1. [50] A robot is moving on a horizon grid of size 10 as shown in Fig. 1. The state of the robot at each time step k is given by its location with respect to the grid, i.e. $\xi_k \in \{1,2,\dots,10\}$. The robot can only move to the right by 1 unit or stay in the current cell at each time step through its control action $u_k \in \{1,0,-1\}$ where 1 for "move right" action, 0 for "stay" and -1 for "move left" action. The robot has a vision sensor to detect whether the wall, i.e. $z_k \in \{1,0\}$ where 1 for "facing wall" and 0 for "facing empty space". The system model and the sensor model are given by the following probabilities.

   i) The initial position of the robot is located at 4, 5, or 6 with equal probability:

$$bel(\xi_0 = i) = \begin{cases} \frac{1}{3}, & i \in \{4,5,6\} \\ 0, & otherwise \end{cases}$$

   ii) The measurement probability of the vision sensor with respect to the wall is given as:
   - Correct positive: $p(z_k = \mathbf{1} | \xi_k \in \{2,6,7\}) = 0.9$,
   - False negative: $p(z_k = \mathbf{0} | \xi_k \in \{2,6,7\}) = 0.1$,
   - False positive: $p(z_k = \mathbf{1} | \xi_k \notin \{2,6,7\}) = 0.05$,
   - Correct negative: $p(z_k = \mathbf{0} | \xi_k \notin \{2,6,7\}) = 0.95$

   iii) The robot motion model:
   - The "move right" command will cause it to move right at 90% rate (if $i \neq 10$), and stay otherwise; $p(\xi_{k+1} = i+1 | \xi_k = i, u_k = \mathbf{1}) = 0.9$, $p(\xi_{k+1} = i | \xi_k = i, u_k = \mathbf{1}) = 0.1$ (for $i < 10$)
   - The "move left" command will cause it to move to left at 80% rate (if $i \neq 1$), and stay otherwise; $p(\xi_{k+1} = i-1 | \xi_k = i, u_k = -\mathbf{1}) = 0.8$, $p(\xi_{k+1} = i | \xi_k = i, u_k = -\mathbf{1}) = 0.2$ (for $i > 1$)
   - If $\xi_k = 1$ or $\xi_k = 10$, then the robot will stay for any command to move it out of state range; $p(\xi_{k+1} = 1 | \xi_k = 1, u_k = -\mathbf{1}) = p(\xi_{k+1} = 10 | \xi_k = 10, u_k = \mathbf{1}) = 1$
   - The "stay" command keeps the robot still, i.e. $p(\xi_{k+1} = i | \xi_k = i, u_k = \mathbf{0}) = 1$, $\forall i \in \{1,\dots,10\}$.

   Assume that the robot control action sequence is given by $u_{0:6} = \{1,1,1,-1,0,-1,0\}$. Using Matlab, implement the Bayes filter for localization of the robot, i.e. compute $bel(\xi_k = i)$ with $k = 0,1,2,3,4,5,6$ for each location $i = 1,2,\dots,10$. Construct $bel(\xi_k)$ as a 8×10 matrix of the following form;

$$bel = \begin{bmatrix} bel(\xi_0 = 1) & bel(\xi_0 = 2) & \cdots & bel(\xi_0 = 10) \\ bel(\xi_1 = 1) & bel(\xi_1 = 2) & \cdots & bel(\xi_1 = 10) \\ \vdots & \vdots & \vdots & \vdots \\ bel(\xi_7 = 1) & bel(\xi_7 = 2) & \cdots & bel(\xi_7 = 10) \end{bmatrix}$$

   Get the plot of each row of bel matrix. Based on the result, find the most-likely location of the robot at time step $k = 6$ and $k = 7$. The skeleton code ("HW4Q1skeleton.m") is provided and you can fill up the commented section.

