Algorithmic Robot Motion Planning - Final Project Report
Topic: Motion Planning for the F1Tenth Autonomous Small-Scale Racecar

Student: Yotam Granov
E-mail: yotam.g@campus.technion.ac.il

Semester: Winter 2022/23
Course Number: 236901

Abstract

In this work, we will investigate the implementation of motion planning algorithms on an autonomous small-scale competitive racecar being developed under the F1Tenth framework. The aim of the work is the successful implementation of two such algorithms, namely Follow-The-Gap and Pure Pursuit, on the physical car (provided by the University of Pennsylvania), along with an investigation into the implementation of more advanced motion planning algorithms such as Rapidly-Exploring Random Trees (RRT).

Figure 1: Stock photo of an F1Tenth car

Contents

1 Introduction

1.1 The F1Tenth Organization

F1Tenth is an international community of researchers, engineers, and autonomous systems enthusiasts founded at the University of Pennsylvania in 2016 [4]. F1Tenth's mission is to provide an open-source platform for autonomous systems research and education, along with holding a number of autonomous race car competitions each year where teams from all around the world gather to compete.

The Technion F1Tenth Team was established in July 2022 and launched in October 2022, with 4 team members across 4 faculties receiving guidance from Dr. Kiril Solovey of the Technion's Electrical & Computer Engineering Faculty. The team currently has one F1Tenth car ready, and is in the process of building four more. Our current goal is to race our car at the 11th edition of the F1Tenth Grand Prix competition, being held at ICRA 2023 in London in late May. As part of the project, we are aiming to implement advanced motion planning algorithms on the car in order to ensure its optimal performance at the competition, where it will need to race around a previously unknown racetrack in the shortest amount of time possible alone as well as against an adversary.

1.2 The F1Tenth Racecar Platform

The F1Tenth autonomous racing platform is designed by a team led by the Safe Autonomous Systems Lab at the University of Pennsylvania since 2016. The racecar consists of a Traxxas RC car chassis, a Hokuyo UST-10LX (10 meter) planar LiDAR sensor for perception, a Vedder electronic speed controller (VESC) for control and actuation, an NVIDIA Jetson NX computer for computation, and a custom-designed (by UPenn) powerboard for power management and distribution. The Jetson computer runs the Ubuntu 18.04 Linux operating system, and the main framework used for interacting with the car is ROS Melodic.

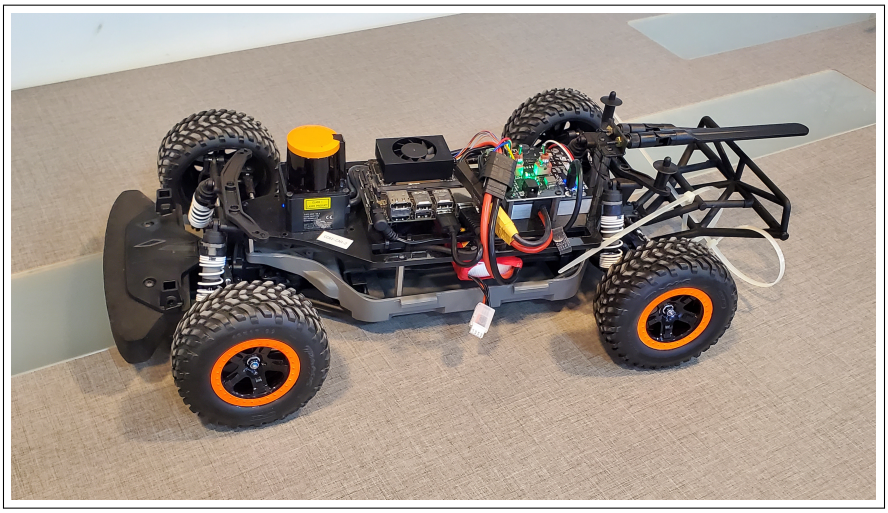

Figure 2: The Technion F1Tenth team's first racecar

The simulation for this project has already been built by the University of Pennsylvania, and it is written using ROS2 Foxy (which will be the main distribution used throughout this project) with a single track (bicycle) model from the Technical University of Munich's CommonRoad software package to simulate a rear wheel drive car with Ackermann steering. The simulation uses RViz for visualization, and its GUI appears as follows:

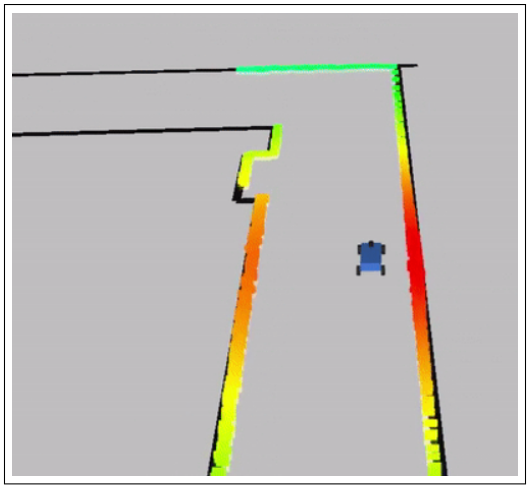

Figure 3: Screenshot from the F1Tenth simulator, prepared by UPenn

The blue box represents the car itself, the grey background represents free space, the black lines mark the boundaries of the racetrack, and the colorful points along the boundaries represent LiDAR data received by the car.

2 The Follow-The-Gap Algorithm

The first algorithm we will discuss is known as Follow-The-Gap. In this approach, the car can locally avoid obstacles (reactive method) without any map, and this allows for surprisingly great single-vehicle racing. The idea of the algorithm is as follows: given an array of distance measurements from the LiDAR sensor (each element corresponds to the angle at which it was scanned), we look for all elements whose value is higher than some threshold t (for example, $5[m]$). We will define a gap to be any subsequence of consecutive elements in the LiDAR data array who all have distance values above the threshold - for example, given the array $[1, 2, 6, 7, 2, 1, 8, 1]$ and a threshold of 5, there are two gaps: $[6, 7]$ and $[8]$. We also define a parameter n, such that any gap we consider must have at least n elements. For example, in Figure (4) we see that we will choose Gap 1 (here we choose the widest gap among all valid gaps).

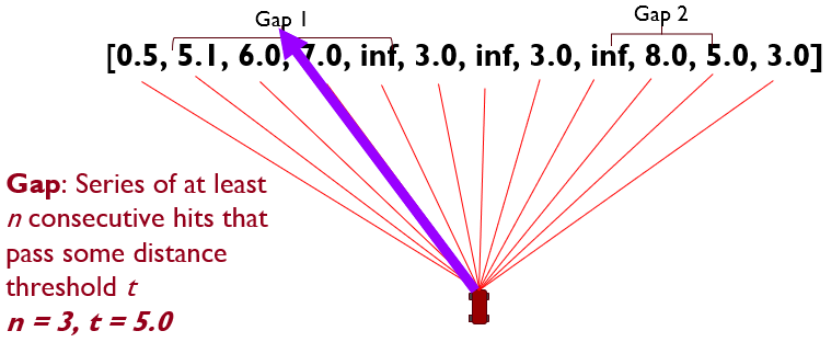

Figure 4: An example of the decision taken by the Follow-The-Gap algorithm

Among such gaps, we will then choose the gap to pursue in an intelligent manner (there are some variants on how to do this, some choose the largest gap and some choose the "deepest" gap), and we will identify some point within this gap to steer towards (usually the middle of the gap). We then set the steering angle δ of the car to be the (negative) scan angle that returned that point in the gap - this method doesn't not directly account for the non-holonomic nature of the car (yet this doesn't seem to affect its performance much). We let the velocity

be linearly dependent on the steering angle:

$$v[m/s] = v_{max} - \alpha \cdot |\delta[^\circ]| \tag{1}$$

where $\alpha = \frac{1}{50}$. We tested the algorithm on the racetrack shown in Figure (5) with threshold $t = 5$ and minimum gap length $n = 3$, and we saw that the car was able to complete at least 10 laps successfully (no wall collisions) up to velocities of $v_{max} = 3.2[m/s]$.

