Hokua – a Wavelet Method for Audio Fingerprinting

Steven Scott Lutz

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mathematics

Brigham Young University

December 2009

Abstract

Hokua – a Wavelet Method for Audio Fingerprinting

Steven Scott Lutz

Department of Mathematics

Master of Science

In recent years, multimedia identification has become important as the volume of digital media has dramatically increased. With music files, one method of identification is audio fingerprinting. The underlying method for most algorithms is the Fourier transform. However, due to a lack of temporal resolution, these algorithms rely on the short-time Fourier transform. We propose an audio fingerprinting algorithm that uses a wavelet transform, which has good temporal resolution.

In this thesis, we examine the basics of certain topics that are needed in understanding audio fingerprinting techniques. We also look at a brief history of work done in this field. We introduce a new algorithm, called the Hokua algorithm. We developed Hokua to take advantage of certain properties of the wavelet transform. The algorithm uses coefficient peaks of wavelet transforms to identify a sample query. The various algorithms are compared.

Acknowledgments

I would like to thank Dr. Jeffrey Humpherys for the support he lent me during this endeavor. His confidence in my abilities often exceeded my own. I am not sure what he saw in me, but I am glad that he pushed me to keep at it. His advice has been unbelievably helpful. I am truly appreciative of him.

Thanks is also owed to the BYU Math Department for the financial support throughout my graduate studies. Also I am indebted to the support of the National Science Foundation (Grant No. 0639328).

Special thanks to the students of IMPACT who have listened to my presentation on several instances and always given me good feedback. They have been true friends. Likewise the wonderful mathematicians of my office, 300 TMCB. They have kept me sane, and let me chat their ears off while I waited for some MATLAB script to run.

I thank my parents for their examples in so many ways. And not getting too upset when I didn't answer their questions as to how my thesis was coming.

Finally, I wish to thank my wife. Her support has come in countless ways–her patience, her encouragement, her service and her trust have been so helpful. I look forward to her support throughout the years to come, and hope I can be a pillar to her.

N̄an e im ad Iroij ij mūri aolep.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Imagine the following scene: You start up your media player, only to find the confusion that abounds from seven songs all labeled “track 05.” You’re pretty sure one of them is the song you want to listen to, so you start clicking on each one, listening to a few seconds, and then moving on. All the while the song you really want is “track 15.” After you give up the search in vain, you settle with some Shania Twain song, since all of your sister’s music is properly labeled.

Audio fingerprinting is a process that takes an unknown audio signal and returns a fingerprint, which can then be matched against a database of known songs to provide information about the unknown audio. In the above scenario the information we desired was contained in the audio’s metadata, which is information associated with the audio clip and includes the song’s title, artist, and album. However we may be interested in answering many different kinds of questions about a given collection of audio music. Audio fingerprinting has a wide variety of applications:

(i) Automatically populate metadata

(ii) Identify a currently playing song

(iii) Monitor radio broadcasts

(iv) Peer-2-peer copyright issues

(v) Manage sound effect libraries

(vi) Speech recognition

The first two items deal with obtaining metadata. In each, the audio fingerprinting system would query a database for the required information. With automatically populating metadata the system would pull missing data from a metadata repository to complete a music library. This is the case with MusicBrainz , a server that will “automatically identify and clean up the metadata tags in your digital music collections.”

Another area where we would like to access metadata is in the identification of a song that we hear, for instance one playing on the radio. Here the song needs to be identified, and then the metadata can be used to not only inform the listener, but also provide links to purchase the song or find similar songs. This is currently done with the application Shazam.

The next two uses ensure that content owners are appropriately compensated for their music. Monitoring of radio broadcasts is important for royalty collection, but also to determine whether commercials get fair air time. Prior to audio fingerprinting, radio stations were monitored by individuals. Now companies such as Nielson BDS or Mediaguide offer automatic reporting of air time for both audio tracks and advertisements.

Peer-2-peer (P2P) networks provide an interesting challenge. Initial attempts to restrict the unauthorized sharing of copyrighted material have been easily circumvented. One method, known as cryptographic hashing, involves running the total bit representation of the file through an algorithm that combines the bits in such a way and returns a string of letters and numbers. This string can be compared against a list to find the matching file. However the slightest change in the bit representation creates an entirely different hash. Thus by removing a portion of the song, those illegally sharing music could bypass hashing methods. Another method used the song's metadata to restrict sharing. However metadata is easily changed, allowing a mislabeled song to be shared. Audio fingerprinting relies on the content of the music itself, and is thus more robust to these methods. Companies such as Audible Magic and SNOCAP provide P2P monitoring.

The penultimate item is an organization tool. In a sound effect library, we may not be as interested in what the song is called, or who recorded it, but what a defining characteristic is of the clip. For instance, we may want to group together all the tracks that contain people laughing, cricket sounds, applause, etc. We attempt to group sound effect clips by perceptual similarity.

While the final item has been researched much more fully than audio fingerprinting, some of the ideas from audio fingerprinting may be beneficial in speech recognition. In this area, we attempt to identify a speaker based on sound characteristics in their voice.

1.1 Challenges

There are several challenges that an audio fingerprinting system will need to address. Identifying a song may be difficult for the following reasons:

- Different formats
- 'Cropping'
- Interference and other degradations
- Huge database to search

Some audio signals, such as radio broadcasts or live performances, may reach the fingerprinting system in analog form. Music already in digital form may be in a variety of formats, such as compressed as an MP3, or losslessly stored as a FLAC.

Another hurdle, 'cropping,' involves identifying an audio track when only a portion of the audio clip is available. Instances where this may be a problem is when identifying a song on the radio, where we began recording the query after the song had started. Another example is the intentional removal of a portion of audio, as discussed in the last section on P2P networks. An audio fingerprinting algorithm must be able to overcome this method of deception.

Interference problems, such as static from the radio or any background noise, could potentially alter the fingerprint, and any fingerprinting system would need to account for the possible addition of noise to the audio clip. In addition, other degradations to a signal such as pitching (where the audio is sped up to result in a higher pitched version of the original signal) or the effect of recording a clip through a cellphone microphone, may make it difficult for the fingerprinting system to correctly identify the audio.

The final challenge, namely the need to search through a huge database of songs is an issue of matching audio fingerprints. With several million audio tracks, the number of comparisons could be formidable if a brute force method was chosen. The audio fingerprinting system needs to quickly comb through the database for matches.

1.2 Outline

This thesis proceeds as follows. In Chapter 2 we review the basic ideas of Fourier theory. We then provide an introduction to wavelets in Chapter 3. We discuss the general framework of the audio fingerprinting process in Chapter 4, as well as examine previous work in this area. In Chapter 5 we present the Hokua algorithm, the main contribution of this work. In Chapter 6 we compare the performance of Hokua to the performance of two well-known algorithms. Finally we conclude the paper in Chapter 7, as well as look at further work we can do with the Hokua algorithm.

Chapter 2. Review of Fourier Theory

In this chapter we provide a basic review of Fourier theory that is fundamental to many audio fingerprinting systems. We also motivate the reasons for looking into wavelet methods for solutions to problems in audio fingerprinting. In Sections 2.1 and 2.2 we introduce basic ideas of Fourier analysis, namely Fourier Series and the Fourier Transform. In Section 2.3 we relate these methods to digital signal processing. As the most common transform in audio fingerprinting algorithms, we will look at the short-time Fourier transform in Section 2.4. Finally, in Section 2.5 we present an important result in digital sampling theory which dictates sampling rate parameters in audio fingerprinting systems.

2.1 Fourier Series

Often times in mathematics, we are interested in a function that we may not know a lot about. However, if we approximate this function with a set of functions that we are quite familiar with, we may be able to understand the original function better. In this section, we investigate the decomposition of a function using trigonometric functions.

Throughout this section, we will assume that f is a periodic function of a real variable with period T, that is, f satisfies the following equation,

$$f(x) = f(x+T), \qquad \text{for all } x \in \mathbb{R}.$$

We will refer to these functions as T-periodic. Note that for any integer n, we have that $f(x+nT) = f(x)$, and so a T-periodic function is also nT-periodic. We can restrict our attention to only one period, and work on the interval $[-T/2, T/2]$.

Since we want to approximate the periodic function f, we will use $\sin(x)$ and $\cos(x)$. Although these functions are 2π-periodic, we perform a horizontal stretch by multiplying the argument by $2\pi/T$ to obtain $\sin(2\pi x/T)$ and $\cos(2\pi x/T)$, which are T-periodic.

These functions will be used to approximate the function f. There is some redundancy, however, if we let $n \in \mathbb{Z}$. Since cosine is an even function, we need only restrict our attention to $n \geq 0$. Similarly, since sine is an odd function, we can once again restrict our attention to $n \geq 0$.

Furthermore, for $n = 0$ we have $\cos(2\pi nx/T) = 1$ and $\sin(2\pi nx/T) = 0$. Therefore only the T-periodic functions with $n > 0$ are used, as long as we include the constant function in our set. For convenience, we choose the value $\frac{1}{\sqrt{2}}$. Thus we have arrived at our set of functions:

$$\mathcal{E} = \left\{ \frac{1}{\sqrt{2}}, \cos\left(\frac{2\pi nx}{T}\right), \sin\left(\frac{2\pi nx}{T}\right) \Big| n \in \mathbb{Z}, n \geq 1 \right\} \tag{2.1}$$

Since the weighted sum of T-periodic functions is T-periodic, we will form a series to approximate f using this set.

Definition 2.1. The *Fourier series of the function f* is the equation

$$f = \frac{a_0}{\sqrt{2}} + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{2\pi nx}{T}\right) + b_n \sin\left(\frac{2\pi nx}{T}\right) \right), \tag{2.2}$$

where the coefficients $\{a_0, a_n, b_n | n \in \mathbb{Z}, n \geq 1\}$ are called the *Fourier coefficients* for the Fourier series of f.

We can calculate the Fourier coefficients using the following result from inner product spaces.

Theorem 2.2. *Let V be an inner product space and $\mathcal{E} = \{\mathbf{e}_n\}_{n=1}^{\infty}$ a complete orthonormal set in V. If $f = \sum_{j=1}^{\infty} a_j \mathbf{e}_j$, then $a_i = \langle \mathbf{e}_i, f \rangle$ for each i.*

Proof. Assume $f = \sum_{j=1}^{\infty} a_j \mathbf{e}_j$. Then

$$\begin{aligned}\langle \mathbf{e}_i, f \rangle &= \langle \mathbf{e}_i, \sum_{j=1}^{\infty} a_j \mathbf{e}_j \rangle \\ &= \sum_{j=1}^{\infty} a_j \langle \mathbf{e}_i, \mathbf{e}_j \rangle \\ &= \sum_{j=1}^{\infty} a_j \delta_{i,j} \\ &= a_j\end{aligned}$$

□

Recall that in $L^2([-T/2, T/2]; \mathbb{R})$, an inner product is given by

$$\langle f, g \rangle = \frac{2}{T} \int_{-T/2}^{T/2} f(x) g(x)\, dx.$$

The constant $\frac{2}{T}$ is for convenience.

Theorem 2.3. *The set $\mathcal{E}$ in (2.1) is an orthonormal set in $L^2([-\frac{T}{2}, \frac{T}{2}]; \mathbb{R})$.*

Proof. We need to show that the following inner products are all zero:

$$\begin{aligned} &\left\langle \tfrac{1}{\sqrt{2}}, \cos\left(\tfrac{2\pi n x}{T}\right) \right\rangle && n \neq 0, \\ &\left\langle \tfrac{1}{\sqrt{2}}, \sin\left(\tfrac{2\pi n x}{T}\right) \right\rangle && n \neq 0, \\ &\left\langle \sin\left(\tfrac{2\pi n x}{T}\right), \cos\left(\tfrac{2\pi m x}{T}\right) \right\rangle, && \\ &\left\langle \cos\left(\tfrac{2\pi n x}{T}\right), \cos\left(\tfrac{2\pi m x}{T}\right) \right\rangle && n \neq m, \\ &\left\langle \sin\left(\tfrac{2\pi n x}{T}\right), \sin\left(\tfrac{2\pi m x}{T}\right) \right\rangle && n \neq m, \end{aligned} \tag{2.3}$$

and that the following inner products are all one:

$$\begin{gathered}\left\langle \tfrac{1}{\sqrt{2}}, \tfrac{1}{\sqrt{2}} \right\rangle \\ \left\langle \sin\left(\tfrac{2\pi nx}{T}\right), \sin\left(\tfrac{2\pi nx}{T}\right) \right\rangle \\ \left\langle \cos\left(\tfrac{2\pi nx}{T}\right), \cos\left(\tfrac{2\pi nx}{T}\right) \right\rangle. \end{gathered} \tag{2.4}$$

These require several trig identities, but the integration is not difficult. This shows that for any $\mathbf{e}_i, \mathbf{e}_j \in \mathcal{E}$,

$$\langle \mathbf{e}_i, \mathbf{e}_j \rangle = \delta_{i,j},$$

and thus $\mathcal{E}$ is an orthonormal set. $\square$

Since the orthonormal set $\mathcal{E}$ of (2.1) is countable, then by Theorem 2.2, the Fourier coefficients are found by taking the inner product of the function, f, with each function in $\mathcal{E}$.

We thus obtain the Fourier coefficients for (2.2):

$$\begin{aligned} a_0 &= \left\langle \frac{1}{\sqrt{2}}, f \right\rangle \\ a_n &= \left\langle \cos\left(\frac{2\pi nx}{T}\right), f \right\rangle \\ b_n &= \left\langle \sin\left(\frac{2\pi nx}{T}\right), f \right\rangle \end{aligned}$$

This representation of a periodic function f as the sum of sinusoidal waves can be simplified somewhat when we consider that we can write

$$\sin(x) = \frac{1}{2i}\left(e^{ix} - e^{-ix}\right) \quad \text{and} \quad \cos(x) = \frac{1}{2}\left(e^{ix} + e^{-ix}\right) \tag{2.5}$$

(The algebra that follows could get a little messy... I would suggest wearing gloves.)

$$
\begin{aligned}
f &= \frac{a_0}{\sqrt{2}} + \sum_{n=1}^{\infty} \left(a_n \cos(\frac{2\pi n x}{T}) + b_n \sin(\frac{2\pi n x}{T}) \right) \\
&= \frac{a_0}{\sqrt{2}} + \sum_{n=1}^{\infty} \left(\frac{a_n}{2} \left(e^{i\frac{2\pi n x}{T}} + e^{-i\frac{2\pi n x}{T}} \right) + \frac{b_n}{2i} \left(e^{i\frac{2\pi n x}{T}} - e^{-i\frac{2\pi n x}{T}} \right) \right) \\
&= \frac{a_0}{\sqrt{2}} + \sum_{n=1}^{\infty} \left(\frac{a_n}{2} e^{i\frac{2\pi n x}{T}} + \frac{b_n}{2i} e^{i\frac{2\pi n x}{T}} \right) + \sum_{n=1}^{\infty} \left(\frac{a_n}{2} e^{-i\frac{2\pi n x}{T}} - \frac{b_n}{2i} e^{-i\frac{2\pi n x}{T}} \right) \\
&= \frac{a_0}{\sqrt{2}} e^{i\frac{2\pi(0)x}{T}} + \sum_{n=1}^{\infty} \left(\frac{a_n - i b_n}{2} e^{i\frac{2\pi n x}{T}} \right) + \sum_{n=-1}^{-\infty} \left(\frac{a_{-n} + i b_{-n}}{2} e^{i\frac{2\pi n x}{T}} \right) \\
&= \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi n x}{T}} \qquad (2.6)
\end{aligned}
$$

where

$$
\begin{aligned}
c_0 &= \frac{a_0}{\sqrt{2}} \\
c_n &= \frac{1}{2}(a_n - i b_n) \qquad \text{for } n > 0 \\
c_n &= \frac{1}{2}(a_{-n} + i b_{-n}) \qquad \text{for } n < 0
\end{aligned}
$$

This complex form in (2.6) leads us to define another special set, this time of complex exponentials, which is once again an orthonormal set. Thus by Theorem 2.2, we have that each c_n is the inner product of the corresponding complex exponential and the function f. Since we are allowing complex values, the inner product space is $L^2([-T/2, T/2]; \mathbb{C})$. An inner product is given by

$$\langle f, g \rangle = \frac{1}{T} \int_{-T/2}^{T/2} \overline{f(x)} g(x)\, dx.$$

Thus we can write the Fourier Series for a function f as

$$f = \sum_{n=-\infty}^{\infty} \langle e^{i\frac{2\pi n x}{T}}, f \rangle e^{i\frac{2\pi n x}{T}}.$$

To illustrate, consider the following example.

Example 2.4. Consider the T-periodic function $f(x) = |x|$ on $[-T/2, T/2]$, shown in Figure 2.1. We will compute the Fourier Series of this function.

We begin by finding the complex Fourier coefficients, c_n. For $n = 0$, we have

$$\begin{aligned}
c_0 = \langle 1, f\rangle &= \frac{1}{T}\int_{-T/2}^{T/2} |x|\,dx \\
&= \frac{1}{T}\left(-\int_{-T/2}^{0} x\,dx + \int_{0}^{T/2} x\,dx\right) \\
&= \frac{1}{T}\left(-\frac{x^2}{2}\Big|_{-T/2}^{0} + \frac{x^2}{2}\Big|_{0}^{T/2}\right) \\
&= \frac{1}{T}\left(\frac{T^2}{4}\right) = \frac{T}{4}.
\end{aligned}$$

For $n \neq 0$, using integration by parts and the identities in (2.5), we find

$$\begin{aligned}
c_n = \langle e^{i\frac{2\pi nx}{T}}, f\rangle &= \frac{1}{T}\int_{-T/2}^{T/2} |x| e^{-i\frac{2\pi nx}{T}}\,dx \\
&= \frac{1}{T}\left(-\int_{-T/2}^{0} x e^{-i\frac{2\pi nx}{T}}\,dx + \int_{0}^{T/2} x e^{-i\frac{2\pi nx}{T}}\,dx\right) \\
&= \frac{1}{T}\Big(-\left(\frac{-Tx}{2\pi ni}e^{-i\frac{2\pi nx}{T}} - \frac{-T^2}{4\pi^2n^2}e^{-i\frac{2\pi nx}{T}}\right)\Big|_{-T/2}^{0} \\
&\quad + \left(\frac{-Tx}{2\pi ni}e^{-i\frac{2\pi nx}{T}} - \frac{-T^2}{4\pi^2n^2}e^{-i\frac{2\pi nx}{T}}\right)\Big|_{0}^{T/2}\Big) \\
&= \frac{1}{T}\left(\frac{T^2}{4\pi ni}e^{i\pi n} + \frac{T^2}{4\pi^2n^2}\left(e^{i\pi n}-1\right) - \frac{T^2}{4\pi ni}e^{-i\pi n} + \frac{T^2}{4\pi^2n^2}\left(e^{-i\pi n}-1\right)\right) \\
&= \frac{1}{T}\left(\frac{T^2}{2\pi n}\left(\frac{1}{2i}\left(e^{i\pi n}-e^{-i\pi n}\right)\right) + \frac{T^2}{4\pi^2n^2}\left(e^{i\pi n}+e^{-i\pi n}-2\right)\right) \\
&= \frac{1}{T}\left(\frac{T^2}{2\pi n}\sin(n\pi) + \frac{T^2}{4\pi^2n^2}\left(2\cos(n\pi)-2\right)\right) \\
&= \frac{T}{4\pi^2n^2}\left(2\cos(n\pi)-2\right) \\
&= \begin{cases} 0 & n \text{ is even}, n\neq 0, \\ \frac{-T}{\pi^2n^2} & n \text{ is odd.} \end{cases}
\end{aligned}$$

We approximate f using the following partial sum, where the index is altered to only consider odd terms:

$$S_n = \frac{T}{4} - \sum_{k=-n}^{n} \frac{T}{\pi^2(2k-1)^2} e^{i\frac{2\pi(2k-1)x}{T}}$$

Figure 2.2 illustrates the progressive approximations of f by S_1,S_2 and S_{60}, and also provides a close-up of the origin.

In Example 2.4, we saw that the partial sum of the Fourier Series came quite close to approximating the

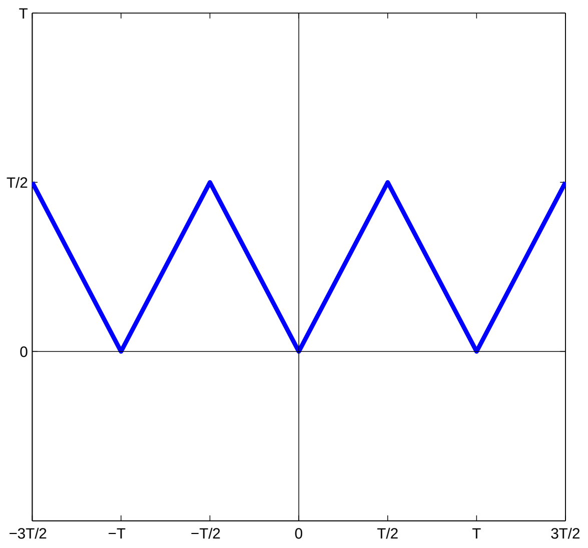
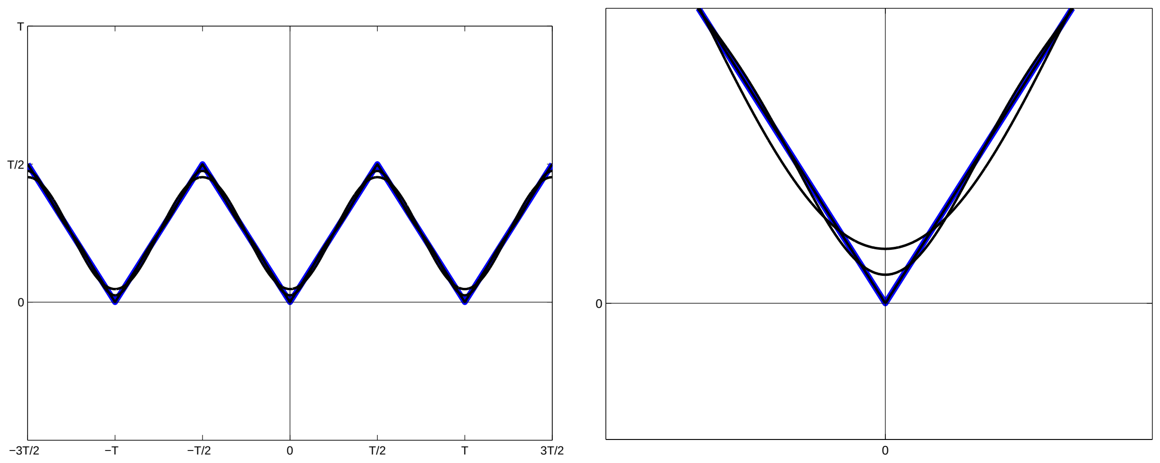


Figure 2.1: Fourier Series Example: $f(x)$



Approximations Zoom-in

Figure 2.2: Fourier Series Example (cont.)

signal. In this case we could store only the first few Fourier coefficients, and reconstruct the signal nearly exactly. This signal is called *sparse* in the Fourier basis, because we only need a few nonzero coefficients to obtain the signal.