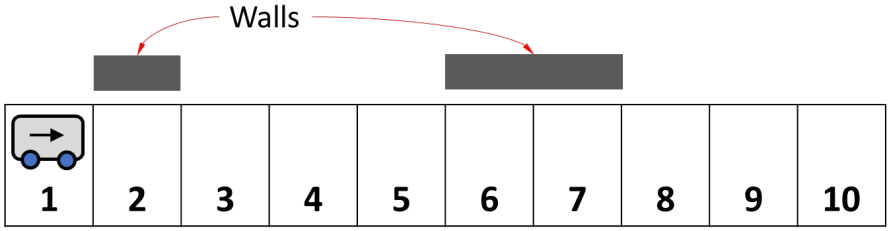
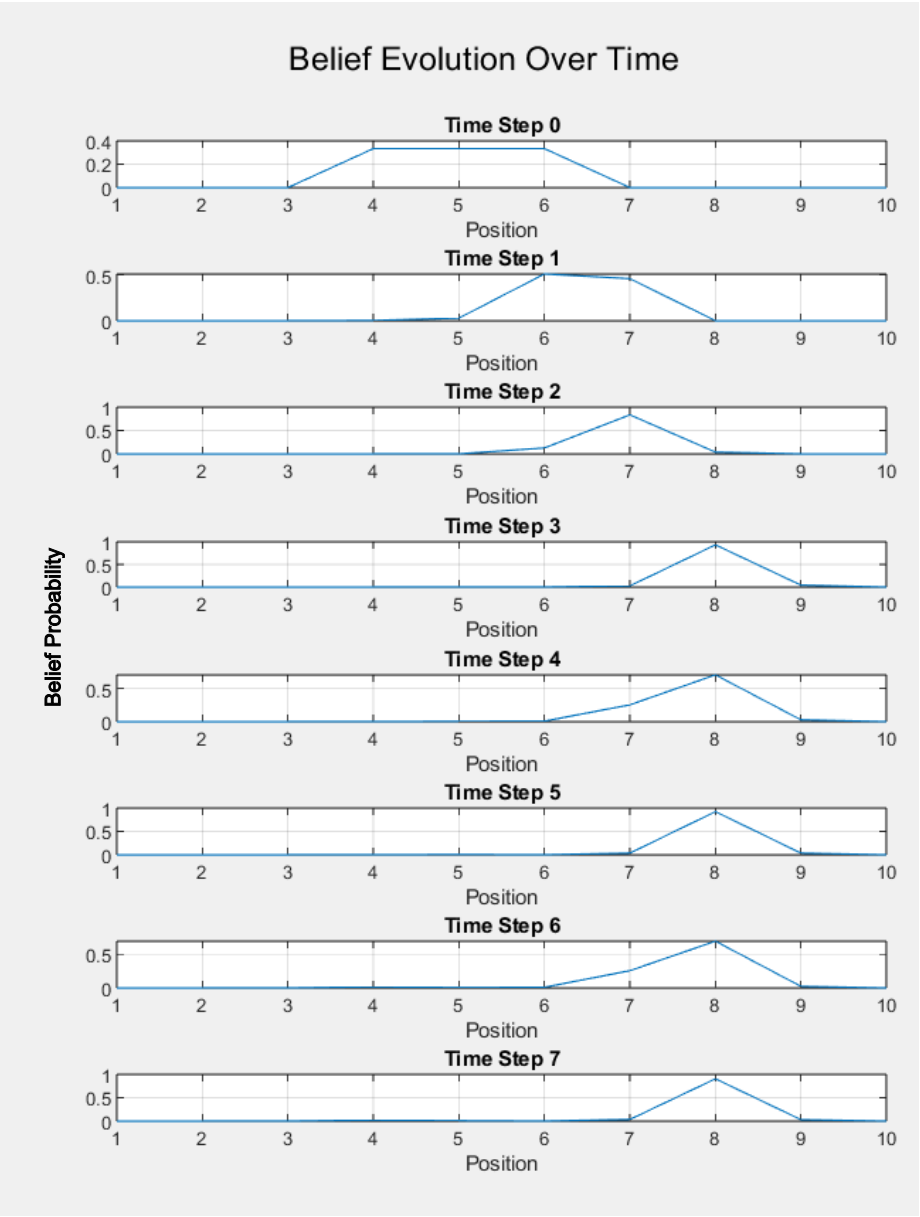


Fig. 1 Robot moving on a horizontal grid



At time step k=6, belief states robot is at 8, at time step k=7, belief states robot is in 8

