0.wav	A single note played on a bell.
bell-octave1.wav	A single note played on a bell.
bell-octave2.wav	A single note played on a bell.
child-imitate-owl.wav	A vocal sample consisting of a child imitating an owl call (2 sound events).
child-imitate-shaker-hi-up.wav	A vocal sample consisting of a child imitating a shaker (1 sound event).
clarinet-C-octave0.wav	A single note played on a clarinet.
clarinet-C-octave0-vib.wav	A single note (with vibrato) played on a clarinet.
conga-muffled-1.wav	A single hit on a conga drum (muted).
conga-open-1.wav	A single hit on a conga drum.
cymbal-hihat-foot-open.wav	A single hit on a hi-hat cymbal.
cymbal-hihat-lick.wav	A sequence of 8 hits on a hi-hat cymbal.
cymbal-hihat-openclose-stick.wav	Two hits on a hi-hat cymbal (1 open and 1 closed).
cymbal-splash-stick.wav	A single hit on a cymbal.
drum-bass-hi-1.wav	A single hit on a bass drum.
drum-snare-drag.wav	A sequence of 3 hits on a snare drum.
drum-snare-roll-short.wav	A roll on a snare drum (9 hits).
drum-surdo-large-lick.wav	A sequence of 9 hits on a surdo drum.
female-glissdown.wav	A female vocal glissando.
femalevoices_aa2_A3.wav	Female voices singing a single note.

femalevoices_oo2_C#4.wav	Female voices singing a single note.
flugel-C.wav	A single note played on a flugelhorn.
flugel-lick.wav	A sequence of 8 notes played on a flugelhorn.
flute-alto-C.wav	A single note played on a flute.
flute-alto-lick.wav	A sequence of 15 notes played on a flute.
glitch_groove_7.wav	A sequence of 30 percussive sounds that are electronically generated and/or manipulated.
guitar-ac-E-octave1.wav	A single note played on an acoustic guitar.
guitar_chord1.wav	A single chord played on an acoustic guitar.
guitar_chord2.wav	A single chord played on an acoustic guitar.
guitar_chord3.wav	A single chord played on an acoustic guitar.
guitar-classical-E-octave1-vib.wav	A single note (with vibrato) played on a classical guitar.
guitar-classical-lick.wav	A sequence of 2 notes and 3 chords played on a classical guitar.
guitar-elec-hollow-lick.wav	A sequence of 15 notes played on an electric guitar.
guitar-elec-solid-dist-E-octave1-long.wav	A single note played on an electric guitar (with distortion).
guitar-elec-solid-dist-EBfifths-octave1-long.wav	A double stop played on an electric guitar (with distortion).
guitar-elec-solid-dist-lick.wav	A sequence of 8 notes played on an electric guitar (with distortion).

guitar-elec-solid-lick.wav	A sequence of 9 overlapping notes and chords played on an electric guitar.
guitar_riff_cutup1.wav	A sequence of 17 chords played on an acoustic guitar.
guitar_riff_cutup3.wav	A sequence of 12 chords played on an acoustic guitar.
malevoice_aa_A2.wav	A male voice singing a single note.
malevoices_oo2_C#4.wav	Male voices singing a single note.
metal_beat_1.wav	A sequence of 11 percussive hits on a metallic surface.
metal_beat_2.wav	A sequence of 12 percussive hits on a metallic surface.
mix1.wav	A sequence of 9 notes played on a combination of string and percussive instruments.
mix2.wav	A sequence of 7 notes played on a combination of woodwind, string and pitched percussive instruments.
mix3.wav	A sequence of 4 notes played using a combination of string instruments and voices.
mix4.wav	A sequence of 26 notes played using a combination of string instruments, woodwind instruments, percussive instruments and voices.
piano_B4.wav	A single note played on a piano.
piano_chord.wav	A broken chord (4 notes) played on a piano.
piano_G2.wav	A single note played on a piano.
piano-studio-octave1.wav	A single note played on a piano.

prep_piano_C0_2.wav	A single note played on a prepared piano.
prep_pianoE0_2.wav	A single note played on a prepared piano.
prep_piano_hit1.wav	A single chord played on a prepared piano.
sax-bari-C-lo.wav	A single note played on a baritone saxophone.
sax-bari-lick.wav	A sequence of 9 notes played on a baritone saxophone.
sax-tenor-lick.wav	A sequence of 9 notes played on a tenor saxophone.
shaker.wav	A recording of a shaker (2 sound events).
shenai-C.wav	A sequence of 2 notes played on a shehnai.
singing-female1-C-hi.wav	A single note sung by a female voice.
singing-womanMA-C-oo.wav	A single note sung by a female voice.
tabla-lick.wav	A sequence of 6 notes played on a tabla.
tabla-lick-voiceanddrums.wav	A sequence of 6 notes played using a combination of tabla and voice.
techno_sequence3.wav	A sequence of 109 electronic sounds.
thisIsTheSoundOfAVocoder_edited.wav	A sequence of 14 notes created by processing a speech sample using a vocoder.
timbale-lick.wav	A sequence of 13 notes played on timbales.
trpt-C-lo.wav	A single note played on a trumpet.
trpt-lick.wav	A sequence of 13 notes played on a trumpet.

twangs_1.wav	A sequence of 10 electronic notes.
vocoder1.wav	A sequence of 31 notes created by processing a speech sample using a vocoder.

Table 4.1: Modal database samples.

4.4.2 Modal software

As well as containing a database of samples and reference onset locations, Modal includes software implementations of all of the ODFs that are discussed in this chapter as well an implementation of the real-time onset detection system that is presented in Section 4.2. ODFs can be computed in a real-time (streaming) mode or in non-real-time. The software is written in C++ and Python and can be used as a library in C++ applications or loaded as a Python module. Python code for plotting onset data is also provided. The Modal software library is free software and is available under the terms of the GNU General Public License (GPL).

Modal onset detection example

Listing 4.1 consists of a Python program that uses Modal to detect onsets in a short audio sample. A piano sample is read on line 7. An instance of the Modal spectral difference ODF class is created on line 12, and configured on lines 13-15 to use a hop size of 512 samples, an analysis frame size of 2048 samples and to use the same sampling rate as the input file. A NumPy array that will hold the computed ODF values is then allocated on line 16, and passed to the ODF object's `process` method along with the input audio file on line 17. Onset locations are then calculated us-

ing a Modal `OnsetDetection` object. The returned onset locations are relative to the ODF which consists of 1 value for every `hop_size` samples. To get the onset location in samples, these values are therefore multiplied by the hop size.

The results of the onset detection process are plotted on lines 24-51, with the image that is produced shown in Figure 4.5. Three subplots are shown: the first (top) is the original waveform. The second (middle) plot shows the normalised ODF (solid line), the dynamic threshold (horizontal dashed line) and the detected onsets (vertical dashed lines). The final plot (bottom) shows the onset locations plotted against the original waveform.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import simpl
4  import modal
5
6
7  audio, sampling_rate = simpl.read_wav('piano.wav')
8
9  frame_size = 2048
10 hop_size = 512
11
12 odf = modal.SpectralDifferenceODF()
13 odf.set_hop_size(hop_size)
14 odf.set_frame_size(frame_size)
15 odf.set_sampling_rate(sampling_rate)
16 odf_values = np.zeros(len(audio) / hop_size, dtype=np.double)
17 odf.process(audio, odf_values)
18
19 onset_det = modal.OnsetDetection()
20 onsets = onset_det.find_onsets(odf_values) * hop_size
21
22 # plot the original sample, the ODF and the threshold and onset location(s),
23 # and the original file with the detected onset location(s)
24 fig = plt.figure(1)
25
26 plt.subplot(3, 1, 1)
27 plt.ylim([-1.1, 1.1])
28 plt.title('Original waveform')
29 plt.xlabel('Sample Number')
30 plt.ylabel('Sample Value')
31 plt.tick_params(labelsize=14)
32 plt.plot(audio)
33
34 plt.subplot(3, 1, 2)
35 plt.title('ODF, threshold and detected onset location(s)')
36 plt.xlabel('Sample Number')
37 plt.ylabel('ODF Value')
38 modal.plot_detection_function(odf_values, hop_size=hop_size)
```

