RBE550: Motion Planning

Project 4 Rigid Body Motion Planning

Student 1: UdayGirish Maradana
Student 2: Pradnya Sushil Shinde

- Theoretical Questions

1. Questions about Asymptotically Optimal Planners

   (a) The core idea behind RRT* is to find an optimal path as the number of samples increases to infinity, at the same time, RRT provides a zig-zag path and with an increase in the number of samples, there is no considerable decrease in the number of steps taken to reach the goal. Therefore, RRT* succeeds in finding the shortest path while RRT does not succeed.
   The following methods are changed and highlight the difference between RRT and RRT*:

      i. Method of adding new edges to the tree
      RRT* estimates a set of nodes in the neighborhood of the new node(q_{new}) and selects the best neighbor of the new node. While making a selection of the best neighbor, the total cost of reaching the new node is evaluated ($q_{near} \rightarrow q_{new}$), and the neighbor node that yields the lowest cost is then selected as the best neighbor. Then the new node, (q_{new}) and edge joining (q_new) and nearest neighbor node (q_{near}) are added to the tree.

      ii. Method of rewiring to find the optimal path
      The newly added node q_{new} is again checked for rewiring. The total cost is now calculated from ($q_{new} \rightarrow q_{near}$) and is compared with the current cost to reach q_{near}. If it is lower than the current cost, q_{new} becomes the new parent of q_{near}.

   (b)
   - Informed RRT* adopts both optimality and completeness from RRT* but improves upon the time taken to reach optimality i.e. convergence rate.
   - This is because of a different sampling strategy used by Informed RRT*, where the sampling happens based on a prolate ellipsoid formed based on the start and goal paths. This sampling is adaptive in that until the algorithm finds a goal point it acts as RRT* but once it finds a goal state, it adapts to the ellipsoid sampling.
   - Overall because of these improvements, informed RRT* is optimized on the cost to reach the goal state.
   - RRT* can be much better in case of denser gaps or where there are highly dense obstacles between the start and goal endpoint. Here Informed RRT* struggles as the sampling happens like RRT* only till it finds a goal point which takes a long time, further it is hard to fit an ellipsoid strategy in narrow paths when the dimension of the robot and all needs to be considered. In this case, we might not see any advantage over RRT* instead, it can happen that Informed RRT* might take a longer time because of the different sampling strategy.
   - When the goal itself is a moving object even though we have a track of it, as Informed RRT* aims at finding an optimal path based on the sampling but in this case the initial sampling after the first goal found changes, as in a dynamic goal case the goal is not constant. So, here Informed RRT* can go in a forever loop which is not useful and that is why in most of the dynamic real-life use cases we see RRT*.

2. Indoor Vaccum Robot Problem - Roomba

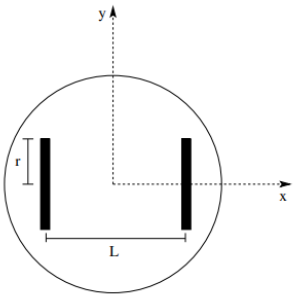


Figure 1: The Indoor Vacuum Robot

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{r}{2}(u_l + u_r)\cos(\theta) \\ \frac{r}{2}(u_l + u_r)\sin(\theta) \\ \frac{r}{L}(u_r - u_l) \end{pmatrix}$$

(a) The C-Space of the robot is: $\mathbb{R}^2 \times \mathbb{S}$
State Space: (x, y, θ)
Control Space: (u_r, u_l)

(b) The current state of the robot is given by the following:

$$\begin{pmatrix} x_0 \\ y_0 \\ \theta_0 \end{pmatrix}$$

The change in state using Euler approximation is given by:

$$x(\Delta t) \approx \begin{pmatrix} x_0 \\ y_0 \\ \theta_0 \end{pmatrix} + \Delta t \begin{pmatrix} \frac{r}{2}(u_l + u_r)\cos\theta_0 \\ \frac{r}{2}(u_l + u_r)\sin\theta_0 \\ \frac{r}{L}(u_r - u_l) \end{pmatrix} \tag{1}$$

Given values: $\Delta t = 1$ s, $u_r = 0.5$ and $u_l = 1$.

$$\begin{aligned} x(\Delta t) &\approx \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \Delta t \begin{pmatrix} \frac{r}{2}(1 + 0.5)\cos 0 \\ \frac{r}{2}(1 + 0.5)\sin 0 \\ \frac{r}{L}(0.5 - 1) \end{pmatrix} \\ &\approx \begin{pmatrix} \frac{1.5r}{2} \\ \frac{1.5r}{2} \times 0 \\ \frac{-0.5r}{L} \end{pmatrix} \\ &\approx \begin{pmatrix} 0.75r \\ 0 \\ \frac{-0.5r}{L} \end{pmatrix} \end{aligned}$$

