ADC Final Project

Kim Winter, Lauren Pudvan, March Saper

Overview

Our final project for Introduction to Analog and Digital Communications employed 2 USRP radios and sent data from one to the other through Quadrature Amplitude Modulation. We were able to send bit strings back and forth, achieving a data rate of about 16kb/s. Our going beyond element was to understand and implement (7,4) Hamming codes. All of our code is here: https://github.com/kwinter213/ADC

System Overview

Sending

First we decided on two words to send and encoded these words into bits. The we put them through a (7,4) binary Hamming code and upsampled the resulting bit string to form boxes. Since the USRPs are capable of Quadrature Amplitude Modulation, we then wrote our signal complex vector of $\pm 1 \pm j$ and sent this to the USRP. The system diagram for this can be seen in Figure 1.

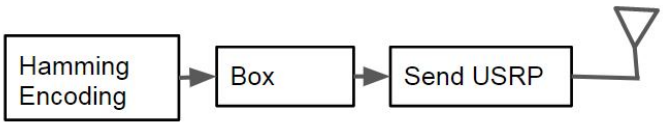

Figure 1: The block diagram for sending the data.

Receiving

Once the message was received we trimmed the file and adjusted for phase and frequency offset. Then, we unboxed the sent message, put it through Hamming decoding and found the error rate. The system diagram for this can be seen in Figure 2 on the next page.

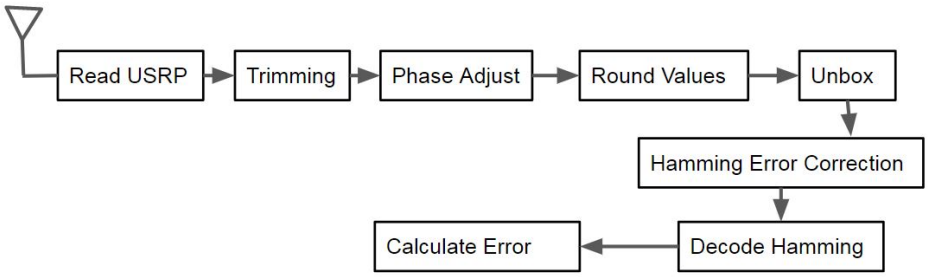

Figure 2: The block diagram of how we received data.

Trimming our Signal

For all of our transmissions, we sent the following stream of bits: 1000 1's, 1000 0's, a known header, the encoded message, 1000 0's, 1000 1's. An example of this is shown in Figure 3.

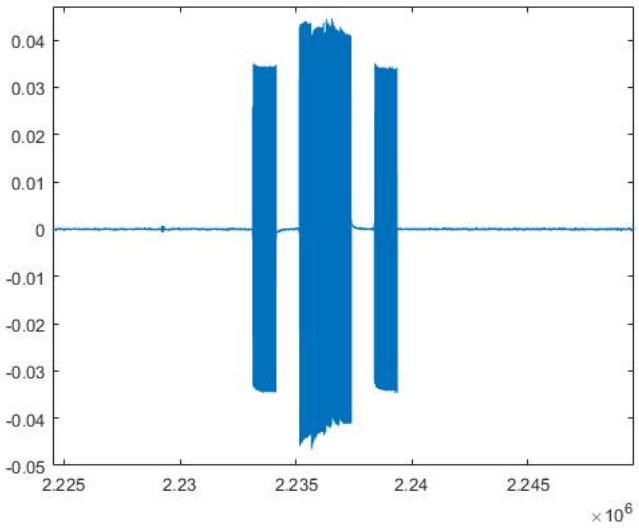

Figure 3: This is an untrimmed received signal.

