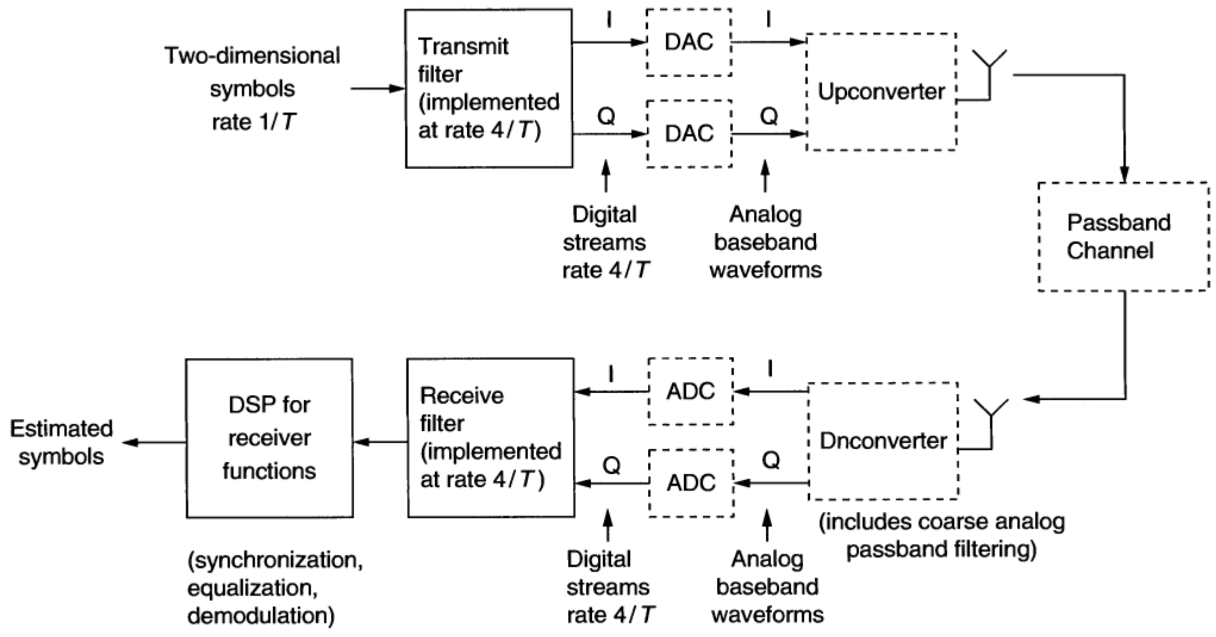
Linear Modulation Over a Noiseless Ideal Channel

ECE 146 Lab 3

November 23, 2019



Two-dimensional symbols rate 1/T
Transmit filter (implemented at rate 4/T)
I
Q
DAC
DAC
I
Q
Upconverter
Digital streams rate 4/T
Analog baseband waveforms
Passband Channel
Dnconverter
(includes coarse analog passband filtering)
ADC
ADC
I
Q
Receive filter (implemented at rate 4/T)
DSP for receiver functions
(synchronization, equalization, demodulation)
Estimated symbols
Digital streams rate 4/T
Analog baseband waveforms

Student:	Ivan Arevalo
Perm Number:	5613567
Email:	ifa@ucsb.com

Department of Electrical and Computer Engineering, UCSB

0 Introduction

This lab implements a linearly modulated system (Figure 1) over a noiseless channel using the complex-baseband representation. Our model does not include DAC and ADC effects for simplicity. We are choosing the cascaded transmitter and receiver filters to be a raised cosine (RC) pulse by choosing each to be a square root raised cosine pulse (SRRC) which inherits the Nyquist property of the sinc pulse. The excess bandwidth and smooth frequency shape of a RC pulse results in a time domain attenuation proportional to $\frac{1}{t^3}$, mitigating intersymbol interference (ISI) in presence of slight sampling perturbations at the receiver. We will simulate sending 100 random bits under 3 different scenarios and determine the probability error of recovering the original bit sequence with one of two encoding systems.

