2013

DISCRETE TRANSFORMS WITH GOOD TIME-FREQUENCY AND SPATIAL-FREQUENCY LOCALIZATION

David Chisholm
CUNY City College

How does access to this work benefit you? Let us know!
Follow this and additional works at: http://academicworks.cuny.edu/cc_etds_theses
Part of the Computer Sciences Commons

Recommended Citation
Chisholm, David, "DISCRETE TRANSFORMS WITH GOOD TIME-FREQUENCY AND SPATIAL-FREQUENCY LOCALIZATION" (2013). CUNY Academic Works.
http://academicworks.cuny.edu/cc_etds_theses/550

This Thesis is brought to you for free and open access by the City College of New York at CUNY Academic Works. It has been accepted for inclusion in Master's Theses by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.
DISCRETE TRANSFORMS WITH GOOD TIME-FREQUENCY AND SPATIAL-FREQUENCY LOCALIZATION

A thesis submitted in partial fulfillment of the requirements for the degree

Master of Science
at
The City College of New York
of the
City University of New York

by
David Chisholm
August 2013

Approved:

____________________________________
Professor Izidor Gertner, Thesis Advisor

____________________________________
Professor Douglas Troeger, Chairman

Department of Computer Science
**ABSTRACT**

Discrete orthonormal time-frequency basis functions are described and used for both analysis and synthesis of complex-valued signals. We derive expressions for complex-valued expansion coefficients in time-frequency lattices in the discrete one dimensional case. This derivation is based on Professor I. Gertner's previous construction of a complete orthonormal set of basis functions well localized in the temporal-spatial-frequency domain in the continuous case. We describe how these can be generalized to any number of dimensions. Example applications are presented in one and two dimensions. Three dimensional basis functions are visualized and discussed. Finally, a full Matlab implementation of this work is provided. Chapter two of this thesis has been submitted for publication as a self-contained paper.
ACKNOWLEDGEMENTS

I would foremost like to thank Professor Izidor Gertner, whose continual support, guidance and mentorship has helped me complete this thesis and much more.

I would also like to thank Tamara Johnson for helping and supporting me during the writing process.

I would also like to thank Professor Chris Chafe, Mr. Gary Johnson and Mr. Gary D’Aries, each of whose academic and professional mentorship in previous years helped me get to this point.

Finally I would like to thank the Computer Science department at CCNY for giving me the opportunity to teach here while completing my degree.
**TABLE OF CONTENTS**

1. Introduction ........................................................................................................................................... 7
   1.1. Thesis Statement .............................................................................................................................. 7
   1.2. Technical Background and Literature Review .................................................................................. 8
      Fourier Transform ............................................................................................................................... 8
      Short Time Fourier Transform .......................................................................................................... 9
      Finite, Discrete STFT ......................................................................................................................... 11
      Gabor Transforms ............................................................................................................................. 12
      Modified Discrete Cosine Transform ............................................................................................... 14
      Wilson Bases ..................................................................................................................................... 15
2. Discrete Transform with Good Time-Frequency Localization [submitted for publication] .................. 17
3. Discussion and examples in one dimension ......................................................................................... 23
   3.1. Fast Computation based on the FFT .............................................................................................. 23
   3.2. Visualizations ............................................................................................................................... 26
   3.3. Example applications .................................................................................................................... 27
      Reconstruction, Discontinuities and Compression .............................................................................. 27
      Noise Reduction ............................................................................................................................... 27
      Visualizing a polynomial ................................................................................................................... 30
4. Discussion and examples in two dimensions ....................................................................................... 31
   4.1. Coefficient expressions .................................................................................................................. 31
   4.2. Visualizations ............................................................................................................................... 34
   4.3. Example applications .................................................................................................................... 40
      Reconstruction & Compression ........................................................................................................... 40
5. Discussion and examples in higher dimensions .................................................................................. 48
   5.1. Extending basis functions to any D dimensions ............................................................................ 48
   5.2. Visualization of 3D Basis Functions .............................................................................................. 50
6. Conclusions .......................................................................................................................................... 54
7. References ............................................................................................................................................. 55
8. Appendix – complete code listing ..................................................................................................... 58
   8.1. Example1DReconstruction ........................................................................................................... 58
   8.2. Example2DReconstruction .............................................................................................................. 62
   8.3. ExampleAudioDenoising .................................................................................................................. 64
   8.4. ExampleFastComputation ............................................................................................................. 66
   8.5. ExamplePolynomial ....................................................................................................................... 68
   8.6. ExampleShowBasis1D ...................................................................................................................... 69
   8.7. ExampleShowBasis2D ...................................................................................................................... 71
   8.8. ExampleShowBasis3D ...................................................................................................................... 73
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Visualizations of 1D basis functions</td>
</tr>
<tr>
<td>2</td>
<td>Cosine channel lattice of original audio</td>
</tr>
<tr>
<td>3</td>
<td>Cosine channel lattice of audio with noise added</td>
</tr>
<tr>
<td>4</td>
<td>Cosine channel lattice of denoised audio</td>
</tr>
<tr>
<td>5</td>
<td>Time-frequency lattice for polynomial expression</td>
</tr>
<tr>
<td>6</td>
<td>$C_\cos \times \cos(0,0,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>7</td>
<td>$C_\cos \times \cos(0,4,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>8</td>
<td>$C_\cos \times \cos(4,0,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>9</td>
<td>$C_\cos \times \cos(4,4,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>10</td>
<td>$C_\cos \times \cos(1,1,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>11</td>
<td>$C_\sin \times \cos(1,1,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>12</td>
<td>$C_\cos \times \sin(1,1,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>13</td>
<td>$C_\sin \times \sin(1,1,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>14</td>
<td>$C_\cos \times \cos(8,1,0)$ basis function displayed from overhead and from a 3D perspective</td>
</tr>
<tr>
<td>15</td>
<td>Original image composed of a pair of cross faded 2D chirp functions</td>
</tr>
<tr>
<td>16</td>
<td>Chirp function image reconstructed with 100% of coefficients</td>
</tr>
<tr>
<td>17</td>
<td>Chirp function image reconstructed with 10% of coefficients</td>
</tr>
<tr>
<td>18</td>
<td>Chirp function image reconstructed with 3% of coefficients</td>
</tr>
<tr>
<td>19</td>
<td>Original reference image</td>
</tr>
<tr>
<td>20</td>
<td>Reference image reconstructed using 100% of coefficients</td>
</tr>
<tr>
<td>21</td>
<td>Reference image reconstructed using 10% of coefficients</td>
</tr>
<tr>
<td>22</td>
<td>Reference image reconstructed using 3% of coefficients</td>
</tr>
<tr>
<td>23</td>
<td>$\Psi_{0,1,0,0}$ basis function displayed as a series of overhead intensity images along the Z axis</td>
</tr>
<tr>
<td>24</td>
<td>$\Psi_{0,1,0,0}$ basis function displayed as a series of 3D perspective images along the Z axis</td>
</tr>
<tr>
<td>25</td>
<td>$\Psi_{0,1,1,0}$ basis function displayed as a series of overhead intensity images along the Z axis</td>
</tr>
<tr>
<td>26</td>
<td>$\Psi_{0,1,1,0}$ basis function displayed as a series of 3D perspective images along the Z axis</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The most common representation of many signals is the time domain representation, which shows how the amplitude of a signal changes over time. Audio data, seismograms, temperature readings and many more “real-world” signals are usually represented this way. Image data is similarly represented in the spatial domain by denoting amplitude changes over location.

In some cases a frequency domain representation is more useful. In this representation signal information is represented as amplitudes over frequency rather than over time (or space). Concepts such as the pitch of a recorded musical note are often better described by frequency domain representations.

There are many applications for which yet another domain – the time-frequency domain – is the most useful one. This thesis deals with a particular method of representing signals in the time-frequency domain. Such representations allow us to capitalize on common properties in signals; for instance, frequency content is often consistent for short intervals of a signal’s duration but changes greatly over the entire duration. These properties can often be more easily exploited in the time frequency domain than other domains for purposes such as compression, enhancement and analysis.

1.1. Thesis Statement

Discrete orthonormal time-frequency basis functions are described and used for both analysis and synthesis of complex-valued signals. We derive expressions for complex-valued expansion coefficients in time-frequency lattices in the discrete one dimensional case. This derivation is based on Professor I. Gertner's previous construction of a complete orthonormal set of basis functions well localized in the temporal-spatial-frequency domain in the continuous case [Gert12] [Gert13A]. We describe how these can be generalized to any number of dimensions. Example applications are presented in one and two dimensions. Three dimensional basis functions are visualized and discussed. Finally, a full a Matlab implementation of this work is provided. Chapter two of this thesis has been submitted for publication as a self-contained paper.
1.2. TECHNICAL BACKGROUND AND LITERATURE REVIEW

FOURIER TRANSFORM

The fundamental tool for any frequency or time frequency signal analysis is the Fourier transform. The continuous Fourier transform $F$ of a time domain function $f$ is defined as [Tol76]:

$$ F(\nu) = \int_{-\infty}^{\infty} f(t) e^{-i2\pi \nu t} dt $$

where:
- $\nu$ represents continuous frequency\(^1\)
- $t$ represents continuous time

The Fourier transform allows us to view the content of a signal as a function of frequency rather than time; $F$ and $f$ are different representations of the same signal content.

An equivalent representation of the Fourier transform is

$$ F(\nu) = \int_{-\infty}^{\infty} f(t) (\cos(2\pi \nu t) - i \sin(2\pi \nu t)) dt $$

This representation simply expands the complex exponential into real and imaginary components. Note that the oscillators in the real and imaginary terms have a phase offset difference of $\pi/2$ and are orthogonal.

The frequency content information provided by the Fourier transform is completely unlocalized in time. That is to say, the energy coefficients for a particular frequency represent the content of that frequency over the entire duration of the signal. There is no way to tell via the Fourier transform whether a certain frequency has more or less energy during a particular portion of the signal.

---

\(^1\) Note here $\nu$ indicates temporal rather than angular frequency. Both notations are commonly used.
The short time Fourier transform (STFT) is an analysis method that adds the notion of time dependency to the Fourier transform. In the STFT, the signal is windowed around a particular instant in time\(^2\), and the frequency content of the result is then measured. Unlike in the standard Fourier transform, this allows us to determine the energy at a certain frequency for a particular portion of the signal, rather than the signal as a whole. The STFT is defined as

\[
STFT(n, \nu; h) = \int_{-\infty}^{\infty} f(t) h^*(t - n) e^{-i2\pi \nu t} dt
\]

where:

- \( h(t) \) is a “window” function\(^3\) localized in time around \( t = 0 \) and in frequency around \( \nu = 0 \)
- \( n \) is the chosen instant in time
- \( \nu \) is the frequency to be analyzed

Note that the window is translated in time by subtracting \( n \) when evaluating \( h^* \). Similarly, it is translated in frequency by multiplying by the complex exponential. Thus our analysis is localized around a particular time and frequency.

Selection of the window function affects the results of this transformation in many ways [Har78]. Ideally, we would like to select a window function to provide perfect resolution for localization in both frequency and time. If we consider the time-frequency domain as a two dimensional plane\(^4\) such a perfectly localized signal would correspond to a single point on this plane. Unfortunately this is impossible due to the Uncertainty principle. In order to illustrate this notion we will follow a presentation used in [Aug10]. To do this, we consider a probability distribution \( f(t) \) and its Fourier transform \( F(\nu) \).

---

\(^2\) An analysis window that converges to zero after some duration in either direction in time is typically chosen. This suppresses the signal beyond the near vicinity of the chosen instant.

\(^3\) The * in the preceding equation denotes complex conjugation

\(^4\) With time as one axis and frequency as the other
First define the total energy of \( f(t) \), assumed to be finite:

\[
E_{\text{sig}} = \int_{-\infty}^{\infty} |f(t)|^2 \, dt
\]

Next the average time:

\[
t_{\text{avg}} = \frac{1}{E_{\text{sig}}} \int_{-\infty}^{\infty} t |f(t)|^2 \, dt
\]

Next the average frequency:

\[
v_{\text{avg}} = \frac{1}{E_{\text{sig}}} \int_{-\infty}^{\infty} t |F(v)|^2 \, dv
\]

Next the variance in time:

\[
T^2 = \frac{4\pi}{E_{\text{sig}}} \int_{-\infty}^{\infty} (t - t_{\text{avg}})^2 |f(t)|^2 \, dt
\]

And the variance in frequency:

\[
B^2 = \frac{4\pi}{E_{\text{sig}}} \int_{-\infty}^{\infty} (v - v_{\text{avg}})^2 |F(v)|^2 \, dv
\]

Finally define the **time-bandwidth product** as \( T \times B \) [Aug10].

For all signals there is a positive lower bound\(^5\) to this time-bandwidth product. A Gaussian signal of the form:

\[
x(t) = Ce^{i\alpha(t-t_{\text{avg}})^2 + i2\pi v_{\text{avg}}(t-t_{\text{avg}})}
\]

with \( C \in \mathbb{R}, \alpha \in \mathbb{R}_+ \) is well localized in time and frequency, and thus achieves this bound.

---

\(^5\) We have chosen a formulation for time-bandwidth product which results in a lower bound of 1 [Aug10]. However, there are several different formulations widely used. For a formulation that results in a limit of .25 see [Gad10]; for formulations that result in limits of .5 or \( \pi \) see [Smit13]. All express the same underlying principle.
Finite, Discrete STFT

Until now we have been describing signals in the infinite continuous case. Here we introduce the STFT for a finite discrete signal \( f(t) \), defined as [Søn07]:

\[
STFT(n, \nu; h) = \sum_{t=0}^{L-1} f(t) h^*(t - n) e^{-i2\pi \nu t/L}
\]

where:
- \( f(t) \in \mathbb{C}^L \) (That is, a complex signal of length \( L \))
- \( h(t) \) is a discrete window function localized \( t = 0 \) and \( \nu = 0 \)
- \( n \) is the chosen discrete sample in time with \( 0 \leq n < L \)
- \( \nu \) is the frequency to be analyzed with \( 0 \leq \nu < L \)

Another desirable property of a time frequency analysis technique is that it be invertible, allowing the original time domain signal to be reconstructed from the time-frequency representation. The discrete STFT is invertible using [Søn07]:

\[
f(t) = \sum_{n,\nu=0}^{L-1} \frac{1}{\|h\|^2} \ STFT(n, \nu; h) h(t - n) e^{i2\pi \nu t/L}
\]

This formula illustrates a drawback of the STFT. Note that our original signal \( f(t) \) was represented by \( L \) coefficients. However, due to \( 0 \leq \nu, n < L \), the discrete STFT of \( f(t) \) has \( L^2 \) coefficients, all of which are processed when reconstructing the signal. Thus the STFT is highly redundant\(^6\), and its size grows quadratically as the amount of information in the original signal increases linearly. In order to address this shortcoming, the Gabor transform is often used.

\(^6\) Or “oversampled”
**GA\B\O\R\ TRANSFORMS**

The Gabor transform is described as a subsampled case of the STFT in [Søn07]. In the STFT the chosen time $n$ could vary by the smallest discrete time increment (a single sample) and the chosen frequency $\nu$ could vary by the smallest discrete frequency\(^7\) increment ($\frac{1}{L}$). The Gabor transform, along with its inverse the Gabor expansion\(^8\), introduces the concepts of time separation and frequency separation between those instants and frequencies that can be chosen for analysis. It is defined:

$$GT(n, \nu; h) = \sum_{t=0}^{L-1} f(t) h^*(t - an) e^{-i2\pi\nu t \over M}$$

where:

- $f(t) \in \mathbb{C}^L$ (That is, a complex signal of length $L$)
- $h(t)$ is a discrete window function localized $t = 0$ and $\nu = 0$
- $a$ is the time separation
- $N$ is the total number of time intervals with $N = \frac{L}{a}$
- $b$ is the frequency separation
- $M$ is the total number of frequency bands with $M = \frac{L}{b}$
- $n$ is the chosen time interval with $0 \leq n < N$
- $\nu$ is the frequency band to be analyzed with $0 \leq \nu < M$

The number of coefficients in a Gabor system depends on the choices for $a$ and $b$ and will equal $MN$. Gabor systems may be under, over or critically sampled\(^9\).

