Assignment 3: Manipulation Estimation and Control

Submitted by: Sushanth Jayanth (AndrewID: sushantj)

1. Dead Reckoning and Kalman Filter

1. a. linearized approximation of the system as a function of the current state

1.) a.) given state model:

$$q[k+1] = \begin{bmatrix} q_1[k] + T(u_1[k] + v_1[k])\cos q_3[k] \\ q_2[k] + T(u_1[k] + v_1[k])\sin q_3[k] \\ q_3[k] + T(u_2[k] + v_2[k]) \end{bmatrix} - (1)$$

where $q = \begin{bmatrix} x_k \\ y_k \\ \theta \end{bmatrix}$

$\therefore$ eq (1) is of the form

$$\hat{x}(k+1|k) = f(\hat{x}(k|k), u(k), k)$$

$$\therefore F(k) = \left.\frac{\partial f}{\partial x}\right|_{x = \hat{x}(k|k)}$$

in our case, to find $F(q[k], u[k])$ we consider:

$$F(q[k], u[k]) = \left.\frac{\partial f}{\partial x}\right|_{x = \hat{x}(k|k)}$$

i.e

$$F(q[k], u[k]) = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \frac{\partial f_1}{\partial q_2} & \frac{\partial f_1}{\partial q_3} \\ \frac{\partial f_2}{\partial q_1} & \frac{\partial f_2}{\partial q_2} & \frac{\partial f_2}{\partial q_3} \\ \frac{\partial f_3}{\partial q_1} & \frac{\partial f_3}{\partial q_2} & \frac{\partial f_3}{\partial q_3} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & -T(u_1[k]) \sin q_3[k] \\ 0 & 1 & T(u_1[k]) \cos q_3[k] \\ 0 & 0 & 1 \end{bmatrix}$$

similarly,

$$G(q[k], u[k]) = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} \\ \frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2} \end{bmatrix}$$

$$= \begin{bmatrix} T\cos q_3[k] & 0 \\ T\sin q_3[k] & 0 \\ 0 & T \end{bmatrix}$$

Similarly,

$$\gamma\left(q[k], u[k]\right) = \begin{bmatrix} \frac{\partial f_1}{\partial v_1} & , & \frac{\partial f_1}{\partial v_2} \\ \frac{\partial f_2}{\partial v_1} & , & \frac{\partial f_2}{\partial v_2} \\ \frac{\partial f_3}{\partial v_1} & , & \frac{\partial f_3}{\partial v_2} \end{bmatrix}$$

$$= \begin{bmatrix} T\cos q_3[k] & 0 \\ T\sin q_3[k] & 0 \\ 0 & T \end{bmatrix}$$

∴ Having found F, G & γ: we can use this in the linearized system equation:

$$q[k+1] = F\left(q[k], u[k]\right).q[k] \quad G\left(q[k], u[k]\right).u[k]$$

$$\gamma\left(q[k], u[k]\right).v[k]$$

1. b. Estimation of Covariance Matrix for process and measurement noise

```
% import the calibrate.mat file to get:
% Train time params
% - t_groundtruth (absolute time values)
% - q_groundtruth (robot states at the above mentioned time values)
% - u (inputs to robot at time values without any noise)
%
% Test time values
% - t_y (a vector of times associated with the GPS measurement)
% - y (noisy GPS measurements taken at the times in t y)

load CMU/Assignment_Sem_1/MEC/Assignment_3/'Problem1 (EKF)'/calibration.mat
```

The error between noisy measurement from actual sensor and groundtruth is recorded. This is then used to calculate the covariance of the **measurement noise**.

```
% get length of time vector
time_duration = size(t_groundTruth,2);

% create a dummy array which will hold the error values
measure_errors = zeros(2,250);

for t = 1:time_duration
    if mod(t,10) == 0
        time = t/10;
        measure_errors(:,time) = (y(:,time) - q_groundTruth(1:2,t));
    end
end

W = cov(measure_errors.')
```

```
W = 2x2
    1.8817    0.0632
    0.0632    2.1384
```