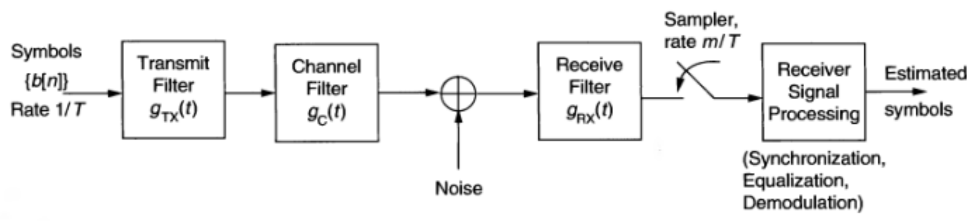

Symbols
{b[n]}
Rate 1/T
Transmit Filter $g_{TX}(t)$
Channel Filter $g_C(t)$
Noise
Receive Filter $g_{RX}(t)$
Sampler, rate m/T
Receiver Signal Processing
(Synchronization, Equalization, Demodulation)
Estimated symbols

Figure 1: Block diagram of a linearly modulated system, modeled in complex baseband

1 Linearly Modulation Modeled in Complex Baseband

1.1 Function to generate a SRRC pulse.

```
function [rc,time_axis] = sqrtraised_cosine(a,m,length)
length_os = floor(length*m); %number of samples on each side of peak
%time vector (in units of symbol interval) on one side of the peak
t = cumsum(ones(length_os,1))/m;
A= 4*a*cos(pi*(1+a)*t); %term 1
B= sin(pi*(1-a)*t)./t; %term 2
C= pi*(1-16*a^2*t.^2); %term 3
zerotest = m/(4*a); %location of zero in denominator
%check whether any sample coincides with zero location
if (zerotest == floor(zerotest))
    t_perturbed = t(zerotest)+0.001;
    A(zerotest) = 4*a*cos(pi*(1+a)*t_perturbed);
    B(zerotest) = sin(pi*(1-a)*t_perturbed)./t_perturbed;
    C(zerotest) = pi*(1-16*a^2*t_perturbed.^2);
end
D = (A+B)./C; %response to one side of peak
rc = [flipud(D);(4*a+pi*(1-a))/pi;D]; %add in peak and other side
time_axis = [flipud(-t);0;t];
```

1.2 Transmitter and receiver filters with a 22% excess bandwidth, sampled at $\frac{4}{T}$ and truncated to [-5T, 5T].

```
a = 0.22; % desired excess bandwidth
m = 4; %oversample by a lot to get smooth plot
```

```
length_RC = 5; % where to truncate the time domain response
%(one-sided, multiple of symbol time)
[transmit_filter,transmit_filter_time] = sqrtraised_cosine(a,m,length_RC);
figure('Name', 'Section 1.2');
plot(transmit_filter_time,transmit_filter);
```

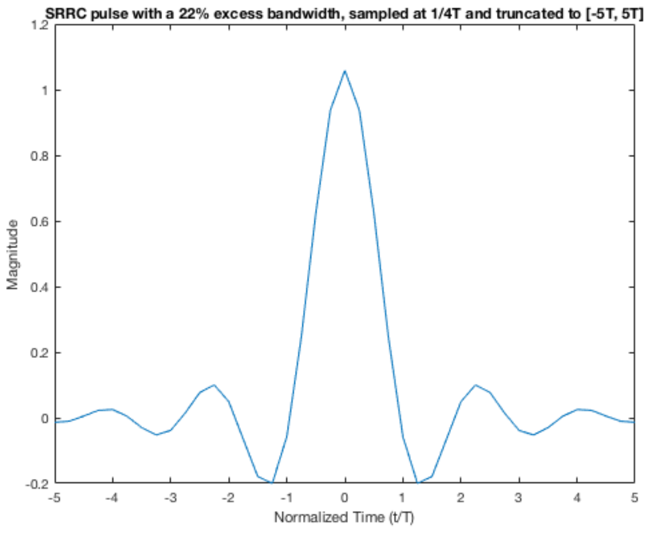


Figure 2: Transmitter and receiver filters with a 22% excess bandwidth, sampled at $\frac{4}{T}$ and truncated to [-5T, 5T].

