ENPM661 Project 5 Phase 3

Justin Albrecht, Brian Bock, Mahmoud Dahmani

Spring 2020

TABLE OF CONTENTS

I. INTRODUCTION

A. Motivation

In 2018, 36,560 people were killed in motor vehicle crashes. Of these fatalities, 10,511 (28.75%) were due to alcohol impairment [1]. 9 out of 10 (94%) serious roadway crashes occur due to human behavior [2], [3]. While there has been a downward trend in traffic fatalities over the last 40 years due to vehicle improvements like airbags, seat belts, and electronic stability control [1], there could be significant further reduction in vehicular fatalities with the integration of computer assisted driving and the phasing out of human drivers in favor of autonomously driven cars. The development of reliable autonomous vehicles is an important area of research in robotics, with vast life-saving potential. This field spans many complex sub-disciplines, often including perception, planning, controls, machine learning, and tremendous data processing, often at high speed and in real time. As ENPM661 is primarily a planning course, our focus will be on a small slice of the planning aspect of autonomous cars.

B. Background

Testing new autonomous vehicle algorithms in the real world can be both expensive and dangerous, especially when human occupied vehicles are nearby. If an experimental autonomous car is going to fail, it's safer for it to fail in a low risk setting, and not at high speed on a highway surrounded by unsuspecting human drivers. For this reason, autonomous cars must be tested with close human supervision, and usually in controlled, sequestered settings. According to the Bureau of Transportation Statistics, in 2017, vehicles drove over 5.5 million miles on US highways [4]. Experimental autonomous vehicles can only be driven comparatively short distances in their controlled environments, which limits their exposure to but a slim slice of the multitude of diverse road conditions and driving experiences that human driven cars face. Furthermore, car manufacturers only have finite human resources to devote to the supervision and testing of new autonomous cars, and they also have financial capital tied into each physical vehicle they build and test [5]. To test and train autonomous vehicles in necessarily large numbers and over millions of miles of varied road without risking product or human lives, many companies are turning towards simulation. In simulation, an arbitrarily large number of cars can be driven through greatly varied situations, offering companies a safe venue to test their algorithms and strategies and train their autonomous driving intelligences [5]. The more conditions that can be explored digitally, the better the physical car will be able to fare in the varied conditions of the real world [6].

The goal of this project is to implement an advanced driver-assistance system (similar to what is found in L2 autonomous cars like Tesla) that renders a car capable of autonomously navigating a straight road at a predefined speed without hitting other cars. The problem will first be framed as a moving car on a road with stationary obstacles, such as parked cars and road obstructions. Time permitting, we will expand our solution to work with dynamic obstacles, such as vehicles moving at constant velocities (which may differ from our car's velocity). Finally, if time permits, we will expand the solution to work with highly dynamic systems - cars that can change lanes and move with varying speed.

Accordingly, this project will be an original research project, as we implement our own search algorithm approach to this particular problem. Later, we will compare our results with other similar works from existing literature related to path planning and autonomous vehicles.

Due to the prohibitively high cost of the requisite hardware (multiple vehicles with advanced sensor suites and sophisticated instrumentation that enables user-defined control signals on a real road) and legal obstacles surrounding autonomous vehicles, this project will be a simulation of an autonomous vehicle driving in a straight road with other vehicles. There is therefore no real hardware requirement besides the minimal computer specification requirements for our simulation software(s).

C. Literature Review

It is clear from our literature review both that planning is a vital component of car autonomy and also that RRT (and it's variants) are a popular solution for many aspects of such a problem. RRT has been used for path planning for parking [7], [8],

Collision avoidance is obviously a paramount element of autonomous car operation. Multi-agent RRT has been used for this purpose to plan to avoid collisions between two objects [9]. Time based RRT (TB-RRT) has been used for the exact opposite problem - planning rendezvous for two dynamic systems objects [10]. While rendezvous (also known as collisions) are avoided at all costs in the vehicular world, this TB-RRT approach, in which each node specifies a specific state at a specific time [10], may prove to be useful for planning collision avoidance trajectories for highly dynamic systems.