However not all periodic signals are sparse in the Fourier basis, as we shall see in the next example.

Example 2.5. Consider the T-periodic function on $[-T/2, T/2]$,

$$f(x) = \begin{cases} 1 & -T/2 \le x \le 0, \\ 0 & 0 < x \le T/2, \end{cases}$$

shown in Figure 2.3. We will compute the Fourier Series of this function.

We begin by finding the complex Fourier coefficients, c_n. For $n = 0$, we have

$$\begin{aligned} c_0 = \langle 1, f \rangle &= \frac{1}{T} \int_{-T/2}^{T/2} f(x)\, dx \\ &= \frac{1}{T} \int_{-T/2}^{0} x\, dx \\ &= \frac{1}{T}\left(\frac{T}{2}\right) = \frac{1}{2}. \end{aligned}$$

For $n \neq 0$, using integration by parts and the identities in (2.5), we find

$$\begin{aligned} c_n = \langle e^{i\frac{2\pi n x}{T}}, f \rangle &= \frac{1}{T} \int_{-T/2}^{T/2} f(x) e^{-i\frac{2\pi n x}{T}}\, dx \\ &= \frac{1}{T} \int_{-T/2}^{0} e^{-i\frac{2\pi n x}{T}}\, dx \\ &= \frac{1}{T}\left(\frac{-T}{2\pi n i}\right)\left(e^{-i\frac{2\pi n x}{T}}\right)\Big|_{-T/2}^{0} \\ &= \frac{-1}{2\pi n i}\left(e^{i\pi n} - 1\right) \\ &= \frac{-1}{2\pi n i}\left(cos(n\pi) - 1\right) \\ &= \begin{cases} 0 & n \text{ is even}, n \neq 0, \\ \frac{1}{\pi n i} & n \text{ is odd}. \end{cases} \end{aligned}$$

We approximate f using the following partial sum, where the index is altered to only consider odd terms:

$$S_n = \frac{1}{2} - \sum_{k=-n}^{n} \frac{1}{\pi(2k-1)i} e^{i\frac{2\pi(2k-1)x}{T}}$$

Figure 2.4 illustrates the progressive approximations of f by S_1,S_6 and S_{61}, and also provides a close-up of the origin.

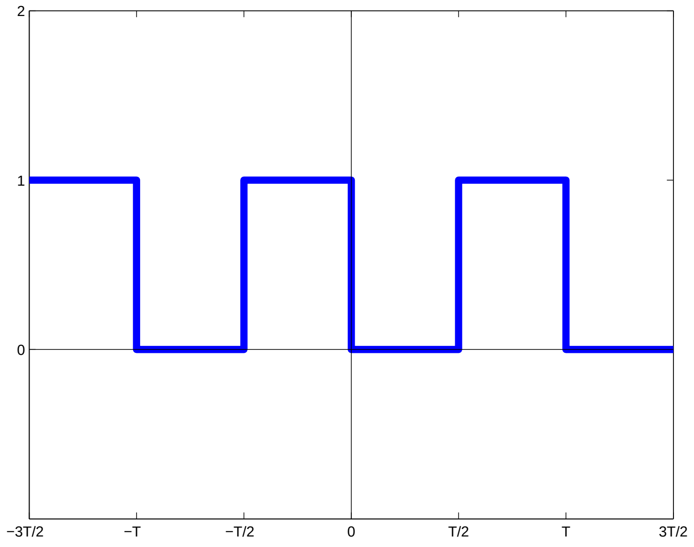
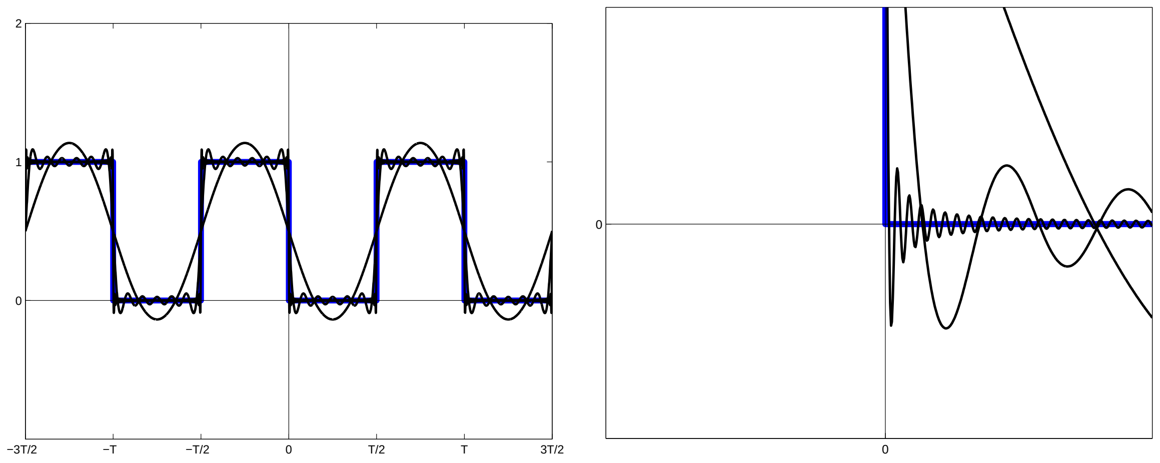


Figure 2.3: Fourier Series Discontinuity Example: $f(x)$



Approximations

Zoom-in

Figure 2.4: Fourier Series Discontinuity Example (cont.)

What we see in Example 2.5 is what is known as Gibb's phenomenon. While the partial sum approximation approaches the signal, we see more and more oscillations as we approach the discontinuity. In fact, every finite sum will result in the oscillatory behavior, above. Thus no finite set of Fourier coefficients are sufficient to reconstruct the signal, and so it is not sparse in the Fourier basis.

Considering that most real-world signals contain discontinuities, a Fourier basis seems less than ideal for approximation. As we shall see in Chapter 3, Example 3.4, a better method exists for approximating signals with discontinuities.

2.2 Fourier Transform

The Fourier coefficients represent the amount of certain oscillations that are present in a function. In many applications, this can be quite helpful in understanding the function. However the Fourier coefficients above are for a discrete collection of oscillations. The method for finding the amount of oscillation at any arbitrary frequency can be generalized from the work done for Fourier Series above.

The Fourier coefficients in (2.6) were determined by an inner product:

$$c_n = \langle e^{i\frac{2\pi nx}{T}}, f\rangle = \frac{1}{T}\int_{-T/2}^{T/2} f(x)e^{-i\frac{2\pi nx}{T}}\,dx.$$

We define a new function, $\hat{f}$, that maps the space of frequencies, $\mathbb{R}$, to $\mathbb{C}$ similarly. $\hat{f}$ returns the amount of a certain frequency present in the function f.

Definition 2.6. The *Fourier Transform* of a function f is $\hat{f}$, where

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i\xi x}\,dx.$$

This expresses the Fourier Transform of f as a function of ξ in units of ordinary frequency, Hertz (Hz), where one Hertz is one revolution per second. Alternately, we can express $\hat{f}$ as a function of ω in units of radians per second, using the substitution $\omega = 2\pi\xi$.

The Fourier Transform $\hat{f}(\xi)$ returns the amplitude and phase of a sinusoid oscillating at the frequency ξ within the signal f. As with Fourier series, these values can be used to reconstruct f, given f satisfies certain conditions.

Definition 2.7. The *inverse Fourier transform* of a function $\hat{f}$ is

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi)e^{2\pi i\xi x}\,d\xi$$

The inverse Fourier transform when $\hat{f}$ is a function of ω is given by

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega x} \, d\omega.$$

The Fourier Transform gives good information on how much of a frequency is present in a function, if at all. However the transform provides no information on where the frequency occurred in time. To show this, consider the following example.

Example 2.8. Let s_8 and s_{24} be two sinusoids that oscillate 8 times per second and 24 times per second, resp., each with a duration of .5 seconds:

$$s_8 = \begin{cases} \sin(16\pi x) & \text{for } 0 \leq x < \frac{1}{2}\,, \\ 0 & \text{otherwise }\,. \end{cases} \qquad s_{24} = \begin{cases} \sin(48\pi x) & \text{for } 0 \leq x < \frac{1}{2}\,, \\ 0 & \text{otherwise }\,. \end{cases}$$

Consider the following two signals:

$$\begin{aligned} f_{\text{split}}(x) &= s_8(x) + s_{24}(x - \frac{1}{2}) \\ f_{\text{merged}}(x) &= s_8(x - \frac{1}{4}) + s_{24}(x - \frac{1}{4}), \end{aligned}$$

illustrated in Figure 2.5, each having duration of 1 second. Both signals are composed of only two sinusoids, s_8 and s_{24}. The first, which we will call the split signal, separates the two, where s_8 comes first, followed by s_{24}. The second signal, which we will call the merged signal, sums the two waves in the middle of the signal.

Figure 2.5 also shows the Fourier transform of each signal. As can be seen, the transforms are nearly identical. The spikes at 8 and 24 represent the large presence of these frequencies in each signal.

The similar Fourier transform is not surprising. Consider a translation of f, $g(t) = f(t - a)$. Then

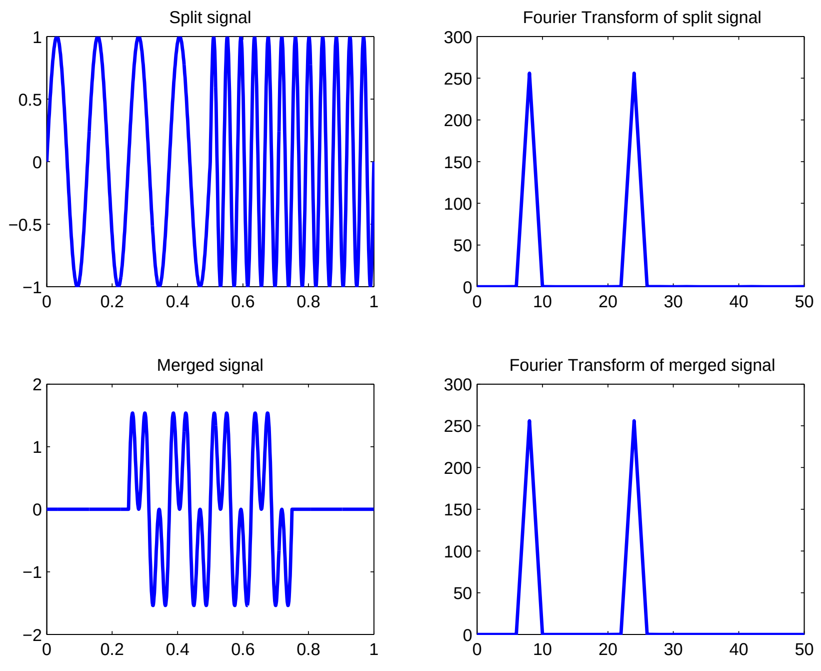


Figure 2.5: Lack of Temporal Resolution in Fourier Transform

$$\begin{aligned}\hat{g}(\xi) &= \int_{-\infty}^{\infty} g(x)e^{-2\pi i\xi x}\,dx\\ &= \int_{-\infty}^{\infty} f(x-a)e^{-2\pi i\xi x}\,dx\\ &= \int_{-\infty}^{\infty} f(t)e^{-2\pi i\xi(t+a)}\,dt\\ &= e^{-2\pi i\xi a}\int_{-\infty}^{\infty} f(t)e^{-2\pi i\xi t}\,dt\\ &= e^{-2\pi i\xi a}\hat{f}(\xi).\end{aligned}$$

Note that since the magnitude of $e^{-2\pi i\xi a}$ is one, $\hat{g}$ differs from $\hat{f}$ by a rotation in the complex plane. Thus the translation merely affects the phase of the Fourier transform. As we generally only use the magnitude of $\hat{f}$, translations can be ignored.

This inability of the Fourier transform to pinpoint a time where frequencies occur is referred to as the "lack of temporal resolution" of the transform. For signals where the frequencies do not change, this is not a problem. In the real world, however, most signals change over time.

One remedy for this problem is the short-time Fourier Transform, also known as a windowed Fourier transform. We will discuss this transform in Section 2.4. As a brief overview, this transform is performed by taking only short pieces of the function at a time. Each piece is treated as though the frequencies are unchanging. Thus the Fourier transform of the individual piece represents what frequencies are present for the duration of the short time frame. As we take the Fourier transform of several consecutive pieces, we are able to see how the frequencies in a signal change over time.

2.3 Digital Signal Processing

In this section we examine how these Fourier methods can be used to understand a digital signal. We then examine discrete-time Fourier transforms, the fast Fourier transform, and the power spectrum.

A digital signal is a sequences of values in time or space. Generally, these values are samples of some continuous process, called an analog signal, taken at uniform intervals. Digital signals are ubiquitous, composing most of the multimedia transmitted and stored in the world today. While images represent a 2-dimensional digital signal with values in space, we will confine our discussion to audio signals, which are sequences of samples in time. Our domain is therefore the real line.

The uniform spacing of samples presents an interesting problem. Section 2.5 presents the Shannon-Nyquist sampling theorem, which involves the rate with which an analog signal needs to be sampled in order to perfectly recreate the original signal. This rate will play a role in some of the audio fingerprinting systems discussed in Chapter 4.

The work done with Fourier series and transforms supposed that the function was defined at almost every point in the time domain. In order to use Fourier methods to understand digital signals, we use discrete-time Fourier transforms. As with continuous-time Fourier transforms, these compute the amount of different frequencies present in a discretely sampled signal.

Definition 2.9. The *discrete-time Fourier transform* of a length N discrete signal $f(n)$, is the function $F(m)$, where

$$F(m) = \sum_{0}^{N-1} f(n)e^{2\pi inm/N}.$$

$F(m)$ is referred to as the *frequency spectrum* of the signal.

The discrete-time Fourier Transform calculates the presence of certain frequencies from a discrete set of frequencies. These frequencies are spread uniformly over the angular frequency interval $[0, 2\pi]$.

The frequencies are determined by the number of samples, N, and the rate at which the samples were taken, ξ_{sr}. For $k = 0, 1, \ldots, N-1$, the k-th frequency is given by

$$\xi_k = \frac{k\xi_{sr}}{N}.$$

We may also denote these frequencies in terms of angular frequency,

$$\omega_k = \frac{2\pi k}{N}.$$

The Fast Fourier Transform (FFT) is an algorithm that calculates the discrete-time Fourier Transform quite efficiently. The length of the signal must be a power of 2, and if not, we pad the signal with zeros. The algorithm breaks the DTFT summation up into 2 summations of the even n and odd n. Each of these shorter summations are broken into 2 additional summations, and the process continues. In the final step, each sum is added together. Both [1] and [2] give good coverage of the FFT, as well as further explanation of most of the information found in this section.

Definition 2.10. The *power spectrum* of a signal is the square magnitude of each coefficient, that is $|F(m)|^2$.

The power spectrum at ξ_k represents the energy of the kth frequency in the signal.

2.4 Short-time Fourier transforms

In Section 2.1, we covered a common technique for analyzing a T-periodic function f by representing f as a linear combination of other functions.

However, what about functions that are not T-periodic. In particular let us consider a function $f \in L^2$, the space of square-integrable functions. An inner product on this space is

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(t)\overline{g(t)}\, dt.$$

One property of L^2 functions is that they decay to zero as we go off to infinity. Recall that the Fourier basis consists of sine and cosine functions of various frequencies. Note that these functions are not even in our space!

This problem can be remedied by forcing our basis functions to zero for large $|t|$. One way to bring these functions into L^2 is to incorporate a function g that will narrow each function to a specified window. This leads to the *short-time Fourier Transform*, also known as the *window Fourier transform.* As alluded to in Section 2.2, this will also help with the lack of temporal resolution, because the window narrows our analysis to only a segment of the function.

We choose the window function g so that its total mass is equal to 1, i.e., $\int g(x)\,dx = 1$. Also we choose g to be concentrated at 0, so that either $0 \in \text{supp}(g)$, where this support is compact, or the function has a max at zero, and decays to 0 as $|t| \to \infty$.

Figure 2.6 shows a few functions that are commonly used as window functions. A wide variety of windows exist, and one is chosen based on the application. The Hann window is also called the Hanning window. The equations for those pictured are as follows:

$$\begin{aligned} \text{Hamming:} \quad & g(t) = \tfrac{27}{50} - \tfrac{23}{50}\cos\left(\tfrac{\pi t}{a}\right) \\ \text{Hann:} \quad & g(t) = \tfrac{1}{2}(1 + \cos\left(\tfrac{\pi t}{a}\right) \\ \text{Gauss:} \quad & g(t) = e^{-t^2/(2a^2)} \end{aligned}$$

The parameter a is used to determine the width of the window, each of the windows in Figure 2.6 use $a = 1$.

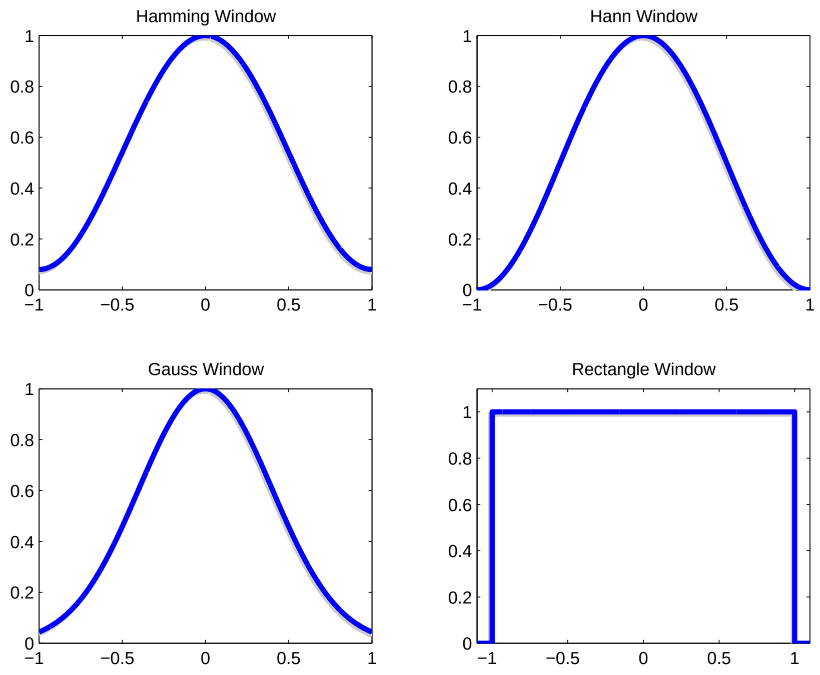


Figure 2.6: Example Window Functions

Definition 2.11. Given the window function $g(t)$, let $g_s(t) = g(t - s)$. The *short-time Fourier transform* of

f is defined by $Gf : \mathbb{R} \times \mathbb{R} \to \mathbb{C}$, where

$$Gf(\xi, s) = \frac{1}{\sqrt{T}} \int_{-\infty}^{\infty} f(t) g_s(t) e^{-2\pi i \xi t}\, dt.$$

This transformation will give us localized information about the frequencies of the function f. By translating the window function along the real line, we gain information about the function on each interval.

In digital signal processing, we can extend the idea of the STFT to discrete signals by weighting each sample according to a discrete window. Since the window is zero outside some region, this results in a segment of the digital signal of a given length, referred to as frames. Since the FFT requires a signal length that is a power of 2, the window is generally chosen to have 2^k non-zero entries. We will usually choose the value, k, and then sample a continuous window $g(x)$ at 2^k uniformly spaced points on its support.

Just as we move the window along the continuous signal, we will frame the digital signal starting at different points. The frames are overlapped so that no information is loss near the ends of a frame, which are generally diminished close to zero.

While the STFT gives better temporal resolution than Fourier transforms, problems still occur. For a window with 'width' h, difficulties arise when frequencies are present that are either much larger or much smaller than $2/h$. With the former, the oscillations happen so frequently that the window can't adequately localize the frequency (we still don't know where it occurs.) With the latter, we can't even see a full oscillation. The wavelet transform that we discuss in Chapter 3 addresses this problem.

2.5 Shannon-Nyquist Sampling Theorem

In the field of information theory, a common problem is that of taking a signal and converting it into a numeric sequence that can be transmitted. Most often the signal is continuous in time, e.g., a sound, or in space, e.g., an image, while the numeric sequence is discrete.

In this section we will prove the Shannon-Nyquist Sampling theorem, which plays an important role in the conversion from an analog signal to a digital signal.

