Faculty of Engineering and Applied Science

ENGR 5945G: Mobile Robotic Systems

Instructor: Scott Nokleby, PhD, PEng

Project Report

Title: Path Planning and Path Tracking of a Mobile Robot

First Name: Anirban

Last Name: Barua

[redacted]

PREFACE

List of Figures

1. Introduction

Path planning and path tracking are the two fundamental concepts in mobile robotics, especially in autonomous navigation. Robots often function in situations where human involvement is infeasible or unworkable, and this is where these two techniques come into play. Through path planning and tracking, robots can autonomously navigate these environments, completing tasks efficiently and safely.

Path planning entails a sequence of steps or waypoints for a robot to move from its current position to its goal position by avoiding obstacles and following specific constraints. So, first, a robot gathers data about its environment using sensors such as cameras, LIDARs, etc., to map out the obstacles and understand the layout. Then, using the collected data, the robot represents its environment, often in the form of a grid map, occupancy map, or a geometric model. After that, a feasible path will be generated from the robot's current position to the goal position using path planning algorithms such as A*, RRT, Bi-RRT, RRT*, etc., and once the path is generated, it will go through optimization to improve the efficiency or address specific constraints such as minimizing energy consumption or avoiding obstacles.

Path tracking involves following the generated path during the path-planning process as closely as possible by controlling the robot's motions. In the path-tracking phase, the robot continuously estimates its position and orientation using onboard sensors such as GPS, odometry, or visual odometry. It will also implement control algorithms to adjust its motion based on the difference between its trajectory and the desired path. Moreover, the robot will continuously update control inputs based on sensor feedback to ensure accurate tracking, even in disturbances or uncertainties.

For my project, I have simulated a mobile robot's path planning and path tracking in a virtual environment with obstacles. I am using RRT (Rapidly Exploring Random Tree) and Bi-RRT (Bidirectional Rapidly Exploring Random Tree) for the path planning algorithm. In the case of path tracking, I am using the "Follow the Carrot" algorithm to follow the generated path effectively.

2. Simulation Flow chart

The flow chart of the simulation is given below:

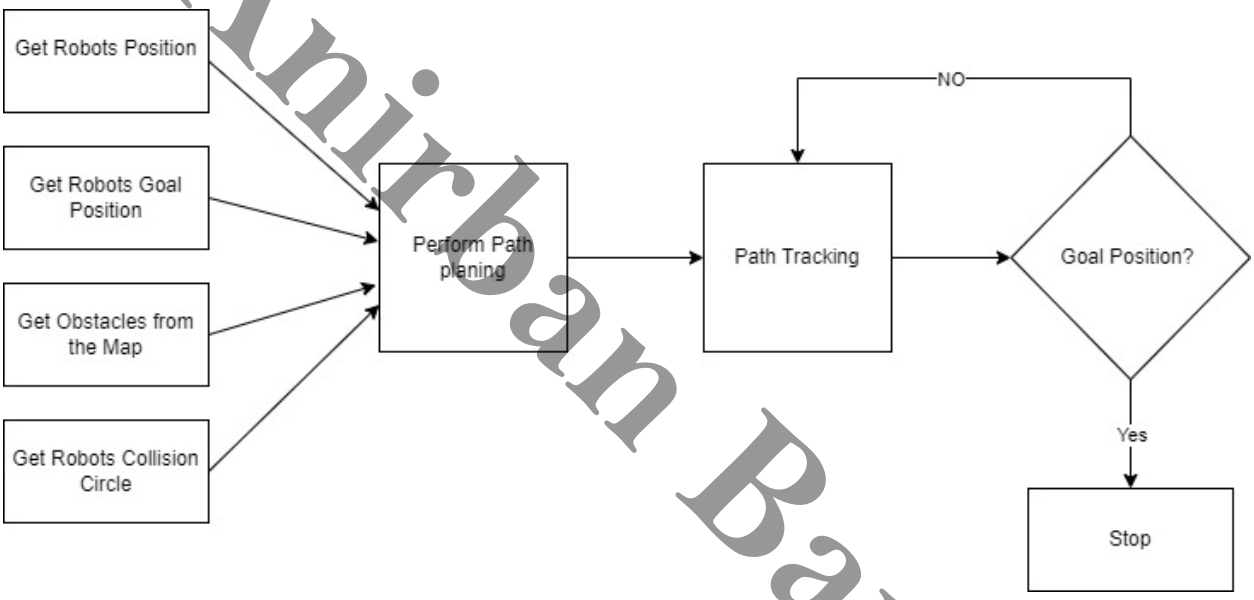

Fig 1: Simulation flow chart of the system.

According to the flow chart, first, the path planner will take the robot's current position, goal position, obstacles from the simulation map, and collision circle and use an algorithm (RRT or Bi-RRT) to generate a collision-free path. After the path is generated, the robot will start following it, and during that time, it will constantly check if the desired or goal position is reached or not. Once the goal position is reached, the robot will stop its motion.

3. Path Planning

For the simulation, during the path planning stage, two path planning algorithms such as Bi-RRT and RRT are used to visualize their performance under same obstacle course and to find out the effective one.

3.1. RRT (Rapidly Exploring Random Tree)

RRT path planning algorithm generates a path by avoiding obstacles and expanding the tree of nodes in a state space or a map from start to goal position.