The ability to reduce redundancy compared to the STFT is an advantage for the Gabor transform. However, there are still some drawbacks to the Gabor transform. Most notably, windows used for reconstruction in a critically sampled Gabor system are not numerically

---

\(^7\) Here we refer to frequencies whose period can be expressed as a whole number of samples. Thus the increment for frequency or time instant is the same: 1/L. Consider that a finite discrete signal $f(t)$ has the same number of time samples as its discrete Fourier transform $F(t)$ has frequency samples.

\(^8\) Together, a Gabor transform and expansion are called a **Gabor system**

\(^9\) Here we refer to the redundancy (or lack therof) of a system. A system where $ab < L$ is redundant and will have more samples than the original time domain signal (aka over sampled). When $ab = L$ the system has the same amount of samples and is critically sampled; otherwise it is under sampled and discards some of the information in the original signal.
stable. In order to achieve perfect reconstruction with well-behaved windows a Gabor system must be over sampled. These over sampled Gabor systems are thus frames\textsuperscript{10} instead of bases, and use a pair of dual windows for analysis and reconstruction. Algorithmic generation of these windows is a non-trivial process and discussed at length in [Søn07].

\textsuperscript{10} For a detailed explanation of frames, see section 6.4.1 of [Søn07]
**Modified Discrete Cosine Transform**

The modified discrete cosine transform (MDCT) was introduced in [Prin87]. Many invertible time-frequency transforms break data into smaller blocks prior to processing; often during this process artifacts are introduced at the boundaries of adjacent blocks. The MDCT addresses this issue by overlapping adjacent blocks and applying an appropriate window function to each block. In addition, it is critically sampled. The MDCT is a real, rather than complex, valued transform. There are several formulations of the MDCT; a slight modification from one presented in [Smit13] is:

\[
MDAC(v; h) = \sum_{t=0}^{B-1} f(t) h(t) \cos\left(\frac{\pi}{2B} \left( 2t + 1 + \frac{B}{2} \right) (2v + 1) \right)
\]

where:

- \(f(t) \in R^L\) (That is, a real signal of length \(L\))
- \(h(t)\) is a discrete window function meeting several conditions\(^\text{11}\)
- \(B\) is the block length
- \(N\) is the total number of blocks with \(N = \frac{L}{B}\)
- \(n\) is the chosen block with \(0 \leq n < N\)
- \(\nu\) is the frequency band to be analyzed with \(0 \leq \nu < \frac{B}{2} - 1\)

The IMDAC gives a single block of data; blocks are then added to reconstruct the original signal. A single block of the IMDAC is defined:

\[
IMDAC(i; h) = \sum_{\nu=0}^{\frac{B}{2} - 1} h(i) \cos\left(\frac{\pi}{2B} \left( 2i + 1 + \frac{B}{2} \right) (2\nu + 1) \right)
\]

where:

- \(i\) specifies the sample within the block and \(0 \leq i < B\)

\(^{11}\) Usually the same window is used for both analysis and reconstruction and it is localized around \(t = 0\) and \(\nu = 0\), although these conditions are not strictly required. The window must be of length \(B\) and must satisfy \([h(t+B/2)]^2 + [h(t)]^2 = 1\) for \(0 \leq t < B/2\). For more detail on window requirements, see [Smit13] or [Mal99]. The cosine window is a typical choice.
Continuous Wilson bases are described in [Daub91] and a discrete transform is described in [Böl96]. A slight modification on the definition of the discrete Wilson transform in [Søn07] is:

**If** \( m = 0 \)

\[
\text{DWILT}(m; n) = \sum_{t=0}^{L-1} f(t) h(t - 2nA)
\]

**If** \( m \) **is odd and** \( m < M \)

\[
\text{DWILT}(m; n) = \sqrt{2} \sum_{t=0}^{L-1} f(t) \sin(\pi \frac{m}{M} l) h(t - 2nA)
\]

\[
\text{DWILT}(m; n) = \sqrt{2} \sum_{t=0}^{L-1} f(t) \cos(\pi \frac{m}{M} l) h(t - (2n + 1)A)
\]

**If** \( m \) **is even and** \( m < M \)

\[
\text{DWILT}(m; n) = \sqrt{2} \sum_{t=0}^{L-1} f(t) \cos(\pi \frac{m}{M} l) h(t - 2nA)
\]

\[
\text{DWILT}(m; n) = \sqrt{2} \sum_{t=0}^{L-1} f(t) \sin(\pi \frac{m}{M} l) h(t - (2n + 1)A)
\]

**If** \( m \) **is even and** \( m = M \)

\[
\text{DWILT}(m; n) = \sum_{t=0}^{L-1} f(t) (-1)^t h(t - 2nA)
\]

**If** \( m \) **is odd and** \( m = M \)

\[
\text{DWILT}(m; n) = \sum_{t=0}^{L-1} f(t) (-1)^t h(t - (2n + 1)A)
\]
where:

- \( f(t) \in \mathbb{C}^L \) (That is, a complex signal of length \( L \))
- \( M \) is one-half the number of frequency coefficients per time interval.
- \( A \) is the time separation
- \( N \) is the total number of time intervals with \( N = \frac{L}{A} \)
- \( n \) is the chosen block with \( 0 \leq n < N \)
- \( m \) is the frequency index \( 0 \leq m < 2M \)
- \( h(t) \) is an appropriate window function. Requirements for an appropriate window function are strict. Such a function was first constructed in [Daub91] and another was later constructed in [Gert13A] – see these sources for more information on and [Søn07] for an implementation of window construction.
Izidor Gertner, Senior Member, IEEE, Dave Chisholm, Student Member, IEEE

Abstract—Discrete orthonormal time-frequency basis functions are described and used for both analysis and synthesis of complex-valued signals.

Index Terms—Gabor Analysis, Time-Frequency Analysis, Zak Transform, Wilson Basis

I. INTRODUCTION

A goal of any time frequency analysis technique is to provide information that is well localized in the time-frequency plane. Gabor analysis attempts to provide time frequency information by using time and frequency shifts of a Gaussian multiplied by complex exponential. Unfortunately, the Balian-Low theorem states that if a square integrable function generates a Gabor Riesz basis on an integer lattice then it is not well-localized either in time or in frequency [8]. Wilson bases where constructed in the continuous case by Daubechies et al in [4] and in [7]. The technique in [7] is based on constructing a basis generating function using the normalized Zak transform and then using it with an alternating sines and cosines in a specific way to construct orthonormal basis functions well localized in time and frequency with redundancy two. Zak in 2002 [3] constructed a unique expression for the complete and orthonormal set of basis functions, for all discrete frequency and time indexes $m, l = -\infty, \ldots, \infty$ on the lattice, in the form of a $kq$-representation for any real function $\phi(x)$. Gertner [1],[2] derived an explicit, single expression for a complex-valued, orthonormal basis well localized in time-frequency domain from Gaussian divided by the square root of a Jacobi theta function.

A discrete Wilson basis was discussed by Bölcskei et al in [5]. Søndergaard extensively discussed algorithmic approaches to discrete Gabor problem in [6].

Here we present a discrete version of [1],[2] to create orthonormal time-frequency basis functions used for both analysis and synthesis of complex-valued signals.

The paper is organized as follows: Section II reviews known results from [1],[2], Section III presents the discrete version of the time-frequency analysis algorithm, Section IV describes the synthesis algorithm from the expansion coefficients, and Section V discusses an example.
II. BACKGROUND

In [1],[2] the basis generating function was obtained from Gaussian by dividing it with a square root of Jacobi theta function and it was simplified to:

$$\psi_{00}(x) = \frac{4}{\alpha \pi \lambda^2} \frac{1}{\Theta_3 \left( \frac{x}{\lambda}, \alpha \pi \lambda, e^{-\alpha \pi} \right)}$$  \hspace{1cm} (1)

for $\lambda > 0$, $\alpha > 2.25$. $\Theta_3$ denotes Jacobi theta function.

The set of functions $[1],[2]$:

$$\psi_{lm}(x) = \cos \left[ \frac{2\pi}{T} \left( x - \frac{T}{4} m \right) \right] \psi_{00}(x - T m) + \sin \left[ \frac{2\pi}{T} \left( x - \frac{T}{4} m \right) \right] \psi_{00}(x - T m - \frac{T}{2}), \quad x \in \mathbb{R}$$  \hspace{1cm} (2)

form a complete, orthonormal basis well localized in time and frequency. Where $l = -\infty, \ldots, -1, 0, 1, \ldots, \infty$ is the discrete frequency index and $m = -\infty, \ldots, -1, 0, 1, \ldots, \infty$ is the discrete time index, and $T = 2\lambda \sqrt{\Delta t}$.

A. Complex-valued Signal Expansion into Time-Frequency Series

We consider an expansion of a complex-valued signal $S(x) = S_{Re}(x) + \text{i} S_{Im}(x)$ into time-frequency series [1],[2]

$$S(x) = \sum_{l} \sum_{m} (C_{\cos}(l, m) + \text{i} C_{\sin}(l, m)) \psi_{lm}(x),$$  \hspace{1cm} (3)

where $\psi_{lm}(x)$ denotes complex conjugate of (2). Because of orthogonality of $\psi_{lm}(x)$ the expansion coefficients

$$C_{\cos}(l, m) = W \int_{-\infty}^{\infty} (S_{Re}(x) + \text{i} S_{Im}(x)) \cdot \cos \left[ \frac{2\pi}{T} \left( x - \frac{T}{4} m \right) \right] \psi_{00}(x - m \cdot T) dx$$  \hspace{1cm} (4)

$$C_{\sin}(l, m) = W \int_{-\infty}^{\infty} (S_{Re}(x) + \text{i} S_{Im}(x)) \cdot \sin \left[ \frac{2\pi}{T} \left( x - \frac{T}{4} m \right) \right] \psi_{00}(x - m \cdot T - \frac{T}{2}) dx$$  \hspace{1cm} (5)

$W$ is a normalization constant.

In the next section we present discrete version of (4),(5).

III. DISCRETE-TIME ORTHONORMAL BASIS

Let $N$ be an even integer. Denote $N$ the number of discrete time intervals in $T$, and the sampling rate $\frac{1}{\Delta x}$, where $\Delta x = \frac{T}{N}$. The actual value of $N$ is determined based on the required frequency resolution $\frac{2\pi}{\Delta x \cdot N}$. Then for $T$ the discrete samples are $n \Delta x, n = 0, \ldots, N-1$. For $l = -\infty, \ldots, -1, 0, 1, \ldots, \infty, m = -\infty, \ldots, -1, 0, 1, \ldots, \infty$ (2) can be written in the form
\[ \psi_{1m}(n) = \cos \left[ \frac{2\pi}{N} \left( n - \frac{N}{4} \right) \right] \cdot \psi_{00}(n - Nm) + \sin \left[ \frac{2\pi}{N} \left( n - \frac{N}{4} \right) \right] \cdot \psi_{00}(n - Nm - \frac{N}{2}) \quad n = \ldots -2, -1, 0, 1, 2, \ldots \infty \]  

Discretize \( S(x) \) into \( L \) complex-valued samples \( S(n) = S_{Re}(n) + is_{Im}(n) \) \( n = 0, \ldots, L-1 \), and extend \( S(n) \) periodically \( S(n) = S(n + pL) \) for \( p \in \mathbb{Z} \). Select \( N \) such \( L = 0 \mod(N) \). Then \( m = 0, 1, \ldots \frac{L}{N} - 1 \) and \( l = 0, 1, \ldots, \frac{N}{2} \).

A. Description of the computational algorithm

We compute the expansion coefficients in (3) for \( C_{\cos}(l, m) \) and \( C_{\sin}(l, m) \) separately.

To compute \( C_{\cos}(l, m) \), change the integration variable \( x' = x - m \cdot T \) in (4) to yield

\[
C_{\cos}(l, m) = W \int_{-\infty}^{\infty} \left( S_{Re}(x + m \cdot T) + iS_{Im}(x + m \cdot T) \right) \cdot \cos \left( \frac{2\pi}{T} \left( x - \frac{T}{4} \right) \right) \cdot \psi_{00}(x) \, dx
\]

The discrete version of (4) has expression for \( m = 0, 1, \ldots \frac{L}{N} - 1, l = 0, 1, \ldots, \frac{N}{2} \).

\[
C_{\cos}(l, m) = W \sum_{n=\infty}^{\infty} \left( S_{Re}(n + m \cdot T) + iS_{Im}(n + m \cdot T) \right) \cdot \cos \left( \frac{2\pi}{T} \left( n + \frac{T}{4} \right) \right) \cdot \psi_{00}(n) \quad (7)
\]

Similarly, for \( C_{\sin}(l, m) \), change integration variable \( x' = x - m \cdot T - \frac{T}{2} \) in (5) to yield

\[
C_{\sin}(l, m) = W \int_{-\infty}^{\infty} \left( S_{Re}(x + m \cdot T + \frac{T}{2}) + iS_{Im}(x + m \cdot T + \frac{T}{2}) \right) \cdot \sin \left( \frac{2\pi}{T} \left( x + \frac{T}{4} \right) \right) \cdot \psi_{00}(x) \, dx
\]

The discrete version of (5) has expression for \( m = 0, 1, \ldots \frac{L}{N} - 1, l = 0, 1, \ldots, \frac{N}{2} \).

\[
C_{\sin}(l, m) = W \sum_{n=\infty}^{\infty} \left( S_{Re}(n + m \cdot T + \frac{T}{2}) + iS_{Im}(n + m \cdot T + \frac{T}{2}) \right) \cdot \sin \left( \frac{2\pi}{T} \left( n + \frac{T}{4} \right) \right) \cdot \psi_{00}(n) \quad (8)
\]

The normalizing factor \( W \) is chosen to satisfy \( \sum_{n} \left( \psi_{00}^2(n) + \psi_{00}^2\left( n - \frac{N}{2} \right) \right) = 1 \)

and \( W = \sqrt{\frac{2}{N}} \) when \( l = 0 \mod\left( \frac{N}{2} \right) \) and \( W = \sqrt{\frac{1}{N}} \) when \( l \neq 0 \mod\left( \frac{N}{2} \right) \).

Since \( \psi_{00}(n) \) decays exponentially, assume that \( \psi_{00}(n) \) vanishes outside \( -r \frac{N}{2}, \ldots, -1, 0, 1, \ldots, r \frac{N}{2} - 1 \), where \( r \) is a small integer.

