EE360 - Digital Communication Systems
Final Project: Matlab Radio Receiver

David Tolsma

June 7, 2019

Abstract

This lab explores the implementation of a Mat-lab based software radio receiver. Given several different test vectors, we designed a robust radio receiver that could interpret the test vectors by doing carrier recovery, demodulation, baud timing recovery, adaptive channel equalization, and decoding. After designing and tuning the receiver, we tested the robustness of the system by attempting decoding of a previously unknown test vector to ensure that our receiver was general purpose.

Contents

1 Introduction

For our final project in EE360 we were tasked with creating an example software radio receiver and implement it in Mat-lab. This receiver needed to do all of the things that a real radio receiver does, but just in an easily accessible developing environment. A real life radio receiver needs to do many things in order to interpret the data that it is receiving.

The first is receiving the signal. An antenna (or a transmission line) collects energy from the environment and brings it into the radio receiver. This very weak signal is then amplified and band-pass filtered in order to remove some of the broadband noise that is present (this could happen in the opposite order as well). The signal is then passed to an automatic gain control system where the gain control effects the amplification in order to make the signal have an even amplitude over time. Since most radio communications is done at a very high frequency (often up to several Ghz), the signal needs to be down converted to an intermediate frequency before sampling. In our test situation, the signal was down converted to an intermediate frequency of 2 MHz. This down converted signal is then run into a sampler in order to take it from the analogue domain into the digital domain. The sample rate in this simulation was at 850 kHz.

Once sampled, this raw data is then fed into our Mat-lab radio receiver simulation. The next steps are carrier recovery, carrier demodulation, matched filtering, baud timing, channel equalization, soft decision making, and decoding. In Figure 1.1 we can see the steps (along with the adaptive parts to make each section work properly - highlighted in gray) needed to decode the input signal and to output the data.

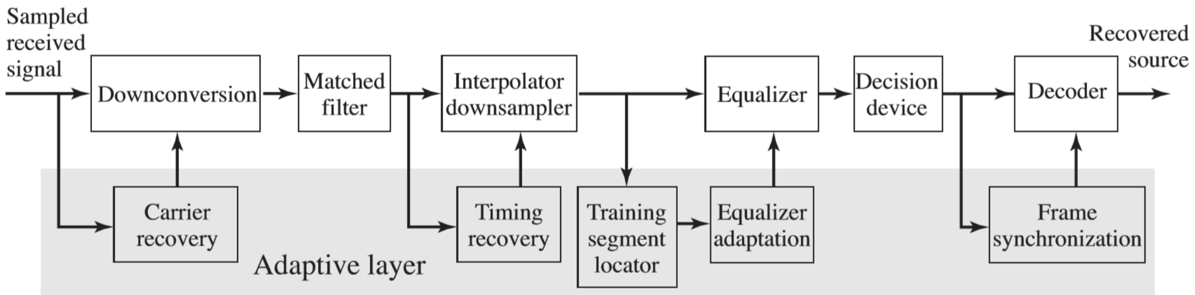

Figure 1.1: General data processing flow diagram inside the Mat-lab radio receiver

For each section, I quote the parts of the code as they become relevent to each section. The full uncut code can be found in Appendix A.

2 Methods and Discussion

In order to document how this radio receiver works, each section seen in Figure 1.1 will be broken down into its own subsection to describe each system in detail.

2.1 Pre-Processing and Adaptive Carrier Recovery

The data that is being streamed into the receiver is a base-band signal that was up-converted to some unknown high frequency, then down-converted to an intermediate frequency of 2 MHz. After down-converting the resulting signal was sampled at 850 kHz. Since 2 MHz is a much higher frequency than the Nyquist frequency corresponding to our sampling rate of 425 kHz, we cannot sample the intermediate frequency directly. When sampling, we create copies of the signal in the digital domain every 850 kHz centered around the intermediate frequency of 2 MHz. This means that we will see a copy of the up-converted base-band signal at 2 MHz, 1.15 MHz, 300 kHz, ... , etc. The 300 kHz signal is a low enough frequency that it falls underneath the Nyquist limit of our sampling rate, as 300 kHz < (850 kHz / 2).

In order to do down-conversion, we need to create a cosine wave that exactly matches the frequency and phase offset that exists in the transmitter. If we are unable to match both the frequency and phase we will be unable to properly down-convert the signal to base-band and will instead not be at base-band if the frequency is wrong, or be heavily attenuated if the phase is incorrect. This receiver uses a dual Phase Locked Loop (PLL) to do carrier recovery. As there is no energy at the actual carrier frequency, we can get around this by squaring the signal to give us energy at $2 * f_c$, where f_c is the intermediate carrier frequency of 300 kHz, with double the carrier phase offset. After squaring the input vector and creating a 600 kHz version of the carrier we band-pass the result to isolate the squared carrier. We can see in Listing 1 that the necessary code to create a band pass filter using Mat-labs built in firpm() function has been commented out on lines 3-6. This is because after extensive testing, it was realized that the ripple of the firpm filter produced with this method was making very accurate carrier recovery impossible. This was later replaced with a 6th order IIR Butterwoth filter created on line 7. A Butterworth filter much less stop band ripple, and allows the carrier recovery algorithm to be more accurate. The phase and frequency response curves of each of these filters are seen in Figure 2.1 and Figure 2.2.

```
1  %Carrier Preprocessing
2      ppSquaredR = r.^2;
3      %ppFilterOrder = 500;
4      %ppFilterFreq = [0 .56 .58 .61 .63 1];
5      %ppFilterMags = [0 0 1 1 0 0];
6      %ppFilterB = firpm(ppFilterOrder, ppFilterFreq, ppFilterMags);
7      [ppFilterB, ppFilterA] = butter(6,[.58,.61]);
8      ppRPreProcess = filter(ppFilterB, ppFilterA, ppSquaredR);
9
10 %Carrier Frequency Recovery
11     ppFFTrPreProcess = fft(ppRPreProcess);
12     [ppM, ppImax] = max(abs(ppFFTrPreProcess(1:floor(end/2))));
13     ppSsf = (0:length(ppRPreProcess))/(Ts*length(ppRPreProcess));
14     ppFreqS = ppSsf(ppImax);
15     ppPhaseP=angle(ppFFTrPreProcess(ppImax));
16     [ppIR, ppF] = freqz(ppFilterB,1,length(ppFFTrPreProcess), sampleFreq);
17     [ppMi, ppIm] = min(abs(ppF - ppFreqS));
18     ppPhaseBPF = angle(ppIR(ppIm));
19     ppPhaseS = mod(ppPhaseP - ppPhaseBPF, pi);
```