To calculate the process noise, we use the linearized system model derived in **1.a.**

$$q[k+1] \approx F(q[k], u[k])q[k] + G(q[k])u[k] + \Gamma(q[k])v[k]$$

In the above equation, we know all values except $v[k]$. Therefore we solve the equation for $v[k]$ as shown in the below code to get the **process noise**.

```
% create a dummy array which will hold the error values
process_errors = zeros(2,2500);
timestep = 0.01
```

```
timestep = 0.0100
```

```
for t = 1:time_duration-1
    % time = t/10;

    % calculate the F matrix:
    F = [1 0 -timestep*u(1,t)*sin(q_groundTruth(3,t));
         0 1 timestep*u(1,t)*cos(q_groundTruth(3,t));
         0 0 1];

    % calculate the G matrix:
    G = [timestep*cos(q_groundTruth(3,t))  0;
         timestep*sin(q_groundTruth(3,t))   0;
         0                          timestep];

    gamma = G;

    process_errors(:,t) = gamma \ (q_groundTruth(:,t+1) - F*q_groundTruth(:,t) - G*u(:,t));
end

V = cov(process_errors.')
```

```
V = 2x2
    0.2591    0.0010
    0.0010    0.0625
```

1. c. i. Dead Reckoner

After finding the process and measurement noise, we can make predictions on the position of the robot using sensor readings alone

```
% V and W calculated from previous question
W = [1.8817 0.0632; 0.0632 2.1384]
```

```
W = 2x2
    1.8817    0.0632
    0.0632    2.1384
```

```
V = [0.2591 0.0010; 0.0010 0.0625]
```

```
V = 2x2
    0.2591    0.0010
    0.0010    0.0625
```

```
load CMU/Assignment_Sem_1/MEC/Assignment_3/'Problem1 (EKF)'/kfData.mat



% define placeholder for states and P
q_hat = zeros(3,2501);


% define the intitial state estimate
q_hat(:,1) = [0.355 ; -1.590; 0.682];
```

```
% define the initial P
P = [25 0 0; 0 25 0; 0 0 0.154];

time_duration = size(q_groundtruth,2);
timestep = 0.01;

for t1 = 1:time_duration-1
    v = mvnrnd([0;0], V);
    q_hat(:,t1+1) = [ q_hat(1,t1) + timestep.*(u(1,t1) + v(1)).*cos(q_hat(3,t1));
                      q_hat(2,t1) + timestep.*(u(1,t1) + v(1)).*sin(q_hat(3,t1));
                      q_hat(3,t1) + timestep.*((u(2,t1) + v(2)))];
end

plot(q_hat(1,:), q_hat(2,:))
hold on
plot(q_groundtruth(1,:), q_groundtruth(2,:))
hold off
ylabel('y (m)')
xlabel('x (m)')
legend({'prediction', 'ground truth'})
```

