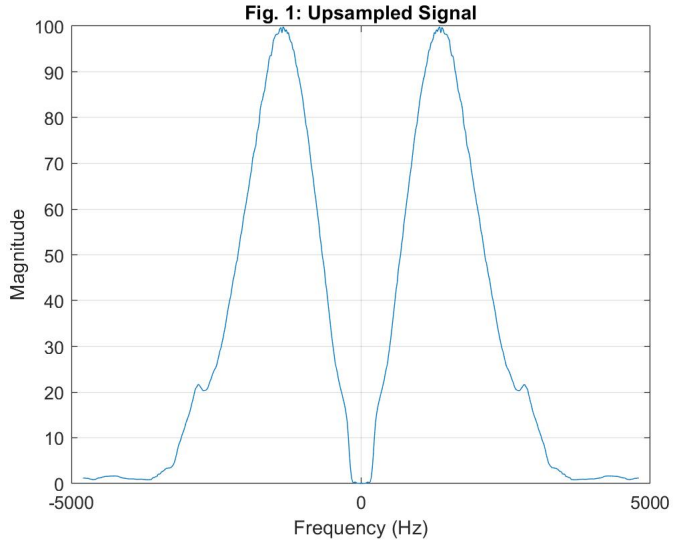
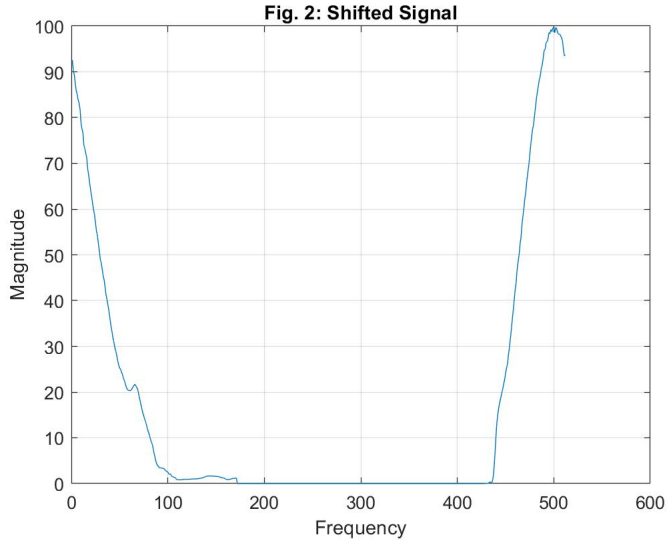
EC ENGR 113, Fall 2021: Kellen Cheng (905155544)
Channel Project
Due 3 December, 2021 at 11:00 PM

1 Complex Baseband Representation of Channel c0

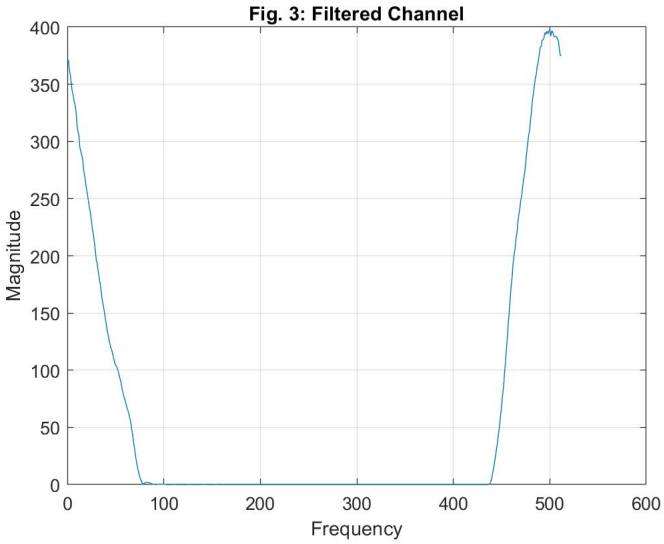
(a) **Solution:** Below is the plot of the absolute values of the fast Fourier transform of the channel samples (i.e. Figure 1). Note that the frequency peaks are roughly in the 1600 Hz vicinity.



