California Institute of Technology

Pasadena, California

ME/CS/EE 169 : Robotics

Spring 2022-2023

Final Project Report

for

Non Holonomic RRT with Dynamic Replanning and Obstacle Mapping

Submitted by

Team Name : Astrorangers

Team Members : Sri Aditya Deevi & Jeff Chen

9 June, 2023

I Problem Description

1 Problem Statement

The aim of this project is build a mobile robot that has the following functionalities:

(i) It can globally localize itself initially (without the need for Initial 2D Pose Estimate) using Monte Carlo Localization.

(ii) It can drive like a car (without turning in place) in a smooth way, while locally localizing itself.

(iii) It can plan its way *quickly* into narrow and challenging areas of the map. This allows it to effectively execute tasks such as entering a narrow garage and parallel parking.

(iv) It can detect new obstacles not in the map and stop to avoid any collisions.

(v) It can map the new obstacles into an "Obstacle Map" and dynamically replan to find a path around them to the Goal.

(vi) It should also try to enter new narrow garages and parallel park around new obstacles that were not in the map (which is a much more challenging problem) fairly well.

2 Transitioning from Goals to Project

2.1 Differential Drive to Nonholonomic Car

We initially had a Differential Drive based robot but we would like to "treat" like a non-holonomic car that cannot turn in place. Essentially our inputs change from (v_x, ω_z) to (v, ϕ), where ϕ is the steering angle. So, we would like to use the current odometry setup and therefore, we need write some equations to convert (v, ϕ) commands to (v_x, ω_z) commands so that the wheel control can execute accordingly.

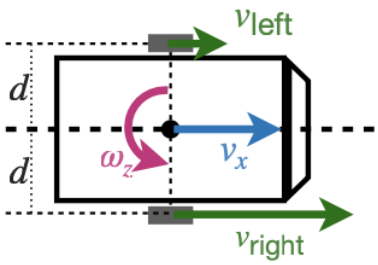


Figure 1: Schematic of a Differential Drive Robot

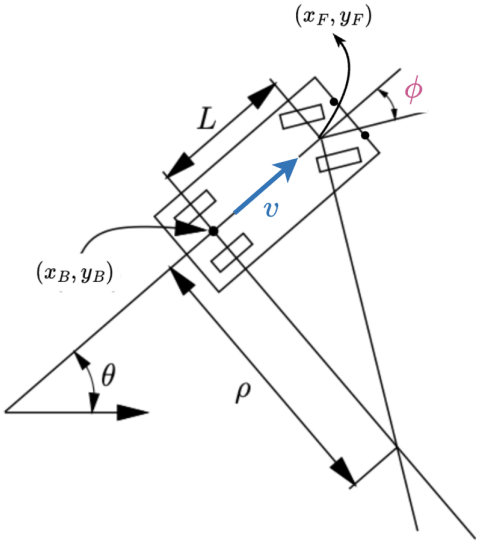


Figure 2: Schematic of a Non-Holonomic Car

From the above schematic, we can see that:

$$\rho = \frac{L}{tan(\phi)} \tag{1}$$

Using the convention in the Vehicle Kinematics Handout and assuming that there is no slippage, we can write the following equation involving the wheel rotational speeds as follows:

$$\begin{aligned} \frac{\dot{\psi}_{\text{left}}}{\rho + d} &= \frac{\dot{\psi}_{\text{right}}}{\rho - d} \\ \implies \dot{\psi}_{\text{right}} &= \dot{\psi}_{\text{left}} \left(\frac{\rho - d}{\rho + d} \right) = \dot{\psi}_{\text{left}} \, \alpha(\rho) \end{aligned} \tag{2}$$

Then let us consider the expressions of v_x and w_z in terms of $\dot{\psi}_{\text{left}}$ and $\dot{\psi}_{\text{right}}$. According to the current conventions:

$$v = v_x = -\frac{R}{2}\left(\dot{\psi}_{\text{left}} + \dot{\psi}_{\text{right}}\right)$$
$$v = -\frac{R}{2}\dot{\psi}_{\text{left}}\left[(1+\alpha(\rho)\right]$$
$$\implies \dot{\psi}_{\text{left}} = \frac{-2v}{R(1+\alpha(\rho))} \tag{3}$$

Then:

$$w_z = \frac{R}{2d}\left(\dot{\psi}_{\text{left}} - \dot{\psi}_{\text{right}}\right)$$
$$w_z = \frac{-v}{d} \times \left(\frac{1-\alpha(\rho)}{1+\alpha(\rho)}\right) \qquad \text{[Using (2) and (3)]}$$
$$w_z = \frac{-v}{d} \times \frac{-d}{\rho}$$
$$w_z = \frac{v tan(\phi)}{L} \qquad \text{[Using (1)]}$$

Finally, we can use the following equations to convert from (v, ϕ) to (v_x, w_z) that can be used by the current odometry setup to control the wheels:

$$v_x = v \tag{4}$$
$$w_z = v\frac{tan(\phi)}{L} \tag{5}$$

In order to test if everything works correct, we tried to drive the car in circles of different radii as shown in the following screenshots from RVIZ:

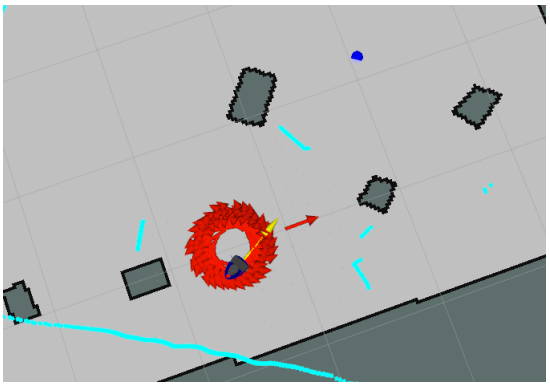


Figure 3: A Circle corresponding to $\phi = 30°$

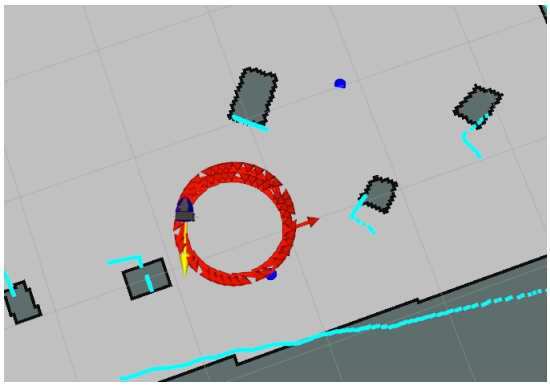


Figure 4: A Circle corresponding to $\phi = 15°$

2.2 Offloading

All nodes, except wheelcontrol, odometry, and lidar, are offloaded to the laptop to decrease the CPU requirements on the RPi. Currently, the RPi is utilizing approximately 120% of the CPU, with 60% allocated to both wheelcontrol and odometry.

We have decided to run the other nodes on Ubuntu instead of a virtual machine (VM) due to the VM running sluggishly and consuming all the available CPU power.

3 Visuals

In this subsection, we discuss about the new visuals added in addition to existing ones such as lidar scans, pose estimate, path trail. Some of them are listed below:

- RRT Path Waypoints: We basically dynamically change the color of the markers (which represent the waypoints) once they are reached for clarity.

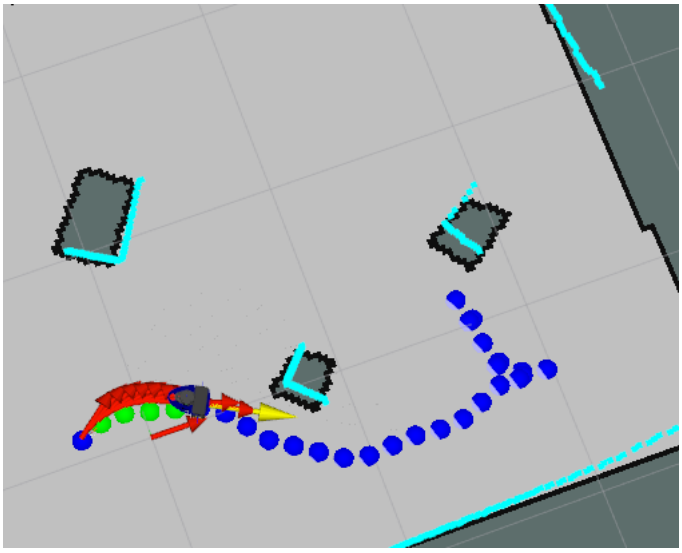


Figure 5: Markers in RVIZ indicating Waypoints. Note that, green markers are the ones the robot has already reached.

- Mapping Visuals:

  While mapping (1m radius) we consider adding and forgetting (useful in case of dynamic obstacles) new obstacles.

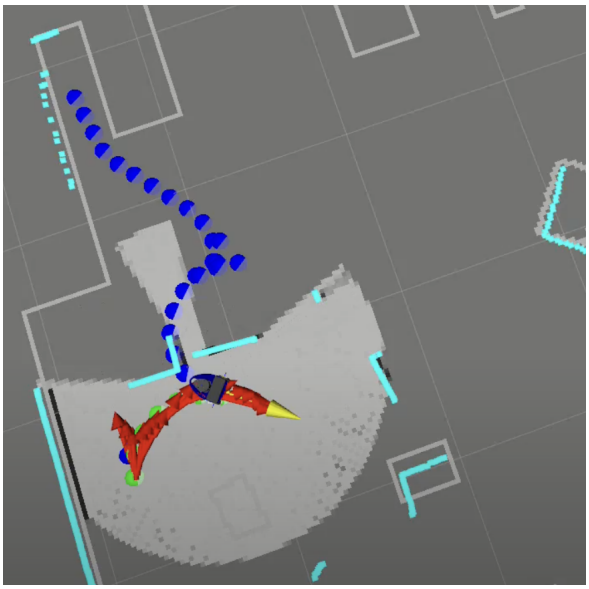


Figure 6: Mapping in action. New obstacles are added as black lines in the obstacle map.

- Particle Filter :

  We show the pose of the particles in the process of global (monte-carlo) localization. Also we vary the colour the arrows to indicate their scores.