Code on last page

2. [50] You are given the map of an environment as occupancy grid (Fig. 2), obtained from range sensors, the values of which are between [0,1], indicating the likelihood of having an obstacle in a cell of the grid. Dijkstra algorithm can be used to find a feasible path (collision-free) from a starting point to a goal point, by assuming that the cells of the occupancy grid are vertices connected to their neighboring cells. Complete the following functions in the given Matlab code:
   a) get_neighbors, which has arguments of the current cell (as coordinates $[x,y]$), and the occupancy grid map. This function should return all the neighbors of your current cell as a cell array of coordinates;
   b) get_edge_cost, which has arguments of the prior node and the current node (as coordinates $[x,y]$), and the occupancy grid map. This function should return the edge cost between the prior node and the current node. The costs should allow you to plan the path using Dijkstra's algorithm, by taking into account the occupancy information (i.e. give priority to the cells with lower probabilities);
   c) complete the update step in the while loop (which implements Dijkstra's algorithm), by checking the neighbors (using get_neighbors) of the prior node and calculating the costs (using get_edge_cost).

   The code might be quite slow, you can comment the 'pause' in the plotting functions.

   In your report, attach the code segments you've implemented, and the figures resulting from the code for the given start = [40,15] and goal = [21,37] points. [Optional] You can also try other starting and goal points and see how things change.