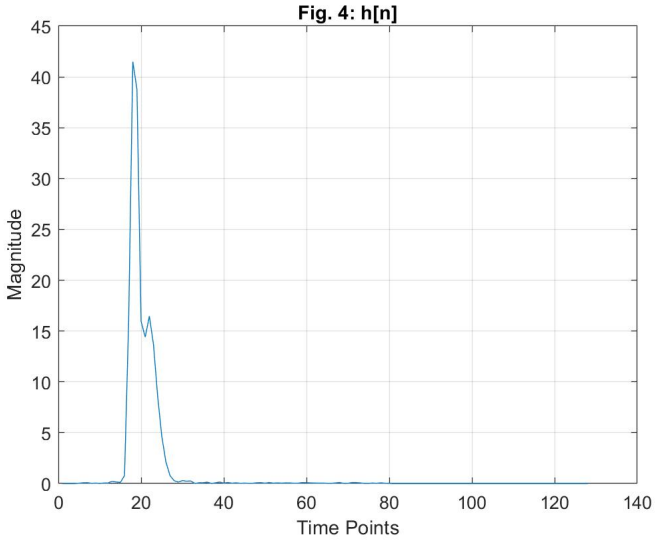
(b) **Solution:** Below is the plot of the cyclically shifted frequencies (i.e. Figure 2). Note that the shift value in the code is determined from the lecture 16 class notes posted on CCLE.



(c) **Solution:** Below is the plot of the SRRC filtered frequencies (i.e. Figure 3). Note that the causal square root raised cosine filter has been truncated and recomputed for values under the 30 dB point.

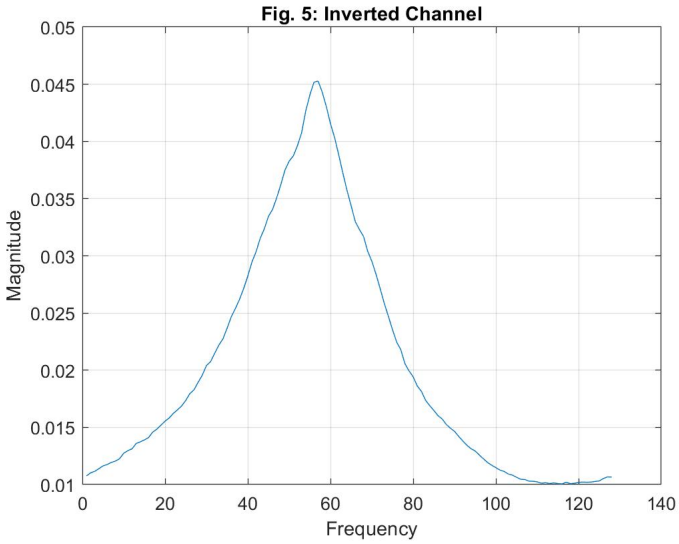


(d) **Solution:** Below is the plot of the downsampled signal in the time domain (i.e. Figure 4). Note that the downsample occurs by a factor of 4 to 2400 Hz, resulting in an hpower of 4507.

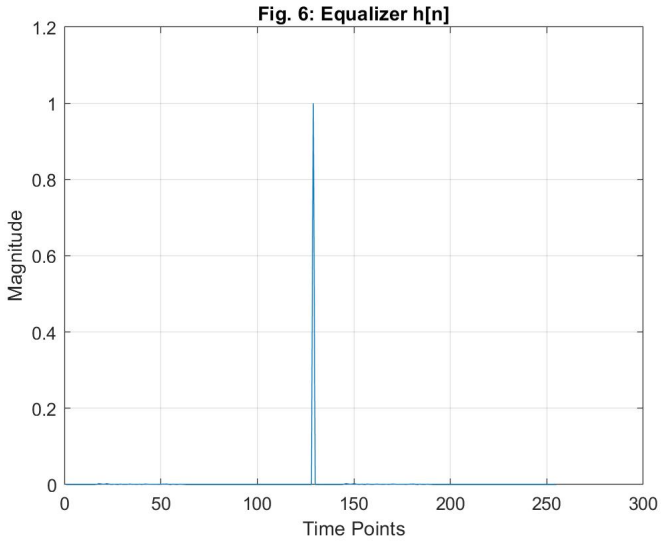


2 Fixed Linear Equalizers

(a) **Solution:** Below is the plot of the inverted frequencies of our complex channel (i.e. Figure 5).



