RRT and RRT* Motion Planning for a robot in a Parking Lot Scenario

RO47005 Planning and Decision Making
Group 35
Bas van Vliet, 4594959
H.A. Jekel, 5609593
Y.J.P. le Gars, 5609577
Matti Lang 5632935

***Abstract*— This report presents a sampling-based motion planning to autonomously move a robot along a trajectory in a static environment in a parking lot scenario. Both RRT and RRT* single query path planning algorithms were implemented without the use of a planner module, and investigated. A PD controller allows the robot, represented by a kinematic bicycle model, to follow the computed path. The results suggest that RRT* finds a shorter path in exchange for a higher run time. Future research can extend this work by looking into limited search RRT* algorithms.**

I. INTRODUCTION

Solving the autonomous driving problem is one of the biggest challenges in modern day engineering. In 2018, nearly 1.35 million people in the world life's were lost in traffic accidents [1]. Mobile communication technology is being developed to reduce traffic accidents caused by human errors. Companies like Tesla, Waymo, and Mobileye all have large teams working on this problem, progressing to scalable full autonomous solutions. To ensure safety, an autonomous vehicle must have the ability to identify and avoid obstacles. An autonomous vehicle has a sense layer to collect data about the environment, a perceive layer that interprets the received data, a motion planner layer that creates a path to the goal and a control layer to control the vehicle to follow the path.

This report focuses on the third layer of the autonomous navigation problem: the motion planning of a robot within a given environment map of a parking lot. Two popular approaches for creating a graph through a known environment are search-based algorithms and sampling-based algorithms. Sampling-based algorithms perform better than search-based methods in simple scenarios. The scenario explored in this report is a parking lot with static obstacles and only one actor, which is why this report focuses on sampling-based algorithms. Various sampling-based algorithms exist, RRT is memory efficient as it does not need to keep a discretized map of the environment in memory and it is a lot faster other methods like A*, however it does not find the optimal path. RRT* does a rerouting step in addition to the steps in RRT, which enables it to asymptotically find the optimum, with the downside that it is slower then RRT. State of the art algorithms like informed RRT*, RRT*-Smart, Q-RRT* and F-RRT*, build further on RRT*, focusing on constraining the search area after an initial path, but are more complicated [3].

The team wanted a fundamental understanding of the methods, therefore, it was decided to focus on RRT and RRT*, after which more advanced algorithms can be investigated later. Nonetheless, RRT and RRT* provide an interesting trade-off as aforementioned and therefore makes for an interesting comparison.

The following work has been done by the team. The team has implemented a kinematic bicycle model to show the non-holonomic behavior or the robot and build a PD controller to track the path generated by the path planner. During the planning stage, all objects were given a certain clearance from objects. This way, we could guaranty that the path did not lead to any collisions. Inline with our motivation of getting a fundamental understanding of the algorithms, the team has implemented both RRT and RRT* without the use of a planner module, using only the modules Pygame, math, random, time, os, and matlibplot. The RRT method and visualisation was implemented by following a video tutorial [2]. However, the RRT* method and the bicycle model and controller have been implemented without following any tutorials. Nonetheless, A number of sources were used for understanding various parts of these problems such as: [3], [4] and [5].

II. ROBOT MODEL

A robot moving in a planar workspace, $W \subset R^2$, has a 3-dimensional configuration space defined as $C = R^2 \times S^1$. The motion of the robot can therefore be described with respect to 3 variables (x, y, θ) that are being

subject to rolling and sliding constraints due to the non-holonomic nature of the robot. Furthermore, the scenario requires the robot to drive at low speeds. As a result, the authors chose a simple constant velocity kinematic bicycle model to represent the motion of the vehicle [6] This is one of the simplest ways of representing the motion of the robot under no-slip conditions while still having the non-holonomic constraints in place. This kinematic bicycle model also allows more focus on the path planning and less time on the exact dynamics of the vehicle. Moreover, a PD controller was used to follow the path. While a PD controller gives no guarantees that the robot will not clash into objects, it does provide an effective and simple method to follow the path.