We let $f(t)$ denote the continuous signal that we are interested in sampling. If $f(t) \in L^2$, then the continuous Fourier transform of f exists, and we denote this as $\hat{f}(\xi)$, where ξ is the ordinary frequency in hertz:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \xi t}\, dt$$

Definition 2.12. A signal is *bandlimited* to a one-sided baseband bandwidth B if it contains no energy at frequencies higher than B, that is

$$\hat{f}(\xi) = 0 \qquad \text{for all } |\xi| > B$$

What this tells us is that no frequencies of the signal are outside the range $[-B, B]$. We are now ready to state the main theorem of this section.

Theorem 2.13. ***Shannon-Nyquist Sampling Theorem*** *If a function $f(t)$ contains no frequencies higher than B hz, it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart.*

Proof. Let $\hat{f}(\xi)$ be the spectrum of $f(t)$, that is

$$f(t) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i \xi t}\, d\xi.$$

Since f is bandlimited, then $\hat{f}(\xi) = 0$ whenever $|\xi| > B$. Thus our integral simplifies to

$$f(t) = \int_{-B}^{B} \hat{f}(\xi) e^{2\pi i \xi t} d\xi.$$

Now if $t = \frac{n}{2B}$ where $n \in \mathbb{Z}$, then we can rewrite this as:

$$f\left(\frac{n}{2B}\right) = \int_{-B}^{B} \hat{f}(\xi) e^{i\frac{2\pi n \xi}{2B}}\, d\xi.$$

Recall from (2.6) that the Fourier series for a function $g(x)$ is given by

$$g(x) = \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi n x}{T}},$$

where T is the fundamental period and the Fourier coefficients are

$$c_n = \frac{1}{T} \int_{-T/2}^{T/2} g(x) e^{-i\frac{2\pi n x}{T}}\, dx.$$

The Fourier coefficients, then, of a function that is $2B$-periodic and matches $\hat{f}(\xi)$ on $[-B, B]$ are $c_n = \frac{1}{2B} \int_{-B}^{B} \hat{f}(\xi) e^{-i\frac{2\pi n \xi}{2B}}\, d\xi$. Thus $c_n = \frac{1}{2B} f(\frac{n}{2B})$. By the uniqueness property of Fourier transforms, $f(t)$ is uniquely determined by $\hat{f}(\xi)$. Since $\hat{f}(\xi)$ is obtained from the sequence of Fourier coefficients, $\{c_n\}$, (and set to zero outside of $[-B, B]$), then $f(t)$ is determined by $f(\frac{n}{2B})$. $\square$

Since the signal is "completely determined" by evaluating the signal every $1/(2B)$ seconds, then we can perfectly reconstruct the signal as long as we sample quick enough.

We will let ξ_{sr} be our uniform sampling frequency. Thus if $\xi_{sr} > 2B$, then we can exactly reconstruct our original signal. This $2B$ is known as the *Nyquist rate*, and is a property of the signal that tells us how fast we must sample. Equivalently, we can write the above equation as $B < \frac{\xi_{sr}}{2}$. This value, $\frac{\xi_{sr}}{2}$, is referred to as the *Nyquist frequency*, and is a property of the sampling system that tells us the types of signals that can be perfectly reconstructed from the system.

Consider the samples of a single continuous sinusoid pictured in Figure 2.7. These samples where taken every .2 seconds, i.e. our sampling rate, $\xi_{sr} = 5$. Thus as long as the largest frequency is smaller than 2.5 Hz, we can obtain the original signal. The left image in Figure 2.8 shows this case, where the sinusoid is 1.5 Hz, below the Nyquist frequency for our system. However the right image in Figure 2.8 shows the problems that occurs when the signal is not bandlimited.

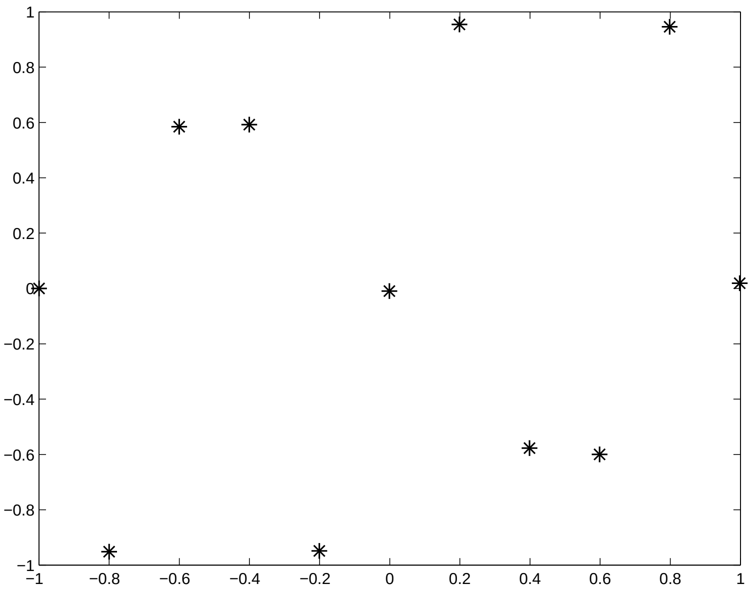


Figure 2.7: Sampling Rate Example: Discrete Samples

The time interval between successive samples is referred to as the *sampling interval*, and is given by $\tau = \frac{1}{\xi_{sr}}$. Thus we can write our discretized signal as $x_n = f(n\tau)$, for $n \in \mathbb{Z}$.

The reconstruction is achieved using the sinc function, illustrated in Figure 2.9:

$$\operatorname{sinc}(x) = \sin(\pi x)/(\pi x).$$

The analog signal is the linear combination of translated sinc functions. For each sample point, a corre-

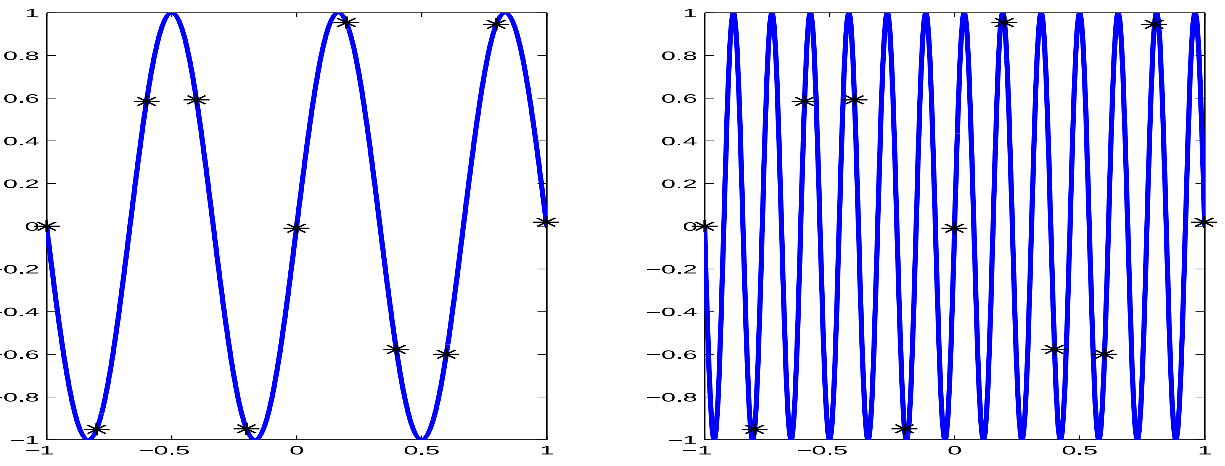


Figure 2.8: Sampling Rate Example: Potential Reconstructions

sponding sinc function is weighted by the sample value and shifted by the sample point's position in time.

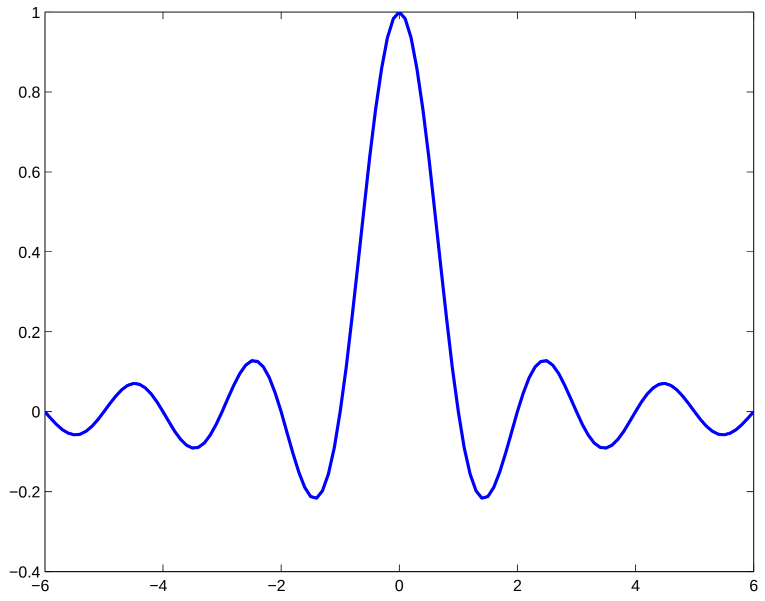


Figure 2.9: Sinc Function

Chapter 3. Introduction to Wavelets

In this chapter, we introduce the concept of the Wavelet transform. We look first at continuous wavelet transforms in Section 3.1, which bear resemblance to Fourier transforms. In Section 3.2 we examine the discrete Wavelet transform, which we utilize in Hokua.

3.1 Wavelets

As in our review of Fourier theory, we desire to approximate a function using a set of functions that we understand well. We begin with the mother wavelet. This function, ψ, will be used to construct our entire basis of functions, $\{\mathbf{e}_\alpha\}$, which are used in decomposing a signal. In choosing the mother wavelet, we are restricted by certain conditions. First $\psi \in L^2$, which means that ψ is square integrable, or that it has finite energy. Second, we require that ψ has a total mass of zero, that is $\int_{-\infty}^{\infty} \psi \, dt = 0$. A third condition, which applies only for complex valued mother wavelets is that the Fourier transform, $\hat{\psi}$ is real and has no negative frequencies. Beyond this, we have extensive freedom in choosing ψ, and later in Subsection 3.1.1 we will examine additional conditions we would like the mother wavelet to satisfy.

As an example of wavelets, consider Figure 3.1. These mother wavelets satisfy each of the conditions above. Along the top row are the Haar wavelet, the Gaussian wave, and the Mexican Hat wavelet. The bottom row features the Morlet wavelet, the Daubechies-4 wavelet, and the Coiflet-30.

First we will look at the functions that make up our basis.

Definition 3.1. For a given mother wavelet ψ, we define the (a, b) wavelet as

$$\psi_{a,b}(t) = \frac{1}{|a|^{1/2}} \psi\left(\frac{t-b}{a}\right).$$

This wavelet is a translation by b of a scaled copy (by a factor of a) of the mother wavelet. These wavelet functions will form a basis for L^2. As an example of these functions consider Figure 3.2. The wavelet is called the Gaussian wave, and is the first derivative of the Gaussian $e^{-t^2/2}$. The mother wavelet is shown in heavy black. A few other (a, b) wavelets are illustrated.

In order to analyze a signal, we will 'interrogate' the signal with the different (a, b) wavelets. This interrogation tells us how well the signal resonates with the wavelet. If the signal is constant, then the zero mass constraint will result in no resonance, while if the signal "oscillates" at the same rate as the wavelet, the magnitude of the resonance will be large.

Definition 3.2. The continuous wavelet transform of f with mother wavelet ψ is given by $W_\psi f : \mathbb{R}^* \times \mathbb{R} \to \mathbb{C}$, where

$$W_\psi f(a, b) = \langle \psi_{a,b}, f \rangle = |a|^{-1/2} \int_{-\infty}^{\infty} f(t) \overline{\psi\left(\frac{t-b}{a}\right)} \, dt.$$

We will usually suppress the $_\psi$ when the mother wavelet being used is understood.

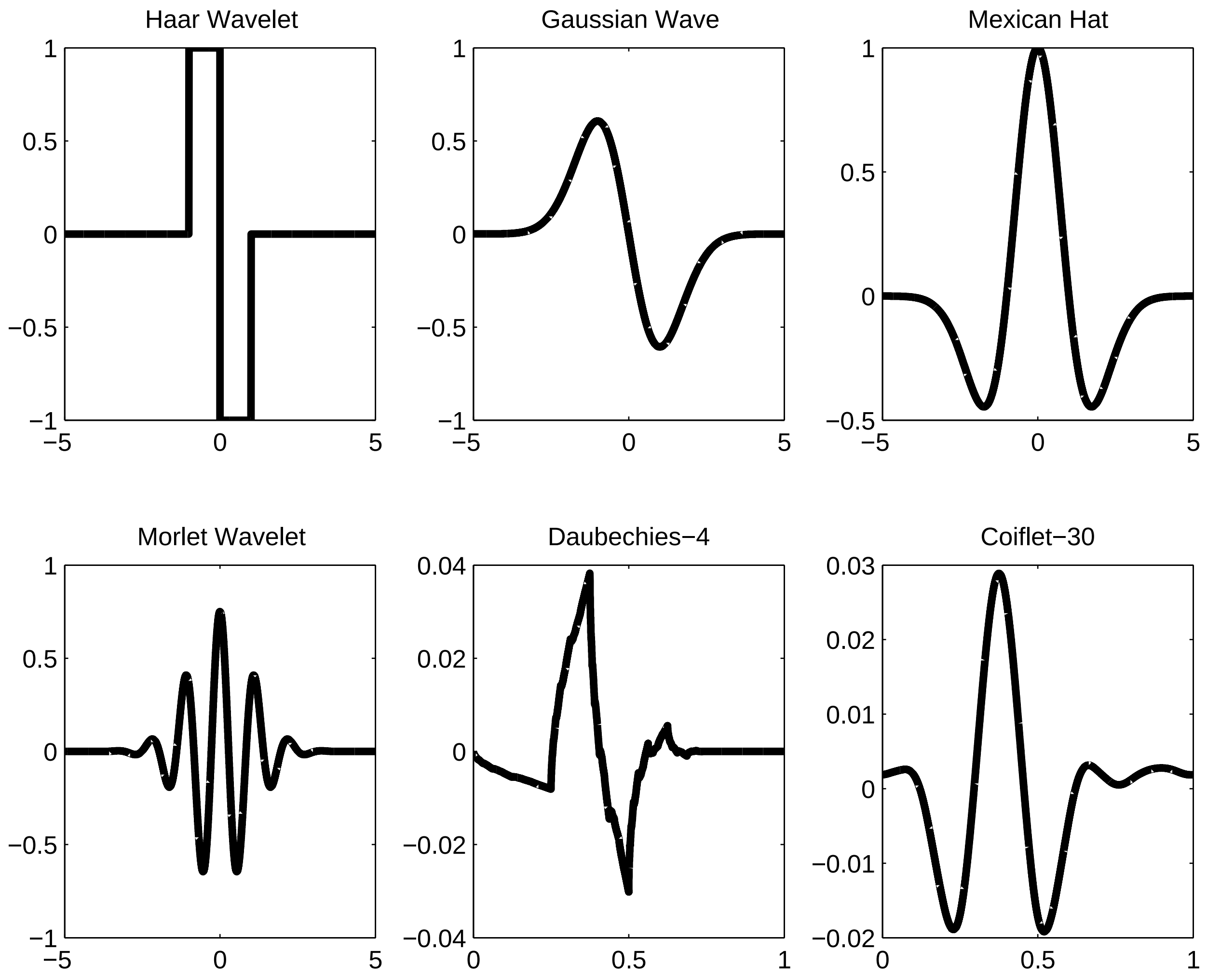


Figure 3.1: Examples of Several Mother Wavelets

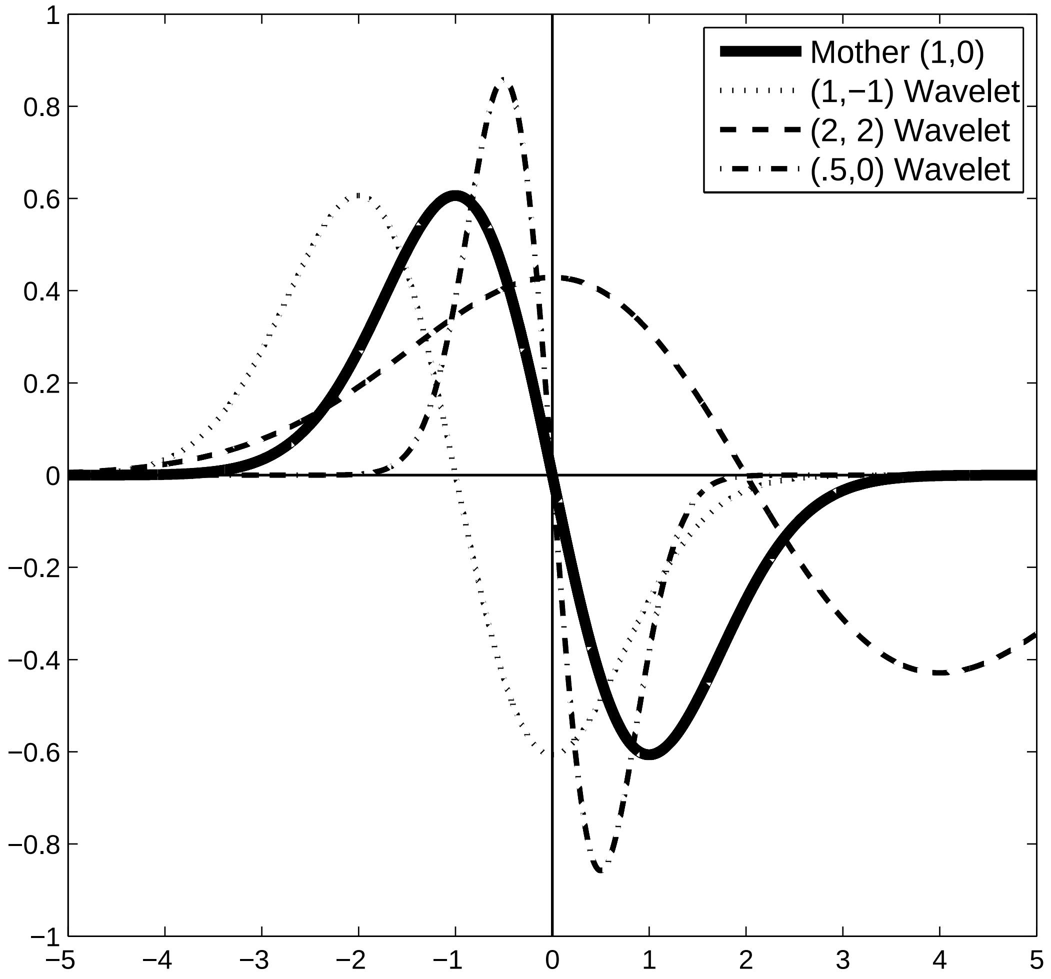


Figure 3.2: Mother Wavelet with Scaled and Translated Children

For the wavelet function $\psi_{a,b}$, a is called the scaling parameter and corresponds to the dilation of the mother wavelet. Smaller values of $|a|$ lead to narrower interrogating windows, while larger values of $|a|$ result in wider windows. Relating the wavelet back to the dilemma we faced with short-time Fourier transforms, we could use $0 < |a| \ll 1$ to localize the high frequencies of our signal, while $|a| \gg 1$ would give us information about the presence of low frequencies.

The translation parameter b serves the same role that s did in the short-time Fourier transform. It allows us to move the localized interrogating window through $\mathbb{R}$.

Definition 3.3. The inversion formula to obtain f from Wf is given by

$$f = \frac{1}{C_\psi} \int_{\mathbb{R}^* \times \mathbb{R}} \frac{1}{|a|^2} Wf(a,b)\psi_{a,b}\, da\, db,$$

where C_ψ is a constant that depends only on the mother wavelet.

The inversion formula is useful for reconstructing the original signal. In situations where the signal is sparse in the given wavelet basis, this is quite useful for storage purposes. The `JPEG-2000` standard makes use of wavelets in image compression.

As with the Fourier methods described in Section 2.1, we can use wavelets in a series to approximate a function f. The wavelet series expansion of a function f can be chosen by discretizing our parameters a and b. Generally the discretization will be logarithmic in scale parameter. For some fixed a_0, the scale is $a_r = (a_0)^r$, $r \in \mathbb{Z}$. The translation values are also discretized, and usually depend on the scale parameter. As the scale decreases, we need to take smaller steps, while we can use larger steps as the scale increases. Thus we will index b by two parameters, allowing our step size to change with the scale, a_r. That is we let $b_{r,k} = k a_0^r \beta$, where β is a base step size. Thus we can specify $c_{r,k} = Wf(a_r, b_{r,k})$, in which case

$$f \approx \sum_{\substack{r\in\mathbb{Z} \\ k\in\mathbb{Z}}} c_{r,k}\psi_{r,k}.$$

As an example, we will return to the signal introduced in Example 2.5. In this case we will use the Haar wavelet.

Example 3.4. Consider the T-periodic function on $[-T/2, T/2]$,

$$g(x) = \begin{cases} 1 & -T/2 \le x \le 0, \\ 0 & 0 < x \le T/2. \end{cases}$$

We will compute the Haar wavelet series for the function $f(x) = g(x) - 1/2$, shown in Figure 3.3.

