THESIS

DEVELOPMENT OF A SENSORY SUBSTITUTION API

Submitted by

Marco Martinez

Department of Mechanical Engineering

In partial fulfillment of the requirements for

The Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2018

Master's Committee:

Advisor: John Williams
Co-Advisor: Leslie Stone-Roy

David Alciatore
Ross McConnell

Copyright by Marco Martinez, 2018

All Rights Reserved

ABSTRACT

DEVELOPMENT OF A SENSORY SUBSTITUTION API

Sensory substitution – or the practice of mapping information from one sensory modality to another – has been shown to be a viable technique for non-invasive sensory replacement and augmentation. With the rise in popularity, ubiquity, and capability of mobile devices and wearable electronics, sensory substitution research has seen a resurgence in recent years. Due to the standard features of mobile/wearable electronics such as Bluetooth, multicore processing, and audio recording, these devices can be used to drive sensory substitution systems. Therefore, there exists a need for a flexible, extensible software package capable of performing the required real-time data processing for sensory substitution, on modern mobile devices. The primary contribution of this thesis is the development and release of an Open Source Application Programming Interface (API) capable of managing an audio stream from the source of sound to a sensory stimulus interface on the body. The API (named *Tactile Waves*) is written in the Java programming language and packaged as both a Java library (JAR) and Android library (AAR). The development and design of the library is presented, and its primary functions are explained. Implementation details for each primary function are discussed. Performance evaluation of all processing routines is performed to ensure real-time capability, and the results are summarized. Finally, future improvements to the library and additional applications of sensory substitution are proposed.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the continued support and guidance of many generous mentors and contributors. First and foremost, I would like to thank my parents and brothers for always believing in me, encouraging me, and trusting in my vision. I owe all my accomplishments to them. Additionally, I thank my mentor Kyle Krabtree for his advice, honesty, and wisdom at any hour of the day.

I would not be involved in this project without the influence, insight, and leadership of JJ Moritz at Sapien LLC. My work stands on the shoulders of his research and design and he has made endless contributions to both this thesis and the field of sensory substitution, for which I give many thanks. Similarly, my advisors Dr. John Williams and Dr. Leslie Stone-Roy laid the groundwork for this project and guided me through the program. I cannot thank them enough for their patience, assistance, and hard work.

I would also like to thank the members of my thesis committee, Dr. Dave Alciatore and Dr. Ross McConnell for their time and efforts in seeing me through this program. Lastly, I thank CSU Ventures for providing the funding that made this project possible, as well as the students, faculty, and friends that have contributed to this thesis both directly and indirectly throughout the course of my time at CSU.

TABLE OF CONTENTS

1. Introduction 1
  1.1. Sensory Substitution 2
  1.2. Mapping Sound to Touch 3
  1.3. Tactile Waves 3
  1.4. Summary 5
2. Real-Time Digital Signal Processing 7
3. Development 15
  3.1. Programming Environment 15
  3.2. Implementation 16
  3.3. Object Oriented Programming – A Technical Primer 16
  3.4. Tactile Waves Package Overview 20
  3.5. The Audio Engine 21
  3.6. The Toolbox 29
  3.7. Utilities 64
  3.8. COM 71
4. Testing & Validation 76
  4.1. Unit Testing 76
  4.2. Performance Testing 82
5. Conclusion 90
  5.1. Future Work 91
  5.2. Final Thoughts 93
References 94

1. Introduction

Worldwide, around 1.3 billion people suffer from hearing impairment, with an estimated 360 million of those afflictions severe enough to be considered disabling [1]. Although a variety of treatments exist, such as amplification devices, implants, and speech therapy, many individuals remain untreated due to severe damage of the auditory pathway, medical risk factors, and/or monetary limitations. Amplification devices, such as in/over-ear hearing aids, work by applying selective gain adjustments to certain frequency ranges in the audio spectrum to compensate for the wearer's audiogram. When properly configured, amplification devices are an effective solution for patients with only partial hearing loss, such as noise induced and age-related hearing loss. Unfortunately, many people who are fitted with these products do not use them, commonly due to device value, fit/comfort, and maintenance of the device [2]. While some of these reports can be attributed to the quality of the hearing aid device itself, individual biases such as social stigma, perception of the amplified sound, and monetary or motivational obstacles with the device maintenance also play a role [2]. For those with more severe loss, implants are used to bypass damaged or non-functioning structures within the auditory pathway. For example, Cochlear implants (manufactured by The Cochlear Corporation), are used to treat individuals with severe bilateral hearing loss, and attempt to collect and process audio signals and directly stimulate the auditory nerve. These devices are highly effective for persons with both pre-lingual and post-lingual hearing loss, although the age of implantation influences success in prelingually deafened patients [3-4]. Furthermore, many who could benefit from an implant are unable to afford the device and procedure, or simply do not wish to undergo an invasive surgery with risk of complication or risk jeopardizing their identity in the deaf community. The obstacles to implants are worsened in developing countries lacking trained medical personal able to perform these operations

and servicing the device and care for patients. There is a need for an affordable, accessible, and non-invasive treatment capable of catering to a wide range of people suffering from hearing afflictions.

By reducing audio information from spoken language into simplified digital representations, language can be projected tactilely to virtually any location on the body through the use of tactile stimulation hardware. The goal of the Tactile Waves software package is to provide a research tool to facilitate the evaluation of encoding schemes to better understand how to best provide auditory information to the brain through the somatosensory pathway.

1.1. Sensory Substitution

The fundamental idea behind sensory substitution is simple: information normally received by one sensory organ can be transmitted to the brain through a different sensory organ. Sensory substitution takes advantage of the plasticity of the human brain: with training the brain can learn to interpret meaningful signals presented controllably through a non-conventional sensory pathway. The most successful example of sensory substitution is a blind person's use of their sense of touch to acquire information that would normally be acquired through their eyes. Braille is used to communicate written text through the nerve receptors in the fingertips, while vibrations transmitted through a walking cane yield information about the surrounding world. Sign language and Closed Captioning can be considered examples of sensory substitution as well, as a deaf user is able to use their sense of sight to acquire lingual information that would normally be spoken or played aloud.

Since the 1960's sensory substitution research has been used to convey information to a subject's brain that would otherwise have been lost. With the use of electrotactile or vibrotactile stimulation, information traditionally acquired through one sensory pathway can be transmitted through the somatosensory pathway. Researchers have used the sense of touch to transmit many different data sources to the brain including audio, visual, and vestibular information. Various locations

on the body have been used, with varying success [5-18]. Sensory interfaces used have ranged in size and configuration, from a single vibration motor, to a 2D array of 400 distinctly vibrating tips.

1.2. Mapping Sound to Touch

The primary task of a sensory substitution system is to take some arbitrary information and map it to a discrete set of sensory receptors in the body, while ensuring the input information is supplied within the sampling constraints of nerve receptors for proper transmission to the brain. The input data must be processed in a way that captures the necessary features to maintain useful intelligibility, while reducing the data to the channel configuration and maximum throughput of the sensory stimulation device. If too much information is present at the input, it must be either compressed or filtered to reduce the excess or redundant data. This presents the primary difficulty for designing a software package for sound-to-touch sensory substitution systems: How should a system encode time-varying air pressure values for representation on an arbitrary number of receptors on the body to provide useful information to the brain?

This thesis aims to provide a software library and design framework to aid in further research investigating this question. Design considerations include determining the constraints imposed by the electronic processing hardware that is used. It must be defined and compared against the temporal acuity of human sensory systems to verify the efficacy of typical modern computational hardware. Can today's typical mobile (laptop and smartphone) processing hardware provide robust audio processing for sound-to-touch sensory substitution with latencies at or below the threshold (60 ms) needed for distinct perception of tactile stimuli?

1.3. Tactile Waves

Currently, a software library capable of performing audio signal management and processing for sound-to-touch sensory substitution systems does not exist. Previous sensory substitution research has

made use of custom built hardware and software to acquire audio and represent it tacitly on various locations on the body [5-18], and there are fundamental systemic components that these systems share.

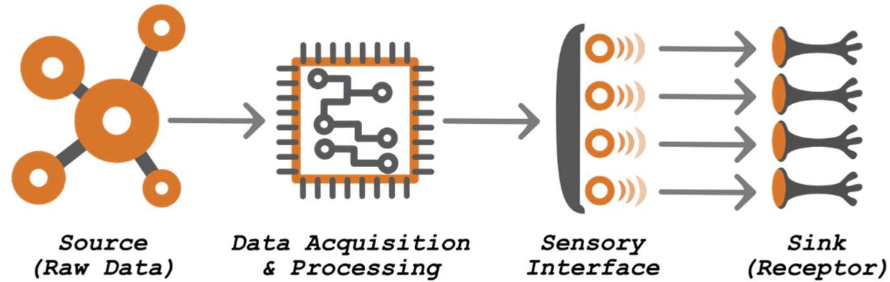


Figure 1.1: Visualization of the main components of a generalized sensory substitution system.

As shown in Figure 1.1, a sensory substitution system from source to sink has 4 main components:

1. Information/Data Source – The raw signal of interest. In the case of sound-to-touch this would be an audio source such as a person speaking or playback of an audio file.
2. Data Acquisition and Processing– The raw input signal must be captured and buffered before further processing. This involves discretizing and digitizing the source via sampling.
3. Sensory Interface – An array of N transducers on the body such that each channel can transmit time varying signals to separate receptors in the body, distinct from all other channels.
4. Information/Data Sink – Sensory receptors in the body that communicate with pathways that can deliver the electrical signals to the brain in response to stimulus from the sensory interface.

The primary contribution of this thesis is the development of a dedicated Java library containing both a toolbox and a prebuilt, general-use audio engine for audio acquisition, preprocessing, analysis, and feature extraction. This library allows researchers to use either a Java Virtual Machine (JVM) or Dalvik Virtual Machine (DVM) to serve as the data acquisition and processing component of a sound-to-touch sensory substitution system. Because the JVM is directly supported on Windows, Mac, and Linux, Java programs can be built for virtually any computer. The DVM is specific to Android devices, so the

same programs built for a computer can be built for nearly any Android device. Additionally, Java can run on iOS devices through Oracle's ADF Mobile, as well as on embedded devices with the use of one of Oracle's embedded runtime environments. This broad platform compatibility allows Tactile Waves to target as many potential users and applications as possible.

Many general-purpose DSP libraries are capable of performing a variety of useful audio analysis. What do these libraries lack for sensory substitution applications? Why would a researcher use Tactile Waves over another library, or combination of libraries? The primary problem (for sensory substitution applications) with these more traditional DSP libraries is their audio-in, audio-out design. They are designed to capture audio from a microphone or media file, perform some processing on the sound, and play back the resulting audio to a speaker or file. For sound-to-touch sensory substitution, audio playback is not needed. Instead, some signal or set of commands must be generated to activate channels on the sensory interface in response to the audio input. Additionally, all processing and analysis must be capable of low-latency real-time operation. Currently, the only Java library capable of performing real-time audio analysis and feature extraction is Tarsos DSP. This library was used in the preliminary stages of this research, but it was designed primarily for music information retrieval, synthesis, and DSP education. As a result, sensory substitution applications are outside of its intended scope. There is a need for a Java DSP library capable of real-time mapping of sound-to-touch, and Tactile Waves was developed to meet this need.

1.4. Summary

The primary contribution of this thesis has been to develop a software library for the Java Virtual Machine (JVM) that provides a toolbox of audio processing objects that can be used to perform preprocessing, feature extraction/analysis, and transmission of speech signals for sound-to-touch sensory substitution devices. The challenges and design considerations for real-time audio processing

are discussed, and a playback latency limit is established. Each subpackage within Tactile Waves is presented and the function and implementation of each object is detailed. Computationally expensive methods from objects within the *toolbox* package are tested on both Windows and Android to verify real-time operation with various audio buffer lengths. Finally, future improvements to the library are summarized and commercial applications are explored.

2. Real-Time Digital Signal Processing

The primary design consideration and bottleneck of real time audio processing is latency. An audio system whose sole task is to capture, digitize and playback a signal will have some amount of delay between the original and replayed signal. Additional analysis and processing of the signal by the system will increase this latency at a rate proportional to the computation complexity of the processing performed. For musical applications of DSP, latencies of 20-30 ms are considered acceptable, with 10 ms being ideal [19-20]. Higher latencies will introduce issues for listeners, as the delay between the source and processed signals becomes perceptible and destructive phase interference, known as comb filtering, can be introduced. Comb filtering is an artifact that is produced when two similar but time delayed (out of phase) signals are present at perceptible levels, as is the case when both source and processed signal are played together. However, due to spatial reverb effects that are always present in real listening environments, comb filtering is always present to some degree. This suggests that the brain is accustomed to adjusting for this degradation. Additionally, for applications involving hearing-to-touch substitution, the source and processed signals are not perceived simultaneously. Therefore, phase cancellation effects are disregarded and only the problems of perceived synchronization and continuous playback are addressed.

Most audio processing procedures require the signal being analyzed to be continuous and stationary. When an audio source is sampled by an analog-to-digital converter (ADC), it is discretized into a series of periodically spaced values representing the amplitude of the signal at each point in time. Audio signals are, therefore, a 1-dimensional, time varying sampling of relative air pressure amplitude. Real world audio that contains useful information such as language, music and environmental sounds is constantly varying and is therefore non-stationary. To perform useful processing on this information, a technique known as short-time analysis is used. Short time analysis takes a small chunk of these

consecutive audio samples acquired from the ADC, and stores them in a region of computer memory known as a buffer. This buffer contains a short segment of the audio signal, the length of which is typically chosen as a power of 2 and will typically vary from 256 to 8192 samples, depending on the type of analysis performed. This buffer is assumed to be stationary and continuous by treating it as being circular. That is, during analysis of the buffer, it is assumed to repeat end to end out to infinity. This allows a DSP system to process a continuous audio source piece by piece, yielding information about the audio signal at each "short-time" snapshot.

Because each buffer contains only a short glimpse of a complex, time varying signal, discontinues are introduced at the beginning and end of the buffer where the signal is truncated. These discontinues lead to errors and inaccurate results during analysis. Applying a window function (also called a smoothing window) to each buffer prior to analysis is therefore a standard preprocessing step. Windowing functions aim to remove or soften the discontinuities at the ends of the buffer by tapering, or smoothing, the signal down to near zero at each end. The simplest windowing function available is the Rectangular Window, shown for a buffer size of 256 below in Figure 2.1:

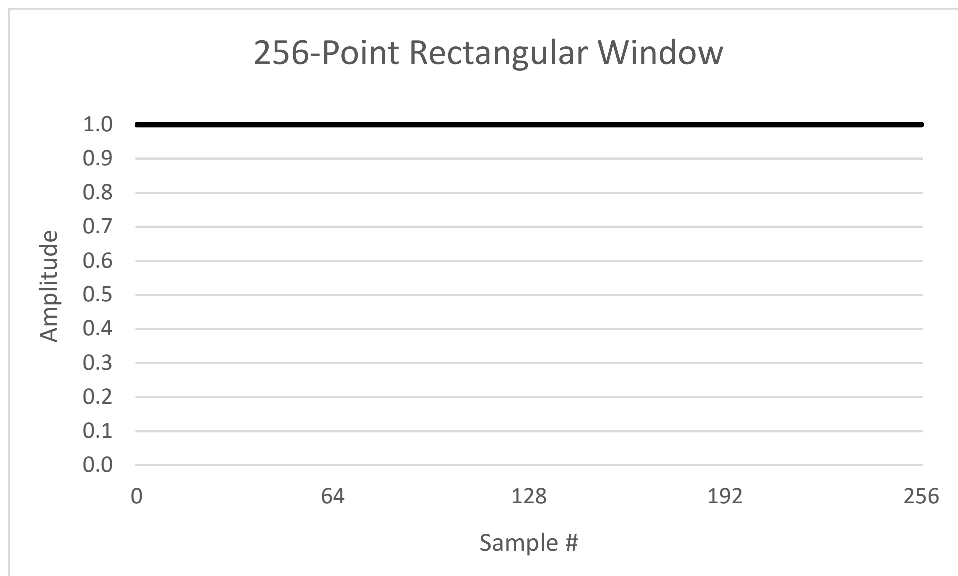


Figure 2.1: A 256-point rectangular windowing function.

The rectangular window contains only unity amplitude, so applying a rectangular window is precisely the same as performing no windowing at all. Therefore, using no window function is described as using a rectangular window. More advanced windowing functions aim to produce unity amplitude (or near unity) at the center of the buffer, and taper down to zero (or near zero) at either end. Various window functions are available, one of which known as the Hann Window is shown for a 256-sample buffer in Figure 2.2.

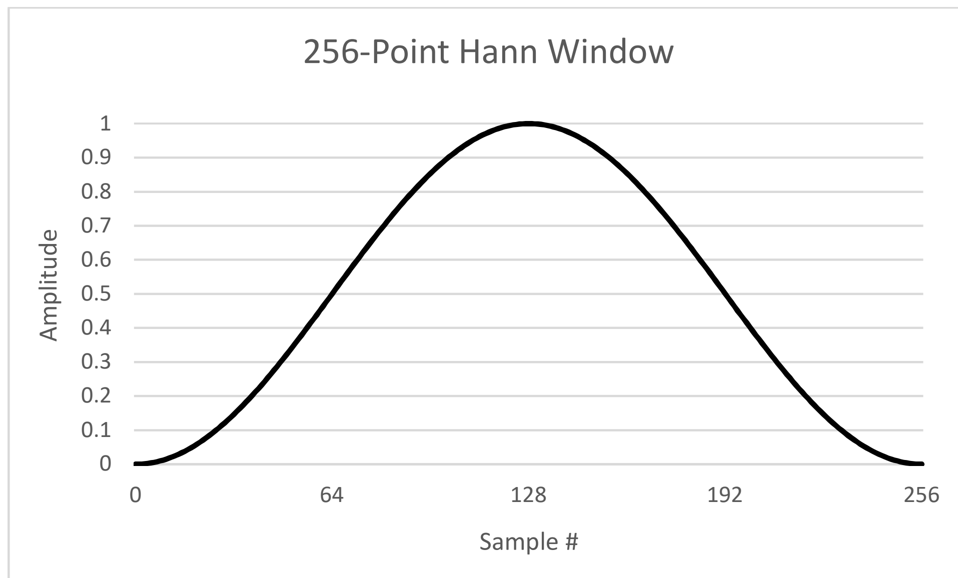


Figure 2.2: A 256-point Hann Windowing Function

This window applies unity amplitude at the center of the buffer and gradually tapers down to an amplitude of zero at either end. Specific windowing functions are discussed in more detail in the Development chapter of this document.

Consider the 1024-sample audio signal containing pseudo-random noise shown in Figure 2.3, generated in Microsoft Excel with a random number generator bounded between -1 and 1.

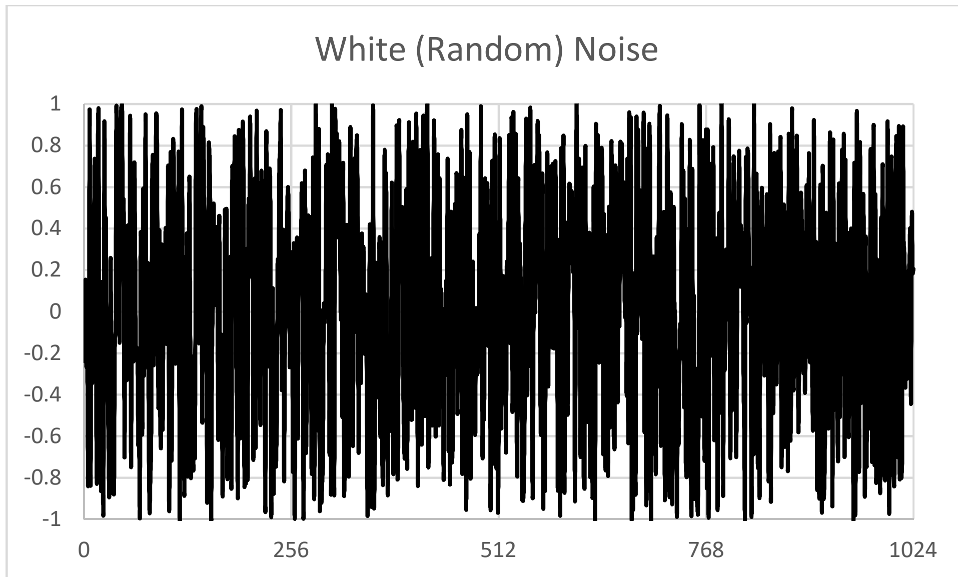


Figure 2.3: 1024-sample random noise signal

To perform short-time analysis on this signal, it is first broken into consecutive, non-overlapping buffers of 256 samples and each buffer is multiplied by a 256-point Hann window function. The windowed signal is shown below in Figure 2.4.

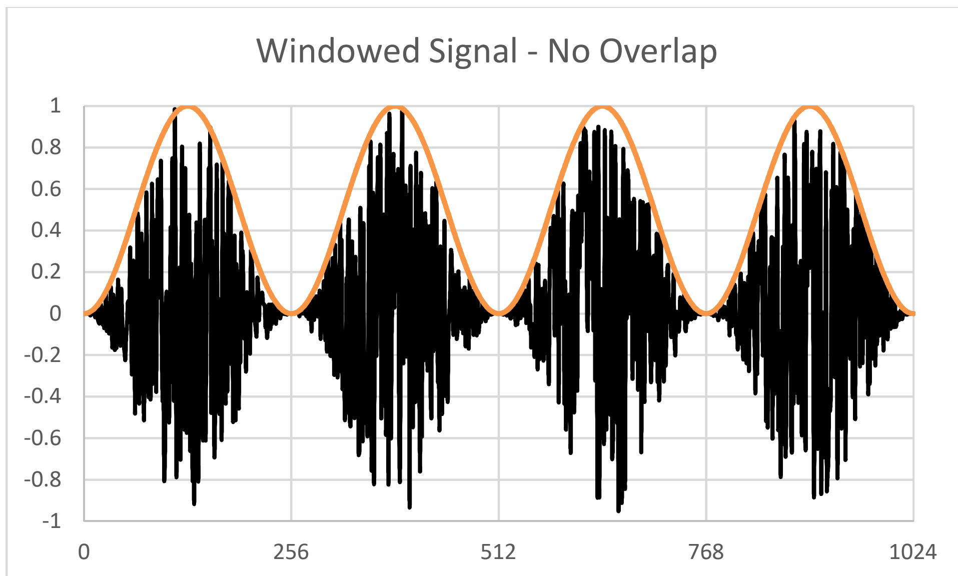


Figure 2.4: 1024-sample white noise signal windowed with 256-point Hann with no overlap

Unfortunately, applying the window function in this manner has degraded the signal due to the amplitude modulation performed by the window. Information is lost at the boundaries of each buffer, and the total signal energy has been cut in half, as shown in Figure 2.4.