(b) **Solution:** Below is the plot of the equalizer response in the time domain (i.e. Figure 6). Note that the delay of the impulse occurs at a time point of 129. Additionally, compared to the previous impulse we plotted [in Part 1], note that there is strong interference suppression, evidenced by the cleaner graph and also by a large SIR of 10326.



(c) **Solution:** Below is the plot of the frequency magnitudes for the minimum mean squared error linear equalizer (i.e. Figure 7). Note that the optimal delay is calculated to be 52, resulting in an optimal SIR of 1545300000 (also $1.5453 * 10^9$). Compared to the zero-forcing linear equalizer, it is evident that the minimum mean squared error equalizer produces a much larger SIR and a smaller delay, as it finds the more optimal solution.

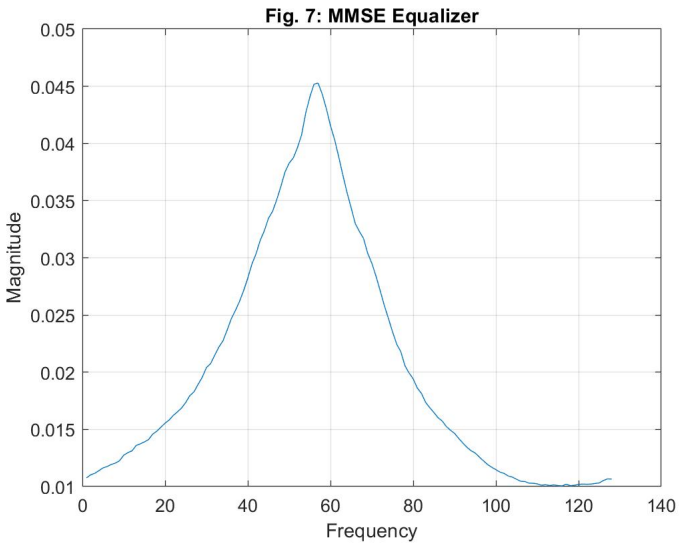
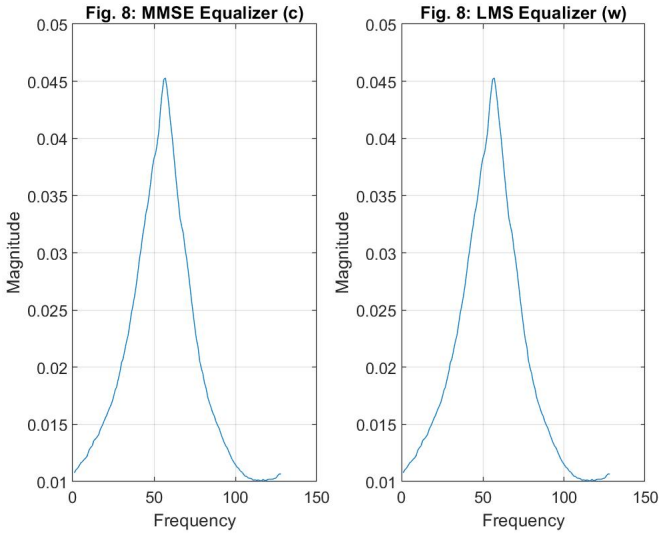