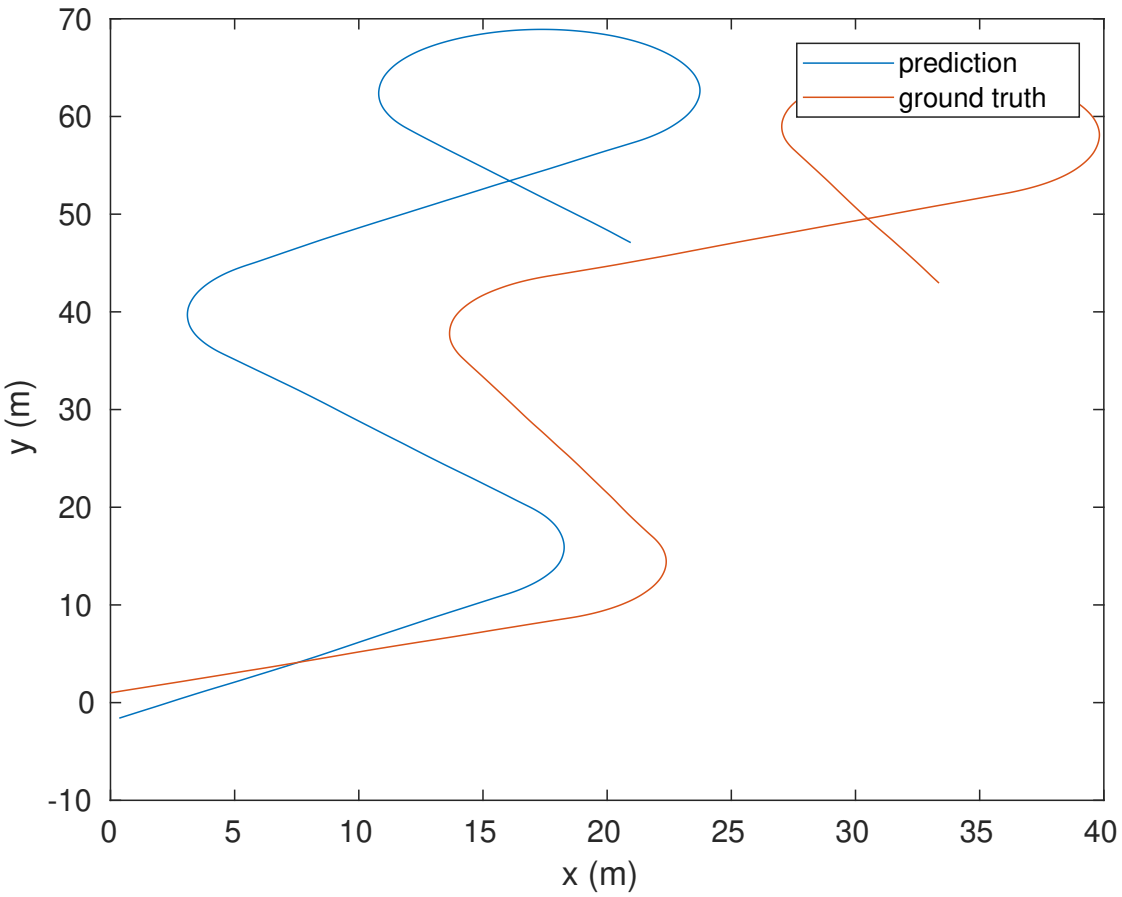

Figure 1. Dead Reckoning Tracking

1. c. ii. Fully Extended Kalman Filter

```
clear q_hat
% define placeholder for states and P
q_hat = zeros(3,2501);


% define the intitial state estimate
q_hat(:,1) = [0.355 ; -1.590; 0.682];

H = [1 0 0; 0 1 0];

% iterate over 2500 time entries (defined as time_duration)
for t1 = 1:time_duration-1
    % prediction step
    v = mvnrnd([0;0], V);

    q_hat(:,t1+1) = [ q_hat(1,t1) + timestep.*(u(1,t1) + v(1)).*cos(q_hat(3,t1));
                   q_hat(2,t1) + timestep.*(u(1,t1) + v(1)).*sin(q_hat(3,t1));
                   q_hat(3,t1) + timestep.*((u(2,t1) + v(2)))]; % eq. 8.37
    if mod(t1,10) == 0
        % time is not absolute time, but just an index position
        time = t1/10;

        % calculate the F matrix:
        F = [1 0 -timestep*u(1,t1)*sin(q_groundtruth(3,t1));
             0 1 timestep*u(1,t1)*cos(q_groundtruth(3,t1));
             0 0 1];

        gamma = [timestep*cos(q_groundtruth(3,t1))  0;
                 timestep*sin(q_groundtruth(3,t1))   0;
                 0                          timestep];

        P = F*P*transpose(F) + gamma*V*transpose(gamma); % eq. 8.38
        S = H*P*transpose(H) + W; % eq. 8.43
        R = P*(transpose(H))*inv(S); % eq. 8.44

        % RECHECK THIS!!! (should be actual - predicted value)
        v2 = y(:,time) - q_hat(1:2,t1+1); % eq. 8.42

        P = P - R*H*P;
        q_hat(:,t1+1) =  q_hat(:,t1+1) + R*v2;
    end
end
% EKF estimates
plot(q_hat(1,:), q_hat(2,:))
hold on
% ground truths
plot(q_groundtruth(1,:), q_groundtruth(2,:))
% GPS measurements
scatter(y(1,:), y(2,:))
hold off
ylabel('y (m)')
xlabel('x (m)')
legend({'prediction', 'ground truth', 'GPS measurement'})
```