To trim the signal we started by cross-correlating the known header with the received signal using the code that follows

```
[corrWithStarter, starterLag] = (xcorr(recievedFileToBeTrimmed,starter));
[~,I] = max(abs(corrWithStarter));
lagStart = starterLag(I) + 1;
```

Then we subtracted 300 from the index lagStart and continued looking forward for a large magnitude. This large magnitude is the start of the signal. We subtract the 300 because it corrects the common error where the cross-correlation is off by a little bit. This is possible because right before the header and message we send 1000 1000 0's. So, we back into the section of 0s and move forward until we find our message start. Our trimming function also takes in the expected length

of the message and trims the end by being that amount of points from the start. We tried using an ender message in a similar manner as the header, but had significant issues getting the ender to autocorrelate.

Header

We found we had the most success when we used a header that was not random. The real and imaginary parts of the header we employed are shown in Figure 4 and Figure 5.

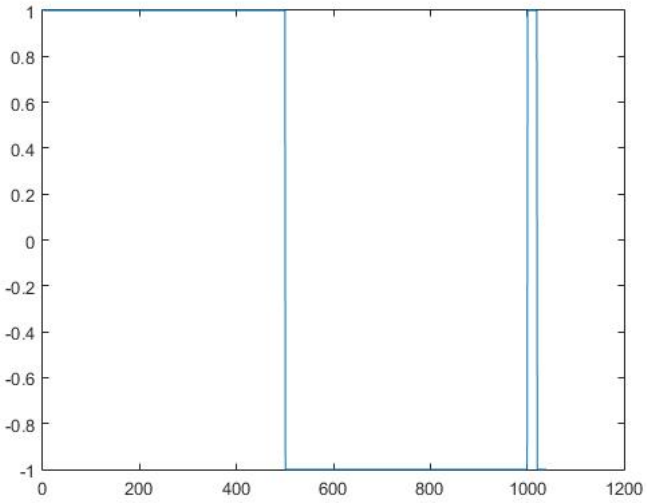

Figure 4: Imaginary part of our header

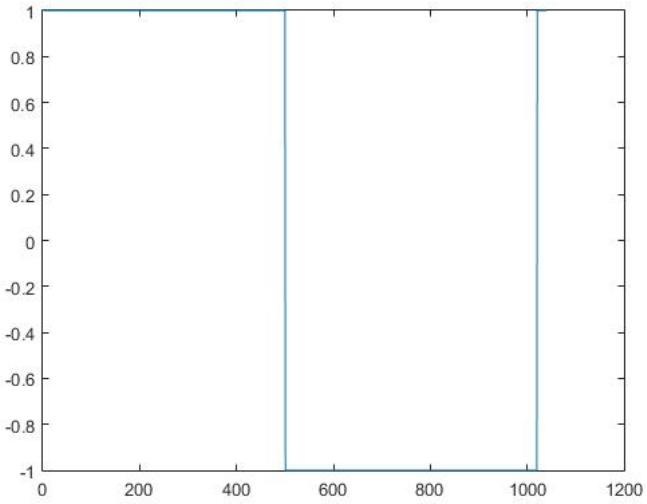

Figure 5: Real part of our header

Importantly, any header we used needed to end with at least 20 points which fell in the same quadrant in the complex plane. The reasoning behind this will be explained in the next section.

Phase Adjustment

USRPs are not perfect and there is a mismatch between the transmitting USRP's frequency and the receiving USRP's frequency. We adjusted for this offset by examining the received signal and assuming a Rayleigh Flat Fading channel model.

The model assumes that the bandwith of our signal is much smaller than the carrier frequency and that our received signal can be modelled as $y[k] \approx hx[k] + n[k]$ where h is a complex constant. If we rewrite our equation to include frequency and phase offset, we get $y[k] = he^{(2\pi f + \omega)j} + n[k]$. If we assume no noise and push the phase offset value to the constant h we can simplify our equation to $y[k] = h_{new}e^{2\pi fj}x[k]$,

If we remember that all values of $x[k]$ are $\pm 1 \pm j$, we can see that taking $y[k]$ to the fourth power will give something that is scaled only by the complex value h_{new} and the frequency offset: $y[k]^4 = e^{4\omega j}e^{4(2\pi f_d + \omega)j}$ In this equation h_{new} has been written in its complex exponential form. Plotting the Fourier transform of $y[k]^4$ gives a plot similar to Figure 6.

