

3SA

(Spread Spectrum Simulation Application)

By Terrence Irving
Advisor: Professor Bruce McNair
July 22, 2005

National Science Foundation Research Experience for Undergraduates in Software-Defined Radio
Stevens Institute of Technology
Hoboken, NJ USA
Summer Research Duration: May 23 2005 – July 29, 2005

Table of Contents

1. Research Background	Page 1
2. Spread Spectrum Communications Overview	Page 1
3. Project Introduction	Page 2
4. 3SA Progress and Development	Page 2
▪ Component Descriptions	Page 2
▪ Issues Encountered	Page 4
▪ Component and Algorithm Changes	Page 5
▪ 3SA as Software	Page 5
5. Conclusions	Page 6
▪ The Research Process	Page 6
▪ Future Improvements	Page 6
▪ 3SA as a Learning Tool	Page 6
6. References	Page 7
▪ Works Cited	Page 7
▪ Other Resources Consulted	Page 8
7. Other Resources Consulted	Page 8
8. Appendices	Page 10

■ Appendix A: System Block Diagram	Page 10
■ Appendix B: Weekly Reports	Page 11
■ Section 1 – Week 1	Page 11
■ Section 2 – Week 2	Page 12
■ Section 3 – Week 3	Page 16
■ Section 4 – Week 4	Page 20
■ Section 5 – Week 5	Page 23
■ Section 6 – Week 6	Page 26
■ Section 7 – Week 7	Page 30
■ Section 8 – Week 8	Page 33
■ Appendix C: Key MATLAB Plots	Page 37
■ Appendix D: 3SA Screenshots	Page 50

Research Background

The National Science Foundation's 2005 Research Experience for Undergraduates in Software-Designed Radio took place at the Stevens Institute of Technology in Hoboken, NJ between the dates of May 23 and July 29. It was the NSF's goal to bring together college undergraduates of similar educational backgrounds (engineering and computer science) to work cooperatively with Stevens faculty and graduate students, exposing them to the wonders of conducting research.

NSF REU students were teamed with undergraduate researchers sponsored by the Department of Defense and the Stevens Scholar Program. The 19 students were broken into smaller groups and placed under the mentorship of individual Stevens faculty advisors. Those students who worked with Professor Bruce McNair (a Stevens specialist in software-defined radio) undertook projects covering topics such as cryptography, smart card technology, numerically controlled oscillators, and direct sequence spread spectrum (DSSS) communications. The latter is the focus of this report.

The development of the 3SA MATLAB application started as a mere interest in the field of wireless communications. As the knowledgebase grew, however, so did 3SA's potential for being a useful tool in the field of software-define radio education. This growth—as the student team members have learned—is the essence of research.

Spread Spectrum Communications Overview

Spread spectrum came about in the 1940s when actress Hedy Lamarr and pianist George Antheil brainstormed the idea in order to describe a new way for the military to securely control torpedoes (Maxim). The pair received a patent for their work, but the United States Army did not immediately put spread spectrum to use. In 1980, the Federal Communications Commission expressed an interest in implementing spread spectrum systems outside of homeland defense (Bible). It was at this time that spread spectrum was introduced to the realm of amateur radio, and the general public was able to see its potential as a radio frequency communications method.

Today, spread spectrum is the basis for many popular and emerging wireless communications standards, including portable telephones, global positioning systems, Bluetooth, and some cellular telephone network schemes. The two most widely used versions of spread spectrum are frequency hopping spread spectrum (FHSS) and direct sequence spread spectrum (DSSS). Though each has its own place in modern wireless radio frequency applications, they share some common characteristics:

- spread spectrum systems are bandwidth inefficient—the bandwidth of the transmitted signal is always greater than the bandwidth of the original input signal, which causes the transmission to appear noise-like to radio spectrum outsiders (Maxim)
- spread spectrum transmitters use similar transmission power levels as narrowband transmitters, but because SS signals are very wide (due to pre-transmission signal

spreading operations), they travel at a lower spectral power density than narrowband signals, accounting for the lack of interference between spread spectrum systems and narrowband systems

- spread spectrum communications depend on some kind of pattern to make successful transmissions—DSSS implementations typically use a code while FHSS systems operate with a predefined set of frequencies (Spread Spectrum Scene)

As mentioned earlier, the focus of this research was DSSS. Spread spectrum as a whole, however, was studied for some time before the direct sequence method was chosen as a research path.

Project Introduction

The concept of 3SA (Spread Spectrum Simulation Application) was born during Week 2's team meeting. Professor McNair suggested that the engineering software environment MATLAB be considered as a primary aid in developing the DSSS project.

Even from the onset of its development, 3SA was intended to be more than just the culmination of the research conducted during the REU program. The developer, as well as Professor McNair, hoped that 3SA would "be useful to others in the spirit of dissemination of knowledge that NSF strives for" (McNair). In other words, simply completing the research would not be enough—contributing to the fields of software-defined radio and wireless communications in a worthwhile manner was also an objective.

3SA Progress and Development

Component Descriptions

3SA's block diagram can be found in Appendix A. The following is a brief description of each of the 10 component's purpose in the DSSS simulation system:

- **Voice Acquisition (acquire.m):** The Voice Acquisition component acquires the system's input signal, which takes the form of recorded audio data (via a PC microphone) or a Microsoft Wave Sound (file extension ".wav"). The input signal's left channel is arbitrarily kept while the right is discarded. This is done with respect to telecommunications—the average cellular telephone has only one earpiece, for example. Currently, one second of data sampled at 8 kHz is the standard.
- **Spread (spread.m):** This is the only component represented by a true MATLAB function. Given a desired Walsh Code length n (also known as a "spreading factor"), the Spread component generates an n -by- n Hadamard Matrix and randomly selects a row of that matrix, which serves as the Code. The input signal and Code are then

extended (by n and the length of the input, respectively), and their product is the spread signal.

- **Analog to Digital Conversion (adc.m):** The Analog to Digital Conversion process is the third component of the transmission chain. This component first compresses the spread input data using a Mu-law compander (compressor-expander) and then quantizes the compressed signal according to a specific codebook. The codebook (which, in a quantization algorithm, helps to describe the organization or grouping technique used to quantize the data) is based on a normalized version of the compressed signal. It consists of normalized values in order to guarantee integer results after quantization. These integer values are then made positive and converted to binary. An array containing polarity information is appended to the end of the binary sequence.
- **Binary Phase Shift Keying (bpsk.m):** Binary Phase Shift Keying occurs when a sinusoidal carrier signal is generated (with one cycle per bit output by the ADC), the binary sequence's 0s are replaced with -1s, the binary sequence is extended (resulting in one bit per cell of the sinusoid array), and the two are multiplied together. This inverts the appropriate cycles of the sinusoid, resulting in an analog representation of the binary sequence.
- **Transmitter's Square Root Raised Cosine Filter (srrc_trans_filter.m):** Professor McNair suggested the inclusion of two SRRC filters for completeness. The transmitter's filter is responsible for ensuring that the transmission does not become wider than necessary prior to detection by the receiver. A two-to-one ratio governs the filter's parameters (filter sampling frequency and input signal sampling frequency, respectively), leading to an upsampling factor of two. The filter's output (i.e. the signal transmitted across the radio spectrum) also features amplitude that is lower than the original BPSK signal's. It is worth noting that this filter was the reason for the one second input specification—MATLAB's square root raised cosine filter operation is very memory-intensive.
- **Receiver's Square Root Raised Cosine Filter (srrc_rec_filter.m):** The receiver's SRRC filter is the first component in the receiver chain, and it completes the filter cascade. It corrects the amplitude change applied by the transmitter's filter, but leaves the upsampling characteristic unchanged.
- **De-Binary Phase Shift Keying (debpsk.m):** This component undoes (or demodulates) the BPSK carrier wave, determining the received binary sequence from this signal. It uses a correlation technique, multiplying each cycle of the received signal with a reference cycle. Because this reference cycle represents a binary 1, the product can be used to determine each received bit.
- **Digital to Analog Conversion (dac.m):** Digital to Analog Conversion is performed on the received binary sequence. It is converted to decimal form, de-normalized, and expanded with the Mu-law compander. The result is the received spread input signal.

- **De-Spread (despread.m):** The De-Spread operation removes the Walsh Code from the received signal. Any amplitude values above 1 or below -1 are clipped to 1 or -1, respectively.
- **Voice Playback:** Though it is no longer its own M-file (3SA's playback of the received signal was more easily implemented within the application itself, rather than as a separate M-file representing a system component), the Playback component is the final piece of the DSSS simulation. It is here that the input and received signals can be compared both visually and audibly.

Issues Encountered

A few troublesome issues were encountered and overcome during the development of 3SA:

- **Excessive memory usage:** Originally, the Analog to Digital Conversion component consisted of a Pulse Code Modulator, using the built-in *uencode* function with eight-bit precision in MATLAB (Appendix B, Section 4). Though the PCM method made intuitive sense to the researcher, it was repeatedly tested with a signal that had been spread using an eight-bit Walsh Code. The size of the changing data was growing very quickly, reaching eclipsing 135 megabytes in the MATLAB workspace. This led to “out of memory” errors when attempting to filter the resulting BPSK signal with the transmitter's SRRC filter, especially due to the filter's upsampling. The first attempt to avoid this issue consisted of cutting the bit operations in half (i.e. using four-bit spreading and four-bit quantization), but the memory accumulation was still quite large. Also, the recovered voice input was of very low quality. Finally, per Professor McNair's suggestion, the Mu-law compression algorithm was implemented (see Component and Algorithm Changes).
- **Trouble identifying bits represented by received BPSK carrier wave:** *debpsk.m*'s original algorithm identified bits in a “visual” manner, seeking out and keeping track of certain qualitative properties of the received signal (Appendix B, Section 5). Preliminary tests revealed a problem with this component, as roughly 25% of the bits represented by the modulated signal were incorrectly identified by the De-BPSK algorithm. The root of the problem turned out to be a breakdown in the BPSK method, which generated some erratic data in the modulated signal (Appendix C, Figure 4). Correcting the BPSK algorithm solved the problem (Appendix B, Section 6), and the addition of a correlation demodulation method made the De-BPSK algorithm more efficient (see Component and Algorithm Changes).

Component and Algorithm Changes

Several key changes were made to the base simulation during its development, including:

- **Spreading factor reduction:** The simulation's spreading factor was reduced to cut the total number of bits down (Walsh Code lengths were originally eight bits, then four, and finally two). This improved the simulation speed by reducing the amount of data the transmitter's filter needed to process. The signal was still spread, but the length of the Walsh Code was not deemed important when considering 3SA as a program. To the user, it makes no difference whether their input was spread with two bits or 256 bits. The memory saved, however, made the reduction necessary and helpful. Interactive features of 3SA make up for this reduction in the base simulation's Walsh Code length (Appendix D, Figure 5).
- **Replacement of Pulse Code Modulator with Mu-law compression algorithm:** This, coupled with the Walsh Code length reduction, helped to decrease the total number of bits processed by the simulation components. The algorithm compresses the data to the point that four bits are sufficient for binary conversion (see 3SA Progress and Development: Component Descriptions). The overall result is a better sounding output signal.
- **De-Binary Phase Shift Keying correlation method:** The original BPSK demodulation method (Appendix B, Section 5), though it worked perfectly, was inefficient and unrealistic. Professor McNair suggested that a correlation method be implemented, where the received signal was compared to some sort of reference signal (Appendix B, Section 7). Not only is the correlation method more efficient, but it is also resistant to low levels of random noise (see Appendix C, Figure 11).

3SA as Software

The development as 3SA as a program was a straightforward but very involved process. Because it is a medium between the completed research and the outside world, the user interface is very important. Basic features, including allowing the user to input a voice signal with a microphone, see the simulation unfold through graphics, and examine the result of each component's work, were considered early on in the research period. Other aspects, such as stability, were not considered until the software development phase began.

Interactivity between the DSSS simulation and the user was an important consideration. The layout of 3SA's main window resembles a block diagram of the system (Appendix A), and each component is represented by a labeled pushbutton. After the full simulation is complete, the user is able to click on a pushbutton to learn more about that particular component's purpose and implementation. In order to further accommodate the user's interest in the simulation, certain component windows (formally referred to as "figure windows" in MATLAB terms) allow the user to input data and see the results of their parameters. With the Spread component window, for example, the user is able to specify a Walsh Code length, generate a Code of that size, and spread their input signal. This interactive operation is independent of the full simulation, of course, as it is not enabled until the simulation has finished running.

A large portion of time was devoted to removing software “bugs.” For example, after spreading their input signal with a valid Walsh Code, the user is able to play back the spread result. As 3SA supports spreading factors of up to 256, user-spread signals can become quite long with respect to time. The ability to stop the playback of these signals needs to be clear and apparent, which is why the user is provided with a pushbutton labeled “Stop Spread Signal.” The software cannot be certain, however, that the user will actually make use of this button. Because of this, the MATLAB *CloseRequestFcn* figure window property (specifies what should be done if a figure window is closed) was utilized for the Spread window.

Conclusions

The Research Process

The research proved to be a new and exciting experience. A strong sense of responsibility was present throughout the summer program as the students were left to their own devices to plan, pursue, and execute their chosen or assigned investigations. The most rewarding part of the research program was the strong sense of contribution gained while developing 3SA; the notion that the researcher was given the opportunity to do work that would have an impact in the area of wireless technology education.

Future Improvements

In future versions of the software, it is hoped that the simulation itself can be made more realistic. This might include implementing other, more complex characteristics of DSSS systems, such as Walsh Code chip rates and the exploration of signal spectral density. With only eight to nine weeks to develop 3SA’s base simulation, these additions were just too much to handle. Also, the one second input signal limit is something that, with more time, could possibly be overcome. Professor Yu-Dong Yao suggested that longer inputs be broken up somehow, possibly in some sort of packet transfer fashion. Another worthwhile improvement would be the introduction of more transmitter/receiver pairs into the system. With this, the user would be able to witness the effectiveness of Walsh Code assignment. Extending the simulation to hardware was also suggested by Professor McNair, but time did not allow for this.

3SA as a Learning Tool

As mentioned before, one of the objectives in mind during the development of 3SA was its potential effectiveness as an educational tool. The researcher feels that this goal has been accomplished, as porting the simulation components to a functional program was a success. Though an attempt will be made to develop a stand-alone version of the application, it is probably more useful as a learning tool while being run from a MATLAB environment. This way, the user is able to view variable information, access the simulation’s data, and alter the code to fit their needs and the needs of their students. Next steps will include finding individuals or educational communities that may have a use for 3SA, and then finding a means of distributing the application.

References

Works Cited

Bible, Steve. *Spread Spectrum - It's not just for breakfast anymore!*, 1995. Tucson Amateur Packet Radio. 29 May 2005. <http://www.tapr.org/ss_intro.html>.

Maxim Integrated Products. *An Introduction to Direct-Sequence Spread-Spectrum Communications*, 2003. Maxim Integrated Products. 31 May 2005. <http://www.maxim-ic.com/appnotes.cfm/appnote_number/1890>.

McNair, Bruce. "Re: Ideas." Email to the author. 2 June 2005.

Roberts, Randy. *Introduction to Spread Spectrum*, 1998. Spread Spectrum Scene Online. 28 May 2005. <<http://www.sss-mag.com/ss.html>>.

Other Resources Consulted

The following resources were consulted extensively during the research period but were not referenced in the final report:

Barr, Michael, and Brian Wagner. *Introduction to Digital Filters*, 2002. Embedded.com. 16 June 2005. <<http://www.embedded.com/story/OEG20021119S0020>>.

Clapham, Matthew. "CDMA Communication System." Project results. The University of Newcastle, Australia.

Dartmouth College, Committee on Sources. Sources: Their Use and Acknowledgement. 1998. 21 July 2005. <<http://www.dartmouth.edu/~sources/>>.

Farley, Tom, and Mark van der Hoek. *Cellular Telephone Basics: AMPS and Beyond*. Privateline.com. 30 May 2005. <<http://www.privateline.com/Cellbasics/Cellbasics.html>>.

Gentile, Ken. "The care and feeding of digital, pulse-shaping filters." RF Design.com (2002): 50-61.

Irving, Terrence. *Stevens Research*, 2005. National Science Foundation Research Experience for Undergraduates in Software-Defined Radio. 20 July 2005. <http://www.cs.dartmouth.edu/~tmi/stevens/stevens_index.htm>.

Mandayam, Dr. Shreekanth. *Making Music with MATLAB*, 1997. Rowan University. 4 June 2005. <<http://users.rowan.edu/~shreek/networks1/music.html>>.

Marchand, Patrick, and O. Thomas Holland. *Graphics and GUIs with MATLAB: Third Edition*. New York: Chapman & Hall/CRC, 2003.

