Worcester Polytechnic Institute

Path Planning and Threat Modeling for Self Driving Cars

Directed Research End-of-Term Report

Authors:

John O'Neill

Evan Kelly

Supervisor:

Raghvendra V. Cowlagi, Ph.D.

May 26, 2020

Contents

1 Introduction . . . 1
2 Cost Model . . . 2
3 Ray Tracing . . . 5
4 Parameterization . . . 7
5 3D Constant Velocity Path Planning . . . 9
5.1 Node Spacing . . . 9
5.2 Adjacency . . . 9
5.3 Limitations . . . 10
6 4D Dijkstra . . . 11
6.1 Helper Function Descriptions . . . 11
6.2 Description of Path Planning Algorithm . . . 13
6.3 Scenarios Tested and Results . . . 14
6.4 Sensor Data Addition . . . 14
6.5 Heuristic Implementation . . . 16
6.6 Sensor Path Planning Results . . . 17
7 Time-Varying Transition-based RRT* . . . 18
7.1 Modification of the RRT* Algorithm . . . 18
7.2 Motion Constraints . . . 20
7.3 Minimization of Jerk and Steering Rate . . . 21
7.4 Refinement Control . . . 22
7.5 Map and Scenario Definitions . . . 22
7.6 Carla Simulator Implementation . . . 24
7.7 Limitations . . . 25
References . . . 26

1 Introduction

Autonomous vehicles in production today likely employ light-based detection systems such as LIDAR (light detection and ranging) in processing their surrounding environment [1]. Since the mid-20^{th} century, LIDAR technology has quickly developed into likely the most feasible method for implementation into the world of self-driving vehicles. High fidelity sensors can produce quite precise representations of the world around the vehicle, with one caveat: anything behind an obstacle in line with the line of sight of the LIDAR sensor will inherently be left undetected, causing the driverless vehicle to have an incomplete knowledge of the surrounding environment. This could lead to potentially fatal situations if an obstacle like a pedestrian or vehicle is obstructed by another object. The ultimate goal of this project is to design a solution to this major drawback of the LIDAR sensor platform using a decentralized network of vehicles that can inform any one vehicle of objects on the road which may be outside of its view.

This summer Evan and Jack worked to develop path planning algorithms necessary for the self-driving vehicles, beginning with a time-varying implementation of Dijkstra's minimum-cost pathfinding algorithm using parameterized cost maps. They then branched off to develop two more viable pathfinding algorithms. Evan modified Dijkstra's algorithm to create a four-dimensional pathfinding algorithm, and Jack modified the Rapidly exploring Random Trees searching algorithm (RRT*) to account for operational limitations of the driverless vehicle. These pathfinding algorithms are currently being finalized. They will be used in conjunction with the Carla simulator [2] to test validate the importance of a decentralized network of vehicles in avoiding collisions.

2 Cost Model

In order to implement any form of optimal path planning algorithm, the first step that we needed to accomplish was finding a way to mathematically model the threat field produced by a car. This model needed to be a function of the agent self driving car's position and velocity as well as the position and velocity of the other car producing threat. It additionally, needed to have an elliptical skewed contour shape to reflect that the space in front of a car is more threatening than the area behind it where the maximum threat is located at the actual position of the car itself. With these basic requirements, I investigated several different functions that could be used to model.

Ultimately, I decided upon using a modified form of a bivariate lognormal probability distribution function to model the threat produced by a given car, the overall form of which was taken form a paper detailing the use of such function to model asbestos fibers [3].The the mean value reflected the threat producing car's position and the variance was used as a scaling parameter. The two independent variables that this field depended upon were the spacial position of the car, which created a three dimensional threat curve upon a two dimensional spacial field representing the positions of this car. The function was also dependent upon a number of other parameters including the velocity of the threat producing car, the position and velocity of the self driving car, as well as a number of scaling parameters. These parameters were chosen such that the car was roughly five meters long and two and one half meters wide and that the maximum intensity existed at the position specified as the car's location. The parameters sigma x and sigma y reflected how the curve was skewed in the x and y axis respectively. Therefore, sigma y was set to a small value as to limit the skew to only existing only in the x direction as the cost field should be roughly symmetrical about this axis. The value of sigma x determines how much area in front of the car is threatened. As such is value was based upon the relative velocity between the self driving and threat producing car so that a greater difference in velocity leads to more area being threatened. This distance was scaled from a basic driving rule stating that for every 10 miles per hour in relative velocity, roughly one car length should be left ahead of a given car. The parameters cx and cy work similarly to their respective sigmas, however, instead of scaling the skew of the distribution, they uniformly scale the distribution over their respective axis. These values were experimentally set such that the car itself would have roughly the desired length and width. The equation for which is shown below.

$$\phi(x, y, (t) \mid p_{x_n}(t), p_{y_n}(t), V_{x_n}(t), V_{y_n}(t), p_{x_0}(t), p_{y_0}(t), V_{x_0}(t), V_{y_0}(t)) \tag{1}$$
$$= \sum_{n=1}^{Nc} \frac{4e^{-(.05-.00025Vrel_n)Prel_n x}}{(1+e^{-(.05-.00025Vrel_n)Prel_n x})^2} \frac{1}{2\pi\sigma_x\sigma_y c_x sgn(V_{x_n})(x - p_{x_n} + sgn(V_{x_n})8)c_y(y - p_{y_n} - 5)}$$
$$* e^{\frac{\frac{ln(c_y(y-p_{y_n}-5))^2}{\sigma_y} + \frac{ln(c_x sgn(V_{x_n}(x-p_{x_n}+sgn(V_{x_n}8))^2}{\sigma_x}}{-2}}$$

$$\sigma_x = .1 + .0092Vrel_n \tag{2}$$

$$\sigma_y = .1 \tag{3}$$

$$Prel_n = \sqrt{(p_{x_n} - p_{x_0})^2 + (p_{y_n} - p_{y_0})^2} \tag{4}$$

$$Prel_n = \sqrt{(V_{x_n} - V_{x_0})^2 + (V_{y_n} - V_{y_0})^2} \tag{5}$$

ϕ =total cost; Nc =number of monitored cars; (p_{x_0}, p_{y_0}) =position of self driving car; (V_{x_0}, V_{y_0}) =Velocity of self driving car; (p_{x_n}, p_{y_n}) =position of n^{th} car; (V_{x_n}, V_{y_n}) =Velocity of n^{th} car; c_x =scaling parameter=.1; c_y =scaling parameter=.2;

This function overall generates a threat field for a given car, where the complete threat field is the superposition of the threats generated from all cars on the road. The threat field is concentrated at the position of the car and extends in the direction of its motion by a distance based upon its relative velocity. The intensity of the distribution itself is based upon the car's relative distance, where the threat field increases at a decreasing rate as the threat approaches the self driving car and decreases at a similar rate as the threat passes. This was again based upon a basic road rule, where one should be aware of all cars within roughly 300 feet. The figure below shows two cars for which such a threat field is created for. The longer distribution corresponds to a car that is moving significantly faster than the shorter distribution.

The cost model does, however, have a number of limitations mostly owing to the use of a logarithmic function. The domain is restricted to logarithm inputs greater than zero. This does not at first seem to be an issue, however, because the function is horizontally and vertically translated, there are areas within the positive x and y field that lead to negative logarithm inputs and therefore imaginary threat fields. This can be remedied fairly easily by setting any imaginary cost to zero as they do not fall anywhere in the distribution and are negligible.