Figure 5: The racetrack setup used for this study

This algorithm performs exceptionally well given its simplistic and local nature - in fact, it was the winning algorithm at the 2019 F1Tenth Montreal Grand Prix! We are interested in more advanced and intelligent approaches to motion planning for our racecar, though, and this leads us to our next algorithm.

2.1 Materials

The code for our implementation (as a ROS Melodic node) of this algorithm can be found in the `reactive_node.py` file, and a video of the algorithm's performance on the real car can be seen in the `FollowTheGap_V=3.2.mp4` file.

3 The Pure Pursuit Algorithm

While the Follow-The-Gap algorithm ignored the richer geometric structure of the racetrack in favor of faster computational speed, the construction of a map of the racetrack and online localization to this map during a race should allow for the implementation of more intelligent motion planning algorithms. One of the simplest such algorithms is called Pure Pursuit [2], where the car pursues a moving target point while localizing itself to a given map (which it can produce itself).

3.1 Algorithm Description

This algorithm is also relatively simple: given a set of waypoints along a racetrack and the ability to localize to the racetrack's map, the pure pursuit algorithm will identify the nearest waypoint to the car at a distance of (at least) L in front of it - this is the main parameter of the algorithm, called the lookahead distance - and then drives towards it along an arc (accounting for the car's non-holonomic nature). Figure (6) shows the choice of a waypoint to pursue, given the car's current location, the lookahead distance L, and the set of waypoints.

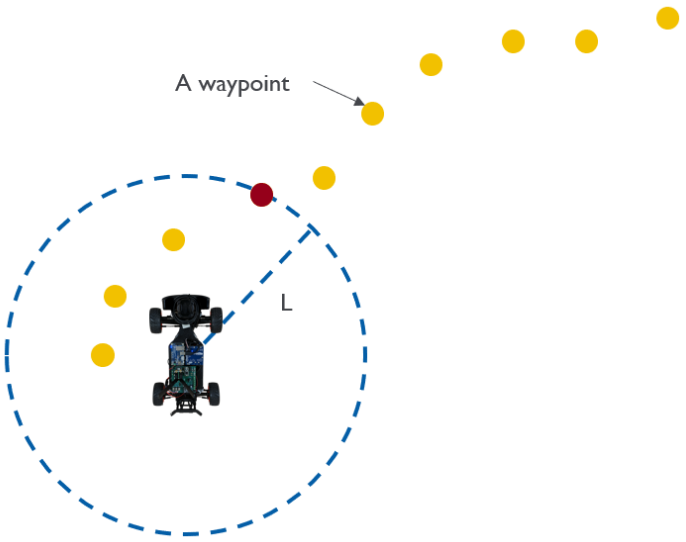

Figure 6: Visualization of the Pure Pursuit algorithm

In order to calculate the necessary steering angle δ to lead the car towards the waypoint along an arc, we will need to first define the arc itself and then obtain its curvature. A diagram of the arc connecting the car to a waypoint located at (x, y) in the car's reference frame is shown in Figure (7).

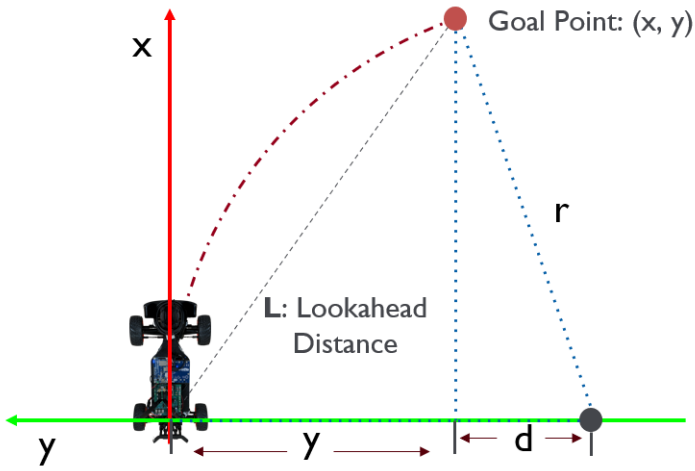

Figure 7: Diagram used for the arc calculations for obtaining the steering angle in Pure Pursuit