```
39  modal.plot_detection_function(onset_det.threshold, hop_size=hop_size,
40                                colour='green', linestyle='--')
41  modal.plot_onsets(onsets, min_height=0)
42
43  plt.subplot(3, 1, 3)
44  plt.title('Original waveform and detection onset location(s)')
45  plt.xlabel('Sample Number')
46  plt.ylabel('Sample Value')
47  plt.ylim([-1.1, 1.1])
48  modal.plot_onsets(onsets)
49  plt.plot(audio)
50
51  plt.show()
```

Listing 4.1: Detecting onsets in an audio file using Modal.

Modal software overview

Lists of the modules, classes and functions that are provided in Modal are given in Tables 4.2 and 4.3. Functions that are marked with an asterisk (*) currently have no C++ implementation and so are only available from Python[8]. Classes begin with capital letters while functions start with lowercase letters. The full source code to Modal can be found on the accompanying CD.

[8] In the case of the `num_onsets` and `samples` functions this is because they depend on the third-party `h5py` Python library. The functions `plot_onsets` and `plot_detection_function` depend on the Python Matplotlib library. The `autocorrelation` and `covariance` functions were not implemented in C++ as it was found that obtaining linear prediction coefficients using the Burg method resulted in improved detection accuracy (linear prediction is discussed in more detail in Section 4.6).

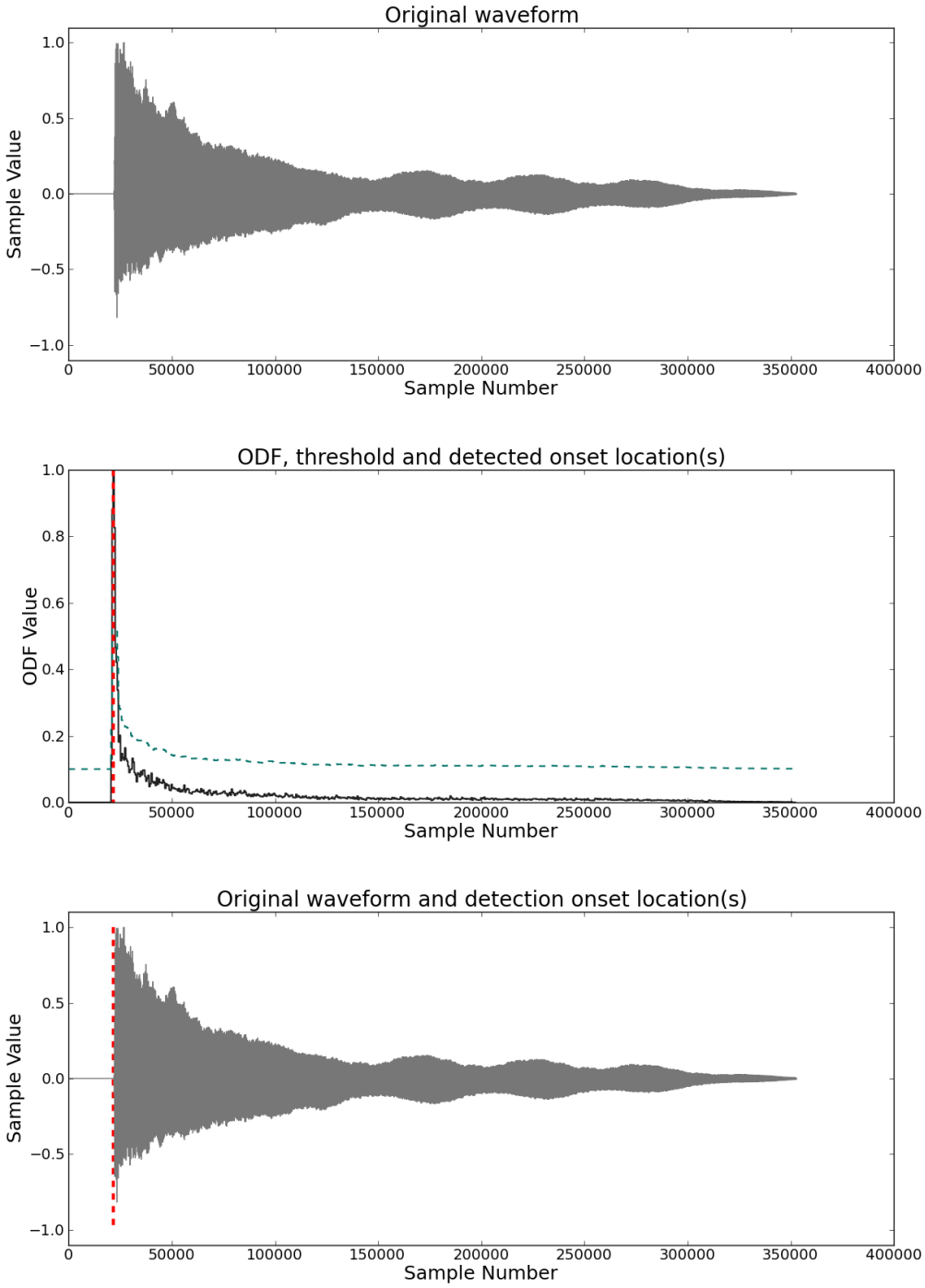

Figure 4.5: Figure produced by the onset detection example that is given in Listing 4.1. The first plot (top) is the original waveform (a piano sample). The second (middle) plot shows the normalised ODF (solid line), the dynamic threshold (horizontal dashed line) and the detected onsets (vertical dashed lines). The final plot (bottom) shows the onset locations plotted against the original waveform.

Python Module	Classes and Functions in Module
modal.db	num_onsets
	samples
modal.detectionfunctions	OnsetDetectionFunction
	EnergyODF
	SpectralDifferenceODF
	ComplexODF
	LinearPredictionODF
	LPEnergyODF
	LPSpectralDifferenceODF
	LPComplexODF
	PeakAmpDifferenceODF
	lpf
	moving_average
	savitzky_golay
	normalise
modal.detectionfunctions.lp	autocorrelation
	covariance
	burg
	predict
modal.detectionfunctions.mq	MQPeakDetection
	MQPartialTracking
	Peak
modal.onsetdetection	OnsetDetection
	RTOnsetDetection

modal.ui	plot_onsets plot_detection_function

Table 4.2: Modal Python modules.

Class or Function	Description
num_onsets*	Get the total number of reference onsets that are currently in the Modal database.
samples*	Get samples from the Modal database.
OnsetDetectionFunction	Base class for all onset detection functions, containing common variables and accessor methods.
EnergyODF	The energy ODF (Section 4.3.1).
SpectralDifferenceODF	The spectral difference ODF (Section 4.3.2).
ComplexODF	The complex ODF (Section 4.3.3)
LinearPredictionODF	Base class for all onset detection functions that use linear prediction.
LPEnergyODF	The energy LP ODF (Section 4.6.2).
LPSpectralDifferenceODF	The spectral difference LP ODF (Section 4.6.3).
LPComplexODF	The complex LP ODF (Section 4.6.4).
PeakAmpDifferenceODF	The peak amplitude difference ODF (Section 4.8.2).
autocorrelation*	Compute linear prediction coefficients using the autocorrelation method [58].
covariance*	Compute linear prediction coefficients using the covariance method [75].
burg	Compute linear prediction coefficients using the Burg method [62].

`predict`	Predict the next value of a signal using the supplied linear prediction coefficients.
`MQPeakDetection`	Peak detection using the MQ method.
`MQPartialTracking`	Partial tracking using the MQ method.
`Peak`	Class representing a spectral peak.
`OnsetDetection`	Detect onsets using a supplied ODF (non-real-time).
`RTOnsetDetection`	Detect onsets from a stream of ODF values (real-time).
`plot_onsets*`	Plot onset locations using Matplotlib.
`plot_detection_function*`	Plot ODFs using Matplotlib.

Table 4.3: Modal classes and functions (available from both C++ and Python).

4.5 Initial evaluation results

This section presents the detection accuracy and performance benchmarks for the energy ODF, spectral difference ODF and complex ODF.

4.5.1 Onset detection accuracy

The detection accuracy of the ODFs was measured by comparing the onsets detected using each method with the reference samples in the Modal database. To be marked as 'correctly detected', the onset must be located within 50 ms of a reference onset. Merged or double onsets were not penalised. The database currently contains 501 onsets from annotated sounds that are mainly monophonic and so this must be taken into consideration when viewing the results. The annotations were

also all made by one person, and while it has been shown in [72] that this is not ideal, the chosen detection window of 50 ms should compensate for some of the inevitable inconsistencies.

The results are summarised by three measurements that are common in the field of Information Retrieval [82]: the precision (P), the recall (R), and the F-measure (F) defined here as follows:

$$P = \frac{C}{C + f_{\mathrm{p}}} \tag{4.10}$$

$$R = \frac{C}{C + f_{\mathrm{n}}} \tag{4.11}$$

$$F = \frac{2PR}{P + R} \tag{4.12}$$

where C is the number of correctly detected onsets, f_{p} is the number of false positives (detected onsets with no matching reference onset), and f_{n} is the number of false negatives (reference onsets with no matching detected onset).

Every reference sample in the database was streamed one frame at a time to each ODF, with ODF values for each frame being passed immediately to a real-time peak-picking system (Algorithm 4.2). Dynamic thresholding was applied according to Equation 4.1, with $\lambda = 1.0$, $\alpha = 2.0$, and w in Equation 4.2 set to 0.05. A median window of seven previous values was used. These parameters were kept constant for each ODF.

The precision, recall and F-measure results for each ODF are given in Figures 4.6, 4.7 and 4.8, respectively. The precision value for the energy ODF is significantly lower than the values for the spectral difference and complex ODFs. The energy and spectral difference ODFs have similar recall values while the complex ODF fares slightly worse in this metric. For the overall F-measure however, the

spectral difference ODF is marginally ahead of the complex ODF, with the energy ODF having the lowest F-measure in our evaluation.

Accuracy results by sound type

It is useful to consider how the detection accuracy of each ODF changes with regards to changes in the characteristics of the input sound, as it may be possible to improve performance in domains where the class of input sound is restricted. The metadata for the reference samples in the Modal database includes a "type" classification. Each sample is assigned to one of four type categories: non-pitched percussive (NPP), pitched percussive (PP), pitched non-percussive (PNP) or mixed (M) (consisting of multiple sound sources that belong to different type categories).

Table 4.4 shows the F-measure results for each ODF, categorised according to sound type. The methods perform quite similarly when dealing with non-pitched percussive sound and mixed sound types. However, the spectral difference ODF and complex ODF show a noticeable improvement when analysing pitched percussive and pitched non-percussive sound types. In particular, there is a noticeable decrease in accuracy for the energy ODF when working with pitched non-percussive sounds (this category contains the majority of the "soft" onsets in the Modal reference set).

	NPP	PP	PNP	M
$\mathrm{ODF_E}$	0.61	0.67	0.49	0.66
$\mathrm{ODF_{SD}}$	0.62	0.72	0.62	0.64
$\mathrm{ODF_{CD}}$	0.60	0.70	0.60	0.65

Table 4.4: F-measure results for each ODF, categorised according to sound "type". The sound types are non-pitched percussive (NPP), pitched percussive (PP), pitched non-percussive (PNP) and mixed (M).

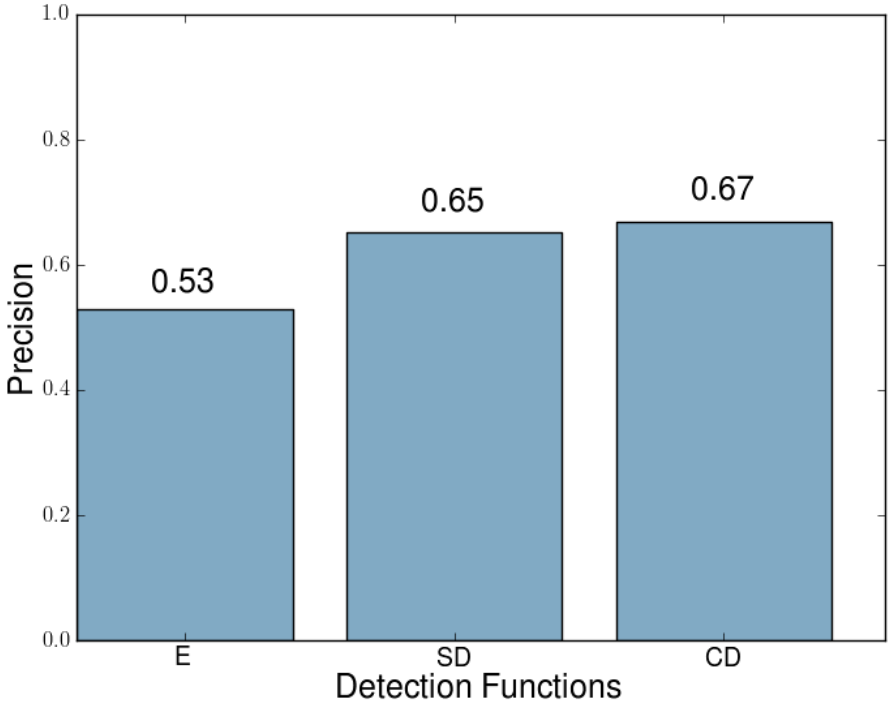

Figure 4.6: Precision results for the energy ODF, spectral difference ODF and the complex ODF.

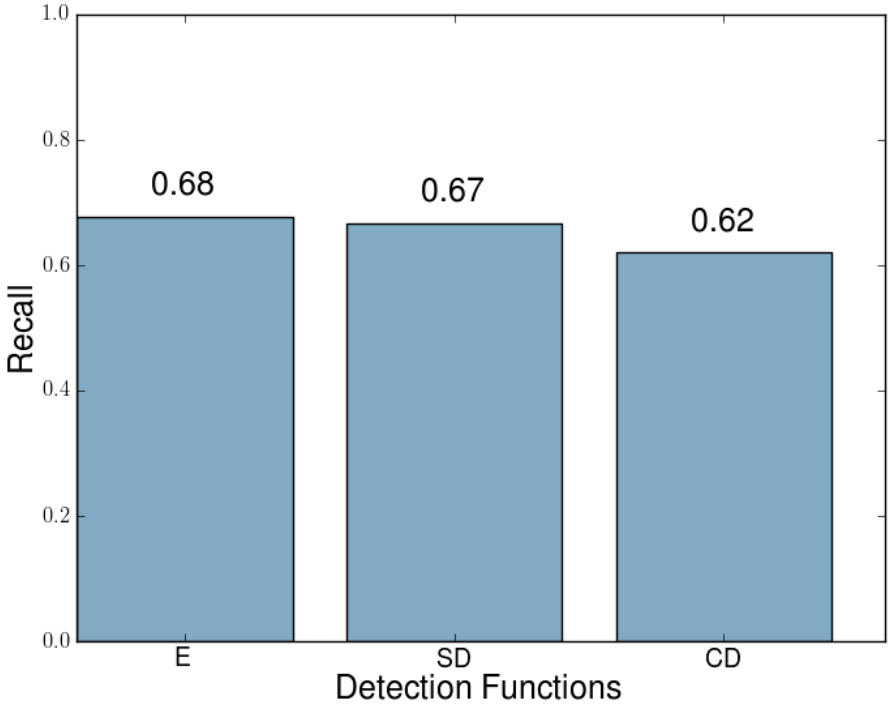

Figure 4.7: Recall results for the energy ODF, spectral difference ODF and the complex ODF.

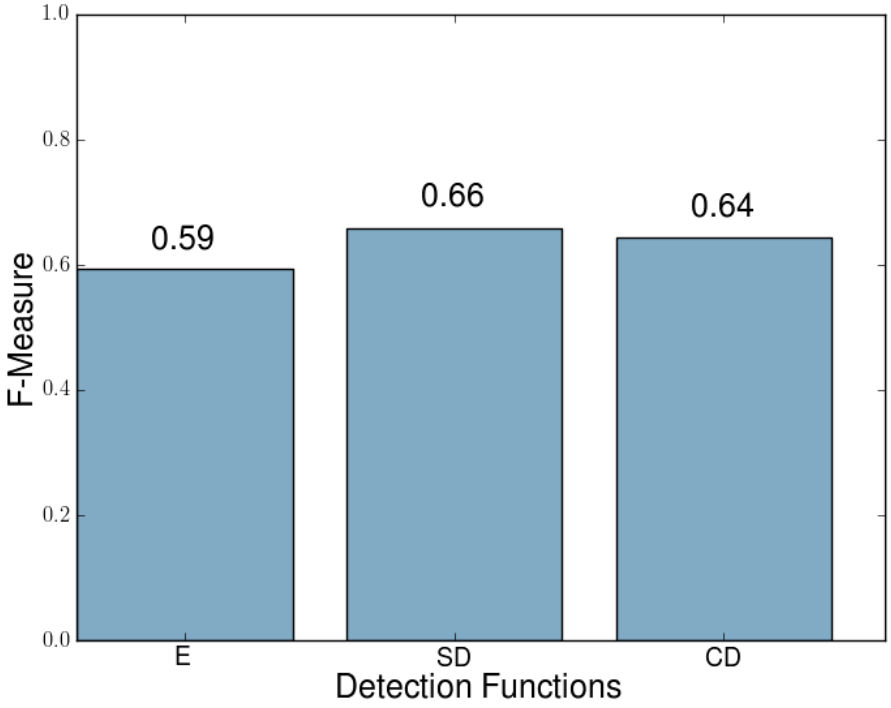

Figure 4.8: F-Measure results for the energy ODF, spectral difference ODF and the complex ODF.

4.5.2 Onset detection performance

In Table 4.5, we give the worst-case number of floating-point operations per second (FLOPS) required by each ODF to process real-time audio streams, based on our implementations in the Modal library. This analysis does not include data from the setup/initialisation periods of any of the algorithms, or data from the peak detection stage of the onset-detection system. As specified in Section 4.1, the audio frame size is 2,048 samples, the buffer size is 512 samples, and the sampling rate is 44.1 kHz.

These totals were calculated by counting the number of floating-point operations required by each ODF to process 1 frame of audio, where we define a floating-point operation to be an addition, subtraction, multiplication, division or assignment involving a floating-point number. As we have a buffer size of 512 samples measured

at 44.1 kHz, we have 86.133 frames of audio per second, and so the number of operations required by each ODF per frame of audio was multiplied by 86.133 to get the FLOPS total for the corresponding ODF.

To simplify the calculations, the following assumptions were made when calculating the totals:

- As we are using the real fast Fourier transform (FFT) computed using the FFTW3 library [38], the processing time required for a FFT is $2.5N \log_2(N)$ where N is the FFT size, as given in [37].
- The complexity of basic arithmetic functions in the C++ standard library such as $\sqrt{}$, $\cos$, $\sin$, and $\log$ is $O(M)$, where M is the number of digits of precision at which the function is to be evaluated.
- All integer operations can be ignored.
- All function call overheads can be ignored.

As Table 4.5 shows, the energy ODF (ODF_E) requires far less computation than any of the others. The spectral difference ODF is the second fastest, needing about half the number of operations that are required by the complex domain method.

	FLOPS
ODF_E	529,718
ODF_SD	7,587,542
ODF_CD	14,473,789

Table 4.5: Number of floating-point operations per second (FLOPS) required by the energy, spectral difference and complex ODFs to process real-time audio streams.

To give a more intuitive view of the algorithmic complexity, in Table 4.6, we also give the estimated real-time CPU usage for each ODF given as a percentage of the

maximum number of FLOPS that can be achieved by two different processors: an Intel Core 2 Duo and an Analog Devices ADSP-TS201S (TigerSHARC). The Core 2 Duo was chosen as it is a relatively common PC processor and will be a good indicator of the performance of the algorithms using current standard consumer hardware. The ADSP-TS201S was selected as it scores relatively well on the BDTI DSP Kernel Benchmarks [13] and so provides an indication of how the algorithms would perform on current embedded processors. The Core 2 Duo has a clock speed of 2.8 GHz, a 6 MB L2 cache and a bus speed of 1.07 GHz, providing a theoretical best-case performance of 22.4 GFLOPS [51]. The ADSP-TS201S has a clock speed of 600 MHz and a best-case performance of 3.6 GFLOPS [5]. Any value that is less than 100% in Table 4.6 shows that the ODF can be calculated in real-time on this processor.

	Core 2 Duo (%)	ADSP-TS201S (%)
$\mathrm{ODF_E}$	0.002	0.015
$\mathrm{ODF_{SD}}$	0.034	0.211
$\mathrm{ODF_{CD}}$	0.065	0.402

Table 4.6: Estimated real-time CPU usage for the energy, spectral difference and complex ODFs, shown as a percentage of the maximum number of FLOPS that can be achieved on two processors: an Intel Core 2 Duo and an Analog Devices ADSP-TS201S (TigerSHARC).

4.5.3 Initial evaluation conclusions

The F-measure results (Figure 4.8) for the three ODFs that are described in this section are lower than those given elsewhere in the literature, but this was expected as the peak-picking and thresholding stages are significantly more challenging when forced to meet our real-time constraints. The various parameter settings can have a large impact on overall performance [28]. We tried to select a parameter set that

gave a fair reflection on each algorithm, but it must be noted that every method can probably be improved by some parameter adjustments, especially if prior knowledge of the sound source is available.

The spectral difference method generally performs the best on our sample set, with an average F-measure of 0.66. It is considerably more computationally expensive than the energy ODF but is still well within the real-time limits of modern hardware. The next section describes our first attempt to improve upon these accuracy measurements.

4.6 Improving onset detection function estimations using linear prediction

The ODFs that are described in Section 4.3, and many of those found elsewhere in the literature [11], are trying to distinguish between the steady-state and transient regions of an audio signal by making predictions based on information about the most recent frame of audio and one or two preceding frames. In this section, we present methods that use the same basic signal information to the approaches described in Section 4.3. Instead of making predictions based on just one or two frames of these data however, an arbitrary number of previous values are combined with linear prediction (LP) to improve the accuracy of the estimates. The ODF is then calculated as the absolute value of the differences between the actual frame measurements and the new predictions. The ODF values are low when the prediction is accurate, but larger in regions of the signal that are more unpredictable, which should correspond with note onset locations.

This is not the first time that LP errors have been used to create an ODF. The authors in [71] describe a somewhat similar system in which an audio signal is first filtered into six non-overlapping sub-bands. The first five bands are then decimated by a factor of 20:1 before being passed to a LP error filter, while just the amplitude envelope is taken from the sixth band (everything above the note B7 which is 3,951 kHz). Their ODF is the sum of the five LP error signals and the amplitude envelope from the sixth band.

Our approach differs in a number of ways. Here we show that LP can be used to improve the detection accuracy of the three ODFs that are described in Section 4.3[9]. As this approach involves making predictions about the time-varying changes in signal features (energy, spectral difference and complex phasor positions) rather than in the signal itself, the same technique could be applied to many existing ODFs from the literature. It can therefore be viewed as an additional post-processing step that can potentially improve the detection accuracy of many existing ODFs. Our algorithms are suitable for real-time use, and the results were compiled from real-time data. In contrast, the results given in [71] are based on off-line processing, and include an initial pre-processing step to normalise the input audio files, and so it is not clear how well this method performs in a real-time situation.

The LP process that is used in this chapter is described in Section 4.6.1. In Sections 4.6.2, 4.6.3 and 4.6.4, we show that this can be used to create new ODFs based on the energy, spectral difference and complex domain ODFs respectively.

[9]Onset detection results are provided in at the end of this section

4.6.1 Linear prediction

In the LP model, also known as the autoregressive model, the current input sample $x(n)$ is estimated by a weighted combination of the past values of the signal. The predicted value $\hat{x}(n)$ is computed by FIR filtering according to Equation 4.13, where O is the order of the LP model and a_k are the prediction coefficients.

$$\hat{x}(n) = \sum_{k=1}^{O} a_k x(n-k) \tag{4.13}$$

The challenge is then to calculate the LP coefficients. There are a number of methods given in the literature, the most widespread among which are the autocorrelation method [58], covariance method [75] and the Burg method [62]. Each of the three methods was evaluated, but the Burg method was selected as it produced the most accurate and consistent results. Like the autocorrelation method, it has a minimum phase, and like the covariance method it estimates the coefficients on a finite support [62]. It can also be efficiently implemented in real time [58].

The Burg algorithm

The LP error is the difference between the predicted and the actual values.

$$e(n) = x(n) - \hat{x}(n) \tag{4.14}$$

The Burg algorithm minimises the average of the forward prediction error $f_m(n)$ and the backward prediction error $b_m(n)$. The initial (order 0) forward and backward errors are given by

$$f_0(n) = x(n) \tag{4.15}$$

$$b_0(n) = x(n) \tag{4.16}$$

over the interval $n = 0, \ldots, N-1$, where N is the block length. For the remaining $m = 1, \ldots, p$, the m-th coefficient is calculated from

$$k_m = \frac{-2\sum_{n=m}^{N-1}[f_{m-1}(n)b_{m-1}(n-1)]}{\sum_{n=m}^{N-1}[f_{m-1}^2(n) + b_{m-1}^2(n-1)]} \tag{4.17}$$

and then the forward and backward prediction errors are recursively calculated from

$$f_m(n) = f_{m-1}(n) - k_m b_{m-1}(n-1) \tag{4.18}$$

for $n = m+1, \ldots, N-1$, and

$$b_m(n) = b_{m-1}(n-1) - k_m f_{m-1}(n) \tag{4.19}$$

for $n = m, \ldots, N-1$, respectively. Pseudocode for this process is given in Figure 4.9, taken from [62].

$f \longleftarrow x$;
$b \longleftarrow x$;
$a \longleftarrow x$;
for $m \leftarrow 0$ **to** $p-1$ **do**
  $fp \longleftarrow f$ without its first element ;
  $bp \longleftarrow b$ without its last element ;
  $k \longleftarrow -2bp \cdot fp/(fp \cdot fp + fp \cdot fp)$;
  $f \longleftarrow fp + k \cdot bp$;
  $b \longleftarrow bp + k \cdot fp$;
  $a \longleftarrow (a[0], a[1], \ldots, a[m], 0) + k(0, a[m], a[m-1], \ldots, a[0])$;

Figure 4.9: The Burg method.

4.6.2 Energy LP ODF

The energy ODF is derived from the absolute value of the energy difference between two frames. This can be viewed as using the energy value of the first frame as an estimate of the energy of the second, with the difference being the prediction error. We can therefore try and improve this estimate by taking the energy values from the previous O frames, using them to create a set of LP coefficients, and then using these coefficients to compute an estimate of the signal energy for the current frame. The O previous frame energy values can be represented using the sequence

$$E(l-1), E(l-2), \ldots, E(l-O).$$

Using (4.15) – (4.19), O coefficients are calculated based on this sequence, and then a one-sample prediction is made using (4.13). Hence, for each frame, the energy LP ODF (ODF_{ELP}) is given by

$$\text{ODF}_{\text{ELP}}(l) = |E(l) - P_E(l)| \tag{4.20}$$

where $P_E(n)$ is the predicted energy value for frame l.

4.6.3 Spectral difference LP ODF

A similar methodology can be applied to the spectral difference and complex domain ODFs. The spectral difference ODF is formed from the absolute value of the magnitude differences between corresponding bins in adjacent frames. This can therefore be viewed as a prediction that the magnitude in a given bin will remain constant between adjacent frames, with the magnitude difference being the predic-

tion error. In the spectral difference LP ODF (ODF_{SDLP}), the predicted magnitude value for each of the k bins in frame l is calculated by taking the magnitude values from the corresponding bins in the previous O frames, using them to find O LP coefficients then filtering the result with (4.13). Hence, for each k in l the magnitude prediction coefficients are formed using (4.15) – (4.19) on the sequence

$$|X_{l-1}(k)|, |X_{l-2}(k)|, \ldots, |X_{l-O}(k)|.$$

If $P_{\text{SD}}(k, l)$ is the predicted spectral difference for bin k in l then

$$\text{ODF}_{\text{SDLP}}(l) = \sum_{k=0}^{N/2} ||X_l(k)| - P_{\text{SD}}(k, l)|. \tag{4.21}$$

As is shown in Section 4.7, this is a significant amount of extra computation per frame compared with the ODF_{SD} given by (4.5). However if the chosen LP model order O is low enough then the algorithm can still be run in a real-time context using commodity hardware. We found that an order of 5 was enough to significantly improve the detection accuracy while still comfortably meeting the real-time processing requirements.

4.6.4 Complex domain LP ODF

The complex domain ODF works by measuring the Euclidean distance between the predicted and the actual complex phasors for a given bin. There are a number of different ways by which LP could be applied in an attempt to improve this estimate. The bin magnitudes and phases could be predicted separately, based on their values over the previous O frames, and then combined to form an estimated phasor value

for the current frame. Another possibility would be to only use LP to estimate either the magnitude or the phase parameters, but not both.

However we found that the biggest improvement came from using LP to estimate the value of the Euclidean distance that separates the complex phasors for a given bin between consecutive frames. Hence, for each bin k in frame l, the complex distances between the k-th bin in each of the last O frames are used to calculate the LP coefficients. If $R_l(k)$ is the magnitude of the k-th bin in frame l, and $\phi_l(k)$ is the phase of the bin, then the distance between the k-th bins in frames l and $l-1$ is give by Equation 4.22.

$$\Gamma_l(k) = \sqrt{R_l(k)^2 + R_{l-1}(k)^2 - 2R_l(k)R_{l-1}(k)\cos(\phi_l(k) - \phi_{l-1}(k))} \quad (4.22)$$

LP coefficients are formed from the values

$$\Gamma_{l-1}(k), \Gamma_{l-2}(k), \ldots, \Gamma_{l-O}(k)$$

using (4.15) – (4.19), and predictions $P_{\mathrm{CD}}(k,l)$ are calculated using (4.13). The complex domain LP ODF ($\mathrm{ODF}_{\mathrm{CDLP}}$) is then given by Equation 4.23.

$$\mathrm{ODF}_{\mathrm{CDLP}}(l) = \sum_{k=0}^{N/2} |\Gamma_l(k) - P_{\mathrm{CD}}(k,l)| \quad (4.23)$$

4.7 Evaluation of linear prediction ODFs

This section presents the detection accuracy and performance benchmarks for the energy LP ODF, spectral difference LP ODF and complex LP ODF, and compares them with the results for the ODFs from the literature. The results were obtained

using the same process that was described in Section 4.5. The only change is the addition of a LP model order parameter for the energy LP, spectral difference LP and complex LP ODFs, which was set to 5.

4.7.1 Onset detection accuracy

The precision, recall and F-measure results for each ODF are given in Figures 4.10, 4.11 and 4.12, respectively. The blue bars give the results for the methods from the literature, while the results for our new LP methods are given by the brown bars. The addition of LP noticeably improves the precision value for each ODF to which it is applied. LP has improved the recall value for the energy ODF but has made the recall result of the spectral difference and complex domain ODFs slightly worse. However for the general F-measure result, all ODFs are improved by the addition of LP, with the spectral difference LP ODF performing best of all.

Accuracy results by sound type

Table 4.7 shows the F-measure results for each ODF, categorised according to sound type. The addition of linear prediction has generally improved the detection accuracy on all sound types for all ODFs, with the only notable exception being the degradation in the F-measure result for the complex ODF when analysing non-pitched percussive sounds. The most significant accuracy gains are in the detection of onsets in pitched percussive sounds, where the new linear prediction methods all perform well. There are also significant improvements in the accuracy of onset detection in signals containing mixed sound types for the spectral difference LP ODF and complex domain LP ODF.

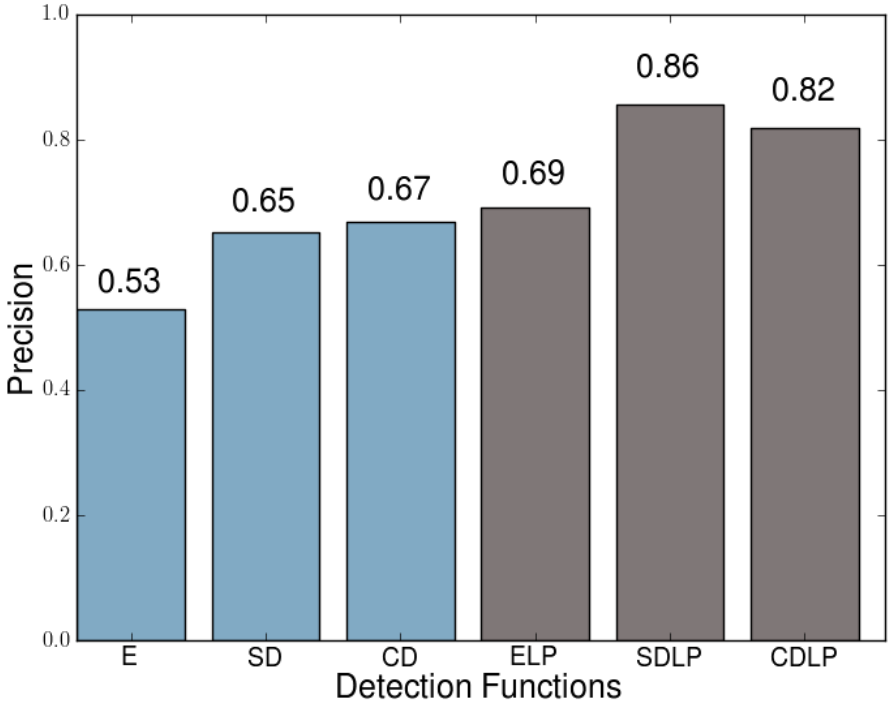

Figure 4.10: Precision results for the energy ODF, spectral difference ODF, complex ODF and their corresponding LP-based methods.

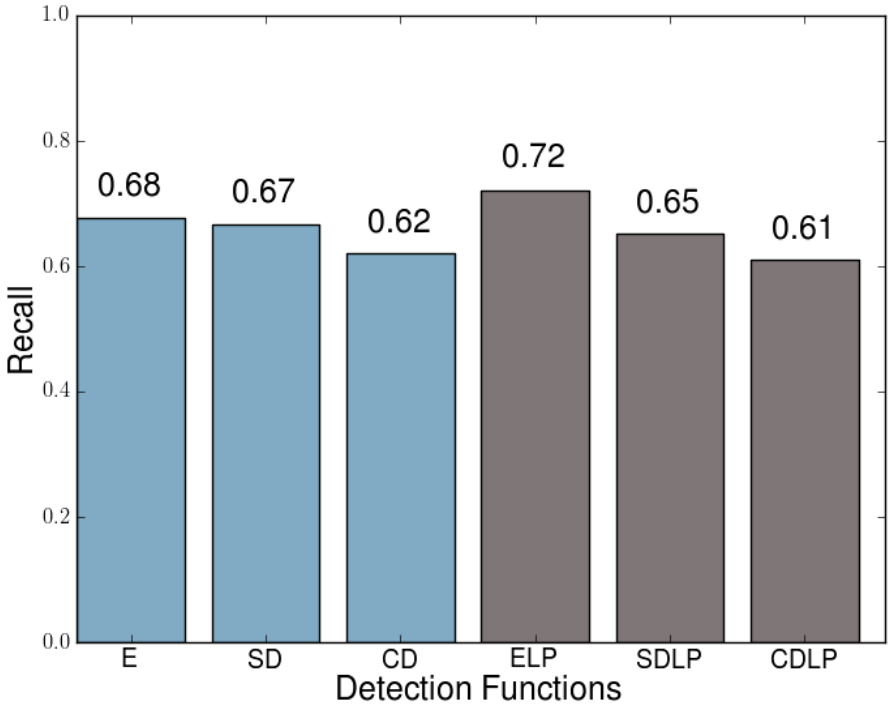

Figure 4.11: Recall results for the energy ODF, spectral difference ODF, complex ODF and their corresponding LP-based methods.

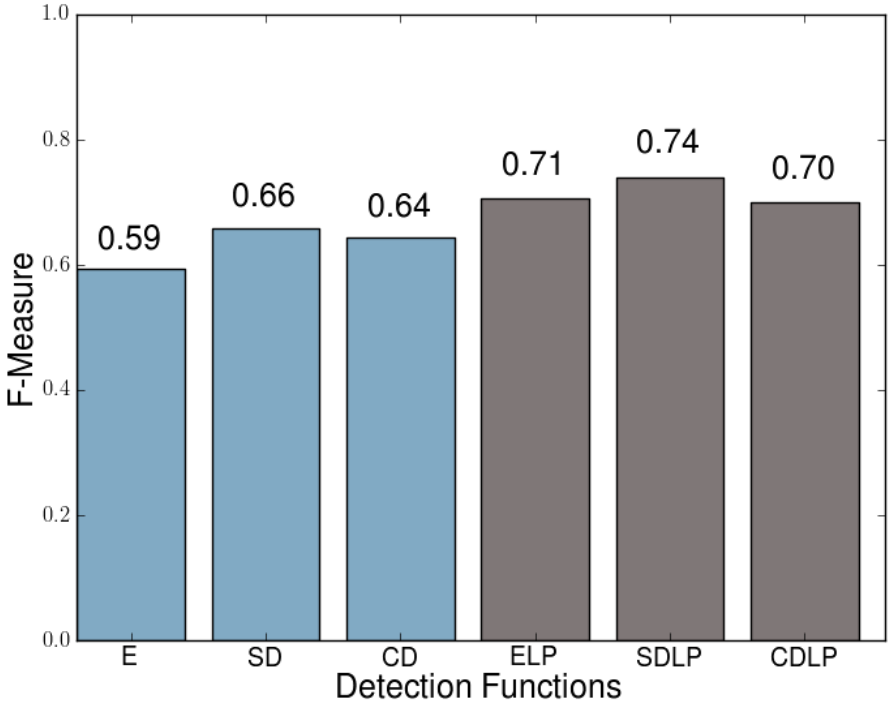

Figure 4.12: F-Measure results for the energy ODF, spectral difference ODF, complex ODF and their corresponding LP-based methods.

	NPP	PP	PNP	M
ODF_{E}	0.61	0.67	0.49	0.66
ODF_{SD}	0.62	0.72	0.62	0.64
ODF_{CD}	0.60	0.70	0.60	0.65
ODF_{ELP}	0.69	0.90	0.59	0.65
ODF_{SDLP}	0.63	0.91	0.62	0.79
ODF_{CDLP}	0.55	0.89	0.59	0.76

Table 4.7: F-measure results for each ODF, categorised according to sound "type". The sound types are non-pitched percussive (NPP), pitched percussive (PP), pitched non-percussive (PNP) and mixed (M).

4.7.2 Onset detection performance

The worst-case number of FLOPS that are required by our new LP-based ODFs to process audio streams are given in Table 4.8 and the estimated CPU usage is given in Table 4.9. The LP-based ODFs are all significantly slower than their coun-

terparts. In particular, the addition of LP to the spectral difference and complex domain ODFs makes them significantly more expensive computationally than any other technique. However, even the most computationally expensive algorithm can run with an estimated real-time CPU usage of just over 6% on the ADSP-TS201S (TigerSHARC) processor, and so they are still more than capable in respect of real-time performance. The energy LP ODF in particular is extremely cheap computationally, and yet has relatively good detection accuracy for this sample set. The nature of the sample set must also be taken into account however, as evidently the heavy bias towards monophonic sounds is reflected by the surprisingly strong performance of the energy-based methods.

Whether the improvement in onset detection accuracy is worth the increased computational complexity that the LP-based techniques introduce will largely depend on the musical context. If the computation that must be performed in addition to the onset detection system is not significant, or the perceived cost of onset detection errors is suitably high, then the LP-based methods can be extremely useful. However, as our real-time sinusoidal synthesis by analysis system will already be computationally expensive, the cost of the spectral difference LP ODF and complex domain LP ODF was deemed to be too great. Section 4.8 describes a way to improve upon the accuracy of the literature methods without significantly increasing the computational complexity.

	FLOPS
$\mathrm{ODF_E}$	529,718
$\mathrm{ODF_{SD}}$	7,587,542
$\mathrm{ODF_{CD}}$	14,473,789
$\mathrm{ODF_{ELP}}$	734,370
$\mathrm{ODF_{SDLP}}$	217,179,364
$\mathrm{ODF_{CDLP}}$	217,709,168

Table 4.8: Number of floating-point operations per second (FLOPS) required by the energy ODF, spectral difference ODF, complex ODF and their corresponding LP-based ODFs in order to process real-time audio streams.

	Core 2 Duo (%)	ADSP-TS201S (%)
$\mathrm{ODF_E}$	0.002	0.015
$\mathrm{ODF_{SD}}$	0.034	0.211
$\mathrm{ODF_{CD}}$	0.065	0.402
$\mathrm{ODF_{ELP}}$	0.003	0.020
$\mathrm{ODF_{SDLP}}$	0.970	6.033
$\mathrm{ODF_{CDLP}}$	0.972	6.047

Table 4.9: Estimated real-time CPU usage for the energy ODF, spectral difference ODF, complex ODF and their corresponding LP-based ODFs, shown as a percentage of the maximum number of FLOPS that can be achieved on two processors: an Intel Core 2 Duo and an Analog Devices ADSP-TS201S (TigerSHARC).

4.8 Combining onset detection with real-time sinusoidal modelling

Section 4.6 described a method than can improve the detection accuracy of three common ODFs from the literature by using linear prediction to enhance estimates of the frame-by-frame evolution of audio signal properties. This improvement in detection accuracy comes at the expense of much greater computational cost. In some contexts this accuracy improvement would justify the additional processing requirements. However, as this onset detection system is part of a larger real-time sinusoidal synthesis by analysis framework, we investigated to see if we could improve upon the performance of the ODFs from the literature without significantly increasing the computational complexity.

In this section, a novel ODF is presented that has significantly better real-time performance than the LP-based spectral methods. It works by analysing the partials formed during the sinusoidal modelling process and measuring the amplitude differences between peaks in consecutive frames. This makes it particularly easy to integrate with our real-time sinusoidal modelling system, and incurs a relatively low additional computational cost as the sinusoidal analysis stage must be performed anyway. Some existing approaches to onset detection that use sinusoidal modelling are described in Section 4.8.1, followed by an in depth explanation of our new ODF in Section 4.8.2.

4.8.1 Existing approaches to onset detection using sinusoidal modelling

Although it was originally designed to model transient components from musical signals, the system described in [121] (discussed in Section 2.6.2) could also be adopted to detect note onsets. The authors show that transient signals in the time domain can be mapped onto sinusoidal signals in a frequency domain, in this case by using the DCT. Roughly speaking, the DCT of a transient time-domain signal produces a signal with a frequency that depends only on the time shift of the transient. This information could then be used to identify when the onset occurred. However, it is not suitable for real-time applications as it requires a DCT frame size that makes the transients appear as a small entity, with a frame duration of about 1 second recommended. As the onset detection system is intended to be used in a real-time musical performance context this latency value is unacceptably high.

Another system that combines sinusoidal modelling and onset detection was introduced by Levine [73]. It creates an ODF that is a combination of two energy measurements. The first is the energy in the audio signal over a 512 sample frame. If the energy of the current frame is larger than that of a given number of previous frames, then the current frame is a candidate for being an onset location. A multi-resolution sinusoidal model is then applied to the signal in order to isolate the harmonic component of the sound. This differs from the sinusoidal modelling implementation described above in that the audio signal is first split into five octave spaced frequency bands. Currently, only the lower three are used, while the upper two (frequencies above about 5 kHz) are discarded. Each band is then analysed using different window lengths, allowing for more frequency resolution in the lower

band at the expense of worse time resolution. Sinusoidal amplitude, frequency and phase parameters are estimated separately for each band, and linked together to form partials. An additional post-processing step is then applied, removing any partials that have an average amplitude that is less than an adaptive psychoacoustic masking threshold, and removing any partials that are less than 46 ms in duration.

As it stands, it is unclear whether or not the system described in [73] is suitable for use as a real-time onset detector. The stipulation that all sinusoidal partials must be at least 46 ms in duration implies that there must be a minimum latency of 46 ms in the sinusoidal modelling process, putting it very close to our 50 ms limit. If this ODF is used in the onset-detection system that is described in Section 4.2, the additional 11.6 ms of latency incurred by the peak-detection stage would put the total latency outside this 50-ms window. However, their method uses a rising edge detector instead of looking for peaks, and so it may still meet our real-time requirements. Although as it was designed as part of a larger system that was primarily intended to encode audio for compression, no onset-detection accuracy or performance results are given by the authors.

In contrast, the ODF that is presented in Section 4.8.2 was designed specifically as a real-time onset detector, and so has a latency of just two buffer sizes (23.2 ms in our implementation). As we show in Section 4.9, it compares favourably to leading approaches from the literature in terms of computational efficiency, and it is also more accurate than the reviewed methods.

4.8.2 The peak amplitude difference ODF

Implicit in the sinusoidal model is the assumption that a quasi-harmonic musical signal can be well represented as a sum of sinusoids. These sinusoids should evolve slowly in time, and should therefore be accurately characterised by the partials detected by the sinusoidal modelling process. It follows then that during the steady state region of a note, the absolute values of the frame-to-frame differences in the sinusoidal peak amplitudes and frequencies should be quite low. In comparison, transient regions at note onset locations should show considerably more frame-by-frame variation in both peak frequency and amplitude values. This is due to two main factors:

1. Many musical notes have an increase in signal energy during their attack regions, corresponding to a physical excitation being applied, which increases the amplitude of the detected sinusoidal components.

2. As transients are by definition less predictable and less harmonic, the basic premise of the sinusoidal model breaks down in these regions. This can result in spurious spectral peaks being detected in these regions that are not part of any underlying harmonic component. Often they will remain "unmatched" in the sinusoidal modelling process, and so do not form long-duration partials. Alternatively, if they are (incorrectly) matched, then it can result in relatively large amplitude and/or frequency deviations in the resulting partial. In either case, the difference between the parameters of the noisy peak and the parameters of any peaks before and after it in a partial will often differ significantly.

Both these factors should lead to larger frame-to-frame sinusoidal peak amplitude differences in transient regions than in steady-state regions. This can therefore be

used to create an ODF by measuring the differences in peak amplitude values between consecutive frames, as illustrated by Figure 4.13.

The sinusoidal modelling algorithm that we used to create the peak amplitude difference ODF is very close to the MQ method [80], but has a number of additions to the peak-detection process. Firstly, the number of peaks per frame can be limited to M_p, reducing the computation required for the partial-tracking stage [67, 68]. If the number of detected peaks $N_p > M_p$, then the M_p largest amplitude peaks will be selected. In order to allow for consistent evaluation with the other frequency domain ODFs that are described in this Chapter, the input frame size is kept constant during the analysis process (2,048 samples). The partial-tracking process is identical to the one given in [80]. As this partial-tracking algorithm has a delay of one buffer size, this ODF has an additional latency of 512 samples, bringing the total detection latency to 1,536 samples or 34.8 ms when sampled at 44.1 kHz[10].

For a given frame l, let $P_l(k)$ be the peak amplitude of the k-th partial. The peak amplitude difference ODF ($\mathrm{ODF}_{\mathrm{PAD}}$) is given by Equation 4.24.

$$\mathrm{ODF}_{\mathrm{PAD}}(l) = \sum_{k=0}^{M_p} |P_l(k) - P_{l-1}(k)| \tag{4.24}$$

In the steady-state, frame-to-frame peak amplitude differences for matched peaks should be relatively low. A lower number of peak matching errors is expected in this region as the partial tracking process is significantly easier during the steady-state section of a note than in transient regions. At note onsets, matched peaks should have larger amplitude deviations due to more energy in the signal, and there should also be more unmatched or incorrectly matched noisy peaks, increasing the ODF

[10]This includes the one frame delay that is introduced from the ODF peak detection process that is described in Section 4.2.

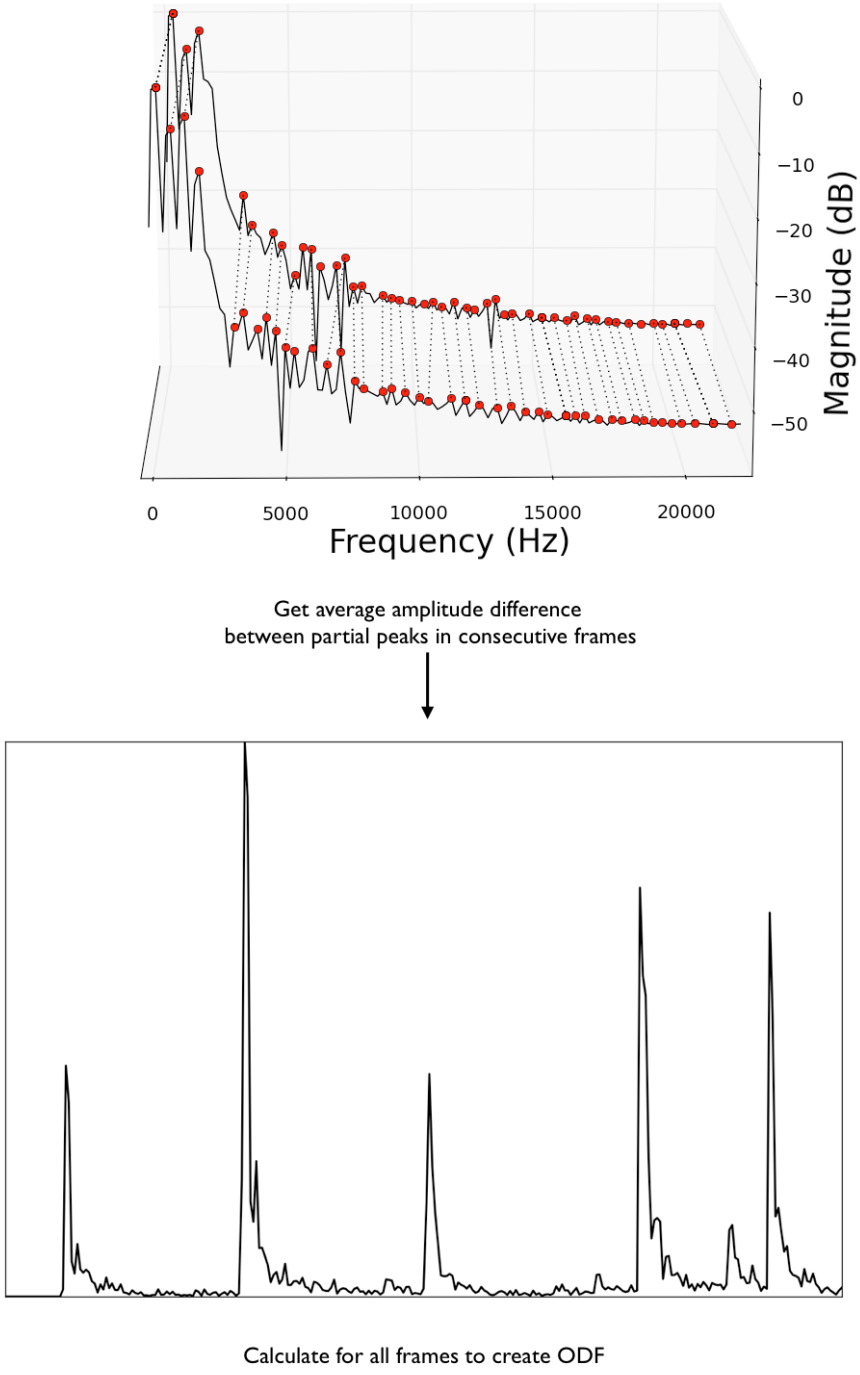

Figure 4.13: The peak amplitude difference ODF. It is based on the premise that the differences between the amplitude values of matched spectral peaks in consecutive frames will be larger at note onset locations.

value. Similarly to the process that is described in [80], unmatched peaks for a frame are taken to be the start of a partial and so the amplitude difference is equal to the amplitude of the peak $P_l(k)$.

4.9 Final onset detection results

This section presents the detection accuracy and performance benchmarks for all of the ODFs that are discussed in this chapter. The process for compiling the results is described in Section 4.5. A model order of 5 was used for the energy LP, spectral difference LP and complex LP ODFs. The number of peaks in the $\mathrm{ODF}_{\mathrm{PAD}}$ was limited to 20.

4.9.1 Onset detection accuracy

Figure 4.14 shows that the precision values for all our methods are higher than the methods from the literature. The addition of LP noticeably improves each ODF to which it is applied. The precision values for the peak amplitude difference method is better than the literature methods and the energy with LP method, but worse than the two spectral-based LP methods. The recall results for each ODF are given in Figure 4.15. The peak amplitude difference method has a greater recall than all of the literature methods and is only second to the energy with LP ODF. Figure 4.16 gives the F-measure for each ODF. All of our proposed methods are shown to perform better than the methods from the literature. The spectral difference with LP ODF has the best detection accuracy, while the energy with LP, complex domain with LP and peak amplitude difference methods are all closely matched.

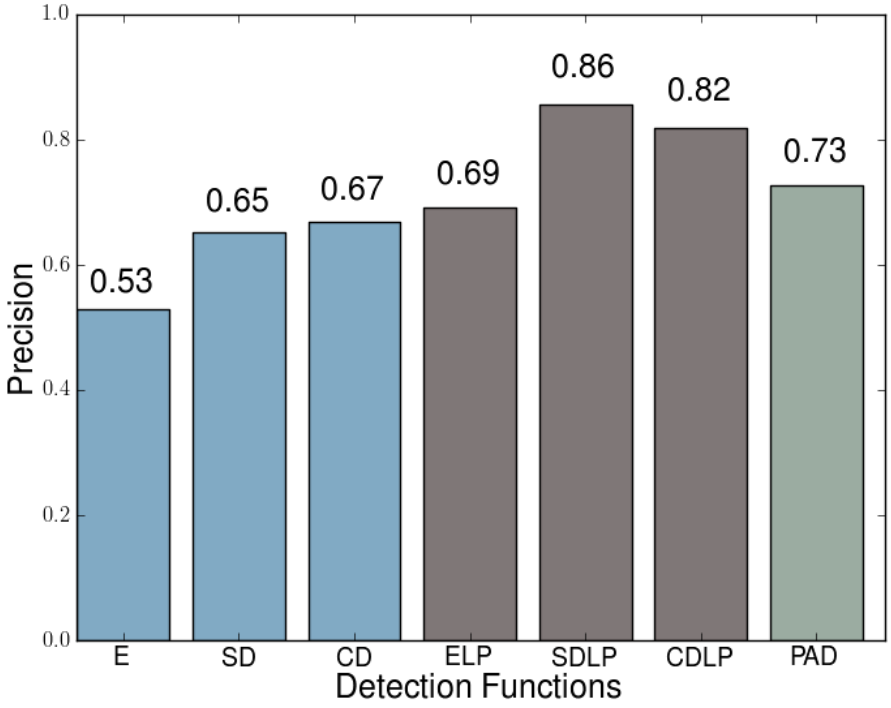

Figure 4.14: Precision results for all ODFs that are described in this chapter.

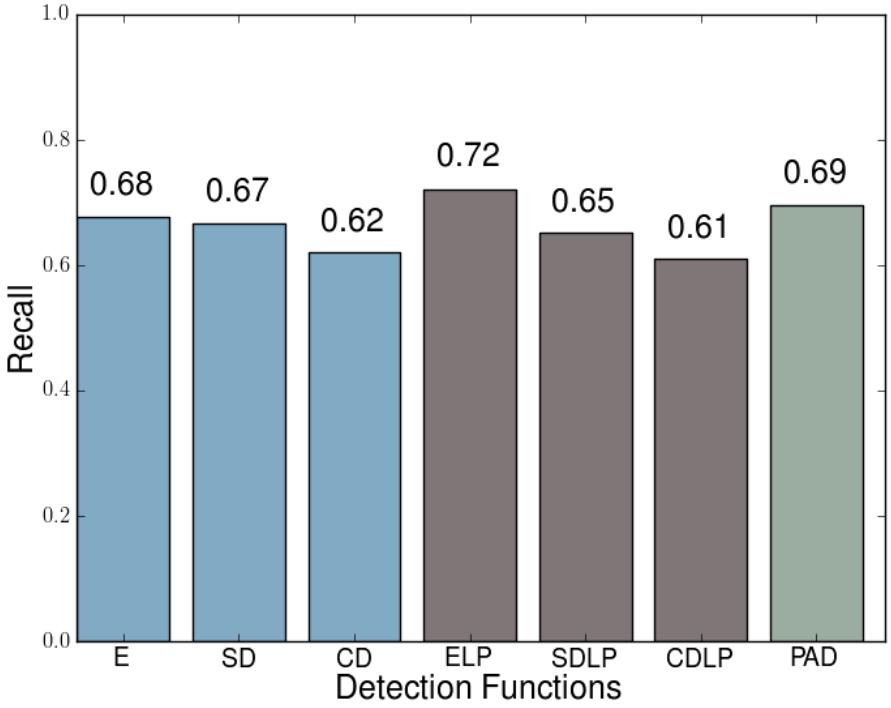

Figure 4.15: Recall results for all ODFs that are described in this chapter.

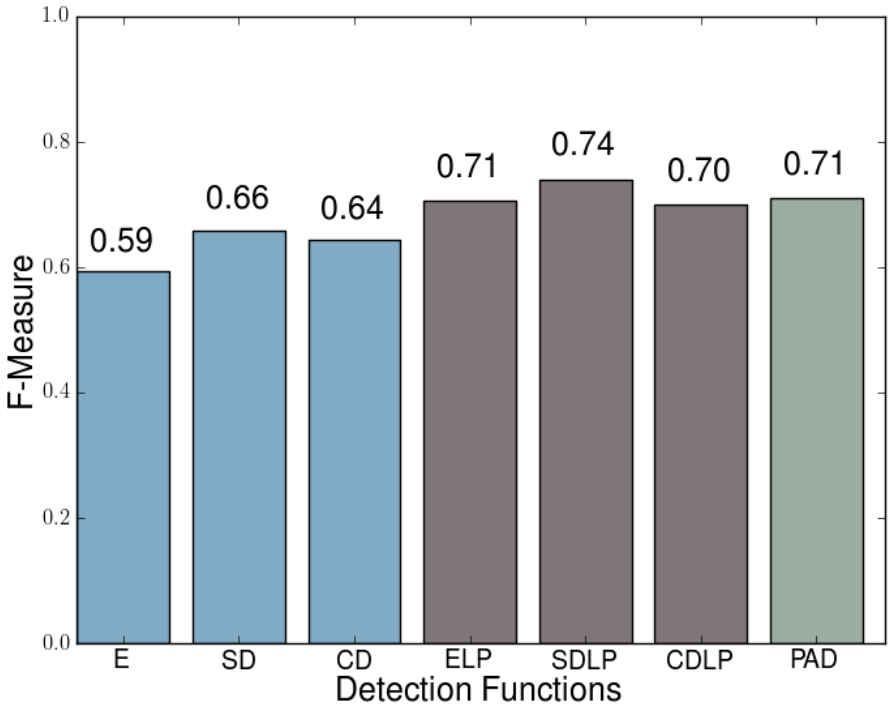

Figure 4.16: F-measure results for all ODFs that are described in this chapter.

Accuracy results by sound type

Table 4.10 shows the F-measure results for all of the ODFs, categorised according to sound type. The peak amplitude difference method performs better than the literature methods for all sound types. It is also marginally more accurate than the linear prediction methods for detecting onsets in pitched non-percussive sounds, but the linear prediction methods are notably stronger for pitched percussive and mixed sound types.

	NPP	PP	PNP	M
$\mathrm{ODF_E}$	0.61	0.67	0.49	0.66
$\mathrm{ODF_{SD}}$	0.62	0.72	0.62	0.64
$\mathrm{ODF_{CD}}$	0.60	0.70	0.60	0.65
$\mathrm{ODF_{ELP}}$	0.69	0.90	0.59	0.65
$\mathrm{ODF_{SDLP}}$	0.63	0.91	0.62	0.79
$\mathrm{ODF_{CDLP}}$	0.55	0.89	0.59	0.76
$\mathrm{ODF_{PAD}}$	0.65	0.83	0.63	0.70

Table 4.10: F-measure results for each ODF, categorised according to sound "type". The sound types are non-pitched percussive (NPP), pitched percussive (PP), pitched non-percussive (PNP) and mixed (M).

4.9.2 Onset detection performance

The worst-case number of FLOPS that each ODF requires to process audio streams are given in Table 4.11 and the estimated CPU usage is given in Table 4.12. The worst-case requirements for the peak amplitude difference method are still relatively close to the spectral difference ODF and noticeably quicker than the complex domain ODF.

	FLOPS
$\mathrm{ODF_E}$	529,718
$\mathrm{ODF_{SD}}$	7,587,542
$\mathrm{ODF_{CD}}$	14,473,789
$\mathrm{ODF_{ELP}}$	734,370
$\mathrm{ODF_{SDLP}}$	217,179,364
$\mathrm{ODF_{CDLP}}$	217,709,168
$\mathrm{ODF_{PAD}}$	9,555,940

Table 4.11: Number of floating-point operations per second (FLOPS) required by each ODF in order to process real-time audio streams.

	Core 2 Duo (%)	ADSP-TS201S (%)
$\mathrm{ODF_E}$	0.002	0.015
$\mathrm{ODF_{SD}}$	0.034	0.211
$\mathrm{ODF_{CD}}$	0.065	0.402
$\mathrm{ODF_{ELP}}$	0.003	0.020
$\mathrm{ODF_{SDLP}}$	0.970	6.033
$\mathrm{ODF_{CDLP}}$	0.972	6.047
$\mathrm{ODF_{PAD}}$	0.043	0.265

Table 4.12: Estimated real-time CPU usage for each ODF, shown as a percentage of the maximum number of FLOPS that can be achieved on two processors: an Intel Core 2 Duo and an Analog Devices ADSP-TS201S (TigerSHARC).

4.10 Conclusions

Sinusoidal models are not well suited to synthesising transient signal components. The transients at note onsets in particular are very important to the perception of timbre, and so it is therefore desirable to be able to accurately identify these regions and to adapt the model in order to be able to reproduce them with a high degree of fidelity. An important first step in transient location is note onset detection. However, accurate onset detection is also of great benefit in other areas of real-time sound analysis and synthesis such as score followers [93] and beat-synchronous analysis systems [113, 105].

This chapter introduced a general approach to onset detection. The first stage in the process is a data reduction step, which transforms the audio signal into an onset detection function (ODF). Onsets are then located by searching for local maxima in this ODF. Two new approaches to real-time musical onset detection were introduced, the first using linear prediction and the second using sinusoidal modelling. A new open source software library and sample database called Modal was created in order to evaluate these techniques. We compared these approaches to some of the leading real-time musical onset-detection algorithms from the literature, and found that they can offer either improved accuracy, computational efficiency, or both. It is recognised that onset-detection results are very context sensitive, and so without a more extensive sample set it is hard to make completely conclusive comparisons to other methods. However, our software and our sample database are both released under open source licences and are freely redistributable, so hopefully other researchers in the field will contribute.

Choosing a real-time ODF remains a complex issue and depends on the nature

of the input sound, the available processing power and the penalties that will be experienced for producing false negatives and false positives. However, some recommendations can be made based on the results in this chapter. For our sample set the spectral difference with LP method produced the most accurate results, and so if computational complexity is not an issue then this would be a good choice. On the other hand, if low complexity is an important requirement then the energy LP ODF is an attractive option. It produced accurate results at a fraction of the computational cost of some of the established methods.

The peak amplitude difference ODF is also noteworthy and should prove to be useful in areas such as real-time sound synthesis by analysis. It was shown to provide more accurate results than the well-established complex domain method with noticeably lower computation requirements, and as it integrates seamlessly with the sinusoidal modelling process, it can be added to the existing sinusoidal modelling systems at very little cost.

Chapter 5

Note segmentation

Sinusoidal models are based on the underlying premise that a musical sound consists of components that vary slowly in time. It was noted in Chapter 4 that this assumption can lead to synthesis artifacts being produced when short-lived or transient components are present in the input signal. During synthesis, analysis parameters are smoothly interpolated between consecutive frames which can cause transient components to become diffused. This problem is significant as in monophonic musical sounds, transient locations often correspond with the attack section of notes[1], and it has been shown that this region is important to our perception of timbre [42, 45].

To improve the quality of the synthesis of attack transients it is first necessary to be able to accurately identify them. The automatic identification of note attack regions can be broken up into two distinct steps:

[1]The word "note" is used in the widest sense in this thesis, referring to any single coherent musical sound or sonic object.

1. Find note onset locations.

2. Calculate the duration of the transient region following a given note onset.

Chapter 4 examined the first step in detail, culminating in the evaluation of seven different real-time onset detection systems. This chapter discusses the second step, describing a method for locating attack transients in real-time.

However, the attack section of a note is not the only temporal region that must be considered when designing sound transformation tools. It may be desirable to only apply transformations such as time-scaling to the steady state of notes for example, leaving the attack and decay regions unmodified. Sound morphing can be described as a technique which aims to produce a sound timbre which lies somewhere perceptually between two (or more) existing sounds. Caetano and Rodet have shown that considering the temporal evolution of the sound sources can lead to sound morphing approaches that are more "perceptually meaningful" than methods that simply interpolate spectral parameters [20]. Therefore in addition to locating attack regions, it is desirable to be able to accurately identify other temporal sections in a musical note that have distinct characteristics.

In this chapter, we present a new technique for the real-time automatic temporal segmentation of musical sounds. Attack, sustain and release segments are defined using cues from a combination of the amplitude envelope, the spectral centroid, and a measurement of the stability of the sound that is derived from an onset detection function. In Section 5.1 we describe some existing approaches to automatic segmentation. Our new method is given in Section 5.2. An evaluation of our method is then provided in Section 5.3, followed by conclusions in Section 5.4.

5.1 Automatic note segmentation

The segmentation of musical instrument sounds into contiguous regions with distinct characteristics has become an important process in studies of timbre perception [45] and sound modelling and manipulation [20]. Since the time of Helmholtz, it has been known that the temporal evolution of musical sounds plays an important role in our perception of timbre. Helmholtz described musical sounds as being a waveform shaped by an amplitude envelope consisting of attack, steady state and decay segments [48]. Here the attack is the time from the onset until the amplitude reaches its peak value, the steady state is the segment during which the amplitude is approximately constant, and the decay is the region where the amplitude decreases again.

A number of automatic segmentation techniques have been developed based on this model, creating temporal region boundaries based solely on the evolution of the amplitude envelope [95, 55]. Automatic segmentation consists of the identification of boundaries between contiguous regions in a musical note. Typically the boundaries are one or more of the following:

Onset: a single instant marking the beginning of a note.

End of attack / start of sustain: end of the initial transient.

End of sustain / start of release: end of the steady state region.

Offset: end of the note.

The regions and boundaries can vary however, firstly depending on the model used by the segmentation technique, and secondly based on the nature of the sound being analysed as not all instrumental sounds are composed of the same temporal events.

This section begins by examining note segmentation based on the evolution of the amplitude envelope in Section 5.1.1. This is followed in Section 5.1.2 by an overview of an approach to automatic note segmentation that also considers the changes in short-time spectra when defining region boundaries.

5.1.1 Amplitude-based note segmentation

Jensen proposed an automatic segmentation technique that analyses the temporal evolution of the derivative of the amplitude envelope [55]. Onset, attack, sustain, release and offset locations are defined for the amplitude envelope of each sinusoidal partial that is detected in a note, however the process could also be used to detect these breakpoints in the overall amplitude envelope. Partial onset and offset times are located as part of the sinusoidal modelling process. To find the remaining boundaries, the amplitude envelope is first smoothed by convolution with a Gaussian according to Equation 5.1 where $A_p(t)$ is the amplitude of the p-th partial at time t and σ is the standard deviation.

$$env_\sigma(t) = A_p(t) * g_\sigma(t), \quad g_\sigma(t) = \frac{1}{2\pi\sigma} e^{-\frac{t^2}{2\sigma^2}} \tag{5.1}$$

The attack and release regions are sections where the envelope slope is positive and negative respectively. The middle of the attack at_m is therefore defined as the maximum of the derivative of the smoothed envelope (Equation 5.2).

$$at_m = \max\left(\frac{\partial}{\partial t} env_\sigma(t)\right) \tag{5.2}$$

The beginning and end of the attack is found by following the derivative from this centre point backwards and forwards in time until it is less than a constant multiplied by the maximum value of the amplitude. Similarly, the middle of the release rt_m is found at the minimum value of the envelope derivative (Equation 5.3). The beginning and end of the release are identified by following the derivative backwards and forwards in time until the value reaches a certain threshold.

$$rt_m = \min\left(\frac{\partial}{\partial t} env_\sigma(t)\right) \tag{5.3}$$

This segmentation technique was examined by Caetano et al. [19]. They found that derivative-based methods were not robust enough and too sensitive to ripples in the amplitude curve and so this approach was not investigated further.

