Adaptive Equalization

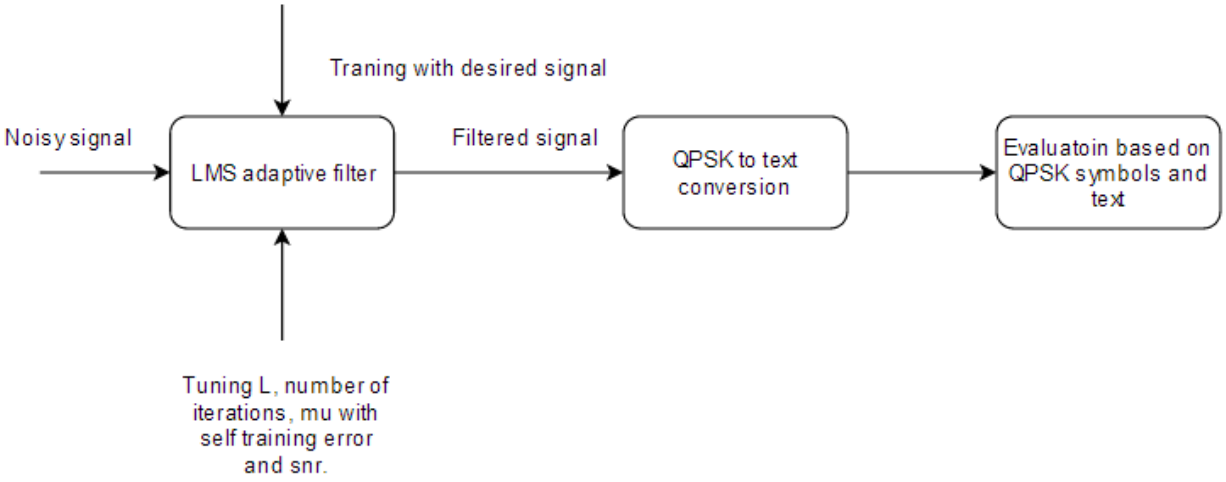

Figure 1. The block diagram of the algorithm.

OVERVIEW

We retrieve noisy signals from the provided signals and we also generate lower SNR signals in order to challenge our design. To design adaptive filter, we first train the filter with the desired signal. The desired signal is an eighty-symbol QPSK sequence which indicates all possible characters in our ground truth text file. The ground truth text file used for testing is the provided by the professor. After we train the desired signal, we run LMS algorithm over the noisy signal and evaluate the performance with the self-test training error as MSE and SNR ratio of the generated QPSK sequence. These two metrics are only used for determining an ambiguous range of the idea order of our filter. We then manually experimenting the order of our filter to achieve an overall better result. Then the filtered signal which is a sequence of QPSK signal is converted into text sequence. Eventually, the text sequence will be used to evaluate the ultimate performance of our algorithm by counting the error rate of text characters and QPSK symbols.

DESCRIPTION

Parameters

Filter order

In my lms_decode.m function, I run through 2 to 40 possible orders of our filter and evaluate the immediate performance with the MSE error and SNR ratio of them. Order that generates smaller MSE error and SNR ratio will be displayed. This gives me a range of orders I need to tune for my adaptive filter. Mostly I get order 2 to 11 for different kinds of signals. Then we manually tune these order numbers by comparing the ultimate error rate. After extensive experiments, I decided to use order 6 and 8 for my adaptive filter. Order 8 is mostly used for time-varying noisy signals based on our results. This is because the time-varying signals need more information when we use longer orders in optimal situation. If we use order 11 however, the error rate

increases, at 10 db level, by 3%. This tells us that, there is a local minimal for optimizing the number of orders. Order 6 is used for time-invariant signals, because taking many time-invariant signals in one iteration will degrade the performance of our filter. Smaller order is also not acceptable because they do not carry enough information to train the proper model. With lower orders, only-noisy signals at 15db sometimes will also have error which could only occur at 5-10 db with the proper order for our filter. The performance of using order 6 and 8 is negligible if the number of iterations is fixed. Therefore, we use order 6 for general use.

```
[~,~,error,snr] = lms_filter(x,2,d);
L = 2;
E = error;
SNR_QPSK = snr;
for i = 3:40;
    [~,~,error,snr] = lms_filter(x,i,d);
    E = [E error];
    SNR_QPSK = [SNR_QPSK,snr];
end
[min_E,order_desired_E] = min(E);
[max_SNR,order_desired_SNR] = max(SNR_QPSK);
```

Filter initiation

I have tried to use random number, zeros, and ones to setup the original filter. Zeros and ones yield the exactly same result. While the random numbers yield unstable performance. I choose all zeros for my filter initiation because based on my experiments, mostly of time, initiating with zeros have 1-2% improvement for the overall accuracy.

```
h = zeros(L,1);
% h = 2*(rand(L,1)-0.5)*(2/L)
```

Adaptive constant mu:

I have tried to use both regular mu and normalized mu for my filter. The benefit of using normalized mu is that we do not have to know SNR of our signal beforehand. However, in this case, we do know the noise will always be complex zero mean unit variance Gaussian noise. Therefore we can generate a sample noise signal to calculate the SNR ratio of our noisy signal. This tends out to work quite well when testing with lower SNR channel from 5-10 db. Based on my experiment results, I decide to use regular mu. However, if the noise is no longer deterministic, we might need to change to normalized mu because we do not know SNR beforehand.

```
mu_max = 1/L/(S^2/2);
mu = mu_max/100;
```

OR

```
% mu_k = 0.5/(xk'*xk+0.001);
```

N

N is the number of iterations for training the filter. I conduct experiments with 1000:1000:10000 sequence of N values. The plot of MSE vs iterations shows that when MSE starts to converge when the number of iterations is close to the length of signal. The conclusion is that, for time-

varying signals, having N close to the length of the noisy signals can achieve the best performance. For example, the error rate of time-varying signals with order 8 drops from 6.3% to 3.5% when using N values close to the length of signal instead of using 10000 at 15db. However, for time-invariant signals, we do need more iterations to lower down the MSE of our filter. In the case that 15db time-invariant signals with order 6 and 4000 iterations, the result has error rate is 0.13% which is not perfect since we can easily achieve zero error rate in this situation with more iterations. Therefore, I choose 10000 iterations to make sure that our algorithm will not break for the time-invariant signals when SNR is still high enough. Losing some accuracy is acceptable for time-varying signals since they are very hard to reach 100% accuracy.

```
MSE_arr = [];
% iterations = 1000:1000:10000;
iterations = length(V);
for N = iterations
    for k = L:N
        xk = x_arr(k:-1:k-L+1);
        output = (h')*xk;
        error = d_arr(k) - output;
%         mu_k = 0.5/(xk'*xk+0.001);
%         h = h + mu_k*conj(error)*xk;
        h = h+mu*conj(error)*(xk);
    end
self_test = conv(h',x);
E = d - self_test(L:end);
MSE = sum(abs(E).^2)/N;
MSE_arr = [MSE_arr MSE];
end
% plot(iterations,MSE_arr)
% title('MSE vs iterations')
% xlabel('iterations')
% ylabel('MSE')
```

Important code snippets

Training sequence

Since we know the desired signal in advance, we can use the desired signal to train our adaptive filter. The original sequence of desired signal is short. Therefore, we can concatenate its copy and the first 80 samples from the noisy signals N/L times to themselves respectively. We end up having two arrays that contains the copy of the desired signals and the 80 sample from the noisy signals. The length of the array is greater than or equal to the number of iterations. We can then use these two arrays to train our adaptive filter.

```
x_arr = [];% d is desired signal
d_arr =[];
x = V(1:length(d));
for i = 1: ceil(N/L)
    x_arr = [x_arr;x];
    d_arr = [d_arr;d];
end
```

