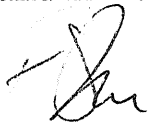
EECS 844 – Fall 2016

Exam 4 Cover page*

Each student is expected to complete the exam individually using only course notes, the book, and technical literature, and without aid from outside sources.

Aside from the most general conversation of the exam material, I assert that I have neither provided help nor accepted help from another student in completing this exam. As such, the work herein is mine and mine alone.



Signature

12/6/16

Date

RICH SIMEON

Name (printed)

2909995

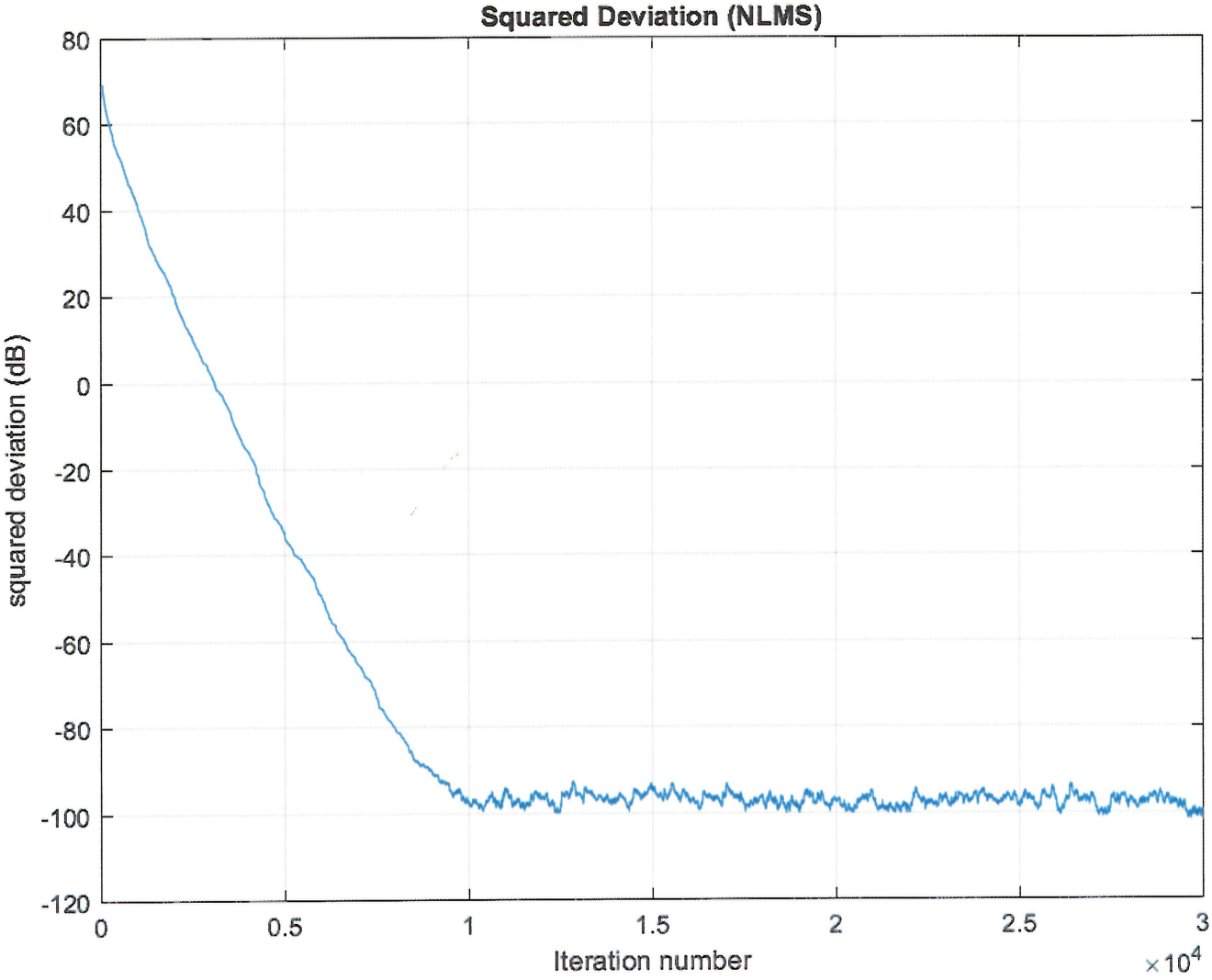
Student ID #

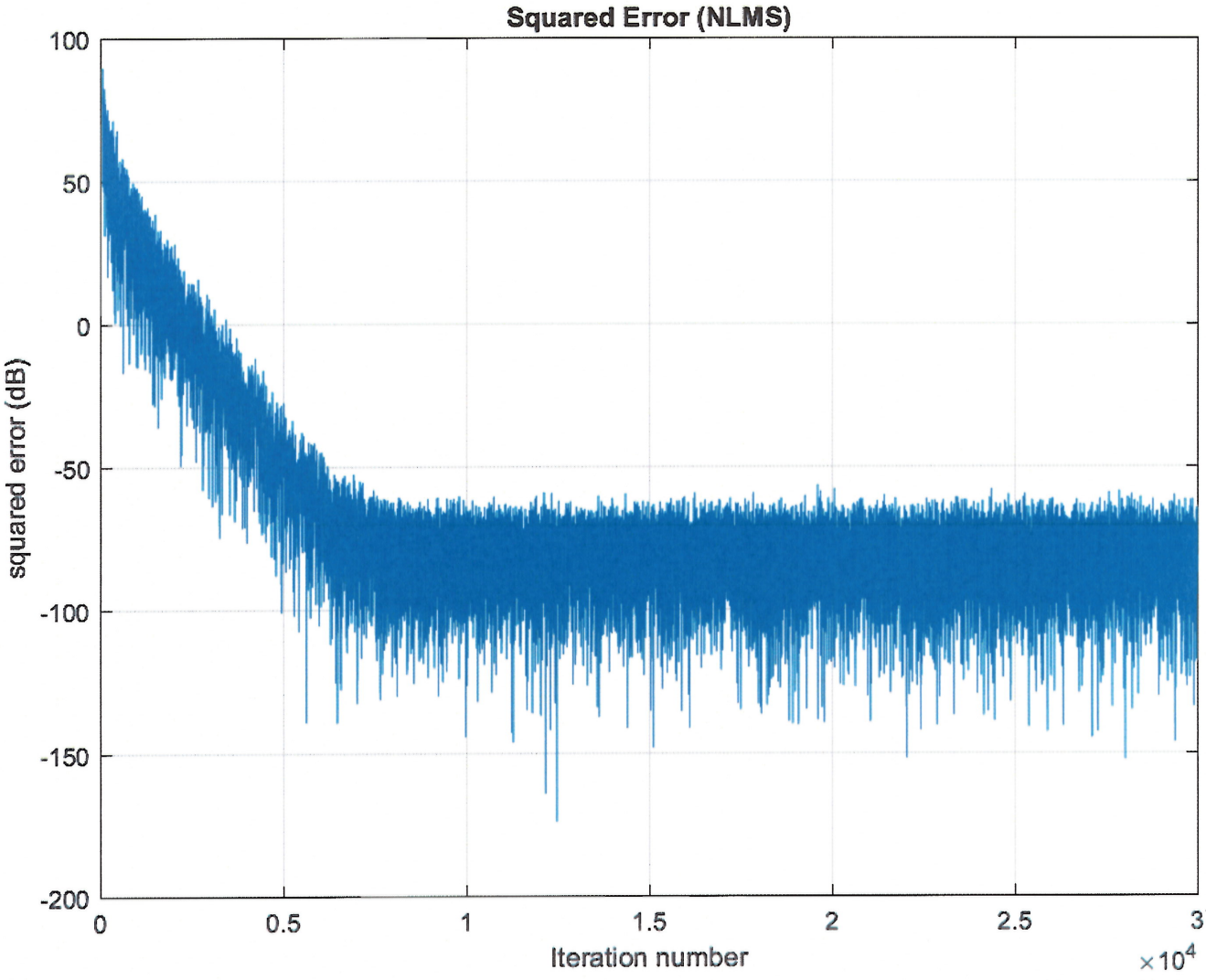
*** Attach as cover page to completed exam.**

Problem 1

Based on the coefficients of the Wiener filter, the model appears to be of order 33; the 60-tap LMS filter was able to converge on the correct solution in 10000 iterations with a very small MSE (-100dB). Due to the logarithmic scale, the convergence was linear in the log sense, meaning convergence in an exponentially-decaying manner, as expected for the LMS algorithm.

Also as expected, the final convergence vacillated about the Wiener filter solution, but in a very small fashion. Adjustment of the step size would show that any variation less than or greater than the normalized step size would slow down the convergence rate (but with better misadjustment) or speed up convergence (at the expense of a higher ending MSE).


