EE5114 Autonomous Robot Navigation

EE5114 Continuous Assessment 2

NAME: MADKAIKAR ATHARVA ATUL

MATRICULATION NO: A0268376M

1. Please explain all your filled-in codes (6 parts in total) with a snapshot of those lines of code. List formulas you have used for that part of codes if applicable.

A) Motion Prediction: The filled-in code in '**motion_estimation.m**' is responsible for performing motion estimation in the FAST SLAM algorithm using point cloud scan matching. The following are code snippets and explanations:

- **Reshaping P and Q:**
  P and Q, representing matched corners in consecutive lidar scans, are reshaped from (1 x n) arrays to (n x 1) arrays.
  Code:
  ```
  P_reshaped = P';
  Q_reshaped = Q';
  ```
- **Finding Centroids:**
  Centroids of P and Q are evaluated by finding mean of each matrix, representing the average position of the matched corners.
  Code:
  ```
  P_centroid = mean(P_reshaped, 2);
  Q_centroid = mean(Q_reshaped, 2);
  ```
- **Centering P and Q:**
  The Lidar scans in P and Q are centered around their respective centroids by subtracting the centroids.
  Code:
  ```
  P_decentered = P_reshaped - P_centroid;
  Q_decentered = Q_reshaped - Q_centroid;
  ```
- **Computing H:**
  The matrix H is computed as the product of the centred matrices P and Q transposed.
  Code:
  ```
  H = P_decentered * Q_decentered';
  ```
- **Performing SVD on H:**
  SVD is performed on the H matrix to obtain the matrices U, V and S
  Code:
  ```
  [U, S, V] = svd(H);
  ```
- **Determining Rotation Matrix R:**
  The rotation matrix R is determined from the SVD results. Handling the sign of the determinant ensures the proper orientation of the rotation matrix.
  Code:
  ```
  d = sign(det(V * U'));
  R = V * [1, 0; 0, d] * U';
  ```
- **Calculating Translation:**
  The translation is computed using rotation matrix R and the centroids of P and Q
  Code:
  ```
  T = R * P_centroid - Q_centroid;
  translation = T';
  ```
- **Orientation (Theta) Calculation:**
  The orientation (theta) of the robot is computed using the arctangent function (**atan2**). The elements of the rotation matrix R are used to determine the rotation angle.
  Code:
  ```
  theta = -atan2(R(2,1), R(1,1));
  ```

MATLAB code:

```
% Missing codes start here ...
% Use matched corners to calculate rotation and translation between
% the two frames

% Reshaping P and Q
P_reshaped = P';
Q_reshaped = Q';

% Find the centroid
P_centroid = mean(P_reshaped, 2);
Q_centroid = mean(Q_reshaped, 2);

% Decentroiding P and Q
P_decentered = P_reshaped - P_centroid;
Q_decentered = Q_reshaped - Q_centroid;

% Calculating the H
H = P_decentered * Q_decentered';

% Performing SVD to get U, S and V matrices
[U, S, V] = svd(H);

% Determines correct orientation of Rotation Matrix
d = sign(det(V * U'));

% Evaluating Rotation Matrix R
R = V * [1, 0; 0, d] * U';

% Get the translation vector
T = R * P_centroid - Q_centroid;
translation = T';

% Evaluating orienation theta of the robot
theta = -atan2(R(2,1), R(1,1));

% Missing codes end here ...
```

Figure 1: Filled in code in motion_estimation.m

B) Feature Extraction: The filled-in code in '**slam_lidar_feat_extrn.m**' computes the angle span of a potential corner, ensuring the angle falls between 60 to 120 degrees. If a corner meets the criteria, its heading direction is calculated and added to the list of detected corners. Following are code snippets and explanations:

- **Vector Calculation:**
  'va' and '**vb'** are calculated as vectors representing the direction of consecutive line segments which intersect to form a potential corner.
  Code:

```
va = [lines(i+1).p1.x - lines(i+1).p2.x, lines(i+1).p1.y - lines(i+1).p2.y];
vb = [lines(i).p2.x - lines(i).p1.x, lines(i).p2.y - lines(i).p1.y];
```

- **Angle Calculation:**
  The angle of span between these vectors is calculated using the dot product and magnitudes.
  Code:

```
dot_product = dot(va, vb);
mag_va = norm(va);
mag_vb = norm(vb);
angle = acos(dot_product / (mag_va * mag_vb));
```

- **Filtering Corners:**
  Corners are considered only if the calculated angle is within the range of 60 to 120 degrees.

- **Heading Direction:**
  If a corner meets the criteria, the heading direction of the corner is calculated based on the vectors **'va'** and **'vb'**.
  Code:

```
delta_x = va(1)/mag_va + vb(1)/mag_vb;
delta_y = va(2)/mag_va + vb(2)/mag_vb;
heading = atan2(delta_y, delta_x);
```

MATLAB Code:

```
% Missing codes start here ...

% Calculate vectors va and vb
va = [lines(i+1).p1.x - lines(i+1).p2.x, lines(i+1).p1.y - lines(i+1).p2.y];
vb = [lines(i).p2.x - lines(i).p1.x, lines(i).p2.y - lines(i).p1.y];

% Calculate angle span of the corner

% Compute the dot product of vectors va and vb
dot_product = dot(va, vb);

% Compute the magnitude of vectors va and vb
mag_va = norm(va);
mag_vb = norm(vb);

% Computing angle span
angle = acos(dot_product / (mag_va * mag_vb));

% Only use corner features having angle from 60 to 120 degrees
if (angle > deg2rad(60) && angle < deg2rad(120))

    % Calculate heading direction of the corner
    % Calculate delta_x and delta_y
    delta_x = va(1)/mag_va + vb(1)/mag_vb;
    delta_y = va(2)/mag_va + vb(2)/mag_vb;

    heading = slam_in_pi(atan2(delta_y, delta_x));

% Missing codes end here ...
```

Figure 2: Filled in code in slam_lidar_feat_extrn.m

Important Formulae:

Formula for calculating angle span of the corner

$$\overrightarrow{v_a} = p_1 - p_2$$

$$\overrightarrow{v_b} = p_3 - p_2$$

$$\alpha = \cos^{-1} \frac{\overrightarrow{v_a} \cdot \overrightarrow{v_b}}{|v_a||v_b|}$$

Formula for calculating heading of the corner

$$\frac{\Delta x}{\Delta y} = \frac{\overrightarrow{v_a}}{|v_a|} + \frac{\overrightarrow{v_b}}{|v_b|}$$

$$\beta = \text{atan2}(\Delta y, \Delta x)$$

C) Split and Merge: The filled-in code in '**slam_lidar_split_merge.m**' efficiently finds the point that is furthest from the line defined by the endpoints. It calculates the perpendicular distances and identifies the point with the maximum distance. This is crucial for subsequent steps in Split-and-Merge algorithm in which if the maximum distance is less than the threshold, all points between first and last point belong to the same line. This is done recursively to fit lines for lidar scans. Following are code snippets and explanations:

- **Initialization:**
  Initialize arrays and variables ('point_distances', 'winner_value', 'winner_index') to store distances and the index of the furthest point.
  Code:

```
point_distances = zeros(1, last);
winner_value = abs(point_distances(1));
winner_index = 1;
```

- **Iteration through dataset to get winner value and index:**
  The code iterates through the points in the dataset and computes the distance of each point from the line formed by the first and the last points. It returns the distance and index of the point that is farthest from the line by performing following:

- **Distance Calculation:**
  For each point in the dataset, calculate the perpendicular distance from the point to the line defined by the endpoints (x1, y1) and (x2, y2).
  Code:
  ```
  distance = abs(((x2 - x1) * (y1 - y_int) - (x1 - x_int) * (y2 - y1)) / sqrt((x2 - x1)^2 + (y2 - y1)^2));
  ```
- **Store Distances:**
  Store the calculated distances in the 'point_distances' array.
  Code:
  ```
  point_distances(i) = distance;
  ```
- **Update Winner:**
  Check if the current point has a greater distance than the current winner. If so, update 'winner_value' and 'winner_index'.
  Code:
  ```
  if distance >= winner_value
      winner_value = distance;
      winner_index = i;
  end
  ```

MATLAB Code

```
% Missing codes start here ...

% Find furthest point to this line
% Initialzie arrays and variables to store distance and the winner
point_distances = zeros(1, last);
winner_value = abs(point_distances(1));
winner_index = 1;

% Iterate through all points in the dataset
for i = 1:last

    % Load the point from the dataset
    x_int = points(i).x;
    y_int = points(i).y;

    % Calculate the distance of the point from the line
    distance = abs(((x2 - x1) * (y1 - y_int) - (x1 - x_int) * (y2 - y1)) / sqrt((x2 - x1)^2 + (y2 - y1)^2));

    % Store the distance in the array
    point_distances(i) = distance;

    % Check if the point is furthest from the line
    if distance >= winner_value
        winner_value = distance;
        winner_index = i;
    end

end

% Missing codes end here ...
```

Figure 3: Filled in code in slam_lidar_split_merge.m

Important Formulae:

Formula used for calculating perpendicular distance of a point from the line.

$$d = \left| \frac{(x_2 - x_1)(y_1 - y) - (x_1 - x)(y_2 - y_1)}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \right|$$

D) Proposal Generation: The filled in code in '**slam.m**' is responsible for generating proposals for particle motion based on motion estimates and noise. The introduced noise helps model the inherent uncertainties and inaccuracies in the motion estimation process. Following are code snippets and explanations:

- **Rotation Update:**
  For each particle (**j**), the code updates the particle's orientation (**theta**) based on the corresponding motion estimate and introduces noise. Both additive and proportional noise are applied to simulate uncertainties in rotation.
  Code:

theta_noise = motion_estimate(j).theta*(1+theta_noise_proportion*randn(1))
+theta_noise_add*randn(1);

particles(j).theta = particles(j).theta + theta_noise;

- **Translation Update:**
  The code also updates the particle's position (**x** and **y**) based on the corresponding motion estimate with additive and proportional noises. The noise accounts for uncertainties in translation.
  Code:

x_noise = motion_estimate(j).x*(1 + translation_noise_proportion*randn(1)) +
translation_noise_add*randn(1);

y_noise = motion_estimate(j).y*(1 + translation_noise_proportion*randn(1)) +
translation_noise_add*randn(1);

particles(j).x = particles(j).x + cos(particles(j).theta)*x_noise - sin(particles(j).theta)*y_noise;

particles(j).y = particles(j).y + sin(particles(j).theta)*x_noise + cos(particles(j).theta)*y_noise;

MATLAB Code

```
%% Motion prediction
for j = 1:particles_count
    % Missing code start here ...

    % Apply rotation to all particles with (additive and proportional) noises

    theta_noise = motion_estimate(j).theta*(1 + theta_noise_proportion*randn(1)) + theta_noise_add*randn(1);
    particles(j).theta = particles(j).theta + theta_noise;

    % Apply translation to all particles with (additive and proportional) noises

    x_noise = motion_estimate(j).x*(1 + translation_noise_proportion*randn(1)) + translation_noise_add*randn(1);
    y_noise = motion_estimate(j).y*(1 + translation_noise_proportion*randn(1)) + translation_noise_add*randn(1);


    particles(j).x = particles(j).x + cos(particles(j).theta)*x_noise - sin(particles(j).theta)*y_noise;
    particles(j).y = particles(j).y + sin(particles(j).theta)*x_noise + cos(particles(j).theta)*y_noise;

    % Missing code end here ...
end
```