Meel, Jan. "Spread Spectrum (SS) Introduction." Project results. De Nayer Instituut, 1999.

Pelletier, Benoit. *CDMA Technology*. McGill Telecommunications and Signal Processing Laboratory. 5 June 2005. <<http://www.tsp.ece.mcgill.ca/Telecom/Docs/math.html>>.

Pelletier, Benoit. *Orthogonal Variable Spreading Factor Codes*. McGill Telecommunications and Signal Processing Laboratory. 5 June 2005. <<http://www.tsp.ece.mcgill.ca/Telecom/Docs/OVSF.html>>.

Poland, K.L. "Quantization and Pulse Code Modulation." University of Illinois at Chicago: 17-22.

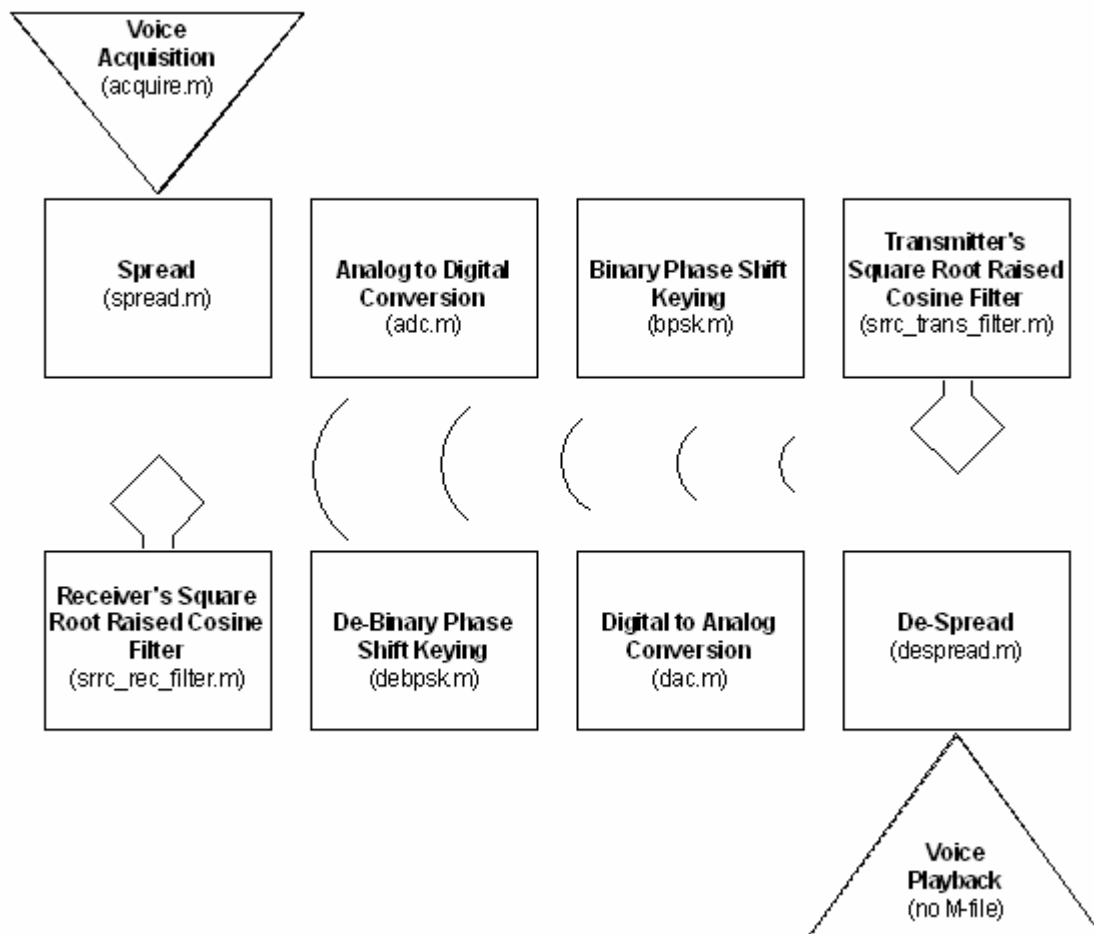
Pulse Code Modulation, 2002. Everything2.com. 16 June 2005. <http://www.everything2.com/index.pl?node_id=1375155>.

Wikipedia. *Mu-law algorithm*. Wikimedia Foundation. 22 June 2005. < <http://en.wikipedia.org/wiki/Mu-law>>.

Wikipedia. *Phase-shift keying*. Wikimedia Foundation. 8 June 2005. <http://en.wikipedia.org/wiki/Phase_shift_keying>.

Appendices

Appendix A: System Block Diagram



Appendix B: Weekly Reports

The following are reproductions of weekly progress logs ranging from the research program's beginning to the time this report was written. The original reports can be found [here](#).

Section 1 — Week 1

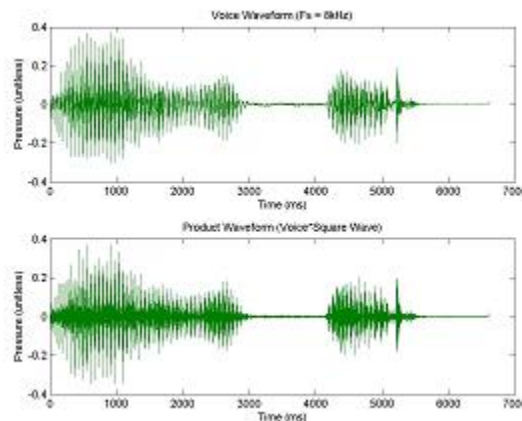
- Created personal research website
- Met with advisor, Professor Bruce McNair, and discussed possible research paths
- Completed preliminary research on wireless communication and RF aspects in general
- Began researching spread spectrum technology

Section 2 — Week 2

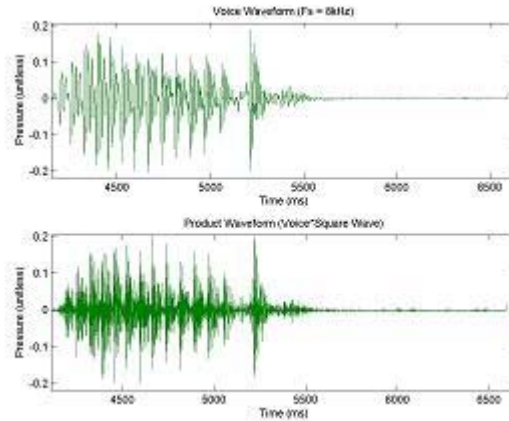
After further consideration, research, and discussion with Professor McNair, I have decided to take my initial research topic (spread spectrum communications) and develop an **interactive MATLAB application** as my project. I hope that the next several weeks that I spend developing this program will extend my knowledge of SS communications and programming in MATLAB, but I am also aiming to create something that others can make use of. If all goes according to plan, the application will consist of a graphical user interface that allows the user to speak into a microphone and then **see and hear** the effects of direct sequence spread spectrum on their voice data. Time allowing, Professor McNair mentioned the possibility of attempting to further the simulation through the use of two PCs (one to transmit and the other to receive) and ultrasonic transducers.

To get started, my advisor, Professor McNair, suggested that I try and familiarize myself with the effects of applying different techniques to voice data with MATLAB. Therefore, my first goal was to take a [wav file](#) and apply a [square wave](#) to it. In short, this consisted of reading the file into MATLAB (and storing its data in a two-column matrix--one column for each of the left and right channels that the file utilizes), multiplying its data points with the corresponding data points of a 4 kHz square wave, plotting the resulting waveform, and converting the resulting data into another wav file for playback.

You'll notice that the [result](#) sounds strange, but the speech can still be made out if you listen carefully (if you can't quite understand it, the voice is saying "Hi, what's up?"). The time-domain waveforms for the original and resulting wav files support this (see below). They look similar, but closer inspection shows that they are not exactly the same.



The entireties of the original and "square waved" waveforms look similar.



A closer look, however, shows that the resulting waveform is quite different.

Because the y-axis values of the square wave alternate between -1 and 1, it makes some sense that the resulting wav file sounds somewhat different from the original.

My next task was to go further with this notion and examine a real-life version of it: Walsh codes. Walsh codes come from Hadamard matrices, which are based on the following definition:

$$H_1 = [1]$$

$$H_{2^n} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}$$

where **n is a power of 2** and the resulting matrix is square. The Walsh codes themselves come from the rows of the matrices. For example, the Walsh codes of length two and four (meaning they have "spreading factors" of two and four, respectively) come from the following Hadamard matrices:

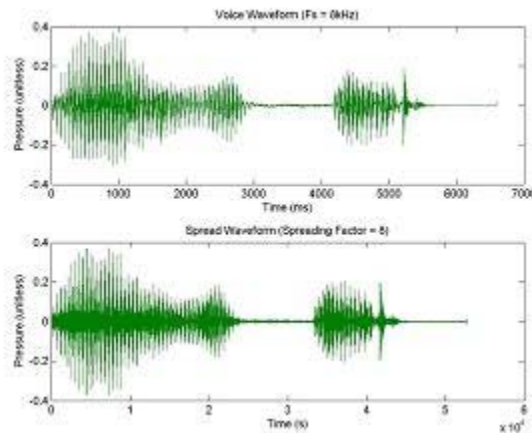
$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

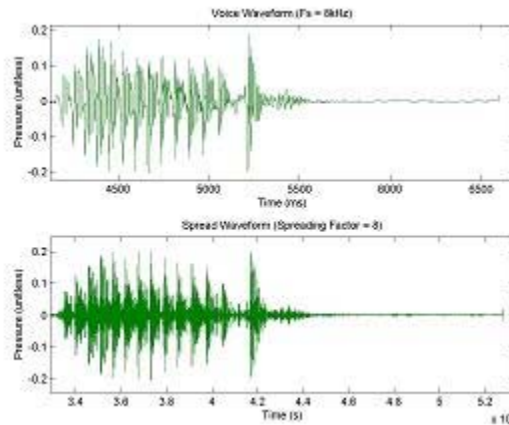
When the Walsh codes are taken from their Hadamard matrices, **0s become 1s and 1s become -1s**. That is, the Walsh codes with spreading factor two are (1, 1) and (1, -1), **not** (0, 0) and (0, 1). The reason for this (as will be seen shortly) is that applying 0s to voice data (or any kind of data) effectively erases that data, and erasing data isn't very useful in cellular communications.

Similar to the simulation described above, I next wrote some MATLAB code to apply a Walsh code with a spreading factor of eight to the same wav file. The code I chose to use was (1, 1, -1, -1, -1, -1, 1, 1). To make a long story short, the matrices that held the voice data and Walsh code data needed to be the same size, so some extending took place before the voice data was spread.

A close look at the resulting waveform emphasizes the purpose of the **spreading** involved in DSSS. The resulting waveform is thicker than the original, indicating that the original bandwidth has been increased by the application of the Walsh code (purposely increasing the signal's bandwidth before transmission is one of the keys to spread spectrum communications). If the waveforms themselves are not convincing enough, the [resulting wav file](#) should indicate the effects of applying the Walsh code.



Even from this view it is clear that resulting waveform is much fuller than the original. This is a direct result of the spreading that took place during the application of the Walsh code.



A closer look shows more detail of the effect of applying the Walsh code (also notice the difference in time scales).

It's important to remember that a spreading factor of only eight was used for this simulation--in some real-life systems, factors of 64 are used. Methods such as QPSK (quadrature phase shift keying) increase the availability of Walsh codes.

The importance of Walsh codes is that they allow for multiple signals to be transmitted and received on the same radio frequency without major problems with interference and security. For example, the use of a code system helps ensure that the correct receiver receives the correct transmission, and the noise-like result of spreading a signal helps to keep adversaries or non-intended receivers from intercepting a transmission (the military took advantage of this characteristic in the early days of spread spectrum communications). This, in simple terms, is the basis of some [CDMA](#) cellular systems.

This week I will go further with examining the effects of Walsh codes and signal spreading as I continue working on my project.

Section 3 — Week 3

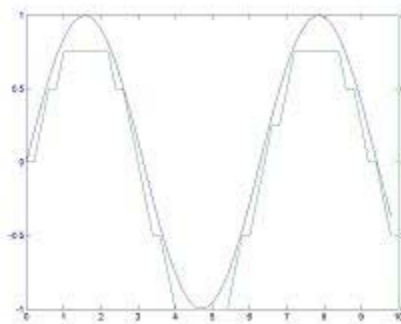
In order to make the development of the DSSS MATLAB simulation application more like real a world spread spectrum system, it was necessary to update the list of steps needed to prepare for transmission:

1. get the analog input signal (i.e. the user's voice)
2. **sample** and **quantize** the analog signal to get a binary version of its data
3. possibly randomize the binary version in order to increase its "noisiness"
4. spread the signal using a Walsh code
5. generate a **BPSK** (binary phase shift keyed) signal
6. filter the resulting signal to remove unnecessary bandwidth

Steps four and five are probably interchangeable, in that the injection of the Walsh code is only required to happen before transmission to the receiver.

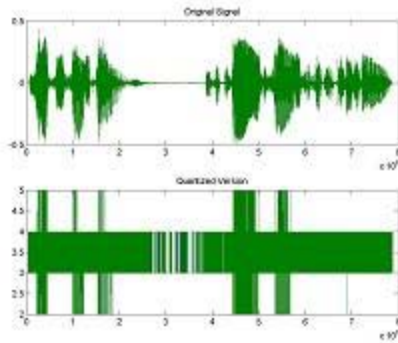
Step two touches on another interesting aspect of wireless communications: **analog to digital conversion**. Just as a review, analog signals and systems are continuously variable, meaning that they do not contain discrete or sampled information (in fact, the word "analog" itself implies "analogy," meaning "what comes in goes out unchanged, but possibly in a different form"). Examples of analog systems include speech (from mouth to ear) or clocks with hands on them. Digital signals and systems, on the other hand, contain or convey information that has been quantized (subdivided) and sampled from a discrete notion (the word digital implies "digit," and describes the use of numbers to convey data). Examples of digital systems include MP3s and powered clocks with numerical displays instead of hands.

In the case of direct sequence spread spectrum systems, the input signal is quantized and encoded into a binary form. The image below is a simple example of the beginning of this process.



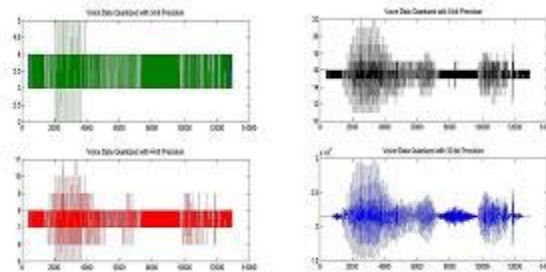
This plot shows a typical sinusoid (blue) and its eight-bit quantized version (green).

I haven't worked out the details for my project yet, but applying the same quantization function to a wav file of voice data leads to a plot that looks something like this:



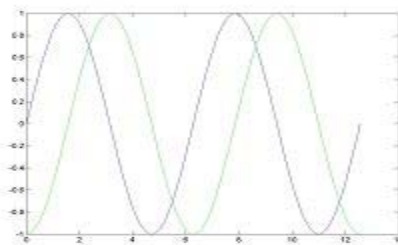
This plot shows the original and 3-bit quantized versions of a voice wav file's waveform in the time domain.

Altering the precision level (which ranges from two to 32 bits with the MATLAB implementation I have been using) can lead to higher **fidelity**, or accuracy of the quantized version as compared to the original.



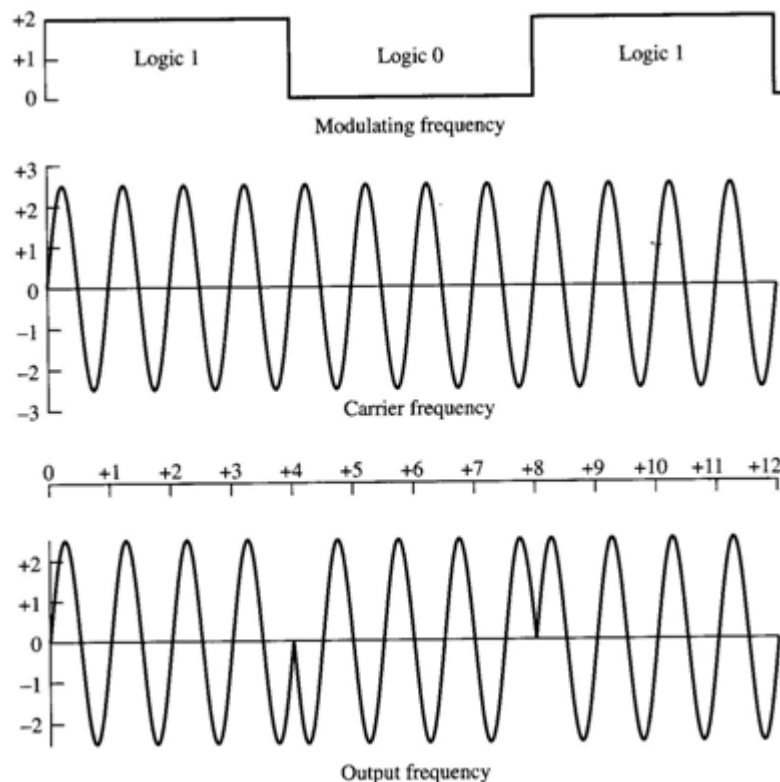
With a high level of precision (such as 32 bits), the quantized version of the data looks more like the original signal.