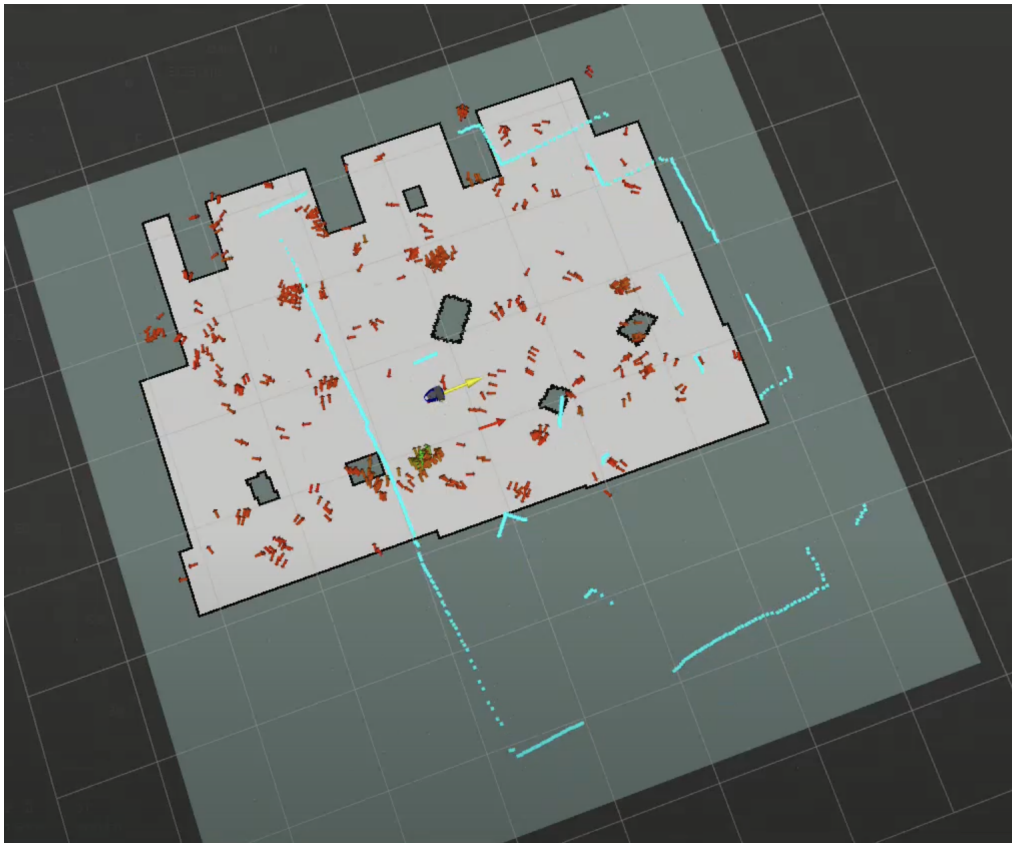


Figure 7: Particle Filter in Action. Note that, near the actual location of the robot, there are more “green" particles (with high scores).

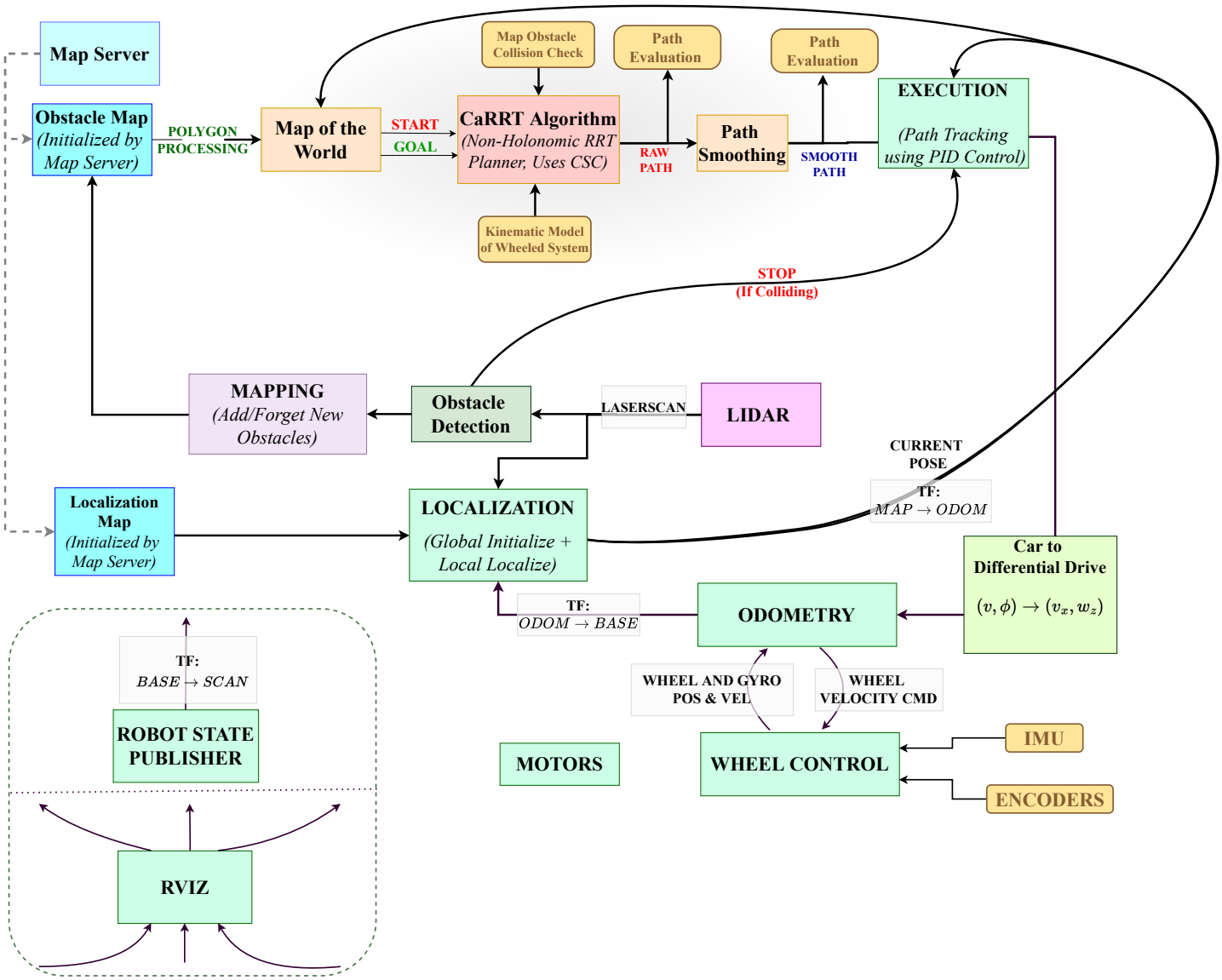
4 Applications

We can use this project as a motivation and further improve its performance so that it can be used in a wide range of applications. For example, it can very useful space/planetary exploration rover that has a roughly initial map but has to map new obstacles and plan dynamically around them. It can be very useful in search and rescue operations of an known location (map). Moreover, the robot could be used for a variety of tasks on a farm, such as moving supplies or monitoring crop growth. The initial map and new obstacle mapping could be useful in these changing environments.

II Methodology

1 High Level Block Diagram

The High Level Block Diagram of the entire approach is as follows:



2 Planning

We plan to reuse some of the “blocks" designed during 133b final project for this project. We briefly review them in this subsection.