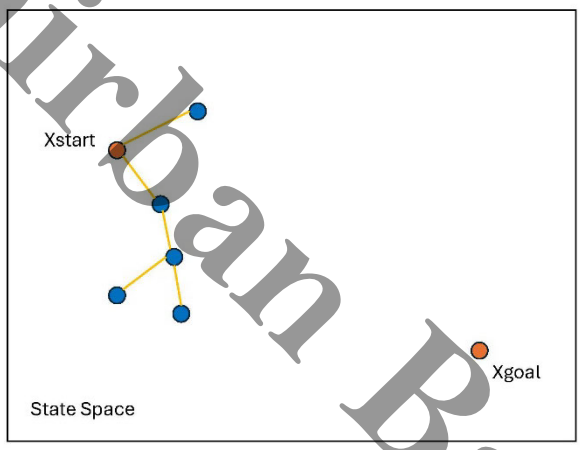

Fig 2: Initial tree of RRT.

Let's assume a state space where the path will be generated where, Xstart and Xgoal are the start node and goal node respectively. According to the Fig 2 we can see a initial tree that has grown from Xstart. The main objective is to generate branches of random nodes to get close to Xgoal. This step is called expansion step. The expansion step is shown below.

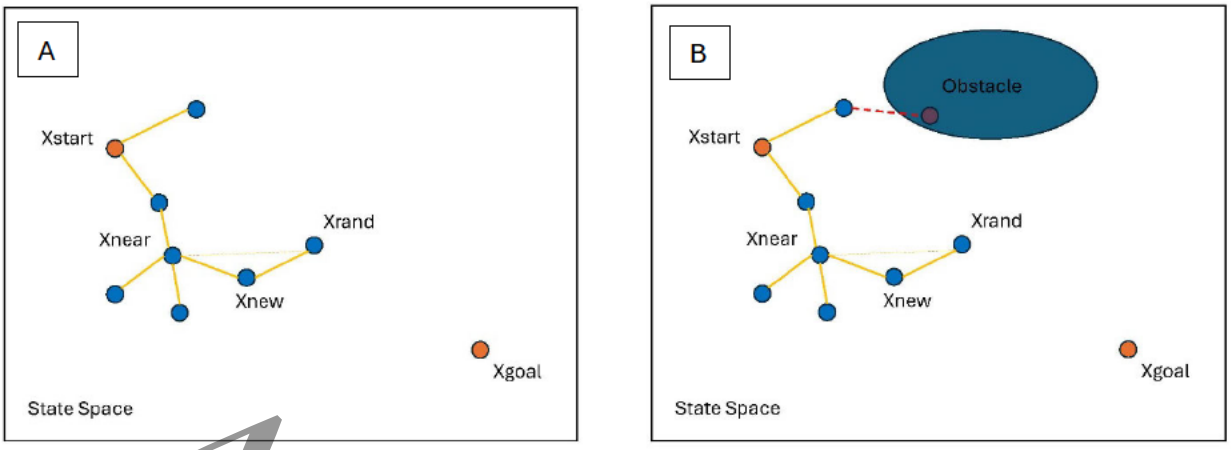

Fig 3: RRT tree expansion.

Expansion Step:

First randomly sample a node Xrand from the state space and then find the nearest node in the existing tree to the sampled node. This random sampled node can be anywhere in the state space. So, the algorithm creates another node Xnew between Xrand and Xnear to expand the tree (Fig 3(A)). If the random node is in the obstacle, its not created (Fig 3(B)).

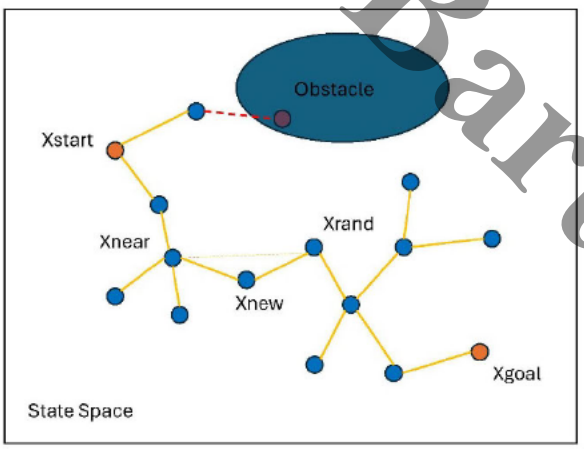

Fig 4: Tree reaches the goal node.

The algorithm repeats the expansion step until the tree reaching the goal node or a maximum number of iterations is reached (Fig 4).

When the tree reaches the goal node, the algorithm constructs a path from the initial node to the goal node by backtracking through the tree (Fig 5).

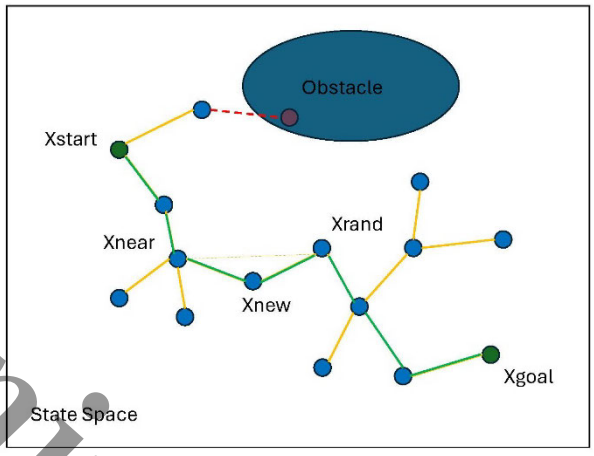

Fig 5: Constructed path shown in green.

3.2. Bi-RRT (Bidirectional Rapidly Exploring Random Tree)

In Bi-RRT the Expansion step is same as RRT but the fundamental difference is that the tree initializes from both directions (Xstart and Xgoal). The algorithm iteratively grows both trees towards random nodes in the space.