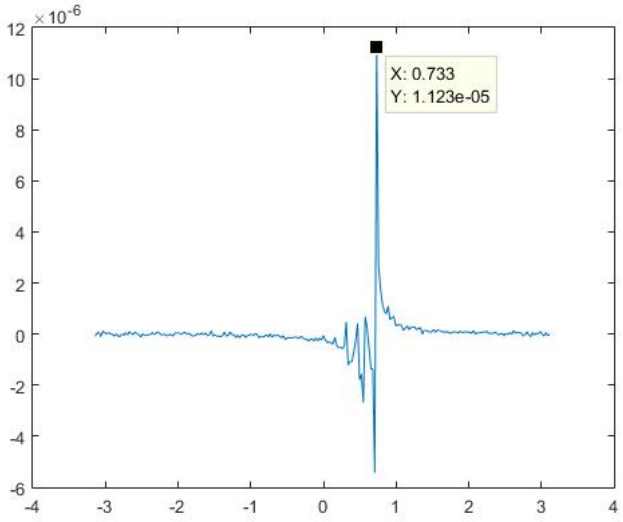

Figure 6: FFT of one recieved signal

Since we took our entire equation to the fourth power, we can figure out f and ω using the peak which we assume to be the impulse represented by

$$FFT(e^{4(fkj+\omega j)}) = (f - 4f_dkj + \omega j)$$

f is equal to the frequency location of the peak divided by 4. ω is equal to the phase of this peak divided by 4. Using this, we can apply an initial offset to all of our received points by dividing them by $e^{kfj} + \omega j$. Below, you can see this, where the red is our initial received, trimmed message, and the blue indicates the initial phase adjustment.

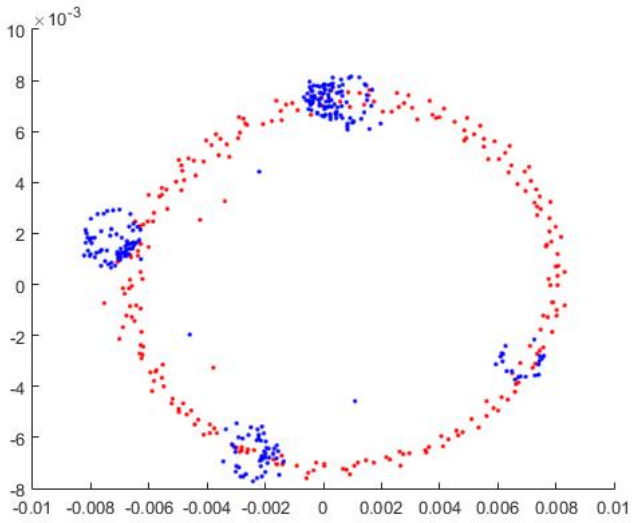

Figure 7: In red, one can see our original received message, having been trimmed, plotted in the real and imaginary axes. In blue, one can view our phase adjusted message.

These four blobs are a great sign of success in our system. Each blob represents one of four possible received signals, $\pm 1 \pm j$. The quadrant of the blob indicates its value. However, are all of these blobs in the correct quadrants? We wrote more code to find out!

Because we have a constant 20 bits at the end of our known header, we can compare phase of the received header points to the known phase of the final 20 header points (before they went through the channel). This allows us to calculate how much our constellation blobs should be rotated to put everything in the correct quadrant.

This was accomplished with the following code:

```
% correct the quadrant
% figure out where the last 20 values of the Header (the first 20 of
% tocorrect) should go
knownPhase = 0;
for i = 1:20
    kno = phase(knownHead(i));
    if kno < 0
        kno = 2*pi + kno;
    end
    knownPhase = knownPhase + kno;
end
knownPhase = knownPhase/20; % What the phase should be

offPhase=0;
for k=1:20
    off = phase(corrected(k));  % First 20 points of corrected correspond to final
        20 points of  header
    if off < 0
        off = 2*pi + off;
    end
    offPhase=offPhase+ off;
end

offPhase = offPhase/20; % what the phase is

```

```
phaseCorrection = knownPhase - offPhase;     % How much we need to correct


for k = 1:length(corrected)
    corrected(k) = corrected(k)*exp((phaseCorrection)*1j);    % adjust for  offset
end
end
```

Below, in green, you can see the final points from this process mapped to their proper quadrants.