1.3 Magnitude of transmitter transfer function

```
ts=1/4; % Normalized sampling interval T = 1
signal_timedomain = transmit_filter; %sinusoidal pulse in our example
fs_desired = 1/64; %desired frequency granularity
Nmin = ceil(1/(fs_desired*ts)); %minimum length DFT for desired frequency granularity
%for efficient computation, choose FFT size to be power of 2
Nfft = 2^(nextpow2(Nmin)); %FFT size = the next power of 2 at least as big as Nmin
%Alternatively, one could also use DFT size equal to the minimum length
%Nfft=Nmin;
%note: fft function in Matlab is just the DFT when Nfft is not a power of 2
%freq domain signal computed using DFT
%fft function of size Nfft automatically zeropads as needed
signal_freqdomain = ts*fft(signal_timedomain,Nfft);
%fftshift function shifts DC to center of spectrum
signal_freqdomain_centered = fftshift(signal_freqdomain);
fs=1/(Nfft*ts); %actual frequency resolution attained
%set of frequencies for which Fourier transform has been computed using DFT
freqs = ((1:Nfft)-1-Nfft/2)*fs;
%plot the magnitude spectrum
figure('Name', 'Section 1.3');
plot(freqs,abs(signal_freqdomain_centered));
title('Magnitude of transmitter transfer function');
xlabel('Normalized Frequency (fT)');
ylabel('Magnitude Spectrum');
```

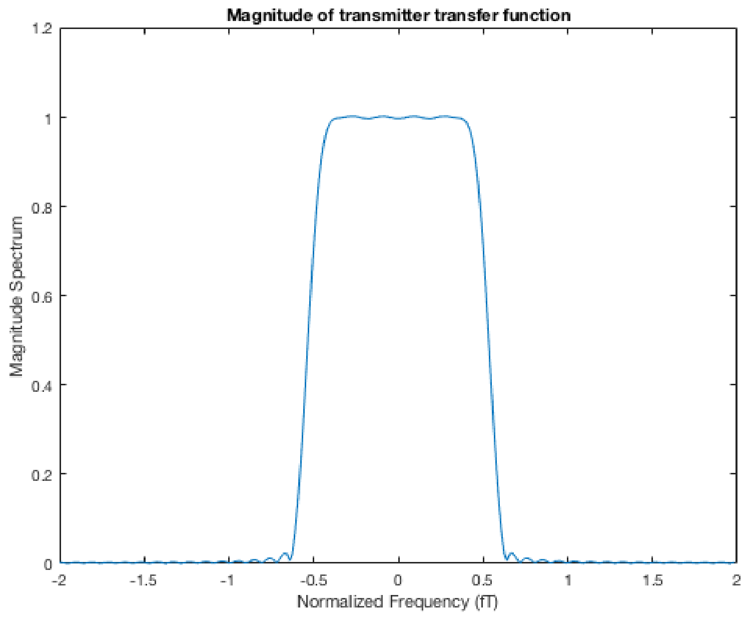


Figure 3: Magnitude of transmitter transfer function

Observing Figure 3, we notice that the normalized bandwidth is well predicted by the nominal excess bandwidth given that the pulse normalized bandwidth is 1.22. This matched our expected normalized bandwidth with an excess of 22%.

1.4 Sending a symbol through a system

The following snippet of code allows us to a symbols through a system by covolving it with the transmitter and receiver filter.

```
oversampling_factor = 4;
m = oversampling_factor;
%parameters for sampled raised cosine pulse
a = 0.22;
length_RC = 5;% (truncated outside [-length*T,length*T])
[transmit_filter, transmit_filter_time] = sqrtraised_cosine(a,m,length_RC);
receive_filter = transmit_filter(end:-1:1); % matched filter
receive_filter_time = transmit_filter_time;

%NUMBER OF SYMBOLS
nsymbols_p4 = 1;
%BPSK SYMBOL GENERATION
symbols_p4 = sign(rand(nsymbols_p4,1) -.5);
%UPSAMPLE BY m
nsymbols_upsampled_p4 = 1+(nsymbols_p4-1)*m;%length of upsampled symbol sequence
symbols_upsampled_p4 = zeros(nsymbols_upsampled_p4,1);%initialize
symbols_upsampled_p4(1:m:nsymbols_upsampled_p4)=symbols_p4;%insert symbols with spacing m
```

```
t_s = receive_filter_time(end)-receive_filter_time(end-1);
symbols_upsampled_p4_time = (0:nsymbols_upsampled_p4)*t_s;

%NOISELESS MODULATED SIGNAL
[tx_output_p4, tx_output_p4_time] = contconv2(symbols_upsampled_p4,transmit_filter,
    symbols_upsampled_p4_time(1),transmit_filter_time(1), ts);
tx_output_p4 = m*tx_output_p4; % Scale by m
[rx_output_p4, rx_output_p4_time] = contconv2(tx_output_p4, receive_filter,
    tx_output_p4_time(1), receive_filter_time(1), ts);

figure('Name', 'Section 1.4');
subplot(2,1,1);
plot(tx_output_p4_time, tx_output_p4);
title('Transmitter Output'); ylabel('Magnitude'); xlabel('Time');
subplot(2,1,2);
plot(rx_output_p4_time, rx_output_p4);
title('Receiver Output'); ylabel('Magnitude'); xlabel('Time');
```