Filter Adaptation

The filter coefficients are trained with following code. The loop will runs N-L times and starts from L in order to avoid the filter transient. As mentioned before, N is chosen as a large value to make sure the perfect accuracy in the high SNR situation for time-invariant signals. We also use regular mu instead of normalized mu as explained before.

```
        xk = x_arr(k:-1:k-L+1);
        output = (h')*xk;
        error = d_arr(k) - output;
%         mu_k = 0.5/(xk'*xk+0.001);
%         h = h + mu_k*conj(error)*xk;
        h = h+mu*conj(error)*(xk);
```

Filtering

To actually apply our filter for the give noisy signal, we can simply convolve h' with the given signals. Then we choose the L:end of the filtered signal in order to avoid filter delays.

```
QPSK = conv(h',V);
QPSK = QPSK(L:length(V));
```

QPSK to Text Conversion

The filtered signals are a sequence of complex values which are all QPSK symbols. Therefore, we then convert the result QPSK symbol sequence to text message based on the definition of QPSK. For the real values of QPSK symbol, above zero means a binary bit 1 and below is 0, same for the imaginary values. After that, we will have a sequence of demodulated binary values. We use `bin2dec and reshape` to generate our version of ascii values for the characters. Finally, just convert these custom ascii values back into text messages.

```
for index = 1:length(bs)
    b1 = real(bs(index));
    b2 = imag(bs(index));
    if(b1 < -Threshold)
        b1 = '1';
    elseif(b1 > Threshold)
        b1 = '0';
    else
        display('threshold situation detected')
        b1 = '1';
    end

    if(b2 < -Threshold)
        b2 = '1';
    elseif(b2 > Threshold)
        b2 = '0';
    else
        b2 = '1';
        display('threshold situation detected')
    end
    bits = [bits b1 b2];
end
demodBn = reshape(bits, bn_length , length(bits)/bn_length);
demodBn = demodBn';
```

```
ascii = bin2dec(demodBn);
```

Evaluation

The evaluation is straightforward. We use the text message from gettysburg.txt as ground truth and first convert it into custom ascii version and QPSK symbols. Then we compare our result of custom ascii version and QPSK symbols with the ground truth. We shit our result by 85 QPSK symbols because of the leading sequence in front of the given signals. If the length of leading sequence changes, the `head_signal` should also be updated.

```
head_signal = 85; %[0:31 4 5 asciiseq(find(~isnan(asciiseq))) 0 0 0 0 0 0],
[0:31 4 5] length is 34, 34*5/2 = 85
[QPSK_real, ascii] = file2bin(filename);
txt_real = char(ascii);
txt_test = bin2text(QPSK_test(head_signal+1-order_desired+1:end));
txt_test=txt_test(1:length(txt_real));
```

CONSTELLATION PLOT

Noise-only

We can achieve around 8.0% error rate give 7db SNR for noise-only signals. We use order 6 and 10000 iterations. If it is 6db, the error rate jumps up to 16.1%. The constellation plots for the "before" and "after" signals are shown below.

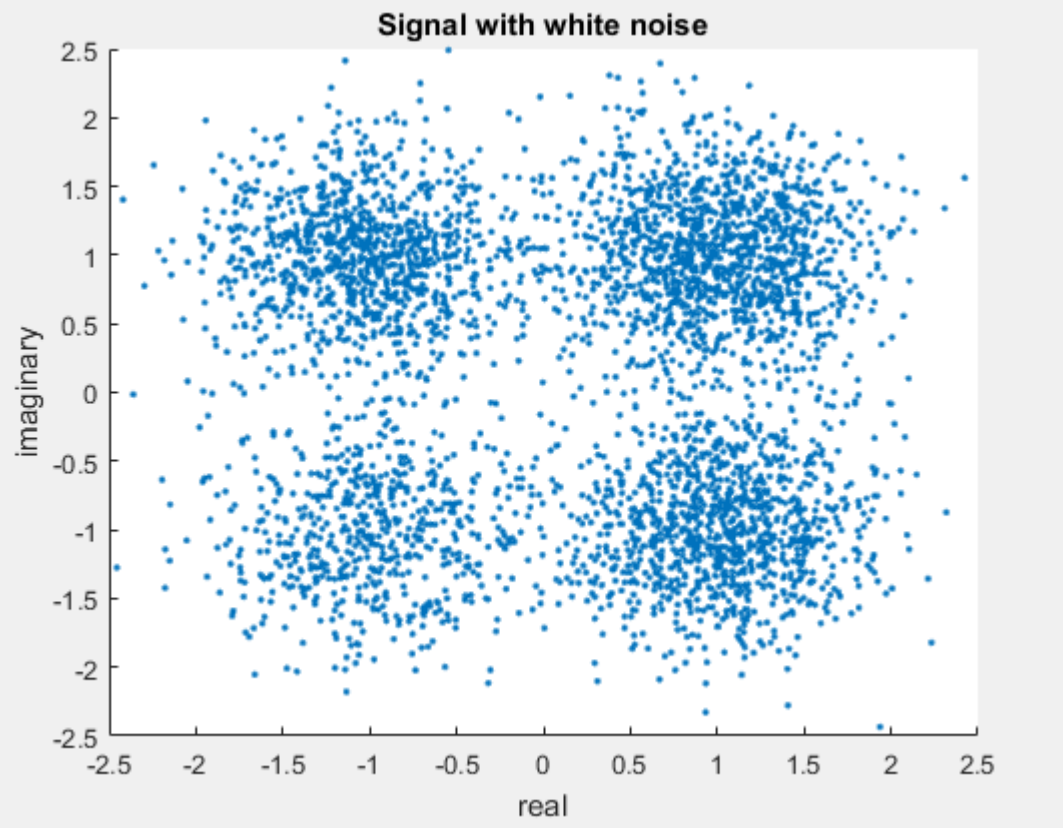

Figure 2. Constellation plot of SNR = 7db of noise-only signals before filtering

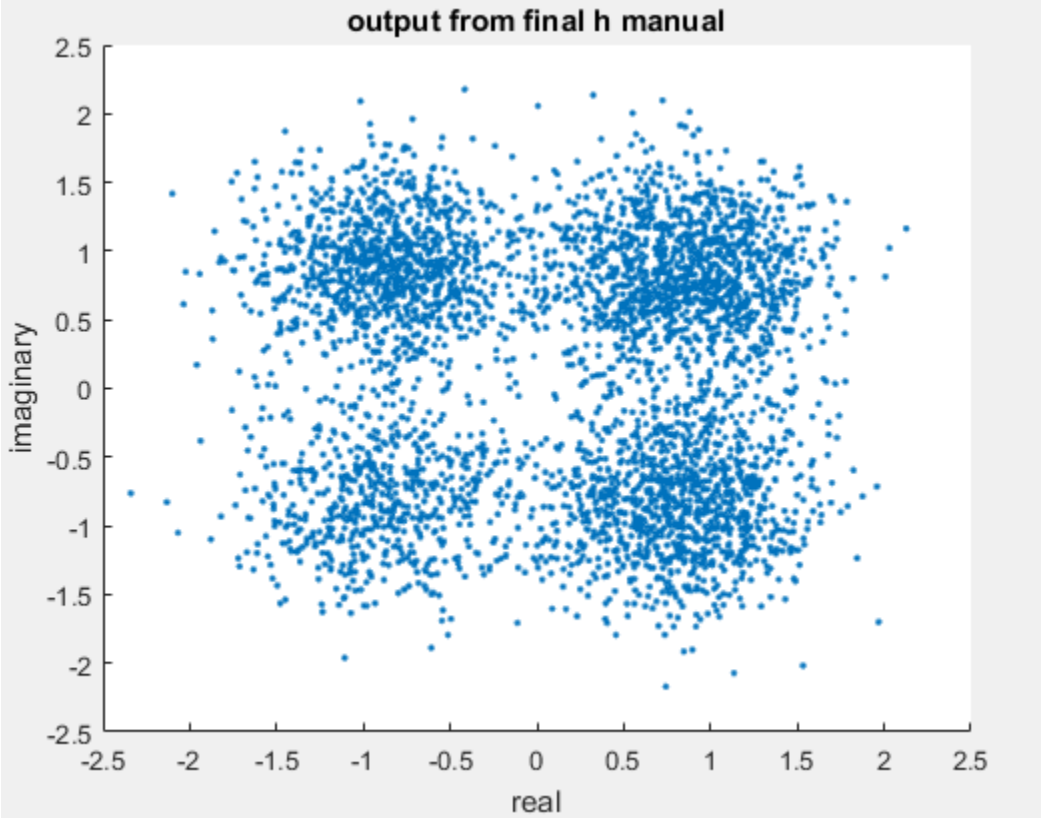

Figure 3. Constellation plot of SNR = 7db of noise-only signals after filtering

output from final h manually choose order 6
Error rate is 3.181258% over 4055 QPSK symbols
Error rate is 7.964602% over 1582 text

Time-Invariant

We can achieve around 8.5% error rate give 9db SNR for noise-only signals. We use order 6 and 10000 iterations. If it is 8db, the error rate jumps up to 19.9%. The constellation plots for the "before" and "after" signals are shown below.