Step five introduces another modulation scheme, binary phase shift keying, or **BPSK** (modulation is basically placing a signal “on top of” a different carrier signal before transmission). This process enters the picture after the input signal has been digitized and then creates yet another representation of the data, this time with a sinusoid (although it doesn't have to be). When referring to waveforms, the word "phase" is a relative term--it relates the position of a waveform's feature (such as a peak) to that same feature on the same waveform or to the same feature on another waveform.



These two sinusoids are out of phase--the first peak of each sinusoid would be overlapping if the two waveforms were in phase.

Because this mode of phase shift keying is **binary**, this means that there will be two possible phase shifts: 0 and 180 degrees. The carrier, therefore, will be describing digital information (which is why this step comes after the quantization in the algorithm list above). For example, the carrier signal might start out (at $t=0$) following an "up, down, up, down" pattern in order to convey one piece of digital information. When a different piece of information needs to be described, however, the phase of the carrier signal will change, and the pattern will become "down, up, down, up." One way to put it is that the "sense" of the carrier signal has changed once it became necessary to describe another piece of the digital information. The result is a strange-looking sinusoid such as the one shown below.



The bottom plot of this image from ICT Technologies's Phase Shift Key Modulation [article](#) shows a resultant BPSK waveform.

The usefulness in implementing BPSK in wireless communications has to do with this modulation method's good performance in the presence of interference.

My next steps few steps will be to finalize my application's quantization scheme, create a BPSK generator, and then to create a filter to remove unnecessary bandwidth before transmission. Professor McNair also suggested that I randomize the quantized data (using a chaotic function created by Professor Yao) before spreading, so this is something that I will be looking into as well.

In closing, [here](#) is a 15 second clip of the result of spreading [last week's wav file](#) with a 256-bit Walsh code (the actual result is 3:31 long). For comparison purposes, the application of an 8-bit Walsh code to that same wav file sounds like [this](#).

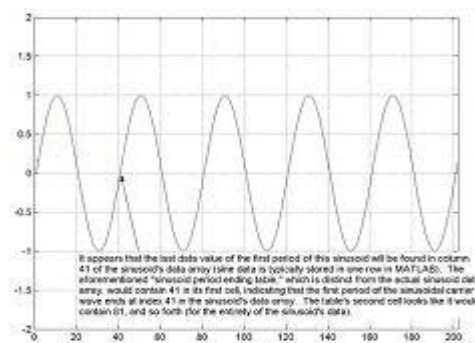
Section 4 — Week 4

The transmission chain of my project (an application written in MATLAB that allows the user to input, via a PC microphone, a voice signal and examine the effects of a spread spectrum system on that signal) is nearly complete. I altered the transmission steps again this week

1. get the "analog" input signal
2. spread the signal with a Walsh code
3. sample and quantize the analog signal to get a binary version of its data
4. generate a BPSK carrier signal
5. filter the BPSK signal with the first of two square root raised cosine filters

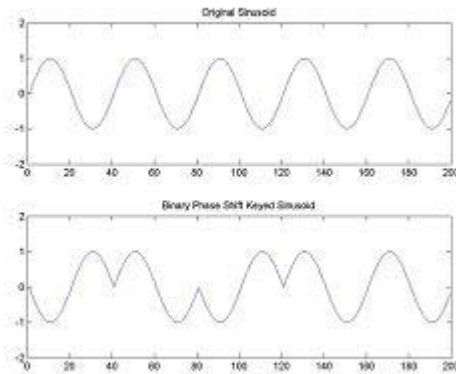
Last week, steps 2 and 3 were switched. After giving it more thought, I decided that spreading the user's signal before quantizing it made more sense to me than sampling and quantizing the signal before spreading it.

My binary phase shift keying component takes the encoded data (that is, the binary representation of the quantized signal) as an input. I decided to use a sinusoid as the base BPSK carrier signal, as this seems to be common in spread spectrum systems. In order to convey digital information with the sinusoid accurately, **there needs to be one period of the sine wave for every bit present in the binary sequence** (as a review, the encoded data consists of 0s and 1s). Therefore, the first step in my BPSK algorithm is the generation of a time vector (based on the length of the encoded sequence) that has "room" for one period of a sine wave for every bit of the binary sequence. Next, the sinusoid's data is created from this time vector. A table of the ending indices of each period of the sinusoid is created. This is worth explaining further. The numerical data of the sinusoid is stored in an array in MATLAB. For each period of the sinusoid, there is a piece of data in the sinusoid array that represents **the last piece of data for that particular period**. At this point, the current period ends, and the next position in the array is where the following period begins. The table I was talking about before (the "table of the ending indices of each period") contains all of the indices of these period-ending values. The illustration below will hopefully make this more clear:



After identifying a pattern for finding the indices of the ends of sinusoid cycles, an array was created that held the index numbers.

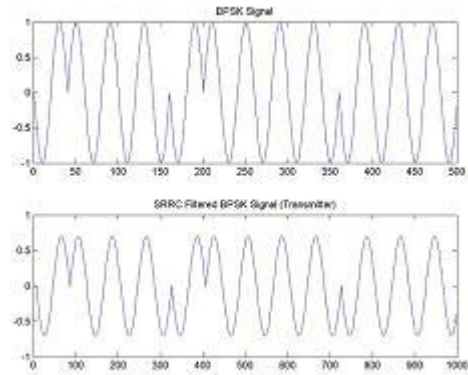
After generating this index table, the 0s in the encoded data array are changed to -1s. This makes the next step easier, which is multiplying the altered encoded data array with the sinusoid data array (after it has been confirmed that they have the same dimensions). The following plot shows the resulting BPSK carrier wave, which conveys the binary sequence with a sinusoid:



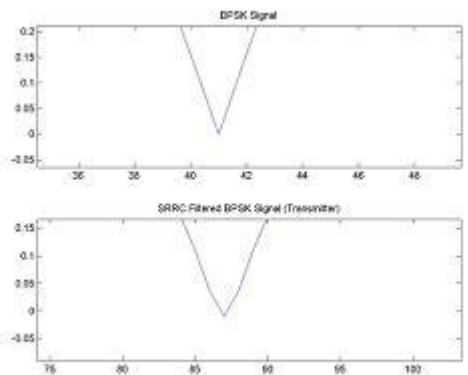
The changes in phase of this BPSK signal convey a digital sequence.

Finishing the BPSK algorithm led to another problem—memory usage. I was running the BPSK simulation (unwisely) on voice data that had not been spread (I think this was unwise because had I been simulating with spread data, I would have noticed the problem sooner). When I ran it for the first time with a spread signal, I realized how quickly the size of the simulation's data was growing. The time and sinusoid data arrays took up over 135 MB of memory themselves (each) when using an 8-bit Walsh code and 8-bit quantization. I've decided that a good way to avoid this is to process the user's with 4-bit operations, and then allow the user to examine their input with higher bit operations separately (i.e. 4-bit will be the application's standard, but a separate function of the program will allow the user to view waveforms of, say, the result of applying a 256-bit Walsh code, but the audio itself will not be processed through the entire spread spectrum algorithm with anything greater than 4-bit operations).

Professor McNair informed that in order to make my project more realistic, a **square root raised cosine filter** should be present at the end of the transmission chain. Such a filter allows for alteration of a signal's characteristics before transmission, such as its bandwidth (for a complete explanation, see [this](#) page). Upon further investigation, I've found out that placing filter at the end of the transmission chain and another identical filter at the beginning of the reception chain will be optimal for my spread spectrum simulation. The purpose of this **cascade** is to "partly filter" the BPSK signal before transmission and then filter it the rest of the way upon its arrival at the receiver. This is advantageous because the full benefit of a single square root raised cosine filter is preserved (i.e. not destroyed by interference, etc.) by filtering the signal twice with the same filter (once before it is transmitted and again after it has faced the environment of the radio spectrum). I'll be exploring square root raised cosine filter design some more in the coming week, but here are some plots.



This image shows the effects of filtering an example BPSK signal with a square root raised cosine filter. Altering the filter's parameters change the x-axis characteristics of the filtered signal. Notice that the amplitudes of the two waves are not the same. Filtering the signal a second time (at the receiver) will restore the amplitude to (close) to its original magnitude.



Another result of filtering with a square root raised filter is that the input signal's transitions are rounded. That is, sharp edges or points of the input waveform appear more gentle or gradual after the signal has been filtered. Again, altering the filter's parameters, such as its rolloff factor, affect this result.

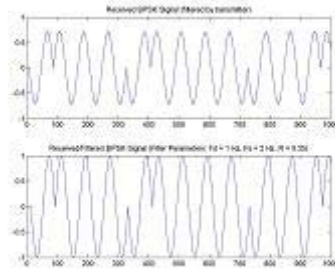
Next week will bring the completion of the filter design (and the transmission side of the application). Then it will be time to begin working on the receiver and the graphical user interface.

Section 5 — Week 5

This week I continued looking for ways to improve the speed of my algorithms, and Professor McNair suggested that I look into a compression system such as Mu-law or a-law. Because it is a North American (and Japanese) standard (a-law is used in Europe), I devoted most of my attention on this topic to Mu-law. The basis of the Mu-law telecommunications compression standard is that the input signal's dynamic range (spectrum of frequencies) is more efficiently represented by optimizing the range before the signal is digitized. The most important difference between Mu-law encoding and the method that my application currently uses is that Mu-law is based on logarithmic mathematics. Mu-law's equation includes some natural logs, absolute values, the signum function (which in MATLAB creates an array the same size as the input and fills that array with 1, -1, or 0 depending on the corresponding element's relation to 0), and μ (which is equal to a unitless 255--the μ in a-law compression is equal to 87.5). Implementing Mu-law in MATLAB requires the use of companding (a combination of *compressing* before quantization and *expanding* to undo the operation) and quantization functions that I'm not used to yet, so I'm going to leave the compression algorithms alone for now.

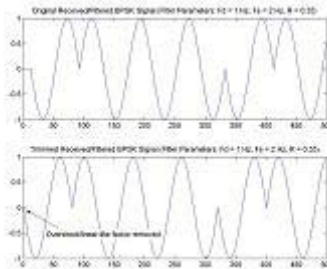
Week 5's most significant task was the creation of the "de-BPSK" algorithm, which undoes the carrier-wave-creating algorithm that I had been working on in previous weeks. This was probably the most difficult component so far, as there was no clear-cut way (that I could immediately see) to go about implementing this.

Square root raised cosine filter design was the first part of this component, as one SRRC filter sits at the end of the transmitter chain and another sits at the beginning of the receiver chain. They are only involved in my project detail (in other words, they are used in real-world spread spectrum communications). I designed the filters in a cascaded setup (i.e. the output of one becomes the input of the other) using MATLAB Communications Toolbox functions. In short, MATLAB design of these filters includes upsampling the original BPSK signal with the transmitter SRRC filter, but the receiver filter is designed not to further upsample the signal (MATLAB doesn't allow the sampling frequency of the input signal and the filter to be the same, which would eliminate the upsampling as far as I can tell--also, the ratio of the two sampling frequencies is *about* the magnitude of the upsampling factor). This implementation also gives some interesting properties of the two waveform's amplitudes--the first filter's output has a smaller amplitude than the original BPSK signal, but the amplitude returns to "normal" after passing through the receiver filter. The image below shows this:

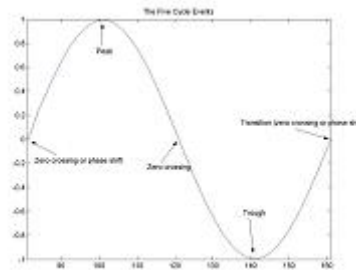


The output of the transmitter SRRC filter (top) has a smaller amplitude than the original BPSK signal, but the output of the receiver filter (whose input is the output of the transmitter filter) features an amplitude that is more like the original.

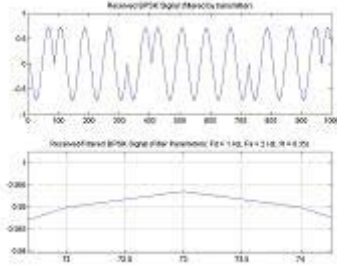
Moving on to the de-BPSK algorithm itself, I soon realized that it would be a difficult task. What first got my attention was the fact that the output of the receiver SRRC filter was not a true sinusoid, but a sine-like waveform with overshoot and other slight imperfections. But these differences, however, would become useful later. The following is my attempt to briefly explain this otherwise complex de-BPSK algorithm with text supplementing the images, rather than the other way around:



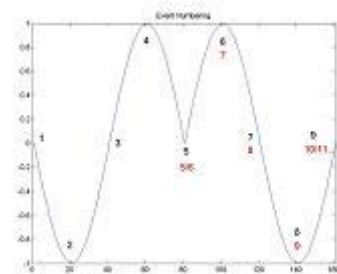
The first thing the algorithm does is remove the filter's residual beginning and ending characteristics.



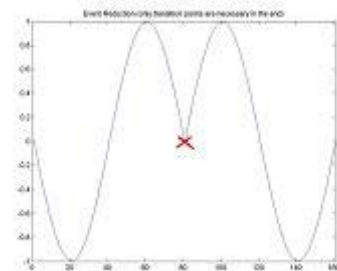
The signal's cycles are characterized by five events. A "normal" sinusoid cycle is shown for clarity (in other words, a BPSK sinusoid includes phase changes). The de-BPSK component traverses the entire waveform and creates a table that indexes every event. This table is later reduced to contain only the important parts--the transitions. Implementation of this step was very slow at first because of MATLAB function calls within the algorithm, but is now much faster because of a mathematical approach.



Here is an up-close view of a peak. Troughs and phase changes also have this triangular characteristic. Comparing the values of the three points (which can be referred to as "previous," "current," and "next") allows the algorithm to identify what the point is.



The first time through, events are numbered in black. Later on in the algorithm, however, they are numbered in red. The difference is that transition points are shared--the end of one cycle is the beginning of the next. This allows every cycle to possess the aforementioned five event standard.



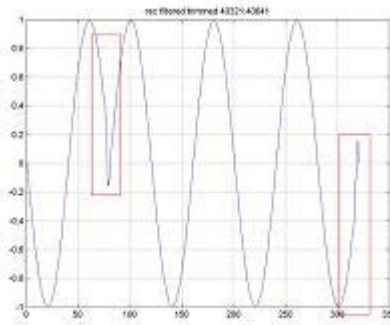
The final version of the event array only contains transition locations, as those are the only important events for determining bits. With this implementation, the total number of bits will be one more than the total number of transitions (the final cycle of waveform is considered to lack a transition point at its end). The last step of the de-BPSK algorithm is the bit determination. The first bit conveyed by the waveform is manually set based on the slope of the beginning of the signal. Subsequent bits, however, are determined by considering both the value of the previous bit and the type of transition between the previous cycle and the current cycle. The binary sequence shown here is "01."

At this point, the de-BPSK algorithm is complete but is not yet correct (it's only recognizing about 75% of the total bits). I plan on fixing it early in Week 6 and then move on to the user interface, after reconsidering the Mu-law compression possibility.

Section 6 — Week 6

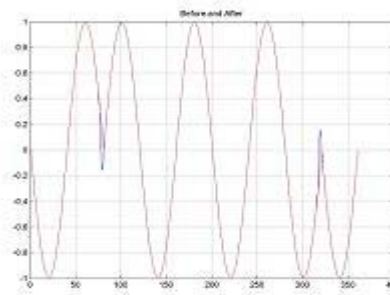
Some major changes and developments were made in Week 6. Week 5 ended with a de-BPSK algorithm that didn't work (it only identified about 75% of the bits represented by the carrier wave, and there was no saying whether those bits were correct or not). Also, the implementation of some kind of a Mu-law compression algorithm was wanted, but its status was still undetermined.

Before moving forward with the application, I had to identify the problems with the BPSK/de-BPSK transition. I thought that the filter or some other key element of the system was causing a situation that I wasn't noticing, but the root of the problem lay in the fact that my cycle transition-identifying code was the culprit. Somewhere near the middle of the pre-transmission BPSK wave were cycles that looked like this:



Erratic data (caused by the formerly over-complicated transition-identifying code) led to the poor 75% bit identification success in Week 5.

