EE5114 - AUTONOMOUS ROBOT NAVIGATION

NATIONAL UNIVERSITY OF SINGAPORE

DEPARTMENT OF MECHANICAL ENGINEERING

EE5114-CA2

Author:
LIU XIAO(ID: A0304126U)

Date: October 7, 2024

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.

Part-1 slam_lidar_split_merge:

This part of the code calculates the shortest distance from any point (x, y) in the set to the line formed by the first point (x_1, y_1) and last point (x_2, y_2). The distance could be expressed as:

$$d = \left| \frac{(x_2 - x_1)(y_1 - y) - (x_1 - x)(y_2 - y_1)}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \right| \tag{1}$$

```
43      % Missing codes start here ...
44
45      % Find furthest point to this line
46      point_distances = zeros(1, last);
47      winner_value = abs(point_distances(1));
48      winner_index = 1;
49
50      for index = 2 : last - 1
51          x = points(index).x; y = points(index).y;
52          d = abs((x2 - x1) * (y1 - y) - (x1 - x) * (y2 - y1));
53          d = d / (sqrt((x2 - x1) ^ 2 + (y2 - y1) ^ 2));
54          point_distances(index) = d;
55          if d > winner_value
56              winner_value = d;
57              winner_index = index;
58          end
59      end
60
61      % Missing codes end here ...
```

Part-2 slam_lidar_feature_extrn:

This part of the code calculates the angle between two feature lines and determines the orientation of the corner point formed by these lines. These two lines are directed, so the angle is calculated using vector operations where vec_1 and vec_2 are the unit vectors of the two lines.

$$angle = \frac{vec_1 * vec_2}{|vec_1| * |vec_2|} = vec_1 * vec_2 \tag{2}$$

$$heading = atan2((vec_1 + vec_2)[1], (vec_1 + vec_2)[0]) \tag{3}$$

```
27      % Missing codes start here ...
28
29
30      % Calculate angle span of the corner
31      vec_1 = [lines(i).p1.x - lines(i).p2.x,
32               lines(i).p1.y - lines(i).p2.y];
33      vec_2 = [lines(i + 1).p2.x - lines(i + 1).p1.x,
34               lines(i + 1).p2.y - lines(i + 1).p1.y];
35      angle = acos(vec_1' * vec_2) / (norm(vec_1) * norm(vec_2));
36
37      % Only use corner features having angle from 60 to 120 degrees
38      if (angle > deg2rad(60) && angle < deg2rad(120))
39          % Calculate heading direction of the corner
40          heading_vec = vec_1 ./ norm(vec_1) + vec_2 ./ norm(vec_2);
41          heading = atan2(heading_vec(2), heading_vec(1));
42
43
44      % Missing codes end here ...
```

Part-3 motion_estimate:

In this section, I use two methods: one is an algorithm based on Singular Value Decomposition (SVD), as shown in SLAM Part 1 slides, and the other is an algorithm based on the Gauss-Newton nonlinear optimization method, as shown in SLAM Part 2 slides.

Singular Value Decomposition

It's actually based on Kabsch algorithm. From data association, we obtain the corners' position matrix P and Q of the same corner in two different coordinate frame. Given P and Q, the target is to find rotation matrix and translation value. $P = R * Q + t$. To simplify the problem the first step is to calculate the centroid of the feature points for both frames:

$$\begin{aligned} \tilde{p}_i &= p_i - \bar{p} \\ \tilde{q}_i &= q_i - \bar{q} \end{aligned} \tag{4}$$

After that, the optimization objective turns out to be the equation below where R is an orthogonal matrix:

$$\begin{aligned} E &= \sum_i (\tilde{p}_i - R * \tilde{q}_i)^T (\tilde{p}_i - R * \tilde{q}_i) \\ &= \sum_i (\tilde{p}_i^T \tilde{p}_i + \tilde{q}_i^T R^T R \tilde{q}_i - 2 * \tilde{p}_i^T * R * \tilde{q}_i) \\ &= \sum_i (\tilde{p}_i^T \tilde{p}_i + \tilde{q}_i^T \tilde{q}_i - 2 * \tilde{p}_i^T * R * \tilde{q}_i) \end{aligned} \tag{5}$$

From the equation above, the objective turns to maximize $\sum_i \tilde{p}_i^T * R * \tilde{q}_i = trace(R \sum_i \tilde{q}_i \tilde{p}_i^T) = trace(RQP^T)$. Using SVD to decompose the matrix $QP^T = USV^T$, where U and V are also orthogonal matrix.

$$\begin{aligned} trace(RQP^T) &= trace(RUSV^T) \\ &= trace(V^T RUS) \end{aligned} \tag{6}$$