Squared Deviation (NLMS)
80
60
40
20
0
-20
-40
-60
-80
-100
-120
squared deviation (dB)
0
0.5
1
1.5
2
2.5
3
Iteration number
×10^4


Squared Error (NLMS)
100
50
0
-50
-100
-150
-200
squared error (dB)
0
0.5
1
1.5
2
2.5
3
Iteration number
×10^4

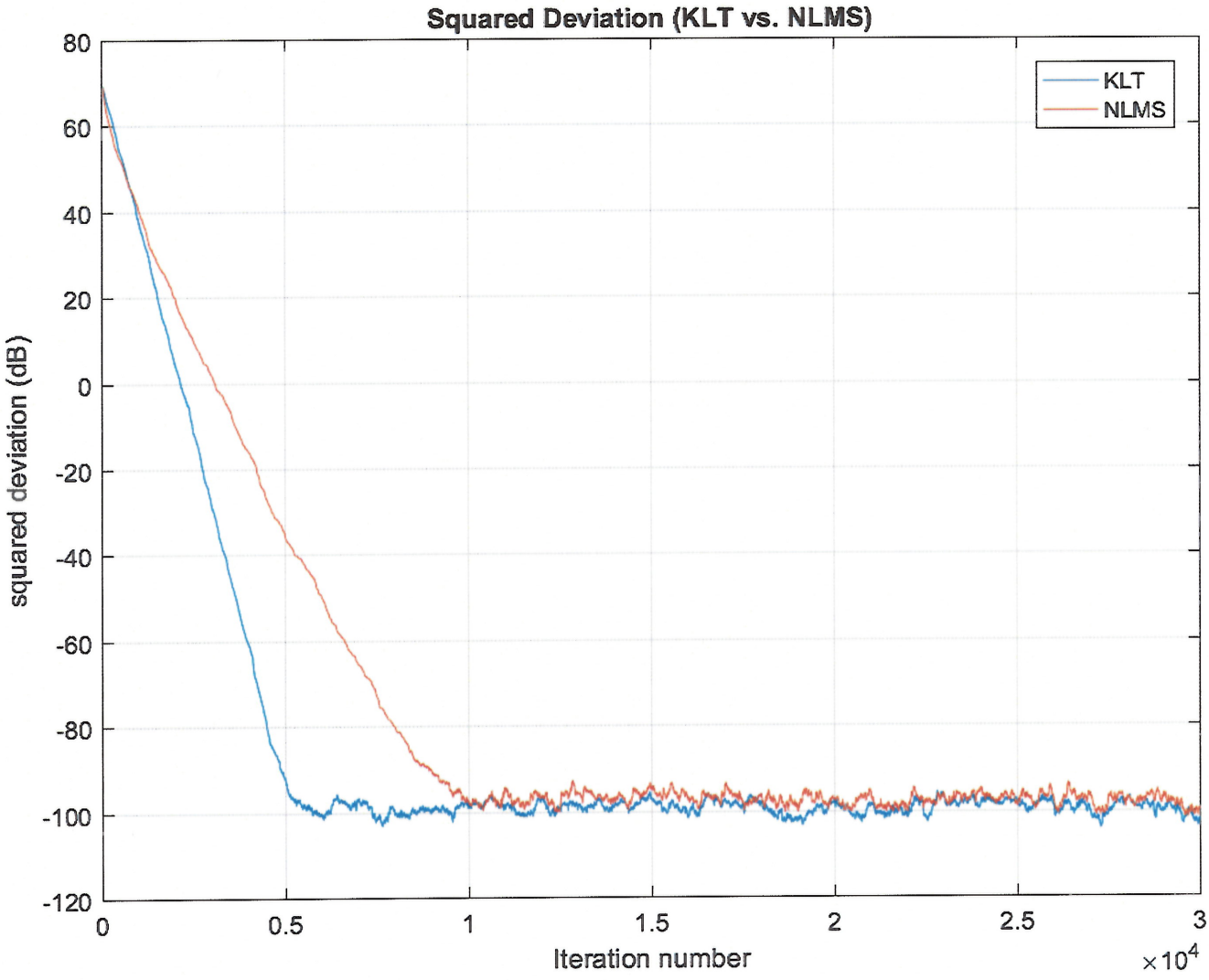
```
% Exam 4 Problem 1

clear all;
close all;
hold off;

load p1.mat; % loads x,d

M = 60; % Wiener filter order
K = length(x);
N = K-M+1;

mu_hat = 0.5;  % step size
delta  = 0.03; % leakage factor

%% Wiener filter----------------------

% Create X snapshot matrix
X = zeros(M,K);
for k=1:N
    X(:,k) = flipud(x(k:k+M-1));
end

% Calculate autocorrelation matrix R
Rxx = (1/N)*X*ctranspose(X);

% Calculate P
P = zeros(M,1);
for k=M:K
    P = P + (flipud(x(k-M+1:k)) * conj(d(k)));
end
P = P / N;

% Calculate Wiener filter (wo)
wo = inv(Rxx)*P;

%% Normalized LMS
w    = zeros(M,1);           % init NLMS weights to zero
e_squared   = zeros(K,1);    % squared error
squared_dev = zeros(K,1);    % squared deviation

for n=M:K
    y_l = ctranspose(w)*flipud(x(n-M+1:n));     % compute LMS filter output
    e = d(n) - y_l;                             % compute error
    x_norm = ctranspose(x(n-M+1:n))*x(n-M+1:n);% get input energy for normalization
    mu = mu_hat / (delta + x_norm);             % compute step size
    w = w + mu * conj(e) * flipud(x(n-M+1:n)); % LMS update

    e_squared(n) = conj(e)*e;                   % squared error
    squared_dev(n) = ctranspose(w-wo)*(w-wo);   % squared deviation
end
```

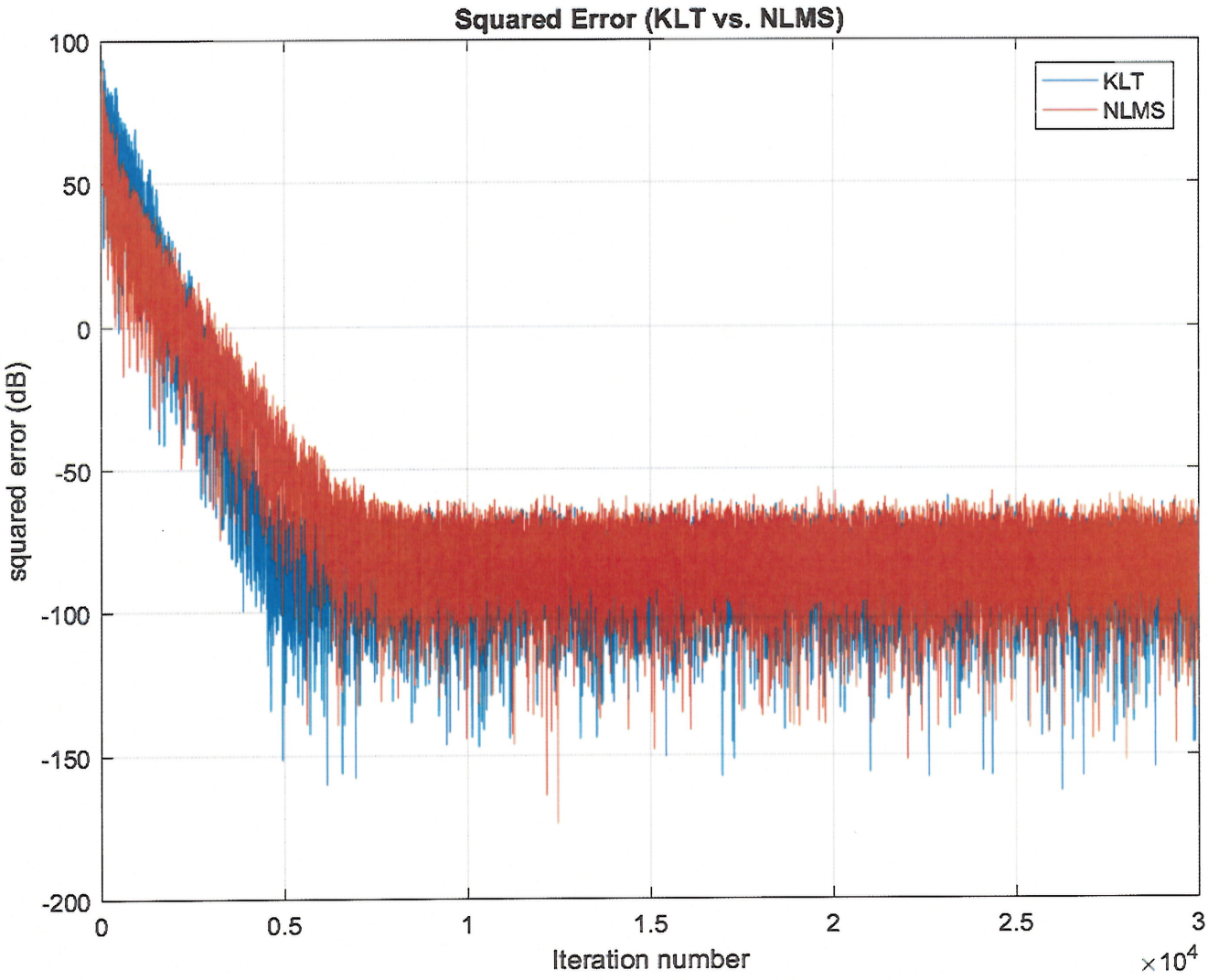
```
%% Plots----------------------------

figure(1);
plot(20*log10(e_squared));
title('Squared Error (NLMS)');
xlabel('Iteration number');
ylabel('squared error (dB)');
grid on;

figure(2);
plot(20*log10(squared_dev));
title('Squared Deviation (NLMS)');
xlabel('Iteration number');
ylabel('squared deviation (dB)');
grid on;
```

Problem 2

The KLT LMS filter solution yielded a much faster convergence at 5000 iterations (almost half the time of the normalized LMS), even though the final MSE was the same at about -100dB. Final misadjustment was also comparable to the LMS.

By de-correlating the input vector, a faster convergence is possible for the same normalized step size. The drawback for KLT is required knowledge of the eigenvectors for linearly mapping the input vector to the decorrelated transformed domain, and the eigenvalues for proper scaling of the step size for each filter tap.