To begin, we will use as our mother wavelet:

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \le x \le \frac{T}{2}, \\ -1 & \text{if } \frac{T}{2} \le x \le T, \\ 0 & \text{otherwise.} \end{cases}$$

Let us discretize logarithmically in scale, $a_r = 2^r$, and discretize the translation parameter in terms of scale, $b_{r,k} = Tk2^r$. We need to find the wavelet coefficients, $c_{r,k}$. For $r > 0$, we have

$$\begin{aligned} c_{r,k} = \langle \psi_{r,k}, f \rangle = |2|^{-r/2} \int_{-\infty}^{\infty} f(t)\psi\left(\frac{t - 2^r kT}{2^r}\right) dt \\ = |2|^{-r/2} \int_{2^r kT}^{2^r kT + 2^r T} f(t)\psi\left(\frac{t - 2^r kT}{2^r}\right) dt \\ = |2|^{-r/2} \left(\int_{2^r kT}^{2^r kT + 2^r T/2} f(t)\, dx - \int_{2^r kT + 2^r T/2}^{2^r kT + 2^r T} f(t)\, dt \right) \\ = |2|^{-r/2} \left(\int_{2^r kT}^{2^r kT + 2^r T/2} f(t) - f(t + 2^r T/2)\, dt \right) = 0, \end{aligned}$$

since f is $2^{r-1}T$-periodic.

For $r < 0$, we have

$$\begin{aligned} c_{r,k} = \langle \psi_{r,k}, f \rangle = |2|^{-r/2} \int_{-\infty}^{\infty} f(t)\psi\left(\frac{t - 2^r kT}{2^r}\right) dt \\ = |2|^{-r/2} \int_{2^r kT}^{2^r kT + 2^r T} f(t)\psi\left(\frac{t - 2^r kT}{2^r}\right) dt \\ = |2|^{-r/2} \left(\int_{2^r kT}^{2^r kT + 2^r T/2} f(t)\, dx - \int_{2^r kT + 2^r T/2}^{2^r kT + 2^r T} f(t)\, dt \right) \\ = |2|^{-r/2} \left(\int_{2^r kT}^{2^r kT + 2^r T/2} f(t) - f(t + 2^r T/2)\, dt \right) = 0, \end{aligned}$$

since f is constant from $2^r kT$ to $2^r kT + 2^r T$.

Finally, for $r = 0$, which is the mother wavelet, we find

$$\begin{aligned}
c_{0,k} &= \int_{-\infty}^{\infty} f(t)\psi\left(t - kT\right)\, dt \\
&= \int_{kT}^{(k+1)T} f(t)\psi\left(t - kT\right)\, dt \\
&= \int_{kT}^{(k+1/2)T} f(t)\, dt - \int_{(k+1/2)T}^{(k+1)T} f(t)\, dt \\
&= \int_{0}^{T/2} f(t + kT)\, dt - \int_{0}^{T/2} f(t + kT + T/2)\, dt \\
&= \int_{0}^{T/2} f(t) - f(t + T/2)\, dt \\
&= \int_{0}^{T/2} -dt = \frac{-T}{2}.
\end{aligned}$$

We approximate f using the following sum:

$$f = -\sum_{k=-\infty}^{\infty} \frac{T}{2}\psi_{0,k}.$$

Figure 3.3 illustrates this approximation of f. As can be seen, the signal is sparse in the orthonormal basis of this example. Wavelet transforms are thus better equipped to handle signals with discontinuities.

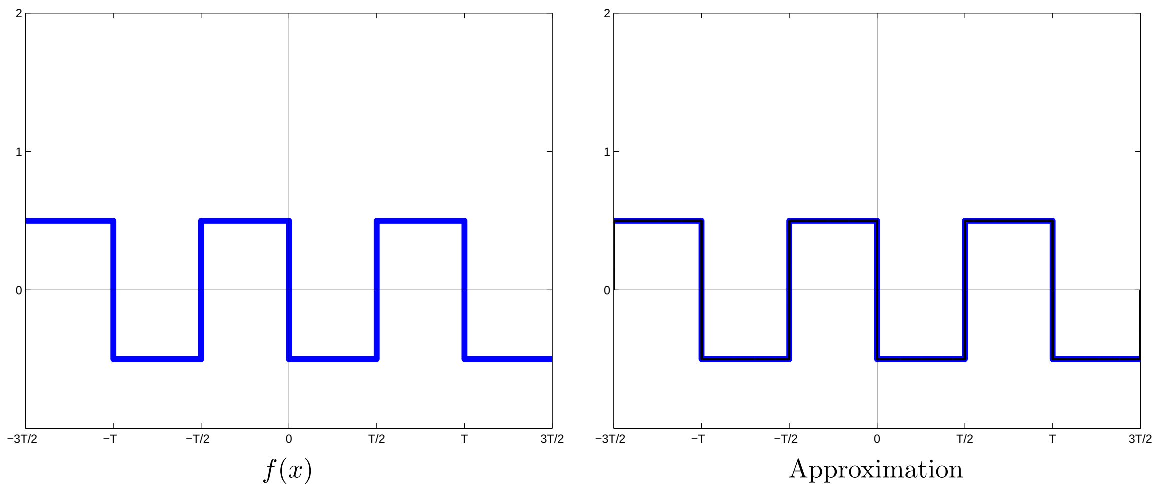


Figure 3.3: Wavelet Series Discontinuity Example

The theory of wavelet frames (see [4] and [10]) allow us to determine how closely the above approximation of 3.1 is to the function. While we do not cover this theory, one result deals with perfect approximation. In the case that the wavelets $\{\psi_{r,k}\}$ form an orthonormal basis, the approximation is exact.

3.1.1 Additional Conditions.

In addition to orthonormality, other factors we may require in choosing our mother wavelet, ψ, include:

- ψ has some symmetry.
- ψ has compact support.
- ψ is sufficiently differentiable.
- ψ has a certain number of vanishing moments.
- There are fast algorithms for calculating the transform coefficients.
- The coefficients have optimal decay as $r \to -\infty$.

The decision as to which wavelet to choose often depends on an application. For instance symmetric wavelets are often chosen for image applications because the human eye is sensitive to non-symmetric distortions [10].

In our implementation of Hokua, we use the 'Coiflet' mother wavelet. Using the discretization outlined in Section 3.2 below, it forms an orthogonal basis. Additionally, the Coiflet is designed to be symmetric and have a certain number of vanishing moments. We choose the Coiflet-30, which has 10 vanishing moments, for use in Hokua. The vanishing moment condition for a wavelet refers to how well the wavelet represents polynomial behavior in a signal. For example, the Coiflet-30 wavelet can approximate polynomials up to order 9 using only the approximation coefficients.

3.2 Discrete Wavelet Transform

Now we turn our focus to using Wavelet methods with discrete signals. Much of this is similar to handling discrete signals with Fourier methods. In order to begin, however, we must first decide upon a mother wavelet, ψ, as well as the scaling values a_r and transition values $b_{r,k}$ used in (3.1). As is customary, we use powers of 2 for our scales, i.e. $a_r = 2^r, r \in \mathbb{Z}$, and the transition values at each scale will be integer multiples of the scale, i.e. $b_{r,k} = k2^r, r, k \in \mathbb{Z}$. Finally we must choose the number of scales we wish to compute. This is constrained by the length of our signal, N, which must be a power of 2. The number of scales, J, must be less than than $\log_2 N$.

The inner product in the next definition is in $l^2(\mathbb{C}^N)$.

Definition 3.5. The *discrete Wavelet transform* of a length N discrete signal $f(n)$, is the function $W_\psi\, f(j,k)$, where

$$W_\psi\, f(j,k) = \langle f, \psi_{2^{-j},k2^{-j}} \rangle.$$

The scale $j \in \{1,\ldots,J\}$, and the translation $k \in \{0,\ldots,2^{-j}N-1\}$. We will suppress the $_\psi$ when the mother wavelet is understood. The $W f(j,k)$ are called the *wavelet coefficients* or *detail coefficients.*

The discrete Wavelet transform (DWT) calculates the presence, or resonance, of certain scales of the mother wavelet at certain times. We can distinguish the type of Wavelet transform by the number of scales it uses. The definition is for a general J-scale wavelet transform.

A portion of the DWT is missing from the above definition. In fact there are $N/2^J$ more coefficients. Functions known as scaling functions, denoted $\phi_{J,k}$, where $k \in \{0,\ldots,N/2^J-1\}$, are used to "fill in holes" in the basis. These *approximation coefficients*, $\langle f, \phi_{J,k}\rangle$, incorporate what remains of the signal after the J scales of wavelet coefficients have been calculated. These remainder coefficients are used in the reconstruction of the original discrete signal.

$$f(m) = \sum_{j=1}^{J} \left(\sum_{k=0}^{N/2^j-1} \langle f, \psi_{j,k}\rangle \psi_{j,k}(m) \right) + \sum_{k=0}^{N/2^J-1} \langle f, \phi_{J,k}\rangle \phi_{J,k}(m). \tag{3.1}$$

In fact the scaling functions play an important role in the DWT. Let S_J be a vector of the approximation coefficients of a J-scale DWT. We can think of the original signal as the remainder after a 0-scale wavelet transform. Thus the original samples of our signal make up our approximation coefficients, that is $S_0(n) = f(n)$, and we have $N/2^0 = N$ approximation coefficients. If we take the 1-scale DWT, we obtain $W\, f(1,k)$, the $N/2^1$ detail coefficients, and what remains are the $N/2$ approximation coefficients S_1. Now we can once again take the 1-scale DWT of S_1, to obtain $W\, f(2,k)$ and S_2. This process is continued for J scales. Figure 3.4 illustrates this process.

As with the FFT, quick and efficient ways of computing the detail coefficients are available to us. Using the idea above of obtaining the next level's approximation and detail coefficients from the current level's approximation coefficients, we are able to quickly calculate the J-scale DWT. This is facilitated by a relationship between the scaling functions and mother wavelet.

When calculating the DWT of a signal of length N, we arrive at a dilemma as to how to handle the ends of the signal. For a few translations of each scale, we interrogate beyond the edge of the signal. Several methods exist for handling the boundary, as shown in Figure 3.5. The vertical lines represent the beginning and ending of the segment being transformed. For more information on these boundary effects, see [10].

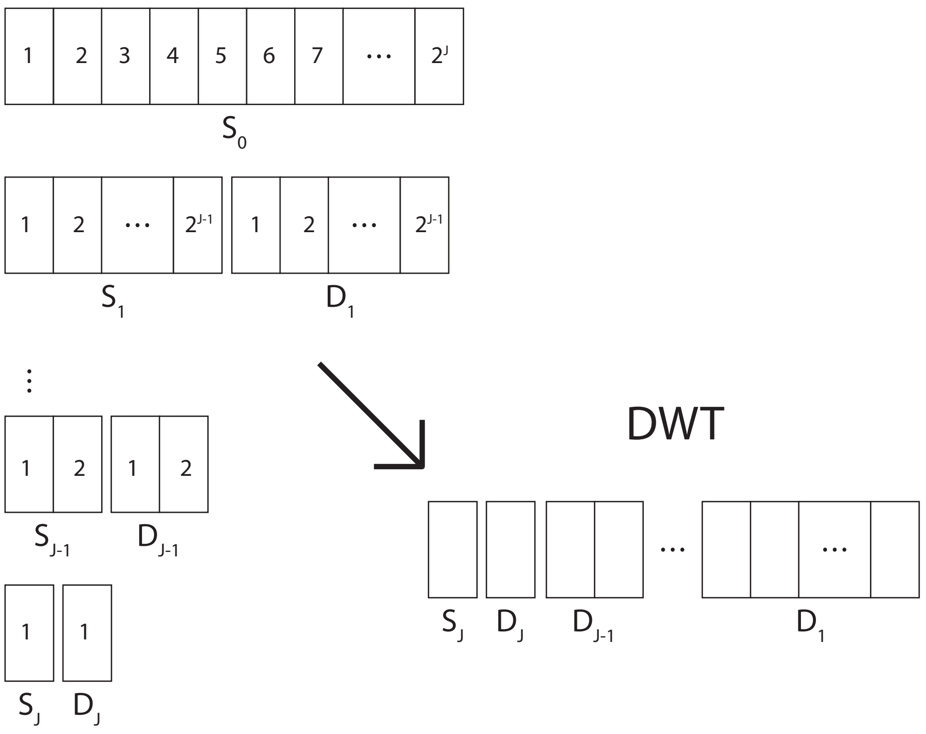


Figure 3.4: Calculating the DWT

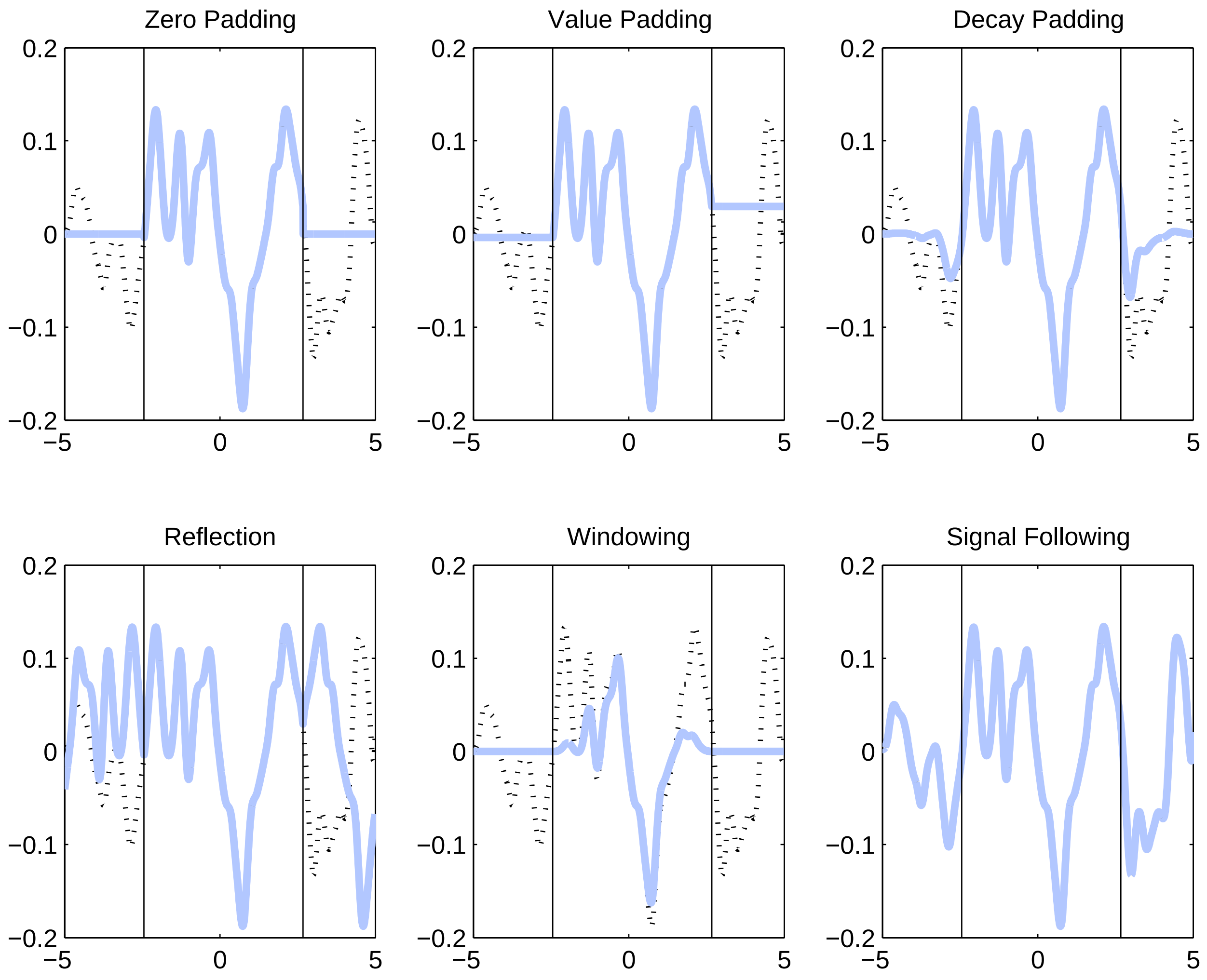


Figure 3.5: Examples of Boundary Handling

Another method, which we will adopt, resembles that which is done with Fourier methods, where we assume the segment is part of a periodic signal. In our implementation, we use a periodic wavelet transform, where the term periodic describes how the DWT handles calculating the coefficients near the end of the signal on any given scale. In this case, the DWT treats the signal as periodic, and so the beginning of the segment is appended to the end. It has the affect of wrapping the wavelet back to the beginning. Figure 3.6 illustrates this behavior, showing a wavelet of some scale translated through time. The dotted line represents some arbitrary signal.

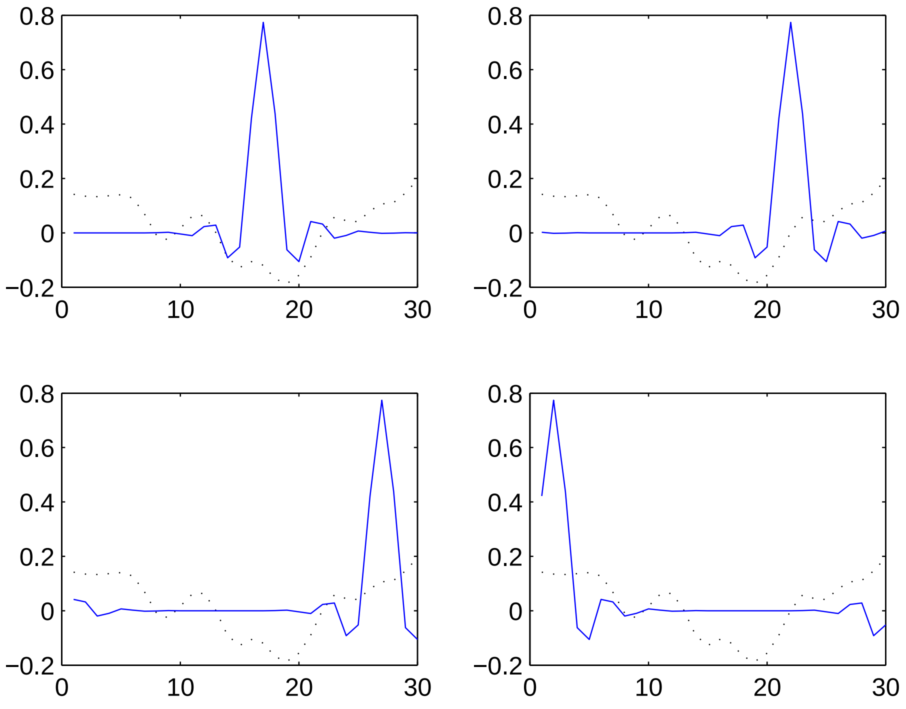


Figure 3.6: Example of periodic wrapping

Chapter 4. Audio Fingerprinting Methods

In this chapter we will discuss the current state of audio fingerprinting. Most of this chapter relies on the review article of Cano et al., [14], who summarized and reviewed different audio fingerprinting methods. In Section 4.1, we discuss the general framework of audio fingerprinting systems. We give a brief history of different fingerprinting algorithms in Section 4.2. In Sections 4.3 and 4.4 we discuss two influential algorithms, a Highly Robust Algorithm (HRA) of Haitsma and Kalker [16], and an Industrial-Strength Algorithm (Shazam) of Wang [17].

4.1 Framework

Audio fingerprinting lends itself to a division of two main processes. The first process is that of fingerprinting an audio clip. The resulting fingerprint is used to either expand the database of known songs, or query the database when the clip is unknown. The second process is that of matching the query fingerprint against the database to find a match, as well as testing the resulting match to ensure accuracy. We will discuss both of these processes here.

4.1.1 Fingerprinting. As pointed out in [14], the first process, fingerprinting, is broken up into two blocks. The first block, known as the front-end, takes the input signal and standardizes the format, performs certain transforms, and returns values of interest about the signal. This output is then fed into the second block, known as the fingerprint model. This block creates the fingerprint from the values returned by the front-end.

The front-end is broken up into the following phases:

(i) Preprocessing

(ii) Framing & Overlap

(iii) Transform

(iv) Feature Extraction

(v) Post-Processing

Preprocessing: This phase is used to ensure that each signal is in a common format. Generally this will involve averaging channels so that the signal is in mono, as well as resampling the audio at a specified rate.

Framing and Overlap: Generally we would like to treat the signal as stationary to perform analysis on it. In order to accomplish this, we split the signal up into short duration frames. The frames give us a small window at which to look at the signal, over which we treat the signal as stationary. In order to ensure that signal features occurring on the boundaries of these frames are accounted for, we overlap the frames. A common percentage of overlap is 50%.

Transform: At this point the signal is transformed into a different domain. Generally this transform is a Fourier transform, though other transforms, such as the wavelet transform, are used.

Feature Extraction: For the front-end, this phase is the most variable, as the feature that is calculated depends entirely on the algorithm. Some examples of features include energy, derivative of energy, Mel-frequency cepstral coefficients, and subband centroids, as well as high level approximations of pitch and tempo.

Post-Processing: In this stage, the feature vectors are manipulated to arrive at a form used in the fingerprinting model. Values are quantized, and vectors or hashes are formed from the calculated values of the feature extraction.

The second block, the fingerprinting model, also varies greatly depending on the algorithm. Within the fingerprinting model, the feature vectors returned by the front-end are encoded as a fingerprint. The model will combine the vectors into a fingerprint that is used in either expanding or querying the database.

4.1.2 Matching. This process of matching a query fingerprint against a database can be divided into three areas:

(i) Distance

(ii) Search

(iii) Verification

Distance: In order to compare two fingerprints, we need to define a metric between them. The distance should give a general idea of how similar two fingerprints are. Examples of distances include the Euclidean distance for finding the distance between vectors in $\mathbb{R}^n$, or the hamming distance, an adaptation of which is used in Section 4.3 to define the BER.

Search: Even after defining a good measure of distance between two fingerprints, we still need to find a suitable fingerprint from the database that has a small distance to the query fingerprint. This problem proves to be quite difficult, but several solutions exist. The search side of matching will scour the database for potential matches to a query fingerprint, and each potential match can be compared to the query using the distance metric. Schemes were the fingerprints are indexed can speed the search.

