Rapidly Exploring Random Trees for Self-Driving Vehicles and Autonomous Parking Lot

1st Diane Ngo
Robotics Engineering, ENPM
University of Maryland
College Park, USA
dngo13@umd.edu

2nd Eitan Griboff
Robotics Engineering, ENPM
University of Maryland
College Park, USA
egriboff@gmail.com

***Abstract*—As the industry advances in technology, autonomous robots are increasingly gaining popularity. In order to perform a task or travel from start to goal point, path planning is implemented alongside its operation. Sampling based path planning algorithms are widely used due to efficiency in computation for higher dimensional graphs and collision detection. Generally, random points are chosen from the graph and then connected if they aren't through obstacles. This process is repeated until a path between the start and goal nodes are found. For this project, Rapidly Exploring Random Trees (RRT) is implemented for an autonomous parking for self-driving vehicles. The simulation is created using ROS Gazebo and RVIZ, with Turtlebot3 acting as the vehicle.**

***Index Terms*—path planning, sampling methods, rrt, random trees, autonomous parking, smart garages, turtlebot, gazebo, ros**

I. Introduction

Technology is advancing at a rapid rate, making the present and future smarter through autonomous machinery. Robotics has become a hot topic, especially with self-driving vehicles. With a rising population and crowded cities, navigating a city and finding parking is difficult. Car companies are tackling this issue through electric and autonomous vehicles. Creation of smart parking garages are in development through Internet of Things, and sensors are communicating whether or not a parking space is occupied. Self-driving vehicles are equipped with a multitude of sensors, ranging from RGB cameras, ultrasonic sensors, to LIDAR and distance sensors. Figure 1 is an image to show autonomous parking.

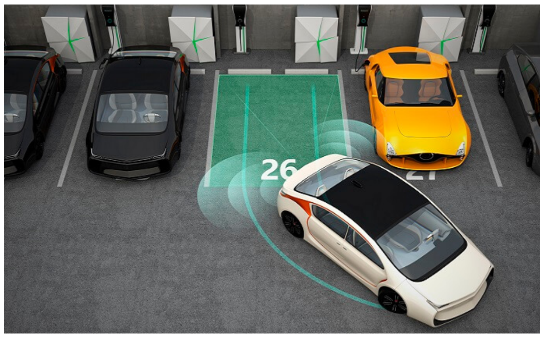

Fig. 1. Autonomous Parking

Autonomous robots require path planning in order to perform a task without being user controlled. There are many different types of path planning. The two main categories fall under Systematic Planning and Probabilistic Methods. Systematic Planning uses Actions Based planning or Systematic Road-maps. Probabilistic Methods use Probabilistic Road-maps (PRM) or Rapidly Exploring Random Trees (RRT). This project focuses on the RRT algorithm for path planning. In general, Probabilistic Methods are a type of Sampling-Based Road maps. They pre-compute a graph or road-map of the configuration space and need to be reachable from start and goal. The configuration space, or C-space, is a way of visualizing the system as one point in a higher dimensional space. Graphs are a way to visualize the C-space, where edges represent trajectories and nodes represent configurations.

Unlike Probabilistic Methods, where a graph is pre-computed, RRT does not generate a graph beforehand. Given a start and goal point, the algorithm chooses random points in the C-space if it not in an obstacle, connects the points and stops when a path is formed between the start and goal. To simulate RRT in a parking lot environment, a simple 5(spaces)x2(columns) with 3 lanes environment has been created in ROS Gazebo. Turtlebot3 burger model is chosen as its wheels have better traction control over the other models and uses differential drive.

We utilize a few different libraries in order to visualize different aspects of the project. OpenCV, most notably, draws the parking lot in 2D and numbers each space so that the user has a visual reference when choosing what space to park in. Matplotlib displays the branches that the algorithm created in RRT and the path from start to goal. ROS is heavily used for simulating the parking in a 3D environment with Gazebo, and a 2D visualization tool with RVIZ. In general, the project starts by using OpenCV to prompt for goal parking space, Matplotlib to show the graph and path, Gazebo for 3D simulation, and RVIZ for 2D visualization while the robot is moving in Gazebo.

II. Background

In path planning, a configuration space has to be defined. It is a map or graph that consists of the environment with any possible obstacles that needs to be avoided. It can be written C-space. However, a configuration space can be set with only the open area for the robot to move through, defined as C-free.