The kinematic bicycle model and controller are given in Equation 1 to Equation 4. Here, the states are the x and y position of the robot and the yaw angle ψ and the input is the steering angle θ

$$\begin{bmatrix}\dot{\psi}\\ \dot{x}\\ \dot{y}\end{bmatrix} = \begin{bmatrix}\frac{v}{L_{front}+L_{rear}}cos(\beta)tan(\theta)\\ v\ cos(\phi+\beta)\\ v\ sin(\phi+\beta)\end{bmatrix} \tag{1}$$

Where v is the forward velocity, L_{rear} and L_{front} are the positions of the rear wheel and the front wheel from the center of the robot respectively, and β the slip angle given by the equation:

$$\beta = tan^{-1}(\frac{L_{rear}}{L_{front}+L_{rear}})tan(\theta) \tag{2}$$

The steering angle θ is given by the PD controller using the error e. The error e is defined as the shortest distance (normal distance) between the point of the robot on the front wheel and the 'current' edge. The front wheel position is chosen instead of the center of the robot because it causes less overshoot at the nodes. The error e is calculated as shown by Figure 1. Only the 'current' (L_1) and the 'next' (L_2) edge are used. These edges are shifted by 1 index if e becomes e_2. Then the error is given by $e = min(e_2, e_3)$ and so forth.

$$\theta = P * e + D * \dot{e} \tag{3}$$

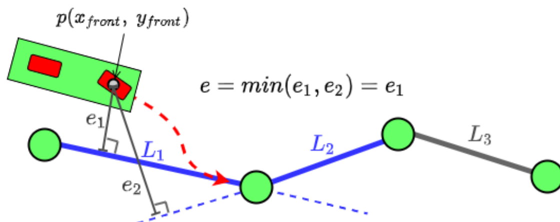

Fig. 1: How the error is calculated for the controller

The bicycle model is then simulated using:

$$\begin{bmatrix}\psi_{t+1}\\ x_{t+1}\\ y_{t+1}\end{bmatrix} = \begin{bmatrix}\psi_t+\dot{\psi}dt\\ x_t+\dot{x}dt\\ y_t+\dot{y}dt\end{bmatrix} \tag{4}$$

III. Map

The authors have decided to use Pygame to simulate the environment. Pygame is a set of Python modules designed for writing video games, simplifying the generation of a map, robot and moving obstacles. A static scenario with objects generated at random locations was used to implement the planning algorithms. Subsequently, a robot parking scenario is generated as the final project. The Pygame visualization is shown in Figure 2.

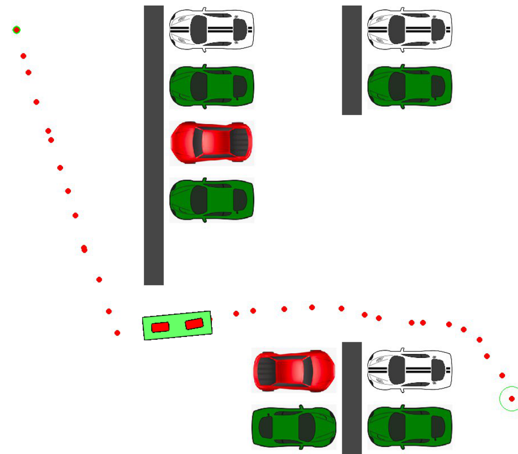
Fig. 2: The map with the bicycle model following a path found using RRT*

During the planning stage of the problem, the bounding boxes are drawn around the parked cars to make sure the path generated has enough clearance around all the objects. This will generally make sure the robot will not crash into these objects.

IV. MOTION PLANNING

The motion planning algorithm will depend on the state of the environment. In other words, moving obstacles will require local planning algorithms, whereas global planning algorithms are best suited for static environments. Due to its complexity, dynamical environments have been discarded and the focus will be on implemented efficient global planning algorithms. Finding a balance between minimizing cost consumption like distance travelled and the run time of the planning algorithm will be a primary goal of this work. The first step in the selection of the appropriate motion planning algorithm for our robot model is to choose between single or multi query methods. The difference is that single query methods plan for one specific start and goal configuration whereas multi query methods use the

same graph for multiple start and goal configurations [5].A true autonomous car needs to navigate from an initial state to a goal state without human intervention. It needs to plan its own path and move along this path until it reaches the destination. Since we have selected a car-like robot it is more realistic to assume that our environment is constantly changing and therefore we will focus on single query methods in this report, namely RRT and RRT*. RRT finds a cheaper solution to find a feasible path from point A to point B. In RRT, points are randomly generated and connected to the closest available node [4]. At each iteration the algorithm checks if the vertex and edges are collision free. When the goal is reached, the algorithm backtracks to the start and the shortest path is returned. However, RRT is not optimal whereas A* is. RRT* is an optimized version of RRT. There are two key additions to the algorithm: RRT* records the distance each vertex has traveled relative to its parent vertex and also it allows for rerouting [4]. The process of selecting least cost parent and rerouting tree are the two most promising features of RRT* that contribute to its asymptotic optimal property [8]. In our approach we decided to first implement RRT and then RRT* to easily compare the two algorithms.