Verification: Once we obtain a candidate match from the database, we need to verify the match. Some algorithms use a simple threshold to determine whether a candidate should be accepted, while others use hypothesis testing or compute Bayes factors to determine validity.

There are certain desired qualities of the matching process:

(i) Fast: Linear searches through an entire database can be much too slow. The search should quickly identify potential matches, and the distance computed quickly for each potential song.

(ii) Correct: Incorrect identifications should be minimized (a low false acceptance or false positive rate), while not missing the true song (a low false rejection or false negative rate).

(iii) Memory efficient: With millions of songs, the database can be quite large. The data should be stored in a compact form, with little overhead.

(iv) Easily updatable: As new recordings become available, the database can become outdated. Songs should easily be inserted, deleted or updated within the database.

Each of these are important, and will form the basis for some of the comparisons we make in Chapter 6. It should be noted, however, that these qualities are often in contradiction, such as a smaller fingerprint may result in more false identifications, or a fast indexing system may add complexity in updating the database.

4.2 History of Audio Fingerprinters

Prior to audio fingerprinting, the only methods for identifying an audio clip were either by listening to it directly (which you may not recognize) or by the clip's metadata (which may be incorrect). The idea for content-based identification is intended to use the perceptual qualities of the audio to identify the clip. Two methods arose–watermarking and fingerprinting.

The first method, watermarking, imbeds identifying information into the song itself. The goal is to imbed it in such a way that it is not perceived, yet a program can pick it out and return the desired information. This method, however, requires that the source of the audio clip has included the watermark, and that it hasn't been tampered with. While an interesting approach to the problem, it is not the one we consider here.

The second method, fingerprinting, endeavors to create a database against which an unknown audio clip can be matched in order to identify the query clip. This method is the approach that we consider.

An audio fingerprinting algorithm with widespread influence and stands as a standard for most audio fingerprinters is the algorithm of Jaap Haitsma and Ton Kalker that they propose in [16], which we will refer to as HRA (Highly Robust Algorithm). This algorithm uses the Fourier transform, and focuses on the energy in different frequency bands to create a fingerprint. As the standard, we compare the performance of this algorithm against Hokua in Chapter 6. More about this algorithm can be found in Section 4.3.

Several other algorithms are derivatives of HRA. Those found in [19], [22] modify or build off of HRA, in order to improve some of Haitsma and Kalker's results. Other work, such as [20], use HRA as the fingerprinter for their searching algorithm.

Avery Wang, in [17], presents another approach, which is used by the Shazam application. This approach uses the Fourier transform to obtain a spectrogram of an audio signal. Local maxima of the spectrogram are selected and used to create several different hashes that are used in matching to find the most likely song. Since this system performs well, and forms the basis for our algorithm, we compare it against Hokua in Chapter 6. More about this algorithm can be found in Section 4.4.

As Hokua uses a wavelet transform, it is worthwhile to mention other algorithms that also rely on wavelet transforms. One proposed by Hsieh and Wang in [21] is based on the Haitsma and Kalker algorithm. The main difference is that instead of a STFT, Hsieh et al. perform a DWT. We also attempted this approach, but abandoned it due to redundancy in time. Instead of banding frequency space as is done with HRA, Hsieh et al. band the approximation coefficients after a 2-scale Haar wavelet transform. Thus instead of grouping frequencies, they group the "leftover" coefficients in time. With the large overlap, we see diagonal patterns that represent the slow shift of coefficients. Due to the temporal resolution offered by the wavelet transform, the overlap resulted in the exact same coefficients.

Another algorithm, introduced by Google employees Shumeet Baluja and Michele Covell in [18], also utilizes wavelets, but in an image compression context. Using the Fourier transform, the WavePrint algorithm produces a spectrogram for an audio clip. This spectrogram image is then transformed using a 2-dimensional wavelet transform. Of the resulting coefficients, only the K largest are retained. These coefficients, with their corresponding scale and translation, are encoded as the fingerprint for the clip.

Other wavelet-based methods such as those found in [11] and [23] are used in audio classification, as opposed to identification.

4.3 A Highly Robust Algorithm

We first look at the algorithm proposed by Haitsma and Kalker in their article "A Highly Robust Audio Fingerprinting System," [16].

The algorithm has been split into two pieces, that portion which fingerprints an audio signal, and that portion which matches fingerprints against a database of known audio clips. We have implemented this algorithm using MATLAB, and we first introduce the algorithm, then discuss its implementation, and finally we discuss the matching methods.

4.3.1 Fingerprinting Algorithm. This algorithm follows the steps outlined in Section 4.1. The pre-processing converts the signal into double-precision, normalized samples, and downsamples the signal to about 5 kHz. As we are interested in frequencies only up to 2 kHz, this rate is sufficient according to the results of Section 2.5. Also, when the signal is in stereo, the channels are averaged to obtain a mono signal.

In the framing and overlap step, the signal is framed using a Hanning window, with an overlap of 31/32. This large overlap results in each frame being quite similar to the previous frame, and so we would expect the signal to not change much from frame to frame.

The transform that is performed in this algorithm is a discrete Fourier transform. This maps each frame into frequency space, where the analysis and fingerprinting of the signal takes place. In particular for this algorithm, we take the magnitude of each coefficient, which represents the amount of energy of each frequency.

At this point, we have the energy of various frequencies for each frame. The feature that we are interested in extracting is how the energy is changing. Using a log scale, we create 33 bands with log spacing. As the authors note in [16], the reason for this log scale is that it closely resembles how the ear reacts to different frequencies. The energy in each band is aggregated, and then the difference in energy between a band and the next band is calculated.

At this point, in the post-processing step, we encode how this band energy difference changes from frame to frame. We quantize the change as a 0 if the band energy decreases, and 1 if the band energy increases. This results in a fingerprint for the original signal, which can then either be used to expand a database, or as a query of the database for identifying purposes.

4.3.2 Implementation. The MATLAB scripts `fingerprint.m` and `basefinger.m` both fingerprint an audio signal using this algorithm. The former is used to create a fingerprint for a database query, and returns only the fundamental 256×32 fingerprint needed for matching. The latter fingerprints an entire song, which is then used to expand the database of recognizable songs. In each, the audio signal is converted using either `wavread.m` or a similar m-file `mp3read.m` (available at [27]) to return `y`, a vector of the double-precision, normalized samples, as well as the clip's sampling rate, `Fps`. The `Fps` is used in downsampling, so that we have a signal sampled around 5kHz. In particular, for a song sampled at 44.1kHz, the downsampled audio will be 5292 Hz. The sample rate, in Hz, of the downsampled signal is denoted `fs`.

We frame the signal so that each frame has a length of 2048 samples. The overlap of 31/32 results in each frame starting 64 samples after the previous frame. Each frame is approximately 38 seconds long, and due to the large overlap, a new frame starts every 12 milliseconds. The frame is weighted using a Hanning (or Hann) window to minimize the effect of discontinuities at the endpoints.

Since we are interested in taking the DFT of each frame, we use the Fast Fourier Transform (FFT) to more efficiently calculate the coefficients for each frequency. Since the human auditory system is insensitive to phase shift, we record only the magnitude of each coefficient.

We use the command `logspace(2.477,3.301,34)` to divide the frequency range from $10^{2.477} \approx 300$ Hz to $10^{3.301} \approx 2000$ Hz into 33 intervals. In order to find which entries these frequencies correspond to in the transformed signal, we use the relation that

$$k = \frac{f_k * 2048}{\texttt{fs}}.$$

We find the total energy for each band by summing the coefficients of the frequencies between these separating frequencies. This gives us a matrix E, the size of which is the number of frames in our fingerprint by 33.

We then compute the difference in energy between two consecutive bands for each frame, DE. The fingerprint is encoded logically, where the (i, j) entry (corresponding to the i^{th} frame and the j^{th} band) is 1 if $DE_{i,j}$ is larger than $DE_{i-1,j}$, and 0 otherwise. We convert these logical values into unsigned, 8-bit integers.

Figure 4.1 shows a sample query fingerprint of a popular children's song. Black pixels correspond to 1.

4.3.3 Matching.

In their paper, [16], Haitsma and Kalker list a variety of ideas for the matching portion of their algorithm. We choose only a few of these ideas to include in our implementation of the matching process, found in the file `matchfingerprint.m`.

The first step for the matching algorithm is to have a database of fingerprinted originals. For this algorithm, I created a script, `expanddatabase.m`, which takes a list of songs and either creates a new database from them, or appends to an existing database. The script goes through the list, song by song, and performs two actions. First, the song is fingerprinted, using `basefinger.m` as discussed in 4.3.2. The second procedure extends the database by inserting every subfingerprint for each song.

The database contains four components. The first two components store the actual information about the songs. The matrices `SONGS` and `METADATA` are each 150×1 cell arrays, the former stores the entire fingerprint for each of the 150 songs in our database. The corresponding cell of the latter is the song's metadata. The remaining two components are for indexing the subfingerprints.

The index scheme is known as a linked list. This list is implemented in the position matrix, `posmat`. Each entry stores the song ID and time index, i.e. the position, of a certain subfingerprint within that song, as well as a link to the next entry in the list.

Since each subfingerprint is a 1×32 vector of either zeros or ones, there are 2^{32} different subfingerprints. The final component is the lookup table, `LUT`, a matrix large enough to hold every possible subfingerprint.

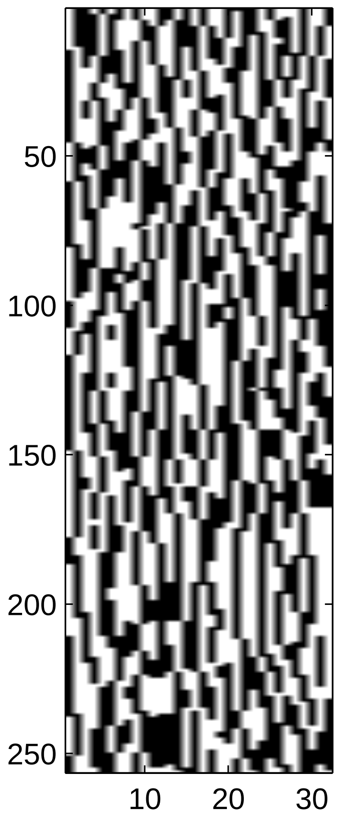


Figure 4.1: Sample HRA Query Fingerprint

Since we don't expect to see even a fraction of all of these possible subfingerprints, this matrix is created as a sparse matrix with alloted memory for 750,000 nonzero entries, though this could be expanded easily enough. The values stored in the `LUT` are addresses to the first entry of the linked list for the given subfingerprint.

In order to match, we need a metric to determine how close two fingerprints are. We do this using a Hamming distance, counting the number of bits that are different between the fingerprints, and dividing by the total number of bits. We will call this number the Bit Error Rate (BER). For two 256×32 queries, P and Q, where $\ominus$ denotes subtraction mod 2,

$$\text{BER} = \frac{\sum (Q_{i,j} \ominus P_{i,j})}{8192}.$$

Matching proceeds as follows: For each subfingerprint, s_i in our query fingerprint, we find the corresponding position in the `LUT`, and read the first address for our linked list. If the entry is zero, then there are no matching subfingerprints in the database, and we move on to s_{i+1}. Otherwise we follow the linked list to the final node, checking the query fingerprint against each fingerprint identified in the list by song id and time position. Having done this for every subfingerprint in the query, the algorithm returns the database track with the smallest BER, as long as this value is below some predetermined threshold. Haitsma and Kalker suggest using 0.35 as the cutoff. If no song has a BER with the query below this tolerance, than no match is declared.

This matching system requires that at least one of the subfingerprints matches exactly the corresponding subfingerprint of the database in order for it to match the fingerprint. The authors of HRA propose another method to weaken this constraint. When s_i is being checked, they also check the database for $s_i \oplus \mathbf{e}_j, j \in \{1, 2, \dots, 32\}$, where $\mathbf{e}_j$ is the row vector of zeros with a one in the j^{th} entry, and $\oplus$ denotes addition mod 2. This flips each of the bits successfully, trying each subfingerprint. This extends the search to all subfingerprints within one hamming distance of the query subfingerprint, s_i. We have implemented both approaches.

4.4 An Industrial-Strength Algorithm

We will now look at an algorithm proposed by Avery Wang in the article "An Industrial-Strength Audio Search Algorithm,"[17]. As Wang went on to create the audio fingerprinting application Shazam, we will refer to this algorithm by that name.

As in Section 4.3, the algorithm is split into a portion which fingerprints an audio signal, and a portion which matches fingerprints against a database of known audio clips. We use a MATLAB implementation of this algorithm by Dan Ellis found at Columbia [28]. We first introduce the algorithm, then discuss its implementation, and finally we briefly discuss how matching is performed.

4.4.1 Fingerprinting Algorithm.

This algorithm follows the steps outlined in section 4. The preprocessing converts the signal into double-precision, normalized samples, and resamples the signal to 8000 Hz. Also, when the signal is in stereo, the channels are averaged to obtain a mono signal.

In the framing and overlap step, the signal is framed using a Hanning window, with an overlap of 1/2. Each frame is 512 samples long. The signal is then transformed into the time-frequency domain using the fast Fourier transform. In this algorithm, we take the magnitude of each coefficient, to obtain the energy in each frame at each frequency. The energy is known as the spectrogram.

The feature that we are interested in extracting is the location of local maxima in the spectrogram. We take each local maximum as an anchor point, and pair it with other maxima with in a target zone of the spectrogram (that spreads out from the anchor in frequency, and forward in time).

During the post-processing, Shazam encodes the frequency magnitudes of the maxima in each pair, as well as the time between them, into a hash. It then combines the time onset of the anchor point, and, in the case of extending a database, the song id, with the hash to form a portion of the fingerprint. The fingerprinting model combines all the hashes for a query or database fingerprint.

4.4.2 Implementation.

As mentioned, we use Dan Ellis' implementation of the fingerprinting system. We made slight modifications to his code to create a permanent database, as well as some changes to the driving script file, `demo_fingerprint.m`, in order to suit our needs. A given audio signal is read in, as in 4.3, with either `wavread.m` or `mp3read.m`. This yields `y`, a vector of the double-precision, normalized samples, as well as the clip's sampling rate, `Fps`. These values are passed in to the function `find_landmarks.m`. Here stereo data is combined into a mono signal, and then resampled to 8kHz.

The signal is framed so that each frame has a length of 512 samples, which at the resampled rate constitutes 64 milliseconds of audio. The overlap of 1/2 results in a new frame starting every 32 milliseconds. The frame is weighted using a Hanning (or Hann) window to minimize the effect of discontinuities at the endpoints. Once again, the FFT is used to calculate the discrete Fourier transform coefficients. We record only the magnitude of each coefficient.

At this point, we have the spectrogram, and `find_landmarks.m` searches through the matrix to find local maxima. Having found these, pairs are formed and a matrix, `L`, of hashes is set up. Each row of `L` is a hash, where the columns are the time of the anchor point, the frequency of the anchor point, the frequency of the paired point, and the change in time between the two points.

If we fingerprint the audio clip in order to expand the database, the hashes in `L` are all appended with a song Id used in identifying which clip the hash comes from. These hashes are all added to `HashTable`, which is saved as a `.mat` file for later use. On the other hand, if the audio clip is a query, the hashes in `L` are passed on to the matching algorithm.

Figure 4.2 illustrates a sample query fingerprint of a popular children's song.

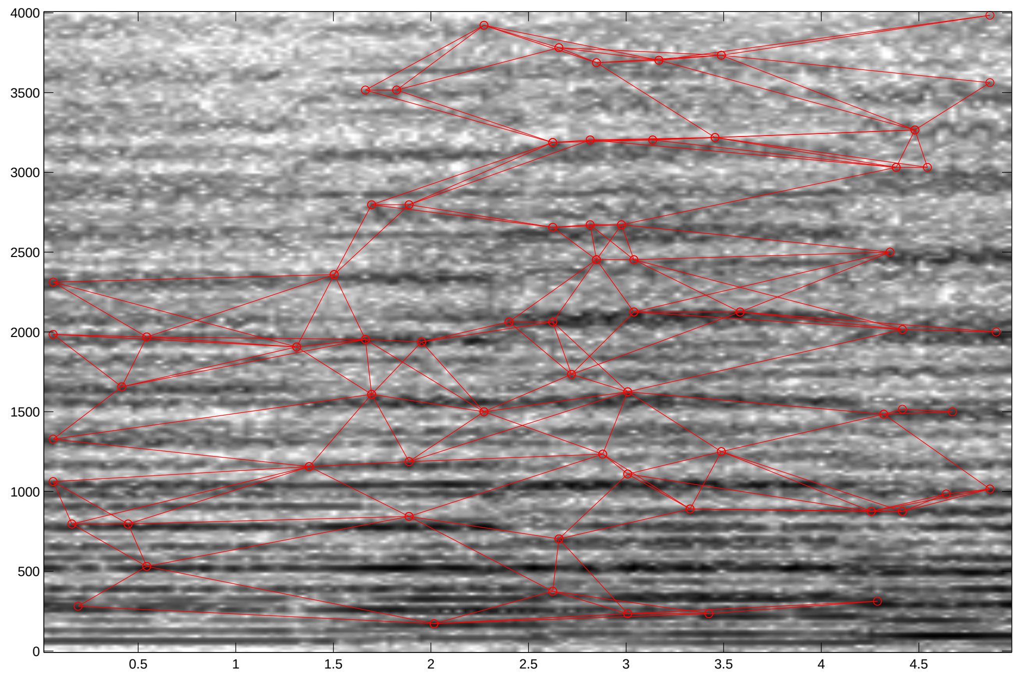


Figure 4.2: Sample Shazam Query Fingerprint

4.4.3 Matching Algorithm.

Matching for a query Q is performed by finding all hashes in the song database that match the hashes of Q. These are sorted by song ID, and for each song with at least one matching hash, the most popular offset time is calculated. The offset time is the difference between the anchor point time of the query hash and the matching database hash. The total number of query hash-database hash pairs that have the most popular offset time is the song's score. The song with the largest score is declared to be the correct match. No match is returned if no matching hashes are found.

Chapter 5. The Hokua Algorithm

In this chapter we present our algorithm, Hokua, which addresses the audio fingerprinting problem. Hokua uses a wavelet transform to obtain feature vectors for identification purposes. It extends the ideas of Wang's algorithm, presented in Section 4.4, to wavelets.

The algorithm has been split into two pieces, that portion which fingerprints an audio signal, and that portion which matches fingerprints against a database of known audio clips. We introduce the fingerprinting algorithm in Section 5.1, then discuss its implementation using MATLAB in Section 5.2. In Section 5.3 we discuss Hokua's matching method. Finally, we discuss methods for validating the results of matching in Section 5.4.

5.1 Fingerprinting Algorithm

We follow the steps outlined by Cano et al., found in Section 4.1. The preprocessing step converts the signal into double-precision, normalized samples, and downsamples the signal to about 5000 Hz. Also, when the signal is in stereo, the channels are averaged to obtain a mono signal.

In the framing and overlap step, the signal is broken into overlapping pieces. Each segment is $2^13 = 8192$ samples long, which allows for a fast implementation of the discrete wavelet transform. In the resampled audio, this corresponds to approximately 1.55 seconds. The frame overlap is 1/4, so the next segment starts 6144 samples after the previous segment. This overlap is to diminish the effects the use of a periodic wavelet causes at the edges of each segment.

The transform that is performed in this algorithm is a periodic orthogonal discrete Wavelet transform. The orthogonal mother wavelet we use is the 'Coiflet' wavelet. We use 10 scales, that is $J = 10$.

Every segment of the audio clip is transformed. At this point we have wavelet coefficients for all translations of each of the 10 scales. Figure 5.1 illustrates the DWT for one sample segment.

Continuing with the front end, we find the location and local energy of the largest magnitude coefficients as part of our feature extraction. For each scale we pick the largest 4 coefficients, with some constraint on the minimum distance between two peaks. For a query fingerprint, which consists of just 2 segments, this amounts to 88 peaks. As a side note, this is where the algorithm gets its name. Hokua is the Hawaiian word for the tip of a high wave.