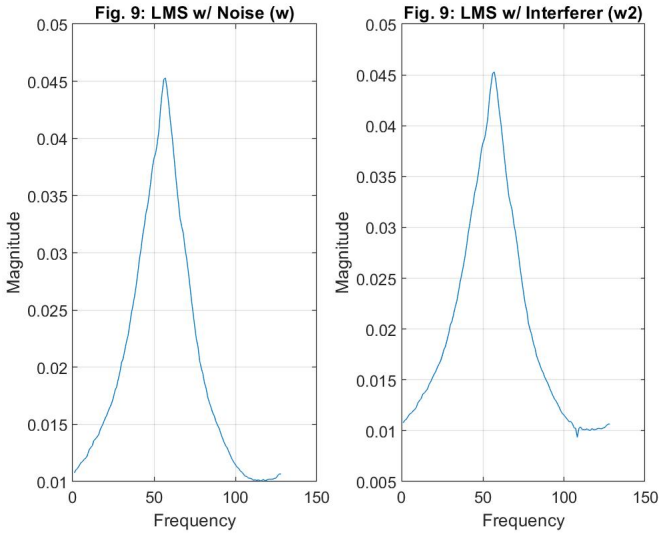
Fig. 7: MMSE Equalizer
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
Magnitude
0
20
40
60
80
100
120
140
Frequency

3 Adaptive Equalizers

(a) **Solution:** Below is the plot of the Wiener Filter frequencies compared to the minimum mean squared error equalizer (i.e. Figure 8). Note that the calculated SINR is 1415500000 (also $1.4155 * 10^9$), which is in roughly the same order of magnitude as our result from the minimum mean squared error equalizer.



(b) **Solution:** Below is the plot of the Wiener Filter frequencies computed with noise alone compared to the Wiener Filter frequencies computed with the interferer (i.e. Figure 9). Note that the calculated SINR is 1045.3 (also $1.0453 * 10^3$), which is much smaller due to the sheer magnitude of our interferer. Notice that there is still a sufficient amount of suppression in the interference region, such that the resultant plot is still very similar to the calculated filter with noise alone (although differences in the graph can be seen at the edge frequencies). Additionally, this adaptive equalizer is much better suited to handle this scenario than the cascade of a notch filter and zero-forcing equalizer, since inverting the notch filter in the frequency domain results in massive amplification at the interference (i.e. the exact opposite behavior we want).



(c) **Solution:** N.B. For parts (a) and (b) of this section of the final project, the "del" parameter (i.e. our learning rate for stochastic gradient descent) is increased in order to converge to a solution at a faster rate. For iteration number, both parts utilized 2 million iterations.