Figure 4: Filled in code in slam.m

Important Formulae:

Formulae used in Proposal Generation Code

$$\Delta c_t^{[m]} = \Delta\bar{c_t}(1+\alpha_1 \text{randN}(1)) + \alpha_2 \text{randN}(1)$$
$$\Delta x_t^{[m]} = \Delta\bar{x_t}(1+\alpha_3 \text{randN}(1)) + \alpha_4 \text{randN}(1)$$
$$\Delta y_t^{[m]} = \Delta\bar{y_t}(1+\alpha_3 \text{randN}(1)) + \alpha_4 \text{randN}(1)$$
$$P_t^{[m]}.c = P_t^{[m]}.c + \Delta c_t^{[m]}$$
$$P_t^{[m]}.x = P_t^{[m]}.x + \cos(P_t^{[m]}.c)\Delta x_t^{[m]} - \sin(P_t^{[m]}.c)\Delta y_t^{[m]}$$
$$P_t^{[m]}.y = P_t^{[m]}.y + \sin(P_t^{[m]}.c)\Delta x_t^{[m]} + \cos(P_t^{[m]}.c)\Delta y_t^{[m]}$$

E) Particle Resampling: The filled in code in '**slam_resample.m'** essentially implements the core logic of resampling particles, ensuring that particles with higher weights are more likely to be selected during the resampling process. Following are code snippets and explanations:

- **Initialization**:
  Initialize a copy of the total weight (**weight_copy**) to keep track of the cumulative weight updates during the iteration.
  Code:
  ```
  weight_copy = weight_total;
  ```
- **Random Weight Generation**:
  Generate a random weight value (**W_random**) within the range of 0 to the total weight.
  Code:
  ```
  W_random = (weight_total - 0) * rand;
  ```
- **Particle Selection, Assignment and Weight Reset:**
  Iterate through the particles to find the one whose cumulative weight is less than or equal to that of the random weight. This simulates the process of selecting particles with a probability proportional to their weight. Assign the selected particle to the (**new_particles)** array, and set its weight to the initial weight (**init_weight**). This ensures that the resampled particles have a consistent initial weight.
  Code:
  ```
  for j = 1:particles_count
      weight_copy = weight_copy - particles(j).weight;

      if weight_copy <= W_random
          new_particles(i) = particles(j);
          new_particles(i).weight = init_weight;
          break;
      end
  end
  ```
- **Return the new set of particles:**
  Assigns the (**new_particles**) array after the resampling has been performed to (**particles**) array and returns this array back.
  Code:
  ```
  particles = new_particles;
  ```

MATLAB Code

```
new_particles = struct();

% Copy fields from the original particles to the new_particles
for i = 1:length(fields)
    field = fields{i};
    new_particles.(field) = particles.(field);
end

for i = 1:particles_count

    % Missing codes start here

    % Initialize a copy of the total weight
    weight_copy = weight_total;

    % Generate a random weight value within the total weight range
    W_random = (weight_total - 0) * rand;

    for j = 1:particles_count
        weight_copy = weight_copy - particles(j).weight;

        % Check if the cumulative weight is within the random weight range
        if weight_copy <= W_random
             % Assign the selected particle to the new_particles
            new_particles(i) = particles(j);

            % Set the weight of the new particle to the initial weight
            new_particles(i).weight = init_weight;
            break;
        end
    end

    % Missing codes end here
end

% Return the new set of particles after resampling
particles = new_particles;
```

Figure 5: Filled in code in slam_resample.m

F) Per Particle Measurement Update: The filled in code in '**slam_crnr_kf.m**' integrates new sensor measurements (detected corners) into the existing map representation (known corners for each particle) by updating the mean and covariance using a Kalman filter. The code snippets and explanations are given below:

- **Mean Update:**
  Updates the x, y, heading, and angle of the known corner using the Kalman gain (**K**) and the innovation vector. This step adjusts the estimated position and orientation of the corner based on the new sensor measurement.
  Code:

```
known_corner.x = known_corner.x + K(1,1) * innovation(1);
known_corner.y = known_corner.y + K(2,2) * innovation(2);
known_corner.heading = known_corner.heading + K(3,3) * innovation(3);
known_corner.angle = known_corner.angle + K(4,4) * innovation(4);
```

- **Covariance Update:**
  Updates the covariance matrix of the known corner using the Kalman gain and the identity matrix. Updates the covariance matrix of the known corner using the Kalman gain and the identity matrix. This update refines the uncertainty based on the new sensor measurement.
  Code:

```
known_corner.covariance = (eye(4) - K) * known_corner.covariance;
```

MATLAB Code

```
% Missing codes start here ...

% Update mean of this corner
known_corner.x = known_corner.x + K(1,1) * innovation(1);
known_corner.y = known_corner.y + K(2,2) * innovation(2);
known_corner.heading = known_corner.heading + K(3,3) * innovation(3);
known_corner.angle = known_corner.angle + K(4,4) * innovation(4);

% Update covariance of this corner
known_corner.covariance = (eye(4) - K) * known_corner.covariance;

% Missing codes end here ...
```

Figure 6: Filled in code in slam_crnr_kf.m

Important Formulae:

The mentioned formulae update the mean and covariance by using the innovation, Kalman gain and Known corner covariance.

$$\mu_{n,t}^{[m]} = \mu_{n,t-1}^{[m]} + K_{n,t}^{[m]}(z_{n,t} - \hat{z}_{n,t})$$
$$\Sigma_{n,t}^{[m]} = (I - K_{n,t}^{[m]})\Sigma_{n,t-1}$$

2. Please explain what problem is line 11-40 in the original 'slam_lidar_split_merge.m' trying to solve? Why does the solution need to use if/else to consider two cases?

The code segment checks whether the line is more vertical or horizontal by comparing the absolute differences in x and y coordinates of the endpoints. The code further calculates the parameters necessary to get the equation of the line. If the line is more vertical '**abs(x2-x1) <= abs(y2-y1)**', it assumes a representation in the form '**y = mx + c'** and assumes representation in the form '**x = my + c'** and assumes if it is more horizontal. Based on the that, it calculates gradient **m** and y-intercept **c** and determines the polar coordinates '**r'** and '**theta'** of the line. The '**if/else**' statement checks vertical or horizontal nature of the line which helps in correctly determining the polar coordinates of the line, representing the position and orientation of the line.

3. Please explain what line 51-97 in 'motion_estimation.m' is trying to do?

The code is performs the feature association step in the algorithm. Feature association is crucial for tracking features, in this case corners, across different frames, aiding in the motion estimation and mapping process in SLAM.

- The code checks if there are corners detected in the current and previous LiDAR frames (corners1 and corners2).
- If corners are detected, it calculates the Mahalanobis distance matrix between pairs of corners.
- The while loop iteratively associates corners by finding the minimum distance in the Mahalanobis distance matrix.
- If the minimum distance is below a certain threshold '**threshold**', it associates the pair of corners, increments '**assoc_count'**, and eliminates the associated row and column from further consideration.
- The loop continues until there are no more valid associations below the threshold.

4. Please explain what 'slam_in_pi.m' function is trying to achieve. Where are the locations this function is called. Why are they necessary?

The '**slam_in_pi.m**' function restricts the parameter passed to it (mostly the heading angle) between -pi and pi. The function is used in lines 157, 159 and 162 in '**motion_estimation.m**'. The purpose of the function is to constrain the angles between a defined interval and maintain a consistent angle representation. It is also used in '**slam_lidar_split_merge.m**' in lines 23 and 37 to constrain polar coordinate theta of the line. It is further used in code '**slam_crnr_kf.m**' in lines 13, 14 and 22, 23 respectively. The function constrains the '**heading**' and '**angle**' between -pi and pi. The reason to constrain angles and headings concerned with features is to ensure consistency in measurement and to make sure that we don't have multiple valid representations for the same orientation.