The post-processing takes the wavelet peaks to create hashes. Each hash comes from a pair of peaks,

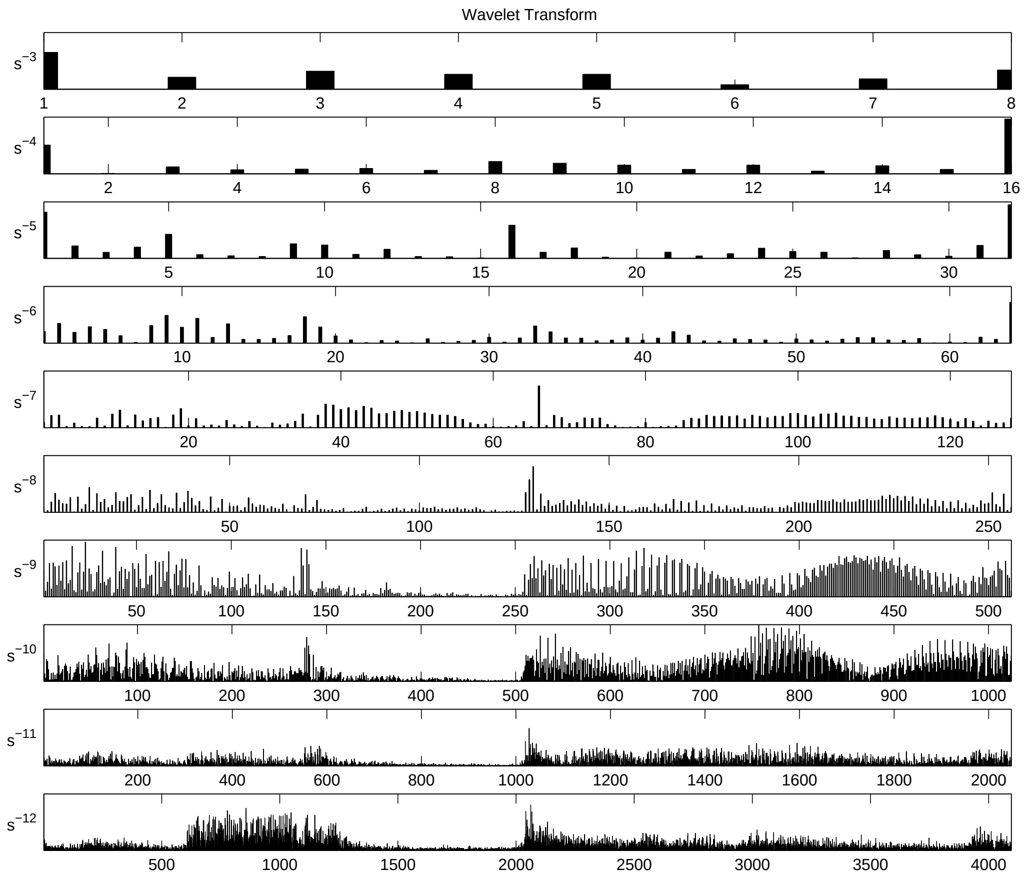


Figure 5.1: Sample Wavelet Decomposition

each pair being chosen systematically: each peak is paired with the two following peaks in its own scale, as well as with the following peaks in the scales above and below. Figure 5.2 illustrates the peaks (starred coefficients inside the box) which are paired with a given base peak (starred coefficient outside the box). For each pair, the algorithm encodes the scale of the base peak, as well as the scale offset to the paired peak, to index the hash. The algorithm then combines this with the relative change in energy (a quantized value, relative to the base peak's energy), the temporal difference between peaks, and the temporal location of the base peak. All of this information is stored as the hash for one pair.

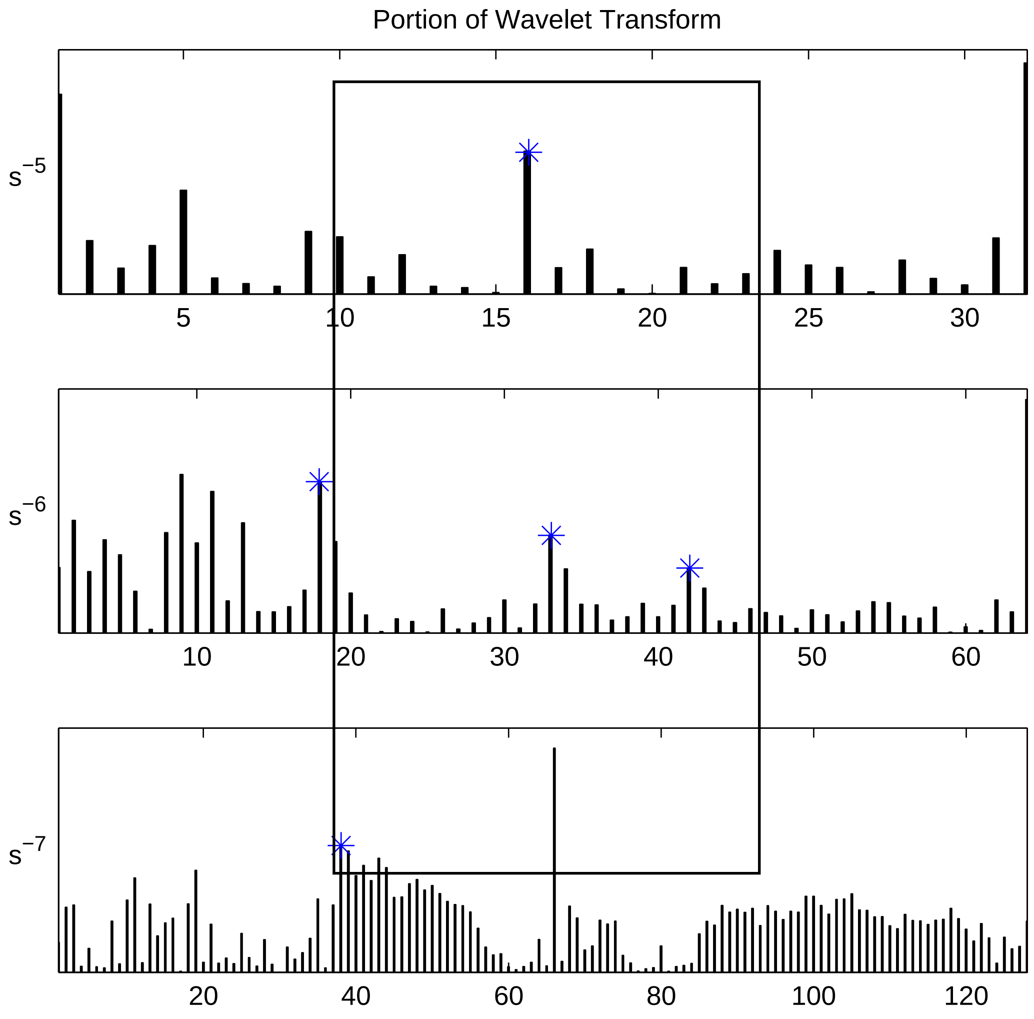


Figure 5.2: Sample of Paired Peaks

The fingerprint model collects the hashes for every pair to form the fingerprint of the audio clip. The database is created by fingerprinting the entire song. A query fingerprint consists of 2 frames, which is about 3 seconds of a query audio clip.

5.2 Implementation

The MATLAB scripts `hokuaFinger.m` and `hokuaDBfinger.m` both fingerprint an audio signal using this algorithm. The former is used to create a fingerprint for a database query, and returns about 250 hashes used in matching. The latter m-file fingerprints an entire song, which is then used to expand the database of recognizable songs. In each, the audio signal is converted using either `wavread.m` or a similar m-file `ff3read.m` to return a vector of the double-precision, normalized samples, `y`, as well as the sampling rate of the clip, `Fps`. The `Fps` is used in downsampling, so that we have a signal sampled around 5kHz. In particular, for a song sampled at the common 44.1kHz, the downsampled audio will be sampled at 5292 Hz. The sample rate, in Hz, of the downsampled signal is denoted `fs`

In the framing and overlap step, the signal is broken into overlapping pieces. The 8192-length segments are not framed by a window (as is done with STFT). However, we do overlap the segments by 1/4, in order that we may ignore the coefficients on the boundary of the segment, which are affected by the periodic condition of the wavelet transform. The overlapping results in a new segment starting every 1.16 seconds. For a query fingerprint, consisting of only two segments, the minimum length required is 2.71 seconds.

We perform a periodic orthogonal discrete Wavelet transform, using `FWT_PO.m`, a function found in `WaveLab 850`, a wavelet toolbox for MATLAB. We pass three arguments: the signal frame, the coursest level, and a vector called the quadrature mirror filter (`QMF`). The coursest level determines the number of scales we use. Here we choose 3 as our coursest level, meaning the largest scale is 2^{-3}. Since our frame is 2^{13} samples, then we have $13 - 3 = 10$ scales, that is $J = 10$. The `QMF` is used in the DWT to quickly calculate the coefficients of the wavelet transform. It is created by another `WaveLab 850` m-file, `MakeONFilter`. The `QMF` is created based on a given wavelet and any parameters associated with it. In our algorithm, we use the 'Coiflet' wavelet with 10 vanishing moments.

Using the built in `max` function, we find the locations of the 4 largest magnitude coefficients for each scale. When picking the largest coefficients, we constrain the algorithm so that peaks are separated by some minimum distance. At these locations, we sum the coefficients within an open region containing the peak. The reason for finding this local energy is to account for misalignment of wavelet translations. Also, the location that we save is not the peak coefficients index in the scale—which would be a number between 1 and 2^{13-s}, where s is the scale—but rather an absolute index value, measured from the beginning of the clip in units of the most refined scale. After doing this for each frame of the signal, we have matrices of peak locations and of peak energy.

We then pass these two matrices into `sortPeaks.m` and `encodePeaks3.m`. The former sorts the peaks so they are in increasing order of translation for each scale. The latter implements the post-processing, and returns a list of hashes.

In `encodePeaks3.m`, we begin by creating the hashes for the pairs that stay on the same scale, and then create hashes for those pairs that spread up or down a scale. Using mostly the `reshape` function and vector subtraction, we obtain a hash that contains the following information:

(i) Scale of the base peak

(ii) Difference in scale between the two peaks

(iii) Temporal location of the base peak

(iv) Difference in translation between the two peaks

(v) A quantized value that represents the change in energy.

The hashes returned by `encodePeaks3.m` constitute the Hokua fingerprint. Figure (5.3) illustrates a sample query fingerprint of a popular children's song.

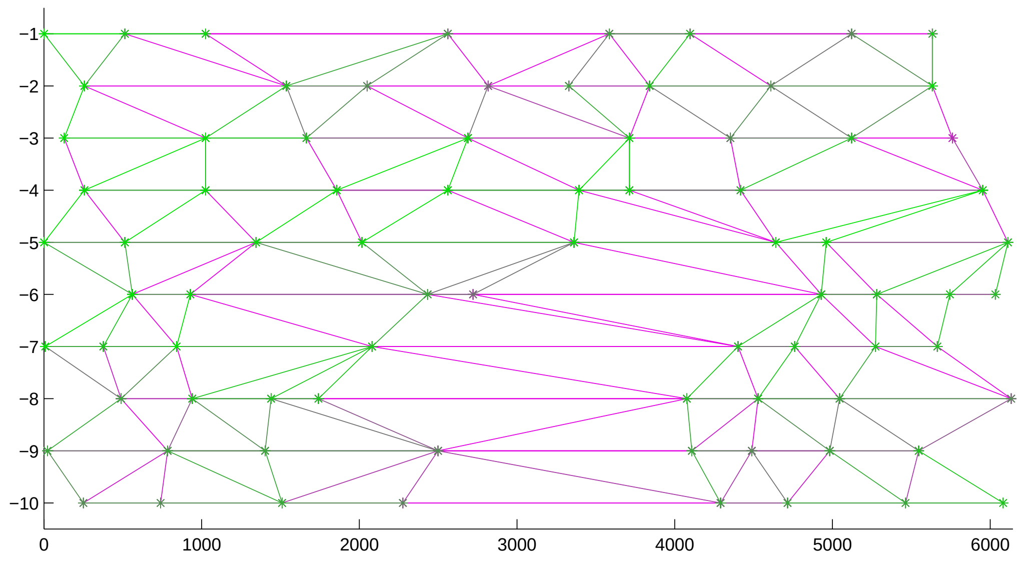


Figure 5.3: Sample Hokua Query Fingerprint

5.3 Matching

As the Hokua fingerprinter is based on the work done by Wang [17], we perform similar matching. Per the framework for matching, we fingerprint each original song. A song id is appended to the resulting hashes, which are collected in a database. This step is carried out in MATLAB using the script `hokuaExpandDB.m`. This script goes through a list of .mp3 files, performing two actions. First, the song is fingerprinted, using `hokuaDBfinger.m`. This function fingerprints the entire song as outlined in 5.2. Second, the database is extended, inserting each hash in the appropriate bins.

As the structure of the database plays an important role in the matching, we will consider it now. Since each hash contains the scale of the base peak and the scale difference between the two peaks, we use these as indexing values. Thus the hashes are stored in a 10×3 cell array, corresponding to the 10 different scales and 3 scale offset values, $(\{-1, 0, 1\})$.

The matching of a query fingerprint, Q, against the database is performed using `hokuaMatch.m`. The algorithm collects those hashes from the database that have the same base peak scale, scale offset, time offset, and energy as a hash from Q.

At this point, the collection of pulled database hits contains the time locations of the base peaks of both the query clip and the corresponding database hits, as well as the song id for each database hit. For each song in the database, the time location of the query hash is subtracted from each matching database hit. The mode is found (ie, the most common difference in time locations), and the number of query hash-database hash pairs that differ by this amount is the score given to the song. The song with the highest score is the candidate match.

5.4 Verification

We would like to make some inference on how likely the candidate match is the true identification. In this section, we will examine some ideas we pursue for ensuring that we report the true song.

While the matching hashes are found for all the scales in the general matching algorithm, we will modify this so that we use a subset of the scales. We have found that the most refined scales have more discriminating power, where as the coarsest scales will result in a dozen or so more matches for every song. In our experiments, we run the matching algorithm searching only the 2^{-4} to 2^{-10} scales. This eliminates some of the more noisy coarse level scales.

To support this choice, consider the histograms found in Figures 5.4 and 5.7. These show the histograms of a given query for both matching and non-matching songs in the database, for a variety of scales. Figures

5.4 and 5.5 consider each scale separately. In the case of the true match, we see several of the scales do not correctly identify the offset (which was approximately 87,000) as the mode. In Figures 5.6 and 5.7, we aggregate all the matching time offsets for scales s^{-i} to s^{-10} into one matrix, and then find the mode. The histogram shows heavy clustering of time offsets around the mode. Thus aggregating the offsets for multiple scales improves our performance.

Supporting the choice to focus on scales 4 to 10, consider Table 5.1. We ran the matching algorithm for the first 5 seconds of every song. This eliminates any phase effects that diminish the performance of Hokua. We fingerprint each intro clip, and then loop through the matching algorithm for each range of scales, starting with aggregating scales 1 to 10, 2 to 10, and so on until we use just scale 10. The first column of Table 5.1 reports the starting scale. The columns represent the average true song score (tss), the average proportion of the true song score to the next highest false song score (tss/hfss), the average proportion of the true song score to the number of matches found for the true song (tss/tmf),the average time to match (match (s)), and the average original rank of the true song (o. rank).

Scale	tss	tss/hfss	tss/tmf	match (s)	o. rank
1	252.5946	14.2812	0.057386	2.5407	55.9392
2	233	17.6635	0.084414	1.5381	48.0203
3	206.4527	20.8568	0.12908	1.1203	38.3986
4	179.6486	23.587	0.19627	0.80444	26.5743
5	152.6486	25.7476	0.30467	0.60135	13.2568
6	125.5541	27.62	0.46398	0.46795	4.2568
7	98.9527	27.7316	0.5814	0.42923	2.0608
8	72.7415	25.1059	0.66906	0.33899	1.7007
9	45.9592	20.5204	0.72722	0.28851	1.3197
10	19.4694	14.2602	0.78364	0.22308	1.5986

Table 5.1: Scale Results

As can be seen from Table 5.1, it would appear starting with scales 4, 5, or 6 would all produce good results. The different columns would suggest some measurements we may like to use in verifying the candidate match.

Currently, the method for verifying a candidate match is to compare the song's score with the next highest score. In general, if the difference is greater than some constant, `NOMATCH`, then we declare the song as a match. With `NOMATCH` about 5, we are confident that the identified song is the true match. However this will sometimes result in us rejecting the true song. We have found that even using `NOMATCH` as small as 1, we often times have the correct song. As we will discuss in the comparison section, Section 6, the choice depends largely on the goals of the fingerprinting system.

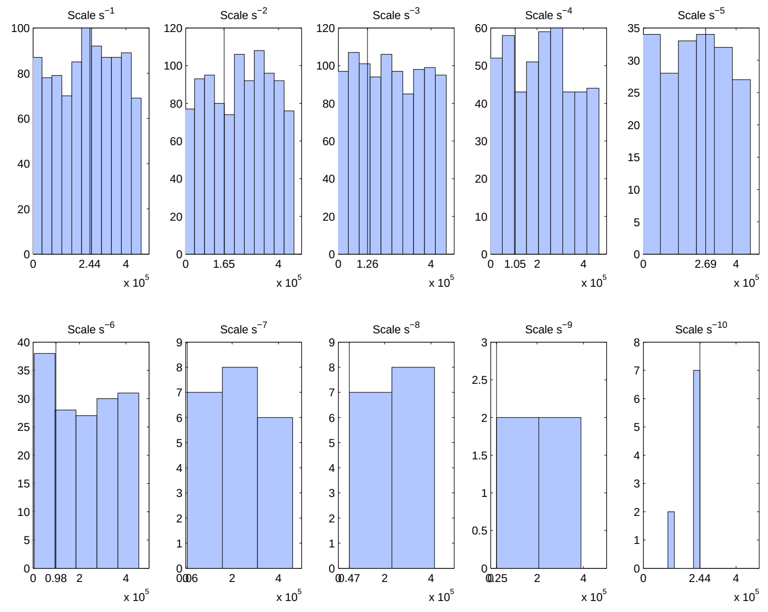


Figure 5.4: Scale Histograms for True Match to Query

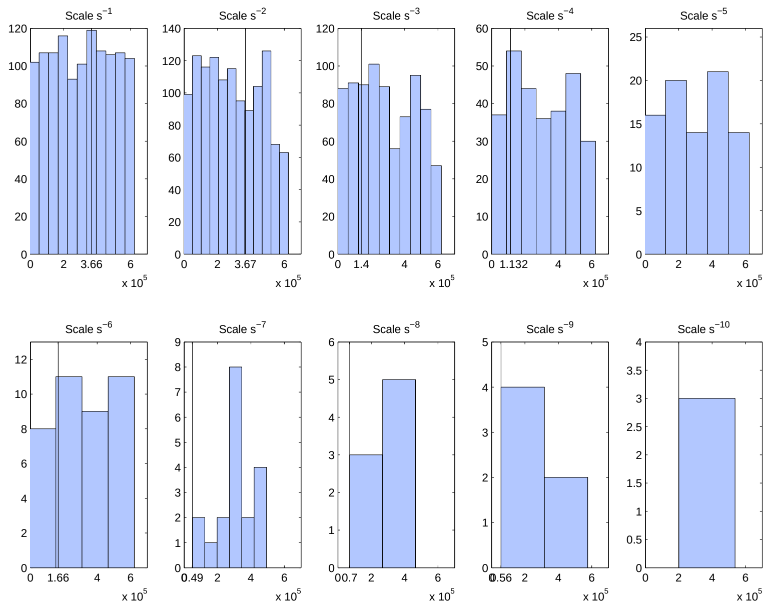


Figure 5.5: Scale Histograms for False Match to Query

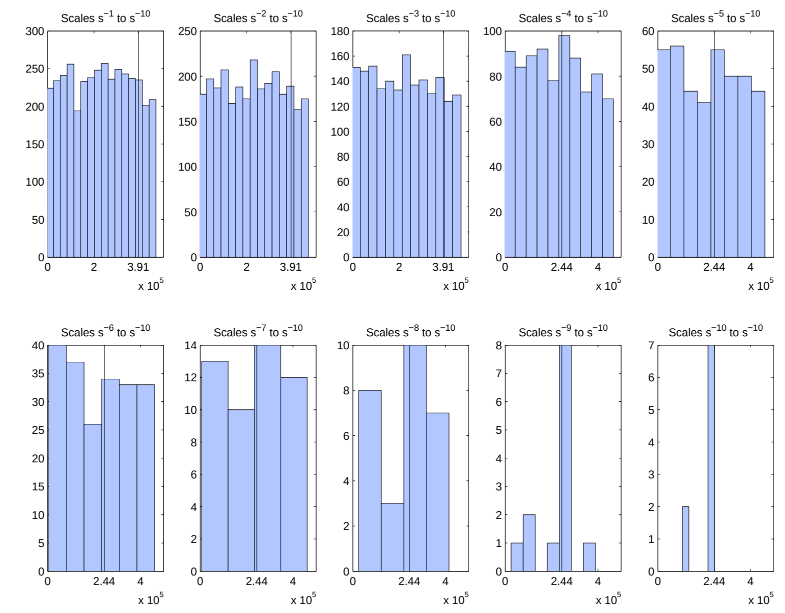


Figure 5.6: Cumulative Scale Histograms for True Match to Query

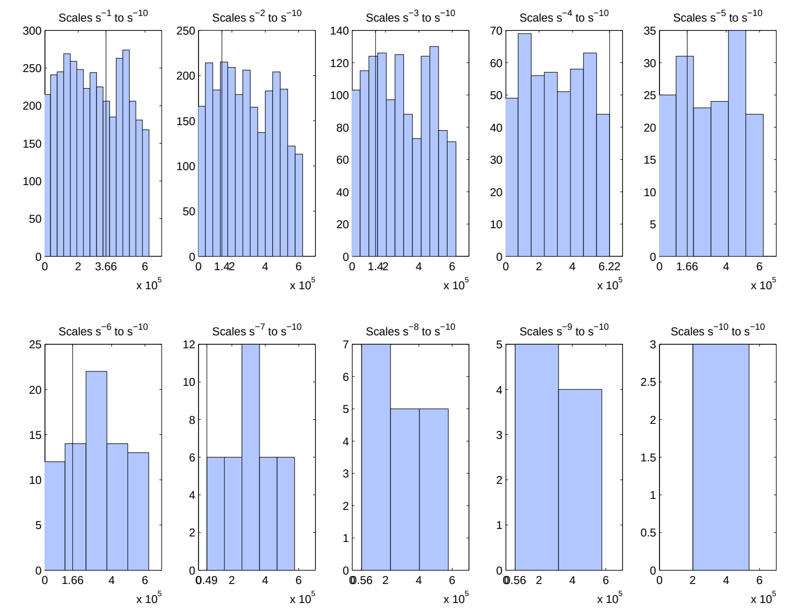


Figure 5.7: Cumulative Scale Histograms for False Match to Query

Chapter 6. Comparisons

In this chapter, we will compare three algorithms–Hokua, discussed in Chapter 5, as well as HRA and Shazam, discussed in Chapter 4. There are several aspects of a fingerprinting system that we can compare, based on various goals. As discussed by Cano et al. in [14], the aspects that every audio fingerprinting system should address are:

- Accuracy
- Reliability
- Robustness
- Granularity
- Security
- Versatility
- Scalability
- Complexity
- Fragility

We will explain each of the above requirements in their own sections. If they are areas that Hokua addresses, we will provide comparisons among the algorithms, as well of a discussion of how Hokua handles the requirement, including strengths and weaknesses.