To overcome these issues, overlapping buffers can be used. By overlapping buffers, samples at the end of one buffer will be reused in the beginning of the next buffer. Therefore, information near the ends of each windowed buffer is not lost. The number of overlapping samples between one buffer and the next determines the amount of overlap, which is chosen based on the total buffer length, the window function being used, and the analysis performed on the windowed samples. For example, to account for the Hann window's halving of the signal energy, a 50% buffer overlap is used. By applying a 256-point Hann window to the white noise signal in Figure 3.3 with an overlap of 128-samples (50%), the energy of the signal is preserved. By overlap-adding each buffer, the original signal can be perfectly reconstructed, as shown in Figure 2.5.

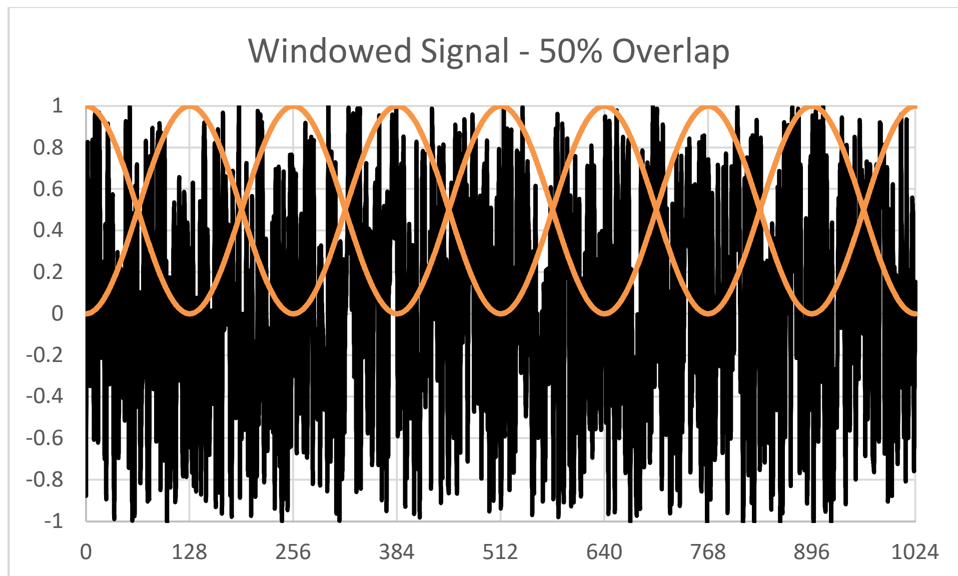


Figure 2.5: 1024-sample white noise signal windowed with 256-point Hann with 50% overlap

The length in seconds of each of these audio buffers, t, can be calculated with the following equation:

$$t = \frac{N - overlap}{f_s}$$

Where N is the length of the buffer, in samples, $overlap$ is the number of overlapping samples, and f_s is the sampling frequency. Typical sampling frequencies for speech processing range from 8000 to 24000 Hz, with music or other more complex audio signals utilizing higher sample rates ranging from 44100 to 192000 Hz. For example, an audio buffer of 1024 samples acquired at 44.1 kHz represents a 23.2 ms duration of audio. This value is significant because it represents the maximum latency limit of the processing system. For continuous playback of the processed audio there must be an audio buffer available at every 23.2 ms interval, otherwise there will be skips and drop outs in the output stream. If overlapping buffers are used, this value is reduced by the amount of overlap. The system must, therefore, be able to process each buffer in less than the time it takes for the buffer to be played as audio. Because each 1024 sample buffer will take 23.2 ms to play aloud as audio, the system must be able to perform its processing on each buffer in 23.2 ms or (ideally) less. Scaling the size of the buffer will scale this maximum latency limit by the same factor. Because of this practical latency limit, all experiments performed herein to investigate the real-time performance of the software modules are analyzed based on the subroutines ability to process a buffer in less than the time-domain length of the buffer.

It is important to consider the effect of buffer length on computation time. Choosing a larger buffer size allows for longer processing times, so one might be tempted to choose the largest buffer possible while still maintaining perceived synchronization. However, the amount of processing needed also scales proportionally with the size of the buffer. By doubling the buffer from 1024 to 2048 samples, the number of samples that require processing at each step has also been doubled. Depending on the computational complexity, this may have a substantial impact on the required processing time. For example, the common radix-2 FFT algorithm, made popular by J.W. Cooley and John Tukey in 1965, is

bound by $O(n \log n)$ complexity, while its less efficient counterpart, the DFT, is bound by a complexity of $O(n^2)$. Figure 2.6 shows the required computations for each complexity for increasing buffer sizes, as well as a basic O(n) complexity for comparison.

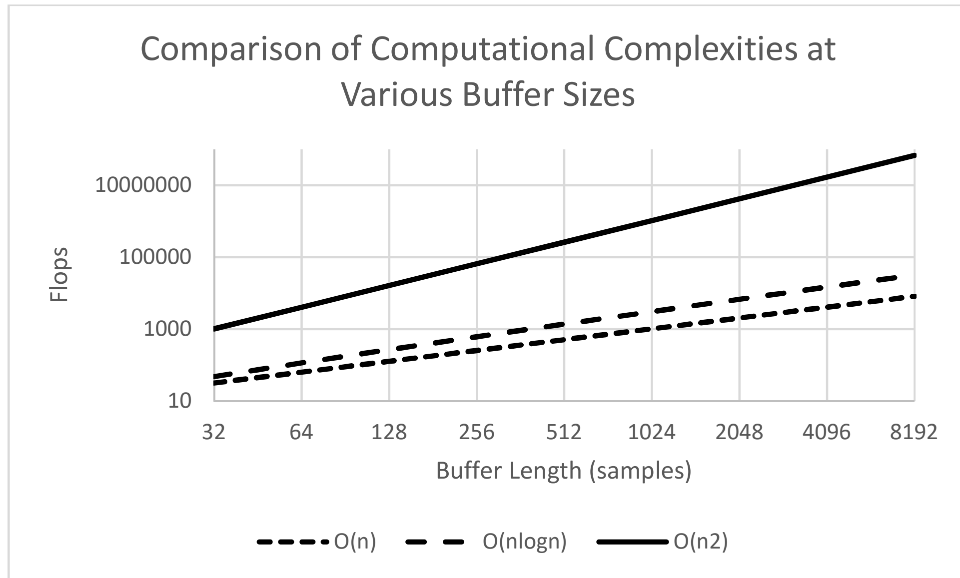


Figure 2.6: Required processing steps as a function of buffer size for different computational complexities

For algorithms that are bounded by any complexity greater than $O(n)$, an increase of the buffer size results in an even larger increase of required processing steps. These results indicate that buffer size selection should consider the computational complexity of the required processing. Additionally, not all DSP algorithms are optimized at the same sample rate and buffer size. A typical algorithm for estimating the fundamental frequency (pitch) of a sound, for example, requires a buffer length that is at least a long as one period of the fundamental [21]. Each DSP subroutine must be evaluated individually to ensure real-time usage criteria are met over a range of acceptable buffer sizes.

Perceived synchronization refers to the tendency of an audio-visual observer to not detect desynchronization of audio and visual cues when the time delay between each is below a certain value. The maximum value of this imperceptible delay varies between persons, but it has been found that +/- 80 ms is an acceptable value [20]. If the time delay between an audio signal, and its corresponding visual

cue is less than +/-80 ms, the delay will most likely not be perceived. Certain factors can drastically change this value. For example, increasing distance between listener and the visual content can increase the acceptable delay [20]. As a result of this effect, all audio processing performed on a single audio buffer must be performed in less than 80 ms. A conservative limit of 60 ms is chosen to ensure a user will not be able to perceive the delay between visual cues. From this value, a practical buffer size limit can be selected based on the chosen/available sample rate. For example, at 44.1 kHz a buffer size of 2048 results in a playback latency of 46.4 ms, which is below the 60ms perception limit.

An audio processing system must adhere to these considerations to be capable of real-time operation. Tactile Waves provides a programming interface that allows users to build such a system, customized for their application.

3. Development

In this chapter, the Tactile Waves sensory substitution library is discussed in detail. The programming language, development environment, and publishing platforms that were used to create and release the library are presented. Implementation requirements are summarized before providing a brief technical primer in Object Oriented Programming. Finally, the API is described in detail via a walkthrough of each subpackage in the library. Each object within each subpackage is exposed, and their functionality and implementation details are provided.

3.1. Programming Environment

The Tactile Waves API is written in the Java programming language. Java was chosen for its multi-platform interoperability. Programs built and compiled on one machine can run on any other machine that supports the Java Virtual Machine (JVM). Currently, the JVM is directly supported on Windows, Mac OS, and Linux. Android's Dalvik Virtual Machine is used to run Java code on Android devices such as smartphones and watches. Any programming logic written for the JVM is compatible with the DVM, and vice versa, allowing Tactile Waves to be used for both PC and mobile applications.

The development of Tactile Waves was performed in an Integrated Development Environment (IDE) to simplify project management, testing, and documentation. Because the Android platform is the primary deployment target for the API, the Android Studio 3.0 IDE was used. Java 8 is the latest version of Java provided by Oracle. However, Version 3 of Android studio uses Java 7, with a subset of features from Java 8, so the library was designed within these constraints to ensure Android compatibility. While Tactile Waves can be used in iPhone applications through Oracle's ADF Mobile, this platform has not been tested due to the prohibitive cost and restrictive ecosystem of Apple devices.

Git was used for version control, and the source code for the library is stored in a public repository on GitHub [22]. GitHub is also used to host a static web page with the library containing usage/install instructions, documentation, download links, etc. Documentation is generated with JavaDoc and is bundled with the library download, as well as included in the web page. Downloads are provided through Bintray and remote linking is available from the popular jCenter Android library repository.

3.2. Implementation

The primary function that Tactile Waves aims to accomplish is the management of an audio stream from source to sensory interface in a sensory substitution system (Figure 1.1). This requires sampling of audio from a microphone or audio file, performing some useful processing on the audio, transforming the audio data into the correct dimensionality and size, and transmitting the transformed data to a sensory interface worn on the body. In the case of live speech, these operations must be performed continuously with acceptable latencies for real-time operation. The processing stream must also be designed in a way that is flexible and extensible. The interface should allow a user to customize the processing chain without writing custom code, but also allow a user to extend the functionality with their own custom code if desired. Finally, verbose documentation must be provided for all public classes and methods to allow users to see the operation of each function without having to view the API source code.

3.3. Object Oriented Programming – A Technical Primer

The following sections describe the code design of the libraries main packages in detail, and therefore requires the use of terms that may be new to those unfamiliar with object oriented programming (OOP). Readers who are themselves programmers, or have experience with OOP, may wish to skip over this section.

In the following sections, whenever a piece of code or code object is discussed in text, it will be displayed in a `monospace` font to distinguish it as such.

Object oriented programming, or OOP, is a term used to describe a common programming design paradigm that is based around the creation and interaction of *objects.* An *object* is a collection of related state and behavior. State information is stored in data members known as *attributes* while behaviors are chunks of encapsulated code known as *methods*. A programmer can invoke the methods of an object to illicit the object's behavior, which may include changing the state of the object. Because each object has its own state and behavior, multiple instances of the same object can be created and used simultaneously. Objects can interact with or depend on one another, changing the state, or invoking behaviors to create complex interactions. This is the main idea behind OOP.

A *class* is a blueprint, or a template that describes the default state and behavior that is used to create an object. The relationship between classes and objects is best described with an example. Suppose a program needed to store information about several different dogs. A `Dog` class could be created that describes the state and behavior of any `Dog` objects. An example of a possible implementation of the `Dog` class is shown below (in Java-esque pseudocode):

```
class Dog {
      attributes: breed, age, name
      behaviors: get recommended diet
}
```

The program can create any number of objects that describe different dogs using this `Dog` class as a prototype. The above code *defines* the class `Dog`, but to create an actual object of type `Dog`, an instance of `Dog` must be *instantiated*. Each Dog instance will have its own breed, age, and name that is separate from any other instances of the `Dog` class. Rover, a 4-year-old golden retriever, and Spot, an 8-year-old border collie, will be represented by 2 separate objects that are both of the class `Dog`. An instance of a

class might change its behaviors based on the state of the object defined by its attributes. For example, the behavior “get recommended diet” could be designed to return a recommended diet for a dog based on the breed and age of the dog stored in the instance of each `Dog` object. A benefit of this design is that the program can reuse the code contained in the `Dog` class for all instances, rather than rewriting nearly identical code for each dog that the program needs to store.

Now suppose the program required the storage of a new type of a dog: a service dog. To properly describe a service dog, additional state information and behaviors are required beyond what the basic `Dog` class provides. A subclass called `ServiceDog` could be created that extends from the original `Dog` class:

```
class ServiceDog extends Dog {
      attributes: training
      behaviors: get service type
}
```

Any objects that are creating using the `ServiceDog` class will contain all the attributes and behaviors of the original `Dog` class, as well as the new attributes and behaviors defined in the `ServiceDog` class. This idea is called *inheritance* in OOP, as the subclass `ServiceDog` *inherits* the attributes and behaviors of the `Dog` superclass. Again, this design allows for code reuse, which generally leads to smaller programs and easier code maintenance.

In the above example `ServiceDog` inherits from a concrete superclass called `Dog`, and either can be used to create objects. But subclasses can also inherit from an *abstract* superclass, or a class that cannot be instantiated into a concrete object. Consider the abstract class below called `Pet`:

```
abstract class Pet {
      attributes: age, name, owner
      abstract behaviors: get recommended diet
}
```

This class is declared *abstract*, meaning that it cannot be directly instantiated and must instead be subclassed into concrete classes that extend from the abstract class `Pet`. Abstract classes can also contain abstract methods (behaviors). An abstract method is not implemented in the abstract class, but is instead implemented in the concrete subclasses of the abstract class. A new `Dog` class that is a subclass of `Pet` is shown below:

```
class Dog extends Pet {
      attributes: breed
      behaviors: get recommended diet
}
```

Just as with the ServiceDog example, the Dog class contains all of the attributes and behaviors defined by its superclass, Pet, as well as the new attributes and behaviors defined in the subclass. A brand-new object type can be created that is different than Dog, but still extends the abstract Pet class:

```
class Cat extends Pet {
      attributes: breed
      behaviors: get recommended diet
}
```

Now the program can describe both cats and dogs, which are completely different objects, but both are subtypes of the abstract idea of a pet, and can therefore be used interchangeably as varying types of pets. It can be said that in the real world, a dog is a type of pet, and a cat is a type of pet, but a pet is not a type of cat or dog. In fact, a pet is not a type of any animal. It is an abstract concept that describes any animal that a person keeps and cares for. The same statements can be said about the abstract `Pet`, and concrete `Dog` and `Cat` classes. In this way, objects can be created in ways that mimic humans real-world understanding of objects and types.

An *Interface* is a construct in OOP that defines a contract between classes and the programmer. Any class that implements an interface is guaranteed to implement the behavior described by that

interface, i.e. it must adhere to the contract laid out by the interface. Because interfaces only describe the required behavior of implementing classes, they cannot have attributes. An interface called Drawable is shown below:

```
interface Drawable {
      behaviors: draw
}
```

Any objects that implement the `Drawable` interface, are "contractually obligated" to implement a behavior called `draw`. For example, the `Dog` class could implement the `Drawable` interface, and define a `draw` method that draws a dog to the screen. Meanwhile the `Cat` class could also implement the `Drawable` interface, and define a `draw` method that draws a cat to the screen. Each class contains its own implementation of the `Drawable` interface, and although the implementation differs, the behavior is the same: both objects are capable of drawing a visual representation of the object they represent to the screen. By using interfaces to define certain behaviors, objects of different types can be replaced in a program without altering the correctness of that program.

And finally, a *package* is a collection of classes and interfaces, and a *subpackage* is simply a package within a package. Packages are used to logically organize code into groups of related code files, and can be thought of as being analogous to a folder system on a computer. A folder contains a group of files, and can contain other folders that also contain files and folders and so on. Much like folders on a computer help users organize their files, packages help programmers organize their code.

3.4. Tactile Waves Package Overview

An overview of the *tactilewaves* package is shown in Figure 3.1. The *com* subpackage contains classes for sending data over Bluetooth, and the *android* subpackage within contains Bluetooth code that is specific to the Android platform. The *dsp* subpackage contains all the classes that make up the core audio processing engine, as well as 2 additional subpackages, *toolbox* and *utilities*. The *toolbox*

subpackage contains a collection of useful algorithms such as filtering and Fourier transforms, while the *utilities* subpackage contains a collection of classes to perform utility operations such as data sorting. Finally, the *io* subpackage contains all the necessary code for the input and output of audio data from an audio file or a microphone, and the *android* subpackage within contains audio input/output code that is specific to the Android platform.

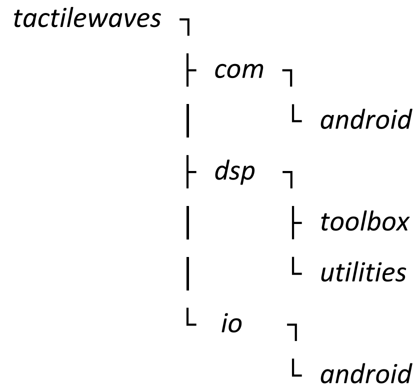


Figure 3.1: Tactile Waves Package structure overview

3.5. The Audio Engine

Tactile Waves uses a custom audio engine that was built specifically for sensory substitution applications. A set of objects are used to manage an audio stream from acquisition, through a chain of processing, and finally transform the audio signal into signals or commands that can then be sent to stimulation hardware worn on the body. The objects that make up the audio engine are contained within the *dsp* and *io* subpackages and are summarized in Tables 1 and 2.

Table 1: Tactile Waves Audio Engine Class Summary

Class Name	Description
`WaveFormat`	Describes the format of audio being used
`WaveInputStream`	Manages reading from an input stream
`WaveFloatConverter`	Converts byte encoded audio to floats (-1,1)
`WaveFrame`	Stores 1 frame of audio and supporting data/objects

WaveManager	Core audio processing object – manages thread to prepare and process frames of audio

Table 2: Tactile Waves Audio Engine Interface Summary

Interface Name	Description
WaveFrameListener	Defines listener functionality to create objects that listen for processed WaveFrame events
WaveProcessor	Defines processor functionality for building objects that perform some processing on a WaveFrame

WaveFormat

The WaveFormat class allows the creation of WaveFormat objects that house the parameters that describe the format of the audio in use. The format parameter member variables are summarized in Table 3.

Table 3: WaveFormat Member Summary

Member Name	Type	Meaning
mEncoding	Integer Constant	Represents the encoding method used using integer constants
mBigEndian	Boolean	True for Big Endian byte order, False for Little Endian
mSampleRate	Integer	The sample rate (Hz) of the audio
mBitDepth	Integer	The number of quantization bits used (bits/sample)
mChannels	Integer	The number of audio channels
mBytesPerSample	Integer	The number of bytes needed for 1 sample of audio

The mEncoding parameter is an integer constant that represents the type of encoding used to byte-encode the underlying audio associated with this format. Three static integer constants are defined in the WaveFormat class that define the state of the mEncoding variable. If the encoding has not been specified, mEncoding will equal NOT_SPECIFIED. This is the default value of the mEncoding

parameter. Currently, only PCM (linear quantization) encoding is supported, in both signed and unsigned forms. Depending on the encoding method used, this parameter will take the state of either ENCODING_PCM_SIGNED, or ENCODING_PCM_UNSIGNED. The `mBigEndian` parameter is a Boolean (true/false) data type that specifies the byte order (endianness) used in the underlying audio stream. If true, the bytes in the stream are ordered in Big Endian byte order – meaning the most significant byte will be read first, followed by the second most significant byte. If false, Little Endian byte order is used, and the bytes will be ordered with the least significant bytes first – the opposite of Big Endian. The `mSampleRate` member is an integer variable that holds the sample rate, in Hz, that was used to record the audio in question. On Android devices, this is almost always 44100 Hz (the default sampling rate on Android), but any sample rate is supported by `WaveFormat`. The `mBitDepth` describes the number of bits used to quantize the audio samples in the given encoding scheme. As little as 8 bits can be used for speech without overly degrading the sound quality, but 16 bits or more are needed for audio files containing music or other more complex audio content. When recording from the Android microphone, this bit depth will almost always be 16 bits (the default for Android). Additionally, most audio files are stored at 16 bits, due to its balance between quality and file size. Regardless, any bit depth that is a multiple of 8 bits is supported (8, 16, 24, 32, etc). The `channels` parameter stores the channel count of the audio described by this `WaveFormat` object. A mono file will have 1 channel, while a stereo file has 2 (left and right) channels. More than 2 channels indicate some type of multichannel surround sound type audio, and is not currently supported by *Tactile Waves*. Finally, `mBytesPerSample` is an additional parameter that is not needed to correctly describe the audio format, but it has been included because of its usefulness in other classes. It stores the number of bytes needed per sample of audio and can be calculated from other `WaveFormat` parameters:

$$BytesPerSample = Channels * \frac{BitDepth}{8}$$

Because this can be calculated using 2 other preexisting format parameters, it is not necessary. However, it is useful when decoding audio as it describes how many bytes must be read at once to read one sample of audio from an encoded stream so its inclusion saves users from having to repeatedly implement the calculation.