Squared Deviation (KLT vs. NLMS)
KLT
NLMS
squared deviation (dB)
80
60
40
20
0
-20
-40
-60
-80
-100
-120
0
0.5
1
1.5
2
2.5
3
Iteration number
×10^4


Squared Error (KLT vs. NLMS)
KLT
NLMS
squared error (dB)
100
50
0
-50
-100
-150
-200
0
0.5
1
1.5
2
2.5
3
Iteration number
×10^4

```
% Exam 4 Problem 2

clear all;
close all;
hold off;

load p1.mat; % loads x,d

M = 60; % Wiener filter order
K = length(x);
N = K-M+1;

mu_hat_lms = 0.5;  % step size
mu_hat_KLT = 0.125/M;  % step size
delta      = 0.03;     % leakage factor

%% Wiener filter----------------------

% Create X snapshot matrix
X = zeros(M,K);
for k=1:N
    X(:,k) = flipud(x(k:k+M-1));
end

% Calculate autocorrelation matrix R
Rxx = (1/N)*X*ctranspose(X);

% Calculate P
P = zeros(M,1);
for k=M:K
    P = P + (flipud(x(k-M+1:k)) * conj(d(k)));
end
P = P / N;

% Calculate Wiener filter (wo)
wo = inv(Rxx)*P;

%% KLT transform LMS and Normalized LMS ----------------------
[Q,D] = eig(Rxx);
D = D + delta*eye(size(D));

% KLT initialization
wt              = zeros(M,1);   % init weights to zero
w_klt           = zeros(M,1);   % init weights to zero
e_squared_klt   = zeros(K,1);   % squared error
squared_dev_klt = zeros(K,1);   % squared deviation

% NLMS initialization
w_lms   = zeros(M,1);           % init NLMS weights to zero
e_squared_lms   = zeros(K,1);   % squared error
squared_dev_lms = zeros(K,1);   % squared deviation
```

```
for n=M:K
    % Do KLT transform LMS-------------
    y = ctranspose(w_klt)*flipud(x(n-M+1:n));       % compute KLT filter output
    e = d(n) - y;                                   % compute error
    z = ctranspose(Q)*flipud(x(n-M+1:n));           % compute transformed input vector
    wt = wt + (mu_hat_KLT * inv(D) * z * conj(e)); % compute transformed weight vector
update
    w_klt = Q*wt;                                   % re-map weight vector back to
"normal" domain

    e_squared_klt(n)    = conj(e)*e;                          % KLT squared error
    squared_dev_klt(n) = ctranspose(w_klt-wo)*(w_klt-wo);  % KLT squared deviation

    % Do NLMS-------------------------
    y = ctranspose(w_lms)*flipud(x(n-M+1:n));  % compute LMS filter output
    e = d(n) - y;                              % compute error
    x_norm = ctranspose(x(n-M+1:n))*x(n-M+1:n);% get input energy for normalization
    mu = mu_hat_lms / (delta + x_norm);        % compute step size
    w_lms = w_lms + mu * conj(e) * flipud(x(n-M+1:n)); % LMS update

    e_squared_lms(n) = conj(e)*e;                          % NLMS squared error
    squared_dev_lms(n) = ctranspose(w_lms-wo)*(w_lms-wo);  % NLMS squared deviation
end

%% Plots---------------------------

figure(1);
plot(20*log10(e_squared_klt));
hold on;
plot(20*log10(e_squared_lms));
title('Squared Error (KLT vs. NLMS)');
xlabel('Iteration number');
ylabel('squared error (dB)');
grid on;
legend({'KLT','NLMS'});

figure(2);
plot(20*log10(squared_dev_klt));
hold on;
plot(20*log10(squared_dev_lms));
title('Squared Deviation (KLT vs. NLMS)');
xlabel('Iteration number');
ylabel('squared deviation (dB)');
grid on;
legend({'KLT','NLMS'});
```

Problem 3

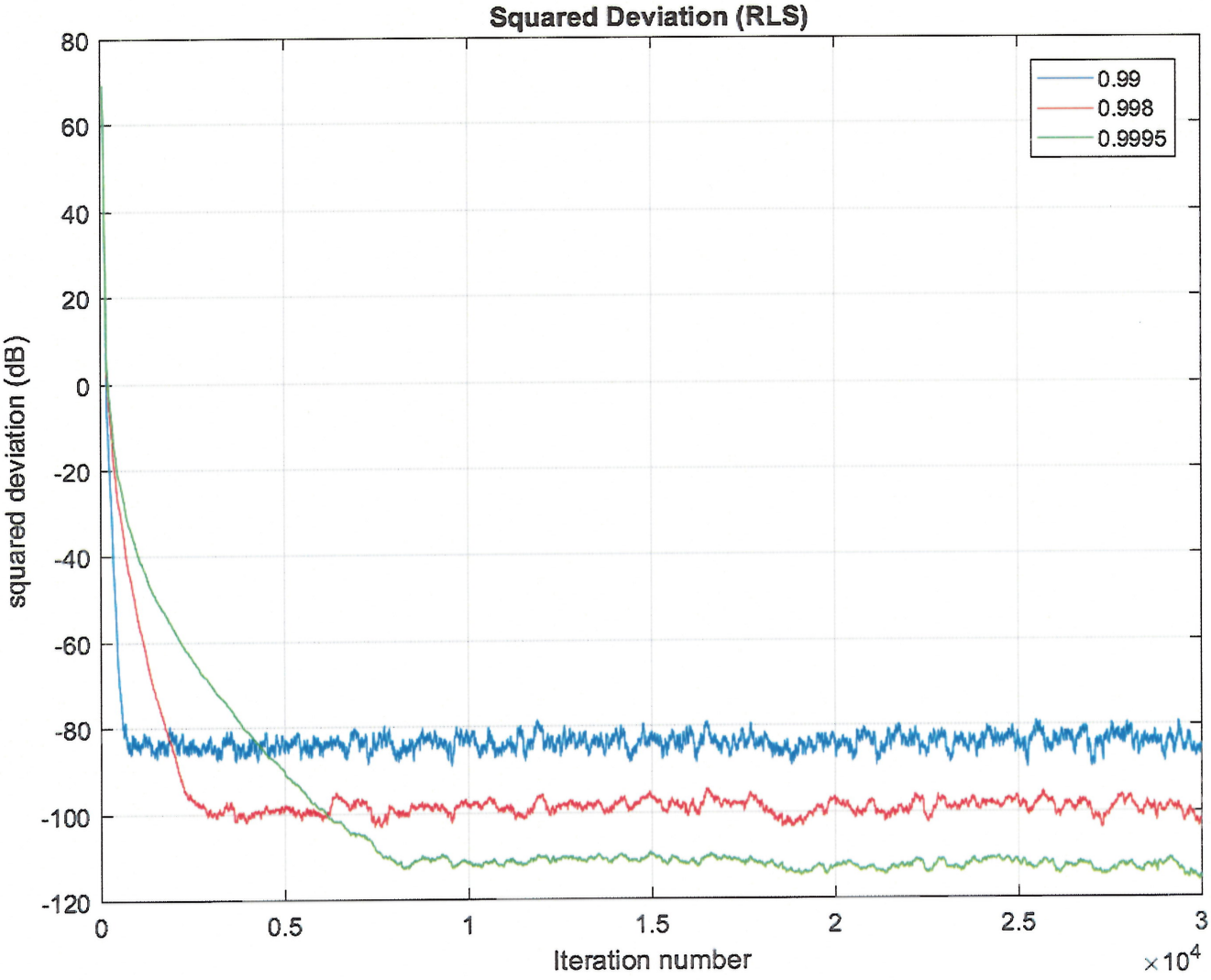
The RLS filter solution was overall the best in terms of convergence speed and final MSE (though obviously the most computationally-expensive to implement).

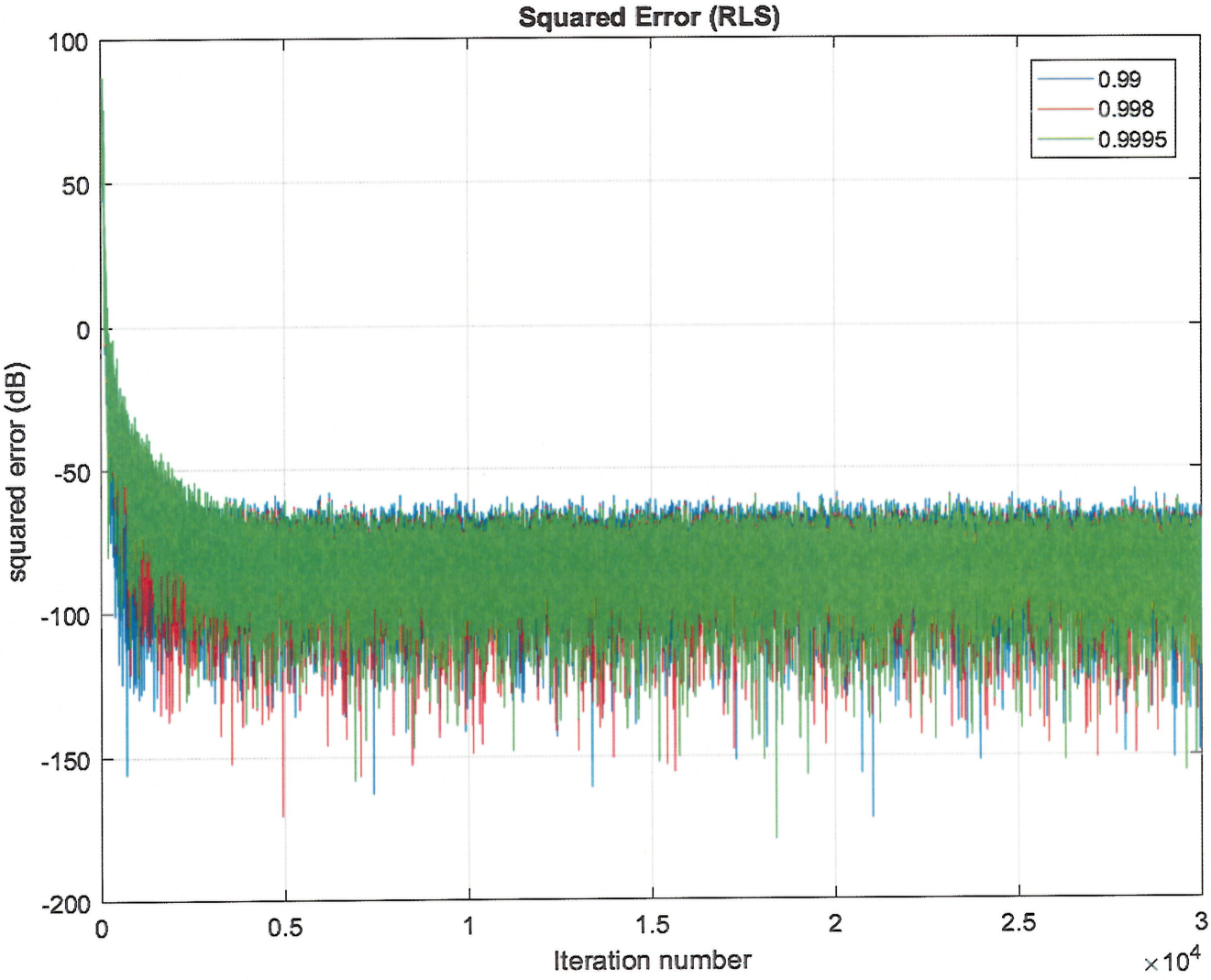
At a lambda forgetting factor of 0.99, convergence was an order of magnitude better than the NLMS at about 1000 iterations, but ended with a higher squared deviation at -85dB, or higher than the original NLMS.