The project uses RRT, not RRT*, meaning it is not optimized. Running the program multiple times with the same goal can lead to the program taking a different path and time taken to run the program every time. This is because each time the program is ran, it selects different random points and not all the points could be optimal.

III. Methodology

For this section, the methodology for the project will be explained. Implementation through pseudocode is provided.

A. Explanation

For the implementation of the project, the navigation launch file first sets the default Turtlebot model. Then, it starts up Gazebo by calling the parking world launch file, which contains the starting position and orientation of the turtlebot, as well as the parking lot environment. Finally, it pulls up the RVIZ map generated previously by running the map server. The Gazebo world consists of a grid which is 5m in the x-direction and 3m in the y-direction. The origin of this box is (0,0) in the Gazebo world. Within the grid are three sets of two rows of parking spaces. Each row of parking spaces has 5 spaces, and the spaces are approximately 0.4m tall, and 0.5m wide (Figure 4).

After running the launch file, the RRT ros file needs to be ran inside another terminal. This prompts the user for a desired parking space and sets it as the goal for the robot to travel to. figure 2 is the image that the user will use to choose the goal. The program will then run the algorithm until a path to the goal is found. Once the program displays the generated graph containing the path to the goal and the branches connection, the user can exit the graph to start running the turtlebot to the goal. Figure 3 is one of the visualization images the user will see when the algorithm is completed.

The RRT method was implemented using the following algorithm. Using the space that the user had input, a goal node would be determined. As part of the setup of the environment, the start and end coordinates for the 30 spaces were chosen. Therefore, once the user input to which space the robot should drive, it could be determined where that was in the environment given the values that had been chosen as part of the environment setup. This space would then be popped out of a list which had been previously created which contained the limits of each of the parking spaces. For example, if the user chose space 13, then the coordinates

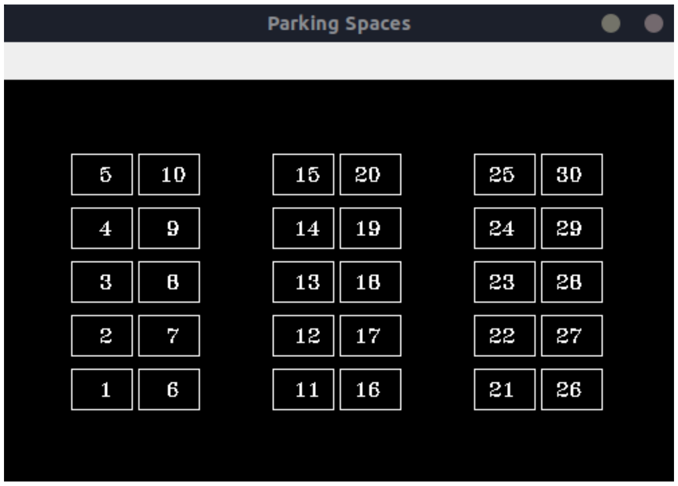

Fig. 2. Parking Spaces

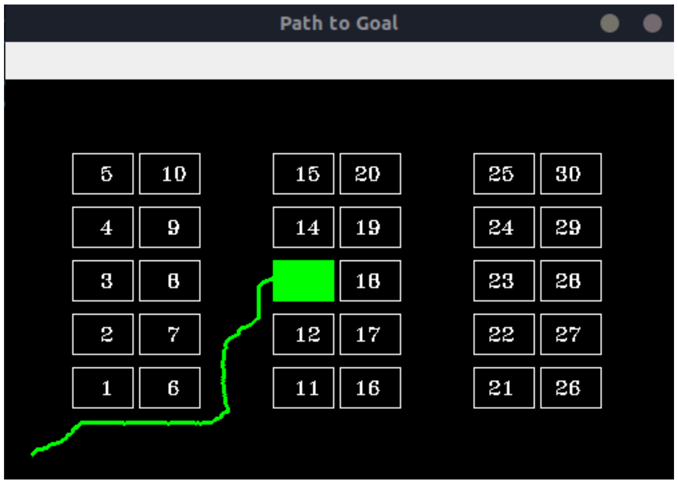

Fig. 3. Path to Goal Space

for space 13, which happen to be x=[200:245], y=[135:165], would be popped out of the list which contained the values for each space. Now the program knows where the obstacles are, and knows that space 13 is not an obstacle. However, in space 13's place are the top and bottom lines of space 13. That way the algorithm will not find a path that would go through the top and bottom lines which are walls in the Gazebo environment. Given that the start location was always chosen to be (20, 20), the RRT algorithm can now begin.