WaveInputStream

The `WaveInputStream` class is an abstract class that was created to allow the creation of any number of subclasses that house different types of input streams. By wrapping input streams into WaveInputStream subclasses, implementation differences between the streams can be abstracted away. This allows Tactile Waves to pipe audio from any audio source without requiring specific code for each source. Code can be reused between all subclasses of an abstract class, so this design also reduces code complexity and eases maintenance.

Any object that is a subclass of `WaveInputStream` must implement the following methods: `read()`, `readSample()`, `getFormat()`, and `close()`. The `read()` method attempts to read a single byte from the input stream and return it. The `readSample()` method attempts to read a single *sample* from the input stream. The number of bytes required to construct one sample is defined by the `WaveFormat` associated with the input stream. Using the `read()` and `readSample()` methods from each subclass, all `WaveInputStream` objects can utilize additional `read()` methods to read multiple samples at once, without needing to implement code to do so. The methods `getFormat()`, and `close()` are necessary utility methods that allow the user to obtain the `WaveFormat` associated with the input stream and close the input stream, respectively.

WaveFloatConverter

The `WaveFloatConverter` class is a static class whose sole responsibility is the conversion of multi-channel encoded audio bytes to single channel audio samples (floats), and back again. Its primary methods are `toMonoFloat()` and `toMonoFloatArray()` which convert an array of bytes to a single sample or array of samples, using single precision floating point numbers. The methods `toMonoDouble()` and `toMonoDoubleArray()` perform the same task, but instead return audio samples as double precision (64-bit) floating point numbers. Finally, the `toBytes()` method performs the reverse operation, converting single or double precision audio samples back to encoded bytes, according to the supplied `WaveFormat`.

WaveFrame

The `WaveFrame` class represents a single frame of audio. The member objects of this class are listed below in Table 4.

Table 4: WaveFrame Member Summary

Member Name	Type	Meaning
`mFormat`	`WaveFormat`	The audio format of this frame
`mSamples`	float array	The sample buffer, or the audio frame itself
`mLength`	integer	The length of the frame (sample buffer)
`mFeatures`	HashMap	Collection of audio features in the frame such as the pitch, or a list of formant frequencies

Each instance of the `WaveFrame` class contains the actual audio samples and `WaveFormat` associated with one frame of audio that has been read from a `WaveInputStream`. The frames audio data is stored in the `mSamples` object, with `mLength` storing the length of the frame. The `WaveFormat` describing the underlying audio format is stored in the `mFormat` variable. Apart from

energy/volume calculations on the frame, this class is not responsible for performing any actual signal processing on the audio buffer. The sole purpose of this object is the storage of a frame of sampled audio data. These objects are created and passed through a processing chain by the `WaveManager` object, and sent to any `WaveFrameListener` objects upon completion. During processing, extracted audio features can be stored in the objects `mFeatures` object, which is described in more detail below.

WaveManager

The `WaveManager` class contains the core audio processing thread and is responsible for the acquisition and management of sampled audio data. Its member objects are summarized below in Table 5.

Table 5: WaveManager Member Summary

Member Name	Type	Meaning
`mRunning`	`boolean`	Is the audio thread currently running?
`mInput`	`WaveInputStream`	The input stream to read audio from
`mFormat`	`WaveFormat`	The audio format in use
`mListeners`	List	List of WaveFrameListeners to notify of completed WaveFrame objects
`mSamples`	Float array	A buffer to store audio samples
`mOverlap`	integer	# of samples to overlap between frames
`mLength`	integer	The number of samples used in each audio frame
`mFramesProcessed`	integer	The number of audio frames that have been successfully processed
`mTotalSamplesRead`	integer	The total number of samples that have been read from the `WaveInputStream`
`mFXChain`	LinkedList	A list of `WaveProcessor` objects that are executed sequentially on each `WaveFrame` produced

An instance of `WaveManager` will attempt the read audio encoded as bytes from `mInput`. These bytes are immediately converted to audio samples according to the format parameters specified by `mFormat`. For each frame, `mOverlap` samples from the previous frame are combined with

```
mLength – mOverlap
```

new samples from the input stream. These samples are then copied into a new `WaveFrame` object, and this object is then passed to the `process()` method of each object in `mFXChain` for processing. After executing the entire processing chain, the processed `WaveFrame` is sent to any listeners stored in the `mListeners` member, and the `WaveManager` repeats these steps for the next frame, until it is stopped.

This object is the primary component of Tactile Waves' audio engine, and is the object that users will primarily interact with when using the library. An instance of `WaveManager` is created by supplying its constructor with 3 parameters: a `WaveInputStream`, and 2 integers that specify the number of samples used for the frame length and overlap. At this point, the `WaveManager` will be ready to begin reading audio samples from the input stream and assembling them into `WaveFrame` objects for further processing. However, when the object is created, it is initialized with an empty processing chain. In order to perform some useful processing on each `WaveFrame`, one or more `WaveProcessor` objects must be added to the processing chain using the `WaveManager`'s `addEffectToChain()` method. It is important to note that the processors within mFXChain will be executed sequentially, in the order they were added. Immediately after creation, each WaveFrame is sent to the first processor in the chain, and once it has completed its processing, the WaveFrame is sent to the second processor in the chain, and so on until all processors in the chain have been executed.

WaveProcessor

The `WaveProcessor` interface defines the required functionality that an object must implement to be used in a `WaveManager`'s processing chain. The interface specifies two methods: `process()` and `processingFinished()`. The `process()` method should contain all of the code required to perform whatever processing is required for each `WaveFrame`. The processor can make use of the `WaveFrame` method `getSamples()` to obtain a reference to the frame's samples and perform some useful processing with or on the buffer. Optionally, a `WaveProcessor` can add any number of arbitrarily formatted "features" to the `WaveFrame` using its `addFeature()` method. Each processors `process()` method is called once by the `WaveManager` for every frame of audio. The `processingFinished()` method is called after all processing has been completed on a `WaveFrame` and should therefore contain any clean-up code that requires execution after the processor has finished. This might include deallocation of resources, and resetting of variables or parameters. If no clean-up code is required by the processor, the `processingFinished()` method can be left blank.

WaveFrameListener

The `WaveFrameListener` interface specifies the required functionality of listener objects with just one method: `newFrameAvailable()`. After successfully executing the processing chain, a `WaveManager` will call any active listener's `newFrameAvailable()` method, passing it the newly processed `WaveFrame` object. The listener can then trigger some response to this event such as updating the UI, or sending `WaveFrame` features over Bluetooth. This allows heavy, time consuming tasks like these to be performed in a dedicated thread, separate from the `WaveManager`'s audio processing thread.

In general, the `WaveProcessor` interface should be used for any tasks that should be run in the audio processing thread, while the `WaveFrameListener` interface should be used for any tasks that require access to the processed `WaveFrame` objects, but should not be performed within the audio thread.

3.6. The Toolbox

Tactile Waves features a list of toolbox objects that are used to perform single DSP tasks such as computing the Fourier Transform of an audio buffer. These methods are listed in Table 6 and described in more detail in this section.

Table 6: Tactile Waves Toolbox Class Summary

Object Name	Responsibility	Requires Instantiation?
`FFT`	Fourier Transforms of audio buffers	Optional instantiation for speed, otherwise no
`Window`	Windows audio buffers	Instantiation available, but not required
`DCT`	Discrete Cosine Transforms, Type-I and II	No
`Filter`	Filters an audio buffer	Custom filters require instantiation, otherwise no
`Cepstrum`	Cepstral Transforms on audio buffers	No
`MFCC`	Computes Mel Frequency Cepstrum Coefficients	No
`LPC`	Linear Predictive Coding Analysis	No
`YIN`	Implements YIN Pitch Detection Algorithm	Yes, required for algorithm correctness
`ZCR`	Computes the zero-crossing rate of an audio signal	No

These classes and their methods are designed to provide as much static access as possible. That is, toolbox methods should be available to the user without requiring creation of an instance of the containing object, wherever possible. This design favors performance and reduced memory usage.

FFT

The `FFT` class is an all in one tool for performing Discrete Fourier Transforms on arrays. Converting a signal to the frequency domain is a useful processing step in its own right, but many other tools in the package require the use of FFT's in their algorithms. Therefore, performance optimizations made within the `FFT` class will necessarily translate to optimizations in other algorithms that rely on the FFT (such as the DCT). Iterative, rather than recursive, implementations are used because they are significantly faster. Additionally, because Tactile Waves is an audio processing library, and audio is always a real signal, the real DFT is used over the complex DFT wherever possible, and has been optimized to yield a 20-30% speed increase over the complex DFT.

`FFT` supports any transform size. Power-of-2 buffer lengths are supported through the famous Cooley-Tukey Radix-2 Fast Fourier Transform [23], and are the most performant ($O(n \log n)$). All other buffer sizes are supported using Leo Bluestein's algorithm [24]. Several data types are supported. Arrays of both single and double precision floating point numbers as well as arrays of Complex objects (see Utilities section) can be used with `FFT`. The algorithm is implemented with an iterative decimation-in-time (DIT) approach, summarized and explained in detail below:

Complex FFT Algorithm Summary

1. Decompose N sample signal into N 1-sample signals
2. Transform each of the 1-sample signals into the frequency domain
3. Recombine N 1-point spectra into a single N-point spectrum

The first step in the DIT complex FFT algorithm is to decompose 1 signal of N samples into N signals of 1 sample in the time domain (hence decimation-in-time). An example of this decomposition is shown for an 8-sample signal in Table 7. In the first step, the 8-sample signal is decomposed into 2

signals of 4 samples. In the next step, both 4-sample signals are decomposed into a total of 4 signals of 2 samples, and so on until the signal is broken down into N signals of 1 sample.

Table 7: Example of interlace decomposition with an 8-sample signal

1 signal of 8 samples	0 1 2 3 4 5 6 7 ↙ ↘							
2 signals of 4 samples	0 2 4 6 ↙ ↘				1 3 5 7 ↙ ↘			
4 signals of 2 samples	0 4 ↓ ↓		2 6 ↓ ↓		1 5 ↓ ↓		3 7 ↓ ↓	
8 signals of 1 sample	0	4	2	6	1	5	3	7

A technique known as interlace decomposition is used at each decomposition step, which separates the signal into even and odd components. To accomplish this even/odd reordering efficiently with code, a technique known as bit reversal sorting is used. This operation simply takes the binary address of each sample and reverses it to yield the samples new address. For the 8-sample signal, the sample at address 0 (binary 000) is not moved because the reverse of 000 is 000. The sample at address 1 (binary 001) is moved to address 7 because the reversal of 001 is 100, or a decimal 7. The 8-sample signal decomposition example is shown again using bit reversal sorting in Table 8.

Table 8: Example of bit reversal sorting with an 8-sample signal

Sample indices before bit reversal		Sample Indices after bit reversal	
Decimal	Binary	Decimal	Binary
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

Note how the ordering of sample indices after bit reversal sorting is identical to the ordering seen in the final step of the interlace decomposition in Table 7.

Once the time domain signal has been decomposed into N 1-sample signals, the algorithm must convert the signals into the frequency domain by calculating the spectrum of each signal. Here lies the beauty of the algorithm: the spectrum of a 1-sample signal is itself, so the algorithm needs to do *nothing* to convert each signal to the frequency domain.

All that is left for the algorithm is to recombine the N 1-sample frequency spectra into 1 N-sample frequency spectrum. The recombination is done in the exact opposite order that the interlace decomposition was performed. An example of a single recombination step is shown in Table 9. At each step, the signals are combined by duplicating each frequency spectrum, and adding the results together. This is known as a “butterfly calculation”, due to the shape outlined by the diagram in Table 9 that resembles a butterfly’s wings. For the spectra to match up when added, the odd point spectrum is shifted by 1 sample. Shifting in the frequency domain is performed by multiplication with a sinusoid. After recombination, the algorithm yields a single N-point frequency spectrum of the original N-sample signal and the algorithm terminates.

Table 9: Example of a recombination of 2, 1-sample spectra into a single 2-sample spectrum

2 1-sample Spectra Input	Even Spectrum ↓	Odd Spectrum ↓
Shift Odd Spectrum	↓ ↓	*Sinusoid ↓
Butterfly Calculation	duplicate ↙ ↘ E E ↓ ↘ ↓ ↘ ↓ ↙	duplicate ↙ ↘ O*S O*S ↙ *-1 ↙ ↓ ↘ ↓
1 2-sample Spectrum Output	E + O*S Positive Frequencies	E – O*S Negative Frequencies

Because this is a *complex* FFT, it operates on complex signals. A complex signal is a signal that is comprised of complex numbers, or numbers that have both a real and imaginary part, as shown by the equation:

$$z = x + yi$$

Where z is a complex number comprised of real part x and imaginary part y, and i is the imaginary number $\sqrt{-1}$. The complex FFT of a complex signal results in a complex spectrum, or a spectrum comprised of both real and imaginary parts. But what if the input signal is not complex? The complex FFT will work fine in this situation, as a real signal can be represented as a complex signal with all imaginary components equal to zero. However, many computation steps are "wasted" in this case on the calculations involving the all zero valued imaginary components. Audio signals are *always* real signals, never complex, so it would be preferable to not have to waste time on the complex signal calculations. Luckily, there exists an alternate algorithm that takes advantage of this situation by eliminating the unnecessary calculations associated with the real valued signal: the real FFT.

Notice how the very first step of the complex FFT works: An N sample signal is split in half and broken down into its even and odd decompositions, and the step is repeated on the even/odd halves. In other words, in the first step an N sample complex signal is divided into 2 N/2 sample complex signals. The real FFT uses a clever trick that takes advantage of this step and "fools" the complex FFT into doing the majority of its work for it. The real FFT takes an N sample real signal and breaks it into its even/odd halves, just as with the complex FFT. The 2 halves of the real signal are then reinterpreted as the real and imaginary components of a complex signal, yielding a single N/2 sample complex signal. This complex signal is then passed to the complex FFT which returns a N/2 complex spectrum. The real FFT can again reinterpret the real and imaginary components of this spectrum as 2 halves of a real

spectrum. The real FFT can now perform the final recombination step, combining the 2 N/2 sample spectra into a single N sample spectrum. A summary of this algorithm is described below:

Real FFT Algorithm Summary

1. Separate N point real signal into N/2 point even and odd halves
2. Compute the complex FFT of the N/2 point even/odd halves
3. Recombine N/2 length spectra into N length spectrum

Therefore, real FFT of an N point signal is essentially just an N/2 point complex FFT, plus a few extra computations for the final recombination step. This results in up to a 40% increase in performance over the complex FFT.

Window

The `Window` class is responsible for applying windowing functions to audio/data buffers. At the time of this writing, `Window` contains 6 different window functions, summarized in Figure 3.2. `Window` also includes a rectangular window, which is computed by doing *nothing* to the incoming signal and has been omitted from the graphs.

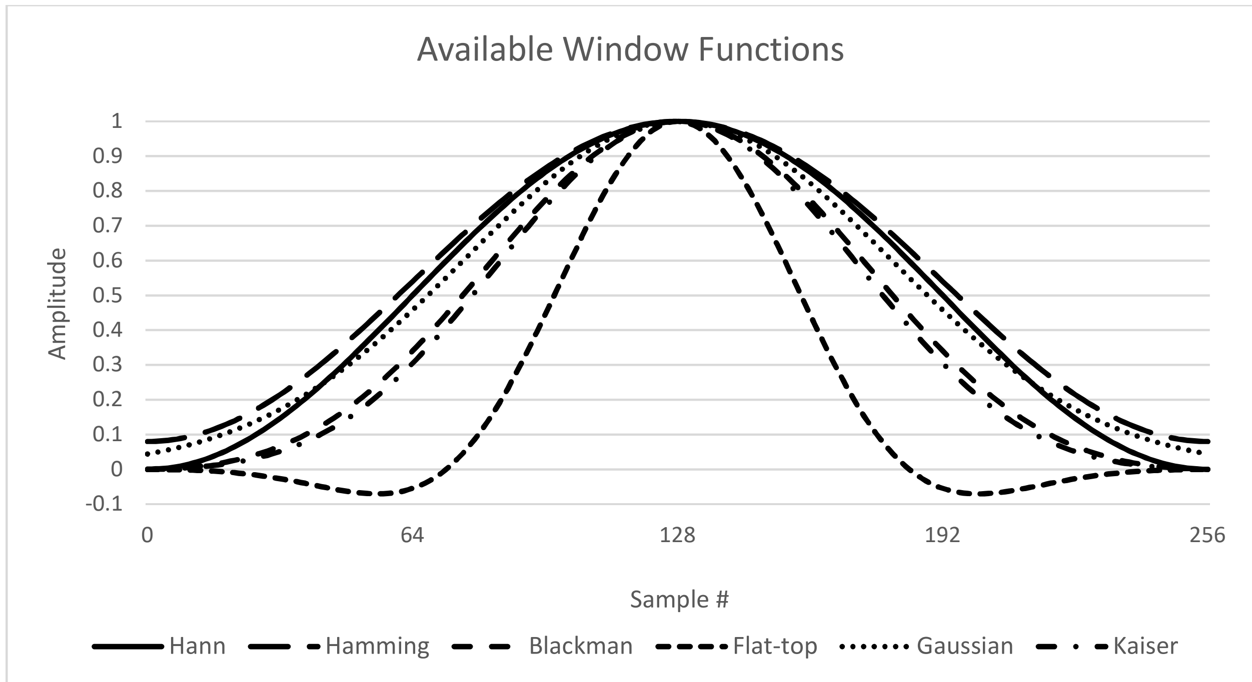


Figure 3.2: Summary of all available window functions in Tactile Waves

The response of an N-point window function can be computed by finding the normalized magnitude spectrum of the window itself, and shifting the result by $N/2$. Alternatively, the exact response can be computed directly using the following equations [25]:

$$a_r(f) = \sum_{j=0}^{N-1} w_j \cos(2\pi f j/N), \qquad (real\ part)$$

$$a_i(f) = \sum_{j=0}^{N-1} w_j \sin(2\pi f j/N), \qquad (imaginary\ part)$$

$$a(f) = \frac{\sqrt{a_r^2 + a_i^2}}{\sum_{j=0}^{N-1} w_i}$$

The `Window` class includes two `getResponse()` methods that return the response of a window function using these equations. These methods were used to generate the response of each window in the class (excluding rectangular).

The Hann window (sometimes called the Hanning window) is probably the most used window function in speech and music processing. This function is equivalent to one period of a 0.5 amplitude sine wave, shifted up by 0.5 and over by $\frac{-\pi}{2}$, and can be computed using the following simplified equation:

$$w_i = sin^2\left(\frac{\pi i}{N}\right)$$

This results in a window that is exactly zero at 0 and $N + 1$, one half at $N/4$ and $3N/4$, and one at $N/2$, leading to a side lobe roll off of about 18 dB/octave as shown in Figure 3.3.

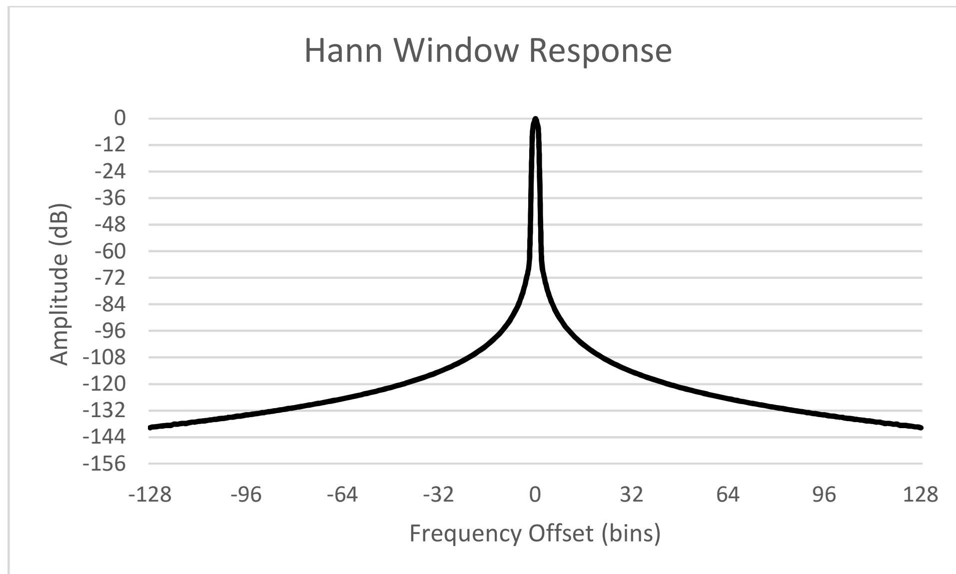


Figure 3.3: Hann Window Response

The Hamming window is very similar to the Hann window, in that it is comprised of a shifted sine wave. It is defined as one period of a 0.46 amplitude sine wave, shifted up by 0.54 and over by $\frac{\pi}{2}$. Substituting $\sin(\theta + \frac{\pi}{2})$ for $\cos(\theta)$ yields the final equation for the Hamming window:

$$w_i = 0.54 - 0.46\cos\left(\frac{2\pi i}{N}\right)$$

Unlike the Hann window, this window does not touch zero at its ends, resulting in more spectral leakage in the side lobes. The response of the Hamming window is shown in Figure 3.4 below.