Listing 1: Carrier Pre-Processing and Carrier Recovery

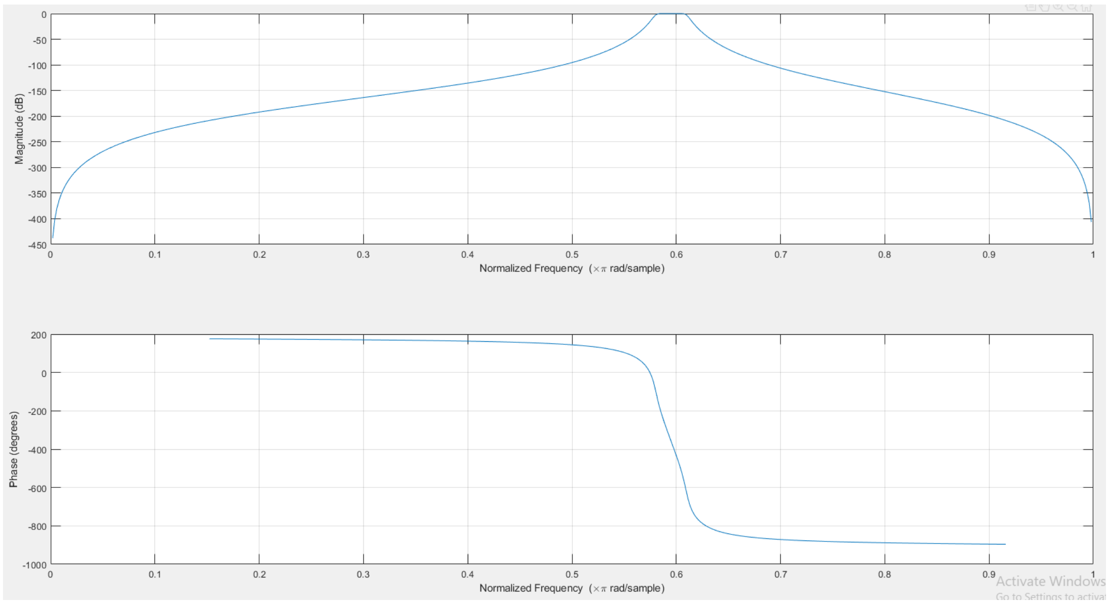

Figure 2.1: Frequency and Phase response of 6th order IIR Butterworth Band Pass Filter

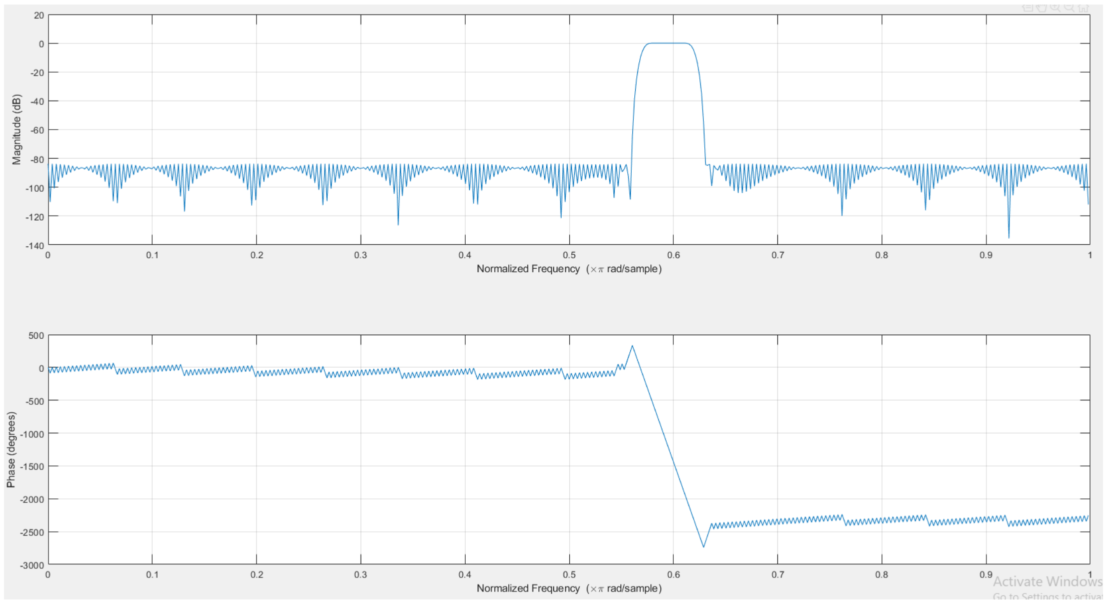

Figure 2.2: Frequency and Phase response of 500th order FIR Band Pass Filter

The cutoff frequencies for these band pass filters were chosen to isolate the squared carrier frequency. Since the squared carrier frequency is 300kHz $*2 =$ 600kHz which is above the Nyquist rate of 425 kHz, we cannot represent this 600 kHz wave with a sample rate of 850 kHz. Since this squared carrier is higher than the Nyquist rate of the sampler, the squared carrier gets shifted to the 'virtual' frequency corresponding to 600 kHz. This frequency is at $425kHz - (600\text{kHz} - 425\text{kHz}) = 250\text{kHz}$. This is the frequency we are going to see our squared carrier at. This gives us the location of our squared carrier at a normalized frequency of 250kHz/425kHz $= 0.588$. I chose cutoffs that were slightly above and below this normalized frequency. In the case of the Butterworth filter seen in Listing 1 I chose cutoffs of 0.58 and 0.61.

We then use an FFT to find the largest frequency component in this pre-processed signal and extract the phase and frequency information of the squared carrier. This information should be at 2x the frequency and 2x the phase offset of the actual carrier.