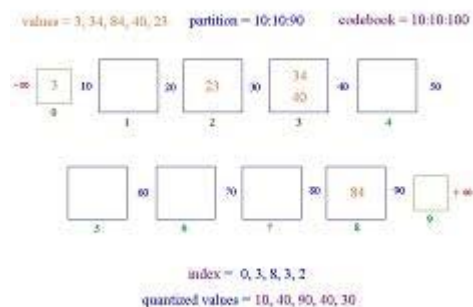
I simplified the code by noticing that the time vector from which the original, pre-BPSK sinusoid was created had a step size of $\pi/20$, which meant that a transition would occur every 40 data points after the first data point, and every 80 data points for the received BPSK signal (because of the square root raised cosine filters' upsampling). I actually noticed this some time ago, but ignored it because it seemed too simple of a concept to take advantage of.



This plot shows the before and after correction of the received BPSK signal.

To make sure that everything regarding the BPSK signal was working, I added some testing purposes code that compares the received binary sequence to the sent sequence and calculates the accuracy of the received data (this accuracy was low in the beginning, but now hits 100% every time).

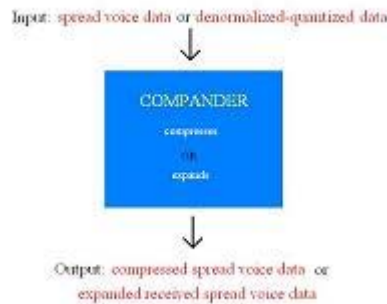
With the BPSK issues taken care of, I next moved on to Mu-law compression, which I was interested in implementing (from Professor McNair's advice) in order to replace the pulse code modulation component. I realized that in order to get a Mu-law compression system to work how I wanted (i.e. without needing to change much else in the application), I was going to have to quantize data differently with MATLAB. The image below depicts a quantization example found in MATLAB Help:



An explanation of the quantization function that the application currently uses.
This example (from MATLAB's Help) rounds five integers up to the nearest multiple of 10.

The reason I failed at first was because of my choice of partition--it resulted in quantized data that was composed of decimal values, not integers like I had hoped for. My next thought was to somehow normalize the quantized data so that it could be easily converted to binary and then transmitted, but I did not come up with a good implementation until the end of Week 6. The results data-wise (tens of thousands less bits to filter and transmit) and quality-wise (the output that comes from the Mu-law implementation is MUCH clearer than the PCM result) were very, very good.

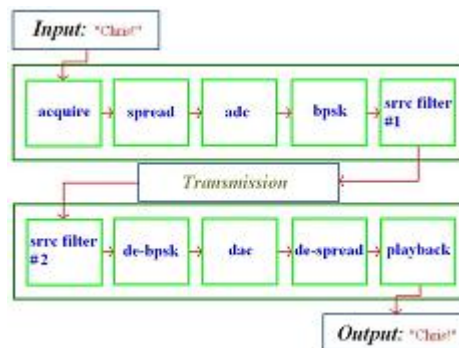
In order to make sure that I hadn't molded the Mu-law component to work only with my own voice (I also integrated microphone input into the system this week), I asked [Chris](#) for his help. Here is [my voice input](#) and the [Mu-law/ADC result](#). Chris was particularly enthusiastic, offering [this](#) for an input, and it resulted in [this](#) output. For reference, [here](#) is the result of my same input when sent through the old 4-bit PCM system. The image below shows a graphical representation of the Mu-law component's core--the compander (compressor/expander).



Companding a voice signal is the key to many telephone compression systems. Mu-law companders are based on the equation below (from Wikipedia.org).

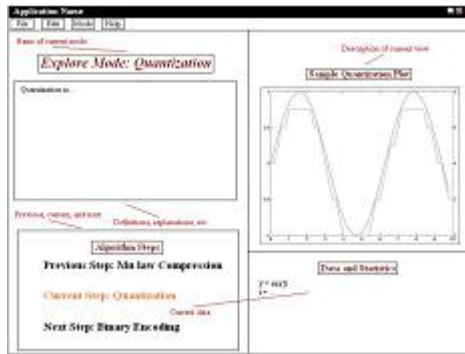
$$F(x) = \text{sgn}(x) \frac{\ln(1 + \mu|x|)}{\ln(1 + \mu)} \quad -1 \leq x \leq 1$$

Because of the important addition of the Mu-law algorithm, the PCM and de-PCM components are gone, replaced by ADC (analog to digital conversion) in the transmitter and DAC (digital to analog conversion) in the receiver. A diagram of the system can be found below.



The layout of my project.

Beginning in Week 7, the user interface will be my main focus. I have already begun considering it, as shown by the image below.



A rough draft of the user interface for the application.

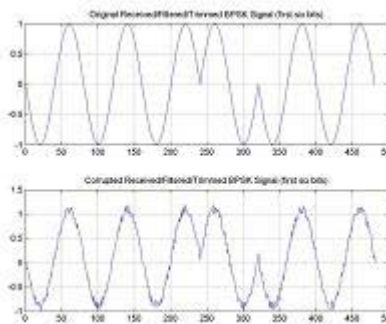
Section 7 — Week 7

During our meeting at the end of Week 6, Professor McNair suggested that I look into a more realistic approach for my de-BPSK algorithm. At the time, that component was taking sort of a qualitative/visual approach to determining the bits conveyed by the carrier signal, checking on the shapes of transition points (see [Week 5 Extended Description](#) for more details). I took his advice, and the results were more significant than I had anticipated.

What Professor McNair was suggesting was that I try and implement a **correlation** algorithm. In basic terms within the scope of my de-BPSK component, correlation refers to comparing the received BPSK carrier wave (with all of its phase changes to represent binary 1s and 0s) to some kind of reference signal in order to determine the binary sequence. Because the REU program's time is growing short, I have implemented the de-BPSK correlation in the simplest way that I could think of:

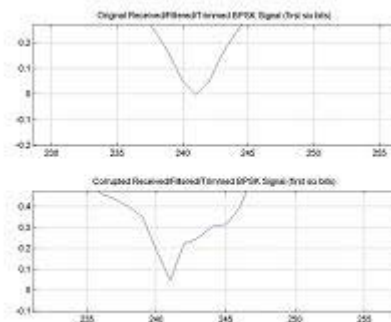
- the reference signal is one cycle of a "normal" sinusoid, representing a binary 1
- one by one, each cycle of the received, filtered, and trimmed BPSK signal is multiplied with the reference cycle
- if the resulting product is positive, this indicates that the phases match (1)
- if the resulting product is negative, this indicates that the phases do not match (0)

During our discussion, Professor McNair mentioned that this implementation would prove to be more realistic (real-world spread spectrum systems use correlation algorithms, although they are far more complicated than mine and are generally implemented with hardware) and also much more resistant to noise than my former de-BPSK component (because noise is not a part of my simulation, I hadn't taken it into account when developing the first de-BPSK algorithm). This second characteristic made sense to me, but I went ahead and generated some plots to see it in action. This image shows the first six bits (000100) of a received BPSK signal in the top plot, and the same signal corrupted with noise in the bottom plot. I ran the noisy signal through my correlation de-BPSK component and it passed, identifying all six bits correctly.



The bottom plot shows the effect of transmission noise on the BPSK signal on the top.

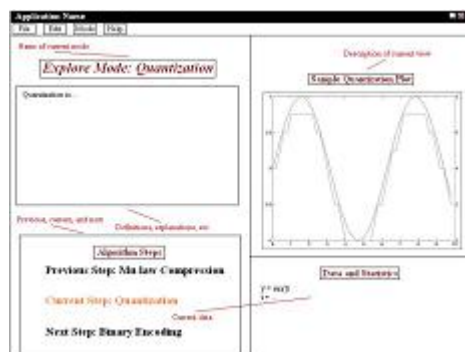
The next image shows why my old de-BPSK algorithm would fail with this noisy signal--the phase changes of the noisy signal are no longer ideal (i.e. composed of three points like the phase changes of the clean BPSK signal).



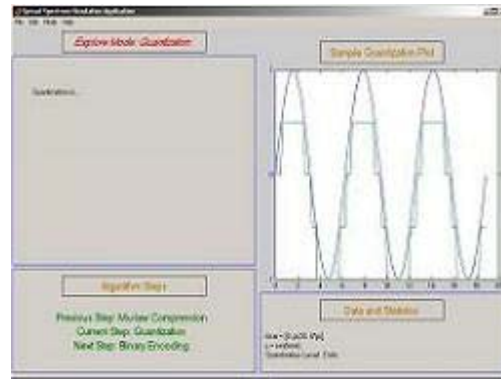
This close-up view of the two signal's phase changes shows why the old de-BPSK algorithm would fail when given a noisy input.

After putting this finishing touch on the inner workings of the project, I returned to the graphical user interface portion of the application. I completed the GUI chapter of *Graphics and GUIs with MATLAB* and coded several small GUIs in order to get myself more familiar with MATLAB's user interface objects and how they may interact with the actual spread spectrum components of my project. Following an example from the book, I also explored different ways to go about coding a functional user interface (i.e. storing the user interface objects in the base workspace, in the figure window's UserData field, etc.) and created a GUI that allows the user to plot a function.

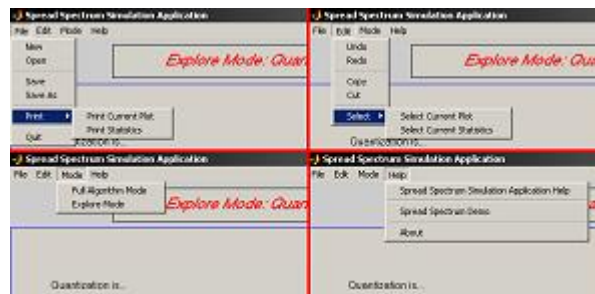
[Last week](#) I presented an image that showed what I thought the GUI's layout might be. Here is that image again, along with the actual non-functional MATLAB user interface version:



Last week's layout idea.



The actual MATLAB (non-functional) GUI that I created from that plan.



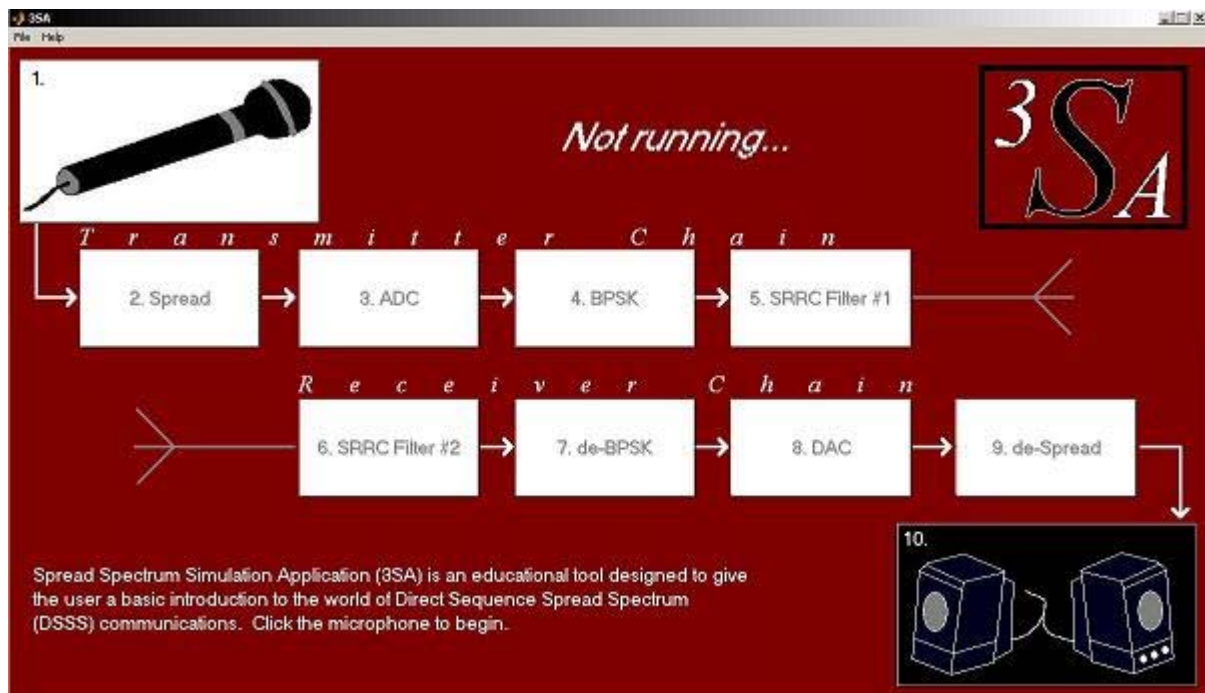
A close-up view of the menubar.

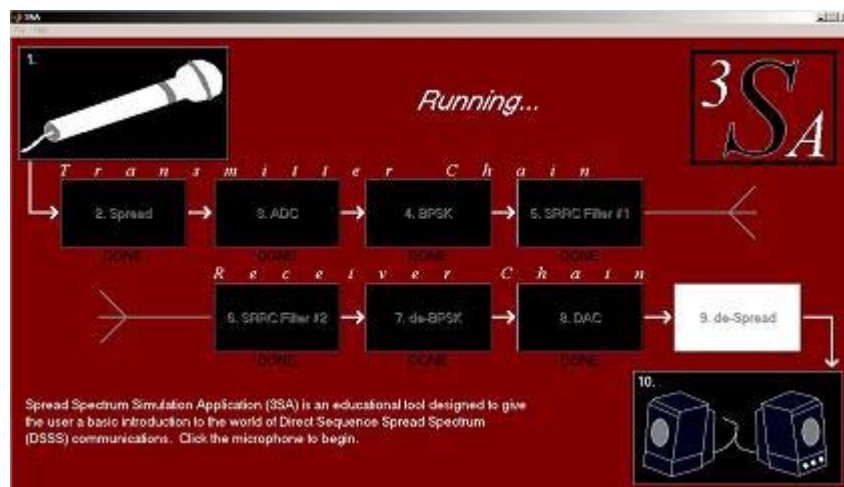
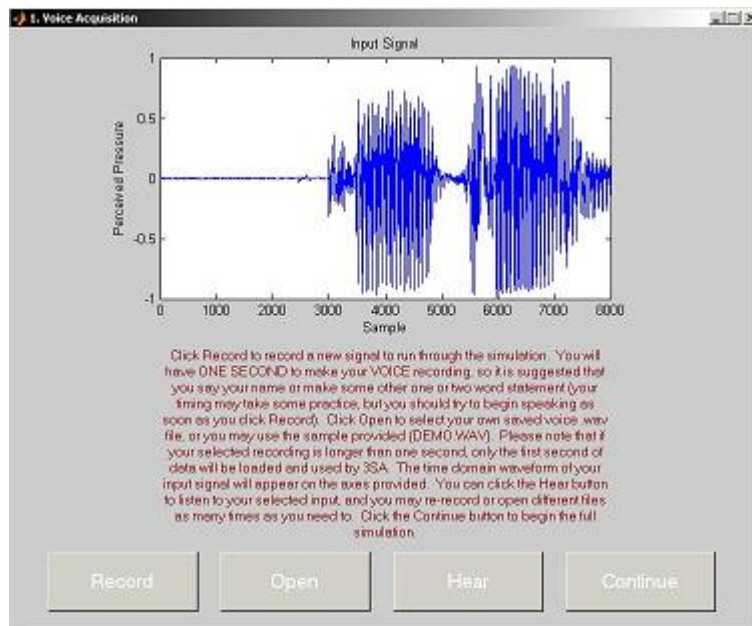
In Week 8 I will build the application's graphical user interface and then move on to the final report.

