Mobile Robotics EE/CE 468

Homework Assignment 04

Lyeba Abid - la07309
Ali Muhammad - aa07190

Question:	1	2	3	4	5	Total
Points:	25	25	25	0	25	100
Score:						

Problem 1 [20 Points]

Solution: A sensor measurement $z = (r, \theta)^T$ where r is the measured distance to the landmark, and θ is the bearing to the landmark.

Assuming that we are in a 2d-plane, the robot pose can be assumed to be $x = (x_x, x_y, x_\theta)$ where x_x, x_y are the x and y coordinates, and x_θ is its orientation The landmark position can be assumed to be $l = (l_x, l_y)$.

The sensor model $p(z \mid x, l)$ represents the probability of obtaining a measurement z given the true position x and the location of the landmark l. Since both the range and bearing measurements are subject to zero-mean Guassian noise variances σ_r^2 and σ_θ^2, their measurements can be described as:
$r = r_t + \varepsilon_r$
$\theta = \theta_t + \varepsilon_\theta$
where r_t and θ_t are the true values, and ε_r and ε_θ are the noise.

Then the sensor model can be constructed as a joint probability distribution of these independent Guassian distributions:

$$p(z \mid x, l) = p(r, \theta \mid x, l) = p(r \mid x, l)p(\theta \mid x, l) \tag{1}$$

Then we define each of the independent probability distributions.

For the range measurement r, calculate the expected range r_t from the robot's position x, to the landmark l. Then
$r_t = \sqrt{(l_x - x_x)^2 + (l_y - x_y)^2}$
Then our change in the range is $\triangle r = r - r_t$.
We then define our probability density function for the range r as:

$$p(r \mid x, l) = \frac{1}{\sqrt{2\pi\sigma_r^2}} exp(-\frac{(\triangle r)^2}{2\sigma_r^2}) \tag{2}$$

Similarly, for the bearing, we first calculate the expected bearing θ_t from the robots orientation x_θ to the landmark. Then
$\theta_t = \arctan 2(l_y - x_y, l_x - x_x) - x_\theta$
Our angle difference then becomes $\triangle\theta$ (taking the smallest angle difference between measured bearing and expected bearing). We define our probability density function for θ as:

$$p(\theta \mid x, l) = \frac{1}{\sqrt{2\pi\sigma_\theta^2}} exp(-\frac{(\triangle\theta)^2}{2\sigma_\theta^2}) \tag{3}$$

Substituting (2) and (3) into (1), we get the joint probability density for the measurement $z_i = (r, \theta)^T$:

$$p(z_i \mid x, l) = \frac{1}{\sqrt{2\pi\sigma_r^2}} exp(-\frac{(\triangle r)^2}{2\sigma_r^2}) \times \frac{1}{\sqrt{2\pi\sigma_\theta^2}} exp(-\frac{(\triangle\theta)^2}{2\sigma_\theta^2})$$

$$p(z_i \mid x, l) = \frac{1}{2\pi\sigma_r\sigma_\theta} exp\left(-\left[\frac{(\triangle r)^2}{2\sigma_r^2} + \frac{(\triangle\theta)^2}{2\sigma_\theta^2}\right]\right) \tag{4}$$

which gives us the complete sensor model.

Problem 2 [**20 Points**]

