Principles of Robot Autonomy I: Midterm 2024

Wei Lin Pai

10/30/24

Problem 1

(i)

$$\dot{x} = \begin{bmatrix} vcos(\theta) \\ vsin(\theta) \\ \frac{v}{l}tan(\phi) \\ \omega \end{bmatrix} \tag{1}$$

To get Jacobian, we need to differentiate it with respect to each state variable

$$J_s = \begin{bmatrix} \frac{\partial \dot{q}_x}{\partial q_x} = 0, & \frac{\partial \dot{q}_x}{\partial q_y} = 0, & \frac{\partial \dot{q}_x}{\partial \theta} = -v\sin(\theta), & \frac{\partial \dot{q}_x}{\partial \phi} = 0, \\ \frac{\partial \dot{q}_y}{\partial q_x} = 0, & \frac{\partial \dot{q}_y}{\partial q_y} = 0, & \frac{\partial \dot{q}_y}{\partial \theta} = v\cos(\theta), & \frac{\partial \dot{q}_y}{\partial \phi} = 0 \\ \frac{\partial \dot{\theta}}{\partial q_x} = 0, & \frac{\partial \dot{\theta}}{\partial q_y} = 0, & \frac{\partial \dot{\theta}}{\partial \theta} = 0, & \frac{\partial \dot{\theta}}{\partial \phi} = \frac{v}{l}\sec^2(\phi) \\ \frac{\partial \dot{\phi}}{\partial q_x} = 0, & \frac{\partial \dot{\phi}}{\partial q_y} = 0, & \frac{\partial \dot{\phi}}{\partial \theta} = 0, & \frac{\partial \dot{\phi}}{\partial \phi} = 0 \end{bmatrix}$$

Thus, the state Jacobian J_s is:

$$J_s = \begin{bmatrix} 0 & 0 & -v\sin(\theta) & 0 \\ 0 & 0 & v\cos(\theta) & 0 \\ 0 & 0 & 0 & \frac{v}{l}\sec^2(\phi) \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(ii) Control Jacobian will be

$$J_c = \begin{bmatrix} \frac{\partial \dot{q}_x}{\partial v} = \cos(\theta), & \frac{\partial \dot{q}_x}{\partial \omega} = 0 \\ \frac{\partial \dot{q}_y}{\partial v} = \sin(\theta), & \frac{\partial \dot{q}_y}{\partial \omega} = 0 \\ \frac{\partial \dot{\theta}}{\partial v} = \frac{1}{l}\tan(\phi), & \frac{\partial \dot{\theta}}{\partial \omega} = 0 \\ \frac{\partial \dot{\phi}}{\partial v} = 0, & \frac{\partial \dot{\phi}}{\partial \omega} = 1 \end{bmatrix}$$

Thus, the control Jacobian J_c is:

$$J_c = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ \frac{1}{l}\tan(\phi) & 0 \\ 0 & 1 \end{bmatrix}$$

(iii)

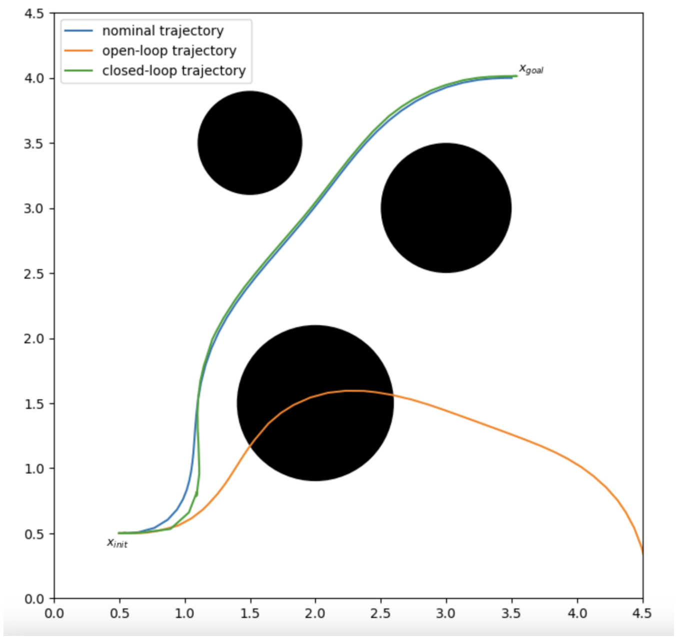


Figure 1: trajectory plot

Problem 2

(i) Transformation Matrix:

$$\begin{bmatrix} \cos(5*x) & -\sin(5*x) & x \\ \sin(5*x) & \cos(5*x) & x \\ 0 & 0 & 1 \end{bmatrix}$$

(ii) x = 0 in my case and

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

No it did not reach the target after the transformation, because the point after transformation is [0.5,0.5], distance between target is more than 1.