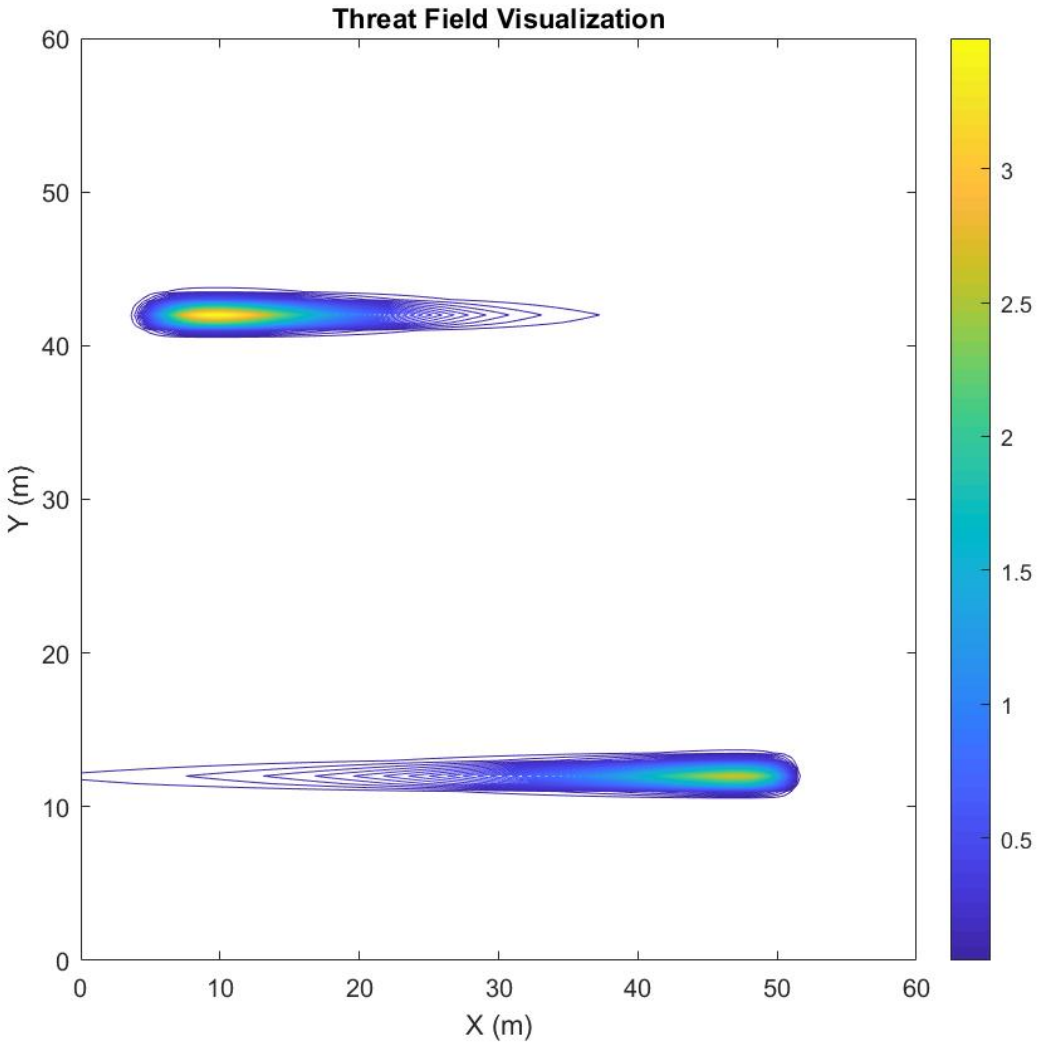


Fig. 1: Plot of parameterized threat field overlayed with true threat field.

3 Ray Tracing

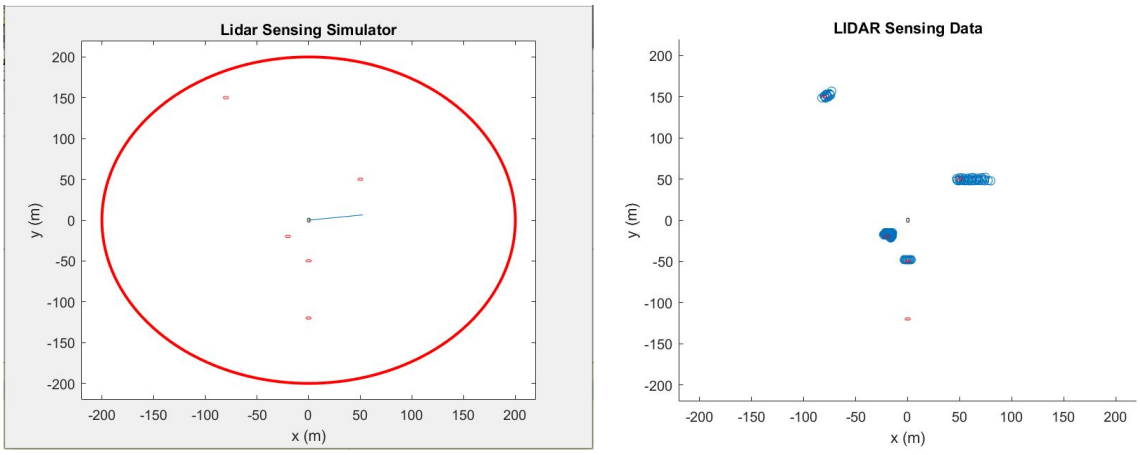
The current state of practice way that self driving cars map the world around them and discover threats is to use a sensor package consisting of a RADAR, LIDAR, cameras, and sometimes magnetic sensors. In order to generate paths that would reflect what could be done on a current self driving car, we needed to develop a way to simulate what data a car would have from these sensors, to build paths around, to form a baseline to compare more state of the art methods such as car connectivity.

We developed a function in MATLAB that allows one to input the position, velocity, and heading angle of any number of cars on a given road and outputs the "sensed" position and velocities of these cars. This function relies on a method called ray tracing where rays are projected out from a measurement point and bounced back to the source and a basic time of flight calculation is performed on them to measure the distance of between the source and the obstacle, mimicking the functionality of a LIDAR.

Firstly, the function initializes by generating obstacles in the form of rectangles with a length and width set to five and two and one half meters respectively, centered around the positions of the cars set by the user. Ray tracing was implemented in the function by specifying the measurement source as the self driving car and tracing out lines from the source out to a specified measurement radius. If the line at any point comes within a certain tolerance of an obstacle it stores the coordinates of that collision and stops tracing the line, to reflect the limitations of the LIDAR, not being able to see past objects. The function then repeats this process in a complete circle around the source at a specified resolution. A user could set this angular resolution and measurement range based upon information taken from a LIDAR data sheet if they wished to simulate a certain sensor.

This process is then iterated over time at discrete time steps, where at the start of every time step the obstacles are updated based upon the vehicle's imputed velocities and heading angles. Once a full 360 degree sweep is completed the function sorts the position data it collects by car and finds mean values of the x and y values of these points for every car to obtain an estimate of the positions of all cars at that given time step. Once all of these sweeps are completed the function numerically differentiates these positions to obtain their velocities and heading angles for all time steps. The overall output of the function is then the "sensed" position and velocity of the obstacle cars. The two plots below show the results of this function. The plot on the left shows the process

by which collisions are detected, and the plot on the right is a scatter plot of collision data collected over the simulation.



(a) (b)

Fig. 2: (a) Ray Tracing Plot, (b) Scatter Plot of Ray Tracing Data

Overall, the function is able to accurately sense the position and velocity of cars that it inputs. There is a small amount of error even in this ideal noiseless sensor due to the unequal distribution of data points around a car due to the ways in which the rays spread out as they travel further from the source, which is consistent with how a true LIDAR would operate. Additionally, as expected the simulation is not able to detect cars that are being blocked by other obstacles e.g. other cars, trees, or bushes. This leads to the creation of limited threat fields which was one of our main goals with this portion of the project.

To improve the fidelity and accuracy of this model in the future we could artificially corrupt the sensor data with white noise with a variance similar to the noise encountered in a typical LIDAR sensor. We would then have to filter this noise with a Kalman filter in order to have usable data. This would be an area of future work that would make the sensor data more true to reality if we were to implement it.