For the purpose of numerical computation, we have chosen \( r = 4 \) where the value of \( \psi_{00}(n) \) outside this interval is less than \( 10^{-k} \). Then (7) become for the discrete frequency index \( l = 0 \mod\left( \frac{N}{2} \right) \).
\[ C_{\cos}(0,m) = C_{\cos}(0,m) + iC_{\cos}(0,m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} (s_{\Re}(n+m \cdot N) + i s_{\Im}(n+m \cdot N)) \cdot \cos\left(\frac{2 \pi (n - N)}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (9) \]

\[ C_{\cos}(\frac{N}{2}, m) = C_{\cos}(\frac{N}{2}, m) + iC_{\cos}(\frac{N}{2}, m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} (s_{\Re}(n+m \cdot N) + i s_{\Im}(n+m \cdot N)) \cdot \cos\left(\frac{2 \pi (n - N)}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (9) \]

and (8) become for discrete frequency indexes \( l = 0 \), \( \text{mod}\left(\frac{N}{2}\right) \)

\[ C_{\sin}(0,m) = 0, \quad m = 0, 1, \ldots \frac{L}{N} - 1 \quad \text{, } \quad C_{\sin}(\frac{N}{2}, m) = 0, \quad m = 0, 1, \ldots \frac{L}{N} - 1 \quad \text{, } \quad (10) \]

For the discrete frequencies \( l \neq 0 \), \( \text{mod}\left(\frac{N}{2}\right) \), the coefficients \( C_{\cos}(l,m), C_{\sin}(l,m) \) have the following expressions:

\[ C_{\cos}(l,m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} (s_{\Re}(n+m \cdot N) + i s_{\Im}(n+m \cdot N)) \cdot \cos\left(\frac{2 \pi (n - N)}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (11) \]

\[ C_{\sin}(l,m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} (s_{\Re}(n+m \cdot N) + i s_{\Im}(n+m \cdot N)) \cdot \sin\left(\frac{2 \pi (n - N)}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (12) \]

Denote the real part and imaginary part of \( C_{\cos}(l,m) \) (11) as

\[ C_{\cos}(l,m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} s_{\Re}(n+m \cdot N) \cdot \cos\left(\frac{2 \pi (n - N)}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (13) \]

\[ C_{\cos}(l,m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} s_{\Im}(n+m \cdot N) \cdot \cos\left(\frac{2 \pi (n - N)}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (14) \]

and denote the real and imaginary part of \( C_{\sin}(l,m) \) (12) as

\[ C_{\sin}(l,m) = \begin{aligned} \frac{r}{N} \sum_{n = -\frac{r}{2}}^{\frac{r}{2}-1} s_{\Re}(n+m \cdot N + \frac{N}{2}) \cdot \sin\left(\frac{2 \pi (n + \frac{N}{2})}{N} \cdot \frac{1}{l} \right) \cdot \psi_{00}(n), \quad m = 0, 1, \ldots \frac{L}{N} - 1 \end{aligned}, \quad (15) \]
\[
C_{\text{sin}}^{\text{Im}}(l, m) = \frac{2}{q^N} \sum_{n = -\frac{N}{2}}^{\frac{N}{2} - 1} S_{\text{sin}}^{\text{Im}} \left( n + m \cdot N + \frac{N}{2} \right) \cdot \sin \left( \frac{2\pi}{N} \left( n + \frac{N}{2} \right) \right) \cdot \psi_0(n), \quad m = 0, 1, \ldots, \frac{L}{N} - 1 \\
S_{\text{cos}}^{\text{Re}}(i) = \sum_{l=0}^{\frac{N}{2} - 1} \sum_{m=0}^{\frac{L}{N} - 1} \sum_{i=0}^{\frac{L}{N} - 1} W \left[ C_{\text{cos}}^{\text{Re}}(l, m) \cdot \cos \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) + C_{\text{sin}}^{\text{Re}}(l, m) \cdot \sin \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) \right] \\
S_{\text{sin}}^{(i)} = \sum_{l=0}^{\frac{N}{2} - 1} \sum_{m=0}^{\frac{L}{N} - 1} \sum_{i=0}^{\frac{L}{N} - 1} W \left[ C_{\text{cos}}^{\text{Im}}(l, m) \cdot \cos \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) + C_{\text{sin}}^{\text{Im}}(l, m) \cdot \sin \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) \right]
\]

Coefficients (9), (10), and (13), (14), (15), (16) completely represent the complex valued signal (3).

IV. RECONSTRUCTION OF COMPLEX-VALUED SIGNAL FROM TIME-FREQUENCY EXPANSION COEFFICIENTS

Because of orthogonality of the basis functions \( \psi_{lm} \), the analysis and synthesis expressions are the same. We synthesize the complex-valued signal (3), for \( i = 0, 1, \ldots, L - 1 \) using coefficients \( C_{\text{cos}}^{\text{Re}}(l, m), C_{\text{cos}}^{\text{Im}}(l, m), C_{\text{sin}}^{\text{Re}}(l, m), C_{\text{sin}}^{\text{Im}}(l, m) \) for

\[
m = 0, 1, \ldots, \frac{L}{N} - 1, l = 0, 1, \ldots, \frac{N}{2},
\]

where \( L \) is number of samples in the signal and \( N \) is the number of samples in \( T \). We require that \( L \equiv 0 \mod(N) \).

Then

\[
S_{\text{Re}}^{(i)} = \sum_{l=0}^{\frac{N}{2} - 1} \sum_{m=0}^{\frac{L}{N} - 1} W \left[ C_{\text{cos}}^{\text{Re}}(l, m) \cdot \cos \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) + C_{\text{sin}}^{\text{Re}}(l, m) \cdot \sin \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) \right]
\]

(17)

\[
S_{\text{Im}}^{(i)} = \sum_{l=0}^{\frac{N}{2} - 1} \sum_{m=0}^{\frac{L}{N} - 1} W \left[ C_{\text{cos}}^{\text{Im}}(l, m) \cdot \cos \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) + C_{\text{sin}}^{\text{Im}}(l, m) \cdot \sin \left( \frac{2\pi}{N} \left( i - \frac{N}{2} \right) \right) \cdot \psi_0(i - mN + \frac{N}{2}) \right]
\]

(18)

From (9) and (11) it follows that there are \( 2 \frac{L}{N} + \frac{L}{N} \cdot \left( \frac{N}{2} - 1 \right) = \frac{L}{2} + \frac{L}{N} \) \( C_{\text{cos}}(l, m) \) coefficients. From (10) and (12) it follows that there are \( \frac{L}{N} + \frac{L}{N} \cdot \left( \frac{N}{2} - 1 \right) = \frac{L}{2} + \frac{L}{N} \) non-zero \( C_{\text{sin}}(l, m) \) coefficients. \( C_{\text{sin}}(l, m) \) coefficients (10) vanish for frequency indexes \( l = 0 \mod(\frac{N}{2}) \). The total number of coefficients is then \( L \).

V. EXAMPLE

For illustration select a signal

\[
S(n\Delta t) = \left\{
\begin{array}{ll}
0, & 0 \leq n \leq 127 \\
\cos \left( \frac{2\pi n}{200} \right), & 128 \leq n \leq 327 \\
\cos \left( \frac{2\pi n}{15.8} \right), & 328 \leq n \leq 602 \\
\cos \left( \frac{2\pi n}{4.7} \right), & 603 \leq n \leq 877 \\
\cos \left( \frac{2\pi n}{2.4} \right), & 878 \leq n \leq 1151 \\
0, & 1152 \leq n < 1280
\end{array}
\right.
\]

(19)
Figure 1. (Left) Reconstruction of (19) using 320 of 1280 coefficients. RMS error is 0.081. (Right) Reconstruction of (19) using all 1280 coefficients. RMS error is 0.017. N=128, n+mN, i=0,1,...,N-1; m=0,1,...,9, L=1280.

Figure 2. (Left) Coefficients $C_{00}(l,m)$ (13). (Right) Coefficients $C_{30}(l,m)$ (15). m = 0, 1, ... 9, l = 0, 1, ... 64.

REFERENCES


3. DISCUSSION AND EXAMPLES IN ONE DIMENSION

3.1. Fast Computation based on the FFT

Like many frequency and time-frequency transforms a straightforward implementation of our transform has computational complexity $O(N^2)$ for a signal of length $N$. Here we present a simple implementation of our discrete transform that is based on the Fast Fourier Transform\(^{13}\). Thus it has complexity $O(N \log N)$. A sample implementation is provided in “ExampleFastComputation.m” Assume we have:

- $f(n) \in \mathbb{C}^L$ (That is, a discrete complex signal of length $L$)
- $f'(t)$, an infinite periodic extension of $f(t)$ such that $f'(n + pL) = f(n)$ with $p \in \mathbb{Z}$
- $N$, the number of discrete samples in $T$ such that $T = N \Delta t$
- $g(t) : \mathbb{C} \to \mathbb{C}$, a window function
- $r \in \mathbb{N}$, a window length factor\(^ {14} \)
- $l \in \mathbb{N}_0$, an index to represent an oscillator at frequency $l/N$, $0 \leq l \leq \frac{N}{2}$
- $m \in \mathbb{Z}$, an index to represent a shift of our time window, $0 \leq m < \frac{L}{N}$

Then the cosine channel coefficient $C_{cos}(l, m)$ is:

$$C_{cos}(l, m) = W \sum_{t=-\frac{trN}{2}}^{\frac{trN}{2}-1} f'(t + mN) g(t) \cos \left( \frac{2\pi lt}{N} - \frac{\pi l}{2} \right)$$

where energy factor $W$ is defined as:

$$W = \begin{cases} \sqrt{\frac{2}{N}} & \text{if } l = 0 \text{ or } l = \frac{N}{2} \\ \frac{2}{\sqrt{N}} & \text{otherwise} \end{cases}$$

\(^ {13} \) Modestly faster implementations could be based on the Fast Cosine and Fast Sine transforms; however here we use the FFT as it is the most widely known of these fast transforms, and FFT routines are readily available in many environments, allowing for quick implementation.

\(^ {14} \) Such that our window function centered at $t = 0$ is assumed to be negligible outside of the interval $[-rN/2, rN/2-1]$. See accompanying paper for formal definition of $r$. 
Consider the trigonometric identity:

\[ \cos(\theta) = \frac{1}{2} \left( e^{i\theta} + e^{-i\theta} \right) \]

This allows us to instead express \( C_{\cos l,m} \) as:

\[
C_{\cos}(l, m) = W \sum_{t=-\frac{lrN}{2}}^{\frac{lrN}{2}-1} f'(t + mN)g(t)(e^{\frac{i2\pi lt}{N}} + e^{\frac{-i2\pi lt}{N}} + e^{\frac{intl}{2}} + e^{\frac{-intl}{2}})
\]

We further rearrange as:

\[
C_{\cos}(l, m) = W \sum_{t=-\frac{lrN}{2}}^{\frac{lrN}{2}-1} \left( e^{\frac{-intl}{2}} f(t + mN)g(t) e^{\frac{i2\pi lt}{N}} + e^{\frac{intl}{2}} f(t + mN)g(t) e^{\frac{-i2\pi lt}{N}} \right)
\]

We introduce \( h_m(t) \) as a notational substitute for \( f() \) and \( g() \) such that:

\[ h_m(t) = f'(t + mN)g(t) \]

Allowing us to finally rearrange as:

\[
C_{\cos}(l, m) = e^{-\frac{intl}{2}} W \sum_{t=-\frac{lrN}{2}}^{\frac{lrN}{2}-1} h_m(t) e^{\frac{i2\pi lt}{N}} + e^{\frac{intl}{2}} W \sum_{t=-\frac{lrN}{2}}^{\frac{lrN}{2}-1} h_m(t) e^{\frac{-i2\pi lt}{N}}
\]

Note that \( e^{-\frac{intl}{2}} \) and \( e^{\frac{intl}{2}} \) are constants with respect to time \( t \) representing phase shifts dependent on frequency index \( l \). This allows us to move them outside the summations over \( t \).
Consider the FFT defined as:

\[ \text{FFT}_{T,k}(x) = \frac{1}{\sqrt{T}} \sum_{t=0}^{T-1} x(t) e^{-i2\pi tk/T} \]

And similarly the IFFT defined as:

\[ \text{IFFT}_{T,k}(x) = \frac{1}{\sqrt{T}} \sum_{t=0}^{T-1} x(t) e^{i2\pi tk/T} \]

where:
- \( x(t) \in \mathbb{C}^T \), a discrete complex signal of length \( T \) to be processed by FFT
- \( T \), the length of the FFT
- \( k \), a discrete frequency returned by FFT, \( 0 \leq k < T \)

We can now define \( C_{\cos l,m} \) in terms of the FFT and IFFT as follows\(^{15}\):

\[ C_{\cos l,m} = e^{-intl} \frac{\sqrt{rN}}{2} \text{IFFT}_{rN,rl}(h_m) + e^{intl} \frac{\sqrt{rN}}{2} \text{FFT}_{rN,rl}(h_m) \]

Similarly, using the trigonometric identity:

\[ \sin(\theta) = \frac{1}{2i} (e^{i\theta} - e^{-i\theta}) \]

\( C_{\sin l,m} \) can be defined as:

\[ C_{\sin l,m} = e^{-intl} \frac{\sqrt{rN}}{2i} \text{IFFT}_{rN,rl}(h_m) - e^{intl} \frac{\sqrt{rN}}{2i} \text{FFT}_{rN,rl}(h_m) \]

Where:

\[ h_m(t) = f'(t + mN + \frac{N}{2}) g(t) \]

\(^{15}\) The inclusion of \( \sqrt{rN} \) cancels the energy factor \( \frac{1}{\sqrt{T}} \) imparted by the FFT/IFFT (\( T = rN \))
3.2. Visualizations

Below, we display and annotate some of the basis functions for various values of frequency index $I$ and time interval index $m$. The implementation of these figures is found in “ExampleShowBasis1D.m”

![Visualizations of 1D Basis Functions](image)

**Figure 1 - Visualizations of 1D Basis Functions**

Observe a few key points regarding these basis functions:

- As frequency index $I$ is increased the oscillation frequency is increased to the next multiple of $\frac{2\pi}{N}$ and phase is shifted by $\frac{\pi}{2}$
- As time index $m$ is increased function is shifted in time by $N$
- Cosine and sine functions with the same time index $m$ are offset from each other by $\frac{N}{2}$ samples
- Sine functions are null for DC frequency ($l = 0$). They are likewise null for $l = \frac{N}{2}$
3.3. Example Applications

Reconstruction, discontinuities and compression

This topic is discussed in chapter 2. The implementation of the figures and concepts is included in “Example1DReconstruction.m”

Noise Reduction\textsuperscript{16}

An implementation of noise reduction in audio is found in “ExampleAudioDenoising.m”\textsuperscript{17}. A brief description is given here, but the reader is recommended to listen to the example as the aural demonstration is more illustrative\textsuperscript{18}.

Typically in audio files noise is evenly distributed throughout the entire frequency spectrum, but “important” information is concentrated in relatively few frequencies. Denoising capitalizes on this by reducing any time-frequency coefficients below a certain threshold. In our demo we load an audio file, add noise to it, and then de-noise it by reducing time-frequency coefficients whose magnitude is below a chosen threshold. By applying such a technique, we are able to increase the signal to noise ratio from approximately 6db (for the noisy signal) to 10db (for the denoised signal).

Examine the time-frequency lattice of the original, noisy, and de-noised files below. The original lattice appears “clean” – energy is concentrated in a few areas of the lattice, and the portions corresponding to individual notes are quite clear. In the noisy lattice, the added energy is easily visible and spread throughout the lattice. Finally in the denoised lattice the appearance is again “clean” and much more similar to the original source.

\textsuperscript{16} Audio sample is a portion of track 8 (violin) taken from the EBU Sound Quality Assessment Material available at http://tech.ebu.ch/publications/sqamcd (See reference section for more info)

\textsuperscript{17} This example requires ltfat library to run for implementation of \texttt{thresh()} function, see [Søn12]

\textsuperscript{18} In this case, a sound might be worth a thousand words...
Figure 2 - Cosine Channel Lattice of Original Audio

Figure 3 - Cosine Channel Lattice of Audio with Noise Added
Figure 4 - Cosine Channel Lattice of Denoised Audio
**VISUALIZING A POLYNOMIAL**

Consider the function:

\[ f(t) = e^{-i2\pi t^3} \]

\( f(t) \) gives us the value of a complex exponential whose phase varies as the cube of time \( t \). As instantaneous frequency is the derivative of phase, a time frequency plot of this function should appear as a quadratic equation.

In “ExamplePolynomial.m” we generate time-frequency lattices for such a function. Examining the figure below, the lattice\(^{19} \) generated by our transform does indeed appear quadratic.