Solution:

```
clc; clear; close all;

% Define environment parameters
map_length = 2;          % Length of the map (lane) in meters
cell_size = 0.1;         % Size of each grid cell in meters
num_cells = map_length / cell_size;  % Number of cells

% Initialize the occupancy grid
occupancy_grid = zeros(1, num_cells) + 0.5;  % Initially all cells are uncertain (0.5)

% Robot position (left-most cell)
robot_position = 0;

% Range sensor measurements
measurements = [101, 82, 91, 112, 99, 151, 96, 85, 99, 105]/100;

% Sensor model parameters
prob_occupied_near = 0.3;
prob_occupied_far = 0.6;
max_distance = 0.2;  % 20 cm

% Process measurements and update the occupancy grid
for measurement_index = 1:length(measurements)
    measurement = measurements(measurement_index);

    % Update the occupancy grid based on the measurement
    for cell_index = 1:num_cells
        cell_position = (cell_index - 1) * cell_size;  % Representative point for the cell
        distance_to_cell = cell_position - robot_position;

        if distance_to_cell < measurement && distance_to_cell > -max_distance
            % Update probability based on sensor model
            occupancy_grid(cell_index) = prob_occupied_near * occupancy_grid(cell_index);
        elseif distance_to_cell >= measurement
            % Update probability based on sensor model
            occupancy_grid(cell_index) = prob_occupied_far * occupancy_grid(cell_index);
        end
    end
end
```

```

% Normalize the probability values
occupancy_grid = occupancy_grid / sum(occupancy_grid);

% Compute representative points
representative_points = (1:num_cells) * cell_size - cell_size / 2;

% Plot the probability values against representative points
figure;
bar(representative_points, occupancy_grid, 'BarWidth', 0.9);
xlabel('Position (meters)');
ylabel('Normalized Probability');
title('Normalized Probability Mass Function (PMF)');

% Threshold for binary occupancy
occupancy_threshold = 0.5;
binary_occupancy = occupancy_grid >= occupancy_threshold;


% Draw the map based on the probabilities
figure;
hold on;
for i = 1:num_cells
    if occupancy_grid(i) > 0.1
        % If the cell is likely occupied
        rectangle('Position', [i*cell_size-cell_size/2, -0.05,
            cell_size, 0.1], 'FaceColor', 'k');
    else
        % If the cell is likely unoccupied
        rectangle('Position', [i*cell_size-cell_size/2, -0.05,
            cell_size, 0.1], 'FaceColor', 'w');
    end
end
axis equal;
axis([-0.1, 2.1, -0.2, 0.2]);
xlabel('Position (m)');
title('Occupancy Grid Map');
```

Listing 1: Grid-Based Occupancy Mapping code

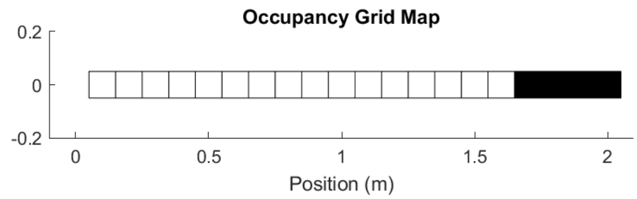

Figure 1: Occpancy Grid Map

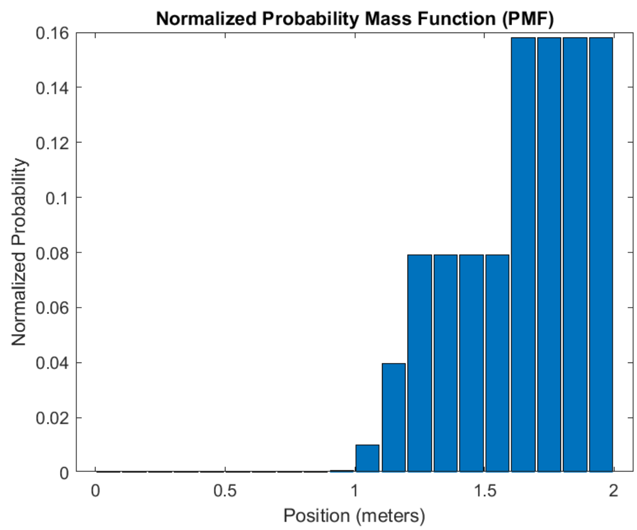

Figure 2: Probability Mass Function

The provided MATLAB code simulates the update of an occupancy grid, a representation of a physical environment where each cell indicates the likelihood of being occupied. The environment parameters, such as the map length and cell size, are defined. The occupancy grid is initialized with uncertain values. Range sensor measurements, representing distances from obstacles, are processed to update the occupancy probabilities using a sensor model. The sensor model incorporates parameters for the probability of occupancy near and far from the robot. The resulting probabilities are normalized, and a bar plot is generated to visualize the normalized probability mass function. A binary occupancy map is created based on a threshold, and a map of the environment is drawn, indicating likely occupied and unoccupied cells based on the updated probabilities.