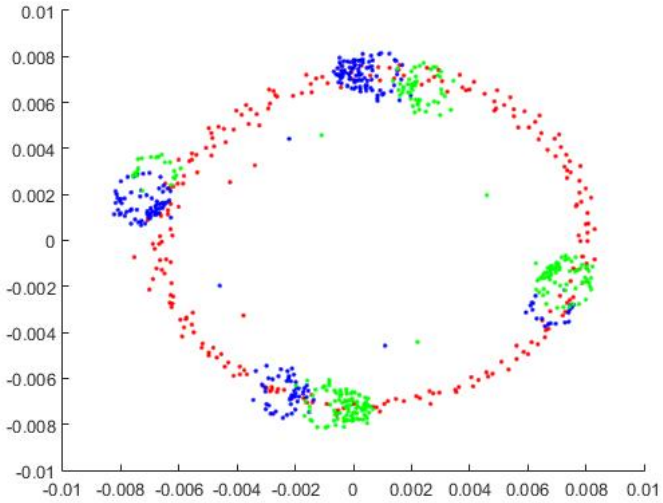

Figure 8: In red is our original received signal. In blue is our initially adjusted signal, and in green is our final values.

After this, we were able to take the sign of these newly adjust bits, in the real and imaginary plane separately, to determine what bits we had received.

Hamming

Our going beyond element was to learn about and implement Hamming codes in our system. Hamming codes are linear error control codes and come in several varieties. The Hamming code we chose to implement was the (7,4) Hamming code. This code sends 4 known bits and 3 parity bits for error correction if a mistake is made during transmission. Linear error control codes are very handy for implementations in systems such as ours because all encoding and decoding can be accomplished through matrix multiplication under mod(2).

Depending on how the parity check bits and message bits are arranged, different matrices (called Generator and Parity Check matrices respectively) are used. Hamming codes have a unique property that when a Parity Check matrix of increasing binary numbers is used and if only one error occurs in transmission, the location of the error can be directly found.

Hamming Encode

The following code snipped shows how we encoded our messages. Notice that because the Parity Check matrix of increasing binary numbers was used, we had to use a Generator matrix which

corresponded to it. Therefore, the Generator matrix is not in standard form.

```
function encodedMessage = hamming_encode(Inputmessage)
% Function generates encoded message using
% Hamming coding method. Message to be encoded. Message must multple of 4.
encodedMessage = [];
for i = 1:4:length(Inputmessage)

    message = Inputmessage(i:i+3); % encode this

if length(message) ~= 4
    disp('Input vector wrong length');
end

G = [1 1 1 0 0 0 0;
     1 0 0 1 1 0 0;
     0 1 0 1 0 1 0;
     1 1 0 1 0 0 1];

encoded = (message*G);
encoded = mod(encoded, 2);
disp(encoded);

encodedMessage = horzcat(encodedMessage, encoded);
end

end

end
```