4 Parameterization

With the overall cost model reflecting the threat field produced by an agent car finished, we needed to parameterize the threat field over a field of bivariate gaussian distributions. This was done by expressing the formulation of the cost field in the form of the equation below, where the field is equal to a summation of parameters, theta, multiplied by the Gaussian distributions, phi.

$$c(x) = \sum_{n=1}^{N_p} \theta_n \phi_n(x) \tag{6}$$

These parameters were constants that we calculated using the inner product between the Gaussian distributions and the cost field we previously derived. The equation to determine the parameters is shown below.

$$\theta_n = \int_x \int_y c(\underline{x}, t)\phi_n(\underline{x})\, dx\, dy \tag{7}$$

When we implemented this parameterization, we needed to choose several parameters of the Gaussian distributions in order to have an accurate depiction of the overall cost field. We needed to decide how many distributions the threat would be expressed over, this resulted in a trade off between accuracy of the threat field, with more distributions over a given threat field area, and processing efficiency, where higher fidelity led to higher processing times, making running any subsequent scenarios using this process difficult to run efficiently. Selecting this placement was an iterative process of trying various resolutions, but we eventually decide upon a resolution of 300 distributions over a special distance of roughly 10 meters. This gave us a reasonable trade off between acceptable accuracy and usability. Once we deiced upon this spacing and therefore placement, we next had to set the variance of the distributions. In order for this parameterization to be compatible with IPAS, we needed the distributions to be non-overlapping so it would be possible to treat them as distinct points. With the spacing of the distribution decided by the specified resolution, we determined these variances so that the peaks of the distribution would be separated by a distance of 4*sigma. It was then assumed that this spacing would lead to a negligible amount of overlap.

Upon generating the Gaussian field, we calculated the inner product and performed the summation over a number of terms equal to the number of Gaussian distributions in the field. The results of which are shown in the figure below with the cost field

overlaid above the distribution.

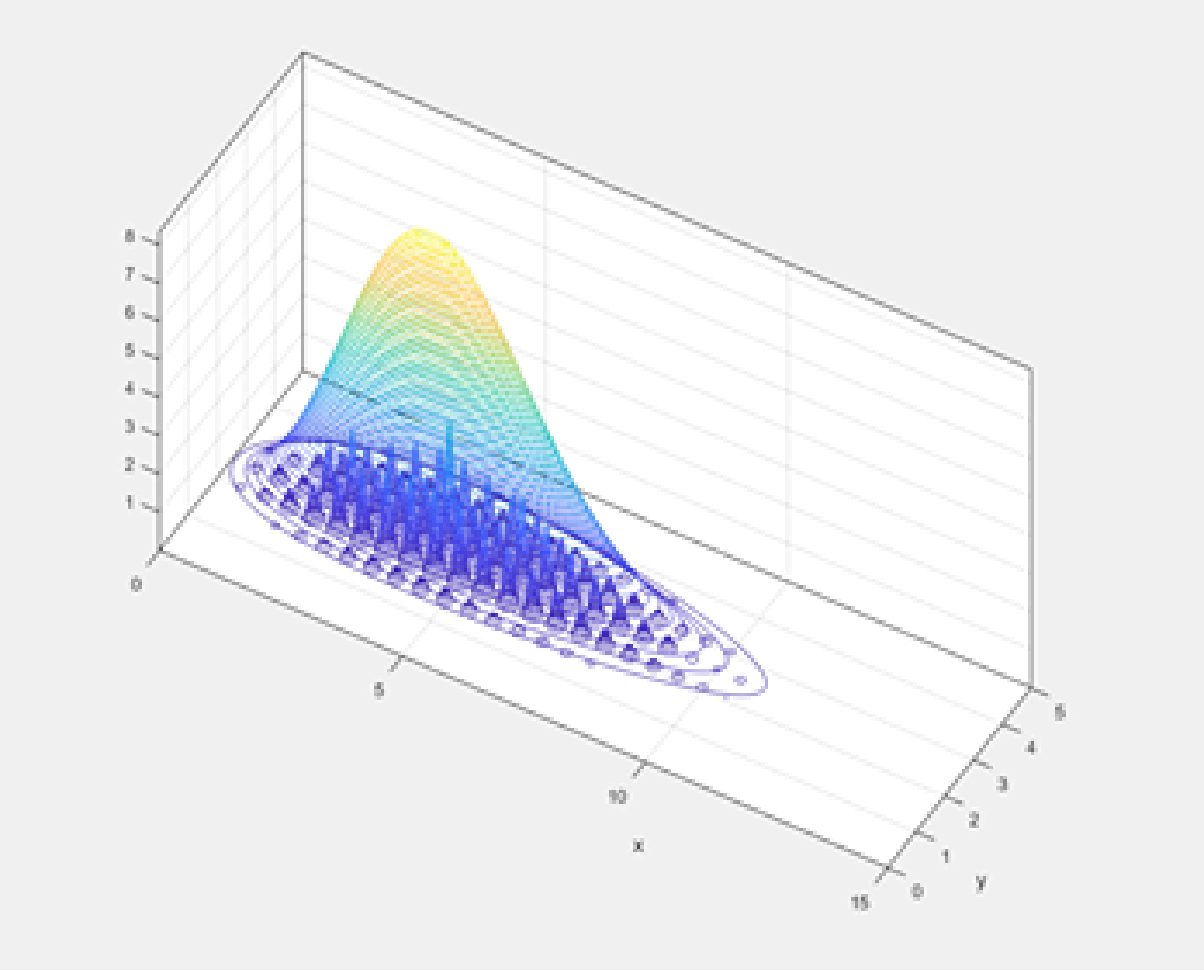


Fig. 3: Plot of parameterized threat field overlayed with true threat field.

It can be seen from the figure that the distributions follow a correct overall shape and trend, however, we were unable to properly normalize the intensity of the parameterized field such that it would have the same values as the original cost field. We attempted to find a constant coefficient that the Gaussian intensities could be multiplied by to roughly equate the original cost field using the ratio between the maximum intensity of the true and parameterized threat. We were never able to adequately equate the two threats, however, where the absolute intensity of the threat field itself is somewhat arbitrary we did not view this as a major issue, however, this is an area that could be expanded upon in the future to improve the accuracy of this parameterization.

5 3D Constant Velocity Path Planning

With the completed parameterization code we could now begin incorporating Dijkstra's minimum cost path finding algorithm to a parameterized cost field. Since our cost field only represents a single point in time, an optimal path calculated for a model of, say, a highway with multiple vehicles, would make no sense since the path would not account for the time-varying cost field due to the movement of the vehicles. Therefore, it was necessary to implement a third dimension of time into the graph. With insight from the paper "Path-planning with waiting in spatiotemporally-varying threat fields" [4] we defined a three-dimensional cost field with time as the third dimension. For each time step there is an associated two-dimensional cost field which we call a cost layer. For a "full-knowledge" scenario each cost layer is defined as a parameterized cost field associated with the position of all the actors at that layer's time point. This results in a parameterized time-varying cost field, allowing the actor to navigate through position and time domains at a constant time step. This code provided a framework to replicate user-defined maps. Using this three-dimensional pathfinding algorithm (which we call 3D Dijkstra) we explored scenarios on two map layouts: a three-lane highway, and a four-way intersection.

5.1 Node Spacing

Our graph contained uniformly distributed positional nodes for each layer, and each layer was separated by a uniform time step. The positional node distance on a layer were chosen so that the user-defined maximum allowable speed V_{max} would be equal to the positional node distance divided by the time step. Since computational demand was also an issue, the resolution of these time steps and positional distances needed to be sufficiently large to allow MATLAB to complete the script in a reasonable amount of time.

5.2 Adjacency

The driving "rules" differed between the two maps we explored, necessitating two adjacency models. For each node in the four-way intersection the adjacent nodes are defined as being one single time step above the node at either the same spatial location, or immediately within one node distance in the x or y directions. This allows the actor to move in any direction at the defined maximum velocity, or to wait if necessary. For the

highway, since a driver would not be able to stop or to move backwards, for each node its adjacent nodes are defined as being one time step above the node at immediately one node distance in the positive or negative y direction, or one node distance in the positive x direction. This forces the actor to only move in the positive x direction or "steering" on the positive or negative y direction, but never stopping or moving in the negative x direction.