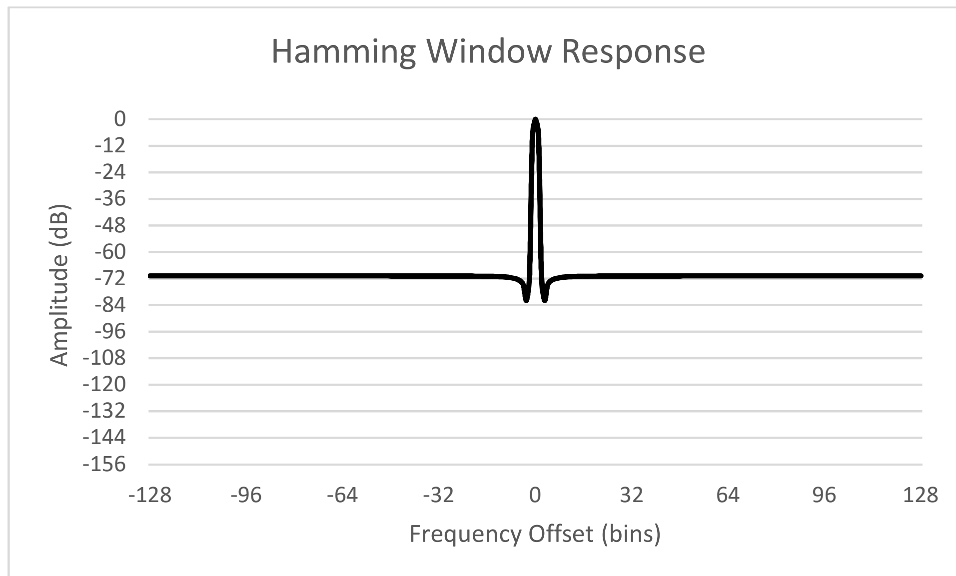


Figure 3.4: Hamming Window Response

The Blackman window is a modified version of the Hamming window that is meant to increase side lode attenuation. It is nearly identical to the Hamming window with an added sin function, defined by the following equation:

$$w_i = 0.42 - 0.5\cos\left(\frac{2\pi i}{N}\right) - 0.08\cos\left(\frac{4\pi i}{N}\right)$$

This window combines the first lobe cancelation of the Hamming window, with the improved attenuation of the Hann Window. The Blackman window response is shown below in Figure 3.5.

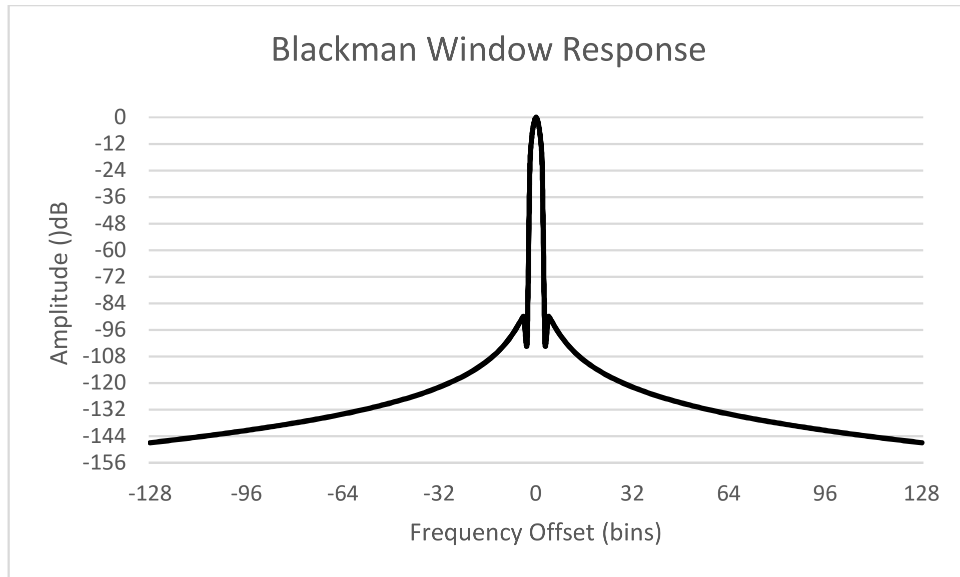


Figure 3.5: Blackman Window Response

The flat-top window differs drastically from previous windows in that it is partially negative. `Window` implements a cosine-sum version of the flat-top window, defined by the following equation:

$$w_i = 0.216 - 0.417\cos\left(\frac{2\pi i}{N}\right) + 0.277\cos\left(\frac{4\pi i}{N}\right) - 0.084\cos\left(\frac{6\pi i}{N}\right) + 0.007\cos\left(\frac{6\pi i}{N}\right)$$

This window's response can be seen in Figure 3.6. It has poor frequency resolution (indicated by the wide main lobe), but it is the best window for measuring the actual *amplitude* of a frequency component.

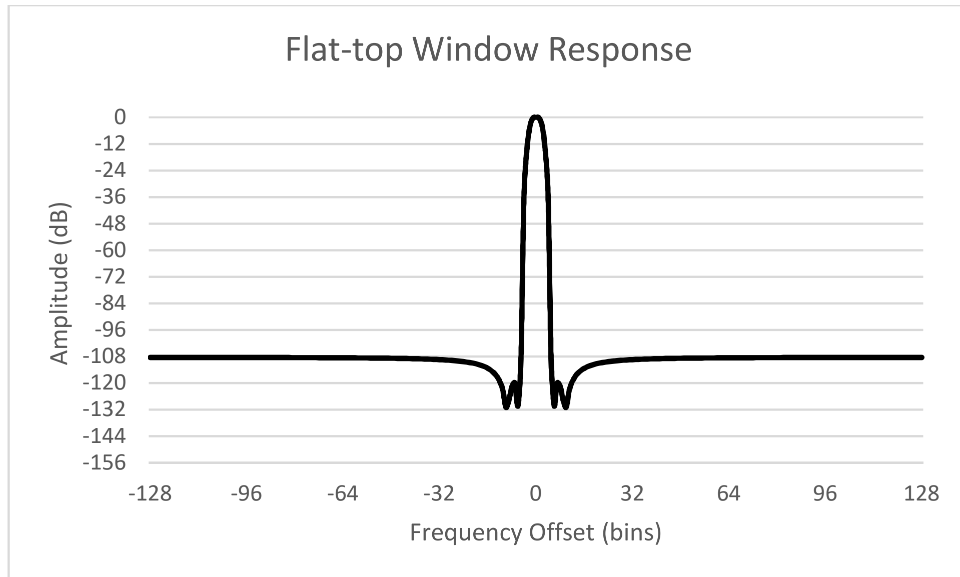


Figure 3.6: Flat-top Window Response

The Window class includes two adjustable windows that allow the user to customize the window shape to fit their needs. The first of which is the Gaussian window, defined by:

$$w_i = e^{-n^2/\sigma^2}$$

Where $n = i - \frac{N}{2}$ and $\sigma = \frac{N}{2\propto}$. The parameter $\propto$ is used to adjust the window shape. At $\propto = 2.5$, the shape of the Gaussian window is similar to the Hamming window, but with reduced sidelobe attenuation, and a slightly wider main lobe. Increasing $\propto$ results in greater sidelobe attenuation, at the cost of a wider main lobe. The response of the Gaussian window is shown in Figure 3.7.

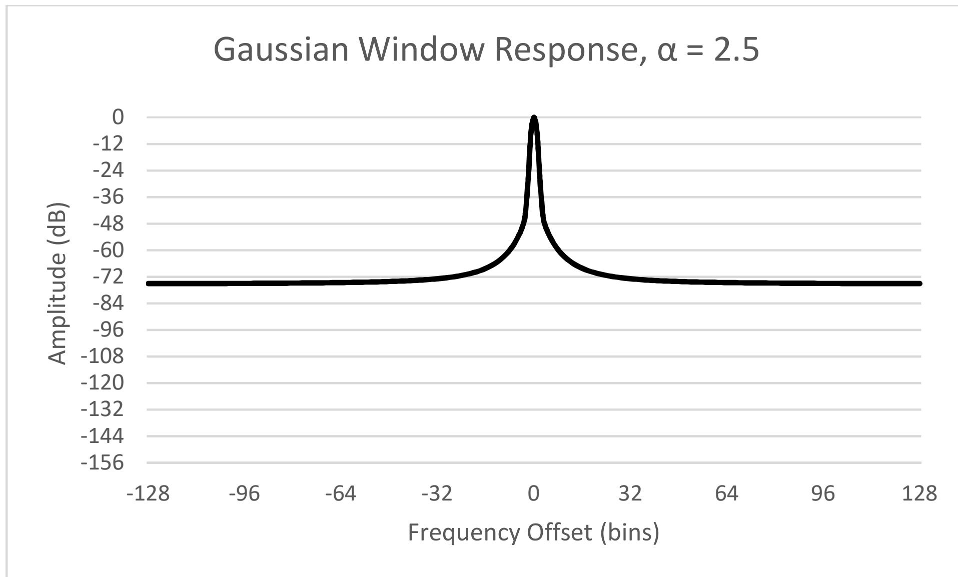


Figure 3.7: Gaussian Window Response

The second adjustable window, the Kaiser window, approximates a window known as the discrete prolate spheroidal sequence (DPSS) or Slepian window. The DPSS window maximizes the energy of the main lobe, but is very difficult to compute. Therefore, the Kaiser window is commonly used in its place. The Kaiser window function is defined by the equation:

$$w_i = \frac{I_0\left(\pi \propto \sqrt{1-\left(\frac{i-N/2}{N/2}\right)^2}\right)}{I_0(\pi \propto)}$$

Where I_0 is the zero-th order modified Bessel function of the first kind:

$$I_0(z) = \frac{1}{\pi}\int_0^{\pi} e^{z\cos(\theta)} d\theta$$

To numerically evaluate an integral of a continuous function in code, it must be discretized. Any integral can be discretely approximated using the trapezoidal rule:

$$\int_a^b f(x)\, dx \approx b - a\left(\frac{f(a)+f(b)}{2}\right)$$

The accuracy of the approximation can be increased by dividing the integral into n sub integrals, evaluating the trapezoid rule for each, and summing the results. This technique is known as the composite trapezoidal rule:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{n}\left(\frac{f(a)}{2} + \sum_{k=1}^{n-1}\left(f\left(a + k\frac{b-a}{n}\right)\right) + \frac{f(b)}{2}\right)$$

The Kaiser window uses the composite trapezoidal rule with $n = 20$ to closely approximate I_0. The window's response is shown in Figure 3.8.

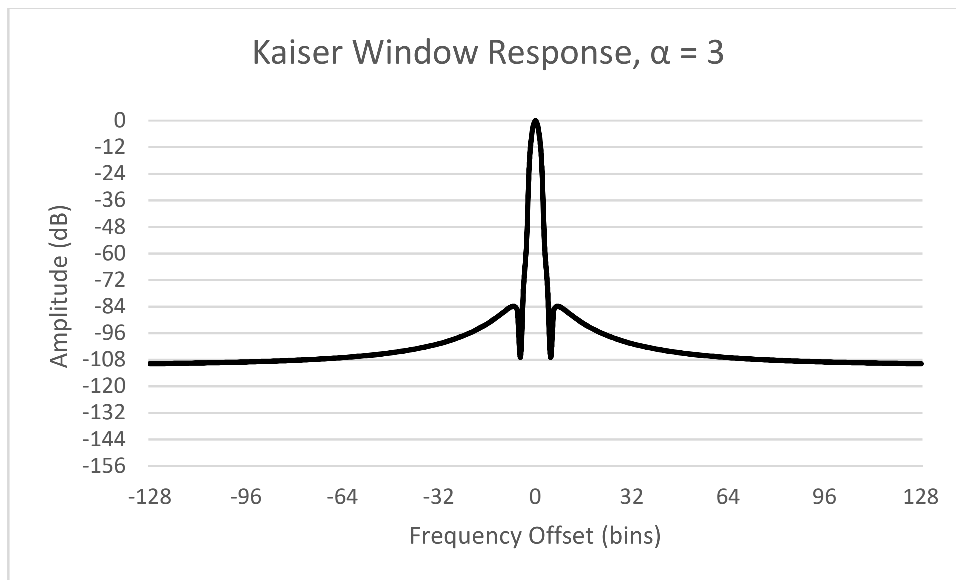


Figure 3.8: Kaiser Window Response

At $\propto = 3$, the Kaiser window resembles a combination of the Blackman and Flat-top windows, with the narrower peak like the Blackman, and flatter side lobe roll off like that of the Flat-top. Just like the Gaussian window, increasing $\propto$ results in greater sidelobe attenuation, at the cost of a wider main lobe.

DCT

The `DCT` class implements a Type-II and Type-III discrete cosine transform, or DCT. A DCT is closely related to the discrete Fourier transform (DFT), but instead represents a finite data series as a sum of cosine waves, rather than a sum of complex sinusoids. There are eight different DCT variations, four of which are commonly used in DSP. The Type-II DCT is the most common, and is therefore simply referred to as “the DCT”. The Type-II DCT of a data series x of N points is defined by:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{2N}(2n+1)k\right), \qquad 0 \le k < N$$

To find the inverse (of any DCT), k and n must be switched. In the case of the Type-II DCT, its inverse is the Type-III DCT, referred to as “the inverse DCT” or “the IDCT” and is defined for a data series x of N points by:

$$y(k) = \frac{x(0)}{2} + \sum_{n=1}^{N-1} x(n) \cos\left(\frac{\pi}{2N} n(2k+1)\right), \qquad 0 \le k < N$$

A DCT matrix is often normalized to ensure its orthogonality after transformation. For example, MATLAB uses this orthogonal normalization scheme by default in its `dct()` and i`dct()` functions. The DCT is normalized by scaling the entire matrix by $\sqrt{\frac{2}{N}}$ and dividing the $y(0)$ term by $\sqrt{2}$. The normalized DCT is then defined by:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{2N}(2n+1)k\right), \qquad 0 \le k < N$$

$$y(0) = \frac{y(0)}{\sqrt{2}}$$

Similarly, the IDCT is normalized by dividing the $x(0)$ term by $\sqrt{2}$ instead of 2, and scaling the entire matrix by $\sqrt{\frac{2}{N}}$. The normalized IDCT is then defined by:

$$y(k) = \sqrt{\frac{2}{N}}\left(\frac{x(0)}{\sqrt{2}} + \sum_{n=1}^{N-1} x(n)\cos\left(\frac{\pi}{2N}n(2k+1)\right)\right), \qquad 0 \le k < N$$

Computing an N length DCT of an N length input signal requires $O(n^2)$ processing complexity with any of the above equations. Fortunately, there is an alternate algorithm that takes advantage of the DCT's close relation to the DFT to perform the transformation with reduced complexity. The Type-II DCT of an N-length input signal is exactly equivalent to a DFT of a $4N$-length signal with even symmetry and all the even indexed input values set to zero. Because the DFT can be computed with the $O(n \log n)$ FFT algorithm, this alternative is more performant. However, the $4N$-length FFT limits the algorithms performance, making it slower than the standard DCT at smaller input sizes. Therefore, for optimal performance the algorithm must be modified to use an N-length FFT. The modified DCT and IDCT algorithms are summarized in Table 10 and 11 with an example 10-point input signal. These algorithms are computationally equivalent to a single N-length real FFT, plus some additional overhead for the rearranging and multiplication.

Table 10: Modified DCT algorithm with 10-point input

10-point input signal	0	1	2	3	4	5	6	7	8	9
					↓					
Rearrange input	0	2	4	6	8	9	7	5	3	1
					↓					
Take real FFT of input	realFFT() ↓									
Multiply by half sample shift	$* e^{\frac{-i\pi}{2N}k}$ ↙ ↘									
Discard imaginary	real					~~imaginary~~				

Table 11: Modified IDCT algorithm with 10-point input

10-point input signal	0	1	2	3	4	5	6	7	8	9
Multiply by half sample shift	$* \, e^{\frac{-i\pi}{2N}k}$									
Take real FFT of input	complexFFT()									
Discard imaginary	real					imaginary				
Real part	0	2	4	6	8	9	7	5	3	1
Rearrange output	0	1	2	3	4	5	6	7	8	9

The DCT is primarily used for its energy compaction property: The DCT of a signal will contain nearly all the signals energy in the first few DCT bins. That is, nearly all the signal's energy (information) will be contained in a small number of the transformed samples. This property can be seen in Figure 3.9 for the 10-point input signal $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$.

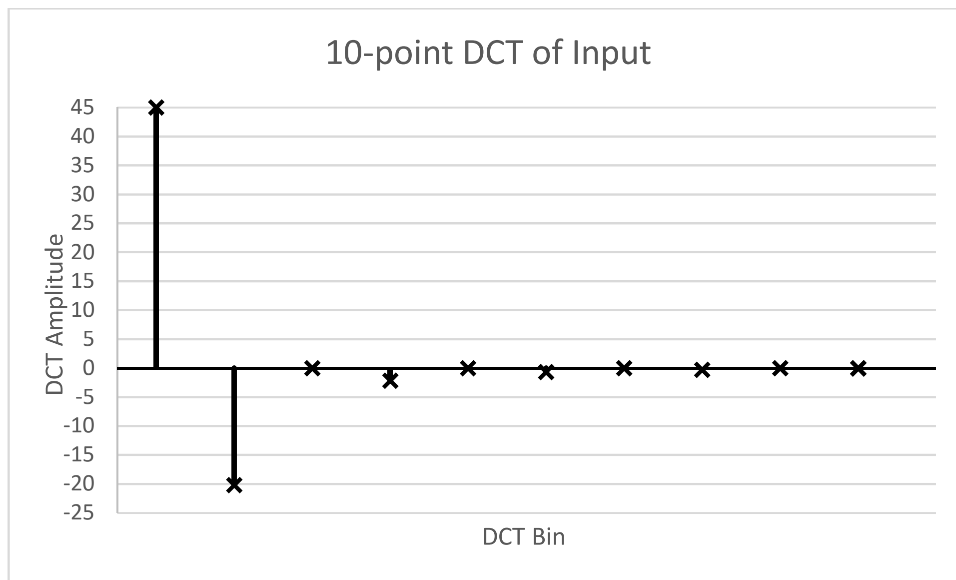


Figure 3.9: Type-II DCT of Input Sequence

Here, the DCT of $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ contains nearly all the signals energy in the first 2 bins. The other 8 bins are nearly zero. In fact, the first 2 bins contain 99.89% of the original signal's total

energy, and the original signal can be represented using only these first 2 bins. By discarding the remaining 8 bins, and taking the IDCT of the 2 retained bins, the original signal can be reconstructed. The reconstructed signal is shown overlaid with the original signal in Figure 3.10.

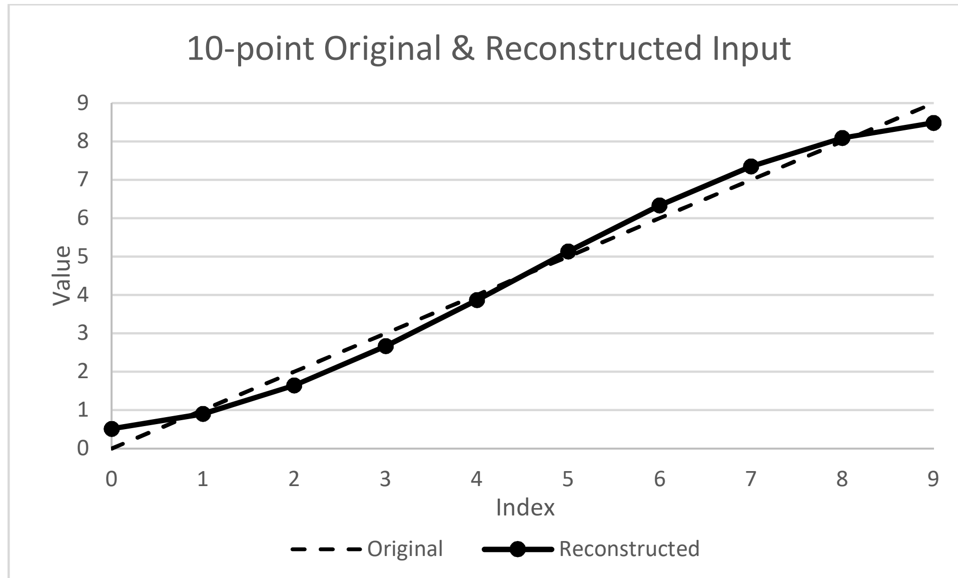


Figure 3.10: Input reconstructed via DCT with 80% data reduction

After an 80% data reduction, the signal can still be reconstructed with only a small amount of error. This energy compaction by the DCT is the primary mechanism used in JPEG image compression and MP3 audio compression.

Filter

The `Filter` class aims to be a comprehensive data filtering toolkit for the Tactile Waves library. In DSP applications, filters are used to remove, attenuate, or amplify specific frequency regions in an audio signal. `Filter` implements a single, general purpose recursive filter algorithm in the method `filter()`. This method accepts 3 input arrays containing the filter's numerator coefficients, denominator coefficients, and the signal/data series to be filtered. The data array can be an array of single or double precision floating point numbers, while the coefficients must be double precision

floating point arrays. Both finite impulse response (FIR) and infinite impulse response (IIR) filters can be described by the rational transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_{n_b} z^{-n_b}}{a_0 + a_1 z^{-1} + \cdots + a_{n_a} z^{-n_a}}$$

Where $X(z)$ is the Laplace Transform of the input signal x, $Y(z)$ is the Laplace Transform of the filtered output signal y, b is the list of n_b numerator coefficients, and a is the list of n_a denominator coefficients. The normalized form of this transfer function divides all the coefficients by a_0:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\frac{b_0}{a_0} + \left(\frac{b_1}{a_0}\right) z^{-1} + \cdots + \left(\frac{b_{n_b}}{a_0}\right) z^{-n_b}}{1 + \left(\frac{a_1}{a_0}\right) z^{-1} + \cdots + \left(\frac{a_{n_a}}{a_0}\right) z^{-n_a}}$$

$H(z)$ can then be expressed as the difference equation:

$$a(0)y(n) = b(0)x(n) + b(1)x(n-1) + \cdots + b(n_b)x(n-n_b) - a(1)y(n+1) - \cdots - a(n_a)y(n-n_a)$$

Which can be easily implemented in code. Here the numerator coefficients, b, are called the feedforward coefficients, and n_b is the feedforward filter order. Similarly, the denominator coefficients, a, are called the feedback coefficients, and n_a is the feedback filter order. `Filter` uses this equation in the `filter()` method to execute all filtering operations that `Filter` performs.