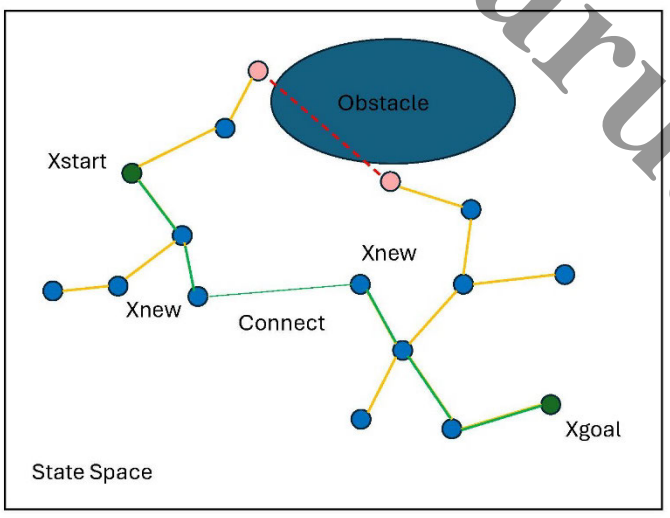

Fig 6: Bi-RRT path generation.

The algorithm would connect the trees if a new node added (Xnew) to one tree is within a certain distance threshold of a node of the other tree. The trees won't connect, or the connection branch won't be created if there is an obstacle between the nodes which we can observe from Fig 6.

Once the trees meet, the algorithm constructs a path by backtracking from the start node to the common node, and then from the common node to the goal node which is indicated in green.

3.3 Bi-RRT VS RRT

To visualize Bi-RRT and RRT algorithm, I have written animation codes for both algorithms in python. These two files named "BITRRT.py" and "RRT.py" are in the submitted zip file with this report.

The figure shown below is the output of "BITRRT.py" with iteration number set to 100.

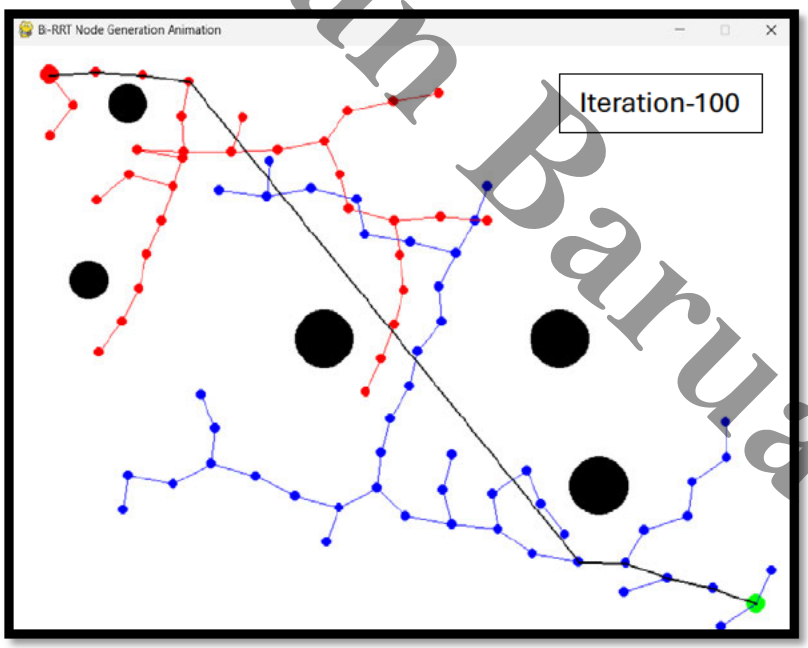

Fig 7: Visualization of Bi-RRT.

The start node and goal node are shown in red and green respectively and the black circles indicate obstacles. When the 100 iterations are complete, the algorithm will look for collision free

connection between two trees within the certain threshold value. Once the connection is made, it will construct a path indicated in black.

The figure shown below is the output of "RRT.py" with iteration number set to 100 and 300.

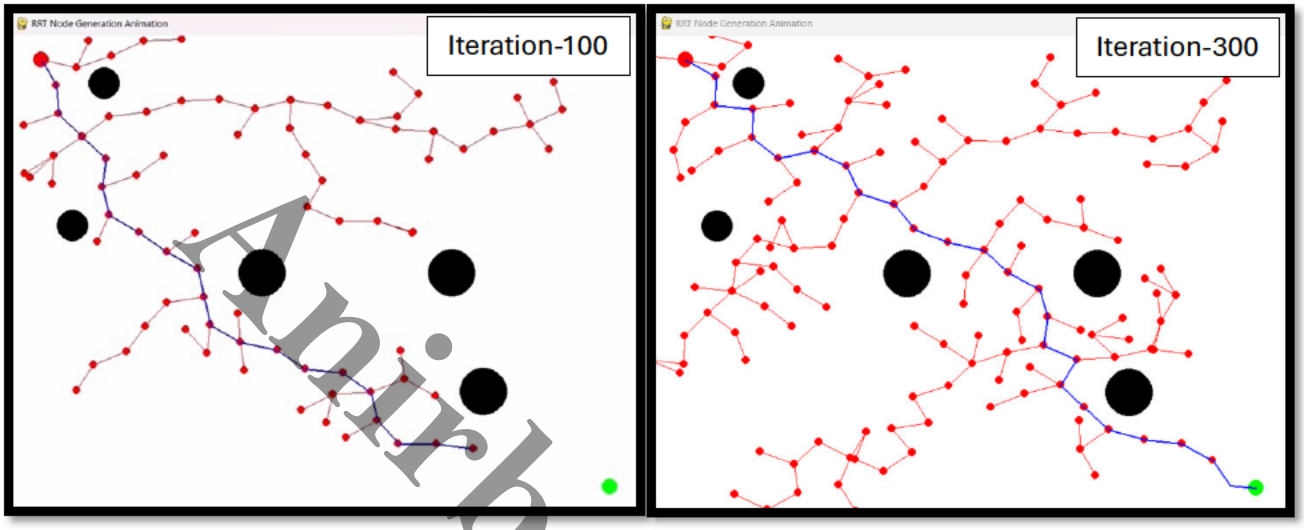

Fig 8: Visualization of RRT.