From here, we can calculate the radius of curvature of the arc as follows:

$$r[m] = \frac{L^2}{2|y|} \tag{2}$$

The curvature of the arc is just the inverse of this radius:

$$\gamma[rad] = \frac{1}{r} = \frac{2|y|}{L^2} \tag{3}$$

Finally, the required steering angle is obtained by multiplying the curvature of the arc by a proportionality factor k_p (which turns out to be roughly equal to the distance from the front axle to the rear axle of the car -

$0.325[m]$), in what is essentially proportional control:

$$\delta[rad] = k_p \gamma = \frac{2k_p|y|}{L^2} \tag{4}$$

We then let the velocity be linearly dependent on the steering angle (same as for Follow-The-Gap):

$$v[m/s] = v_{max} - \alpha \cdot |\delta[^\circ]| \tag{5}$$

where $\alpha = \frac{1}{50}$. We tested the algorithm on the racetrack shown in Figure (5) with lookahead distance $L = 1.6[m]$, and we saw that the car was able to complete at least 10 laps successfully (no wall collisions) up to velocities of $v_{max} = 2.8[m/s]$. While not yet as fast as the Follow-The-Gap algorithm, we believe that this algorithm (as well as the SLAM needed to run it) can be tuned further to return better performance. One serious drawback of this algorithm is that it relies on waypoints that are created offline, and so it can't avoid dynamic obstacles or opponents. To this end, we hope to implement online waypoint generation using the RRT algorithm (discussed further in Section 4), but first we will elaborate a bit further on the mapping and localization process used in our work.

3.2 Mapping & Localization

To produce the map for an unknown racetrack, we used the Google Cartographer SLAM package [5]. We conduct the SLAM using only LiDAR data, since the inclusion of odometry data seemed to incur large errors that have yet to be well understood on our part (this likely just needs to be tuned). To map the track, we enable Cartographer and drive the car slowly ($0.8[m/s]$) around the track using the Follow-The-Gap node (this method drives quite smoothly even on unknown maps). Cartographer uses the Ceres solver to formulate a nonlinear least-squares correlative scan matching problem [1]. The tuning for Cartographer was quite a challenging process, and we believe that a more methodical approach to it would serve us well for the future (in order to construct higher quality maps). An example of a map constructed by Cartographer for the racetrack from Figure (5) is shown in Figure (8). This map can be found in the files `mp_assignment.pgm` and `mp_assignment.yaml` under the `maps` folder.