At 0.998, convergence was better than both the NLMS and KLT (converged at 2500 iterations, vs. 5000 for KLT and 10000 for NLMS), for the same steady-state performance of -100dB squared deviation.

Finally, at 0.9995, the convergence was between the KLT and NLMS at around 7800 iterations, but had a lower ending squared deviation at -110dB.

The tradeoff of minimum MSE/squared deviation vs. convergence speed is evident in the plots. Minimum squared deviation is dependent on the forgetting factor; factors close to 1.0 are ideal for stationary signals, and will value past errors almost as much as current errors. This, in turn, yields more accurate error estimates since error variances are smaller (the decaying effect of the forgetting factor effectively shrinks the number of samples used for the error estimate, thus increasing the error variance). A higher error variance manifests itself as a higher misadjustment through a slightly higher gain vector (increase in gradient noise); however, a higher gain vector increases convergence speed.


Squared Deviation (RLS)
0.99
0.998
0.9995
80
60
40
20
0
-20
-40
-60
-80
-100
-120
squared deviation (dB)
0
0.5
1
1.5
2
2.5
3
Iteration number
×10^4


Squared Error (RLS)
0.99
0.998
0.9995
100
50
0
-50
-100
-150
-200
squared error (dB)
0
0.5
1
1.5
2
2.5
3
Iteration number
×10^4

```
% Exam 4 Problem 3

clear all;
close all;
hold off;

load p1.mat; % loads x,d

M = 60; % Wiener filter order
K = length(x);
N = K-M+1;

lambda_array = [0.99 0.998 0.9995];

%% Wiener filter---------------------

% Create X snapshot matrix
X = zeros(M,K);
for k=1:N
    X(:,k) = flipud(x(k:k+M-1));
end

% Calculate autocorrelation matrix R
Rxx = (1/N)*X*ctranspose(X);

% Calculate Pdx
Pdx = zeros(M,1);
for k=M:K
    Pdx = Pdx + (flipud(x(k-M+1:k)) * conj(d(k)));
end
Pdx = Pdx / N;

% Calculate Wiener filter (wo)
wo = inv(Rxx)*Pdx;

%% RLS----------------------
e_squared   = zeros(K,length(lambda_array));% squared error
squared_dev = zeros(K,length(lambda_array));% squared deviation

% Cycle through all values of lambda
for m=1:length(lambda_array)
    P      = eye(M);            % init inverse correlation matrix inv(R)
    w      = zeros(M,1);        % init weights to zero
    lambda = lambda_array(m);   % load lambda for this run

    for n=M:K
        xn = flipud(x(n-M+1:n));    % define input vector

        % RLS update
        pi = P*xn;
        k = pi / (lambda + ctranspose(xn)*pi);      % compute gain vector
```

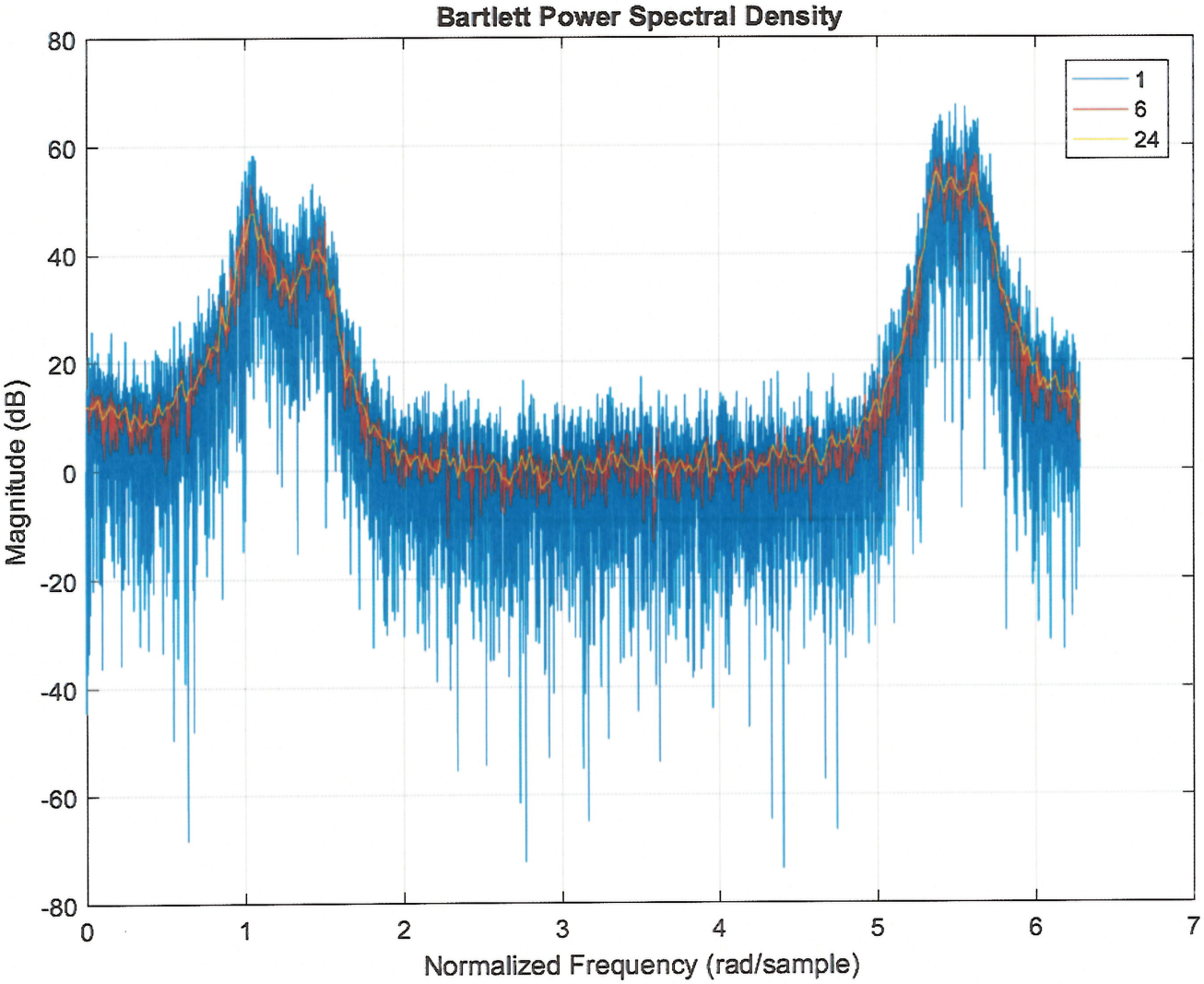
```
        y = ctranspose(w)*xn;                                % compute filter output
        e = d(n) - y;                                        % compute error
        w = w + k*conj(e);                                   % update weight vector
        P = (P/lambda) - (k*ctranspose(xn)*P)/lambda;% update inverse correlation matrix
inv(R)
                                                               % via the Ricatti equation

        e_squared(n,m)    = conj(e)*e;                       % squared error
        squared_dev(n,m) = ctranspose(w-wo)*(w-wo);% squared deviation
    end
end

%% Plots----------------------------

figure(1);
plot(20*log10(e_squared(:,1)));
hold on
plot(20*log10(e_squared(:,2)),'color','red');
plot(20*log10(e_squared(:,3)),'color','green');
title('Squared Error (RLS)');
xlabel('Iteration number');
ylabel('squared error (dB)');
legend({'0.99','0.998','0.9995'});
grid on;

figure(2);
plot(20*log10(squared_dev(:,1)));
hold on;
plot(20*log10(squared_dev(:,2)),'color','red');
plot(20*log10(squared_dev(:,3)),'color','green');
title('Squared Deviation (RLS)');
xlabel('Iteration number');
ylabel('squared deviation (dB)');
legend({'0.99','0.998','0.9995'});
grid on;
```