A list called "nodes" was created and the first entry was given as the start node, which is (20, 20). In this list will be contained all of the visited nodes by the program. Next the program goes through a set number of iterations that it has in order to find a path from the start node to the goal node. For this project, this iteration count was chosen to be 35,000 iterations. For each iteration a random node would be generated. This node would then be checked to see if it was a node close to the previous node. This was done by checking the Euclidean distance between each node. If this distance was less than a certain value, then this node was close to the current node and it would be added to the path as long as it did not collide with one of the obstacles or leave the workspace. Each node which was not chosen, would be added to a branch. This nearest node was then added to the current node to provide the next node in the path.

This process continues until the Euclidean distance between the current node and the final goal node is less than a certain value. For this project, this value was chosen to be equal to 5. At this point, the program would go and start the backtracking process to find the shortest path. For this process, the program would go through each value in the ”edges” list and see if a path could be created between start and goal nodes. If so, then the value of each point in the ”nodes” array that made up the path, would be marked down. For example, if the 1st, 20th, 35th, 50th and 65th values in the ”nodes” array made up the path, then the values (1, 20, 35, 50, 65) would be what are contained in the list. That way, when attempting to draw the visual representation of RRT, the program knows which values of the ”nodes” list to use.

With the path determined, the program would now draw the path. As previously mentioned, to find the shortest path one would use the values in ”nodes” which correspond to the entries which make up the shortest path and these would be plotted in green. All other values in ”nodes”, which make up the branches which come from the RRT method, are then plotted and marked in blue. That way the user can see the options that the program created, and what the final option that the program chose was. When the visualization is complete, the program shows the user the path with the branches in matplotlib, and then afterwards will show the user just the path on the grid showing the parking spaces, as seen in Figure 3. This way the user can see what will happen once the robot begins to drive from the start point to the goal point.

Something important to note is that turtlebot uses differential drive. Both wheels on the robot are separately driven, so it can robot at different speeds. When robots with differential drive turns, this can result in a bit of a swivel or curvature. Thus turning is not accurate, and the program needs to consider differential drive constraints when turning in Gazebo. Equations (1), (2), and (3) are parametric representations of differential constraints. They can be used as configuration transition equations to map the forward speed u_s and the steering angle u_ϕ to a set of allowed velocities.

$$\dot{x} = u_s cos(\theta) \tag{1}$$

$$\dot{y} = u_s sin(\theta) \tag{2}$$

$$\dot{\theta} = \frac{u_s}{L} tan(u_\phi) \tag{3}$$

In order to drive the robot from the starting point to the user-input parking space, the following method was used. The first step was to determine each node of the shortest path. This was taken from the RRT algorithm. With this list, one could now iterate over that list and use those points to begin to drive from one coordinate to the next. To determine where the robot was at any given time, odometry data was collected. Comparing the odometry data, which gave the current position of the robot, to the next node within the shortest path, allowed the robot to know how far it needed to move in order to get to the next node, which would become its next starting point. To drive the robot itself, the velocity message from the Twist package was used. This allowed control of the linear and angular velocities of the robot. Based on the determined angle that the robot needed to move, and the distance that it needed to go, the robot would be given a certain linear and angular velocity and use those values to get to the next node. Once the robot got to the next node, it’s velocities would be set to zero, and it would wait for the next node in the shortest path list. Because this method was not perfect, it was decided that in order for the robot to stop and wait for the next node it would have to be within 0.3m of the normalized distance between the current node and the next node.

This worked well until it came to turning up one of the aisles. At this point, the robot would not turn far enough in order to avoid the walls of the different parking spots. In order to combat this issue, a function was used which would rotate the robot a certain number of degrees, until it was able to complete the necessary turns for reaching the parking space. This function would take a number of degrees to turn and an angular velocity, and would set the linear velocity to zero while the robot was turning. Therefore, the robot would be turning in place, so that it would have the same (x,y) position, but only the orientation would change. The direction of these changes, left or right, were determined based on the current position and orientation of the robot as determined by odometry data. By changing only the orientation, it became more likely that the robot would be able to make future turns. This same method was also used when turning into the parking space itself. To ensure, a smooth entry into the parking space, this same roatate function was used to make sure that the robot would enter the parking spot straight, and stop towards the middle of the space. While this means that the implementation was not a perfect RRT implementation for driving the robot, except for these couple of nudges in the right direction, all other moves of the robot were done using the nodes obtained via the RRT algorithm.