2.2 Dual-PLL and Demodulation

After doing carrier recovery, I implemented a Dual Pll system to use the pre-processed carrier information to demodulate the signal. We can see the code used to do this in Listing 2. The PLL that is to track the frequency of the carrier wave was given a step size value of 0.03, and the PLL that is to track the phase of the carrier was given a step size of 0.00003. The step-size of the first PLL needs to be quite large to be able to track a large frequency offset, but only large enough to track the maximum frequency offset. The step-size of the second PLL can be much larger, as when it converges to a phase offset it doesn't need to track. It needs to be large enough to converge in a reasonable amount of time, but as small as possible to reduce PLL jitter. This system was tuned as much as possible, as if this is not perfect the next steps in decoding will not be nearly as effective.

```
%Dual PLLs
    pllMu1= 0.03; pllMu2 = 0.00003;                  % algorithm stepsizes
    pllF0=300000;                            % assumed freq at receiver
    pllT = 0:Ts:length(r)*Ts-Ts;

    pllTh1 = zeros(1, length(r));
    pllTh2 = zeros(1, length(r));
    pllTh2(1) = 0.05;
    for k=1:length(r)-1                      % combine top PLL th1
      pllTh1(k+1)=pllTh1(k)-pllMu1* ...
      ppRPreProcess(k)*sin(4*pi*pllF0*pllT(k)+2*pllTh1(k)+ppPhaseBPF);
      pllTh2(k+1)=pllTh2(k)-pllMu2* ...
      ppRPreProcess(k)*sin(4*pi*pllF0*pllT(k)+2*pllTh1(k)+2*pllTh2(k)+ppPhaseBPF);
    end
```

```

%Demodulation
    demodF0=300000;                              % assumed freq at receiver

    demodT = 0:Ts:length(r)*Ts - Ts;
    demodCos = cos(2*pi*demodF0*demodT + pllTh1 + pllTh2);
    demodR = demodCos.*r';

    demodFl=250; demodFf=[0 .25 .27 1]; demodFa=[1 1 0 0];
    demodH=firpm(demodFl,demodFf,demodFa);                  % LPF design
    %demodFilteredR = filter(demodH, 1, demodR);
    [demodFilterB, demodFilterA] = butter(6,.25,'low');
    demodFilteredR = filter(demodFilterB, demodFilterA, demodR);
```

Listing 2: Dual PLL and demodulation of input signal

We can see in Figure 2.3 that when running with the 'medium.mat' test vector the Dual PLL converges quite quickly and has very little jitter. This high quality demodulation allows the rest of the system to work properly. We can see that Theta 1 converges to a sloped line indicating good frequency tracking, and Theta 2 converges to a flat line indicating a good phase offset lock. The demodulation is performed by creating a cosine with a frequency of the assumed carrier frequency (300 kHz) and adding the Theta 1 and Theta 2 phase offsets to account for the carrier frequency off set and phase offset. This results in a perfectly demodulated input signal.

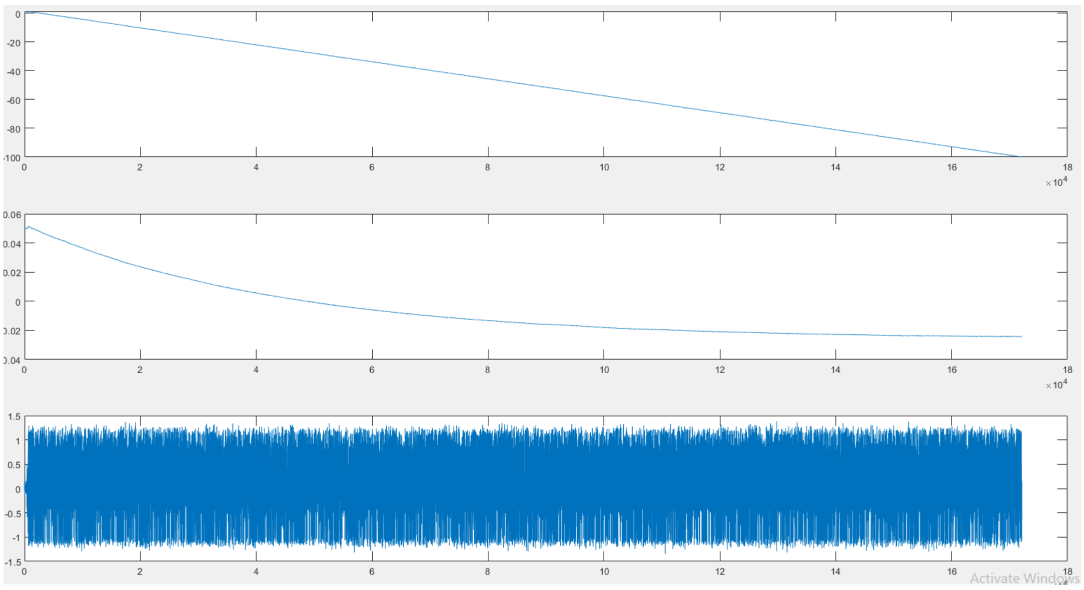

Figure 2.3: Chart of Theta 1 (top) and Theta 2 (middle) when Rx.m is run with the 'medium.mat' test vector. The demodulated input signal (bottom) is shown.

2.3 Matched Filtering

In order to reduce overall bandwidth usage, the transmitter used an SRRC pulse shape to do pulse shaping before it did up-conversion. Now that the receiver has done down-conversion it needs to do matched filtering in order to reduce overall signal noise, and to remove the inter-symbol interference from the pulse shaping. This needs to be done as an SSRC pulse is not a Nyquist pulse and therefore causes some inter-symbol interference. When the received signal is run through a matched SSRC pulse shaped filter the combined response creates a Nyquist pulse which does not exhibit inter-symbol interference. Going through the specifications of the transmitter laid out in Chapter 15 we can see that the SRRC pulse is 8 clock periods wide, has a roll-off factor given in the test signal, and has an oversampling factor of (Symbol Period/Time between Samples) which is equal to 6.4uS/(1/850kHz). In the case with the 'medium.mat' test vector, we had a roll-off factor of 0.3. The code to produce the SRRC matched filter and to perform the filtering is given in Listing 3

```
%Pulse Shape Matched Filtering
```

```
    psmfPulseShape = srrc(srrcWidth/2, rolloff, (symbolPeriod/(Ts)));
    psmfRFiltered = conv(demodFilteredR, psmfPulseShape);
    if(ssrcDebug)
        figure('Name', 'Output of Pulse Shape Matched Filtering');
        plot(psmfRFiltered);
    end
```

Listing 3: Matched filter creation and matched filtering

We can see the shape of this matched filter, seen in Figure 2.4 is similar to a sinc pulse, but is more attenuated at the edges and is truncated.