The matrix $V^T RU$ is orthogonal matrix clearly. It can be prove that to maximize the trace:

$$V^T RU = \begin{bmatrix} 1 & 0 \\ 0 & det(VU^T) \end{bmatrix} \tag{7}$$

So in conclusion, the translation and rotation from P to Q could be written as:

$$R = V \begin{bmatrix} 1 & 0 \\ 0 & det(VU^T) \end{bmatrix} U^T = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{8}$$

$$t = P - RQ = (\Delta x, \Delta y)^T \tag{9}$$

```
    avg_P = mean(P, 1)'; avg_Q = mean(Q, 1)';
    new_P = P' - avg_P; new_Q = Q' - avg_Q;
    [U, S, V] = svd(new_P * new_Q');
    d = sign(det(V * U'));
    R = V * [1, 0; 0, d] * U';
    T = R * avg_P - avg_Q;
    theta = -atan2(R(2), R(1));
    translation = [T(1), T(2)];
```

Nonlinear Optimization

This is a typical problem suitable for solving with nonlinear optimization. The goal is to determine the homogeneous transformation matrix. The robot exists in a two-dimensional space, so the size of the homogeneous transformation matrix is 3×3, with three parameters to optimize: $\mathbf{x} = (x, y, \theta)$.

$$T = \begin{bmatrix} cos(\theta) & -sin(\theta) & x \\ sin(\theta) & cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

Firstly, I transform these coordinate matrix P and Q into homogeneous form.

$$\begin{aligned} P &= \begin{bmatrix} x_{p_1} & x_{p_2} & \dots & x_{p_N} \\ y_{p_1} & y_{p_2} & \dots & y_{p_N} \\ 1 & 1 & 1 & 1 \end{bmatrix} \\ Q &= \begin{bmatrix} x_{q_1} & x_{q_2} & \dots & x_{q_N} \\ y_{q_1} & y_{q_2} & \dots & y_{q_N} \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{aligned} \tag{11}$$

So the error function can be written as:

$$E = P - T * Q \tag{12}$$

And the non-linear optimization objective is to find the optimal state $\mathbf{x}^* = (x^*, y^*, \theta^*)$ that satisfies:

$$\mathbf{x}^* = argmin_{\mathbf{x}} trace(E^T * E) \tag{13}$$

And my code implementation uses Matlab internal optimization function "fminunc" and the initial state $\mathbf{x} = zeros(3)$.

```
    global obj_func_y  obj_func_x;

    function f = obj_func(T)
        global obj_func_y  obj_func_x;
        dx = T(1); dy = T(2); theta = T(3);
        transpose_mat = [cos(theta), -sin(theta), dx;
                         sin(theta), cos(theta), dy;
                         0, 0, 1];
        error_mat = obj_func_y - transpose_mat * obj_func_x;
        f = trace(error_mat' * error_mat);
    end
```

```
    T_0 = [0.0, 0.0, 0.0];
    obj_func_y = [P'; ones(1, size(P, 1))];
    obj_func_x = [Q'; ones(1, size(Q, 1))];
    opts = optimoptions('fminunc', 'Algorithm', 'quasi-newton', 'Display', 'iter');
    [T_opt, fval] = fminunc(@obj_func, T_0, opts);
    theta = T_opt(3); translation = T_opt(1:2);
```

Part-4 slam_resample:

In the resampling function, the cumulative distribution of weights is first calculated. Then, in a loop, a random number is generated from a uniform distribution in the range of 0 to 1. The smallest weight that is greater than or equal to this random number is found, and the corresponding particle is added to the new particle set.

```
    function [particles] = slam_resample(particles, init_weight)

        particles_count = size(particles, 2);
        new_particles = particles(1);

        weights = zeros(particles_count, 1); total_weights = 0;
        for i = 1:particles_count
            weights(i) = particles(i).weight;
            total_weights = total_weights + weights(i);
```

```
        end
        cumulative_weights = cumsum(weights) / total_weights;

        for i = 1:particles_count
            % Missing codes start here

            % Resamples particles based on their weights
            random_num = rand(1);
            index = find(cumulative_weights >= random_num, 1);
            new_particles(i) = particles(index);

            % Afterwards, each new partical should be given the same init_weight
            new_particles(i).weight = init_weight;

            % Missing codes end here
        end
        particles = new_particles;
    end
```

Part-5 slam_crnr_kf:
In the Kalman filter section, the measurement equation can be expressed as the equation below where $\mathbf{w}$ is the measurement noise, $\mathbf{x}$ is the know corner and $\mathbf{y}$ is the newly detected corner.

$$\mathbf{y} = \mathbf{x} + \mathbf{w} \tag{14}$$

So the innovation matrix S and Karman Filter gain W could be expressed as the equation below where P and Q are covariance matrix of known corners and detected corners respectively.

$$\begin{aligned} S &= P + Q \\ W &= P * S^{-1} \end{aligned} \tag{15}$$

Lastly, in Karman Filter update state, updating equation is as below:

$$\begin{aligned} \mathbf{x} &= \mathbf{x} + W * (\mathbf{y} - \mathbf{x}) \\ P &= P - W * S * W^T \end{aligned} \tag{16}$$

```
    % Missing codes start here ...

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

    % Update covariance of this corner
    known_corner.covariance = known_corner.covariance - K * Q * K';

    % Missing codes end here ...
```