2.1 Single Tree CaRRT Algorithm

A high level RRT algorithm for one tree is presented as follows:

Algorithm 1: CaRRT Algorithm for one tree (High Level)

Data: START, GOAL, MAP
Result: PATH
Define Parameters r, p, TOL;
TREE = [START] ;
while *TRUE* **do**
  q_t = Get_Target(GOAL, r) ; // Goal Biased Random Sampling
  q_{nt} = NearestNode_TREE(q_t,TREE) ; // Uses CSC Distance
  q_{next} = NextNode_Integrate(q_{nt},q_t) ; // Uses CSC Distance and avoids collisions
  add2tree(q_{next}) ;
  d = Euclid_Distance(GOAL, q_{next}) ;
  r = $p \times d$;
  if d <*TOL* **then**
    break
  end
end
PATH = Get_Path(q_{next}, TREE) ;
Return PATH

Figure 8: A still of the single tree growing

2.2 Two Tree CaRRT Algorithm

A high level RRT algorithm for two trees is presented as follows:

Algorithm 2: Modified CaRRT Algorithm with Two Trees (High Level)

Data: START, GOAL, MAP
Result: PATH
Define Parameters TOL;
TREE1 = [START] ;
TREE2 = [GOAL] ;
while *TRUE* **do**
  q_t = Get_Target() ; // Totally Random Sampling
  q_{nt} = NearestNode_TREE(q_t,TREE1) ; // Uses CSC Distance
  $q_{next,1}$ = NextNode_Integrate(q_{nt},q_t) ; // Uses CSC Distance and avoids collisions
  add2tree($q_{next,1}$, TREE1) ;
  q_t = $q_{next,1}$; // set next node as the target for another tree
  q_{nt} = NearestNode_TREE(q_t,TREE2) ; // Uses CSC Distance
  $q_{next,2}$ = NextNode_Integrate(q_{nt},q_t) ; // Uses CSC Distance and avoids collisions
  add2tree($q_{next,2}$, TREE2) ;
  d = Euclid_Distance($q_{next,1}$, $q_{next,2}$) ;
  if d <TOL **then**
    break
  end
  SWAP(TREE1, TREE2);
end
PATH1 = Get_Path($q_{next,1}$, TREE1) ;
PATH2 = Get_Path($q_{next,2}$, TREE2) ;
PATH = MERGE(PATH1, PATH2) ;
Return PATH

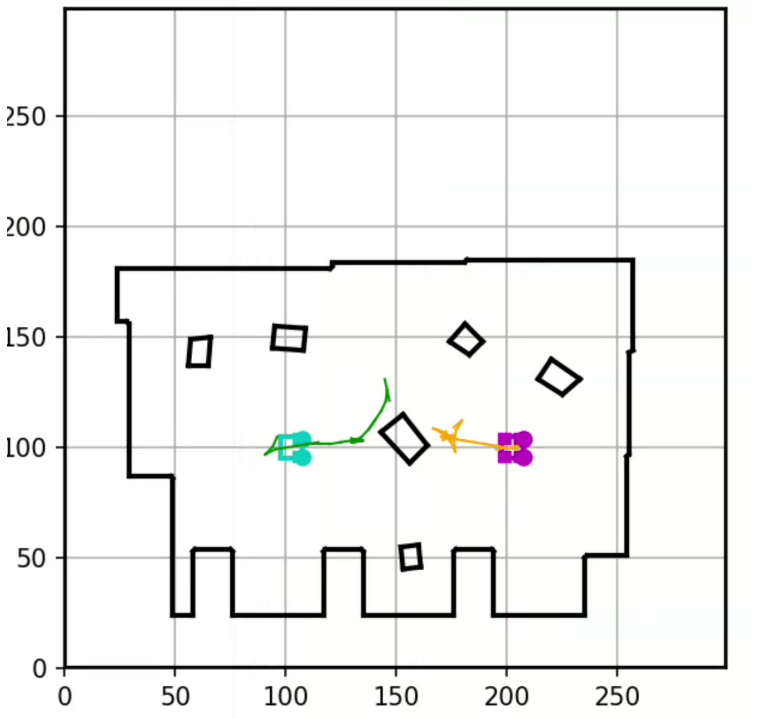


Figure 9: A still of the two trees growing (one from START and other from GOAL)

2.3 Smoothing Function

Algorithm 3: Smoothing Function for Non-Holonomic Mobile Robots

Function `PostProcess_smooth`(*PATH*)**:**
  if *len(PATH) <= 2* **then**
    Return PATH
  end
  NEW_PATH = [PATH[0]] ;
  while *len(NEW_PATH) < len(PATH)* **do**
    q_{next} = NextNode_Integrate(NEW_PATH[-1], PATH[-1]) ; `// Uses CSC Distance and avoids collisions`
    NEW_PATH.append(q_{next});
    if *Euclid_Distance(PATH[-1],* q_{next}*) < TOL* **then**
      Return NEW_PATH
    end
  end
  PATH1, PATH2 = SPLIT(PATH);
  PATH1 = `PostProcess_smooth`(PATH1);
  PATH2 = `PostProcess_smooth`(PATH2);
  NEW_PATH = MERGE(PATH1,PATH2);
  Return NEW_PATH
end