(iii)

$$\begin{bmatrix} 400 \\ 340 \\ 1 \end{bmatrix} = \begin{bmatrix} 100 & 0 & 640 \\ 0 & 100 & 480 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

solve x = -2.4, y = -1.4
Since the center of the image plane is at (640,480) away from the camera coordinate, this means the target is located to the negative x and negative y part of the optical center of the camera's local frame.

(iv) (c)

(v) (b)

Problem 3

(i) (A)

(ii) (B)

(iii) (D)

(iv) (C)

Problem 4

(i) Shortcutting is a post-processing that removes unnecessary turns and redundant nodes by taking ”shortcuts” between points along the path. However, this shortcutting process only heuristically improves path quality; it doesn’t guarantee an optimal path.
RRT* incrementally improves the path during tree expansion. It converges to an optimal path, meaning it can find the shortest or least-cost path.

(ii) If infinite samples, RRT* guaranteed to find the shortest optimal path. This is because RRT* is asymptotically optimal, which means in infinite loops it will converge to optimal solution due to its path optimization in each iteration of connecting new nodes.

(iii) I would expect RRT* still generate better solution because RRT* performs local label correction through rewiring and it always guarantee shorter path solution.

(iv) Plot image

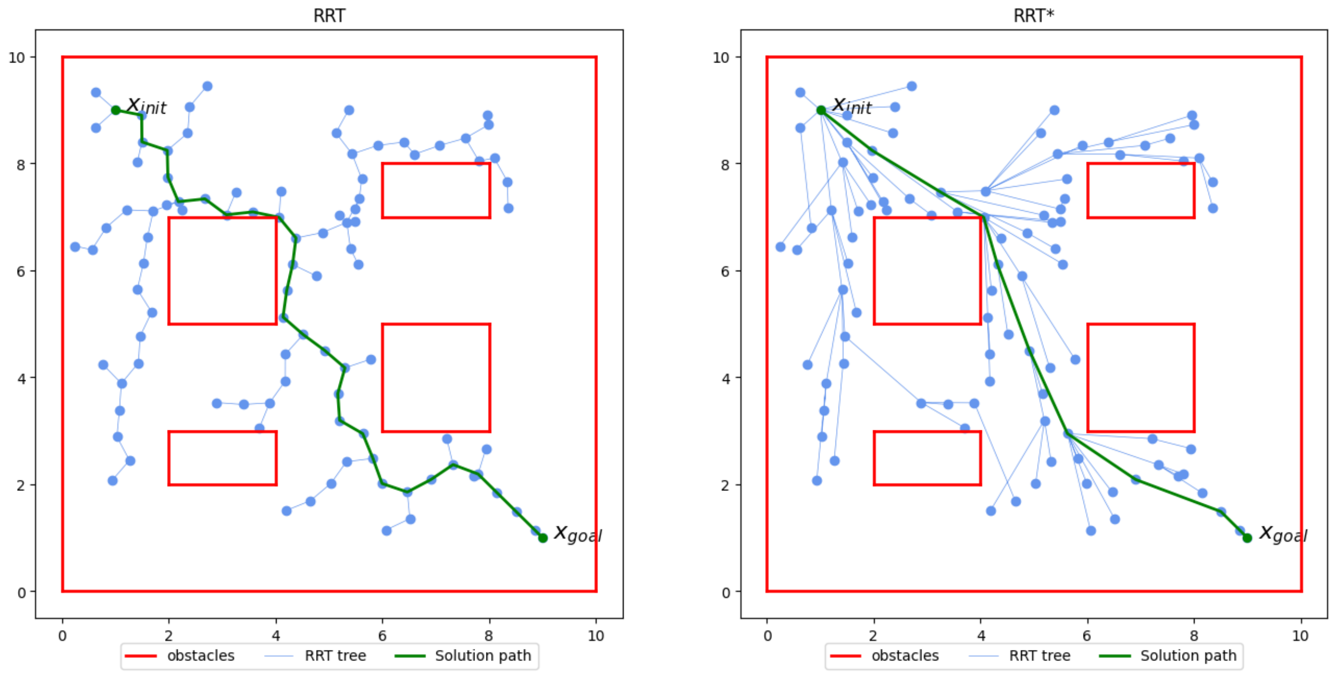
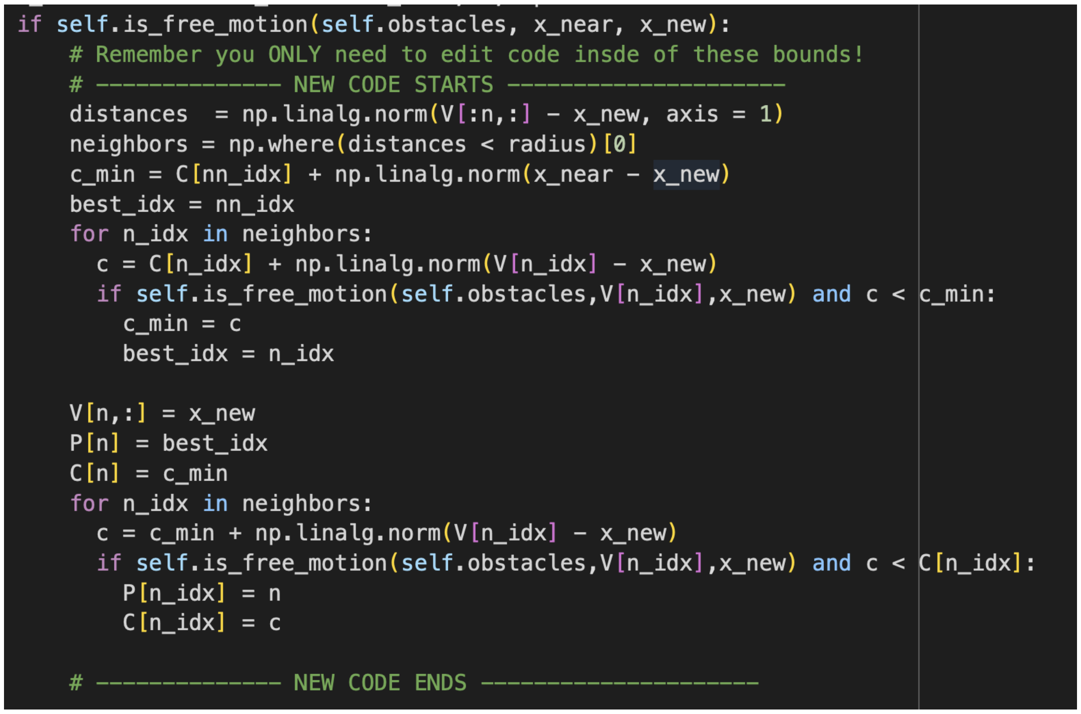