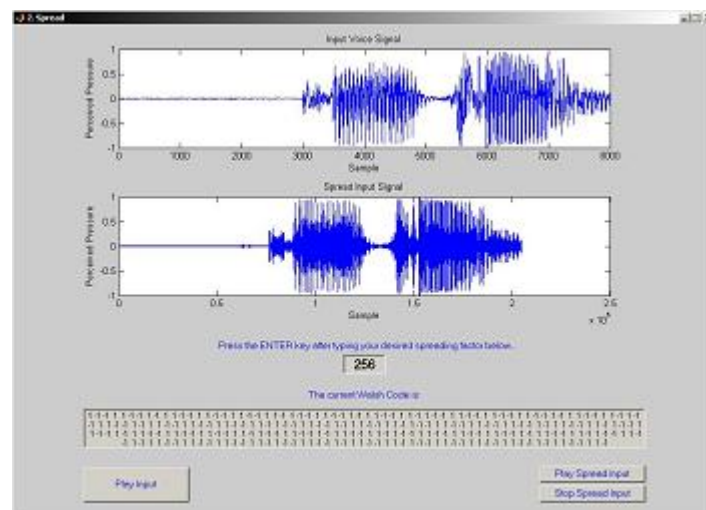
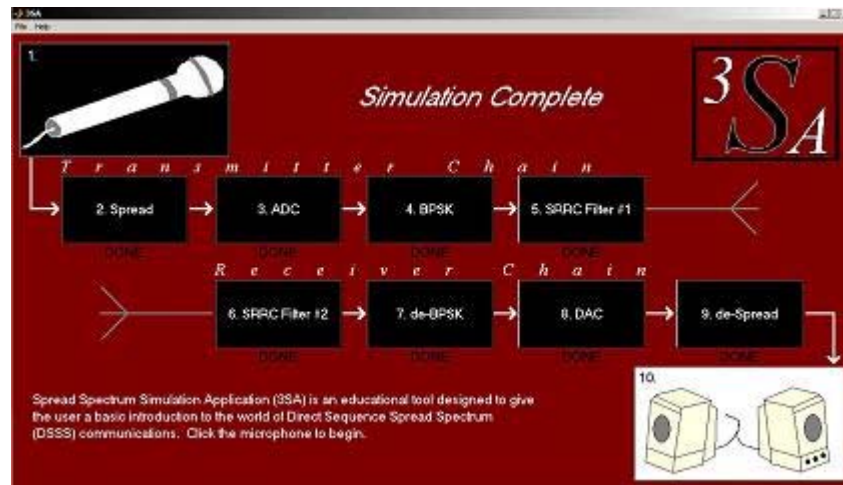
Section 8 — Week 8

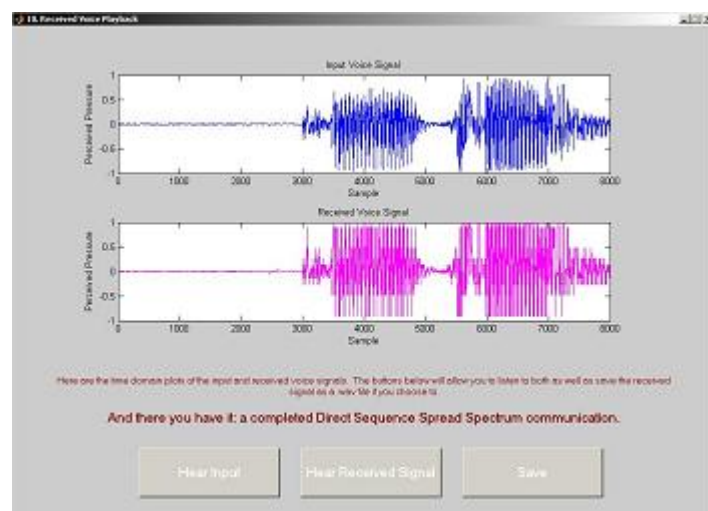
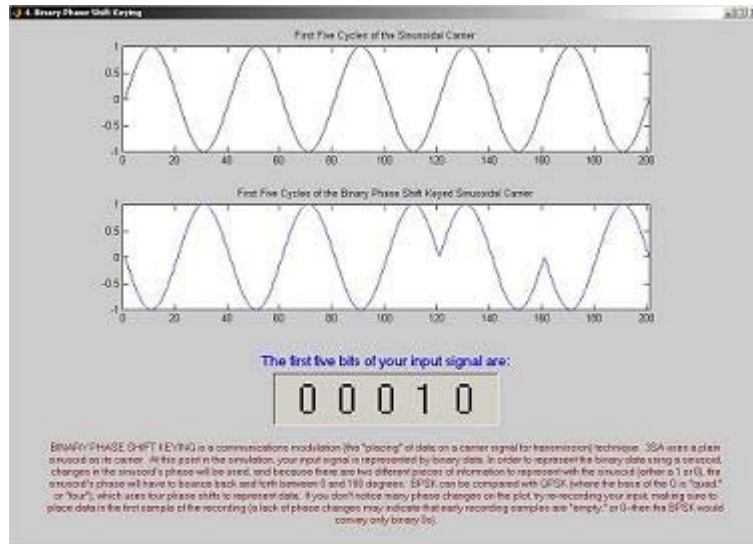
During Week 8 I finished cleaning up the source code of the DSSS simulation components and making the code readable. I also created the layout for my final report and began working on it. It is going well. The big news for this week is that I completed my project. I have not made a stand-alone application out of it yet (and I am beginning to think that it would be best to leave it how it is--an application made to run *with* MATLAB, rather than without it), but I will be looking into that this week.

And without further ado, here are some screenshots of the finished product, entitled **3SA (Spread Spectrum Simulation Application)**:









Appendix C: Key MATLAB Plots

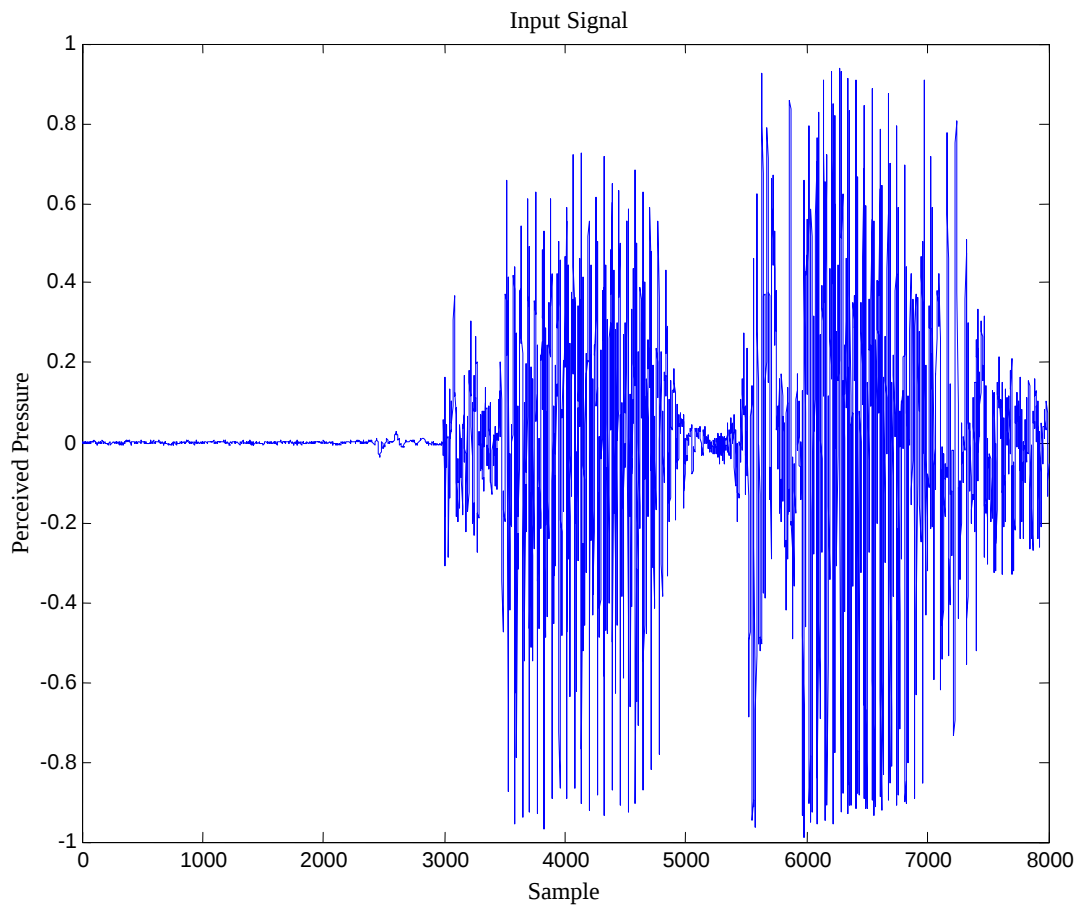


Figure 1: Sample input signal time domain plot. This waveform represents the spoken phrase “Terrence Irving.”

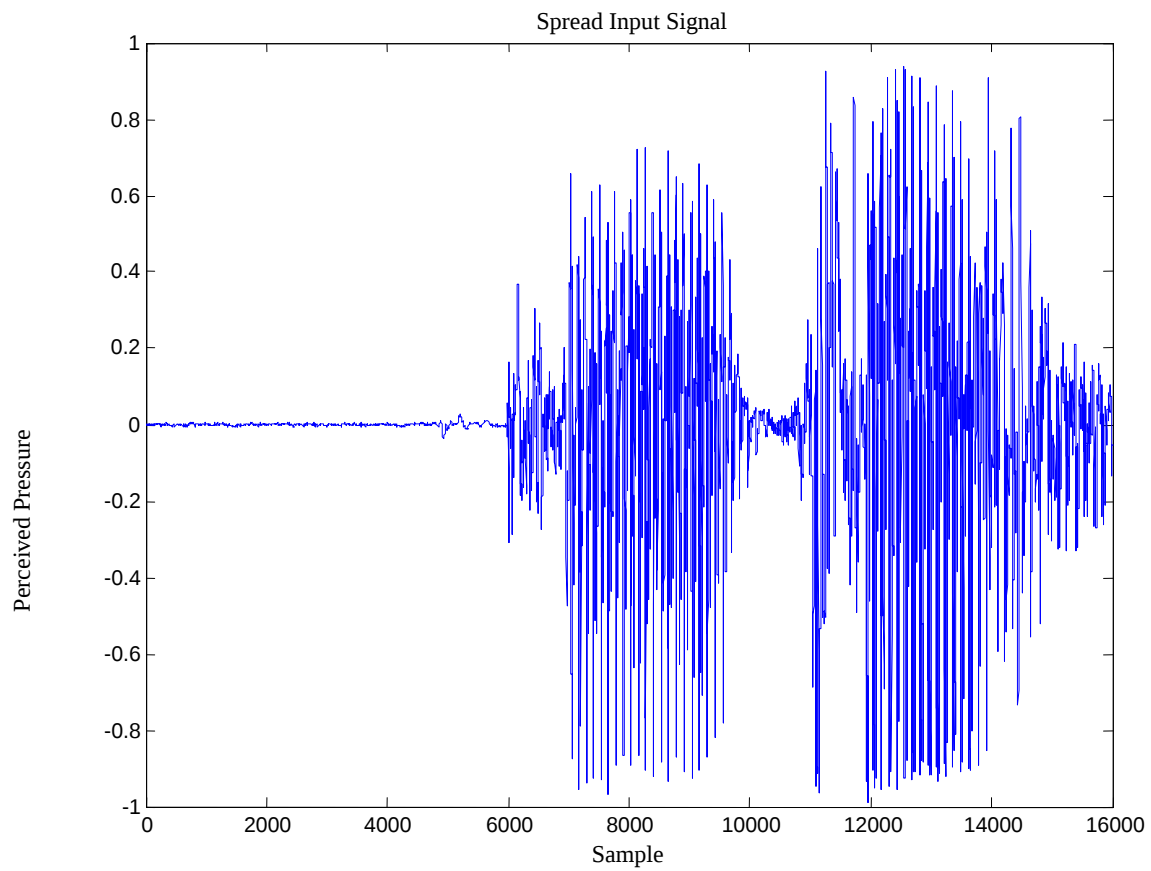


Figure 2: Spread input signal. The random two-bit Walsh Code used was 1 1.

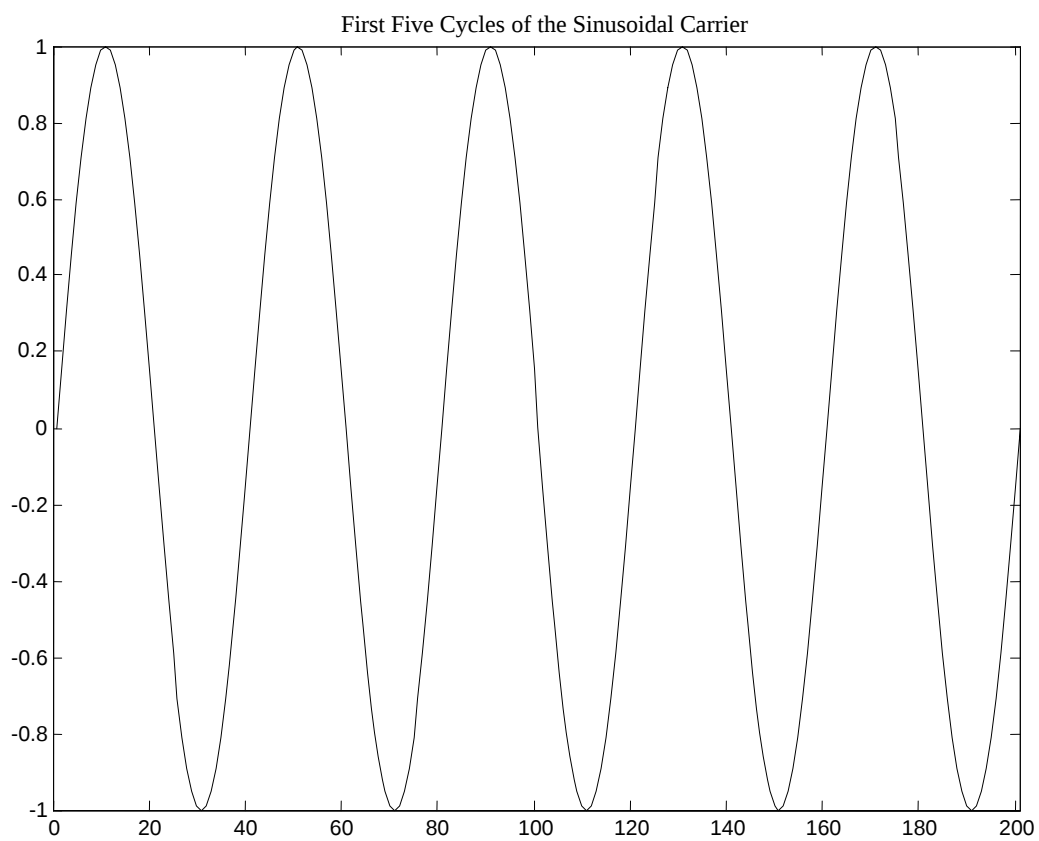


Figure 3: The carrier sinusoid.

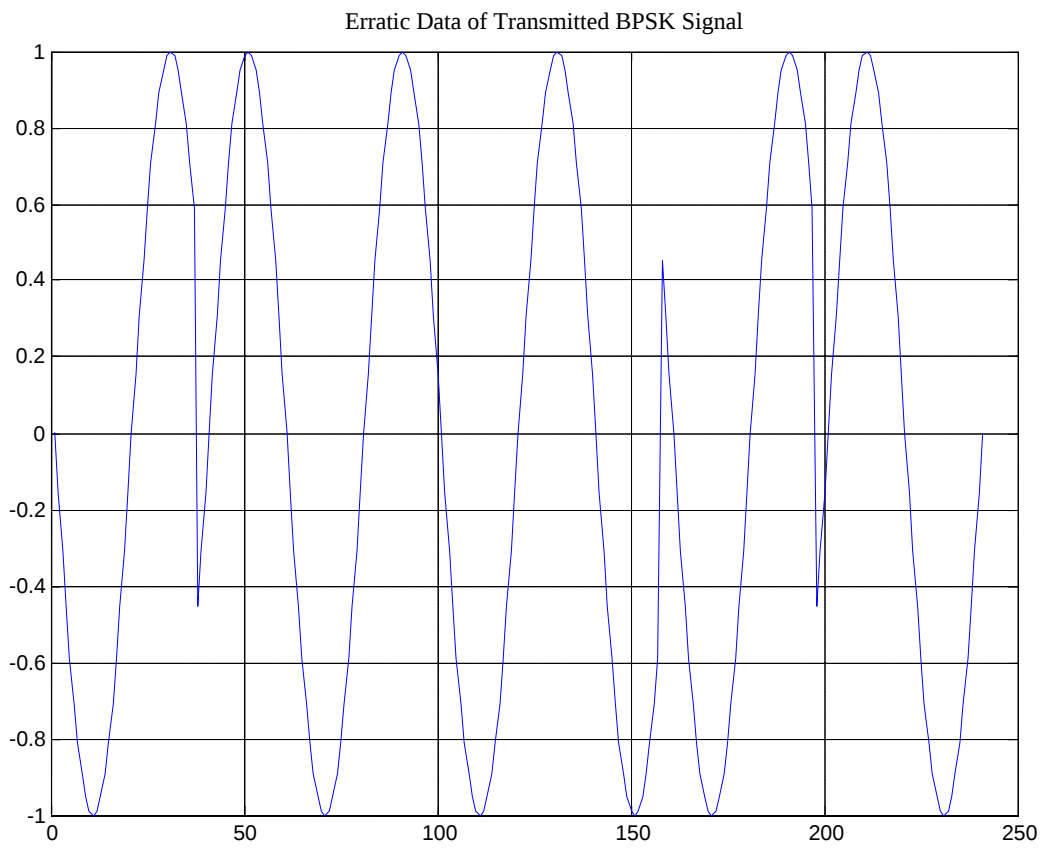


Figure 4: Erratic phase changes of this BPSK carrier signal occur near points 40, 160, and 200 on the horizontal axis.