L = distance between the wheels
r = radius of the wheels

3. PseudoCode: 2D Rectangular Robot

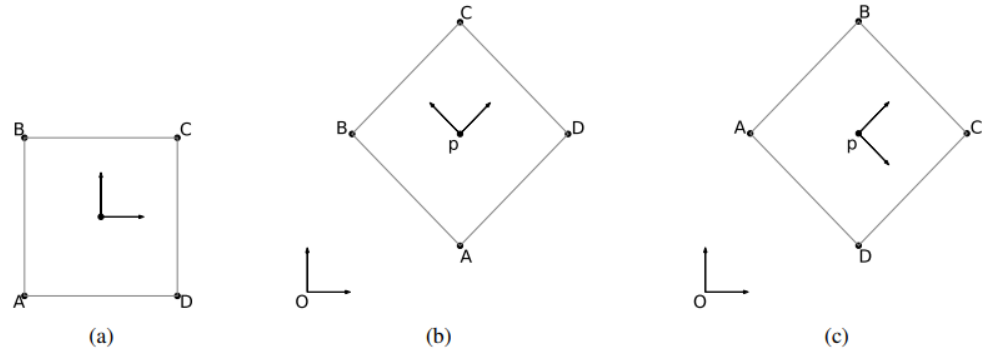


Figure 2: (a) The geometry of the square robot in its local coordinate frame. The points A-D are specified with respect to the local origin at the center of the square. The choice of local origin is arbitrary, but must remain constant. (b) The robot at $p = (x, y)$ and $\pi/4$ rotation in the workspace. (c) The robot at $p = (x, y)$ and $-\pi/4$ rotation in the workspace. Although (b) and (c) look similar, these are two geometrically distinct configurations of the robot.

To check collision between a rectangular robot and AABB obstacles, we need to do four things:

(a) Get the robot corner coordinates (endpoints).
(b) Get edges of the robot.
(c) Check if the corners of the robot lie within the region of the map
(d) Check for every obstacle, if the robot corners collide within the boundary of obstacles

Algorithm 1 AABB Collision Checking

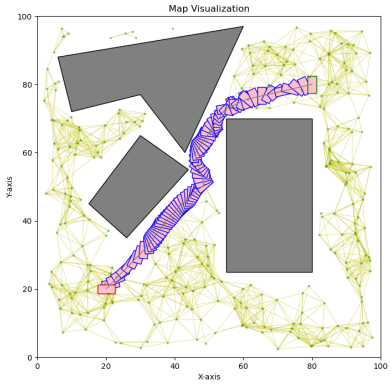
function CHECKAABBCOLLISION(robot, obstacles, map_corners)
▷ robot is a list of parameters: $[x, y, \theta]$
▷ Obstacles is a list of tuples, with each tuple containing parameters, the bottom-left corner coordinates obstacle, (X, Y) and width and height of the obstacle
▷ map_corners is a list of corners of the map
robot_corners = forward_kinematics(robot)
▷ We will get a list of 4 corner points, $[c_1, c_2, c_3, c_4]$ as output.
robot_edges = getEdges(robot_corners)
▷ Output will polygon edges in the form of a list of tuples: $[(c_1, c_2), (c_2, c_3), (c_3, c_4), (c_4, c_1)]$
collision = False
for Obstacle in Obstacles do
for j in range(4) do
if robot_corners[j][0] ≥ Obstacle.X + Obstacle.Width AND robot_corners[j][1] ≥ Obstacle.Y + Obstacle.height then collision = True

obstacle_edges = getEdges(obstacle_edges)
▷ Text Obstacle edges
for obs_edge in obstacle_edges do
for rob_edge in robot_edges do
if line_intersection(obs_edge, rob_edge) then collision = True
return collision

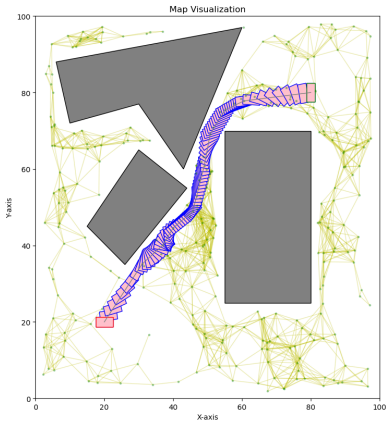
then

- Programming Component

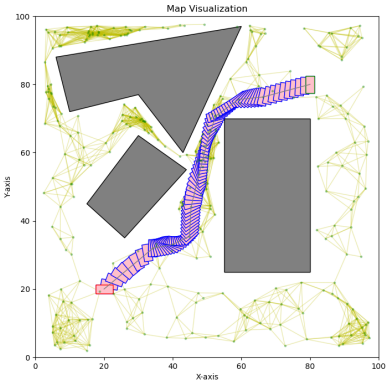
1. **Omnidirectional 2D Robot**
   PRM Planner