5.3 Limitations

A major drawback of the 3D Dijkstra algorithm is the binary nature of how velocity and steering are defined. Since the adjacency model is fixed and cannot be updated to accept higher or lower velocities (that is, adjusting the time step or the distance between nodes in a layer), the actor is constrained either to moving at the maximum velocity or to be completely still, also constraining it to move at a single spatial distance for every time step. Furthermore, the actor is limited to a four headings, pointing north, south, east or west. This pathfinding model, while it might be a step in the right direction, would not be a feasible solution for our project.

6 4D Dijkstra

The next path planning method that we investigated was a four dimensional implementation of Dijkstra's Algorithm. The basic premise of this approach was to apply Dijkstra's Algorithm to a four dimensional graph, consisting of the agent's x coordinate, y coordinate, velocity, and time. The first three of these states existed in a finite discretised grid, similar to the states we previously employed in the three dimensional case, however, in this case we chose time to be continuous to add more fidelity to the model and to allow for the car to have different velocities over the course of the path, making velocity a decision variable of the system. With this continuous 4D grid, I was unable to use a standard implementation of Dijkstra's algorithm where a user specifies node adjacency and edges ahead of time, as the graph was infinitely large in the time direction. Additionally, because the velocity was constantly changing, the adjacent nodes could not be predicted ahead of time, as they would depend upon the velocity of the previous state. Finally, in a typical implementation of Dijkstra's algorithm the visited nodes are removed from a list of all nodes until the destination node is marked as visited. In this case, while the basic principle remains the same, I had to instead append all visited nodes to a list and check for when the destination appeared. In order to implement this algorithm, I created a class and a series of helper functions within MATLAB to each run a specific portion of the problem.

6.1 Helper Function Descriptions

Considering that the nodes reflected in this implementation were of four states [px py V psi], I created a hashing rule that would allow for these for values to be reflected in a single number, to increase ease of storage and searching to improve its overall performance. I created two functions, hash and dehash, that allow one to convert between a node and a hash. To generate these hashes, I began by defining a node position of a given node. I visualized the three dimensional finite grid of the first three states as being in the x, y, and z dimensions of a cubic grid respectively. The node position was therefore equal to the number of nodes away from the origin of the grid, beginning in the bottom left corner and working one's way to the right horizontally, then up vertically, finally moving up through the layers in the third dimension. This position was then designated as the node position and would always be a integer. In order to encode the time in a way such that all four states could be calculated from a given hash, I divided the state's time by

an arbitrary number dubbed t con which could be any value greater than the final time of the simulation. This ensures that the time is always equal to a decimal and is then added to the node position to create the hash. This processes can then be reversed to calculate a node from a hash.

With the hashing rules established, the first major step I had to address in this implementation was to create a function that would generate a list of nodes adjacent to a given starting node and the corresponding costs to move to them. This function could then be iteratively called at each step of path planning to determine the nodes available to the algorithm.

From a specified road type, any of eight possible spacial locations are adjacent to the current node, including up, down, left, right, and any diagonal combination of these moves. These allowances are based upon a specified road type, either highway or four way where in the former only moves corresponding to right, up right, and down right are allowed, and in the latter all eight directions are allowed. Additionally, the location in the spacial grid also constrains possible neighbors as if the actor is on any border of this grid it cannot continue moving in that direction off of the grid. One major assumption that this whole approach takes is a binary constant acceleration, meaning that the actor can accelerate at a rate of either a, -a, or 0, where a is a specified constant. Using this assumption, the actor can arrive at any of the allowed spacial locations at one of three velocities as it could travel there under any of the three accelerations, by the time it reaches the location at that neighbor's time value. The function calculates these velocities and times using basic 1 dimensional kinematics, again under the assumption of constant velocity. The function then outputs these spacial coordinates, velocities, and times of the adjacent nodes. The function then inputs both these neighbor nodes and the starting node into the bivariate cost function detailed above. This then calculates the value of the threat produced by any actors present at both the starting and neighboring nodes based upon the spacial and temporal coordinates of the actor. In order to penalize paths that involve the actor driving in restricted lanes or off of a given road, I created two other sources of threat, lane threat and obstacle threat. These are based solely on the spacial location of a given node and are added to the threat due to actors on the grid. There are two prebuilt scenarios currently, a highway and a four way intersection which add this threat to a grid of any size, although the highway field was designed to be used on a narrow rectangular grid and the four way was made for a smaller square

grid. The values of these threats are given by the user The cost for any given move is then given by the arithmetic average of the threat at the source and neighboring node. The get neighbors function then output these values for the adjacent nodes and their corresponding costs and could be used in the Dijkstra function itself.

6.2 Description of Path Planning Algorithm

The algorithm itself inputs the source and destination of the path along with the threat generating constants described above. The initialization step of the algorithm consisted of setting the current node equal to the source node and by modifying the destination such that it was equal to the spacial component of the destination node hash only. This allows a user to input only a spacial destination and the algorithm will select optimal velocity values. Following the initialization step the overall structure is as follows. While the spacial destination has not yet been visited the algorithm runs the get neighbors algorithm on the current node. It then checks to see if any of the adjacent nodes have been checked thus far, if not it sets their edge weight to infinity and enters them into an array called info. This array consists of three columns, node, cost, and previous step for every node the algorithm checks. The algorithm then checks if the calculated cost is less than the previously stored value, infinity if a node has only just been discovered. If so it updates the cost to this smaller value and updates the previous node entry for that neighbor to the current node, as this is now the least cost path to it. It then marks the current node as visited and sorts all of the entries in the info array by cost and selects the lowest costing node whose spacial position has yet to occur on the visited list. This is then set as the current node and the algorithm repeats until it visits a node with the spacial position of the destination node. This indicates that the algorithm has found the least cost path and then iteratively reconstructs the path it took to get there going from the destination node to its corresponding previous node until it reaches the source node. This reconstruction method prevents the algorithm from having to store the whole path of nodes to all checked nodes, usually on the order of several thousand for a typical example in order to improve processing efficiency. This path along with the final value of cost calculated for the destination node are then outputted from the overall function.

6.3 Scenarios Tested and Results

So far I have tested this function on several different examples for various road types. I intended to use these tests not as any sort of final result but as a proof of concept to demonstrate the effectiveness of this path finding algorithm. The first test was to simulate a car moving at high speeds on a highway, trying to get from one end of the grid to the other when a slow driver was blocking the lane in front of them. As expected the algorithm showed that the car began in the starting lane and then immediately traveled to an adjacent empty lane and passed the car before returning to its original lane to arrive at its final destination. I designed the second scenario to be a foil to the first and to test the simulation's capabilities when it had many more movement choices at a much lower speed where stopping and waiting was an option. To do this I simulated a stop sign at a four way intersection where the self driving car began stopped at a stop sign. It then, again as expected, waited for a car moving perpendicular to it through the intersection to pass before it began to accelerate and move across the intersection itself, avoiding a collision. While these simple examples are by no means exhaustive proof of the algorithm's effectiveness nor do they reflect any form of final results, I do believe that they demonstrate the effectiveness of the algorithm in a variety of different cases.

6.4 Sensor Data Addition

Since the end of the summer, my major goal for improving the performance and fidelity of the Dijkstra path planner was to implement the addition of a ray traced sensor data simulation into the path planning process. This would allow for optimal paths to be planned using a limited knowledge of the environment, resulting from obstacles obscuring the view of a lidar. This could then be used to try and demonstrate scenarios where limited sensor data could lead to collisions. In order to implement this, I developed two new method functions within the continuous sensor class. The first of these, get_sensor_data, inputs both the current node that the path planner is evaluating as well as the car info struct, which contains all of the information regarding threat producing cars on the road that is used to generate the true threat field. The function then runs two sensor sweeps, one at the time indicated by the node as well as one at one time step directly before the node time, in the case of the node time not having a previous time, then the time step following the node was used instead. This sensor sweep utilized the ray tracing algorithm described in its' section above. The first sweep began to populate an array called

car_info_sensed, which was similar to car_info, however, it contained only the position and velocity of cars that the sensor detected during this current time. The velocity of these cars was then estimated by numerically differentiating the two sweeps conducted, which was also added to the car_info_sensed array. This function successfully integrated the ray traced algorithm into a form that could be used for path planning and generated good estimates regarding the position and velocities of cars. The image shown below, compares two threat fields for the same scenario. The upper subplot shows a threat field constructed with the true threat information, whereas the other image is constructed from a limited ray traced threat information using this function.