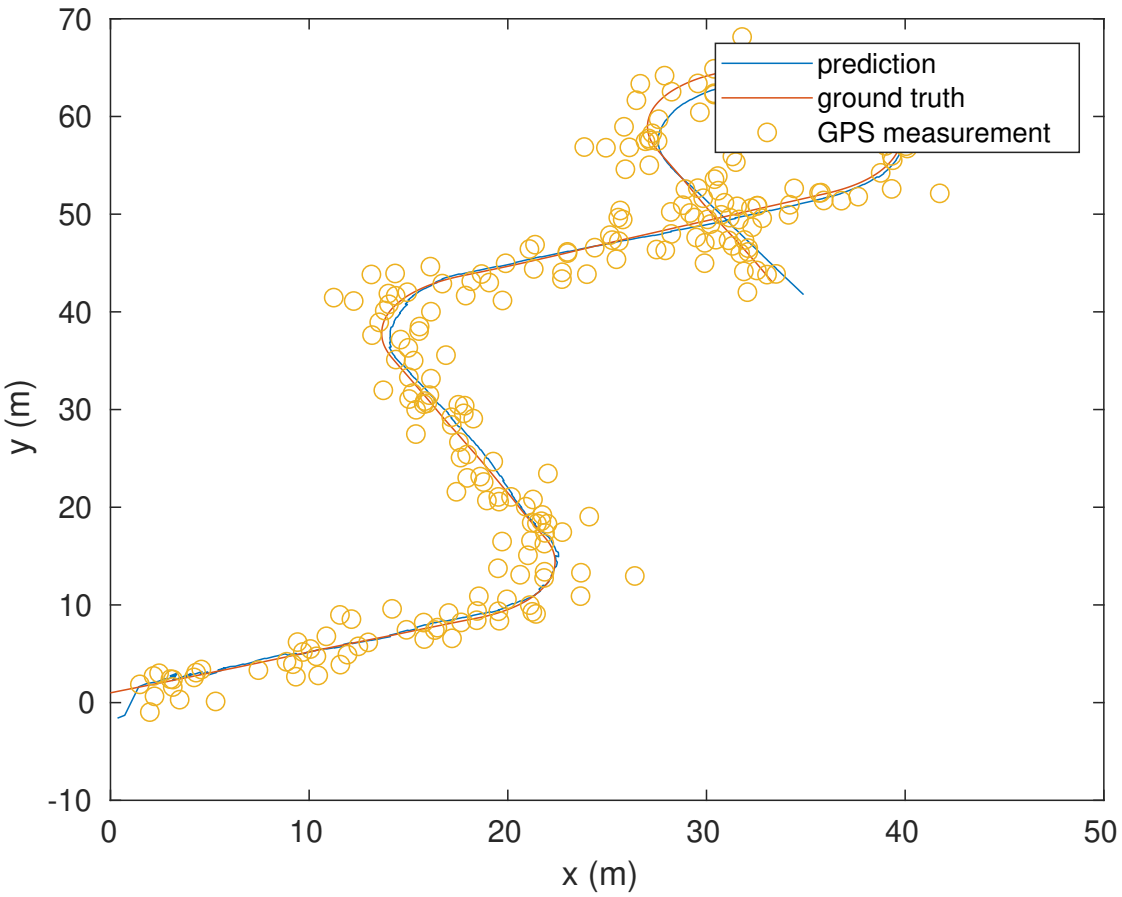

Figure 2. Kalman Filter Estimation

Scaling W down by a factor of 100

```
clear W
W = [1.8817 0.0632; 0.0632 2.1384]*0.01
```

```
W = 2x2
    0.0188    0.0006
    0.0006    0.0214
```

```
clear q_hat
% define placeholder for states and P
q_hat = zeros(3,2501);


% define the intitial state estimate
q_hat(:,1) = [0.355 ; -1.590; 0.682];

H = [1 0 0; 0 1 0];

% iterate over 2500 time entries (defined as time_duration)
for t1 = 1:time_duration-1
    % prediction step
    v = mvnrnd([0;0], V);

    q_hat(:,t1+1) = [ q_hat(1,t1) + timestep.*(u(1,t1) + v(1)).*cos(q_hat(3,t1));
```

```
                    q_hat(2,t1) + timestep.*(u(1,t1) + v(1)).*sin(q_hat(3,t1));
                    q_hat(3,t1) + timestep.*((u(2,t1) + v(2)))]; % eq. 8.37
    if mod(t1,10) == 0
        % time is not absolute time, but just an index position
        time = t1/10;

        % calculate the F matrix:
        F = [1 0 -timestep*u(1,t1)*sin(q_groundtruth(3,t1));
             0 1 timestep*u(1,t1)*cos(q_groundtruth(3,t1));
             0 0 1];

        gamma = [timestep*cos(q_groundtruth(3,t1))  0;
                 timestep*sin(q_groundtruth(3,t1))   0;
                 0                          timestep];

        P = F*P*transpose(F) + gamma*V*transpose(gamma); % eq. 8.38
        S = H*P*transpose(H) + W; % eq. 8.43
        R = P*(transpose(H))*inv(S); % eq. 8.44

        v2 = y(:,time) - q_hat(1:2,t1+1); % eq. 8.42

        P = P - R*H*P;
        q_hat(:,t1+1) =  q_hat(:,t1+1) + R*v2;
    end
end
% EKF estimates
plot(q_hat(1,:), q_hat(2,:))
hold on
% ground truths
plot(q_groundtruth(1,:), q_groundtruth(2,:))
% GPS measurements
scatter(y(1,:), y(2,:))
hold off
ylabel('y (m)')
xlabel('x (m)')
legend({'prediction', 'ground truth', 'GPS measurement'})
```