Hamming Decode

This code snipped shows how we decoded our Hamming code words and corrected for up to 1 error per 7 bit vector. (Note, if more than one error occurred, the entire vector would be incorrectly decoded. (7,4) Hamming codes are implemented when it is assumed that the chance of error is unlikely enough that this restriction will not cause significant issues. Other error control codes exist for situations where large bursts of errors etc may occur etc.)

```
function decoded = hamming_decode(received)
% Function decodes Hamming encoded message using
% It takes an encoded row vector and returns decoded. Can correct up to 1 error per
% every 7 bits. To work input, vector must be multiple of 7.

received = sign(received+1) % Assumes input vector is + 1 - 1. Turns input vector
    into expected 1's and 0's.

decoded = [];

for i = 1:7:length(received)- 6    % Every 7 bits are evaluated to decode into 4
    bits
    todecode = received(i:i+6); % decode this section

if length(todecode) ~= 7
    disp("input vector wrong length");
end

wrongIndex = 0;
```

```
corrected = todecode;

% parity check matrix in order of increasing binary numbers
H = [0 0 0 1 1 1 1;
     0 1 1 0 0 1 1;
     1 0 1 0 1 0 1];

check = mod(todecode'*H',2)          % if 1 bit error, returns vector which matches
    column corresponding to incorrect location

wrongBit =bi2de(fliplr(check))       % convert binary number/column error to index

if wrongBit > 0
    corrected(wrongBit) = corrected(wrongBit) + 1;
    corrected = mod(corrected, 2);  % correct the error (if exists)

end

decoded = horzcat(decoded, [corrected(3), corrected(5), corrected(6), corrected(7)])
    ; % decoded message, removes parity bits

end


end
```

USRP Quirks

Much of our project was spent on trying to figure out how to get the USRPs to transmit and receive anything. We faced certain odd errors throughout this project due to the "quirks" of working with the USRP's. One such oddity is that we weren't quite receiving the entirety of our signal, because our signal was getting cut off at the beginning. After asking T.J. about this issue, he suggested that we add buffers before and after our signal, and that this quirk had been due to the USRP's themselves. So, we added 1000 1's and 1000 0's before the signal as well as 1000 0's and 1000 1's after the signal. This means that our final sent signal in the real plane looked something like this:

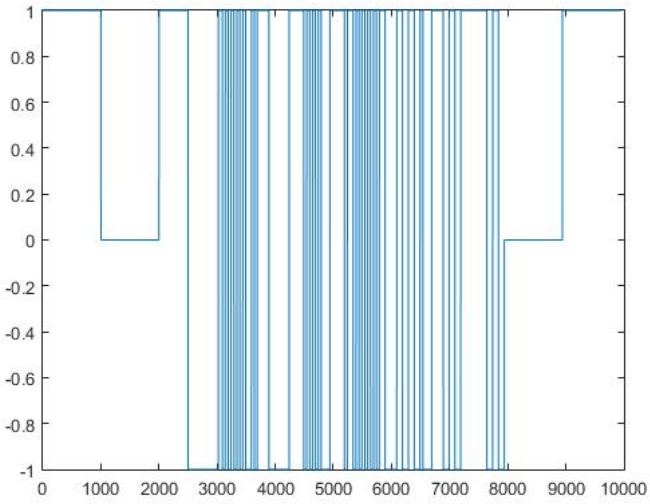

Figure 9: This is the actual contents of the binary file that was sent via USRP in the real plane. This includes 1000 1's, 1000 0's, a header, the message, 1000 0's, and finally 1000 1's.

This solved one of our issues, but for some reason introduced others. Frequently, he USRP would cut off transmission after only half the signal was sent. This happened so often that it made progressing through our project difficult.

Results

The following results represent our most successful transmission. Unfortunately this was also a transmission where the USRP only sent about 60 % of our message signal. Consequently, the error rate was severely affected.

We began our transmission by encoding two strings, "jazziest" and 'buzzword" into bit messages. Our intended transmission, with buffers, a header, and a message can be seen in Figure 10 on the following page.