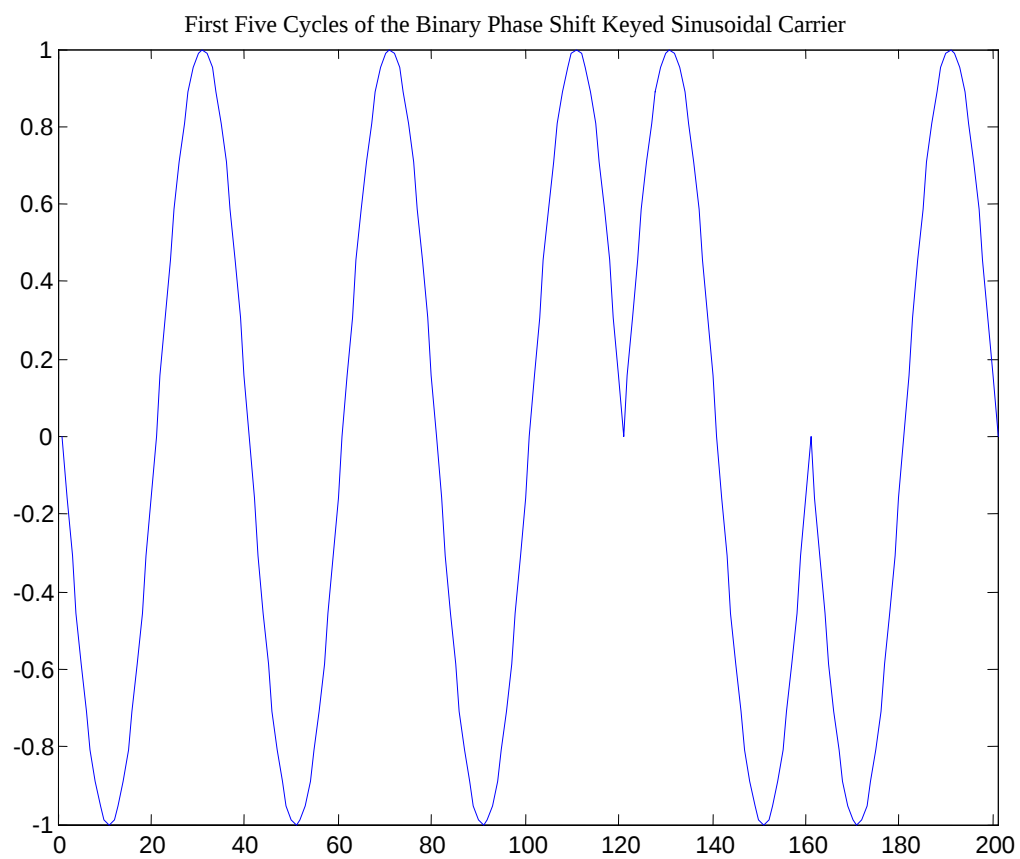


Figure 5: The correct Binary Phase Shift Keyed for the input signal in Figure 1.

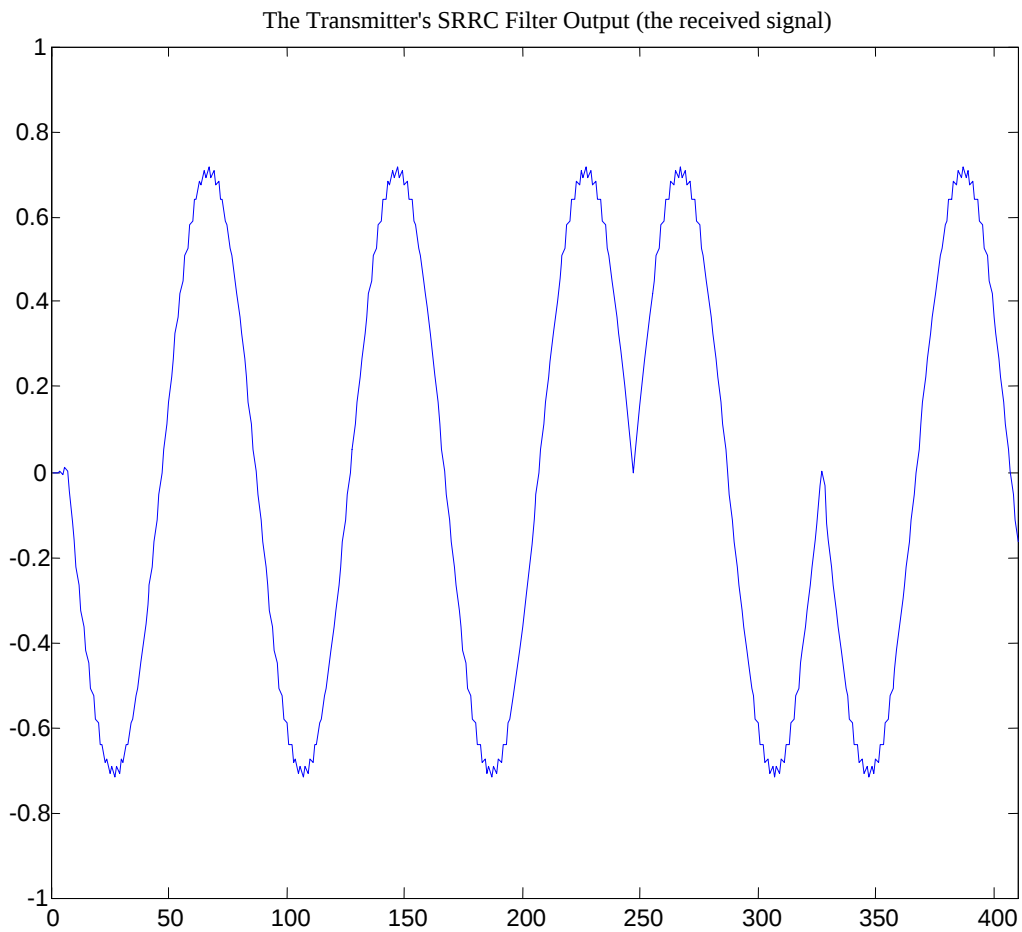


Figure 6: The transmitter's Square Root Raised Cosine Filter's output.

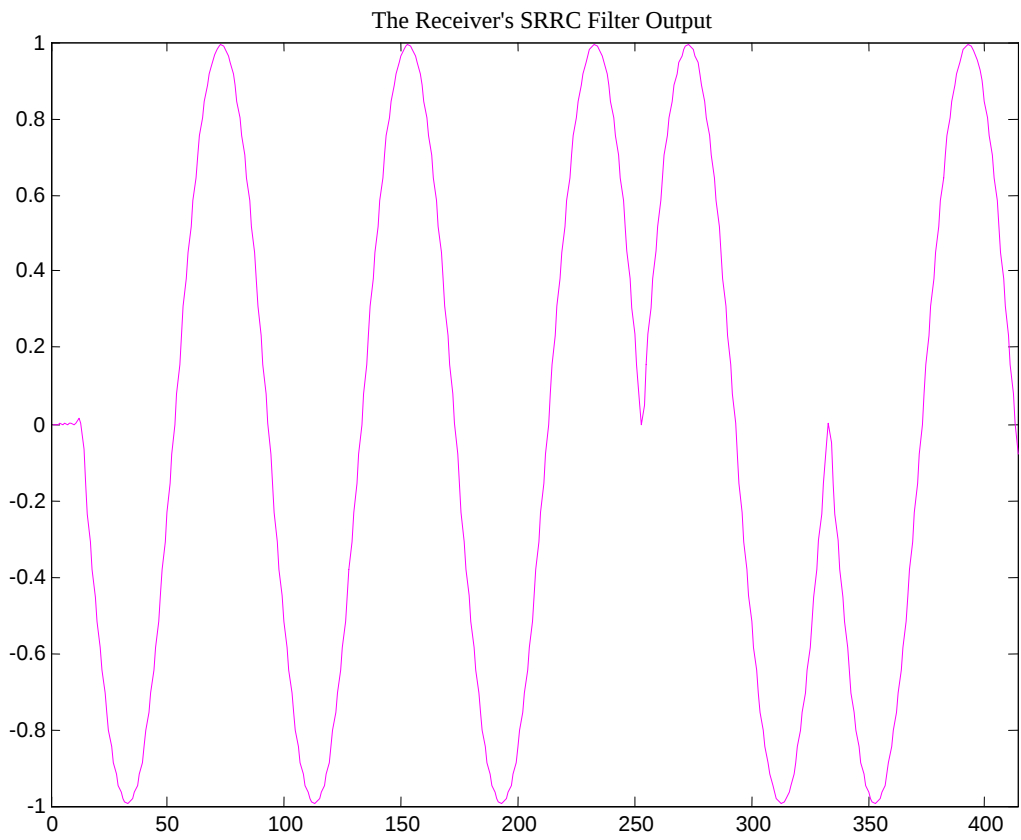


Figure 7: The filter's Square Root Raised Cosine Filter's output.



Figure 8: The output of the receiver's SRRC filter is trimmed to remove spurious data prior to demodulation.

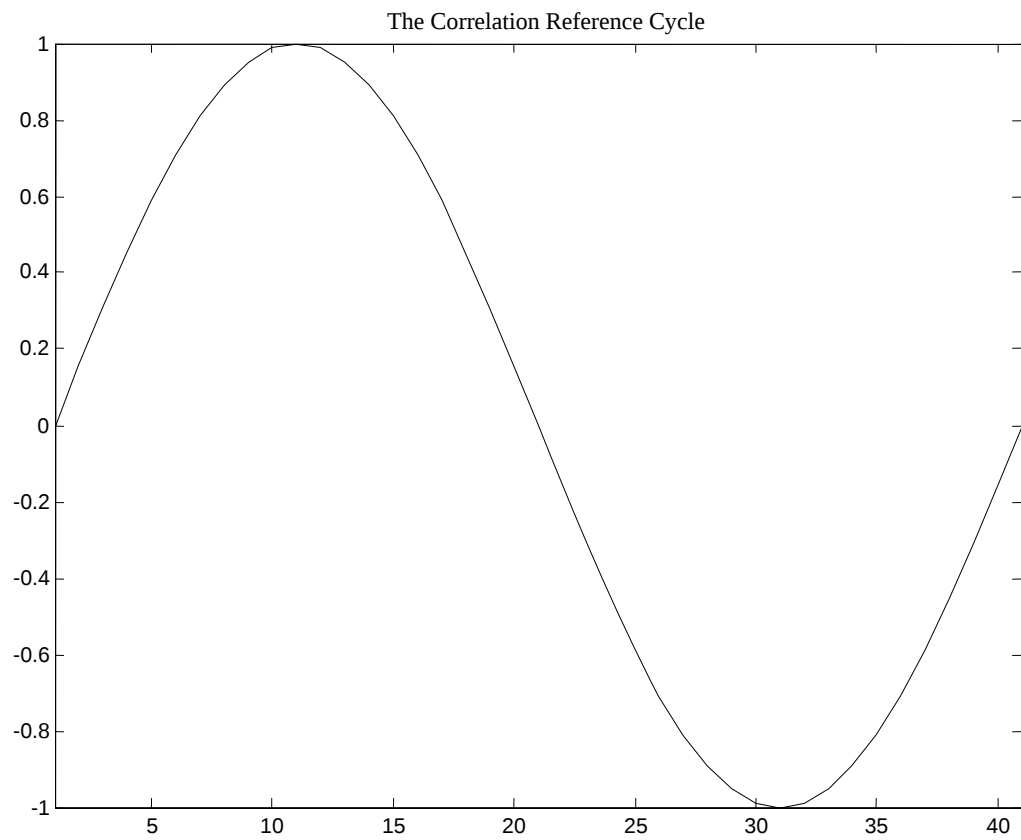


Figure 9: The cycle used for De-BPSK correlation. It represents a binary 1.

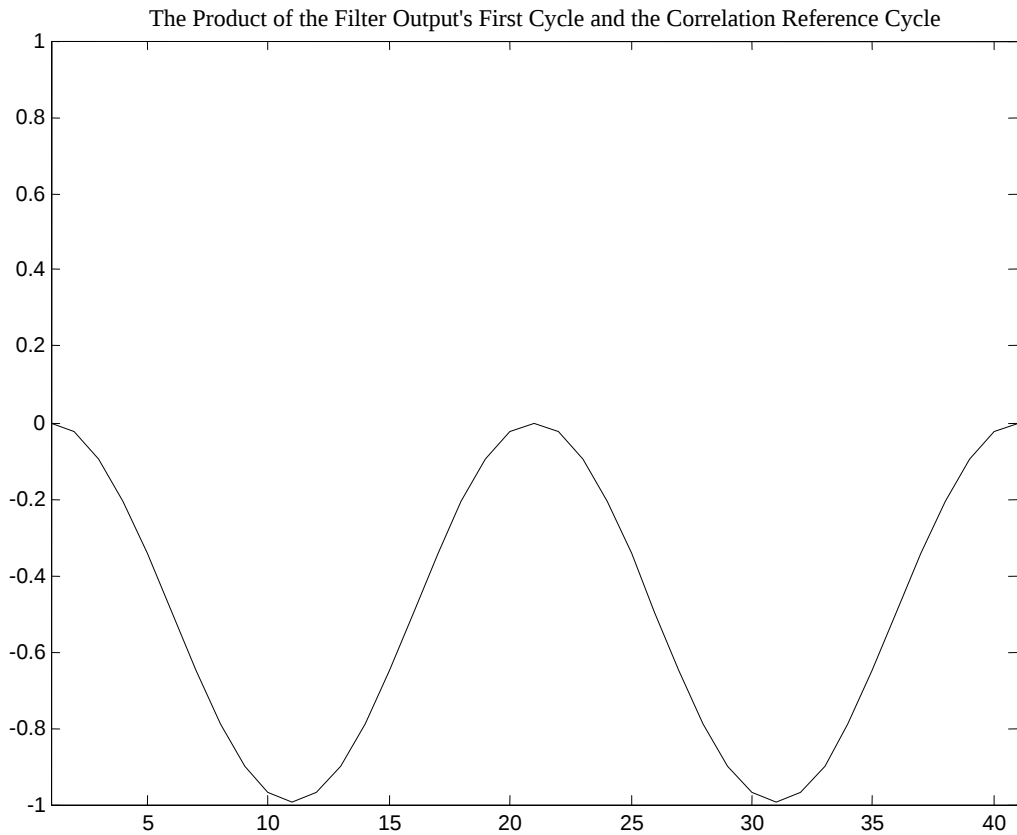


Figure 10: The product of the correlation reference cycle and the received BPSK signal's first cycle. Because the product waveform is negative, this indicates that the reference and received cycles were not the same, meaning that the first bit of the received sequence was a 0.

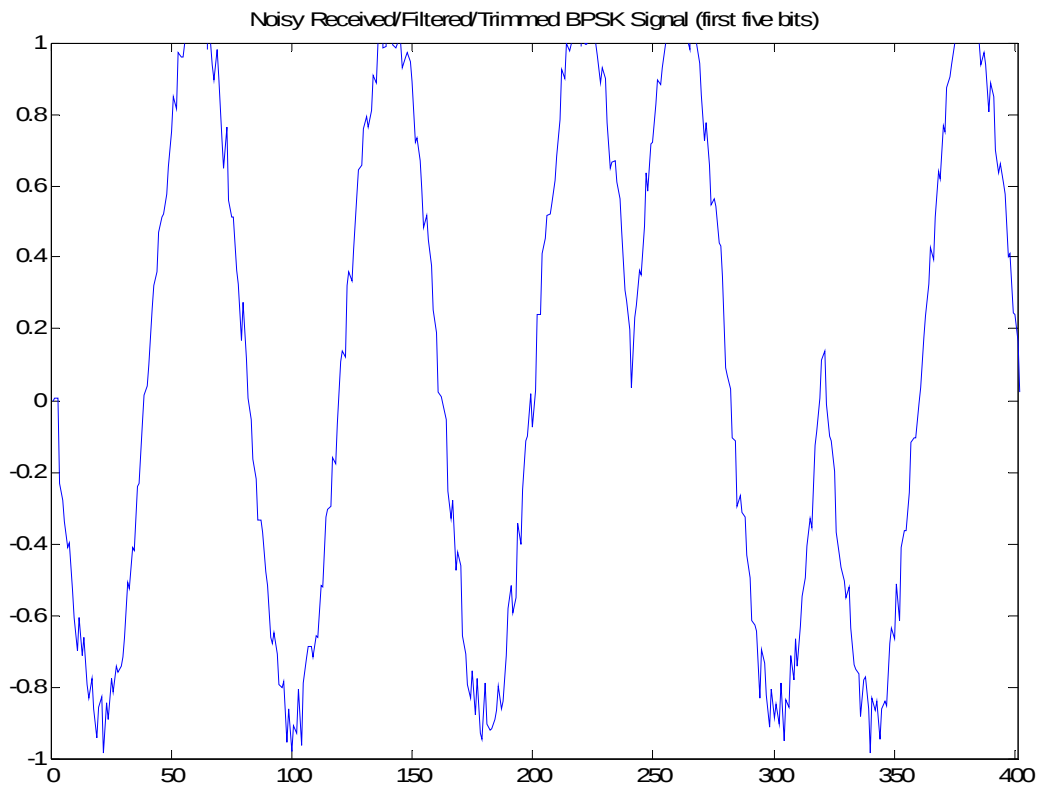


Figure 11: The simulation's correlation demodulation method is able to correctly identify the conveyed bits correctly even in the presence of noise.