From Fig 8 we can see that when the number of iterations is 100 or same as Bi-RRT, the tree doesn't reach the goal position. When the number was increased to 300, from the figure we can observe that the tree reaches the goal position, and a path is constructed. Thus, we can conclude that Bi-RRT is more efficient and faster than RRT.

4. Path Tracking

For the path tracking algorithm, I am using 'Follow the Carrot' algorithm. The 'Follow the Carrot' algorithm offers a balance between simplicity, real-time performance, smoothness, and adaptability, making it a suitable choice for my project.

The ‘Follow the Carrot’ algorithm looks exactly like the image below if you visualize it which is a donkey chasing after the carrot to eat it but can’t.

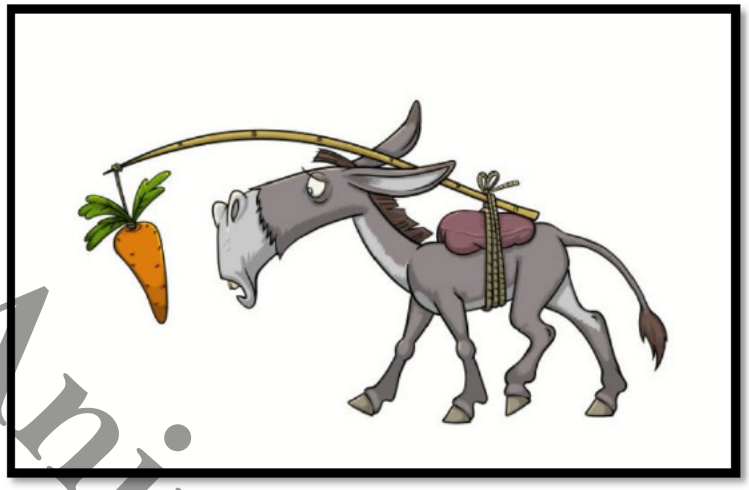

Fig 9: Visualization of “Follow the Carrot”.

4.1. Follow the Carrot

The "Follow the Carrot" path tracking algorithm, also known as the "Carrot-Chasing" algorithm, is a simple yet effective method for steering a mobile robot along a predefined path or trajectory.

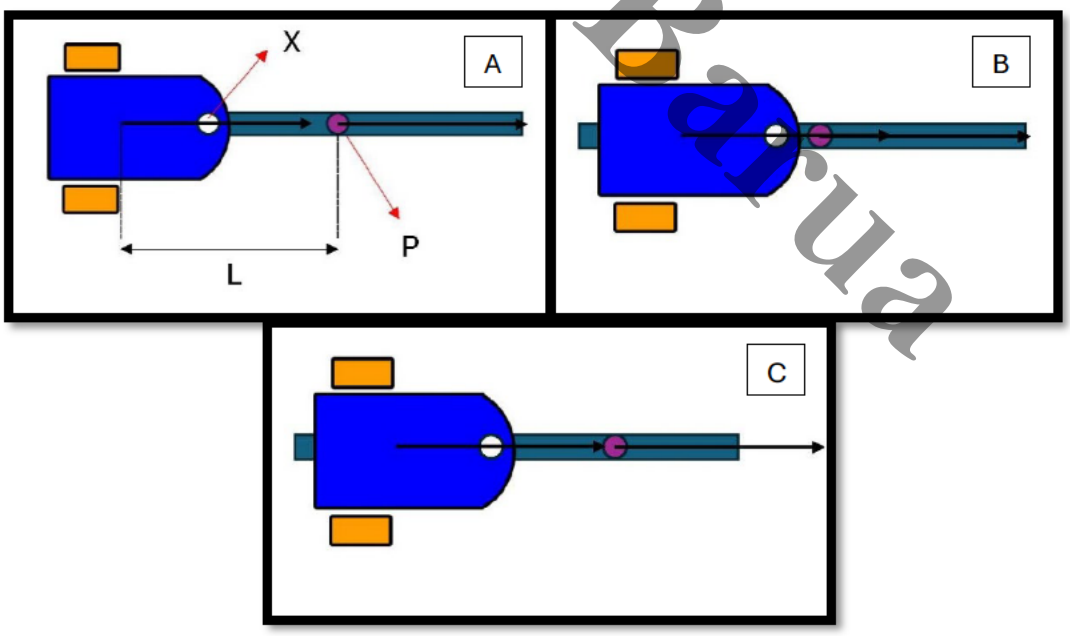

Fig 10: "Follow the Carrot" algorithm steps for a straight path.