Part-6 slam:
This part mainly uses the dx, dy, θ values computed from the motion estimation. The noise in each motion parameters is calculated using the following formula:

$$\begin{aligned} d\theta_{noised} &= d\theta * (1 + theta\_noise\_prop * randn(1)) + theta\_noise\_add * randn(1) \\ dx_{noised} &= dx * (1 + trans\_noise\_prop * randn(1)) + trans\_noise\_add * randn(1) \\ dy_{noised} &= dy * (1 + trans\_noise\_prop * randn(1)) + trans\_noise\_add * randn(1) \end{aligned} \tag{17}$$

So the new pose of the particle could be expressed as:

$$\begin{aligned} x &= x * cos(d\theta_{noised}) - y * sin(d\theta_{noised}) + dx_{noised} \\ y &= x * sin(d\theta_{noised}) + y * cos(d\theta_{noised}) + dy_{noised} \\ \theta &= slam\_in\_pi(\theta + d\theta_{noised}) \end{aligned} \tag{18}$$

```
74      % Missing code start here ...
75
76      x = particles(j).x; y = particles(j).y; theta = particles(j).theta;
77      dx = motion_estimate(i).x;
78      dy = motion_estimate(i).y;
79      d_theta = motion_estimate(i).theta;
80
81      % Apply rotation to all particles with (additive and proportional) noises
82      d_theta_with_noise = d_theta + randn(1) * theta_noise_proportion * d_theta +
83                           randn(1) * theta_noise_add;
84
85      % Apply translation to all particles with (additive and proportional) noises
86      dx_with_noise = dx + randn(1) * translation_noise_proportion * dx +
87                      randn(1) * translation_noise_add;
88      dy_with_noise = dy + randn(1) * translation_noise_proportion * dy +
89                      randn(1) * translation_noise_add;
90
91      particles(j).x = x + dx_with_noise * cos(theta) - dy_with_noise * sin(theta);
92      particles(j).y = y + dx_with_noise * sin(theta) + dy_with_noise * cos(theta);
93      particles(j).theta = slam_in_pi(theta + d_theta_with_noise);
94
95      % Missing code end here ...
```

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?

This part of the code first calculates the equation of the line formed by the first and last points in the set. It then computes the perpendicular distance from the origin to the line, as well as the angle between the perpendicular line and the positive x-axis of robot local frame.

The solution accounts for two cases to avoid situations where the denominator of the line's slope is very small. In such cases, the slope might result in NaN(Not a Number) or an extremely large value approaching infinity. Handling these cases ensures the calculations remain stable and avoid errors.

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

This part of the code performs data association, specifically the matching of landmarks. corner1 is the known set of corner points, and corner2 is the set of corner points detected at the current time step.

First, the algorithm iterates through both sets of points, calculating the Mahalanobis distance between each pair of points to form a Mahalanobis distance matrix. Time complexity is $O(kj)$ where k and j are the size of two corner sets respectively.

Then, a loop begins to match corner points. Each time the loop finds the smallest Mahalanobis distance in distance matrix. If the distance is below threshold, the corresponding two corner points are considered a match, and the matches are saved in sets P and Q, with the corresponding row and column values in the matrix set to $threshold + 1$(No longer match). If the distance is above the threshold, the matching loop terminates.

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 function slam_in_pi is used to constrain angles within the range of $-\pi$ to π, addressing the periodicity and continuity of angles to maintain consistency in angle representation. In CA2, the func-

tion is called in motion_estimation function, slam_crnr_jcbb_assoc function, slam_crnr_kf function, slam_crnr_loc2glo function and slam_lidar_split_merge function.
This function is necessary because angles are periodic, angles like θ and $\theta+2\pi$ represent the same direction. Without such normalization, it would be impossible to correctly compare the differences between directions. For example, when comparing $-\pi$ and π, although we know these two angles represent the same direction, without angle normalization, their difference would be 2π, which would be incorrectly considered unequal by the program.

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?

The advantage of constraining the change of x, y and theta is that we can approximates the corner point coordinates in different frames as being in the same frame when calculating the Mahalanobis distance. As shown in Matlab code of this function, with the assumption that the robot's pose change between two adjacent time steps is small, we can directly approximate variable a, b and c without any rotation matrix which greatly simplify the problem.

```
63      for j = 1:detected_corners_count
64          for k = 1:known_corners_count
65                  % Components of x-mu
66                  a = corners2(j).x-corners1(k).x;
67                  b = corners2(j).y-corners1(k).y;
68                  c = slam_in_pi(corners2(j).angle-corners1(k).angle);
69
70                  mahalanobis_dist = sqrt([a b c] * (covariance^-1) * [a; b; c]);
71                  if (mahalanobis_dist < threshold)
72                          mahalanobis_matrix(j,k) = mahalanobis_dist;
73                  end
74          end
75      end
```