The `filter()` method is a static method, so it does not require instantiation. However, a filter can be saved and reused by instantiating a `Filter` object. The constructer takes two arguments: an array of numerator coefficients and an array of denominator coefficients. These coefficients are saved with the object, and the non-static version of `filter()` can be called at any time to perform filtering with the saved coefficients.

Custom digital filters can be designed from analog prototypes using the `allPole()` `biquad()`, and `chebyshev()` methods. The `allPole()` method can be used to design 2-pole Butterworth, Critically damped, and Bessel IIR filters. Both low and high pass filters are supported, as well as filter cascading. The 3 filter types are compared in Figure 3.11 for a single 100 Hz low-pass design. High-pass versions of these filters would produce an identical response, flipped horizontally about the critical frequency of 100 Hz.

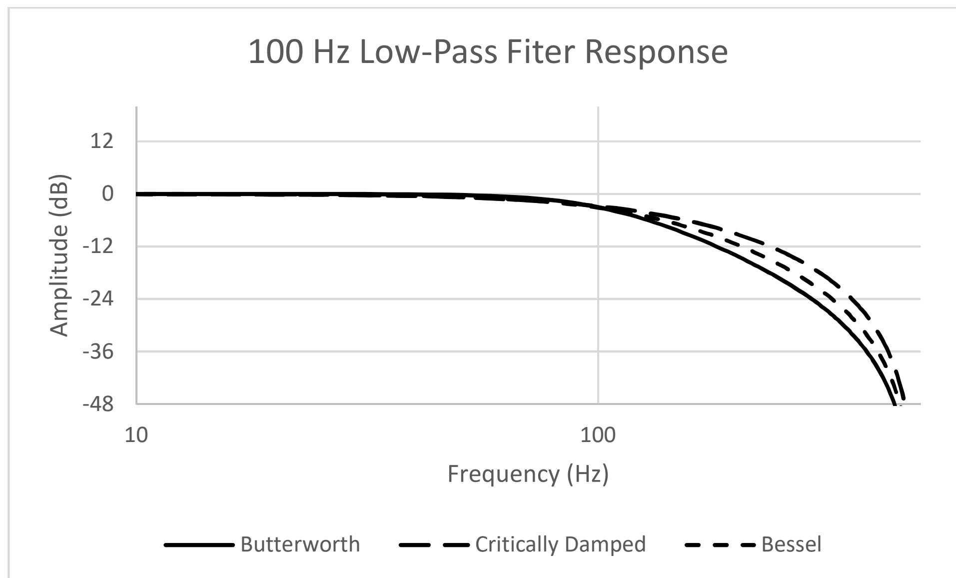


Figure 3.11: Low-pass filter responses of all pole filters

The roll-off of these all-pole filters can be improved by cascading the filter, or passing the signal through multiple times (after adjusting the filter coefficients accordingly). An example of this is shown in the following figure for a Butterworth filter with various passes, or cascades. As the number of filter passes in increased, the stopband roll-off increases.

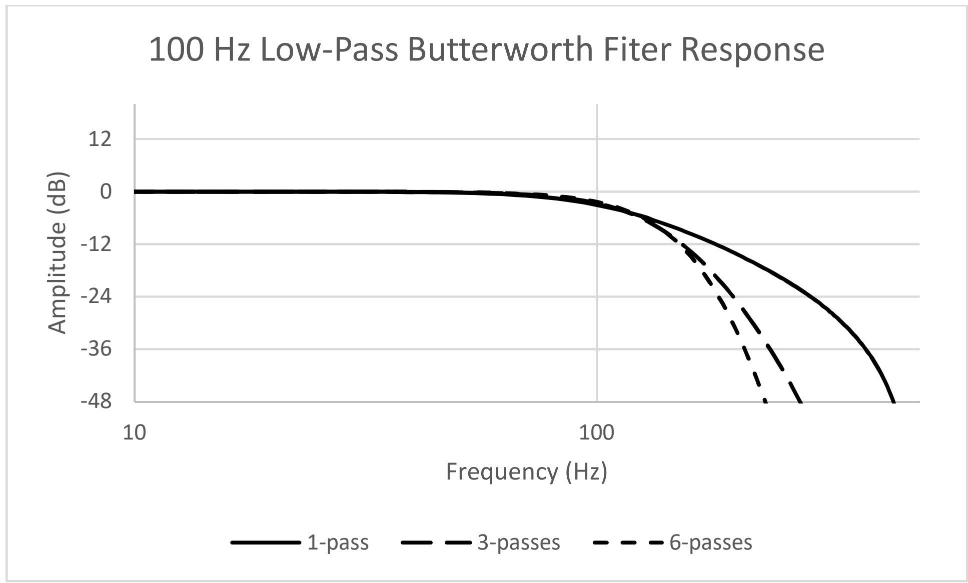
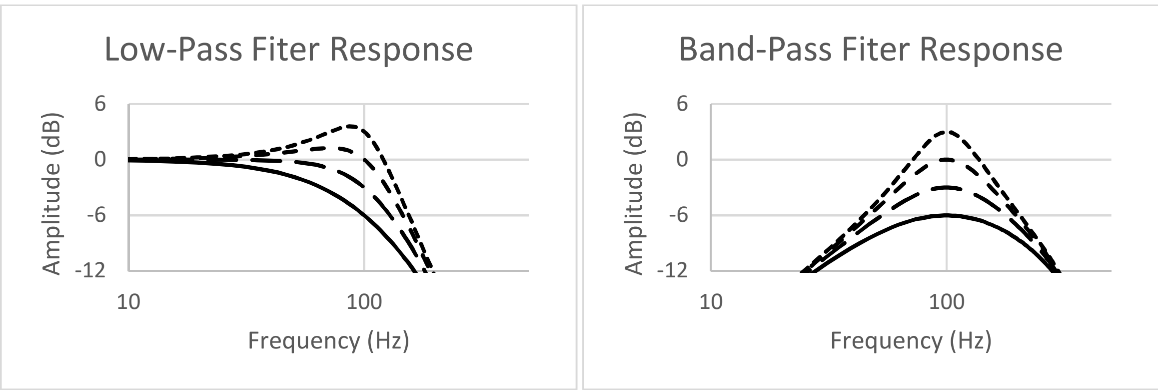


Figure 3.12: Example of Butterworth filter cascading

The biquad() method is capable of designing a diverse pallet of adjustable biquadratic filters. Low/high-pass, all-pass, band-pass, notch, peak (bell), and low/high shelf filter shapes are supported, and each can be adjusted through the design parameters. Each of these filter shapes is shown in Figure 3.13 for a critical frequency of 100 Hz. Lines with smaller dashes indicate an increase in the Q/gain parameter of the filter. High-pass and high-shelf filters have been omitted, as their response is identical to their low-pass and low-shelf counterparts, flipped horizontally about the critical frequency.



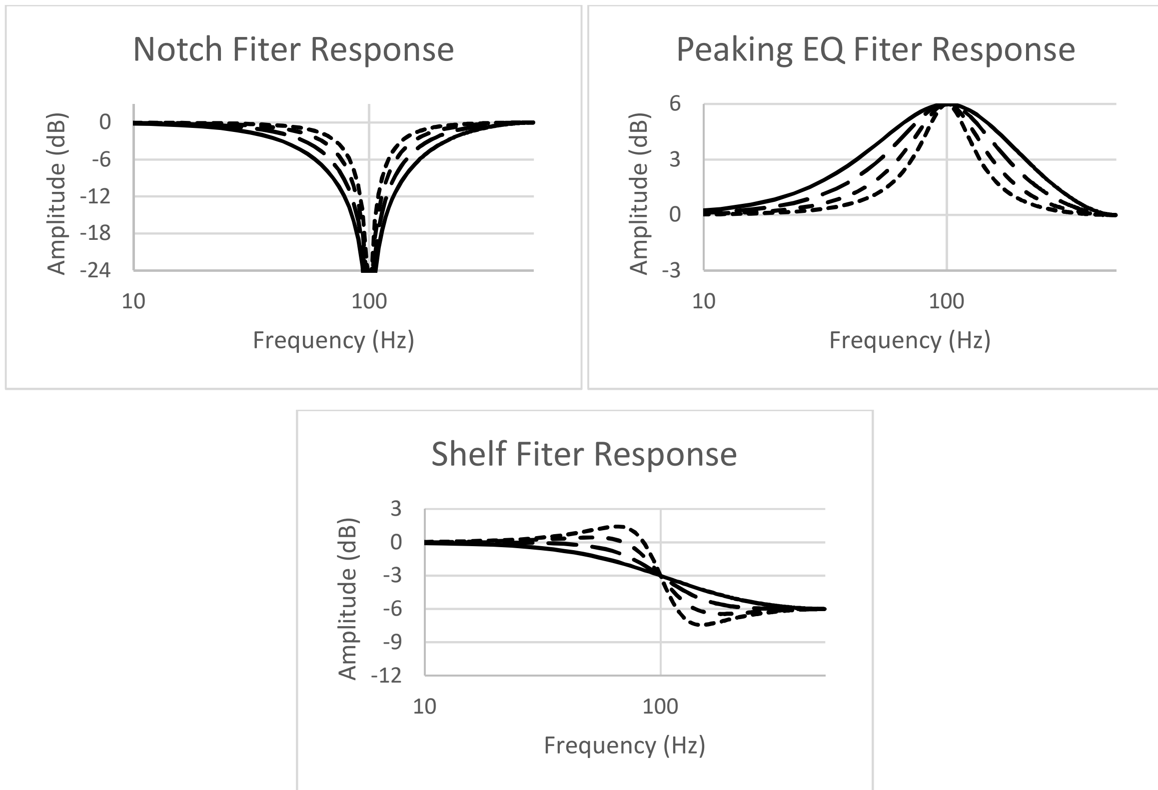


Figure 3.13: Biquadratic Filter Responses

The chebyshev() method allows for the design of N-pole digital Chebyshev IIR filters with adjustable passband ripple. Doubling the number of poles will double the slope of the stopband roll-off at the cost of stability (Figure 3.14), and increasing the passband ripple will increase slope of the stopband roll-off at the cost of ripple in the passband (Figure 3.15). With the potential for such sharp stopband attenuation, the Chebyshev filters are intended to be used in situations that require exact frequency separation, such as resampling.

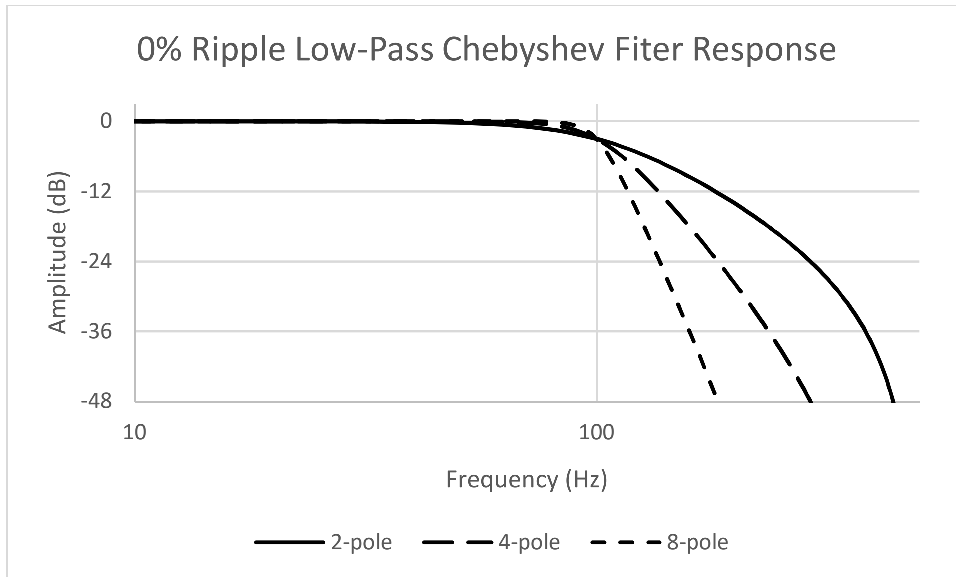


Figure 3.14: 0% ripple Chebyshev filter response with varying number of poles

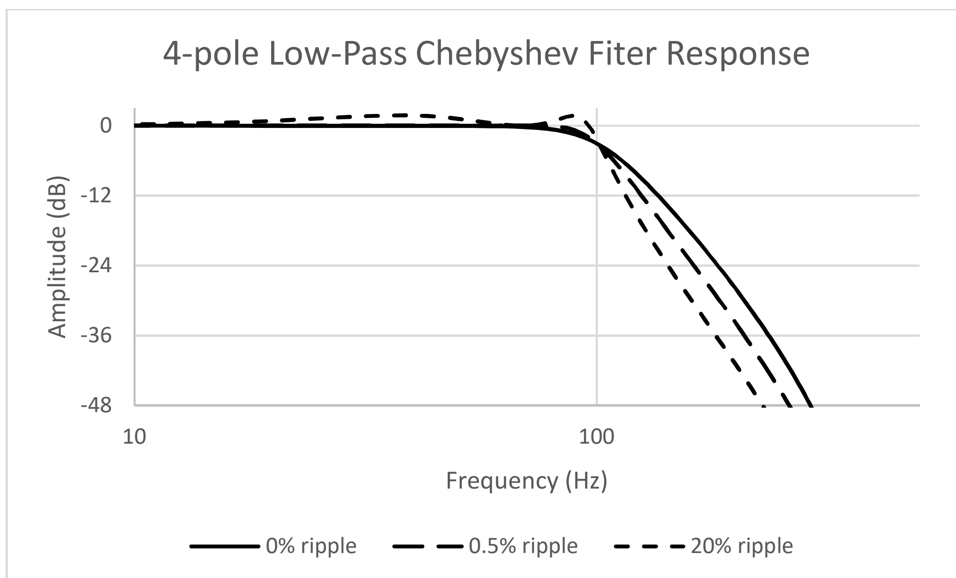


Figure 3.15: 4-pole Chebyshev filter response with varying passband ripple

Cepstrum

The `Cepstrum` class is responsible for performing cepstral transforms on audio data/signals. Much like how a Fourier transform computes the spectrum of a time domain signal by converting it into the frequency domain, a cepstral transform computes the "cepstrum" of a time domain signal by converting it into the "quefrency" domain. The names "cepstral" and "cepstrum" come from reversing

the first four letters of the words "spectral" and "spectrum", and "quefrency" is a reordering of the letters in "frequency". Similarly, a filtering like operation in the quefrency domain is called "liftering". As the name might imply, the cepstral transform is closely related to the Fourier (spectral) transform. A *cepstrum* is defined as the inverse Fourier transform of the logarithm of the spectrum of a signal, and there are several variations based on the type of spectrum used. `Cepstrum` implements the *real cepstrum*, *power cepstrum*, and *complex cepstrum*. The real cepstrum is found by taking the inverse Fourier transform of the log of the *magnitude* spectrum:

$$real\ cepstrum = IFFT(\log(|FFT(x)|))$$

Whereas the power cepstrum uses the *power* spectrum, rather than the magnitude spectrum:

$$power\ cepstrum = IFFT(\log(|FFT(x)|^2))$$

Because the magnitude and power spectrums throw away the signal phase, the original signal cannot be reconstructed from these spectra alone. Similarly, the original signal cannot be reconstructed from the real or power cepstra. When reconstruction of the original signal after transformation is required, the complex cepstrum is used. The complex cepstrum is the result of the inverse Fourier transform of the *complex* log of the *complex* spectrum:

$$complex\ cepstrum = IFFT\big(\log\big(FFT(x)\big)\big)$$

The original signal can then be reconstructed by reversing these steps:

$$inverse\ complex\ cepstrum = IFFT(e^{FFT(x)})$$

Where e is the complex exponential function (the inverse of the complex logarithm).

Cepstral analysis takes advantage of a basic property of logarithms that makes it an important tool in speech processing. The product rule of logarithms states that the logarithm of a product can be separated into a sum of the log of its factors:

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

By defining $FFT(x) = FFT(s) \times FFT(f)$, the definition of a cepstrum can then be rewritten as:

$$cepstrum = IFFT\big(\log\big(FFT(s)\big) + \log\big(FFT(f)\big)\big)$$

Which shows that multiplication in the frequency domain transforms into a linear combination in the cepstral domain. Because multiplication in the frequency domain is equivalent to convolution in the time domain, the cepstral transform turns a time domain convolution into a linear combination (summation) of its components. Unlike convolution, a linear combination can be easily separated. In this way, the cepstral transform can be used for deconvolution, which is achieved through a process known as "liftering"

Liftering is a similar operation to filtering in the frequency domain where a signal's spectrum is multiplied by a desired frequency response to attenuate or amplify certain frequency regions. Liftering is the process of removing either high-time or low-time quefrency components from the cepstrum. Low-time liftering is achieved by multiplying the cepstrum by a rectangular window that covers the quefrency region of interest. A low-time liftering window can be defined as:

$$w(n) = \begin{cases} 1, & 0 \le n \le L_c \\ 0, & L_c < n \le \dfrac{N}{2} \end{cases}$$

Where L_c is the cutoff length of the liftering window, in samples. This window will zero out all quefrency components with periods longer than L_c , and leave the rest of the cepstrum intact. This operation effectively removes low frequency content from the cepstrum.

Similarly, high-time liftering applies a rectangular window that preserves quefrency components above the cutoff, L_c. A high-time liftering window is therefore the opposite of the low-time liftering window:

$$w(n) = \begin{cases} 0, & 0 \leq n < L_c \\ 1, & L_c \leq n \leq \frac{N}{2} \end{cases}$$

This window will zero out all quefrency components with periods shorter than L_c , which effectively removes high-frequency content from the cepstrum.

MFCC

The `MFCC` class computes the Mel-frequency cepstral coefficients (MFCC's) of an audio signal. The MFCC's are simply a group of numbers that collectively make up the Mel-frequency cepstrum (MFC), which is a modified power cepstrum. The MFC is defined as the discrete cosine transform of the logarithm of the Mel-frequency centered triangular filtered power spectrum of a signal, mapped to the Mel scale. The steps to compute the MFC are as follows:

1. Compute the N-length power spectrum of the signal
2. Compute the M Mel-filter bank energies
3. Take the logarithm of the filter bank energies
4. Compute the discrete cosine transform of the log filter bank energies to yield cepstral coefficients, or MFCCs
5. Apply liftering operation (optional step)

To convert a frequency in Hz to the Mel scale, the following equation is used:

$$M(frequency) = 2595 \log_{10}(1 + \frac{frequency}{700})$$

To convert from the Mel scale back to frequency, the inverse operation is performed:

$$M^{-1}(mel) = 700(10^{mel/2595} - 1)$$

These two equations are used to compute the filter bank FFT bins, $f(i)$:

$$f(i) = floor\left(\frac{(N+1)M^{-1}(mel(i))}{f_s}\right)$$

Where $0 \leq i \leq M + 1$, and

$$mel(i) = mel(0) + \frac{i}{M+1}\big(mel(M+1) - mel(0)\big)$$

$$mel(0) = M(minimum\ frequency)$$

$$mel(M+1) = M(maximum\ frequency)$$

Which defines the start, center, and ending FFT bin for each triangular filter in the filter bank. The mth filter will start at $f(m-1)$, reach unity at $f(m)$ and end at $f(m+1)$. The filter bank can then be defined as:

$$h_m(k) = \begin{cases} 0 & k < f(m-1) \\ \dfrac{k - f(m-1)}{f(m) - f(m-1)} & f(m-1) \leq k \leq f(m) \\ \dfrac{f(m+1) - k}{f(m+1) - f(m)} & f(m) \leq k \leq f(m+1) \\ 0 & k > f(m+1) \end{cases}$$

Where $1 \leq m \leq M$ and $0 \leq k \leq N/2$. An example of a Mel-filter bank running with 10 banks in the range 0-8000 Hz is shown in Figure 3.16.