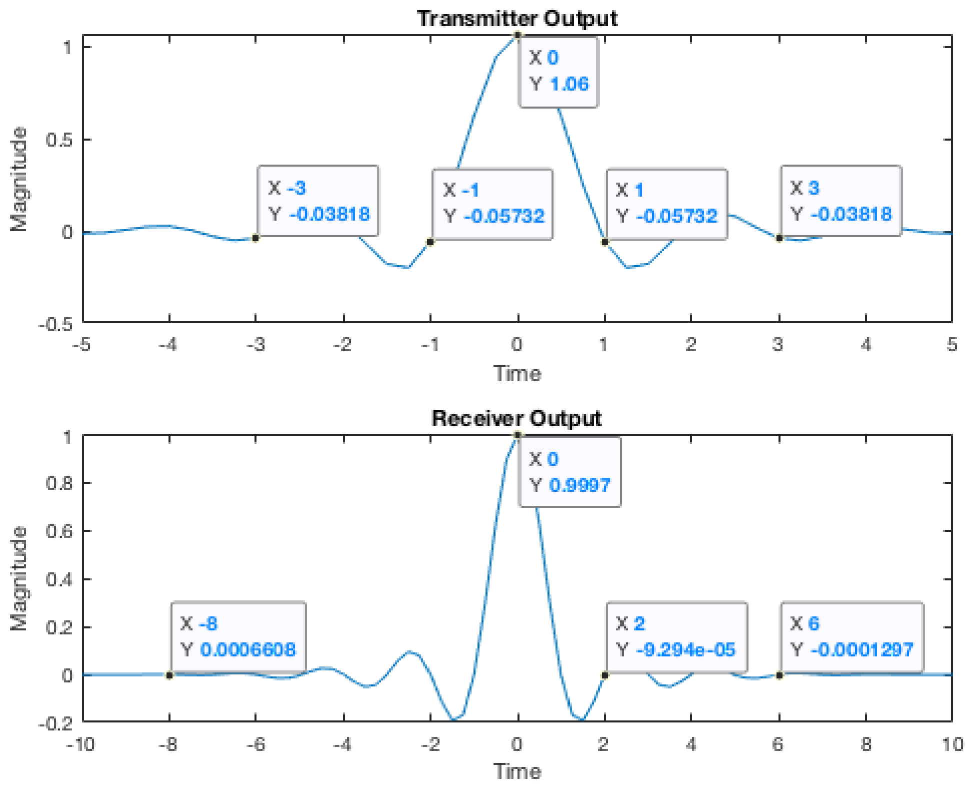


Figure 4: Transmitter and receiver filter output response

Figure 4 shows the output response of both the transmitter and receiver filter. We can see that at integer values of the normalized time with respect to symbol time T, the magnitude of the receiver filter is 1 at mT for m = 0, and 0 for every other integer value of m. As expected, the cascade of the transmitter and receiver filters is Nyquist at rate 1/T.

1.5 Generating 100 random bits

We can now generate 100 random binary bits and map them to symbols of value +1, -1.

```
%NUMBER OF SYMBOLS
nsymbols = 100; % Binary 0 mapped to +1, Binary 1 mapped to -1
%BPSK SYMBOL GENERATION
symbols = sign(rand(nsymbols,1) -.5);
```