![Time-Frequency Lattice for Polynomial Expression](image)

\(^{19}\) Here the columns of the cosine and sine channels have been interlaced to show the data on a single lattice display
4. DISCUSSION AND EXAMPLES IN TWO DIMENSIONS

4.1. COEFFICIENT EXPRESSIONS

Chapter 2 introduced expressions $C_{\text{cos}}(l, m)$ and $C_{\text{sin}}(l, m)$ for coefficients in one dimension with frequency index $l$ and location index $m$. In two dimensions, we have four channels instead of two and our indices are each two dimensional. Assume we have:

- $f(x, y) \in \mathbb{C}^{L_x \times L_y}$ (a two dimensional finite discrete complex signal of length $L_x$ in one dimension and $L_y$ in the other)
- $f'(x, y)$ an infinite periodic extension of $f(x, y)$ such that $f'(x + pL_x, y + qL_y) = f(x, y)$ with $p, q \in \mathbb{Z}$
- $0 \leq x < L_x, 0 \leq y < L_y$
- $T_x$ and $T_y$, continuous analysis periods
- $N_x$, the number of discrete samples in $T_x$ such that $T_x = N_x \Delta t_x$
- $N_y$, the number of discrete samples in $T_y$ such that $T_y = N_y \Delta t_y$
- $g_x(x), g_y(y) : \mathbb{C} \rightarrow \mathbb{C}$, window functions
- $r_x, r_y \in \mathbb{N}$, window length factors
- $l_x, l_y \in \mathbb{N}_0$, indices to represent an oscillator at two dimensional spatial frequency $\left(\frac{2\pi l_x}{N_x}, \frac{2\pi l_y}{N_y}\right)$ with $0 \leq l_x \leq \frac{N_x}{2}$ and $0 \leq l_y \leq \frac{N_y}{2}$
- $m_x, m_y \in \mathbb{Z}$, indices to represent a shift to location $(m_x N_x, m_y N_y)$ with $0 \leq m_x < \frac{L_x}{N_x}$ and $0 \leq m_y < \frac{L_y}{N_y}$
Then the first channel coefficient – denoted $C_{\text{cos}\times\text{cos}}(l_x, l_y, m_x, m_y)$ – is:

$$
C_{\text{cos}\times\text{cos}}(l_x, l_y, m_x, m_y) = W_x W_y \sum_{x = -\frac{l_x r_y N_x}{2}}^{\frac{l_x r_y N_x}{2} - 1} \sum_{y = -\frac{l_y r_y N_y}{2}}^{\frac{l_y r_y N_y}{2} - 1} f'(x + m_x N_x, y + m_y N_y) g_x(x) g_y(y) \cos \left( \frac{2\pi x l_x}{N_x} - \frac{\pi l_x}{2} \right) \cos \left( \frac{2\pi y l_y}{N_y} - \frac{\pi l_y}{2} \right)
$$

Where energy factors are defined as:

$$W_x = \begin{cases} \sqrt{\frac{2}{N_x}} & \text{if } l_x = 0 \text{ or } l_x = \frac{N_x}{2} \\ \frac{2}{\sqrt{N_x}} & \text{otherwise} \end{cases},
W_y = \begin{cases} \sqrt{\frac{2}{N_y}} & \text{if } l_y = 0 \text{ or } l_y = \frac{N_y}{2} \\ \frac{2}{\sqrt{N_y}} & \text{otherwise} \end{cases}$$

Similarly, we also have a second channel $C_{\text{sin}\times\text{cos}}(l_x, l_y, m_x, m_y)$:

$$
C_{\text{sin}\times\text{cos}}(l_x, l_y, m_x, m_y) = W_x W_y \sum_{x = -\frac{l_x r_y N_x}{2}}^{\frac{l_x r_y N_x}{2} - 1} \sum_{y = -\frac{l_y r_y N_y}{2}}^{\frac{l_y r_y N_y}{2} - 1} f'(x + m_x N_x + \frac{N_x}{2}, y + m_y N_y) g_x(x) g_y(y) \sin \left( \frac{2\pi x l_x}{N_x} - \frac{\pi l_x}{2} \right) \cos \left( \frac{2\pi y l_y}{N_y} - \frac{\pi l_y}{2} \right)
$$

A third channel $C_{\text{cos}\times\text{sin}}(l_x, l_y, m_x, m_y)$:

$$
C_{\text{cos}\times\text{sin}}(l_x, l_y, m_x, m_y) = W_x W_y \sum_{x = -\frac{l_x r_y N_x}{2}}^{\frac{l_x r_y N_x}{2} - 1} \sum_{y = -\frac{l_y r_y N_y}{2}}^{\frac{l_y r_y N_y}{2} - 1} f'(x + m_x N_x, y + m_y N_y + \frac{N_y}{2}) g_x(x) g_y(y) \cos \left( \frac{2\pi x l_x}{N_x} - \frac{\pi l_x}{2} \right) \sin \left( \frac{2\pi y l_y}{N_y} - \frac{\pi l_y}{2} \right)
$$
And a fourth channel $C_{\text{sin}\times\text{sin}}(l_x, l_y, m_x, m_y)$:

$$C_{\text{sin}\times\text{sin}}(l_x, l_y, m_x, m_y) = W_x W_y \sum_{x = \frac{-l_x r_x N_x}{2}}^{\frac{l_x r_x N_x}{2} - 1} \sum_{y = \frac{-l_y r_y N_y}{2}}^{\frac{l_y r_y N_y}{2} - 1}$$

$$f'(x + m_x N_x + \frac{N_x}{2}, y + m_y N_y + \frac{N_y}{2}) g_x(x) g_y(y) \sin\left(\frac{2\pi x l_x}{N_x} - \frac{\pi l_x}{2}\right) \sin\left(\frac{2\pi y l_y}{N_y} - \frac{\pi l_y}{2}\right)$$
4.2. Visualizations

Below we display two dimensional basis functions for various values of frequency indices \( l_x \) and \( l_y \). For all figures, the spatial interval indices are \( m_x = 0 \) and \( m_y = 0 \). The implementation of these figures is found in “ExampleShowBasis2D.m”

First, consider the \( \cos \times \cos \) channel at 2D spatial frequency \( (l_x = 0, l_y = 0) \)

![Figure 6 - \( C_{\cos \times \cos}(0, 0, 0) \) basis function displayed from overhead and from a 3D perspective](image)

Note the following:

- Appears as the product of an envelope function along \( x \) axis and another along \( y \) axis
- The function is centered at \( (0,0) \) for \( m_x = 0 \) and \( m_y = 0 \)
- Other channels are null for 2d spatial frequency \( (l_x = 0, l_y = 0) \)

\[\text{Here 2D basis functions are notated as } \Psi_{c,l_x,l_y,m_x,m_y} \text{ with } c \text{ indicating channel in range 0-3, } l_x \text{ and } l_y \text{ indicating horizontal and vertical frequency index respectively in range } 0-\text{N/2}, \text{ and } m_x \text{ and } m_y \text{ indicating spatial and horizontal shift index in range } 0 - \text{L/N}\]
Next consider frequencies \((l_x = 0, l_y = 4)\) and \((l_x = 4, l_y = 0)\) for the \(\cos \times \cos\) channel

- \((l_x = 0, l_y = 4)\) appears as a vertical oscillator with frequency \(\frac{4 \cdot 2\pi}{N}\) and horizontal and vertical envelopes applied. The \(\cos \times \sin\) channel would appear similar, but offset vertically by \(\frac{N}{2}\). \(\sin \times \cos\) and \(\sin \times \sin\) channels are null for this frequency

- \((l_x = 4, l_y = 0)\) appears similar, but oriented horizontally. The \(\sin \times \cos\) channel would appear similar with a horizontal offset of \(\frac{N}{2}\). \(\cos \times \sin\) and \(\sin \times \sin\) channels are null
Next consider spatial frequency \((l_x = 4, l_y = 4)\) for the \(cos \times cos\) channel.

\(-\hspace{1cm}\)

\((l_x = 4, l_y = 4)\) appears as the product of a pair of oscillators, one vertical and one horizontal. This is in contrast to the Gabor transform in which the function at this frequency would appear as a single oscillator oriented in a diagonal direction.
Next consider spatial frequency \((l_x = 1, l_y = 1\)) for each of the four channels.

\[ \text{Figure 10 - } C_{\cos \times \cos}(1, 1, 0, 0) \text{ basis function displayed from overhead and from a 3D perspective} \]

\[ \text{Figure 11 - } C_{\sin \times \cos}(1, 1, 0, 0) \text{ basis function displayed from overhead and from a 3D perspective} \]
- $sin \times cos$ function is offset in the horizontal direction, centered at $\left(\frac{N}{2}, 0\right)$
- $cos \times sin$ function is offset in the vertical direction, centered at $\left(0, \frac{N}{2}\right)$
- $sin \times sin$ function is offset in the both directions, centered at $\left(\frac{N}{2}, \frac{N}{2}\right)$
- Differences in the channel functions due to phase and application of the envelope are quite noticeable at these relatively low frequencies
Finally consider spatial frequency \((l_x = 8, l_y = 1)\) for the \(cos \times cos\) channel

- Oscillates slowly in the vertical direction, but rapidly in the horizontal direction
4.3. Example applications

Reconstruction & Compression

Our two dimensional transform can be applied to image data to generate a lattice of time frequency coefficients. The image can then be perfectly reconstructed using all of the coefficients, or a close approximation can be created by reconstructing using only the coefficients with the largest magnitude. Creation of these approximations illustrates an approach to image compression as they only require a small portion of the lattice data. The implementation of these concepts and figures is found in “Example2DReconstruction.m”

Consider the original source image below:

![Original Image composed of a pair of cross faded 2D chirp functions](Figure 15)
After performing analysis, we can perfectly reconstruct this image using all of the time-frequency coefficients as shown below:

This reconstruction matches the original\textsuperscript{21} and cannot be visually distinguished from it.

\textsuperscript{21} A negligible amount of error is introduced by imprecision of floating-point operations during processing
We can also choose only a subset of the time-frequency coefficients and reconstruct from those values. Below, a reconstruction using only those coefficients whose magnitude is in the top 10% of all coefficients is shown:

![Chirp function image reconstructed with 10% of coefficients](image.png)

There are some visible differences from the original, particularly in the brightest areas in the upper left and middle right portions of the image, but nevertheless a good approximation has been created while discarding 90% of the original coefficients.
Below we reconstruct using only 3% of the coefficients:

Figure 18 - Chirp function image reconstructed with 3% of coefficients

Here the degradation in quality is readily apparent, but in return we are able to discard the vast majority of the coefficients in the image.
Finally we repeat this exercise using a common reference image. The results are shown below:
Figure 20 - Reference Image Reconstructed Using 100% of Coefficients
Figure 21 - Reference image reconstructed using 10% of coefficients
FIGURE 22 - REFERENCE IMAGE RECONSTRUCTED USING 3% OF COEFFICIENTS
5. DISCUSSION AND EXAMPLES IN HIGHER DIMENSIONS

5.1. EXTENDING BASIS FUNCTIONS TO ANY D DIMENSIONS

Previous chapters presented bases using a pair of basis functions specified for one dimension or a quadruplet of functions specified for two dimensions. Thus in one dimension we had two channels and in two dimensions we had four channels. Here we explain how such sets of functions can be generated for any number of dimensions. Let us suppose the following:

Let our data have dimensionality $D$.

We represent a discrete point in $D$ dimensional space as $(x_0, ..., x_{D-1})$

In each dimension $d$ we define a period $N_d$.

$m_d$ represents a shift index along dimension $d$. $m_d \in \mathbb{Z}$ and we will shift by amount $m_d N_d$

$l_d$ represents a frequency index relative to $N_d$. $l_d \in \mathbb{Z}$ and the frequency will be $\frac{2\pi l_d}{N_d}$

$g_d$ represents a window function in dimension $d$

$W_d$ represents an energy factor. $W_d = \frac{2}{\sqrt{N_d}}$ if $l_x = 0$ or $l_x = \frac{N_d}{2}$, $W_d = \frac{2}{\sqrt{N_d}}$ otherwise

Our basis will have $2^D$ channels. We will use $c$ to denote a channel, with $0 < c \leq 2^D - 1$.

Channels differ by whether they are quadrature or not in each dimension. If a channel is quadrature for a particular dimension, then its oscillators are orthogonally shifted in frequency by $\frac{\pi}{2}$ (becoming a sine carrier instead of a cosine carrier) and shifted in time by $\frac{N_d}{2}$. With $2^D$ channels, every possible combination of quadrature relationships for $D$ dimensions is used.

Let $\text{IsQuad}(c, d)$ be a function determining if a particular channel is quadrature for a particular dimension, with $0 < c \leq 2^D - 1$ and $0 < d \leq D - 1$. The function returns 1 if the channel is shifted and 0 otherwise.

---

22 Note the different frequencies each have a harmonic relationship to carrier $l_d = 1$

23 Or in “space” if the particular data, such as images, is seen spatially

24 This can be easily implemented using bitwise logic. If the “$d^{th}$” bit is set in the binary representation of $c$, then 1 is returned
For an individual basis function $\psi$, we must specify:

- the channel $c$
- the shift index $m_d$ in each dimension
- the harmonic frequency index $l_d$ in each dimension

So each function $\psi_{c,l_0,...,l_{D-1},m_0,...,m_{D-1}}(x_0, ..., x_{D-1}) \in \mathbb{C}$ is specified as:

$$\prod_{d=0}^{D-1} W_d g_d(x_d - m_d \cdot N_d - IsQuad(c, d) \cdot \frac{N_d}{2}) \cdot \cos\left(\frac{x_d \cdot 2 \pi l_d}{N_d} + \frac{\pi l_d}{2} + IsQuad(c, d) \cdot \frac{\pi}{2}\right)$$

For example, for one dimension $\psi_{0,0,1}$ is a cosine term with envelope centered at time zero with frequency $\frac{2\pi}{N}$. $\psi_{1,3,2}$ is sine with envelope centered at time $\frac{7N_d}{2}$ with frequency $\frac{2\pi}{N}$.

Likewise in two dimensions, $\psi_{0,0,0,1,1}$ represents a $\cos \cdot \cos$ term (with frequencies $\frac{2\pi}{N_0}$ and $\frac{2\pi}{N_1}$) centered at $(0,0)$, $\psi_{1,0,0,1,1}$ represents a $\sin \cdot \cos$ term centered at $(\frac{N_0}{2},0)$, $\psi_{2,0,0,1,1}$ represents a $\cos \cdot \sin$ term centered at $(0, \frac{N_1}{2})$, $\psi_{3,0,0,1,1}$ represents a $\sin \cdot \sin$ term centered at $(\frac{N_0}{2}, \frac{N_1}{2})$, and so on.

Each time frequency lattice term is the inner product of such a basis function and the function we are analyzing. Thus the lattice structures used in the code supplied have $2D+1$ dimensions; the channel must be specified as well as the frequency index for each dimension and the time shift index for each dimension.
5.2. **Visualization of 3D Basis Functions**

Below we display three dimensional basis functions, considering the x and y axis as horizontal and vertical dimensions and the z axis as a time dimension. The implementation of this section is found in “ExampleShowBasis3D.m”. In addition, from the code these functions may be viewed as animations rather than as static series of images as shown below. We begin with the $\cos \times \cos \times \cos$ channel at 3D spatial frequency ($l_x = 1, l_y = 1, l_z = 0$).

$Z(t) = -4$

$Z(t) = -3$

$Z(t) = -2$

$Z(t) = -1$

$Z(t) = 0$

$Z(t) = 1$

$Z(t) = 2$

$Z(t) = 3$

**Figure 23 - $\psi_{0,1,0,0,0,0}$ Basis Function Displayed as a Series of Overhead Intensity Images Along the Z Axis**
Observe that this function begins in time as null, increases in magnitude for $\frac{N_x}{2}$ until it appears much like the 2D basis function for frequency ($l_x = 1, l_y = 1$) in the previous chapter. It then decays back to null at time $Z = 3$. This behavior is due to the $l_z$ frequency being zero; over time the function is increasing and decaying over time according to our window function.
Next we consider the $cos \times cos \times cos$ channel at 3D spatial frequency ($l_x = 1, l_y = 1, l_z = 1$).

