## Computational Assignment

### Problem 1: Value Iteration/Policy Iteration/Q-Learning and Linear Programming

On thar part, we need to solve our encoder based on some iterations and Q-Learning. For that case, first of all, we have a X which members are B and G. They means fading channel in good and bad state. Also. There is an excoder which says using channel or not. It gets values from U which are zero and one. Our goal of the encoder is send information across the channer. Our aim is making u = 1 and x = G for utilizing successful. Also, we have transition kernels.

For part a.i, we need to make value iteration. We need to take some fixed $\beta \in (0, 1)$ of your choice. Consider $\zeta = 0.75$ and $\eta = 0.8$, $\eta = 0.6$ and $\eta = 0.01$. Interpret the optimal solution for these different values of $\eta$, in view of the application.

On MATLAB, I coded simple Markov Decision Process problem with two states and two actions. First of all, I initialized transition probabilities and cost parameters. They are already given in problem. After that, I coded states and actions which is also already defined in problem. Then, I created a loop for value iteration. The main reason I did this is to try diffrent eta values. Then, I started to my loop code. Firstly, I initialized value functions and policies. Then, the value function is updated repeatedly until the ideal values are reached. The loop continues for as long as ‘max_iterations’ or until the differance between two before value function. Then, for each state-action pair, for using that, I calculated Q-values. It calcuşated for getting the sum of the immediate reward. Then, for iterating, it updates the value function and policy. Finally, I checked for convergence. If our updated value less that 1e-6, then it stops. According to my code, my result is:

“For eta = 0.8
Optimal Policy:
{[1]}
{[1]}

Optimal Value Function:
339.3945
338.6445

For eta = 0.6
Optimal Policy:
{[1]}
{[0]}

Optimal Value Function:
253.7133
253.1133

For eta = 0.01
Optimal Policy:
{[1]}
{[0]}

Optimal Value Function:

4.2286
4.2186”

After then, I wrote same code with policy iteration. The code estimates the value function for the current policy in the policy evaluation phase using the Bellman equation, and then iteratively updates the value function until convergence. According to the existing policy, the value function represents the anticipated cumulative reward starting from each state.
Using the Q-values for each state-action pair, the policy improvement phase, which comes after policy evaluation, determines if the policy has improved. The predicted total reward from doing a particular action in a particular state and then adhering to the existing policy is represented by the Q-value. The code then chooses the action for each state with the highest Q-value to update the policy, leading to a better policy.
Finally, the algorithm repeats to get best value just like value iteration. It also ends with stable policy findings. Then, after my implementation, my codes result is:

“For eta = 0.8
Optimal Policy:
{[1]}
{[1]}

Optimal Value Function:
339.3945
338.6445

For eta = 0.6
Optimal Policy:
{[1]}
{[0]}

Optimal Value Function:
253.7133
253.1133

For eta = 0.01
Optimal Policy:
{[1]}
{[0]}

Optimal Value Function:
4.2286
4.2186”

Finally, for last part of a (a.iii), I need to calculate that value with Q-values. Through episodes, the Q-learning algorithm iteratively engages with the environment. Each episode begins with the agent in a random state. He then chooses a course of action using an epsilon-greedy strategy, sees the outcome, and receives an immediate reward. Based on the Q-learning update algorithm, which integrates the immediate reward and the discounted maximum Q-value of the next state, the Q-values in the Q-function are updated. The Q-values are improved by this iterative process across several episodes, resulting in an ideal policy that chooses the actions with the highest Q-values for every state. Finally, my result is:

“Q-Learning - eta = 0.60:
2
1”

For part a ending, I need to make a comparison about it. Although each of the three solutions use a different algorithm to do so, they all try to identify the best policy and value function for the MDP problem. Dynamic programming algorithms called Value Iteration (a.i) and Policy Iteration (a.ii) act on the value function and policy iteratively. A model-free method that learns from interactions with the environment is Q-learning, in contrast. While the Q-learning solution discovers the best course of action through exploration and exploitation, both the a.i. and a.ii solutions explicitly handle the transition probabilities and apply them in the computations. For the specified MDP problem, all three methods produce the same optimal policy and value function. It is good because we need to get around same answers for it. It did not take them, it means some parts of code works wrong.