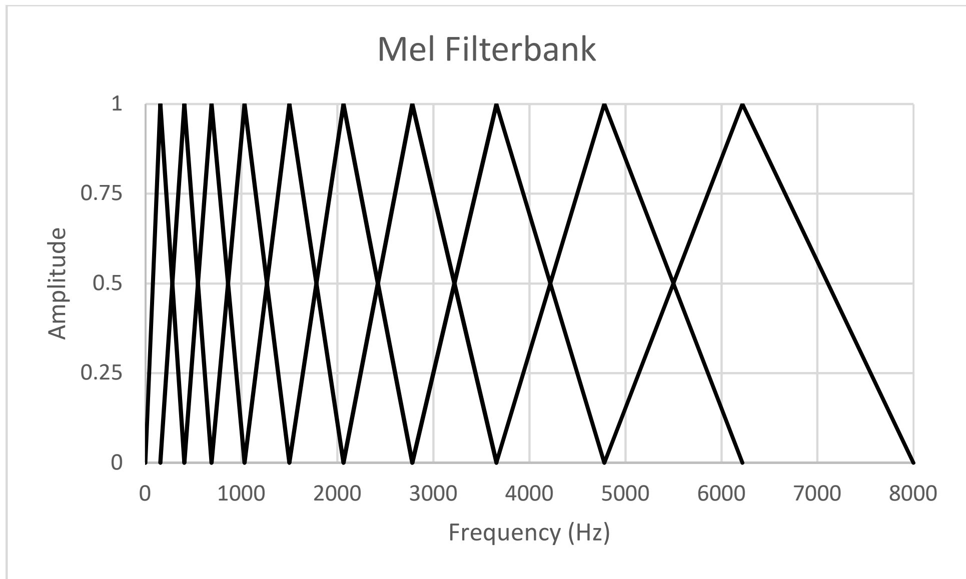


Figure 3.16: A 10-band Mel filter bank from 0-8000 Hz

The entire process for computing the MFCCs of a signal can now be defined by the following equation:

$$MFCC_m = DCT_m\left(\log\left(|FFT(x)|^2 * h_m(k)\right)\right)$$

The optional liftering step above amounts to simply point-wise multiplying the MFCC vector by a weighting function w_i:

$$MFFC_i = w_i * MFFC_i$$

Many different weighting functions exist for various types of liftering. `MFCC` includes three popular liftering methods: linear, sinusoidal, and exponential. The equations for each lifter are listed in Table 12

Table 12: MFCC Liftering Methods

Linear [K.K. Paliwal]	Sinusoidal [Biing-Hwang Juang]	Exponential [Mike Brookes]
$w_i = i$	$w_i = 1 + \frac{D}{2}\sin\left(\frac{\pi i}{D}\right)$	$w_i = i^s e^{-i^2/2\tau^2}$ $s = 1.5, \tau = 5$

LPC

The `LPC` class implements a linear predictive coding algorithm using an autocorrelation method. Linear predictive coding attempts to provide a compressed representation of the spectral envelope of a speech signal using a linear prediction model. A Pth-order FIR filter is fit to a data series x that predicts the current value of x based on previous values:

$$x_p(n) = a(1)x(n-1) + a(2)x(n-2) + \cdots + a(P)x(n-P)$$

Where a is a vector containing the models $P+1$ prediction coefficients, and $a(0) = 1$. `LPC` finds the optimal coefficients by solving the Yule-Walker system of equations:

$$Ra = r$$

Where

$$R = \begin{bmatrix} r(0) & r^*(1) & \cdots & r^*(n-1) \\ r(1) & r(0) & \ddots & r^*(n-2) \\ \vdots & \vdots & \ddots & \vdots \\ r(P-1) & r(P-2) & \dots & r(0) \end{bmatrix}, r = \begin{bmatrix} -r(1) \\ -r(2) \\ \vdots \\ -r(P) \end{bmatrix}$$

And $r(m)$ is the autocorrelation function of a signal x, defined as follows:

$$r(m) = \sum_{n=0}^{N-1-m} x(n)x(n+m)$$

Because R is a *Toeplitz* (diagonal-constant) matrix, the system can be solved for a in $O(n^2)$ flops using Levinson-Durbin recursion.

Levinson-Durbin recursion recurses over the model order to calculate the coefficients for a Pth-order predictor from a $(P-1)^{th}$-order predictor, denoted by the subscripts P, $P-1$, etc. The system of equations representing the $(P-1)^{th}$-order predictor, can be found by omitting the last row and column from R_P:

$$R_{P-1}\begin{bmatrix} a_P(1) \\ a_P(1) \\ \vdots \\ a_P(P-1) \end{bmatrix} = r_{P-1} - a_P(P)\hat{r}_{P-1}$$

Multiplying by in inverse of R_{P-1} gives:

$$\begin{bmatrix} a_P(1) \\ a_P(1) \\ \vdots \\ a_P(P-1) \end{bmatrix} = {R_{P-1}}^{-1} r_{P-1} - {R_{P-1}}^{-1}\hat{r}_{P-1} a_P(P)$$

Substituting $a = R^{-1}r$ yields the final equation for calculating the predictor coefficients:

$$\begin{bmatrix} a_P(1) \\ a_P(1) \\ \vdots \\ a_P(P-1) \end{bmatrix} = a_{P-1} - a_P(P)\hat{a}_{P-1}$$

The ith coefficient for a Pth-order model can be found from:

$$a_P(i) = a_{P-1}(i) - a_P(P)a_{P-i}(P-1)$$

Where

$$a_P(P) = \frac{r(P) - \hat{r}_{P-1}a_{P-1}}{r(0) - \hat{r}_{P-1}\hat{a}_{P-1}}$$

and

$$\hat{r}_P = \begin{bmatrix} r_P(P) \\ r_P(P-1) \\ \vdots \\ r_P(1) \end{bmatrix}, \hat{a}_P = \begin{bmatrix} a_P(P) \\ a_P(P-1) \\ \vdots \\ a_P(1) \end{bmatrix}$$

`LPC` computes the autocorrelation of the input signal x (using `FFT`), then performs Levison-Durbin recursion as described above to find the $P+1$ coefficients of the predictor. These coefficients

define a Pth-order polynomial that approximates the spectral envelop of the signal. This can be used for a variety of useful processing. For example, the formant frequencies of the signal can be found from the roots of the polynomial. For each complex root z, with a positive imaginary component, the corresponding formant frequency and bandwidth can be found from the following equations:

$$frequency = \arg(z) * \frac{f_s}{2\pi}, bandwidth = \log(mag(z)) \frac{-f_s}{4\pi}$$

LPC provides two static methods, `lpc()` and `estimateFormants()`. The `lpc()` method takes a signal x, and an integer specifying the model order, P and returns the $P + 1$ predicator coefficients. The `estimateFormats()` method takes a signal x, and two integers specifying the number of requested formants, and the sampling rate of the signal. The method first calls lpc() with a model order of $2 * \# \, of \; formants + 2$. The roots of the polynomial are found using `RootSolver`, and the formant/bandwidth pairs are calculated and returned.

YIN

The `YIN` class implements the YIN pitch detection algorithm (PDA), as described by de Cheveigné and Kawahara in [21]. A pitch detection algorithm attempts to estimate the fundamental frequency, or pitch, of an audio signal. The fundamental frequency, F_0, is defined as the inverse of the period of a periodic waveform, where the period is defined as the smallest time shift that leaves the signal invariant. Pitch estimation techniques based on autocorrelation attempt to exploit this assumption by comparing a signal to many time delayed versions of itself. If there is a delay time that appears to leave the signal unchanged, then that delay time must be equal to the period of that signal. The autocorrelation function (ACF) is used to accomplish this task efficiently, as it is a measure of the correlation between a signal and a delayed version of itself, as a function of the delay time. In the Yin paper, the autocorrelation function of a signal x is defined as:

$$r(\tau) = \sum_n x_n x_{n+\tau}$$

Which yields the autocorrelation of x at a lag of τ samples. To demonstrate the relation between auto correlation and pitch, consider a 0 dbFS 10 Hz sine wave that has been corrupted with noise at an SNR of 0 dB and sampled at 200 Hz. A 128-sample chunk of this waveform is shown in Figure 3.17.

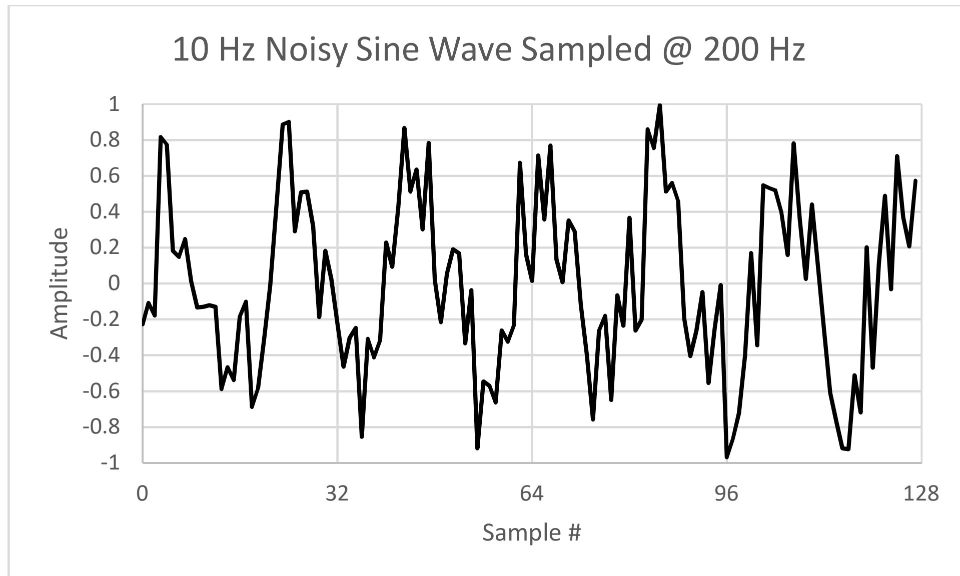


Figure 3.17: 10 Hz sine wave + noise sampled @ 200 Hz

The autocorrelation function of this sine wave is shown in Figure 3.18 for the range $0 < \tau < 64$ samples.

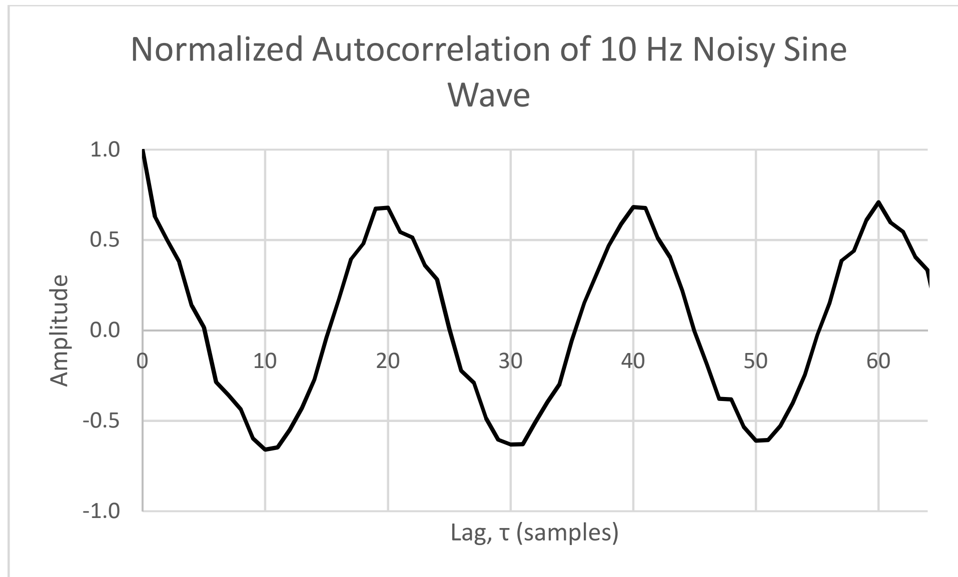


Figure 3.18: Autocorrelation of noisy sine wave

The ACF peaks at 0 samples (no lag), and every integer multiple of 20 samples. The period (in samples) of the waveform is found by selecting the smallest non-zero lag which contains a peak. Dividing the sampling rate (200 Hz) by this value (20 samples), yields the pitch of the signal – 10 Hz. This is the fundamental idea behind autocorrelation based pitch estimation.

The Yin algorithm implements an autocorrelation based pitch estimation, with the addition of several error correction steps that make it one of the most accurate PDAs available [YIN]. Apart from the additional steps, the algorithm has one key difference from the procedure described above. Instead of using the autocorrelation function, the difference function is used. The difference function of a signal x is defined by:

$$d(\tau) = \sum_{n} (x_n - x_{n+\tau})^2$$

Which can be written in terms of the autocorrelation function by:

$$d(\tau) = \sum_{n} x_n x_n - 2x_n x_{n+\tau} + x_{n+\tau} x_{n+\tau} = r(0) + r_\tau(0) - 2r(\tau)$$

The first step in the algorithm is to compute this difference function, followed by a normalization step to normalize the difference function using its cumulative mean. The equation to compute the cumulative mean normalized difference function is defined as:

$$d'(\tau) = \begin{cases} 1 & \tau = 0 \\ \dfrac{d(\tau)}{\frac{1}{\tau}\sum_{j=1}^{\tau} d(j)} & else \end{cases}$$

Where the ACF shows the amount of correlation between a signal and a time delayed copy, the difference function shows the how different a signal is from a time shifted copy. For example, the normalized difference function of the noisy sine wave used above is shown below in Figure 3.19.

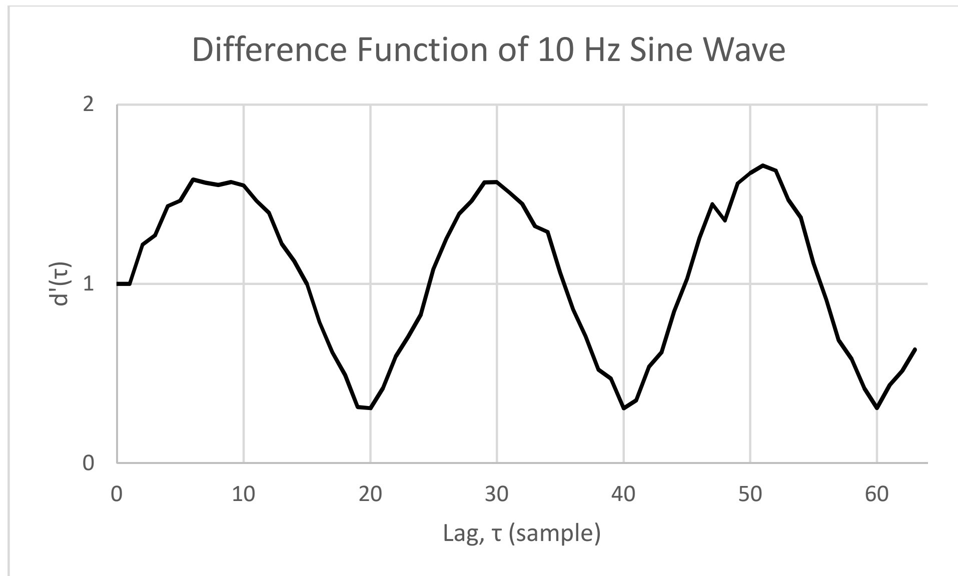


Figure 3.19: Cumulative normalized difference function of noisy sine wave

Here, the function is at a minimum at lag values that are integer multiples of 20 samples. Because the added noise breaks the perfect periodicity of the sine wave, the difference function is non-zero everywhere. A perfectly periodic signal would result in zeros at integer multiples of the period, as any time delay that is evenly divisible by the period leaves the signal invariant.

It is possible for the difference function to contain dips at lag values greater than the period, that are lower than the dips at the actual period. To prevent the algorithm from choosing this incorrect

period, an absolute thresholding scheme is employed as the next step in the algorithm. A threshold is set in the program (or by the user/programmer) and the smallest τ value that gives a minimum of $d'(\tau)$ that is below the threshold is chosen. If no value is found that satisfies the threshold, the global minimum of the difference function should be returned instead. However, in testing this proved to be problematic. The algorithm will *always* report a pitch, even in the presence of un-pitched sounds, such as white noise. A popular solution to this problem is to compute the probability of the estimated pitch, and return this value with the pitch. The program must then perform some post-processing to determine if the pitch reading is valid, based on the probability, the past readings, etc. It would be preferable for the algorithm to report the pitch when a pitched sound is present, and no pitch otherwise. The algorithm was modified slightly to accomplish this. Rather than returning the global minimum if the threshold is not satisfied, `YIN` returns -1, indicating that no pitch was found. Because speech is composed of both pitched (voiced) and un-pitched (unvoiced) sounds, `YIN` can be used as a voiced/unvoiced speech detector, as well as a pitch estimator, simultaneously.

The final steps of the Yin pitch detection algorithm are about refining the period estimate obtained from previous steps. The first is called the “best local estimate” and in it, the value of the normalized difference function at the current τ estimate at time t is compared to values of the difference function in the temporal vicinity $\left[t - \frac{T_{max}}{2}, t + \frac{T_{max}}{2}\right]$ to find a minimum. The Yin paper describes a file based pitch detection, in which the difference function of the entire audio clip is computed at once, allowing the algorithm to look forward into time. Because Tactile Waves implements real-time audio processing, this is impossible to achieve. Instead, `YIN` examines the current audio buffer and searches for a minimum value of the difference function in the range $[\tau - \tau/5, \tau + \tau/5]$.

The final step of the algorithm uses parabolic interpretation to refine the estimated period to a more precise value. Because the lag τ is measured in samples, it is restricted to whole integer numbers.

This limits the maximum accuracy of the reported pitch, as the actual pitch period could fall on a fractional number of samples. For example, the pitch of the note C3 is 261.6 Hz, corresponding to a period of 168.578 samples at a sampling rate of 44100 Hz. Without parabolic interpolation, the algorithm would be report a period of 168 or 169 samples, corresponding to a pitch of 262.5 or 260.947 Hz, respectively. The dip in the normalized difference function corresponding to the estimated τ is modeled as a quadratic function, and the curve is fitted with a parabola and the interpolated minimum is retuned as the final estimated pitch period. The sampling rate of the signal is divided by this value to yield the final pitch estimate.

YIN provides two static methods to estimate the pitch of a signal using the Yin PDA: `estimatePitch()` and `estimatePitchFast()`. The former computes $d(\tau)$ directly from the signal x, as described above, which requires $O(n^2)$ flops. The latter method uses the `FFT` class to compute the autocorrelation of x, then computes $d(\tau)$ from the autocorrelation, resulting in only $O(n \log n)$ flops, and roughly a 90% speed increase over `estimatePitch()`. Other than the differences in computing the difference function, these two methods are completely identical.

ZCR

The `ZCR` class computes the zero-crossing rate of a signal. `ZCR` implements a single static function called `getZCR()`. This method returns the rate that the input signal crosses zero, in units of crossings per sample. This reading can be converted to Hz by multiplying by the signals sample rate. For example, the signal [1.0, 0.5, -0.5, -1.0] contains 1 zero-crossing at a zero-crossing rate of 1/4 crossings/sample. If this signal was sampled at 100 Hz, this would result in a zero-crossing frequency of 25 Hz.

The method `getZCR()` accepts a single argument that contains the signal to analyze. The input signal can be stored in an array of single or double precision floating point numbers, or contained within

a `WaveFrame` object. This method traverses the entire input signal sample-by-sample, and keeps a count of the number of times the signal value changes sign (resulting in $O(n)$ computational complexity). This counter is then divided by the total number of samples in the input signal, and the result is returned.

The `ZCR` class is useful for performing a quick estimate of the noisiness of a signal, or checking the voiced/unvoiced state of a speech signal. A signal with a large amount of high frequency information, or a signal that has been corrupted with wide-band noise will have a higher zero-crossing rate than a signal dominated by low frequency content. For example, voiced speech sounds are pitched, and pitched sounds contain a dominant low frequency component corresponding to the fundamental frequency, whereas unvoiced speech sounds are unpitched and do not have a dominant frequency. Therefore, regions of voiced speech will exhibit lower zero-crossing rates than regions of unvoiced speech. In this way, `ZCR` can be used to distinguish between voiced and unvoiced speech segments in real-time.

3.7. Utilities

The *dsp* subpackage features a subpackage called *utilities* that contains some useful classes and supporting data structures for the library. The classes within the *utilities* subpackage are summarized in Table 13.

Table 13: Class summary of utilities package

Object Name	Responsibility	Requires Instantiation?
`Complex`	Class to represent complex number objects	Yes
`Matrix`	Class to represent Matrix objects	Yes
`RootSolver`	Find the roots of polynomials	No
`SolverNotConvergedException`	Custom exception for RootSolver	Yes

`Sort`	Array sorting and peak finders	No
`StopWatch`	Stop watch to time events	Yes
`Utilities`	Collection of utility methods	No

Complex

The `Complex` class represents complex numbers of the form $z = x + yi$, where z is the complex number, x is its real part, y is its imaginary part, and i is $\sqrt{-1}$. One instance of `Complex` represents a single complex number, and is constructed from 2 floating point numbers representing the real and imaginary components of the complex number. A collection of mathematical operations supported by the `Complex` class are summarized in Table 14.

Table 14: Summary of `Complex` class methods

Method Name	Description
`isReal()`	Is this complex number real ($y = 0$)?
`real()`	Returns the real part of this complex number (x)
`imag()`	Returns the imaginary part of this complex number (y)
`mag()`	Returns the magnitude of this complex number ($\sqrt{x^2 + y^2}$)
`power()`	Returns the power of this complex number ($x^2 + y^2$)
`arg()`	Returns the argument of this complex number ($tan(y/x)$)
`conj()`	Returns the conjugate of this complex number ($x - yi$)
`reciprocate()`	Returns the reciprocal of this complex number ($1/z$)
`exp()`	returns the complex exponential of this complex number (e^z)
`plus()`	Add a (complex) number to this complex number and return result
`minus()`	Subtract a (complex) number from this complex number and return result

`times()`	Multiply this complex number by another (complex) number and return result
`divide()`	Divide this complex number by another (complex) number and return result
`pow()`	Raise this complex number to a power and return result

Matrix

The `Matrix` class represents 2-dimensional matrices of floating-point numbers. An $N \times N$ matrix is stored in column major order in a single array of length N^2. For example, the matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ is stored as $[1, 3, 2, 4]$. This is done for both simplicity and performance, as accessing elements from a 1-dimensinal array is significantly faster than access from a 2-dimensional array.

A collection of mathematical operations are supported by the `Matrix` class, and are summarized in Table 15.