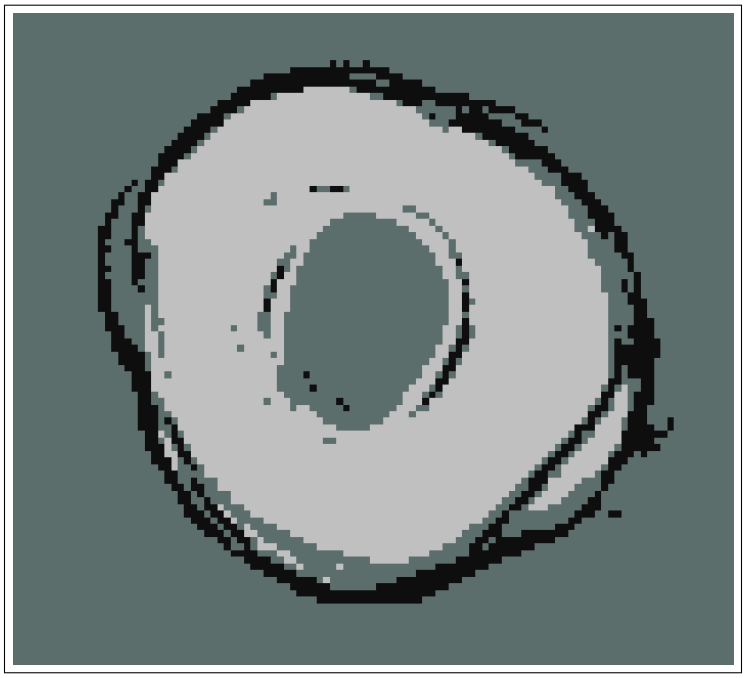

Figure 8: Example map created by Cartographer on our racetrack

We then localize the car according to this map using particle filter (or Monte Carlo) localization, from a library developed at MIT [6]. This library worked quite well for us straight out of the box with minimal tuning required, though improvements can always be made. We used the most simple 2D ray-casting method, Bresenham's Line (BL), but in the future we will try to take advantage of the computational power available in the Jetson computer to use the stronger Ray Marching method on its GPU (RMGPU). In Figure (9), we can see the result of running this particle filter on the car using the map we created previously: the inferred pose of the car is drawn as a blue arrow, the particles are shown as smaller pink arrows, and the yellow points show the matching of the LiDAR scan data to the map.

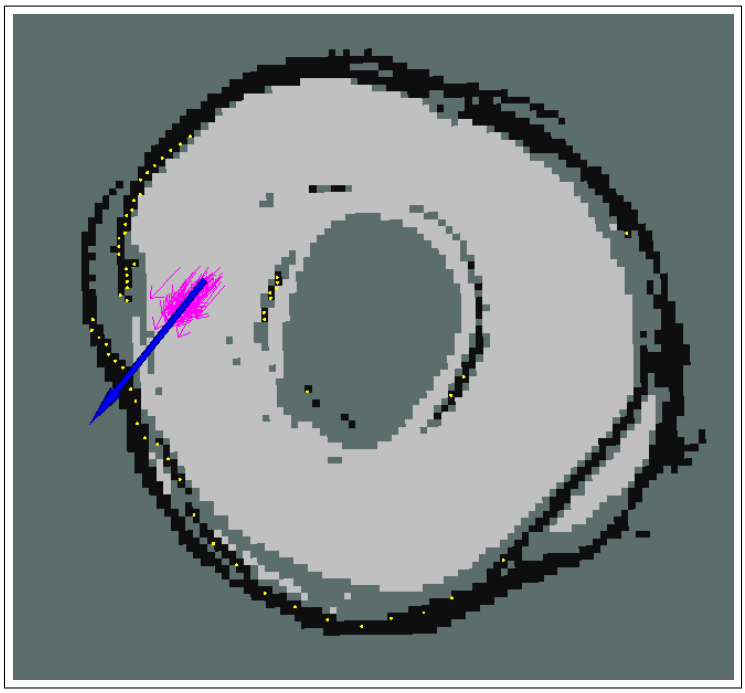

Figure 9: Localization of the car in the given map using the particle filter approach

With the map built and the car localized within it, we then produced the waypoints for the Pure Pursuit algorithm using a waypoint logger script created at the University of Pennsylvania for F1Tenth. While running this script, we allow the car to move slowly around the centerline of the track (again using the Follow-The-Gap node) to produce a series of points along this line (whose positions are obtained thanks to the particle filter). The waypoints are then refined from this set of points, and fed into the Pure Pursuit algorithm as a `.csv` file containing their Cartesian coordinates (in the global reference frame). An example of a set of waypoints produced in this manner for the map from Figure (8) is shown as a set of red points in Figure (10), and the waypoint currently being pursued given the current inferred pose (blue arrow) is marked in green and located (at least) some lookahead distance L ahead of the car. These waypoints can be found in the `mp_assignment.csv` file under the `logs` folder.