Other non-RRT related planning algorithms for autonomous vehicles are also popular in recent (and semi-recent) literature, such as the use of A* with numeric nonlinear optimization to path plan parking lot and U-turn navigation for an autonomous vehicle in an unstructured environment[11]. However, it seems that most recently, RRT and its variants have dominated the autonomous car path planning space.

There are a growing number of RRT variants, and it is clear that this is an active area of research. A bidirectional variant of RRT known as RRT-Connect converges faster than standard RRT and produces a more optimal solution[12]. RRT*-Smart, which uses intelligent sampling and path optimization has faster convergence than RRT or RRT* [13]. Local and global pruning of RRT* results via line of sight path shortening can dramatically reduce the total path from a solution, improving its optimality [14].

II. Method

A. Car Model and Dynamics

The car we are modeling is a front wheel steer (FWS), rear wheel drive (RWD), 4 wheeled vehicle utilizing Ackerman steering. This setup was chosen to closely mirror the steering dynamics of a real car. Physical cars move with non-holonomic motion, so it is important that our simulator replicate this behavor.

Figure 1 defines our key car dimensions. We'll be using the notation in this diagram [15] for the duration of this paper.

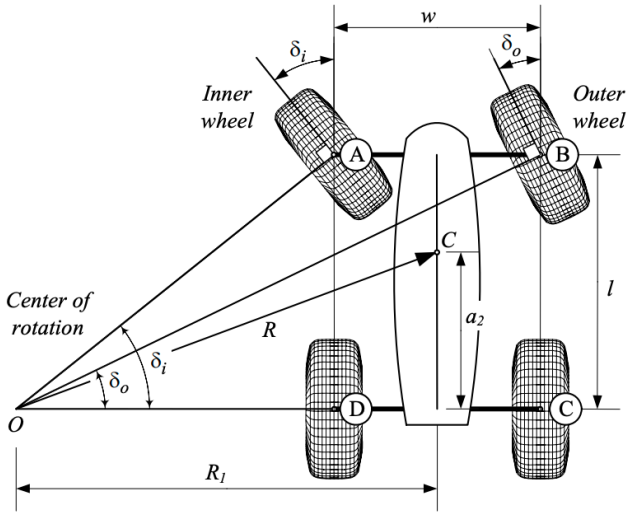

Fig. 1: A front-wheel-steering vehicle and steer angles of the inner and outer wheels.[15]

For simplicity, we assume that the two front wheels must remain mutually parallel. Therefore, we only have one wheel angle δ. We also define C, the vehicle's center of mass, to be in the physical center of the vehicle, and therefore:

$$a_2 = \frac{l}{2} \tag{II.1}$$

Figure 2 adds additional information - the speed of the rear wheels for RWD. Without loss of generality, we'll assume that the two rear wheels are actuated at equal speeds:

$$\omega_i = \omega_o = \omega \tag{II.2}$$

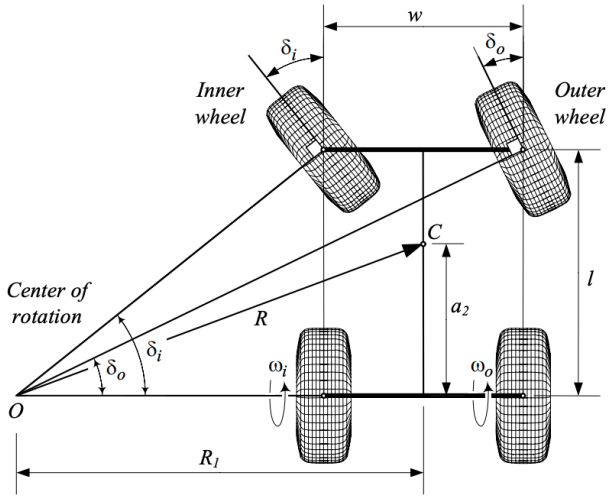

Fig. 2: Kinematic condition of a FWS vehicle using the angular velocity of the inner and outer wheels.[15]