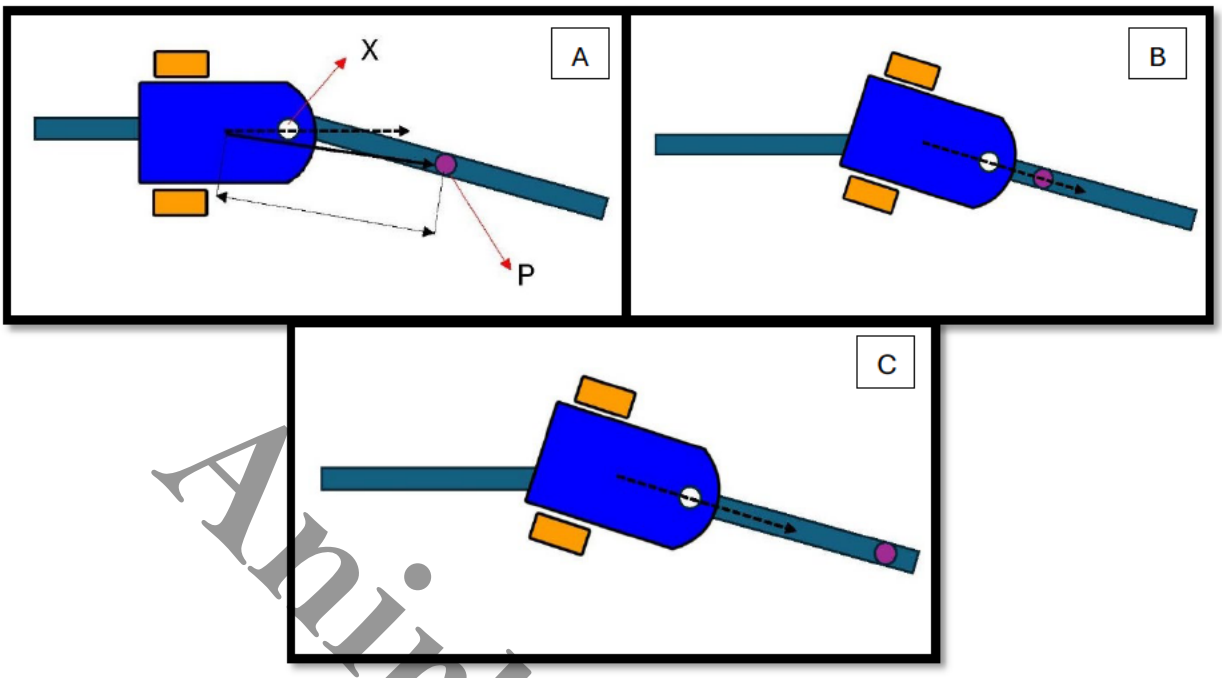

Fig 11: "Follow the Carrot" algorithm steps for a curved path.

First, the algorithm gets a closest position (P) or the carrot on the path relative to robot's front reference point (X). Then computes the point P coordinates relative to the robot's reference frame. From this computation the algorithm gets distance error (L) and orientation error or angle (β) which can be positive or negative (Fig 10(A)).

The robot will start moving using initial velocities Vright = 1.0 and Vleft = 1.0 when β=0.

Otherwise, it will use the following velocity equations:

Vright= (1.0+kp* β) if β < 0 and Vleft= (1.0-kp* β) if β > 0

Where Vright and Vleft are the robots right and left motors velocity respectively and kp is the proportional gain.

For Fig 10 case, β = 0 because the robot's frame is in axis with the point p. Therefore, Vright = 1.0 and Vleft = 1.0.

Thus, the robot will go straight. For Fig 11 case, where the path is curved, $\beta \neq 0$ because the robot's frame is not in axis with the point P. Therefore, Vright will be less than 1.0 and Vleft will remain the same. As a result, the robot will be steering to right until $\beta = 0$.

When the Distance error (L) is equal to the certain distance threshold or close to point P (Fig 10(B) and 11(B)), the algorithm will create another point on the path (Fig 10(C) and 11(C)). This process will continue until the robot reaches the goal node.

5. Simulation

5.1 Simulation Software

For the simulation software, I am using 'CoppeliaSim'. 'CoppeliaSim' is a robotics simulator with an integrated development environment. Each object/model can be individually controlled via an embedded script, a plugin, ROS / ROS2 nodes, remote API clients, or a custom solution in this environment. Moreover, robot controllers can be written in C/C++, Python, Java, Lua, Matlab, or Octave. Its user-friendly UI has built-in environments and robots, so we don't need to set it up from scratch and focus more on the simulation. The installation process is easy. We need to go to the official website and download the 'edu' version, which is free and better than any other paid software.

5.2 Robot

The robot that I am using for my simulation is called ‘Tron’ which is a custom-made differential drive robot with two wheels, a castor wheel and a front view camera mounted on top (Fig 12).

Fig 12: Differential Drive Robot ‘Tron’.

I have developed this robot with the help of ‘Solidworks’ and ‘CoppeliaSim’ software. The parts were created in ‘Solidworks’ and were imported to ‘CoppeliaSim’ where the robot was assembled.

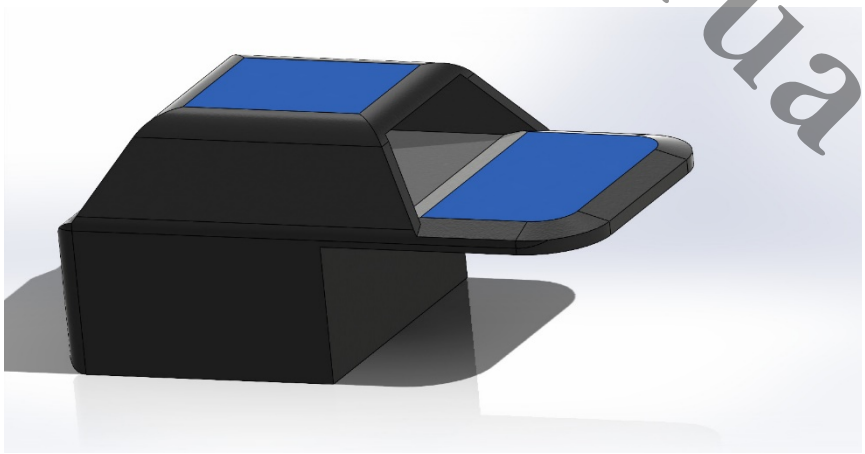

Fig 13: Top part of the robot in ‘Solidworks’.

The figure given below is the assembly tree of the robot.