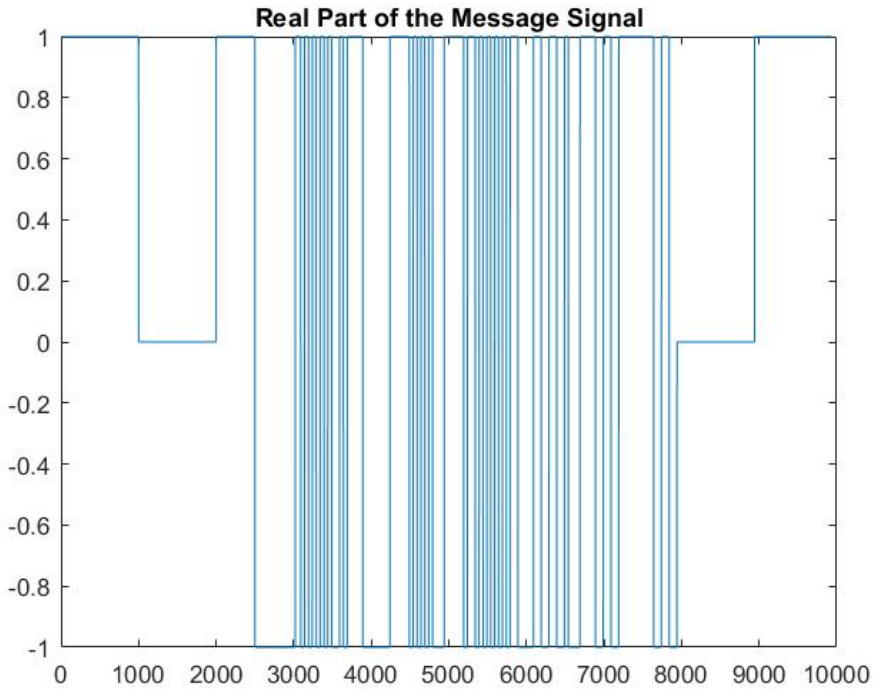

Figure 10: Message signal to send words “jazziest” and “buzzword”.

We wrote our message signal to the USRP and it was sent through Quadrature Amplitude Modulation to another radio. Figure 11 on the next page shows what we got on the other end. As you can see, about 40 % of our message was either cut off or never sent by the USRP. We were expecting an additional blob of 1000 1’s at the end of our received signal. Also, we expected the middle blob of our signal to be 4900 values long. The middle blob was only about 3000 bits long. This implies that there may have been an issue with transmitting or receiving over a laptop, potentially with an underflow or overflow of data that may have stopped our transmission.

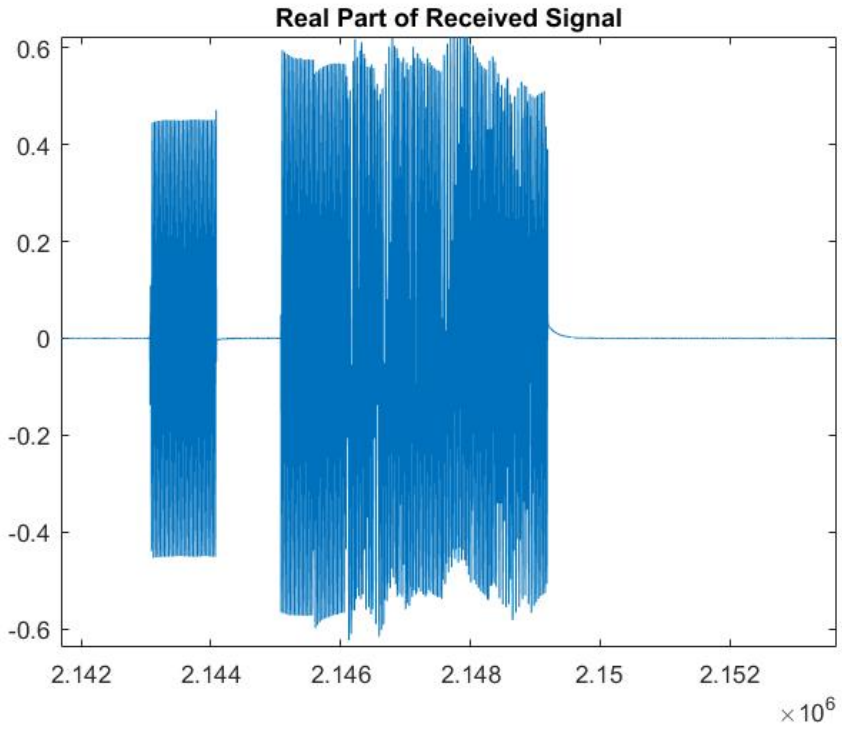

Figure 11: Real part of the received signal. Notice about half of it is missing.