Then, for part b, I need to apply convex analytic method to find optimal policy. Inputs were given just like before questions. Also, for that assignment, I needed to use “linprog” to solve linear programming problems. First of all, I defined all channel states and actions just like part a. Then, I defined transition probabilities and cost parameters given in question. Then, for selecting relevant elements in transition, I transformed transition probabilities in matrix form. Then, I defined equality constraints for getting sum of probabilities. Then, to restrict binary values, I used lower and upper bounds. Then, I solved linear programming by using ‘linprog’. Then I checked unboundedness or something for getting true value. Finally, when we get the true value, I extract optimal policy and I displayed the result:

“Optimal Policy:
When the channel is in state G, the encoder should use the channel (u = 1).
When the channel is in state B, the encoder should not use the channel (u = 0).
Objective Value (Minimum Cost): 0.51”

Finally, I ended to code first part of code.

**Problem 2: The Kalman Filter**

For that problem, we have a linear system driven by Gaussian noise and partially observed Markov chains model. We have a noise process which is i.i.d. is given in problem. For part a, we need to write down the Kalman Filter update equations:

For prediction step:
priori of xt = A * one back posteriori of xt
priori of pt = A * one back posteriori of pt * A’ + Q

For update step:
kt = priori of pt * C’ * inverse of(C * priori of pt * C’ + R)
posteriori of xt = priori of xt + Kt * (yt – C * priori of xt)
posteriori of pt = (size of priori of pt – Kt * C) * priori of pt

Finally, for Ricatti, we have:
one pack posteriori of pt = A * priori pf pt * A’ + Q

Then, for part b, we needed to simulate the system given in problem 2. We also need to print some plots. For getting them, firstly, I initialized Kalman Filter parameters just like given in question. Then, I initialized simulation time parameters just like also given in question. Then, for estimating posteriori of xt, I initialized state and estimation of convariance. Also, we need to store our result for plotting. Thus, I applied some code for storaging results. Then, for predicting next step for priori, I initialized prediction step based on previous posterior estimates by using state transition matrix and process noise convariance. Then, just like first steps, I ordered generating measurement, updating and storing our values. Finally, I plotted them:

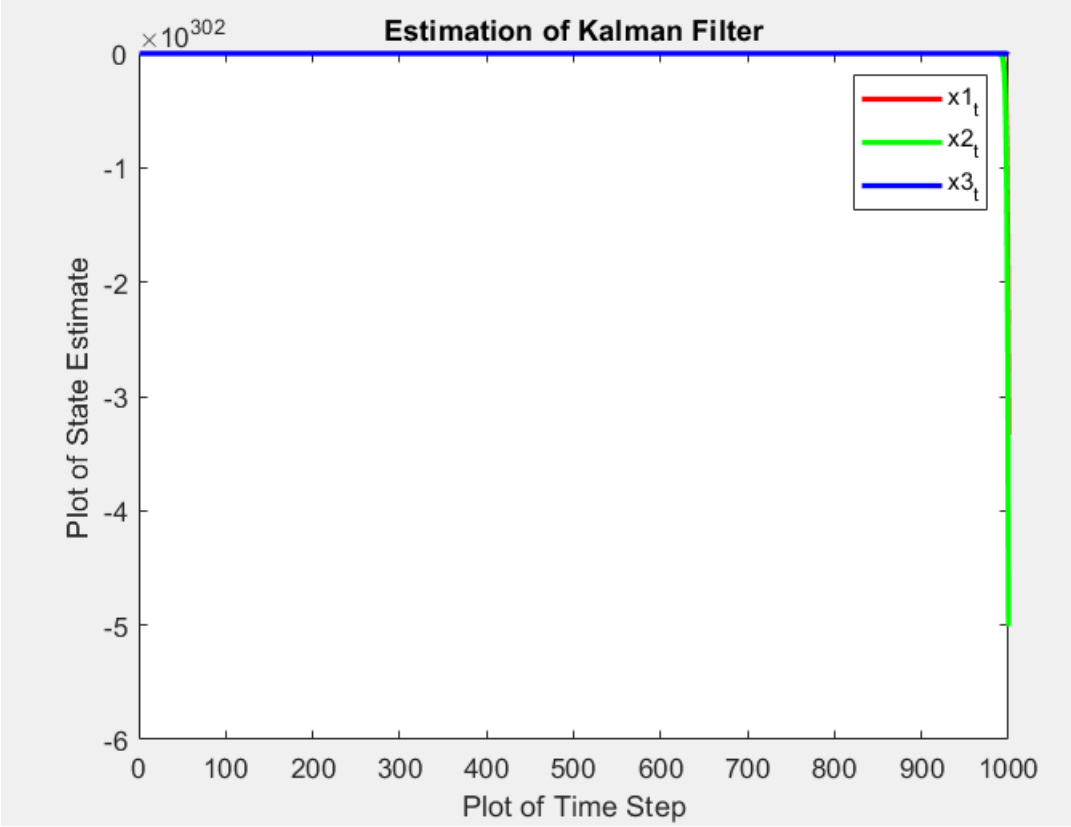


**Image 1:** Plot of Estimation of Kalman Filter

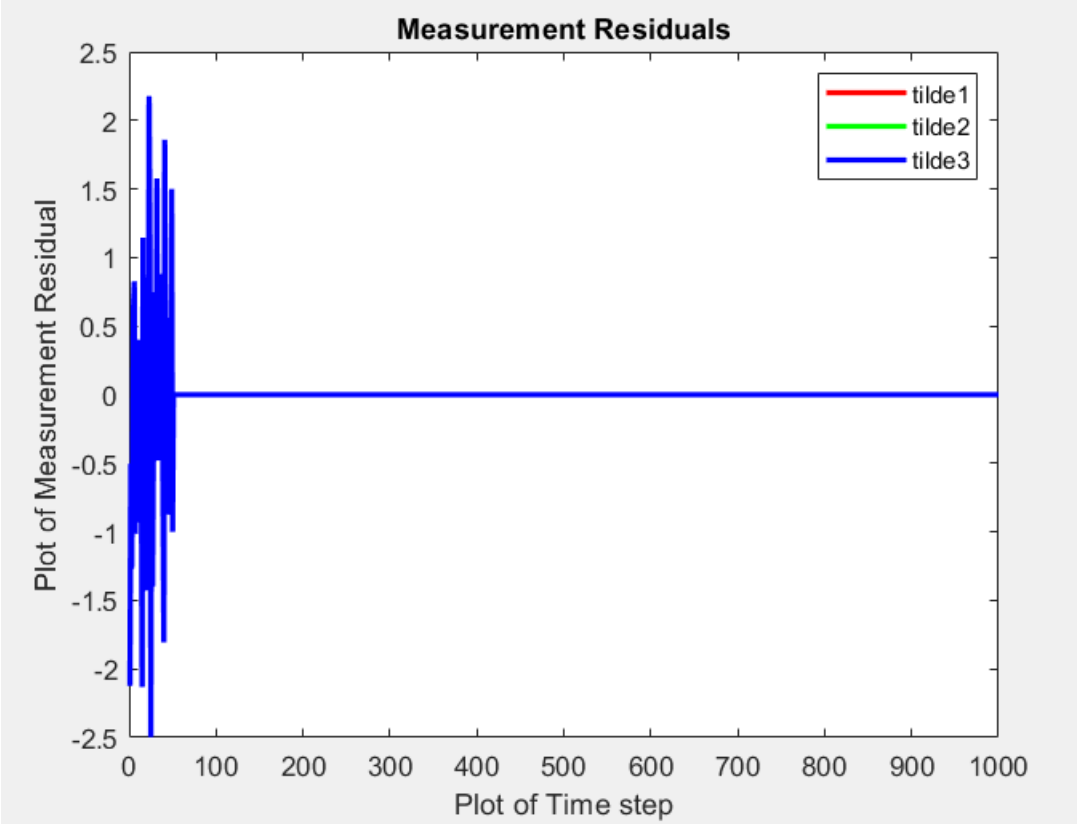


**Image 2:** Plot of Measurement Residuals

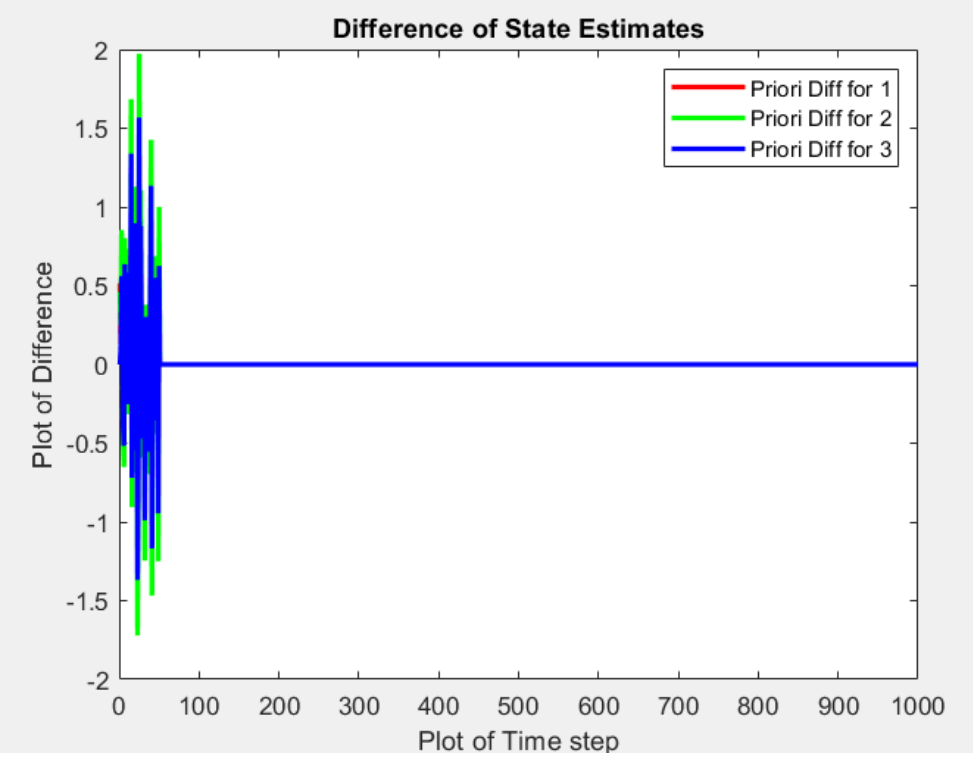


**Image 3:** Plot of Difference of State Estimates

According to those results, we can check them all of components (state estimations, measurement residuals, estimation differences) works perfectly for all of steps. No matter tilde changes, it keeps difference zero for some time. Also, our first image, one of the value goes very bad for estimation. It is also same for our part a answers. When we increase our steps, our optimated result have become infinity which is not should be. Then, we can say that this works good.

For final step we need to answer that question: “Do the Ricatti recursions converge to a limit?”. For answering that question, for getting our covariance matrix, I applied diffrent initial for that part. Then, I applied prediction, measurement and update code just like part b on question 2. Dinally my answer states that:

“Riccati recursions is not unique. It means it did not converge to the same limit.”