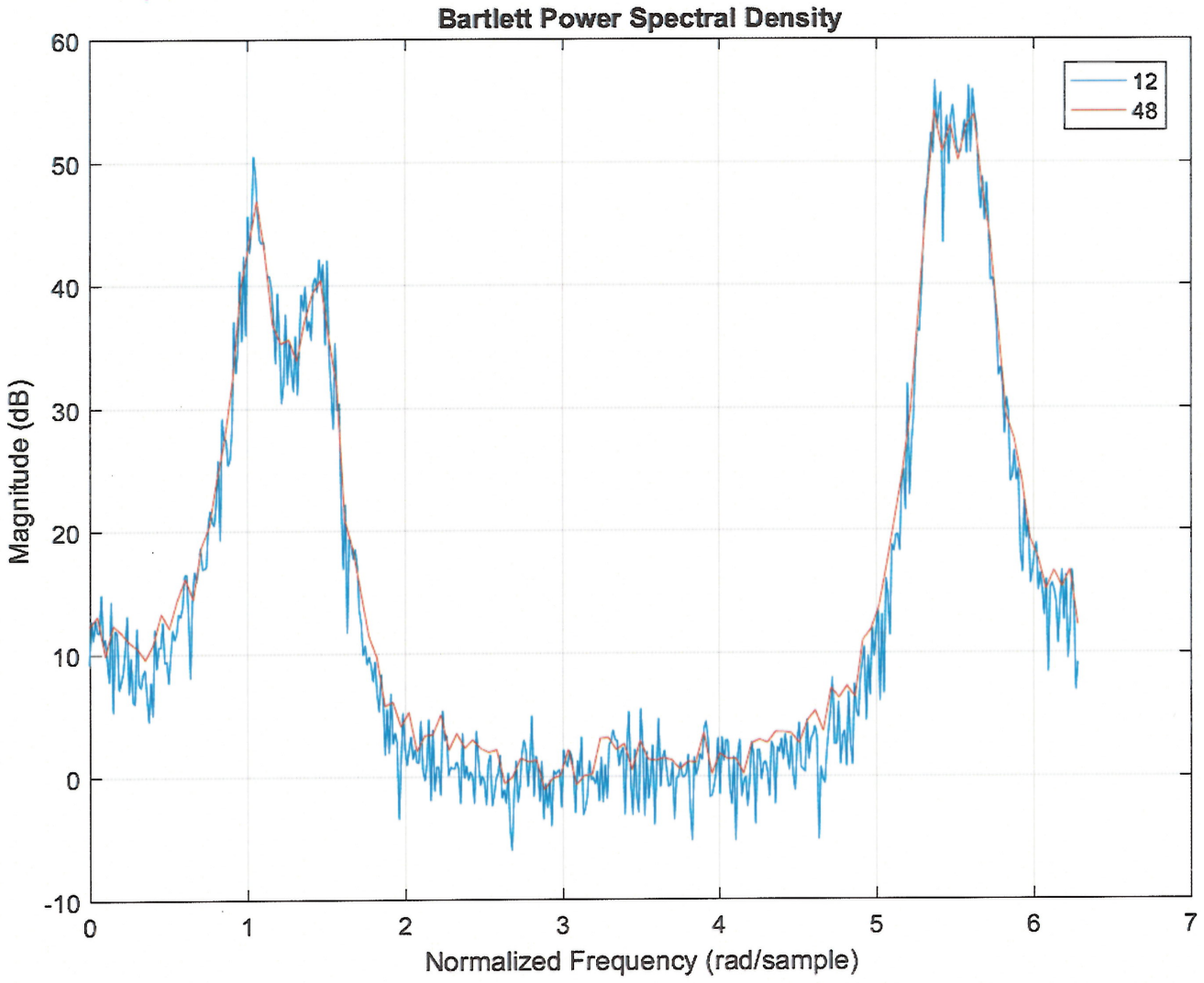
Problem 4

Plots of the Bartlett PSD shows that for K=1 (1 segment), the resolution of the PSD is very high, but is also the noisiest plot by far. As the number of segments increases, the resolution of the PSD decreases, but the plots become smoother.

A comparison of K=1, 6, and 24 shows that the K=1 is very noisy, but K=24 shows more distinct peaks where signals are present. It's very hard to determine the location of the peaks around 1.0 and 1.5 radians, but it's much easier at the K=24 and K=48 case.

Likewise, for K=12 and 48, the K=48 case is smoother than the K=12 case, but still has enough resolution to discern signal peaks at around 1 and 1.5 radians/sample. There appears to be two or more peaks at 5.4 to 5.6 radians.


Bartlett Power Spectral Density
Magnitude (dB)
Normalized Frequency (rad/sample)
1
6
24
80
60
40
20
0
-20
-40
-60
-80
0
1
2
3
4
5
6
7


Bartlett Power Spectral Density
Magnitude (dB)
Normalized Frequency (rad/sample)
12
48
60
50
40
30
20
10
0
-10
0
1
2
3
4
5
6
7

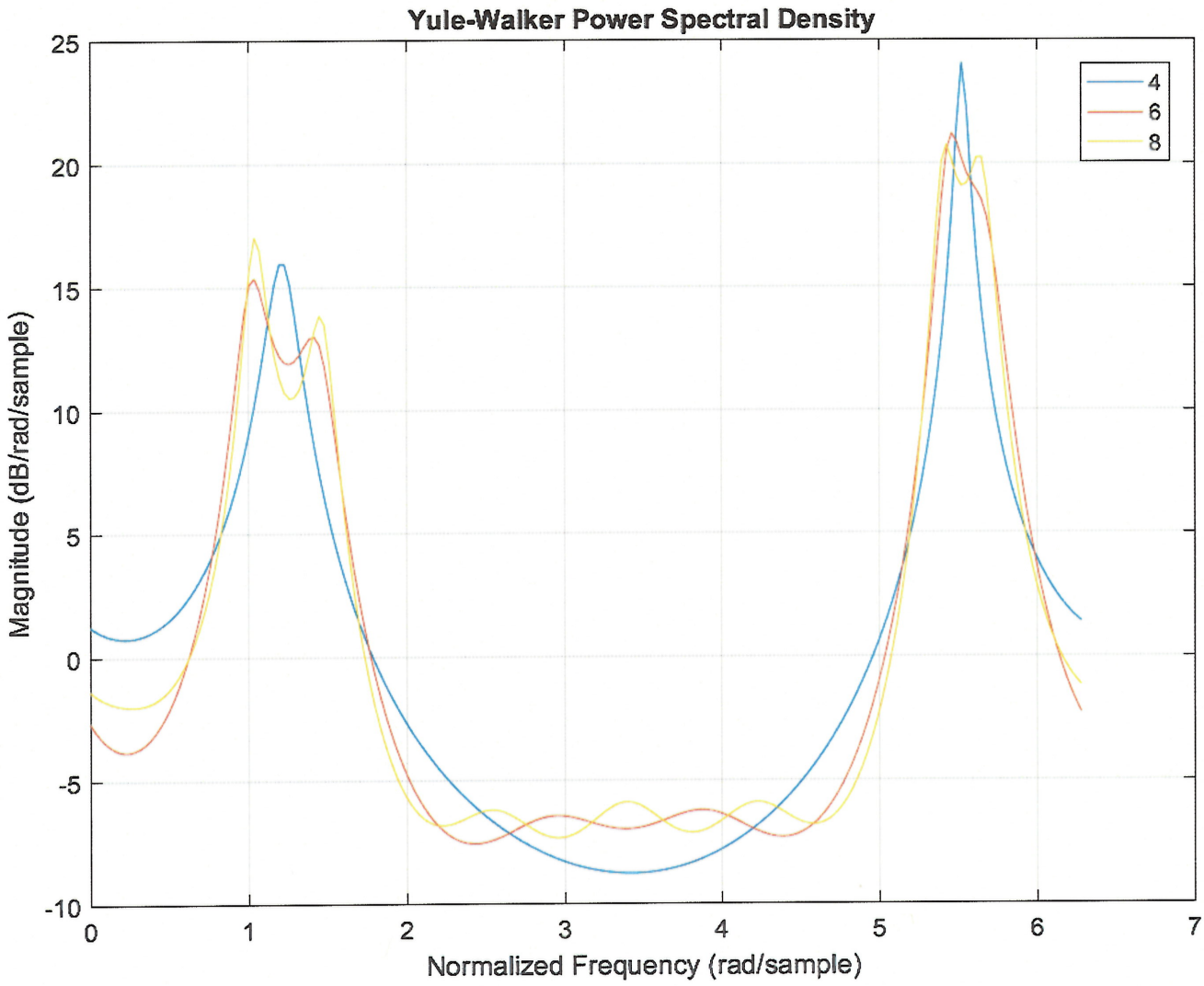
```
% Exam 4 Problem 4

clear all;
close all;
hold off;

load p4.mat; % loads x

segment_lengths = [1 6 12 24 48];
figure(1);

% Cycle through all segment lengths
for idx=1:length(segment_lengths)
    K = segment_lengths(idx);
    M = floor(length(x)/K); % number of bins in this PSD
    w = linspace(0,2*pi,M);
    Pxx = zeros(M,1);       % PSD for each segment

    % Cycle through all K segments
    for i=0:K-1
        % Compute ith Pxx(f) over all f
        for f=1:M
            sum = 0;
            for n=0:M-1
                sum = sum + x(n+1+(i*M))*exp(-j*w(f)*n);
            end
            Pxx(f) = Pxx(f) + (sum * conj(sum))/M;
        end
    end

    % Normalize by number of segments
    Pxx = Pxx / K;

    % Plot Power Spectral Density
    plot(w,20*log10(abs(Pxx)));
    hold on;
end

title('Bartlett Power Spectral Density');
xlabel('Normalized Frequency (rad/sample)');
ylabel('Magnitude (dB)');
legend({'1','6','12','24','48'});
grid on;
```

Problem 5

Using parametric modeling via the Yule-Walker method for PSD estimation, we can see using the MATLAB freqz() function that the spectral peaks are more noticeable, and the PSD is naturally smoother

For order p=8, the YW method shows four peaks at {1.0, 1.4, 5.4, and 5.6} radians/sample. This is in contrast to the Bartlett PSD, which showed (at higher segment values) two peaks at {1.0, 1.4} radians/sample and a wide peak at 5.5 radians. So the YW method identified two distinct peaks around 5.4 and 5.6 radians/sample, as opposed to the Bartlett method which showed a broad peak at 5.5 radians/sample.