1.6 Sending 100 random bits through a system

Taking the 100 random bits generated in 1.5 and the transmitter and receiver filters generated in 1.4:

```
%UPSAMPLE BY m
nsymbols_upsampled = 1+(nsymbols-1)*m;%length of upsampled symbol sequence
symbols_upsampled = zeros(nsymbols_upsampled,1);%initialize
symbols_upsampled(1:m:nsymbols_upsampled)=symbols;%insert symbols with spacing m
t_s = receive_filter_time(end)-receive_filter_time(end-1);
symbols_upsampled_time = (0:nsymbols_upsampled)*t_s;

%NOISELESS MODULATED SIGNAL
[tx_output, tx_output_time] = contconv2(symbols_upsampled,transmit_filter,
    symbols_upsampled_time(1),transmit_filter_time(1), ts);
tx_output = m*tx_output; % Scale by m
[rx_output, rx_output_time] = contconv2(tx_output, receive_filter, tx_output_time(1),
    receive_filter_time(1), ts);

figure('Name', 'Section 1.6');
subplot(2,1,1);
plot(tx_output_time, tx_output);
title('Transmitter Output'); ylabel('Magnitude'); xlabel('Normalized Time (T)');
subplot(2,1,2);
plot(rx_output_time, rx_output);
title('Receiver Output'); ylabel('Magnitude'); xlabel('Normalized Time (T)');

length_transmitter_ouput = length(tx_output)
length_receiver_ouput = length(rx_output)
```

```
length_transmitter_ouput = 437
length_receiver_ouput = 477
```

We calculated that the transmitter and receiver output had a length of 437 and 477 points respectively. This makes sense since the original symbols sequence of length 100 got upsampled by 4, putting 3 zeros in between each sample, for a final length of 397 points. After convolving with a filter of length 41 points, we get a resulting output of length 41 + 397 -1 = 437 points. Likewise, the output of the transmitter filter convolved with the receiver filter gives us a receiver output of length 477 points. This will be an important fact to keep in mind at the receiver when we need to sample the output to recover the mapped bit sequence. As shown in Figure 5, given that the transmitter and receiver filter are truncated at [-5T, 5T], we notice how in time domain, the output of the transmitter and receiver signal span over -5 to 105 and -10 to 110 T respectively.

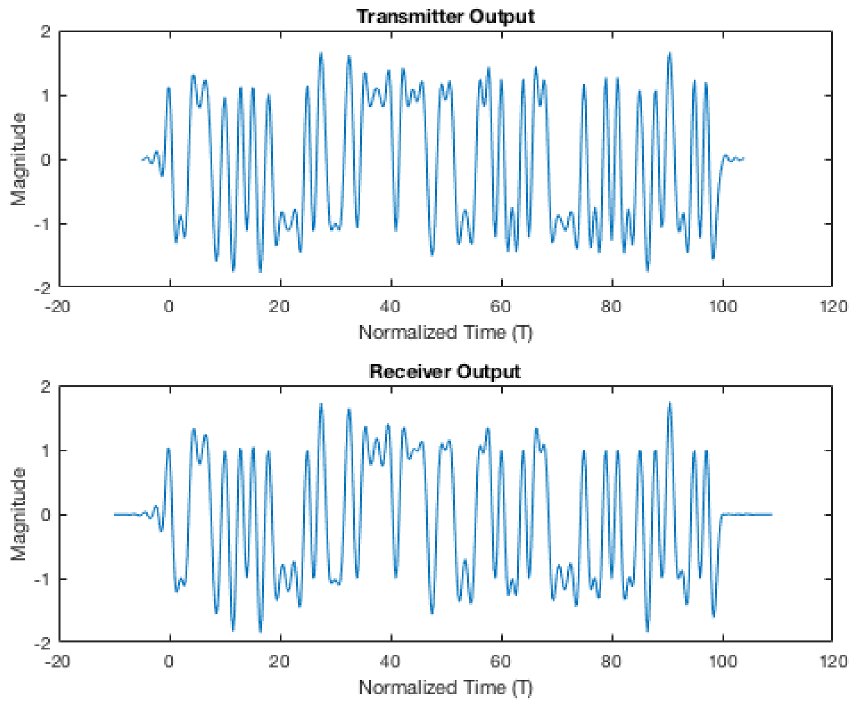


Figure 5: Transmitter and receiver filter output response

1.7 Recovering sent bits from transmitter output

Given that we are modeling a noiseless channel, we can recover the sent bits by sampling the output of the transmitter filter by analyzing the sign of each recovered mapped value and mapping them back to the original binary bit stream sent. It is important to remember that given our convolution of the mapped values with the transmitter filter, there is a delay we must apply before starting to sample the output in order to get the correct mapped bit values.

```
start_index_of_sequence_tx = find(tx_output_time==0);
end_index_of_sequence_tx = find(tx_output_time==99);

select_samples_tx = tx_output(start_index_of_sequence_tx:4:end_index_of_sequence_tx);
recoverd_samples_tx = sign(select_samples_tx);
error_tx = 100*(1/nsymbols)*sum((symbols-recoverd_samples_tx));
fprintf("The percent error measured was " + error_tx + "%% since no noise has been
    added");
```

```
The percent error measured was 0% since no noise has been added
```