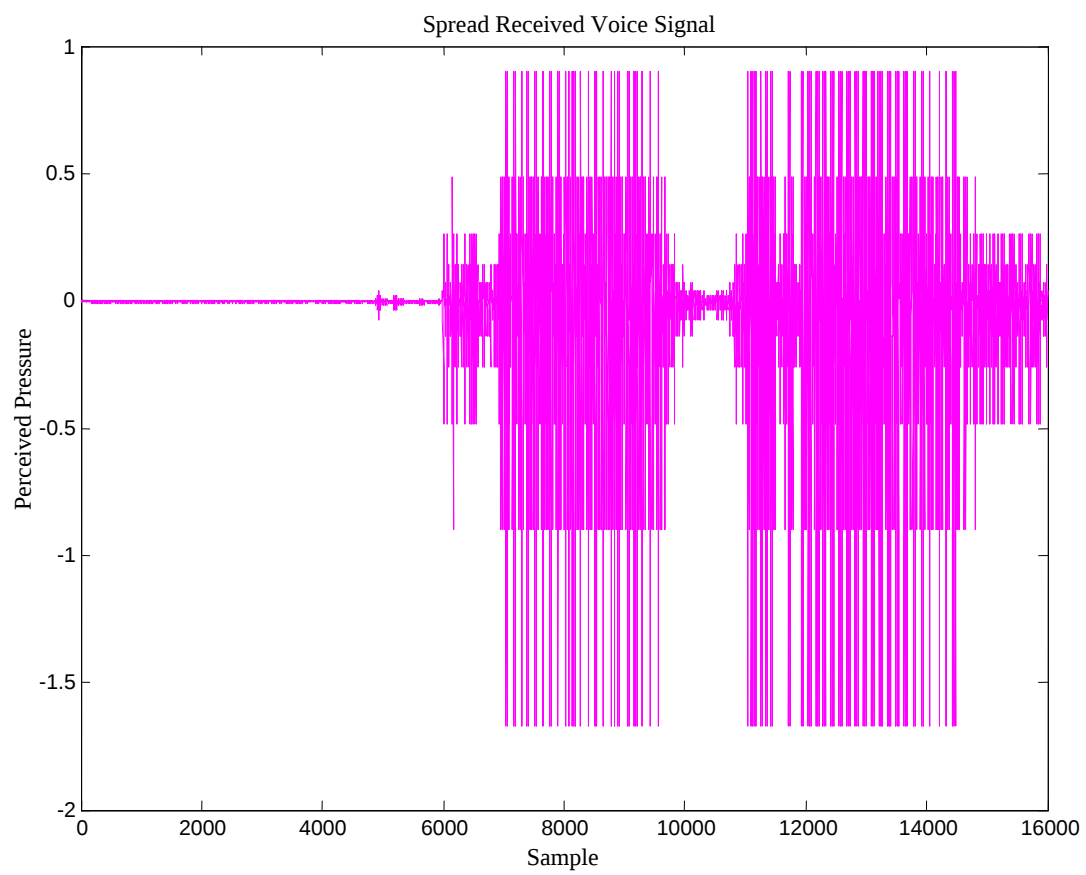


Figure 12: The received spread input signal.

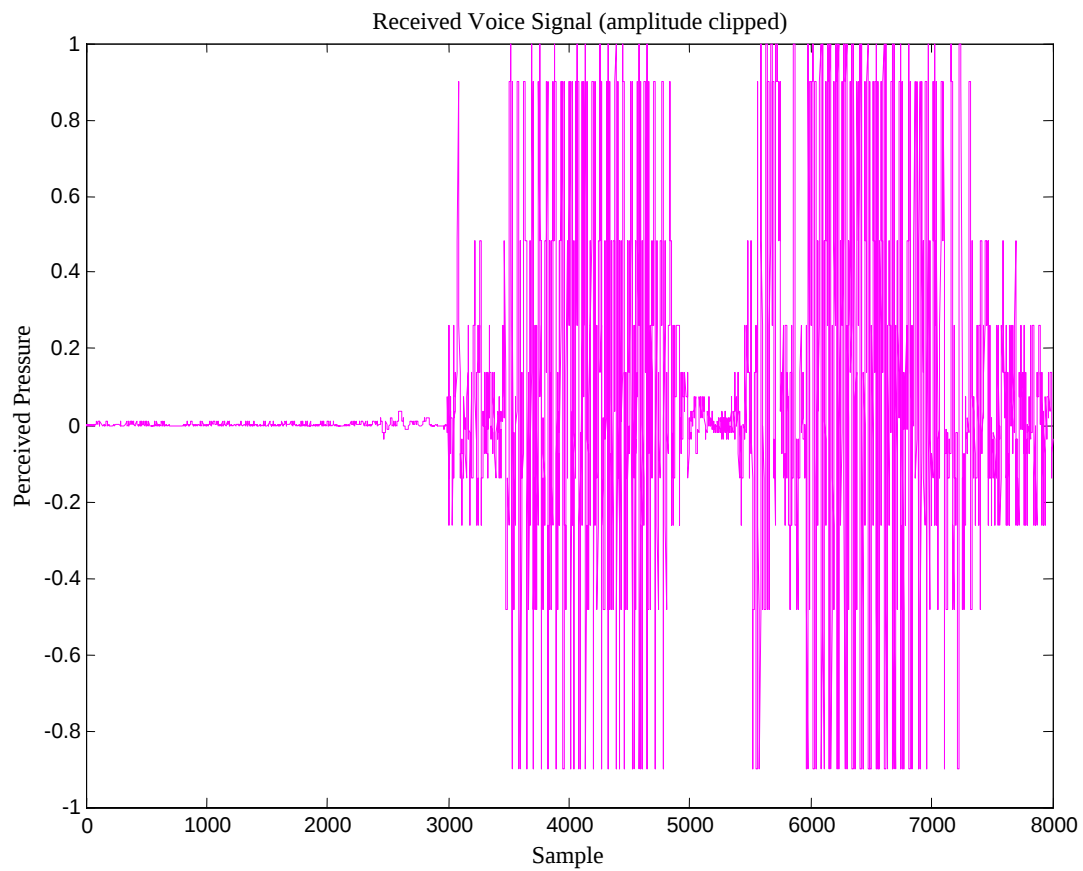


Figure 13: The recovered input signal. Amplitude values above 1 and below -1 are clipped. This waveform is the simulation's output, and the user is able to play back and save the audio.

Appendix D: 3SA Screenshots

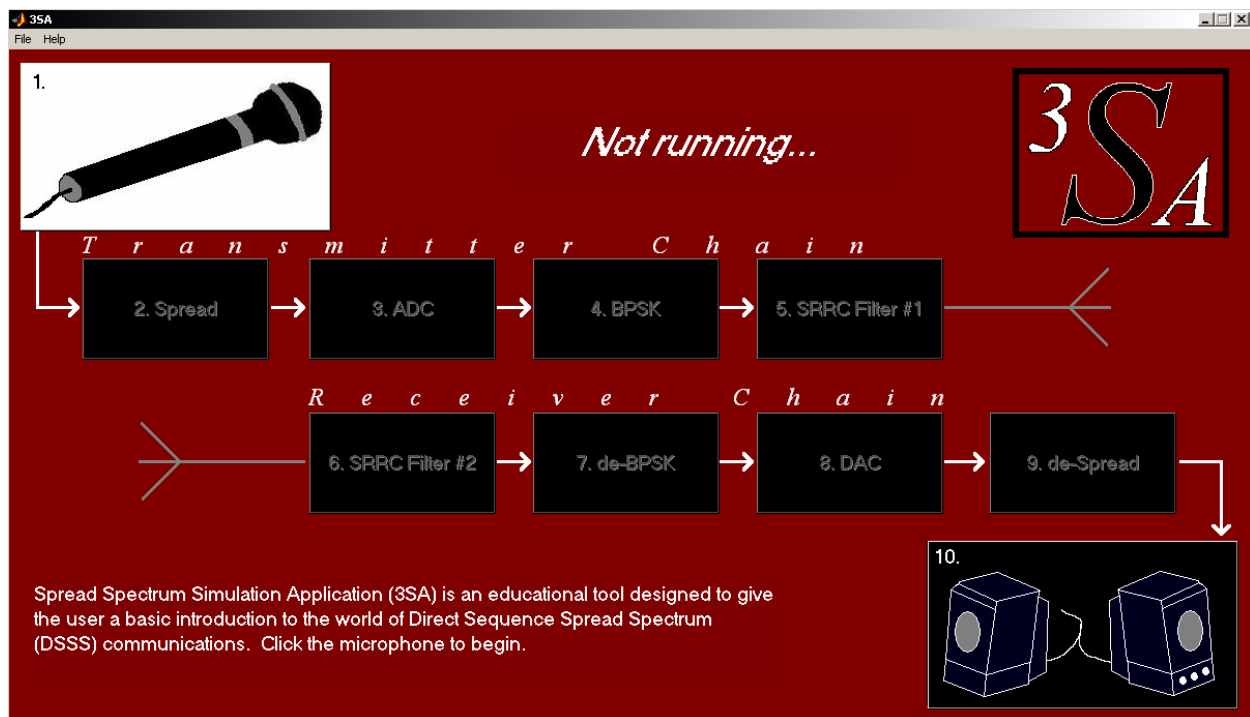


Figure 1: The 3SA base window. Only the microphone (input) pushbutton is enabled when the user first starts the application.

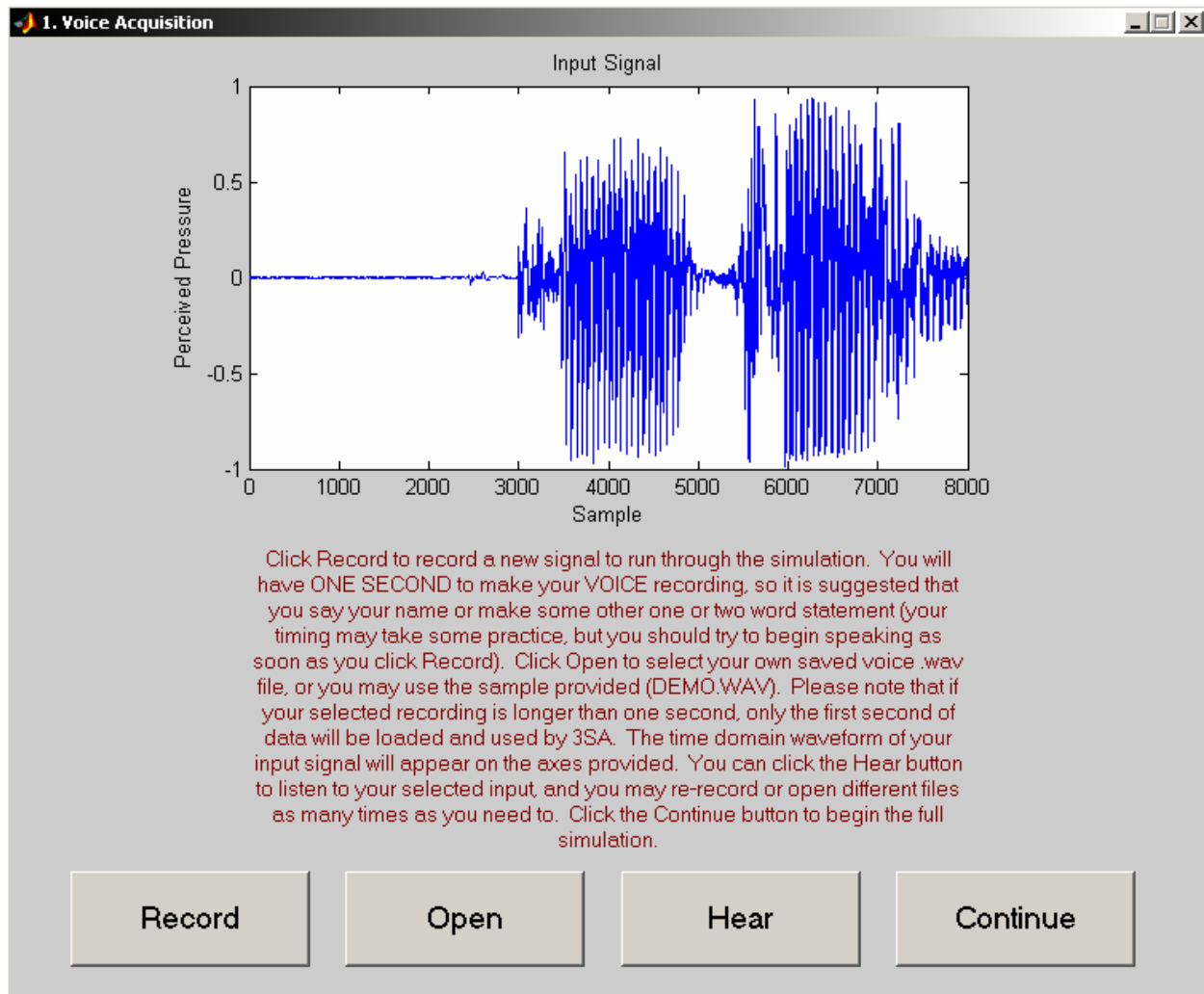


Figure 2: The input specification window allows the user to record a new input signal or select a prerecorded .wav file.

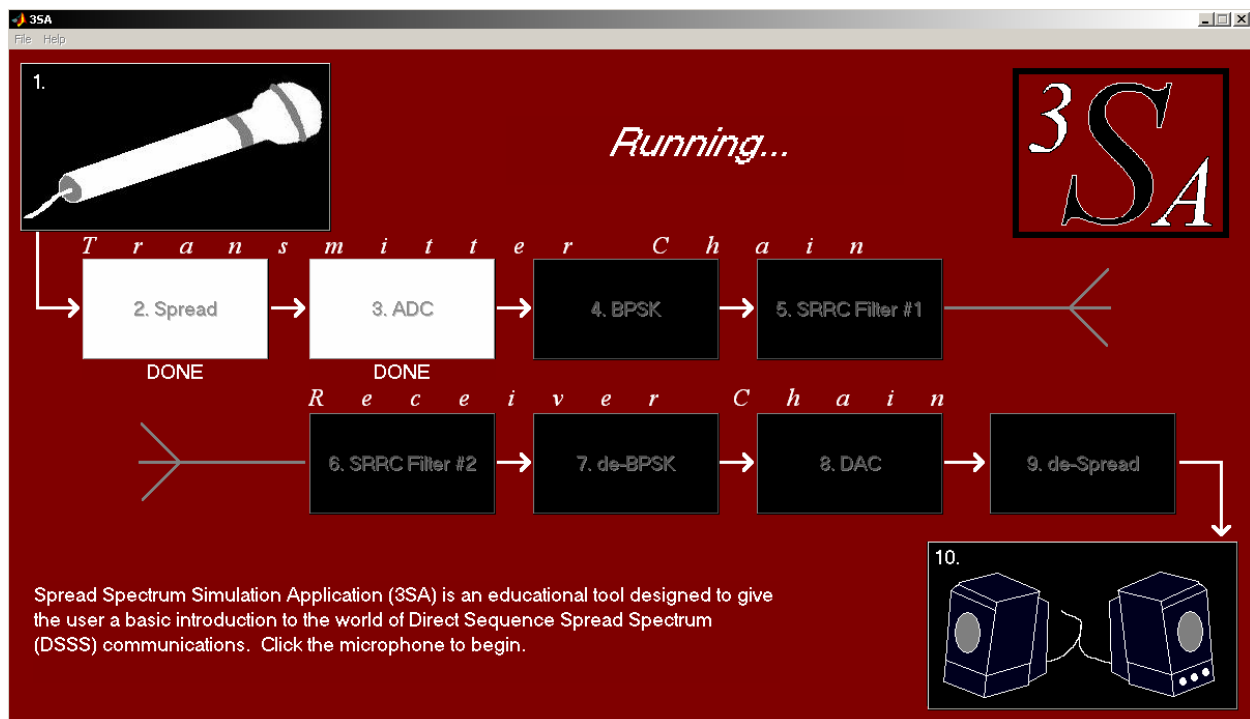


Figure 3: 3SA simulation in progress. Component pushbutton color changes from black (disabled) to white (enabled) as each component completes its portion of the simulation.