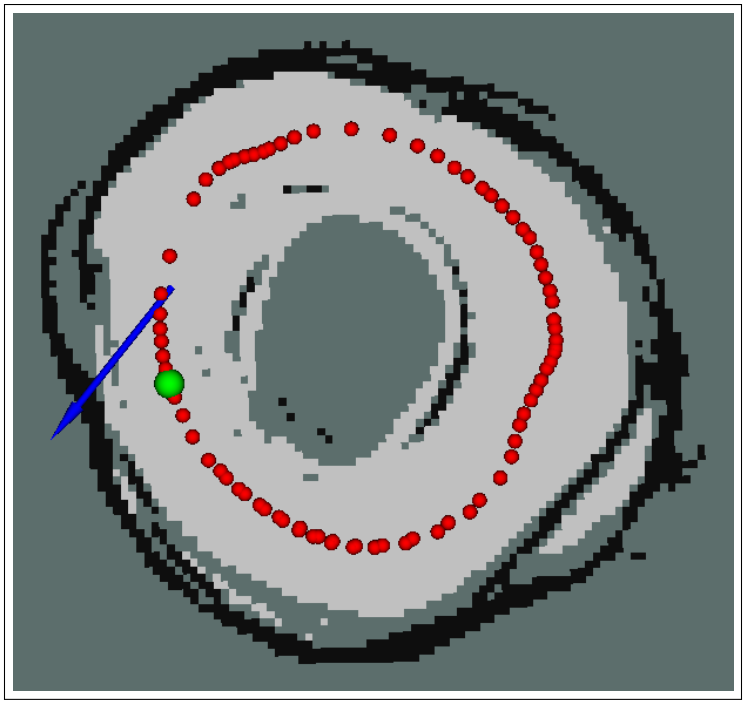

Figure 10: The waypoints (red) fed into the Pure Pursuit algorithm for the given map, and the target waypoint it is currently pursuing (green)

3.3 Materials

The code for our implementation (as a ROS Melodic node) of this algorithm can be found in the `pure_pursuit_node.py` file, and a video of the algorithm's performance on the real car can be seen in the `PurePursuit_L=1.6_V=2.7.mp4` file.

4 The Rapidly-Exploring Random Trees (RRT) Algorithm

While the Follow-The-Gap and Pure Pursuit approaches show promising ability to lead the car smoothly through simple racetracks, it is our goal to combine the online obstacle avoidance capabilities of Follow-The-Gap with the mapping and localization capabilities of Pure Pursuit to produce an algorithm that can intelligently plan the motion of the car along the racetrack given both global and local knowledge of the environment. The Rapidly-Exploring Random Trees (RRT) Algorithm [3] can provide exactly that, as we will generate waypoints using an RRT-generated tree path that can then be fed into a Pure Pursuit motion controller to steer the car safely and quickly through free space - in other words, we use RRT as an intelligent local planner. Unfortunately, our attempts to implement an effective version of RRT on our car have proved unfruitful as of yet, but we will discuss here the approach we are taking and our steps for the future.

4.1 Occupancy Grid

First, we need to choose a way to efficiently detect collisions for sampled nodes and edges in the vicinity of the car. To this end, we implemented an occupancy grid constructor which takes in LiDAR scan data and builds a 25 by 25 cell grid in the car's reference frame (the car is located at the bottom center of the grid) where each cell has a value of either -1 (if the cell has not been explored yet), 0 (if the cell is free), or some value between 0 and 100 (giving the probability that the cell is occupied). Such a grid can be seen in Figure (11).