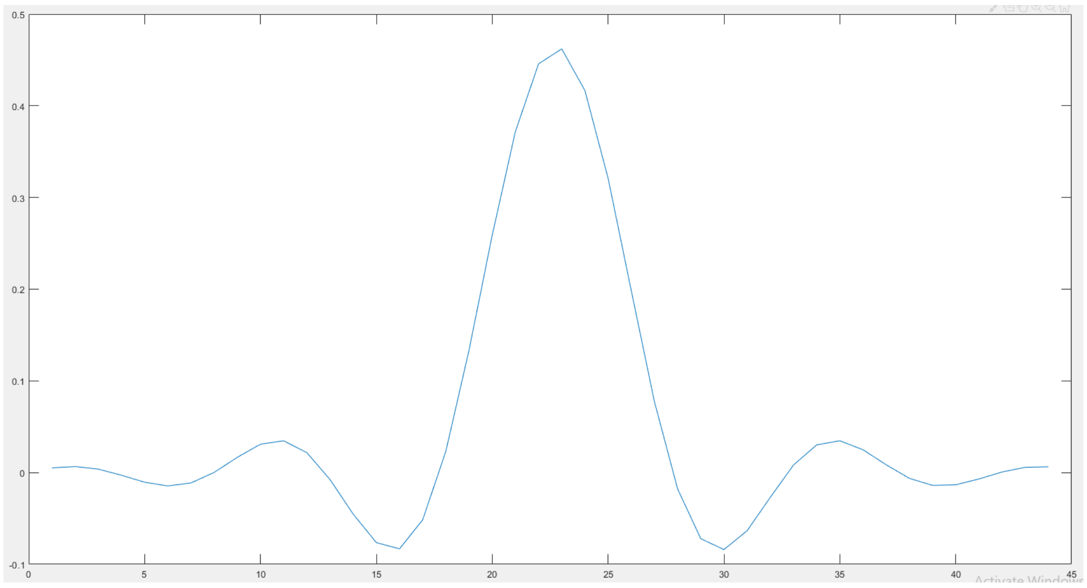

Figure 2.4: SRRC matched filter pulse shape

2.4 Interpolate Down-sampler (Baud Timing Recovery)

After we have demodulated and performed matched filtering, we need to find the right places to take samples. If our baud timing is incorrect, then our samples will not represent what was trying to be sent, but instead will represent the bit being sent plus some scaled combination of the bits around it. If we cannot accurately recover baud timing we will experience significant inter-symbol interference and we will encounter significant issues when we move to quantizing and decoding the data.

Since our data has a frequency content far below the Nyquist frequency of our sampling rate we can use sinc interpolation to recover exactly what the values are at the time between our samples. This allows us to get a more accurate representation of the sent symbols even if we did not take a sample at the exact time the symbol was sent. Where we take this interpolated sample can then move forward and back in time in order to grab exactly where the symbol was sent. There are several algorithms to recover baud timing, which each have their own pros and cons. The two commonly used in this project were gradient decent algorithms, one based on decision direction (where the grabbed symbols are quantized and their offset between themselves and their quantized counter parts are used as the cost) and another based on maximizing output power. The decision directed algorithm has false minima that make it unsuitable to find the absolute baud timing, but has a stronger timing lock. The output power algorithm has no false maxima, but is more jittery than decision direction once locked.

This led me to using a mix of both algorithms, each with their own step size (and therefore weight) to get the best of both worlds. The code for this system can be seen in Listing 4 I chose the output power algorithm to be dominant to ensure that we arrived at a correct baud timing offset, but gave decision direction enough of a weight to stabilize the overall algorithm. I chose a stepsize of 0.004 for the decision direction algorithm, a stepsize of 0.09 for the output power algorithm, and a common delta of 0.46 for both algorithms.

```
%Interpolator Downsampler
```

```
    IDtnow = (srrcWidth/2)*overSampleFactor+1;
    IDtau = 0.3;
    IDxs = zeros(1,length(r));
    IDtauSave = zeros(1, length(r));
    IDi = 0;
    IDmuDD = 0.004;
    IDmuOP = 0.09;
    IDΔ = .46;

    while IDtnow < length(r)- 2*(srrcWidth/2)*overSampleFactor
        IDi = IDi + 1;
        IDxs(IDi) = interpsinc(psmfRFiltered, IDtnow + IDtau, (srrcWidth/2));
        IDx_Δp=interpsinc(psmfRFiltered,IDtnow+IDtau+IDΔ,(srrcWidth/2)); % value to right
        IDx_Δm=interpsinc(psmfRFiltered,IDtnow+IDtau-IDΔ,(srrcWidth/2)); % value to left

        IDdx = IDx_Δp-IDx_Δm;
        %IDtau = IDtau + IDmu*IDdx*IDxs(IDi);

        IDdx=IDx_Δp-IDx_Δm;              % numerical derivative
        IDqx=quantalph(IDxs(IDi),[-3,-1,1,3]);  % quantize to alphabet
        %IDtau=IDtau-IDmu*IDdx*(IDqx-IDxs(IDi));         % alg update: DD

        IDtau=IDtau - IDmuDD*IDdx*(IDqx-IDxs(IDi)) + IDmuOP*IDdx*IDxs(IDi);

        IDtnow=IDtnow+overSampleFactor;
        IDtausave(IDi)=IDtau;      % save for plotting
    end

    if(IDdebug)
        figure('Name', 'Plot of DD Interpolator')
        subplot(2,1,1), plot(IDxs(1:IDi-2),'b.')        % plot constellation diagram
        title('constellation diagram');
        ylabel('estimated symbol values')
        subplot(2,1,2), plot(IDtausave(1:IDi-2))        % plot trajectory of tau
        ylabel('offset estimates'), xlabel('iterations')
    end
```

Listing 4: Interpolator Down-sampler (Baud Timing Recovory)

We can see the output of the interpolator down sampler in Figure 2.5. We can see that the timing offset has converged to a sloped line. This is what we expect as the timing offset is fixed over a single frame as described in the Chapter 15 specifications. Similar to the sloped line in the PLL that accounts for the frequency offset in the carrier recovery system, this system also converges to a sloped line to account for the baud timing frequency offset. This sloped line however is quite jittery as the output power algorithm is relatively unstable due to the output power being different from different symbol amplitudes. This is helped by the decision direction algorithm, but I could not increase the decision direction algorithm weight much before we started to converge to false minima. We can see that the constellation pattern is very clear for 'medium.mat' and is clearly converging to the correct baud timing.