The turning radius about O is [15]:

$$R = \sqrt{a_2^2 + l^2 cot^2(\delta)} \tag{II.3}$$

By simple trigonometry, we define a new angle α, which is the angle between R_1 and R (Figure 3):

$$\alpha = sin^{-1}(\frac{a_2}{R}) \tag{II.4}$$

R_1, the horizontal distance between C and O (Figure 1) is:

$$R_1 = Rcos(\alpha) \tag{II.5}$$

The yaw rate, or angular velocity, of the vehicle can be defined as follows [15]:

$$r = \frac{R_w \omega}{R_1 + \frac{W}{2}} \tag{II.6}$$

From the car's perspective, as the car turns, it drives out an arc with radius R. The angle the car subtends as it travels $d\theta$ is a function of the vehicle's yaw rate r and the time it has to move dt (Figure 3):

$$d\theta = r \cdot dt \tag{II.7}$$

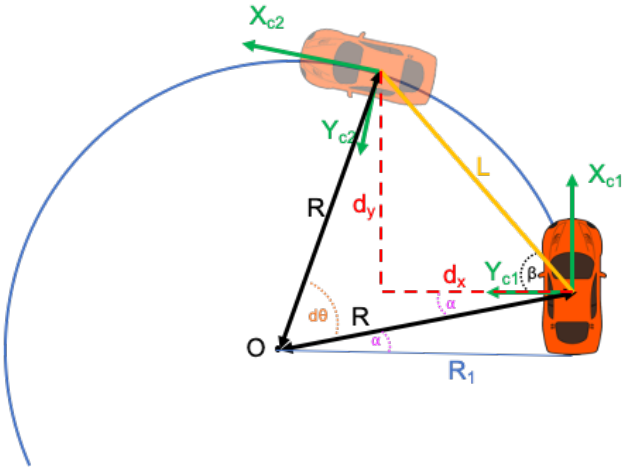

Fig. 3: Diagram showing the car motion (relative to the car frame) which defines our geometric variables

Figure 3 displays the motion of the car (in the car's reference frame) as it moves through it's arc. L is the shortest Euclidean distance between the start position and end position; a chord of the circle of movement, defined by R and $d\theta$:

$$L = 2Rsin(\frac{d\theta}{2}) \tag{II.8}$$

We're interested in finding d_x and d_y, the components of the displacement of the car in the car's reference frame. Note that the center of rotation O also moves with the car. This d_x and d_y exclusively measure the displacement of the car relative to O and do not yet account for the motion of O. d_x and d_y are in directions coincident with X_{c1} and Y_{c_1}, the car's initial direction vectors, and not aligned with the global X_w and Y_w coordinate system. We will later compensate for the initial angle of the car in the world frame, but for now we can work with just the car frame without loss of generality. Since Y_{c1} is defined to be parallel to R_1, we know the angle between R and Y_{c1} must also be α. The triangle ΔRLR must be an isosceles triangle.

$$2(\beta + alpha) + d\theta = 180° \tag{II.9}$$

Rearranging to solve for β:

$$\beta = \frac{180° - d\theta}{2} \tag{II.10}$$

We now have enough information to compute the components of the displacement of the car:

$$d_x = Lsin(\beta) \tag{II.11}$$

$$d_y = Lcos(\beta) \tag{II.12}$$

We need to compensate for angle θ of the car as well as the motion of O as the car moves. The component of d_x and d_y in the X_w direction is:

$$d_{x_w} = d_x cos(\theta) + d_y cos(\theta) \tag{II.13}$$

Similarly,

$$d_{y_w} = d_y sin(\theta) + d_x sin(\theta) \tag{II.14}$$

The center of rotation O moves as a function of the car's forward velocity v, the car's angle θ, and the time it has to move dt:

$$O_x = vcos(\theta)dt \tag{II.15}$$

$$O_y = vsin(\theta)dt \tag{II.16}$$

Finally, the new position of the car can be computed as follows:

$$x_f = d_{x_w} + O_x \tag{II.17}$$
$$x_f = d_x cos(\theta) + d_y cos(\theta) + vcos(\theta)dt \tag{II.18}$$

$$y_f = d_{y_w} + O_y \tag{II.19}$$
$$y_f = d_y sin(\theta) + d_x sin(\theta) + vsin(\theta)dt \tag{II.20}$$

$$\theta_f = \theta + d\theta \tag{II.21}$$

B. Simulation

We are building a highway driving simulation in Pygame. This simulation features a horizontally scrolling top-down view of a four lane highway (Figure 4) and the cars upon it.

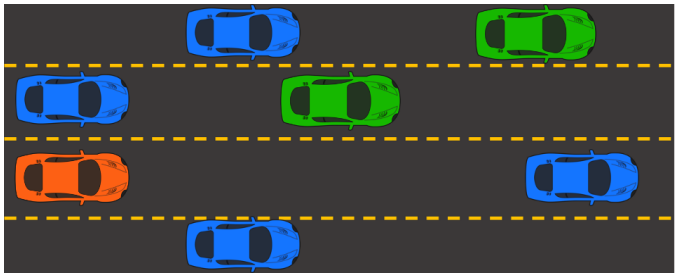

Fig. 4: Sample image from the Pygame simulation

Figure 4 shows a sample image from our Pygame simulation. Our car is shown in orange. Blue cars are disabled or damaged vehicles, serving as stationary obstacles, and can be generated randomly or at predefined locations. Green cars are other highway drivers that travel in their own lane at a constant speed (which may or may not be the same as the default speed of our car). These cars serve as dynamic obstacles. To reduce the complexity of this situation, green cars will be free to pass through any blue car in their path and are not expected to expend any effort attempting to avoid obstacles. For increased challenge, the speed of the green cars may later change as a function of time.

The camera is fixed to the orange car and moves with it.

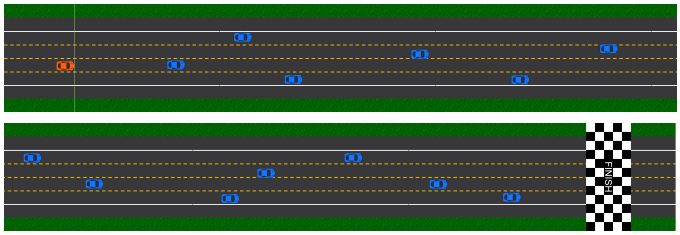

Fig. 5: Full top down view of the course, split onto 2 lines for easier viewing

1) World Builder: We created a tool to build new worlds and course. You define the difficulty (the name of this course), enter in your desired road length, and then run the program. It spawns a blank road of that length and the orange car (Figure 6).

Fig. 6: Initial view of the world builder

You can re-position the window using the arrow keys. Blue cars can be placed with a single mouse click, and will snap to the nearest lane center-line at that x position. When finished, the user hits Q to exit, and the blue car positions are saved. The next time the game is run with that difficulty mode selected, it will load the road length and car positions from the saved files.

C. Motion Planning

The goal is to implement an efficient motion planner for a self-driving car on a highway. By leveraging the nature of highway-driving, we managed to significantly reduce the complexity of the planner and develop our own pipeline, which is depicted in Figure 7. Essentially, we divided the motion planning problem into two simple sub-problems, namely, path planning and action planning. In fact, a car on a highway almost never changes its orientation relative to the lane it's driving in, except when it merges into another lane. As a result, the possible actions in this setting (i.e. highway-driving) are either driving straight to follow the current lane or merge to an adjacent one (right or left). We are also assuming that the car is driving at a constant speed. These two assumptions allowed us to decouple the orientation and the differential constraints from the path planning stage and have them taken care of by the action planner. Therefore, the path planner just outputs an obstacle-free path without having to worry about the differential constraints. The action planner then process the x and y coordinates of the path waypoints to produce a motion plan that satisfies the differential constraints of the car.