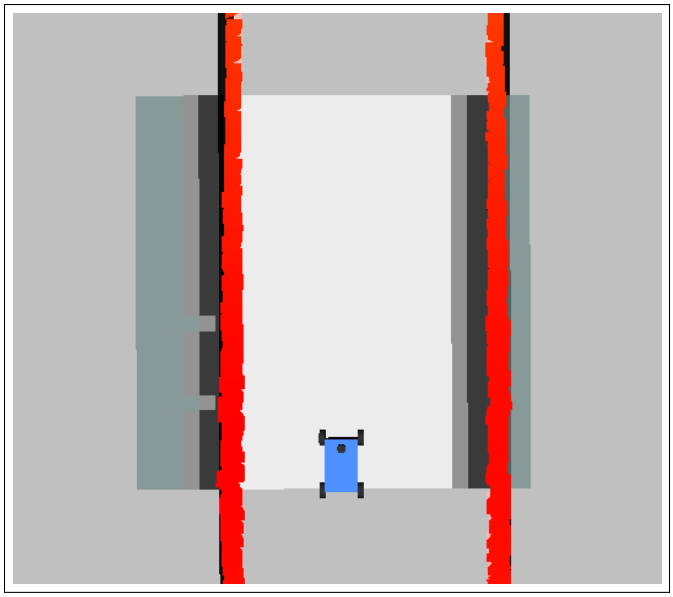

Figure 11: An occupancy grid created in the F1Tenth simulator

The code for this implementation is found under the `scan_callback` function in the `rrt.py` file, and the occupancy grid is rebuilt every time new LiDAR data is received.

4.2 Pure Pursuit for RRT

For now, our implementation involves the construction of an RRT tree in the free space (definde by the occupancy grid), where the goal point(s) is defined by sampled points which are at least some lookahead distance L away from the car. Such points are colored green in Figure (12), while regular samples are colored blue. The path from the car to a goal point is marked in red, and the algorithm will then feed the set of points along this path into the Pure Pursuit algorithm (as the set of input waypoints) in order to control the car's motion (i.e. determine the steering angle and velocity command).

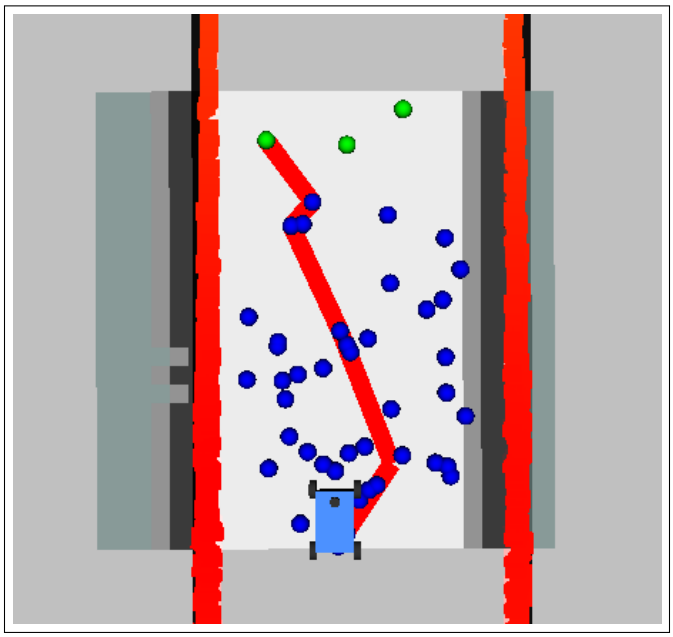

Figure 12: An example RRT tree produced inside the occupancy grid

For now, the edges are represented as straight lines (rather than the more appropriate arcs), and collisions

are detected using only the occupancy grid. The extension mode "E2" (i.e. extending towards sampled point by some step size) has been implemented and works nicely, and in the future the RTT* algorithm will also be implemented to further improve the results. For now, this implementation is not yet able to control the car effectively, and this remains a goal for us to tackle in the coming weeks.

4.3 Materials

See the `RRT` folder for all of our code of the RRT implementation.

References

[1] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 3 2022.

[2] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. 1992.

[3] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[4] Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. In Hugo Jair Escalante and Raia Hadsell, editors, *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, volume 123 of *Proceedings of Machine Learning Research*, pages 77–89. PMLR, 08–14 Dec 2020.

[5] H. Rapp W. Hess, D. Kohler and D. Andor. Real-time loop closure in 2d lidar slam, in robotics and automation. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, page 1271–1278, 2016.

[6] Corey Walsh and Sertac Karaman. Cddt: Fast approximate 2d ray casting for accelerated localization. abs/1705.01167, 2017.