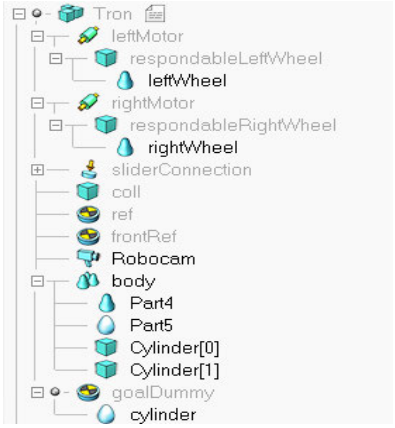

Fig 14: Assembly tree of the robot.

The subtree of the robot 'body' defines the robot's chassis or outer shell. The 'ref' is the point that defines the robot's frame position, and 'front ref' defines a point of a robot at the front which is used to create the carrot point as mentioned at the 'Path Tracking' section of this report. The 'goalDummy' represents the robot's goal position. The 'sliderConnection' represents the castor wheel. For the 'leftMotor' and 'rightMotor', I have used the revolute joint. Moreover, the object properties and dynamic properties of both motors are shown in the figure below.

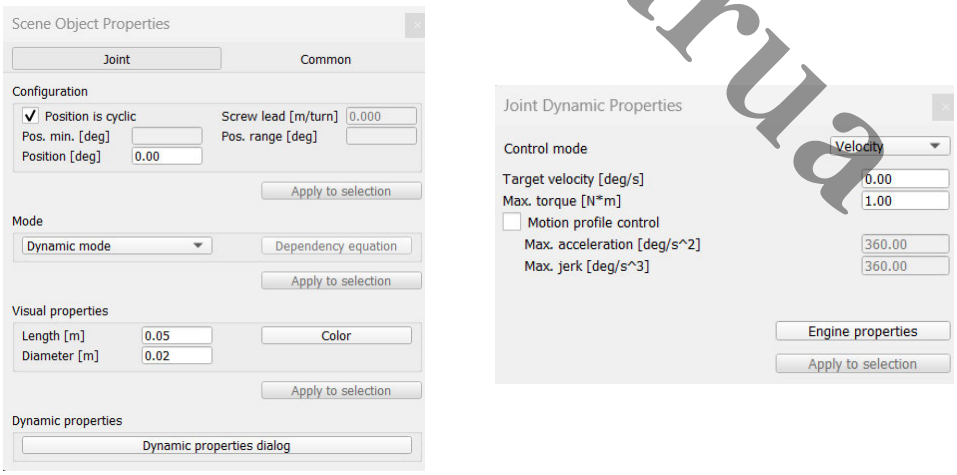

Fig 15: Object properties and Dynamic properties of left and right motor.

The ‘coll’ object demonstrates the robot’s collision circle or cylinder which we can see from the figure below.

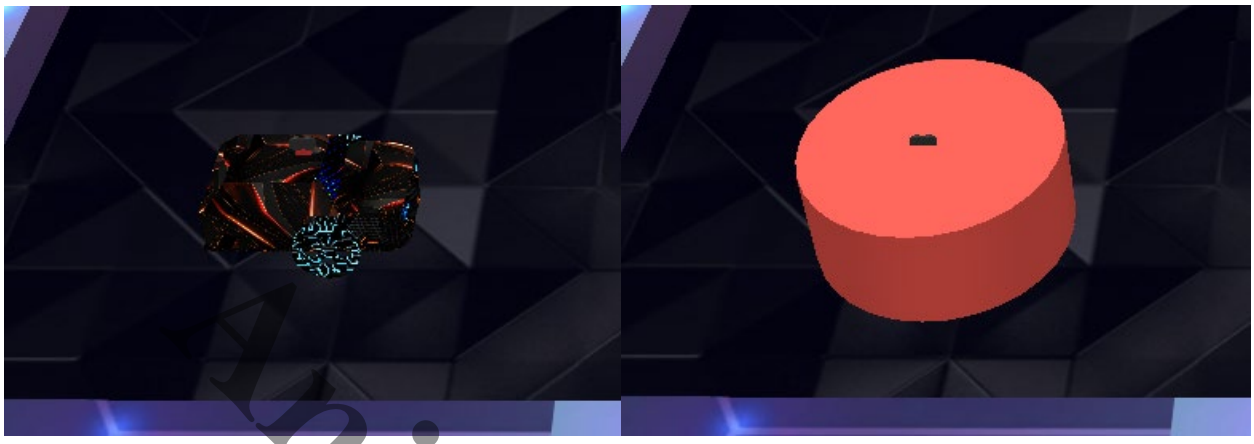

Fig 16: Robot’s collision cylinder.

Moreover, in the object property we need to tick all the ‘special object properties’ so that it can detect collisions. Now, the robot is ready for simulation.

5.3 Simulation Map

I have created an obstacle course or a maze on the floor of the simulation map which is shown in the figure below.

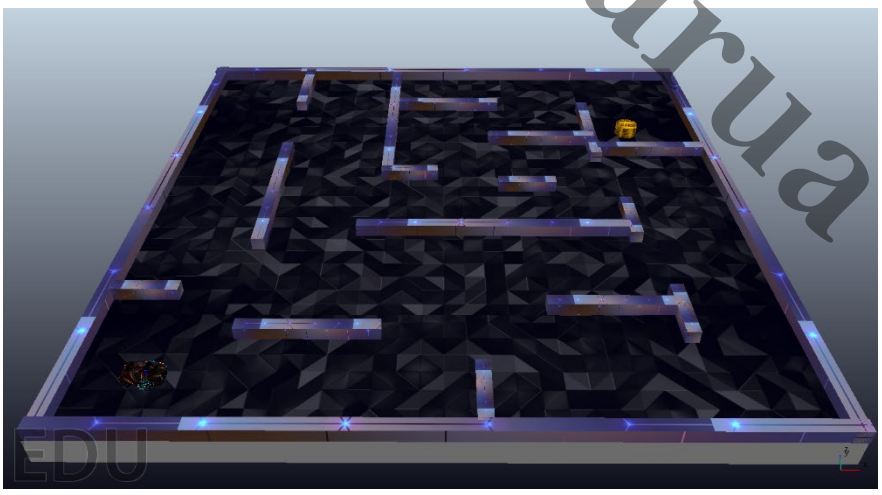

Fig 17: An obstacle course or a maze on the floor.