1.8 Recovering sent bits from receiver output

We can repeat the same process as before to recover the sent bits at the receiver output, again noting that there will be a delay applied before we starts sampling.

```
start_index_of_sequence_rx = find(rx_output_time==0);
end_index_of_sequence_rx = find(rx_output_time==99);

select_samples_rx = rx_output(start_index_of_sequence_rx:m:end_index_of_sequence_rx);
recoverd_samples_rx = sign(select_samples_rx);
error_rx = 100*(1/nsymbols)*sum(abs(symbols-recoverd_samples_rx)/2);
fprintf("The percent error measured was " + error_rx + "%% since no noise has been
    added\n");
```

```
The percent error measured was 0% since no noise has been added
```

Let's now test the reliability of this method given that we choose and incorrect delay at the receiver, offset from the correct delay by T/2.

```
% Choosing a wrong delay offset by T/2.
select_samples_rx_2 =
    rx_output((start_index_of_sequence_rx:m:end_index_of_sequence_rx)+m/2);
recoverd_samples_rx_2 = sign(select_samples_rx_2);
error_rx_2 = 100*(1/nsymbols)*sum(abs(symbols-recoverd_samples_rx_2)/2);
fprintf("The percent error measured was " + error_rx_2 + "%% since no noise has been
    added\n");
```

```
The percent error measured was 22% given a delay offset of T/2
```

1.9 Frequency and phase mismatch at receiver

Given that the local oscillator used for downconversion at the receiver has a frequency and phase offset relative to the incoming wave by $\triangle f = \frac{1}{40T}$ and $\frac{\pi}{2}$, can we modify our baseband model to include the the effects of this carrier frequency offset?

```
nsymbols_upsampled = 1+(nsymbols-1)*m;%length of upsampled symbol sequence
symbols_upsampled = zeros(nsymbols_upsampled,1);%initialize
symbols_upsampled(1:m:nsymbols_upsampled)=symbols;%insert symbols with spacing m
t_s = receive_filter_time(end)-receive_filter_time(end-1);
symbols_upsampled_time = (0:nsymbols_upsampled)*t_s;

%NOISELESS MODULATED SIGNAL
[tx_output, tx_output_time] = contconv2(symbols_upsampled,transmit_filter,
    symbols_upsampled_time(1),transmit_filter_time(1), ts);
tx_output = m*tx_output; % Scale by m

offset_freq = 1/40;
phase = pi/2;
tx_output = tx_output.*exp(-1i*(2*pi*offset_freq*tx_output_time' + phase)); % Carrier
    and phase effects
[rx_output, rx_output_time] = contconv2(tx_output, receive_filter, tx_output_time(1),
    receive_filter_time(1), ts);

% Calculate delay and sampling space
start_index_of_sequence_rx = find(rx_output_time==0);
end_index_of_sequence_rx = find(rx_output_time==99);

select_samples_rx = rx_output(start_index_of_sequence_rx:m:end_index_of_sequence_rx);
recoverd_samples_rx = sign(real(select_samples_rx));
figure('Name', 'Section 1.9');
plot(select_samples_rx, 'o');
title('Recovered complex values from receiver output');
xlabel('Real'); ylabel('Imaginary');
```