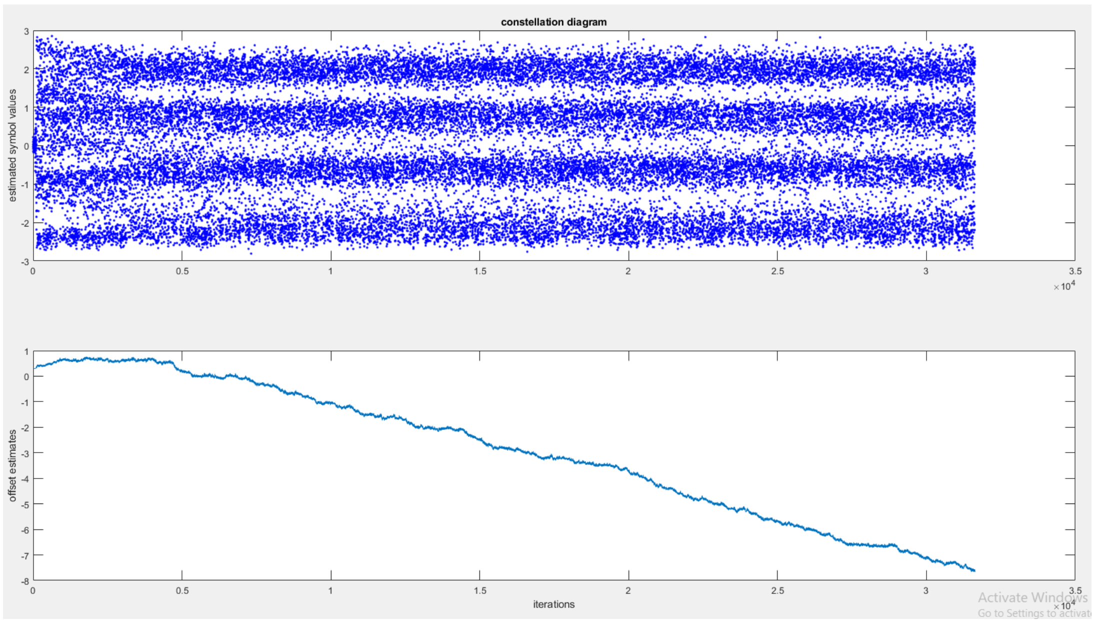

Figure 2.5: Baud timing offset (bottom) and output symbol constellation from the interpolator down-sampler (top) with an input sequence of ’medium.mat’

2.5 Locating Equalizer Training Sequences

In order to train our equalizer, we need to locate the training sequences, otherwise known as a header. In order to locate the headers I implemented a system do do correlation of the known header sequence and the output of the interpolator down-sampler. The code to do this is shown in Listing 5.

```
%Find Training Sequences
%Create training sequence vector
    TS = 'A0Oh well whatever Nevermind';
    TSpam = letters2pam2(TS);
    [TScorr, TScorrLags] = xcorr(IDxs, TSpam); %Find correlation between preamble and XS

    [TScorrPeaksMags, ¬] = findpeaks(TScorr, 'NPeaks', 11, 'MinPeakDistance', 1000, ...
        'MinPeakHeight', 500);
    if(isempty(TScorrPeaksMags))
        TScorr = -TScorr;
        IDxs = -IDxs;
    end

    [TScorrPeaksMags, TScorrPeaksLocs] = findpeaks(TScorr, 'NPeaks', 11, 'MinPeakDistance', ...
        1000, 'MinPeakHeight', 500);
    TSxsPeakLocs = TScorrLags(TScorrPeaksLocs); %Locations of preamble starts in terms of XS

    if(TSdebug)
        figure('Name', 'Correlation of Training Sequence')
        stem(TScorr);

        figure('Name', 'TScorr Peaks')
        stem(TScorrLags(TScorrPeaksLocs), TScorrPeaksMags);
    end
```

Listing 5: Correlation with the header to locate training sequences

The code checks for correlation peaks above a certain threshold, and if it does not locate one assumes the signal is inverted and proceeds to invert the signal. The correlation peaks are then searched for and the locations of which are listed into a vector. The output of the correlate can be seen in Figure 2.6. We can see very distinct peaks for each of the header locations.

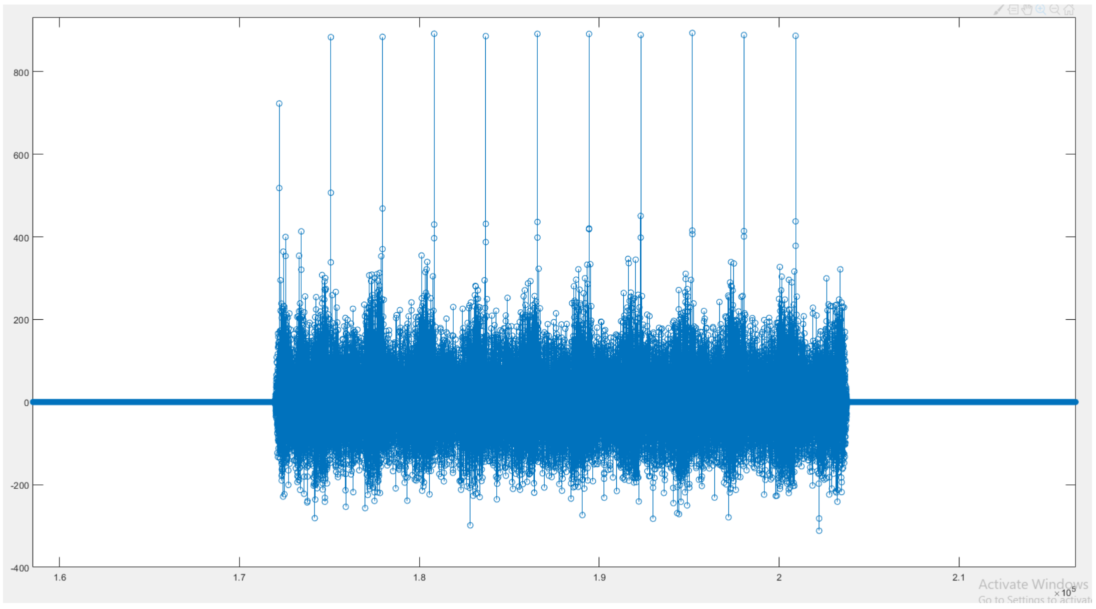

Figure 2.6: Output of the header correlator with an input sequence of 'medium.mat'

2.6 Equalization

When using the 'easy.mat' test vector everything described so far is enough to get perfect symbol extraction. In harder test vectors we do not have an echo-less channel. Since we have a channel that introduces echos into the signal, we need to create an equalizer in order to adaptivly remove these echos in order to reduce the inter-symbol interference. Since we are given a known training sequence at the beginning of each frame, we can use this to find the parameters of the channel and produce a filter to remove those echos as best as possible. I chose to perform a decision directed equalization algorithm that would train on the known header sequences and then freeze those settings while decoding the user data part of the frame. The code used to do this is seen in Listing 6.