This report will introduce the model in Section 2 and explain the map and work/configuration space in Section 3, Then, Section 4 addresses the planning algorithms. Section 5 and 6 show and discuss the results and finally future work are considered.

V. RESULTS

A number of metrics are used to compare RRT and RRT*. In general, we can divide the results into 2 parts, first the metrics on the first path found, and second on the final path found. The list below gives a short description of the metrics that are used to compare the methods. All the metrics are averaged over 10 runs to minimize the significance of outliers.

- Avg iterations until path found - the average number of iterations needed to find the first path
- Avg nr. of vertices until path found - the average number of vertices needed to find the first path
- Number of vertices (-) - the number of vertices that are generated in a fixed number of iterations
- Avg run-time per iteration (s) - the average time needed to complete 1 iteration
- Path length (UoL) - the length of the found path in the local Unit of Length (UoL)
- accumulated error (UoL) - the total sum of errors over the entire path. A metric used as a way to judge how well the robot is able to follow the path
- normalized accumulated error (UoL) - the accumulated error divided by the path length

Metric	RRT	RRT*
Avg iterations until path found	483	515
Avg nr. of vertices until path found	246	249
Avg run-time per iteration (ms)	5.7	5.9
Path length (UoL)	1786	1602

TABLE I: Comparison of RRT and RRT* on various performance metrics for the first path found, averaged over 5 runs

Metric	RRT	RRT*
Avg run-time per iteration (ms)	9.2	12.7
Number of vertices (-)	1883	1854
Path length (UoL)	1775	1331
accumulated error (UoL)	4456	4770
normalized accumulated error (UoL)	2.5	3.6

TABLE II: Comparison of RRT and RRT* on various performance metrics for the final path found in 3000 iterations, averaged over 5 runs

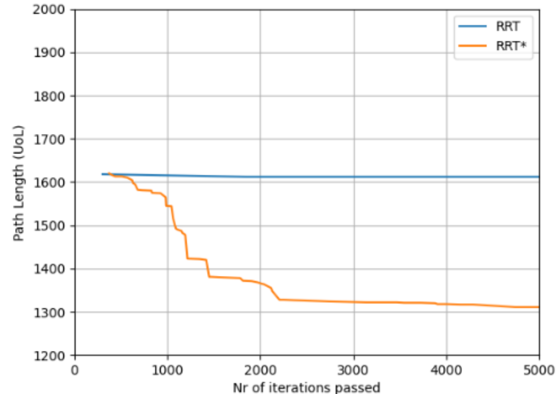

Fig. 3: The path length found using RRT and RRT star over the number of iterations

Figure 3 shows compares the path length of RRT and RRT* with respect to the number of iterations. We can see that the number of iterations does not affect RRT. However, with RRT* the path length reduces sharply until about 2000 iterations. After this threshold, the path length continues to decrease with the number of iterations but only minimally. Figures 4 and 5 show visual results from our Pygame implementation of both RRT and RRT* respectively The obstacles are represented in gray and we added bounding boxes (represented in red) so that our robot model avoids the obstacles with some clearance. In Figure 4 we can see that RRT returns a successful solution from start to goal but clearly does not take the optimal path. In Figure 5 we see that RRT* with 3000 iterations shows an improved solution in comparison to RRT.

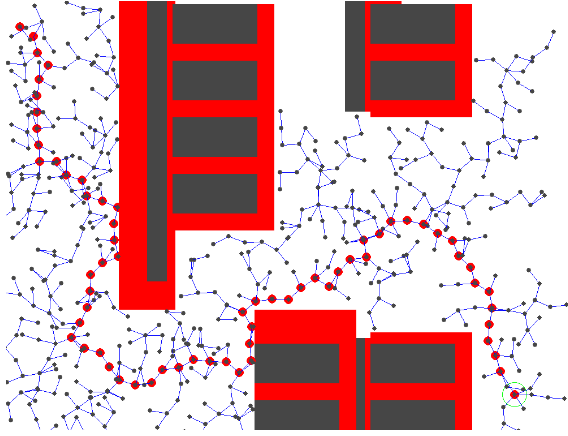

Fig. 4: A path found using RRT.