For order p=6, only three peaks are seen at {1.0, 1.4, 5.4} radians/sample. Since the order is reduced, only three peaks can be discerned. Likewise, at p=4, only two peaks at {1.2, 5.5} radians/sample are detected, which is roughly centered between the peaks seen for the higher-order estimators.


Yule-Walker Power Spectral Density
Magnitude (dB/rad/sample)
Normalized Frequency (rad/sample)
4
6
8
25
20
15
10
5
0
-5
-10
0
1
2
3
4
5
6
7

```
% Exam 4 Problem 5

clear all;
close all;
hold off;

load p4.mat; % loads x

K = length(x);
p_vals = [4 6 8];
L = 200; % number of PSD bins
w = linspace(0,2*pi,L);

figure(1);

% Cycle through all model orders
for idx=1:length(p_vals)
    p = p_vals(idx);
    M = p+1;
    N = K-M+1;

    % Create X snapshot matrix
    X = zeros(M,N);
    for k=1:N
        X(:,k) = flipud(x(k:k+M-1));
    end

    % Calculate autocorrelation matrix R
    R = (1/N)*X*ctranspose(X);
    rxx = R(2:M,1);
    Rxx = R(1:p,1:p);

    % Calculate AR coefficients
    a = -inv(Rxx)*rxx;

    % Evaluate using freqz
    ar = zeros(M,1);
    ar(1) = 1.0;
    ar(2:M) = conj(a);

    % Plot frequency response using freqz
    [H,W] = freqz(1,ar,L,'whole');
    plot(w,20*log10(abs(H)));
    hold on;
end

title('Yule-Walker Power Spectral Density');
xlabel('Normalized Frequency (rad/sample)');
ylabel('Magnitude (dB/rad/sample)');
legend({'4','6','8'});
grid on;
```

Problem 6

	X1	X2	X3	X4
Signal Covariance Matrix	4	1	4	1

Temporally-correlated signals (e.g. from multi-path effects) can affect the number of sources that are estimated by the BIC algorithm, by artificially lowering the rank of the autocorrelation matrix. Also, two (or more) signals arriving from very close locations (angles) could lower the rank of the autocorrelation matrix since their angles of arrival are almost the same (signal separation would be difficult).

The signal is also assumed to be stationary over the period used to form the correlation matrix. Furthermore, the noise is assumed to be white (in order to keep the noise power on the main diagonal of the autocorrelation matrix) and the SNR should be high enough such that the magnitudes of the eigenvalues can accurately reflect the number of sources (a low SNR would tend to equalize the eigenvalue magnitudes and "hide" the dominant signal eigenvalues).

```
% Problem 6, Exam 4

clear variables;

load p6.mat; % loads X1..X4

% Cycle through all four data sets
for a=1:4
    % Select data set to analyze
    switch a
        case 1
            A = X1;
        case 2
            A = X2;
        case 3
            A = X3;
        case 4
            A = X4;
    end

    [M,N] = size(A); % N = number of time samples
    p = M; % number of sensors
    n = N; % number of time samples

    % Calculate autocorrelation matrix R
    R = (1/N)*A*ctranspose(A);

    % since R is square PSDH, it can be decomposed via eigen decomposition
    % since R is Hermitian, lambdas are real-valued
    % Get eigenvalues and eigenvectors of R
    [V, lambda] = eig(R);
    lambda = diag(lambda);
    l = real(flipud(sort(lambda)));

    % Calculate the BIC (Bayesian Information Criterion)
    BIC = zeros(p,1);
    for q=0:p-1
        sum1 = 0; % first summation term
        sum2 = 0; % second summation term
        for i=q+1:p
            sum1 = sum1 + log(l(i));
            sum2 = sum2 + (l(i)/(p-q));
        end
        log_lq = n * (sum1 - (p-q)*log(sum2));
        BIC(q+1) = (-2*log_lq) + (((q*((2*p)-q))+1)*log(n));
    end

    [min_BIC, min_index] = min(BIC);
    fprintf('Estimated number of signals = %d\n',min_index-1);
end
```

Problem 7

	X1	X2	X3	X4
Signal Covariance Matrix	4	1	4	1
Forward-Backward Averaging	8	2	4	2

When forward-backward (FB) averaging is performed on the signals, all signals except X3 increase by a factor of two. From the explanation below, the increase can be good (number of signals is closer to the actual number) or bad (number of signals is being overestimated).

For ideal uniform linear arrays, forward-backward averaging has the benefit of adding twice as many independent samples, thus possibly adding to the rank of a rank-deficient autocorrelation matrix (or lowering the condition number of that matrix). It also acts to increase the signal sub-space for temporally-correlated signals impinging on the array by a factor of two, to the point where the signal sub-space is closer or equal to the actual number of signals.

FB averaging can be detrimental if the linear array is not necessarily uniform, or has uncalibrated arrays where each element is slightly different (e.g. magnitude/phase response). When the samples are flipped and conjugated, the signal structure is slightly different, possibly enough to create a signal that looks independent enough to be construed as a different signal; the effect of this can be seen as a false increase in the number of detected signals.

Since we don't know the actual number of signals (yet), the increase in number of signals can be beneficial (the FB averaging moved the number of signals closer to the actual number), or detrimental (FB caused the number of signals to be overestimated, e.g. if a non-ideal array is used).

```
% Problem 7, Exam 4

clear variables;

load p6.mat; % loads X1..X4

% Cycle through all four data sets
for a=1:4
    % Select data set to analyze
    switch a
        case 1
            A = X1;
        case 2
            A = X2;
        case 3
            A = X3;
        case 4
            A = X4;
    end

    [M,N] = size(A); % N = number of time samples
    p = M; % number of sensors
    n = N; % number of time samples

    % Form a reflection matrix and forward-backward estimate of R
    J = zeros(M,M);
    for idx=1:M
        J(M-idx+1,idx) = 1;
    end
    Rfb = (1/(2*N))*(A*ctranspose(A) + J*conj(A)*transpose(A)*J);

    % since R is square PSDH, it can be decomposed via eigen decomposition
    % since R is Hermitian, lambdas are real-valued
    % Get eigenvalues and eigenvectors of Rfb
    [V, lambda] = eig(Rfb);
    lambda = diag(lambda);
    l = real(flipud(sort(lambda)));

    % Calculate the BIC (Bayesian Information Criterion)
    BIC = zeros(p,1);
    for q=0:p-1
        sum1 = 0; % first summation term
        sum2 = 0; % second summation term
        for i=q+1:p
            sum1 = sum1 + log(l(i));
            sum2 = sum2 + (l(i)/(p-q));
        end
        log_lq = n * (sum1 - (p-q)*log(sum2));
        BIC(q+1) = (-2*log_lq) + (((q*((2*p)-q))+1)*log(n));
    end
```

```
    [min_BIC, min_index] = min(BIC);
    fprintf('Estimated number of signals = %d\n',min_index-1);
end
```

Problem 8

	X1	X2	X3	X4
Signal Covariance Matrix	4	1	4	1
Forward-Backward Averaging	8	2	4	2
Spatial Smoothing	4	4	4	11

Spatial smoothing helps remove the effects of temporally-correlated signals. In this simulation, the number of sub-arrays (L) is 14. As seen in this simulation run, it appears to help X1 (FB averaging may have falsely overestimated the number of sources, while SS brings it back to the actual number of sources). X3 did not change, possibly because this signal is well-behaved (ideal array receiver elements, no multi-path, independent signals). X2 also seemed to improve as well, indicating that SCM might have been previously underestimated the number of sources in X2 due to multipath.

Interestingly, spatial smoothing dramatically increased the number of sources from 1 (in the SCM case) to 11. Since the maximum rank of the signal subspace for spatial smoothing is the minimum of L (14) or M-1 (11), it appears that the X4 signal has at least 11 paths (1 direct and 10 multi-path reflections) but possibly more.

```
% Problem 8, Exam 4

clear variables;

load p6.mat; % loads X1..X4
M_hat = 12;  % sub-array size

% Cycle through all four data sets
for a=1:4
    % Select data set to analyze
    switch a
        case 1
            A = X1;
        case 2
            A = X2;
        case 3
            A = X3;
        case 4
            A = X4;
    end

    [M,N] = size(A); % N = number of time samples

    n = N;           % number of time samples
    L = N;           % number of time samples
    K = M-M_hat+1;   % number of sub-arrays to average
    p = M_hat;       % number of sub-array sensors

    % Perform spatial smoothing on the received signals
    Rss = zeros(M_hat,M_hat);
    for m=1:K
        % Create Xm matrix
        Xm = A(m:m+M_hat-1,:);

        % Calculate autocorrelation matrix R
        Rss = Rss + Xm*ctranspose(Xm);
    end
    Rss = Rss / (L*K);

    % since R is square PSDH, it can be decomposed via eigen decomposition
    % since R is Hermitian, lambdas are real-valued
    % Get eigenvalues and eigenvectors of Rss
    [V, lambda] = eig(Rss);
    lambda = diag(lambda);
    l = real(flipud(sort(lambda)));

    % Calculate the BIC (Bayesian Information Criterion)
    BIC = zeros(p,1);
    for q=0:p-1
        sum1 = 0; % first summation term
        sum2 = 0; % second summation term
```

```
        for i=q+1:p
            sum1 = sum1 + log(l(i));
            sum2 = sum2 + (l(i)/(p-q));
        end
        log_lq = n * (sum1 - (p-q)*log(sum2));
        BIC(q+1) = (-2*log_lq) + (((q*((2*p)-q))+1)*log(n));
    end

    [min_BIC, min_index] = min(BIC);
    fprintf('Estimated number of signals = %d\n',min_index-1);
end
```

Problem 9

The cumulative results of MATLAB simulations of for SCM, FB averaging, and spatial smoothing are shown below:

	X1	X2	X3	X4
Signal Covariance Matrix	4	1	4	1
Forward-Backward Averaging	8	2	4	2
Spatial Smoothing	4	4	4	11

X1 and X3 both have 4 signals detected, while X2 and X4 have 1 signal detected. Therefore, we can say that X2 and X4 both are signals that contain temporally-correlated signals, while X1 and X3 are temporally uncorrelated.