(a) PRM: Uniform Sampling
Path Length: 95.15

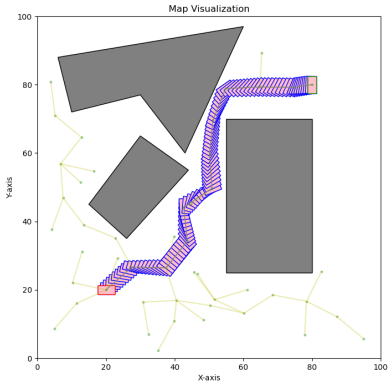


(b) PRM: Gaussian Sampling
Path Length: 94.09

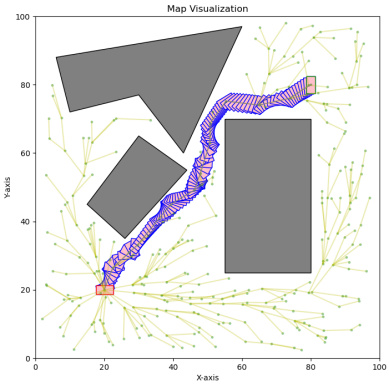


(c) PRM: Bridge Sampling
Path Length: 96.32

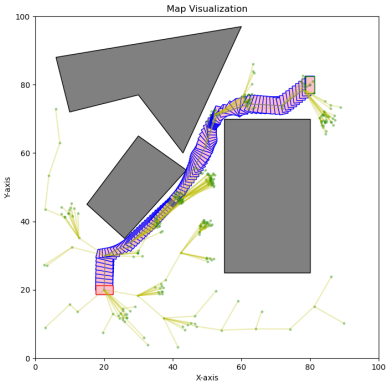
RRT and its variants



(a) RRT
Path Length: 105.27

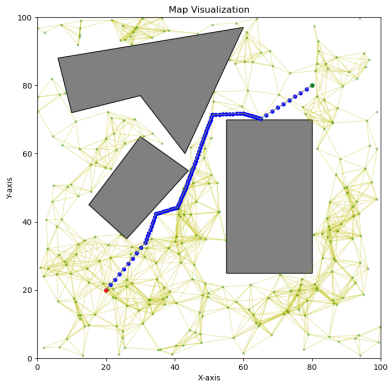


(b) RRT*
Path Length: 96.76

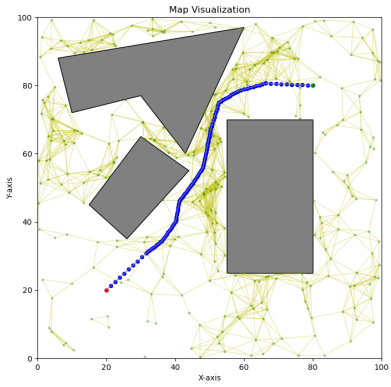


(c) Informed RRT*
Path Length: 94.52

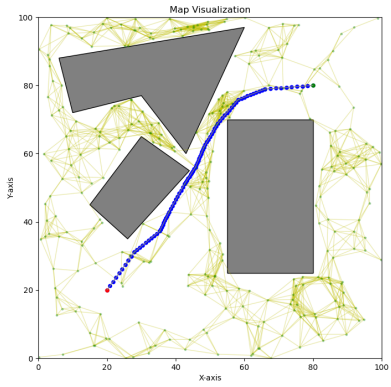
2. **Point Robot**
   PRM Planner



(a) PRM: Uniform Sampling
Path Length: 94.71

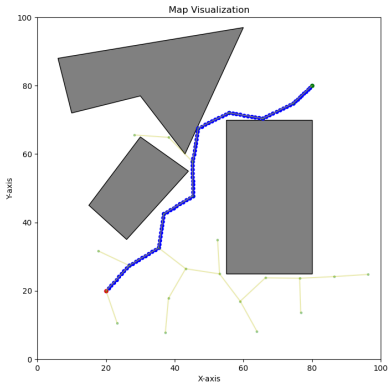


(b) PRM: Gaussian Sampling
Path Length: 94.94

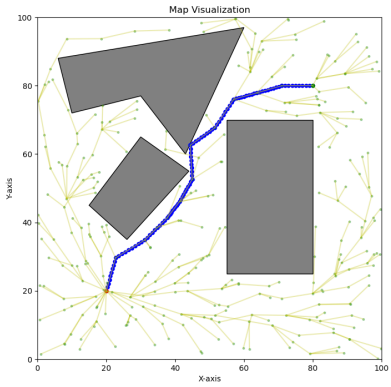


(c) PRM: Bridge Sampling
Path Length: 90.77

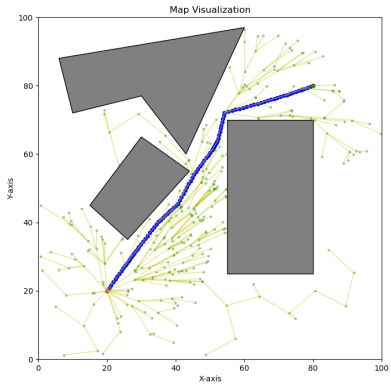