Another amplitude-based segmentation method was proposed by Peeters [95], after noting that the well-known ADSR envelope does not apply to most natural sounds as depending on the nature of the sound, one or more of the segments is often missing. He therefore proposed segmenting musical sounds into two regions named *attack* and *rest*. Only two region boundaries needs to be calculated in this model: the start and end of the attack region (no automatic method is suggested for detecting the note onsets or offsets). Two techniques are described for detecting these boundaries:

Fixed Threshold: The peak value of the energy envelope is found. In order to account for signal noise, the start of the attack is the time when the amplitude envelope reaches 20% of the peak value, and the end of the attack is the time when the envelope reaches 90% of the maximum[2].

[2]It is noted that these values should be set empirically based on the type of sound.

Adaptive Threshold (Weakest Effort Method): The range of values from 0 to the peak value of the energy envelope is divided up into a number of equally spaced regions called *thresholds*. An *effort* as defined as the time it takes the signal to go from one threshold value to the next. The average effort value w is calculated, then the start of the attack is taken to be the first threshold at which the effort is below $M \times w$. The end of the attack is calculated as the first threshold at which the effort is above $M \times w$. A value of $M = 3$ is recommended. This process is depicted in Figure 5.1.

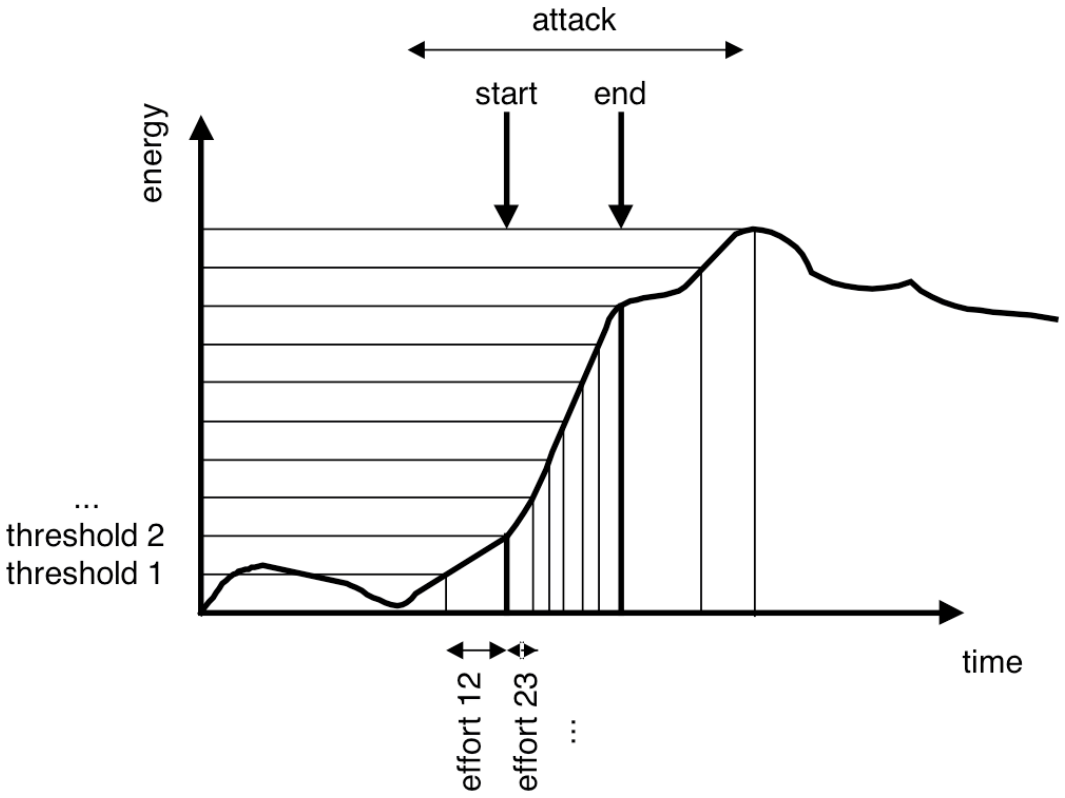

Figure 5.1: Weakest Effort Method. Figure taken from [95].

Although the amplitude envelope often provides a good approximation of the temporal evolution of the internal structure of a musical sound, it simply does not

provide enough information to allow for accurate, robust and meaningful temporal segmentation of the musical audio signal. In particular the attack region, which has often become synonymous with the amplitude rise time [11], is not well delineated by the amplitude envelope. The attack is a transient part of the signal that lasts from the onset until a relatively stable periodicity is established, and as a result the steady state is generally achieved before the end of the initial amplitude rise time [45]. During the steady state the amplitude envelope can often show considerable variation, particularly in the presence of tremolo and/or vibrato. This makes it difficult to detect the boundary between the steady state and the release using just the amplitude envelope, especially if operating under the constraints of a real-time system. The Amplitude/Centroid Trajectory model, which is described in Section 5.1.2, has addressed many of these issues.

5.1.2 Automatic segmentation using the Amplitude/Centroid Trajectory model

It has been shown that in order to better understand the temporal evolution of sounds, it is necessary to also consider the way in which the audio spectrum changes over time [45]. Hajda proposed a new model for the partitioning of isolated non-percussive musical sounds [44], based on observations by Beauchamp that for certain signals the root mean square (RMS) amplitude and spectral centroid have a monotonic relationship during the steady state region [7]. An example of this relationship is shown for a clarinet sample in Figure 5.2. The spectral centroid is given by Equation 5.4, where f is frequency (in Hz) and a is linear amplitude of frequency band b (up to m bands) which are computed by Fast Fourier Transform. The Fourier

Transform is performed on Bartlett windowed analysis frames that are 64 samples in duration. This results in 32 evenly spaced frequency bands (up to 11025 Hz), each with a bandwidth of about 345 Hz.

$$\text{centroid}(t) = \frac{\sum_{b=1}^{m} f_b(t) \times a_b(t)}{\sum_{b=1}^{m} a_b(t)} \tag{5.4}$$

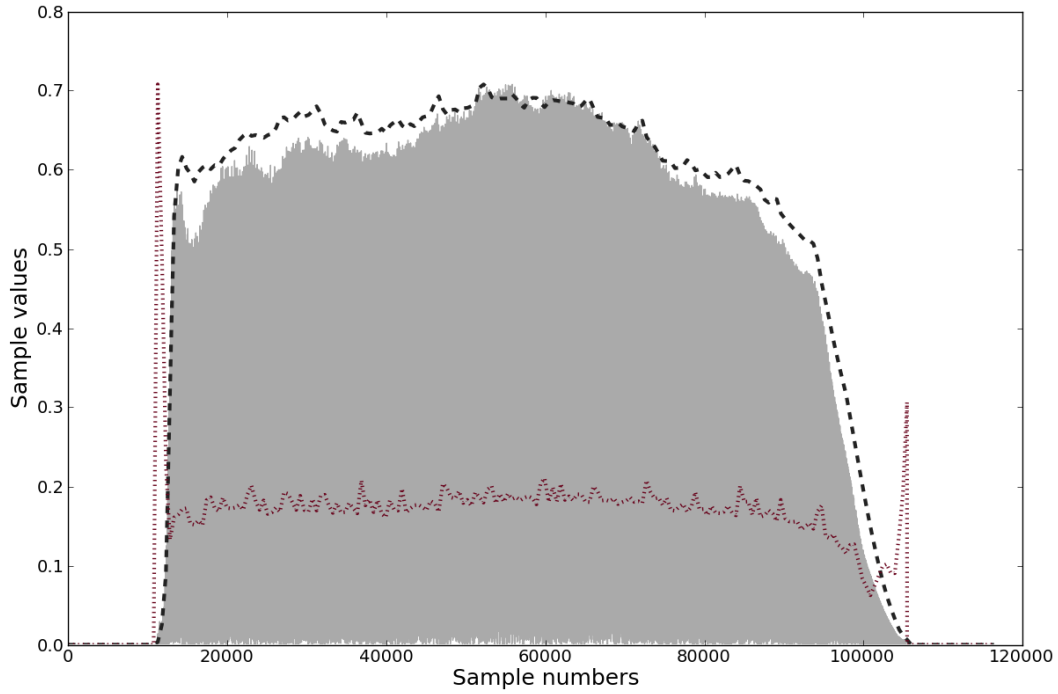

Figure 5.2: The full-wave-rectified version of a clarinet sample, the RMS amplitude envelope (dashed line) and the spectral centroid (dotted line). The RMS amplitude envelope and the spectral centroid have both been normalised and scaled by the maximum signal value.

Hajda's model, called the Amplitude/Centroid Trajectory (ACT), identifies the boundaries for four contiguous regions in a musical tone:

Attack: the portion of the signal in which the RMS amplitude is rising and the spectral centroid is falling after an initial maximum. The attack ends when the centroid slope changes direction (centroid reaches a local minimum).

Attack/steady state transition: the region from the end of the attack to the first local maximum in the RMS amplitude envelope.

Steady state: the segment in which the amplitude and spectral centroid both vary around mean values.

Decay: the section during which the amplitude and spectral centroid both rapidly decrease. At the end of the decay (near the note offset), the centroid value can rise again however as the signal amplitude can become so low that denominator in Equation 5.4 will approach 0. This can be seen in Figure 5.2 (starting at approximately sample number 100200).

Hajda initially applied the ACT model only to non-percussive sounds. However, Caetano et al. introduced an automatic segmentation technique based on the ACT model [19] and proposed that it could be applied to a large variety of acoustic instrument tones. It uses cues taken from a combination of the amplitude envelope and the spectral centroid. The amplitude envelope is calculated using a technique called the true amplitude envelope, which is a time domain implementation of the true envelope.

The True Envelope and the True Amplitude Envelope

The true envelope [50, 101, 21] is a method for estimating a spectral envelope by iteratively calculating the filtered cepstrum, then modifying it so that the original spectral peaks are maintained while the cepstral filter is used to fill the valleys between the peaks. The real cepstrum of a signal is the inverse Fourier transform of the log magnitude spectrum and is defined by Equation 5.5, where X is the spectrum

produced by a N-point DFT.

$$C(n) = \sum_{k=0}^{N-1} log(|X(k)|)e^{j2\pi kn/N} \tag{5.5}$$

The cepstrum can be low-pass filtered (also known as *liftering*) to produce a smoother version of the log magnitude spectrum. This smoothed version of the spectrum can be calculated according to Equation 5.6 where w_n is a low-pass window in the cepstral domain (defined by Equation 5.7) with a cut-off frequency of n_c.

$$\hat{X}(n) = \sum_{k=0}^{N-1} w_n C(k)e^{-j2\pi kn/N} \tag{5.6}$$

$$w(n) = \begin{cases} 1 & |n| < n_c \\ 0.5 & |n| = n_c \\ 0 & |n| > n_c \end{cases} \tag{5.7}$$

If $\hat{X}_i(n)$ is the smoothed version of the spectrum at iteration i, then the true envelope is found by iteratively updating the current envelope A_i according to Equation 5.8.

$$A_i(n) = max(A_{i-1}(k), \hat{X}_{i-1}(k)) \tag{5.8}$$

The algorithm is initialised by setting $A_0(n) = log(|X(k)|)$ and $\hat{X}_0(n) = -\infty$. This process results in the envelope gradually growing to cover the spectral peaks, with the areas between spectral peaks being filled in by the cepstral filter. An additional parameter Δ is specified in order to stop the algorithm, specifying the maximum value that a spectral peak can have above the envelope.

Caetano and Rodet used this enveloping technique in the time domain in order to

create the true amplitude envelope (TAE) [21]. The first step in the TAE is to obtain a rectified version of the audio waveform so that there are no negative amplitude values. The signal is then zero-padded to the nearest power of two, a time-reversed version of it is appended to the end of the signal and the amplitude values are exponentiated as the true envelope assumes that the envelope curve is being created in the log spectrum. Finally, the true envelope algorithm is applied to the time domain signal instead of the Fourier spectrum so that the resulting envelope accurately follows the time domain amplitude peaks.

Automatic segmentation using the ACT model

For each musical tone the method presented by Caetano et al. locates onset, end of attack, start of sustain, start of release and offset boundaries as follows:

Onset: start of the note, found by using the automatic onset detection method described in [100][3].

End of attack: position of the first local minima in the spectral centroid that is between the onset and the start of sustain.

Start of sustain: boundary detected using a modified version of Peeters' weakest effort method.

Start of release: also detected using a version of the weakest effort method, but starting at the offset and working backwards.

[3]This technique basically involves looking for signal regions in which the center of gravity of the instantaneous energy of the windowed signal is above a given threshold. Or in other words, if most of the energy in a spectral frame is located towards the leading edge of the analysis window, then the frame is likely to contain a note onset.

Offset: the last point that the TAE attains the same energy (amplitude squared) as the onset.

Notably, they allow the same point to define the boundary of two distinct contiguous regions. This signifies that the region is too short to be detected as a separate segment and makes the model more robust in dealing with different types of sounds.

A plot of a clarinet sample and the boundaries detected by our implementation of this segmentation method are shown in Figure 5.3. From just a visual inspection of the waveform, the attack and sustain sections look to be well detected. There are some changes between our implementation and the technique described here (which are discussed in more detail in Section 5.3) which partly account for the lack of precision in the identification of the onset and offset. The identification of the release region for this sample does not seem accurate however.

Evaluation of the ACT model

Caetano et al. compare the performance of their automatic segmentation technique to that of the one described by Peeters [95]. They do this by visual inspection of plots of the waveform, spectrogram and detected boundaries produced by both methods, showing 16 analysed samples consisting of isolated tones from western orchestral instruments (plus the acoustic guitar). They found that their model outperformed the Peeters method in all cases, although for one sample (a marimba recording) the amplitude envelope and spectral centroid do not behave in the manner that is assumed by the model and so neither method gives good results. However, this provides strong evidence that the ACT model assumptions can be applied to a wide variety of sounds, and shows that using a combination of the amplitude en-

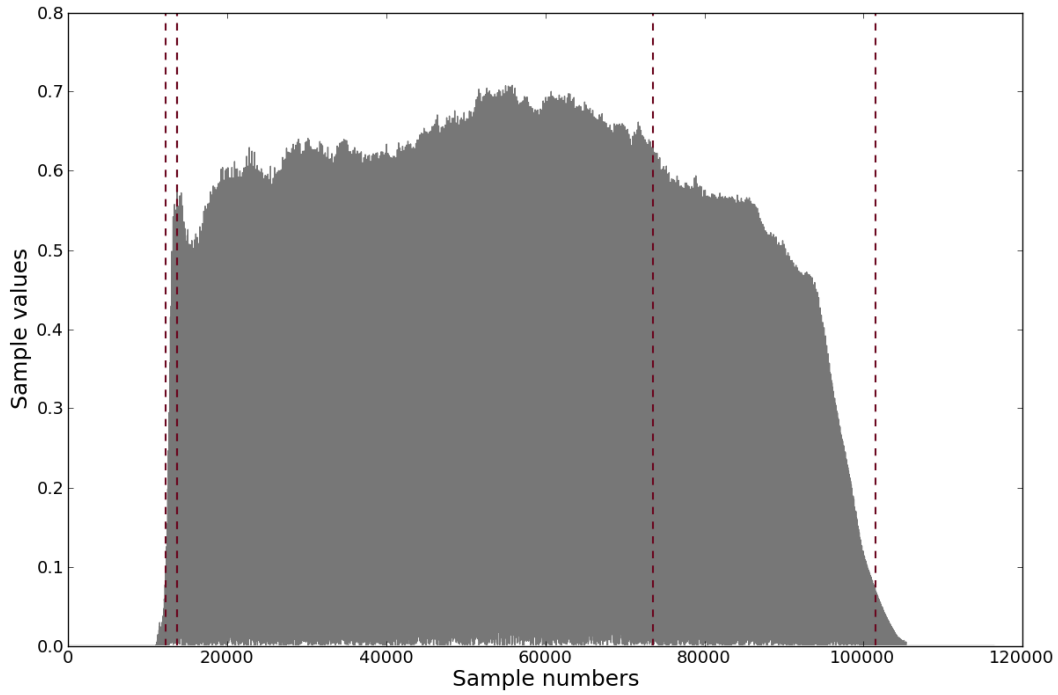

Figure 5.3: A clarinet sample and the boundaries (vertical dashed lines) detected by our implementation of the automatic segmentation technique proposed by Caetano et al. [19]. From left to right, they are the onset, end of attack, start of sustain, start of release and offset.

velope and the spectral centroid can lead to more accurate note segmentation than methods based on the amplitude envelope alone.

The automatic segmentation technique proposed by Caetano et al. cannot be used to improve the performance of real-time synthesis by analysis systems however, as the method for detecting the start of sustain and start of release boundaries requires knowledge of future signal values. The spectral centroid has been shown to be a useful indirect indicator as to the extent of the attack region. However in order to help reduce synthesis artifacts in real-time sinusoidal modelling systems, it is desirable to have a more accurate and direct measure of the attack transient duration by locating signal regions in which the spectral components are changing rapidly or

unpredictably. Both of these issues are addressed by the new segmentation model that is proposed in Section 5.2.

5.2 Real-time automatic note segmentation

The desire to be able to automatically segment notes in streaming audio signals lead to the development of a new segmentation method. It uses cues from a combination of the RMS amplitude envelope, the spectral centroid and an onset detection function. Here we refer to this as a real-time method, with the caveat that we are using the same definitions and constraints that were defined in relation to real-time onset detection systems in Section 4.1. The real-time segmentation model defines boundaries for the onset, start of sustain, start of release and offset as follows:

Onset: start of the note, detected using the peak amplitude difference onset detection method (described in Section 4.8).

Start of sustain (end of attack): a region that begins as soon as the attack transient has finished. This calculation is described in detail in Section 5.2.1.

Start of release (end of sustain): a region that begins when the following conditions are met:

(1) The RMS amplitude envelope is less than 80% of the largest amplitude value seen between the onset and the current frame.

(2) The RMS amplitude envelope is decreasing for 5 consecutive frames.

(3) The current value of the spectral centroid is below the cumulative moving average of the values of the centroid from the onset to the current frame.

This boundary also occurs if the RMS amplitude value drops to less than 33% of the peak value. The RMS amplitude here is subject to a 3 point moving average filter and the spectral centroid is given by Equation 5.4.

Offset: the point at which the RMS amplitude value drops to less than a value Δ below the peak amplitude value. This value should be adapted depending on the nature of the environment in which the sound is captured. In the context of a studio with little background noise we set $\Delta = 60\,\text{dB}$.

A frame size of 512 samples is used, resulting in a latency of 11.6 ms when operating at a sampling rate of 44.1 kHz. The maximum delay in detecting a boundary is 5 frames (or 58 ms).

To be robust, a real-time segmentation model must consider the fact that not all musical musical sounds contain distinct attack, sustain and release segments. One or more of these segments may not be present at all in the sound. Our model manages this situation a similar manner to Caetano et al. method, allowing multiple segment boundaries to occur at the same time index. Transients that occur at time points other than at the start of a note or sound event will generally cause a new onset to be recorded and the model will reset. Sound sources that contain transient components outside of note attack locations will therefore not interfere with distinct sound events that may follow. However, the exact consequences of a model reset on the current sound will vary depending on whether transformations are being applied to the audio signal or not, and if so, on the nature of these transformations.

An example of the boundaries detected by our method is given in Figure 5.4. Detected region boundary positions are indicated by vertical dashed lines. From a

visual inspection, the end of attack and offset boundaries look accurate. There is a slight delay in detecting the onset. The location of the release boundary also looks more accurate than the Caetano et al. method although it still could be improved further. A full comparison between the two approaches is provided in Section 5.3.

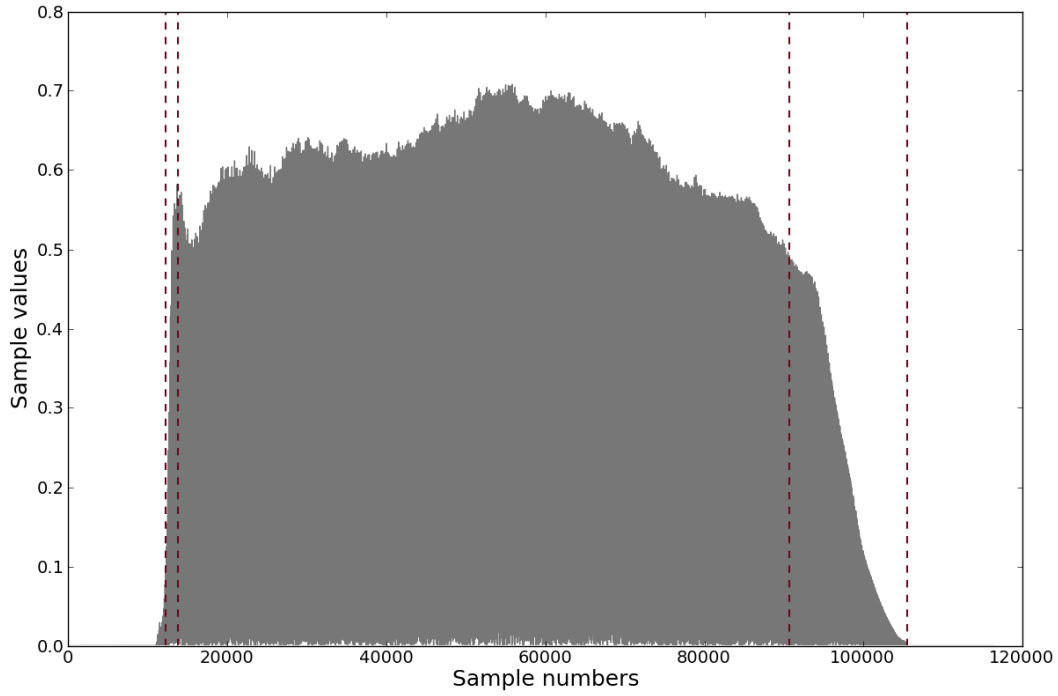

Figure 5.4: A clarinet sample and region boundaries (dashed lines) detected by the proposed real-time segmentation method. The boundaries (from left to right) are the onset, start of sustain (end of attack), start of release and offset.

5.2.1 Calculating the duration of the attack region

Onset locations are typically defined as being the start of the attack transient. The problem of finding their position is therefore linked to the problem of detecting transient regions in the signal. Another way to phrase this is to say that onset detection is the process of identifying which parts of a signal are relatively unpredictable.

The peak amplitude difference method for computing onset detection functions, introduced in Section 4.8, was one of the techniques that performed the best in our ODF evaluation. As it effectively measures errors in the partial tracking stage of sinusoidal modelling, it can also be used to measure the stability of the detected sinusoidal partials in the audio signal. Peaks in the ODF should occur at regions where the spectral components in the signal are most unstable or are changing unpredictably[4].

The ODF can therefore be used to identify transient signal regions. In our real-time note segmentation model the attack transient is defined as being the region from the onset until the next local minima in the ODF. We also signal the end of the attack segment if the RMS amplitude envelope reaches a local maxima. This technique is similar to the transient detection method proposed in [30], where the authors detect transient regions based on peaks in the energy of the noise signal resulting from the identification and removal of the deterministic signal component. However as we do not separate the deterministic and stochastic components, our method should require considerably less computation. In addition, we do not low-pass filter the resulting ODF as doing so widens the ODF peak (and in turn, the detected transient region) without presenting an obvious way to compensate for this deviation.

An example of the ODF and corresponding transient region can be seen in Figure 5.5. The boundaries are detected one frame later than their "correct" position in the ODF as the real-time peak picking algorithm has a latency of one frame. This algorithm is identical to the one that is described in Section 4.2.

[4]This does not only apply to the peak amplitude difference ODF, but indeed to any ODF that measures the variability of spectral components in the audio signal.

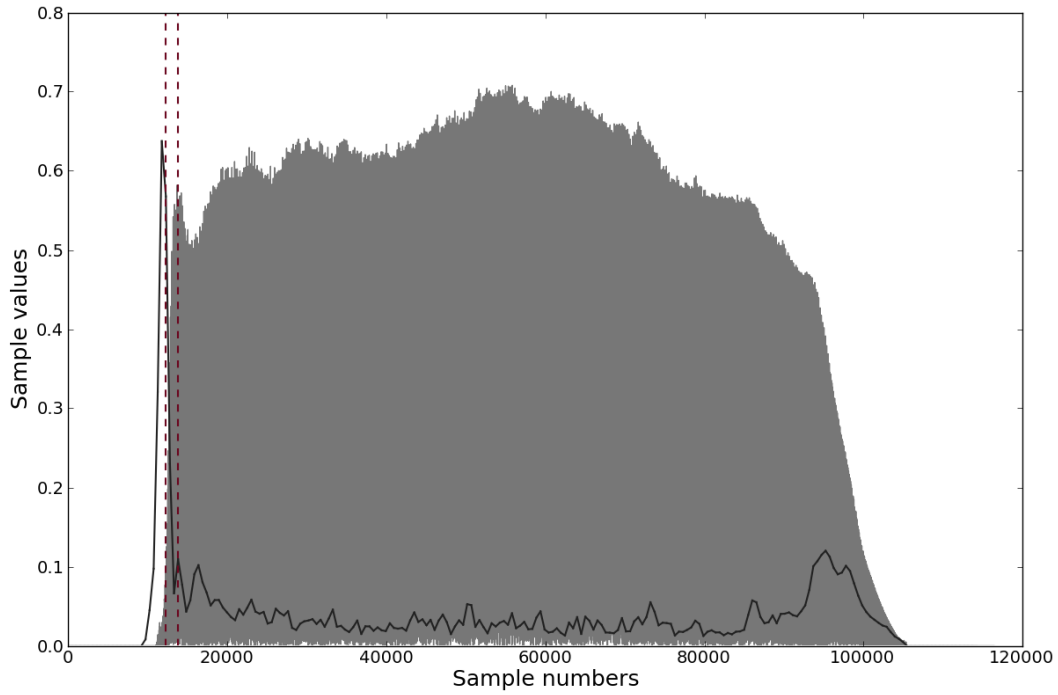

Figure 5.5: A clarinet sample, the onset detection function (solid line), computed using the peak amplitude difference method and the detected transient region (between vertical dashed lines).

5.3 Note segmentation evaluation

The real-time segmentation model was evaluated by comparing the locations of the detected region boundaries with the region boundaries that were calculated using the Caetano et al. method [19]. The first step in this process was to create software implementations of the two techniques. This is described in Section 5.3.1. The second phase of the evaluation process was to compare the locations of the detected region boundaries with a set of reference boundary locations. This procedure and the outcome of the evaluation are discussed in Section 5.3.2.

5.3.1 Real-time note segmentation software

A new Python software library (called *notesegmentation*) was created, consisting of implementations of the Caetano et al. automatic segmentation method and our new real-time method. The real-time method is also implemented in C++. It can be built as a dynamic library or as a Python extension module (using the provided Cython wrapper class). The code that is needed to reproduce our evaluation results is also available. The notesegmentation library is free software released under the terms of the GNU General Public License. It can be found on the accompanying CD.

Lists of the modules, classes and functions that are provided in the note segmentation library are given in Tables 5.1 and 5.2. Functions that are marked with an asterisk (*) currently have no C++ implementation and so are only available in Python[5]. Classes begin with capital letters while functions start with lowercase letters.

[5]The Caetano et al. method cannot be used in a real-time context, and as the Python implementation was suitable for evaluating the accuracy of the segment boundaries, the method was not implemented in C++. The other functions that are omitted from the C++ library are only required for the evaluation process.

Python Module	Classes and Functions in Module
amplitude_envelopes	rms
	tae
partial_stability	get_stability
	get_transients
segmentation	spectral_centroid
	cbr
	rt
util	next_minima
	next_minima_rt
	next_maxima
	next_maxima_rt
	find_peaks
	cumulative_moving_average

Table 5.1: Note segmentation Python modules.

Class or Function	Description
rms	Calculate the RMS amplitude envelope of a signal. Can return the RMS of the current frame (when used in a real-time context) or the RMS of each frame of audio in a full sample.
tae*	Calculate the true amplitude envelope of a signal.
get_stability*	Calculate a partial stability signal for a given audio signal (this process is discussed in Section 5.3.2).
get_transients*	Return the estimated location of attack transients based on the partial stability signal.
spectral_centroid	Calculate the spectral centroid for either a signal audio frame or for each frame in a signal.
cbr*	Calculate region boundaries using the Caetano et al. method.
rtsegmentation*	Calculate region boundaries using our proposed real-time method.
RTSegmentation	C++ class for calculating region boundaries using our proposed real-time method.
next_minima*	Calculate the next local minima in a signal (from a given time index).
next_minima_rt	Calculate the next local minima in a signal (from a given time index) in real-time, requiring a latency of 1 frame.
next_maxima*	Calculate the next local maxima in a signal (from a given time index).

`next_maxima_rt`	Calculate the next local maxima in a signal (from a given time index) in real-time, requiring a latency of 1 frame.
`cumulative_moving_average`	Return the current value of the cumulative moving average of a sequence of values.

Table 5.2: Note segmentation classes and functions (available from both C++ and Python).

5.3.2 Evaluation of segmentation algorithms

To evaluate the performance of the note segmentation algorithms a set of samples with "correct" region boundary locations was needed. No such reference set was found in the literature so a new sample set was created. It consists of a selection of 36 samples of isolated musical sounds that are part of the Modal database and so can be freely redistributed. These particular samples were selected to ensure that sounds with differing types of temporal evolution were considered in the evaluation process. The sample set contains sound sources ranging from short, percussive sonic events to musical notes with longer, sustained excitations. A list of the samples that were used in the evaluation is given in Table 5.3. Descriptions of the sound sources that make up each sample can be found in Table 4.1[6].

As they are from the Modal database each sample comes with hand-annotated onset locations. Additional annotations were added for each sample to denote the start of sustain, start of release and note offset locations. The annotations were all made by one person, which will inevitably lead to some degree of inaccuracy and inconsistency [72], however they should still give some indication as to the

[6]The samples that are marked with an asterisk (*) were not part of the original modal database release. They all consist of single notes played on a cello.

performance of the automatic segmentation methods.

In addition to the hand-annotated segment boundary locations, we also developed an automatic technique for identifying regions in the audio signal with the highest level of sinusoidal partial instability. This was done by first performing sinusoidal analysis on each sample using the SMS method, then calculating a detection function from the sum of the frame by frame variations in log frequency (scaled by log amplitude) for each partial. Areas with unstable partials were then defined as the area around peaks in this detection function. An example of the partial stability signal for a clarinet sample is given in Figure 5.6. The detected region of partial instability is the area between the vertical dashed lines.

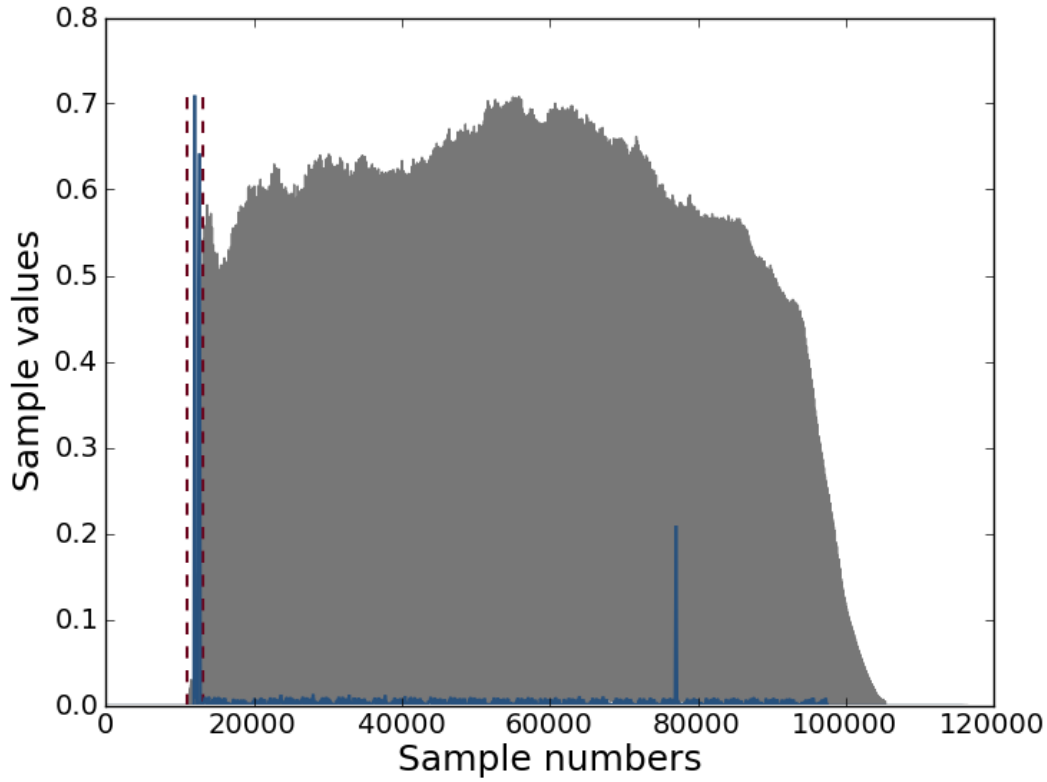

Figure 5.6: A clarinet sample, the partial stability signal (solid line) and the detected region of partial instability around the note onset (between vertical dashed lines).

bass-elec-C-lo.wav	female-glissdown.wav
bell-octave0.wav	femalevoices_aa2_A3.wav
bell-octave1.wav	femalevoices_oo2_C#4.wav
bell-octave2.wav	flugel-C.wav
cello-C-octave0.wav*	flute-alto-C.wav
cello-C-octave1-plucked.wav*	guitar-ac-E-octave1.wav
cello-C-octave1.wav*	guitar-classical-E-octave1-vib.wav
cello-C-octave2-plucked.wav*	guitar-elec-solid-dist-E-octave1-long.wav
cello-G-octave0.wav*	malevoice_aa_A2.wav
cello-G-octave1.wav*	piano-studio-octave1.wav
child-imitate-shaker-hi-up.wav	piano_B4.wav
clarinet-C-octave0-vib.wav	piano_G2.wav
clarinet-C-octave0.wav	prep_pianoE0_2.wav
conga-muffled-1.wav	prep_piano_C0_2.wav
conga-open-1.wav	sax-bari-C-lo.wav
cymbal-hihat-foot-open.wav	singing-female1-C-hi.wav
cymbal-splash-stick.wav	singing-womanMA-C-oo.wav
drum-bass-hi-1.wav	trpt-C-lo.wav

Table 5.3: Note segmentation evaluation samples. Descriptions of the sound sources that make up each sample can be found in Table 4.1.

Evaluation results

The region boundaries that were found by analysing the full sample set with each segmentation algorithm were compared to the hand-annotated reference boundaries

plus the additional partial instability region measurement. However, there is a key difference between our implementation of the Caetano et al. method and the description of the method in the literature: the same onset detection algorithm (the peak amplitude difference method) was used for both segmentation models. As this is a real-time method the detection accuracy will inevitably be worse than if a state-of-the-art non-real-time method was used, so this must be taken into account when interpreting the results.

Table 5.4 gives the average difference in milliseconds between the automatically detected boundary and reference boundary for our method and the Caetano et al. method. Each average value is followed immediately by the standard deviation (σ) in brackets. The average difference between the automatically detected start of sustain region and the value from the reference samples is almost identical for the two methods, but the Caetano et al. method has a lower standard deviation. However, the sustain section of the real-time method is considerably closer to the end of the region with the highest level of partial instability (and with lower standard deviation).

The real-time method also fares better in detecting start of release and note offset locations in comparison with the Caetano et al. method. However, a large part of the error in the Caetano et al. offset detection can be attributed to the fact that they define this boundary based on the energy the signal has at the onset location, and as our onset detector is a real-time method there is a slight latency before it responds, by which stage the signal energy has already started to increase.

When evaluating onset detection algorithms, an onset is commonly regarded as being correctly detected if it falls within 50 ms of the reference onset location in order to allow for human error when creating the set of reference values [11, 72].

Boundary	PM Avg. Dev. (ms)		CBR Avg. Dev. (ms)	
Onset	16.57	($\sigma = 21.64$)	16.57	($\sigma = 21.64$)
Start of sustain	64.87	($\sigma = 108.53$)	64.86	($\sigma = 82.37$)
End of unstable partials	46.50	($\sigma = 52.34$)	83.47	($\sigma = 106.40$)
Start of release	586.46	($\sigma = 831.81$)	900.72	($\sigma = 1115.00$)
Offset	331.32	($\sigma = 999.49$)	1597.75	($\sigma = 2099.44$)

Table 5.4: Average deviation from boundaries in reference samples for our proposed method (PM) and for the Caetano et al. method (CBR).

Table 5.5 gives the percentage of automatically detected boundaries that fall within 50 ms of the reference values for both segmentation methods. Here, our proposed method is more accurate in detecting the sustain boundary and is again closer to the end of the unstable partial region. The Caetano et al. method is more accurate in detecting the release, with our method performing better at note offset detection.

Boundary	PM Accuracy (%)	CBR Accuracy (%)
Onset	91.67	91.67
Start of sustain	77.78	63.89
End of unstable partials	69.44	58.33
Start of release	25.00	33.33
Offset	44.44	19.44

Table 5.5: The percentage of automatically detected boundaries that fall within 50 ms of the reference values for our proposed method (PM) and for the Caetano et al. method (CBR).

Table 5.6 gives the percentage of automatically detected boundaries that fall within 50 ms of the reference values, categorised by sound type, for the two segmentation methods. The sound types are non-pitched percussive (NPP), pitched percussive (PP) and pitched non-percussive (PNP). The two methods have the same performance for the non-pitched percussive sound category. For pitched percussive sounds, the methods have the same results for the sustain boundary. The Caetano et al. method has a higher percentage of correct detections of the end of unstable partials and release boundaries, with our proposed method correctly locating more

offset boundaries. In the pitched non-percussive sound category our method performs better for all boundaries (except for the onset boundary, as the onset detection algorithm is identical).

		NPP	PP	PNP
PM	Onset	100.00	100.00	86.36
	Start of sustain	80.00	88.89	72.73
	End of unstable partials	80.00	77.78	63.64
	Start of release	40.00	0.00	31.82
	Offset	20.00	33.33	54.55
CBR	Onset	100.00	100.00	86.36
	Start of sustain	80.00	88.89	50.00
	End of unstable partials	80.00	66.67	50.00
	Start of release	40.00	55.56	22.73
	Offset	20.00	0.00	27.27

Table 5.6: The percentage of automatically detected boundaries that fall within 50 ms of the reference values, categorised by sound type, for our proposed method (PM) and for the Caetano et al. method (CBR). The sound types are non-pitched percussive (NPP), pitched percussive (PP) and pitched non-percussive (PNP).

Evaluation conclusions

The results show that both methods perform reasonably well at detecting the start of the sustain region, although our start of the sustain region is significantly closer to the end of the region with high partial instability. Neither method performs particularly well in detecting the release and offset with high accuracy, although on average our proposed model behaves more robustly. However unlike the Caetano et al. method, our model calculates region boundaries from streams of audio signals and is suitable for use in a real-time musical performance context.

5.4 Conclusions

The sound transformation process can be improved by considering the temporal characteristics of musical notes. In particular, to improve the quality of the synthesis of attack transients in sinusoidal models, it is first necessary to be able to accurately identify them. This chapter introduced a new model for the real-time segmentation of the temporal evolution of musical sounds. Attack, sustain and release regions are identified by the model, using cues from the amplitude envelope, spectral centroid and an onset detection function that is based on measuring errors in sinusoidal partial tracking. We evaluated our method by comparing it with the technique proposed by Caetano et al., and found that it generally performs better and is more robust. Note onsets, attack and sustain boundaries are identified with a relatively high degree of accuracy, but neither method was particularly accurate in detecting the release and offset boundaries. Our method can run in real-time and with considerably lower computation requirements as it does not calculate the computationally costly true amplitude envelope.

Chapter 6

Metamorph: Real-time high-level sound transformations based on a sinusoids plus noise plus transients model

One of the goals of this research is to provide new software synthesis tools for composers, musicians and researchers, enabling them to perform timbral transformations on live audio streams. As the temporal evolution of the sound spectrum has a large influence on the perception of timbre, it seems natural to use a sound model that is based on the frequency spectrum as a tool to manipulate timbre. The Simpl software library for sinusoidal modelling was introduced in Chapter 3. To improve the synthesis of attack transients, software libraries for performing real-time onset detection and note segmentation were created, discussed in Chapters 4 and 5 respectively. This chapter combines these three components, creating a real-time

sound synthesis by analysis tool called *Metamorph*. Metamorph is an open source software library (available under the terms of the GNU General Public License) for performing high-level sound transformations based on a sinusoids plus noise plus transients model. It is written in C++, can be built as both a Python extension module and a Csound opcode, and currently runs on Mac OS X and Linux. It is designed to work primarily on monophonic, quasi-harmonic sound sources and can be used in a non-real-time context to process pre-recorded sound files or can operate in a real-time (streaming) mode.

The sinusoids plus noise plus transients model that is used by Metamorph is discussed in Section 6.1. This is followed by a description of the sound transformations that are currently available in Metamorph in Section 6.2. An overview of the implementation of the software is provided in Section 6.3, followed by sound transformation examples in Section 6.4. Conclusions are given in Section 6.5.

6.1 The Metamorph sinusoids plus noise plus transients model

Several systems have been proposed that combine a representation of transient signal components and a sinusoids plus noise model. The method described by Masri [77] aims to reproduce the sharpness of the original transient during synthesis. First a pre-analysis scan of the audio signal is performed in order to detect transient regions, which are defined as being the areas between a note onset and the point at which the onset detection function (based on an amplitude envelope follower) falls below a fixed threshold or reaches a maximum duration (whichever is shorter).

This information is then used during sinusoidal analysis to make sure that the edges of the analysis windows are snapped to the region boundaries. During synthesis, the missing overlap at the region boundaries is reconstructed by extrapolating the waveforms from the centres of both regions and then performing a short cross-fade. However, this method can not run in real-time due to the need for a pre-analysis scan of the audio signal.

Levine [73] introduced a sinusoids plus noise model that includes transform-coded transients. Note onsets are located using a combination of an amplitude rising edge detector and by analysing the energy in the stochastic component. Transient regions are then taken to be fixed-duration (66 ms) sections immediately following a note onset. The transients are translated in time during time-scaling and pitch transposition, however as the primary application of this work was for use in data compression there is no ability to musically manipulate the transients. In the general case, it would also appear to be desirable to determine the duration of transient regions based on measurable signal characteristics, as it does not seem obvious that all transients will be of the same duration.

Verma and Meng proposed a system that extends SMS with a model for transients in [121]. They show that transient signals in the time domain can be mapped onto sinusoidal signals in a frequency domain using the discrete cosine transform (DCT). However, it is not suitable for real-time applications as it requires a DCT frame size that makes the transients appear as a small entity, with a frame duration of about 1 second recommended. This is far too much a latency to allow it to be used in a performance context[1].

[1] According to the real-time performance requirements that are described in Section 4.1

6.1.1 The Metamorph model

Metamorph uses a flexible real-time sinusoids plus noise plus transients model which is summarised in Figure 6.1. The input sound is first segmented using the real-time note segmentation method, which locates attack, sustain and release temporal regions. The sound is analysed using the chosen sinusoidal modelling implementation from the Simpl library. Any of the available peak detection, partial tracking, harmonic synthesis and noise synthesis algorithms can be selected, but by default Metamorph currently uses the bandwidth-enhanced model for peak detection and SMS for all other analysis and synthesis stages. If the current frame is not in an attack region then the output frame will be computed by synthesising the identified deterministic and stochastic components and summing the result. Transformations may optionally be applied to deterministic and stochastic parameters before synthesis.

If the current frame is in an attack transient region then transient modifications are applied (if any have been selected) and then the raw samples are copied into the output frame. During synthesis the output of the transient region is extended slightly (for a single hop size) in order to perform a short cross-fade between the unmodified sample values in the transient region and the synthesised signal in the note region that follows. If the hop size is small enough then this cross-fade will not be noticeable [78]. Metamorph has a default hop size of 512 samples (a duration of 11.6 ms when sampled at the default rate of 44.1 kHz).

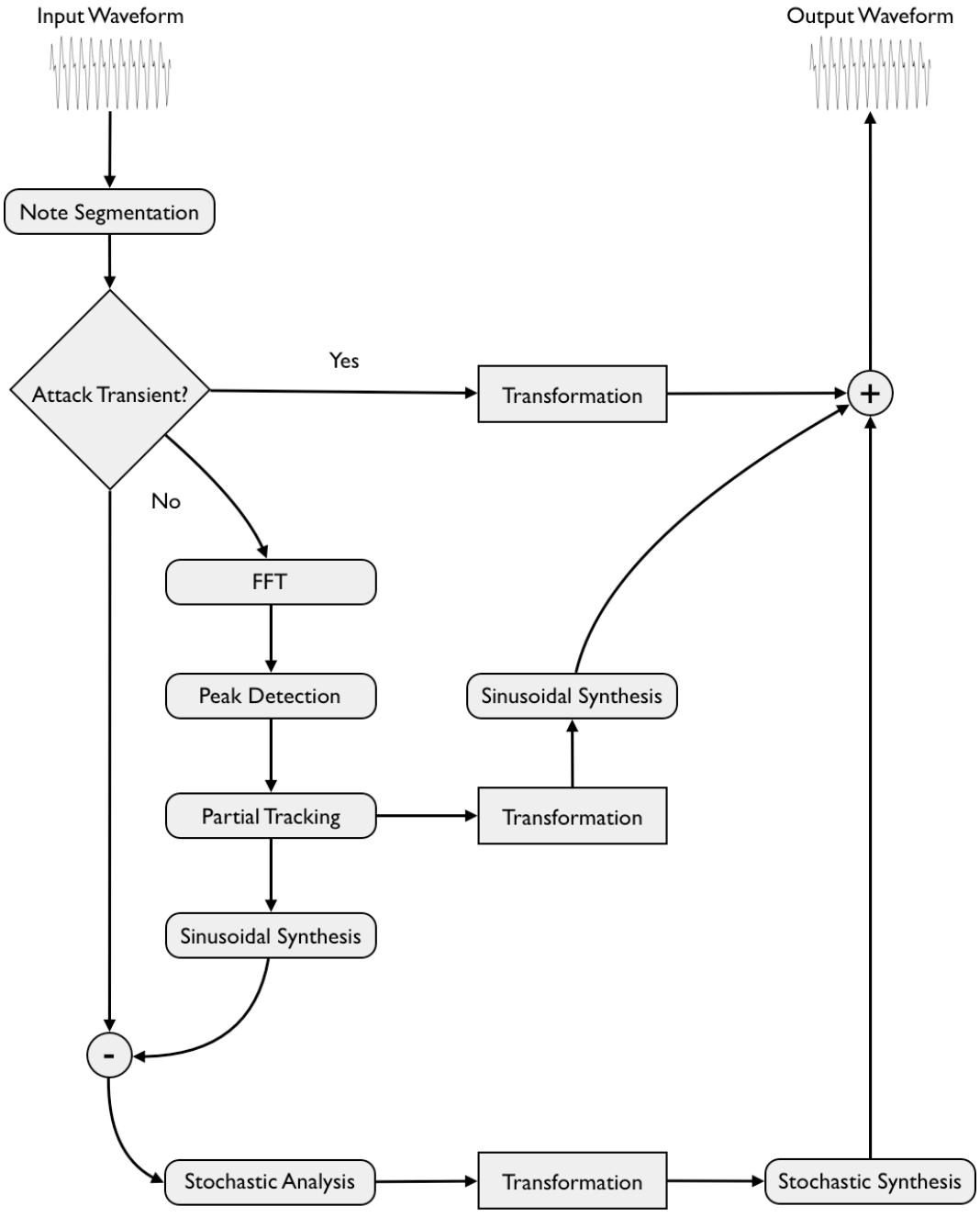

Figure 6.1: The sinusoids plus noise plus transients model that is used in Metamorph.

6.2 Metamorph sound transformations

This section describes the sound transformations that are currently available in Metamorph, which can generally be defined as adaptive digital audio effects [120]. Examples that show how these transformations can be applied to streaming audio signals are given in Section 6.4.

6.2.1 Harmonic distortion

The harmonic distortion of a sound [109] is a measure of the degree of the deviation of the measured partials from ideal harmonic partials. The Metamorph harmonic distortion transformation allows the user to alter the deviation of each synthesised partial in a sound from the ideal harmonic spectrum according to Equation 6.1, where i is the partial number, f is the analysed partial frequency, F_0 is the estimated fundamental frequency, α is the input parameter (between 0 and 1) and F is the output frequency of the synthesised partial.

$$F_i = (\alpha \times f_i) + ((1 - \alpha) \times (F_0 \times i)) \tag{6.1}$$

6.2.2 Noisiness and Transience

The noisiness [109] of a synthesised frame is calculated by taking the ratio of the amplitude of the residual component to the total signal amplitude. This is given by Equation 6.2, where s is the original signal and s_R is the residual.

$$Noisiness = \frac{\sum_{n=0}^{M-1} |s_R(n)|}{\sum_{n=0}^{M-1} |s(n)|} \tag{6.2}$$

Metamorph allows the user to easily adjust this balance by altering the amplitudes of the deterministic and stochastic components independently. It also enables the independent manipulation of the amplitude of transient regions. This effect is called changing the signal *transience*.

6.2.3 Spectral envelope manipulation

A spectral envelope is a curve in the spectrum of an audio signal that approximates the distribution of the signal's energy over frequency. Ideally this curve should pass through all of the prominent spectral peaks in the frame and be relatively smooth, preserving the basic formant structure of the frame without oscillating too much or containing discontinuities. Many different techniques for estimating the spectral envelope have been proposed, with most based on either linear prediction [75] or on the real cepstrum [91]. Spectral envelopes in Metamorph are calculated using the discrete cepstrum envelope method [39, 40, 22] which provides a smooth interpolation between the detected sinusoidal peaks. This method was found to produce more accurate spectral envelopes to those produced by linear prediction or the real cepstrum [107, 101].

Further comparison between the discrete cepstrum envelope and the true envelope methods would be interesting as in [101, 102] it was shown to produce spectral envelopes that are as good (if not better) than the discrete cepstrum envelope, and it can also be computed efficiently. However, this study only compared the envelopes in two example transformations. Samples of singing voice sounds were transposed with spectra warped by each envelope. Envelope parameters were set so that the output synthesised sound was subjectively judged to be of the highest quality. In

the first, the difference between the two methods was described as being "small and hardly perceptually relevant". The discrete cepstrum envelope does perform worse with the second example, creating an artificial boost in the magnitude of the frequencies below the fundamental frequency.

Another one of the main problems that the authors in [101] had with the discrete cepstrum envelope is that it requires a potentially computationally expensive fundamental frequency analysis or other means of identifying spectral peaks. As Metamorph manipulates audio using a sinusoidal model, spectral peaks have already been identified before the spectral envelope is calculated, and so there is no additional computational cost associated with this process. This will have an influence on the relative performance costs for the two methods that are reported in [101].

The discrete cepstrum envelope

The real cepstrum was defined in Equation 5.5. For reference this is repeated in Equation 6.3.

$$C(n) = \sum_{k=0}^{N-1} log(|X(k)|)e^{j2\pi kn/N} \tag{6.3}$$

The log magnitude spectrum can be recovered from the cepstral coefficients by DFT according to Equation 6.4.

$$log(|X(k)|) = \sum_{k=0}^{N-1} C(k)e^{-j2\pi kn/N} \tag{6.4}$$

$log(|X(k)|)$ is even symmetric with $log(|X(k)|) = log(|X(N-k)|)$ and so the cepstral coefficients are also symmetric with $C(k) = C(N-k)$. Equation 6.4 can

therefore be rewritten as Equation 6.5.

$$log(|X(k)|) = C(0) + 2\sum_{k=1}^{\frac{N}{2}-1} C(k)\, cos(2\pi kn/N) + C(N/2)\, cos(\pi k) \qquad (6.5)$$

A smoothed version of the original log magnitude spectrum can be created by setting all but the leading P terms to 0, where P is called the *order* of the cepstrum. This smoothed magnitude spectrum S can therefore be computed according to Equation 6.6 where f is the normalised frequency.

$$log(S(f)) = C(0) + 2\sum_{p=0}^{P} C(p)\, cos(2\pi fp) \qquad (6.6)$$

Instead of computing the cepstrum directly from the spectrum in this manner, Galas and Rodet proposed a way to calculate a cepstrum envelope from a set of discrete points in the frequency/magnitude plane [39, 40]. Their method is called the *discrete cepstrum* and the resulting spectral envelope is called the *discrete cepstrum envelope* (DCE). Given a set of K spectral magnitude and normalised frequency values denoted by a_k and f_k, the discrete cepstrum coefficients are found by minimising the error function ε given in Equation 6.7.

$$\varepsilon = \sum_{k=1}^{K} \mid log(a_k) - log(S(f_k)) \mid^2 \qquad (6.7)$$

The least-square solution is given in Equation 6.8 where

$A = [log(a_1), log(a_2), \ldots, log(a_K)]^{\mathrm{T}}$ are the magnitude values,

$C = [C(0), C(1), \ldots, C(P)]^{\mathrm{T}}$ are the computed cepstral coefficients and M is de-

fined by Equation 6.9.

$$C = (M^{\mathrm{T}}M)^{-1}M^{\mathrm{T}}A \tag{6.8}$$

$$M = \begin{bmatrix} 1 & 2cos(2\pi f_1) & 2cos(2\pi 2f_1) & \dots & 2cos(2\pi P f_1) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 2cos(2\pi f_K) & 2cos(2\pi 2f_K) & \dots & 2cos(2\pi P f_K) \end{bmatrix} \tag{6.9}$$

To improve the smoothness of the spectral envelope a regularisation term can be introduced to Equation 6.8 which effectively penalises rapid changes in the spectral envelope [22]. The regularised envelope can therefore be calculated by Equation 6.10 where R is a diagonal matrix with diagonal elements
$8\pi^2[0, 1^2, 2^2, \dots, P^2]$ and λ is the regularisation parameter.

$$C = (M^{\mathrm{T}}M + \lambda R)^{-1}M^{\mathrm{T}}A \tag{6.10}$$

An example of the DCE calculated from one frame of a saxophone sample is given in Figure 6.2. The spectral peaks are denoted by circles, the magnitude spectrum is shown by the dashed line and the DCE shown by the solid line.

Transformations using the spectral envelope

The spectral envelope transformation in Metamorph allows the amplitudes of the synthesised sinusoidal partials to be altered to match the corresponding amplitude values in a different spectral envelope. This can be a fixed envelope, or a different sound source can be specified and its envelope is then extracted as the target envelope. Synthesised partial amplitudes may also be linearly interpolated between the original spectral envelope and the target envelope. The amplitude of each partial is then given by Equation 6.11, where α is the input parameter (between 0 and 1), O

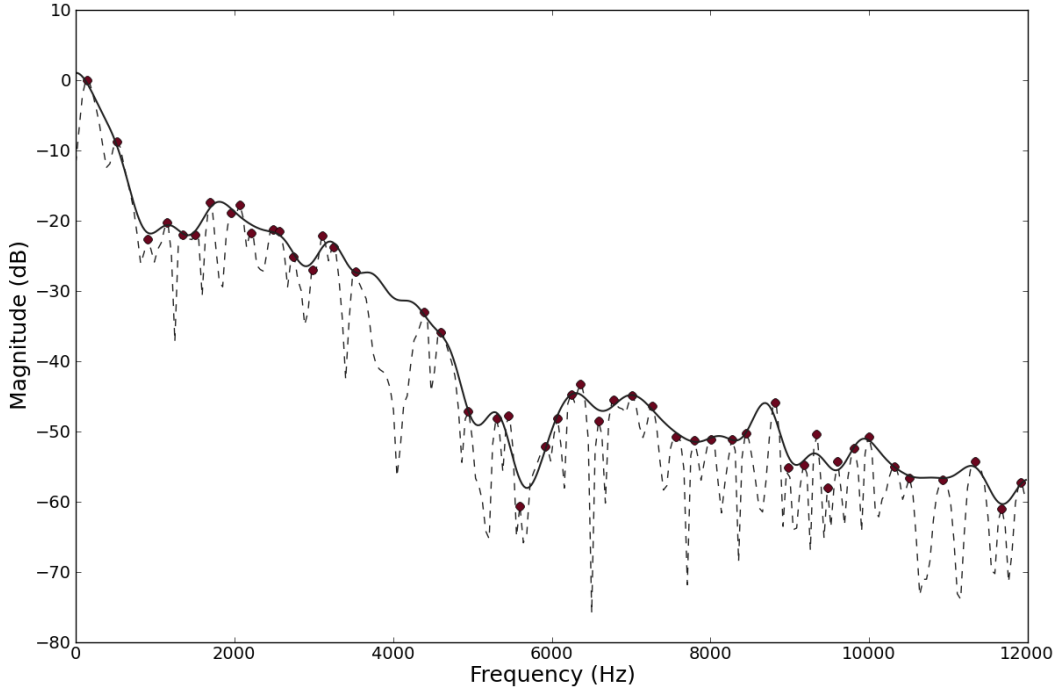

Figure 6.2: Spectrum of one frame from a saxophone sample (dashed line), detected spectral peaks (circles) and the discrete cepstrum envelope (solid line). The envelope order is 40 and regularisation parameter λ is set to 0.0005.

is the original partial amplitude and T is the value of the target spectral envelope at the frequency of the partial.

$$Amplitude = (\alpha \times O) + (|(\alpha - 1)| \times T) \tag{6.11}$$

Although similar techniques can be performed using tools such as the phase vocoder, spectral envelope manipulation using sinusoidal models enables the preservation (or independent manipulation) of the stochastic signal component, offering greater possibilities for sound transformation. In Metamorph this can be taken a step further, altering the spectral envelope of a musical tone while preserving the initial note attack.

6.2.4 Transposition

Sounds can be transposed in Metamorph in two different ways. Both techniques involve initially multiplying the frequency values of all synthesised partials by the same constant factor. The second process additionally adjusts all of the partial amplitude values so that they match those of the original spectral envelope. The latter approach preserves the original formant structure which results in a more natural sounding transposition for certain types of sounds. The transient region can be either modified or unmodified for both types of transposition.

6.2.5 Time-scaling

Time-scaling is the only Metamorph transformation that is not available in real-time mode. The time-scaling algorithm works by keeping the analysis and synthesis frame rates identical, but instead of passing each analysis frame directly to the synthesis module, frames may be repeated (or skipped) depending on the required time-scale factor. This approach does not result in any synthesis artifacts or discontinuities as the synthesis module interpolates smoothly between input frames, and has been shown to produce high-quality time-scaling [15]. Frames from transient regions are treated differently by default however; they are always passed to the synthesis module in the original order, with the sample values passed unmodified to the output so that the transient is maintained. This means that the time-scale factor has to be adjusted slightly during non-transient regions in order to make sure that the final output signal is of the required length. It is also possible to time-scale the transient regions if desired.

6.2.6 Transient processing

Most transformations in Metamorph aim to preserve the original transient region by default, but it is also possible to explicitly alter the output transient. The most basic effect is to either filter the transient using either low- or high-pass filters, which although relatively simple can have quite a large impact on the resulting sound. The transient can also be removed altogether. Another interesting effect is transient substitution, where the transient regions in the audio signal can be replaced by a different set of audio samples (which may or may not themselves be transients). This allows for the creation of various hybrid instruments, for example combining the attack of a drum sound with a sustained woodwind instrument tone.

6.3 Implementation

This Section provides an overview of the design and implementation of Metamorph. Similarly to the Simpl, Modal and the Note Segmentation libraries (described in Chapters 3, 4 and 5 respectively), Metamorph is open source software, released under the terms of the GNU General Public License. It is written in C++ and can be built as a dynamic library or as a Python extension module using the included Cython wrapper code. In addition, a Metamorph Csound opcode is also available, making it easy to integrate the Metamorph real-time sound transformations with the wide variety of sound design and signal processing tools that are included in Csound or have been provided by the Csound community. The most important class in Metamorph, called `FX`, is discussed in Section 6.3.1. Section 6.3.2 describes how Metamorph can be extended by using `Transformation` classes, illustrated by an outline of the `Transposition` class. A full list of the Metamorph modules, classes,

functions and Csound opcodes is given in Section 6.3.3. The full source code can be found on the accompanying CD.

6.3.1 The `FX` class

The `FX` class contains the core Metamorph processing functionality. It is the largest class in the library, consisting of 39 public methods. The majority of these methods are basic getter and setter methods for the analysis/synthesis model parameters. The most important method in the class, `process_frame`, is examined in detail in this section. This method performs the real-time analysis and synthesis, as well as providing plugin points that enable arbitrary transformations to be applied to the three signal components (sinusoids, noise and transients) independently.

The start of the `process_frame` method is shown in Listing 6.1. It is called once for each `hop_size` block of samples and has 4 input parameters: the size of the input frame, a pointer to an array of input samples, the size of the output array and a pointer to an array that will contain the computed output samples. The input size and output size should be equal to the signal hop (or buffer) size.