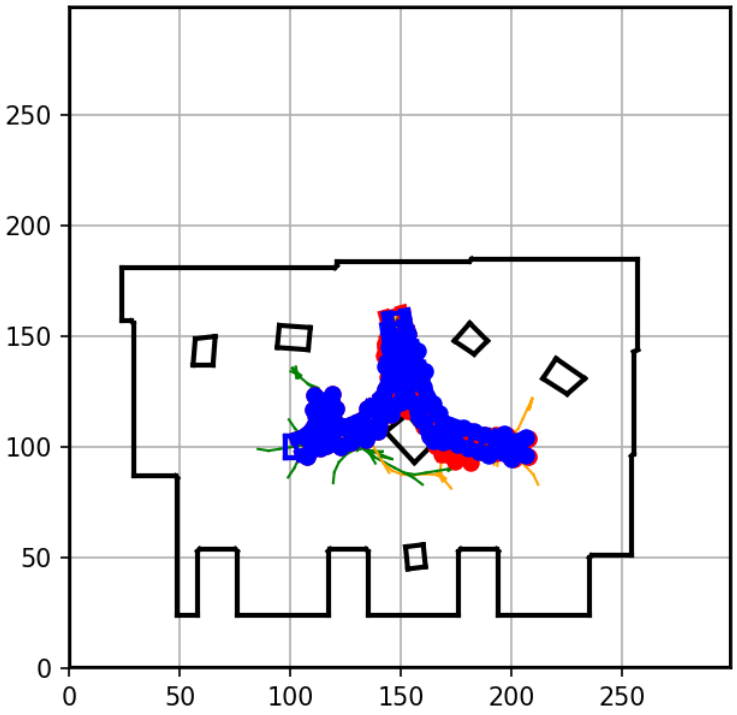


Figure 10: A Minimal example of Path Smoothing

2.4 Quantitative Metrics

We are planning to consider the following metrics to analyze the path found:

1. # Nodes Sampled : We record the number of nodes sampled as a part of the algorithm.
2. Tree Size : We record the number of nodes that are added to the tree, which is the size of the tree.
3. # Nodes in Path : We determine number of nodes that are a part of the path found (Raw or Smooth).
4. Path Length: We determine distance travelled by the mobile robot while traversing from the START to GOAL. This includes the both the curved segments and the straight line segments. For straight line segments the length of the segment is the 2D Euclidean distance between the (x, y) coordinates of the end nodes of the segment. For curved segments, the length of the segment is:

$$L_{curve} = \left| \frac{L}{\tan\phi} \left(\theta_0^A - \theta_0^B \right) \right|$$

   where θ_0^A and θ_0^B correspond to the orientations of end nodes of the segment.

5. Path Smoothness : The overall smoothness of the path. A score close to zero indicate smoother paths. We consider that straight line paths have a score of zero ($\phi = 0$ and $R = \frac{L}{\tan\phi} \to \infty$) and paths with larger radius of curvature ($R = \frac{L}{\tan\phi}$) to have lower score. Therefore, the final metric can be calculated as follows:

$$\text{Path Smoothness} = \sum_{segments} \left(\frac{L}{R_{segment}} \right)^2 = \sum_{segments} \tan^2 \left(\phi_{segment} \right)$$

6. Planning Time (in s) : We record the time taken to find a path frmo START to GOAL.
6. Smoothing Time (in s) : We record the time taken to find postprocess the raw path to get the smooth path using the smoothing function.
6. Outcome : If the planner cannot find a path, the outcome is FAILURE, otherwise it is SUCCESS.

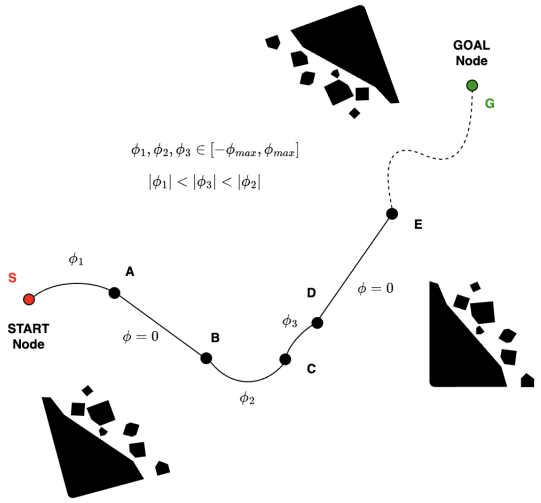


Figure 11: Defining a Typical Path. Here, AB & DE are some straight line segments and SA, BC & CD are some curved segments of the path.

2.5 Some Modifications

In order to make the trajectory "smoother" and more feasible we used the following values:

- $d\_step = 12.7$ cm
- $\phi_{traj} \in \{-18°, 0°, 18°\}$

3 Path Tracking

After a path is found and is post processed, we want the movement between nodes to be as smooth as possible. We basically assigned the velocity profile to the list of waypoints for the low level to process by keeping in mind the following :

- If the steering angle don't change too much and the car keeps moving forward/backward, the car should increase the speed until it hits a saturation limit (V_{MAX}) and then drives with that constant speed.
- Otherwise, the car should slowly slow down.

After the car reaches a waypoint, we record the various errors (described later) between the current position and the next waypoint

After the car reach one node, it should record the error between car and the node and use that information as feedback to make sure the car drive one the path.

3.1 "Reaching" a Waypoint

Since the car is driving under a continuous, non-zero speed, the closest distance for the car to a node would happens at the moment the car drive pass the horizontal line of the node (Or the horizontal line of the car drive pass the node). We check this by the following equation

$$isFront = (\vec{P}_c - \vec{P}_n) \cdot (cos(\theta_n), sin(\theta_n))$$
$$hasReached = (v \times isFront > 0)$$

At that moment, we consider the car has reach the node and we request the position of next node and the corresponding velocity and steering angle. We also record the error of the position and orientation between the car and the node to adjust next steering angle with PID control.