RRT and its variants



(a) RRT
Path Length: 97.45

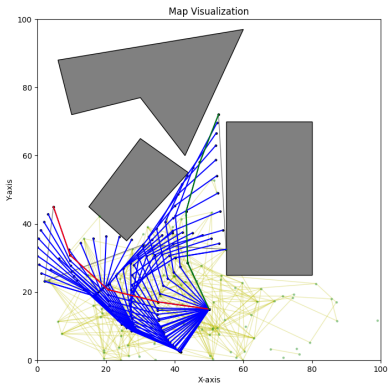


(b) RRT∗
Path Length: 94/87

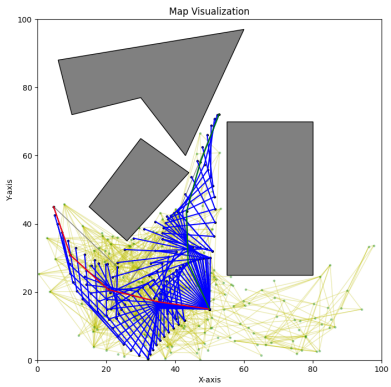


(c) Informed RRT∗
Path Length: 90.11

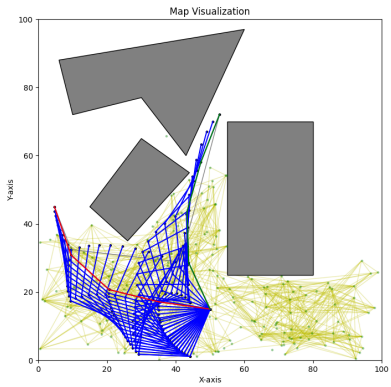
3. **Kinematic Chain**
   PRM Planner



(a) PRM: Uniform Sampling
Path Length: 7.32

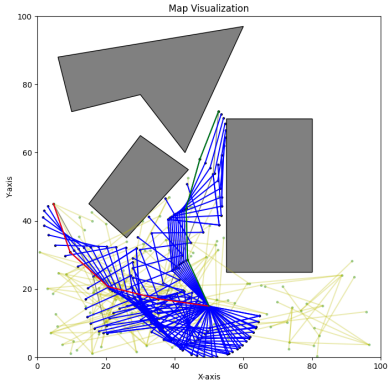


(b) PRM - Gaussian Sampling
Path Length: 11.02

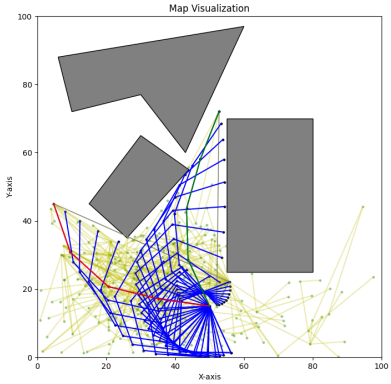


(c) PRM - Bridge Sampling
Path Length: 9.24

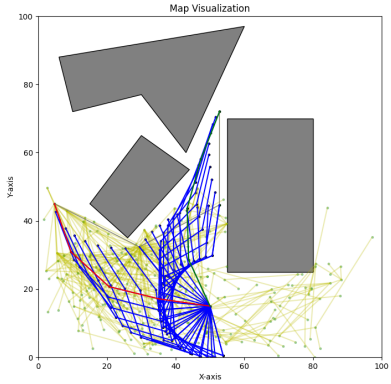
RRT and its variants



(a) RRT
Path Length: 15.01



(b) RRT∗
Path Length: 7.41



(c) Informed RRT∗
Path Length: 7.20

- **Please check videos and outputs in the Output Videos and Output Images folder in the zip**
- **Added main_arg python script for running with argparser**