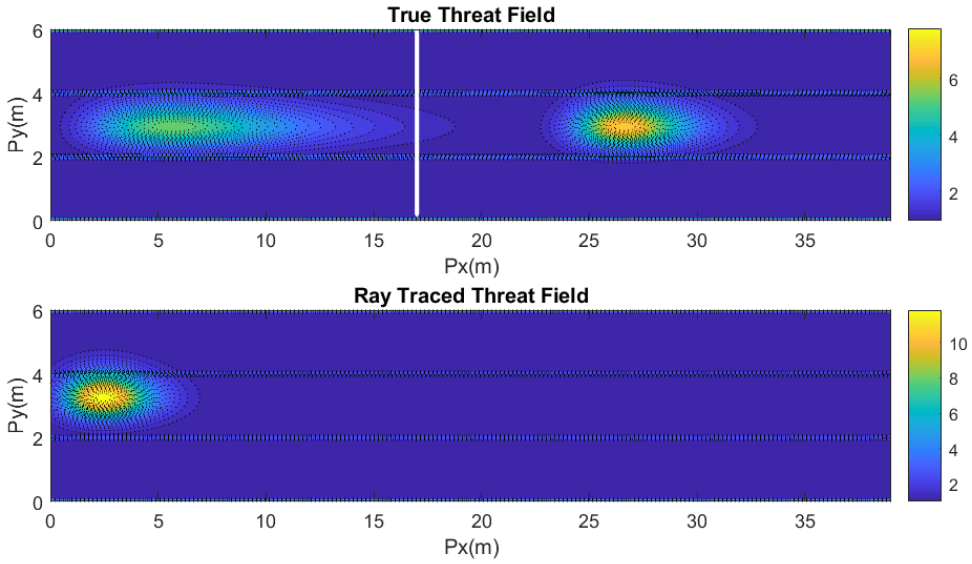


Fig. 4: Plot of true threat field and plot of sensed threat field.

With this ray traced function in place, I created one additional function which was needed to utilize get_sensor_data in path planning. When the algorithm is running and calculating the cost to travel from its' current node to a given adjacent node, it must first generate a value of threat for its' current node and for the adjacent node. Once it does this the cost is simply the average threat value of the two nodes. With the framework described above in place, the algorithm can run the get_sensor_data function to calculate a threat of a current node, but it must estimate the positions of threat producing cars at the adjacent node in order to calculate their threat. In order to accomplish this, I created a function called update_car_info_sensed, where the function inputs a simulated set of sensor data, generated from the above function, and estimates the position and velocity for all cars at this future node. To do this I assumed that the threat producing cars would have a constant velocity and heading angle over the

time span between nodes and used basic kinematics to calculate the car's positions from this information and the time difference between the nodes. Once this function ran, the updated sensor information could then be inputted into the get_threat function and the threat for a future node could be easily calculated. Once both the threat of the current and future node was calculated, the algorithm could then plan paths using a threat field iteratively produced using sensor data.

6.5 Heuristic Implementation

While using the method for generating sensor data described above does add more functionality and fidelity to the path planning algorithm, it comes at the cost of decrease efficiency. I found that with these new functions in place, the algorithm's performance drastically dropped due to the time spent running the ray tracing algorithm repeatedly. In order to make use of the path planner feasible again, I had to begin converting it from Dijkstra's algorithm to an implementation of the A* algorithm through the use of Heuristics.

The first step in doing so was modifying the threat field itself to have an ambient cost associated with every node of the threat field. This is a threat that is uniformly added to the whole field which is small compared to the value of threat produced by obstacles and cars that serves the purpose of penalizing longer paths, when compared to shorter paths. This inclusion alone helped to marginally improve performance, as the algorithm would not investigate paths that previously had no cost associated with them but traveled away from the destination.

The A* search algorithm is very similar to Dijkstra's, however, instead of selecting future nodes based upon cost alone, it uses the sum of the cost and an artificial cost called a heuristic. This heuristic must be carefully developed in order to ensure optimality and efficiency, which are often competing factors and one must choose an acceptable balance between the two. If the heuristic of a given node is ever higher than the minimum cost that must be paid in order to travel from the current node to the destination, then the algorithm will loose optimality as it will put more faith in this heuristic than the true cost and could potentially select higher cost nodes to make it to the destination in a shorter distance. In order to ensure that this would not happen and to calculate this heuristic, I implemented a function called get_heuristic, where a given node and the destination of the path planner were inputted and the function calculated the heuristic associated with

that node. In this case the lowest cost that one would ever pay to travel from any node in the field to the destination node would be the fewest number of nodes that the path would have to pass through on it's way to the destination times the ambient cost per node. This would occur in the case that no obstacles or threat cars were encountered, and the actor car only would have to pay a cost for distance. This, therefore, was how the function calculated the heuristic for a given node. Once the function calculated this heuristic for the adjacent nodes being investigated, the heuristic was saved along with the node hash number and cost. The cost and heuristic were then added together, and this formed the basis for sorting nodes in order to determine the next node to set as current.

With the addition of this heuristic, the algorithm was able to run much more efficiently as it checked far fewer nodes than without the addition. With this framework in place, it made it possible to run the sensor function on grids that were large enough to obtain meaningful results in a reasonable amount of time as opposed to before, when it was running the standard Dijkstra's algorithm.

6.6 Sensor Path Planning Results

Once I had implemented both new additions to the path planner, I was able to run more realistic scenarios to plan paths around. I was able to try several examples of differing scenarios, some of which led to collision free paths, however, with the right scenario parameters, the actor car did collide with a threat producing car. I also created a way to plot the path being followed by the car in real time when plotted against the time varying threat field as a way to visualize the data.

The main scenario of interest involved a car cutting in front of a self-driving car, while another car in front of it suddenly began breaking. The car that cut in front of the self-driving car, would then veer into another lane, however, by the time the actor car sensed the car in front of it, in the event that the other lanes around it were not free, would have no choice but to collide with the car in front of it. I was able to replicate this scenario within the path planning environment and it did indeed show the collision that I expected it to. This is shown in the figure below, where the red line shows the path the self driving car is taking.

A major next step to be implemented would be the addition of a function that would allow the cars to share sensor information with one another to see if the same collision could be avoided with sensor connectivity.