Some other projects used the decision directed algorithm for the header and then transitioned into a blind equalizer when decoding the user data portion of the frame, however I did not find that necessary to do accurate decoding. Using a more complex algorithm like this or even a non-linear algorithm may improve performance more, possibly allowing this receiver to accurately decode even the 'hard.mat' test vector.

```
%Equalizer
    %Chop TSxs into blocks for processing
        EQout = [];
        EQn=12;
        EQmu=.025;
        EQ∆=4;             % stepsize and delay ∆
        EQf=zeros(EQn,1);           % initialize equalizer at 0

        for(EQi = 1:length(TSxsPeakLocs))

            EQr = IDxs(TSxsPeakLocs(EQi):TSxsPeakLocs(EQi) + length(TSpam) - 1);
            EQframe = IDxs(TSxsPeakLocs(EQi):TSxsPeakLocs(EQi)+2870-1);

            for EQj=EQn+1:length(TSpam)                   % iterate
                EQrr=EQr(EQj:-1:EQj-EQn+1)';          % vector of received signal
                EQe=TSpam(EQj-EQ∆)-EQrr'*EQf;         % calculate error
                EQf=EQf+EQmu*EQe*EQrr;                % update equalizer coefficients
            end

            EQout = [EQout,conv(EQf,EQframe)];
        end

        if(eqDebug)
            figure('Name', 'After EQ')
            plot(EQout, '.');
        end

        EQoutQuant = quantalph(EQout,[-3,-1,1,3]);
```

Listing 6: Equalization Algorithm

For this equalizer I settled on an equalizer filter length of 12 and an equalizer delay of 4. This length along with the step size of 0.025 gave me the best mix of equalizer performance vs. convergence speed. As the equalizer length increases the better job it can do at removing echos, but the longer it takes to converge. Since our training sequence was relatively short, and the channel changes quite quickly, we need our equalizer to be very responsive. We can see the constellation diagram of the equalizer with the input vector 'medium.mat' in Figure 2.7. When comparing it to the before equalized constellation diagram in Figure 2.5, we can see that the output of the equalizer has much tighter bands than before equalization.

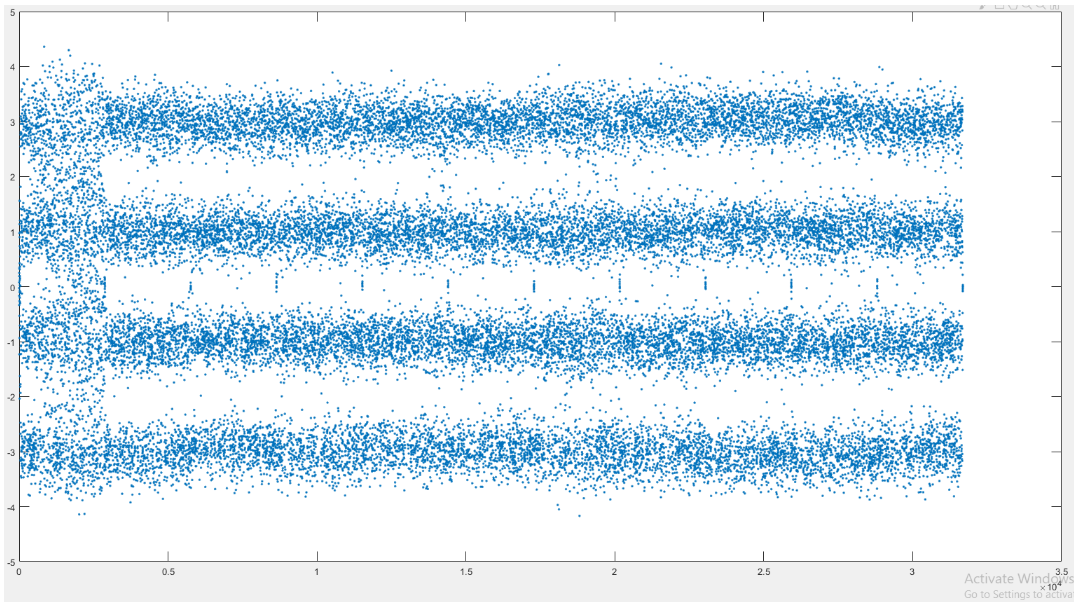

Figure 2.7: Output of the equalizer with an input sequence of 'medium.mat'

2.7 Re-Correlation and Decoding

After equalization we introduce delays and extra symbols between frames due to the frame by frame processing that was done in the last step. We need to redo the correlation in order to relocate the start of each frame and then grab the symbols that correspond to the user data that we are interested in decoding. This code where this is done in Listing 7. This data is then decoded into text.

```
%Find Training Sequences
    %Create training sequence vector
        [RTScorr, RTScorrLags] = xcorr(EQoutQuant, TSpam); %Find correlation between preamble ...
            and XS

        [RTScorrPeaksMags, RTScorrPeaksLocs] = findpeaks(RTScorr, 'NPeaks', 11, ...
            'MinPeakDistance', 2500, 'MinPeakHeight', 500);
        RTSxsPeakLocs = RTScorrLags(RTScorrPeaksLocs); %Locations of preamble starts in terms ...
            of XS

        if(RTSdebug)
            figure('Name', 'ReCorrelation of Training Sequence')
            stem(RTScorr);

            figure('Name', 'TScorr Peaks')
            stem(RTScorrLags(RTScorrPeaksLocs), RTScorrPeaksMags);
        end

        RTSUserOffset = [0, 875, 875*2];

        RTSout = [];
        for(RTSi = 1:length(RTSxsPeakLocs))
            RTSout = [RTSout, EQoutQuant(RTSxsPeakLocs(RTSi) + length(TSpam) + 1 + ...
                RTSUserOffset(desireduser): RTSxsPeakLocs(RTSi) + length(TSpam) + ...
                RTSUserOffset(desireduser) + 875)];
        end

        decoded_text = pam2letters2(RTSout);
```

Listing 7: Re-correlation and user data decoding

3 Results

After extensive testing and tuning, acceptable results were ac hived. The results are listed in Table 1.

	Bit Error Rate	Calculated Grade
'easy.mat'	0.0	100%
'medium.mat'	0.00095238	100%
'hard.mat'	0.0071429	90%
'mystery.mat'	unknown	100%

Table 1: Bit error rates and calculated grade for test vectors

Decoded Text for 'easy.mat':

> This is the first frame which you probably shouldn't able to decode perfectly unless you "cheat" and give your receiver the initial points. Now we're into the fourth frame. You might be able to decode this one, and now the fifth. The next frame is where your receiver should definitely be operatingwithout errors. You might want to re-test your receiver by using different initial parameters, and different stepsizes to see what the effect is. It's probably helpful to plot the time history of the adaptive parameter elements, too, so you can see if they're taking too long to converge, if they seem unstable, etc. This easy test vector has no impairments, i.e. no carrier frequency offset, a simple unity gain channel, no broadband noise, no narrowband interferers, no adjacent users, no baud timing clock offset, no phase noise. So, the algorithms in your receiver should settle fairly quickly, and remain constant... If your receiver has made it this far with no errors, and performs error-free even when you change the initial parameter values, Good job!! Now it's time to add impairments!'