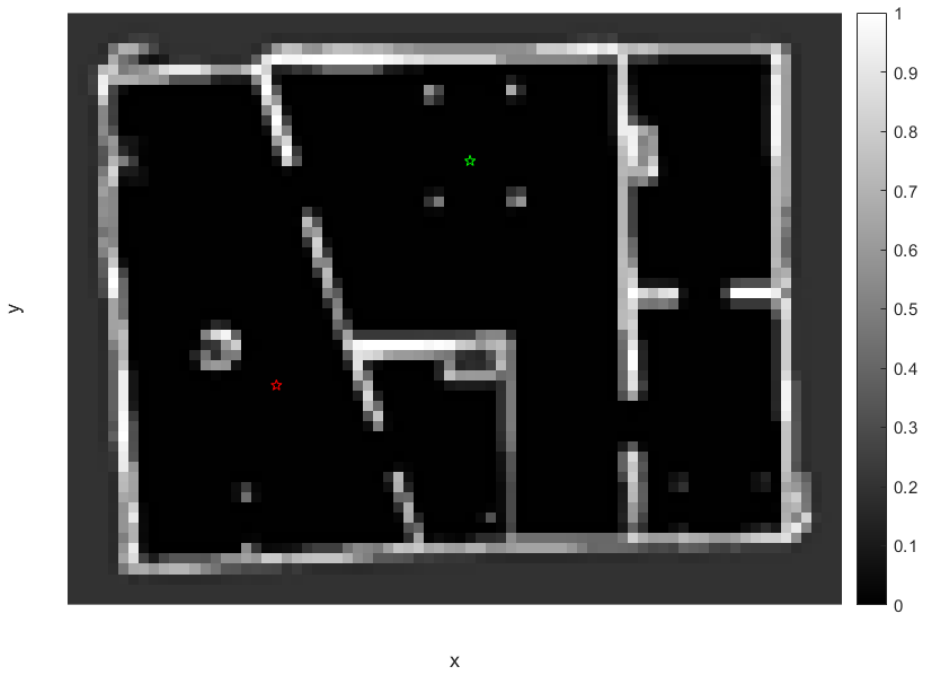
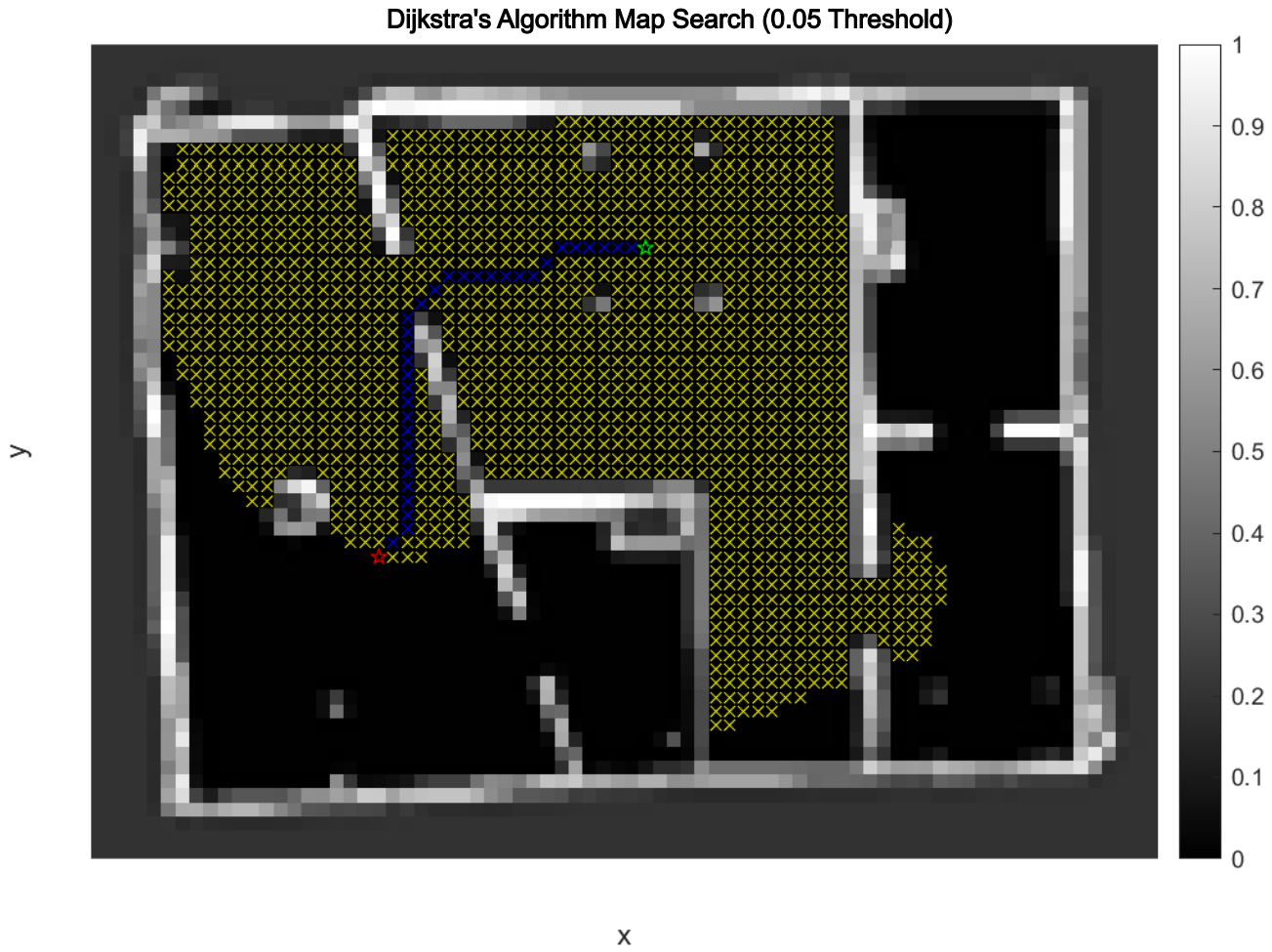