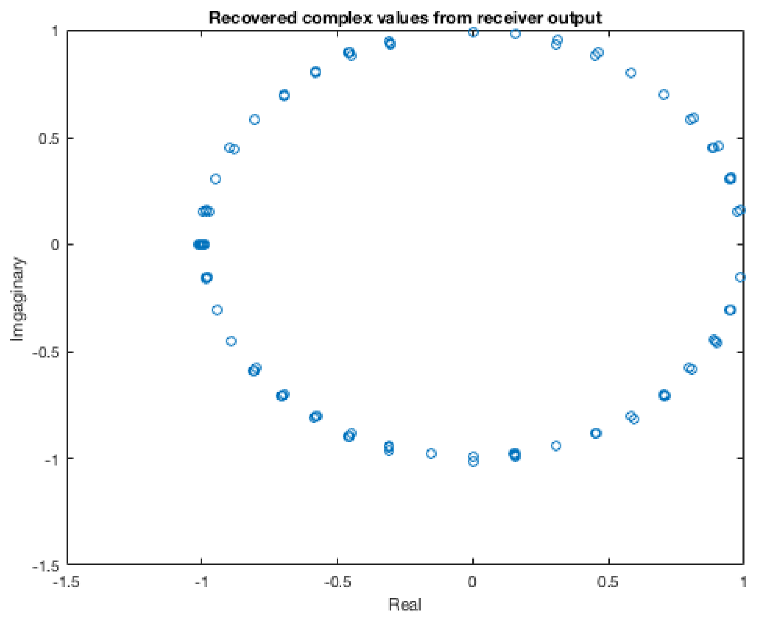


Figure 6: Recovered complex values from receiver output

In order to compensate for the frequency and phase being ahead at the receiver, we retard the mapped complex-baseband representation after it goes through the transmitter filter. We can now downconvert with respect to the receiver reference and recover the retarded complex-baseband values as shown in Figure 6. As we will show, using the previous method of sampling the receiver output at the correct delay and determining the sent binary bit by inspecting the sign of the real part of the recovered value, does not work for this case.

```
% Run 100 trials to estimate error probability
num_trials = 100;
error_rx = zeros(num_trials, 1);
for i=1:100
    %NUMBER OF SYMBOLS
    nsymbols = 100; % Binary 0 mapped to +1, Binary 1 mapped to -1
    %BPSK SYMBOL GENERATION
    symbols = sign(rand(nsymbols,1) -.5);

    nsymbols_upsampled = 1+(nsymbols-1)*m;%length of upsampled symbol sequence
    symbols_upsampled = zeros(nsymbols_upsampled,1);%initialize
    symbols_upsampled(1:m:nsymbols_upsampled)=symbols;%insert symbols with spacing m
    t_s = receive_filter_time(end)-receive_filter_time(end-1);
    symbols_upsampled_time = (0:nsymbols_upsampled)*t_s;

    %NOISELESS MODULATED SIGNAL
    [tx_output, tx_output_time] = contconv2(symbols_upsampled,transmit_filter,
        symbols_upsampled_time(1),transmit_filter_time(1), ts);
    tx_output = m*tx_output; % Scale by m
    offset_freq = 1/40; %Offset
    phase = pi/2; %Phase
    tx_output = tx_output.*exp(-1i*(2*pi*offset_freq*tx_output_time' + phase)); %
```

```
        Carrier and phase effects
    [rx_output, rx_output_time] = contconv2(tx_output, receive_filter,
        tx_output_time(1), receive_filter_time(1), ts);

    % Calculate delay and sampling space
    start_index_of_sequence_rx = find(rx_output_time==0);
    end_index_of_sequence_rx = find(rx_output_time==99);
    select_samples_rx = rx_output(start_index_of_sequence_rx:m:end_index_of_sequence_rx);
    recoverd_samples_rx = sign(real(select_samples_rx));
    error_rx(i) = (1/nsymbols)*sum(abs(symbols-recoverd_samples_rx)/2);
end

error_probability = 100*(1/num_trials)*sum(error_rx);
fprintf("Given 100 trials, the percent error measured was " + error_probability + "%% ");
```

```
Given 100 trials, the percent error measured was 59.46%
```

As shown above, we estimated the probability of error to be an unacceptable 59.46%.