Problem 3 [20 Points]

Solution: For the particle filter, or in other words, the Monte Carlo Localization (MCL), the main script, "turtlebotMCLLocalization" was updated as follows:

```
% ... Existing Code ...
N = 1000;
particles = initializeParticles(N, M);

SIGMA = repmat(diag([0.0001; 0.0001; 0.0001]), 1,N);

% ... Existing Code ...

% ... main while loop ...
    % ... Existing Code ...

    % Compute linear and angular velocities from robot wheel
      velocities
    v = (groundTruth(4) + groundTruth(5))*params.WHEEL_DIA/4; % v =
       (wl + wr)* d /4
    w = (groundTruth(5) - groundTruth(4))*params.WHEEL_DIA/(2*
      params.WHEEL_SEP);
    % Compute odometry and store the ground truth.


    u = ([groundTruth(4); groundTruth(5)] ) * sampleTime * params.
      WHEEL_DIA/2;
    Q = eye(2)*.000001;

    % Extract lines
    cart(:,1) = cart(:,1)-0.032;
    [theta,rho] = cart2pol(cart(:,1),cart(:,2));
    [particles, SIGMA] = mclIncrementalLocalization(particles,
      SIGMA, u, Q, [theta'; rho'], M, params, sqrt(10), params.
      WHEEL_SEP);
    % MCL_incrementalLocalization(particles, SIGMA_seq, u, Q, S, M,
       params, g, b)
    waitfor(rate);
    it = it + 1;
    disp("Time")
    rate.TotalElapsedTime
end
```

Listing 2: Main script for MCL

The above function first initializes the number of particles to 1000, ensuring a dense particle cloud, then sending this number to the initialize particles function within the environment defined by matrix "M".

The initialization function basically generates an initial distribution of particles. It determines the boundaries of the map 'M', and initializes 'N' particles within those bounds. It is defined as follows:

```
function [particles]= initializeParticles(N, M)
    i = 0;
    left_max = 0;
    right_max = 0;
    up_max = 0;
    down_max = 0;
    for angle = M(1,:)
        i = i + 1;
        if angle == 0
            if M(2,i) > right_max
                right_max = M(2,i);
            end
        end
        if angle == pi
            if M(2,i) > left_max
                left_max = M(2,i);
            end
        end
        if angle == pi/2
            if M(2,i) > up_max
                up_max = M(2,i);
            end
        end
        if angle == -pi/2
            if M(2,i) > down_max
                down_max = M(2,i);
            end
        end
    end

    particles = zeros(4,N);

    particles(1,:) = unifrnd(-left_max, right_max,1, N );
    particles(2,:) = unifrnd(-down_max, up_max,1, N );
    particles(3,:) = unifrnd(0, 2*pi,1, N );
    particles(4,:) = ones(1, N);
end
```