**Figure 25** - $\Psi_{0,1,1,0,0,0}$ basis function displayed as a series of overhead intensity images along the Z axis.
Note we have changed \( l_z \) from zero to one. The function now increases in magnitude more rapidly for a duration of \( \frac{N_z}{4} \) and then decays to null at time \( Z = 0 \). Then the cycle begins again, with the values inverted – thus the oscillatory nature of the function over the third dimension \( Z \) is apparent.
6. Conclusions

The discrete orthonormal time-frequency basis functions described in this thesis are suitable for various signal processing tasks as shown by the examples presented above. The Matlab implementation verified that the one and two dimensional analytic expressions shown in chapters 2 and 4 could be used successfully. Three dimensional basis functions were successfully generated and displayed using the approach for extension to N dimensions described in section 5.1. The identical results and marked decrease in execution time when using the fast algorithm described in section 3.1 verify its suitability for efficient implementation of this analysis technique.
7. REFERENCES


8. APPENDIX – COMPLETE CODE LISTING

[Electronic version of code is available upon request]

8.1. EXAMPLE 1D RECONSTRUCTION

```matlab
%% Example1DReconstruction.m
% dave chisholm 2013

close all;
clear all;

set(0,'defaulttextinterpreter','latex');
set(0,'DefaultAxesFontName','Courier New');
set(0,'DefaultAxesFontSize',10);

% here choose between pulse and oscillator examples
testType = 'pulse';
% testType = 'multipleOscillators';

% here choose the types of window to use in basis functions
windowsToUse = {'default', 'cosine', 'rectangular'};

% affects how the TF lattices are displayed
aColorMapScale = .1; % lower values give higher contrast; 1 is linear

% select value for N – 32 gives best results
N = 32;

% reconstruct using only the top 25% of coefficients
compressFactor = .75;

if (strcmpi(testType, 'pulse'))
duration = 640;
pointOfStepUp = 150;
pointOfStepDown = 400;

time = [0:duration-1];
testSig = [-1*ones(1, pointOfStepUp) ones(1, pointOfStepDown - pointOfStepUp) -1*ones(1, duration-pointOfStepDown)];
elseif (strcmpi(testType, 'multipleOscillators'))
oscPeriod1 = 200;
oscPeriod2 = 15.8;
oscPeriod3 = 4.7;
oscPeriod4 = 2.4;
oscDuration1 = 200;
oscDuration2 = 275;
oscDuration3 = 275;
oscDuration4 = 274;
oscPhase1 = 0;
oscPhase2 = 0; %.7*pi/4;
oscPhase3 = 0; %.12*pi/4;
```

oscPhase4 = 0; \% 0.96*pi/4;
time = [0:1:oscDuration1 + oscDuration2 + oscDuration3 + oscDuration4 - 1];
testSig = [ zeros(1, N) ... 
sin(2*pi/oscPeriod1*time(1:oscDuration1) + oscPhase1) ... 
sin(2*pi/oscPeriod2*time(oscDuration1+1:oscDuration1 + oscDuration2) + oscPhase2) ... 
sin(2*pi/oscPeriod3*time(oscDuration1 + oscDuration2 + 1:oscDuration1 + oscDuration2 + oscDuration3) + oscPhase3) ... 
sin(2*pi/oscPeriod4*time(end - oscDuration4 + 1:end) + oscPhase4) ... 
zeros(1, N)];

else
  error('Unknown test type');
end

lenInT = numel(testSig) / N;
tickFactor = ceil(lenInT / 10); \% for display

% translate percentage into actual number of coefficients
numCoeffToUse = numel(testSig) - ceil(compressFactor*numel(testSig));

% plot the original signal
HEIGHT = 320;
WIDTH = 480;
FIGURE_LABEL_PAD = .11;
fig = figure('Position', [25, 600, WIDTH, HEIGHT]);
plot(testSig); ylim([-1.1 1.1]); set(gcf, 'color', 'w'); xlim([0 numel(testSig)]);
ylabel('f(n)');

for ii = 1 : numel(windowsToUse)
  windowType = windowsToUse{ii};
  basis = GetBasis([N], [4], 1, windowType);
  windowPosOffset = -(ii-1) * 200;
  lattice = AnalyzeSignal(testSig, basis);

  clims = [0 0];
  clims(1) = min(abs(lattice(:)));
  clims(2) = max(abs(lattice(:)));
  %clims = [0 7];

  % linear indexing vs. coarse/fine grain indexing
% linSig(i) = cfSig(i mod N, i div N)
% cfSig(x, m) = cfSig(m*N + x)
reconstructedSignal = ReconstructSignal(lattice, basis);

% here is where we do the "compression" - basically we copy and then sort
% the coefficients, setting compressFactor% of them to zero, before
reconstruction
    tempLattice = lattice(:);
    [sortedLattice, sortedIndexes] = sort(abs(tempLattice));
    toNullIndexes = sortedIndexes(1:numel(sortedIndexes) - numCoeffToUse);
    tempLattice(toNullIndexes) = 0;
    compressedLattice = tempLattice;
    compressedLattice(:) = tempLattice;
    compressedSignal = ReconstructSignal(compressedLattice, basis);

% calc the root mean squared errors
    [~, rmse_all] = RootMeanSquaredError(testSig, reconstructedSignal);
    [~, rmse_compressed] = RootMeanSquaredError(testSig, compressedSignal);
    fprintf('RMSE using all coefficients is %f. RMSE using %d
coefficients is %f. ', rmse_all, numCoeffToUse, rmse_compressed));

% cos TF lattice
    fig = figure('Position', [100, 500 + windowPosOffset, WIDTH, HEIGHT]);
    GraphLattice(squeeze(lattice(1,:,:)), '', '', 0, aColorMapScale);set(gcf,'color','w');
    h = xla
label({ ... '
    hspacet(0.5in) Time ($m \cdot N$)'; ... '
    sprintf('Cosine channel lattice using %s window', windowType) });
    axpos = get(gca,'pos');
    set(gca,'pos',[axpos(1) (axpos(2)+FIGURE_LABEL_PAD) axpos(3) (axpos(4)-FIGURE_LABEL_PAD)]);

% sine TF lattice
    fig = figure('Position', [400, 500 + windowPosOffset, WIDTH, HEIGHT]);
    GraphLattice(squeeze(lattice(2,:,:)), '', '', .5, aColorMapScale);set(gcf,'color','w');
    h = xla
label({ ... '
    hspacet(0.5in) Time ($m \cdot N$)'; ... '
    sprintf('Sine channel lattice using %s window', windowType) });
    axpos = get(gca,'pos');
    set(gca,'pos',[axpos(1) (axpos(2)+FIGURE_LABEL_PAD) axpos(3) (axpos(4)-FIGURE_LABEL_PAD)]);

% since our TF approach has critical sampling, the total number of
% coefficients is equal to the number of samples. We count it here; note
% that we omit the DC and nyquist frequencies for the sine channel, since
% these are zero by definition
totalNumCoeff = numel(squeeze(lattice(1,:,:))) +
numel(squeeze(lattice(2,2:end-1,:)));

% reconstruction using all coefficients
fig = figure('Position', [400, 500, WIDTH, HEIGHT]);
plot(reconstructedSignal); ylim([-1.1 1.1]); set(gcf,'color','w'); xlim([0
numel(testSig)]);
set(gca,'XTick',[0:tickFactor*N:lenInT*N-1]); tickLabels =
[0:tickFactor:lenInT-1]; set(gca,'XTickLabel',tickLabels);
xlabel('Time (N)'); ylabel('f(n)');
title(sprintf('Reconstruction using all %d coefficients. RMSE is %f',
totalNumCoeff, rmse_all));
h = xlabel({ ...
'\hspace{1.8in} Time ($m \cdot N$)'; ...
'});
axpos = get(gca,'pos');
set(gca,'pos',[axpos(1) (axpos(2)+FIGURE_LABEL_PAD) axpos(3) (axpos(4)-
FIGURE_LABEL_PAD)]);

% compressed reconstruction
fig = figure('Position', [700, 500 + windowPosOffset, WIDTH+35, HEIGHT]);
plot(compressedSignal); ylim([-1.1 1.1]); set(gcf,'color','w'); xlim([0
numel(testSig)]);
set(gca,'XTick',[0:tickFactor*N:lenInT*N-1]); tickLabels =
[0:tickFactor:lenInT-1]; set(gca,'XTickLabel',tickLabels);
xlabel('Time (N)'); ylabel('f(n)');
title(sprintf('Reconstruction using %d of %d coefficients and %s
window. RMSE is %1.3f', numCoeffToUse, totalNumCoeff, windowType, rmse_compressed));
h = xlabel({ ...
'\hspace{1.8in} Time ($m \cdot N$)'; ...
'});
axpos = get(gca,'pos');
set(gca,'pos',[axpos(1) (axpos(2)+FIGURE_LABEL_PAD) axpos(3) (axpos(4)-
FIGURE_LABEL_PAD)]);

end

% window response graphs
% defWin = GetEnvelope(64, 4, 'default');
% cosWin = GetEnvelope(64, 4, 'cosine');
% rectWin = GetEnvelope(64, 4, 'rect');
wvtool(defWin, cosWin, rectWin)
8.2. Example2DReconstruction

```matlab
%% Example2DReconstruction.m
% Demonstrates analysis and reconstruction of 2D images using all
% coefficients (uncompressed) and only most relevant coefficients
% (compression). Can analyze either a standard reference image or an
% image generated from spatially varying patterns
% dave chisholm - 2013

close all;
clear all;

%if true, analysis of a set of patterns automatically generated
%otherwise just the standard boat reference image
usePattern = true;
sampleFilename = 'boat.512.tiff';
compressionRatios = [0, .9, .97 ]; % use all, 10%, 3% respectively

N = 32;

if (usePattern)
    aSignalToAnalyze = Get2DPattern(.0002, .0002, -pi/6, 512);
else
    aSignalToAnalyze = imread(sampleFilename);
    aSignalToAnalyze = double(aSignalToAnalyze) / 255;
    %aSignalToAnalyze = aSignalToAnalyze(129:384,129:384);
end

windowType = 'default';
basis = GetBasis([N, N], [4, 4], 2, windowType);

tic();
lattice = AnalyzeSignal(aSignalToAnalyze, basis);
toc();

figure; set(gcf,'color','w');
imshow(aSignalToAnalyze); title('Original Image');
set(gca,'xtick',[]); set(gca,'xticklabel',[]); set(gca,'yticklabel',[]);
set(gca,'ytick',[]);

for i = 1:numel(compressionRatios)
    tic();
    compressFactor = compressionRatios(i);
    % translate percentage into actual number of coefficients
    numCoeffToUse = numel(aSignalToAnalyze) -
    ceil(compressFactor*numel(aSignalToAnalyze));

    % here is where we do the "compression" - basically we copy and then sort
    % the coefficients, % setting compressFactor% of them to zero, before
    % reconstruction
    tempLattice = lattice(:,);
    [sortedLattice, sortedIndexes] = sort(abs(tempLattice));
    toNullIndexes = sortedIndexes(1:numel(sortedIndexes)-numCoeffToUse);
    tempLattice(toNullIndexes) = 0;
    compressedLattice = lattice;
```
compressedLattice( :) = tempLattice;
compressedSignal = ReconstructSignal(compressedLattice, basis);
figure; set(gcf, 'color', 'w');
imshow(compressedSignal); title(sprintf('Reconstruction using %d%% of coefficients', round((1-compressFactor)*100)));
set(gca, 'xtick', []); set(gca, 'xticklabel', []); set(gca, 'ytick', []); set(gca, 'yticklabel', []);
toc();
end
8.3. Example Audio Denoising

%% ExampleAudioDenoising.m
% loads a sample sound signal, adds noise
% then removes via time-frequency techniques
% dave chisholm 2013

% portions of code based off demo_audiodenoise.m from ltfat.sourceforge.net
% requires ltfat to be installed for "thresh" function

%% after running this file, you may listen to audio via these commands...
% run the line below to hear the original, clean signal
% sound(audioSignal, sampleRate)

% run the line below to hear the signal with noise added
% sound(noisySignal, sampleRate)

% run the line below to hear the signal reconstructed from TF coefficients
% with noise reduction applied
% sound(denoisedAudioSignal, sampleRate)

clear all;
close all;

%% Basic parameters for analysis
% Use glockenspiel sample
% Audio sample is taken from the EBUSound Quality Assessment Material
% available at http://tech.ebu.ch/publications/sqamcd
[audioSignal, sampleRate] = wavread('violin_16_44.wav');

%audioSignal = audioSignal';
N = 1024;
sigLenInSamples = 2^17;

set(0, 'defaulttextinterpreter', 'latex');

noiseStdDev = .5;
Relative_Threshold = 0.25;
tau = Relative_Threshold*noiseStdDev;

basis = GetBasis(N, 4, 1, 'default');

%% truncate the signal to a manageable size evenly divisible by N
audioSignal = audioSignal(1:sigLenInSamples);
numN = floor(sigLenInSamples/N);
audioSignal = audioSignal(1: numN * N);

%% add noise to the signal
noiseStdDev = noiseStdDev * std(audioSignal);
nosySignal = audioSignal + (noiseStdDev * randn(size(audioSignal)));

%% Analyze signals
perfectLattice = AnalyzeSignal(audioSignal, basis);
noisyLattice = AnalyzeSignal(noisySignal, basis);
softThreshLattice = thresh(noisyLattice, tau, 'soft');

%% Reconstruct signals with noise reduction via soft threshholding
denoisedAudioSignal = ReconstructSignal(softThreshLattice, basis);

%% Determine SNRs
InputSNR = 20 * log10(std(audioSignal)/std(noisySignal- audioSignal));
OutputSNR_s = 20 * log10(std(audioSignal)/std(denoisedAudioSignal- audioSignal));

%showOutput = false;
%uncomment this to see various output
showOutput = true;
if (showOutput)
    fprintf(' RESULTS:
');
    fprintf('     Input SNR: %f dB.
', InputSNR);
    fprintf('     Output SNR: %f dB.
', OutputSNR_s);
    fprintf(' Signals are stored in variables audioSignal, noisySignal, denoisedAudioSignal
');
end

figure;
GraphLattice(squeeze(perfectLattice(1,:,:)), '', '', .5, .1);set(gcf,'color','w');
xlabel({'\hspace{0.5in} Time ($m \cdot N$)' ; sprintf('Cosine channel lattice of clean audio')});
figure;
GraphLattice(squeeze(noisyLattice(1,:,:)), '', '', .5, .1);set(gcf,'color','w');
xlabel({'\hspace{0.5in} Time ($m \cdot N$)' ; sprintf('Cosine channel lattice of noisy audio')});
figure;
GraphLattice(squeeze(softThreshLattice(1,:,:)), '', '', .5, .1);set(gcf,'color','w');
xlabel({'\hspace{0.5in} Time ($m \cdot N$)' ; sprintf('Cosine channel lattice of denoised audio')});
8.4. ExampleFastComputation

%% ExampleFastComputation.m
% Creates a sample complex signal, runs both the traditional and the fast
% algorithms to compute lattices, compares the lattices to make sure they
% are equal, displays different running times, optionally shows graphs of
% lattices
% dave chisholm 2013

%% Basic parameters for analysis
N = 128;
numWindows = 10;
time = [0:1:numWindows*N-1];
SHOW_GRAPHS = false;
MAX_TOLERANCE = 10e-8;
r = 4; % window length factor
windowLength = r*N;
testSignal = 0.2*ones(1, 10*N) + 0.5*1j*cos(2*pi*13*time/N);
bufferedSignal = BufferSignal(testSignal, 0, numel(testSignal), 1,
windowLength, N/2, 'buffer');
basis = GetBasis(N, r, 1, 'default');
envelope = GetEnvelope(N, r, 'default');

display('Starting true computation');
tic();
trueLattice = AnalyzeSignal(testSignal, basis);
toc();
display('Finished true computation');

fastLattice = zeros(size(trueLattice));

l = [0:1:N/2];
shift = exp(lj*1*pi/2);
iShift = exp(-lj*1*pi/2);
energyFactor = sqrt(r); % aka (sqrt(r*N)/2) * W

% matlab uses a slightly different definition of FFT/IFFT than our paper
% hence here we mult and div by sqrt(r*N);
fFactor = shift .* energyFactor / sqrt(r*N);
iFactor = iShift .* energyFactor * sqrt(r*N);

% adjust DC and nyquist freqs
fFactor(1) = fFactor(1) / sqrt(2);
ifactor(N/2+1) = ifactor(N/2+1) / sqrt(2);
ifactor(1) = ifactor(1) / sqrt(2); % moot point, since sin chan will be
zero...
ifactor(N/2+1) = ifactor(N/2+1) / sqrt(2); % moot point, since sin chan will be zero...