6.1 Accuracy

The accuracy measure gives us an idea of how often a fingerprint system is correct in its identification. When we present a query from the database, the system can return three different results:

I. Correct identification

II. Incorrect identification

III. Missed identification

A type I result is clearly the best, known as a true positive. We would like this to be as high as possible. Type II results, known as false positives, occur when the system claims a query as song i when in fact it is song j. Type III results, known as false negatives, occur when the system claims the query is not in the database. For some applications, such as copyright issues, a type III result may be better tolerated than type II, that is its better that a copyright song slip by unidentified than a non-copyrighted song be incorrectly tagged as copyrighted. At other times, the roles may be reversed.

Table 6.1 illustrates experimental results for the three algorithms. We created a test set of 100 clips, each 5 seconds long, from songs in the database. We tested each algorithm against this test set, and report the percentage of each type of result.

	Type I	Type II	Type III
HRA	17	0	83
Shazam	99	1	0
Hokua	47	7	46

Table 6.1: Accuracy of Algorithms

As noted in section 4.3, the authors of HRA suggested using a cut off BER of 0.35 to distinguish if a query is in the database. As none of the BERs for the 83 type III results were under this tolerance, no songs were reported, though the algorithm always had candidates.

We fear that the poor performance of HRA has something to do with our implementation of the matching algorithm. As is probably evident, we had no part in implementing the Shazam algorithm.

6.1.1 Hokua Discussion. The low Type II percentage is due to the matching algorithm. As mentioned in Section refHokua:Inference, the candidate match needs to beat every other song's score by more than one in order for Hokua to declare that the song is not in the database. If, on the other hand, we wanted the best guess, we could remove this restriction, i.e. set `NOMATCH` to zero, and simply report the database song with the highest score. As every fingerprint has at least one matching hash with every song in the database, this second method would result in zero type III errors. As mentioned, the decision is based on the requirements of the particular application.

Further experiments have sparked some interesting modifications that can improve performance. In what we call a super fingerprint, we combine several fingerprints together to form a more accurate fingerprint. We discuss the method for combining fingerprints in section 6.4 when we discuss granularity. For now, we will detail the results of the accuracy experiment for Hokua when we use these super fingerprints. The results are shown in Table 6.2, where the first column is the number of fingerprints combined in the super fingerprint.

	Type I	Type II	Type III
1	47	7	46
2	75	0	25
3	85	0	15
4	93	0	7

Table 6.2: Hokua Accuracy with Super Fingerprints

6.2 Reliability

The goal of reliability relates closely with that of accuracy. We measure how reliable the system is at determining whether or not a given query is found in the database. As Cano et al. point out, the reliability is crucial in some applications, while in other applications, not so much. In our development we do not address this requirement, but thought it may be worthwhile to report some information on the results when each algorithm is presented with a query not in the database. We chose 5 second clips from 10 songs not found in the database. We report arithmetic means.

For HRA, the average BER was 0.46, above the suggested 0.35 tolerance for declaring a match. With shazam, the algorithm returned all type II results, claiming all the songs were present in the database. The average largest number of matching hashes was 2.9. Finally Hokua, which marked all of the test clips as being outside the database, had an average highest score of 4.9.

6.2.1 Hokua Discussion. Using different number of scales for comparisons results in varying results. For instance, using all of the scales (1 to 10), results in an average high score of 18.3, including four false positive matches. However using only scales 8 to 10, the average score is 3.4.

6.3 Robustness

The most often studied goal in the literature on audio fingerprinting is robustness. This deals with how the system handles degradation in the audio signal. Various degradations include noise addition, echo addition, resampling a clip to 22kHz and then back to 44.1kHz, volume normalizations, and mp3 compression at various rates. While we hoped to also look at two other common degradations, time shifts and pitch shifts, we did not implement these in MATLAB.

In order to measure robustness, we will examine the three algorithms against the same set of distorted audio. We will use the 100 song set from the accuracy experiment, and add the various distortions. The results can be found in Table 6.3. As was seen in the accuracy section, my implementations of HRA and Hokua performed poorly in matching. Thus we report a similarity measure between the distorted and clean clip. For each of the algorithms, we will report the average percentage of the clean fingerprint that was matched by the distorted clip.

6.3.1 Hokua Discussion. It should be noted that while the similarity is low, often the matching can occur with even as few as 10 matching hashes. Given the approximate 265 hashes per query, this is only 3.77% similarity. The rub lies in whether the clip we compared against was found in the database.

	Noise	Echo	Resample	Volume Normalization	MP3 Compression 32kb/s	MP3 Compression 64 kb/s
HRA	78.22	81.99	99.99	100.00	81.13	84.17
Shazam	15.56	42.82	100.00	100.00	2.92	11.28
Hokua	18.92	17.56	99.89	100.00	5.83	6.86

Table 6.3: Robustness of Algorithms

We also found that there is a delay of 1066 samples (approximately 24 milliseconds) when the experiments converted the .wav to 32kbps .mp3 and back to .wav. Accounting for this gave somewhat better results. The method for performing the degradations should be scrutinized.

6.4 Granularity

A fingerprinting system's granularity is measured by the fingerprint length, in seconds, required for matching a query. While it is possible to match two fingerprints with very short queries, a longer clip can improve some of the previous goals. Thus a good balance should be found between accuracy/reliability and granularity. The goal is to determine the smallest possible granularity for a given level of correctness.

Table 6.4 lists the granularity for the three algorithms. The HRA length is standard, based on 256 subfingerprints in a query fingerprint, and the frame length and overlap. For shazam, Wang ran experiments with 15-, 10-, and 5-second clips. As he showed, identification rates were better for longer clips, but the system still performed at the shorter rate. For Hokua, we decided to require at least 2 segments for a query fingerprint, which with the given segment length and overlap, results in the stated length.

	Time (s)
HRA	3.48
Shazam	5.00
Hokua	2.71

Table 6.4: Granularity of Algorithms

6.4.1 Hokua Discussion. As mentioned above, Hokua uses only 2 segments for its fingerprint. However, it is interesting to note what happens as we change the number of segments with which to fingerprint. Table 6.5 gives results of using different number of segments in the fingerprinter.

The method for fingerprinting with a different number of segments went as follows. The 1-segment fingerprint takes a segment from the end of the 5 second clip. The 2-segment fingerprint was taken from the beginning of the 5 second clip. The 3-segment fingerprint was the combination of the 1- and 2-segment fingerprints, adding a fixed offset of 9134 to each of the time locations of the 1-segment fingerprint. Finally the 4-segment fingerprint combined the 2-segment fingerprint with another 2-segment fingerprint taken from the end of the 5 second clip. This second 2-segment fingerprint had a fixed offset of 6062 added to each time location.

The fixed offsets serve the purpose of aligning the combined fingerprints temporally with the original 2 segment fingerprint. To find these offsets, we first compute the resampled signal length–a 5 second clip is 26460 samples long. For the 1 segment fingerprint, we need to start the wavelet transform 8192 samples before the end of the clip, at sample 18268. Since our transformed signal is indexed by the most refined scale, which gives 4096 coefficients per 8192 samples, than there is one coefficient for every two samples, and so the first coefficient of the 1-segment fingerprint is at $18268/2 = 9134$. The fixed offset for the 2-segment fingerprint is found similarly. We need the samples starting $7*8192/4$ from the end, that is at sample 12124. Division by two yields the offset, 6062. This idea can be used to combine multiple fingerprints.

Num. of Segments	Average Hashes	Type I Accuracy %	Type II Accuracy %	Type III Accuracy %	Granularity
1	111.80	30	5	65	1.5480
2	263.73	47	7	46	2.7090
3	375.53	64	4	32	5.0000
4	527.53	82	3	15	5.0000

Table 6.5: Hokua Granularity

6.5 Security

Security measures how well the audio fingerprinting system defends against concerted attacks. Often times these malicious attacks come in the form of slight distortions which result in perceptually little or no change, but render the system unable to give a good match. We reference [26], which studies various attacks on the security of the YouTube audio fingerprinting system. Since we studied several distortions in the Robustness section which could be used as attacks, we will not do a comparison of how each algorithm handles the security requirement.

6.6 Versatility

The versatility of a system measures how well the database can be used to identify audio clips that are in different audio formats. While we looked at compression rates in robustness, we will look at how well each algorithm performs with audio in different formats.

To examine each of the algorithms, we took an .m4a file, and created several copies in the following formats: .au, .ac3, .mp2, .mp3, .wav, and .wma. The results are found in Table 6.6. One thing we note is the similarity between several formats, leading to the assumption that the encoder/decoder pair of the program we used is the same. Apparently .au, .m4a, and .wav are all read in the same. Similarly .mp3 and .wma are read in the same.

	.ac3	.au	.m4a	.mp2	.mp3	.wav	.wma
HRA	79.30	85.29	85.29	87.88	91.37	85.29	91.37
Shazam	9.68	34.19	34.19	14.19	49.03	34.19	49.03
Hokua	3.80	3.80	3.80	2.28	32.31	3.80	32.31

Table 6.6: Versatility of Algorithms

6.6.1 Hokua Discussion. One thing that should be noted with this measure is the format problems present in the database. Due to the unwieldy size of 16-bit PCM audio (the .wav files), most songs are compressed in some other format. In our database, all the songs were originally .m4a files. However MATLAB only provides support for reading in .wav and .au files (there is a SIMULINK block in the signal processing library entitled *From Multimedia File* that can, however we were not able to incorporate it). We used a free program [25] to convert the .m4a file to an .mp3 file. From here we use either `MP3read.m` or `ff3read.m` to read in the .mp3 file, which involves converting the file to a temporary .wav file, and then reading this temporary file using `wavread.m`.

Since all of the sound files are manipulated versions of the .m4a files and must be read into MATLAB as .wav files, the results in Table 6.6 are not surprising.

6.7 Scalability

Scalability measures how well an audio fingerprinting system handles two concerns: larger databases and more identification queries. One problem that occurs as the database grows is that distances between a song and its nearest neighbor will decrease. As we are only looking at a small database, this concern is not addressed.

We likewise do not address the second concern. It is most often a concern in a server implementation of the audio fingerprinting system. As more queries are made, the audio fingerprinting system may become inundated. As we are merely implementing a MATLAB prototype, we only handle one identification at a time.

6.8 Complexity

The complexity of a system measures the computational requirements of creating and storing the database and query fingerprints, as well as in performing matching. We report complexity by listing the size of the database and query, as well as the time to create or match fingerprints. The goal is for a compact database to save on storage, and a compact fingerprint in applications where the query is made at a remote server. Also, the time to create a database is a measure of the difficulty in updating the database.

Table 6.7 compares the three algorithms in terms of complexity described above. Once again, this is for a database of 150 songs. The size of the database is the MATLAB compression of the variables, created using the `save` command.

	Size (B)		Time (s)		
	Database	Query	Database	Query	Matching
HRA	16578560	8192	9831.325	0.1564	1.1363
Shazam	6236160	1740	3247.487	0.1520	0.0216
Hokua	13816832	1961	2887.159	0.0843	0.7212

Table 6.7: Complexity of Algorithms

6.8.1 Hokua Discussion. The size of both the Hokua database and query are based on utilizing information about potential values for the different pieces of the hash. For instance, we store the energy change (since it is a quantized value between -4 and 4) as an `int8`, where as the time location is stored as an `uint32`, since it is always positive and for a 4 minute song can be nearly 750,000.

We point out the speed with which the database and the query are generated. Hokua's performance over the other algorithms is due to the fact that Hokua only computes 2 DWTs, while HRA computes 256 FFTs and Shazam computes 155 FFTs.

6.9 Fragility

Fragility measures the ability of the audio fingerprinting system to detect when audio content has been altered. While important for some applications, we do not consider it here.

Chapter 7. Conclusion

In this final chapter, we make concluding remarks of the Hokua algorithm, as well as outline aspects of Hokua that could be studied further for improving performance.

We have proposed an algorithm, Hokua, that uses the wavelet transform in order to identify audio signals. Using several measures set forth by Cano et al., we compared its performance with two other algorithms, the Highly Robust Algorithm developed by Haitsma et al., and the precursor to Shazam, developed by Wang.

While Hokua did not compare favorably with HRA in terms of Robustness, nor with Shazam in terms of Accuracy, we did find that Hokua is certainly capable of matching, and with modifications or further study has great potential for improved performance in audio fingerprinting applications.

7.1 Future Work

As we have seen from the comparison chapter, the Hokua algorithm has considerable potential in audio fingerprinting. However, there are quite a few of parameters that may be tweaked to potentially obtain better accuracy results.

One of the first areas that should be analyzed is the choice of mother wavelet. The decision to use the 'coiflet' wavelet with 10 vanishing moments was completely arbitrary. With the large amount of work done in wavelet transforms, there are a considerable number of mother wavelets that could be used. There is also the possibility of designing a mother wavelet specifically to work well with audio signals.

Further work could also examine a variety of parameters. The number of segments used as a query fingerprint, as was briefly examined in the Hokua discussion of granularity in chapter 6, could be altered. Using the fixed offset idea, we may find that segments need not be overlapping.

Another fingerprint parameter is the number of peaks to find per segment. Currently we only find the largest four, however increasing this may yield better results. A related issue is how close we allow peaks to be. Currently the neighborhood is the same width for each scale, basically one sixth of the segment. Thus we could only find about 6 peaks before the constraint that peaks be a certain distance apart would eliminate any more options. If we decreased this neighborhood as we decrease scale size, we would allow more peaks, especially on the scales that are more discriminatory.

A similar fingerprint parameter that could be adjusted are those affecting pairing. For each base peak, we could pair it with more peaks at the same scale, more peaks in neighboring scales, or even push out to further scales (i.e. pairing a base peak with a peak 2 or 3 scales above it). While this will increase complexity, it may improve accuracy.

With the matching algorithm, the `NOMATCH` parameter can be changed to get different types of accuracy results. While the rates of false positives and false negatives will be dependent on the application, the settings could be determined that minimize one or the other.

Another matching parameter worth considering is which scales should be used for matching. The tradeoff between having more hashes with which to match and having more spurious matches to other songs can be examined. One thing that relates to this is in query fingerprint creation. If we chose to only use a subset of scales to match, we can change the length of our fingerprint (4096 samples as opposed to 8192 samples), change the coarsest level, etc., based on what subset we are interested in for matching.

Appendix A. Hokua MATLAB Code

This appendix contains the MATLAB code for the Hokua algorithm. The m-files contained herein are not sufficient for Hokua. In addition to many built-in MATLAB files, the code requires `WAVELAB 850`, a wavelet toolbox for MATLAB. It is available at http://www-stat.stanford.edu/w̃avelab/.

A.1 hokuaExpandDB.m

This section contains the code for creating or extending a database.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   hokuaExpandDB.m
%   PURPOSE:
%     hokuaExpandDB.m creates or extends the hokua
%     database SDB.
%   INPUT:
%     none (requires list of audio songs, wavlist.dat.
%           See makewavlist.py for script.)
%   OUTPUT:
%     none (note that it clears the workspace)
%       Created 2009, Steven Scott Lutz
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Reads a list of files to be used in extending or creating the database.
[wavefile titlex artist album] =...
    textread('wavlist.dat','%s %s %s %s','delimiter','\t');
```

```
startclean = 0;     % Flag: 0-extend database, 1-create database

%#ok<*UNRCH>

if startclean
    stclean = input('Starting a clean database--Are you sure?   ','s')
    if strcmpi(stclean,'yes')
        disp('Start anew')
    else
        error('Please set startclean = 0, and restart hokuaExpandDB.m')
    end
    SI = 0;                   % The number of songs already in the database
else
    load hokuaSDB.mat         % Previously created database
    SI = length(METADATA);    % The number of songs already in the database
end
niter = length(wavefile);
ti = zeros(niter,1);
if startclean
    METADATA = cell(niter,1); % Create database of song metadata
    SDB = cell(10,3);         % Create database of song hashes
else
    METADATA = [METADATA;cell(niter,1)]; % Extend database of song metadata
end
% Iterate over songs in wavlist.dat
for ind = 1:niter
    tic;
    disp(['Fingerprinting ' char(titlex(ind+SI))])
    fing = hokuaDBfinger(wavefile{ind+SI});      % Fingerprint current song
    % Insert current song's metadata
    METADATA{SI+ind,1} = struct('Title',titlex(ind+SI),...
        'Artist',artist(ind+SI),'Album',album(ind+SI));
    disp('Extending database')

    fing = sortrows(fing);

    % Iterate over base scales

    for Iint = 1:10

        tfing1 = fing(fing(:,1)==Iint,2:5);  % Hashes with scale Iint

        % Iterate over scale difference

        for Jint = 1:3

            tfing2 = tfing1(tfing1(:,1)==Jint-2,2:4); % Hashes w/ diff Jint

            ltf2 = length(tfing2);           % Number of hashes to insert

            % Insert hashes in the current bin

            SDB{Iint,Jint} = [SDB{Iint,Jint};tfing2 (ind+SI)*ones(ltf2,1)];

        end

    end

    ti(ind) = toc;      % Time to obtain and insert current song's hashes

end

hokua_database_add_time = sum(ti)

average_time = mean(ti)

longest_time = max(ti)


% This sorts the SDB matrix in order of increasing distances

for Iint = 1:10

    for Jint = 1:3

        SDB{Iint,Jint} = sortrows(SDB{Iint,Jint});

    end

end


save hokuaSDB.mat SDB METADATA

clear
```

A.2 hokuaDBfinger.m

This section contains the code for computing the Hokua fingerprint of an entire song.

```
function fing = hokuaDBfinger(wavefile)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   fing = hokuaDBfinger(wavefile)
%   PURPOSE:
%     hokuaDBfinger.m returns the fingerprint of an
%     entire song, used in extending the database.
%   INPUT:
%     wavefile:  filename of song to be fingerprinted
%   OUTPUT:
%     fing:      fingerprint of wavefile
%       Created 2009, Steven Scott Lutz
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% modifiable parameters
N = 8192;                         % Signal length
CoarseLevel = 3;                  % Coarsest level of the DWT
numdyads = log2(N)-CoarseLevel;   % Number of scales
qmf = MakeONFilter('Coiflet',5);  % Orthonormal wavelet: Coiflet-30
numpeaks = 4;                     % Number of peaks per segment per scale
leveldeep = 2;                    % Number of pairs to form per scale
levelspread = 1;                  % Number of scales from base scale to use
                                  % in forming pairs


% Gather information about wavefile,
% obtains signal, y, and sampling rate, Fps
wvin = mmfileinfo(wavefile);
if strcmp(wvin.Audio.Format,'PCM')              % .wav file
    if wvin.Duration > 250
        [y Fps] = wavread(wavefile,44100*250);
    else
        [y Fps] = wavread(wavefile);
    end
    if wvin.Audio.NumberOfChannels == 2
        y = (y(:,1)+y(:,2))/2;                  % Average Stereo into Mono
    end
elseif strcmp(wvin.Audio.Format,'MPEGLAYER3')   % .mp3 file
%     if wvin.Duration > 250
%         sampnum = 250*44100;
%     else
%         sampnum = 0;
%     end
    [y Fps] = ff3read(wavefile,0,1); %sampnum instead of 0
else
    error('Only .mp3 and .wav files are currently supported')
end

% Obtain parameters for upsampling and downsampling
if Fps == 44100                 % Most common sampling rate
    p = 3; q = 25;
elseif Fps == 48000
    p = 5;q = 48;
else
    error('Currently only music sampled at 44.1 or 48 kHz are supported')
end

ymonores = resample(y,p,q);     % Resampled audio signal
leny = length(ymonores);        % Length of resampled audio

maxint = floor(4*leny/(3*N))-1; % Maximum number of segments from signal


% Initializations
Xwt = zeros(N,maxint);                          % Wavelet transform of y
```

```
peakTS = zeros(numdyads,numpeaks*maxint);       % Location of peaks
peakE = zeros(numdyads,numpeaks*maxint);        % Local energy of peaks


% Compute DWT
for Iint = 1:maxint
    inte = (Iint-1)*3*N/4;
    Xwt(:,Iint) = abs(FWT_PO(ymonores(1+inte:N+inte),3,qmf));
end


% Location and local energy of Coarsest dyad, needs separate handling.
coeff = 1;
startindex = 2^(CoarseLevel);       % Number of coefficients in dyad
multx = 2^(numdyads-1);             % Multiplier to get absolute time index
                                    % from scale dependent time index.
eyeX = 0:3*startindex/4:3*startindex*maxint/4-1;
                                    % Starting absolute index for each
                                    % segment.
ddd=dyad(CoarseLevel);              % Index for coarsest dyad in DWT
ddd = ddd(coeff+1:end-coeff);       % Removes edge coefficients to ignore
                                    % boundary effects.
tempXwt = Xwt(ddd,:);               % Temporary DWT of signal, used in
                                    % constraining distance between peaks.
unosos = ones(1,maxint);            % Preallocation of a ones column vector


% Iterate over the number of peaks per segment
for Jint = 1:numpeaks
    [tempMax, tempLocal] = max(tempXwt);        % Find peak locations
    indexer = Jint:numpeaks:maxint*numpeaks;    % Save room for numpeaks
                                                % per segment
    % Stores the absolute time location of peaks
    peakTS(1,indexer) = (tempLocal-unosos+eyeX).*multx+1;
    % Iterate over the number of segments
    for Kint = 1:maxint
        begind = tempLocal(Kint);   % Location of peak

        % Calculate square of largest remaining coefficient
        peakE(1,Jint+(Kint-1)*numpeaks) = Xwt(startindex+1+begind,Kint).^2;
        tempXwt(begind,Kint)=0;     % Set to 0 so we find next largest peak
    end
end

% Location and local energy of remaining dyads' peaks
for Iint = 2:numdyads
    coeff = 2^(Iint-1);             % Number of coeff. in current dyad
                                    % that constitute neighborhood.

    startindex = 2^(CoarseLevel+Iint-1);% Number of coefficients in dyad
    multx = 2^(numdyads-Iint);      % Multiplier to get absolute time index
    eyeX = 0:3*startindex/4:3*startindex*maxint/4-1;
                                    % Starting absolute index for each
                                    % segment.
    ddd=dyad(CoarseLevel+Iint-1);   % Index for coarsest dyad in DWT
    ddd = ddd(coeff+1:end-coeff);   % Removes edge coefficients to ignore
                                    % boundary effects.
    tempXwt = Xwt(ddd,:);           % Temporary DWT of signal, used in
                                    % constraining distance between peaks.
    lenddd = length(ddd);

    % Iterate over the number of peaks per segment
    for Jint = 1:numpeaks
        [tempMax, tempLocal] = max(tempXwt);       % Find peak locations
        indexer = Jint:numpeaks:maxint*numpeaks;   % Save room for numpeaks
                                                   % per segment
        % Stores the absolute time location of peaks
        peakTS(Iint,indexer) = (tempLocal-unosos+eyeX).*multx+1;
        % Iterate over the number of segments
        for Kint = 1:maxint
            begind = tempLocal(Kint)-coeff/2;   % Begin of peak nbhd
            endind = tempLocal(Kint)+coeff/2;   % End of peak neighborhood
            nbhd = startindex+coeff+begind:startindex+coeff+endind;
                                                % Location of nbhd in
                                                % original DWT of signal.