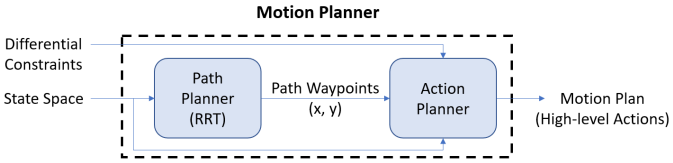

Fig. 7: Motion Planning Pipeline

1) Path planner: While the environment that car is navigating is initially assumed to be static (i.e. the obstacles are stationary), the ultimate goal of this project is to implement dynamic planning that simulates driving on a real highway. Therefore, the planning algorithm for the proposed system is going to be rapidly exploring random tree (RRT) [16], a sampling-based method that tends to offer better performance by sacrificing optimality, which is not a requirement in our application. Efficiency, on the other hand, is of paramount importance since the planner needs to be invoked every time the current motion plan becomes invalid and thus, RRT is very suitable for the nature of this problem. The main intuition behind the performance boost gained by sampling-based planners is that by randomly sampling the configuration space, they generate fewer nodes that encode the connectivity of the space compared to grid-based alternatives.

The inputs to our RRT algorithm are the start configuration q_s, goal configuration q_g, and the configuration space C, which is constructed by processing the localization information coming from the car's sensors.

The RRT algorithm comprises the following steps:

1) Initialize the roadmap tree T with the start configuration q_s.
2) Randomly sample a configuration q_{rand} from a bounded region centered around q_s.
3) Find the vertex q_n in T closest to q_{rand} using any distance metric (such as Euclidean) defined on C.
4) Use a collision detection algorithm to find a configuration q_{new} in C_{free} that is along the path from q_n to q_{rand} such that the distance from q_n to q_{new} is at most a preset branch size. Repeat step 2 if such a configuration doesn't exist.

5) Add an edge in T from q_n to q_{new} if there is an obstacle-free straight-line from q_n to q_{new}. Repeat step 2 otherwise.
6) Repeat Steps 2 through 5 until the goal configuration q_g is reached.
7) Once the car has reached q_g, define a new $q_s = q_g$ and a new q_g at the edge of the car's sensing range.
8) Repeat steps 2 through 7 until the car reaches the final goal.

It's worth Noting that the algorithm fails and wastes much effort in running into C_{obs} instead of exploring the space, if q_g is picked every iteration. That's why q_g was rather sampled with a 5% probability to make RRT properly work.

2) Action Planner: The design of the action planner is actually quite simple. We first defined 3 motion primitives that satisfy the differential constraints of the car, namely drive straight a given distance and turn right (or left) with a given steering radius (which changes the orientation of the car). We then utilize them to construct our actual action set, that is, follow the current lane, merge into the right lane and merge into the left lane. Thus, The action planning algorithm will just need to need to detect whenever a path waypoint no longer lies within the current lane, and choose accordingly the merge action corresponding to the new lane the car is in. If there is no such detection, the algorithm merely drives the car forward to follow the current lane.

III. Results

A. Generating the Plan

Once we had the world we had to generate equations that allowed us to determine whether or not any of the sampled points for the RRT fell inside our obstacles. We chose to model all of the sprites as simple rectangles for simplicity then used half plane equations to check if a sampled point fell within the bounds of any of the obstacles. We also added clearance values to ensure that the sampled point would never fall in the same lane on the top and bottom of the car. The clearance in front and back of the car were equal to the distance we traveled during a merge. This ensures that the planner recognizes that the car needs to change lanes well before the cars would collide.