Decoded Text for 'medium.mat':

> Twau Brylm)g,!!nd'the clithq toves Did eyre avd gimbhE yn txe wabe; Bu, mimsy were the bos/gow%s(And the mome raths outgrabe. "Beware the Jabberwock, my son! The jaws that bite, the claws that catch! Beware the Jubjub bird, and shun The frumious Bandersnatch!" He took his vorpal sword in hand: Loog time the manxome foe he sought– So rested he by the Tumtum tree, And stood awhile in thought. And as in uffish thought he stood, The Jabberwock, with eyes of flame, Came whiffling through the tulgey wood, And burbled as it came! One, two! One, two! and through and through The vorpal blade went soicker-snack! He left it dead, and with its head He went galumphing back. "And hast thou slain the Jabberwock? Come to my arms, my beamish boy! O frabjous day! Callooh! Callay!" He chortled in his joy/ 'Twas brillig, and the slithy toves Did gyre and gimble in the wabe; All mimsy were the borogoves, And the mome raths outgrabe. "Jabberwocky", a poem by Lewis Carroll which appeared in his 1872 novel Through the Looking-Glass, and What Alice Found There, a sequel to Alice's Adventures in Wonderlang

Decoded Text for 'hard.mat':

I Kick It Rowt Do{n I Put My Romt Down It's Nor A Put Down I!st!y G/ot Hown Ajd RXen I!!ke Soue Love, I Put My Root Down Like Sweetie Pie!y The Stone Alliance Everbody Knows I'm Known For Dropping Science I'm Electric Like Dick Hyman I Guess You'd Expect To Catch The Crew Rhymin' Never Let You Down With The Stereo Sound So Mike, Get On The Mic And Turn It Out We're Talking Root Down, I Put My Boot Down And If You Want To Battle Me, You're Putting Loot Down I Said Root Down, It's Time To Scoot Down I'm A Step Up To The Mic In My Goose Down Come Up Representing From The Upper West Money Makin' PuttYng Me To The Test Sometimes I Feel Bs Though I've Been Blessed Because I'm Doing What I Want So I Never Rest Well, I'm Not Coming Out Goofy Like The Fruit Of The Loom Guys Just Strutting Like The Meters With The Look-ka Py Py GCause Downtown Brooklyn Is Where I Was Born but Win The KnoW! sFalling, Then I'm Gone You Might Think Ti!t I'm A Fbnaric A Phone Call@Erom Utah And I'm Throwing A

As we can see, as we enter the frame, on harder vectors we are unable to decode the first part of the text, but as the various adaptive algorithms have time to converge, we get better and better decoding of the User data.

4 Conclusion

After beating every single problem that we encounter while trying to decode the test vectors with the sledge hammer known as the 'gradient decent algorithm' we were able to decode even quite difficult vectors. As we moved through each step of decoding it was very interesting to see how the output of each stage gets closer and closer to the transmitted data. There is plenty of room for improvement however. This receiver was still unable to decode the 'hard.mat' test vector completely. Better algorithms, especially when it comes to equalization could be used to improve this significantly. The naive equalizer used here only is adjusting during the training sequences and not during user data. Since the Chanel is contently changing, the equalize has a hard time keeping up when only training on the headers.

More tuning of the step-size parameters could also help, as the current settings are not necessarily ideal. Using an iterative process with bit error as a cost function could possibly be used to tune the various parameters for even better performance.

Being able to build a radio receiver from first principles is surprisingly complicated and ended up being a very rewarding process. That being said however, I hope next time we have to build a transmitter.