Signal	Case A (ideal, uncorr.)	Case B (ideal, corr)	Case C (real, uncorr)	Case D (real, corr)
X1	X		X	
X2		X		X
X3	X		X	
X4		X		X

When FB averaging is performed, all signals except X3 increased the number of detected signals by a factor of 2. It's known that FB averaging will increase the number of signals in uncalibrated antenna arrays, since flipping the signal array presents a slightly different signal structure (and therefore appears as another independent signal). Therefore, X3 is likely the signal that has an ideal array with temporally uncorrelated signals (case A). By process of elimination, X1 is likely case C.

Signal	Case A (ideal, uncorr.)	Case B (ideal, corr)	Case C (real, uncorr)	Case D (real, corr)
X1			X	
X2		X		X
X3	X			
X4		X		X

When spatial smoothing is performed, all signals except X4 indicated 4 signals present, while X4 indicated 11 signals. Spatial smoothing is known to increase the signal subspace by a factor equal to the number of overlapping sub-arrays; in the case of the realistic array with temporally correlated signals, the signal subspace increased enough to indicate extra signals for X4, but more than X2, meaning that the uncalibrated array created enough difference in signal structure from the reflected (false) signals to appear as distinct signals. Therefore, X4 is likely case D, while by process of elimination X2 is left to be case B. X2 was also the signal that steadily increased the signal subspace after each improvement.

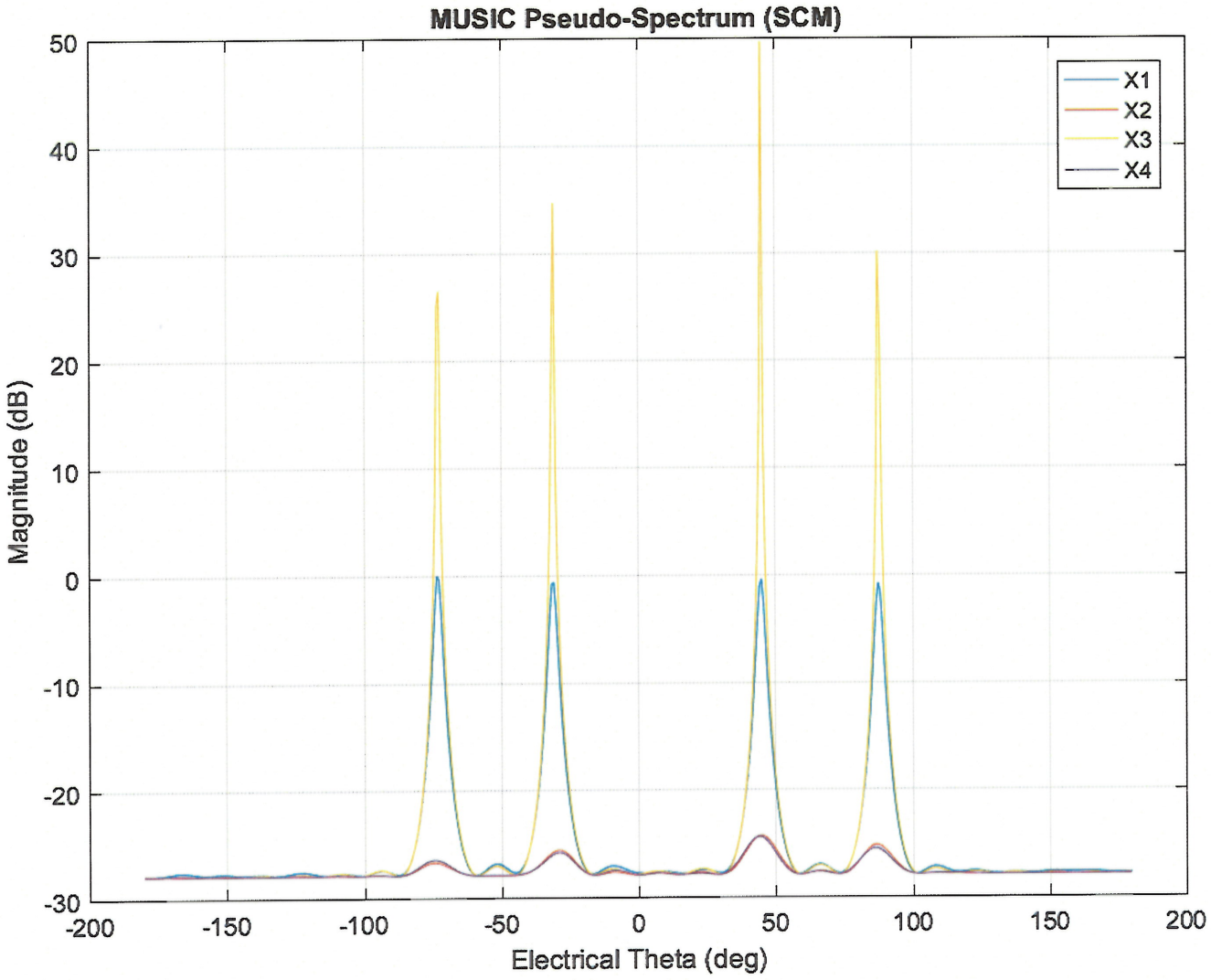
Signal	Case A (ideal, uncorr.)	Case B (ideal, corr)	Case C (real, uncorr)	Case D (real, corr)
X1			X	
X2		X		
X3	X			
X4				X