```
509 void FX::process_frame(int input_size, sample* input,
510                         int output_size, sample* output) {
511     // setup for current frame
512     _input.assign(input, input + _hop_size);
513     setup_frame(input_size, output_size);
514 
515     // calculate current temporal region
516     _previous_segment = _current_segment;
517     _current_segment = _ns.segment(input_size, input);
518 
519     // if at onset, reset any processes that rely on the current note segment
520     if(_current_segment == notesegmentation::ONSET) {
521         reset();
522     }
523 
524     // find sinusoidal peaks and partials
525     _pd->find_peaks_in_frame(_frame);
526     _pt->update_partials(_frame);
```

Listing 6.1: The Metamorph FX class (lines 509-526).

A copy of the input samples is taken on line 512. The setup_frame method is then called on line 513 which makes sure that the Simpl Frame objects are cleared of any previous analysis data and they contain the appropriate audio for the current frame.

On line 517 the Note Segmentation library is used to calculate the current temporal region. If the current frame is a note onset, the reset method is called (on line 521) to clear any data that should change on a note-by-note basis, such as sinusoidal partial tracking data. Simpl PeakDetection and PartialTracking instances are then used to find the spectral peaks and sinusoidal partials in the current frame respectively on lines 525 and 526.

The transient preservation code is shown in Listing 6.2. First, any specified transformations are applied to the raw transient samples by calling the process_frame method of the TransientTransformation objects[2] on lines 534-536, then performing the transient substitution transformation (which is included in the FX class) on lines 538-548. The transient samples are then copied directly to the output buffer, scaled by the _transient_scale parameter on lines 550-552.

```
528     // don't use synthesis output for transient region if
529     // _preserve_transients is set to true
530     if(_preserve_transients && (_transient_scale > 0) &&
531        (_current_segment == notesegmentation::ONSET ||
532         _current_segment == notesegmentation::ATTACK)) {
533         // perform all transient transformations
534         for(int i = 0; i < _transient_trans.size(); i++) {
535             _transient_trans[i]->process_frame(_input);
536         }
537 
538         if(_transient_substitution) {
539             for(int i = 0; i < _hop_size; i++) {
```

[2]Metamorph Transformation classes are discussed in Section 6.3.2.

```
540                 if(_transient_sample < _new_transient_size) {
541                     _input[i] = _new_transient[i];
542                     _transient_sample++;
543                 }
544                 else {
545                     break;
546                 }
547             }
548         }
549 
550         for(int i = 0; i < _hop_size; i++) {
551             output[i] += _input[i] * _transient_scale;
552         }
553     }
```

Listing 6.2: The Metamorph FX class (lines 528-553).

If transient preservation is not enabled, or the current frame is not in a transient note segment, the output buffer will be the sum of synthesised deterministic and stochastic components (potentially after a short cross-fade if this current segment follows a transient region). The synthesis of the harmonic and noise components is shown in Listing 6.3. The former is computed first, beginning on line 556. The `create_envelope` method is called on line 557, which computes the spectral envelope from the sinusoidal partials in the current frame using the discrete cepstrum envelope technique.

```
554     else {
555         // perform all harmonic transformations
556         if(_harmonic_scale > 0) {
557             create_envelope(_frame);
558 
559             for(int i = 0; i < _harm_trans.size(); i++) {
560                 _harm_trans[i]->process_frame(_frame);
561             }
562 
563             for(int i = 0; i < _specenv_trans.size(); i++) {
564                 _specenv_trans[i]->process_frame(_frame, _new_env);
565             }
566 
567             apply_envelope(_frame);
568             _synth->synth_frame(_frame);
569         }
570 
571         // perform all residual transformations
572         if(_residual_scale > 0) {
573             for(int i = 0; i < _residual_trans.size(); i++) {
```

```
574                 _residual_trans[i]->process_frame(_residual_frame);
575             }
576
577             _residual->synth_frame(_residual_frame);
578         }
```

Listing 6.3: The Metamorph `FX` class (lines 554-578).

Following this, sinusoidal transformations are applied by calling the `process_frame` method of all `HarmonicTransformation` and `SpecEnvTransformation` instances (lines 559-565). If the spectral envelope has been modified, the partials magnitudes are changed accordingly by calling the `apply_envelope` method on line 567. The deterministic component is synthesised on line 568, with the stochastic component then computed on lines 571-578. Residual transformations are applied by calling the `process_frame` method of any `ResidualTransformation` instances that are attached to the `FX` object on lines 573-575, followed by the synthesis of the final residual signal on line 577.

```
580         if(_preserve_transients &&
581            _current_segment == notesegmentation::SUSTAIN &&
582            (_previous_segment == notesegmentation::ONSET ||
583             _previous_segment == notesegmentation::ATTACK)) {
584
585             // perform all transient transformations
586             for(int i = 0; i < _transient_trans.size(); i++) {
587                 _transient_trans[i]->process_frame(_input);
588             }
589
590             if(_transient_substitution) {
591                 for(int i = 0; i < _hop_size; i++) {
592                     if(_transient_sample < _new_transient_size) {
593                         _input[i] = _new_transient[i];
594                         _transient_sample++;
595                     }
596                     else {
597                         break;
598                     }
599                 }
600             }
601
602             // end of transient section, crossfade
603             for(int i = 0; i < _fade_duration; i++) {
604                 output[i] += _input[i] * _fade_out[i] * _transient_scale;
605                 output[i] += _frame->synth()[i] * _fade_in[i] *
606                              _harmonic_scale;
```

```
607                     output[i] += _residual_frame->synth_residual()[i] *
608                                  _fade_in[i] * _residual_scale;
609                 }
610 
611                 for(int i = _fade_duration; i < _hop_size; i++) {
612                     output[i] += _frame->synth()[i] * _harmonic_scale;
613                     output[i] += _residual_frame->synth_residual()[i] *
614                                  _residual_scale;
615                 }
616             }
617             else {
618                 for(int i = 0; i < output_size; i++) {
619                     output[i] += _frame->synth()[i] * _harmonic_scale;
620                     output[i] += _residual_frame->synth_residual()[i] *
621                                  _residual_scale;
622                 }
623             }
624         }
625 }
```

Listing 6.4: The Metamorph `FX` class (lines 580-625).

The contents of the output buffer for the non-transient signal regions is computed on lines 580-625 and shown in Listing 6.4. If the current frame is immediately following a transient section, transient transformations are applied to the input sample block as before on lines 580-600. This is then cross-faded with the synthesised deterministic and stochastic components on lines 602-616. If the current frame does not follow a transient frame, the output buffer is computed by summing the deterministic and stochastic components on lines 617-623.

6.3.2 Extending Metamorph using `Transformation` classes

The `FX.process_frame` method has a number of points where transformations can be applied to sinusoidal, noise or transient data before the output signal is synthesised. It is therefore possible to extend Metamorph by creating new classes that are derived from the abstract classes defined in *transformations.h*: `HarmonicTransformation`, `SpecEnvTransformation`, `ResidualTransformation` and `TransientTransformation`. Each class that is derived from `Transformation` must implement a `process_frame`

method, taking a reference to a Simpl `Frame` object as input. The `process_frame` method in classes derived from `SpecEnvTransformation` have an additional parameter: a reference to a `std::vector` of samples that contains the spectral envelope. A `Transformation` object can be added to an `FX` instance by calling the `FX.add_transformation` method.

The use of `Transformation` classes is exemplified by the Metamorph `Transposition` class, which is derived from the abstract class `HarmonicTransformation` (shown in Listing 6.5). The definition of the `Transposition` class is given in Listing 6.6. It consists of private variables to hold the transposition amount, getter and setter methods to change this transposition amount, and a `process_frame` method that will be called by the main `FX` object.

```
23 class Transformation {
24     public:
25         virtual void process_frame(simpl::Frame* frame) = 0;
26 };
27
28
29 class HarmonicTransformation : public Transformation {};
```

Listing 6.5: The class definitions for `Transformation` and `HarmonicTransformation`.

```
48 class Transposition : public HarmonicTransformation {
49     private:
50         sample _transposition;
51         sample semitones_to_freq(sample semitones);
52
53     public:
54         Transposition();
55         Transposition(sample new_transposition);
56         sample transposition();
57         void transposition(sample new_transposition);
58         void process_frame(simpl::Frame* frame);
59 };
```

Listing 6.6: The class definition for `Transposition`.

The implementation of the `Transposition` class is shown in Listing 6.7. The `process_frame` method loops over each sinusoidal partial in the supplied Simpl `Frame` object, multiplying the frequency of the partial by the transposition amount (in Hz). This method is called on line 544 in *FX.cpp* (Listing 6.3), before spectral envelope transformations are applied.

```
9  Transposition::Transposition() {
10     _transposition = 0;
11 }
12
13 Transposition::Transposition(sample new_transposition) {
14     _transposition = new_transposition;
15 }
16
17 sample Transposition::transposition() {
18     return _transposition;
19 }
20
21 void Transposition::transposition(sample new_transposition) {
22     _transposition = new_transposition;
23 }
24
25 sample Transposition::semitones_to_freq(sample semitones) {
26     return pow(TWELFTH_ROOT_2, semitones);
27 }
28
29 void Transposition::process_frame(simpl::Frame* frame) {
30     if(_transposition != 0) {
31         for(int i = 0; i < frame->num_partials(); i++) {
32             frame->partial(i)->frequency *= semitones_to_freq(_transposition);
33         }
34     }
35 }
```

Listing 6.7: The implementation of the `Transposition` class.

6.3.3 Metamorph modules, classes, functions and Csound opcode

Lists of the modules, classes and functions that are available in Metamorph are given in Tables 6.1 and 6.2. Classes begin with capital letters while functions start with lowercase letters.

Python Module	Classes and Functions in Module
metamorph	FX SpectralEnvelope Transformation HarmonicTransformation SpecEnvTransformation ResidualTransformation TransientTransformation Transposition HarmonicDistortion TransientLPF TransientHPF TimeScale

Table 6.1: Metamorph Python module.

Class or Function	Description
FX	Main Metamorph class. Implements the main analysis and synthesis process and provides plugin points so that Transformation classes can modify analysis data before synthesis.
SpectralEnvelope	Code for computing spectral envelopes using the discrete cepstrum envelope method.
Transformation	Abstract base class for Metamorph transformations.
HarmonicTransformation	Abstract base class for Metamorph harmonic transformations.
SpecEnvTransformation	Abstract base class for Metamorph transformations that manipulate the spectral envelope.
ResidualTransformation	Abstract base class for Metamorph residual component transformations.
TransientTransformation	Abstract base class for Metamorph transient region transformations.
Transposition	Class for performing transposition of the sinusoidal component. Derived from HarmonicTransformation.
HarmonicDistortion	Class for performing harmonic distortion of the sinusoidal component. Derived from HarmonicTransformation.
TimeScale	Class for time-scaling of an audio file. Derived from FX.
TransientLPF	Class for low-pass filtering transient signal components. Derived from TransientTransformation.

`TransientHPF`	Class for high-pass filtering transient signal components. Derived from `TransientTransformation`.

Table 6.2: Metamorph classes and functions (available from both C++ and Python).

Metamorph Csound opcode

The syntax of the Metamorph Csound opcode is as follows:

Opcode name: `mm`

Syntax: `aTransformedSignal` **mm** `aInputSignal, kHarmonicScale,`

`kResidualScale, kTransientScale, kPreserveTransients,`

`kTranspositionFactor, kPreserveEnvelope, kHarmonicDistortion, kFundamentalFr`

The opcode input parameters are described in Table 6.3.

6.4 Metamorph examples

This section presents five examples that illustrate how Metamorph can be used to manipulate audio streams using Python, Csound and C++. The Python examples are described first in Sections 6.4.1 and 6.4.2, performing harmonic distortion and time-scaling transformations respectively. Sections 6.4.3 and 6.4.4 show how the Metamorph Csound opcode can be used to transform real-time audio signals. In the final example (Section 6.4.5) a new `Transformation` class is created in order to adjust the spectral envelope of a recorded sample. This example also demonstrates the use of Metamorph from C++ and shows how to create new sound transformations using the library.

Csound opcode parameter	Description
aInputSignal	The input audio signal.
kHarmonicScale	Factor used to scale the harmonic component.
kResidualScale	Factor used to scale the residual component.
kTransientScale	Factor used to scale the transient component.
kPreserveTransients	(0 or 1) Whether to preserve transients or not.
kTranspositionFactor	Transposition amount (in semitones).
kPreserveEnvelope	(0 or 1) Whether to preserve the orignal spectral envelope or not.
kHarmonicDistortion	(0 ... 1) Harmonic distortion amount.
kFundamentalFrequency	Harmonic distortion fundamental frequency.

Table 6.3: Metamorph Csound opcode.

6.4.1 Harmonic distortion

Listing 6.8 shows the application of the Metamorph harmonic distortion transformation to a sampled piano note. On line 7, a Metamorph `FX` object is created. The `residual_scale` parameter is set to `0` on line 8, so the stochastic component will not be present in the synthesised signal. A `HarmonicDistortion` instance is then created on line 10. The two input parameters to the constructor refer to the distortion amount α and the fundamental frequency F_0 in Equation 6.1. The `HarmonicDistortion` instance is added to the `FX` object on line 11. The transformed audio signal is computed on line 13 by calling `fx.process`. This method segments the input audio file into contiguous frames and calls `FX.process_frame`

on each frame in sequence. Finally, the transformed signal is saved to a file called *piano_hdist.wav* on lines 14 and 15.

```
1  import numpy as np
2  import scipy.io.wavfile as wav
3  import metamorph
4
5  audio, sampling_rate = metamorph.read_wav('piano.wav')
6
7  fx = metamorph.FX()
8  fx.residual_scale = 0
9
10 hdist = metamorph.HarmonicDistortion(0, 440)
11 fx.add_transformation(hdist)
12
13 output = fx.process(audio)
14 wav.write('piano_hdist.wav', sampling_rate,
15           np.array(output * 32768, dtype=np.int16))
```

Listing 6.8: Metamorph harmonic distortion transformation using Python.

6.4.2 Time-scaling

As noted in Section 6.2.5, time-scaling is the only Metamorph transformation that can not be performed in real-time because it requires an initial analysis pass. The first uses our real-time note segmentation technique to locate all transient regions in the signal. As these regions will not be time-scaled[3], the overall time-scale factor in non-transient segments must be adjusted so that the final scaled audio signal is of the required duration. Once the time-scale factor for non-transient regions has been calculated, the deterministic and stochastic components are then synthesised, and connected to transient regions using a short cross-fade.

The `TimeScale` transformation is therefore quite different to the other harmonic and stochastic transformations, and cannot be implemented using a Metamorph

[3]Transient regions are not time-scaled by default, although it is possible to change this.

`Transformation` class. Instead, `TimeScale` is derived directly from `FX`. Listing 6.9 shows how this transformation can be used to time-scale a vocal sample by a factor of 3. The `TimeScale` object is created on line 7, the scale factor is set on line 8 and then the scaled signal is computed using a overwritten version of the `process` method on line 9.

```
1  import numpy as np
2  import scipy.io.wavfile as wav
3  import metamorph
4
5  audio, sampling_rate = metamorph.read_wav('vocal.wav')
6
7  ts = metamorph.TimeScale()
8  ts.scale_factor = 3
9  output = ts.process(audio)
10
11 wav.write('vocal_transposed.wav', sampling_rate,
12           np.array(output * 32768, dtype=np.int16))
```

Listing 6.9: Metamorph time-scale transformation using Python.

6.4.3 Real-time synthesis of the stochastic component

The use of the Metamorph Csound opcode is demonstrated in Listing 6.10. It takes real-time input from the sound card (device 0) and outputs audio back to the default output device. The Csound instrument is defined on lines 14-18. `aIn` is set to read audio from input channel 1 on line 15. The Metamorph opcode `mm` is used to synthesise the stochastic component of the input audio stream on line 16 by setting the scale factors for the harmonic and transient components to 0. The resulting signal is saved to `aProcessed` which is output on line 17. Lines 21-23 define a short score that plays this instrument for 60 seconds.