display('Starting fast computation');
tic();
for curCol = 1:numWindows
cosStart = 1 + (curCol-1)*N;
sinStart = cosStart + N/2;
cosSig = bufferedSignal(cosStart: cosStart+windowLength-1) .* envelope;
sinSig = bufferedSignal(sinStart: sinStart+windowLength-1) .* envelope;

fft_result = fft(cosSig);
ifft_result = ifft(cosSig);
fft_result = fft_result(1:4: (r*N/2)+1);
ifft_result = ifft_result(1:4: (r*N/2)+1);
fastLattice(1, :, curCol) = iFactor.*ifft_result + fFactor.*fft_result;

fft_result = fft(sinSig);
ifft_result = ifft(sinSig);
fft_result = fft_result(1:4: (r*N/2)+1);
ifft_result = ifft_result(1:4: (r*N/2)+1);
fastLattice(2, :, curCol) = (iFactor.*ifft_result - fFactor.*fft_result) * -1j; % mult by -j as we store lattice in form Cos -iSin
end

toc();
display('Finished fast computation');
%trueReconstructedSignal = ReconstructSignal(trueLattice, basis);
%fastReconstructedSignal = ReconstructSignal(fastLattice, basis);

%GraphLattice(trueLattice, '', '')
if (SHOW_GRAPHS)
    GraphLattice(squeeze(trueLattice(1,:,:)), '', '', 0, .1); title('True cos lattice');
    figure;
    GraphLattice(squeeze(trueLattice(2,:,:)), '', '', 0, .1); title('True sin lattice');
    figure;
    GraphLattice(squeeze(fastLattice(1,:,:)), '', '', 0, .1); title('Fast cos lattice');
    figure;
    GraphLattice(squeeze(fastLattice(2,:,:)), '', '', 0, .1); title('Fast sin lattice');
end;
diffLattice = trueLattice - fastLattice;
if (numel(find(abs(diffLattice) > MAX_TOLERANCE)) > 0)
    display('TEST FAILED, significant differences found between lattices')
else
    display('Fast computation test passed')
end
8.5. Example Polynomial

```matlab
%% ExamplePolynomial.m
% Generates a cubic complex exponential and analyzes
% dave chisholm 2013

clear all;
close all;

%% plot some one dimensional basis functions
N = 256;
oneDimBasis = GetBasis(N, 4, 1, 'default');

%% plot TF lattice of a complex exponential cubed
aChirpRate = .00000001;
chirpDelay = 0;
chirpTime = [(20*N)-1];
chirpDelay = .072;
chirpSignal = exp(1i * pi * aChirpRate .* (chirpTime - chirpDelay).^3);
chirpLattice = AnalyzeSignal(chirpSignal, oneDimBasis);
[~, numL, numN] = size(chirpLattice);
interlacedLattice = zeros(numL, 2*numN);
interlacedLattice(:, 1:2:end) = squeeze(chirpLattice(1,:,:));
interlacedLattice(:, 2:2:end) = squeeze(chirpLattice(2,:,:));
figure;
GraphLattice(interlacedLattice, '', '', .5, .1);set(gcf,'color','w');
xlabel({'\hspace{0.5in} Time ($m \cdot N$)'; '' ; sprintf('Cosine channel lattice of noisy audio')});
tickLabels = [-20:5:19];
set(gca,'XTickLabel',tickLabels);
```
8.6. Example Show Basis 1D

% ExampleShowBasis1D.m
% Creates some illustrative plots of various basis functions in 1D
% Dave Chisholm 2013

clear all;
close all;

%% plot some one dimensional basis functions
N = 256;
tic()
oneDimBasis = GetBasis(N, 1, 1, 'default');
toc()
time = [-N/2:1:N-1];
padding = zeros(N/2,1);

longTime = [-N/2:1:2*N-1];
paddingT = zeros(N,1);

figure('Position', [100, 100, 960, 640]);
set(gcf,'color','w');

subplot(4,2,1);
plot(longTime, [squeeze(oneDimBasis(1,1, :)); paddingT; padding], 'b');
title('Cosine Channel Basis Functions with l = 0, m = 0', ...;'FontWeight','bold');
xlabel('Time (N)', 'FontSize',10); ylabel('Cos{\psi}_{0,0}(n)', 'FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = ['   '; ' 0 '; '   '; ' N '; '   '; '2N '];
set(gca,'XTickLabel',tickLabels);
ylim([-2 .2]);

subplot(4,2,3);
plot(longTime, [squeeze(oneDimBasis(1,2, :)); paddingT; padding], 'b');
title('Cosine Channel Basis Functions with l = 1, m = 0', ...;'FontWeight','bold');
xlabel('Time (N)', 'FontSize',10); ylabel('Cos{\psi}_{1,0}(n)', 'FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = ['   '; ' 0 '; '   '; ' N '; '   '; '2N '];
set(gca,'XTickLabel',tickLabels);

subplot(4,2,5);
plot(longTime, [paddingT; squeeze(oneDimBasis(1,3, :)); padding], 'b');
title('Cosine Channel Basis Functions with l = 2, m = 1', ...;'FontWeight','bold');
xlabel('Time (N)', 'FontSize',10); ylabel('Cos{\psi}_{2,1}(n)', 'FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = ['   '; ' 0 '; '   '; ' N '; '   '; '2N '];
set(gca,'XTickLabel',tickLabels);
subplot(4,2,7);
plot(longTime, [paddingT; squeeze(oneDimBasis(1,4, :)); padding], 'b');
title('Cosine Channel Basis Functions with l = 3, m = 1',...
' FontWeight','bold');
xlabel('Time (N)','FontSize',10); ylabel('Cos{\psi}_{3,1}(n)','FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = [' '; ' 0 '; ' '; ' N '; ' '; '2N '];
set(gca,'XTickLabel',tickLabels);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

subplot(4,2,2);
plot(longTime, [padding; squeeze(oneDimBasis(2,1, :)); paddingT], 'r');
title('Sine Channel Basis Functions with l = 0, m = 0',...
' FontWeight','bold');
xlabel('Time (N)','FontSize',10); ylabel('Sin{\psi}_{0,0}(n)','FontSize',10);
set(gca,'XTick',[-N/2:N/2:N]);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = [' '; ' 0 '; ' '; ' N '; ' '; '2N '];
set(gca,'XTickLabel',tickLabels);
ylim([-1.2 .2]);

subplot(4,2,4);
plot(longTime, [padding; squeeze(oneDimBasis(2,2, :)); paddingT], 'r');
title('Sine Channel Basis Functions with l = 1, m = 0',...
' FontWeight','bold');
xlabel('Time (N)','FontSize',10); ylabel('Sin{\psi}_{1,0}(n)','FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = [' '; ' 0 '; ' '; ' N '; ' '; '2N '];
set(gca,'XTickLabel',tickLabels);

subplot(4,2,6);
plot(longTime, [padding; paddingT; squeeze(oneDimBasis(2,3, :))], 'r');
title('Sine Channel Basis Functions with l = 2, m = 1',...
' FontWeight','bold');
xlabel('Time (N)','FontSize',10); ylabel('Sin{\psi}_{2,1}(n)','FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = [' '; ' 0 '; ' '; ' N '; ' '; '2N '];
set(gca,'XTickLabel',tickLabels);

subplot(4,2,8);
plot(longTime, [padding; paddingT; squeeze(oneDimBasis(2,4, :))], 'r');
title('Sine Channel Basis Functions with l = 3, m = 1',...
' FontWeight','bold');
xlabel('Time (N)','FontSize',10); ylabel('Sin{\psi}_{3,1}(n)','FontSize',10);
set(gca,'XTick',[-N/2:N/2:2*N]);
tickLabels = [' '; ' 0 '; ' '; ' N '; ' '; '2N '];
set(gca,'XTickLabel',tickLabels);
8.7. ExampleShowBasis2D

```matlab
% ExampleShowBasis2D.m
% Creates some illustrative plots of various basis functions in 2D
% dave chisholm 2013

close all;
clear all;
tic();
N = 64;
windowType = 'default';

basis = GetBasis([N, N], [1, 1], 2, windowType);
my_max = max(basis(:));
my_min = min(basis(:));

startVals = zeros(1.5*N, 1.5*N);

% first create and display the DC carrier
DCBasisFn = startVals; b0 = startVals; b1 = startVals; b2 = startVals; b3 = startVals;
DCBasisFn(1:N, 1:N) = squeeze(basis(1,1,1,:,:));
Display2DBasisFunction(DCBasisFn, N, my_min, my_max);

% the display DC in one direction, 4/N in the other
b0 = startVals; b1 = startVals;
b0(1:N, 1:N) = squeeze(basis(1,5,1,:,:));
b1(1:N, 1:N) = squeeze(basis(1,1,5,:,:));
Display2DBasisFunction(b0, N, my_min, my_max);
Display2DBasisFunction(b1, N, my_min, my_max);

% now 4/N in each dimension
b0 = startVals;
b0(1:N, 1:N) = squeeze(basis(1,5,5,:,:));
Display2DBasisFunction(b0, N, my_min, my_max);

% now 1/N in each channel
b0 = startVals; b1 = startVals; b2 = startVals; b3 = startVals;
b0(1:N, 1:N) = squeeze(basis(1,2,2,:,:));
%note matlab display considers 1st dimension "vertical"
%so we take some liberties with placing of the channels in the display
%to accomodate reader expectation of x being the first channel and
%horizontal
b1(N/2+1:end, 1:N) = squeeze(basis(3,2,2,:,:));
b2(1:N, 1:N) = squeeze(basis(2,2,2,:,:));
b3(N/2+1:end, N/2+1:end) = squeeze(basis(4,2,2,:,:));
Display2DBasisFunction(b0, N, my_min, my_max);
Display2DBasisFunction(b1, N, my_min, my_max);
Display2DBasisFunction(b2, N, my_min, my_max);
Display2DBasisFunction(b3, N, my_min, my_max);

%finally 1/N in y, 8/N in x
b0 = startVals;
b0(1:N, 1:N) = squeeze(basis(1,2,9,:,:));
```

71
Display2DBasisFunction(b0, N, my_min, my_max);
8.8. ExampleShowBasis3D

```matlab
% ExampleShowBasis3D.m
% Creates some illustrative plots of various basis functions in 3D
% Can be shown as static images or a bit like a "movie"...
% dave chisholm 2013

clear all;
close all;

useAnimation = false;
use3DViewpoint = false;

xN = 32;
yN = 32;
zN = 8;

basis = GetBasis([xN, yN, zN], [1, 1, 1], 3, 'default');

animationDelay = .5;

xyMax = max([xN yN]);
tmpFn = squeeze(basis(1,2, 2, 1, :, :, :));
maxAmp = max(abs(tmpFn(:)));
if useAnimation
    for i = 1:zN
        figure; set(gcf,'color','w');
        mesh(tmpFn(:, :, i));
        axis([0 xyMax 0 xyMax -maxAmp maxAmp]);
        tickLabels = ['-N/2'; ' 0 '; ' N/2'; ' N '];
        set(gca,'XTick',[0:xN/2:3*xN/2]); set(gca,'YTick',[0:yN/2:3*yN/2]);
        set(gca,'XTickLabel',tickLabels); set(gca,'YTickLabel',tickLabels);
        caxis([-maxAmp, maxAmp]);
        pause(animationDelay);
    end
else
    figure('Position', [100, 100, 720, 840]); set(gcf,'color','w');
    my_max = max(tmpFn(:));
    my_min = min(tmpFn(:));
    for i = 1:zN
        subplot(zN/2, 2, i);
        title(sprintf('Z (time) = %d', i-round(zN/2)));
        surface(tmpFn(:, :, i));
        zlim([my_min*1.05 my_max*1.05]);
        tickLabels = ['-N/2'; ' 0 '; ' N/2'; ' N '];
        set(gca,'XTick',[0:xN/2:3*xN/2]); set(gca,'YTick',[0:yN/2:3*yN/2]);
        set(gca,'XTickLabel',tickLabels); set(gca,'YTickLabel',tickLabels);
        caxis([-maxAmp, maxAmp]);
        if use3DViewpoint
            view([-1,-1,.5]);
        end
    end
end
```
end
end

tmpFn = squeeze(basis(1, 2, 2, 2, :, :, :));
maxAmp = max(abs(tmpFn(:))); if useAnimation
   for i = 1:zN
      figure; set(gcf,'color','w');
      mesh(tmpFn(:, :, i));
      axis([0 xyMax 0 xyMax -maxAmp maxAmp]);
      pause(animationDelay);
   end
else
   figure('Position', [100, 100, 720, 840]); set(gcf,'color','w');
   my_max = max(tmpFn(:));
   my_min = min(tmpFn(:));
   for i = 1:zN
      subplot(zN/2, 2, i);
      title(sprintf('Z (time) = %d', i - 1 - round(zN/2)));
      surface(tmpFn(:, :, i));
      zlim([my_min*1.05 my_max*1.05]); %caxis([my_min, my_max]);
      tickLabels = ['-N/2'; ' 0 '; ' N/2'; ' N '];
      set(gca,'XTick',[0:xN/2:3*xN/2]); set(gca,'YTick',[0:yN/2:3*yN/2]);
      set(gca,'XTickLabel',tickLabels); set(gca,'YTickLabel',tickLabels);
      caxis([-maxAmp, maxAmp]);
      if use3DViewpoint
         view([-1,-1,.5]);
      end
   end
end
8.9. AnalyzeSignal

% AnalyzeSignal.m
% returns a lattice given a signal and basis
% lattice will have 2^D+1 dimensions for signal of dimensionality D
% first dimension of lattice specifies the channel and is size 2^D
% next D dimensions are of size N/2+1 and specify the frequency carriers l
% final D dimensions size depends on signal and are the time/space offset
% indexes m
% dave chisholm 2013

function lattice = AnalyzeSignal(aSignal, aBasis)

sigSize = size(aSignal);
numDimensions = sum(sigSize>1);
if (numDimensions == 1)
    aSignal = aSignal(:);
    %force a column vector
    sigSize = size(aSignal);
    %force a column vector
end

basisSize = size(aBasis);
umBasisDimensions = sum(basisSize>1);
if (numBasisDimensions ~= (2*numDimensions+1))
    error('Invalid basis - should be a %dD matrix for %dD signal data,
    instead is a %dD matrix', (2*numDimensions+1), numDimensions,
    numBasisDimensions);
end

numChannels = 2^numDimensions;
if (basisSize(1) ~= numChannels)
    error('Invalid basis - should have % channels for %dD signal data,
    instead has %d channels', numChannels, numDimensions,
    numBasisDimensions, basisSize(1));
end

Ns = zeros(1, numDimensions);
halfNs = zeros(1, numDimensions);
umWindows = zeros(1, numDimensions);
umHarmonics = zeros(1, numDimensions);
windowSizes = zeros(1, numDimensions);
latticesSize = zeros(1, 2*numDimensions + 1);
latticesSize(1) = numChannels;
for curDimension = 1 : numDimensions
    Ns(curDimension) = (basisSize(curDimension+1) - 1) * 2;
    halfNs(curDimension) = fix(Ns(curDimension)/2);
    if ( (~IsIntegerValue(sigSize(curDimension)/Ns(curDimension))) ||
    (sigSize(curDimension)/Ns(curDimension) < 4))
        error('Signal size must be evenly divisible by N in every dimension
    and at least 4N');
    end
    numWindows(curDimension) = fix(sigSize(curDimension)/Ns(curDimension));
    numHarmonics(curDimension) = basisSize(curDimension+1);
    windowSizes(curDimension) = basisSize(curDimension+1+numDimensions);
    latticesSize(curDimension + 1) = numHarmonics(curDimension);
latticesSize(curDimension + 1 + numDimensions) =
numWindows(curDimension);
end

lattice = zeros(latticesSize);

%bufferedSignal contains a copy of the final windowsize/2 worth of data in
each %dimension copied to the begining and end, to handle the "wrap" around when m == 0
bufferedSignal = BufferSignal(aSignal, 0, sigSize, numDimensions, windowSizes, halfNs, 'buffer');
[harmonicCombinations, locationCombinations] =
GetLatticeCombinations(numDimensions, numHarmonics, numWindows);

[lattice, ~] = IterateLattice('Analyze', lattice, bufferedSignal, aBasis, Ns, halfNs, ...
   windowSizes, numDimensions, numChannels, locationCombinations, harmonicCombinations);