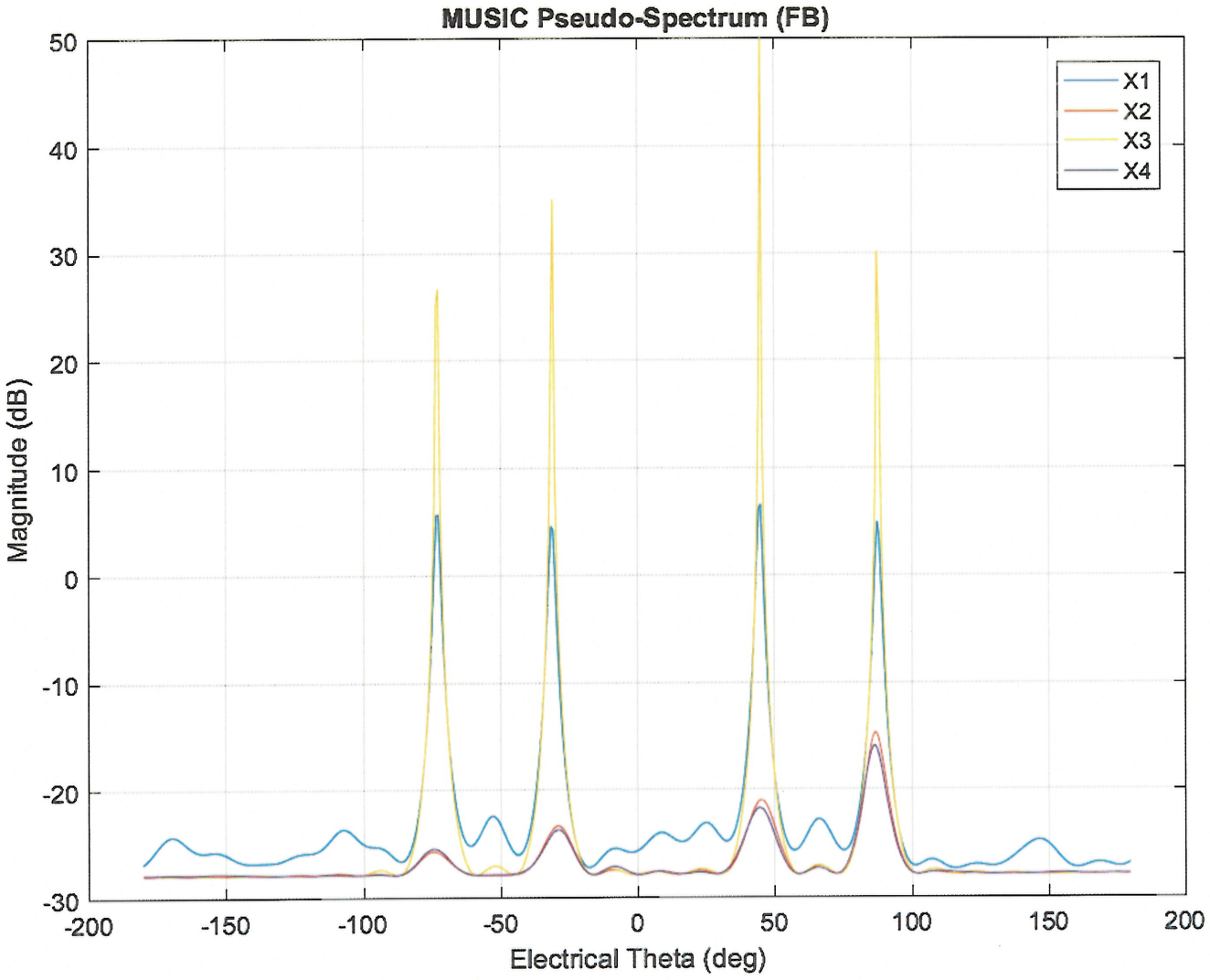
Problem 10

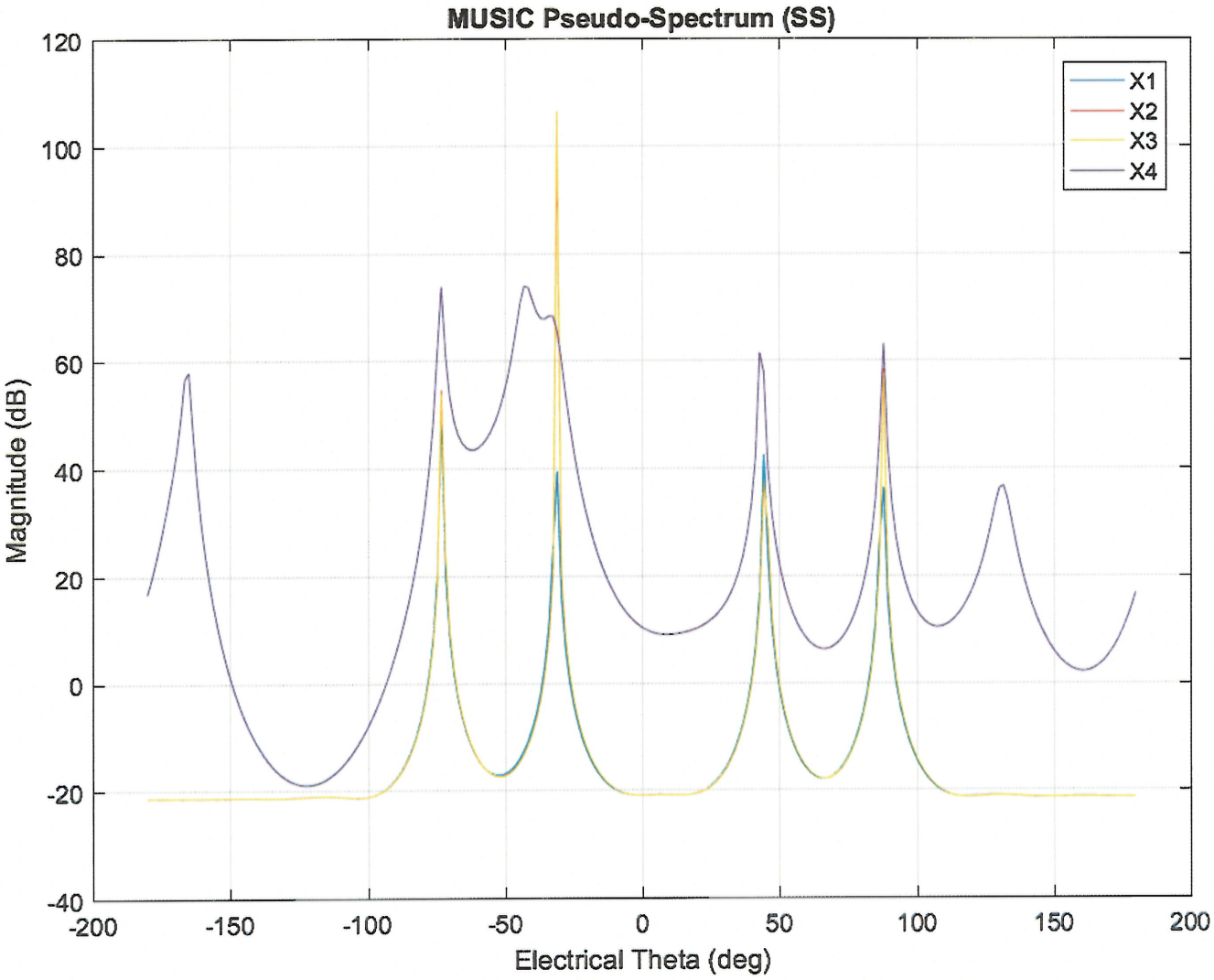
For the SCM case, the MUSIC pseudo-spectrum plots reveal that X1 and X3 show four strong peaks at {-75, -31, 45, 87} degrees, while X2 and X4 don't have very strong peaks at any angle (nor are there any sharp peaks). The strongest peak for X2 and X4 appears to be at 45 degrees, which is likely the sole signal source estimated by the BIC algorithm; it could be argued that the peak at 87 degrees is present for X2 and X4.

For the FB-averaging case, the MUSIC plots show four peaks at the same locations for all four signals. However, peaks for X2 and X4 are still small relative to the X1 and X3 peaks, but the peaks are higher, indicating that FB averaging did help to decrease the condition number of the covariance matrix by de-correlating the signals somewhat. It's clear that the BIC probably indicated that the peaks at 45 and 87 degrees were the two that it considered "significant"; however, from visual inspection, it's possible to say that there are four peaks for X2 and X4. Finally, X1 showed a slightly higher "noise floor" than X3, presumably due to the negative effects of FB averaging on non-ideal arrays not duplicating the original signal structure when flipping/conjugating the array signals.

Finally, for the SS case, all signals (X1-X4) all showed four sharp peaks at the same locations. However, X4 showed four extra peaks at {-165, -41, 132, 180} degrees; in this instance, it's possible that spatial smoothing could not average-out the effects of the non-ideal antenna array, since X2 had a well-behaved MUSIC response.


MUSIC Pseudo-Spectrum (SCM)
X1
X2
X3
X4
Magnitude (dB)
50
40
30
20
10
0
-10
-20
-30
-200
-150
-100
-50
0
50
100
150
200
Electrical Theta (deg)


MUSIC Pseudo-Spectrum (FB)
X1
X2
X3
X4
Magnitude (dB)
50
40
30
20
10
0
-10
-20
-30
-200
-150
-100
-50
0
50
100
150
200
Electrical Theta (deg)


MUSIC Pseudo-Spectrum (SS)
Magnitude (dB)
Electrical Theta (deg)
X1
X2
X3
X4
120
100
80
60
40
20
0
-20
-40
-200
-150
-100
-50
0
50
100
150
200

```
% Problem 10, Exam 4

clear variables;
close all;
hold off;

load p6.mat; % loads X1..X4
M_hat = 12;  % sub-array size for spatial smoothing case

% Cycle through all four data sets
for a=1:4
    % Select data set to analyze
    switch a
        case 1
            A = X1;
        case 2
            A = X2;
        case 3
            A = X3;
        case 4
            A = X4;
    end

    [M,N] = size(A); % N = number of time samples

    n = N;           % number of time samples
    L = N;           % number of time samples
    K = M-M_hat+1;   % number of sub-arrays to average
    p = M_hat;       % number of sub-array sensors

    % Calculate autocorrelation matrix Rxx
    Rxx = (1/N)*A*ctranspose(A);

    % Form a reflection matrix and forward-backward estimate of Rfb
    J = zeros(M,M);
    for idx=1:M
        J(M-idx+1,idx) = 1;
    end
    Rfb = (1/(2*N))*(A*ctranspose(A) + J*conj(A)*transpose(A)*J);

    % Perform spatial smoothing on the received signals
    Rss = zeros(M_hat,M_hat);
    for m=1:K
        % Create Xm matrix
        Xm = A(m:m+M_hat-1,:);

        % Calculate autocorrelation matrix Rss
        Rss = Rss + Xm*ctranspose(Xm);
    end
    Rss = Rss / (L*K);
```

```
% Choose autocorrelation matrix to use for analysis
%R = Rxx;            % uncomment for SCM
%R = Rfb;            % uncomment for FB
R = Rss; M = M_hat; % uncomment for SS

% since R is square PSDH, it can be decomposed via eigen decomposition
% since R is Hermitian, lambdas are real-valued
% Get eigenvalues and eigenvectors of R
[V, lambda] = eig(R);
lambda = real(diag(lambda)); % force real-only
l = flipud(sort(lambda));

% Check to see if MATLAB didn't order the dominant eigenvalues
% and eigenvectors at the beginning of the matrix. If not, then
% use fliplr to order it correctly. Or course this assumes that
% the eigenvalues and associated eigenvectors are sorted as well.
if( lambda(1) ~= l(1) )
    V = fliplr(V);
end

% Calculate the BIC (Bayesian Information Criterion)
BIC = zeros(p,1);
for q=0:p-1
    sum1 = 0; % first summation term
    sum2 = 0; % second summation term
    for i=q+1:p
        sum1 = sum1 + log(l(i));
        sum2 = sum2 + (l(i)/(p-q));
    end
    log_lq = n * (sum1 - (p-q)*log(sum2));
    BIC(q+1) = (-2*log_lq) + (((q*((2*p)-q))+1)*log(n));
end

[min_BIC, min_index] = min(BIC);
fprintf('Estimated number of signals = %d\n',min_index-1);

% Declare number of detected signals
p = min_index-1;

% Compute pseudo-spectrum
numsamps = 20*M;
theta = linspace(-pi,pi,numsamps);
degrees = linspace(-180,180,numsamps);

PS = zeros(numsamps,1); % pseudo-spectrum
for idx=1:numsamps
    U = 0.0;
    for k=p+1:M
        % Compute steering matrix (electrical angle)
        s = transpose(exp(-j*theta(idx)*linspace(0,M-1,M)));
        U = U + abs(ctranspose(s)*V(:,k))^2;
```

```
        end
        PS(idx) = 1/U;
    end

    % MUSIC pseudo-spectrum vs. electrical angle
    plot(degrees,20*log10(abs(PS)));
    hold on;
end

title('MUSIC Pseudo-Spectrum');
xlabel('Electrical Theta (deg)');
ylabel('Magnitude (dB)');
grid on;
legend({'X1','X2','X3','X4'});
```