For answering that state why, we need to understand when Ricatti recursion is not converge to the same limit. When that thing happens, it means that we do not have an unique steady state convariance. It can happens when our dynamics has multiple possible steadt state covariance matrices. Nonlinearity, observability and initial conditions can be reasons why it happens. It's fundamental to note that non-uniqueness of the steady-state covariance does not fundamentally suggest that the Kalman Channel isn't working accurately. The Kalman Channel gives a great appraise of the framework state and covariance given the accessible estimations and framework elements. In any case, when the framework has different conceivable steady-state arrangements, the covariance gauges might vary based on the starting conditions and other variables.
To superior get it the behavior of the framework and its covariance gauges, it is advantageous to analyze the framework flow, discernibleness, and beginning conditions in more detail. In commonsense applications, the non-uniqueness of the steady-state covariance can be overseen by carefully setting the introductory conditions and utilizing suitable sifting methods to guarantee steadiness and vigor.

**Conclusion**

In conclusion, on that assignment, we applied so many contents which we learned in lesson. Using many of the concepts we have learnt in the course in an application such as MATLAB will enable us to reinforce this education both in quizzes and in our future life. While doing this homework, we use the information that is only in theory in real life and we clearly understand which connections can be made in which situations.

## Appendix

```
%% Warning! To work those codes, plese run by section.
%% Q1
%% a.i
% Transition probabilities
Prob1good = 0.1; Prob1bad = 0.9; Prob0good = 0.9; Prob0bad = 0.1;
% Cost parameters
zeta = 0.75;
etas = [0.8, 0.6, 0.01]; % Our diffrent eta values
beta = 0.9; % This is our discount factor
max_iterations = 10; % Maximum number of iterations for value iteration

% Our states and action
X = {'B', 'G'};
U = {0, 1};

% Main code (Value Iteration)
for i = 1:length(etas)
    eta = etas(i);

    % Initialize value function and policy
    V = zeros(length(X), 1); % Initialize V to zeros
    policy = cell(length(X), 1);

    for iter = 1:max_iterations
        V_prev = V;

        for j = 1:length(X)
            x = X{j};
            Q = zeros(length(U), 1);

            for k = 1:length(U)
                u = U{k};
                Q(k) = -zeta*(strcmp(x, 'G') && u==1) + eta*u;
                for m = 1:length(X)
                    x_next = X{m};
                    Q(k) = Q(k) + beta*(Prob1good*V_prev(strcmp(X, x_next)) + Prob1bad*V_prev(strcmp(X, 'B')));
                end
            end

            [V(j), idx] = max(Q);
            policy{j} = U{idx};
        end

        if max(abs(V - V_prev)) < 1e-6
            break;
        end
    end

    % This is our results
    disp(['For eta = ', num2str(eta)]);
    disp('Optimal Policy:');
    disp(policy);
    disp('Optimal Value Function:');
    disp(V);
end
```

```
%% a.ii
% This part is as same as a.i
Prob1good = 0.1; Prob1bad = 0.9; Prob0good = 0.9; Prob0bad = 0.1;
zeta = 0.75;
etas = [0.8, 0.6, 0.01];
beta = 0.9;
max_iterations = 10;

X = {'B', 'G'};
U = {0, 1};

% I initialized random policy
policy = cell(length(X), 1);