Figure 12 shows the real part of our phase adjusted signal plotted on top of the original message. We feel that this demonstrates that our system generally worked and received a signal that was able to be phase-adjusted correctly, even if it did get cut off.

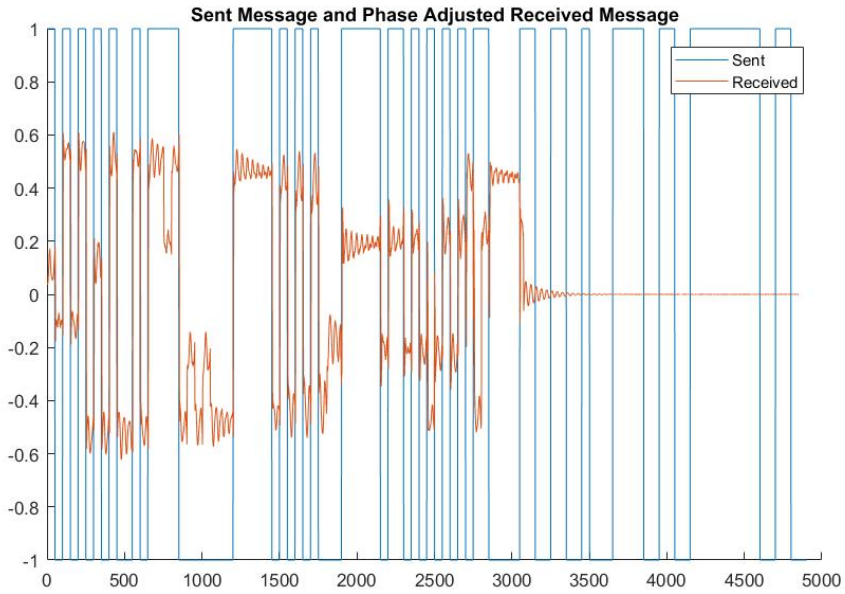

Figure 12: Real values of sent signal and phase adjusted signal before sign was taken.

After we downsampled the real and imaginary parts of our received signal, we found that the error rate of the first 60% of our received message (the part the USRP actuallyl sent) was around 15 %.

Finally, we decoded the Hamming codes and converted the resulting bit strings into characters, we received the following words: "jazzwK+c" and "bu {U". Considering the fact that we could only get 60% of the signal to send in the first place, we are especially proud of decoding "jazzwK+c" from the initial word "jazziest".

Future Work

If we were to do this again would have liked our code to be more modular. Many of our functions have weird restrictions and we would like it to become more universal. For example our message length needs to be divisible by 4 so the (7,4) Hamming function will work. This is very restrictive and it would be awesome to account for messages of different lengths! We would have liked to have an ender that works successfully. In the real world the receiver does not know the message length. Our starter works pretty well and we used the same process when we attempted the ender but it was always super unreliable. We would also like to get a better net data rate. Our USRP transmitting rate was 260e3 and at this rate (with Hamming) it is impossible to do any boxing and meet the minimum 64kb/s goal. Boxing by 10 really helped reduce error so we decided to keep our upsampling to small boxes. Finally, we would have liked to make our phase adjustment more robust. While the Raleigh Flat Fading model worked for the short messages we sent, we would liked to have implemented a phase lock loop or Costas loop to allow for longer messages and more robust phase adjustment.

Conclusion

We learned a lot about how application is much messier than theory. This is especially prevalent when trying to optimize an error rate and transmission rate. We are confident that with a little more time we could meet the net data rate of 64kb/s. It would take a lot of fine tuning to improve our error rate but we understand how to make our signal clearer. This project taught us a lot about how to actually use QAM instead of just understanding it as a concept!