Where the yellow cylinder is the destination where the robot wants to go. The walls of the maze were created with rectangular objects.

5.4 Simulation Code

The code is written in Lua and the code is inside the scenes that has been provided in the zip file. The Explanation of the code is given below.

Initialization and Setup:

```
sim=require'sim'
simOMPL=require'simOMPL'
function sysCall_init()
  -- Obtain handles to various objects/components in the simulation environment
  robotHandle=sim.getObject('.')
  refHandle=sim.getObject('./ref')
  frontRefHandle=sim.getObject('./frontRef')
  leftMotorHandle=sim.getObject('./leftMotor')
  rightMotorHandle=sim.getObject('./rightMotor')
  collVolumeHandle=sim.getObject('./coll')
  goalDummyHandle=sim.getObject('./goalDummy')
  frontCam=sim.getObject('./Robocam')
   -- Create a floating view to display the robot's front camera feed
  frontView=sim.floatingViewAdd(0.9,0.9,0.9,0.9,0)
  sim.adjustView(frontView,frontCam,0)
 -- Create a collection for robot-related obstacles
  robotObstaclesCollection=sim.createCollection(0)
  sim.addItemToCollection(robotObstaclesCollection,sim.handle_all,-1,0)
  sim.addItemToCollection(robotObstaclesCollection,sim.handle_tree,robotHandle,1)
  collPairs={collVolumeHandle,robotObstaclesCollection}
  -- Set the parent of the goalDummyHandle to none initially
  sim.setObjectParent(goalDummyHandle,-1,true)
  -- Define various parameters and settings for the robot's behavior
  velocity=180*math.pi/180
  searchRange=5
  searchDuration=0.1
  searchAlgo=simOMPL.Algorithm.BiTRRT
  displayCollisionFreeNodes=true
  showRealTarget=true
  showTrackPos=true
```

This section initializes the script and sets up various parameters, including obtaining handles for objects in the simulation environment (e.g., robot components, sensors). It also creates a dynamic view to show the robot's front camera footage, creates a group for obstructions related to the robot, sets motion-related variables, and configures collision detection settings. It also uses motion planning algorithms provided by the 'simOMPL' library to generate collision-free paths for the robot to follow while avoiding obstacles. At the variable 'searchAlgo', we can see that the selected algorithm is Bi-RRT. To use the RRT algorithm just write 'simOMPL.Algorithm.RRT'.

Cleanup:

```
function sysCall_cleanup()
  sim.setObjectParent(goalDummyHandle,robotHandle,true)
end
```

When the simulation ends or the script is stopped, this routine is executed. It resets the parent of the 'goalDummyHandle' to the robot's handle to clean up after the simulation.

Collision Checking:

```
function checkCollidesAt(pos)
  -- Check if the robot collides at a given position
  local tmp=sim.getObjectPosition(collVolumeHandle)
  sim.setObjectPosition(collVolumeHandle,pos)
  local r=sim.checkCollision(collPairs[1],collPairs[2])
  sim.setObjectPosition(collVolumeHandle,tmp)
  return r>0
end
```

By setting the position of a collision volume and checking for collisions with obstacles in the environment, this function examines if the robot collides at a specified position.

Visualization Functions:

```
function visualizeCollisionFreeNodes(states)
  -- Visualize collision-free nodes in the motion planning algorithm

end

function visualizePath(path)
  -- Visualize the computed path for the robot to follow

end
```

These functions handle the visualization of collision-free nodes as well as the computed path for the robot's motion planning algorithm.

Main Thread:

```
function sysCall_thread()
  while true do
    -- Main loop for controlling the robot's behavior
  end
end
```

This is the main thread of the script that continuously runs while the simulation is active. It contains the logic for controlling the robot's behavior, including collision avoidance, motion planning, and

path following. It comprises the logic for managing the robot's actions, such as motion planning, collision avoidance, and path following.

5.5 Running the Simulation

First the simulation has been executed using the Bi-RRT path planning algorithm.