After the obstacle space was generated we used the RRT algorithm to generate and search the space. This gave us a plan of way points that we needed to traverse in order to reach the goal. We adjusted the minimum branch size for the tree to optimize for both speed and accuracy of the plan. Once we had the series of way points for the planner we needed to use the local planner to generate a set of actions for the car to take to follow the way points. The results of the local planner is a list of either a left or right merge and a location (in the road direction) to start the merging process.

B. Simulating the Driving

This action list is then then fed into the simulation. In pygame we wrote a function `turnCar`. This function takes the wheel angle as an argument and calculates the change in position and angle of the car given the intrinsic car properties and a specified (currently 0.1 seconds). We then wrote a function that merges by turning the wheel in one direction for a specified about of time, then turns in the opposite direction for an equal amount of time. We needed to tune the wheel angle, and timing so that the car would move in a reasonable fashion. Currently there is only one configuration for changing lanes but the program would easily support faster or slower merges by increasing or decreasing the wheel angle during the merge operation.

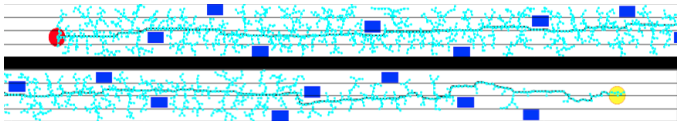

Fig. 8: Matplotlib visualization of RRT tree and waypoints

After we had a function that could merge the car we had the car go through the list of actions, executing a left or right merge at the specified location. The results of the simulation can be seen in this video: https://youtu.be/YY4P6OunSC4

IV. Future Work

A. Obstacles

Currently the planner only supports stationary objects (blue cars). We included functionality to randomly spawn dynamic obstacles (green cars) which drive at either constant or varied speeds, but due to time constraints were not able to integrate these with the planner.

In addition to dynamic obstacles we would also like to build out a more varied obstacle set. Not every car on a highway is the same size so a more robust simulation would allow us to include obstacles such as large trucks or even motorcycles.

B. Planner Improvements

Currently, the planner has access to the entire length of road, and plans a path from the start to the finish. More realistically, a car would only be aware of obstacles within a finite sensing range of the vehicle, and need to update it's plan as new information becomes available. In a future version of this project, we'd modify our planner to only have knowledge of obstacles within a certain distance of the car, forcing it to plan only from it's current position to the edge of it's sensing range, and then reevaluating and re-planning that path as the car progressed further.

The planner also needs to be able to handle dynamic obstacles. There are a few different variants of RRT that would allow us to generate a time based tree and future versions of the code should use those.

Currently the action planner only produces two different actions, 'Merge Left' or 'Merge Right'. In future versions of the code we should included variations on the cars speed as well as faster or slower merges depending on what is needed to best follow the way points.

C. Simulation Improvements

Collision Detection

Pygame natively has the ability to detect collision between sprites. Since all objects on the road are rendered as sprites it would not be difficult to add this as a feature. The planner could be stopped at any point where there is a collision and we could collect data to see what types of planning architectures lead to the fewest number of collisions.

Green Cars

We'd like to add additional functionality to the world builder in the future. Currently, green cars (although not enabled) spawn at random locations and at random speeds, irrespective of the lane line boundaries. It would not be difficult to mirror the blue car lane constraints to the green cars, so they spawn centered within their lanes. In addition, the world builder tool should have functionality for placing green cars, perhaps via a different key or click type (left click). As this functionality is expanded, it would make sense to include options to add other types of vehicles.

Lanes and World Structure

The world builder is somewhat restrictive, limiting the user to only straight roads with 4 lanes. Future versions of the world builder should include a lane count to allow for an arbitrary number of lanes. In reality, straight roads are but a small part of the full driving experience. We'd love to modify the world builder with a segment builder, allowing users to build out custom lengths of road. Instead of having one long road, the road would be comprised of a series of segments - straight, 45° or 90° left or right turns, intersections, exits, traffic circles, interchanges, etc. To further improve the flexibility of the world builder, users should be able to define the starting lane/position of the orange car.