B. Pseudocode

The main RRT algorithm we have created can be simplified as pseudocode below. It repeats a process of selecting a random point from the C-space, checking if the point is inside an obstacle, and calculating the heuristic. If the distance between the current point and the goal is within a threshold, then the goal is reached.

RRT ALGORITHM

```
get start and goal nodes, obstacles
set max iterations N = 35000
**for** $i = 0$ in range $N$ **do**
    $newnode \leftarrow randompointinsideC - space$
    get closest nodes from new node
    check if new node is inside obstacle
    **if** not in obstacle **then**
        add new node to edges
    **end if**
    **if** heuristic of $edge \leq 5$ **then**
        find path between the edges
    **end if**
**end for**
```

A flowchart containing a visual representation of launching the project up until calculating the RRT algorithm is seen in Figure 9. After the program has calculated the path to the goal, the robot will start driving. A basic flowchart explaining how the driving works is seen in Figure 10. The Gazebo world environment can be seen with Figure 4. It uses a thin wall to define the lanes because there was no way to draw lines on the world. This also benefits us by having an easier time detecting obstacles with the simulated lidar sensor through a laser scan.

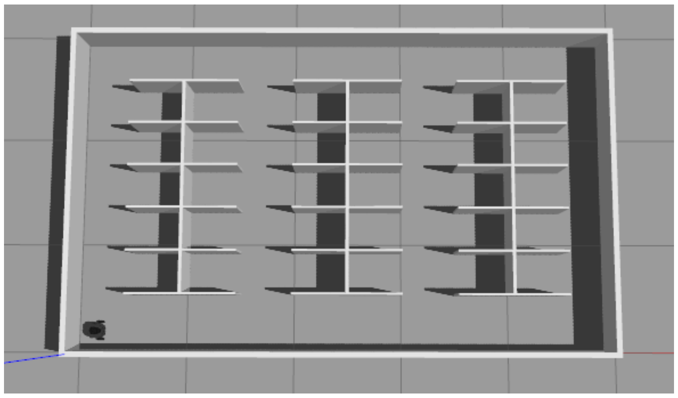
Fig. 4. Gazebo Environment

IV. Results

This project consists of different visualizations to show processes separately. OpenCV is used to start the project off by asking the user to choose which parking space to be designated as the goal. After getting this parameter, the program proceeds to utilize RRT to find a path to the goal before running the robot. Matplotlib, a data visualization library from Python, is used to graph a 2D plot of the RRT branches and the path found. Figure 5, 6, and 7 are some of the cases we've tested for the RRT algorithm. The time computed for each case will always be different, and can take a longer time than others.