4 Appendix: MATLAB Codes

(a) **Solution:** Note that all project sections are demarcated into different coding sections in the MATLAB code. Refer to comments when the algorithm logic may appear unclear.

```
% srrc.m
function srrc = srrc(beta, n)
    if n == 0
        srrc = (4*beta + pi - beta*pi) / pi;
    else
        constant = 4 * beta / pi;
        numerator = cos((1+beta)*pi*n/4) + sin((1-beta)*pi*n/4) / (n*beta);
        denominator = 1 - (beta*n)^2;
        srrc = constant * numerator / denominator;
    end
end
```

```
% channel_project.m
%% Load Prerequisites
M = readmatrix("c0.txt")';
M = M(1:end-2)'; N = 512; L = 256;
%% Project Channel Response (Part 1)
% Upsample to 9600 Hz (i.e. 299 samples) -> Zero-stuff to 512 samples
c_n = interp1([1:1:L], M(:, 1), linspace(1, L, 299), "spline");
c_n = [c_n zeros(1, (N - length(c_n)))];

% Take the FFT and plot the magnitudes
C_k = fftshift(fft(c_n));
figure(); plot(linspace(-4800, 4800, 512), abs(C_k)); title("Fig. 1: Upsampled Signal");
xlabel("Frequency (Hz)"); ylabel("Magnitude"); grid on;

% Zero out negative frequencies -> Cyclically shift carrier frequency
C_k(1:256) = zeros(1, 256);
C_k = circshift(C_k, -85-256); % Shift number determined from lec. 16

figure(); plot(abs(C_k)); title("Fig. 2: Shifted Signal");
xlabel("Frequency"); ylabel("Magnitude"); grid on;
%% Project SRRC Response (Part 1)
% Initialize full-length SRRC filter
srrc_signal = zeros(1, N); beta = 0.15;
for i=-256:255
    srrc_signal(i+257) = srrc(beta, i);
end

% Determine 30 dB cutoff point
thirty_dB = srrc_signal(257) / sqrt(1000);
idx = 1;
for i=1:257
    % Traverse from tail to peak to determine cutoff
```

```
    if abs(srrc_signal(i)) > thirty_dB
        break;
    else
        idx = i;
    end
end

% Initialize truncated SRRC filter
idx = idx + 1; idx2 = 2*(257 - idx) + 1;
srrc_new = zeros(1, idx2);
for i=(-floor(idx2 / 2)):(floor(idx2 / 2))
    srrc_new(i + (floor(idx2 / 2)) + 1) = srrc(beta, i);
end

% Zero-stuff to 512 -> Take the FFT
srrc_new = [srrc_new zeros(1, (N - length(srrc_new)))];
srrc_filter = (fft(srrc_new));

% Multiply and plot the filtered channel
channel = srrc_filter .* C_k;
figure(); plot(abs(channel)); title("Fig. 3: Filtered Channel");
xlabel("Frequency"); ylabel("Magnitude"); grid on;

% Find and plot the impulse response
channel_n = ifft(channel);
h_n = channel_n(1:4:end); % Downsample by factor of 4

figure(); plot(abs(h_n)); title("Fig. 4: h[n]");
xlabel("Time Points"); ylabel("Magnitude"); grid on;

% Compute power for noise normalization
hpower = sum(abs(h_n).^2); % hpower = 4507 or 4.5070e3
%% Project Zero-Forcing Equalizer (Part 2)
% Find and plot the inverted channel
G_k = 1 ./ fft(h_n);
figure(); plot(abs(G_k)); title("Fig. 5: Inverted Channel");
xlabel("Frequency"); ylabel("Magnitude"); grid on;

% Find and plot the equalizer
zf_equalizer = conv(ifft(G_k), h_n);
figure(); plot(abs(zf_equalizer)); title("Fig. 6: Equalizer h[n]");
xlabel("Time Points"); ylabel("Magnitude"); grid on;

% Compute the SIR
msav = max(abs(zf_equalizer).^2);
SIR = msav / (sum(abs(zf_equalizer).^2) - msav); % SIR = 10,326 or 1.0326e4
%% Project MMSE Linear Equalizer (Part 2)
% Initialize MMSE loop parameters
```

```
R = zeros(128, 128); hd = zeros(128, 1); nvar = 1 / (sum(abs(h_n).^2) * 2000);
SIR1 = -1; c = -1; d = -1; mmse_h = -1;

% Initialize R matrix
for i=1:128
    for j=1:128
        val = 0;

        for k=1:256
            % If indices invalid, skip iteration
            if (k+1-i < 1 || k+1-i > 128 ||k+1-j < 1 || k+1-j > 128)
                continue
            end

            val = val + conj(h_n(k+1-i)) * h_n(k+1-j);
            % Add noise if diagonal element
            if i==j
                val = val + nvar;
            end
        end

        R(i, j) = val;
    end
end

% Calculate optimum delay
for k=1:128
    hd_tmp = zeros(128, 1); % Placeholder hd array

    for i=1:128
        if (k+1-i >= 1) && (k+1-i <= 128)
            hd_tmp(i) = h_n(k+1-i);
        end
    end

    % Convolve with the channel to compute SIR
    c_tmp = R \ conj(hd_tmp);