A Rx.m

```matlab
% Rx.m - Decodes a Transmitted signal
% David Tolsma
function [decoded_text, y] = Rx(r, rolloff, desireduser)


addpath('C:\Users\David\Documents\MATLAB\MatLab Files')
%load easy.mat
%load medium.mat
%load hard.mat


%desireduser = desired_user;

%Debug Bits

debugPll = 0;
ssrcDebug = 0;
IDdebug = 0;
TSdebug = 0;
eqDebug = 0;
```

```
RTSdebug = 0;

%Parameters

    interFreq = 2e6;
    sampleFreq = 850e3;
    Ts = 1/sampleFreq;
    symbolPeriod = 6.4e-6;
    srrcWidth = 8;
    overSampleFactor = symbolPeriod/Ts;


%Carrier Recovery

    %Carrier Preprocessing
        ppSquaredR = r.^2;
        ppFilterOrder = 500;
        ppFilterFreq = [0 .56 .58 .61 .63 1];
        ppFilterMags = [0 0 1 1 0 0];
        %ppFilterB = firpm(ppFilterOrder, ppFilterFreq, ppFilterMags);
        [ppFilterB, ppFilterA] = butter(6,[.58,.61]);
        ppRPreProcess = filter(ppFilterB, ppFilterA, ppSquaredR);

    %Carrier Frequency Recovery
        ppFFTrPreProcess = fft(ppRPreProcess);
        [ppM, ppImax] = max(abs(ppFFTrPreProcess(1:floor(end/2))));
        ppSsf = (0:length(ppRPreProcess))/(Ts*length(ppRPreProcess));
        ppFreqS = ppSsf(ppImax);
        ppPhaseP=angle(ppFFTrPreProcess(ppImax));
        [ppIR, ppF] = freqz(ppFilterB,1,length(ppFFTrPreProcess), sampleFreq);
        [ppMi, ppIm] = min(abs(ppF - ppFreqS));
        ppPhaseBPF = angle(ppIR(ppIm));
        ppPhaseS = mod(ppPhaseP - ppPhaseBPF, pi);

    %Dual PLLs
        pllMu1= 0.03; pllMu2 = 0.00003;                  % algorithm stepsizes
        pllF0=300000;                            % assumed freq at receiver
        pllT = 0:Ts:length(r)*Ts-Ts;

        pllTh1 = zeros(1, length(r));
        pllTh2 = zeros(1, length(r));
        pllTh2(1) = 0.05;
        for k=1:length(r)-1                      % combine top PLL th1
          pllTh1(k+1)=pllTh1(k)-pllMu1*ppRPreProcess(k)*sin(4*pi*pllF0 ...
          *pllT(k)+2*pllTh1(k)+ppPhaseBPF);
          pllTh2(k+1)=pllTh2(k)-pllMu2*ppRPreProcess(k)*sin(4*pi*pllF0 ...
          *pllT(k)+2*pllTh1(k)+2*pllTh2(k)+ppPhaseBPF);
        end

%Demodulation
    demodF0=300000;                             % assumed freq at receiver

    demodT = 0:Ts:length(r)*Ts - Ts;
    demodCos = cos(2*pi*demodF0*demodT + pllTh1 + pllTh2);
    demodR = demodCos.*r';

    demodFl=250; demodFf=[0 .25 .27 1]; demodFa=[1 1 0 0];
    demodH=firpm(demodFl,demodFf,demodFa);                    % LPF design
    %demodFilteredR = filter(demodH, 1, demodR);
    [demodFilterB, demodFilterA] = butter(6,.25,'low');
    demodFilteredR = filter(demodFilterB, demodFilterA, demodR);

    if(debugPll)
        figure('Name', 'Th1, Th2, DemodR')
        subplot(3,1,1)
        plot(pllTh1)
        subplot(3,1,2)
        plot(pllTh2)
        subplot(3,1,3)
        plot(demodFilteredR)
    end

%Pulse Shape Matched Filtering
    psmfPulseShape = srrc(srrcWidth/2, rolloff, (symbolPeriod/(Ts)));
    psmfRFiltered = conv(demodFilteredR, psmfPulseShape);
    if(ssrcDebug)
        figure('Name', 'Output of Pulse Shape Matched Filtering');
        plot(psmfRFiltered);
    end

%Interpolator Downsampler
    IDtnow = (srrcWidth/2)*overSampleFactor+1;
    IDtau = 0.3;
    IDxs = zeros(1,length(r));
    IDtauSave = zeros(1, length(r));
    IDi = 0;
    IDmuDD = 0.004;
    IDmuOP = 0.09;
    IDΔ = .46;


    while IDtnow < length(r)- 2*(srrcWidth/2)*overSampleFactor
        IDi = IDi + 1;
        IDxs(IDi) = interpsinc(psmfRFiltered, IDtnow + IDtau, (srrcWidth/2));
        IDx_Δp=interpsinc(psmfRFiltered,IDtnow+IDtau+IDΔ,(srrcWidth/2)); % value to right
        IDx_Δm=interpsinc(psmfRFiltered,IDtnow+IDtau-IDΔ,(srrcWidth/2)); % value to left

        IDdx = IDx_Δp-IDx_Δm;
        %IDtau = IDtau + IDmu*IDdx*IDxs(IDi);

        IDdx=IDx_Δp-IDx_Δm;              % numerical derivative
        IDqx=quantalph(IDxs(IDi),[-3,-1,1,3]);  % quantize to alphabet
        %IDtau=IDtau-IDmu*IDdx*(IDqx-IDxs(IDi));         % alg update: DD

        IDtau=IDtau - IDmuDD*IDdx*(IDqx-IDxs(IDi)) + IDmuOP*IDdx*IDxs(IDi);

        IDtnow=IDtnow+overSampleFactor;
        IDtausave(IDi)=IDtau;      % save for plotting
    end

    if(IDdebug)
        figure('Name', 'Plot of DD Interpolator')
        subplot(2,1,1), plot(IDxs(1:IDi-2),'b.')        % plot constellation diagram
        title('constellation diagram');
        ylabel('estimated symbol values')
        subplot(2,1,2), plot(IDtausave(1:IDi-2))        % plot trajectory of tau
        ylabel('offset estimates'), xlabel('iterations')
    end

%Find Training Sequences
    %Create training sequence vector
        TS = 'A0Oh well whatever Nevermind';
        TSpam = letters2pam2(TS);
        [TScorr, TScorrLags] = xcorr(IDxs, TSpam); %Find correlation between preamble and XS

        [TScorrPeaksMags, ¬] = findpeaks(TScorr, 'NPeaks', 11, 'MinPeakDistance', 1000, ...
            'MinPeakHeight', 500);
        if(isempty(TScorrPeaksMags))
            TScorr = -TScorr;
            IDxs = -IDxs;
        end

        [TScorrPeaksMags, TScorrPeaksLocs] = findpeaks(TScorr, 'NPeaks', 11, ...
            'MinPeakDistance', 1000, 'MinPeakHeight', 500);
        TSxsPeakLocs = TScorrLags(TScorrPeaksLocs); %Locations of preamble starts in terms of XS

        if(TSdebug)
            figure('Name', 'Correlation of Training Sequence')
            stem(TScorr);

            figure('Name', 'TScorr Peaks')
            stem(TScorrLags(TScorrPeaksLocs), TScorrPeaksMags);
        end

%Equalizer
    %Chop TSxs into blocks for processing
        EQout = [];
        EQn=12;
        EQmu=.025;
        EQΔ=4;                % stepsize and delay Δ
        EQf=zeros(EQn,1);            % initialize equalizer at 0

        for(EQi = 1:length(TSxsPeakLocs))

```

```
            EQr = IDxs(TSxsPeakLocs(EQi):TSxsPeakLocs(EQi) + length(TSpam) - 1);
            EQframe = IDxs(TSxsPeakLocs(EQi):TSxsPeakLocs(EQi)+2870-1);

            for EQj=EQn+1:length(TSpam)                  % iterate
                EQrr=EQr(EQj:-1:EQj-EQn+1)';         % vector of received signal
                EQe=TSpam(EQj-EQ∆)-EQrr'*EQf;        % calculate error
                EQf=EQf+EQmu*EQe*EQrr;               % update equalizer coefficients
            end

            EQout = [EQout,conv(EQf,EQframe)];
        end

        if(eqDebug)
            figure('Name', 'After EQ')
            plot(EQout, '.');
        end

        EQoutQuant = quantalph(EQout,[-3,-1,1,3]);


%Re-Correlate

%Find Training Sequences
    %Create training sequence vector
        [RTScorr, RTScorrLags] = xcorr(EQoutQuant, TSpam); %Find correlation between preamble ...
            and XS

        [RTScorrPeaksMags, RTScorrPeaksLocs] = findpeaks(RTScorr, 'NPeaks', 11, ...
            'MinPeakDistance', 2500, 'MinPeakHeight', 500);
        RTSxsPeakLocs = RTScorrLags(RTScorrPeaksLocs); %Locations of preamble starts in terms ...
            of XS

        if(RTSdebug)
            figure('Name', 'ReCorrelation of Training Sequence')
            stem(RTScorr);

            figure('Name', 'TScorr Peaks')
            stem(RTScorrLags(RTScorrPeaksLocs), RTScorrPeaksMags);
        end

        RTSUserOffset = [0, 875, 875*2];

        RTSout = [];
        for(RTSi = 1:length(RTSxsPeakLocs))
            RTSout = [RTSout, EQoutQuant(RTSxsPeakLocs(RTSi) + length(TSpam) + 1 + ...
                RTSUserOffset(desireduser): RTSxsPeakLocs(RTSi) + length(TSpam) + ...
                RTSUserOffset(desireduser) + 875)];
        end

        decoded_text = pam2letters2(RTSout);
y = EQout;
```

Listing 8: Full Rx.m Code

References

[1] A. Klein, ”Digital Communication Systems Spring 2019 Final Project” Western Washington University, Spring 2019.

[2] A. Klein, ”Chapter 15: Make It So” Western Washington University, Spring 2019.