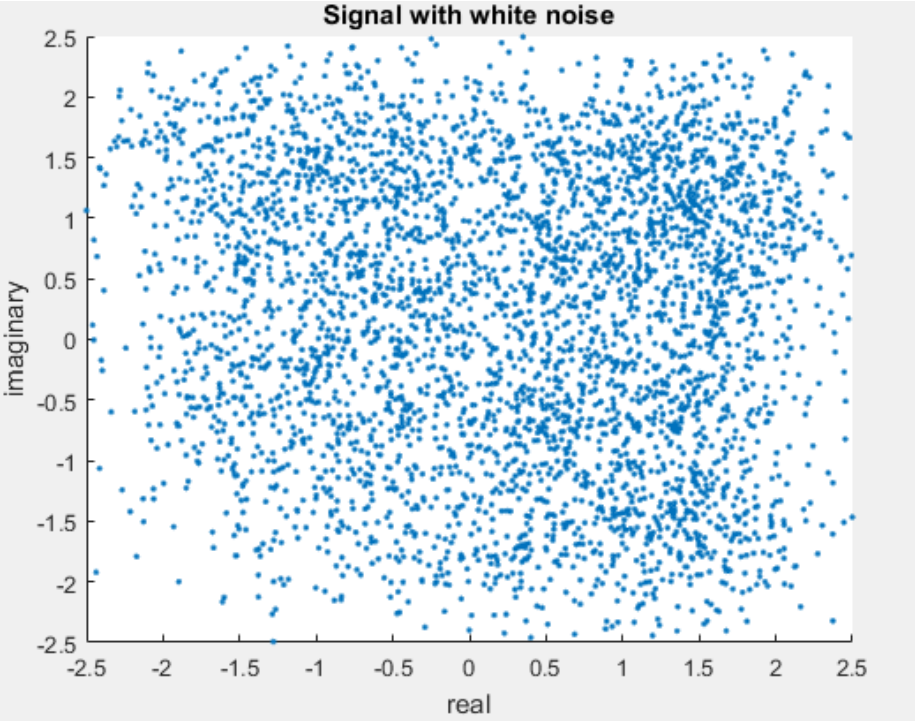

Figure 4. Constellation plot of SNR = 9db of time-invariant signals before filtering

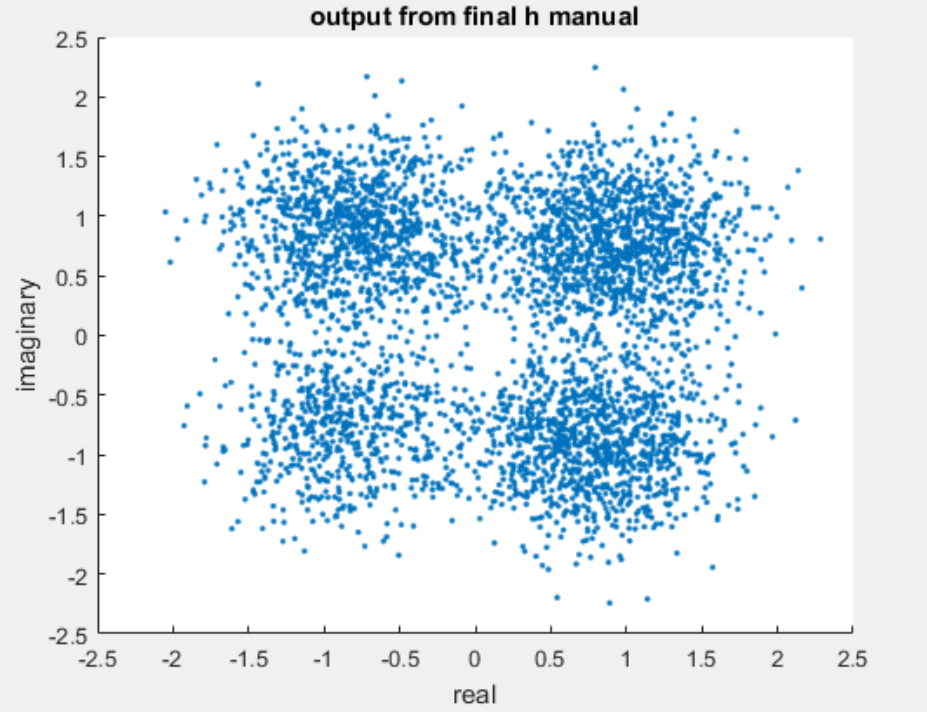

Figure 5. Constellation plot of SNR = 9db of time-invariant signals after filtering

output from final h manually choose order 6
Error rate is 3.575832% over 4055 QPSK symbols
Error rate is 8.470291% over 1582 text

Time-varying dispersion

We can achieve around 7.6% error rate give 13db SNR for time-varying signals. We use order 6 and 10000 iterations. If it is 12db, the error rate jumps up to 14.0%. If we use order 8 and 4000 iterations, the error rate drops to 4.9%, recalling what we discussed before. The constellation plots for the "before" and "after" signals are shown below.

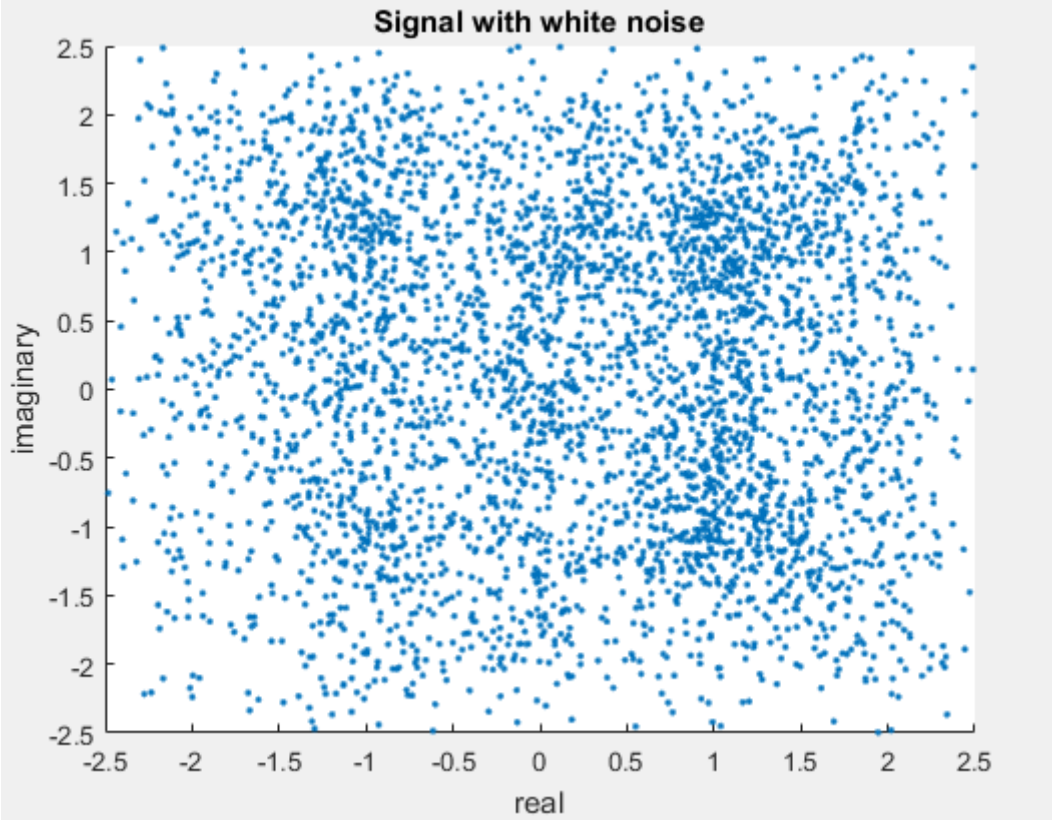

Figure 6. Constellation plot of SNR = 13db of time-varying signals before filtering