Table 15: Summary of `Matrix` class methods

Method Name	Description
`identity()`	Returns an identity matrix
`getRows()`	Returns the number of rows in this matrix
`getCols()`	Returns the number of columns in this matrix
`getData()`	Returns an array containing all the data in this matrix in column major order
`get()`	Returns the value at a specific location in the matrix
`set()`	Sets the value at a specific location in the matrix
`add()`	Adds a number to the value at a specific location in the matrix
`transpose()`	transpose this matrix, overwriting the original
`inverse()`	Returns the inverse of this matrix

`swap()`	swap 2 specified rows in this matrix
`times()`	Multiply this matrix by another matrix or vector
`solve()`	Solves the system of linear equations $A * x = B$, where A is this matrix, and B is another matrix or vector

RootSolver

The `RootSolver` class implements a root-finding algorithm to solve for the roots of polynomials using the Durand-Kerner Method. The Durand-Kerner method works as follows: if the array $[a, b, c, d]$ represents the polynomial $f(x) = x^4 + ax^3 + bx^2 + cx + d$, and the complex numbers $[P, Q, R, S]$ are the roots of this polynomial then it follows that:

$$f(x) = (x - P)(x - Q)(x - R)(x - S)$$

Which can be rearranged as:

$$P = x - \frac{f(x)}{(x - Q)(x - R)(x - S)}$$

And because f(P) = 0:

$$P = P - \frac{f(P)}{(P - Q)(P - R)(P - S)} = P - 0 = P$$

Which implies that P is a fixed point of the above equation and can therefore be found using a fixed-point iterator:

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{(p_{n-1} - q_{n-1})(p_{n-1} - r_{n-1})(p_{n-1} - s_{n-1})}$$

The above process is repeated for Q, R, and S to yield:

$$q_n = q_{n-1} - \frac{f(q_{n-1})}{(q_{n-1} - p_n)(q_{n-1} - r_{n-1})(q_{n-1} - s_{n-1})}$$

$$r_n = r_{n-1} - \frac{f(r_{n-1})}{(r_{n-1} - p_n)(r_{n-1} - r_n)(r_{n-1} - s_{n-1})}$$

$$s_n = s_{n-1} - \frac{f(s_{n-1})}{(s_{n-1} - p_n)(s_{n-1} - q_n)(s_{n-1} - r_n)}$$

With these 4 equations, the complex roots $[P, Q, R, S]$ can now be found by selecting initial guesses $[p_0, q_0, r_0, s_0]$, and iterating until the complex values $[p, q, r, s]$ stop changing between iterations, or if the change between successive iterations is below a specified threshold.

`RootSolver` provides the method `roots()` to return the complex roots of a polynomial represented by a floating-point array of polynomial coefficients. There are two flavors of the `roots()` method. The first accepts the array of polynomial coefficients, a stop threshold, and a max iteration limit. This method will return the roots of the polynomial if the maximum change between successive iterations drops below the stop threshold or throw a SolverNotConvergedException if the max number of iterations is reached before the stop threshold is satisfied. The second accepts the same array of polynomial coefficients, and a single boolean variable that indicates whether the method should proceed with maximum accuracy. If true, the method will find the roots as accurately as possible, at a greater computational cost. If false, the method will operate with a balance of accuracy and performance. This allows the programmer to effortlessly select the proper method for their use, without having to worry about optimizing the stop threshold and iteration limit.

SolverNotConvergedException

The `SolverNotConvergedException` class extends Java's built in `Exception` class which is a form of `Throwable` used to "throw" errors or conditions that an application may want to "catch" and attempt to correct, rather than immediately crashing.

SolverNotConvergedException provides a custom Exception object that RootSolver can throw if the root solver cannot converge to a solution, rather than getting stuck in an infinite loop attempting to solve a diverging iteration.

Sort

The Sort class provides a specialized sorting functionality that is highly useful in spectral analysis applications. Typical sorting methods, like those used by Java's own Arrays.sort(), take a list of unsorted numbers and rearrange them into ascending numerical order. The list gets sorted, but the original ordering of the numbers has been lost. But what if a program needs to sort a list of numbers while maintaining a link to their original positions? For example, to find the N highest spectral peak in a signal, its spectrum could be sorted. The sorted spectrum would now contain the amplitudes of the highest points in the spectrum, but it is impossible to find the corresponding frequency of each amplitude without knowing its original position in the spectrum. The Sort class addresses this use case with a customized merge sort algorithm. The method sort() takes an array of integers to be sorted, and an array of floating point numbers to sort with. The integer array is then sorted using an $O(n \log n)$ merge sort algorithm based on the contents of the floating-point array.

StopWatch

The StopWatch class implements a simple stop watch object for the library. One instantiated, a StopWatch object can be started and stopped with the methods start() and stop(). At any point, the stop watch timer can be queried using the getElapsedTime() or getElapsedTimeSecs() method to get the elapsed time in units of nanoseconds or seconds, respectively. If either of these methods are called while the stop watch is running, the elapsed time

since the last call to `start()` is returned, otherwise the time between the last calls to `start()` and `stop()` is returned.

This class is used heavily in the testing package to time library methods to verify real-time operation, or compare the execution speed of different algorithms/implementations. During development, the latter was performed extensively to ensure every class was designed with optimal performance on both PC and Android. See the *Testing* section for more details.

Utilities

The `Utilities` class is a collection of useful methods that are included with the library, but do not have a dedicated class. These methods will most likely be refactored into specific classes in the future. A summary of all methods provided by the Utilities class is shown in Table 16.

Table 16: Summary of `Utilities` class methods

Method Name	Description
`ansiBands()`	Returns center frequencies of ANSI Octave-bands and Fractional-Octave-Bands [ANSI Spec]
`ansiBandLimits()`	Returns band-limit frequencies of ANSI Octave-bands and Fractional-Octave-Bands [ANSI Spec]
`max()`	Returns the maximum value in an array
`maxLoc()`	Returns the location of the maximum value in an array
`arrayAvg()`	Returns the average of all the values in an array
`findHighestPeaks()`	Finds the N highest peaks in a data series
`findLowestPeaks()`	Finds the N lowest peaks in a data series
`findOrderedPeaks()`	Finds the first N peaks in a data series, in the order they appear
`isAndroid()`	Returns true if application is running on Android, false otherwise

3.8. COM

The *com* subpackage contains classes for data transmission over Bluetooth. The classes within the *com* subpackage are summarized below in Table 17 & 18.

Table 17: Class summary of com package

Object Name	Responsibility	Requires Instantiation?
`Bluetooth`	Abstract Bluetooth class	Abstract
`android/BluetoothAndroid`	Concrete Bluetooth class for Android	Yes
`BluetoothJava`	Concrete Bluetooth class for Java	Yes
`PacketPacker`	Packs data into any number of bits	No

Table 18: Interface summary of com package

Interface Name	Description
`BluetoothEventListener`	Defines listener functionality to create objects that listen for important Bluetooth events

Bluetooth

The abstract `Bluetooth` class defines the cross-platform functionality required to send data over a Bluetooth communication socket. The class's pubic methods are summarized in Table 19.

Table 19: Summary of `Bluetooth` class methods

Method Name	Description
`setListener()`	Attaches a BluetoothEventListener object to this `Bluetooth` object
`getState()`	Gets the current state of this `Bluetooth` object
`getPairedDevices()`	Gets the list of paired devices from the Bluetooth hardware
`connect()`	Connects to a specified paired device

`send()`	Sends bytes over Bluetooth to the connected device
`terminateConnection()`	Terminate the connection to the remote device, if connected

The classes `BluetoothAndroid` and `BluetoothJava` extend the abstract Bluetooth class and implement the required platform specific Bluetooth code for both Android and Java. `BluetoothAndroid` uses Android's own Bluetooth framework from the *android.bluetooth* package, while `BluetoothJava` uses the JSR-82 Java API's for Bluetooth implementation [26], from the Java library *BlueCove* [27]. Because both classes extend `Bluetooth`, they can be used interchangeably as `Bluetooth` objects, without needing to deal with the different platform code. To eliminate UI blocking, both `BluetoothAndroid` and `BluetoothJava` utilize separate threads for connecting devices and managing successful connections.

A Bluetooth connection can be made by first calling `getPairedDevices()` to get a list of previously paired devices. The index of the device to connect to is then passed to the `connect()` method to attempt to initiate a connection to the selected paired device. If a successful connection is established, the `send()` method can be used to send data to the remote device until the connection is terminated with the `terminateConnection()` method, or the connection is lost. Throughout this process, important events are communicated through the `BluetoothEventListener` object associated with the `Bluetooth` object.

BluetoothEventListener

The `BluetoothEventListener` interface is used in tandem with the `getState()` method from `Bluetooth` to allow the application to manage and react to Bluetooth communication events. The interface defines three methods that listen for important events from a Bluetooth instance.

The method bluetoothNotAvailable() is called if Bluetooth could not find any Bluetooth hardware, of if the hardware is currently unavailable. If Bluetooth is not available, this method will be called shortly after instantiation of the Bluetooth object. The application can then respond to this condition and disable any Bluetooth dependent functionally or instruct the user to turn on/allow Bluetooth on their device. Additionally, this condition causes a state change to either STATE_NONE, indicating that Bluetooth hardware was not found, or STATE_OK, indicating that Bluetooth hardware was found, but it is not currently enabled.

The listener method bluetoothStateChanged() is called anytime a Bluetooth object undergoes a change in state. A list of all possible states is shown in Table 20. If hardware was found and is enabled/ready to use, the object state will change to STATE_READY indicating that it is ready to begin connecting to a remote device. When attempting to establish a connection, the object will be in the state STATE_CONNECTING. Once a successful connection is obtained, the state is changed to STATE_CONNECTED. Upon disconnection from the remote device, the Bluetooth object will return to STATE_READY. If the device happens to be in discovery mode (used when pairing to a new device) the state is changed to STATE_DISCOVERY and the object must wait until the device is taken out of discovery mode (either by the user or by timeout).

Table 20: State summary of Bluetooth object

State Name	Description
STATE_NONE	Bluetooth is not available or not supported
STATE_OK	Bluetooth is supported, but not ready/available
STATE_READY	Bluetooth is available and ready for use
STATE_DISCOVERY	Bluetooth is currently in discovery mode
STATE_CONNECTING	Bluetooth is currently attempting to connect to a device

`STATE_CONNECTED`	Bluetooth is currently connected and ready to send/receive data

Finally, the method `bluetoothDataAvailable()` is called whenever data is *received* from a connected device. The received data (array of bytes) is passed as an argument to this method.

PacketPacker

`PacketPacker` is a class that packs any amount of data into a specified number of bits. It is intended to be used with `Bluetooth` to compress large data packets for wireless transmission. The static method `pack()` takes an array of normalized floating-point numbers to pack (the data packet), and an integer specifying the bit depth to use. Each normalized datum is converted to the specified number of bits using the following equation:

$$packed = value * (2^{bit\ depth} - 1)$$

The packed bits are then strung together to form an array of bytes. If the case of bit depths greater than 8, the bits are placed in little-endian order in the array. The returned byte array will have a length of:

$$ceiling\left(\frac{\#\ of\ data\ points * bit\ depth}{8}\right)$$

This byte array can then be sent over Bluetooth and the receiving device can perform the reverse of these operations to obtain the original data, with some quantization error from the bit reduction.

To pack unnormalized data, `PacketPacker` must be instantiated. It is initialized with the bit depth, and a minimum and maximum value for the incoming data. Data packets can then be passed to the non-static version of `pack()`, and each point is normalized with the following equation:

$$normalized = \frac{value - minimum}{maximum - minimum}$$

After normalization, each value is packed into a byte array exactly as described above.

4. Testing & Validation

Tactile Waves was built using a Test-Driven Development (TTD) methodology. In this approach, specific test cases are generated for each required feature or function, and then tested to ensure they function and fail as expected. The actual implementation is then written to pass these specific tests *after* the tests have been defined. The tests are run against this new code to verify its correctness, and the process is repeated, expanding or adding new functionality. The main idea behind this process is that it forces the programmer to focus on interaction rather than implementation. That is, the programmer must first define how the code will be used, rather than how it will be written. This is a favored development strategy as it focuses on only what is needed to produce the required functionality.

4.1. Unit Testing

Unit testing is the process of testing a single unit of code (referred to as the Unit-Under-Test or UUT) in a self-contained and isolated scope. In OOP, a unit is a single class or method whose behavior is being examined. The purpose of unit testing is to ensure that each unit performs exactly as it should, assuming that everything else in the system is working properly. To make this assumption valid, a unit test must have no dependence on external objects or state. If a unit requires data or state from another object or external source (such as a microphone) for proper execution, those data should be mocked within the testing scope. The steps of a unit test are as follows:

1. Setup the UUT and any mocked data/dependencies
2. Execute the test by passing the mock data and triggering the target behavior
3. Collect all output from the UUT and compare it to the correct output defined by the test

4. Cleanup the testing environment for the next test by freeing memory, resetting objects/data, etc.

In TDD, every object should have a set of unit tests *before* the actual object is implemented. Additional unit tests may be added during development, but the core functionality should be pre-defined by unit tests. Additionally, all unit tests should be executed whenever the code is changed to ensure the new changes do not break the defined functionality.

Before Tactile Waves was even conceived, research was performed using MatLab to investigate various speech processing techniques and obtain a better understanding of audio processing as a whole. Nearly every analysis procedure available in Tactile Waves was first prototyped in MatLab to flesh out the required steps, and gain insight into how these processes work. The development of Tactile Waves then became a process of redesigning these MatLab experiments for real-time usage and object-oriented design. As a result, much of the `toolbox` package was made to mimic certain Matlab functions. For example, the static `filter()` method from the `Filter` class was designed to mimic MatLab's own `filter()` function [28]. First, several MatLab scripts were created that (1) perform some useful filtering on a generated data set, and (2) output the filtered result. These generated data sets became the mocked data in step 1 of `filter()`'s unit test, and the output from the MatLab scripts was used as the correct output in step 3. This process was repeated until every requirement defined by the MatLab experiments had a corresponding unit test in Java. The library was then designed to satisfy these tests.

Unit testing in Tactile Waves was structed as follows. For each class in the library, a corresponding unit testing class was created. For example, the `FFT` class is tested by the `FFTUnitTest` class, `LPC` by `LPCUnitTest`, etc. Within each testing class, there is a unit test for each method in the class being tested.

FFT

The `FFT` class is tested using various methods to ensure that proper discrete Fourier transformation is performed. The first group of unit tests uses the test output from the `FFT.java` code provided in the textbook *Algorithms, 4th Edition* by Robert Sedgewick and Kevin Wayne [29]. Fourier transformation, inverse Fourier transformation, and convolution are all tested for correct output when given a 4-sample input signal. All supported data types are tested (`float`, `double`, and `Complex`). The next group of tests were written to verify known properties of the DFT. A unity amplitude signal is a signal in which every sample is equal to positive one. By definition, this signal has no frequency components and its spectrum should therefore contain all of the signals energy in the first bin (zero frequency, or DC) with all other frequency bins equal to zero. Additionally, the inverse DFT should revert a complex spectrum back to the original signal from which it was generated. Therefore, the inverse FFT of the FFT of a signal should result in the original signal (plus some numerical noise). Finally, a known property of the complex DFT is its left/right symmetry, so the complex FFT of any signal should always be symmetric about the middle of the spectrum. Each FFT subroutine in the class is tested to meet these conditions.

Window

The `Window` class is tested to ensure that each window function produced exactly the same output as MatLab's window functions. In MatLab, a 100-point window of each type was generated and output to the console. These outputs were written into the `WindowUnitTest` class and used to check the correctness of each window function in the library.

DCT

The Type-I and Type-II DCT algorithms implemented in the DCT class are tested against MatLab's dct() function. Each DCT is tested with both 8-point and 9-point input signals (to test both even and odd length), and the 9-point input is tested with and without orthogonal normalization for a total of 6 distinct unit tests.

Filter

The `filter()` method from the `Filter` class is tested against MatLab's filter() function using a 64-point impulse signal for various filter coefficients. Additionally, the coefficients of the 2-pole Butterworth filter generated with the allPole() filter design method are checked against the coefficients output by MatLab's butter() function, which designs a digital Butterworth filter. Because a Chebyshev filter with 0% passband ripple is a Butterworth filter, chebyshev() was also compared against these coefficients for 2-poles and 0% ripple. Unfortunately, MatLab does not have functions for creating Bessel or Critically Damped digital filters, and its cheby() function is incapable of designing a filter with no passband ripple. Therefore, these filters could not be directly tested against MatLab. Instead, the filter frequency responses were generated in MatLab and compared against known responses for each filter type. Once the frequency responses were verified, the coefficients were saved into the FilterUnitTest class and are used for testing the Bessel, Critically Damped, and Chebyshev filters. A similar procedure was used to check the coefficients of the biquad filters, except Ableton Live was used to verify the responses. A *.wav* file was created containing an impulse signal, and this file was imported into a new Live set. Live's native EQ Eight and Auto Filter devices were used to filter the impulse signal with various bi-quadratic filters, and the filtered audio from each filter was recorded into a blank audio track. These recorded impulse responses are included in the testing class for testing the biquad() method.

Cepstrum

The various cepstral transforms implemented in the Cepstrum class are tested against corresponding functions in MatLab. The output of the real cepstrum is compared against the MatLab function rceps(), while the complex cepstrum and inverse complex cepstrum are compared against the MatLab functions cceps() and iceps(), respectively. Each is tested with a 128-sample input signal containing a 45 Hz sine wave with an echo, sampled at 100 Hz. MatLab does not have a function for computing the power cepstrum, so the real cepstrum is used to verify the power cepstrum. The power cepstrum is equal to the square of two times the real cepstrum. This definition is used to compute the power cepstrum from the real cepstrum, the result of which is compared against the output of the power cepstrum.

MFCC

MatLab does not provide any functions for directly computing the MFCC's of an audio signal. There are a variety of MFCC subroutines available online for various programming environments, but each seems to use its own variant of the algorithm. For example, the Hidden Markov Model Toolkit (HTK) uses a slightly different Mel scale, and a Type-III DCT instead of a Type-II, while the CMU Sphinx package uses the standard Mel scale with a Type-II DCT, but normalizes the output by its length. This makes it difficult to develop an effective unit test for the `MFCC` class, as the output cannot be directly compared to a known correct output. Instead, certain numerical properties defined by the MFCC are tested. Because the Mel filter banks are designed to be constant energy, a signal with a flat spectrum should produce a flat MFC. An impulse signal is used as the test input as it contains all frequencies and therefore has a flat spectrum. The output is then compared to the output of the DCT of a flat spectrum normalized to the same total energy as the impulse. This test is repeated for both a low and high energy

impulse. Additionally, the filter bank frequencies are tested separately to ensure the correct Mel-scaled filter frequencies are used by the algorithm.

LPC

The `LPC` class is compared directly to MatLab's own `lpc()` function, as it was designed to emulate this implementation exactly. A 4-sample input signal is used (the same signal used in `FFTUnitTest`), and the output of a 4th order predictor from `lpc()` is compared to the output produced by MatLab's `lpc()`. The formant/bandwidth pairs produced by the estimateFormants() method are also compared to those produced by MatLab, using the procedure outlined in [].

YIN

The sole responsibility of the `YIN` class is to report the pitch of an audio signal, or -1 if no pitch exists. This functionality is tested with various signals and sample rates to ensure reliable operation over a wide range of input signals. For all tests, both `estimatePitch()` and `estimatePitchFast()` are tested in the same way. The first unit test in the `YINUnitTest` class uses a simple sine wave as input. The methods are tested with both a 0.0 dBFS 120 Hz sine wave sampled at 16000 Hz and a 0.0 dBFS 1000 Hz sine wave sampled at 44100 Hz. The output of each method should be equal to the frequency of the input sine wave. This test is repeated with the same sine waves, except the first and second harmonics are added to each with 60% and 40% of the fundamental's amplitude, respectively. This test is repeated, but random noise is added instead of the additional harmonics. The same 120 and 1000 Hz sine waves are generated, and then mixed with randomly generated noise at a ratio of 5:1 signal to noise. In all cases, the methods should report the fundamental frequency of either 120 or 1000 +/- 0.1 Hz.

ZCR

The `ZCR` class is tested by ensuring that it reports the correct number of zero crossings for a known signal. The correct number of zero crossings can be calculated for a pure sine wave by multiplying the frequency of the wave by two and dividing by the sampling rate. Using this relation, the output of the `getZCR()` method is checked using the same 120 and 1000 Hz clean sine waves from the `YinUnitTest` class.

4.2. Performance Testing

At the time of this writing, Tactile Waves is the only published software library for real-time sensory substitution. Each algorithm and subroutine in the library must be tested to ensure it can be used in real-time on both personal computers and Android phones. As discussed in Chapter 2, a continuous audio signal must be sampled at discrete points in time for digital representation. These samples are then processed in groups called frames, or buffers of audio. Much like how a video rapidly displays single stationary images to create the illusion of continuous motion, audio can be generated by rapidly processing frames of audio samples. Processed samples are sent to a digital-to-analog converter (DAC) which converts the discrete samples to a pressure wave. If a new audio frame is not sent to the DAC by the time it finishes playing the last, there will be a dropout in the audio stream. In the case of sensory substitution, the audio may not actually be played aloud, but the principal remains. New audio data must be available to be sent to the sensory interface at a rate equal to or faster than it takes to represent that audio in the time domain, or there will be data loss. Based on this criterion, "real-time operation/processing" is defined in this thesis as the ability to process an audio task/analysis/effect on a frame in less than the time it takes to play that frame as actual audio.

As stated in Chapter 2, the length in seconds of each of an audio frame, t, can be calculated with the following equation:

$$t = \frac{N}{f_s}$$

Where N is the number of samples in the frame, and f_s is the sampling rate. This relation was used to develop the scoring system used herein. Scoring is based on a ratio of computation time and audio frame length, defined by the following equation:

$$Score = \frac{N/f_s}{time\ to\ compute}$$

Which states that the score is equal to the number of times an operation can be performed on one audio frame without exceeding the playback latency of that frame. For example, if a processing operation takes 20 ms to compute for a 20 ms frame of audio, it would receive score of 1. Similarly, another operation that takes 10 ms to compute 20 ms of audio would receive a score of 2. A score less than 1 would indicate that the processing requires more time than there is audio, indicating a failure to satisfy real-time operation. A score of 1 may not be suitable for real-time operation, because of additional processing overhead needed to prepare a frame of audio, and convert it back to an analog signal after processing. Additionally, it is rarely useful to perform only one processing operation on a frame of audio. A real-world application will require a sequence of audio processing steps to achieve its goal. For example, a sensory substitution software may need to window and filter a frame with `Window` and `Filter` before estimating its pitch using `YIN`, and then finally extracting its formant frequencies with the `LPC` class. For these reasons, a score of 1 for a single computational process is not satisfactory. A conservative lower limit of 4 was chosen for real-time validation testing of Tactile Waves, with 10 or above being an ideal score. That is to say, real-time operation is validated if the processing can be completed in less than or equal to one quarter of the frame time, with one tenth (or less) of the frame time being ideal, which would allow 4-10 processes to be performed without imparting dropouts or artifacts due to buffer underruns.