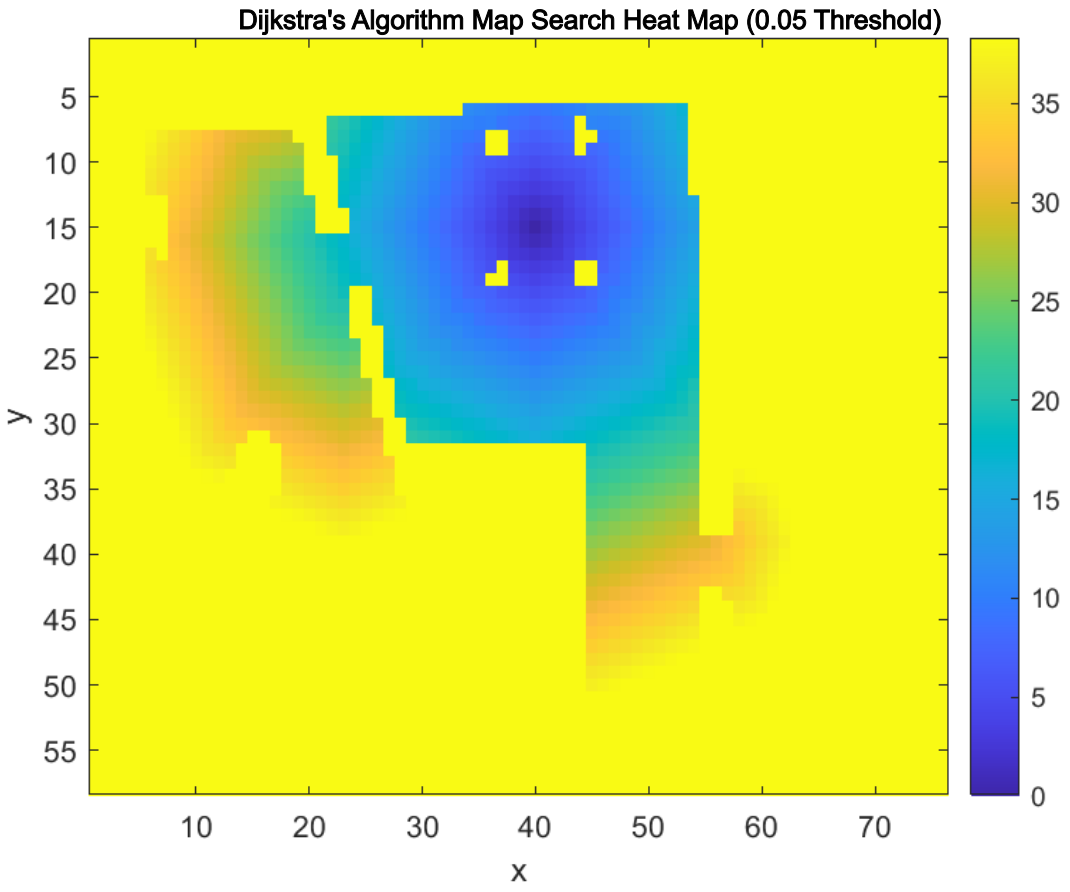