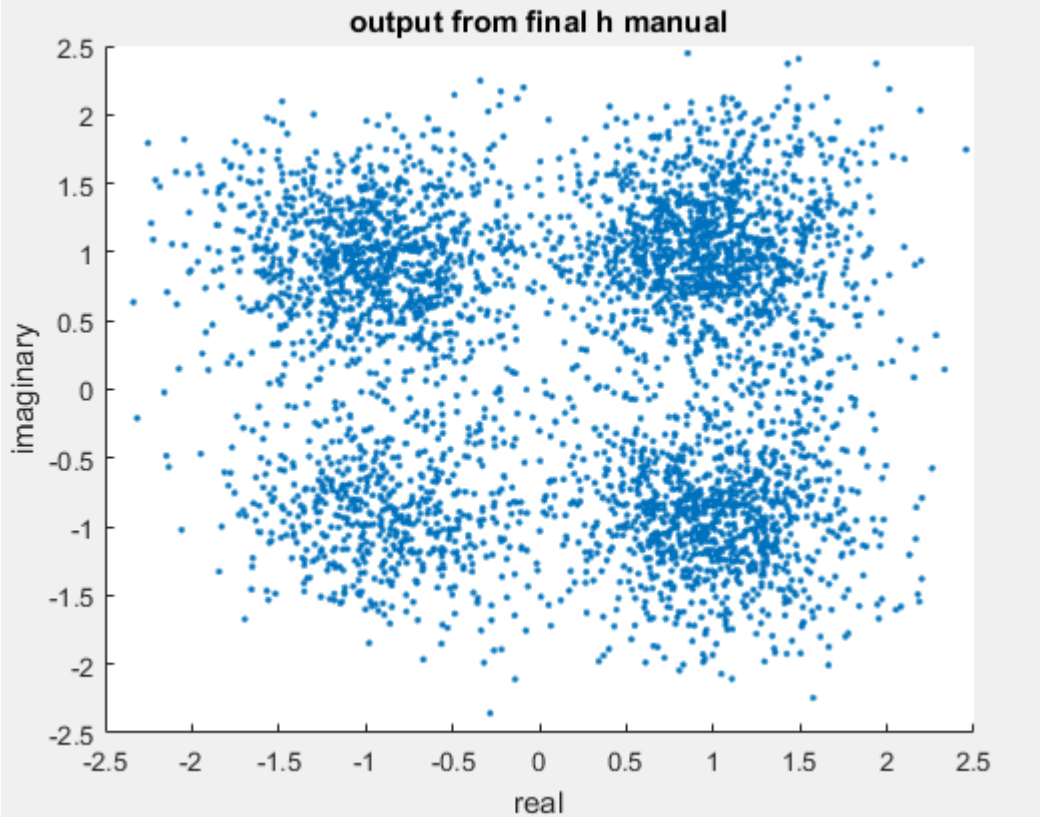

Figure 7. Constellation plot of SNR = 13db of time-varying signals after filtering

output from final h manually choose order 6
Error rate is 3.082614% over 4055 QPSK symbols
Error rate is 7.585335% over 1582 text