```
<CsoundSynthesizer>

<CsOptions>
-iadc0 -odac -b512 -B2048
</CsOptions>

<CsInstruments>
sr = 44100
kr = 86.1328125
ksmps = 512
nchnls = 1
0dbfs = 1

instr 1
    aIn inch 1
    aProcessed mm aIn, 0, 1, 0
    out aProcessed
endin
</CsInstruments>

<CsScore>
i1 0 60
</CsScore>

</CsoundSynthesizer>
```

Listing 6.10: Using Metamorph to synthesise the stochastic component in real-time using Csound.

6.4.4 Transposition

A second example that uses the real-time Metamorph Csound opcode is given in Listing 6.11, this time transposing the input audio signal. As in the example in Section 6.4.3, input device 0 and the default output device are used for real-time sound input and output. However, different parameters are given to the `mm` opcode on line 16. Only the harmonic component is synthesised by Metamorph as the scale values for the harmonic, noise and transient components are set to 1, 0 and 0 respectively. The fifth parameter is also 0, so transient components will not be preserved in this transformation. The final parameter passes the value of the orchestra macro `$transpose` to Metamorph which sets the desired transposition factor[4]

[4]This value for this parameter is specified in the command-line arguments.

```
<CsoundSynthesizer>

<CsOptions>
-iadc0 -odac -b512 -B2048
</CsOptions>

<CsInstruments>
sr = 44100
kr = 86.1328125
ksmps = 512
nchnls = 1
0dbfs = 1

instr 1
    aIn inch 1
    aProcessed mm aIn, 1, 0, 0, 0, $transpose
    out aProcessed
endin
</CsInstruments>

<CsScore>
i 1 0 60
</CsScore>

</CsoundSynthesizer>
```

Listing 6.11: Using Metamorph to transpose the harmonic component in real-time using Csound.

6.4.5 Spectral Envelope Interpolation

This section shows that the Metamorph C++ library can be used to manipulate the spectral envelope of an audio file. It also demonstrates how to create new effects using Metamorph's analysis data. The first part of the example is given in Listing 6.12. The first three lines include the C++ header files for the *iostream* library, *libsndfile* and the Metamorph library.

```
#include <iostream>
#include <sndfile.hh>
#include "metamorph.h"

using namespace metamorph;


class EnvInterp : public SpecEnvTransformation {
    private:
```

```
10          FX* _fx;
11          int _num_frames;
12          double _interp;
13          double _interp_step;
14          std::vector<double> _env;
15
16      public:
17          EnvInterp(FX* fx, int num_frames) {
18              _fx = fx;
19              _num_frames = num_frames;
20
21              _interp = 0.0;
22              _interp_step = 1.0 / num_frames;
23
24              _env.resize(fx->env_size());
25
26              // create a linear ramp from 0.5 to 0 over the duration
27              // of the envelope
28              for(int n = _env.size() - 1; n >= 0; n--) {
29                  _env[n] = ((double)n / 2.0) / (_env.size() - 1);
30              }
31              fx->apply_envelope(_env);
32          }
33
34          void process_frame(simpl::Frame* frame, std::vector<double>& env) {
35              // gradually change envelope interpolation until the applied
36              // envelope is the linear ramp
37              _interp += _interp_step;
38              _fx->env_interp(_interp);
39          }
40  };
```

Listing 6.12: Using Metamorph to manipulate the spectral envelope of an audio file (lines 1-40).

A definition of a new class called `EnvInterp` begins on line 8, which derives from `SpecEnvTransformation`. The goal of the transformation is to begin synthesising output using the original spectral envelope, but to smoothly interpolate the envelope for a specified time duration so that by the end it is using a new envelope. The constructor for the class is shown on lines 17-32. The input parameters are a pointer to a Metamorph `FX` instance and the desired interpolation duration (in frames). A new spectral envelope is created on lines 24-31 which consists of a linear ramp over the duration of the envelope between 0.5 and 0. This envelope is saved in the `FX` instance on line 31.

The `process_frame` method is defined on lines 34-39. This method is called once for each frame of audio by the `process_frame` method in the `FX` instance, just

before the spectral envelope is applied to the current sinusoidal partials. Lines 37 and 38 slowly adjust the linear interpolation between the original envelope and the new envelope over the duration of the sample.

The main function is shown in Listing 6.13. The paths to the input and output files are passed as command line parameters, so the number of supplied parameters is checked on lines 44-48. A libsndfile `SndfileHandle` is created on line 50, which is used to read the input audio file. A Metamorph `FX` object is instantiated on line 56. The hop size and frame size are set on lines 57 and 58 respectively, then on line 59 spectral envelope preservation is set to `true` so that the spectral envelope of the input sound will be calculated and then applied (possibly with modifications) before synthesis of the deterministic component. An instance of the new `EnvInterp` class is created on line 61 and then added to the `FX` object on line 62. On lines 69-71 the `fx.process_frame` method is called on each contiguous frame in the input signal. The output waveform is saved to the `std::vector` of samples called `output`, which is then written to the output file using libsndfile on lines 77-82.

```
43  int main(int argc, char* argv[]) {
44      if(argc != 3) {
45          std::cout << "Usage: " << argv[0] << " <input file> <output file>";
46          std::cout << std::endl;
47          exit(1);
48      }
49
50      SndfileHandle input_file = SndfileHandle(argv[1]);
51      int num_samples = (int)input_file.frames();
52      int hop_size = 512;
53      int frame_size = 2048;
54      int num_frames = (num_samples - frame_size) / hop_size;
55
56      FX fx;
57      fx.hop_size(hop_size);
58      fx.frame_size(frame_size);
59      fx.preserve_envelope(true);
60
61      EnvInterp env_interp = EnvInterp(&fx, num_frames);
62      fx.add_transformation(&env_interp);
63
```

```
64      std::vector<double> input(num_samples);
65      std::vector<double> output(num_samples);
66
67      input_file.read(&(input[0]), num_samples);
68
69      for(int n = 0; n < (num_samples - hop_size); n += hop_size) {
70          fx.process_frame(hop_size, &(input[n]), hop_size, &(output[n]));
71      }
72
73      int output_format = SF_FORMAT_WAV | SF_FORMAT_PCM_16;
74      int output_channels = 1;
75      int output_sampling_rate = 44100;
76
77      SndfileHandle output_file = SndfileHandle(argv[2],
78                                                SFM_WRITE,
79                                                output_format,
80                                                output_channels,
81                                                output_sampling_rate);
82      output_file.write(&(output[0]), num_samples);
83
84      return 0;
85  }
```

Listing 6.13: Using Metamorph to manipulate the spectral envelope of an audio file (lines 43-85).

6.5 Conclusions

This chapter introduced Metamorph, a software library which provides a new environment for performing high-level sound manipulation. It is based on a real-time sinusoids plus noise plus transients model, combining the sinusoidal modelling, onset detection and note segmentation libraries that were discussed in chapters 3, 4 and 5 respectively. Metamorph includes a collection of flexible and powerful sound transformations: harmonic distortion, adjusting the noisiness and transience of an audio signal, manipulating the spectral envelope before synthesising the harmonic component, transposition (with and without spectral envelope preservation), time-scaling and transient processing. An overview of the design and implementation of Metamorph was presented in Section 6.3, which included a discussion of how additional transformations can be added to Metamorph in order to extend the

functionality of the library. Section 6.4 described five examples that demonstrated how Metamorph could be used to manipulate audio streams using Python, C++ and Csound.

6.5.1 Metamorph in comparison to existing tools for spectral modelling and manipulation of sound

Section 2.7 provided an overview of existing software packages for spectral processing of audio signals. These software packages were further categorised into two groups; systems that specialised in spectral processing, and general purpose systems that contained tools for spectral manipulation. As Metamorph belongs in the former category, some comparisons with the other software tools in the group will be made in this section.

Metamorph does not include a GUI for viewing and editing spectral data unlike ATS, SPEAR and AudioSculpt. It also does not directly support importing or exporting sinusoidal partial data to common file formats such as SDIF. However, Metamorph is available as a Python extension module and it integrates well with Python's libraries for scientific computing, and so both of these tasks can be performed using a relatively small amount of Python code. This is evident when looking at the code required to plot sinusoidal analysis data using the Simpl library in Section 3.4.2.

The majority of the software packages in the specialised tools category integrate well with one or more of the general purpose signal processing applications that are listed in Section 2.7.2. Like Loris, Metamorph is available as a Csound opcode, allowing it to be used with a powerful system for audio signal processing

and sound design. Similarly, Libsms can be used with Pure Data, ATS can be used with SuperCollider, Csound and Pure Data, and SuperVP is available as an external object for Max/MSP.

However, only Metamorph and SuperVP manipulate audio signals in real-time using a model that is based on separating sounds into deterministic, stochastic and transient components. Metamorph and SuperVP both include a means of estimating and manipulating the spectral envelope of the analysed sound. Both are also modular systems, but unlike SuperVP, Metamorph includes a series of "plugin" points. This enables developers to create new transformations by altering the data that is created at every stage of the analysis process. Another important distinction between the two systems is that Metamorph is open source software, which allows other researchers and developers to extend the current functionality and also makes it useful as a teaching tool. The additional note segmentation phase in the Metamorph model could also be used to improve the quality of existing sound morphing techniques. Caetano and Rodet showed that by considering the temporal evolution of a sound source, morphing effects can be created that are judged to be more "perceptually meaningful" than methods that simply interpolate spectral parameters [20].

Chapter 7

Conclusions

The preceding chapters documented the design and implementation of new open source software tools for sound manipulation. The main motivation behind the development of these systems was to enable composers, musicians and researchers to perform flexible and intuitive transformations on live audio streams, with a particular focus on manipulating the timbres of live musical sounds. As the perception of timbre is closely linked to the temporal evolution of the spectrum, the sinusoidal model was chosen as the underlying synthesis by analysis system. There are well-documented weaknesses with sinusoidal models however:

1. They are not ideally suited to providing meaningful control of signals that contain noise-like components.

2. As sinusoidal models are based on the assumption that the underlying signal components will evolve slowly in time, the quality of the synthesis of transient components can be compromised. This is particularly problematic during the attack region of musical sounds as these regions are perceptually salient.

3. Sinusoidal models can potentially suffer from the problem of having too many control parameters to allow the synthesised sounds to be manipulated in a meaningful manner.

Addressing the first problem lead to the development of sinusoids plus noise models of sound such as Spectral Modelling Synthesis and the bandwidth-enhanced model. The open source software packages *libsms* and *Loris* can be used to analyse and synthesise sounds using these two sinusoidal models. However, when working with sinusoidal models it is often desirable to have an expressive, interactive system that facilitates rapid-prototyping of ideas, and with a set of tools for performing standard signal processing techniques and data visualisation. The experience of experimenting with different sound models can also be simplified if it is possible to interact with analysis data and parameters in a consistent manner.

The appeal of such a system motivated the development of the Simpl sinusoidal modelling library. It is an open source software library that is written in C++ and Python. It provides a consistent API for accessing implementations of Spectral Modelling Synthesis, the bandwidth-enhanced model, the sinusoidal modelling system that is part of the SndObj library and an implementation of the McAulay-Quatieri method. Analysis and synthesis can be performed on pre-recorded audio samples or on streams of audio frames (as long as this is supported by the underlying sinusoidal modelling algorithm). When used with Python, Simpl uses NumPy arrays to store audio samples, making it trivial to integrate the functionality provided by the library with the SciPy suite of software for scientific computing. Simpl also includes a module that visualises sinusoidal analysis data using Matplotlib. Simpl is discussed in Chapter 3, which provides an overview of the software library followed

by an in-depth look at the implementation of one of the peak detection modules. The chapter concludes with some examples that demonstrate the functionality of the library.

In order to improve the synthesis of note attack regions it is necessary to be able to accurately locate and identify these transient signal components. The attack portion of a musical note is usually taken to be a region (possibly of varying duration) that immediately follows a note onset, and so the first stage in our solution to this problem was to detect note onset locations. Chapter 4 describes Modal, a new open source software library for real-time onset detection, written in C++ and Python. It includes implementations of onset detection functions from the literature that are based on measuring frame-by-frame variations in signal energy, average spectral bin magnitude values, and a combination of spectral magnitude and frequency values. Modal also includes a database of 71 samples that have creative commons licensing which allows them to be freely redistributed. Each sample has associated metadata containing hand-annotated onset locations for each note in the sample. The database contains 501 onsets in total.

The Modal sample database was used to evaluate the detection accuracy of all of the onset detection functions. We then show how the use of linear prediction can improve the detection accuracy of the literature methods, but this improvement comes at the expense of much greater computational cost. A novel onset detection function that is based on measuring changes between spectral peak amplitude values in sinusoidal partials is then presented, which is shown to be more accurate than the literature methods while having similar computational performance characteristics.

Chapter 5 describes a method for detecting the duration of the attack transient region that follows a note onset. However in addition to locating attack regions, it is

desirable to be able to accurately identify other temporal sections in a musical note that have distinct characteristics. Therefore Chapter 5 presented a new technique for the real-time automatic temporal segmentation of musical sounds. Attack, sustain and release segments were defined using cues from a combination of the amplitude envelope, the spectral centroid, and a measurement of the stability of the sound that is derived from an onset detection function. This segmentation method is then compared to an implementation of a leading non-real-time technique from the literature and it is shown to generally be an improvement. The note segmentation software is free software, also written in C++ and Python.

The Simpl sinusoidal modelling library, Modal onset detection library and note segmentation library are then combined in a real-time sound synthesis by analysis tool called Metamorph which is described in Chapter 6. Metamorph is an open source software library for performing high-level sound transformations based on a sinusoids plus noise plus transients model. It is written in C++ and can be built as both a Python extension module and a Csound opcode. The chapter examines the sinusoids plus noise plus transients model that is used in Metamorph before describing each of the sound transformations that can be performed using the library: harmonic distortion, adjusting the noisiness and transcience of an audio signal, manipulating the spectral envelope before synthesising the harmonic component, transposition (with and without spectral envelope preservation), time scaling and transient processing. The implementation of Metamorph is then discussed, followed by examples of transformations performing using Metamorph from C++, Python and Csound.

7.1 Discussion and suggestions for future work

The Simpl library provides a consistent API for interacting with some leading sinusoidal modelling implementations from the literature. Simpl adds another layer of abstraction on top of the existing implementations, however the C++ library is still suitable for real-time analysis and synthesis. As sinusoidal models have been an active research area for several decades there are more analysis models that could be implemented and added to the library [94, 8, 26, 73, 121, 98, 62, 63]. The majority of this research has focused on the identification and modelling of the harmonic content in an audio signal, but there is potentially still room for improvement in the modelling and synthesis of the residual component. It would also be of great benefit to see a comprehensive evaluation of the performance of sinusoidal analysis and synthesis methods, and indeed some progress towards such an evaluation has occurred in recent years [61, 87].

The topic of real-time onset detection was discussed in Chapter 4. The performance of seven onset detection functions is compared using a set of reference samples (the Modal database). Our linear prediction and peak amplitude difference methods were shown to perform better than techniques from the literature, however the composition and size of the sample set must be taken into account. The samples are quite diverse in nature, consisting of percussive and non-percussive sounds from a mixture of western orchestral instruments, contemporary western instruments, electronic sounds and vocal samples. Ideally the size of the database would be larger in order to increase the confidence in the results. However, the ability to freely distribute both the samples, their associated metadata, and all of the analysis code was deemed to be extremely important as it enables other researchers

to reproduce the results and to potentially improve upon the proposed methods. At the time of publication there are no other known publicly available onset detection resources that include these three components, but hopefully more sample sets and source code will be made available in future which will make it possible to extend Modal.

Our real-time note segmentation method was introduced in Chapter 5, and is shown to compare favourably to a leading non-real-time technique from the literature. Both methods identify note onset, attack and sustain locations with a high degree of accuracy. Neither method was particularly accurate in detecting the positions of release and offset events however, and so both region boundaries would be good targets for potential improvement.

Chapter 6 introduced a new open source software tool for real-time sound manipulation called Metamorph, consisting of six transformations that are based on high-level parameters [109]. Improvements could be made to the existing code base in order to expose some of the model parameters that are currently hard-coded. For example, there is a minimum time that must elapse between the detection of consecutive onsets (currently set to 200 ms), which limits the number of notes per second that can be accurately transformed by the model. Ideally, the user should be able to adjust this value depending on the nature of the input sound. The number of model parameters that are available in the Csound opcode could also be increased, as some of the existing C++ parameters cannot currently be controlled from Csound.

There are more spectral transformations from the literature that could be added to the library [4]. The ability to manipulate spectral envelopes that are created using the discrete cepstrum envelope has proven to be very useful, but a thorough evaluation of the discrete cepstrum envelope and the True Envelope would certainly be

of interest to the research community. Using these spectral envelopes to manipulate the sound spectrum can definitely reduce the number of parameters that must be adjusted when creating sound modifications, but much work still remains in the quest to create meaningful ways of interacting with these envelopes, particularly in a real-time musical performance situation.

7.2 Closing remarks

It was noted in the opening chapter that the appeal of additive synthesis and the promise of a general-purpose sound model that can provide flexible and intuitive control of transformations has proven hard to resist for many researchers. While there are still improvements that can be made in order to increase the perceived quality of real-time sinusoidal synthesis by analysis methods, and to provide perceptually intuitive means of controlling the real-time synthesis process, it is hopefully clear that this research represents an important step towards achieving these goals. A strong emphasis was placed on providing free, open-source software libraries, enabling the results from each stage of the research to be independently evaluated and reproduced, and ideally leading to a wider dissemination of the work.

Appendices

Appendix A

Contents of the accompanying data CD

The thesis comes with a data CD that contains the source code for the software that is discussed in Chapters 3, 4, 5 and 6. A set of audio examples are also included. The CD contains the following directories:

simpl: The source code to the Simpl sinusoidal modelling library (discussed in Chapter 3).

modal: The source code to the Modal real-time onset detection library (discussed in Chapter 4).

modal_evaluation: The code used to evaluate the onset detection functions that are implemented in Modal.

notesegmentation: The source code to the Note Segmentation library (discussed in Chapter 5).

notesegmentation_evaluation: The code used to evaluate the performance of our real-time note segmentation method and our implementation of the Caetano et al. method.

metamorph: The source code to Metamorph (discussed in Chapter 6).

audio_examples: A collection of audio examples that were created using Simpl and Metamorph.

The most recent versions of the four software libraries can be found online at the author's website: http://www.johnglover.net.

Appendix B

Simpl: A Python library for sinusoidal modelling

Original publication:

John Glover, Victor Lazzarini, and Joseph Timoney. Simpl: A Python library for sinusoidal modelling. In *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, Como, Italy, September 2009.

SIMPL: A PYTHON LIBRARY FOR SINUSOIDAL MODELLING

John Glover, Victor Lazzarini, Joseph Timoney

The Sound and Digital Music Research Group
National University of Ireland, Maynooth
Ireland
John.C.Glover@nuim.ie
Victor.Lazzarini@nuim.ie
JTimoney@cs.nuim.ie

ABSTRACT

This paper introduces Simpl, a new open source library for sinusoidal modelling written in Python. The library is presented as a resource for researchers in spectral signal processing, who might like to access existing methods and techniques. The text provides an overview of the design of the library, describing its data abstractions and integration with other systems. This is complemented by some brief examples exploring the functionality of the library.

1.INTRODUCTION

Simpl is an open source library for sinusoidal modelling [1] written in the Python programming language [2] and making use of Scientific Python (SciPy) [3]. The aim of this project is to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API, as well as providing implementations of some recently published sinusoidal modelling algorithms, many of which have yet to be released in software.

Simpl is primarily intended as a tool for other researchers in the field, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms. There are currently several open source software projects that either include or are dedicated solely to sinusoidal modelling such as PARSHL [4], the Sound Object Library [5], Csound [6], Loris [7], CLAM [8], libsms [9] and SAS [10]. All of these systems exist as separate entities, and due to their internal workings it can often be awkward to exchange analysis data between them for comparison. However, they generally share common ideas, terminology and abstractions (such as the concepts of spectral peaks and partial tracking). Simpl allows these abstract data types to be exchanged between different underlying implementations. For example, one might wish to compare the sinusoidal peaks detected with the SMS algorithm with those found by the Loris implementation. Due to the flexible, modular design of Simpl this sort of operation is straightforward. Simpl analysis/synthesis is able to render audio files in non-real-time as well as operate in real-time streaming mode, as long as the underlying algorithms are able to do so.

1.1.Sinusoidal Modelling

Sinusoidal modelling is based on Fourier's theorem, which states that any periodic waveform can be modelled as the sum of sinusoids at various amplitudes and harmonic frequencies. For stationary pseudo-periodic sounds, these amplitudes and frequencies evolve slowly with time. They can be used as parameters to control pseudo-sinusoidal oscillators, commonly referred to as *partials*. The audio signal s can be calculated from the sum of the partials using:

$$s(t)=\sum_{1}^{N_p} A_{p(t)}\cos(\theta_{p(t)}) \quad (1)$$

$$\theta_{p(t)}=\theta_{p(0)}+2\pi\int_{0}^{t} f_{p(u)}\,du \quad (2)$$

where N_p is the number of partials and A_p, f_p and Θ_p are the
amplitude, frequency and phase of the p-th partial respectively.
Typically, the parameters are measured for every:

$$t=nH/F_s \quad (3)$$

where n is the sample number, H is the hop size and F_s is the sampling rate. To calculate the audio signal, the parameters must then be interpolated between measurements. Calculating these parameters

for each frame is referred to in this document as *peak detection*, while the process of connecting these peaks between frames is called *partial tracking*.

In [11] McAulay and Quatieri proposed to represent a speech signal as a sum of sinusoids with time-varying amplitude, frequency and phase. While it is possible to model noisy signals with sinusoids, it is not very efficient, as large numbers of partials are often required. It is also not particularly meaningful, and does not seek to represent the underlying structure of the sound.

Serra and Smith extended this idea in [12], making a distinction between the pseudo-periodic or *deterministic* component of a sound and the more noise-like or *stochastic* component, modelling and synthesising the two components separately. Fitz and Haken keep this distinction in [13], but use *bandwidth-enhanced* oscillators to create a homogeneous additive sound model.

Later advances and refinements in the field have mostly been in the details of the analysis algorithms, in particular in the peak detection and partial tracking processes. A good overview of peak detection techniques can be found in [14]. In [15] Depalle and Rodet use the Hidden Markov Model to improve partial tracking, while in [16] Lagrange et al achieve this using Linear Prediction.

1.2.SciPy

SciPy is a cross-platform, open source software package for mathematics, science and engineering. It depends on NumPy [17], which provides fast array processing. It has a syntax that is very similar to Matlab [18], with implementations of many of Matlab's functions: it contains packages for matrix manipulation, statistics, linear algebra as well as signal processing. SciPy also supports Matlab-style plotting and visualisation of data through the Matplotlib [19] language extension. The vast library of functions combined with the readability and power of the Python language make SciPy a great tool for quick prototyping as well as for the development of larger applications.

2.THE SIMPL LIBRARY

Simpl is an object-orientated Python library for sinusoidal modelling. Spectral data is represented by two main object types: Peak (represents a spectral peak) and Partial. A Partial is basically just an ordered collection of Peak objects.

Simpl includes a module with plotting functions that use Matplotlib to plot analysis data from the peak detection and partial tracking analysis phases, but generating additional plots is trivial using Matplotlib's Matlab-like interface.

All audio in Simpl is stored in NumPy arrays. This means that SciPy functions can be used for basic tasks such as reading and writing audio files, as well as more complex procedures such as performing additional processing, analysis or visualisation of the data.

Each supported analysis/synthesis method has associated wrapper objects that allows it to be used with Simpl Peak and Partial data, which facilitates the exchange of information between what were originally unrelated sinusoidal modelling systems. The implementations that are currently supported are the Sound Object Library, Spectral Modelling Synthesis (SMS, using libsms) and Loris. Additionally, the following algorithms are included: McAulay-Quatieri (MQ) analysis and synthesis as given in [11], partial tracking using the Hidden Markov Model (HMM) as detailed in [15] and partial tracking using Linear Prediction (LP) as detailed in [16].

Currently in Simpl the sinusoidal modelling process is broken down into three distinct steps: peak detection, partial tracking and sound synthesis. Python objects exist for each step, which all of the analysis/synthesis wrapper objects derive from. Each object has a method for real-time interaction as well as non-real-time or batch mode processing, as long as these modes are supported by the underlying algorithm. For any given step, every analysis/synthesis object returns data in the same format, irrespective of its underlying implementation. This allows analysis/synthesis networks to be created in which the algorithm that is used for a particular step can be changed without effecting the rest of the network. The process is summarised in figure 1.

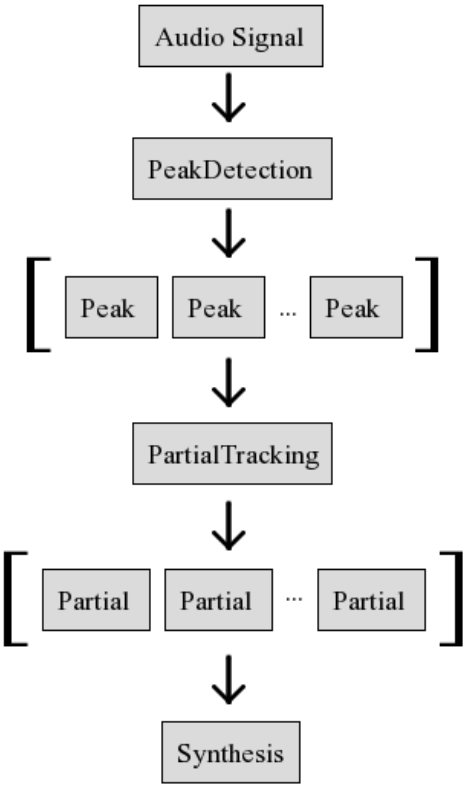

Figure 1: *Simpl analysis-synthesis process*

2.1.Peak Detection

PeakDetection objects take a NumPy array of audio samples as input. This can be just a single frame of audio, or a longer signal of arbitrary length that will be cut up into frames for further processing internally and zero padded if necessary. For each frame, spectral peaks are calculated. If the input was a single audio frame, then a single list of Peak objects is returned. If it was a longer signal, a separate list of Peaks is returned for each audio frame.

2.2.Partial Tracking

The input to PartialTracking objects is either a list of Peaks or an arbitrary number of lists of Peaks. This information is used by the partial tracking algorithm to form Partial objects, which are ordered lists of Peaks. The return value is always a list of Partials.

2.3.Sound Synthesis

SoundSynthesis objects take a list of Partials and a NumPy array of audio samples (the original signal) as input. They use this data in various ways depending on the synthesis algorithm, but the general process is to use the Partial data to synthesise the harmonic (deterministic) sound component, then subtract this from the original signal in order to obtain the residual (stochastic) component. All derived objects can return a fully synthesised signal, as well as these two components in isolation if supported. For example, the MQ algorithm does not make this distinction and between components and so the MQSoundSynthesis object returns a synthesised signal based only on the Partial data. SMSSoundSynthesis on the other hand can return all three signal types. Audio signals are returned as NumPy arrays.

3.EXAMPLES

In this section we will present three examples, demonstrating the system. The first deals with the basic manipulation of audio data using SciPy. The next two provide examples of the Simpl library proper for two basic tasks of analysis-synthesis and spectral display.

3.1 Using SciPy

The following example shows how SciPy can be used to read in an audio file called piano.wav and plot it using Matplotlib. The resulting plot is displayed in figure 2.

```
from scipy.io.wavfile import read
from pylab import plot, xlabel,
ylabel, \
                  title, show

input_data = read('piano.wav')

# store samples as floats between
-1 and 1
audio_samples = input_data[1] /
32768.0

# plot the first 4096 samples
plot(audio_samples[0:4096])
ylabel('Amplitude')
xlabel('Time (samples)')
title('piano.wav')
show()
```

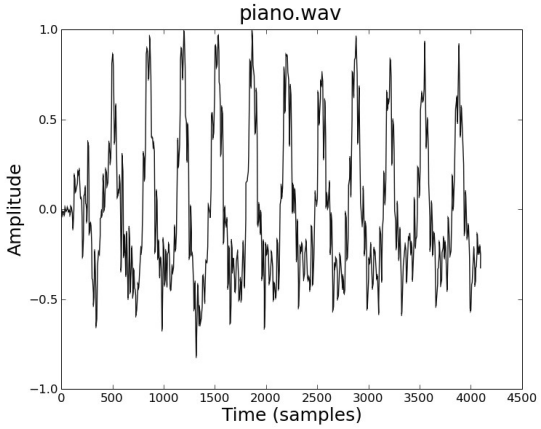

Figure 2: *Resulting waveform plot*

3.2.Using Simpl

This data can now be passed directly to the Simpl analysis objects. In the following example, peak detection is performed using the Sound Object Library, followed by partial tracking from the MQ algorithm before finally the sound is resynthesised using SMS. All operations are performed in non-real-time.

```
from scipy.io.wavfile import read
from scipy import asarray, float32
from SimplSndObj import SndObj-
PeakDetection
from SimplMQ import MQPartial-
Tracking
from SimplSMS import SMSSynthesis
input_data = read('piano.wav')
# store audio samples as 32-bit
floats,
# with values between -1 and 1
audio_samples =
asarray(input_data[1], \
   float32) / 32768.0

# This detects peaks using the
SndObj lib
# and stores them in a numpy array
pd = SndObjPeakDetection()
peaks = pd.find_peaks(audio_sam-
ples)

# Here we have partial tracking
using
# McAulay-Quatieri method
pt = MQPartialTracking()
partials = pt.find_partials(peaks)
# finally we synthesise the audio
# using SMS
synth = SMSSynthesis()
# our detected partials will be
used to
# form the harmonic component, and
the
# original audio signal will be
used when
# calculating the residual
audio_out =
synth.synth(partials, \
   audio_samples)
```

3.3 Simpl Data Visualisation

Partial tracking data can be displayed using the Simpl plotting module. The plot produced by this example is shown in figure 3.

```
from scipy.io.wavfile import read
from scipy import asarray, float32
from SimplSndObj import SndObj-
PeakDetection
from SimplMQ import MQPartial-
Tracking
from SimplPlots import plot_par-
tials

# read audio data
input_data = read('piano.wav')
audio_samples =
asarray(input_data[1], \
   float32) / 32768.0

# detect up to a maximum of 20
peaks. If
# there are more, the 20 with the
largest
# amplitudes will be selected
pd = SndObjPeakDetection()
pd.max_peaks = 20
peaks = pd.find_peaks(audio_sam-
ples)

# track peaks
pt = MQPartialTracking()
partials = pt.find_partials(peaks)
# display them
plot_partials(partials)
show()
```

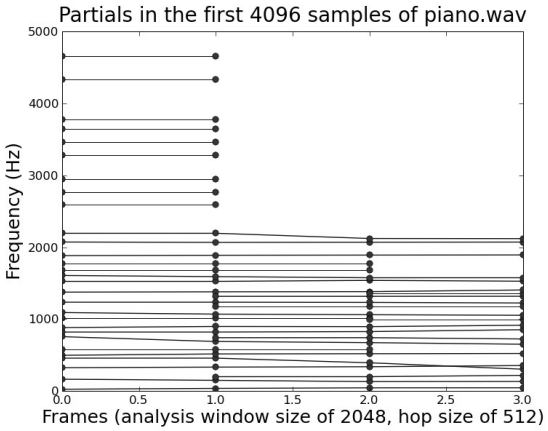

Figure 3: *Plot of partial data. Circles represent peaks, lines show the resulting partials. Some extra peaks were created by the MQ partial tracking algorithm, for partial 'birth' and 'death'.*

4. FUTURE WORK

More visualisation functions will be added to the library. In particular, we want to add the ability to display data during real-time analysis. The current plotting functions are not efficient enough to achieve this. It is expected that more analysis/synthesis algorithms will be added to the library, adapted from the many published papers in the field. We would also like to add the ability to control algorithm parameters in real-time using OpenSound Control [20]. Developers are encouraged to contribute to the project, and can contact the authors via email.

5.CONCLUSION

Simpl provides a new environment for developing sinusoidal modelling applications, unifying several of the existing solutions in addition to implementing some of the most important advances in the field. Together with the flexibility of Python and the extensive range of SciPy functions, Simpl should be a valuable tool for other researchers and developers.

Simpl is free software, available under the terms of the GNU GPL. To download it or for more information go to:
http://simplsound.sourceforge.net

6.ACKNOWLEDGEMENTS

The authors would like to acknowledge the generous support of An Foras Feasa, who funded this research.

7.REFERENCES

[1] X. Amatriain, J. Bonada, A. Loscos, X. Serra, "DAFX - Digital Audio Effects", chapter Spectral Processing, pp 373-438, Udo Zölzer Ed, John Wiley & Sons, Chichester, UK, 2002.

[2] G. Van Rossum, F. Drake, "The Python Language Reference Manual", Network Theory, Bristol, UK, 2006.

[3] E. Jones, T. Oliphant, P. Peterson and others, "SciPy: Open Source Scientific Tools for Python", http://www.scipy.org, accessed April 6, 2009.

[4] J. Smith, X. Serra, "PARSHL: An Analysis/Synthesis Program for Non-Harmonic Sounds Based on a Sinusoidal Representation." Proceedings of the International Computer Music Conference (ICMC), San Francisco, USA, 1987.

[5] V. Lazzarini, "The Sound Object Library", Organised Sound 5 (1), pp 35-49, Cambridge University Press, Cambridge, UK, 2000.

[6] J. Ffitch, "On the Design of Csound5", Proceedings of the 3rd Linux Audio Conference (LAC), pp. 37-42, ZKM, Karlsruhe, Germany, 2005.

[7] K. Fitz, L. Haken, S. Lefvert, M. O'Donnel, "Sound Morphing using Loris and the Reassigned Bandwdith-Enhanced Additive Sound Model: Practice and Applications", Proceedings of the International Computer Music Conference, Gotenborg, Sweden, 2002.

[8] X. Amatriain, P. Arumi, D. Garcia,"CLAM: A Framework for Efficient and Rapid Development of Cross-platform Audio Applications", Proceedings of ACM Multimedia, Santa Barbara, California, USA, 2006.

[9] R. Eakin, X. Serra, "libsms Library for Spectral Modeling Synthesis" http://www.mtg.upf.edu/static/libsms/, accessed April 06, 2009.

[10] M. Desainte-Catherine, S. Marchand, "Structured Additive Synthesis: Towards a Model of Sound Timbre and Electroacoustic Music Forms", Proceedings of the International Computer Music Conference (ICMC), pp. 260-263, Beijing, China, 1999.

[11] R. McAulay, T. Quatieri, "Speech Analysis/Synthesis Based on a Sinusoidal Representation", IEEE Transaction on Acoustics, Speech and Signal Processing, vol. 34, no. 4, pp. 744–754, 1986.

[12] X. Serra, J. Smith, "Spectral Modeling Synthesis A Sound Analysis/Synthesis Based on a Deterministic plus Stochastic Decomposition", Computer Music Journal, Vol. 14, No. 4 (Winter), 12-24, 1990.

[13] K. Fitz, "The Reassigned Bandwidth-Enhanced Method of Additive Synthesis", Ph. D. dissertation, Dept. of Electrical and Computer

Engineering, University of Illinois at Urbana-Champaign, USA, 1999.

[14] F. Keiler, S. Marchand, "Survey on Extraction of Sinusoids in Stationary Sounds", Proceedings of the 5th International Conference on Digital Audio Effects (DAFx), Hamburg, Germany, 2002.

[15] P. Depalle, G. Garcia, X. Rodet, "Tracking of Partials for Additive Sound Synthesis Using Hidden Markov Models", Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Minneapolis, Minnesota, USA, 1993.

[16] M. Lagrange, S. Marchand, M. Raspaud, J. Rault, "Enhanced Partial Tracking Using Linear Prediction ", Proceedings of the 6th International Conference on Digital Audio Effects (DAFx), London, UK, 2003.

[17] T. Oliphant, "Guide to NumPy", http://numpy.scipy.org/numpybook.pdf, accessed April 06, 2009.

[18] The MathWorks, "MATLAB - The Language of Technical Computing", http://www.mathworks.com/products/matlab, accessed April 06, 2009.

[19] J. Hunter and others, "Matplotlib - Python Plotting", http://matplotlib.sourceforge.net/, accessed April 06, 2009.

[20] M. Wright, A. Freed, A. Momeni, "OpenSound Control: State of the Art 2003", Proceedings of the Conference on New Interfaces for Musical Expression (NIME), Montreal, Canada, 2003.

Appendix C

Sound manipulation using spectral modeling synthesis

Original publication:

John Glover, *The Audio Programming Book*, Richard Boulanger and Victor Lazzarini (ed.), chapter 19. The MIT Press, 2010.

DVD Chapter 19 – *John Glover*

Sound Manipulation Using Spectral Modeling Synthesis

1. Introduction

There are many ways that sound can be modeled and in turn recreated by computers, ranging from mathematical models such as Frequency Modulation (FM) synthesis[1], to time-based techniques such as Granular synthesis[2], to physical models that aim to simulate the acoustic properties of a sound source[3]. All of these techniques are extremely powerful and each one can in theory be used to create any possible sound, but one of the biggest problems that still remains in sound synthesis is how to maintain the flexibility to create a wide variety of sound timbres and yet provide an intuitive means of controlling the synthesis process, preferably in real-time. Recreating specific sounds using FM synthesis is generally not an intuitive task, often requiring a lot of trial and error, although there have been recent improvements in this area[4]. Granular synthesis can require specifying hundreds of parameters even for short segments of sound, making simple yet precise real-time control difficult. Physical modeling can offer control parameters based on the properties of the modeled instrument, which can be very intuitive. For example, a physically modeled guitar may offer controls such as string thickness or wood type. However, physical models are by nature very specific to a particular sound creation process, so creating an arbitrary sound may require constructing a completely new physical model. This is not a trivial task, and may be time-consuming and unintuitive if it is not based on an existing physical instrument.

An alternative to the above sound models is to use a frequency domain or spectral model, which represents audio signals as a sum of sine waves with different frequencies and amplitudes. This representation is somewhat similar to the human hearing system, and in comparison to physical modeling it models the sound that arrives at the ear rather than the sound emitted by the source. Research shows that the perception of timbre is largely dependent on the temporal evolution of the sound spectrum[5], and so it seems natural to use a sound model that is based on the frequency spectrum as a tool to manipulate timbre. Spectral models provide ways to transform audio signals that can be perceptually and musically intuitive, although like granular synthesis they can suffer from the problem of having too many parameters to allow for meaningful real-time control.

This chapter focuses specifically on spectral transformations using a technique

called *Spectral Modeling Synthesis*[6] (SMS), showing how an open source C library called *libsms*[7] can be used to manipulate sound files in a variety of ways. Section 2 begins by giving a brief overview of SMS, and then a simple example is given that demonstrates how libsms can be used to recreate a given input sound. This will serve as the basic template for later examples. In Section 3 we look in detail at each step in the SMS algorithm, focusing on how these spectral processing concepts are implemented in libsms. It may help if you are familiar with the section on *Spectral Processing* (from Book Chapters 7-9) before reading it, as many of the ideas here are based on these basic principles. In Section 4, detailed examples are given that show how we can use libsms to transform the input sound. In particular, we show how the pitch of a sound can be changed without changing its duration, how the duration of the sound can be changed without altering the pitch and how we can use spectral envelope information to create a hybrid of two input sounds.

2. Spectral Modeling Synthesis

Additive synthesis is based on Fourier's theorem, which basically says that any sound can be modeled as the sum of sinusoids at various amplitudes and harmonic frequencies. For sounds with a lot of harmonic content, which includes a lot of musical sounds, these amplitudes and frequencies evolve slowly with time. They can be used as parameters to control sinusoidal oscillators that synthesize individual sinusoidal components or *partials*. The audio signal *s* can be calculated from the sum of the partials using:

$$s(t) = \sum_{1}^{N_p} A_{p(t)} \cos\left(\theta_{p(t)}\right) \tag{1}$$

$$\theta_{p(t)} = \theta_{p(0)} + 2\pi \int_{0}^{t} f_{p(u)}\, du \tag{2}$$

where N_p is the number of partials and A_p, f_p and θ_p are the amplitude, frequency and phase of the *p*-th partial respectively. The problem is that using additive synthesis to create many musically interesting sounds (such as a piano note for example) may require large numbers of sinusoids, each with individual frequency, amplitude and phase parameters. These parameters could be found by hand using trial and error, but as this is extremely time-consuming it is usually preferable to find these parameters by doing some sort of analysis on recorded sound files.

The Short-Time Fourier Transform[8] (STFT) is the usual starting point for sinusoidal analysis, providing a list of bin number, amplitude and phase parameters for each frame of analyzed audio. This principle can be extended, allowing us to track the individual frequencies that are present in the sound. The *phase vocoder*[9] has been used quite successfully for this purpose since the 1970's. However, it does have problems. Firstly, as it uses a fixed-frequency filter bank, the frequency of each sinusoid cannot normally vary outside of the bandwidth of its channel. As the phase vocoder is really set up for analyzing harmonic sounds, it can be inconvenient to use it to work with inharmonic sounds. It also represents the entire audio signal using sinusoids, even if it includes noise-like elements, such as the key noise of a piano for example or the breath noise in a flute note. This is not a problem for straight re-synthesis of the sound, but if any transformation is applied (such as time-stretching), these noisy components are modified along with the harmonic content, often resulting in audible artifacts. Modeling noisy components with sinusoids is computationally expensive, as theoretically recreating noise requires sinusoids at every frequency within the band limits. A sinusoidal representation of noise is also unintuitive and does not provide an obvious way manipulate the sound in a meaningful way.

Two similar systems were developed independently in the 1980's that overcame the phase vocoder problems of sinusoids varying outside the channel bandwidth and analyzing inharmonic sounds, *PARSHL*[10] and another system designed for speech analysis by McAulay and Quatieri[11]. Both work by performing a STFT on the input audio signal, then selecting the most prominent spectral *peaks* (sinusoidal components with the largest magnitudes) that are tracked from frame to frame. These peaks are interpolated across frames to form partials, which may or may not be harmonically related.

This still left the problem of how to model and manipulate noisy sound components. SMS addresses this problem by breaking an audio signal down into two components: a harmonic component (that will be represented by a sum of sinusoids) and a noise or residual component (that will be represented by filtered noise). First, a similar scheme to that used in PARSHL and by McAulay and Quatieri is used to find and model any harmonic content from the sound signal. The harmonic component is then synthesized using this information and subtracted from the original signal, leaving a noise-like residual sound that can then be modeled separately. These two parts of the sound can be manipulated independently and then recombined to create the final synthesized sound. The entire process is summarized in Figure 1.

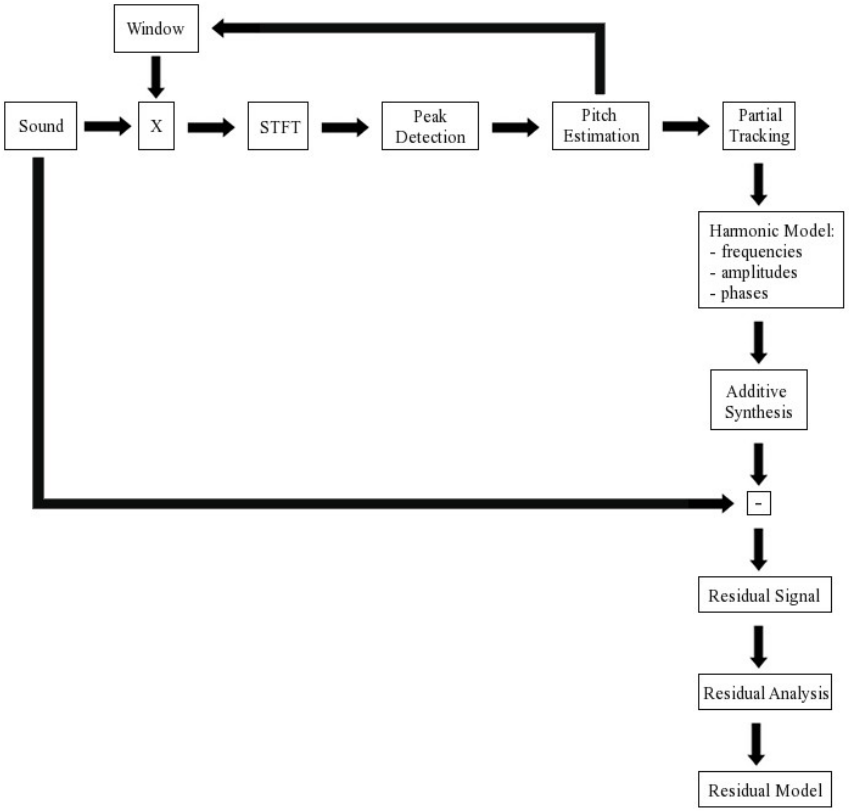

Figure 1 SMS analysis/synthesis.

2.1 A First Look At Libsms

This section looks at how libsms can be used to make a C program (*analysis_synthesis.c*) that performs SMS analysis and synthesis of a given input audio file. No transformations are applied to the sound, so the output should sound very similar to the input, but this program will serve as the template that later examples will build upon. The full source code and build instructions can be found in the *examples* folder.

The first few lines include the libsms header file *sms.h* and define named constants for Boolean values so that the code is easier to read.

```
#include "sms.h"

#define TRUE 1
#define FALSE 0
```

Next we begin the `main` function and make sure that the user has supplied the correct number of command line arguments. There must be two in this case, the path to the input sound file and the path to the new output file that will be created.

```
int main(int argc, char *argv[])
{
    /* make sure that we have the correct number of
     * command line arguments
     */
    if(argc != 3)
    {
        printf("Error: please specify an input file and "
               "an output file\n");
        exit(1);
    }
```

Following this, we declare the variables that will be used in the rest of the program. The first five are all libsms data structures. `analysisData` will hold the result of analyzing one frame of audio. `smsHeader` contains information that is used throughout the entire analysis/synthesis process such as the sample rate, the total number of frames, etc. `soundHeader` will contain information about the input audio file such as the total number of samples, the number of audio channels and the sampling rate. `analysisParams` and `synthParams` are used to define parameters used during analysis and synthesis respectively. Some of these parameters will be used in the example programs discussed in this text, however there are many more, and depending on the audio file being analyzed each parameter can have a large impact on the perceived quality of the synthesized sound. For a full reference of libsms analysis and synthesis parameters consult the online documentation.

The remaining variables are standard C variable types. `inputFile` and `outputFile` are strings assigned to the two command line arguments. `currentFrame`, which will hold one frame of input audio samples, is a floating point array the same size as the largest possible analysis window used by libsms, `SMS_MAX_WINDOW` samples long. `doAnalysis` is a Boolean variable that will be used to control when to stop the main analysis/synthesis loop. `status` will contain information about the result of analyzing the most recent frame of audio, such as whether or not the analysis was completed successfully. `numSamplesRead` contains the total number of samples read from the audio input file at any given time. `frameSize` contains the size of the current audio frame, which can vary from frame to frame depending on the estimated pitch of the input audio file. This process is described in more detail in Section 3.4. `audioOutput` will contain one frame of synthesized audio.

```
    SMS_Data analysisData;
    SMS_Header smsHeader;
    SMS_SndHeader soundHeader;
```

```
SMS_AnalParams analysisParams;
SMS_SynthParams synthParams;

char *inputFile = argv[1];
char *outputFile = argv[2];
float currentFrame[SMS_MAX_WINDOW];
int doAnalysis = TRUE;
long status = 0, numSamplesRead = 0, frameSize = 0;
float *audioOutput;
```

After defining these variables, a call is made to `sms_openSF` to open the input audio file for reading. Behind the scenes, this uses *libsndfile*[12] to open and read the sound file. The return value from this function is checked to make sure that no problems were encountered.

```
/* open input sound */
if(sms_openSF(inputFile, &soundHeader))
{
    printf("Error: Could not open the input sound file\n");
    exit(1);
}
```

Next we initialize the libsms variables that we defined previously. These function calls allocate memory that will be used during analysis and synthesis, and set the parameters to some sensible default values.

```
/* initialize libsms */
sms_init();
sms_initAnalParams(&analysisParams);

/* at 44.1 Khz this frame rate results in a hop size of 128 */
analysisParams.iFrameRate = 344;
sms_initAnalysis(&analysisParams, &soundHeader);
sms_fillHeader(&smsHeader, &analysisParams,
               "AnalysisParameters");
sms_allocFrameH(&smsHeader, &analysisData);
sms_initSynthParams(&synthParams);
synthParams.sizeHop = 128;
sms_initSynth(&smsHeader, &synthParams);
```

It is important to note here that in libsms analysis, we specify a frame rate that will alter the analysis hop size depending on the sample rate of the input audio file. The formula used is:

$$\text{hop size} = (\text{int})\frac{\text{sampling rate}}{\text{frame rate}} \tag{3}$$

However, for our purposes, we want to be able to analyze a frame and synthesize it

again immediately, so we need to the analysis hop size to be the same as the synthesis hop size. We set the analysis frame rate to `344`, which is equivalent to the synthesis hop size of `128` for audio files recorded at a sampling rate of 44.1 KHz. If you are using audio files at different sample rates you will have to adjust this line accordingly to obtain accurate results.

Next a call is made to `sms_createSF` that opens the output audio file for writing, again using libsndfile. `synthParams.iSamplingRate` is set to the same sampling rate as the input audio file by the function `sms_initSynth`. The third argument specifies an output file format. `0` is a floating point *wav* files, `1` is a floating point *aiff* and `2` is a 16-bit *wav*. After this, memory is allocated for one frame of audio output.

```
/* initialize libsndfile for writing a soundfile */
sms_createSF(outputFile, synthParams.iSamplingRate, 0);

/* allocate memory for one frame of audio output */

if((audioOutput = (float *)calloc(synthParams.sizeHop,
   sizeof(float))) == NULL)
{
    printf("Error: Could not allocate memory for audio "
           "output\n");
    exit(1);
}
```

With all the necessary variables declared, parameters set and memory allocated, it is now time to being the main analysis/synthesis loop. This will run until the entire input file has been analyzed by libsms and the output file has been created.

```
/* analysis/synthesis loop */
while(doAnalysis)
{
    /* update the number of samples read so far */
    numSamplesRead += frameSize;

    /* get the size of the next frame */
    if((numSamplesRead + analysisParams.sizeNextRead) <
       soundHeader.nSamples)
    {
        frameSize = analysisParams.sizeNextRead;
    }
    else
    {
        frameSize = soundHeader.nSamples - numSamplesRead;
    }
```

First, the `numSamplesRead` count is updated. It is then compared to the size of the next audio frame, initially specified in `sms_initAnalysis`, but the value can change

as analysis continues. As long as we have enough samples of the input file remaining, this value is used as our new frame size. If we don't have enough samples left, the frame size is simply the number of remaining samples.

Now that we know how many samples need to be read from the input file, `sms_getSound` is called which copies the next `frameSize` worth of samples into the `currentFrame` array. Then, these audio samples are passed to `sms_analyze`, which stores one frame of analysis data in `analysisData` if analysis was successful.