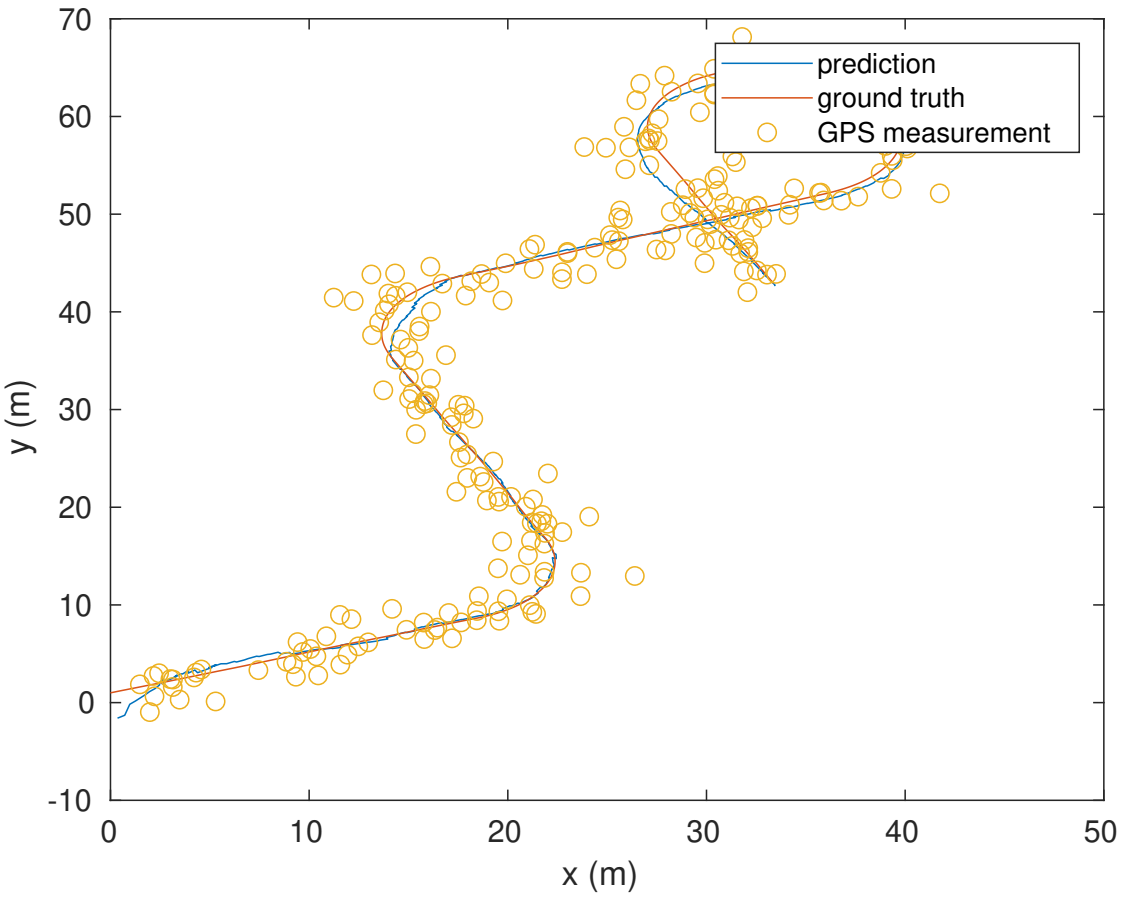

Figure 3. Kalman Filter Estimation with scaled down W

As seen in Figure. 3. scaling down the covarice of measurement noise causes the tracking to be slightly worse. The reason behind this tracking is:

- Reducing the covariance vales of the measurement noise indicates that the GPS measurements are highly accurate (i.e. they are almost very close to ground truth)
- Therefore, during the update step, the EKF will give more weightage to the GPS measurements (even over the prediction values) causing the tracking to become worse (as GPS measurements as seen in scatter plot are not always near the ground truth)

2. Particle Filter

The following parameters were varied in creating the filter:

- Number of particles
- Covariance matrix of measurement noise (W)
- Covariance matrix of process noise (V)

The number of particles were initially 100. After increasing the number of particles to 2000, the tracking seemed to improve. This seems to be because having more particles (denser particle cloud) around the robot would give a better estimate of the robot pose. This also causes better tracking.