5. In the last part of 'motion_estimation.m' (before plotting), what do you think is the advantage of constraining the change of x, y and theta?

Constraining x, y and theta helps avoid discontinuities and maintain consistency in reading and processing data. It clips the data to specified high and low values which helps minimizing jitters in the readings and keeps the code consistent. Moreover, it minimizes the sensitivity of the algorithm to sudden abrupt changes or spikes, making sure the motion predictions and estimations are smooth over time. Also, clipping values between a certain range also helps in simplifying computations, improving performance.

Result:

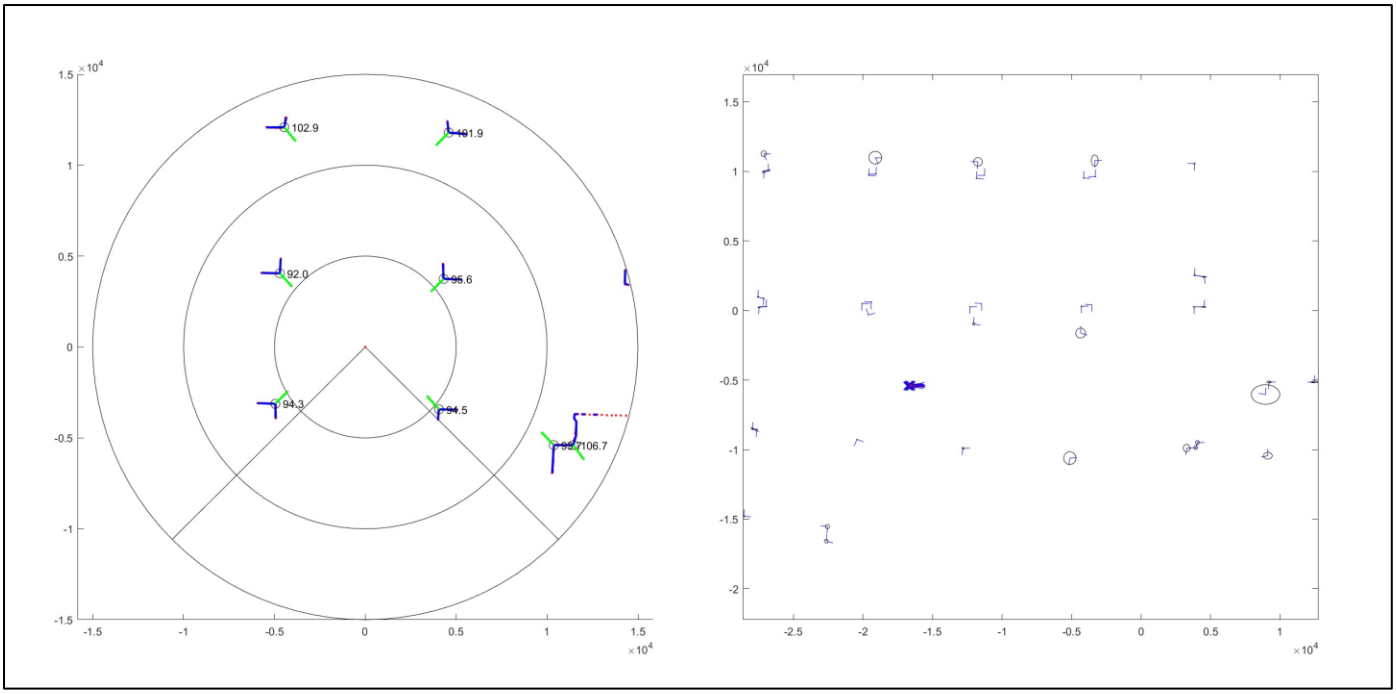

Figure 7: Fast SLAM Result