    mmse_h_tmp = conv(c_tmp, h_n);

    % Compare to optimum values
    m = max(abs(mmse_h_tmp).^2);
    SIR_tmp = m / (sum(abs(mmse_h_tmp).^2) - m);

    % Store the optimal values
    if SIR_tmp > SIR1
        SIR1 = SIR_tmp;
        d = k;
        hd = hd_tmp;
```

```
        c = c_tmp;
        mmse_h = mmse_h_tmp;
    end
end

% Find and plot optimal fft(c) -> delay = 52
mmse_H = fft(c); % SIR = 1,545,300,000 or 1.5453e9
figure(); plot(abs(mmse_H)); title("Fig. 7: MMSE Equalizer");
xlabel("Frequency"); ylabel("Magnitude"); grid on;
%% Project LMS Adaptive Equalizer (Part 3)
% Initialize LMS algorithm loop parameters
w = zeros(128, 1); w(d) = 1; del = 2 / (128 * hpower * 100); j = sqrt(-1);
data = zeros(128, 1); u = zeros(128, 1); squared_error = 0; lim = 2000000;
errors = []; % Keeps track of error through iterations

for iter=1:lim
    if mod(iter, 10000) == 0
        disp(iter);
    elseif mod(iter, 1000) == 0
        errors = [errors e];
    end

    % Randomly compute sampled data
    val = randsample([-1, 1], 1) * (1/sqrt(2)) + randsample([-1, 1], 1) * (j/sqrt(2));

    % Shift sampled data into data array, adding new vals in front
    data = circshift(data, 1); data(1) = val;

    % Compute zero mean complex Gaussian noise
    z = (normrnd(0, sqrt(nvar)) + j*normrnd(0, sqrt(nvar))) / sqrt(2);

    % Compute channel output
    val = 0;
    for i=1:128
        val = val + h_n(i) * data(i);
    end

    % Shift channel output into output array, adding new vals in front
    u = circshift(u, 1); u(1) = val + z;

    % Calculate the filter error
    y = 0;
    for i=1:128
        y = y + u(i) * w(i);
    end
    e = data(d) - y;

    % Calculate squared error when appropriate
```

```
    if iter >= 0.95 * lim
        squared_error = squared_error + abs(e)^2;
    end

    % Adaptively reduce learning rate
    if iter > 1000000
        del = 2 / (128 * hpower * 1000);
    end

    % Update step in stochastic gradient descent
    w = w + del * e * conj(u);
end

SINR = 0.05 * lim / squared_error; % SINR = 1,415,500,000 or 1.4155e9

% Plot FFT comparison between c and w
figure();
subplot(1, 2, 1); plot(abs(mmse_H)); title("Fig. 8: MMSE Equalizer (c)");
xlabel("Frequency"); ylabel("Magnitude"); grid on;
subplot(1, 2, 2); plot(abs(fft(w))); title("Fig. 8: LMS Equalizer (w)");
xlabel("Frequency"); ylabel("Magnitude"); grid on;
%% Project LMS Adaptive w/ Interference Equalizer (Part 3)
w2 = zeros(128, 1); w2(d) = 1; data2 = zeros(128, 1); u2 = zeros(128, 1);
j = sqrt(-1); squared_error2 = 0; lim = 2000000;
errors2 = []; % Keeps track of error through iterations

for iter=1:lim
    if mod(iter, 10000) == 0
        disp(iter);
    elseif mod(iter, 1000) == 0
        errors2 = [errors2 e];
    end

    % Randomly compute sampled data
    val = randsample([-1, 1], 1) * (1/sqrt(2)) + randsample([-1, 1], 1) * (j/sqrt(2));

    % Shift sampled data into data array, adding new vals in front
    data2 = circshift(data2, 1); data2(1) = val;

    % Compute channel output
    val = 0;
    for i=1:128
        val = val + h_n(i) * data2(i);
    end
    val = val + sqrt(10) * exp(-sqrt(-1) * pi * iter / 3);

    % Shift channel output into output array, adding new vals in front
    u2 = circshift(u2, 1); u2(1) = val;
```

```
    % Calculate the filter error
    y = 0;
    for i=1:128
        y = y + u2(i) * w2(i);
    end
    e = data2(d) - y;

    % Calculate squared error when appropriate
    if iter >= 0.95 * lim
        squared_error2 = squared_error2 + abs(e)^2;
    end

    % Update step in stochastic gradient descent
    w2 = w2 + del * e * conj(u2);
end

SINR2 = 0.05 * lim / squared_error2; % SINR2 = 1,045.3 or 1.0453e3

% Plot FFT comparison between noise and interference
figure();
subplot(1, 2, 1); plot(abs(fft(w))); title("Fig. 9: LMS w/ Noise (w)");
xlabel("Frequency"); ylabel("Magnitude"); grid on;
subplot(1, 2, 2); plot(abs(fft(w2))); title("Fig. 9: LMS w/ Interferer (w2)");
xlabel("Frequency"); ylabel("Magnitude"); grid on;
%% Testing Module -> Plot Errors
figure(); plot(abs(errors));
figure(); plot(abs(errors2));
```