FINAL FILTER COEFFICIENT PLOT

Noise-only

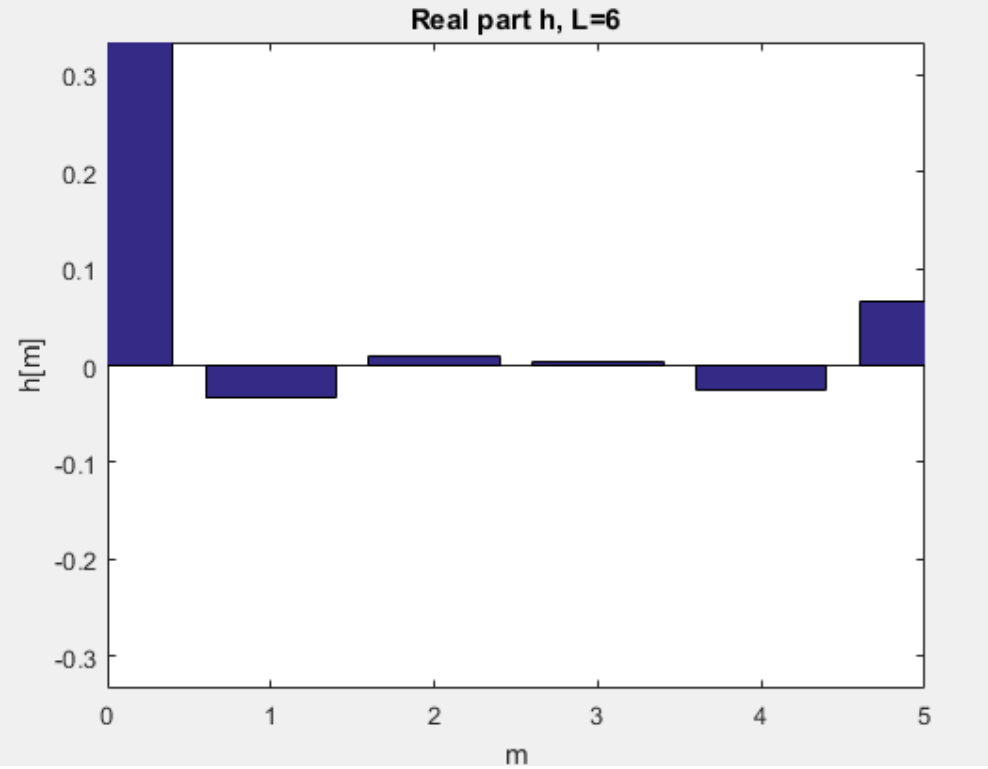

Figure 8. Final filter real coefficient of filter for 7db noise-only signals

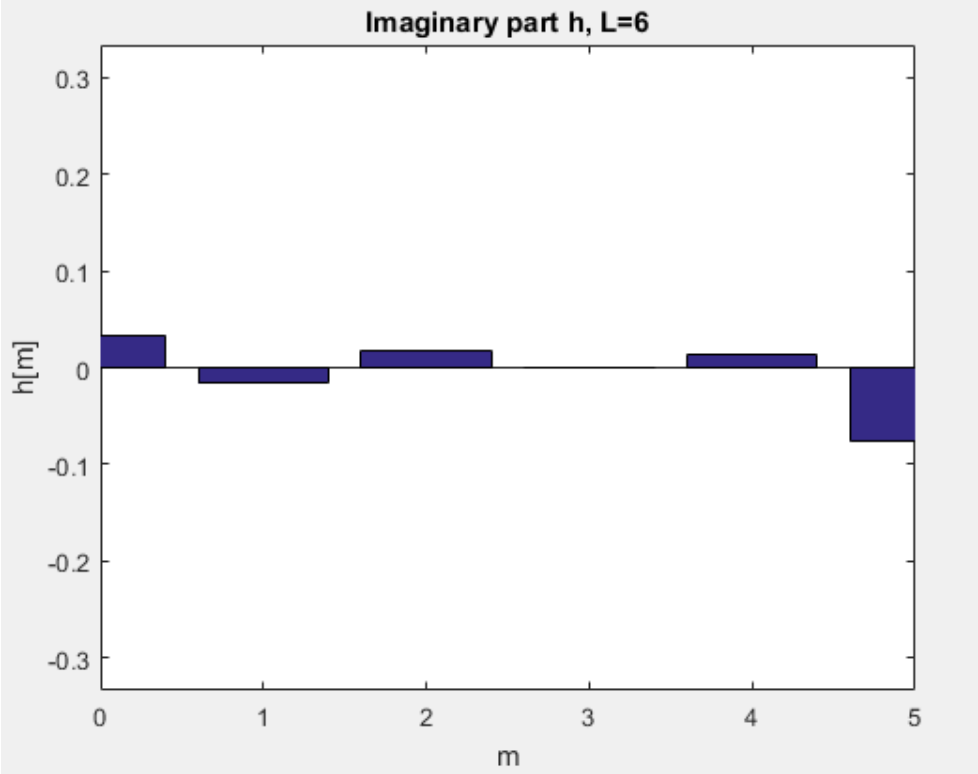

Figure 9. Final filter imaginary coefficient of filter for 7db noise-only signals

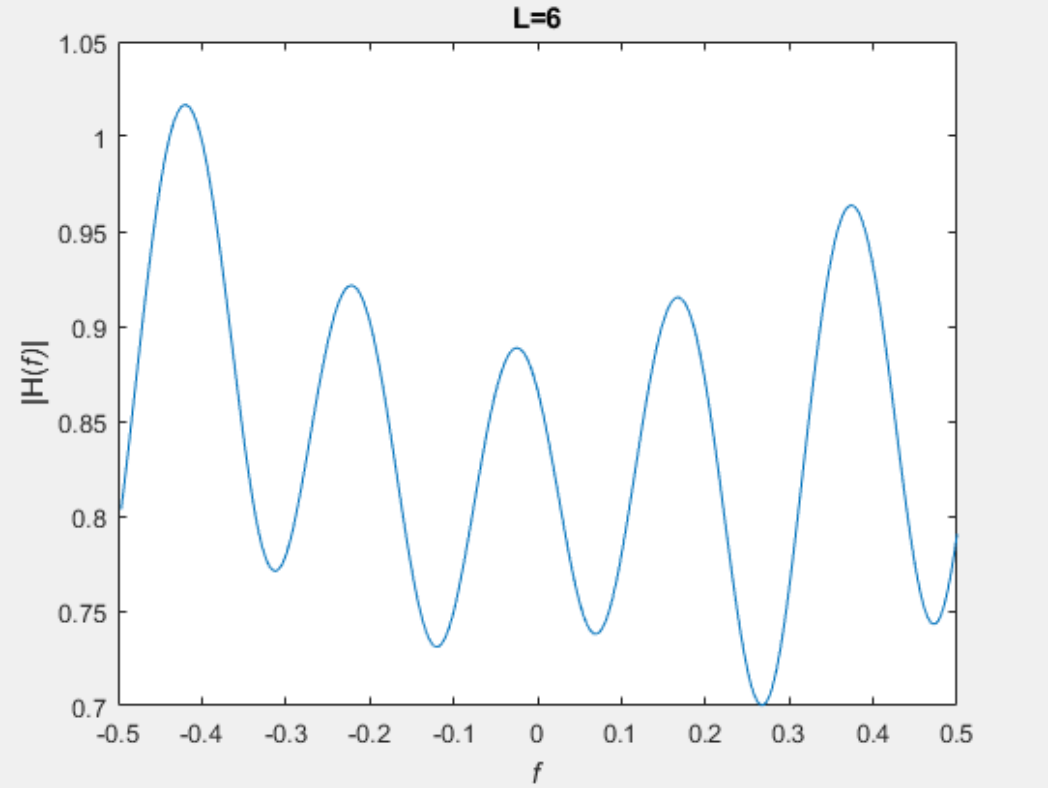

Figure 10. Final filter frequency response of filter for 7db noise-only signals

Time-Invariant

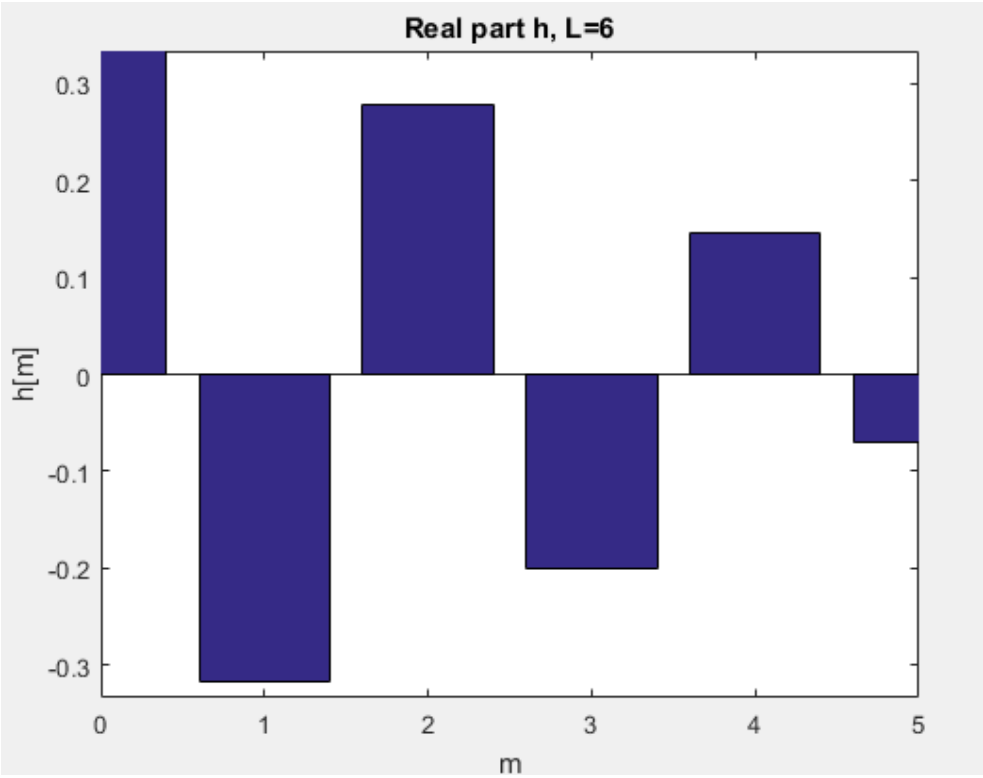

Figure 11. Final filter real coefficient of filter for 9db time-invariant signals