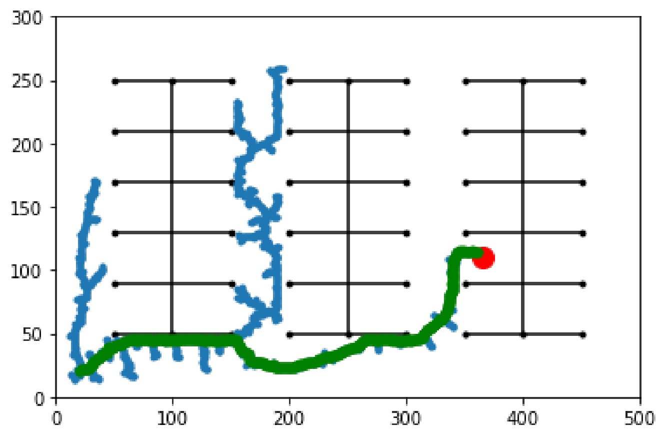

Fig. 5. Case 1

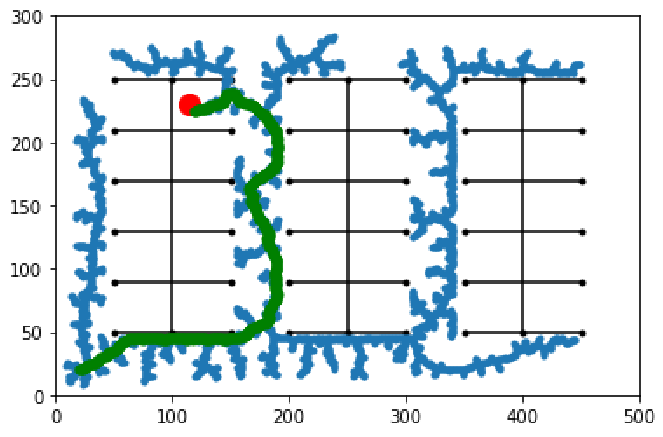

Fig. 6. Case 2

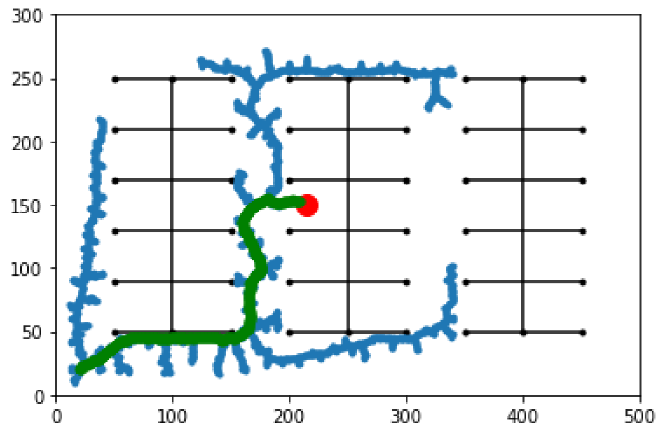

Fig. 7. Case 3

In each of the cases, the program has taken the randomly generated nodes, and plotted them as branches, which are in blue. It then takes the different nodes which make up the shortest path between the start point and end point and marks those nodes in green. It's important to note that each branch will not overlap with an obstacle, whether it's a different parking spot, or the edges of the desired parking space.

Another interesting observation from the results is that due to the random nature of the nodes, there will not be two paths that are identical to one another. One can run the program to go to one parking space multiple times and obtain multiple paths. This can actually cause issues with the driving of the robot to the parking space, as it is possible that there will be a path that contains too sharp of a turn for the robot to make, thus making it impossible for the robot to reach the desired parking space. However, this does not happen all the time and in a majority of the cases, the robot will be able to take the developed path to the desired parking space.

To demonstrate the fact that each run will take a different amount of time finding a path to the goal, Table IV is shown below. Five different runs were tested with the same parking space, space 13, and each had varying times, with the shortest being three seconds and the longest taking 151 seconds, or two minutes and 31 seconds. The longest case can be seen in Figure 8. The program almost filled all of the empty space because the random points calculated weren't able to find connecting nodes.

Trial Number	Calculation Time (s)
1	3.456
2	3.127
3	5.522
4	151.456
5	51.215

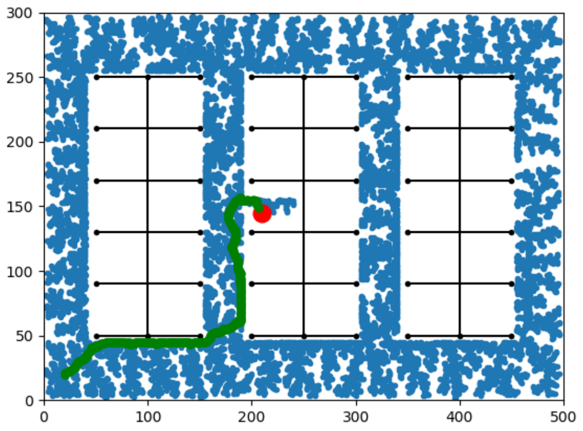

Fig. 8. Longest Case, Space 13

V. Future Work

Unfortunately, not all of the desired functionalities were able to be implemented in this project and report. As such, there is still room for future work on this project. One such aspect which needs more work is the driving of the robot from the start point to the goal point. While it does work decently at the current moment, it does not work every single time that the program is run. This can be due to a couple of factors. First, the algorithm for determining the next location for the robot does not drive the robot perfectly to that location. It only comes with 0.3m of the variable distance-to-goal. As such the robot will be slightly off. This algorithm also does not perfectly account for the correct orientation of the robot. While the robot will start the turn in the correct direction, it will not end up at the perfect orientation. As such, some help needed to be added for the robot so that it could reach the correct orientation at different points throughout its attempt to park the robot in the parking space. With more time, it is likely that an algorithm that would provide for a more exact orientation and position of the robot could be developed.