APPENDIX

A. Definitions

According to SAE J3016[17][2], there are 6 relevant tiers of autonomy for vehicles:

0) **No automation** - the driver is fully responsible for all vehicular operation under all circumstances
1) **Driver Assistance** - Vehicle is controlled by the driver, but some driving assist features may be included in the vehicle design
2) **Partial Automation** - Vehicle has combined automated functions, like acceleration and steering, but the driver must remain engaged with the driving task and monitor the environment at all times.
3) **Conditional Automation** - Driver is a necessity, but is not required to monitor the environment. The driver must be ready to take control of the vehicle at all times with notice.
4) **High Automation** - The vehicle is capable of performing all driving functions under certain conditions. The driver may have the option to control the vehicle.
5) **Full Automation** - The car is fully responsible for all vehicular operation under all circumstances. No human driver is necessary for any portion of operating the vehicle

RRT - Randomly exploring Random tree

FWS - Front Wheel Steering, a common steering method for cars in which only the front two wheels can be angled. The rear two wheels remain parallel to the car chassis

RWD - Rear Wheel Drive, a common driving method for cars in which only the rear two wheels are powered. The front two wheels are free to spin but not connected to the engine.

AWD - All Wheel Drive, a common driving method for cars in which all 4 wheels are driven by the engine.

B. Important Variables

$\boldsymbol{R_1}$ - the distance between the center of mass (C) of the vehicle and the Center of Rotation (O). This distance is parallel to the rear wheel axle. See Figure 1. Not to be confused with R, R_w, or r.

$\boldsymbol{R}$ - the radial distance from the center of mass (C) to the Center of Rotation (O). Not to be confused with R_1, R_w, r.

$\boldsymbol{R_w}$ - the radius of the wheels, in meters. Not to be confused with R, R_1, or r.

$\boldsymbol{r}$ - the yaw rate (or angular velocity) of the vehicle, in rad/s. Not to be confused with R, R_w, or R_1.

$\boldsymbol{C}$ - the center of mass of the car, defined to be the physical center of the vehicle

$\boldsymbol{l}$ - the distance between the front and rear axles of the car. See Figure 1.

$\boldsymbol{a_2}$ - the distance between the rear axle and the center of mass (C). See Figure 1. Defined to be $l/2$. Not to be confused with α

$\boldsymbol{\alpha}$ - the angle between R_1 and R. Also the angle between Y_{c1} and R. See Figure 3. Not to be confused with a_2

$\boldsymbol{W}$ - the distance between the left and right wheels. See Figure 1. Not to be confused with ω.

$\boldsymbol{\omega}$ - the angular velocity of the rear wheels, in rad/s. Not to be confused with W

$\boldsymbol{d\theta}$ - the angle subtended by the car as it rotates through it's arc. See Figure 3. Not to be confused with θ

$\boldsymbol{\theta}$ - the angle of the car, defined in degrees as positive in the counter clockwise direction. 0° is the car facing horizontally to the right. Not to be confused with $d\theta$.

$\boldsymbol{\beta}$ - the angle between Y_{c1} and L. See Figure 3

$\boldsymbol{X_c}$ - the X direction of the car frame. This is always the direction of forward motion for the car (regardless of it's world orientation). See Figure 3. Not to be confused with X_w.

$\boldsymbol{X_w}$ - the X direction of the world frame. This is defined to be horizontal to the right. Not to be confused with X_c.

$\boldsymbol{Y_c}$ - the Y direction of the car frame. This is always the direction to the left of the car (regardless of it's world orientation). See Figure 3. Not to be confused with Y_w.

$\boldsymbol{Y_w}$ - the Y direction of the world frame. This is defined to be vertical, with positive Y in the downward direction. Not to be confused with Y_c.