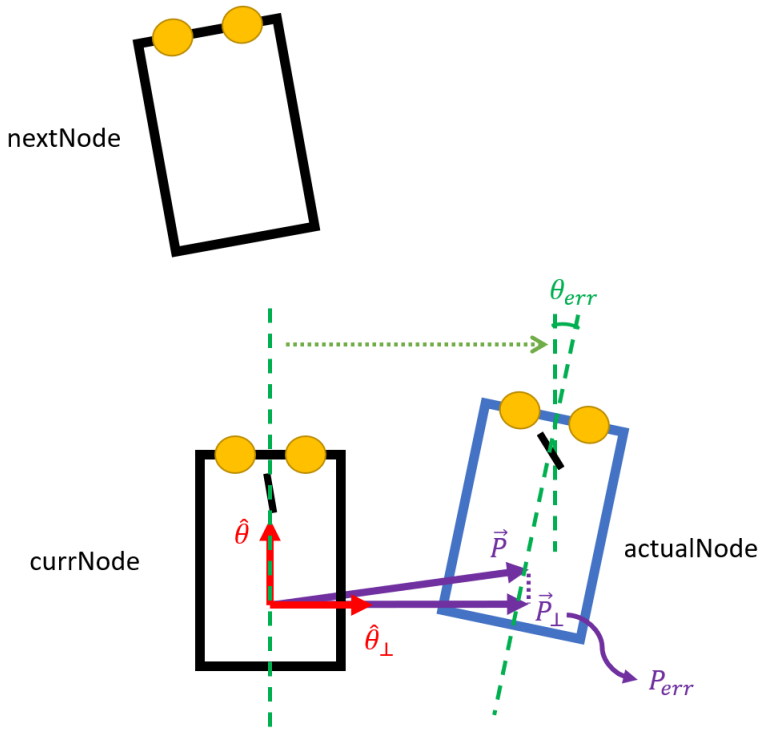


Figure 12: Illustration of Position and Orientation Error

3.2 Projection Error

We first consider the error as the horizontal distance between the node and the car. It can be calculated by:

$$proj\_err = (\vec{P}_c - \vec{P}_n) \cdot (sin(\theta_n), -cos(\theta_n))$$

3.3 Orientation Error

We also consider the mismatch in the orientation between the node and the car so that it can be corrected via feedback.

$$orient\_err = wrap180(\theta_n - \theta_c)$$

3.4 PID Controller

We can treat both errors as a error vector as follows:

$$e[n] = \begin{bmatrix} proj\_error \\ orient\_error \end{bmatrix}$$

Then we consider the following equation for the PID controller for correcting ϕ, steering angle:

$$\phi = sat(\phi_{nom} + [K_P^T e[n] + K_D^T (e[n] - e[n-1]) + K_I^T \Sigma e[n]] sign(v), \frac{\pi}{3})$$

where ϕ is the nominal steering angle obtained from the RRT planner and $K_I, K_P, K_D \in \mathbb{R}^2$.

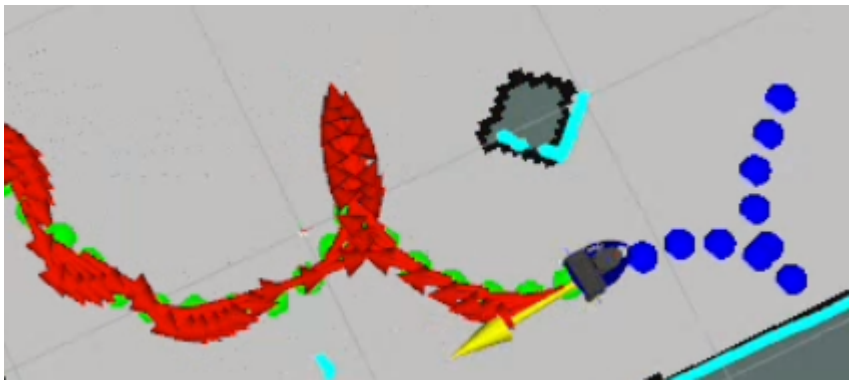


Figure 13: Robot Following a Path Found using the mentioned Path Tracking scheme.

4 Global Localization (Particle Filter)

4.1 Description

We utilize the particle filter algorithm to achieve global localization for the car on the map. Initially, we randomly initialize 500 nodes throughout the map. Each node's score is determined by the degree of match between the scan points and the surrounding walls. Subsequently, we employ local localization techniques to shift the nodes to more favorable positions within their local surroundings. Finally, we resample the nodes, taking into account their scores and updated locations. Nodes with higher scores are given a greater probability of selection. Additionally, we introduce some noise to the nodes after resampling, allowing for further exploration of the map. This entire process is repeated 10 times, and the node with the highest score is ultimately chosen as the final decision for localization.

We are running both local localize and global localize (particle filter) at the same time in different threads, and we subscript to the score of local localize, thus if we manually send a initial estimate location and get a high score, then we can break the particle filter without running the entire iteration.

It is worth noting that for the local localization function we use in both function, we set the scan point weight to r, the distance from the bot to the scan point, because we think when a obstacle is near the bot, their will be a lot of scan points falls on it, so for each scan point we don't want them to have a high score. Besides, when we are playing with obstacle mapping, usually there would be a new obstacle show up near the bot that is actually not in the localization map, and the local localization will try to map it towards the nearest wall if their weight is too high, thus we should actually rely more on the obstacles that is far away from the bot, such as the walls.

4.2 Algorithmic View

Algorithm 4: Particle Filter Localization

```
Define Parameters N, T ;
Function particleFilter ():
    nodes = initializeNodes(N);
    for iter = 0 to T-1 do
        scores = computeScores(nodes)
        nodes = localLocalization(nodes);
        nodes = resample(nodes, scores) + noise();
        nodes → visualize();
    end
    map2odom = getHighestScore();
```

5 Obstacle Mapping and Dynamic Replanning

5.1 Mapping Strategy

When an obstacle appears on the path, the robot stops and initiates the rebuilding of the map for replanning. In order to update the information within the initially empty map called "Obstacles_map," the robot focuses solely on the region within a one-meter radius of its current position, as this area holds the most significance for remapping. To determine the impact of a scan point to the Obstacles_map, we define the distance from a scan point to the robot to be r, and update the log odds ratio by the logic outlined below:

- If r<1, for the pixel corresponding to the scan point, the log odds ratio is incremented by Δl_{inc}.
- If r<1, we consider the space between scan points and the bot, otherwise, we consider the space within a one-meter radius, the log odds ratio of these spaces should be decreased. To determine which pixels lie on the line between the scan point and the robot, the Bresenham's Line Algorithm is employed. The log odds ratio of the corresponding pixels is then decreased by Δl_{dec}.
- if the absolute value of log odds ratio exceeds a certain threshold, it is saturated to that threshold to avoid overconfidence in the accuracy of the Obstacles_map..