Another desired function is after the program had calculated the path to the goal with RRT, it would display those branches on RVIZ through the use of markers or lines. RVIZ has a Marker class from the navigation messages in which we thought we could use to publish x and y values to it, which would then draw lines for each x and y coordinate. However, custom RVIZ environments weren't really covered in depth over the course, and google mainly had C++ examples. With more time and research, this function would've had a higher chance of success. Thankfully, RRT visualization was shown in this project in a 2D manner, which is enough to inform the user of how the algorithm works and the different branches that could have been chosen as paths.

References

[1] S. Klemm, et al., "Autonomous multi-story navigation for valet parking," 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), 2016, pp. 1126-1133, doi: 10.1109/ITSC.2016.7795698.

[2] A. Correa, G. Boquet, A. Morell, and J. L. Vicario, "Autonomous car parking system through a cooperative vehicular positioning network," Sensors, MDPI, April 2017. Barcelona, Spain. doi:10.3390/s17040848.

VI. Appendix

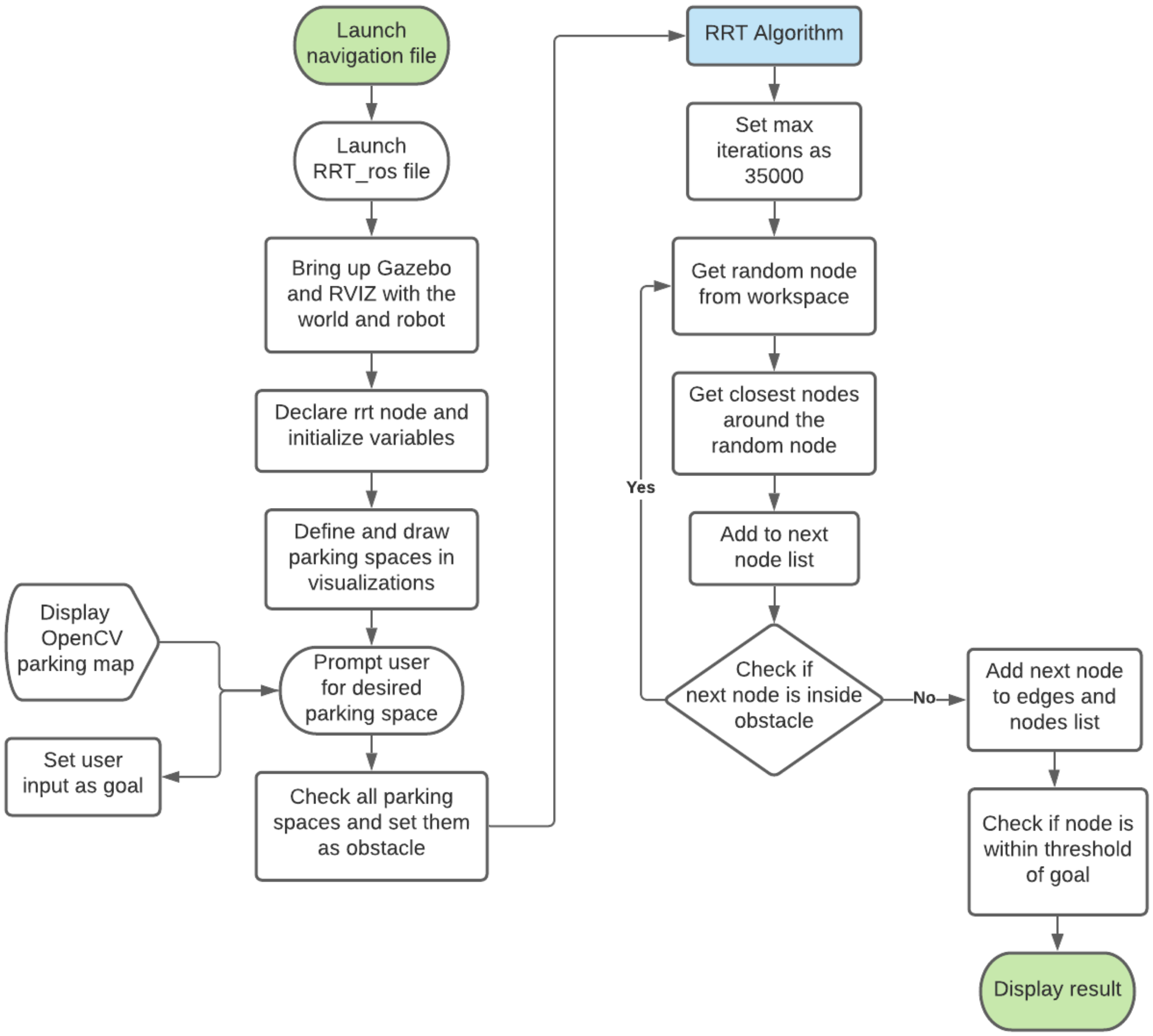
Launch navigation file
Launch RRT_ros file
Bring up Gazebo and RVIZ with the world and robot
Declare rrt node and initialize variables
Define and draw parking spaces in visualizations
Display OpenCV parking map
Prompt user for desired parking space
Set user input as goal
Check all parking spaces and set them as obstacle
RRT Algorithm
Set max iterations as 35000
Get random node from workspace
Get closest nodes around the random node
Add to next node list
Yes
Check if next node is inside obstacle
No
Add next node to edges and nodes list
Check if node is within threshold of goal
Display result