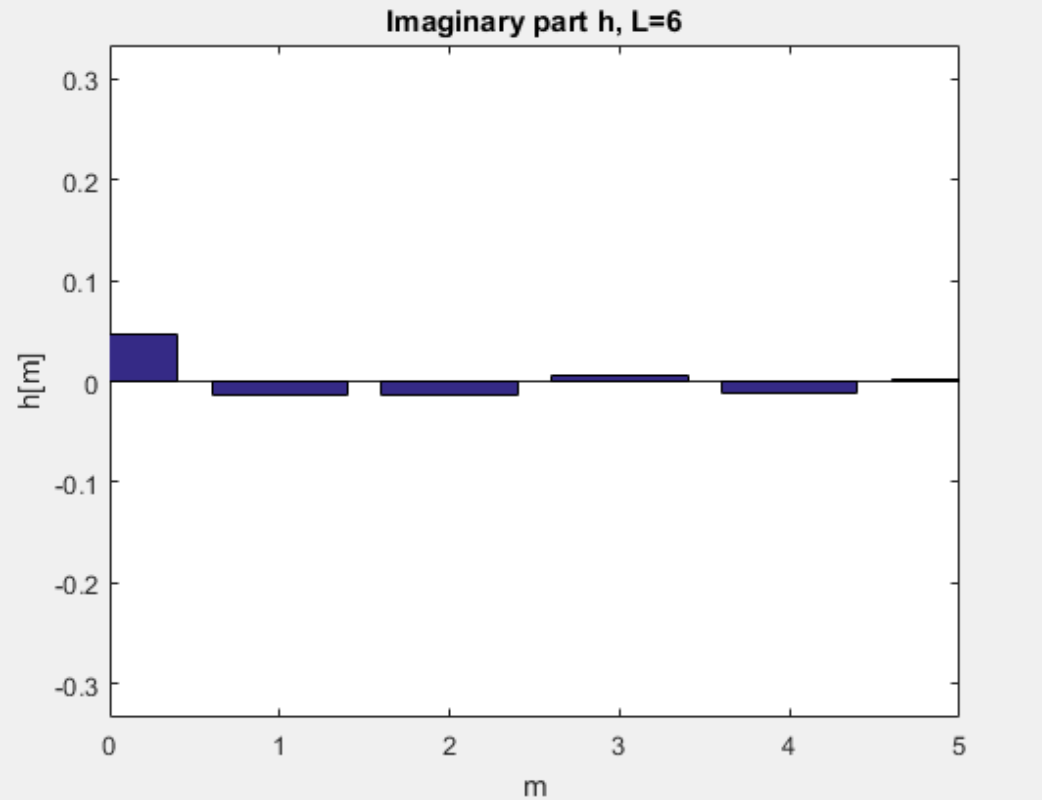

Figure 12. Final filter imaginary coefficient of filter for 9db time-invariant signals

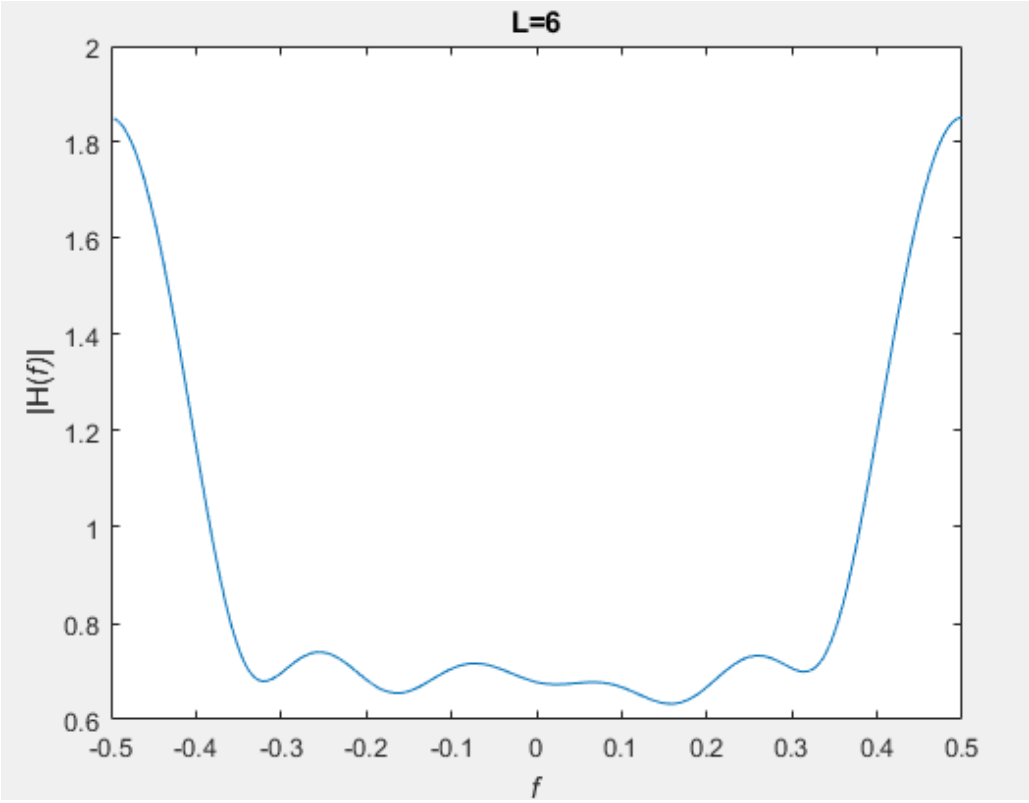

Figure 13. Final filter frequency response of filter for 9db time-invariant signals