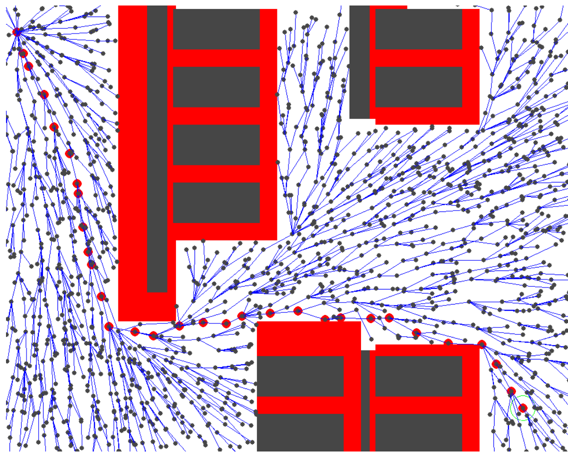

Fig. 5: RRT* - 3000 iterations. note the difference in pattern with RRT and observe visually that the path of RRT* is shorter

VI. DISCUSSION

The results demonstrated that RRT* has better performance over RRT with regards to finding the shortest path. Table I show that the initial path found by RRT* is 11% shorter than that found by RRT. This difference only becomes large when looking at Table I where this increases to nearly 30%. This demonstrates that as iterations increase RRT* keeps reducing the cost. On the other hand RRT* still has a greedier run time explained by its more complex nature. This is especially true as the number of iterations are increased. RRT is probabilistic complete which means that the probability of finding a path is 1 with the running time of the algorithm going to infinity, as long as a solution exists. RRT* in addition, ensures asymptotic optimality. According to [3] RRT* never reaches that optimality in finite time and the rate of convergence is rather slow. Another limitation is that these methods work in the configuration space, provide a path sequence of configurations that the robot needs to be in but they do not account for time. Therefore, moving obstacles are not accounted for and obstacle avoidance methods need to be implemented in addition. The implemented controller is simple and works reasonably well as long as the error stays small. However, if the robot deviates to much of the path, it could switch to the error of the 'next' error even though it should still follow the 'current' error. This especially happens at tight corners, when the turning radius of the robot is too large to follow the path. Interestingly, the average nomalized accumulated error of RRT* was larger then that of RRT, even though the path generated by RRT* is smoother along straight lines. The explanation here is that RRT* makes sharper corners around obstacles, and this path is harder to follow by the PD controller which makes it deviate from the path a lot. This effect could be mitigated by minimizing the length of the robot such that the turning radius is smaller but should be addressed further.

Finally, the authors like to discuss avenues for future work. Derivatives of RRT* have been proposed to address the shortcomings of RRT and RRT* such as RRT*-Smart proposed in [3] which instead of employing a purely random space exploration performs an informed exploration of the search space. Since the authors now have a code base with a large amount of flexibility, it would be an interesting step to extend the current code with such an algorithm. Different paths generators such as Dubins path which connects the path vertices in a smooth fashion can make the path more smooth and improve the path following. Different controllers can be explored such as model predictive control to improve the path following of the robot. Finally, more challenging scenarios can be explored, for example with moving objects and local obstacle avoidance algorithms.

REFERENCES

[1] World health organization. In: *Global Status Report on Road Safety*, (2018).

[2] M. Boumediene, Autonomous car simulated with pyhton. In: *https://www.youtube.com/watch?v=Tllz7Ox2B3gt=381s*, (2021).

[3] J. Nasir, F. Islam, U. Malik, Y. Ayaz, O. Hasan, M. Khan, M. Muhammad, RRT*-SMART: A Rapid Convergence Implementation of RRT*. In: *International Journal of Advanced Robotic Systems, Vol 10, issue 7*, (2013).

[4] T. Chin, Robotic Path Planning: RRT and RRT*. In: *https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378*, 2019.

[5] S. M. LaValle, Planning Algorithms. In: *Cambridge University Press*, (2006).

[6] Y. Ding, Simple Understanding of Kinematic Bicycle Model. In: *https://dingyan89.medium.com/simple-understanding-of-kinematic-bicycle-model-81cac6420357*, (2020).

[7] S. Karaman, E. Frazzoli, ”Sampling-based algorithms for optimal planning”, *Int J Rob Res, vol. 30,pp. 846-894, 2011*.
[8] B. Liao, F. Wan, Y. Hua, R. Ma, S. Zhu, X. Qing, F-RRT*: An improved path planning algorithm with improved initial solution and convergence rate. In: *Expert Systems with Applications, Vol. 184*, 115457-115473, (2021).