Fig. 2 Occupancy grid map with start and goal


Dijkstra's Algorithm Map Search (0.05 Threshold)
y
x
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0


Dijkstra's Algorithm Map Search Heat Map (0.05 Threshold)
y
x
5
10
15
20
25
30
35
40
45
50
55
10
20
30
40
50
60
70
35
30
25
20
15
10
5
0

Question 2 Code

```
%%%% Code for MTE544 - HW4Q2
%%%% Complete the code where indicated
%%%% The expected outcome is: given an occupancy map, the
shortest path from given start to given goal
clear;
close all;

% Load the given occupancy map
load occupancy_map.mat
disp(["Size of the map: ", size(omap)]);
disp(["Number of cells: ", numel(omap)]);

% Coordinates of the starting cell (try to set different starts and goals
and see what happens)
start = [40,15];
% Coordinates of the goal cell
goal = [21,37];
% Plot the occupancy map with the start and goal cells
plot_map(omap, start, goal)

% Matrix to store the cost values for each cell
% Initialize with infinity
costs = ones(size(omap)) * Inf;

% Matrix to store cells that have already been visited -> closed set
% Initialize with zeros (1 means that the cell is part of the closed list)
closed = zeros(size(omap));

% Cell array to store the coordinates and cost of priors for each
visited cell
priors = cell(size(omap) + [2,2]);
% Each element of the cell array element is defined as: [x,y,cost]
% Initialize coordinates to Inf and costs to -1
priors(:,:) = {[Inf,Inf,-1]};

% Initial condition: first prior_node is set to be the starting cell, with
cost set to zero
prior_node = start;
costs(start(1), start(2)) = 0;
while ~isequal(prior_node, goal)
    % Initialize open_cost as costs
    open_costs = costs;
    % Find all those cells that have already been expanded (in the
closed set) and set their cost to Inf
    open_costs(find(closed==1)) = Inf;

    % Find the cell with the minimum cost in the open list
    [mx,my] = find(open_costs==min(min(open_costs)));
    mincost = ind2sub(size(open_costs), [mx ,my]);
    % Take the first found
    x = mincost(1,1);
    y = mincost(1,2);
    % % Find linear index of minimum open cost
    % [minval, linidx] = min(open_costs(:));
    % % If no finite open cell, break
    % if isinf(minval)
    %     break
    % end
    % [x, y] = ind2sub(size(open_costs), linidx);


    % Break the loop if there are no more open cells
    if open_costs(x,y) == Inf
      break
    end

    % Set the minimum cost cell as prior_node and assign it in the
closed list
    prior_node = [x,y];
    closed(x,y) = 1;

    % Update costs and prior nodes for the neighbors
    neighbors = get_neighbors(prior_node, size(omap));
    for i = 1:length(neighbors)
        neighbor = neighbors{i};
        edge_cost = get_edge_cost(prior_node, neighbor, omap);
        new_cost = costs(prior_node(1), prior_node(2)) + edge_cost;
        if new_cost < costs(neighbor(1), neighbor(2))
            costs(neighbor(1), neighbor(2)) = new_cost;
            priors{neighbor(1), neighbor(2)} = [prior_node(1),
prior_node(2), new_cost];
        end
    end

    % Visualize the cells that have already been expanded
    plot_expanded(prior_node, start, goal)
end

% When done, rewind the path from goal to start
if isequal(prior_node, goal)
    disp(["N. of cells expanded : ", nnz(closed)])
    disp(["Cost of the path : ", costs(goal(1), goal(2))])
    path_length = 0;
    while priors{prior_node(1), prior_node(2)}(3) >= 0
      plot_path(prior_node, goal)
      previous = priors{prior_node(1), prior_node(2)}(1:2);
      path_length = path_length + norm(prior_node - previous);
      prior_node = previous;
    end
    disp(["Length of the path: ", path_length])
else
    disp("No feasible path could be found.")
end

% Visualize the costs as a colored map
plot_costs(costs)

%   Find all the neighbors of the given cell.
%   Arguments:
%   current_cell: coordinates of a cell as [x, y]
%   omap_size: size of the occupancy map (nx, ny)
%
%   Output:
%   neighbors: list of up to eight neighbor coordinate as cell array {(x1,
y1), (x2, y2), ...}
function neighbors = get_neighbors(current_cell, omap_size)
    neighbors = {};
    %%%%%%%% Complete code here (a) %%%%%%%%%%
    directions = [-1,-1; -1,0; -1,1; 0,-1; 0,1; 1,-1; 1,0; 1,1];
    for i = 1:8
        nx = current_cell(1) + directions(i,1);
        ny = current_cell(2) + directions(i,2);
        if nx >= 1 && nx <= omap_size(1) && ny >= 1 && ny <=
omap_size(2)
            neighbors{end+1} = [nx, ny];
        end
    end
end

%   Calculate the cost to move from prior_node to current_node.
%   Arguments:
%   prior_node, current_node: coordinates of a cell as [x, y]
%   omap: occupancy map
%
%   Output:
%   edge_cost: calculated cost
function edge_cost = get_edge_cost(prior_node, current_node, omap)
    %%%%%%%% Complete code here (b) %%%%%%%%%%
    if omap(current_node(1), current_node(2)) <= 0.05
        edge_cost = norm(current_node - prior_node);
    else
        edge_cost = Inf;
    end
end

%%%%%% Plotting functions %%%%%%%
function plot_map(omap, start, goal)
  imshow(omap', 'InitialMagnification', 1000)
  colorbar
  hold on
  plot(start(1), start(2), 'gp');
  plot(goal(1), goal(2), 'rp');
  xlabel('x')
  ylabel('y')
end

function plot_expanded(expanded, start, goal)
  if isequal(expanded, start) || isequal(expanded, goal)
    return
  end
  plot(expanded(1), expanded(2), 'yx');
  pause(1e-6)
end

function plot_path(path, goal)
  if isequal(path, goal)
    return
  end
  plot(path(1), path(2), 'bx')
  pause(1e-6)
end

function plot_costs(cost)
  figure;
  imagesc(cost')
  colorbar
  xlabel('x')
  ylabel('y')
end
```