2019/path_planning_example_collision.PNG

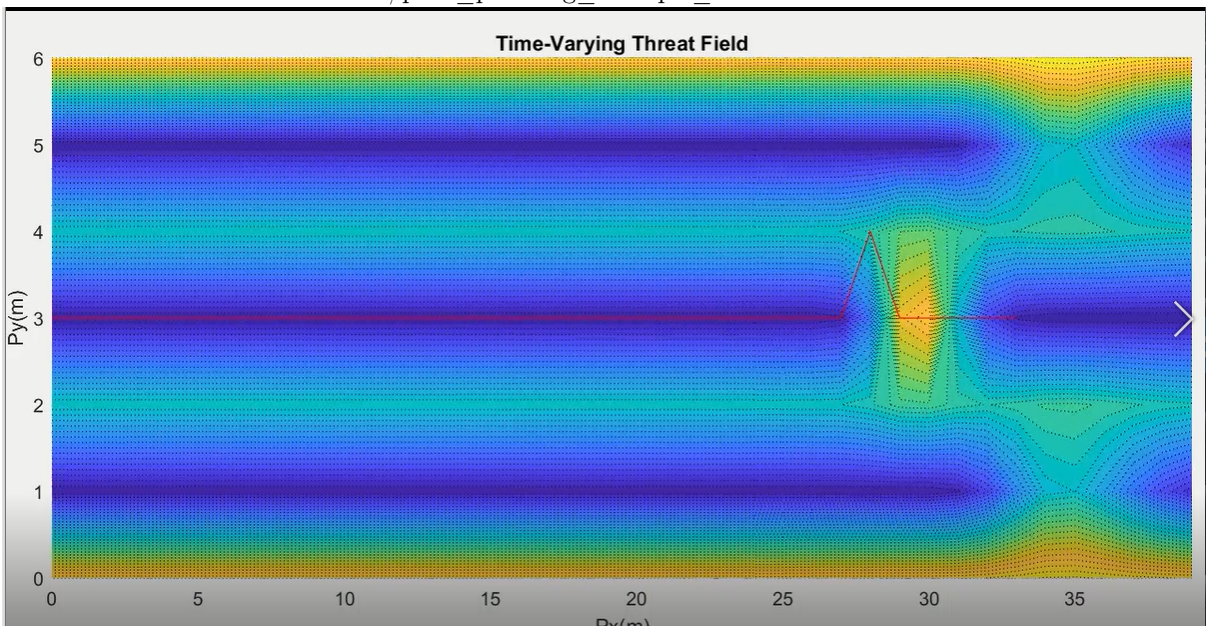


Fig. 5: Collision path planning example.

7 Time-Varying Transition-based RRT*

Another approach to the pathfinding component of this project was through the use of the RRT* algorithm. This algorithm traditionally operates on a non-continuous cost field, searching for optimal paths which avoid obstacles. In our case we needed a continuous cost field in order to implement Evan's vehicle cost model and to set lane-change costs.

7.1 Modification of the RRT* Algorithm

In order to do this I looked to "Transition-based RRT for Path Planning in Continuous Cost Spaces" written by Leonard Jaillet, Juan Cortes and Thierry Simeon [5] for insight on how to implement a continuous cost field into the existing RRT* algorithm. They developed a modified RRT* algorithm called Transition-based RRT or (T-RRT). Using the PythonRobotics [6] RRT* algorithm I modified the RRT* algorithm to become T-RRT*. In this algorithm, a leaf is defined as the last physical point, or node, of a branch. A branch is defined as a set of connected nodes, and a tree is the set of all existing branches. The T-RRT* algorithm differs from RRT* by using a function which adjusts the probability that a randomly chosen node will become a leaf of the tree. The

probability function is defined by Jaillet et. al. and is shown below:

$$p_{ij} = \begin{cases} \exp\left(-\frac{\Delta c^*_{ij}}{K*T}\right) & \text{if } \Delta c^*_{ij} > 0 \\ 1 & \text{otherwise} \end{cases} \tag{8}$$

This "transition test" equation compares the nearest existing node in the tree to a new potential leaf. The change in threat between these two nodes with respect to the physical distance between them Δc^*_{ij} is taken, where Δc^*_{ij} is defined in equation 9.

$$\Delta c^*_{ij} = \frac{c_j - c_i}{d_{ij}} \tag{9}$$

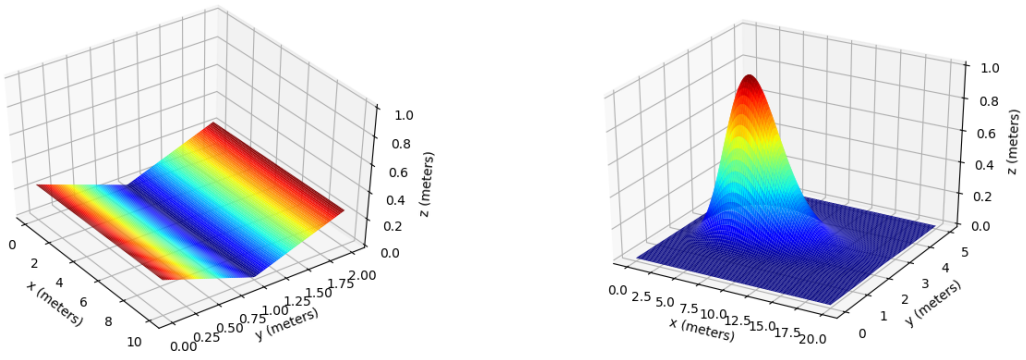
A probability p_{ij} which is a function of this value along with two coefficients T and K is then calculated, where T is the "temperature," or likelihood of the path branching to higher threat areas, and K is a constant defined as the average threat between the start and goal nodes:

$$K = \frac{c_{initial} + c_{goal}}{2} \tag{10}$$

The resulting value of this equation becomes the probability that the new potential leaf will be considered for that existing nearest node. This effectively minimizes the change in slope of the tree while it explores the map, and has a significant impact on the resulting path. Jaillet et. al. defined this as a work minimization algorithm rather than an integral cost minimization since it reduces the change in threat rather than the total accrued threat.

The work minimization behavior of the T-RRT* algorithm is useful for the way we define lane lines, barriers, and vehicle cost models. The lane, shown below, is defined as an inverted triangle with user-defined length, width, and maximum threat. The cost at the center of the lane is equal to zero, and the maximum cost is located on the edges of the lane. The actor cost model, also shown below, uses the log-normal distribution function defined previously, with its maximum threat normalized to 1.0.

The work-minimization equation is useful because it will minimize the time the actor is ever on or near the edges of the lane. This, along with a user-defined actor length and width, will pull the actor to stay near the center of a lane if it is not changing lanes. As with the 3D Dijkstra's algorithm, I needed to incorporate time into the algorithm. I used the same cost layering system as discussed earlier, except each cost layer is now defined as a continuous cost field rather than a parameterized one. I will shorten the name of the time-varying T-RRT* algorithm to T-RRT*-TV.



(a) (b)

Fig. 6: (a) Lane cost model, (b) Height-normalized lognormal vehicle cost model

7.2 Motion Constraints

To ensure the T-RRT*-TV searching algorithm finds a feasible vehicle path it must be given a number of physical constraints defined through vehicle dynamics. I constrained the algorithm to only allow branching in ways which satisfy the physical constraints imposed upon it. The constraints allow the algorithm to mimic the motion of a given vehicle resulting in realistic paths for any given scenario. They work by comparing a potential new leaf to the closest existing node on the tree, then calculating what the resulting speed, acceleration, difference in angular position, and angular velocity would be if that potential new leaf is chosen. If there are no existing nodes on the tree which satisfy the imposed constraints for that potential leaf, that potential leaf is considered not adjacent to any node of the tree, and will be discarded.

Previously, the method for choosing random points to branch to was simply to acquire a random point anywhere on the map, then perform the motion constraint check. Choosing any random point on the map resulted in significant run times (on the order of tens of minutes) because the majority of these random points would be nowhere close to meeting the motion constraints for the existing leaves of the tree. To solve this issue, I needed to limit the area on the map in which to choose a random point. For each node in the tree I defined a minimum and maximum extension radius along with a minimum and maximum extension angle. The extension radii and extension angles for each node were defined using the following methods:

```
1 def get_expansion_angle(self, node):
```

```
        node.exp_angle = [
            node.psi + self.steer_range[0],
            node.psi + self.steer_range[1]
        ]
def get_r_bounds(self, node):
        dt = self.map.t_step
        a = self.accel_range

        r_accel = [
            node.speed*dt + 0.5*a[0]*dt**2,
            node.speed*dt + 0.5*a[1]*dt**2
        ]
        r_speed = [
            self.speed_range[0]*dt,
            self.speed_range[1]*dt
        ]
        node.r = [max([r_accel[0], r_speed[0]]), min([r_accel[1], r_speed[1]])]
        if node.r[0] > node.r[1]:
            node.r = np.flip(node.r)
```

Summarizing the code above, we can see that a node's expansion angle is defined as its heading angle ψ added to the user-defined steering angle bounds. The expansion radii r are computed by comparing the distance that would be traveled given the vehicle is travelling at the minimum or maximum allowable speed, and at the minimum or maximum allowable acceleration. For the maximum extension radius, whichever outcome produces the larger distance will be chosen. For the minimum extension radius, whichever outcome produces the shortest distance will be chosen. This code defines a circle sector for each node. The area and position of this sector on the map signifies the range of locations that produce viable connections to that node, given the simple motion constraints.