C. Graphics Sources

Orange, blue, and green car graphics from http://clipart-library.com/overhead-car-cliparts.html

Grass from https://www.moddb.com/groups/indie-devs/tutorials/how-to-make-a-simple-grass-texture-in-gimp

[1] National Center for Statistics and Analysis, "2018 Fatal Motor Vehicle Crashes: Overview. (Traffic Safety Facts Research Note. Report No. DOT HS 812 826)," October 2019. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812826

[2] NHTSA, "AUTOMATED DRIVING SYSTEMS 2.0 A Vision for Safety," September 2017. [Online]. Available: https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf

[3] Santok Singh, "Critical reasons for crashes investigated in the National Motor Vehicle Crash Causation Survey. (Traffic Safety Facts Crash Stats. Report No. DOT HS 812 115)," February 2015. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115

[4] Bureau of Transportation Statistics, "U.S. Passenger-Miles," 2018. [Online]. Available: https://www.bts.gov/content/us-passenger-miles

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a/dosovitskiy17a.pdf

[6] P. Robbel, M. Maass, R. D. T. M. Meijs, M. Harb, and et al., "Safety first for automated driving," 2019. [Online]. Available: http://apollo.auto/docment/Safety_First_for_Automated_Driving_handover_to_PR.pdf

[7] L. Han, Q. H. Do, and S. Mita, "Unified path planner for parking an autonomous vehicle based on rrt," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 5622–5627.

[8] Y. Wang, D. K. Jha, and Y. Akemi, "A two-stage rrt path planner for automated parking," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, 2017, pp. 496–502. [Online]. Available: https://ieeexplore.ieee.org/document/8256153

[9] M. Čáp, P. Novák, J. Vokřínek, and M. Pěchouček, "Multi-agent rrt*: Sampling-based cooperative pathfinding (extended abstract)," 2013.

[10] A. Sintov and A. Shapiro, "Time-based rrt algorithm for rendezvous planning of two dynamic systems," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 6745–6750.

[11] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Path planning for autonomous vehicles in unknown semi-structured environments," *The International Journal of Robotics Research*, vol. 29, no. 5, pp. 485–501, 2010. [Online]. Available: https://doi.org/10.1177/0278364909359210

[12] S. Klemm, J. Oberländer, A. Hermann, A. Roennau, T. Schamm, J. M. Zollner, and R. Dillmann, "Rrt*-connect: Faster, asymptotically optimal motion planning," in *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2015, pp. 1670–1677.

[13] I. Noreen, A. Khan, and Z. Habib, "A

Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms," *IJCSNS International Journal of Computer Science and Network Security*, vol. 16, no. 10, October 2016. [Online]. Available: http://222.124.154.59/politala/1.%20Jurusan/Teknik%20Informatika/19.%20e-journal/Jurnal%20Internasional%20TI/IJCSNS/2016%20Vol.%2016%20No.%2010/20161004_A%20Compar%20ison%20of%20RRT,%20RRT%20and%20RRT%20-%20Smart%20Path%20Planning%20Algorithms.pdf

[14] ——, "Optimal Path Planning using RRT* based Approaches: A Survey and Future Directions," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 7, no. 11, 2016. [Online]. Available: https://pdfs.semanticscholar.org/9c35/2fec7a86c875eec17fc054106414b6914b7d.pdf

[15] R. N. Jazar, *Steering Dynamics*. New York, NY: Springer New York, 2014, pp. 387–495. [Online]. Available: http://ckw.phys.ncku.edu.tw/public/pub/Notes/GeneralPhysics/Powerpoint/Extra/05/11_0_0_Steering_Theroy.pdf

[16] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. [Online]. Available: http://planning.cs.uiuc.edu/

[17] SAE, "Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems J3016_201401," January 2014. [Online]. Available: https://saemobilus.sae.org/download/?saetkn=7Ab97uLiFw&method=downloadDocument&contentType=pdf&prodCode=J3016_201806&cid=1000408663