Once 100 sets of scan messages are received, it is assumed that the remapping process is complete. At this stage, two updated maps are published: the Combined_map for visualization and the Obstacles_map for RRT. These maps will be discussed in further detail later.

Considering that information about obstacles is more crucial than empty space, the original map and the Obstacles_map are combined by taking the maximum values between two maps for each pixel. This approach ensures that obstacles present in only one map will still be visible in the Combined_map. Once the Combined_map is updated, it is published to RVIZ for map visualization.

5.2 Replanning Strategy

The log odds ratio of the Obstacles_map is converted to occupancy probability and transmitted to the RRT node. Within the RRT node, a wall threshold of 0.58 is set, meaning any points with an occupancy probability above 0.58 are considered as walls. The HoughLinesP function from OpenCV is utilized to detect linear walls. It is worth noting that the minimum acceptable line length is set to 5 pixels, ensuring that only obstacles with a length of at least 5 inches are considered. The detected walls are then integrated into the RRT function. Subsequently, the callback function for /goal_pose is invoked, assuming a new goal_pose has been received, thereby triggering the replanning process with updated walls.

5.3 Algorithmic View

We present an algorithmic view of the current Obstacle Mapping and Replanning strategy being used as follows:

Algorithm 5: Obstacle Mapping and Dynamic Replanning

```
Define Parameters linc, ldec, WALL_THRESH, MAP, LSAT ;
while TRUE do
    if LIDAR Detects Collision then
        STOP → wheel_control;
        COUNT = 0 ;                                                    // Begin Mapping
        LOG_ODDS = 0;
        while COUNT< 100 do
            COUNT += 1;
            if r < 1 metre then
                LOG_ODDS += sat(linc,LSAT);          // LIDAR sees an Obstacle within 1m radius
                LOG_ODDS -= sat(ldec,-LSAT);                  // LIDAR doesn't see an Obstacle
            else
                LOG_ODDS -= sat(ldec,-LSAT);                  // LIDAR doesn't see an Obstacle
            end
            OBS_MAP = log2prob(LOG_ODDS);
            COMBINED = combineMaps(MAP, OBS_MAP) ;
            COMBINED → PublishMap();
        end
        NEW_WALLS = LineDetect(OBS_MAP, WALL_THRESH) ;        // Uses OpenCV's HoughLinesP
        WALLS = concat(WALLS, NEW_WALLS);
        WALLS → RRT_PlanSmooth();
    end
end
```

6 Further Finetuning!

6.1 Proximity in RRT

To facilitate for finding paths that are not too close to the known obstacles we modified the CaRRT algorithm to use a proximity parameter. This makes sure that all the nodes in the path have a certain clearance distance from the obstacles.

6.2 Noise to fix START and GOAL

Sometimes (especially in narrow spaces) where the localization is a little bit off, the robot thinks that the GOAL or START might be in a wall or obstacle, so to handle this we perturb the START and GOAL by adding small random noises so that this problem doesn't arise.

6.3 STOP and Leave Obstacle Strategy

We experimented with a few strategies for mapping and collision detection. We found that the following strategy works the best for our use case:

- Whenever a new obstacle is found, we stop and make a move to the previous waypoint in the path. Since the waypoints are close, we can safely assume that a new obstacle may not have showed up.
- Then we make sure that the motion is complete by waiting for a certain period of time (5 seconds) and then we start remapping. We found that this wait helps stabilize the localization too and hence led to better mapping.

6.4 Weighting Least Squares Local Localization

We interesting found that when the robot is in a narrow space of new obstacles, such as being surrounded by new "walls", weighting the distant far points more than close scan points improves the localization significantly, helping it converge faster and in a more accurate way.

6.5 Adding LIDAR Hats to Robots

Since all the robots are shorter than what the LIDAR can see we decided to add a LIDAR “hat" and have seen that many other teams have done so too. This can help them “see” each other to avoid collision.

III Qualitative Results and Analyses

1 Notable Qualitative Results (with Inferences)

We ran a number of experiments in various scenarios and our method was successfully able to complete the task. We discuss some of them here briefly* as follows:

- ***Single Tree CaRRT vs Two Tree CaRRT*** :

  We compared the Single Tree version of the CaRRT algorithm and found that it is much slower than the two tree version, as we did for some of the cases in the 133b Final Project. This might be because of the bigger size of the map (as compared to 133b). So, we decided to use the two tree CaRRT for all subsequent experiments.

- ***Smoothing vs No Smoothing*** :

  We tried to see if the smoothing helps in making the path efficient and smooth. We observed that this is indeed the case. The raw path is sometimes unnecessarily complex due to the random nature of the algorithm but the smoothing function simplifies it significantly. So, we decided to use the smoothing function for subsequent experiments.

- ***Particle Filter*** :

  We placed the robot at different locations in the map such as near the Box #2, in free space, in the second desk space, between Box #3 & Box #4 in the presence of people sitting (challenging because of unexpected lidar points). Then we ran the Particle Filter and it was successfully able to “globally” localize itself exactly in the map. The addition of local localization into the particle filter and definition of the score function seemed to play an important role in this.

- ***Narrrow Garage*** :

  We placed the robot at different locations in the map and tried to drive the bot inside the narrow garage. In almost all the cases, the robot successfully finds a path fairly quickly and enters the narrow space without any collisions. The success of this can be attributed to the Two Tree CaRRT algorithm, smoothing function and the PID path tracking. Sometimes, if the localization is slightly off or if the path found is close to the wall, the collision detection is activated and the robot starts remapping and most of the times finds an alternative path to enter the narrow garage.