```
/* read the next frame */
if(sms_getSound(&soundHeader, frameSize, currentFrame,
                numSamplesRead))
{
    printf("Error: could not read audio frame: %s\n",
           sms_errorString());
    break;
}

/* analyse the frame */
status = sms_analyze(frameSize, currentFrame, &analysisData,
                     &analysisParams);
```

Libsms actually requires several frames of audio before it starts producing analysis frames. Internally it always saves some frames in a circular buffer in memory. For this reason, `analysisData` cannot be synthesized immediately on the first few runs through the loop; we must wait until `sms_analyze` reports that it is now producing usable data. This is signaled by the function return value. It returns one of three values: `-1` if analysis is now finished (no more frames remaining to analyze), `0` if `sms_analyze` requires more frames of audio in order to return the first frame of analysis data, and `1` if `analysisData` contains a successfully analyzed frame. Similarly, even when we have read in the entire input audio file, `sms_analyze` will need to be called several times (with no audio input) in order to finish processing the audio frames that it has saved in memory. This is why our analysis/synthesis loop is terminated by checking the return value of `sms_analyze` rather than simply stopping when we have finished reading the input file.

As soon we have usable analysis data, we start synthesizing audio by passing it to `sms_synthesize`, along with a pointer to the audio output buffer and the synthesis parameters. This synthesized audio is then written to file by passing a pointer to the audio buffer and the number of samples of audio that it contains to `sms_writeSound`. When we run out of analysis data, the main loop is terminated by setting `doAnalysis` to `FALSE`.

```
/* if the analysis status is 1, analysis was successful
 * and we have a frame of analysis data
 */
if(status == 1)
{
```

```
            /* synthesise audio output */
            sms_synthesize(&analysisData, audioOutput,
                           &synthParams);

            /* write output file */
            sms_writeSound(audioOutput, synthParams.sizeHop);
        }


        /* if status is -1, there is no more data to process,
         * analysis is finished
         */
        else if(status == -1)
        {
            doAnalysis = FALSE;
        }
    }
```

Finally, the input and output audio files are closed, and all memory that was allocated by libsms is released.

```
    /* close input sound file */
    sms_closeSF();

    /* close output sound file */
    sms_writeSF();

    /* free memory used by libsms */
    sms_freeFrame(&analysisData);
    sms_freeAnalysis(&analysisParams);
    sms_freeSynth(&synthParams);
    sms_free();

    return 0;
}
```

3. Understanding Libsms

We will now have a look at what happens to a frame of audio when calls are made to `sms_analyze` and `sms_synthesize`. Sections 3.1 to 3.7 all deal with steps in the analysis process. Sections 3.1 to 3.5 are steps in the analysis of the harmonic component of the signal while Section 3.6 and 3.7 describe the analysis of the residual sound component. Finally, Section 3.8 looks at sound synthesis using the analysis data.

3.1 Spectral Analysis

The first analysis step is basically just a Short-Time Fourier Transform (STFT), as described in Book Chapter 8. As the audio frame must be windowed, the first step is to calculate an appropriate window size. If this is the first frame being analyzed, a default window size is used. This can be specified in the `analysisParams` structure, but if not it defaults to a value that depends on the sampling rate and the default pitch value (which can also be changed in `analysisParams`). At a sampling rate of 44.1 KHz this value is 1681. Due to the inherent trade-off between time and frequency resolution in the Fourier Transform, the window size is a very important parameter. Any extra information that can be specified about the input sound (such as the pitch) prior to calling `sms_analyze` will result in improved analysis. For later frames this window size value may change depending on the results of the pitch estimation step described in Section 3.3.

The type of window used by default is a Blackman-Harris 70 dB window. This is applied to the audio frame, which is then zero-padded up to a size that is a power of two so that the Fast Fourier Transform algorithm can be used. This zero-padding also introduces interpolation in the frequency domain resulting in an improved frequency resolution. A circular shift is performed on the frame to preserve zero-phase conditions and then the FFT is calculated. Figures 2-4 show the steps involved in this process.

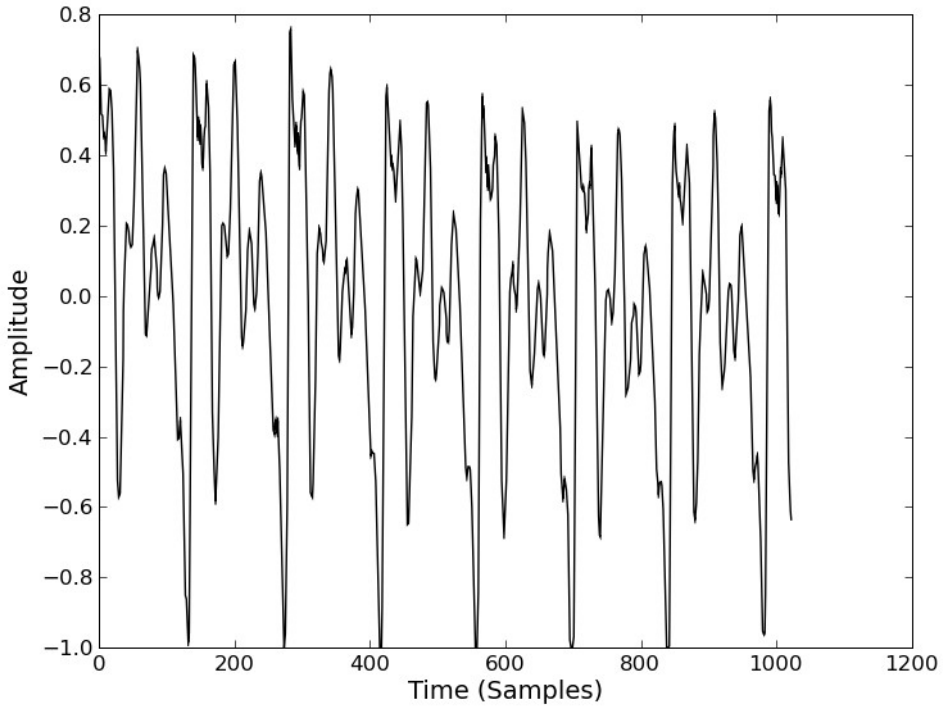

Figure 2 A frame of audio from a vocal sample.

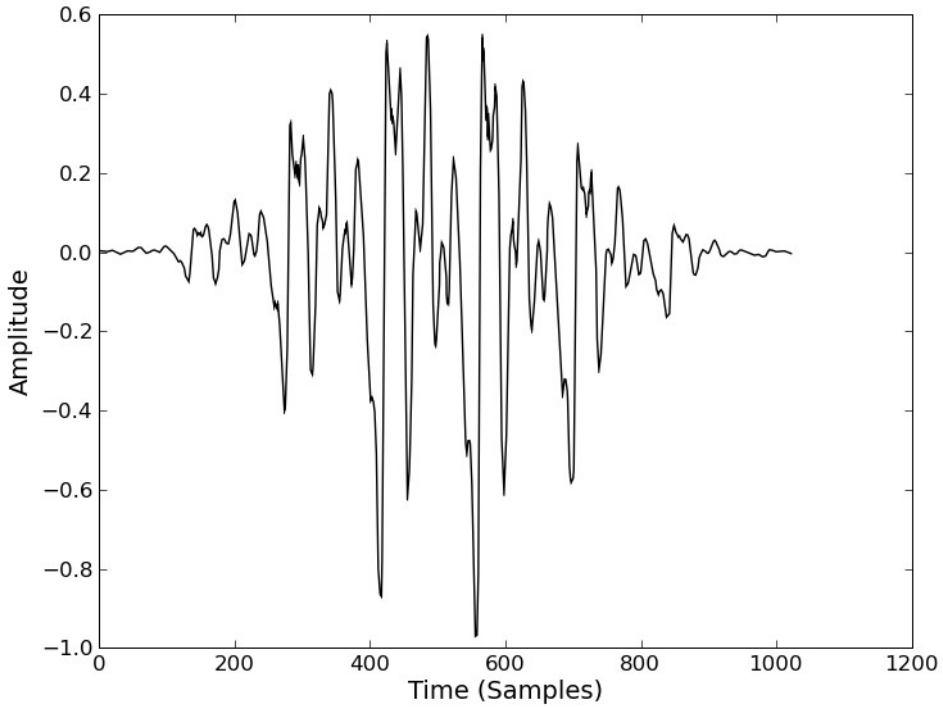

Figure 3 The same audio frame after a window is applied.

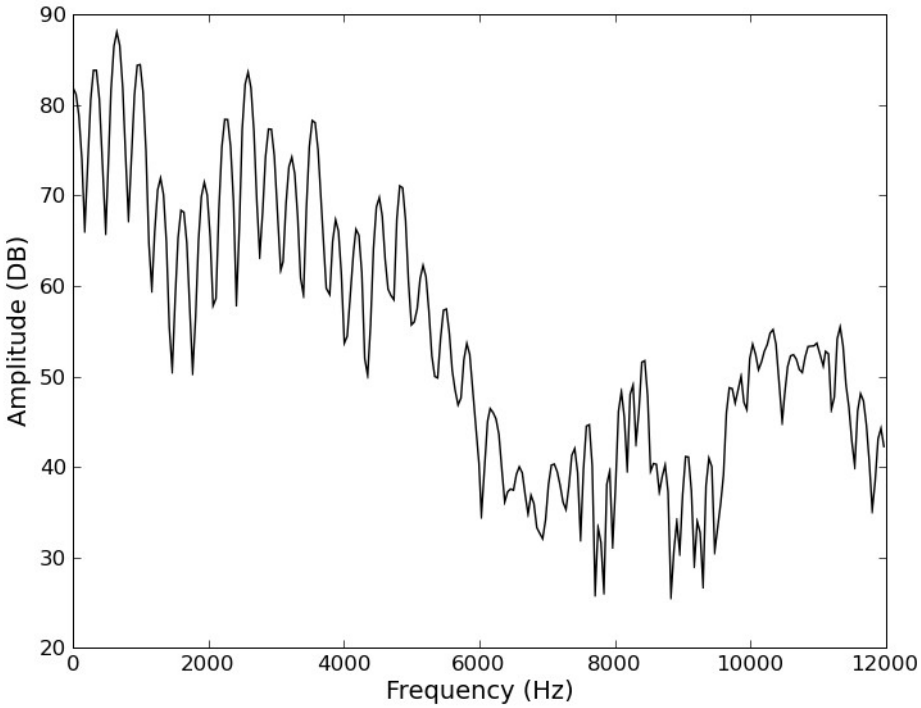

Figure 4 The FFT of the windowed frame (spectral components up to 12 KHz).

3.2 Peak Detection

The aim of harmonic analysis is to represent the stable, periodic parts of a sound as a number of sine waves with different frequencies and amplitudes. Unfortunately for most instrumental sounds it is generally not possible to tell which frequency components are stable and which ones are not from just one frame of spectral analysis data. Some of the sinusoidal components could be due to noise or analysis artifacts. A solution to this is to select the spectral *peaks* from each frame, which are the sinusoidal components with the largest amplitude, then compare peaks from a given frame to those from one or more consecutive frames and see which ones can be used to form stable partials. This process of selecting stable peaks is called *peak continuation* or *partial tracking* and is discussed in Section 3.4.

A peak is defined as a local maximum, a spectral component with a magnitude (expressed in decibels) larger than both its neighbors. Not all peaks are taken to be equal however. Generally an amplitude threshold is defined, below which all peaks are ignored as they are considered to either be noise or not important to the perception of the sound. Peak selection may also take other factors into account. For example, the amplitude threshold may be defined in relation to the amplitude of neighboring

peaks, or it may depend on the frequency of the sinusoidal component. Libsms includes a number of different analysis parameters that can be specified for the peak detection step, such as the frequency range that is searched for peaks, the minimum peak magnitude and the maximum number of peaks detected. As the spectrum returned by the STFT is sampled, each peak is only accurate to within half a bin. One way to improve this would be to use a large zero-padding factor, but as this is computationally inefficient. Instead, accuracy is improved by parabolic interpolation using the peak and the two neighboring samples, where the peak location is taken to be the maximum point of the parabola. The frequency and magnitude of the peak are taken from the magnitude spectrum, and the phase is obtained by unwrapping the phase spectrum and taking the value at the position that corresponds to the frequency of the peak. Figure 5 shows examples of peaks (marked with a red 'x') detected in the magnitude spectrum using libsms.

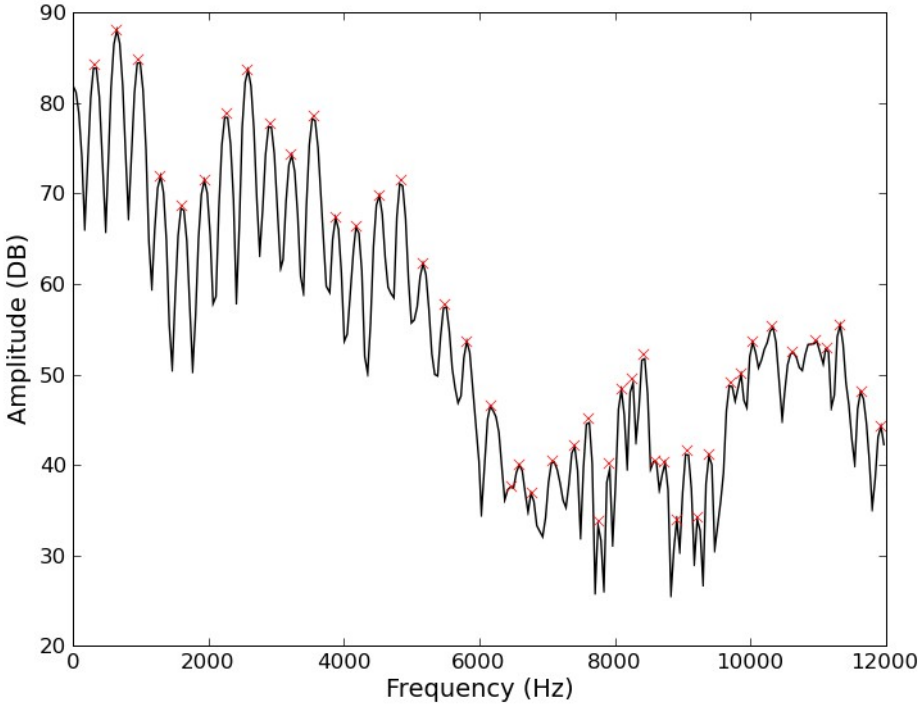

Figure 5 Peaks detected in one frame of a vocal sample after Fourier analysis

3.3 Pitch Estimation

This is an optional analysis step that is used when the input sound is monophonic and pseudo-harmonic (this is the default assumption but it can be changed in the analysis parameters). The objective is to find a fundamental frequency that best explains the

collection of harmonic peaks found in the current analysis frame. This information can then be used to set the guide frequencies in the partial tracking algorithm described in Section 3.4, and to help select an appropriate window size for the next frame of spectral analysis.

Pitch estimation in libsms begins by going through every peak that is within specified frequency limits. If the previous frame had a detected pitch then every peak that is close to it in frequency is marked as a *candidate*, or possible new fundamental frequency. If there was no previous pitch estimate, each peak is assessed to see if it is a candidate. This involves making sure its magnitude is above a certain threshold, that it is not a harmonic of an existing candidate and that the average deviation between its harmonic series and the detected peaks is within an acceptable limit. This last check is ignored if the peak magnitude is very large. After all the peaks have been checked, if there are no candidates then there is no pitch estimation. If there is only one, then its value is the estimated fundamental frequency. If there is more than one, an attempt is made to select the best candidate peak. If there was a previous pitch estimate, then the closest match to this is taken. If not then the peak with the largest amplitude and the lowest frequency deviation from a reference frequency (specified in the analysis parameters) is selected.

3.4 Partial Tracking

The peak detection process results in a set of sinusoidal frequencies, amplitudes and phases for each frame. The final stage in the harmonic analysis is to take these values and track them across frames, interpolating the values between consecutive peaks to form partials. The harmonic component will be synthesized based on this data using additive synthesis, so it is important that peaks are linked together in a way that results in an accurate representation of the original sound. This would be easy if there were a constant number of peaks all with very slowly changing amplitudes and frequencies, however in practice this is rarely the case.

The SMS partial tracking scheme uses a series of *guides* that advance from frame to frame looking for peaks to form partials. A guide can be thought of as a predicted partial trajectory. For harmonic sounds, guide frequencies are initialized according to the harmonic series of the detected fundamental frequency. If the sounds are inharmonic, guides are created dynamically depending on the detected peaks. When a new frame of audio arrives, guide frequencies are updated based on both their existing value and on the values of the harmonic series of the estimated fundamental frequency. Then each guide looks through the detected peaks and attempts to find a match, which is the peak that is closest in frequency, providing that it is within given amplitude and frequency thresholds. A peak can only be matched to one guide, so if a peak could potentially be matched to more than one guide, the guide that is closest in frequency gets priority and the other guide must find a new peak. If a guide does not find a match then it is *turned off*, or matched to itself with an

amplitude of zero. Guides can exist in an off state for a number of frames before being *killed* or made inactive. This frees up space for a new guide to be created, as there is limit to the number of guides that can exist at any one time. New guides are created from any unmatched peaks in the frame, starting with those with the largest amplitudes. At the end of this guide update process, any peaks that have been matched to guides are saved, forming partials. The rest are discarded.

After several frames have been tracked in this manner, two problems may emerge. There could be *gaps* in partials, which are areas where the amplitude has been set to zero because a guide was turned off, but the amplitude increases again in a later frame as a new peak is found for the guide before it is killed. If uncorrected, this will lead to an amplitude modulation effect in some of the partials, which produces audible artifacts in the synthesized sound. There could also be tracks that are not considered to have long enough durations to be stable partials, which again can lead to synthesis artifacts, particularly if the sound is transformed. The final step in the partial tracking algorithm is to search though a specified number of frames and correct these problems. Gaps are fixed by interpolating between the peak values before and after the gap. Short tracks are simply removed.

The partial tracking process results in a collection of peaks that are ordered by partial number. So peak one from frame one is matched to peak one in frame two, etc. This information is stored in `SMS_Data` structures (such as `analysisData` in *analysis_synthesis.c*). The number of tracks is stored in the `nTracks` variable, while peak frequencies, amplitudes and phases are stored in `pFSinAmp`, `pFSinFreq` and `pFSinPha` respectively.

3.5 Spectral Envelope Calculation

This is an optional step. It was not part of the original SMS scheme, but it is supported by libsms as it is useful for certain sound transformations, some of which are discussed in Section 4. A *spectral envelope* is a curve in the spectrum of one frame of audio that approximates the distribution of the signal's energy over frequency. Ideally, this curve should pass through all of the prominent peaks of the frame and be relatively smooth, so it does not oscillate too much or contain any sharp corners or discontinuities. It should preserve the basic formant structure of the frame. Figure 6 shows an example of a spectral envelope (in blue) calculated from one frame of a vocal sample using libsms.

Libsms calculates spectral envelopes using the *Discrete Cepstrum Envelope*[13] method, which basically provides a smooth interpolation between the detected sinusoidal peaks. The envelope is calculated for each analysis frame, and is stored in the `SMS_Data` structure. The variable `nEnvCoeff` gives the number of envelope coefficients, while the envelope itself is stored in `pSpecEnv`.

3.6 Residual Calculation

Now that the stable partials of the sound have been detected, they can be subtracted from the original sound to create the residual component. Libsms performs this subtraction in the time domain. The first step is to use additive synthesis to create the harmonic sound component using the partials found during analysis. This is done on a frame-by-frame basis, by generating a sine wave for each partial then summing the results. To prevent discontinuities at frame boundaries, the parameters of each sinusoidal component (amplitude, frequency and phase) are interpolated across the frame. The instantaneous amplitude is calculated by linear interpolation. The frequency and phase parameters are closely related as frequency is the derivative of phase, and so both control the instantaneous phase that is calculated by using cubic interpolation[14]. The harmonic component is then subtracted from the original sound, which should leave a noise-like residual signal that is largely stochastic.

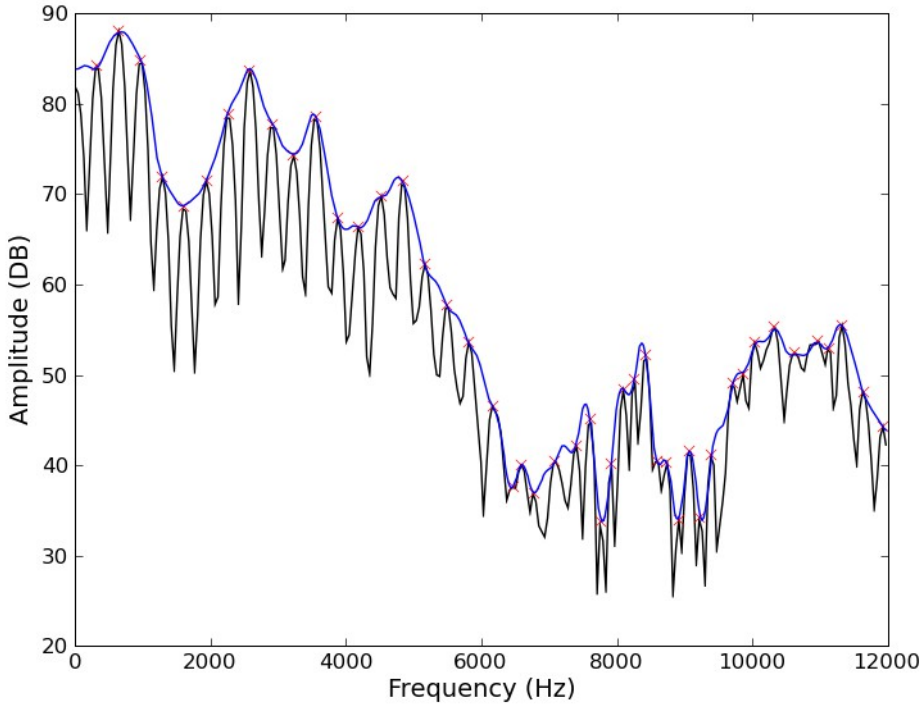

Figure 6 The spectral envelope calculated from one frame of a vocal sample.

3.7 Residual Approximation

As the residual signal is assumed to be stochastic, each frame of it can be characterized by its amplitude and the general shape of its spectrum. Each residual frame can therefore be approximated by applying this spectral envelope shape to a noise signal and scaling the amplitude to match that of the original residual. This process can also be viewed as time-domain filtering of a noise signal, where the envelope shape is used to calculate the filter coefficients.

The first step is to calculate the spectrum of the residual signal. The number of coefficients that will be used to represent this spectrum is specified in the analysis parameters (128 by default). If this is less than the number of spectral bins, then the spectrum must be approximated. In libsms, this is achieved by line-segment approximation. The spectrum is divided into a number of evenly spaced areas, one for each coefficient. The value of each coefficient is the maximum value of the spectrum in this area. These coefficients are then stored in the `pFStocCoeff` variable of the output `SMS_Data` structure. Figure 7 shows an example residual spectrum (128 bins) and Figure 8 shows the resulting spectrum approximation (64 coefficients).

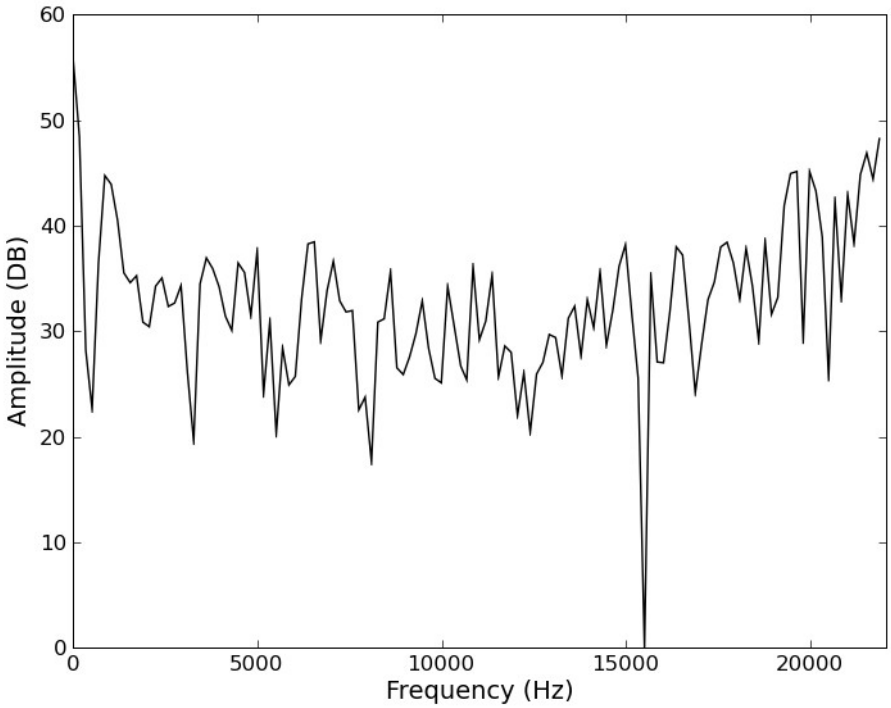

Figure 7 Spectrum of one frame of a residual signal from a vocal sample

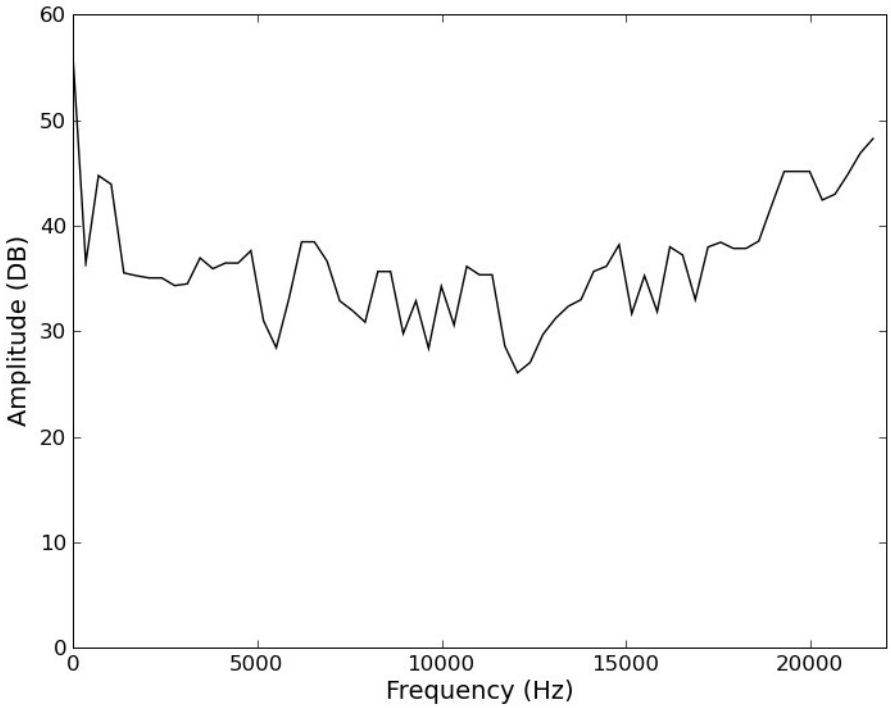

Figure 8 Approximation of the spectrum

3.8 Synthesis

Sound synthesis is performed by making calls to the `sms_synthesize` function. It takes three arguments; a pointer to an `SMS_Data` structure, a pointer to an output audio buffer and a pointer to a structure containing the synthesis parameters (`SMS_SynthParams`). Each call to `sms_synthesize` requires one frame of harmonic and residual analysis data, stored in the `SMS_Data` structure, and creates one frame of audio. `SMS_Data` frames can be passed to `sms_synthesize` in any order, as the data is either interpolated between frames or else overlap-add is used to avoid discontinuities at frame boundaries. We can synthesize just the harmonic component, just the residual component or both at once by changing the `iSynthesisType` variable in the `SMS_SynthParams` structure. By default, both components will be created.

The harmonic component is generated by additive synthesis using one of two schemes. Either the analysis data is used to control the instantaneous amplitudes, frequencies and phases of a bank of oscillators (as described in Section 3.6) or it is used to synthesize audio using the inverse FFT[15], the latter being the default. The choice of additive synthesis method is specified by the `iDetSynthType` variable in `SMS_SynthParams`. Unlike in the residual calculation stage (Section 3.6), the harmonic component is synthesized without using the original phase information, as

it is assumed that it is not crucial to the perception of the sound. This gets rid of the need for the computationally expensive cubic phase interpolation. Both the instantaneous amplitude and the instantaneous frequency are calculated by linear interpolation between two frames, and phase is taken to be the integral of frequency. For new partials, the initial phase values are randomly generated.

The residual component is recreated by inverting the approximation process described in Section 3.7. If the number of residual coefficients is less than the synthesis frame size, an approximate spectrum is created by linear interpolation between coefficients. This provides the amplitude and frequency information, while the phases are generated randomly. Then an IFFT is performed on this spectrum to create the residual audio signal. To prevent discontinuities at frame boundaries, this FFT size is twice the frame size and the final sound is created by an overlap-add between two frames, with 50% overlap. The synthesized audio frame is then created by adding both harmonic and residual components to the output audio buffer.

4. Sound Manipulation Using Spectral Modeling Synthesis

Now that we have a better understanding of how straight synthesis by analysis works using libsms, we can look in detail at the rest of the examples. Each example changes the analysis data in some way so that the output sound is perceptually different to the input. However, as they are all based on the first example (*analysis_synthesis.c*), we will only discuss the differences between the programs here and will not give the full code listing as before. The full source code for each example can be found in the *examples* folder, which also contains information on how to compile and run the examples in the file *examples-guide.doc.* Examples 4.1 and 4.2 show how we can change the pitch of a sound without effecting its duration, while Example 4.3 demonstrates changing the duration of a sound without altering the pitch. Finally, Example 4.4 shows how the spectral envelope from one sound can be used to change the harmonic component of another.

4.1 Transposition

The first additions to this example (*transpose.c*) are the following two lines:

```
#include <stdlib.h>
...
float transpose = atof(argv[3]);
```

Here we are using an extra command line argument to specify the transposition amount in semi-tones. A positive number will transpose to a higher pitch, a negative number will lower the pitch. The file *stdlib.h* is included so that we can use the `atof`

function, which converts a string (the fourth command-line argument) into a floating-point value that is then saved to the variable called `transpose`. As before, variables are declared when libsms is initialized, but there is one extra libsms structure declared that is of type `SMS_ModifyParams`. As the name suggests, this will contain parameters that will specify what modifications (if any) will be made to the analysis data.

```
SMS_ModifyParams modifyParams;
...
sms_initModifyParams(&modifyParams);
sms_initModify(&smsHeader, &modifyParams);
```

`sms_initModifyParams` just sets some sensible default values in the `modifyParams` structure, ensuring that by default nothing will be changed. `sms_initModify` allocates some memory that will be used by the `modifyParams` structure. After this, the output sound is opened for writing and memory and is allocated for one frame of audio. Then the `doTranspose` variable in `modifyParams` is set to `TRUE`, and the `transpose` amount is set to the given command-line value.

```
modifyParams.doTranspose = TRUE;
modifyParams.transpose = transpose;
```

The actual modification of the analysis data happens in the main analysis/synthesis loop, just before `sms_synthesize` is called. The analysis frame that we want to change is passed to `sms_modify` along with the `modifyParams` structure. The changed `analysisData` is then passed to `sms_synthesize` to create the output audio signal.

```
/* transpose the sound */
sms_modify(&analysisData, &modifyParams);

/* synthesise audio output */
sms_synthesize(&analysisData, audioOutput, &synthParams);
```

After the analysis/synthesis loop, audio files are written and closed as before, then the memory used by libsms is deallocated, including the memory used by the `modifyParams` structure.

```
sms_freeModify(&modifyParams);
```

This example shows the general structure that can be used for several different libsms transformations. Analyze the sound, call `sms_modify` to change the analysis data then synthesize this data. The algorithm that is performed by `sms_modify` is very simple in this instance. As noted in Section 3.4, the harmonic components of the sound (sinusoidal peaks) are saved in the `analysisData` structure. To change the perceived

pitch, we just multiply all of the sinusoid frequencies by the transposition amount. The residual component is left unchanged. The basic algorithm is:

```
int i;
for(i = 0; i < analysisData->nTracks; i++)
{
    analysisData->pFSinFreq[i] *= sms_scalarTempered(transpose);
}
```

where `nTracks` is the number of partials, `pFSinFreq` is an array containing the frequencies of each peak, and `sms_scalarTempered` converts a transposition amount in semi-tones into a frequency value in Hertz.

4.2 Transposition While Maintaining The Original Spectral Envelope

The basic transposition described in Section 4.1 works well for some sounds (such as the flute example), but not for others. If this example is run with a vocal sample instead, the results are very synthetic sounding. The reason for this is that when we transpose a sound by simply multiplying the component frequencies, we also change the shape of the resulting audio spectrum as the amplitudes of each partial stays the same. So if there was a very large magnitude partial at 1 KHz for example, and we transpose up one octave, there will now be a large magnitude partial at 2 KHz. As our perception of timbre is strongly linked to the temporal evolution of this spectrum, changes like this will effect our perception of the synthesized sound. This process is shown in Figures 9 and 10, the former being the original spectrum and the latter being the new spectrum after transposition by a major third. Notice that the original formant peaks at about 650 Hz and at about 2.6 KHz have now moved to about 825 Hz and 3.25 KHz.

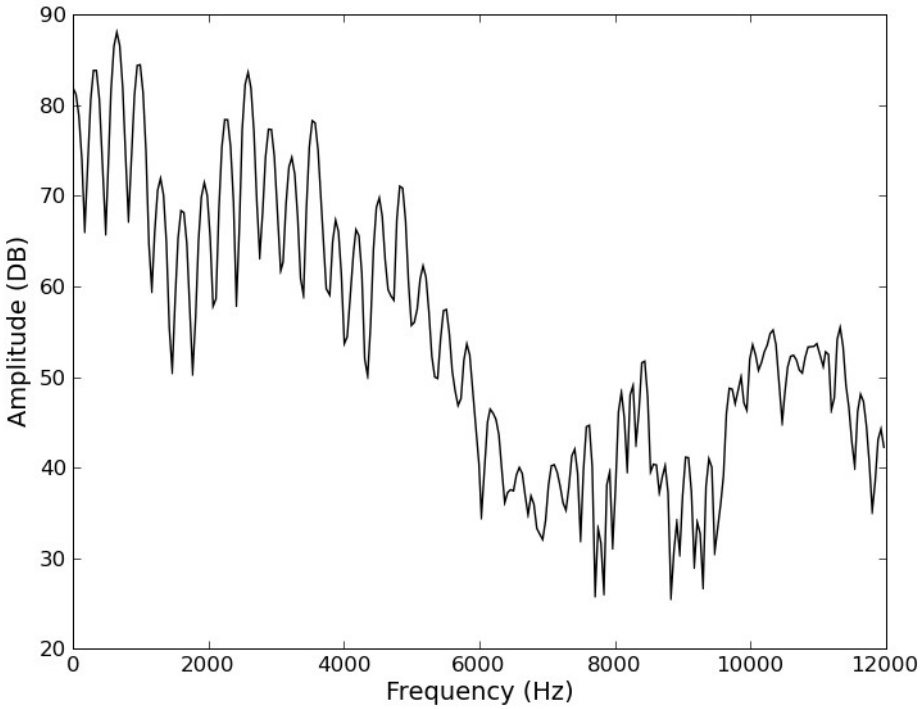

Figure 9 Original spectrum of one frame of a vocal sample.

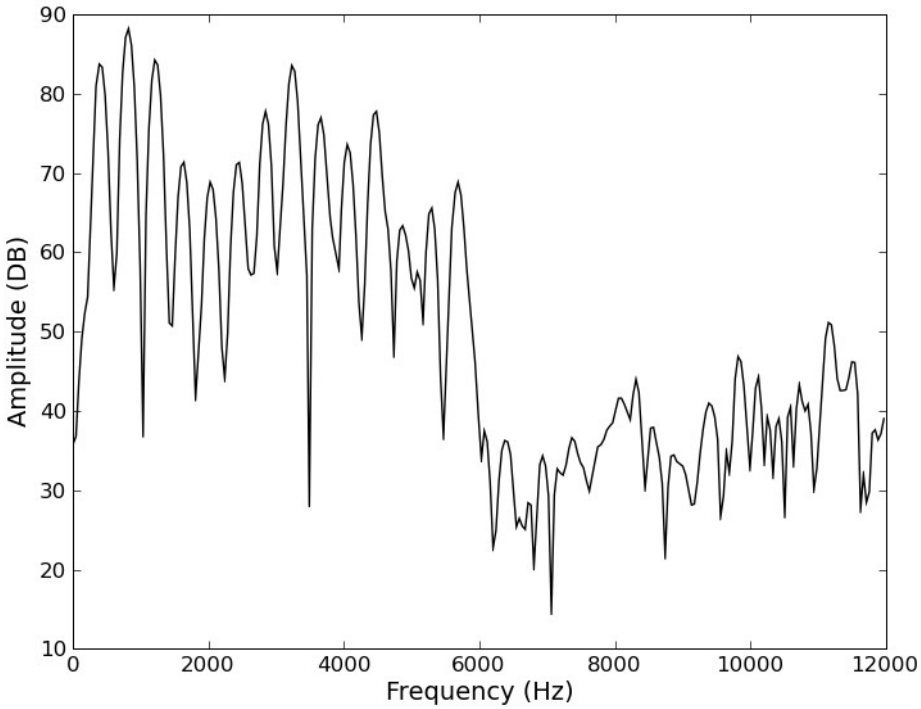

Figure 10 Spectrum after transposition.

For some instrumental sounds this is appropriate, but sources such as the voice tend to have a more rigid formant structure, particularly for relatively small pitch changes by the same singer. To make pitch changes on these sound sources sound more natural, we need to alter the shape of the spectrum after transposition so it is the same as it was before, keeping the formants in the same areas. Example 3 (*transpose_with_env.c*) shows how we can accomplish this using libsms. There are not many changes between this example and *transpose.c*. Two additional analysis parameters are specified. The first one sets the spectral envelope type to `SMS_ENV_FBINS`, which turns on spectral envelope calculation during analysis (it is off by default) and tells libsms to store the envelope as a list of frequency bins and magnitudes. The order of the Discrete Cepstrum Envelope is set to 80, which should ensure that the envelope will pass through every spectral peak and interpolate smoothly between them.

```
analysisParams.specEnvParams.iType = SMS_ENV_FBINS;
analysisParams.specEnvParams.iOrder = 80;
```

One extra modification parameter is also added, telling `sms_modify` to alter the magnitudes of the sinusoidal peaks after transposition so that they match those of the

spectral envelope.

```
modifyParams.doSinEnv = TRUE;
```

`sms_modify` is then called as before, which performs the transposition. It multiplies the frequencies of the sinusoidal components by the transposition amount as shown in Section 4.2. Afterwards it changes the amplitude of each peak so that it matches the amplitude of the original spectral envelope at the new frequency value.

4.3 Time-Stretching

The structure of this example (*timestretch.c*) is a bit different to that of the previous three. We will not be passing analysis frames straight from `sms_analyze` to `sms_synthesize`, and we will not be making calls to `sms_modify`. Instead, we will vary the duration of the output sound by changing the rate at which new analysis frames are passed to `sms_synthesize`. To make the output twice as long as the input for example, each analysis frame is used twice before synthesizing the next one. To make it half as long, the frames are read twice as quickly (so every second frame will be skipped). This process will not result in any glitches or discontinuities as `sms_synthesize` interpolates smoothly between frames.

As we want to change the rate at which analysis frames are passed to `sms_synthesis`, the analysis and synthesis processes had to be separated. We will do this by writing the analysis frames to disk then reading them in later at a different rate. The `main` function has been greatly simplified. It just takes in the command-line arguments, and calls the `analyze` and `synthesize` functions with them.

```
int main(int argc, char *argv[])
{
    /* make sure that we have the correct number of
     * command line arguments
     */
    if(argc != 4)
    {
        printf("Error: please specify an input file, an output"
               " file and a time stretch factor\n");
        exit(1);
    }

    char *inputFile = argv[1];
    char *outputFile = argv[2];
    float timeStretch = atof(argv[3]);

    analyze(inputFile);
    synthesize(outputFile, timeStretch);
```

```
    return 0;
}
```

The `analyze` function begins similarly to the `main` function from previous examples, declaring variables for a frame of analysis data, analysis parameters and header files. The only addition is the inclusion of a file pointer.

```
FILE *smsFile;
```

Following this we open the input file for reading and initialize libsms. We also open a file called *analysis_data.sms* that will be used to store the analysis frames.

```
sms_writeHeader("analysis_data.sms", &smsHeader, &smsFile);
```

The analysis loop again begins by reading in a frame of audio samples that is passed to `sms_analyze`. As soon as we have usable analysis data, instead of calling `sms_modify` or `sms_synthesize` we write this frame to disk and increment the current frame count.

```
sms_writeFrame(smsFile, &smsHeader, &analysisData);
numFrames++;
```

When analysis is finished this frame count is saved to the SMS_Header structure.

```
smsHeader.nFrames = numFrames;
```

After closing the input sound file, the SMS_Header information is written to *analysis_data.sms* and the file is closed.

```
sms_writeFile(smsFile, &smsHeader);
```

The `synthesize` function begins in a similar manner to `analyze`, declaring the same variables that were used in previous examples with the addition of the file pointer. The only real difference in this setup phase is that the file containing the analysis frames is opened for reading.

```
sms_getHeader("analysis_data.sms", &smsHeader, &smsFile);
```

Libsms is initialized as before, the output sound file is opened for writing and memory is allocated to hold one frame of audio samples. The synthesis loop will be different however, as we are no longer calling `sms_synthesize` for each analysis frame. First, the number of samples that the output file should contain is calculated.

This is the total number of analysis frames multiplied by the synthesis hop size, multiplied by the time stretch amount that was read from the command-line.

```
numSamples = smsHeader->nFrames * synthParams.sizeHop *
             timeStretch;
```

The synthesis loop then keeps going until this many samples have been written to the output file.

```
while(numSamplesWritten < numSamples)
{
    /* calculate the current frame number based on
     * the time stretch factor
     */
    currentFrame = numSamplesWritten /
                   (synthParams.sizeHop * timeStretch);

    /* read in a frame of analysis data */
    sms_getFrame(smsFile, smsHeader, currentFrame, &smsFrame);

    /* synthesise audio output */
    sms_synthesize(&smsFrame, audioOutput, &synthParams);

    /* write to output file */
    sms_writeSound(audioOutput, synthParams.sizeHop);

    /* update the number of samples written */
    numSamplesWritten += synthParams.sizeHop;
}
```

The first step in the synthesis loop is to calculate which frame number to read in by essentially imagining that the hop size changes depending on the time-stretch factor[16]. If `timeStretch` is `1`, the current frame is the number of samples written divided by the hop size. As the number of samples written increases by the hop size each time through the loop, this will just be a sequence that increments by one each time. If `timeStretch` is `2` for example, the loop will have to go around twice before the `currentFrame` value is incremented, as the division result is truncated to an `int` value. This frame number is then passed to `sms_getFrame`, which reads an analysis frame from disk into the `smsFrame` variable. This frame is then passed to `sms_synthesize` and the output audio frame is written to disk as before. Finally, the `numSamplesWritten` variable is incremented by the number of samples in the frame that was just created. When the synthesis loop is finished, the output file is closed and libsms memory is deallocated as in previous examples.

4.4 Applying A Different Spectral Envelope

The final example (*change_env.c*) shows how the spectral envelope calculated from the harmonic component of one sound can be used to shape the spectrum of another sound. In our example, the spectral envelope from a vocal sample is applied to the partials of a flute sample, creating the effect of the flute 'talking' while keeping its original pitch contour. The `main` function is similar to that of the previous example. This time the command-line arguments are the source file, the target file and the output file (the spectral envelope will be taken from source sound and applied to the target). The `analyze` function is called twice, producing *source_file.sms* and *target_file.sms*, both of which will be read in by the `synthesize` function.

```
int main(int argc, char *argv[])
{
    /* make sure that we have the correct number of
     * command line arguments
     */
    if(argc != 4)
    {
        printf("Error: please specify a source file, a target "
               " file and an output file\n");
        exit(1);
    }

    char *sourceFile = argv[1];
    char *targetFile = argv[2];
    char *outputFile = argv[3];

    analyze(sourceFile, "source_file.sms");
    analyze(targetFile, "target_file.sms");
    synthesize(outputFile);

    return 0;
}
```

The `analyze` function is similar to the one described in Section 4.3, with the addition of two extra analysis parameters that turn on spectral envelope calculation and set the order of the Discrete Cepstrum Envelope.

```
analysisParams.specEnvParams.iType = SMS_ENV_FBINS;
analysisParams.specEnvParams.iOrder = 80;
```

The setup phase of `synthesize` is also similar to before, but as there are now two input files, there are two `SMS_Header` structures and two `SMS_Data` frames. Both headers are opened for reading, and then memory is allocated for the two frames. There is also an extra variable declared called `envPosition` that will be used to cycle through the samples of the spectral envelope in the synthesis loop.

```
SMS_Header *sourceSmsHeader, *targetSmsHeader;
SMS_Data sourceSmsFrame, targetSmsFrame;
...
int numFrames = 0, currentFrame = 0, envPosition = 0;
...
sms_getHeader("source_file.sms", &sourceSmsHeader,
              &sourceSmsFile);
sms_getHeader("target_file.sms", &targetSmsHeader,
              &targetSmsFile);
...
sms_allocFrameH(sourceSmsHeader, &sourceSmsFrame);
sms_allocFrameH(targetSmsHeader, &targetSmsFrame);
```

The number of frames of output is then set to the number of frames in the shortest duration input file. This is to make sure that both source and target analysis data exists for all synthesis frames.

```
if(sourceSmsHeader->nFrames < targetSmsHeader->nFrames)
{
    numFrames = sourceSmsHeader->nFrames;
}
else
{
    numFrames = targetSmsHeader->nFrames;
}
```

We again set the `doSinEnv` modification parameter so that the amplitudes of the sinusoidal components will be changed to match those of the spectral envelope. However in addition the `sinEnvInterp` parameter is set to `1`. If this is `0` (as it is by default), the original spectral envelope will be used to change the peak amplitudes. If it is `1`, a completely different envelope will be used (this envelope is specified in the synthesis loop, described below). Any value between `0` and `1` will use an envelope that is the appropriate linear interpolation between the two envelopes. For example a value of `0.1` will produce an envelope that is very close to the original envelope, `0.9` will be very close to the new envelope and `0.5` will be half way in between the two.

```
modifyParams.doSinEnv = TRUE;
modifyParams.sinEnvInterp = 1;
```

The synthesis loop begins by reading in two frames of analysis data. Then, the spectral envelope from the source frame is copied into the `sinEnv` array in the `modifyParams` structure. This is the envelope that is used to change the amplitudes of the sinusoidal peaks in the target frame. The modification parameters are passed to `sms_modify` along with the target frame of analysis data, and the resulting frame is then passed to `sms_synthesize` as before. The output frame is then written to disk and the current frame count is incremented by one.