Question 1 Code

```
clear;
Nsteps = 7; % number of time steps
Nstates = 10;   % number of states

% probabilities for motion model
pRR = 0.9;  % move right command and end up advancing
pRS = 0.1;  % move right command but end up staying
pLL = 0.8;  % move left command and end up retracting
pLS = 0.2;  % move left command but end up staying

% probabilities for sensor model
pCP = 0.9;  % correct positive
pFN = 0.1;  % false negative
pCN = 0.95;  % correct negative
pFP = 0.05;  % false positive

% locations of wall
W = [2,6,7];

% sequence of actions
U = [1,1,1,-1,0,-1,0];    % from u_0 to u_6

%% simulated motion
x = zeros(Nsteps,1);
x(1) = randi([4, 6]);   % initial position, random between (including) 3 and 5

% measurement model for the initial position
if ismember(x(1),W)
    if rand < pCP
        Z(1) = 1;
    else
        Z(1) = 0;
    end
else
    if rand < pCN
        Z(1) = 0;
    else
        Z(1) = 1;
    end
end

% running simulaion to generate robot motion
for k = 1:Nsteps-1
    if U(k) == 1
        if rand < pRR
            x(k+1) = x(k) + 1;  % advance to right
        else
            x(k+1) = x(k);
        end
    elseif U(k) == -1
        if rand < pLL
            x(k+1) = x(k) - 1;  % advance to right
            if x(k+1) < 1
                x(k+1) = 1;
            end
        else
            x(k+1) = x(k);
        end
    else
        x(k+1) = x(k);
    end

    % measurement model
    if ismember(x(k+1),W)
        if rand < pCP
            Z(k+1) = 1;
        else
            Z(k+1) = 0;
        end
    else
        if rand < pCN
            Z(k+1) = 0;
        else
            Z(k+1) = 1;
        end
    end

end

%% Bayes Filter
% initialization of a (batch) belief
bel = zeros(Nsteps+1,Nstates);
% bel(xi_0) is one of {3,4,5} with equal probability
bel(1,4:6) = 1/3;

inv_eta = 0;    % the variable for 1/eta
% initialization of a (batch) bar_belief
brbel = zeros(Nsteps,Nstates);

% the main loop
for k = 1:Nsteps
    % prediction step, i.e. computing bar_belief using the motion model
    for i = 1:Nstates
        brbel(k,i) = 0;
        for j = 1:Nstates
            if U(k) == 1
                if i == j + 1
                    brbel(k,i) = brbel(k,i) + bel(k,j) * pRR;
                elseif i == j
                    brbel(k,i) = brbel(k,i) + bel(k,j) * pRS;
                end
            elseif U(k) == -1
                if j > 1
                    if i == j - 1
                        brbel(k,i) = brbel(k,i) + bel(k,j) * pLL;
                    elseif i == j
                        brbel(k,i) = brbel(k,i) + bel(k,j) * pLS;
                    end
                else
                    if i == 1
                        brbel(k,i) = brbel(k,i) + bel(k,j);
                    end
                end
            else
                if i == j
                    brbel(k,i) = brbel(k,i) + bel(k,j);
                end
            end
        end
    end
    % correction step, i.e. computing belief using the sensor output
    for i = 1:Nstates
        if Z(k) == 1
            if ismember(i, W)
                bel(k+1,i) = brbel(k,i) * pCP;
            else
                bel(k+1,i) = brbel(k,i) * pFP;
            end
        else
            if ismember(i, W)
                bel(k+1,i) = brbel(k,i) * pFN;
            else
                bel(k+1,i) = brbel(k,i) * pCN;
            end
        end
        inv_eta = inv_eta + bel(k+1,i);
    end
    % normalize belief using inverse of eta
    bel(k+1,:) = bel(k+1,:)/inv_eta;
    % reset the inv_eta for the next time step
    inv_eta = 0;
end

figure;
sgtitle('Belief Evolution Over Time');
ylabel('Belief Probability');
for i = 1:Nsteps+1
    subplot(Nsteps+1,1,i);plot(bel(i,:)');
    grid on;
    xlabel('State');

    title(['Time Step ', num2str(i-1)]);
end
```