Listing 3: initializeParticles function

The covariance matrix "SIGMA" is initialized with very low uncertainty values (0.0001), reflecting high confidence in the initial pose estimates of the particles, which may be adjusted as per the specific characteristics of the robot and its environment. In the main loop 'v', 'w', and 'u' represent the linear velocity, angular velocity and control signal, computer from the ground truth. The process noise covariance matrix "Q" is initialized from 'u', and is set to a low value to indicate minimal process noise. The laser scanned data represented as Cartesian coordinates, is converted to polar form ("theta" and "rho"), which is then used for updating the particles' weights. The values are then sent to the "mclIncrementalLocalization" function, which updates the particles' weights according to how well they predict the observed data.

```
function [new_particles, SIGMA_posterori] =
    mclIncrementalLocalization(particles, SIGMA_seq, u, Q, S, M,
    params, g, b)
% C_TR represents the covariance matrix for the translation and
    rotation
% of each line segment detected by the sensor, initialized with a
% standard deviation of 0.01 for both translation and rotation.
C_TR = diag([repmat(0.01^2, 1, size(S, 2)) repmat(0.01^2, 1, size(S
    , 2))]);

% Extract lines from the sensor data in polar coordinates, with S
    (1,:)
% being the angles and S(2,:) being the distances. R is the
    measurement
% noise covariance for each detected line, based on C_TR.
[Z, R, ~] = extractLinesPolar(S(1,:), S(2,:), C_TR, params);

% Estimate robot pose
[new_particles, SIGMA_posterori] = mclStep(particles, SIGMA_seq, Z,
     Q, R,u,b,M,g);
```

Listing 4: mclIncrementalLocalization function

The "mclStep" function as defined above, performs a single iteration of particle filtering. It takes the current set of particles (`particles`), each with an associated covariance `SIGMA_seq`, control actions `u` along with additional parameters. It uses odometry to predict the next state, using the previous state and the control actions. For each particle, it associates sensor measurements with map features using another function "mclAssociation", and updates the particles weights using the "measurementModel" function which computes the likelihood of the sensor measurement given the particles predicted state. Then the error covariance for each particle is updated, and the weights are then normalized to represent a probability distribution. The particles are then resampled based on their weights, favoring those that have higher likelihoods to form the new set of particles. The function is defined below:

```
function [new_particles, SIGMA_posterior] = mclStep(particles,
    SIGMA_seq, Z, Q, R,u,b,M,g)
% SIGMA_seq = It contains all the error covariances corresponding
    to
     particles(:,1:5)
     N = size(particles,2);
     X_new = particles(1:3, :);
```

```
    W_new = particles(end, :);
    for i = 1:N
        X_prev = particles(1:3,i);
        [X_bar,Fx, Fu]= odometry(X_prev, u, b);
        X_new(1:3,i)=X_bar;
        [z_, H_seq, R_seq] = mclAssociation(X_new, SIGMA_seq(:,i:i
            +2), Z, R, M, g);
        if size(z_,2) > 0
            for j = 1:size(z_,2)
                % The loop is only for when their are multiple
                    measurements
                xx = R_seq(2*(j-1)+1:2*j,2*(j-1)+1:2*(j));
                pdf = measurementModel(z_(:,j), R_seq(2*(j-1)+1:2*j
                    ,2*(j-1)+1:2*(j)), M(:,j), X_bar);
                W_new(1,i) = pdf*W_new(1,i);
            end
        % Update the SIGMA_seq
        SIGMA_seq(:,i:i+2) = update_sigma(SIGMA_seq(:,i:i+2), H_seq
            , Fu, Fx, Q, R_seq);
        end
    end
    % Normalize
    W_new = W_new/sum(W_new);

    % Resample
    new_particles = stochastic_universal_sampling(X_new, W_new, N);
    SIGMA_posterior = SIGMA_seq;
    new_particles(:, 1:5)
end
```