1.10 Differentially encoded system

Let's now consider a differentially encoded system in which we have a binary bit stream $a[n]$ of size N we want to send. In order to do so, we generate a size N + 1 secondary sequence $b[n]$ of values ± 1 with a $+1$ for its first value ($b[1]$) and determine the rest of its values in the following manner. If the binary bit of $a[n]$ at position n is 0, then $b[n+1] = b[n]$, if it is 1, then $b[n+1] = -b[n]$. We can now repeat the case mentioned previously in section 1.9 where our receiver local oscillator is ahead in frequency and phase relative to the incoming wave by $\triangle f = \frac{1}{40T}$ and $\frac{\pi}{2}$. Let's determine if we are able to better estimate the initial binary bit sequence $a[n]$ from the samples taken at the receiver output.

```
%NUMBER OF SYMBOLS for a[n]
nsymbols = 100;
offset_freq = 1/40;
phase = pi/2;
t_s = receive_filter_time(end)-receive_filter_time(end-1);
num_trials = 100;

error_rx = zeros(num_trials, 1);
for num_trial=1:num_trials
    %BPSK SYMBOL GENERATION
    a_n_symbols = randi([0 1],nsymbols,1);
    b_n_symbols = zeros(nsymbols+1,1);
    b_n_symbols(1) = 1; % first bit is +1

    for i=1:nsymbols
        if a_n_symbols(i) == 0 %binary 0
            b_n_symbols(i+1) = b_n_symbols(i);
        else %binary 1
            b_n_symbols(i+1) = -1*b_n_symbols(i);
        end
    end

    %UPSAMPLE BY m
    nsymbols_upsampled = 1+(nsymbols+1-1)*m;%length of upsampled symbol sequence b[n] =
        len(a[n])+1
    symbols_upsampled = zeros(nsymbols_upsampled,1);%initialize
    symbols_upsampled(1:m:nsymbols_upsampled)=b_n_symbols; %insert symbols with spacing m
    symbols_upsampled_time = (0:nsymbols_upsampled)*t_s;

    %NOISELESS MODULATED SIGNAL
    [tx_output, tx_output_time] = contconv2(symbols_upsampled, transmit_filter,
        symbols_upsampled_time(1),transmit_filter_time(1), ts);
    tx_output = m*tx_output; % Scale by m
    tx_output = tx_output.*exp(-1i*(2*pi*offset_freq*tx_output_time' + phase)); %
        Carrier and phase effects
    [rx_output, rx_output_time] = contconv2(tx_output, receive_filter,
        tx_output_time(1), receive_filter_time(1), ts);

    % Calculate delay and sampling space
    start_index_of_sequence_rx = find(rx_output_time==0);
    end_index_of_sequence_rx = find(rx_output_time==nsymbols);
    % Select samples factoring in delay
    select_samples_rx = rx_output(start_index_of_sequence_rx:m:end_index_of_sequence_rx);
    recoverd_samples_rx = select_samples_rx; % recovered b[n] samples.

    recovered_a_n = zeros(nsymbols, 1);
    for i=2:100
        if recoverd_samples_rx(i)*conj(recoverd_samples_rx(i-1)) >=0 %binary 0
            recovered_a_n(i-1) = 0;
        else %binary 1
            recovered_a_n(i-1) = 1;
        end
    end
    error_rx(num_trial) = (1/nsymbols)*sum((a_n_symbols-recovered_a_n));
end

error_probability = 100*(1/num_trials)*sum(error_rx);
fprintf("Given 100 trials, the percent error measured was " + error_probability + "%% ");
```

```
Given 100 trials, the percent error measured was 0.55%
```

We observe that our probability error measured with the differentially encoded system is significantly lower than our original mapping algorithm for a binary sequence, with a probability of 0.55% error versus 59.46% error.

2 Conclusion

In conclusion, we began this lab by introducing linear modulation and analyzing the modulation pulse used (SRRC) in both frequency and time domain. We proved that a single SRRC pulse and the cascaded SRCC pulses we generated are Nyquist at a rate of 1/T. Furthermore, we simulated sending 100 random bits under 3 different scenario through a noiseless channel. We first encoded the binary sequence to ± 1 values and sent them through ideal conditions where the receiver had no frequency or phase offset and the receiver output was sampled with the correct delay. In this case, we proved that there was 0% probability error. We then sent the same mapped bit sequence through a system that had the local oscillator used for downconversion at the receiver at a frequency and phase offset relative to the incoming wave of $\triangle f = \frac{1}{40T}$ and $\frac{\pi}{2}$. We showed that using the same encoding system used previously would result in an unacceptable probability error of 59%. Last, we introduced a differential encoding system to send the same 100 bits through the same system with a frequency and phase receiver offset. With this encoding system however, we proved that the probability error was much more acceptable than before at 0.55%. We learned that the encoding used to send our information through a communication link is just as important as any other component in our system.