The covariance matrix was initially set to an identity matrix for both process and measurement noise. Reducing the covariance values (especially in the non-diagonal elements) made the tracking better. The tuned covariance matrices are shown below (further tuning could improve it even more):

```
% V and W
W = [0.6 0.3; 0.3 0.6];
V = [0.63 0.25; 0.25 0.63];
```

```
rng(0); % initialize random number generator

b1 = [5,5]; % position of beacon 1
b2 = [15,5]; % position of beacon 2

% load pfData.mat
load pfData.mat

% define timestep for prediction
timestep = 0.1;

numSteps = size(q_groundTruth,2);
% initialize movie array
M(numSteps) = struct('cdata',[],'colormap',[]);

% Define number of particles in filter
num_particles = 2000;

% define placeholder for robot pose at every step
robot_pose = zeros(3,numSteps);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize particles
x_start = 0;
x_end = 20;

y_start = 0;
y_end = 10;

theta_start = 0;
theta_end = 2*pi;

x_rand = (x_end-x_start).*rand(num_particles,1) + x_start;
y_rand = (y_end-y_start).*rand(num_particles,1) + y_start;
theta_rand = (theta_end-theta_start).*rand(num_particles,1) + theta_start;

particles = [x_rand y_rand theta_rand];
particles = transpose(particles);

weights = ones(num_particles, 1)/num_particles;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% here is some code to plot the initial scene
figure(1)
plotParticles(particles); % particle cloud plotting helper function
hold on
plot([b1(1),b2(1)],[b1(2),b2(2)],'s',...
    'LineWidth',2,...
    'MarkerSize',10,...
    'MarkerEdgeColor','r',...
    'MarkerFaceColor',[0.5,0.5,0.5]);
drawRobot(q_groundTruth(:,1), 'cyan'); % robot drawing helper function
axis equal
axis([0 20 0 10])
M(1) = getframe; % capture current view as movie frame

disp('hit return to continue')

% V and W
W = [0.6 0.3; 0.3 0.6];
V = [0.63 0.25; 0.25 0.63];

% beacon 1,2 locations in [x;y]
b1 = [5;5];
b2 = [15;5];

% iterate particle filter in this loop
for k = 2:numSteps

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %              put particle filter prediction step here               %

    for t1 = 1:num_particles
        v = mvnrnd([0;0], V);
        particles(1,t1) = particles(1,t1) + timestep.*(u(1,k) + v(1)).*cos(particles(3,t1));
        particles(2,t1) = particles(2,t1) + timestep.*(u(1,k) + v(1)).*sin(particles(3,t1));
        particles(3,t1) = particles(3,t1) + timestep.*((u(2,k) + v(2))); % eq. 8.37
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                put particle filter update step here
    % weight particles
    for t1 = 1:num_particles
        % get the beacon reading (groundtruth) (2x1 vector)
        current_dist = y(:,k-1);
        % get the output h(k) (2x1 vector)
        h = [ sqrt((particles(1,t1) - b1(1,1))^2 + (particles(2,t1) - b1(2,1))^2);
              sqrt((particles(1,t1) - b2(1,1))^2 + (particles(2,t1) - b2(2,1))^2) ];

        % knowing the current pos of particles and noisy sensor measurement
        % we can merge the distributions of the groundtruth
        % and noisy measurement. We can merge multivariate pdfs using