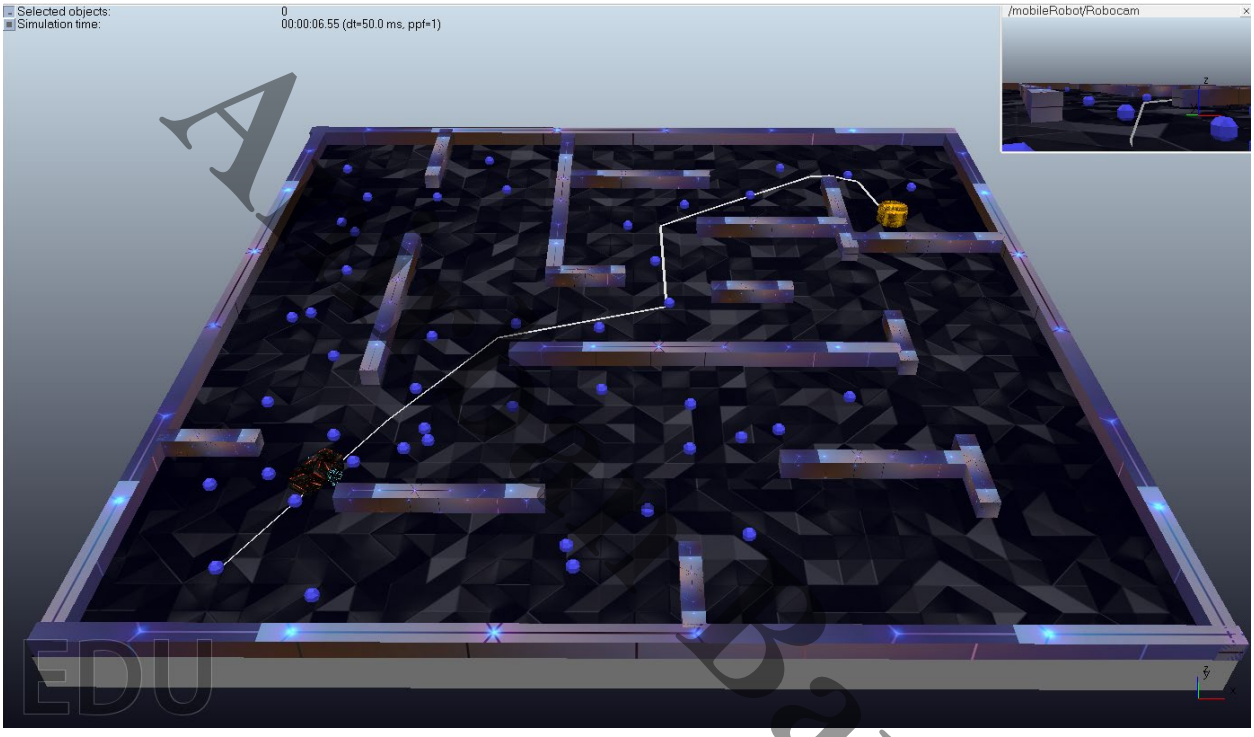

Fig 18: Simulation using Bi-RRT.

The figure shows that a path represented in white has been generated from the robots' initial position to the goal position represented in white. The map's blue spheres visualize the nodes generated during the path-planning process. The robot's front camera feed is at the top right corner of the map. Moreover, for this simulation, the search duration is set to 0.1, which, in other words, is the number of iterations. The figure below shows the carrot, which is an orange semi-circle, and the robot is chasing it.

Fig 19: ‘Follow the Carrot’ simulation.

Furthermore, when the robot reaches the goal, it stops.

Fig 20: Robot reached the destination.

The second simulation has been executed using RRT with the same search duration as Bi-RRT.

Fig 21: Simulation using RRT with the same search duration as Bi-RRT.

We can see that the tree didn't reach the goal position. Moreover, the number of nodes is also low.

Now, for the third simulation, I have increased the search duration from 0.1 to 0.8.

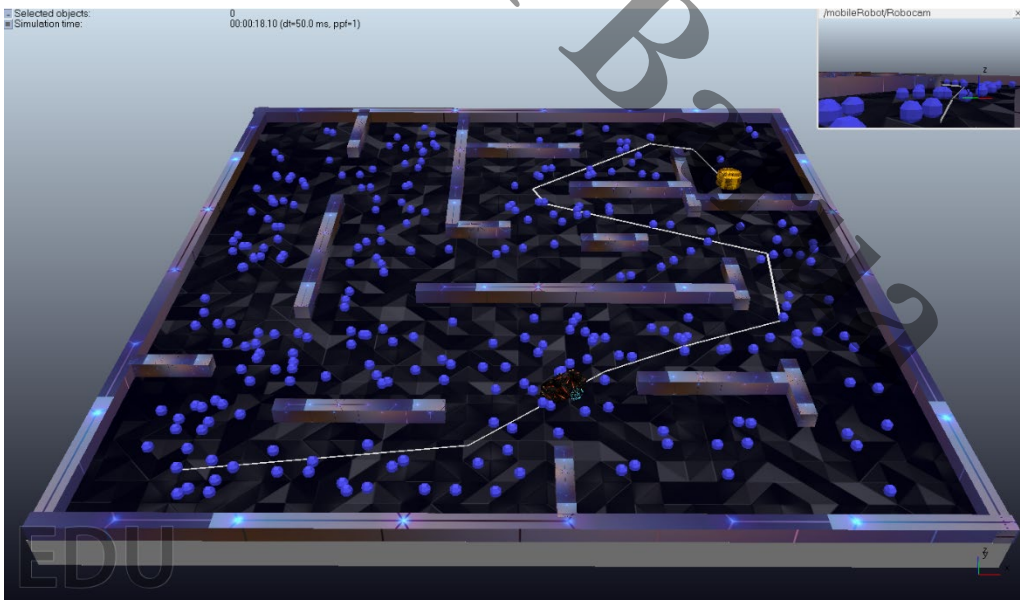

Fig 22: Simulation using RRT with increased search duration.

From the figure, we can see that the number of nodes is more than previous, and the tree reaches the goal position.

5.6 Result and Discussion

Based on my simulations, I concluded that Bi-RRT path planning is more effective and faster than RRT. This is because Bi- RRT requires only 0.1 sec search duration to expand the trees enough to find a path, but in the case of RRT, this algorithm requires 0.8 sec to expand the tree to reach the goal node and construct the path.

5.7 Limitation

My simulation only works for pre-existing or static obstacle maps not for dynamic obstacles.

6. Conclusion

In conclusion, both Bi-RRT and RRT have been explained, visualized, and compared. Moreover, simulation has been conducted to analyze their performance. The 'Follow the Carrot' algorithm is described briefly and simulated. In the simulation, both path planning and path tracking algorithms, the robot successfully generated the optimal path and followed that path effectively without any error.