- ***Desk Parking Lot*** :

  We tried to drive the robot out of the narrow garage several times directly into the garage in the third desk parking lot. The robot successfully was able to accomplish this task with a good path and no collisions. This is again due to the use of two tree CaRRT algorithm, which enable finding paths from narrow spaces quickly.

- ***Adding, Forgetting Obstacles + Mapping + Replanning*** :

  We performed a few dynamic obstacle experiments in the following manner:

  (i) We first ask the planner to plan a path to any location.
  (ii) We placed a new obstacle in its way. This forces the robot to stop and remap.
  (iii) During remapping we can see that the new obstacle gets added to the map.
  (iv) After remapping, the planner finds a new path to the GOAL around the obstacle.
  (v) Then, we move the previous obstacle, hence making it dynamic, in the way of the new path.

*The Final Video has demonstrations, RVIZ & Real, showing the robot succeed in all the scenarios described

(vi) Then the robot remaps again. Now, we can clearly see the robot add the obstacle's new position and forget the previous location, which is the desired behaviour.

(vii) Finally, it finds a path around it again and goes to the GOAL.

We found that the robot is successfully able to execute its task even in the case of unseen dynamic obstacles, which is the main focus of this project. We can attribute this success mostly to the Remapping strategy, STOP and Leave Obstacle Strategy and the collision detection function.

- ***Avoiding Collisions with Other Robots*** :

  We found that our robot can clearly see and plan around the other robots (especially, if they have "LIDAR hats"). This is a good thing and allows unnecessary collisions.

- ***Parallel Park*** :

  We also tried a few small parallel park (though it is not main goal of the project) experiments. The success rate was not as high as compared to other tasks but the car was still able to parallel park (as shown in the video).

2 General Failure Cases

Here, we try to outline some of the situations where our described methods does not work very well:

- *Monte Carlo Localization in Narrow Garage*– We found when the robot is inside the Narrow Garage, the particle filter sometimes fails to localize perfectly. This is mostly because of the very few "good" scan points.
- *Remapping+Collision Detection in Narrow Spaces*– We had designed the Remapping and Collision detection strategy by keeping in mind the Dynamic Obstacle task. We found that this may not be necessarily the best when we want to parallel park in narrow spaces with "new" obstacles and walls.

3 Lessons Learnt, Advice

We learnt a lot of important lessons through this project and we try to list a few of them here as follows:

- Use of Two Trees in RRT algorithm speeds up the planning especially in large maps and for START & GOAL nodes in narrow spaces.
- Use of Smoothing function, post-processing that respects the non-holonomic constraints of the robot is very helpful for removing unnecessary complexities in the found path, thereby improving the efficiency.
- In Monte Carlo Localization (Global) using Particle Filters, the use of local localization improves the accuracy of the finding the correct pose.
- For effective Path Tracking using PID, it is important to consider both position error and orientation error.
- For Mapping, it is important to set a lower and upper saturation limit while incrementing and decrementing Log Odds so that objects can be added/forgotten conveniently. This helps us for dealing with dynamic obstacles.
- The STOP and Leave Obstacle strategy once a collision is detected has many advantages such as giving time for localization to converge, better mapping and easy (also safe) planning.
- In "known" (in map) narrow spaces, the localization can be slightly off, so the START/GOAL may look like they are in the walls even if they aren't. So, perturbing them with noise works really well.
- In "unknown" (new obstacles) narrow spaces, the localization will most likely be off and but this can improved significantly, if we use weighted least squares giving more importance to far away scan points.

IV Conclusion

In conclusion, the presented project showcases the application of a Non Holonomic RRT planner with the ability to map new obstacles (static and dynamic) and dynamically replan. The Particle Filter for global localization and tracking techniques are also major parts. The Particle Filter algorithm enables the car to accurately determine its position on a map by using scan points and resampling nodes based on their scores. The project also includes path planning algorithms, such as the CaRRT algorithm for single and two trees, and a smoothing function to optimize the generated paths. Path tracking is achieved through a PID controller that adjusts the steering angle based on projection and orientation errors. The project further incorporates obstacle mapping and dynamic replanning strategies, where obstacles are detected, mapped, and integrated into the path planning process. Overall, this project demonstrates the potential of these techniques for various applications, including space exploration, search and rescue operations, and farm automation.

The project's methodology encompasses a high-level block diagram that illustrates the different components and their interactions. The diagram includes modules for mapping, path evaluation, collision checking, localization, and execution, among others. The planning phase involves the utilization of the CaRRT algorithm for both single and two trees, allowing for efficient path generation. Additionally, a smoothing function is applied to optimize the paths by iteratively adding new nodes. The methodology also defines quantitative metrics to evaluate the performance of the path planning algorithm, including the number of nodes sampled, tree size, path length, and path smoothness. These metrics provide valuable insights into the efficiency and effectiveness of the algorithm.

The project highlights the importance of accurate path tracking for smooth movement along the planned paths. A detailed explanation of the path tracking process is provided, which involves reaching waypoints, calculating projection and orientation errors, and utilizing a PID controller to adjust the steering angle. By continuously monitoring the errors and making adjustments, the car can maintain precise alignment with the planned path. This ensures smooth and accurate navigation towards the goal. Furthermore, the integration of global localization using a Particle Filter enhances the overall reliability and robustness of the system. By iteratively updating the particle positions based on scan points and resampling, the car can accurately determine its position on the map even in the presence of uncertainties and obstacles.

We would like to acknowledge Professor Günter Niemeyer for his timely advice and support in general. We also would like to thank Lorenzo for many insightful discussions that led to clearing major roadbloks. We also would like to thank Subrahmanya V. Bhide (MS Space Engineering, GALCIT) for helping us design the PID controller for Path Tracking.

We have tagged the final version of our code in the GITLAB repository.

The drive link for the accompanying final video is as follows:

`https://drive.google.com/file/d/1Kte4UpX1izOa7Q48W0lxmmOBrEmmHyp1/view?usp=sharing`