```
while(currentFrame < numFrames)
{
    /* read in both frames of analysis data */
    sms_getFrame(sourceSmsFile, sourceSmsHeader, currentFrame,
                 &sourceSmsFrame);

    sms_getFrame(targetSmsFile, targetSmsHeader, currentFrame,
                 &targetSmsFrame);

    /* copy the source envelope into the modification
     * parameters structure
     */
    for(envPosition = 0; envPosition < modifyParams.sizeSinEnv;
        envPosition++)
    {
        modifyParams.sinEnv[envPosition] =
            sourceSmsFrame.pSpecEnv[envPosition];
    }

    /* call sms_modify to apply the new envelope */
    sms_modify(&targetSmsFrame, &modifyParams);

    /* synthesise audio output */
    sms_synthesize(&targetSmsFrame, audioOutput, &synthParams);

    /* write to output file */
    sms_writeSound(audioOutput, synthParams.sizeHop);
    currentFrame++;
}
```

After the synthesis loop completes, the output file is closed for writing and the memory that was used by libsms is deallocated.

5. Conclusion

This chapter gave an overview of the SMS process and showed how libsms can be used to perform synthesis by analysis on sound files. This was followed by four examples that showed some of the transformations that are possible when using a sinusoidal plus residual model of sound. Even working with just these examples, there is still plenty of room for experimentation. Example 4 (*timestretch.c*) showed that frames of analysis information could be synthesized at any rate and in any order, so the frames do not have to playback in ascending order as they do in the examples. Example 5 (*change_env.c*) demonstrated using the spectral envelope from one sound to shape the amplitudes of the harmonic components of another sound. There are

many more possible transformations based on this technique, such as gradually interpolating between two envelopes, generating completely synthetic envelopes, and so on. There are also many possibilities for sound manipulation using SMS that have not been mentioned here[17], and of course these techniques do not have to be used in isolation. Great results can be achieved when SMS transformations are used in combination with the vast array of audio signal processing techniques that are discussed elsewhere in this book.

[1] Chowning, J. 1973. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation", *Journal of the Audio Engineering Society,* 21: 526-534.

[2] Roads, C. 1988. "Introduction to Granular Synthesis", *Computer Music Journal,* 12(2): 11-13.

[3] Välimäki, V., Pakarinen, J., Erkut, C. and Karjalainen, M. 2006. "Discrete-time Modelling of Musical Instruments", *Reports on Progress in Physics,* 69: 1-78.

[4] Lazzarini, V., Timoney, J. and Lysaght, T. 2008. "The Generation of Natural-Synthetic Spectra by Means of Adaptive Frequency Modulation", *Computer Music Journal,* 32(2): 12-22.

[5] Samson, S.,Zatorre, R. and Ramsay, J. 1997. "Multidimensional Scaling of Synthetic Musical Timbre: Perception of Spectral and Temporal Characteristics", *Canadian Journal of Experimental Psychology*, 51(4): 307-215.

[6] Serra, X., and Smith, J. 1990. "Spectral Modeling Synthesis: A Sound Analysis/Synthesis Based on a Deterministic plus Stochastic Decomposition", *Computer Music Journal*, 14(4): 12-24.

[7] *Libsms* is an open source C library for Spectral Modelling Synthesis. More information can be found on the homepage: http://www.mtg.upf.edu/static/libsms/.

[8] For more information on the Short-Time Fourier Transform see chapter 8, "The STFT and Spectral Processing".

[9] For more information on the Phase Vocoder see Book Chapter 9, "Programming The Phase Vocoder".

[10] Smith, J. O., and X. Serra. 1987. "PARSHL: An Analysis/Synthesis Program for Nonharmonic Sounds based on a Sinusoidal Representation", *Proceedings of the 1987 International Computer Music Conference,* San Francisco.

[11] McAulay, R. J., and Quatieri, T. F. 1986. "Speech Analysis/Synthesis based on a Sinusoidal Representation", *IEEE Transactions on Acoustics, Speech, and Signal Processing,* 34(4): 744-754.

[12] For more information on *libsndfile* see Book Appendix C, "Soundfiles, Soundfile Formats and libsndfile" or the *libsndfile* homepage: http://www.mega-nerd.com/libsndfile/.

[13] Schwarz, D. and Rodet, X. 1999. "Spectral Envelope Estimation and Representation for Sound Analysis-Synthesis", *Proceedings of the International Computer Music Conference*, Beijing.

[14] McAulay, R. J., and Quatieri, T. F. 1986. "Speech Analysis/Synthesis based on a Sinusoidal Representation", *IEEE Transactions on Acoustics, Speech, and Signal Processing,* 34(4): 744-754.

[15] Rodet, X. and Depalle, P. 1992. "Spectral Envelopes and Inverse FFT Synthesis", *Proceedings of the 93rd Audio Engineering Society Convention*, San Francisco.

[16] In fact, this would be another good way to achieve a time-stretching effect. Synthesis would have to use sinusoidal oscillators instead of an IFFT-based approach in order to allow for arbitrary hop sizes, as with the default IFFT synthesis the hop size must be a power of two.

[17] Some more examples of SMS-based transformations are discussed in: Serra, X. and Bonada, J. 1998. “Sound Transformations Based on the SMS High Level Attributes”, *Proceedings of International Conference on Digital Audio Effects,* Barcelona.

Appendix D

Python for audio signal processing

Original publication:

John Glover, Victor Lazzarini, and Joseph Timoney. Python for audio signal processing. In *Proceedings of the 2011 Linux Audio Conference*, Maynooth, Ireland, May 2011.

Python For Audio Signal Processing

John GLOVER, Victor LAZZARINI and **Joseph TIMONEY**
The Sound and Digital Music Research Group
National University of Ireland, Maynooth
Ireland
{John.C.Glover, Victor.Lazzarini}@nuim.ie
JTimoney@cs.nuim.ie

Abstract

This paper discusses the use of Python for developing audio signal processing applications. Overviews of Python language, NumPy, SciPy and Matplotlib are given, which together form a powerful platform for scientific computing. We then show how SciPy was used to create two audio programming libraries, and describe ways that Python can be integrated with the SndObj library and Pure Data, two existing environments for music composition and signal processing.

Keywords

Audio, Music, Signal Processing, Python, Programming

1 Introduction

There are many problems that are common to a wide variety of applications in the field of audio signal processing. Examples include procedures such as loading sound files or communicating between audio processes and sound cards, as well as digital signal processing (DSP) tasks such as filtering and Fourier analysis [Allen and Rabiner, 1977]. It often makes sense to rely on existing code libraries and frameworks to perform these tasks. This is particularly true in the case of building prototypes, a practise common to both consumer application developers and scientific researchers, as these code libraries allows the developer to focus on the novel aspects of their work.

Audio signal processing libraries are available for general purpose programming languages such as the GNU Scientific Library (GSL) for C/C++ [Galassi et al., 2009], which provides a comprehensive array of signal processing tools. However, it generally takes a lot more time to develop applications or prototypes in C/C++ than in a more lightweight scripting language. This is one of the reasons for the popularity of tools such as MATLAB [MathWorks, 2010], which allow the developer to easily manipulate matrices of numerical data, and includes implementations of many standard signal processing techniques. The major downside to MATLAB is that it is not free and not open source, which is a considerable problem for researchers who want to share code and collaborate. GNU Octave [Eaton, 2002] is an open source alternative to MATLAB. It is an interpreted language with a syntax that is very similar to MATLAB, and it is possible to write scripts that will run on both systems. However, with both MATLAB and Octave this increase in short-term productivity comes at a cost. For anything other than very basic tasks, tools such as integrated development environments (IDEs), debuggers and profilers are certainly a useful resource if not a requirement. All of these tools exist in some form for MATLAB/Octave, but

users must invest a considerable amount of time in learning to use a programming language and a set of development tools that have a relatively limited application domain when compared with general purpose programming languages. It is also generally more difficult to integrate MATLAB/Octave programs with compositional tools such as Csound [Vercoe et al., 2011] or Pure Data [Puckette, 1996], or with other technologies such as web frameworks, cloud computing platforms and mobile applications, all of which are becoming increasingly important in the music industry.

For developing and prototyping audio signal processing applications, it would therefore be advantageous to combine the power and flexibility of a widely adopted, open source, general purpose programming language with the quick development process that is possible when using interpreted languages that are focused on signal processing applications. Python [van Rossum and Drake, 2006], when used in conjunction with the extension modules NumPy [Oliphant, 2006], SciPy [Jones et al., 2001] and Matplotlib [Hunter, 2007] has all of these characteristics.

Section 2 provides a brief overview of the Python programming language. In Section 3 we discuss NumPy, SciPy and Matplotlib, which add a rich set of scientific computing functions to the Python language. Section 4 describes two libraries created by the authors that rely on SciPy, Section 5 shows how these Python programs can be integrated with other software tools for music composition, with final conclusions given in Section 6.

2 Python

Python is an open source programming language that runs on many platforms including Linux, Mac OS X and Windows. It is widely used and actively developed, has a vast array of code libraries and development tools, and integrates well with many other programming languages, frameworks and musical applications. Some notable features of the language include:

- It is a mature language and allows for programming in several different paradigms including imperative, object-orientated and functional styles.
- The clean syntax puts an emphasis on producing well structured and readable code. Python source code has often been compared to executable pseudocode.
- Python provides an interactive interpreter, which allows for rapid code development, prototyping and live experimentation.
- The ability to extend Python with modules written in C/C++ means that functionality can be quickly prototyped and then optimised later.
- Python can be embedded into existing applications.
- Documentation can be generated automatically from the comments and source code.
- Python bindings exist for cross-platform GUI toolkits such as Qt [Nokia, 2011].
- The large number of high-quality library modules means that you can quickly build sophisticated programs.

A complete guide to the language, including a comprehensive tutorial is available online at http://python.org.

3 Python for Scientific Computing

Section 3.1 provides an overview of three packages that are widely used for performing efficient numerical calculations and data visualisation using Python. Example programs that make use of these packages are given in Section 3.2.

3.1 NumPy, SciPy and Matplotlib

Python's scientific computing prowess comes largely from the combination of three related extension modules: NumPy, SciPy and Matplotlib. NumPy [Oliphant, 2006] adds a homogenous, multidimensional array object to Python. It also provides functions that perform efficient calculations based on array data. NumPy is written in C, and can be extended easily via its own C-API. As many existing scientific computing libraries are written in Fortran, NumPy comes with a tool called f2py which can parse Fortran files and create a Python extension module that contains all the subroutines and functions in those files as callable Python methods.

SciPy builds on top of NumPy, providing modules that are dedicated to common issues in scientific computing, and so it can be compared to MATLAB toolboxes. The SciPy modules are written in a mixture of pure Python, C and Fortran, and are designed to operate efficiently on NumPy arrays. A complete list of SciPy modules is available online at
http://docs.scipy.org, but examples include:

File input/output (scipy.io):
Provides functions for reading and writing files in many different data formats, including *.wav*, *.csv* and matlab data files (*.mat*).

Fourier transforms (scipy.fftpack):
Contains implementations of 1-D and 2-D fast Fourier transforms, as well as Hilbert and inverse Hilbert transforms.

Signal processing (scipy.signal):
Provides implementations of many useful signal processing techniques, such as waveform generation, FIR and IIR filtering and multi-dimensional convolution.

Interpolation (scipy.interpolate):
Consists of linear interpolation functions and cubic splines in several dimensions.

Matplotlib is a library of 2-dimensional plotting functions that provides the ability to quickly visualise data from NumPy arrays, and produce publication-ready figures in a variety of formats. It can be used interactively from the Python command prompt, providing similar functionality to MATLAB or GNU Plot [Williams et al., 2011]. It can also be used in Python scripts, web applications servers or in combination with several GUI toolkits.

3.2 SciPy Examples

Listing 1 shows how SciPy can be used to read in the samples from a flute recording stored in a file called *flute.wav*, and then plot them using Matplotlib. The call to the *read* function on line 5 returns a tuple containing the sampling rate of the audio file as the first entry and the audio samples as the second entry. The samples are stored in a variable called audio, with the first 1024 samples being plotted in line 8. In lines 10, 11 and 13 the axis labels and the plot title are set, and finally the plot is displayed in line 15. The image produced by Listing 1 is shown in Figure 1.

```
1 from scipy.io.wavfile import read
2 import matplotlib.pyplot as plt
3
```

```
# read audio samples
input_data = read("flute.wav")
audio = input_data[1]
# plot the first 1024 samples
plt.plot(audio[0:1024])
# label the axes
plt.ylabel("Amplitude")
plt.xlabel("Time (samples)")
# set the title
plt.title("Flute Sample")
# display the plot
plt.show()
```

Listing 1: Plotting Audio Files

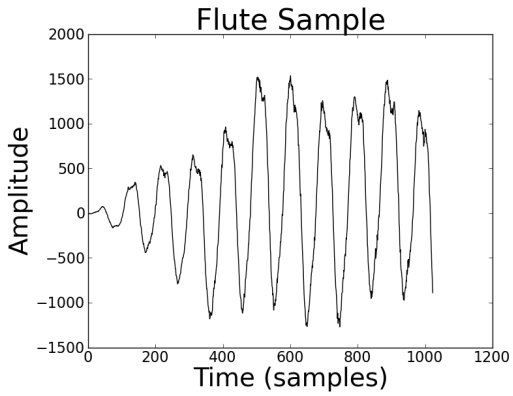

Figure 1: Plot of audio samples, generated by the code given in Listing 1.

In Listing 2, SciPy is used to perform a Fast Fourier Transform (FFT) on a windowed frame of audio samples then plot the resulting magnitude spectrum. In line 11, the SciPy *hann* function is used to compute a 1024 point Hanning window, which is then applied to the first 1024 flute samples in line 12. The FFT is computed in line 14, with the complex coefficients converted into polar form and the magnitude values stored in the variable *mags*. The magnitude values are converted from a linear to a decibel scale in line 16, then normalised to have a maximum value of 0 dB in line 18. In lines 20-26 the magnitude values are plotted and displayed. The resulting image is shown in Figure 2.

```
import scipy
from scipy.io.wavfile import read
from scipy.signal import hann
from scipy.fftpack import rfft
import matplotlib.pyplot as plt

# read audio samples
input_data = read("flute.wav")
audio = input_data[1]
# apply a Hanning window
window = hann(1024)
audio = audio[0:1024] * window
# fft
mags = abs(rfft(audio))
# convert to dB
mags = 20 * scipy.log10(mags)
# normalise to 0 dB max
mags -= max(mags)
# plot
plt.plot(mags)
# label the axes
plt.ylabel("Magnitude (dB)")
plt.xlabel("Frequency Bin")
# set the title
plt.title("Flute Spectrum")
plt.show()
```

Listing 2: Plotting a magnitude spectrum

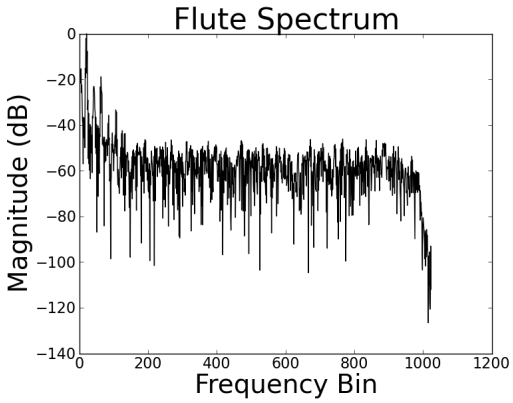

Figure 2: Flute magnitude spectrum produced from code in Listing 2.

4 Audio Signal Processing With Python

This section gives an overview of how SciPy is used in two software libraries that were created by the authors.

Section 4.1 gives an overview of Simpl [Glover et al., 2009], while Section 4.2 introduces Modal, our new library for musical note onset detection.

4.1 Simpl

Simpl [1] is an open source library for sinusoidal modelling [Amatriain et al., 2002] written in C/C++ and Python. The aim of this project is to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API, as well as provide implementations of some recently published sinusoidal modelling algorithms. Simpl is primarily intended as a tool for other researchers in the field, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms.

Simpl breaks the sinusoidal modelling process down into three distinct steps: peak detection, partial tracking and sound synthesis. The supported sinusoidal modelling implementations have a Python module associated with every step which returns data in the same format, irrespective of its underlying implementation. This allows analysis/synthesis networks to be created in which the algorithm that is used for a particular step can be changed without effecting the rest of the network. Each object has a method for real-time interaction as well as non-real-time or batch mode processing, as long as these modes are supported by the underlying algorithm.

All audio in Simpl is stored in NumPy arrays. This means that SciPy functions can be used for basic tasks such as reading and writing audio files, as well as more complex procedures such as performing additional processing, analysis or visualisation of the data. Audio samples are passed into a *PeakDetection* object for analysis, with detected peaks being returned as NumPy arrays that are used to build a list of *Peak* objects. Peaks are then passed to *PartialTracking* objects, which return partials that can be transferred to *Synthesis* objects to create a NumPy array of synthesised audio samples. Simpl also includes a module with plotting functions that use Matplotlib to plot analysis data from the peak detection and partial tracking analysis phases.

An example Python program that uses Simpl is given in Listing 3. Lines 6-8 read in the first 4096 sample values of a recorded flute note. As the default hop size is 512 samples, this will produce 8 frames of analysis data. In line 10 a *SndObjPeakDetection* object is created, which detects sinusoidal peaks in each frame of audio using the algorithm from The SndObj Library [Lazzarini, 2001]. The maximum number of detected peaks per frame is limited to 20 in line 11, before the peaks are detected and returned in line 12. In line 15 a *MQPartialTracking* object is created, which links previously detected sinusoidal peaks together to form partials, using the McAulay-Quatieri algorithm [McAulay and Quatieri, 1986]. The maximum number of partials is limited to 20 in line 16 and the partials are detected and returned in line 17. Lines 18-25 plot the partials, set the figure title, label the axes and display the final plot as shown in Figure 3.

```
1 import simpl
2 import matplotlib.pyplot as plt
3 from scipy.io.wavfile import read
4
5 # read audio samples
6 audio = read("flute.wav")[1]
7 # take just the first few frames
8 audio = audio[0:4096]
9 # Peak detection with SndObj
10 pd = simpl.SndObjPeakDetection()
```

[1] Available at http://simplsound.sourceforge.net

```
pd.max_peaks = 20
pks = pd.find_peaks(audio)
# Partial Tracking with
# the McAulay-Quatieri algorithm
pt = simpl.MQPartialTracking()
pt.max_partials = 20
partls = pt.find_partials(pks)
# plot the detected partials
simpl.plot.plot_partials(partls)
# set title and label axes
plt.title("Flute Partials")
plt.ylabel("Frequency (Hz)")
plt.xlabel("Frame Number")
plt.show()
```

Listing 3: A Simpl example

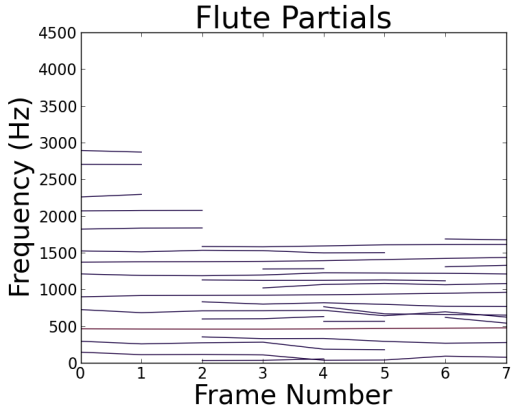

Figure 3: Partials detected in the first 8 frames of a flute sample, produced by the code in Listing 3. Darker colours indicate lower amplitude partials.

4.2 Modal

Modal [2] is a new open source library for musical onset detection, written in C++ and Python and released under the terms of the GNU General Public License (GPL). Modal consists of two main components: a code library and a database of audio samples. The code library includes implementations of three widely used onset detection algorithms from the literature and four novel onset detection systems created by the authors. The onset detection systems can work in a real-time streaming situation as well as in non-real-time. For more information on onset detection in general, a good overview is given in Bello et al. (2005).

The sample database contains a collection of audio samples that have creative commons licensing allowing for free reuse and redistribution, together with hand-annotated onset locations for each sample. It also includes an application that allows for the labelling of onset locations in audio files, which can then be added to the database. To the best of our knowledge, this is the only freely distributable database of audio samples together with their onset locations that is currently available. The Sound Onset Labellizer [Leveau et al., 2004] is a similar reference collection, but was not available at the time of publication. The sample set used by the Sound Onset Labellizer also makes use of files from the RWC database [Goto et al., 2002], which although publicly available is not free and does not allow free redistribution.

Modal makes extensive use of SciPy, with NumPy arrays being used to contain audio samples and analysis data from multiple stages of the onset detection process including computed onset detection functions, peak picking thresholds and the detected onset locations, while Matplotlib is used to plot the analysis results. All of the onset detection algorithms were written in Python and make use of SciPy's signal processing modules. The most computationally expensive part of the onset detection process is the calculation of the onset detection functions, so Modal also includes C++ implementations of all onset detection function modules. These

[2] Available at http://github.com/johnglover/modal

are made into Python extension modules using SWIG [Beazley, 2003]. As SWIG extension modules can manipulate NumPy arrays, the C++ implementations can be seamlessly interchanged with their pure Python counterparts. This allows Python to be used in areas that it excels in such as rapid prototyping and in "glueing" related components together, while languages such as C and C++ can be used later in the development cycle to optimise specific modules if necessary.

Listing 4 gives an example that uses Modal, with the resulting plot shown in Figure 4. In line 12 an audio file consisting of a sequence of percussive notes is read in, with the sample values being converted to floating-point values between -1 and 1 in line 14. The onset detection process in Modal consists of two steps, creating a detection function from the source audio and then finding onsets, which are peaks in this detection function that are above a given threshold value. In line 16 a *ComplexODF* object is created, which calculates a detection function based on the complex domain phase and energy approach described by Bello et al. (2004). This detection function is computed and saved in line 17. Line 19 creates an *OnsetDetection* object which finds peaks in the detection function that are above an adaptive median threshold [Brossier et al., 2004]. The onset locations are calculated and saved on lines 21-22. Lines 24-42 plot the results. The figure is divided into 2 subplots, the first (upper) plot shows the original audio file (dark grey) with the detected onset locations (vertical red dashed lines). The second (lower) plot shows the detection function (dark grey) and the adaptive threshold value (green).

```
1 from modal.onsetdetection \
2     import OnsetDetection
3 from modal.detectionfunctions \
4     import ComplexODF
5 from modal.ui.plot import \
6     (plot_detection_function,
7      plot_onsets)
8 import matplotlib.pyplot as plt
9 from scipy.io.wavfile import read
10
11 # read audio file
12 audio = read("drums.wav")[1]
13 # values between -1 and 1
14 audio = audio / 32768.0
15 # create detection function
16 codf = ComplexODF()
17 odf = codf.process(audio)
18 # create onset detection object
19 od = OnsetDetection()
20 hop_size = codf.get_hop_size()
21 onsets = od.find_onsets(odf) * \
22     hop_size
23 # plot onset detection results
24 plt.subplot(2,1,1)
25 plt.title("Audio And Detected "
26           "Onsets")
27 plt.ylabel("Sample Value")
28 plt.xlabel("Sample Number")
29 plt.plot(audio, "0.4")
30 plot_onsets(onsets)
31 plt.subplot(2,1,2)
32 plt.title("Detection Function "
33           "And Threshold")
34 plt.ylabel("Detection Function "
35            "Value")
36 plt.xlabel("Sample Number")
37 plot_detection_function(odf,
38                         hop_size)
39 thresh = od.threshold
40 plot_detection_function(thresh,
41                         hop_size,
42                         "green")
43 plt.show()
```

Listing 4: Modal example

5 Integration With Other Music Applications

This section provides examples of SciPy integration with two established tools for sound design and composition. Section 5.1 shows SciPy integration with The SndObj Library, with Section 5.2 providing an example of using SciPy in conjunction with Pure Data.

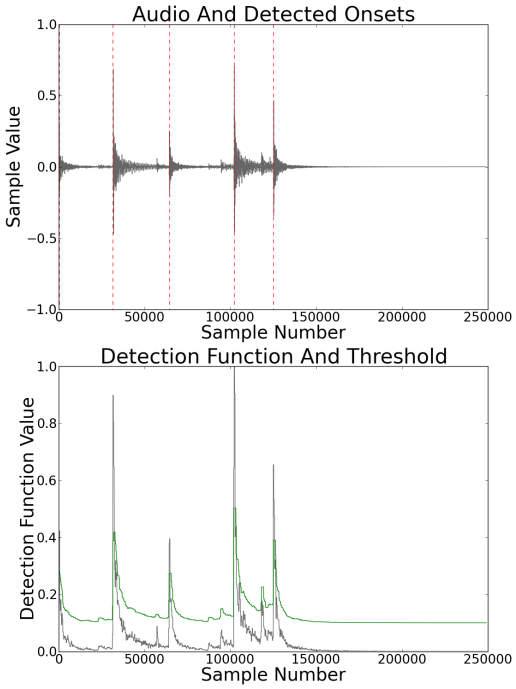

Figure 4: The upper plot shows an audio sample with detected onsets indicated by dashed red lines. The lower plot shows the detection function that was created from the audio file (in grey) and the peak picking threshold (in green).

5.1 The SndObj Library

The most recent version of The SndObj Library comes with support for passing NumPy arrays to and from objects in the library, allowing data to be easily exchanged between SndObj and SciPy audio processing functions. An example of this is shown in Listing 5. An audio file is loaded in line 8, then the *scipy.signal* module is used to low-pass filter it in lines 10-15. The filter cutoff frequency is given as 0.02, with 1.0 being the Nyquist frequency. A *SndObj* called *obj* is created in line 21 that will hold frames of the output audio signal. In lines 24 and 25, a *SndRTIO* object is created and set to write the contents of *obj* to the default sound output. Finally in lines 29-33, each frame of audio is taken, copied into *obj* and then written to the output.

```
1 from sndobj import \
2     SndObj, SndRTIO, SND_OUTPUT
3 import scipy as sp
4 from scipy.signal import firwin
5 from scipy.io.wavfile import read
6
7 # read audio file
8 audio = read("drums.wav")[1]
9 # use SciPy to low pass filter
10 order = 101
11 cutoff = 0.02
12 filter = firwin(order, cutoff)
13 audio = sp.convolve(audio,
14                     filter,
15                     "same")
16 # convert to 32-bit floats
17 audio = sp.asarray(audio,
18                    sp.float32)
19 # create a SndObj that will hold
20 # frames of output audio
21 obj = SndObj()
22 # create a SndObj that will
23 # output to the sound card
24 outp = SndRTIO(1, SND_OUTPUT)
25 outp.SetOutput(1, obj)
26 # get the default frame size
27 f_size = outp.GetVectorSize()
28 # output each frame
29 i = 0
30 while i < len(audio):
31     obj.PushIn(audio[i:i+f_size])
32     outp.Write()
33     i += f_size
```

Listing 5: The SndObj Library and SciPy

5.2 Pure Data

The recently released libpd [3] allows Pure Data to be embedded as a DSP library, and comes with a SWIG wrapper enabling it to be loaded as a Python extension module. Listing 6 shows how SciPy can be used in conjunction with libpd to process an audio file and save the result to disk. In lines 7-13 a *PdManager* object is created, that initialises libpd to work with a single channel of audio at a

[3] Available at http://gitorious.org/pdlib/libpd

sampling rate of 44.1 KHz. A Pure Data patch is opened in lines 14-16, followed by an audio file being loaded in line 20. In lines 22-29, successive audio frames are processed using the signal chain from the Pure Data patch, with the resulting data converted into an array of integer values and appended to the *out* array. Finally, the patch is closed in line 31 and the processed audio is written to disk in line 33.