% original 1D only implementation left as reference
% if (numDimensions ~= 1)
%   error('TODO - support > 1 dimension');
% else
%   for m = 0:numWindows(1)-1
%     for n = 1:numHarmonics(1)
%       %display(sprintf('m=%d , n=%d', m, n));
%       lattice(1, n, m+1) =
dot(squeeze(conj(bufferedSignal(m*Ns(1)+1:(m+1)*Ns(1)))), squeeze(aBasis(1, n, :)));
%       lattice(2, n, m+1) =
dot(squeeze(conj(bufferedSignal(m*Ns(1)+1+halfNs(1):(m+1)*Ns(1)+halfNs(1)))), squeeze(aBasis(2, n, :)));
%     end
%   end
% end

% scrap
% harmonicLists = cell(1, numDimensions);
% locationLists = cell(1, numDimensions);
% for curDim = 1:numDimensions;
%   harmonicLists{curDim} = [1:numHarmonics(curDim)];
%   locationLists{curDim} = [0:numWindows(curDim)-1];
% end
% harmonicCombinations = CartesianProduct(harmonicLists);
% locationCombinations = CartesianProduct(locationLists);
% numHarmonicCombs = numel(harmonicCombinations);
% numLocationCombs = numel(locationCombinations);

% bufSize = sigSize;
% wrapTargetIndices = cell(1, numDimensions);
% wrapSourceIndices = cell(1, numDimensions);
% remainderTargetIndices = cell(1, numDimensions);
% for (curDim = 1 : numDimensions)
%   bufSize(curDim) = bufSize(curDim) + halfNs(curDim);
% end
% bufferedSignal = zeros(bufSize);
% bufferedSignal(wrapTargetIndices{:}) = aSignal(wrapSourceIndices{:});
% bufferedSignal(remainderTargetIndices{:}) = aSignal;

% latticeIndex = cell(1, 2*numDimensions + 1);
% basisIndex = repmat({':'}, 1, 2*numDimensions + 1);
% curWindowIndex = cell(1, numDimensions);
% for curLocationIndex = 1:numel(locationCombinations)
%   curLocationComb = locationCombinations(curLocationIndex);
%   latticeIndex(numDimensions+2:end) = num2cell(curLocationComb+1);
%   for curChan = 1 : numChannels
%     latticeIndex{1} = curChan;
%     basisIndex{1} = curChan;
%     for curHarmIndex = 1:numel(harmonicCombinations)
%       curHarmComb = harmonicCombinations(curHarmIndex);
%       latticeIndex(2:numDimensions+1) = num2cell(curHarmComb);
%       basisIndex(2:numDimensions+1) = num2cell(curHarmComb);
%       for curDim = 1 : numDimensions
%         if (IsShifted(curChan, curDim))
%           curWindowIndex{curDim} = curLocationComb(curDim)*Ns(curDim)+1:curLocationComb(curDim)*Ns(curDim)+1+halfNs(curDim);
%         else
%           curWindowIndex{curDim} = curLocationComb(curDim)*Ns(curDim)+1+halfNs(curDim):curLocationComb(curDim)*Ns(curDim)+1+halfNs(curDim);
%         end
%       end
%     end
%   end
% end
% lattice(latticeIndex{:}) =
% dot(squeeze(conj(bufferedSignal(curWindowIndex{:}))),
% squeeze(aBasis(basisIndex{:})));
8.10. BufferSignal

%% BufferSignal.m
%% prepends an additional half window of data and appends a half window - % N/2 worth of data. this allows us to handle the signal as if it were % periodically extended.
%% dave chisholm 2013

function [result] = BufferSignal(aSignal, aBufferedSignal, aSigSize, aNumDimensions, aWindowSizes, aHalfNs, aAction)

bufSize = aSigSize;
wrapTargetIndices = cell(1, aNumDimensions);
wrapSourceIndices = cell(1, aNumDimensions);
remainderTargetIndices = cell(1, aNumDimensions);
tailTargetIndices = cell(1, aNumDimensions);
tailSourceIndices = cell(1, aNumDimensions);

for (curDim = 1 : aNumDimensions)

    % total additional size is a full window minus N/2
    bufSize(curDim) = bufSize(curDim) + aWindowSizes(curDim) - aHalfNs(curDim);

    % prepend a half window to buffer
    wrapTargetIndices{curDim} = 1 : (aWindowSizes(curDim) / 2);  % aHalfTs(curDim);
    wrapSourceIndices{curDim} = aSigSize(curDim) - (aWindowSizes(curDim) / 2) + 1 : aSigSize(curDim);

    % copy in the central portion of buffer
    remainderTargetIndices{curDim} = (aWindowSizes(curDim) / 2) + 1 : aSigSize(curDim) + (aWindowSizes(curDim) / 2);

    % append a half window minus N/2 to buffer
    tailTargetIndices{curDim} = aSigSize(curDim) + (aWindowSizes(curDim) / 2) + 1 : bufSize(curDim);
    tailSourceIndices{curDim} = 1 : (aWindowSizes(curDim) / 2) - aHalfNs(curDim);

end

% temp workaround for 2D
% todo, correct dimension independent approach...
if (aNumDimensions == 2)

    if (strcmpi(aAction, 'buffer'))
        result = repmat(aSignal, 3, 3);
        startX = aSigSize(1) - (aWindowSizes(1) / 2) + 1;
        startY = aSigSize(2) - (aWindowSizes(2) / 2) + 1;
        result = result(startX:startX+bufSize(1)-1, startY:startY+bufSize(2)-1);
    elseif (strcmpi(aAction, 'debuffer'))
        sourceStart = 1;

end
sourceEnd = (aWindowSizes(1) / 2);
targetStart = aSigSize(1)+1;
targetEnd = aSigSize(1)+(aWindowSizes(1) / 2);
abufferedSignal(targetStart:targetEnd, :) =
abufferedSignal(targetStart:targetEnd, :) +
abufferedSignal(sourceStart:sourceEnd, :);

sourceStart = bufSize(1)-(aWindowSizes(1) / 2)+ 1 + aHalfNs(1);
sourceEnd = bufSize(1);
targetStart = (aWindowSizes(1) / 2) + 1;
targetEnd = targetStart + (sourceEnd - sourceStart);
abufferedSignal(targetStart:targetEnd, :) =
abufferedSignal(targetStart:targetEnd, :) +
abufferedSignal(sourceStart:sourceEnd, :);

sourceStart = 1;
sourceEnd = (aWindowSizes(2) / 2);
targetStart = aSigSize(2)+1;
targetEnd = aSigSize(2)+(aWindowSizes(2) / 2);
abufferedSignal(:, targetStart:targetEnd) = abufferedSignal(:,
targetStart:targetEnd) + abufferedSignal(:, sourceStart:sourceEnd);

sourceStart = bufSize(2)-(aWindowSizes(2) / 2)+ 1 + aHalfNs(2);
sourceEnd = bufSize(2);
targetStart = (aWindowSizes(2) / 2) + 1;
targetEnd = targetStart + (sourceEnd - sourceStart);
abufferedSignal(:, targetStart:targetEnd) = abufferedSignal(:,
targetStart:targetEnd) + abufferedSignal(:, sourceStart:sourceEnd);

result = abufferedSignal(remainderTargetIndices{:});

else
    error('aAction must be "buffer" or "debuffer"');
end

return;
end

if strcmpi(aAction, 'buffer')
    result = zeros(bufSize);
    result(wrapTargetIndices{:}) = aSignal(wrapSourceIndices{:});
    result(remainderTargetIndices{:}) = aSignal;
    result(tailTargetIndices{:}) = aSignal(tailSourceIndices{:});
else if strcmpi(aAction, 'debuffer')
    result = abufferedSignal(remainderTargetIndices{:});
    result(wrapSourceIndices{:}) = result(wrapSourceIndices{:}) +
    abufferedSignal(wrapTargetIndices{:});
    result(tailSourceIndices{:}) = result(tailSourceIndices{:}) +
    abufferedSignal(tailTargetIndices{:});
else
    error('aAction must be "buffer" or "debuffer"');
end
8.11. CARTESIAN PRODUCT

% CartesianProduct.m
% dave chisholm 2013

% this code is based off a StackExchange coding suggestion by Amro found at:
% http://stackoverflow.com/questions/4165859/matlab-generate-all-possible-
combinations-of-the-elements-of-some-vectors

function result = CartesianProduct(sets)
    if (size(sets) == [1 1])
        result = sets{1}(:);
        return
    end
    c = cell(1, numel(sets));
    [c{:}] = ndgrid( sets{:} );
    result = cell2mat( cellfun(@(v)v(:), c, 'UniformOutput',false) );
end
8.12. Display2DBasisFunction

% Display2DBasisFunction
% Displays overhead and 3D views of basis function in a new window
% dave chisholm 2013

function Display2DBasisFunction(aFunction, aN, aMin, aMax)

XX = 25;
YY = 200;
HEIGHT = 480;
WIDTH = 1080;

% 64x64 surface appears too dense, this looks nicer
if (aN >= 64)
    scale3d = 2;
else
    scale3d = 1;
end

figure('Position', [XX, YY, WIDTH, HEIGHT]); set(gcf, 'color', 'w');

subplot(1,2,1);
surface(aFunction);
zlim([aMin*1.05 aMax*1.05]);
caxis([aMin, aMax]);
tickLabels = ['-N/2'; ' 0 '; ' N/2'; '  N '];
set(gca,'XTick',[0:aN/2:3*aN/2]); set(gca,'YTick',[0:aN/2:3*aN/2]);
set(gca,'XTickLabel',tickLabels); set(gca,'YTickLabel',tickLabels);

subplot(1,2,2);
surface(aFunction(1:scale3d:end, 1:scale3d:end));
zlim([aMin*1.05 aMax*1.05]);
caxis([aMin, aMax]);
tickLabels = ['-N/2'; ' 0 '; ' N/2'; '  N '];
set(gca,'XTick',[0:aN/scale3d/2:3*aN/2]); set(gca,'YTick',[0:aN/scale3d/2:3*aN/2]);
set(gca,'XTickLabel',tickLabels); set(gca,'YTickLabel',tickLabels);
view([-1,-1,.5]);

colorbar('location','southoutside');
8.13. GET2DPATTERN

% Get2DPattern.m
% returns a test image consisting of a pair of "blended" 2D chirp signals
% dave chisholm 2013

function pattern = Get2DPattern(aCircularChirpRate, aDiagChirpRate,
aDiagAngle, aPatchSize)

indexes = [0:1:aPatchSize-1];
[xx, yy] = meshgrid(indexes, indexes);

circleDistances = sqrt(xx.^2 + yy.^2);
circles = cos(2*pi*aCircularChirpRate*circleDistances.^2);

diagDistances = cos(aDiagAngle)*yy + sin(aDiagAngle)*xx;
diags = cos(2*pi*aDiagChirpRate*diagDistances.^2);

xxLeft = xx;
xxLeft(xxLeft<100) = 1; xxLeft(xxLeft>=100) = ((aPatchSize-101) -
(xxLeft(xxLeft>=100) - 100)) / (aPatchSize-101);
circles2 = [xxLeft.*circles zeros(aPatchSize, 128)];

xxRight = xx;
xxRight(xxRight<(aPatchSize-100)) = xxRight(xxRight<(aPatchSize-100)) / (aPatchSize-101);
xxRight(xxRight>=(aPatchSize-100)) = 1;
diags2 = [zeros(aPatchSize,128) xxRight.*diags];

pattern = diags2 + circles2;
pattern = pattern / max(abs(pattern(:)));

% GetBasis.m
% Returns a basis given the analysis parameters
% different values of N, window widths and window types may be used for
% different dimensions
% returned basis will have 2*aNumDimensions+1 dimensions
% first dimension specifies channel and is size 2^aNumDimensions
% next aNumDimensions specify frequency and are each size aNs(dim)/2
% final aNumDimensions specify frequency and are each size
% aWindowWidthsInN(dim)*aNs(dim)
% dave chisholm 2013

function basis = GetBasis(aNs, aWindowWidthsInN, aNumDimensions, aWindowType)
    if (~IsIntegerValue(aNumDimensions) || (aNumDimensions < 1))
        error('Number of dimensions must be an integer > 1');
    end
    numChannels = 2^aNumDimensions;
    numHarmonics = zeros(1, aNumDimensions);
    halfNs = zeros(1, aNumDimensions);

    [numNs, ~] = size(aNs(:));
    if (numNs ~= aNumDimensions)
        if (numNs == 1)
            aNs = aNs * ones(1, aNumDimensions);
        else
            error(sprintf('Supplied %d values for N; must supply either one
value, or one N for each dimensions (%d dimensional data supplied)', numNs, aNumDimensions));
        end
    end

    [numWidths, ~] = size(aNs(:));
    if (numWidths ~= aNumDimensions)
        if (numWidths == 1)
            aWindowWidthsInN = aWindowWidthsInN * ones(1, aNumDimensions);
        else
            error(sprintf('Supplied %d values for window widths; must supply
either one value, or one for each dimensions (%d dimensional data supplied)', numWidths, aNumDimensions));
        end
    end

    % todo support multiple window types for multidimensional data
    % [numWindowTypes, ~] = size(aWindowTypes(:));
    % if (numWindowTypes ~= aNumDimensions)
    %     if (numWindowTypes == 1)
    %         tmpWin = aWindowTypes;
    %         for i = 1:aNumDimensions
    %             aWindowTypes(i) = tmpWin;
    %         end
    %     else
    % else
for (i = 1:aNumDimensions)
    if (~IsIntegerValue(log2(aNs(i))))
        error('Length of analysis window N must be a power of two in every dimensions; N for dimension %d is size %d', i, aNs(i));
    end
    halfNs(i) = fix(aNs(i)/2);
    numHarmonics(i) = fix(aNs(i)/2 + 1);
end

basisSize = zeros(1, 2*aNumDimensions + 1);
basisSize(1) = numChannels;
basisFnSize = zeros(1, aNumDimensions);
for curDimension = 1 : aNumDimensions
    basisSize(curDimension+1) = numHarmonics(curDimension);
    basisSize(curDimension+1+aNumDimensions) = aWindowWidthsInN(curDimension) * aNs(curDimension);
    basisFnSize(curDimension) = aWindowWidthsInN(curDimension) * aNs(curDimension);
end

basis = zeros(basisSize);
bases1D = cell(numChannels, aNumDimensions);
for curDimension = 1 : aNumDimensions
    curN = aNs(curDimension);
    curWidthInN = aWindowWidthsInN(curDimension);
    curNumHarmonics = numHarmonics(curDimension);
    curEnv = GetEnvelope(curN, curWidthInN, aWindowType);
    for (curChannel = 0 : numChannels-1)
        if (IsShifted(curChannel, curDimension-1)) %subtract 1 due to zero versus one based indexing
            %curTime = 0:1:aNs(curDimension)-1;
            curTime = halfNs(curDimension) - (halfNs(curDimension) * aWindowWidthsInN(curDimension)) : 1 : halfNs(curDimension) + (halfNs(curDimension) * aWindowWidthsInN(curDimension)) - 1;
            phaseShift = -pi/2;
        else
            %curTime = -halfNs(curDimension):1:halfNs(curDimension)-1;
            curTime = -halfNs(curDimension) : 1 : halfNs(curDimension) * aWindowWidthsInN(curDimension) : 1 :
            phaseShift = 0;
        end
        tmpBasis1D = zeros(curNumHarmonics, curN * curWidthInN);
        for curHarmonic = 0 : curNumHarmonics-1
            if (mod(curHarmonic, halfNs(curDimension)) == 0)
                energyFactor = sqrt(2/aNs(1));
            else
energyFactor = 2/sqrt(aNs(1));
end
tmpBasis1D(curHarmonic+1, :) = energyFactor * cos(2*pi*curTime*curHarmonic/curN + curHarmonic*pi/2 + phaseShift) .* curEnv;
end
bases1D{curChannel+1, curDimension} = tmpBasis1D;
end
end

if (aNumDimensions == 1)
basis(1,:,:) = bases1D{1};
basis(2,:,:) = bases1D(2);
else
harmonicLists = cell(1, aNumDimensions);
for curDim = 1:aNumDimensions
    harmonicLists{curDim} = [1:numHarmonics(curDim)];
end
harmonicCombinations = num2cell(CartesianProduct(harmonicLists), 2);
indices = repmat({':'}, 1, 2*aNumDimensions + 1);
for curChan = 1:numChannels
    indices{1} = curChan;
    for curCombIndex = 1 : numel(harmonicCombinations)
curBasisFn = ones(basisFnSize);
curComb = harmonicCombinations{curCombIndex};
    for curDim = 1:aNumDimensions
        indices{curDim + 1} = curComb(curDim);
curDimBasisFns = bases1D(curChan, curDim);
curDimBasisFn = curDimBasisFns{:}(curComb(curDim),:);
        repetitionSize = basisFnSize;
        repetitionSize(curDim) = 1;
        reshapedFnSize = ones(1,aNumDimensions);
        reshapedFnSize(curDim) = numel(curDimBasisFn);
        reshapedCurDimBasisFn = reshape(curDimBasisFn, reshapedFnSize);
        curBasisFn = curBasisFn .* repmat(reshapedCurDimBasisFn, repetitionSize);
    end
    basis(indices{ (: )}) = curBasisFn;
end
end
end
8.15. GetEnvelope