Time-varying Dispersion

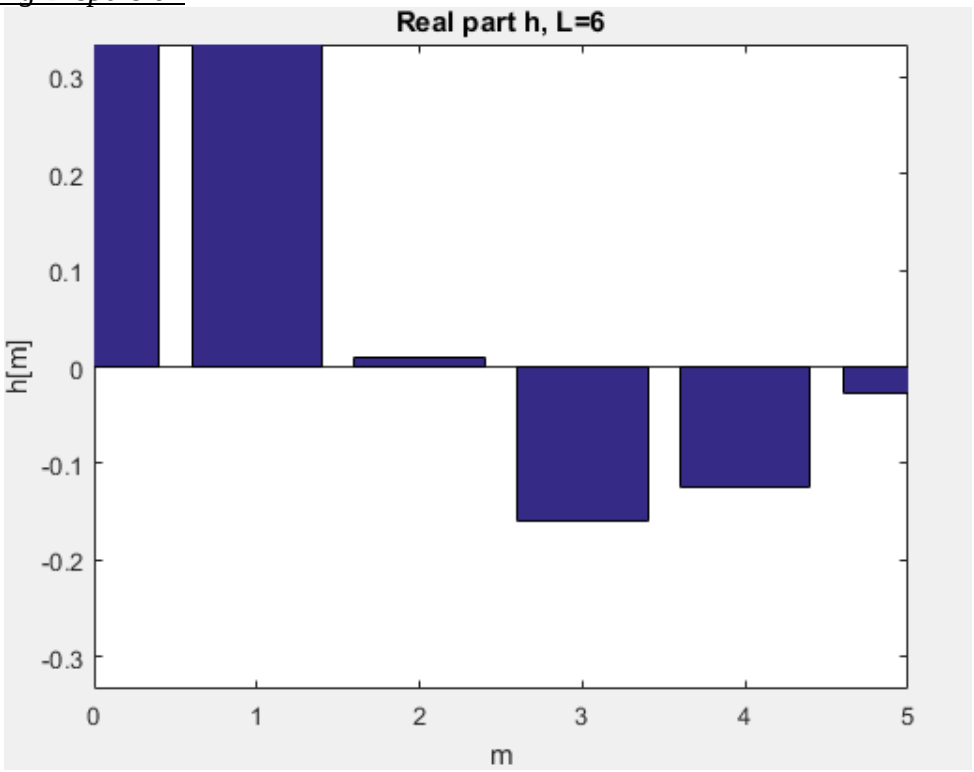

Figure 14. Final filter real coefficient of filter for 13db time-varying signals

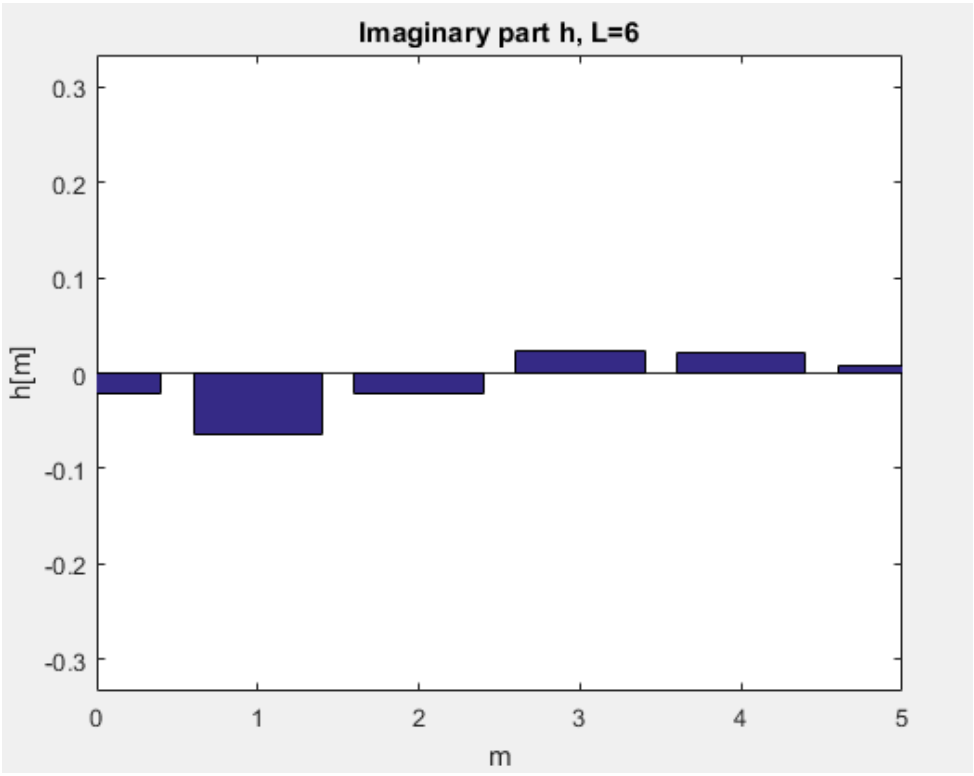

Figure 15. Final filter imaginary coefficient of filter for 13db time-varying signals

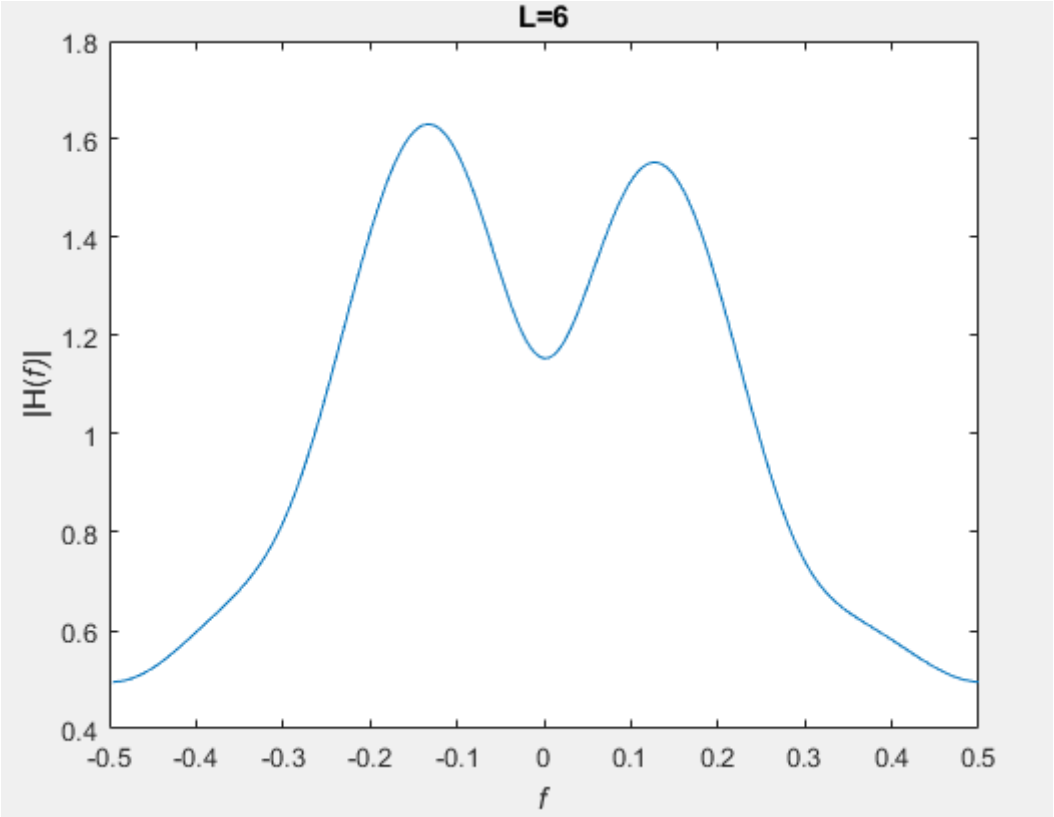

Figure 16. Final filter frequency response of filter for 13db time-varying signals

CODE LISTING

lms_decode(x)

```
function message = lms_decode(x)
V = x;
load desired
% [~,~,error,snr] = lms_filter(x,2,d);
% L = 2;
% E = error;
% SNR_QPSK = snr;
% for i = 3:40;
%     [~,~,error,snr] = lms_filter(x,i,d);
%     E = [E error];
%     SNR_QPSK = [SNR_QPSK,snr];
% end
% [min_E,order_desired_E] = min(E);
% [max_SNR,order_desired_SNR] = max(SNR_QPSK);
order_desired = 6;
% [QPSK_E,h_E,~,~] = lms_filter(V,order_desired_E,d);
% [QPSK_SNR,h_SNR,~,~] = lms_filter(V,order_desired_SNR,d);
[QPSK,h,~,~] = lms_filter(V,order_desired,d);
figure
bar(0:order_desired-1,real(h)); axis([0 order_desired-1,-
2/order_desired,2/order_desired]); title(sprintf('Real part h,
L=%i',order_desired))
```

```
xlabel('m'), ylabel('h[m]')
figure
bar(0:order_desired-1,imag(h)); axis([0 order_desired-1,-
2/order_desired,2/order_desired]); title(sprintf('Imaginary part h,
L=%i',order_desired))
xlabel('m'), ylabel('h[m]')
figure
H = abs(fftshift(fft(h,256)));
f = [1:256]/256-0.5;
plot(f,H);title(sprintf('L=%i',order_desired))
xlabel('\itf'),ylabel('|H(\itf\it)|')

figure
scatter(real(V),imag(V),'.')
xlim([-2.5,2.5])
ylim([-2.5,2.5])
title('Signal with white noise')
xlabel('real')
ylabel('imaginary')
% figure
% scatter(real(QPSK_E),imag(QPSK_E),'.');
% title('output from final h E');
% xlabel('real')
% ylabel('imaginary')
% figure
% scatter(real(QPSK_SNR),imag(QPSK_SNR),'.');
% title('output from final h SNR');
% xlabel('real')
% ylabel('imaginary')
figure
scatter(real(QPSK),imag(QPSK),'.');
xlim([-2.5,2.5])
ylim([-2.5,2.5])
title('output from final h manual');
xlabel('real')
ylabel('imaginary')

filename = 'gettysburg.txt';
% display(sprintf('output from final h_E choose order %i',order_desired_E))
% [txt_test_E, error_rate_QPSK_E, error_rate_txt_E] =
evaluation(filename,QPSK_E,order_desired_E);
% display(sprintf('output from final h_SNR choose
order %i',order_desired_SNR))
% [txt_test_SNR, error_rate_QPSK_SNR, error_rate_txt_SNR] =
evaluation(filename,QPSK_SNR,order_desired_SNR);
display(sprintf('output from final h manually choose
order %i',order_desired))
[txt_test, error_rate, error_rate] = evaluation(filename,QPSK,order_desired);
message = txt_test;
end
```