for i = 1:length(X)
   policy{i} = U{randi(2)};
end

% Main code (policy iteration)
for i = 1:length(etas)
   eta = etas(i);

   for iter = 1:max_iterations
      % It makes policy evaluation
      V = zeros(length(X), 1);
      for j = 1:max_iterations % I also use max_iteration for making policy evaluation
         V_prev = V;

         for j = 1:length(X)
            x = X{j};
            u = policy{j};
            Q = -zeta*(strcmp(x, 'G') && u==1) + eta*u;
            for k = 1:length(X)
               x_next = X{k};
               Q = Q + beta*(Prob1good*V_prev(strcmp(X, x_next)) + Prob1bad*V_prev(strcmp(X, 'B')));
            end

            V(j) = Q;
         end

         diff = max(abs(V - V_prev));
         if diff < 1e-6
            break;
         end
      end

      % Improvement of policy
      policy_stable = true;
      for j = 1:length(X)
         x = X{j};
         old_policy = policy{j};
         Q = zeros(length(U), 1);

         for k = 1:length(U)
            u = U{k};
            Q(k) = -zeta*(strcmp(x, 'G') && u==1) + eta*u;
            for m = 1:length(X)
               x_next = X{m};
               Q(k) = Q(k) + beta*(Prob1good*V(strcmp(X, x_next)) + Prob1bad*V(strcmp(X, 'B')));
            end
         end
```

```
            [V_max, idx] = max(Q);
            policy{j} = U{idx};

            if ~isequal(old_policy, policy{j})
                policy_stable = false;
            end
        end

        if policy_stable
            break;
        end
    end

    disp(['For eta = ', num2str(eta)]);
    disp('Optimal Policy:');
    disp(policy);
    disp('Optimal Value Function:');
    disp(V);
end

%% a.iii
% This initialization part is also as same as a.i and a.ii
beta = 0.9;
zeta = 0.75;
eta = 0.6;

X = {'B', 'G'};
U = [0, 1];

% This is our transition probabilities given
P = zeros(numel(X), numel(X), numel(U));
P(:,:,1) = [
    0.9, 0.1;
    0.9, 0.1;
];
P(:,:,2) = [
    0.1, 0.9;
    0.1, 0.9;
];

% Q-learning functions
num_episodes = 100;
Q = zeros(numel(X), numel(U)); % Q-function initialization

% Main function for Q-learning
for eps = 1:num_episodes
    % Initialize the state
    x_idx = randi(numel(X));
    x = X{x_idx};

    % Action based step chosen
    epsilon = 0.1;
    if rand < epsilon
        u_idx = randi(numel(U));
    else
        [~, u_idx] = max(Q(x_idx, :));
    end
    u = U(u_idx);
```

```
    % Updating
    cost = -zeta * (strcmp(x, 'G') && u == 1) + eta * u;

    x_next_idx = randsample(numel(X), 1, true, P(:, x_idx, u_idx));
    x_next = X{x_next_idx};

    Q(x_idx, u_idx) = Q(x_idx, u_idx) + 0.01 * (cost + beta * max(Q(x_next_idx, :)) - Q(x_idx, u_idx));
end

% Printing values
[~, policy] = max(Q, [], 2);
fprintf('Q-Learning - eta = %.2f:\n', eta);
disp(policy);


%% b

% Channel states and actions defined
X = {'B', 'G'};
U = {0, 1};

% Given transition probabilities
P = [
    0.9 0.1 0.9 0.1;
    0.5 0.5 0.9 0.1
];

% Cost parameters given
zeta = 0.75;
eta = 0.6;

% This is our cost function
c = [zeta; -eta; 0]; % [ζ; -η; 0]

% Transition probabilities in matrix form
P_matrix = [
    P(1, 2) P(1, 4);
    P(2, 2) P(2, 4);
];

% Define the constraint matrix Aeq and vector beq for equalities
Aeq = [
    P_matrix(1, 1) - P_matrix(1, 2),   1, -1;
    P_matrix(2, 1) - P_matrix(2, 2),   0, -1;
    1 - P_matrix(1, 1),                0, 0;
    1 - P_matrix(2, 1),                0, 0;
];