            % Calculate energy in nbhd around peak
            peakE(Iint,Jint+(Kint-1)*numpeaks) = sum(Xwt(nbhd,Kint).^2);

            % Used to 0 coefficients of tempXwt to find next largest peak.
            if begind<=0
                begind = 1;
            end

            if endind >= lenddd
                endind = lenddd;
            end
            numzeros = endind-begind+1;
            tempXwt(begind:endind,Kint)=zeros(numzeros,1);
        end
    end
end



%------SORT------
[peakTS, peakE] = sortPeaks(peakTS, peakE);

%-----ENCODE-----
```

```
fing = encodePeaks(peakTS, peakE, leveldeep, levelspread);
```

A.3 hokuaFinger.m

This section contains the code for computing the Hokua fingerprint of a query clip.

```
function fing = hokuaFinger(varargin)



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   fing = hokuaFinger(wavefile)
%     --OR-- fing = hokuaFinger(y,Fps,printint)
%   PURPOSE:
%     hokuaFinger.m returns the fingerprint of a
%     query clip, used in querying the database.
%   INPUT:
%     wavefile:  filename of song to be fingerprinted
%     y:         signal (previously read in or recorded)
%     Fps:       sample rate of y
%     printint:  (Optional) Print flag, 0-skip, 1-create figure
%   OUTPUT:
%     fing:      fingerprint of query
%       Created 2009, Steven Scott Lutz
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% modifiable parameters
N = 8192;                          % Signal length
CoarseLevel = 3;                   % Coarsest level of the DWT
numdyads = log2(N)-CoarseLevel;    % Number of scales
qmf = MakeONFilter('Coiflet',5);   % Orthonormal wavelet: Coiflet-30
finlen = 2;                        % Number of segments per query
numpeaks = 4;                      % Number of peaks per segment per scale
leveldeep = 2;                     % Number of pairs to form per scale
```

```
levelspread = 1;                 % Number of scales from base scale to use
                                 % in forming pairs




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Pre-processing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

printint = 0;
if nargin == 1          % Only wavefile was passed in
    % Gather information about wavefile,
    % obtains signal, y, and sampling rate, Fps
    wvin = mmfileinfo(varargin{1});
    if strcmp(wvin.Audio.Format,'PCM')
        if wvin.Duration > 250
            [y Fps] = wavread(varargin{1},44100*250);
        else
            [y Fps] = wavread(varargin{1});
        end
        if wvin.Audio.NumberOfChannels == 2
            y = (y(:,1)+y(:,2))/2;              % Average Stereo into Mono
        end
    elseif strcmp(wvin.Audio.Format,'MPEGLAYER3')
        if wvin.Duration > 250
            sampnum = 250*44100;
        else
            sampnum = 0;
        end
        [y Fps] = ff3read(varargin{1},sampnum,1);
    else
        error('Only .mp3 and .wav files are currently supported')
    end
elseif nargin == 2      % Only y and Fps were passed in
    y = varargin{1};                % Signal
    Fps = varargin{2};              % Sample rate
    if size(y,2) == 2
        y = mean(y,2);              % Average Stereo into Mono
    end
elseif nargin == 3      % y, Fps, and printint were passed in
    y = varargin{1};                % Signal
    Fps = varargin{2};              % Sample rate
    printint = varargin{3};         % Print flag
%     disp(['Printing query to subplot ' num2str(printint) ' of 3.'])
    if size(y,2) == 2
        y = (y(:,1)+y(:,2))/2;      % Average Stereo into Mono
    end
else
    error('WPfinger.m takes either 1, 2, or 3 arguments')
end

% Obtain parameters for upsampling and downsampling
if Fps == 44100                 % Most common sampling rate
    p = 3; q = 25;
elseif Fps == 48000
    p = 5;q = 48;
else
    error('Currently only music sampled at 44.1 or 48 kHz are supported')
end

ymonores = resample(y,p,q);     % Resampled audio signal
leny = length(ymonores);        % Length of resampled audio

% The minimum length of audio clip needed to make up
% a query fingerprint, in seconds, is
%                  ((3*finlen+1)*N/4)/(p*Fps/q)


if leny >= (3*finlen+1)*N/4
    base = 0;                  % Used in shifting where the fingerprint
                               % comes from, normally this is zero.
else
    error('Sample not long enough to provide a fingerprint')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Framing, Overlap and Transform
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Initializations
Xwt = zeros(N,finlen);                          % Wavelet transform of y
peakE = zeros(numdyads,numpeaks*finlen);        % Location of peaks
peakTS = zeros(numdyads,numpeaks*finlen);       % Local energy of peaks



% Compute DWT
for Iint = 1:finlen
    inte = (Iint-1)*3*N/4;
    Xwt(:,Iint) = abs(FWT_PO(ymonores((1+inte+base):(N+inte+base)),3,qmf));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Feature Extraction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Location and local energy of Coarsest dyad, needs separate handling.
coeff = 1;
```

```
startindex = 2^(CoarseLevel);       % Number of coefficients in dyad
multx = 2^(numdyads-1);             % Multiplier to get absolute time index
                                    % from scale dependent time index.
eyeX = 0:3*startindex/4:3*startindex*finlen/4-1;
                                    % Starting absolute index for each
                                    % segment.
ddd=dyad(CoarseLevel);              % Index for coarsest dyad in DWT
ddd = ddd(coeff+1:end-coeff);       % Removes edge coefficients to ignore
                                    % boundary effects.
tempXwt = Xwt(ddd,:);               % Temporary DWT of signal, used in
                                    % constraining distance between peaks.

% Iterate over the number of peaks per segment
for Jint = 1:numpeaks
    [tempMax, tempLocal] = max(tempXwt);        % Find peak locations
    indexer = Jint:numpeaks:finlen*numpeaks;    % Save room for numpeaks
                                                % per segment
    % Stores the absolute time location of peaks
    peakTS(1,indexer) = (tempLocal-ones(1,finlen)+eyeX).*multx+1;
    % Iterate over the number of segments
    for Kint = 1:finlen
        begind = tempLocal(Kint);   % Location of peak

        % Calculate square of largest remaining coefficient
        peakE(1,Jint+(Kint-1)*numpeaks) = Xwt(startindex+coeff+begind,Kint).^2;
        tempXwt(begind,Kint)=0;     % Set to 0 so we find next largest peak
    end
end

% Location and local energy of remaining dyads' peaks
for Iint = 2:numdyads
    coeff = 2^(Iint-1);             % Number of coeff. in current dyad
```

```
                                % that constitute neighborhood.

startindex = 2^(CoarseLevel+Iint-1);% Number of coefficients in dyad
multx = 2^(numdyads-Iint);      % Multiplier to get absolute time index
eyeX = 0:3*startindex/4:3*startindex*finlen/4-1;
                                % Starting absolute index for each
                                % segment.
ddd=dyad(CoarseLevel+Iint-1);   % Index for coarsest dyad in DWT
ddd = ddd(coeff+1:end-coeff);   % Removes edge coefficients to ignore
                                % boundary effects.
tempXwt = Xwt(ddd,:);           % Temporary DWT of signal, used in
                                % constraining distance between peaks.

lenddd = length(ddd);

% Iterate over the number of peaks per segment
for Jint = 1:numpeaks
    [tempMax, tempLocal] = max(tempXwt);       % Find peak locations
    indexer = Jint:numpeaks:finlen*numpeaks;   % Save room for numpeaks
                                               % per segment
    % Stores the absolute time location of peaks
    peakTS(Iint,indexer) = (tempLocal-ones(1,finlen)+eyeX).*multx+1;
    % Iterate over the number of segments
    for Kint = 1:finlen
        begind = tempLocal(Kint)-coeff/2;   % Begin of peak nbhd
        endind = tempLocal(Kint)+coeff/2;   % End of peak neighborhood
        nbhd = startindex+coeff+begind:startindex+coeff+endind;
                                            % Location of nbhd in
                                            % original DWT of signal.

        % Calculate energy in nbhd around peak
        peakE(Iint,Jint+(Kint-1)*numpeaks) = sum(Xwt(nbhd,Kint).^2);
```

```
            % Used to 0 coefficients of tempXwt to find next largest peak.
            if begind<=0
                begind = 1;
            end

            if endind >= lenddd
                endind = lenddd;
            end
            numzeros = endind-begind+1;
            tempXwt(begind:endind,Kint)=zeros(numzeros,1);
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Post-processing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%------SORT------
[peakTS, peakE] = sortPeaks(peakTS, peakE);

%-----ENCODE-----

fing = encodePeaks(peakTS, peakE, leveldeep, levelspread);

% Creates a figure of the query fingerprint
if printint~= 0
    lfin = length(fing);
    YVAL = zeros(lfin,2);
    XVAL = [fing(:,5) fing(:,5)+fing(:,4)];
    CVAL = fing(:,3);
    for Jint = 1:lfin
        basy = .05+.1*(10-fing(Jint,1));
        YVAL(Jint,:) = [basy,basy - fing(Jint,2)*.1];
    end
    hold on
    for Kint = -4:4
        indexer = CVAL == Kint;
        plot(XVAL(indexer,:)',YVAL(indexer,:)','Marker','*',...
            'Color',[(4-Kint)/9 (4+Kint)/9 (4-Kint)/9])
%         plot(XVAL(indexer,:)',YVAL(indexer,:)','Marker','*',...
%             'Color',[0 0 0]) % ALL BLACK
    end
    colormap('default')
    set(gca,'xlim',[0 6144],'ylim',[0 1],'ytick',0.05:.1:.95,'yticklabel',-10:-1);
end
```

A.4 hokuaMatch.m

This section contains the code for matching a query fingerprint against the database.

```
function [result,varargout] = hokuaMatch(fing,s2s,SDB,METADATA,TSN)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   [result,varargout] = hokuaMatch(fing,s2s,SDB,METADATA,TSN)
%   PURPOSE:
%     hokuaMatch.m returns the results of matching
%     a query fingerprint against the database SDB.
%   INPUT:
%     fing:      Query Fingerprint, obtained from hokuaFinger
%     s2s:       Subset of scales to match (e.g. 1:10(all), 4:10, 6:10)
%     SDB:       (Optional) Database of song hashes
%     METADATA:  (Optional) Database of song metadata
%     TSN:       (Optional) True song id, for distribution of offset data
%   OUTPUT:
%     result:    Result of matching
%     songscore: (Optional) Table of values for each song, including:
%         Column1:  Song id
%         Column2:  Song score,
%         Column3:  Original number of matching hashes,
%         Column4:  Most popular offset.
%     DistData:  (Optional) Distribution of offset data, requires TSN
%       Created 2009, Steven Scott Lutz
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


if nargin < 4
    load hokuaSDB.mat    % Load SDB and METADATA if not passed in
    TSN = 0;             % Set Flag so DistData is not computed
end
if nargin < 6
    TSN=0;               % Set Flag so DistData is not computed
end


% Set parameters
nbins = length(METADATA);   % Number of songs in database
TopX = nbins;               % Number of songs that are scored

NOMATCH=1;                  % Verification parameter, minimum
                            % difference between songs to declare
                            % match. Set to 0 to pass candidate match.

TOL = 500;                  % Grouping parameter, adds number of
                            % query hash-database hash pairs to
                            % song score that offset is within
                            % TOL/2 of the modal offset.
% Initializations
songnum = zeros(TopX,1);
numhash = zeros(TopX,1);
songscore(1:TopX,5) = (1:TopX)';
maxtobe = 2000000;              % Large constant for preallocation
tobebinned=zeros(maxtobe,3);    % Preallocated to hold all matching hashes
                                % Col1 is QH base time
                                % Col2 is corresponding DBHs base time
                                % Col3 is corresponding DBHs song id
unos = ones(maxtobe,1);         % Large vector of ones
start=1;                        % Used for preallocation
                                % Rows of fing are Query Hashes (QHs)
                                % SDB contains Database Hashes (DBHs)

% Iterate over base scales
for Iint = s2s
    tfing1 = fing(fing(:,1)==Iint,2:5);  % Pulls all QHs with scale Iint

    % Iterate over scale difference
    for Jint = 1:3
        tfing2 = tfing1(tfing1(:,1)==Jint-2,2:4);
                                        % Pulls all QHs w/ difference Jint
        ltf2 = size(tfing2,1);          % Number of QHs in current bin
        thashes = SDB{Iint,Jint};       % Finds all DBHs in current bin

        % Iterate over all QHs in current bin
        for Kint = 1:ltf2
            tf2t=tfing2(Kint,2);      % Time Difference of current QH
            tf2e=tfing2(Kint,1);      % Rel. Energy Change of current QH
            tf2=tfing2(Kint,3);       % Base Time of current QH

            matchhashes2 = thashes(thashes(:,2)==tf2t,[1,3,4]);
```

```
                                    % Find all DBHs that match current QH
                                    % on Time Difference.

            matchhashes = matchhashes2(matchhashes2(:,1)==tf2e,2:3);
                                    % Find all DBHs that match current QH
                                    % on Rel. Energy Change.

%             matchhashes = matchhashes2(abs(matchhashes2(:,1)-tf2e)<=1,2:3);
                                    % Find all DBHs that differ by atmost
                                    % one with the current QH Rel. Energy

            lm=size(matchhashes,1);   % Num. of matching DBHs to current QH

            fin = start+lm-1;         % Used for preallocation
            if fin > maxtobe          % Preallocate more for matching DBHs
                tobebinned = [tobebinned;zeros(2000000,3)]; %#ok<AGROW>
                maxtobe = maxtobe+2000000;
            end
            tobebinned(start:fin,2:3) = matchhashes;
            tobebinned(start:fin,1) = tf2*unos(1:lm,1);
            start = fin+1;
        end
    end
end
tobebinned = tobebinned(1:fin,:);      % Chops off extra preallocated zeros
freqcount = hist(tobebinned(:,3),1:nbins);  % Num. of matching hashes
                                            % for each song

% Iterate over songs with the TopX most matches
for Iint = 1:TopX
    [numhash(Iint),songnum(Iint)] = max(freqcount);
                              % songnum(Iint) is current song id
                              % numhash(Iint) is number of matching hashes
    csg = songnum(Iint);      % current song id
    bts = tobebinned(tobebinned(:,3)==csg,1:2);
                              % Base times of QHs and DBHs with current song
    offsets = bts(:,2)-bts(:,1);
                              % Time offset (time of DBH - time of QH)

    % This calculates DistData for TSN
    if csg == TSN || csg == TSN+1
        lo = length(offsets);
        DistData(1:lo+1,mod(csg,TSN)+1) = [lo;offsets]; %#ok<AGROW>
    end

    md = mode(offsets);      % Most popular offset time for current song
    edg = [md-TOL/2,md+TOL/2];
    tss = histc(offsets,edg);

    songscore(Iint,1:4) = [csg,tss(1),numhash(Iint),md];
            % songscore contains the song id, the song score, the
            % original num. of matching hashes and the most popular offset.

    freqcount(songnum(Iint)) = 0;
end

songscore = sortrows(songscore,-2);         % Descending sort by song score
if songscore(1,2)-songscore(2,2)<= NOMATCH  % Verification: Reject match
    result = 'Failure!';
    varargout = {songscore, DistData};
else                                        % Verification: Accept match
    result = METADATA{songscore(1,1),1};
    matchresults1 = ['There were ' int2str(songscore(1,2))...
        ' hash pairs that pointed to this song.'];
    disp(matchresults1)

    varargout = {songscore, DistData};

end
```

Bibliography

[1] Sophocles J. Orfanidis. *Introduction to Signal Processing.* Prentice Hall, 1996.

[2] Peter M. Clarkson and Henry Stark. *Signal Processing Methods for Audio, Images and Telecommunications.* Academic Press, 1996.

[3] John A. Gubner and Wei bin Chang. Wavelet transforms for discrete-time periodic signals. *Sig. Proc*, 42:167–180, 1995.

[4] Stéphane Mallat. *A Wavelet Tour of Signal Processing, Second Edition (Wavelet Analysis & Its Applications).* Academic Press Inc., San Diego, CA, 1999.

[5] Charles K. Chui. *An introduction to wavelets*, volume 1 of *Wavelet Analysis and its Applications.* Academic Press Inc., Boston, MA, 1992.

[6] Yves Nievergelt. *Wavelets made easy.* Birkhäuser Boston Inc., Boston, MA, 1999.

[7] Christian Blatter. *Wavelets.* A K Peters Ltd., Natick, MA, 1998. A primer.

[8] Gerald Kaiser. *A friendly guide to wavelets.* Birkhäuser Boston Inc., Boston, MA, 1994.

[9] Lokenath Debnath. *Wavelet transforms and their applications.* Birkhäuser Boston Inc., Boston, MA, 2002.

[10] Paul S. Addison. *The illustrated wavelet transform handbook.* IOP Publishing Ltd., Bristol, 2002. Introductory theory and applications in science, engineering, medicine and finance.

[11] G. Tzanetakis, G. Essl, and P. Cook. Audio analysis using the discrete wavelet transform. In *Proc. Conf. in Acoustics and Music Theory Applications.* Citeseer, 2001.

[12] C. Yang. Music Database Retrieval Based on Spectral Similarity. In *Proceedings of the 2nd Annual International Symposium on Music Information Retrieval*, pages 37–38, 2001.

[13] M. Mohri, P. Moreno, and E. Weinstein. Robust Music Identification, Detection, and Analysis. In *Proceedings of the International Conference on Music Information Retrieval (ISMIR), Vienna, Austria*, pages 135–139, 2007.

[14] P. Cano, E. Batlle, T. Kalker, and J. Haitsma. A Review of Audio Fingerprinting. *The Journal of VLSI Signal Processing*, 41(3):271–284, 2005.

[15] Pedro Cano. *Content-Based Audio Search.* VDM Verlag Dr. Müller, 2009.

[16] J. Haitsma and T. Kalker. A Highly Robust Audio Fingerprinting System. In *Proc. ISMIR*, volume 2, 2002.

[17] A. Wang. An Industrial-Strength Audio Search Algorithm. In *Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR 2003)*, 2003.

[18] S. Baluja and M. Covell. Content Fingerprinting Using Wavelets. In *Visual Media Production, 2006. CVMP 2006. 3rd European Conference on*, pages 198–207, 2006.

[19] Y. Ke, D. Hoiem, and R. Sukthankar. Computer Vision for Music Identification. In *IEEE Computer Society Conferance on Computer Vision and Pattern Recognition*, volume 1, page 597. Citeseer, 2005.

[20] M.L. Miller, M.A. Rodriguez, and I.J. Cox. Audio Fingerprinting: Nearest Neighbor Search in High Dimensional Binary Spaces. *The Journal of VLSI Signal Processing*, 41(3):285–291, 2005.

[21] S.L. Hsieh and H.C. Wang. Feature Extraction for Audio Fingerprinting Using Wavelet Transformation. In *National Computer Symposium,(NCS'05)*, 2005.

[22] J.S. Seo, J. Haitsma, and T. Kalker. Linear speed-change resilient audio fingerprinting. In *Proc. IEEE Benelux Workshop on Model Based Processing and Coding of Audio (MPCA) 2002*, 2002.

[23] S.R. Subramanya and Abdou Youssef. Wavelet-based indexing of audio data in audio/multimedia databases. *Multimedia Database Management Systems, International Workshop*, 0:0046, 1998.

[24] V. Venkatachalam, L. Cazzanti, N. Dhillon, M. Wells, M.E. Inc, and WA Kirkland. Automatic Identification of Sound Recordings. *IEEE Signal Processing Magazine*, 21(2):92–99, 2004.

[25] Free m4a to mp3 converter. `http://www.maniactools.com/soft/m4a-to-mp3-converter`, 2009. This is an electronic document. Date retrieved: October 27, 2009.

[26] Scott Smitelli. Fun with youtube's audio content id system. `http://www.csh.rit.edu/~parallax/`, 2009. This is an electronic document. Date retrieved: October 27, 2009.

[27] Dan Ellis. mp3read and mp3write for MATLAB. `http://www.ee.columbia.edu/~dpwe/resources/matlab/mp3read.html`, 2009. This is an electronic document. Date retrieved: October 27, 2009.

[28] Dan Ellis. Robust Landmark-based Audio Fingerprinting. `http://www.ee.columbia.edu/~dpwe/resources/matlab/fingerprint/`, 2009. This is an electronic document. Date retrieved: October 27, 2009.