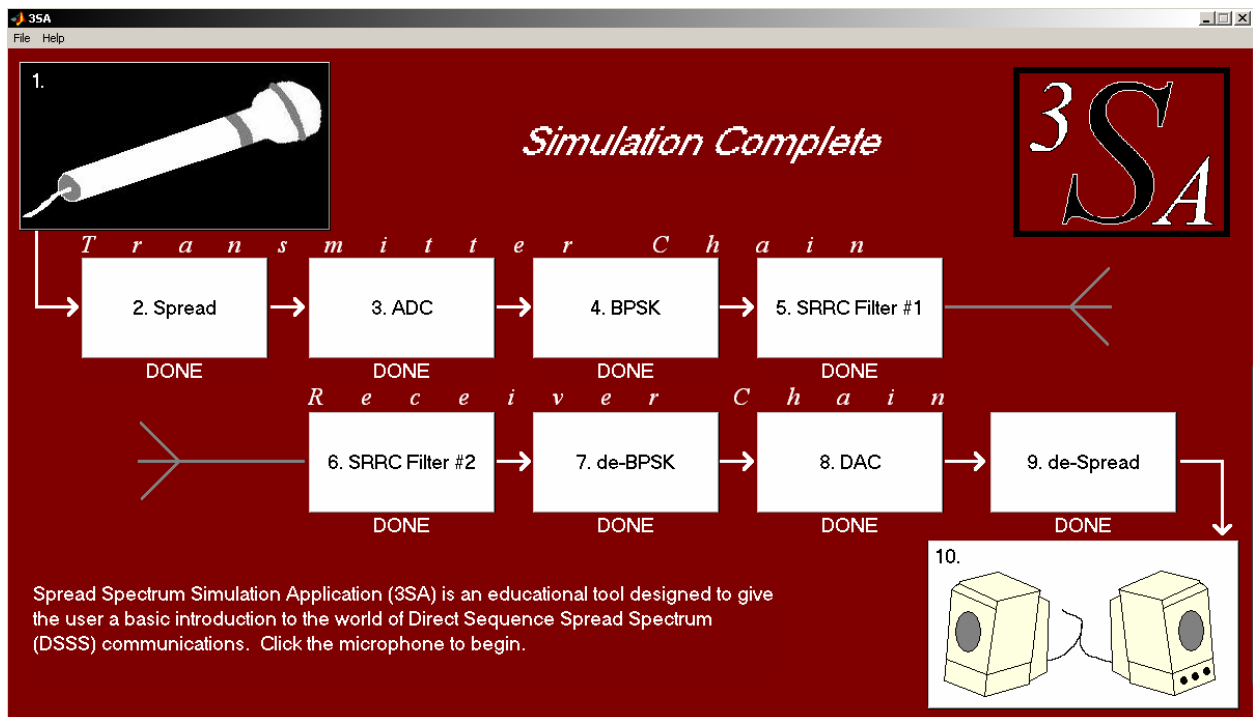


Figure 4: When the simulation is complete, the microphone (input) pushbutton is disabled and all other component pushbuttons are enabled. The user, however, can select “File, New” to start over.

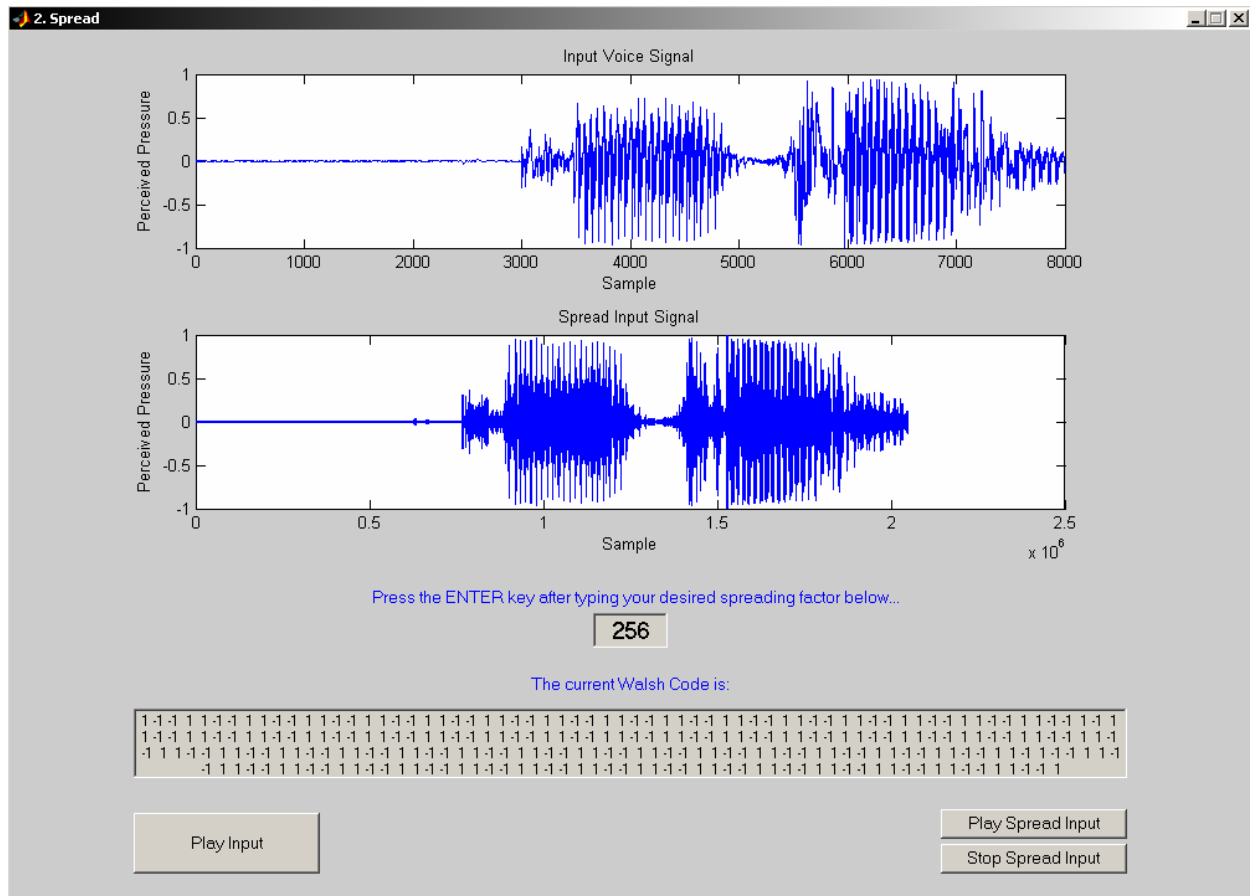


Figure 5: The interactive Spread component window allows the user to listen to their input, manually control the spreading, view the Walsh Code used to spread their input, and listen to the spread signal.

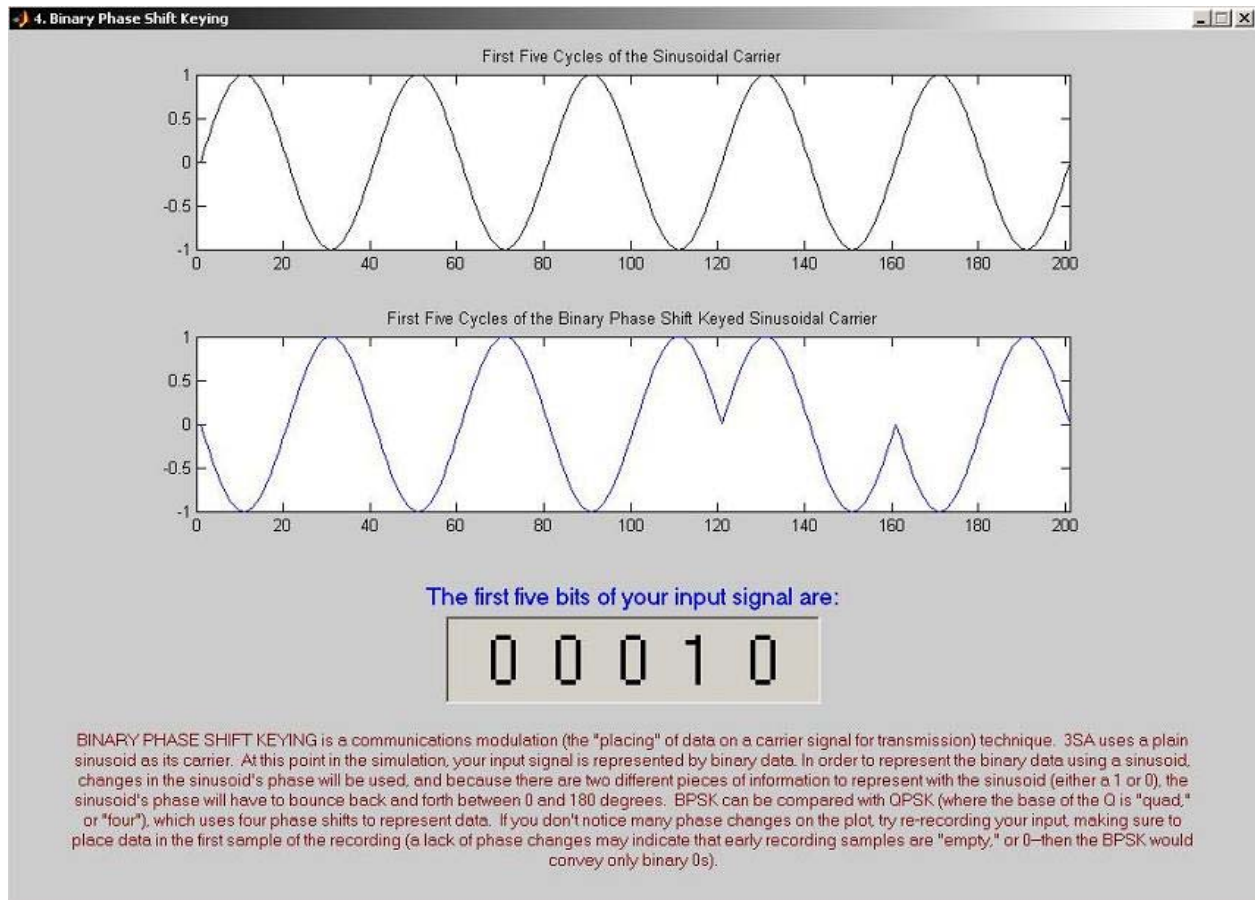


Figure 6: Others, such as the BPSK component window, allow the user to view waveforms and data conversion information.

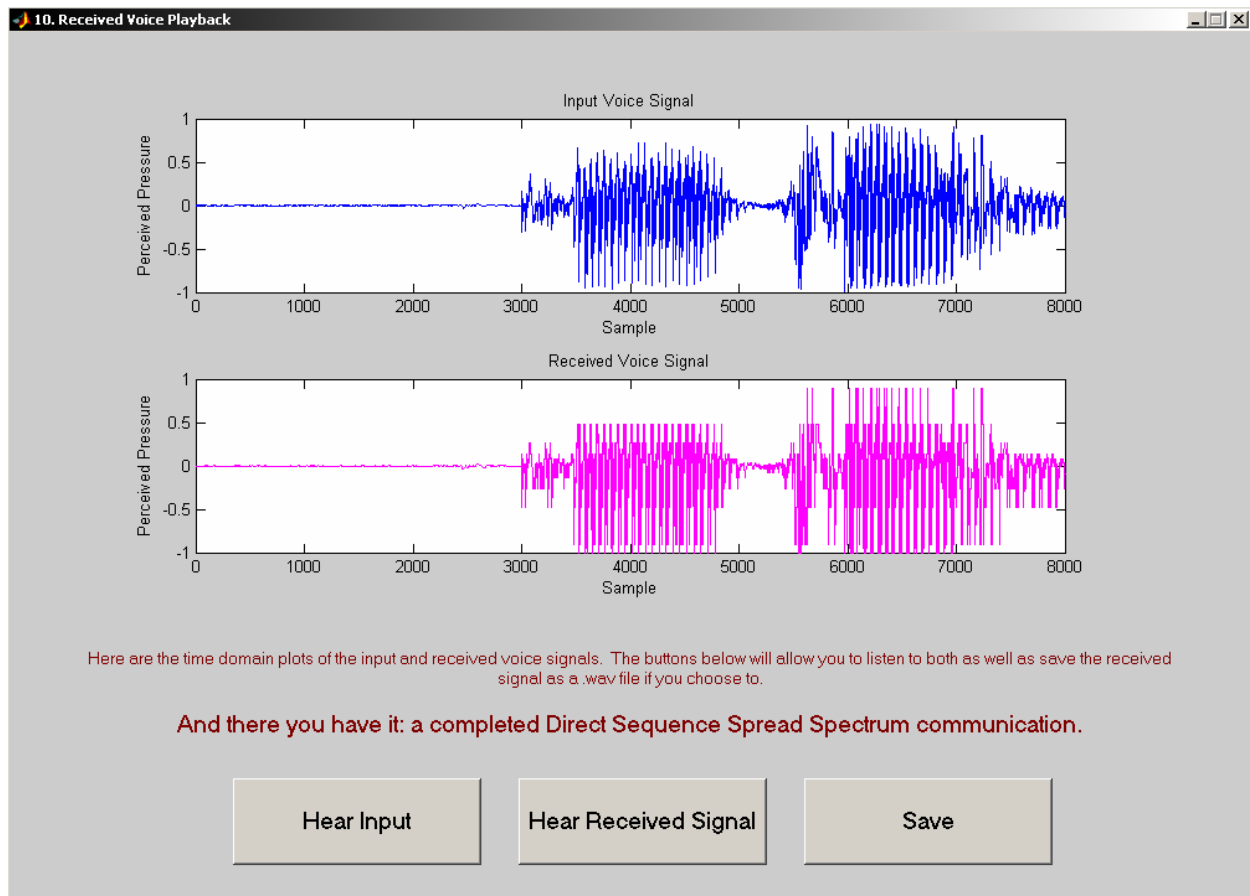


Figure 7: The playback window shows the user both the input and received voice waveforms and allows the user to save the received signal if they choose.

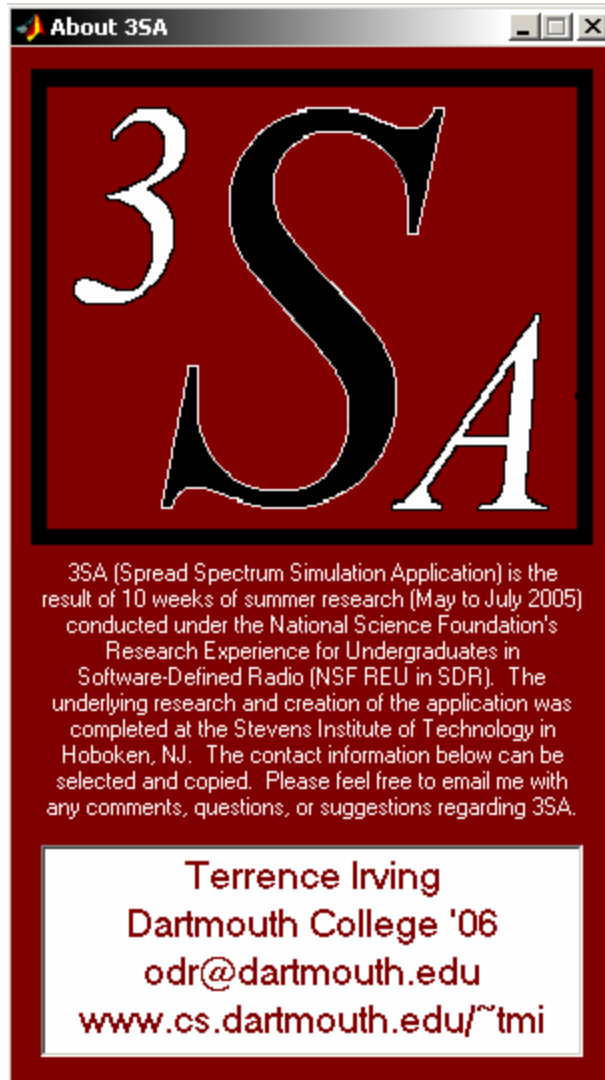


Figure 8: An “About 3SA” window is available under the “Help” menu bar option, complete with 3SA background and contact information.