Listing 5: mclStep function

The associated helper functions are defined as follows:

```
function [z_, H_seq, R_seq] = mclAssociation(x, P, Z, R, M, g)
first_entry = true;
H_seq = [];
R_seq = [];
z_ = [];
Associated_landmarks = zeros(size(M,2)) ;%Associated landmark index
for i = 1:size(Z,2)
    min_d = inf;
    min_m = 0;
    min_Hx = 0;
    found = false;
    for j = 1:size(M,2)
        [h, H_x] = measurementFunction(x, M(:,j));
        v_ijt = Z(:,i) - h;
        Sigma_ijINt = H_x*P*H_x'+R(:,:,i);
```

```
        d_curr = transpose(v_ijt)*Sigma_ijINt^(-1)*v_ijt;
        if d_curr <= min_d
            found = true;
            min_d = d_curr;
            min_m = i;
            min_Hx = H_x;
            Associated_landmarks(j) = 1;
        end
    end
    if found == true
        if first_entry == true && min_d <= g^2
            % disp("A Correspondence is Found")
            first_entry = false;
            z_ = [z_ Z(:, i)];
            H_seq = min_Hx;
            R_seq = R(:,:,min_m);
        elseif min_d <= g^2
            % disp("Another Correspondence is Found")
            z_ = [z_ Z(:, i)];
            % H_seq(end+1:end+2,:) = min_Hx;
            H_seq = [H_seq;
                    min_Hx];
            R_seq(end+1:end+2,end+1:end+2) = R(:,:,min_m);
        end
    end
end
```

Listing 6: mclAssociation

```
function pdf = measurementModel(z, R, M, X)
% The function returns the weight of the i_th particle
mu_z = [angdiff(M(1,1) - X(3,1));
        M(2,1) - ( X(1,1)*cos(M(1,1)) + X(2,1)*sin(M(1,1)));
]; %This is the expected value of z given x_bar
R = R
pdf = mvnpdf(z,mu_z,R);
end
```

Listing 7: measurementModel

```
function [SIGMA] = update_sigma(SIGMA, H, Fu, Fx, Q, R)
R = resize_Tensor(R);
SIGMA_BAR = Fx*SIGMA*Fx'+Fu*Q*Fu';
SigmaINt = H*SIGMA_BAR*H'+R;
Kk = SIGMA_BAR*H'*(SigmaINt)^(-1);
SIGMA = SIGMA_BAR - Kk*SigmaINt*Kk';
end
```

Listing 8: update_sigma

```
function particles = stochastic_universal_sampling(X, w, N)
% The w array needs to be normalized
index = [];
cdf = zeros(1,N);
% create CDF
cdf(1) = w(1);
for i = 2:size(w,2)
    cdf( i) = cdf( i-1) + w(i);
end

u = unifrnd(0,1/N);
for j = 1:N
    i = 1;
    while u > cdf(:, i)
        i = i + 1;
    end
    index = [index i];
    u = u + 1/N;
end
X_new = X(:,index);

particles = [X_new;
    ones(1, N)];
end
```

Listing 9: stochastic_universal_sampling

Based on the above implementation, our robot was able to localize itself within the environment, as shown in the following figure:

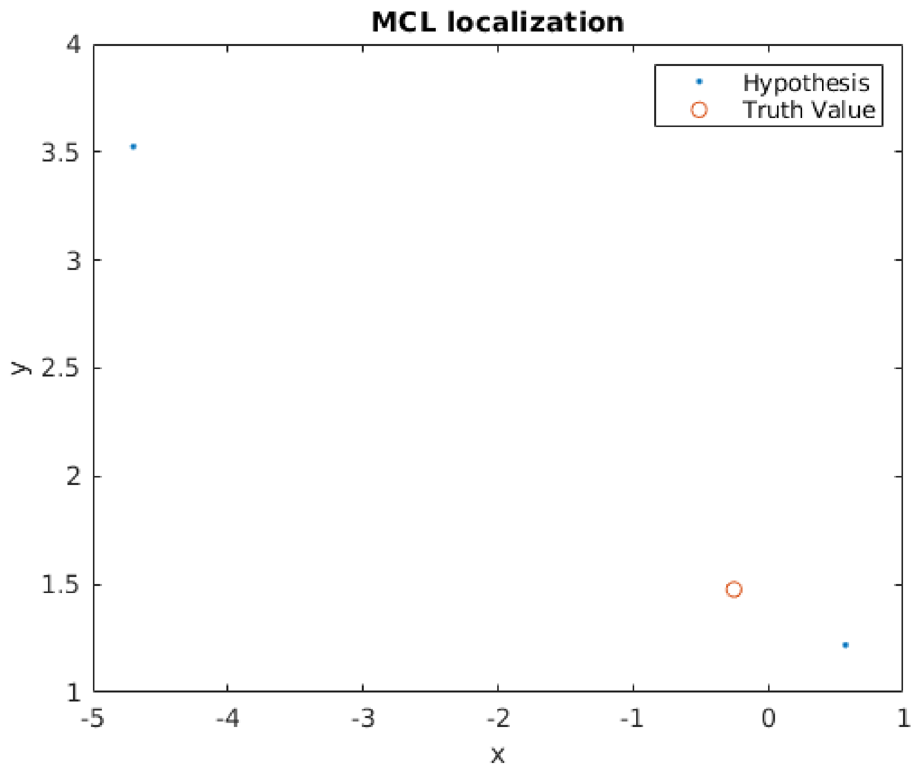

Figure 3: Robot Localization using MCL

Our hypothetical value does come close to the true value, however, not completely accurate. Although we weren't able to resolve this, we believe that maybe the process noise is not accurately captured in the prediction step. Moreover, more iterations may be required for the algorithm to converge all the way. There is also the possibiliy of incorrect modelling of the sensor noise in the measurement model which can lead to incorrect weighting of the particles.

However, we were unable to fix this.

Problem 5 [20 Points]

Solution:

Lyeba Abid

(a) I spent approximately 6-7 hours on this homework assignment.

(b) I took the lead in coding the occupancy mapping algorithm, ensuring the correct implementation of the sensor model and probability updates. I also created the visualizations for the probability mass function and the occupancy grid map. Additionally, I collaborated with team members to debug and optimize the code for better performance.

(c) Understand the fundamentals of grid-based occupancy mapping and sensor models before diving into the code. Pay attention to parameter tuning in the sensor model for accurate probability updates. Debugging is crucial; use print statements or debugging tools to identify errors.

(d) In retrospect, our group's efforts in implementing the grid-based occupancy mapping algorithm were rewarding. Successfully coding the algorithm and observing the visualizations provided a hands-on understanding of how sensor data influences the probabilistic representation of an environment. The assignment solidified my comprehension of probability mass functions and their application in robotics. However, lingering questions persist regarding the algorithm's robustness in dynamic environments with moving obstacles. Fine-tuning the sensor model parameters posed a notable challenge, emphasizing the sensitivity of such algorithms to parameter adjustments. Moving forward, I aspire to explore sensor fusion techniques to broaden my understanding and seek guidance on refining the sensor model for diverse environments. Overall, this assignment has fueled my interest in advancing my skills in robotics and sensor fusion, prompting a desire to explore more complex scenarios and contribute to the field's evolving landscape.

Ali Muhammad Asad

(a) I spent around 10-12 hours on this homework assignment.

(b) I completed Problems 1 and 3; Sensor Model and Particle Filter / Monte Carlo Localization.

(c) Brush up on, and keep your concepts related to probability and statistics fresh. This will help you understand the sensor model and the particle filter better. Especially your concepts on particle filter and the basics of Monte Carlo Localization. And **most importantly**, start this homework early please. It is a lot of work, and you will need time to debug and optimize your code.

(d) The sensor model question was fairly easier. Once I understood what the question was asking, and that a joint probability distribution was required which could be done by splitting z into its respective components since r and θ are independant, it was fairly simple to implement. The particle filter question was a bit more challenging. I had to read up on the particle filter and the Monte Carlo Localization algorithm to understand what was required, and what approach to take. It also meant that I had to understand the existing code, and understand what new functions to implement for MCL. It was a great learning experience though, and although the code is not perfect, I would like to explore more about particle filters and MCL in the future to correct this implementation.