lms_filter(V,L,d)

```
function[QPSK,h,MSE,SNR_QPSK] = lms_filter(V,L,d)
N = length(V);
n = randn(length(V),1)+j*randn(length(V),1);n = n/sqrt(var(n));
SNRdB = snr(V,n);
SNR = 10^(SNRdB/10);
S = sqrt(2*SNR);
mu_max = 1/L/(S^2/2);
mu = mu_max/100;
h = zeros(L,1);
% h = 2*(rand(L,1)-0.5)*(2/L);
% E = [];

x_arr = [];% d is desired signal
d_arr =[];
x = V(1:length(d));
for i = 1: ceil(N/L)
    x_arr = [x_arr;x];
    d_arr = [d_arr;d];
end
MSE_arr = [];
iterations = 1000:1000:10000;
% iterations = length(V);
for N = iterations
    for k = L:N
        xk = x_arr(k:-1:k-L+1);
        output = (h')*xk;
        error = d_arr(k) - output;
%         mu_k = 0.5/(xk'*xk+0.001);
%         h = h + mu_k*conj(error)*xk;
        h = h+mu*conj(error)*(xk);
    end
self_test = conv(h',x);
E = d - self_test(L:end);
MSE = sum(abs(E).^2)/N;
MSE_arr = [MSE_arr MSE];
end
% plot(iterations,MSE_arr)
% title('MSE vs iterations')
% xlabel('iterations')
% ylabel('MSE')

QPSK = conv(h',V);
QPSK = QPSK(L:length(V));
SNR_QPSK = snr(QPSK,n(L:length(V)));
end
```

genSignal.m

```
load gettysburg
load moreSignals
x10=genChannel(s,0,1,10);
x10_ti=genChannel(s,1,1,10);
x10_tv=genChannel(s,2,1,10);
x6=genChannel(s,0,1,6);
x6_ti=genChannel(s,1,1,6);
x6_tv=genChannel(s,2,1,6);
x7=genChannel(s,0,1,7);
x7_ti=genChannel(s,1,1,7);
x7_tv=genChannel(s,2,1,7);
x8=genChannel(s,0,1,8);
x8_ti=genChannel(s,1,1,8);
x8_tv=genChannel(s,2,1,8);
x9=genChannel(s,0,1,9);
x9_ti=genChannel(s,1,1,9);
x9_tv=genChannel(s,2,1,9);
x11=genChannel(s,0,1,11);
x11_ti=genChannel(s,1,1,11);
x11_tv=genChannel(s,2,1,11);
x12=genChannel(s,0,1,12);
x12_ti=genChannel(s,1,1,12);
x12_tv=genChannel(s,2,1,12);
x13=genChannel(s,0,1,13);
x13_ti=genChannel(s,1,1,13);
x13_tv=genChannel(s,2,1,13);
x5=genChannel(s,0,1,5);
x5_ti=genChannel(s,1,1,5);
x5_tv=genChannel(s,2,1,5);
x0=genChannel(s,0,1,0);
x0_ti=genChannel(s,1,1,0);
x0_tv=genChannel(s,2,1,0);
save('AllSignals','x0','x0_ti','x0_tv','x5','x5_ti','x5_tv','x6','x6_ti','x6_
tv','x7','x7_ti','x7_tv','x8','x8_ti','x8_tv','x9','x9_ti','x9_tv','x10','x10
_ti','x10_tv','x12','x12_ti','x12_tv','x11','x11_ti','x11_tv','x13','x13_ti',
'x13_tv','x15','x15_ti','x15_tv','x20','x20_ti','x20_tv')
```

AND
genChannel.m and channel.p provide by the professor

evaluation(filename,QPSK_test,order_desired)

```
function [txt_test, error_rate_QPSK,
error_rate_txt]=evaluation(filename,QPSK_test,order_desired)
head_signal = 85; %[0:31 4 5 asciiseq(find(~isnan(asciiseq))) 0 0 0 0 0 0],
[0:31 4 5] length is 34, 34*5/2 = 85
[QPSK_real, ascii] = file2bin(filename);
txt_real = char(ascii);
txt_test = bin2text(QPSK_test(head_signal+1-order_desired+1:end));
txt_test=txt_test(1:length(txt_real));

dot_idx = find(filename == '.');
```

```
outfilename = strcat(filename(1:dot_idx-1),'_out',filename(dot_idx:end));
fileID_out = fopen(outfilename,'w');
fwrite(fileID_out,txt_test,'uint8');
fclose(fileID_out);
[QPSK_test, ~] = file2bin(outfilename);
error_rate_QPSK = length(find(QPSK_real ~= QPSK_test))/length(QPSK_real);
error_rate_txt = length(find(txt_real ~= txt_test))/length(txt_real);
display(sprintf('Error rate is %f%% over %d QPSK
symbols',error_rate_QPSK*100,length(QPSK_real)));
display(sprintf('Error rate is %f%% over %d
text',error_rate_txt*100,length(txt_real)));
end
```

bin2text(bs)

```
function text = bin2text(bs)
bn_length = 5;
Threshold = 0;
bits = [];
for index = 1:length(bs)
    b1 = real(bs(index));
    b2 = imag(bs(index));
    if(b1 < -Threshold)
        b1 = '1';
    elseif(b1 > Threshold)
        b1 = '0';
    else
        display('threshold situation detected')
        b1 = '1';
    end

    if(b2 < -Threshold)
        b2 = '1';
    elseif(b2 > Threshold)
        b2 = '0';
    else
        b2 = '1';
        display('threshold situation detected')
    end
    bits = [bits b1 b2];
end
demodBn = reshape(bits, bn_length , length(bits)/bn_length);
demodBn = demodBn';
ascii = bin2dec(demodBn);
ascii = ascii';
ascii(ascii >= 6 & ascii <= 31) = ascii(ascii >= 6 & ascii <= 31) + 91;% bn
to a-z
ascii(ascii == 0) = 32;% bn to ' '
ascii(ascii == 1) = 44;% bn to ','
ascii(ascii == 2) = 46;% bn to '.'
ascii(ascii == 3) = 39;% bn to '''
ascii(ascii == 4) = 10;% bn to line feed
ascii(ascii == 5) = 13;% bn to carriage return
text = char(ascii);
end
```

file2bin(file)

```
function [x,ascii] = file2bin(file)

% 'file' should be the filename of a simple text file,
%      including extension, e.g. foo.txt

% read file contents. Every call to fgets retireves one line of text.
% the loop concatenates them, making a variable s were every row is a text
% string correspodning to one line of thetext.
s=[];
fid=fopen(file,'r');
while (~feof(fid));
  s=[s fgets(fid)];
end

as  = double(lower(char(s))); % convert to lower case, then to numeric ASCII
code

% find indices that have spaces, line feeds, and so forth, and those that
% have regular characters
lindex = find((as>96) & (as<123));  % lower case letters
spindex = find(as==32);
lfindex = find(as==10);
rindex  = find(as==13);
cindex  = find(as==44);
pindex  = find(as==46);
qindex  = find(as==39);

% recode the sequence so everyting fits in the range 0:31 and therefore
% into 5 bits.  Anything that isn't space, comma, period, apostropher,
% line feed, return, or 'a' through 'z' is left as a NaN.
asciiseq = NaN*ones(1,length(as));
asciiseq(lindex)  = as(lindex)-91;
asciiseq(spindex) = 0;
asciiseq(lfindex) = 4;
asciiseq(rindex)  = 5;
asciiseq(cindex)  = 1;
asciiseq(pindex)  = 2;
asciiseq(qindex)  = 3;

ascii = NaN*ones(1,length(as));
ascii(lindex)  = as(lindex);
ascii(spindex) = as(spindex);
ascii(lfindex) = as(lfindex);
ascii(rindex)  = as(rindex);
ascii(cindex)  = as(cindex);
ascii(pindex)  = as(pindex);
ascii(qindex)  = as(qindex);
ascii = ascii(find(~isnan(asciiseq)));

% build a new sequence that has the training sequence first, then all
% characters that didn't get left as NaNs, then five spaces.
asciiseq = [0:31 4 5 asciiseq(find(~isnan(asciiseq))) 0 0 0 0 0 0];
```

```
% drop off one of the trailing spaces if need be to make the length even
l=floor(length(asciiseq)/2)*2;
asciiseq = asciiseq(1:l);

%convert characters to binary strings, then to 2-bit symbols. Each column
%of bb is one pair of bits
bb=reshape(dec2bin(asciiseq)',2,length(asciiseq)*5/2);

%QPSK encoding
x = ((-1).^bb') * [1;j];
```

main.m

```
clear
load gettysburg;
load AllSignals;
V = x20;
message = lms_decode(V);
% 7 9 13
```