Rather than choosing a random point on the entire map, the planning algorithm now chooses a random point within a random node sector. This ensures that every randomly chosen node will always be a physically viable option for the path planner.

7.3 Minimization of Jerk and Steering Rate

The newly defined node sectors alone ensure the path remains constrained in steering angle, speed, and acceleration. However it is necessary to constrain the vehicle's jerk and steering rates as well. Previously, to do this I simply constrained jerk and steering rate to a minimum and maximum bounds along with steering angle, speed and acceleration. I modified this approach by instead defining probability functions similar to equation 8, shown in lines 4 and 13 of the code snippet below:

```
    ''' Steer rate minimization function'''
    def minimize_steering_rate(self, d_psi, k=0.2):
        d_t = self.map.t_step
        p = math.exp((-abs(d_psi)/d_t) / k)
        if random.uniform(0, 1) < p:
            return True
        return False

    ''' Jerk minimization function'''
    def minimize_jerk(self, node, accel, k=1.0):
        d_t = self.map.t_step
        jerk = (accel - node.accel) / d_t
        p = math.exp(-abs(jerk) / k)
        if random.uniform(0, 1) < p:
            return True
        return False
```

7.4 Refinement Control

Following a paper *Enhancing the transition-based RRT to deal with complex cost spaces*, written by Didier Devaurs, Thierry Siméon and Juan Cortés I created a method of reducing the number of "refinement nodes" within the tree as a way to force the planning algorithm to mostly "explore" rather than "refine." The refinement control method will prevent any node from developing more than the user-specified maximum allowable children within the tree. Adjusting this maximum allowable number of children per node will increase or decrease the planning algorithm's tendency to explore, and inversely, to refine along the path. The aforementioned paper further develops their refinement control function so that along different points in the path the refinement control will adjust the ratio between "refinement" and "exploration" [7], which allows for a sufficiently-explored path with more refinement near the path's goal. I did not implement this addition to my refinement control method, although it would be interesting to see how doing so would affect the generated paths.

7.5 Map and Scenario Definitions

I created two maps: a highway and a four-way intersection. Three lane lines were created between each of the lanes, each with a width of 1 meter. The first scenario I created using this map layout was a simple lane change on the highway with no other actors present. The figure below shows the branching and final path determined by the

algorithm:

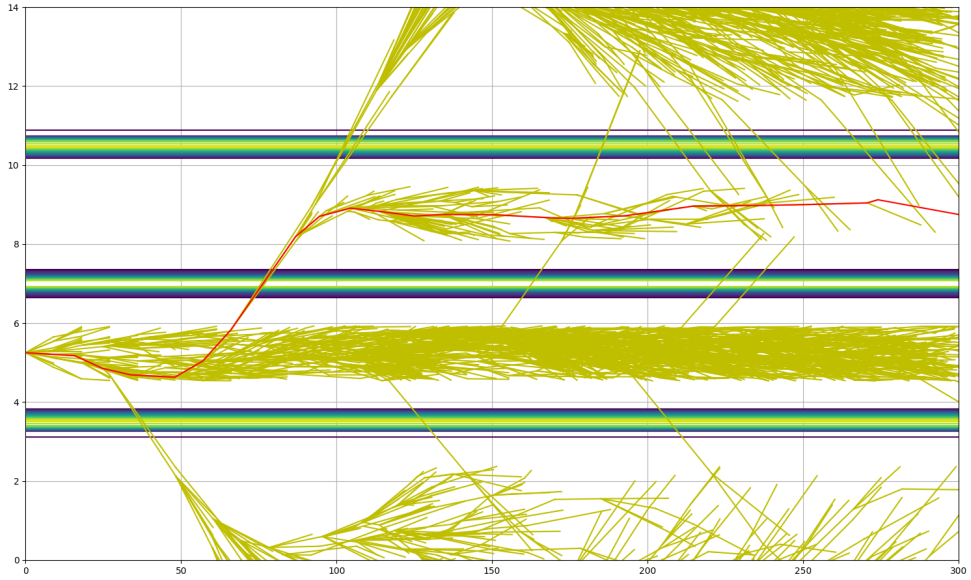


Fig. 7: Simple lane change scenario with no actor vehicles. Yellow lines show the map exploration, and the red line is the computed path.

It is apparent in Figure 7 that the algorithm began searching outside of the allowable map bounds, which was due to the original design of the RRT* algorithm. Searching was constrained to the map bounds in subsequent trials using barrier objects on each side of the road. Video snippets of various highway scenario paths can be found on the T-RRT-TV GitHub page.

The four-way intersection map was created to test the T-RRT*-T algorithm at lower speeds and with more dynamic driving scenarios, involving more acceleration and deceleration and larger steering angles. The intersection contains four lanes, each 3.5 meters wide and 15 meters long. The agent starts at the beginning of the leftmost lane, with a velocity of 10 m/s. An actor is positioned on the bottom lane, driving at a constant velocity forward to the top of the map. The agent's goal is to reach the top of the map without colliding with the actor, all the while satisfying the physical constraints defined for this scenario. The figure below shows the branching and resultant path chosen for this scenario. A video on the GitHub page of this scenario can be found as well, depicting the agent and actor's positions over time. The actor's velocity in that video is 2 m/s.

A highway scenario was also designed in attempt to showcase limitations to

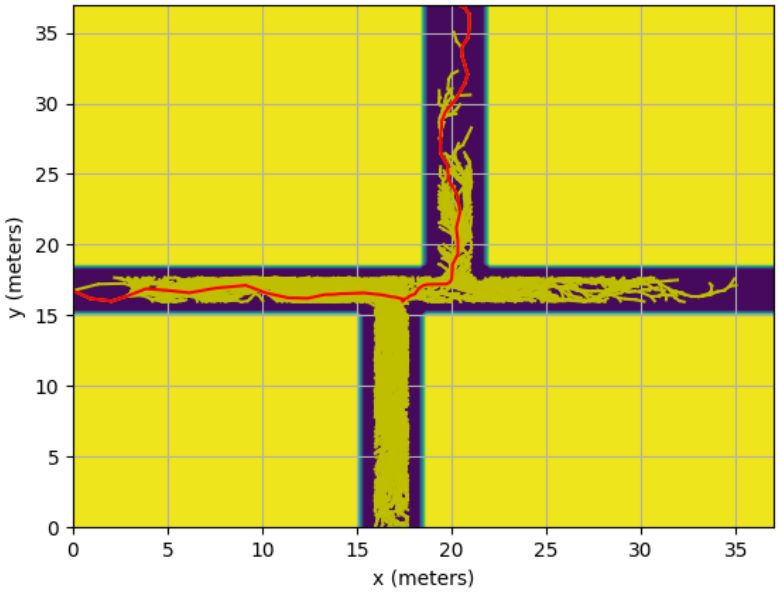


Fig. 8: Four way intersection. In this scenario an actor vehicle drove up through the top lane at a speed of 2 m/s

LiDAR vehicle sensing. The actor vehicles and map layout are replicated from Evan's 4D Dijkstra's highway scenario in order to allow for easy implementation of the resulting paths into the Carla driving simulator. A screenshot of the resulting path from the "full knowledge" (the ego vehicle is fully aware of any vehicles in its vicinity) is shown below. The "partial knowledge" scenario is when the ego vehicle relies on information just from the ray-casting mechanism, resulting in a collision. Videos from each of these scenarios may be found on the GitHub page.