```
1 import scipy as sp
2 from scipy import int16
3 from scipy.io.wavfile import \
4     read, write
5 import pylibpd as pd
6
7 num_chans = 1
8 sampling_rate = 44100
9 # open a Pure Data patch
10 m = pd.PdManager(num_chans,
11                   num_chans,
12                   sampling_rate,
13                   1)
14 p_name = "ring_mod.pd"
15 patch = \
16     pd.libpd_open_patch(p_name)
17 # get the default frame size
18 f_size = pd.libpd_blocksize()
19 # read audio file
20 audio = read("drums.wav")[1]
21 # process each frame
22 i = 0
23 out = sp.array([], dtype=int16)
24 while i < len(audio):
25     f = audio[i:i+f_size]
26     p = m.process(f)
27     p = sp.fromstring(p, int16)
28     out = sp.hstack((out, p))
29     i += f_size
30 # close the patch
31 pd.libpd_close_patch(patch)
32 # write the audio file to disk
33 write("out.wav", 44100, out)
```

Listing 6: Pure Data and SciPy

6 Conclusions

This paper highlighted just a few of the many features that make Python an excellent choice for developing audio signal processing applications. A clean, readable syntax combined with an extensive collection of libraries and an unrestrictive open source license make Python particularly well suited to rapid prototyping and make it an invaluable tool for audio researchers. This was exemplified in the discussion of two open source signal processing libraries created by the authors that both make use of Python and SciPy: Simpl and Modal. Python is easy to extend and integrates well with other programming languages and environments, as demonstrated by the ability to use Python and SciPy in conjunction with established tools for audio signal processing such as The SndObj Library and Pure Data.

7 Acknowledgements

The authors would like to acknowledge the generous support of An Foras Feasa, who funded this research.

References

J.B. Allen and L.R. Rabiner. 1977. A unified approach to short-time Fourier analysis and synthesis. *Proceedings of the IEEE*, 65(11), November.

X. Amatriain, Jordi Bonada, A. Loscos, and Xavier Serra, 2002. *DAFx - Digital Audio Effects*, chapter Spectral Processing, pages 373–438. John Wiley and Sons.

David M. Beazley. 2003. Automated scientific software scripting with SWIG. *Future Generation Computer Systems - Tools for Program Development and Analysis*, 19(5):599–609, July.

Juan Pablo Bello, Chris Duxbury, Mike Davies, and Mark Sandler. 2004. On the use of phase and energy for musical onset detection in the complex domain. *IEEE Signal Processing Letters*, 11(6):553–556, June.

Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark Sandler. 2005. A Tutorial on Onset Detection in Music Signals. *IEEE Transactions on Speech and Audio Processing*, 13(5):1035–1047, September.

Paul Brossier, Juan Pablo Bello, and Mark Plumbley. 2004. Real-time temporal segmentation of note objects in music signals. In *Proceedings of the International Computer Music Conference (ICMC'04)*, pages 458–461.

John W. Eaton. 2002. *GNU Octave Manual*. Network Theory Limited, Bristol, UK.

M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. 2009. *GNU Scientific Library Reference Manual*. Network Theory Limited, Bristol, UK, 3 edition.

John Glover, Victor Lazzarini, and Joseph Timoney. 2009. Simpl: A Python library for sinusoidal modelling. In *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, Como, Italy, September.

Masataka Goto, Hiroki Hashiguchi, Takuichi Nishimura, and Ryuichi Oka. 2002. RWC music database: Popular, classical, and jazz music databases. In *Proceedings of the 3rd International Conference on Music Information Retrieval (ISMIR 2002)*, pages 287–288, October.

John D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001. SciPy: Open source scientific tools for Python. http://www.scipy.org (last accessed 17-02-2011).

Victor Lazzarini. 2001. Sound processing with The SndObj Library: An overview. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, University of Limerick, Ireland, December.

Piere Leveau, Laurent Daudet, and Gael Richard. 2004. Methodology and tools for the evaluation of automatic onset detection algorithms in music. In *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR)*, Barcelona, Spain, October.

The MathWorks. 2010. *MATLAB Release R2010b*. The MathWorks, Natick, Massachusetts.

Robert McAulay and Thomas Quatieri. 1986. Speech analysis/synthesis based on a sinusoidal representation. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-34(4), August.

Nokia. 2011. Qt - a cross-platform application and UI framework. http://qt.nokia.com (last accessed 17-02-2011).

Travis Oliphant. 2006. *Guide To NumPy*. Trelgol Publishing, USA.

Miller Puckette. 1996. Pure Data. In *Proceedings of the International Computer Music Conference (ICMC'96)*, pages 224–227, San Francisco, USA.

Guido van Rossum and Fred L. Drake. 2006. *Python Language Reference Manual*. Network Theory Limited, Bristol, UK.

Barry Vercoe et al. 2011. The Csound Reference Manual. http://

www.csounds.com (last accessed 17-02-2011).

Thomas Williams, Colin Kelley, et al. 2011. Gnuplot. http://www.gnuplot.info (last accessed 17-02-2011).

Appendix E

Real-time detection of musical onsets with linear prediction and sinusoidal modelling

Original publication:

John Glover, Victor Lazzarini, and Joseph Timoney. Real-time detection of musical onsets with linear prediction and sinusoidal modeling. *EURASIP Journal on Advances in Signal Processing*, 2011(1):68, 2011.

Real-time detection of musical onsets with linear prediction and sinusoidal modelling

John Glover*, Victor Lazzarini and Joseph Timoney

The Sound and Digital Music Research Group,

National University of Ireland, Maynooth, Ireland

*Corresponding author: John.C.Glover@nuim.ie

E-mail addresses:

VL: Victor.Lazzarini@nuim.ie

JT: JTimoney@cs.nuim.ie

Abstract

Real-time musical note onset detection plays a vital role in many audio analysis processes, such as score following, beat detection and various sound synthesis by analysis methods. This article provides a review of some of the

most commonly used techniques for real-time onset detection. We suggest ways to improve these techniques by incorporating linear prediction as well as presenting a novel algorithm for real-time onset detection using sinusoidal modelling. We provide comprehensive results for both the detection accuracy and the computational performance of all of the described techniques, evaluated using *Modal*, our new open source library for musical onset detection, which comes with a free database of samples with hand-labelled note onsets.

1 Introduction

Many real-time musical signal-processing applications depend on the temporal segmentation of the audio signal into discrete note events. Systems such as score followers [1] may use detected note events to interact directly with a live performer. Beat-synchronous analysis systems [2, 3] group detected notes into beats, where a beat is the dominant time unit or metric pulse of the music, then use this knowledge to improve an underlying analysis process.

In sound synthesis by analysis, the choice of processing algorithm will often depend on the characteristics of the sound source. Spectral processing tools such as the Phase Vocoder [4] are a well-established means of time-stretching

and pitch-shifting harmonic musical notes, but they have well-documented weaknesses in dealing with noisy or transient signals [5]. For real-time applications of tools such as the Phase Vocoder, it may not be possible to depend on any prior knowledge of the signal to select the processing algorithm, and so we must be able to identify transient regions on-the-fly to reduce synthesis artefacts. It is within this context that onset detection will be studied in this article.

While there have been several recent studies that examined musical note onset detection [6, 7, 8], there have been few that analysed the real-time performance of the published techniques. One of the aims of this article is to provide such an overview. In Section 2, some of the common onset-detection techniques from the literature are described. In Section 3.1, we suggest a way to improve on these techniques by incorporating linear prediction (LP) [9]. In Section 4.1, we present a novel onset-detection method that uses sinusoidal modelling [10]. Section 5.1 introduces *Modal*, our new open source library for musical onset detection. This is then used to evaluate all of the previously described algorithms, with the results being given in Sections 5.2 and 5.3, and then discussed in Section 5.4. This evaluation includes details of the performance of all of the algorithms in terms of both accuracy and computational requirements.

2 Real-time onset detection

2.1 Definitions

This article distinguishes between the terms *audio buffer* and *audio frame* as follows:

Audio buffer: A group of consecutive audio samples taken from the input signal. The algorithms in this article all use a fixed buffer size of 512 samples.

Audio frame: A group of consecutive audio buffers. All the algorithms described here operate on overlapping, fixed-sized frames of audio. These frames are four audio buffers (2,048 samples) in duration, consisting of the most recent audio buffer which is passed directly to the algorithm, combined with the previous three buffers which are saved in memory. The start of each frame is separated by a fixed number of samples, which is equal to the buffer size.

In order to say that an onset-detection system runs in *real time*, we require two characteristics:

1. Low latency: The time between an onset occurring in the input audio stream and the system correctly registering an onset occurrence must be no more than 50 ms. This value was chosen to allow for the difficulty in specifying

reference onsets, which is described in more detail in Section 2.1.1. All of the onset-detection schemes that are described in this article have latency of 1,024 samples (the size of two audio buffers), except for the peak amplitude difference method (given in Section 4.3) which has an additional latency of 512 samples, or 1,536 samples of latency in total. This corresponds to latency times of 23.2 and 34.8 ms respectively, at a sampling rate of 44.1 kHz. The reason for the 1,024 sample delay on all the onset-detection systems is explained in Section 2.2.2, while the cause of the additional latency for the peak amplitude difference method is given in Section 4.3.

2. **Low processing time:** The time taken by the algorithm to process one frame of audio must be less than the duration of audio that is held in each buffer. As the buffer size is fixed at 512 samples, the algorithm must be able to process a frame in 11.6 ms or less when operating at a sampling rate of 44.1 kHz.

It is also important to draw a distinction between the terms *onset*, *transient* and *attack* in relation to musical notes. This article follows the definitions given in [6], summarised as follows:

Attack: The time interval during which the amplitude envelope increases.

Transient: A short interval during which the signal evolves in a relatively unpredictable way. It often corresponds to the time during which the excitation is applied then dampened.

Onset: A single instant marking the beginning of a transient.

2.1.1 The detection window

The process of verifying that an onset has been correctly detected is not straightforward. The ideal situation would be to compare the *detected onsets* produced by an onset-detection system with a list of *reference onsets*. An onset could then be said to be correctly detected if it lies within a chosen time interval around the reference onset, referred to here as the *detection window*. In reality, it is difficult to give exact values for reference onsets, particularly in the case of instruments with a soft attack, such as the flute or bowed violin. Finding reference onsets from natural sounds generally involves human annotation of audio samples. This inevitably leads to inconsistencies, and it was shown in [11] that the annotation process is dependent on the listener, the software used to label the onsets and the type of music being labelled. In [12], Vos and Rasch make a distinction between the *Physical Onset Time* and the *Perceptual Onset Time* of a musical note, which again can lead to differences between the values selected as reference onsets, par-

ticularly if there is a mixture of natural and synthetic sounds. To compensate for these limitations of the annotation process, we follow the decision made in a number of recent studies [6, 7, 8] to use a detection window that is 50 ms in duration.

2.2 The general form of onset-detection algorithms

As onset locations are typically defined as being the start of a transient, the problem of finding their position is linked to the problem of detecting transient intervals in the signal. Another way to phrase this is to say that onset detection is the process of identifying which parts of a signal are relatively unpredictable.

2.2.1 Onset-detection functions

The majority of the algorithms described in the literature involve an initial data reduction step, transforming the audio signal into an *onset-detection function* (ODF), which is a representation of the audio signal at a much lower sampling rate. The ODF usually consists of one value for every frame of audio, and should give a good indication as to the measure of the unpredictability of that frame. Higher values correspond to greater unpredictability. Figure 1 gives an example of a percussive audio sample together with an ODF calculated using the spectral difference method (see Section 2.3.2 for more details on this technique).

2.2.2 Peak detection

The next stage in the onset-detection process is to identify local maxima, also called *peaks*, in the ODF. The location of each peak is recorded as an onset location if the peak value is above a certain threshold. While peak picking and thresholding are described elsewhere in the literature [13], both require special treatment to operate with the limitations of strict real-time operation (defined in Section 2.1). As this article focuses on the evaluation of different ODFs in real-time, the peak-picking and thresholding processes are identical for each ODF.

When processing a real-time stream of ODF values, the first stage in the peak-detection algorithm is to see if the current values are local maxima. In order to make this assessment, the current ODF value must be compared to the two neighbouring values. As we cannot 'look ahead' to get the next ODF value, it is necessary to save both the previous and the current ODF values and wait until the next value has been computed to make the comparison. This means that there must always be some additional latency in the peak-picking process, in this case equal to the buffer size which is fixed at 512 samples. When working with a sampling rate of 44.1 kHz, this results in a total algorithm latency of two buffer sizes or approximately 23.2 ms. The process is summarised in Algorithm 1.

2.2.3 Threshold calculation

Thresholds are calculated using a slight variation of the median/mean function described in [14] and given by Equation 1, where σ_n is the threshold value at frame n, $O[n_m]$ is the previous m values of the ODF at frame n, λ is a positive median weighting value, and α is a positive mean weighting value:

$$\sigma_n = \lambda \times \text{median}(O[n_m]) + \alpha \times \text{mean}(O[n_m]) + N. \quad (1)$$

The difference between (1) and the formula in [14] is the addition of the term N, which is defined as

$$N = w \times v, \quad (2)$$

where v is the value of the largest peak detected so far, and w is a weighting value. For indefinite real-time use, it is advisable to either set $w = 0$ or to update w at regular intervals to account for changes in dynamic level. Figure 2 shows the values of the dynamic threshold (green dashes) of the ODF given in Figure 1, computed using $m = 7$, $\lambda = 1.0$, $\alpha = 2.0$ and $w = 0.05$. Every ODF peak that is above this threshold (highlighted in Figure 2 with red circles) is taken to be a note onset location.

2.3 Onset-detection functions

This section reviews several existing approaches to creating ODFs that can be used in a real-time situation. Each technique operates on frames of *N* samples, with the start of each frame being separated by a fixed buffer size of *h* samples. The ODFs return one value for every frame, corresponding to the likelihood of that frame containing a note onset. A full analysis of the detection accuracy and computational efficiency of each algorithm is given in Section 5.

2.3.1 Energy ODF

This approach, described in [5], is the most simple conceptually and is the most computationally efficient. It is based on the premise that musical note onsets often have more energy than the steady-state component of the note, as in the case of many instruments, this is when the excitation is applied. Larger changes in the amplitude envelope of the signal should therefore coincide with onset locations. For each frame, the energy is given by

$$E(n) = \sum_{m=0}^{N} x(m)^2, \tag{3}$$

where $E(n)$ is the energy of frame n, and $x(m)$ is the value of the mth sample in the frame. The value of the energy ODF (ODF_E) for frame *n* is the absolute value

of the difference in energy values between consecutive frames:

$$\text{ODF}_E(n) = |E(n) - E(n-1)|. \tag{4}$$

2.3.2 Spectral difference ODF

Many recent techniques for creating ODFs have tended towards identifying time-varying changes in a frequency domain representation of an audio signal. These approaches have proven to be successful in a number of areas, such as in detecting onsets in polyphonic signals [15] and in detecting 'soft' onsets created by instruments such as the bowed violin which do not have a percussive attack [16]. The spectral difference ODF (ODF_{SD}) is calculated by examining frame-to-frame changes in the Short-Time Fourier Transform [17] of an audio signal and so falls into this category.

The Fourier transform of the nth frame, windowed using a Hanning window *w(m)* of size *N* is given by

$$X(k,n) = \sum_{m=0}^{N-1} x(m)w(m)e^{-\frac{2j\pi mk}{N}}, \tag{5}$$

where $X(k,n)$ is the kth frequency bin of the nth frame.

The spectral difference [16] is the absolute value of the change in magnitude between corresponding bins in consecutive frames. As a new musical onset will

often result in a sudden change in the frequency content in an audio signal, large changes in the average spectral difference of a frame will often correspond with note onsets. The spectral difference ODF is thus created by summing the spectral difference across all bins in a frame and is given by

$$\mathrm{ODF}_{\mathrm{SD}}(n) = \sum_{k=0}^{N/2} ||X(k,n)| - |X(k,n-1)||. \tag{6}$$

2.3.3 Complex domain ODF

Another way to view the construction of an ODF is in terms of *predictions* and *deviations* from predicted values. For every spectral bin in the Fourier transform of a frame of audio samples, the spectral difference ODF predicts that the next magnitude value will be the same as the current one. In the steady state of a musical note, changes in the magnitude of a given bin between consecutive frames should be relatively low, and so this prediction should be accurate. In transient regions, these variations should be more pronounced, and so the average deviation from the predicted value should be higher, resulting in peaks in the ODF.

Instead of making predictions using only the bin magnitudes, the complex domain ODF [18] attempts to improve the prediction for the next value of a given bin using combined magnitude and phase information. The magnitude prediction

is the magnitude value from the corresponding bin in the previous frame. In polar form, we can write this predicted value as

$$\hat{R}(k,n) = |X(k,n-1)|. \tag{7}$$

The phase prediction is formed by assuming a constant rate of phase change between frames:

$$\hat{\phi}(k,n) = \text{princarg}[2\varphi(k,n-1) - \varphi(k,n-2)], \tag{8}$$

where princarg maps the phase to the $[-\pi, \pi]$ range, and $\varphi(k,n)$ is the phase of the kth bin in the nth frame. If $R(k,n)$ and $\phi(k,n)$ are the actual values of the magnitude and phase, respectively, of bin *k* in frame *n*, then the deviation between the prediction and the actual measurement is the Euclidean distance between the two complex phasors, which can be written as

$$\Gamma(k,n) = \sqrt{R(k,n)^2 + \hat{R}(k,n)^2 - 2R(k,n)\hat{R}(k,n)\cos(\phi(k,n) - \hat{\phi}(k,n))}. \tag{9}$$

The complex domain ODF (ODF_{CD}) is the sum of these deviations across all the bins in a frame, as given in

$$\text{ODF}_{\text{CD}}(n) = \sum_{k=0}^{N/2} \Gamma(k,n). \tag{10}$$

3 Measuring signal predictability

The ODFs that are described in Section 2.3, and the majority of those found elsewhere in the literature [6], are trying to distinguish between the steady-state and transient regions of an audio signal by making predictions based on information about the most recent frame of audio and one or two preceding frames. In this section, we present methods that use the same basic signal information to the approaches described in Section 2.3, but instead of making predictions based on just one or two frames of these data, we use an arbitrary number of previous values combined with LP to improve the accuracy of the estimate. The ODF is then the absolute value of the differences between the actual frame measurements and the LP predictions. The ODF values are low when the LP prediction is accurate, but larger in regions of the signal that are more unpredictable, which should correspond with note onset locations.

This is not the first time that LP errors have been used to create an ODF. The authors in [19] describe a somewhat similar system in which an audio signal is first filtered into six non-overlapping sub-bands. The first five bands are then decimated by a factor of 20:1 before being passed to a LP error filter, while just the amplitude envelope is taken from the sixth band (everything above the note B7 which is 3,951 kHz). Their ODF is the sum of the five LP error signals and the

amplitude envelope from the sixth band.

Our approach differs in a number of ways. In this article we show that LP can be used to improve the detection accuracy of the three ODFs described in Section 2.3 (detection results are given in Section 5). As this approach involves predicting the time-varying changes in signal features (energy, spectral difference and complex phasor positions) rather than in the signal itself, the same technique could be applied to many existing ODFs from the literature, and so it can be viewed as an additional post-processing step that can potentially improve the detection accuracy of existing ODFs. Our algorithms are suitable for real-time use, and the results were compiled from real-time data. In contrast, the results given in [19] are based on off-line processing, and include an initial pre-processing step to normalise the input audio files, and so it is not clear how well this method performs in a real-time situation.

The LP process that is used in this article is described in Section 3.1. In Sections 3.2, 3.3 and 3.4, we show that this can be used to create new ODFs based on the energy, spectral difference and complex domain ODFs, respectively.

3.1 Linear prediction

In the LP model, also known as the autoregressive model, the current input sample $x(n)$ is estimated by a weighted combination of the past values of the signal. The predicted value, $\hat{x}(n)$, is computed by FIR filtering according to

$$\hat{x}(n) = \sum_{k=1}^{p} a_k x(n-k), \tag{11}$$

where p is the order of the LP model and a_k are the prediction coefficients.

The challenge is then to calculate the LP coefficients. There are a number of methods given in the literature, the most widespread among which are the autocorrelation method [20], covariance method [9] and the Burg method [21]. Each of the three methods was evaluated, but the Burg method was selected as it produced the most accurate and consistent results. Like the autocorrelation method, it has a minimum phase, and like the covariance method it estimates the coefficients on a finite support [21]. It can also be efficiently implemented in real time [20].

3.1.1 The Burg algorithm

The LP error is the difference between the predicted and the actual values:

$$e(n) = x(n) - \hat{x}(n). \tag{12}$$

The Burg algorithm minimises average of the forward prediction error $f_m(n)$ and the backward prediction error $b_m(n)$. The initial (order 0) forward and backward errors are given by

$$f_0(n) = x(n), \tag{13}$$

$$b_0(n) = x(n) \tag{14}$$

over the interval $n = 0, \ldots, N-1$, where N is the block length. For the remaining $m = 1, \ldots, p$, the mth coefficient is calculated from

$$k_m = \frac{-2\sum_{n=m}^{N-1}[f_{m-1}(n)b_{m-1}(n-1)]}{\sum_{n=m}^{N-1}[f_{m-1}^2(n) + b_{m-1}^2(n-1)]}, \tag{15}$$

and then the forward and backward prediction errors are recursively calculated from

$$f_m(n) = f_{m-1}(n) - k_m b_{m-1}(n-1) \tag{16}$$

for $n = m+1, \ldots, N-1$, and

$$b_m(n) = b_{m-1}(n-1) - k_m f_{m-1}(n) \tag{17}$$

for $n = m, \ldots, N-1$, respectively. Pseudocode for this process is given in Algorithm 2, taken from [21].

3.2 Energy with LP

The energy ODF (given in Section 2.3.1) is derived from the absolute value of the energy difference between two frames. This can be viewed as using the energy value of the first frame as a prediction of the energy of the second, with the difference being the prediction error. In this context, we try to improve this estimate using LP. Energy values from the past p frames are taken, resulting in the sequence

$$E(n-1), E(n-2), \ldots, E(n-p).$$

Using (13)–(17), p coefficients are calculated based on this sequence, and then a one-sample prediction is made using (11). Hence, for each frame, the energy with LP ODF (ODF_{ELP}) is given by

$$\text{ODF}_{\text{ELP}}(n) = |E(n) - P_E(n)|, \tag{18}$$

where $P_E(n)$ is the predicted energy value for frame n.

3.3 Spectral difference with LP

Similar techniques can be applied to the spectral difference and complex domain ODFs. The spectral difference ODF is formed from the absolute value of the magnitude differences between corresponding bins in adjacent frames. Similarly to the process described in Section 3.2, this can be viewed as a prediction that

the magnitude in a given bin will remain constant between adjacent frames, with the magnitude difference being the prediction error. In the spectral difference with LP ODF (ODF_{SDLP}), the predicted magnitude value for each of the k bins in frame n is calculated by taking the magnitude values from the corresponding bins in the previous p frames, using them to find p LP coefficients then filtering the result with (11). Hence, for each k in n, the magnitude prediction coefficients are formed using (13)–(17) on the sequence

$$|X(k, n-1)|, |X(k, n-2)|, \ldots, |X(k, n-p)|.$$

If $P_{\text{SD}}(k, n)$ is the predicted spectral difference for bin k in n, then

$$\text{ODF}_{\text{SDLP}}(n) = \sum_{k=0}^{N/2} ||X(k, n)| - P_{\text{SD}}(k, n)|. \tag{19}$$

As is shown in Section 5.3, this is a significant amount of extra computation per frame compared with the ODF_{SD} given by Equation 6. However, it is still capable of real-time performance, depending on the chosen LP model order. We found that an order of 5 was enough to significantly improve the detection accuracy while still comfortably meeting the real-time processing requirements. Detailed results are given in Section 5.

3.4 Complex domain with LP

The complex domain method described in Section 2.3.3 is based on measuring the Euclidean distance between the predicted and the actual complex phasors for a given bin. There are a number of different ways by which LP could be applied in an attempt to improve this estimate. The bin magnitudes and phases could be predicted separately, based on their values over the previous p frames, and then combined to form an estimated phasor value for the current frame. Another possibility would be to only apply LP to one of either the magnitude or the phase parameters.

However, we found that the biggest improvement came from using LP to estimate the value of the Euclidean distance that separates the complex phasors for a given bin between consecutive frames. Hence, for each bin k in frame n, the complex distances between the kth bin in each of the last p frames are used to calculate the LP coefficients. If $R(k,n)$ is the magnitude of the kth bin in frame n, and $\phi(k,n)$ is the phase of the bin, then the distance between the kth bins in frames n and $n-1$ is

$$\Gamma(k,n) = \sqrt{R(k,n)^2 + R(k,n-1)^2 - 2R(k,n)R(k,n-1)\cos(\phi(k,n) - \phi(k,n-1))}.$$

LP coefficients are formed from the values

$$\Gamma(k, n-1), \Gamma(k, n-2), \ldots, \Gamma(k, n-p)$$

using (13)–(17), and predictions $P_{\mathrm{CD}}(k, n)$ are calculated using (11). The complex domain with LP ODF ($\mathrm{ODF}_{\mathrm{CDLP}}$) is then given by

$$\mathrm{ODF}_{\mathrm{CDLP}}(n) = \sum_{k=0}^{N/2} |\Gamma(k, n) - P_{\mathrm{CD}}(k, n)|. \tag{20}$$

4 Real-time onset detection using sinusoidal modelling

In Section 3, we describe a way to improve the detection accuracy of several ODFs from the literature using LP to enhance their estimates of the frame-by-frame evolution of an audio signal. This improvement in detection accuracy comes at the expense of much greater computational cost, however (see Section 5 for detection accuracy and performance results).

In this section, we present a novel ODF that has significantly better real-time performance than the LP-based spectral methods. It uses sinusoidal modelling, and so it is particularly useful in areas that include some sort of harmonic analysis. We begin with an overview of sinusoidal modelling in Section 4.1, followed by

a review of previous study that uses sinusoidal modelling for onset detection in Section 4.2 and then concludes with a description of the new ODF in Section 4.3.

4.1 Sinusoidal modelling

Sinusoidal modelling [10] is based on Fourier's theorem, which states that any periodic waveform can be modelled as the sum of sinusoids at various amplitudes and harmonic frequencies. For stationary pseudo-periodic sounds, these amplitudes and frequencies evolve slowly with time. They can be used as parameters to control pseudo-sinusoidal oscillators, commonly referred to as partials. The audio signals can be calculated from the sum of the partials using

$$s(t) = \sum_{p=1}^{N_p} A_p(t) \cos(\theta_p(t)), \tag{21}$$

$$\theta_p(t) = \theta_p(0) + 2\pi \int_0^t f_p(u)\, du, \tag{22}$$

where N_p is the number of partials and A_p, f_p and θ_p are the amplitude, frequency and phase of the pth partial, respectively. Typically, the parameters are measured for every

$$t = nh/F_{\mathrm{s}},$$

where n is the sample number, h is the buffer size and F_{s} is the sampling rate. To calculate the audio signal, the parameters must then be interpolated between

measurements. Calculating these parameters for each frame is referred to in this article as *peak detection*, while the process of connecting these peaks between frames is called *partial tracking*.

4.2 Sinusoidal modelling and onset detection

The sinusoidal modelling process can be extended, creating models of sound based on the separation of the audio signal into a combination of sinusoids and noise [22], and further into combinations of sinusoids, noise and transients [23]. Although primarily intended to model transient components from musical signals, the system described in [23] could also be adopted to detect note onsets. The authors show that transient signals in the time domain can be mapped onto sinusoidal signals in a frequency domain, in this case, using the discrete cosine transform (DCT) [24]. Roughly speaking, the DCT of a transient time-domain signal produces a signal with a frequency that depends only on the time shift of the transient. This information could then be used to identify when the onset occurred. However, it is not suitable for real-time applications as it requires a DCT frame size that makes the transients appear as a small entity, with a frame duration of about 1 s recommended. This is far too much a latency to meet the real-time requirements that were specified in Section 2.1.

Another system that combines sinusoidal modelling and onset detection is presented in [25]. It creates an ODF that is a combination of two energy measurements. The first is simply the energy in the audio signal over a 512 sample frame. If the energy of the current frame is larger than that of a given number of previous frames, then the current frame is a candidate for being an onset location. A multi-resolution sinusoidal model is then applied to the signal to isolate the harmonic component of the sound. This differs from the sinusoidal modelling implementation described above in that the audio signal is first split into five octave spaced frequency bands. Currently, only the lower three are used, while the upper two (frequencies above about 5 kHz) are discarded. Each band is then analysed using different window lengths, allowing for more frequency resolution in the lower band at the expense of worse time resolution. Sinusoidal amplitude, frequency and phase parameters are estimated separately for each band, and linked together to form partials. An additional post-processing step is then applied, removing any partials that have an average amplitude that is less than an adaptive psychoacoustic masking threshold, and removing any partials that are less than 46 ms in duration.

As it stands, it is unclear whether or not the system described in [25] is suitable for use as a real-time onset detector. The stipulation that all sinusoidal partials must be at least 46 ms in duration implies that there must be a minimum latency

of 46 ms in the sinusoidal modelling process, putting it very close to our 50 ms limit. If used purely as an ODF in the onset-detection system described in Section 2.3, the additional 11.6 ms of latency incurred by the peak-detection stage would put the total latency outside this 50-ms window. However, their method uses a rising edge detector instead of looking for peaks, and so it may still meet our real-time requirements. Although as it was designed as part of a larger system that was primarily intended to encode audio for compression, no onset-detection accuracy or performance results are given by the authors.

In contrast, the ODF that is presented in Section 4.3 was designed specifically as a real-time onset detector, and so has a latency of just two buffer sizes (23.2 ms in our implementation). As we discussed in Section 5, it compares favourably to leading approaches from the literature in terms of computational efficiency, and it is also more accurate than the reviewed methods.

4.3 Peak amplitude difference ODF

This ODF is based on the same underlying premise as sinusoidal models, namely that during the steady state of a musical note, the harmonic signal component can be well modelled as a sum of sinusoids. These sinusoids should evolve slowly in time, and should therefore be well represented by the partials detected by the sinu-

soidal modelling process. It follows then that during the steady state, the absolute values of the frame-to-frame differences in the sinusoidal peak amplitudes and frequencies should be quite low. In comparison, transient regions at note onset locations should show considerably more frame-by-frame variation in both peak frequency and amplitude values. This is due to two main factors:

1. Many musical notes have an increase in signal energy during their attack regions, corresponding to a physical excitation being applied, which increases the amplitude of the detected sinusoidal components.

2. As transients are by definition less predictable and less harmonic, the basic premise of the sinusoidal model breaks down in these regions. This can result in peaks existing in these regions that are really noise and not part of any underlying harmonic component. Often they will remain unmatched, and so do not form long-duration partials. Alternatively, if they are incorrectly matched, then it can result in relatively large amplitude and/or frequency deviations in the resulting partial. In either case, the difference between the parameters of the noisy peak and the parameters of any peaks before and after it in a partial will often differ significantly.

Both these factors should lead to larger frame-to-frame sinusoidal peak amplitude differences in transient regions than in steady-state regions. We can therefore create an ODF by analysing the differences in peak amplitude values over consecutive frames.

The sinusoidal modelling algorithm that we used is very close to the one described in [26], with a couple of changes to the peak-detection process. Firstly, the number of peaks per frame can be limited to M_p, reducing the computation required for the partial-tracking stage [27, 28]. If the number of detected peaks $N_p > M_p$, then the M_p largest amplitude peaks will be selected. Also, in order to allow for consistent evaluation with the other frequency domain ODFs described in this article, the frame size is kept constant during the analysis (2,048 samples). The partial-tracking process is identical to the one given in [26]. As this partial-tracking algorithm has a delay of one buffer size, this ODF has an additional latency of 512 samples, bringing the total detection latency (including the peak-picking phase) to 1,536 samples or 34.8 ms when sampled at 44.1 kHz.

For a given frame n, let $P_k(n)$ be the peak amplitude of the kth partial. The peak amplitude difference ODF ($\mathrm{ODF}_{\mathrm{PAD}}$) is given by

$$\mathrm{ODF}_{\mathrm{PAD}}(n) = \sum_{k=0}^{M_p} |P_k(n) - P_k(n-1)|. \tag{23}$$

In the steady state, frame-to-frame peak amplitude differences for matched peaks should be relatively low, and as the matching process here is significantly easier than in transient regions, less matching errors are expected. At note onsets, matched peaks should have larger amplitude deviations due to more energy in the signal, and there should also be more unmatched or incorrectly matched noisy peaks, increasing the ODF value. As specified in [26], unmatched peaks for a frame are taken to be the start of a partial, and so the amplitude difference is equal to the amplitude of the peak, $P_k(n)$.

5 Evaluation of real-time ODFs

This section provides evaluations of all of the ODFs described in this article. Section 5.1 describes a new library of onset-detection software, which includes a database of hand-annotated musical note onsets, which was created as part of this study. This database was adopted to assess the performance of the different algorithms. Section 5.2 evaluates the detection accuracy of each ODF, with their computational complexities described in Section 5.3. Section 5.4 concludes with a discussion of the evaluation results.

5.1 Musical onset database and library (modal)

In order to evaluate the different ODFs described in Sections 2.3, 3 and 4.3, it was necessary to access a set of audio files with reference onset locations. To the best of our knowledge, the Sound Onset Labellizer [11] was the only freely available reference collection, but unfortunately it was not available at the time of publication. Their reference set also made use of files from the RWC database [29], which although publicly available is not free and does not allow free redistribution.

These issues lead to the creation of Modal, which contains a free collection of samples, all with creative commons licensing allowing for free reuse and redistribution, and including hand-annotated onsets for each file. Modal is also a new open source (GPL), cross-platform library for musical onset detection written in C++ and Python, and contains implementations of all of the ODFs discussed in this article in both programming languages. In addition, from Python, there is onset detection and plotting functionality, as well as code for generating our analysis data and results. It also includes an application that allows for the labelling of onset locations in audio files, which can then be added to the database. Modal is available now at http://github.com/johnglover/modal.

5.2 Detection results

The detection accuracy of the ODFs was measured by comparing the onsets detected using each method with the reference samples in the Modal database. To be marked as 'correctly detected', the onset must be located within 50 ms of a reference onset. Merged or double onsets were not penalised. The database currently contains 501 onsets from annotated sounds that are mainly monophonic, and so this must be taken into consideration when viewing the results. The annotations were also all made by one person, and while it has been shown in [11] that this is not ideal, the chosen detection window of 50 ms should compensate for some of the inevitable inconsistencies.

The results are summarised by three measurements that are common in the field of Information Retrieval [15]: the precision (P), the recall (R), and the F-measure (F) defined here as follows:

$$P = \frac{C}{C + f_{\mathrm{p}}}, \tag{24}$$

$$R = \frac{C}{C + f_{\mathrm{n}}}, \tag{25}$$

$$F = \frac{2PR}{P + R}, \tag{26}$$

where C is the number of correctly detected onsets, f_{p} is the number of false positives (detected onsets with no matching reference onset), and f_{n} is the number

of false negatives (reference onsets with no matching detected onset).

Every reference sample in the database was streamed one buffer at a time to each ODF, with ODF values for each buffer being passed immediately to a real-time peak-picking system, as described in Algorithm 1. Dynamic thresholding was applied according to (1), with $\lambda = 1.0$, $\alpha = 2.0$, and w in (2) set to 0.05. A median window of seven previous values was used. These parameters were kept constant for each ODF. Our novel methods that use LP (described in Sections 3.2, 3.3 and 3.4) each used a model order of 5, while our peak amplitude difference method described in Section 4.3 was limited to a maximum of 20 peaks per frame.

The precision, recall and F-measure results for each ODF are given in Figures 3, 4 and 5, respectively. In each figure, the blue bars give the results for the ODFs from the literature (described in Section 2.3), the brown bars give the results for our LP methods, and the green bar gives the results for our peak amplitude difference method.

Figure 3 shows that the precision values for all our methods are higher than the methods from the literature. The addition of LP noticeably improves each ODF to which it is applied to. The precision values for the peak amplitude difference method is better than the literature methods and the energy with LP method, but worse than the two spectral-based LP methods.

The recall results for each ODF are given in Figure 4. In this figure, we see that LP has improved the energy method, but made the spectral difference and complex domain methods slightly worse. The peak amplitude difference method has a greater recall than all of the literature methods and is only second to the energy with LP ODF.

Figure 5 gives the F-measure for each ODF. All of our proposed methods are shown to perform better than the methods from the literature. The spectral difference with LP ODF has the best detection accuracy, while the energy with LP, complex domain with LP and peak amplitude difference methods are all closely matched.

5.3 Performance results

In Table 1, we give the worst-case number of floating-point operations per second (FLOPS) required by each ODF to process real-time audio streams, based on our implementations in the Modal library. This analysis does not include data from the setup/initialisation periods of any of the algorithms, or data from the peak-detection stage of the onset-detection system. As specified in Section 2.1, the audio frame size is 2,048 samples, the buffer size is 512 samples, and the sampling rate is 44.1 kHz. The LP methods all use a model of the order of 5. The number

of peaks in the $\mathrm{ODF}_{\mathrm{PAD}}$ is limited to 20.

These totals were calculated by counting the number of floating-point operations required by each ODF to process 1 frame of audio, where we define a floating-point operation to be an addition, subtraction, multiplication, division or assignment involving a floating-point number. As we have a buffer size of 512 samples measured at 44.1 kHz, we have 86.133 frames of audio per second, and so the number of operations required by each ODF per frame of audio was multiplied by 86.133 to get the FLOPS total for the corresponding ODF.

To simplify the calculations, the following assumptions were made when calculating the totals:

- As we are using the real fast Fourier transform (FFT) computed using the FFTW3 library [30], the processing time required for a FFT is $2.5N \log_2(N)$ where N is the FFT size, as given in [31].

- The complexity of basic arithmetic functions in the C++ standard library such as $\sqrt{}$, $\cos$, $\sin$, and $\log$ is $O(M)$, where M is the number of digits of precision at which the function is to be evaluated.

- All integer operations can be ignored.

- All function call overheads can be ignored.

As Table 1 shows, the energy-based methods (ODF_E and $\mathrm{ODF}_{\mathrm{ELP}}$) require far less computation than any of the others. The spectral difference ODF is the third fastest, needing about half the number of operations that are required by the complex domain method. The worst-case requirements for the peak amplitude difference method are still relatively close to the spectral difference ODF and noticeably quicker than the complex domain ODF. As expected, the addition of LP to the spectral difference and complex domain methods makes them significantly more expensive computationally than any other technique.

To give a more intuitive view of the algorithmic complexity, in Table 2, we also give the estimated real-time CPU usage for each ODF given as a percentage of the maximum number of FLOPS that can be achieved by two different processors: an Intel Core 2 Duo and an Analog Devices ADSP-TS201S (TigerSHARC). The Core 2 Duo has a clock speed of 2.8 GHz, a 6 MB L2 cache and a bus speed of 1.07 GHz, providing a theoretical best-case performance of 22.4 GFLOPS [32]. The ADSP-TS201S has a clock speed of 600 MHz and a best-case performance of 3.6 GFLOPS [33], and scores relatively well on the BDTI DSP Kernel Benchmarks [34]. Any value less than 100% here shows that the ODF can be calculated in real time on this processor.

5.4 Discussion

The F-measure results (shown in Figure 5) for the methods described in Section 2.3 are lower than those given elsewhere in the literature, but this was expected as real-time performance is significantly more challenging at the peak-picking and thresholding stages. The nature of the sample set must also be taken into account, as evidently, the heavy bias towards monophonic sounds is reflected by the surprisingly strong performance of the energy-based methods. As noted in [8], the various parameter settings can have a large impact on overall performance. We tried to select a parameter set that gave a fair reflection on each algorithm, but it must be noted that every method can probably be improved by some parameter adjustments, especially if prior knowledge of the sound source is available.

In terms of performance, the LP methods are all significantly slower than their counterparts. However, even the most computationally expensive algorithm can run with an estimated real-time CPU usage of just over 6% on the ADSP-TS201S (TigerSHARC) processor, and so they are still more than capable in respect of real-time performance. The energy with LP ODF in particular is extremely cheap computationally, and yet has relatively good detection accuracy for this sample set.

The peak amplitude difference method is also notable as it is computationally

cheaper than the complex domain ODF and compares favourably with the spectral difference ODF, while giving better accuracy for our sample set than the other two. For applications such as real-time sound synthesis, which may already include a sinusoidal modelling process, this becomes an extremely quick method of onset detection. One significant difference between the peak amplitude difference ODF and the others is that the computation time is not fixed, but depends on the sound source. Harmonic material will have well-defined partials, potentially requiring more processing time for the partial-tracking process than noisy sound sources, for this sinusoidal modelling implementation at least.

6 Conclusions

In this article, we have described two new approaches to real-time musical onset detection, one using LP and the other using sinusoidal modelling. We compared these approaches to some of the leading real-time musical onset-detection algorithms from the literature, and found that they can offer either improved accuracy, computational efficiency, or both. It is recognised that onset-detection results are very context sensitive, and so without a more extensive sample set it is hard to make completely conclusive comparisons to other methods. However, our soft-

ware and our sample database are both released under open source licences and are freely redistributable, so hopefully other researchers in the field will contribute.

Choosing a real-time ODF remains a complex issue and depends on the nature of the input sound, the available processing power and the penalties that will be experienced for producing false negatives and false positives. However, some recommendations can be made based on the results in this article. For our sample set, the spectral difference with LP method produced the most accurate results, and so, if computational complexity is not an issue, then this would be a good choice. On the other hand, if low complexity is an important requirement then the energy with LP ODF is an attractive option. It produced accurate results at a fraction of the computational cost of some of the established methods.

The peak amplitude difference ODF is also noteworthy and should prove to be useful in areas such as real-time sound synthesis by analysis. Spectral processing techniques such as the Phase Vocoder or sinusoidal models work well during the steady-state regions of musical notes, but have problems in transient areas which follow note onsets [5, 23]. One solution to this problem is to identify these regions and process them differently, which requires accurate onset detection to avoid synthesis artefacts. It is in this context that the peak amplitude difference ODF is particularly useful. It was shown to provide more accurate results than the

well-established complex domain method with noticeably lower computation requirements, and as it integrates seamlessly with the sinusoidal modelling process, it can be added to the existing sinusoidal modelling systems at very little cost.

7 Competing interests

The authors declare that they have no competing interests.

Acknowledgements

The authors would like to acknowledge the generous support received from the Irish research institute *An Foras Feasa* who funded this research.

References

[1] N. Orio, S. Lemouton, and D. Schwarz, "Score following: State of the art and new developments," in *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, (Montreal, Canada), 2003.

[2] A. Stark, D. Matthew, and M. Plumbley, "Real-time beat-synchronous analysis of musical audio," in *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, (Como, Italy), September 2009.

[3] N. Schnell, D. Schwarz, and R. Muller, "X-micks - interactive content based real-time audio processing," in *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)*, (Montreal, Canada), 2006.

[4] M. Dolson, "The phase vocoder: A tutorial," *Computer Music Journal*, vol. 10, pp. 14–27, Winter 1986.

[5] C. Duxbury, M. Davies, and M. Sandler, "Improved time-scaling of musical audio using phase locking at transients," in *112th Audio Engineering Society Convention*, (Munich, Germany), May 2002.

[6] J. P. Bello, L. Daudet, S. Abdallah, C. Duxbury, M. Davies, and M. Sandler, "A Tutorial on Onset Detection in Music Signals," *IEEE Transactions on Speech and Audio Processing*, vol. 13, pp. 1035–1047, Sept. 2005.

[7] D. Stowell and M. Plumbley, "Adaptive whitening for improved real-time audio onset detection," in *Proceedings of the International Computer Music Conference (ICMC'07)*, (Copenhagen, Denmark), pp. 312–319, August 2007.

[8] S. Dixon, "Onset detection revisited," in *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)*, (Montreal, Canada), September 2006.

[9] J. Makhoul, "Linear prediction: A tutorial review," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 561–580, 1975.

[10] X. Amatriain, J. Bonada, A. Loscos, and X. Serra, *DAFx - Digital Audio Effects*, ch. Spectral Processing, pp. 373–438. John Wiley and Sons, 2002.

[11] P. Leveau, L. Daudet, and G. Richard, "Methodology and tools for the evaluation of automatic onset detection algorithms in music," in *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR)*, (Barcelona, Spain), October 2004.

[12] J. Vos and R. Rasch, "The perceptual onset of musical tones," *Perception and Psychophysics*, vol. 29, no. 4, pp. 323–335, 1981.

[13] I. Kauppinen, "Methods for detecting impulsive noise in speech and audio signals," in *Proceedings of the 14th International Conference on Digital Signal Processing (DSP 2002)*, vol. 2, pp. 967–970, 2002.

[14] P. Brossier, J. P. Bello, and M. Plumbley, "Real-time temporal segmentation of note objects in music signals," in *Proceedings of the International Computer Music Conference (ICMC'04)*, pp. 458–461, 2004.

[15] "Mirex 2009 audio onset detection results." http://www.music-ir.org/mirex/wiki/2009:Audio_Onset_Detection_Results (last accessed 05-10-2010).

[16] C. Duxbury, M. Sandler, and M. Davies, "A hybrid approach to musical note onset detection," in *Proceedings of the 5th International Conference on Digital Audio Effects (DAFx-02)*, (Hamburg, Germany), pp. 33–38, September 2002.

[17] J. Allen and L. Rabiner, "A unified approach to short-time Fourier analysis and synthesis," *Proceedings of the IEEE*, vol. 65, pp. 1558–1564, November 1977.

[18] J. P. Bello, C. Duxbury, M. Davies, and M. Sandler, "On the use of phase and energy for musical onset detection in the complex domain," *IEEE Signal Processing Letters*, vol. 11, pp. 553–556, June 2004.

[19] W.-C. Lee and C.-C. J. Kuo, "Musical onset detection based on adaptive linear prediction," in *Proceedings of the 2006 IEEE Conference on Multimedia and Expo, ICME 2006*, (Ontario, Canada), pp. 957–960, July 2006.

[20] F. Keiler, D. Arfib, and U. Zolzer, "Efficient linear prediction for digital audio effects," in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*, (Verona, Italy), December 200.

[21] M. Lagrange, S. Marchand, M. Raspaud, and J.-B. Rault, "Enhanced partial tracking using linear prediction," in *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03)*, (London, UK), September 2003.

[22] X. Serra and J. Smith, "Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decompostion," *Computer Music Journal*, vol. 14, pp. 12–24, Winter 1990.

[23] T. S. Verma and T. H. Y. Meng, "Extending spectral modeling synthesis with transient modeling synthesis," *Computer Music Journal*, vol. 24, pp. 47–59, Summer 2000.

[24] N. Ahmed, T. Natarajan, and K. Rao, "Discrete cosine transfom," *IEEE Transactions on Computers*, vol. C-23, pp. 90–93, January 1974.

[25] S. Levine, *Audio Representations for Data Compression and Compressed Domain Processing*. PhD thesis, Stanford University, 1998.

[26] R. McAulay and T. Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-34, pp. 744–754, August 1986.

[27] V. Lazzarini, J. Timoney, and T. Lysaght, "Alternative analysis-synthesis approaches for timescale, frequency and other transformations of musical signals," in *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx-05)*, (Madrid, Spain), pp. 18–23, 2005.

[28] V. Lazzarini, J. Timoney, and T. Lysaght, "Time-stretching using the instantaneous frequency distribution and partial tracking," in *Proceedings of the International Computer Music Conference (ICMC'05)*, (Barcelona, Spain), 2005.

[29] M. Goto, H. Hashiguchi, T. Nishimura, and R. Oka, "RWC music database: Popular, classical, and jazz music databases," in *Proceedings of the 3rd International Conference on Music Information Retrieval (ISMIR 2002)*, pp. 287–288, October 2002.

[30] M. Frigo and S. G. Johnson, "Fftw3 library." http://www.fftw.org (last accessed 29-01-2011).

[31] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[32] Intel Corporation, "Intel microprocessor export compliance metrics." http://www.intel.com/support/processors/sb/cs-023143.htm (last accessed 13-04-2011).

[33] Analog Devices, "ADSP-TS201S data sheet." http://www.analog.com/static/imported-files/data_sheets/ADSP_TS201S.pdf (last accessed 13-04-2011).

[34] Berkeley Design Technology, Inc., "BDTI DSP kernel benchmarks (BDTIMark200) certified results." http://www.bdti.com/Resources/BenchmarkResults/BDTIMark2000 (last accessed 13-04-2011).

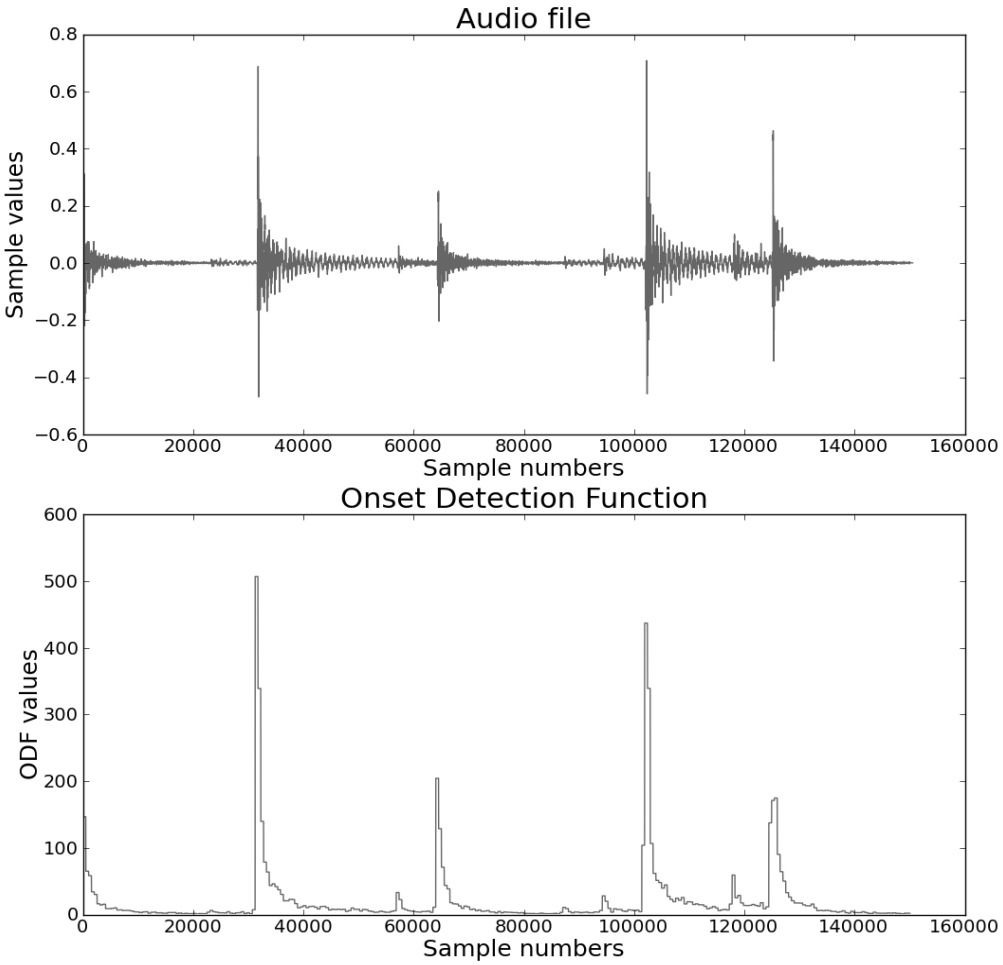

Figure 1: Percussive audio sample with ODF generated using the spectral difference method.

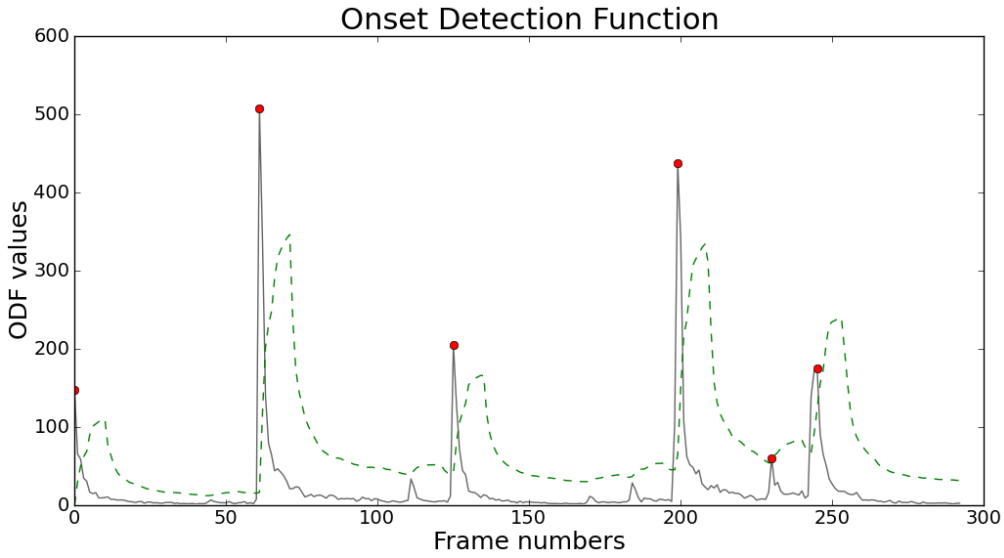

Figure 2: ODF peaks detected (*circled*) **and threshold** (*dashes*) **during real-time peak picking.**

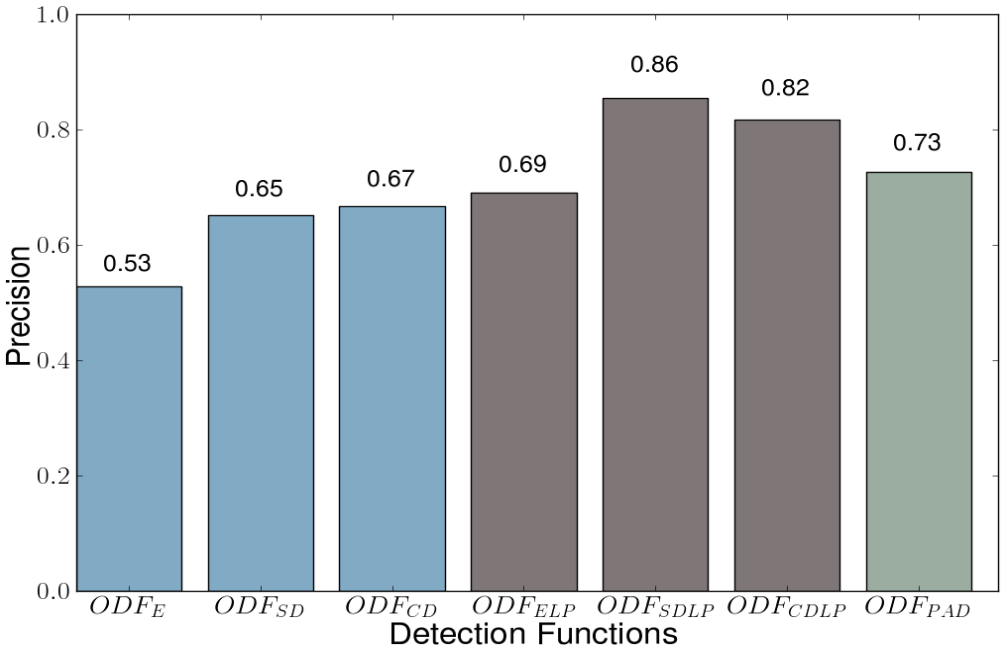

Figure 3: Precision values for each ODF.

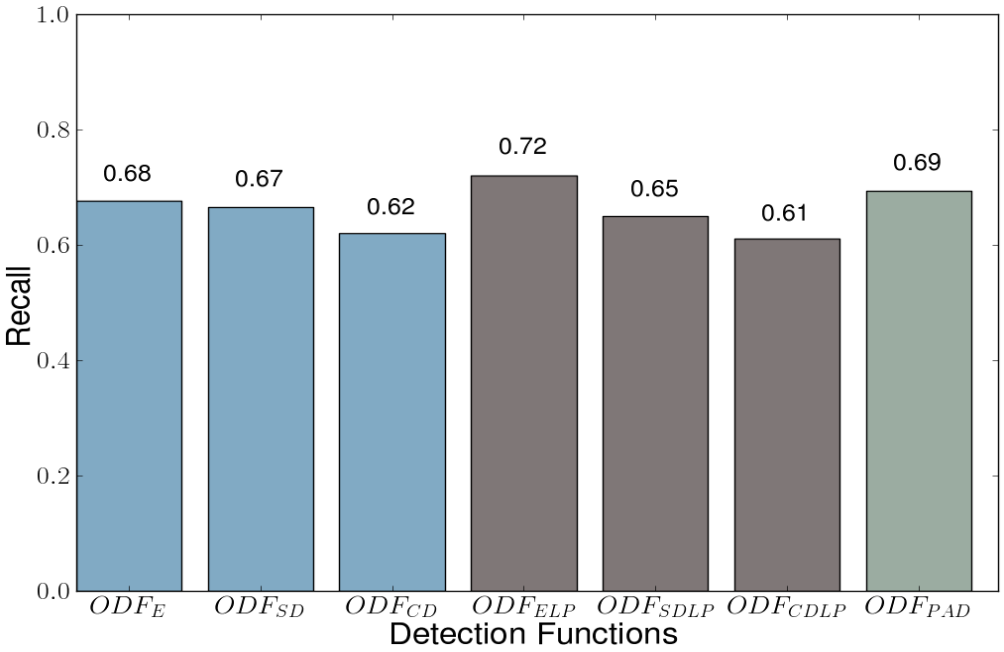

Figure 4: Recall values for each ODF.

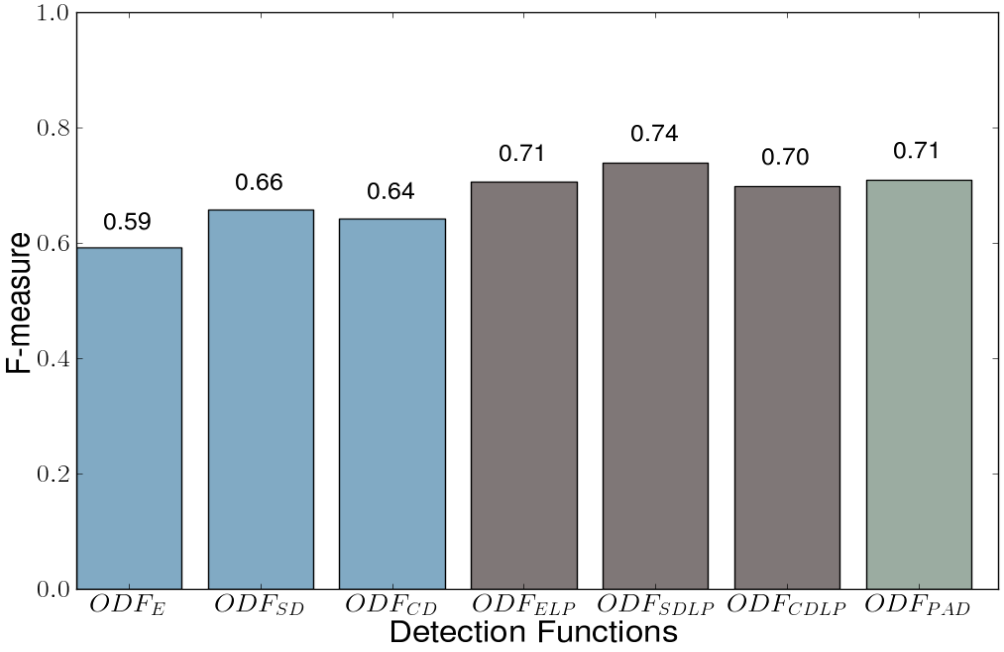

Figure 5: F-measure values for each ODF.

	FLOPS
ODF_{E}	529,718
$\mathrm{ODF}_{\mathrm{SD}}$	7,587,542
$\mathrm{ODF}_{\mathrm{CD}}$	14,473,789
$\mathrm{ODF}_{\mathrm{ELP}}$	734,370
$\mathrm{ODF}_{\mathrm{SDLP}}$	217,179,364
$\mathrm{ODF}_{\mathrm{CDLP}}$	217,709,168
$\mathrm{ODF}_{\mathrm{PAD}}$	9,555,940

Table 1: Number of floating-point operations per second (FLOPS) required by each ODF to process real-time audio streams, with a buffer size of 512 samples, a frame size of 2048 samples, a linear prediction model of the order of 5, and a maximum of 20 peaks per frame for $\mathrm{ODF}_{\mathrm{PAD}}$

	Core 2 Duo (%)	ADSP-TS201S (%)
ODF_{E}	0.002	0.015
$\mathrm{ODF}_{\mathrm{SD}}$	0.034	0.211
$\mathrm{ODF}_{\mathrm{CD}}$	0.065	0.402
$\mathrm{ODF}_{\mathrm{ELP}}$	0.003	0.020
$\mathrm{ODF}_{\mathrm{SDLP}}$	0.970	6.033
$\mathrm{ODF}_{\mathrm{CDLP}}$	0.972	6.047
$\mathrm{ODF}_{\mathrm{PAD}}$	0.043	0.265

Table 2: Estimated real-time CPU usage for each ODF, shown as a percentage of the maximum number of FLOPS that can be achieved on two processors: an Intel Core 2 Duo and an Analog Devices ADSP-TS201S (TigerSHARC)

Input: ODF value

Output: Whether or not previous ODF value represents a peak (Boolean)

$IsOnset \longleftarrow False$

if *PreviousValue* $>$ *CurrentValue* **and** *PreviousValue* $>$ *TwoValuesAgo* **then**

  if *PreviousValue* $>$ *CalculateThreshold()* **then**
    $IsOnset \longleftarrow True$

  end

end

$UpdatePreviousValues()$

return *IsOnset*

Algorithm 1: Real-time peak picking (one buffer delay).

$f \longleftarrow x$

$b \longleftarrow x$

$a \longleftarrow x$

for $m \leftarrow 0$ **to** $p - 1$ **do**

$fp \longleftarrow f$ without its first element

$bp \longleftarrow b$ without its last element

$k \longleftarrow -2bp \cdot fp/(fp \cdot fp + fp \cdot fp)$

$f \longleftarrow fp + k \cdot bp$

$b \longleftarrow bp + k \cdot fp$

$a \longleftarrow (a[0], a[1], \ldots, a[m], 0) + k(0, a[m], a[m-1], \ldots, a[0])$

end

Algorithm 2: The Burg method.

Appendix F

Real-time segmentation of the temporal evolution of musical sounds

Original publication:

John Glover, Victor Lazzarini, and Joseph Timoney. Real-time segmentation of the temporal evolution of musical sounds. In *The Acoustics 2012 Hong Kong Conference*, Hong Kong, China, May 2012.

1. INTRODUCTION

The segmentation of musical instrument sounds into contiguous regions with distinct characteristics has become an important process in studies of timbre perception [1] and sound modelling and manipulation [2]. Since the time of Helmholtz, it has been known that the temporal evolution of musical sounds plays an important role in our perception of timbre. Helmholtz described musical sounds as being a waveform shaped by an amplitude envelope consisting of attack, steady state and decay segments [3]. Here the attack is the time from the onset until the amplitude reaches its peak value, the steady state is the segment during which the amplitude is approximately constant, and the decay is the region where the amplitude decreases again. A number of automatic segmentation techniques have been developed based on this model, taking only the temporal evolution of the signal amplitude into account when calculating region boundaries [4, 5]. However, more recent research has shown that in order to better understand the temporal evolution of sounds, it is necessary to also consider the way in which the audio spectrum changes over time [1]. In [6] Hajda proposed a model for the segmentation of isolated, continuant musical tones based on the relationship between the temporal evolution of the amplitude envelope and the spectral centroid, called the amplitude/centroid trajectory (ACT) model. Caetano et al. devised an automatic segmentation technique based on the ACT model and showed that it outperformed a segmentation method based solely on the temporal evolution of the amplitude envelope [7]. While this method works well in an off-line situation, it cannot be used to improve real-time systems. We are particularly interested in real-time musical performance tools based on sound synthesis by analysis, where the choice of processing algorithm will often depend on the characteristics of the sound source. Spectral processing tools such as the Phase Vocoder [8] are well established means of time-stretching and pitch-shifting harmonic musical notes, but they have well documented weaknesses in dealing with noisy or transient signals [9]. For real-time applications of tools such as the Phase Vocoder, it may not be possible to depend on any prior knowledge of the signal to select the processing algorithm, so being able to accurately identify note regions with specific characteristics on-the-fly is crucial in order to minimize synthesis artifacts.

In this paper, we present a new technique for the real-time automatic temporal segmentation of musical sounds. We define attack, sustain and release segments using cues from a combination of the amplitude envelope, the spectral centroid, and a measurement of the stability of the sound that is derived from an onset detection function. In Section 2 we describe some existing approaches to automatic segmentation. Our new method is given in Section 3. We provide an evaluation of our method in Section 4, followed by conclusions in Section 5.

2. AUTOMATIC SEGMENTATION

Automatic segmentation consists of the identification of boundaries between contiguous regions in a musical note. Typical boundaries are at note onsets, the end of the attack or the start of the sustain, the end of the sustain or the start of the release and at the end of the note (offset). Regions and boundaries can vary however, firstly depending on the model used by the segmentation technique and secondly based on the nature of the sound being analyzed, as not all instrumental sounds are composed of the same temporal events. In [4] Peeters notes that the well-known ADSR envelope does not apply to most natural sounds, as depending on the nature of the sound, one or more of the segments is often missing. Therefore, he proposes segmenting musical sounds into two regions named *attack* and *rest*. This only requires the detection of two region boundaries; the start and end of the attack region. Two techniques are described for detecting these boundaries. The first of these is just to apply a fixed threshold to the amplitude envelope, the start of attack being when the envelope reaches 20% of the peak value and

the end of attack occurring when it reaches 90% of the peak value. The second technique is called the *weakest effort* method, and is based on an indirect measurement of the changes in the slope of the amplitude envelope.

Although the amplitude envelope often provides a good approximation of the temporal evolution of the internal structure of a musical sound, it simply does not provide enough information to allow for accurate, robust and meaningful segmentation of the signal into regions with distinct characteristics. In particular the attack region, which has often become synonymous with the amplitude rise time [10], is not well delineated by the amplitude envelope. The attack is a transient part of the signal that lasts from the onset until a relatively stable periodicity is established, and as a result the steady state is generally achieved before the end of the initial amplitude rise time [1]. During the steady state, the amplitude envelope can often show considerable variation, particularly in the presence of tremolo and/or vibrato. This makes it difficult to detect the boundary between the steady state and the release using just the amplitude envelope, especially if operating under the constraints of a real-time system. The ACT model, which we describe in Section 2.1, has addressed many of these issues.

2.1. Automatic segmentation using the amplitude/centroid trajectory model

Hajda proposed a new model for the partitioning of isolated non-percussive musical sounds [6], based on observations by Beauchamp that for certain signals the root mean square (RMS) amplitude and spectral centroid have a monotonic relationship during the steady state region [11]. An example of this relationship is shown for a clarinet sample in Figure 1. The spectral centroid is given by Equation 1, where f is frequency (in Hz) and a is linear amplitude of frequency band b up to m bands which are computed by Fast Fourier Transform. The Fourier Transform is performed on Bartlett windowed analysis frames that are 64 samples in duration. This results in 32 evenly spaced frequency bands (up to 11025 Hz), each with a bandwidth of about 345 Hz.

$$centroid(t) = \frac{\sum_{b=1}^{m} f_b(t) \times a_b(t)}{\sum_{b=1}^{m} a_b(t)} \tag{1}$$

Hajda's model, called the amplitude/centroid trajectory (ACT), identifies and detects the boundaries for four contiguous regions in a musical tone:

Attack: the portion of the signal in which the RMS amplitude is rising and the spectral centroid is falling after an initial maximum. The attack ends when the centroid slope changes direction (centroid reaches a local minimum).

Attack/steady state transition: the region from the end of the attack to the first local RMS amplitude maximum.

Steady state: the segment in which the amplitude and spectral centroid both vary around mean values.

Decay: the section during which the amplitude and spectral centroid both rapidly decrease. At the end of the decay (near the note offset), the centroid value can rise again however as the signal amplitude can become so low that denominator in Equation 1 will approach 0. This can be seen in Figure 1 (starting at approximately sample number 100200).

Hajda initially applied the ACT model only to non-percussive sounds. However, Caetano et al. introduced an automatic segmentation technique based on the ACT model [7], and proposed that it could be applied to a large variety of acoustic instrument tones. It uses cues taken from a combination of

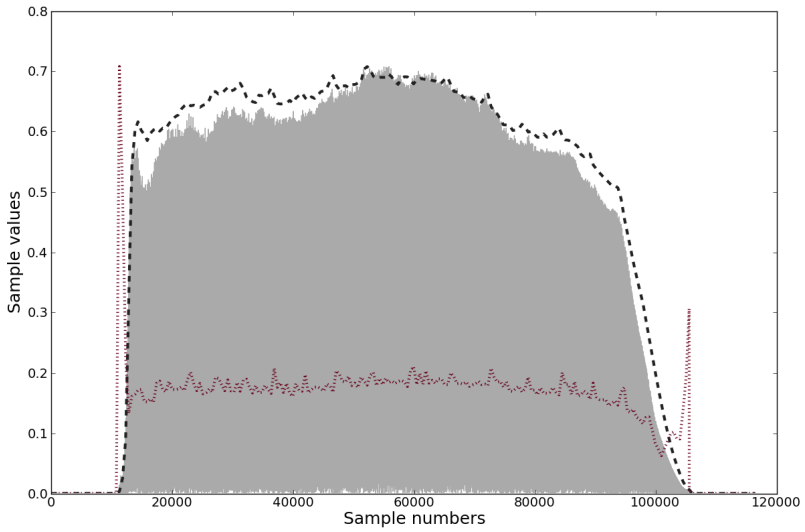

FIGURE 1. The relationship between the RMS amplitude envelope and spectral centroid for a clarinet sample. The full-wave-rectified version of the audio sample is given, with the RMS amplitude envelope shown by the black dashes, and the spectral centroid by the red dots. The RMS amplitude envelope and the spectral centroid have both been normalised and scaled by the maximum signal value.

the amplitude envelope and the spectral centroid, where the amplitude envelope is calculated using a technique called the true amplitude envelope (TAE) [12]. The TAE is a time domain implementation of the true envelope [13], which is a method for estimating a spectral envelope by iteratively calculating the filtered cepstrum, then modifying it so that the original spectral peaks are maintained while the cepstral filter is used to fill the valleys between the peaks. In the TAE this algorithm is applied to the time domain signal instead of the Fourier spectrum, so that the resulting envelope accurately matches the time domain amplitude peaks.

For each musical tone the onset, end of attack, start of sustain, start of release and offset boundaries are detected as follows:

Onset: start of the note, found by using the automatic onset detection method described in [14]. This technique basically involves looking for signal regions in which the center of gravity of the instantaneous energy of the windowed signal is above a given threshold. Or in other words, if most of the energy in a spectral frame is located towards the leading edge of the analysis window, then the frame is likely to contain a note onset.

End of attack: position of the first local minima in the spectral centroid that is between boundaries 1 and 3.

Start of sustain: boundary detected using a modified version of Peeters' weakest effort method.

Start of release: also detected using a version of the weakest effort method, but starting at the offset and working backwards.

Offset: the last point that the TAE attains the same energy (amplitude squared) as the onset.

Notably, they allow the same point to define the boundary of two distinct contiguous regions. This signifies that the region is too short to be detected as a separate segment and makes the model more

robust in dealing with different types of sounds.

Caetano et al. compare the performance of their automatic segmentation technique to that of the one described by Peeters [4]. They do this by visual inspection of plots of the waveform, spectrogram and detected boundaries produced by both methods, showing 16 analyzed samples consisting of isolated tones from western orchestral instruments (plus the acoustic guitar). They find that their model outperformed the Peeters method in all cases, although for one sample (a marimba recording) the amplitude envelope and spectral centroid do not behave in the manner that is assumed by the model, and so neither method gives good results. However, this provides strong evidence that the ACT model assumptions can be applied to a wide variety of sounds, and shows that using a combination of the amplitude envelope spectral centroid can lead to more accurate note segmentation than methods based on the amplitude envelope alone.

The automatic segmentation technique proposed by Caetano et al. cannot be used to improve the performance of real-time synthesis by analysis systems however, as the method for detecting the start of sustain and start of release boundaries requires knowledge of future signal values. Also, although the spectral centroid has been shown to be a useful indirect indicator as to the extent of the attack region, in order to help reduce synthesis artifacts when using tools such as the Phase Vocoder it would be preferable to have a more accurate measure of the attack transient, by locating the signal regions in which the spectral components are changing rapidly and often unpredictably. We address both of these issues in Section 3.

3. A REAL-TIME METHOD FOR THE AUTOMATIC TEMPORAL SEGMENTATION OF MUSICAL SOUNDS

In this section we propose a new method for the real-time automatic segmentation of the temporal evolution of musical sounds, using cues from a combination of the RMS amplitude envelope, the spectral centroid and an onset detection function (the latter is described in Section 3.1). In our segmentation model, boundaries are defined for the onset, start of sustain, start of release and offset as follows:

Onset: start of the note, detected using the peak amplitude difference method [15].

Start of sustain (end of attack): boundary occurs as soon as the attack transient has finished. This calculation is described in detail in Section 3.1.

Start of release (end of sustain): occurs when the following conditions are met:

(1) The RMS amplitude envelope is less than 80% of the largest amplitude value seen between the onset and the current frame.

(2) The RMS amplitude envelope is decreasing for 5 consecutive frames.

(3) The current value of the spectral centroid is below the cumulative moving average of the values of the centroid from the onset to the current frame.

This boundary also occurs if the RMS amplitude value drops to less than 33% of the peak value. The RMS amplitude here is subject to a 3 point moving average filter, and the spectral centroid is given by Equation 1.

Offset: the point at which the RMS amplitude value is 60 db below the peak amplitude value.

Similarly to the Caetano et al. method [7], we allow multiple boundaries to occur at the same time. An example of the boundaries detected by our method is given in Figure 3, with boundary positions shown by vertical blue dashes. We use a frame size of 512 samples, resulting in a latency of 11.6 ms when operating at a sampling rate of 44.1 kHz. The maximum delay in detecting a boundary is 5 frames (or 58 ms).

3.1. Identifying the attack transient from an onset detection function

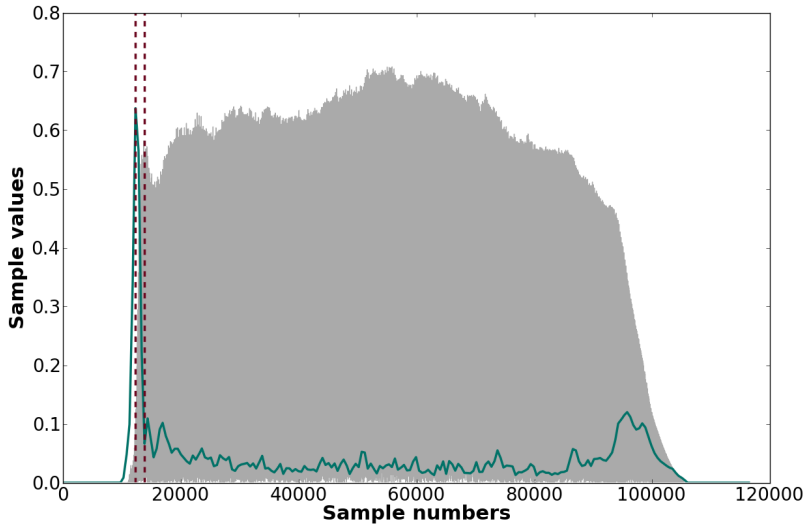

FIGURE 2. Onset detection function (solid green line) for the clarinet sample, and detected transient region (between red vertical dashes).

As onset locations are typically defined as being the start of a transient, the problem of finding their position is linked to the problem of detecting transient regions in the signal. Another way to phrase this is to say that onset detection is the process of identifying which parts of a signal are relatively unpredictable. The majority of the onset detection algorithms described in the literature involve an initial data reduction step, transforming the audio signal into an *onset detection function* (ODF), which is a representation of the audio signal at a much lower sampling rate. The ODF usually consists of one value for every frame of audio and should give a good indication as to the measure of the unpredictability of that frame. Higher values correspond to greater unpredictability. The onset detection function used in our model is the peak amplitude difference method, one of the best-performing methods discussed in [15]. It is based on the premise that during the steady state of a musical note, a quasi-harmonic signal can be well modelled as a sum of sinusoidal partials with slowly evolving amplitudes, frequencies and phases. Therefore, the absolute values of the frame-to-frame differences in the sinusoidal peak amplitudes and frequencies should be quite low. In comparison, transient regions at note onset locations should show considerably more frame-by-frame variation in both peak frequency and amplitude values, so an ODF can be created by measuring these frame-by-frame amplitude and frequency variations. As this effectively measures errors in the partial tracking stage of sinusoidal modelling [17], it can also be used to measure the stability of the detected sinusoidal partials in the audio signal. Peaks in the ODF should therefore occur at regions where the spectral components in the signal are most unstable or are changing unpredictably. It should be noted that this does not only apply to our ODF, but indeed any ODF that measures the variability of spectral components in the audio signal.

We define the attack transient as being the region from the onset until the next local minimum in the ODF. Additionally, we also signal the end of the attack segment if the RMS amplitude envelope reaches a local maxima. This technique is similar to the transient detection method proposed in [9], where the authors detect transient regions based on peaks in the energy of the noise signal resulting from the identification and removal of the deterministic signal component. However, as we do not do a

separation of the deterministic component from the stochastic, our method should require considerably less computation. In addition, we do not low-pass filter the resulting ODF, as doing so widens the ODF peak (and in turn, the detected transient region), without presenting an obvious way to compensate for this deviation. An example of the ODF and corresponding transient region can be seen in Figure 2.

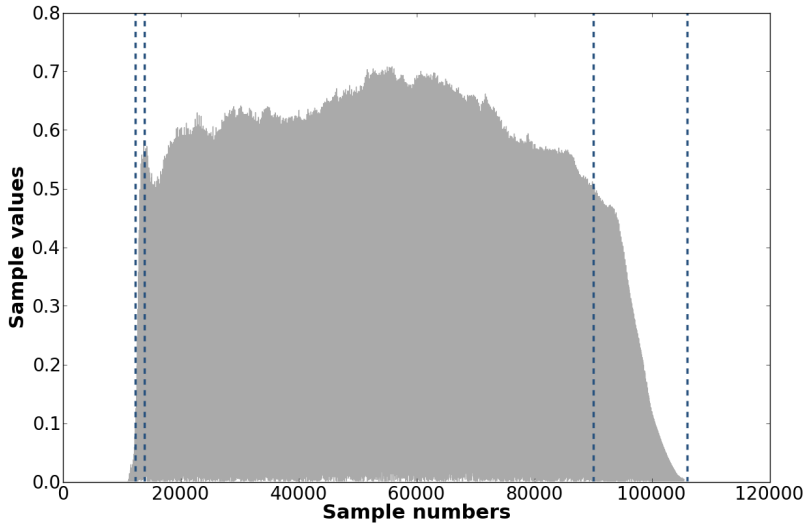

FIGURE 3. Boundaries detected by our proposed real-time segmentation method. The boundaries (from left to right) are the onset, start of sustain (end of attack), start of release (end of sustain) and offset.

4. RESULTS

To evaluate the performance of our proposed segmentation model, we compared the locations of the detected boundaries with those found by our implementation of the method given by Caetano et al. [7]. We took a selection of 36 samples of isolated musical sounds from the Modal database [15], which is a freely available database of samples with creative commons licensing allowing for free reuse and redistribution. More information about the Modal database can be found at `http://www.johnglover.net`. The samples are quite diverse in nature, covering percussive and non-percussive sounds from a mixture of western orchestral instruments, contemporary western instruments and vocal samples. Initially created to evaluate the performance of real-time onset detection algorithms, Modal includes hand-annotated onset locations for each sample. For this work, three additional annotations were added for each sample: start of sustain, start of release and note offset. The annotations were all made by one person, which will inevitably lead to some degree of inaccuracy and inconsistency as is shown in [16], however they should still give some indication as to the performance of the automatic segmentation methods. In addition to the hand-annotated boundaries, we also developed an automatic technique for identifying regions in the audio signal with the highest level of sinusoidal partial instability. This was done by first performing sinusoidal analysis on each sample using the Spectral Modelling Synthesis method [17], then calculating a detection function from the sum of the frame by frame variations in log frequency (scaled by log amplitude) for each partial. Areas with unstable partials were then defined as the area around peaks in this detection function. The automatically detected segmentation boundaries

were compared to each of the hand-annotated boundaries plus the additional partial instability region measurement.

Table 1 gives the average difference in milliseconds between the automatically detected boundary and reference boundary for our method and the Caetano et al. method. The onset detection results are identical, as we used the same onset detection algorithm for both methods (the peak amplitude difference technique). The Caetano et al. method sustain boundary is slightly closer to the hand-annotated sustain locations on average (by 2.2 ms), but our sustain section is considerably closer to the end of the region with the highest level of partial instability. Our method also fares better in detecting start of release and note offset locations in comparison with the Caetano et al. method. A large part of the error in the Caetano et al. offset detection can be attributed to the fact that they define this boundary based on the energy the signal has at the onset location, and as our onset detector is a real-time method there is a slight latency before it responds, by which stage the signal energy has already started to increase.

TABLE 1. Average deviation from boundaries in reference samples for our proposed method and for the Caetano et al. method.

Boundary	**Proposed Method Avg. Dev. (ms)**	**Caetano et al. Method Avg. Dev. (ms)**
Onset	16.6	16.6
Start of sustain	67.1	64.9
End of unstable partials	40.5	80.0
Start of release	541.6	900.7
Offset	329.5	1597.7

TABLE 2. Model accuracy for our proposed method and for the Caetano et al. method.

Boundary	**Proposed Method Accuracy (%)**	**Caetano et al. Method Accuracy (%)**
Onset	97.2	97.2
Start of sustain	83.3	77.8
End of unstable partials	91.7	69.4
Start of release	30.6	38.9
Offset	58.3	25.0

When evaluating onset detection algorithms, an onset is commonly regarded as being correctly detected if it falls within 50 ms of the reference onset location in order to allow for human error when creating the set of reference values [10, 16]. As the note segmentation boundaries are often a lot more difficult to annotate accurately (the start of the sustain section in particular is not easy to define), we have allowed a more lenient detection window of 100 ms. Table 2 gives the percentage of automatically detected boundaries that fall within 100 ms of the reference values for both segmentation methods. Here, our proposed method is slightly more accurate in detecting the sustain boundary, but our sustain section is again considerably closer to the end of the unstable partial region. The Caetano et al. method is more accurate in detecting the release, with our method performing better at note offset detection.

The results show that both methods perform reasonably well at detecting the start of the sustain region, although our start of the sustain region is significantly closer to the end of the region with high partial instability. Neither method performs particularly well in detecting the release and offset with high accuracy, although on average our proposed model behaves more robustly. Our model is also suitable for real-time use unlike the Caetano et al. method.

5. CONCLUSIONS

This paper proposed a new model for the real-time segmentation of the temporal evolution of musical sounds, using cues from the amplitude envelope, spectral centroid and an onset detection function that is based on measuring errors in sinusoidal partial tracking. We evaluated our method by comparing it with the technique proposed by Caetano et al. and found that in the average case it generally performs better and is more robust. Our method can run in real-time and with considerably lower computation requirements as it does not calculate the computationally costly true amplitude envelope. Neither method was particularly accurate in detecting the release and offset boundaries, so future work could include some research in this area. We will also work on integrating the segmentation system with a real-time performance tool based on sinusoidal synthesis by analysis. The code for our segmentation method, our reference samples and all of the code needed to reproduce our results can be found online at `http://www.johnglover.net`.

ACKNOWLEDGMENTS

The authors are grateful for the financial support that was provided by the Postgraduate Travel Fund at the National University of Ireland, Maynooth.

REFERENCES

1. J. M. Hajda, “The Effect of Dynamic Acoustical Features on Musical Timbre”, in Analysis, Synthesis, and Perception of Musical Sounds, J. W. Beauchamp, ed. (Springer, New York), pp. 250-271, 2007.
2. M. Caetano and X. Rodet, “Automatic Timbral Morphing of Musical Instrument Sounds by High-Level Descriptors”, Proc. 2010 Int. Computer Music Conf. (ICMC 2010), New York, 2010.
3. H. V. Helmholtz, “On the Sensations of Tone as a Physiological Basis for the Theory of Music”, Dover, New York, 1877.
4. G. Peeters, “A Large Set of Audio Features for Sound Description (Similarity and Classification)”, The CUIDADO Project, Project Report, 2004. `http://recherche.ircam.fr/anasyn/peeters/ARTICLES/Peeters_2003_cuidadoaudiofeatures.pdf` (last accessed 18-07-2012).
5. K. Jensen, “Envelope Model of Isolated Musical Sounds”, Proc. 2nd COST G-6 Workshop on Digital Audio Effects (DAFx99), Trondheim, Norway, 1999.
6. J. M. Hajda, “A New Model for Segmenting the Envelope of Musical Signals: The Relative Salience of Steady State Versus Attack, Revisited”, Audio Eng. Soc. Paper No. 4391, 1996.
7. M. Caetano, J. J. Burred, and X. Rodet, “Automatic Segmentation of the Temporal Evolution of Isolated Acoustic Musical Instrument Sounds Using Spectro-Temporal Cues”, Proc. 13th Int. Conf. on Digital Audio Effects (DAFx-10), Graz, Austria, 2010.
8. M. Dolson, “The Phase Vocoder: A Tutorial”, Computer Music Journal, 10 (4), pp. 14-27 (1986).
9. C. Duxbury, M. Davies and M. Sandler, “Improved Time-Scaling of Musical Audio Using Phase Locking at Transients”, 112th Audio Eng. Soc. Convention, Paper No. 5530, Munich, Germany, 2002.
10. J. P. Bello, L. Daudet, S. Abdallah, C. Duxbury, M. Davies and M. Sandler, “A Tutorial on Onset Detection in Music Signals”, IEEE Transactions on Speech and Audio Processing, 13, pp. 1035-1047 (2005).
11. J. Beauchamp, “Synthesis by Spectral Amplitude and ‘Brightness’ Matching of Analyzed Musical Instrument Tones” Journal of the Audio Eng. Soc., 30, pp. 396-406 (1982).
12. M. Caetano and X. Rodet, “Improved Estimation of the Amplitude Envelope of Time-Domain Signals Using True Envelope Cepstral Smoothing”, Proc. 2011 Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2011), pp. 4244-4247, Prague, Czech Republic, 2011.
13. A. Röbel and X. Rodet, “Efficient Spectral Envelope Estimation and its Application to Pitch Shifting and Envelope Preservation”, Proc. 8th Int. Conf. on Digital Audio Effects (DAFx-05), Madrid, Spain, 2005.

14. A. Röbel, “A New Approach to Transient Processing in the Phase Vocoder”, Proc. 6th Int. Conf. on Digital Audio Effects (DAFx-03), London, UK, September, 2003.
15. J. Glover, V. Lazzarini and J. Timoney, “Real-Time Detection of Musical Onsets with Linear Prediction and Sinusoidal Modeling”, EURASIP Journal on Advances in Signal Processing, 2011 (1), pp. 68 (2011).
16. P. Leveau, L. Daudet and G. Richard, “Methodology and Tools for the Evaluation of Automatic Onset Detection Algorithms in Music”, Proc. 5th Int. Conf. on Music Information Retrieval (ISMIR 2004), pp. 72-75, Barcelona, Spain, 2004.
17. X. Serra and J. O. Smith, “Spectral Modeling Synthesis: A Sound Analysis/Synthesis System Based on a Deterministic Plus Stochastic Decomposition”, Computer Music Journal, 14 (4), pp. 12-24 (1990).

Appendix G

Metamorph: real-time high-level sound transformations based on a sinusoids plus noise plus transients model

Original publication:

John Glover, Victor Lazzarini, and Joseph Timoney. Metamorph: Real-time high-level sound transformations based on a sinusoids plus noise plus transients model. In *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12)*, York, UK., September 2012.

METAMORPH: REAL-TIME HIGH-LEVEL SOUND TRANSFORMATIONS BASED ON A SINUSOIDS PLUS NOISE PLUS TRANSIENTS MODEL

John Glover, Victor Lazzarini, Joseph Timoney

The Sound and Digital Music Research Group
National University of Ireland, Maynooth
Ireland
john.c.glover@nuim.ie
victor.lazzarini@nuim.ie
jtimoney@cs.nuim.ie

ABSTRACT

Spectral models provide ways to manipulate musical audio signals that can be both powerful and intuitive, but high-level control is often required in order to provide flexible real-time control over the potentially large parameter set. This paper introduces Metamorph, a new open source library for high-level sound transformation. We describe the real-time sinusoids plus noise plus transients model that is used by Metamorph and explain the opportunities that it provides for sound manipulation.

1. INTRODUCTION

When creating software musical instruments or audio effects, we generally want to have the flexibility to create and manipulate a wide variety of musical sounds while providing an intuitive means of controlling the resulting timbres. Ideally we would like to control these instruments and effects in real-time, as this provides valuable feedback for composers and sound designers as well as enabling the possibility of live performance. As it has been shown that the perception of timbre is largely dependent on the temporal evolution of the sound spectrum [1], it seems natural to use a sound model that is based on the frequency spectrum as a tool to manipulate timbre. Spectral models, which represent audio signals as a sum of sine waves with different frequencies and amplitudes, are therefore an excellent choice of tool for performing musical sound transformations. This frequency domain representation of sound is somewhat similar to the analysis performed by the human hearing system, which enables spectral models to provide ways of transforming audio signals that can be perceptually and musically intuitive. However, they generally suffer from the problem of having too many control parameters to allow for meaningful real-time interaction. One solution to this problem is to provide high-level controls that may change many of the spectral model parameters at once but still allow for precise manipulation of the synthesised sound.

This paper introduces Metamorph, a new open source software library for the high-level transformation of sound using a real-time sinusoids plus noise plus transients model. Sinusoidal models in general are described in Section 2, with our sinusoids plus noise plus transients model described in Section 2.2. The high-level transformations that are available in Metamorph are then described in Section 3, with conclusions in Section 4.

2. SINUSOIDAL MODELS

Sinusoidal modelling is based on Fourier's theorem, which states that any periodic waveform can be modelled as the sum of sinusoids at various amplitudes and harmonic frequencies. For stationary pseudo-periodic sounds, these amplitudes and frequencies evolve slowly with time. They can be used as parameters to control pseudo-sinusoidal oscillators, commonly referred to as partials. To obtain the sinusoidal parameters, the Short-Time Fourier Transform (STFT) is often used to analyse an audio stream or a recorded sound file. This results in a list of bin number, amplitude and phase parameters for each frame of analyzed audio, which can then be used to estimate the spectral peak frequency and amplitude values [2]. Although this approach works well for certain musical sounds, problems can arise due to the fact that the entire audio signal is represented using sinusoids, even if it includes noise-like elements (such as the key noise of a piano or the breath noise in a flute note). If any transformation is applied to the model parameters, these noisy components are

modified along with the harmonic content which often produces audible artifacts. A sinusoidal representation of noise is also unintuitive and does not provide an obvious way to manipulate the sound in a meaningful manner. These issues lead to the development of sinusoids plus noise models of sound.

2.1. Sinusoids Plus Noise Models

With Spectral Modelling Synthesis (SMS) [3] Serra and Smith addressed the problems that can occur when noisy sounds are modelled as a sum of sinusoids by splitting the audio signal into two components: a deterministic (harmonic) component and a stochastic (residual) component. Equation 1 shows how the output signal s is constructed from the sum of these two components, where $A_p(t)$ and $\theta_p(t)$ are the instantaneous amplitude and phase of the p-th component, and $e(t)$ is the stochastic component at time t.

$$s(t) = \sum_{p=1}^{N_p} A_p(t)cos(\theta_p(t)) + e(t) \qquad (1)$$

As it is assumed that the sinusoidal partials will change slowly over time, the instantaneous phase is taken to be the integral of the instantaneous frequency. This is given by Equation 2, where $\omega(t)$ is the frequency in radians per second and p is the partial number.

$$\theta_p(t) = \int_0^t \omega_p(t)dt + \theta_p(0) \qquad (2)$$

The sinusoidal parameters are found by computing the STFT, locating sinusoidal peaks in the magnitude spectrum then matching peaks across consecutive frames to form partials. The harmonic component is then generated from the partials using additive synthesis and subtracted from the original signal leaving a noise-like residual signal. This residual component can then optionally be modelled as filtered white noise. The two signal components can now be manipulated independently and then recombined to create the final synthesised sound.

Other sinusoids plus noise models have been proposed since the release of SMS, such as the model introduced by Fitz and Haken in [4]. It also uses the harmonic partials versus noise distinction, but instead uses bandwidth-enhanced oscillators to create a homogeneous additive sound model. The combination of harmonic and stochastic signals is particularly effective when working with sustained notes that have a relatively constant noise component, but in order to successfully manage shorter noise-like signal regions, sinusoidal models have been further extended to include ways to model transients.

2.2. Sinusoids Plus Noise Plus Transients Models

It has been widely recognised that the initial attack of a note plays a vital role in our perception of timbre [5]. These transient signal regions are very short in duration and can often contain rapid fluctuations in spectral content. Although it is possible to model this sort of signal with sinusoids, doing so can result in the same problems that are encountered when trying to model any noisy signal with sinusoids; it is inherently inefficient and does not offer possibilities for meaningful transformations. Transients are also not well modelled as filtered white noise (as is the case with the SMS stochastic component), as they tend to lose the sharpness in their attack and sound dull [6].

Several systems have been proposed that integrate transients with a sinusoids plus noise model. The method described by Masri in [7] aims to reproduce the sharpness of the original transient during synthesis. First a pre-analysis scan of the audio signal is performed in order to detect transient regions, which are defined as being the areas between a note onset and the point at which the onset detection function (based on an amplitude envelope follower) falls below a fixed threshold or reaches a maximum duration, whichever is shorter. This information is then used during sinusoidal analysis to make sure that the edges of the analysis windows are snapped to the region boundaries. During synthesis, the missing overlap at the region boundaries is reconstructed by extrapolating the waveforms from the centres of both regions slightly then performing a short cross-fade. However, this method can not run in real-time due to the need for a pre-analysis scan of the audio signal.

Levine [8] introduced a sinusoids plus noise model that includes transform-coded transients. Note onsets are located using a combination of an amplitude rising edge detector and by analysing the energy in the stochastic component. Transient regions are then taken to be fixed-duration (66 ms) sections immediately following a note onset. The transients are translated in time during time scaling and pitch transposition, however as the primary application of this work was for use in data compression there is no ability to musically manipulate the transients.

Verma and Meng proposed a system that extends SMS with a model for transients in [6]. They show that transient signals in the time domain can be mapped onto sinusoidal signals in a frequency domain using the discrete cosine transform (DCT). However, it is not suitable for real-time applications as it requires a DCT frame size that makes the transients appear as a small entity, with a frame duration of about 1 second recommended. This is far too much a latency to allow it to be used in a performance context.

Metamorph uses a flexible real-time sinusoids

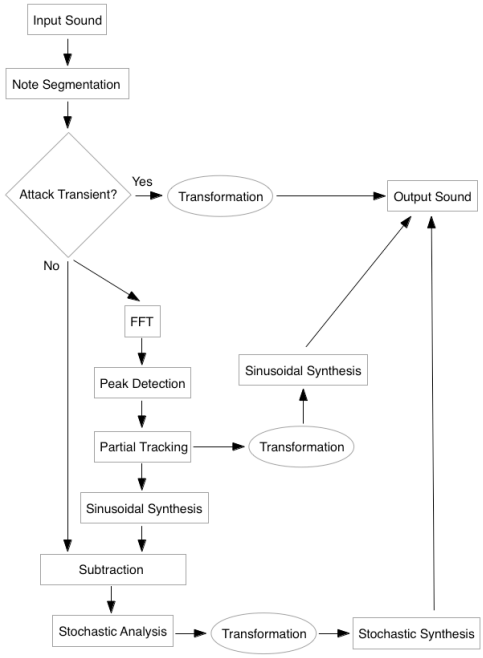

Figure 1: The sinusoids plus noise plus transients model that is used in Metamorph.

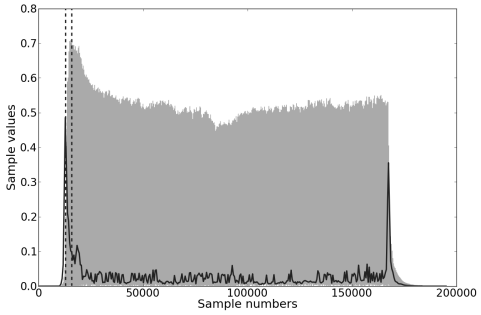

Figure 2: Saxophone sample (grey), onset detection function (solid black line) and detected transient region (between vertical black dashed lines).

plus noise plus transients model which is summarised in Figure 1. Deterministic and stochastic components are identified using Simpl [9], which provides a choice of several different sinusoidal modelling implementations. Onsets are located using the peak amplitude difference method, one of the best-performing methods discussed in [10]. As this effectively measures errors in the partial tracking stage of the sinusoidal modelling process, it can also be used to provide an indication of the stability of the detected sinusoidal partials in the audio signal. Peaks in the onset detection function should occur at regions where the spectral components in the signal are most unstable or are changing unpredictably. We use this information to locate transient regions, which are defined as the region from the onset until the next local minima in the onset detection function. We also mark the end of the transient region if the root mean square amplitude envelope reaches a local maxima or if the transient reaches a maximum duration of 200 ms (whichever is shorter). An example consisting of an audio signal (a saxophone sample), our onset detection function and the corresponding transient region is shown in Figure 2. More information on this process is given in [11]. When no transformations are applied to the signal, transients in Metamorph are simply blocks of unmodified raw sample values, with no sinusoidal analysis or synthesis performed during transient regions. During synthesis, we extend the output of the transient region slightly (currently for a single 512 sample frame) in order to perform a short cross-fade between the unmodified sample values in the transient region and the synthesised signal in the note region that follows. The sample values in transient regions may be altered if a transformation is applied however, as we discuss in Section 3.

3. METAMORPH: HIGH-LEVEL SOUND TRANSFORMATIONS

Metamorph is a new open source library for performing high-level sound transformations based on a sinusoids plus noise plus transients model. It is written in C++, can be built as both a Python extension module and a Csound opcode, and currently runs on Mac OS X and Linux. It is designed to work primarily on monophonic, quasi-harmonic sound sources and can be used in a non-real-time context to process pre-recorded sound files or can operate in a real-time (streaming) mode. Here we define a real-time analysis/synthesis system to be one that can operate with low enough latency and computational requirements so that it is usable in a live musical performance context. While there is no absolute rule that specifies how much latency is acceptable in a live performance context, Metamorph's default latency of 512 samples (or about 11.6 ms) should meet these requirements in many cases. The computational requirements for Metamorph vary depending on the supplied sinusoidal modelling parameters, the type of transformation being applied and on the nature of the sound source itself. However as an example, the noisiness and transience transformation (described in Section 3.2) streaming Csound opcode with de-

fault parameters requires approximately 20% of one CPU core on a 2.4 GHz Mac Intel Core 2 Duo processor.

Metamorph is available under the terms of the GNU General Public License (GPL). For more information and to get the software, go to `http://www.johnglover.net`. In this section, we describe the sound transformations that are currently available in Metamorph.

3.1. Harmonic Distortion

As described in [12], the harmonic distortion of a sound is a measure of the degree of the deviation of the measured partials from ideal harmonic partials. The Metamorph harmonic distortion transformation allows the user to alter the deviation of each synthesised partial in a sound from the ideal harmonic spectrum according to Equation 3, where i is the partial number, f is the analysed partial frequency, F_0 is the estimated fundamental frequency, α is the input parameter (between 0 and 1) and F is the output frequency of the synthesised partial.

$$F_i = (\alpha \times f_i) + ((1 - \alpha) \times (F_0 \times i)) \quad (3)$$

3.2. Noisiness and Transience

The noisiness [12] of a synthesised frame is calculated by taking the ratio of the amplitude of the residual component to the total signal amplitude. Metamorph allows the user to easily adjust this balance by altering the amplitudes of the deterministic and stochastic components independently. It also enables the independent manipulation of the amplitude of transient regions. We call this effect changing the signal *transience*.

3.3. Spectral Envelope Manipulation

A spectral envelope is a curve in the spectrum of an audio signal that approximates the distribution of the signal's energy over frequency. Ideally this curve should pass through all of the prominent peaks of the frame and be relatively smooth, preserving the basic formant structure of the frame without oscillating too much or containing discontinuities. Spectral envelopes in Metamorph are calculated using the discrete cepstrum envelope (DCE) method [13] which provides a smooth interpolation between the detected sinusoidal peaks. However, further comparison with the true envelope method [14] is desirable in future, as it seems to produce spectral envelopes that are as good (if not better) than the DCE and it can now be computed efficiently [15]. This was not an immediate priority as the main problem that the authors in [15] had with the DCE was that it required a potentially computationally expensive fundamental frequency analysis or other means of identifying spectral peaks, but this is already a requirement for other parts of the sinusoidal modelling process so there is no extra cost associated with this step in Metamorph.
The spectral envelope transformation in Metamorph allows the user to alter the amplitudes of the synthesised sinusoidal partials to match the corresponding amplitude values in a different spectral envelope. This can be a fixed envelope, or the user can specify a different sound source to use as the target envelope. The user may also alter the synthesised partial amplitudes based on a linear interpolation between the original spectral envelope and the target envelope. Although similar techniques can be performed using the Phase Vocoder [15], spectral envelope manipulation using sinusoidal models enables the preservation (or independent manipulation) of the stochastic signal component, offering greater possibilities for sound transformation. In Metamorph this is taken a step further, enabling the alteration of a spectral envelope while preserving the initial note attack.

3.4. Transposition

Sounds can be transposed in Metamorph in two different ways. Both techniques involve initially multiplying the frequency values of all synthesised partials by the same factor. The second process additionally adjusts all of the partial amplitude values so that they match those of the original spectral envelope. The latter approach preserves the original formant structure, which results in a more natural sounding transposition for certain types of sounds. The transient region is left unmodified for both types of transposition.

3.5. Time Scaling

Time scaling is the only Metamorph transformation that is not available in real-time mode. The time scaling algorithm works by keeping the analysis and synthesis frame rates identical, but instead of passing each analysis frame directly to the synthesis module, frames may be repeated (or skipped) depending on the required time scale factor. This approach does not result in any synthesis artifacts or discontinuities as the synthesis module interpolates smoothly between input frames, and has been shown to produce high-quality time scaling [16]. Frames from transient regions are treated differently however; they are always passed to the synthesis module in the original order, with the sample values passed unmodified to the output so that the transient is maintained. This means that the time scale factor has to be adjusted slightly during non-transient regions in order

to make sure that the final output signal is of the required length.

3.6. Transient Processing

Most transformations in Metamorph aim to preserve the original transient region, but it is also possible to explicitly alter the output transient. The most basic effect is to either filter the transient using either low- or high-pass filters, which although relatively simple can have quite a large impact on the resulting sound. The transient can also be removed altogether. Another interesting effect is transient substitution, where the transient regions in the audio signal can be replaced by a different set of audio samples (which may or may not themselves be transients). This allows for the creation of various hybrid instruments, for example combining the attack of a drum sound with a sustained woodwind instrument tone.

4. CONCLUSIONS

This paper introduced Metamorph, a software library which provides a new environment for performing high-level sound manipulation. It is based on a real-time sinusoids plus noise plus transients model, which enables it to perform a collection of flexible and powerful sound transformations. Metamorph is free software, can be used as a C++ library, Python extension module or set of Csound opcodes and is available under the terms of the GNU General Public License. To download it or for more information go to `http://www.johnglover.net`.

5. REFERENCES

[1] John M. Hajda, *Analysis, Synthesis, and Perception of Musical Sounds*, chapter The Effect of Dynamic Acoustical Features on Musical timbre, pp. 250–271, Springer, 2007.

[2] Robert McAulay and Thomas Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-34, no. 4, August 1986.

[3] Xavier Serra and Julius O. Smith, "Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decompostion," *Computer Music Journal*, vol. 14, no. 4, pp. 12–24, Winter 1990.

[4] Kelly Fitz, *The Reassigned Bandwidth-Enhanced Method of Additive Synthesis*, Ph.D. thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, USA, 1999.

[5] John M. Grey, *An Exploration of Musical Timbre*, Ph.D. thesis, Stanford University, USA, 1975.

[6] Tony S. Verma and Teresa H. Y. Meng, "Extending spectral modeling synthesis with transient modeling synthesis," *Computer Music Journal*, vol. 24, no. 2, pp. 47–59, Summer 2000.

[7] Paul Masri, *Computer Modeling of Sound for Transformation and Synthesis of Musical Signals*, Ph.D. thesis, University of Bristol, United Kingdom, 1996.

[8] Scott Levine, *Audio Representations for Data Compression and Compressed Domain Processing*, Ph.D. thesis, Stanford University, 1998.

[9] John Glover, Victor Lazzarini, and Joseph Timoney, "Simpl: A Python library for sinusoidal modelling," in *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, Como, Italy, September 2009.

[10] John Glover, Victor Lazzarini, and Joseph Timoney, "Real-time detection of musical onsets with linear prediction and sinusoidal modeling," *EURASIP Journal on Advances in Signal Processing*, vol. 2011, no. 1, pp. 68, 2011.

[11] John Glover, Victor Lazzarini, and Joseph Timoney, "Real-time segmentation of the temporal evolution of musical sounds," in *The Acoustics 2012 Hong Kong Conference*, Hong Kong, China, May 2012.

[12] Xavier Serra and Jordi Bonada, "Sound transformations based on the sms high level attributes," in *Proceedings of the International Conference on Digital Audio Effects*, Barcelona, Spain, 1998.

[13] Thierry Galas and Xavier Rodet, "An improved cepstral method for deconvolution of source filter systems with discrete spectra: Application to musical sound signals," in *Proceedings of the International Computer Music Conference (ICMC'90)*, Glasgow, Scotland, 1990, pp. 82–84.

[14] S. Imai and Y. Abe, "Spectral envelope extraction by improved cepstral method," *Electron. and Commun. in Japan*, vol. 62-A, no. 4, pp. 10–17, 1979.

[15] Axel Röbel and Xavier Rodet, "Efficient spectral envelope estimation and its application

to pitch shifting and envelope preservation," in *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx-05)*, Madrid, Spain, September 2005.

[16] J Bonada, "Automatic technique in frequency domain for near-lossless time-scale modification of audio," in *Proceedings of the International Computer Music Conference (ICMC'00)*, Berlin, Germany, 2000, pp. 396–399.

Bibliography

[1] AHMED, N., NATARAJAN, T., AND RAO, K. Discrete cosine transfom. *IEEE Transactions on Computers C-23*, 1 (January 1974), 90–93.

[2] ALLEN, J. B. Short term spectral analysis, synthesis and modification by discrete Fourier transform. *IEEE Transactions on Acoustics, Speech and Signal Processing ASSP-25* (June 1977), 235–238.

[3] ALLEN, J. B., AND RABINER, L. R. A unified approach to short-time Fourier analysis and synthesis. *Proceedings of the IEEE 65*, 11 (November 1977).

[4] AMATRIAIN, X., BONADA, J., LOSCOS, A., AND SERRA, X. *DAFx - Digital Audio Effects*. John Wiley and Sons, 2002, ch. Spectral Processing, pp. 373–438.

[5] ANALOG DEVICES. ADSP-TS201S data sheet. http://www.analog.com/static/imported-files/data_sheets/ADSP_TS201S.pdf (last accessed 28.03.2013).

[6] AUGER, F., AND FLANDRIN, P. Generalization of the reassignment method to all bilinear time-frequency and time-scale representations. In *Proceedings*

of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-94) (April 1994), vol. 4, pp. 317–320.

[7] BEAUCHAMP, J. Synthesis by spectral amplitude and "brightness" matching of analyzed musical instrument tones. *Journal of the Audio Engineering Society 30*, 6 (1982), 396–406.

[8] BEAUCHAMP, J. Unix workstation software for analysis, graphics, modification, and synthesis of musical sounds. In *Audio Engineering Society Convention 94* (1993).

[9] BEAZLEY, D. M. Automated scientific software scripting with SWIG. *Future Generation Computer Systems - Tools for Program Development and Analysis 19*, 5 (July 2003), 599–609.

[10] BEHNEL, S., BRADSHAW, R., CITRO, C., DALCIN, L., SELJEBOTN, D., AND SMITH, K. Cython: The best of both worlds. *Computing in Science Engineering 13*, 2 (March-April 2011), 31–39.

[11] BELLO, J. P., DAUDET, L., ABDALLAH, S., DUXBURY, C., DAVIES, M., AND SANDLER, M. A Tutorial on Onset Detection in Music Signals. *IEEE Transactions on Speech and Audio Processing 13*, 5 (Sept. 2005), 1035–1047.

[12] BELLO, J. P., DUXBURY, C., DAVIES, M., AND SANDLER, M. On the use of phase and energy for musical onset detection in the complex domain. *IEEE Signal Processing Letters 11*, 6 (June 2004), 553–556.

[13] BERKELEY DESIGN TECHNOLOGY, INC. BDTI DSP kernel benchmarks (BDTIMark200) certified results. http://www.bdti.com/Resources/BenchmarkResults/BDTIMark2000 (last accessed 28.03.2013).

[14] BOGAARDS, N., RÖBEL, A., AND RODET, X. Sound analysis and processing with AudioSculpt 2. In *Proceedings of the International Computer Music Conference (ICMC'04)* (Miami, USA., 2004).

[15] BONADA, J. Automatic technique in frequency domain for near-lossless time-scale modification of audio. In *Proceedings of the International Computer Music Conference (ICMC'00)* (Berlin, Germany, 2000), pp. 396–399.

[16] BOULANGER, R., AND LAZZARINI, V., Eds. *The Audio Programming Book*. The MIT Press, Cambridge, MA, USA, 2010, ch. 7.

[17] BOULANGER, R., AND LAZZARINI, V., Eds. *The Audio Programming Book*. The MIT Press, Cambridge, MA, USA, 2010, ch. 8.

[18] BROSSIER, P., BELLO, J. P., AND PLUMBLEY, M. Real-time temporal segmentation of note objects in music signals. In *Proceedings of the International Computer Music Conference (ICMC'04)* (2004), pp. 458–461.

[19] CAETANO, M., BURRED, J. J., AND RODET, X. Automatic segmentation of the temporal evolution of isolated acoustic musical instrument sounds using spectro-temporal cues. In *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)* (Graz, Austria, September 2010).

[20] CAETANO, M., AND RODET, X. Automatic timbral morphing of musical instrument sounds by high-level descriptors. In *Proceedings of the International Computer Music Conference (ICMC'10)* (New York, June 2010).

[21] CAETANO, M., AND RODET, X. Improved estimation of the amplitude envelope of time-domain signals using true envelope cepstral smoothing. In *Proceedings of the 2011 International Conference on Acoustics, Speech and Signal Processing (ICASSP 2011)* (Prague, Czech Republic, May 2011), pp. 4244–4247.

[22] CAPPÉ, O., AND MOULINES, E. Regularization techniques for discrete cepstrum estimation. *IEEE Signal Processing Letters 3*, 4 (April 1996), 100–102.

[23] CHOWNING, J. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society 21* (1973), 526–534.

[24] COOLEY, J. W., AND TUKEY, J. W. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19* (1965), 297–301.

[25] CROCHIERE, R. A weighted overlap-add method of short-time Fourier analysis/synthesis. *IEEE Transactions on Acoustics, Speech and Signal Processing 28*, 1 (February 1980), 99–102.

[26] DEPALLE, P., GARCIA, G., AND RODET, X. Tracking of partials for additive sound synthesis using hidden Markov models. In *Proceedings of the*

IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-93) (Minneapolis, USA, 1993), pp. 225–228.

[27] DEPALLE, P., AND POIRROT, G. SVP: A modular system for analysis, processing and synthesis of sound signals. In *Proceedings of the International Computer Music Conference (ICMC'91)* (Montreal, Canada, 1991).

[28] DIXON, S. Onset detection revisited. In *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)* (Montreal, Canada, September 2006).

[29] DOLSON, M. The phase vocoder: A tutorial. *Computer Music Journal 10*, 4 (Winter 1986), 14–27.

[30] DUXBURY, C., DAVIES, M., AND SANDLER, M. Improved time-scaling of musical audio using phase locking at transients. In *112th Audio Engineering Society Convention* (Munich, Germany, May 2002).

[31] DUXBURY, C., SANDLER, M., AND DAVIES, M. A hybrid approach to musical note onset detection. In *Proceedings of the 5th International Conference on Digital Audio Effects (DAFx-02)* (Hamburg, Germany, September 2002), pp. 33–38.

[32] EAKIN, R., AND SERRA, X. libsms library for spectral modeling synthesis. http://www.mtg.upf.edu/static/libsms (last accessed 28.03.2013), 2013.

[33] EATON, J. W. *GNU Octave Manual*. Network Theory Limited, Bristol, UK, 2002.

[34] FITZ, K. *The Reassigned Bandwidth-Enhanced Method of Additive Synthesis*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, USA, 1999.

[35] FITZ, K., AND HAKEN, L. Sinusoidal modeling and manipulation using Lemur. *Computer Music Journal 20*, 4 (Winter 1996), 44–59.

[36] FLANAGAN, J. L., AND GOLDEN, R. M. Phase vocoder. *Bell System Technical Journal 45* (1966), 1493–1509.

[37] FRIGO, M., AND JOHNSON, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231.

[38] FRIGO, M., AND JOHNSON, S. G. FFTW3 library. http://www.fftw.org (last accessed 28.03.2013), 2013.

[39] GALAS, T., AND RODET, X. An improved cepstral method for deconvolution of source filter systems with discrete spectra: Application to musical sound signals. In *Proceedings of the International Computer Music Conference (ICMC'90)* (Glasgow, Scotland, 1990), pp. 82–84.

[40] GALAS, T., AND RODET, X. Generalized discrete cepstral analysis for deconvolution of source-filter system with discrete spectra. In *1991 IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics* (October 1991), pp. 71–72.

[41] GOTO, M., HASHIGUCHI, H., NISHIMURA, T., AND OKA, R. RWC music database: Popular, classical, and jazz music databases. In *Proceedings of the*

3rd International Conference on Music Information Retrieval (ISMIR 2002) (October 2002), pp. 287–288.

[42] GREY, J. M. *An Exploration of Musical Timbre*. PhD thesis, Stanford University, USA, 1975.

[43] HAINSWORTH, S., AND MACLEOD, M. On sinusoidal parameter estimation. In *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03)* (London, UK, September 2003).

[44] HAJDA, J. M. A new model for segmenting the envelope of musical signals: The relative salience of steady state versus attack, revisited. In *Audio Engineering Society Convention 101* (November 1996).

[45] HAJDA, J. M. *Analysis, Synthesis, and Perception of Musical Sounds*. Springer, New York, 2007, ch. The Effect of Dynamic Acoustical Features on Musical timbre, pp. 250–271.

[46] HARRIS, F. J. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE 66*, 1 (1978), 51–83.

[47] HDF GROUP. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5 (last accessed 28.03.2013), 2013.

[48] HELMHOLTZ, H. V. *On the sensations of tone as a physiological basis for the theory of music*. Dover, New York, 1877.

[49] HUNTER, J. D. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering 9*, 3 (May-Jun 2007), 90–95.

[50] IMAI, S., AND ABE, Y. Spectral envelope extraction by improved cepstral method. *Electron. and Commun. in Japan 62-A*, 4 (1979), 10–17.

[51] INTEL CORPORATION. Intel microprocessor export compliance metrics. http://www.intel.com/support/processors/sb/CS-032819.htm (last accessed 28.03.2013), 2013.

[52] IRCAM. SuperVP for Max. http://forumnet.ircam.fr/product/supervp-max (last accessed 28.03.2013), 2013.

[53] JAFFE, D. A. Spectrum analysis tutorial, part 1: The discrete Fourier transform. *Computer Music Journal 11*, 2 (Summer 1987), 9–24.

[54] JAFFE, D. A. Spectrum analysis tutorial, part 2: Properties and applications of the discrete Fourier transform. *Computer Music Journal 11*, 3 (Autumn 1987), 17–35.

[55] JENSEN, K. Envelope model of isolated musical sounds. In *Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects (DAFx99)* (Trondheim, Norway, December 1999).

[56] JONES, E., OLIPHANT, T., PETERSON, P., ET AL. SciPy: Open source scientific tools for Python. http://www.scipy.org (last accessed 28.03.2013), 2013.

[57] KAUPPINEN, I. Methods for detecting impulsive noise in speech and audio signals. In *Proceedings of the 14th International Conference on Digital Signal Processing (DSP 2002)* (2002), vol. 2, pp. 967–970.

[58] KEILER, F., ARFIB, D., AND ZOLZER, U. Efficient linear prediction for digital audio effects. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)* (Verona, Italy, December 200).

[59] KLINGBEIL, M. Software for spectral analysis, editing, and synthesis. In *Proceedings of the International Computer Music Conference (ICMC'05)* (Barcelona, Spain, 2005).

[60] KODERA, K., GENDRIN, R., AND VILLEDARY, C. Analysis of time-varying signals with small BT values. *IEEE Transactions on Acoustics, Speech and Signal Processing 26*, 1 (1978), 64–76.

[61] LAGRANGE, M., AND MARCHAND, S. Accessing the quality of the extraction and tracking of sinusoidal components: towards an evaluation methodology. In *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)* (Montreal, Canada, September 2006).

[62] LAGRANGE, M., MARCHAND, S., RASPAUD, M., AND RAULT, J.-B. Enhanced partial tracking using linear prediction. In *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03)* (London, UK, September 2003).

[63] LAGRANGE, M., MARCHAND, S., AND RAULT, J.-B. Partial tracking based on future trajectories exploration. In *Audio Engineering Society Convention 116* (May 2004).

[64] LAROCHE, J., AND DOLSON, M. Phase-vocoder: about this phasiness business. In *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics* (October 1997), pp. 19–22.

[65] LAROCHE, J., AND DOLSON, M. Improved phase vocoder time-scale modification of audio. *IEEE Transactions on Acoustics, Speech and Signal Processing 7*, 3 (May 1999), 323–332.

[66] LAZZARINI, V. Sound processing with The SndObj Library: An overview. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)* (University of Limerick, Ireland, December 2001).

[67] LAZZARINI, V., TIMONEY, J., AND LYSAGHT, T. Alternative analysis-synthesis approaches for timescale, frequency and other transformations of musical signals. In *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx-05)* (Madrid, Spain, 2005), pp. 18–23.

[68] LAZZARINI, V., TIMONEY, J., AND LYSAGHT, T. Time-stretching using the instantaneous frequency distribution and partial tracking. In *Proceedings of the International Computer Music Conference (ICMC'05)* (Barcelona, Spain, 2005).

[69] LAZZARINI, V., TIMONEY, J., AND LYSAGHT, T. Streaming frequency-domain DAFX in Csound5. In *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)* (Montreal, Canada, September 2006), pp. 275–278.

[70] LAZZARINI, V., TIMONEY, J., AND LYSAGHT, T. The generation of natural-synthetic spectra by means of adaptive frequency modulation. *Computer Music Journal 32*, 2 (2008), 12–22.

[71] LEE, W.-C., AND KUO, C.-C. J. Musical onset detection based on adaptive linear prediction. In *Proceedings of the 2006 IEEE Conference on*

Multimedia and Expo, ICME 2006 (Ontario, Canada, July 2006), pp. 957–960.

[72] LEVEAU, P., DAUDET, L., AND RICHARD, G. Methodology and tools for the evaluation of automatic onset detection algorithms in music. In *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR 2004)* (Barcelona, Spain, October 2004), pp. 72–75.

[73] LEVINE, S. *Audio Representations for Data Compression and Compressed Domain Processing*. PhD thesis, Stanford University, 1998.

[74] MAHER, R., AND BEAUCHAMP, J. Fundamental frequency estimation of musical signals using a two-way mismatch procedure. *The Journal of the Acoustical Society of America 95*, 4 (April 1994), 2254–2263.

[75] MAKHOUL, J. Linear prediction: A tutorial review. *Proceedings of the IEEE 63*, 4 (1975), 561–580.

[76] MARCHAND, S., AND LAGRANGE, M. On the equivalence of phase-based methods for the estimation of instantaneous frequency. In *Proceedings of the 14th European Conference on Signal Processing (EUSIPCO2006)* (Florence, Italy, September 2006), EURASIP.

[77] MASRI, P. *Computer Modeling of Sound for Transformation and Synthesis of Musical Signals*. PhD thesis, University of Bristol, United Kingdom, 1996.

[78] MASRI, P., AND BATEMAN, A. Improved modelling of attack transients in music analysis-resynthesis. In *Proceedings of the International Computer Music Conference (ICMC'96)* (Hong Kong, China, 1996).

[79] MATHWORKS, T. *MATLAB Release R2010b*. The MathWorks, Natick, Massachusetts, 2010.

[80] MCAULAY, R., AND QUATIERI, T. Speech analysis/synthesis based on a sinusoidal representation. *IEEE Transactions on Acoustics, Speech and Signal Processing 34*, 4 (August 1986).

[81] MCCARTNEY, J. SuperCollider: A new real time synthesis language. In *Proceedings of the International Computer Music Conference (ICMC'96)* (1996), International Computer Music Association.

[82] MIREX. MIREX 2009 audio onset detection results. http://www.music-ir.org/mirex/wiki/2009:Audio_Onset_Detection_Results (last accessed 28.03.2013), 2009.

[83] MIREX. MIREX audio onset detection. http://www.music-ir.org/mirex/wiki/Audio_Onset_Detection (last accessed 28.03.2013), 2013.

[84] MOORE, F. R. *Elements of Computer Music*. Prentice Hall, New Jersey, USA, 1990.

[85] MOORER, J. A. The use of the phase vocoder in computer music applications. In *Audio Engineering Society Convention 55* (October 1976).

[86] NOKIA. Qt - a cross-platform application and UI framework. http://qt.nokia.com (last accessed 28.03.2013), 2013.

[87] NUNES, L. O., BISCAINHO, L. W., AND ESQUEF, P. A. A. A database of partial tracks for evaluation of sinusoidal models. In *Proceedings of the*

13th International Conference on Digitial Audio Effects (DAFx-10) (Graz, Austria, September 2010).

[88] OLIPHANT, T. *Guide To NumPy*. Trelgol Publishing, USA, 2006.

[89] OLIPHANT, T. Python for scientific computing. *Computing in Science Engineering 9*, 3 (May-Jun 2007), 10–20.

[90] ONE LAPTOP PER CHILD (OLPC). The OLPC free sample library. http://wiki.laptop.org/go/Free_sound_samples (last accessed 28.03.2013), 2013.

[91] OPPENHEIM, A. V., AND SCHAFER, R. W. *Digital Signal Processing*. Prentice Hall, New Jersey, USA, 1975.

[92] OPPENHEIM, A. V., SCHAFER, R. W., AND BUCK, J. R. *Discrete-time signal processing*, 2nd ed. Prentice Hall, 1999, ch. 8.

[93] ORIO, N., LEMOUTON, S., AND SCHWARZ, D. Score following: State of the art and new developments. In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)* (Montreal, Canada, 2003).

[94] PAMPIN, J. ATS: A lisp environment for spectral modeling. In *Proceedings of the International Computer Music Conference (ICMC'99)* (Beijing, China, 1999).

[95] PEETERS, G. A large set of audio features for sound description (similarity and classification) in the CUIDADO project. Project Report, 2004.

[96] PORTNOFF, M. Time-frequency representation of digital signals and systems based on short-time Fourier analysis. *IEEE Transactions on Acoustics, Speech and Signal Processing 28*, 1 (February 1980), 55–69.

[97] PUCKETTE, M. S. Pure Data: another integrated computer music environment. In *Proceedings of the International Computer Music Conference (ICMC'96)* (San Francisco, USA, 1996), International Computer Music Association, pp. 224–227.

[98] PURNHAGEN, H. Parameter estimation and tracking for time-varying sinusoids. In *Proceedings of the 1st IEEE Benelux Workshop on model based processing and coding of audio (MPCA)* (Leuven, Belgium, 2002).

[99] ROADS, C. Automated granular synthesis of sound. *Computer Music Journal 2*, 2 (1978), 61–62.

[100] RÖBEL, A. A new approach to transient processing in the phase vocoder. In *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03)* (London, UK, September 2003).

[101] RÖBEL, A., AND RODET, X. Efficient spectral envelope estimation and its application to pitch shifting and envelope preservation. In *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx-05)* (Madrid, Spain, September 2005).

[102] RÖBEL, A., AND RODET, X. Real time signal transposition with envelope preservation in the phase vocoder. In *Proceedings of the International Computer Music Conference (ICMC'05)* (Barcelona, Spain, 2005), pp. 672–675.

[103] ROEDERER, J. G. *The Physics and Psychophysics of Music: An Introduction*, 4 ed. Springer, New York, 2009.

[104] ROSSUM, G. V., AND DRAKE, F. L. *Python Language Reference Manual*. Network Theory Limited, Bristol, UK, 2006.

[105] SCHNELL, N., SCHWARZ, D., AND MULLER, R. X-micks - interactive content based real-time audio processing. In *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)* (Montreal, Canada, 2006).

[106] SCHOTTSTAEDT, W. Machine tongues XVII: Clm: Music V meets common lisp. *Computer Music Journal 18*, 2 (1994), 30–37.

[107] SCHWARZ, D., AND RODET, X. Spectral envelope estimation and representation for sound analysis-synthesis. In *Proceedings of the International Computer Music Conference (ICMC'99)* (Beijing, China, 1999), pp. 351–354.

[108] SERRA, X. *A System for Sound Analysis/Transformation/Synthesis based on a Deterministic plus Stochastic Decomposition*. PhD thesis, Stanford University, 1989.

[109] SERRA, X., AND BONADA, J. Sound transformations based on the SMS high level attributes. In *Proceedings of the International Conference on Digital Audio Effects* (Barcelona, Spain, 1998).

[110] SERRA, X., AND SMITH, J. O. Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal 14*, 4 (Winter 1990), 12–24.

[111] SMITH, J. O. Viewpoints on the history of digital synthesis. In *Proceedings of the International Computer Music Conference (ICMC'91)* (Montreal, Canada, October 1991), pp. 1–10.

[112] SMITH, J. O., AND SERRA, X. PARSHL: An analysis/synthesis program for non-harmonic sounds based on a sinusoidal representation. In *Proceedings of the International Computer Music Conference (ICMC'87)* (San Francisco, USA, 1987).

[113] STARK, A., MATTHEW, D., AND PLUMBLEY, M. Real-time beat-synchronous analysis of musical audio. In *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)* (Como, Italy, September 2009).

[114] STEIGLITZ, K. *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music*. Prentice Hall, New Jersey, USA, 1996, ch. 7.

[115] STOWELL, D., AND PLUMBLEY, M. Adaptive whitening for improved real-time audio onset detection. In *Proceedings of the International Computer Music Conference (ICMC'07)* (Copenhagen, Denmark, August 2007), pp. 312–319.

[116] TOLONEN, T., VÄLIMÄKI, V., AND KARJALAINEN, M. Evaluation of modern sound synthesis methods. Tech. rep., Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, Espoo, Finland, March 1998.

[117] TREES, H. L. V. *Detection, Estimation, and Modulation Theory, Part I*. John Wiley and Sons, New York, 1968, ch. 3.

[118] TRUAX, B. Real-time granular synthesis with a digital signal processor. *Computer Music Journal 12*, 2 (1988), 14–26.

[119] VERCOE, B., ET AL. The Csound Reference Manual. http://www.csounds.com (last accessed 28.03.2013), 2013.

[120] VERFAILLE, V., ZOLZER, U., AND ARFIB, D. Adaptive digital audio effects (A-DAFx): A new class of sound transformations. *IEEE Transactions on Audio, Speech and Language Procesing 14*, 5 (September 2006), 1817–1831.

[121] VERMA, T. S., AND MENG, T. H. Y. Extending spectral modeling synthesis with transient modeling synthesis. *Computer Music Journal 24*, 2 (Summer 2000), 47–59.

[122] VOS, J., AND RASCH, R. The perceptual onset of musical tones. *Perception and Psychophysics 29*, 4 (1981), 323–335.

[123] WILLIAMS, T., KELLEY, C., ET AL. Gnuplot. http://www.gnuplot.info (last accessed 28.03.2013), 2013.

[124] WISHART., T. *Audible Design*. Orpheus the Pantomime, York, UK., 1996.

[125] WRIGHT, M., CHAUDHARY, A., FREED, A., KHOURY, S., AND WESSEL, D. Audio applications of the sound description interchange format standard. In *Audio Engineering Society Convention 107* (1999), Audio Engineering Society.