beq = [0; 0; P(1, 1); P(2, 1)];

% Lower bound and upper bound values
LowerBound = [0; 0; 0];
upperBound = [1; 1; 1];

% Solve the linear programming problem
opts = optimoptions('linprog', 'Display', 'off');
[optU, valueFinal, chcktrue] = linprog(c, [], [], Aeq, beq, LowerBound, upperBound, opts);

%Optimization succession control
if chcktrue ~= 1
    error('Optimization failed because it is about infeasible or unbounded.');
```

```
end

% Optimal policies
optimalU = round(optU(2));
optimalX = round(optU(1));

% Result
fprintf('Optimal Policy:\n');
fprintf('   When the channel is in state %s, the encoder should use the channel (u = 1).\n', X{optimalX+1});
fprintf('   When the channel is in state %s, the encoder should not use the channel (u = 0).\n', X{2 - optimalX});
fprintf('Objective Value (Minimum Cost): %.2f\n', valueFinal);


%% Q2

%% b

% This is our parameters in Kalman Filter
A = [1/2, 1, 0; 0, 2, 1; 0, 0, 2];
C = [4, 0, 0];
Q = eye(3);
R = 1;

T = 1000; % Total number of steps given in question

% Estimation initialization
postXT = zeros(3, 1);
postPT = eye(3);

% For getting results, we need to make storage
estXT = zeros(3, T);
diffPriXT = zeros(3, T);
mTildeT = zeros(3, T);

% Main function of Kalman filter
for t = 1:T
    % Prediction function
    priXT = A * postXT;
    priPT = A * postPT * A' + Q;

    % Measurement generation
    yt = C * priXT + sqrt(R) * randn();

    % Step of update
    Kt = priPT * C' / (C * priPT * C' + R);
    postXT = priXT + Kt * (yt - C * priXT);
    postPT = (eye(size(priPT)) - Kt * C) * priPT;

    % Storage
    estXT(:, t) = postXT;
    mTildeT(:, t) = yt - C * priXT;
    diffPriXT(:, t) = priXT - postXT;
end

% Plotting
timet = 1:T;
figure;
plot(timet, estXT(1, :), 'r', 'LineWidth', 2);
hold on;
plot(timet, estXT(2, :), 'g', 'LineWidth', 2);
```

```
plot(timet, estXT(3, :), 'b', 'LineWidth', 2);
legend('x1_t', 'x2_t', 'x3_t');
xlabel('Plot of Time Step');
ylabel('Plot of State Estimate');
title('Estimation of Kalman Filter');

figure;
plot(timet, mTildeT(1, :), 'r', 'LineWidth', 2);
hold on;
plot(timet, mTildeT(2, :), 'g', 'LineWidth', 2);
plot(timet, mTildeT(3, :), 'b', 'LineWidth', 2);
legend('tilde1', 'tilde2', 'tilde3');
xlabel('Plot of Time step');
ylabel('Plot of Measurement Residual');
title('Measurement Residuals');

figure;
plot(timet, diffPriXT(1, :), 'r', 'LineWidth', 2);
hold on;
plot(timet, diffPriXT(2, :), 'g', 'LineWidth', 2);
plot(timet, diffPriXT(3, :), 'b', 'LineWidth', 2);
legend('Priori Diff for 1', 'Priori Diff for 2', 'Priori Diff for 3');
xlabel('Plot of Time step');
ylabel('Plot of Difference');
title('Difference of State Estimates');

%% c
% Different initial for our covariance matrix
initPtPostDiff = eye(3) * 10;

% Different init for getting Kalman Filter
initXtPostDiff = zeros(3, T);
for t = 1:T
    % Prediction
    priXT = A * initXtPostDiff(:, t);
    priPT = A * initPtPostDiff * A' + Q;

    % Measurement generate
    yt = C * priXT + sqrt(R) * randn();

    % Update Step
    Kt = priPT * C' / (C * priPT * C' + R);
    initXtPostDiff(:, t) = priXT + Kt * (yt - C * priXT);
    initPtPostDiff = (eye(size(priPT)) - Kt * C) * priPT;
end

if isequal(postPT, initPtPostDiff)
    disp('Riccati recursions is unique. It means it converged to the same limit.');
else
    disp('Riccati recursions is not unique. It means it did not converge to the same limit.');
end
```