Figure 2: RRT*



```
if self.is_free_motion(self.obstacles, x_near, x_new):
    # Remember you ONLY need to edit code insde of these bounds!
    # -------------- NEW CODE STARTS ---------------------
    distances  = np.linalg.norm(V[:n,:] - x_new, axis = 1)
    neighbors = np.where(distances < radius)[0]
    c_min = C[nn_idx] + np.linalg.norm(x_near - x_new)
    best_idx = nn_idx
    for n_idx in neighbors:
      c = C[n_idx] + np.linalg.norm(V[n_idx] - x_new)
      if self.is_free_motion(self.obstacles,V[n_idx],x_new) and c < c_min:
        c_min = c
        best_idx = n_idx

    V[n,:] = x_new
    P[n] = best_idx
    C[n] = c_min
    for n_idx in neighbors:
      c = c_min + np.linalg.norm(V[n_idx] - x_new)
      if self.is_free_motion(self.obstacles,V[n_idx],x_new) and c < C[n_idx]:
        P[n_idx] = n
        C[n_idx] = c

    # -------------- NEW CODE ENDS ---------------------
```

Figure 3: RRT* Code

Problem 5 (Extra Credit)

(i) If we apply one filter, the final dimension is 3 X 3, based on the assumption that stride is 1. If we apply two convolution layer successively, the final dimension will be 1 X 1. We can modify the matrix A using padding to ensure the resulting matrix is of the same size as A. Is this scenario, the padding will be 1.

(ii) Apply 0 padding to A,

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 9 & 0 & 0 \\ 0 & 0 & 9 & 0 & 9 & 0 & 0 \\ 0 & 0 & 9 & 0 & 9 & 0 & 0 \\ 0 & 0 & 9 & 0 & 9 & 0 & 0 \\ 0 & 0 & 9 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Then apply filter S, the resulting matrix will be

$$A = \begin{bmatrix} 2 & 2 & 4 & 2 & 2 \\ 3 & 3 & 6 & 3 & 3 \\ 3 & 3 & 6 & 3 & 3 \\ 3 & 3 & 6 & 3 & 3 \\ 2 & 2 & 4 & 2 & 2 \end{bmatrix}$$

Add 0 padding again,

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 4 & 2 & 2 & 0 \\ 0 & 3 & 3 & 6 & 3 & 3 & 0 \\ 0 & 3 & 3 & 6 & 3 & 3 & 0 \\ 0 & 3 & 3 & 6 & 3 & 3 & 0 \\ 0 & 2 & 2 & 4 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Then apply filter D

$$A = \begin{bmatrix} 7 & 7 & 0 & -7 & -7 \\ 11 & 11 & 0 & -11 & -11 \\ 12 & 12 & 0 & -12 & -12 \\ 11 & 11 & 0 & -11 & -11 \\ 7 & 7 & 0 & -7 & -7 \end{bmatrix}$$

(iii) The final matrix will not be same because convolutional layer is doing one-by-one multiplication over image pixels and it is not the same as what matrix multiplication is doing (row * cols). Hence, multiplying two conv layer (in matrix multiplication fashion) then apply to the image will give different result as applying two conv layer successively.