        % matlabs mvnpdf function
        weights(t1) = weights(t1) * mvnpdf(current_dist, h, W);
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % resample particles

    % normalize the weights
```

```
    weights = weights/sum(weights);

    % make cumulative weight vector
    cumulative_sum = cumsum(weights);

    % make a dummy weight vector
    new_weights = zeros(size(weights));
    newparticles = zeros(size(particles));

    weight_size = size(weights);

    % resample
    for i = 1:weight_size

        % generate a random number between 0,1
        rand_num = rand();

        % find the element which is just lesser than rand_num
        [minval, idx] = min(abs(cumulative_sum - rand_num));
        newparticles(:, i) = particles(:, idx);
        new_weights(i) = 1/num_particles;
    end


    weights = ones(num_particles,1)/num_particles;
    particles = newparticles;

    % calculate the robot estimate from the particle cloud
    particle_estimate = mean(particles,2);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % plot particle cloud, robot, robot estimate, and robot trajectory here %
    % here is some code to plot the initial scene
    figure(1)
    clf
    plotParticles(particles); % particle cloud plotting helper function
    hold on
    plot([b1(1),b2(1)],[b1(2),b2(2)],'s',...
        'LineWidth',2,...
        'MarkerSize',10,...
        'MarkerEdgeColor','r',...
        'MarkerFaceColor',[0.5,0.5,0.5]);
    drawRobot(q_groundTruth(:,k), 'cyan'); % robot drawing helper function
    drawRobot(particle_estimate, 'cyan'); % robot drawing helper function
    plot(q_groundTruth(1,1:k), q_groundTruth(2,1:k))
    axis equal
    axis([0 20 0 10])
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % capture current figure and pause
    M(k) = getframe; % capture current view as movie frame
    pause(0.001)

end

% when you're ready, the following block of code will export the created
% movie to an mp4 file
videoOut = VideoWriter('/home/sush/CMU/Assignment_Sem_1/MEC/Assignment_3/result_2000.mp4','Motion JPE
videoOut.FrameRate=20;
open(videoOut);
```

```
for k=1:numSteps
  writeVideo(videoOut,M(k));
end
close(videoOut);

close

% helper function to plot a particle cloud
function plotParticles(particles)
plot(particles(1, :), particles(2, :), 'go')
line_length = 0.1;
quiver(particles(1, :), particles(2, :), line_length * cos(particles(3, :)), line_length * sin(partic
end


% helper function to plot a differential drive robot
function drawRobot(pose, color)

% draws a SE2 robot at pose
x = pose(1);
y = pose(2);
th = pose(3);

% define robot shape
robot = [-1 .5 1 .5 -1 -1;
          1  1 0 -1  -1 1 ];
tmp = size(robot);
numPts = tmp(2);
% scale robot if desired
scale = 0.5;
robot = robot*scale;

% convert pose into SE2 matrix
H = [ cos(th)   -sin(th)  x;
      sin(th)    cos(th)  y;
      0          0        1];

% create robot in position
robotPose = H*[robot; ones(1,numPts)];

% plot robot
plot(robotPose(1,:),robotPose(2,:),'k','LineWidth',2);
rFill = fill(robotPose(1,:),robotPose(2,:), color);
alpha(rFill,.2); % make fill semi transparent
end
```