%% GetEnvelope.m
% Returns an "envelope" or window function according to the specified parameters
% dave chisholm 2013

function envelope = GetEnvelope(aN, aDurationInN, aWindowType)

if (~strcmpi(aWindowType, 'default'))
    len = aDurationInN * aN;
    padLen = (len - aN) / 2;
    if (strcmpi(aWindowType, 'cosine'))
        x = [0:aN-1];
        middleEnvelope = sin(pi*x/(aN));
    elseif ((strcmpi(aWindowType, 'rect')) || (strcmpi(aWindowType, 'rectangular')))
        middleEnvelope = sqrt(.5) * ones(1, aN);
    elseif (strcmpi(aWindowType, 'hamming'))
        middleEnvelope = window(@hamming, aN)';
    elseif (strcmpi(aWindowType, 'blackman'))
        middleEnvelope = window(@blackmanharris, aN)';
    else
        error('Unsupported window type, set aWindowType parameter to
        "default", "cosine" or "rect"');
    end
end

envelope = [zeros(1, padLen) middleEnvelope zeros(1, padLen)];
return

[end, rawN, rawDurationInN] = GetRawEnvelope();

%trim edges if needed
if (rawDurationInN < aDurationInN)
    error('Maximum supported window length in N is %d', rawDurationInN);
end

%scale to new N
if (rawN ~= aN)
    skipVal = rawN / aN;
    if (~IsIntegerValue(skipVal))
        error('Value for N must evenly divide %d but %d was supplied', rawN,
        aN);
    end
    envelope = envelope(1:skipVal:end);
end

trimSize = ((rawDurationInN - aDurationInN) * aN) / 2;
if (trimSize ~= 0)
    envelope = envelope(1+trimSize:end-trimSize);
end
%% Look into the code below for energy normalization
%normalize so that energy of overlapping envelopes is 1
energy = 0;
for i = 0 : aDurationInN-1
    energy = energy + (envelope(i*aN+1)^2 + envelope(i*aN+1 + (aN/2))^2);
end

envelope = envelope / sqrt(energy);
8.16. **GetLatticeCombinations**

```matlab
function [harmonicCombinations, locationCombinations] = GetLatticeCombinations(aNumDimensions, aNumHarmonics, aNumWindows)

harmonicLists = cell(1, aNumDimensions);
locationLists = cell(1, aNumDimensions);
for curDim = 1:aNumDimensions;
    harmonicLists{curDim} = [1:aNumHarmonics(curDim)];
    locationLists{curDim} = [0:aNumWindows(curDim)-1];
end
harmonicCombinations = CartesianProduct(harmonicLists);
locationCombinations = CartesianProduct(locationLists);
```

% GetLatticeCombinations.m
% determines all combinations of frequency and time indices and wraps them
% in a cell
% dave chisholm 2013
8.17. **GetRawEnvelope (abbreviated)**

```matlab
% GetRawEnvelope.m
% Generated via regexps from a flat text file containing sample window
% dave chisholm 2013

function [envelope, N, durationInN] = GetRawEnvelope()
% June 8 2013
% currently contains values for:
% T = 1.121
% deltaT = T/1024 = 0.00109
% first val = 6.88679*10^-28
% last val = 7.32259*10^-28

N = 1024;
durationInN = 4;

[ ~4000 lines omitted for brevity...]
8.27866*10^-28 7.78597*10^-28 7.32259*10^-28
'];

if (numel(envelope) ~= N * durationInN)
    error('Invalid raw data, incorrect number of samples');
end
```
function GraphLattice( ...
    aLattice,...
    aXLabel,...
    aYLabel,...
    aXAxisShift,...
    aScale)

ramp = nthroot([1:-.01:0]', aScale); %accent smaller values
ramp = [1:-.01:0]'; %linear colormap
colormapToUse = [ramp ramp ramp];

if (strcmp(aXLabel, ''))
    aXLabel = 'Time (m \cdot N)';
end

if (strcmp(aYLabel, ''))
    aYLabel = 'Freq (2\cdot pi\cdot n/N),'
    aYLabel = 'Freq ($\frac{2 \cdot \pi \cdot n}{N}$)';
end

[aTitle = sprintf('%s (T = %g)', aTitle, aT);

[numRows, numCols] = size(aLattice);
x = (0+aXAxisShift:1:numCols-1+aXAxisShift);
y = (0:1:numRows-1);
%if (aTitle ~= '')
%    figure;
%end
%imagesc(x,y,flipud(arrayfun(@norm, interlacedLattice)));
imagesc(x,y,arrayfun(@norm, aLattice));
%if (aTitle ~= '')
%    title(aTitle, 'FontWeight','bold');
%end
xlabel(aXLabel);
ylabel(aYLabel);
colormap(colormapToUse);
set(gca,'YDir','normal');
8.19. IsIntegerValue

function [ isIntegerValue ] = IsIntegerValue(aVal)
    isIntegerValue = rem(aVal, 1) == 0;
end
8.20. IsQuad

Function: IsQuad

This function determines if, for a given dimension, a specified channel is quadrature or not. Both the channel number and dimension should be zero indexed (i.e., 0 to \(D^2-1\) for channels, 0 to \(D-1\) for dimensions).

```matlab
function value = IsQuad(aChannelNum, aDimension);
iChan = uint32(aChannelNum);
iDim = uint32(1);
iDim = bitshift(iDim, aDimension);
value = (bitand(iChan, iDim) ~= uint32(0));
```
8.21. IterateLattice

% IterateLattice.m
% code to traverse the lattice is nearly identical for analysis and
% reconstruction
% only a single operation changes, so we share the code in this helper
% function
% dave chisholm 2013

function [lattice, bufferedSignal] = IterateLattice( aAction, ...
    aLattice, aBufferedSignal, aBasis, aNs, aHalfNs, aWindowSizes, ...
    aNumDimensions, aNumChannels, aLocationCombinations,
    aHarmonicCombinations)

%SHOW_PROGRESS_MESSAGES = true;
SHOW_PROGRESS_MESSAGES = false;

shouldReconstruct = strcmpi(aAction, 'Reconstruct');
if (strcmpi(aAction, 'Analyze'))
    lattice = aLattice;
    bufferedSignal = 0;
elseif (shouldReconstruct)
    bufferedSignal = aBufferedSignal;
    lattice = 0;
else
    error('Action must be "Analyze" or "Reconstruct"');
end

latticeIndex = cell(1, 2*aNumDimensions + 1);
basisIndex = repmat({':'}, 1, 2*aNumDimensions + 1);
curWindowIndex = cell(1, aNumDimensions);
[numLocComb, ~] = size(aLocationCombinations);
%for curLocationIndex = 1:numel(aLocationCombinations)
for curLocationIndex = 1:numLocComb
    if SHOW_PROGRESS_MESSAGES
        display(sprintf('Iterate lattice %d of %d', curLocationIndex,
            numLocComb));
    end
    curLocationComb = aLocationCombinations(curLocationIndex, :);
    latticeIndex(aNumDimensions+2:end) = num2cell(curLocationComb+1);
    for curChan = 1 : aNumChannels
        latticeIndex{1} = curChan;
        basisIndex{1} = curChan;
        for curHarmIndex = 1:numel(aHarmonicCombinations)/aNumDimensions
            curHarmComb = aHarmonicCombinations(curHarmIndex, :);
            latticeIndex(2:aNumDimensions+1) = num2cell(curHarmComb);
            basisIndex(2:aNumDimensions+1) = num2cell(curHarmComb);
            for curDim = 1 : aNumDimensions
                if (IsQuad(curChan-1, curDim-1))
                    start = curLocationComb(curDim)*aNs(curDim)+1+aHalfNs(curDim);
                else
                    start = curLocationComb(curDim)*aNs(curDim)+1;
                end
                stop = start + aWindowSizes(curDim) -1;
                curWindowIndex{curDim} = start:stop;
end
if (shouldReconstruct)
    bufferedSignal(curWindowIndex{:}) = bufferedSignal(curWindowIndex{:}) + aLattice(latticeIndex{:}) * squeeze(aBasis(basisIndex{:}));
else
    %a = squeeze(aBasis(basisIndex{:}));
    %b = squeeze(conj(aBufferedSignal(curWindowIndex{:})));
    %lattice(latticeIndex{:}) = dot(a, b);
    lattice(latticeIndex{:}) = sum(dot(squeeze(conj(aBufferedSignal(curWindowIndex{:}))), squeeze(aBasis(basisIndex{:}))));
end
end
end
% ReconstructSignal.m
% synthesizes signal from a lattice and basis
% dave chisholm 2013

function [reconstructedSignal] = ReconstructSignal(aLattice, aBasis)

basisSize = size(aBasis);
latticeSize = size(aLattice);
if (sum(basisSize>1) ~= sum(latticeSize>1))
    error('Basis and lattice must have same number of dimensions')
end

numDimensions = (sum(basisSize>1)-1)/2;
if ((~IsIntegerValue(numDimensions)) || (numDimensions < 1))
    error('Basis has wrong number of dimensions, should have 2*nDim+1, where
nDim is the dimensionality of the signal to be reconstructed');
end

numChannels = 2^numDimensions;
if (numDimensions == 1)
    sigSize = ones(1, 2);
else
    sigSize = zeros(1, numDimensions);
end

numWindows = zeros(1, numDimensions);
Ns = zeros(1, numDimensions);
halfNs = zeros(1, numDimensions);
numHarmonics = zeros(1, numDimensions);
windowSizes = zeros(1, numDimensions);
for curDim = 1:numDimensions
    numWindows(curDim) = latticeSize(1+numDimensions+curDim);
    Ns(curDim) = (basisSize(curDim+1) - 1) * 2;
    halfNs(curDim) = Ns(curDim) / 2;
    if (~IsIntegerValue(log2(Ns(curDim))))
        error('Malformed basis - length of N must be a power of two in every
dimensions; N for dimension %d is size %d', curDim, Ns(curDim));
    end
    numHarmonics(curDim) = latticeSize(1+curDim);
    if (numHarmonics(curDim) ~= halfNs(curDim) + 1)
        error('Malformed lattice - should have (T/2+1) harmonics in every
dimension, dimension %d has N = %d but %d harmonics', curDim, Ns(curDim),
numHarmonics(curDim));
    end
    sigSize(curDim) = Ns(curDim) * numWindows(curDim);
    windowSizes(curDim) = basisSize(curDim+1+numDimensions);
end

bufferedSignal = BufferSignal(zeros(sigSize), 0, sigSize, numDimensions,
windowSizes, halfNs, 'buffer');
%bufferedSignal = BufferSignal(aSignal, 0, sigSize, numDimensions,
windowSizes, halfNs, 'buffer');
[harmonicCombinations, locationCombinations] =
GetLatticeCombinations(numDimensions, numHarmonics, numWindows);

[~, bufferedSignal] = IterateLattice('Reconstruct', aLattice, bufferedSignal,
aBasis, Ns, halfNs,
    windowSizes, numDimensions, numChannels, locationCombinations,
    harmonicCombinations);

reconstructedSignal = BufferSignal(0, bufferedSignal, sigSize, numDimensions,
    windowSizes, halfNs, 'debuffer');
function [sigError, rmse] = RootMeanSquaredError(a, b)
    sigError = sqrt((a(:) - b(:)).^2);
    rmse = sqrt(sum(sigError(:))/numel(sigError));
8.24. TestOrthonormality

%% TestOrthonormality.m
% Computationally checks orthonormality of a basis
% dave chisholm 2013

clear all;
%script computes dot product of each dual channel function and all adjacent
%and coincident dual channel functions to make sure they are orthogonal
%also checks to make sure energy is equal to one in each channel of
%function

windowType = 'default';
%windowType = 'cosine';
%windowType = 'rect';

%disable this to turn off out of values that are in expected range
verbose = true;
%verbose = false;

N = 32;
todo, investigate why adjacent basis fns of same harmonic have dot prod
values like 0.00039, 0.000776 and increase (slightly) with wider envelopes
%strange
APPROX_ZERO = .001; %used for checking dot products to determine
orthogonality - quick and dirty solution to floating point error

%basis = GetBasis([N], [4], 1, windowType);
basis = GetBasis([N], [2], 1, windowType);
[numChan, numHarm, fnLen] = size(basis);

padding = zeros(1, N/2)';

for i = 1:numHarm
    %pad up front because adjacent channel 2 will start ahead of this
    function
    %pad behind because adjacent channel 1 will end after this function
    cosBasis = [padding; squeeze(basis(1,i,:)); padding; padding];
    sinBasis = [padding; padding; squeeze(basis(2,i,:)); padding];
    dp = dot(cosBasis, sinBasis);
    if (abs(dp > APPROX_ZERO))
        display(sprintf('ERROR*** Channels of basis fn %d is not orthogonal
to itself, dot product is %g', i-1, dp));
    else
        if (verbose), display(sprintf('DP for fn %d and itself = %g', i-1, dp));
    end
end

energy = sqrt(sum(cosBasis.^2));
if (energy > 1 + APPROX_ZERO) | (energy < 1 - APPROX_ZERO)
    display(sprintf('ERROR*** Energy for fn %d chan1 is %g', i-1, energy))
else
if (verbose), display(sprintf('Energy for fn %d chan1 is %g', i-1, energy)); end

if (i ~= 1) && (i ~= N/2+1)
    energy = sqrt(sum(sinBasis.^2));
    if (energy > 1 + APPROX_ZERO) | (energy < 1 - APPROX_ZERO)
        display(sprintf('ERROR*** Energy for fn %d chan2 is %g', i-1, energy))
    else
        if (verbose), display(sprintf('Energy for fn %d chan2 is %g', i-1, energy)); end
    end
end

curBasis = cosBasis - 1j*sinBasis;

for j = i:N/2+1
    if (j ~= i)
        cosBasis = [padding; squeeze(basis(1,j,:)); padding; padding];
        sinBasis = [padding; padding; squeeze(basis(2,j,:)); padding];
        testBasis = cosBasis - 1j*sinBasis;
        dp = dot(curBasis, testBasis);
        if (abs(dp > APPROX_ZERO))
            display(sprintf('ERROR*** Basis fn %d is not orthogonal to %d, dot product is %g', i-1, j-1, dp));
        else
            if (verbose), display(sprintf('DP for fn %d and %d = %g', i-1, j-1, dp)); end
        end
    end
end

cosBasis = [padding; padding; padding; squeeze(basis(1,j,:))];
sinBasis = [squeeze(basis(2,j,:)); padding; padding; padding];

testBasis = cosBasis - 1j*sinBasis;

dp = dot(curBasis, testBasis);
if (abs(dp > APPROX_ZERO))
    display(sprintf('ERROR*** Basis fn %d is not orthogonal to adjacent %d, dot product is %g', i-1, j-1, dp));
else
    if (verbose), display(sprintf('DP for fn %d and adjacent %d = %g', i-1, j-1, dp)); end
end
end
return