7.6 Carla Simulator Implementation

It is important that all constraints and map dimensions introduced to the T-RRT*-TV algorithm are consistent with the Carla simulator since the goal is to incorporate it into the driving simulator. Adding onto Aditya's work this summer on configuring the Carla simulator I developed a way to input actor and ego vehicle motion information from both the modified RRT algorithm and from the 4D Dijkstra algorithm into Carla. I made a low-level controller to allow the vehicles in the simulation to track their respectively assigned paths. Videos of the implementations in the Carla simulator can be found at the T-RRT-TV github page. Screenshots from the simulations are shown below:

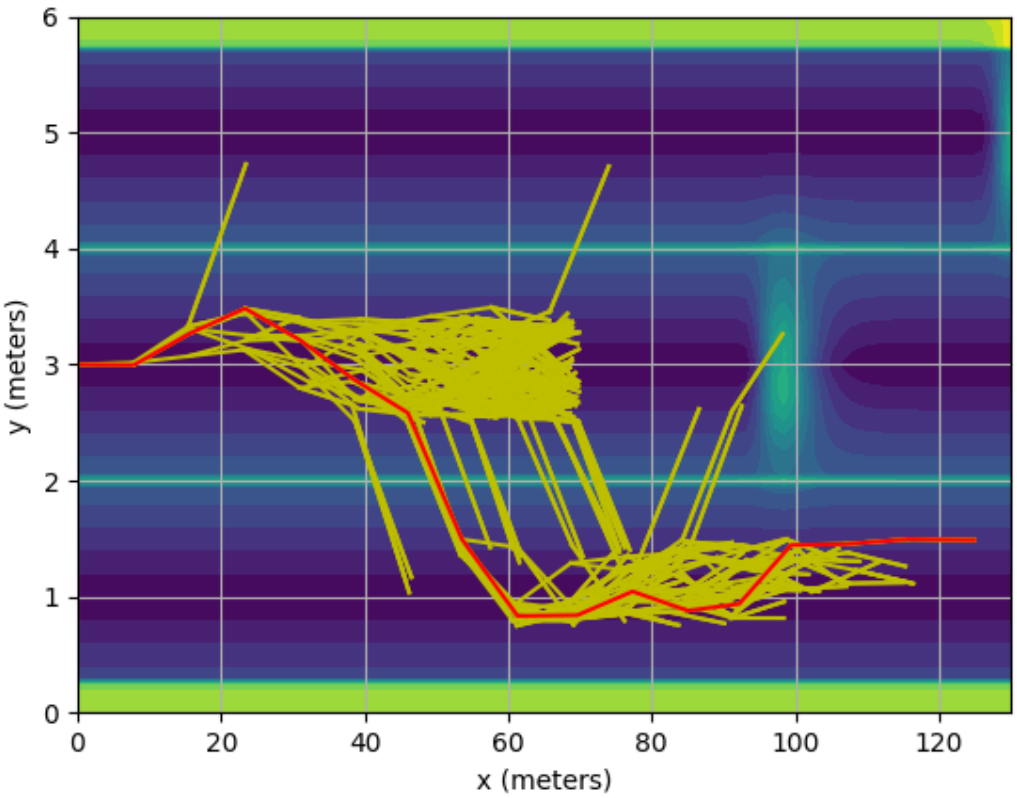


Fig. 9: "Full knowledge" scenario computed path

7.7 Limitations

A major limitation of this path planner is that it does not recompute another path after having determined one at its initial position. Rather, it computes a path then executes it regardless of any new environmental information it could gather along the way. Ideally this algorithm would be performed multiple times a second, constantly updating updated paths as the vehicle is driving, similarly to the RRT-based path planner developed at MIT by Yoshiaki Kuwata et. al., named *Motion Planning for Urban Driving using RRT* [8].

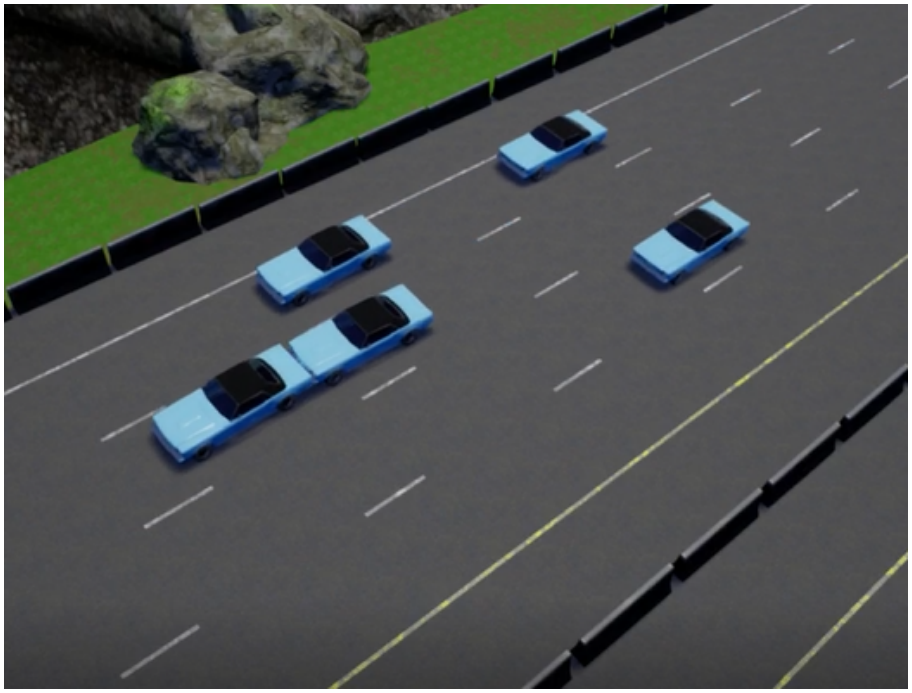


Fig. 10: Carla simulation showing collision event

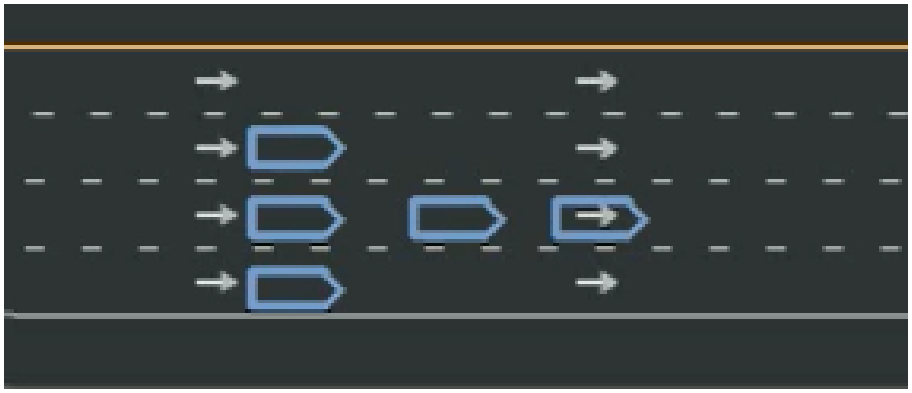


Fig. 11: "Hero" view of the simulation

References

[1] Jeff Hecht. Lidar for self-driving cars. *Optics and Photonics News*, 29(1):26–33, 2018.

[2] Carla simulator.

[3] Yung-Sung Cheng. Bivariate lognormal distribution for characterizing asbestos fiber aerosols. *Aerosol science and technology*, 5(3):359–368, 1986.

[4] Benjamin S Cooper and Raghvendra V Cowlagi. Path-planning with waiting in spatiotemporally-varying threat fields. *PloS one*, 13(8):e0202145, 2018.

[5] Léonard Jaillet, Juan Cortés, and Thierry Siméon. Transition-based rrt for path planning in continuous cost spaces. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2145–2150. IEEE, 2008.

[6] Atsushi Sakai. Pythonrobotics.

[7] Didier Devaurs, Thierry Siméon, and Juan Cortés. Enhancing the transition-based rrt to deal with complex cost spaces. In *2013 IEEE International Conference on Robotics and Automation*, pages 4120–4125. IEEE, 2013.

[8] Yoshiaki Kuwata, Gaston A Fiore, Justin Teo, Emilio Frazzoli, and Jonathan P How. Motion planning for urban driving using rrt. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1681–1686. IEEE, 2008.