Fig. 9. RRT Flowchart

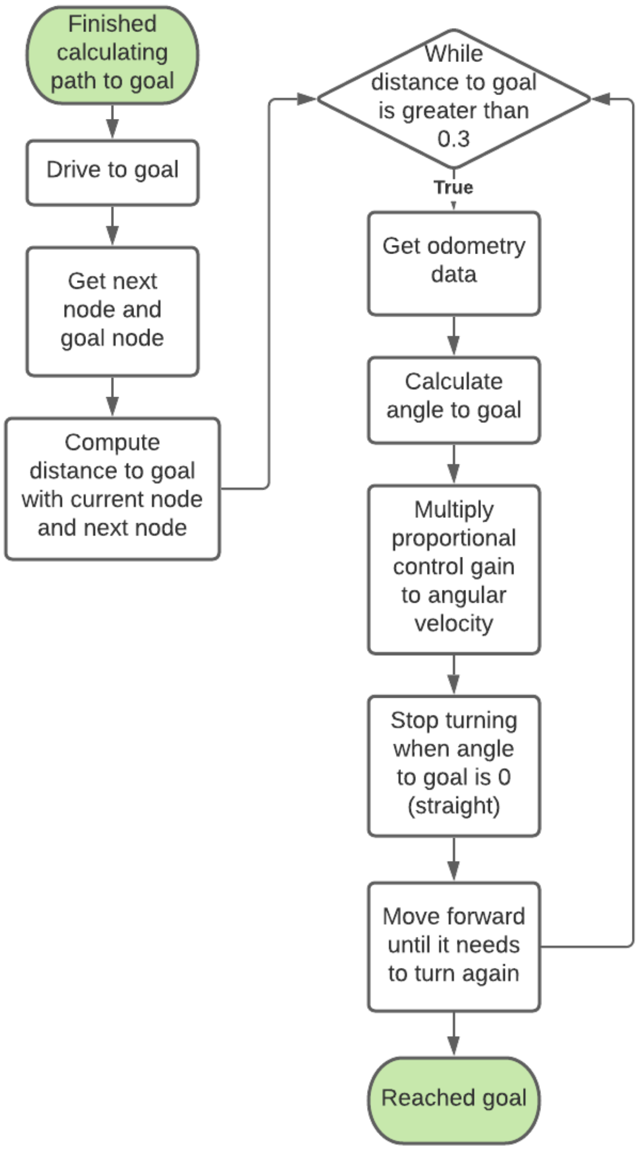
Finished calculating path to goal
Drive to goal
Get next node and goal node
Compute distance to goal with current node and next node
While distance to goal is greater than 0.3
True
Get odometry data
Calculate angle to goal
Multiply proportional control gain to angular velocity
Stop turning when angle to goal is 0 (straight)
Move forward until it needs to turn again
Reached goal

Fig. 10. Driving the Robot Flowchart