As discussed in Chapter 3, computational requirements change based on the length of an audio frame. Each test is performed on 4 different audio frame lengths, to ensure real-time validation over a range of commonly used buffer sizes. The sizes used for each test are 1024, 2048, 4096, and 8192. Because Tactile Waves is intended for use on both mobile phones and desktop/laptop computers, all performance testing has been performed on a Windows laptop, and an Android smartphone.

The Android smartphone used was a Samsung Galaxy S8. An Android application called CPU-Z was used to view the properties of the phones SOC. The Samsung uses a Qualcomm Snapdragon 835 CPU which features eight cores in a big.LITTLE configuration. This means that the chip is setup with two sets of four cores with one set being comprised of low-power cores with a maximum speed of 1.8 GHz, and the other of high-performance cores with a higher maximum clock of 2.36 GHz. The system can rapidly switch between each set depending on the workload to provide a balance between performance and battery life. Each test was ran independently while hooked up to a Windows laptop via the Android Debug Bridge (ADB) to view testing results.

The Windows machine used was an ASUS UX430UNR equipped with an Intel Core i7-8550U CPU and 16 GB of 2133 MHz DDR3L RAM. The 8550U is a quad-core, eight-thread, ultra-low voltage (ULV) CPU with a base clock of 1.8 GHz and a maximum boost clock of 4 GHz. ULV processors are designed specifically for small, thin and lite devices such as ultrabooks, tablets, and 2-in-1's. As a result, the performance of these types of processors is nearly completely dependent on the available power and thermal headroom, and they are optimized for short bursts of power for everyday productivity tasks rather than long sustained workloads. The performance of these chips will therefore degrade as the temperature of the die increases. When the CPU is cold, it will easily run all four of its cores at its maximum rated speed. As a sustained load continues, the power draw and temperature of the CPU package will increase. To prevent the CPU from overheating or drawing too much current, the maximum

speed of each core is reduced. This is highly detrimental for performance testing, as the maximum processor speed might be reduced the longer a test is run, skewing the results.

To mitigate these issues, a standard test configuration was created and used for all Windows testing. ThrottleStop 850 was used to reduce the maximum processor speed to 3.6 GHz (two cores max) and 3.2 GHz (all four cores), as well as disable core-parking. Limiting the maximum processor speed ensures that the processors thermal limit is never reached, eliminating thermal throttling. Disabling core parking ensures that Window's will not shut down cores, which can regularly cause massive and sudden drops in performance during a varying workload. Intel's Extreme Tuning Utility (XTU) was used to increase the maximum boost power to 36 watts to prevent power limit throttling. XTU was also used to undervolt the CPU by -85 mV to decrease the operating temperature, thereby increasing the available thermal headroom. These settings were found after several weeks of testing to provide optimal and consistent performance in various sustained workloads such as audio/video processing and synthesis.

A standard benchmarking routine was developed to provide consistent and accurate micro-benchmarking, which is the process of estimating the real-world execution time of a piece of code. Micro-benchmarking in a compiled language such as C++ is more straightforward and reliable than in Java because a statically compiled language cannot be changed or modified during runtime. Java is not a statically compiled language, and the run-time compilation effects must be considered for accurate benchmarking. The first time a code path is executed by the JVM, it is executed in "interpreted mode", which allows the Java code to be executed directly without compilation, greatly improving the startup time of applications. Only after a certain amount of execution will the JVM obtain enough profiling data to compile the code. This can severely skew the results of a microbenchmark that includes these operations while timing execution as the test will measure the execution of both the optimized machine code and interpreted byte code, as well as the time the compiler spent analyzing and compiling the code path. To eliminate this effect on the timing phase, a so called "warm-up" phase must be performed

before any code is timed. The warm-up phase executes the test code a certain number of times to allow the JVM to compile and replace the interpreted code. With modern VM's, it is difficult to determine how much execution is needed for proper warm up. All benchmark tests were ran with the `-XX:+PrintCompilation` option, which tells the compiler to print a message every time it runs. The test will print messages before and after each warming and timing phase. If the compiler is run during any stage, it will print a message between these statements. The tests were run several times, and if any compilation occurred during a timing phase, the benchmark was modified to attempt to move this compilation to the warming phase. It was found that the best way to warm-up the VM for benchmarking was to run the entire benchmark, including timing phases, multiple times. By the second pass through the benchmark, only compiled code is being used (verified by the fact that the compiler does not print any messages during timing phases of the second pass).

Modern compilers are adept at making optimization decisions to improve performance based on assumptions and observation made about the code being compiled. One of the most common optimizations is code elimination. If a section of code does nothing to alter the program's correctness, a compiler can detect that without this code, the program will still function properly. This code is flagged as "dead code" and removed (or replaced) by the compiler. This is quite useful for deployed code, but presents a problem for benchmarking. Benchmarking code often is dead code. That is, a benchmark will run some code while timing executing, then simply throw away the result of the code being timed, as only the timing results are needed. A compiler may spot this condition, and declare the code being tested as dead because the result is never used, resulting in an empty benchmark. To trick the compiler, dummy methods are used. After a benchmark is completed, the testing data is passed to a dummy method that performs some calculations with the data, and stores it. This prevents the compiler from removing the code being tested from the benchmark.

To eliminate noise in timing measurements, all applications on the testing device were closed. Additionally, “Airplane Mode” was used during all testing to prevent the operating system or any application/service from doing any background work such as downloading updates. For each test, the timing phase is repeated 10 times, and the best results taken. Each test was repeated 3 times to ensure consistent results were obtained. Greatly varying results between testing iterations would indicate some kind of error or unexpected behavior, so the results of each timing phase were inspected. Testing results from the Windows laptop are shown in Table 21, and results from the Android device is shown in Table 22.

Table 21: Real-time validation benchmark results - Windows PC

FFT					
Method	Data Type	1024	2048	4096	8192
fft()	float	2161.041	2067.158	1977.769	1834.573
	double	2048.158	1990.327	1877.253	1519.227
complexFFT()	float	1519.384	1440.501	1307.739	1098.522
	double	1534.664	1411.419	1141.985	673.675
	Complex	275.0202	243.2565	209.3636	170.8312
DCT					
Method	Data Type	1024	2048	4096	8192
dct()	double	219.2690	218.6997	216.9899	193.4675
idct()	double	253.0035	249.4492	233.7784	201.4546
Cepstrum					
Method	Data Type	1024	2048	4096	8192
rCepstrum()	float	158.4087	157.4957	156.8208	154.2499
pCepstrum()	float	162.6465	161.2002	160.5518	157.7844
cCepstrum()	float	139.3341	138.3441	137.6939	135.9715
MFCC					
Method	Data Type	1024	2048	4096	8192
getMFCCs()	float	760.8566	760.8567	1035.832	718.8310
YIN					
Method	Data Type	1024	2048	4096	8192
estimatePitch()	float	77.89865	38.58343	19.17504	9.56549
estimatePitchFast()	float	375.7629	351.7741	215.9140	156.7994
LPC					
Method	Data Type	1024	2048	4096	8192
estimateFormants()	float	300.4420	191.7079	146.1327	124.6823

Table 22: Real-time validation benchmark results - Samsung Galaxy S8

FFT					
Method	Data Type	1024	2048	4096	8192
fft()	float	460.7644	477.8499	441.9865	453.1677
	double	446.4366	353.7387	367.7968	373.2101
complexFFT()	float	402.2071	378.2906	353.5991	330.1951
	double	413.3348	392.4741	365.8644	257.6025
	Complex	18.24707	16.84501	14.49016	12.16456
DCT					
Method	Data Type	1024	2048	4096	8192
dct()	double	223.4289	184.6639	200.8741	174.1264
idct()	double	207.6733	198.1290	182.7861	145.1691
Cepstrum					
Method	Data Type	1024	2048	4096	8192
rCepstrum()	float	210.2004	205.7225	196.4484	184.0477
pCepstrum()	float	188.9545	201.2172	192.9628	182.8314
cCepstrum()	float	196.8173	190.2010	169.6471	172.4575
MFCC					
Method	Data Type	1024	2048	4096	8192
getMFCCs()	float	171.9872	186.2942	188.8579	202.4266
YIN					
Method	Data Type	1024	2048	4096	8192
estimatePitch()	float	15.92646	7.981991	3.994365	1.997635
estimatePitchFast()	float	82.31873	81.91017	77.58937	65.88227
LPC					
Method	Data Type	1024	2048	4096	8192
estimateFormants()	float	63.88614	71.63753	57.87448	53.78447

These results show that all but one of the audio analysis and processing methods included in the library can be used in real-time on modern mobile hardware. One method, the $O(n^2)$ Yin pitch estimator, was not able to satisfy the real-time criteria at higher buffer sizes. A faster version was developed using the FFT to reduce its complexity to $O(n \log n)$, and both methods are available for use in the library. Additionally, many of the methods achieved scores that greatly exceeded the ideal score of 10. This indicated that the library should still function in real-time when used in an environment

where less processing power is available, such as an older smartphone or laptop or wearable device. However, is it impossible to estimate the minimum amount of processing power required without performing additional performance testing on a wider range of mobile devices.

5. Conclusion

The goal of this thesis was to design and develop a speech analysis library and audio processing framework for a sound-to-touch sensory substitution API. A toolbox of static methods was created to mimic a subset of common speech processing functions available in scientific computing environments such as MATLAB. A flexible, portable audio engine was then designed to allow users of the API to manage an audio stream from source to sensory hardware. The engine is highly extensible, allowing users to build custom objects that can be freely inserted into an audio processing chain.

Because any sensory substitution or augmentation device requires real-time operation, the core toolbox methods were micro benchmarked to ensure the required computation can be completed in real-time. One method, the $O(n^2)$ Yin pitch estimator, was not able to satisfy the real-time criteria at higher buffer sizes. A faster version was developed using the FFT to reduce its complexity to $O(n \log n)$, and both methods are available for use in the library.

The toolbox and audio engine were packaged into an Android Java Library called Tactile Waves. The library is licensed under the GNU General Public License Version 3.0, and its source code is available on GitHub [22]. Downloads are available through a public repository on jFrog Bintray [30], and remote linking for build dependencies is available through jCenter. Finally, a website was created with an installation and setup/usage guide, as well as links to GitHub, Bintray, and the API documentation. It is available here: https://funkatronics.github.io/TactileWaves/.

A demonstrative Android application has been created that shows how Tactile Waves can be used to build sensory substitution systems. The application sets up an instance of the audio processing engine with the phones microphone used as an input stream. Two processor objects are used in the processing chain: a pitch detector (YIN) and formant estimator (LPC). These extracted features are then

used to update a graph on the device screen and are sent over Bluetooth. Code for a receiving hardware is not included. This application is meant to provide a demonstration on the usage of Tactile Waves audio engine, processing toolbox, and Bluetooth functionalities. The code for this app is included with the main source code on GitHub [22].

5.1. Future Work

Tactile Waves was developed alongside an ongoing research project aimed at developing a sound-to-touch hearing aid/augmentation system that uses the surface of the tongue as a sensory input [31]. Currently, a primary goal of this research is to determine how to optimally encode speech signals for tactile representation on the tongue. Preliminary human testing has been performed to investigate and compare several encoding methods proposed by JJ Moritz in his thesis [32]. Previously, there was no software capable of extracting the necessary data from speech signals in real-time. As a result, these tests have used a small set of prerecorded audio files from one speaker so that audio features could be extracted and transcribed manually with the help of the popular speech analysis software, Praat [33]. With the completion of Tactile Waves, these experiments can be expanded to use real-time audio processing. This will allow the research team to continue human studies with a broader vocabulary and more diverse range of speakers. The use of smartphones will reduce the size and cost of the necessary hardware, allowing for more subjects to be included. Additionally, the portability of these devices could allow this research to expand into larger scale studies that include real-world listening environments.

The development of the library will certainly not end with the completion of this thesis, and it is expected that the aforementioned research will motivate changes and additions to Tactile Waves. Existing features may be modified or expanded, and new functionality could be added to meet the evolving needs of this research. The current version (Version 1.0.1) of the library will continue to be

updated and improved while still conforming to its primary goal of providing an open sourced, and openly available sound-to-touch sensory substitution API.

Version 1 of Tactile Waves has been designed specifically for speech processing and is therefore not equipped for music applications. There is a rapidly growing market for touch based audio monitoring and feedback devices for audio consumption and creation. Devices like the *Woojer* [34] provide vibrotactile feedback to video gamers, movie watchers, and audiophiles to enhance perception and immersion of low frequency content such as explosions. These devices aim to provide an "IMAX experience at home", without the need for large speaker systems, and are typically referred to as *bass augmentation devices*. Using this principal, devices like the *SUBPAC* and *Basslet* allow music producers and live performers to monitor the low frequency content of their mixes without the expense and hearing damage associated with typical low-frequency acoustic monitoring systems such as floor wedges. The SUBPAC is also is favored by deaf musicians and concert goers as they are no longer left out of these activities due to their hearing impairment. These users never want to feel as though they are deaf persons who like music, but as music lovers who also happen to be deaf. SUBPAC's close interaction with the deaf community and countless success stories with deaf musicians and music lovers shows how effectively music can be experienced through vibrotactile devices [35]. In the realm of music, video games, and movies, bass augmentation devices have exploded in popularity. Both music production and deaf communities are highly accustomed to DIY, ad-hoc approaches to create new tools to achieve desired functionality. As a result, these circles could benefit greatly from a music-to-touch software API.

Version 2 of the library will feature an overhauled design to give creative entrepreneurs a flexible framework to design touch based audio monitoring devices for live performers, producers, and deaf musicians and listeners. The toolbox package will be expanded to include a real-time beat detection engine. Max externals will be created for every object in the library, allowing Tactile Waves to

be used in the Max/MSP visual programming environment. Finally, a real-time audio warping engine will be developed. With these tools, Tactile Waves could be used to build custom bass augmentation devices, tactile hearing aids, and even real-time visual generation programs for musical performances.

5.2. Final Thoughts

It is my intention to continue developing and supporting both versions of the library indefinitely. Being released under an open source license, the library is open for contribution. Any contributions that are be made to the library must first be checked and approved by myself, or any other individuals that I decide to entrust with this responsibility. I both hope and intend for Tactile Waves to motivate and enable further research in the fields of sensory substitution and haptic feedback.

References

[1] World Wide Hearing. (2017). *World Wide Hearing*. [online] Available at: http://www.wwhearing.org/ [Accessed 19 Dec. 2017].

[2] A. McCormack and H. Fortnum, "Why do people fitted with hearing aids not wear them?," *International Journal of Audiology*, vol. 52, no. 5, pp. 360–368, May 2013.

[3] S. W. Teoh, D. B. Pisoni, and R. T. Miyamoto, "Cochlear implantation in adults with prelingual deafness. Part I. Clinical results.," The Laryngoscope, vol. 114, pp. 1536–40, Sept. 2004.

[4] H. Fryauf-Bertschy, R. S. Tyler, D. M. R. Kelsay, B. J. Gantz, and G. G. Woodworth, "Cochlear implant use by prelingually deafened children: the influences of age at implant and length of device use," Journal of Speech, Language, and Hearing Research, vol. 40, pp. 183–199, Feb. 1997.

[5] B. W. White, F. a. Saunders, L. Scadden, P. Bach-y-Rita, and C. C. Collins, "Seeing with the skin," Percept. Psychophys., vol. 7, no. 1, pp. 23–27, 1970.

[6] D. W. Sparks, P. K. Kuhl, A. E. Edmonds, and G. P. Gray, "Investigating the MESA (Multipoint Electrotactile Speech Aid): The transmission of segmental features of speech," The Journal of the Acoustical Society of America, vol. 63, no. 1, pp. 246–257, Jan. 1978.

[7] P. L. Brooks and B. J. Frost, "Evaluation of a tactile vocoder for word recognition," The Journal of the Acoustical Society of America, vol. 74, no. 1, pp. 34–39, Jul. 1983.

[8] P. L. Brooks, B. J. Frost, J. L. Mason, and D. M. Gibson, "Continuing evaluation of the Queen's University tactile vocoder. I: Identification of open set words." J Rehabil Res Dev, vol. 23, no. 1, pp. 119–128, Jan. 1986.

[9] P. L. Brooks, B. J. Frost, J. L. Mason, and D. M. Gibson, "Continuing evaluation of the Queen's University tactile vocoder II: Identification of open set sentences and tracking narrative." J Rehabil Res Dev, vol. 23, no. 1, pp. 129–138, Jan. 1986.

[10] P. L. Brooks, B. J. Frost, J. L. Mason, and D. M. Gibson, "Word and Feature Identification by Profoundly Deaf Teenagers Using the Queen's University Tactile Vocoder," Journal of Speech Language and Hearing Research, vol. 30, no. 1, p. 137, Mar. 1987.

[11] K. A. Kaczmarek, J. G. Webster, P. Bach-y-Rita, and W. J. Tompkins, "Electrotactile and vibrotactile displays for sensory substitution systems," IEEE Transactions on Biomedical Engineering, vol. 38, no. 1, pp. 1–16, Jan. 1991.

[12] T. Ifukube, C. Wada, T. Izumi, and M. Takahashi, "A new display method of vibratory patterns for a fingertip tactile vocoder," in 1992 14th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 1992, vol. 4, pp. 1318–1319.

[13] K. A. Kaczmarek, M. E. Tyler, and P. Bach-y-Rita, "Electrotactile haptic display on the fingertips: Preliminary results," in *Engineering in Medicine and Biology Society, 1994. Engineering Advances: New Opportunities for Biomedical Engineers. Proceedings of the 16th Annual International Conference of the IEEE*, 1994, vol. 2, pp. 940–941.

[14] I. R. Summers and D. A. Gratton, "Choice of speech features for tactile presentation to the profoundly deaf," IEEE Transactions on Rehabilitation Engineering, vol. 3, no. 1, 1995.

[15] P. Bach-y-Rita, et al. "Form perception with a 49-point electrotactile stimulus array on the tongue: a technical note," J. Rehabilitation Research and Develop., vol. 35, pp. 427-430, Oct. 1998.

[16] P. Bach-y-Rita and S. W. Kercel, "Sensory substitution and the human–machine interface," Trends in Cognitive Sciences, vol. 7, no. 12, pp. 541–546, Dec. 2003.

[17] M. Auvray, S. Hanneton, and J. K. O'Regan, "Learning to Perceive with a Visuo — Auditory Substitution System: Localisation and Object Recognition with 'The Voice,'" *Perception*, vol. 36, no. 3, pp. 416–430, Mar. 2007.

[18] K. A. Kaczmarek, "The tongue display unit (TDU) for electrotactile spatiotemporal pattern presentation," Scientia Iranica, vol. 18, no. 6, pp. 1476–1485, Dec. 2011.

[19] N. Lago and F. Kon, "The Quest for Low Latency," in *ICMC*, 2004.

[20] R. Steinmetz, "Human Perception of Audio-Visual Skew," Architecture and Protocols for High-Speed Networks. Springer, Boston, MA, 1994, pp. 235-252

[21] A. de Cheveigné and H. Kawahara, "YIN, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, Apr. 2002.

[22] Funkatronics, "Funkatronics/TactileWaves," GitHub. [Online]. Available: https://github.com/Funkatronics/TactileWaves. [Accessed: 15-May-2018].

[23] J.W. Cooley, J.W. Tukey, "An algorithm for the machine calculation of complex Fourier Series," Mathematics Computation, Vol. 19, 1965, pp 297-301.

[24] L. Bluestein, "A linear filtering approach to the computation of discrete Fourier transform", IEEE Transactions on Audio and Electroacoustics, vol. 18, no. 4, pp. 451-455, 1970.

[25] G. Heinzel, A. Rudiger, and R. Schilling, "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows.," p. 84.

[26] C. Ortiz, "Using the Java APIs for Bluetooth Wireless Technology", Oracle.com, 2018. [Online]. Available: http://www.oracle.com/technetwork/articles/javame/index-156193.html. [Accessed: 17- May- 2018].

[27] V. Skarzhevskyy, "BlueCove - BlueCove JSR-82 project", Bluecove.org, 2018. [Online]. Available: http://bluecove.org/. [Accessed: 17- May- 2018].

[28] "1-D digital filter - MATLAB filter", Mathworks.com, 2018. [Online]. Available: https://www.mathworks.com/help/matlab/ref/filter.html. [Accessed: 07- Mar- 2018].

[29] R. Sedgewick, Algorithms in Java. Boston: Addison-Wesley, 2010.

[30] Tactile Waves, Bintray.com, 2018. [Online]. Available: https://bintray.com/funkatronics/tactilewaves/tactilewaves. [Accessed: 25- Apr- 2018].

[31] J. Moritz Jr., P. Turk, J. Williams and L. Stone-Roy, "Perceived Intensity and Discrimination Ability for Lingual Electrotactile Stimulation Depends on Location and Orientation of Electrodes", Frontiers in Human Neuroscience, vol. 11, 2017.

[32] J. A. Moritz Jr, "Evaluation of electrical tongue stimulation for communication of audio information to the brain," Colorado State University, 2016.

[33] "Praat: doing Phonetics by Computer." [Online]. Available: http://www.fon.hum.uva.nl/praat/. [Accessed: 19-Nov-2015].

[34] Woojer, 2016. [Online]. Available: https://www.woojer.com/. [Accessed: 12- Mar- 2018].

[35] "Deaf Community Archives - FEEL SUBPAC", FEEL SUBPAC, 2018. [Online]. Available: http://feel.subpac.com/category/deaf-community/. [Accessed: 10- May- 2018].