ME5751 Final Project – CPP Truck Simulator

Ryan Chao
Cal Poly Pomona
Pomona, CA
rchao@cpp.edu

Cameron R. Weigel
Cal Poly Pomona
Pomona, CA
crweigel@cpp.edu

Abstract— **In this project, utilization of a PRM path planner, a Stanley controller, and an obstacle avoidance window contributed to the simulation of a truck navigating autonomously on a 2-D map. Key limitations to the motion planning were the truck's max steering angle and linear velocity. The truck followed paths relatively well, but depending on the sharpness of the yaw angle at given points, it would go off track in which it would eventually correct itself through the Stanley controller. Several other issues faced were path deviations due to obstacles, undesirable reactions to obstacles, and poorly generated paths. Compute times deviated greatly based on map complexity and obstacle positions with the limited amount of random points which resulted in multiple failed path generation attempts. Overall, the motion planner exceeded expectations since it was able to properly execute a motion plan and react when necessary.**

Keywords— robot motion planning; costmap generation; PRM path planner; Dijkstra's algorithm; Bresenham's line; KD tree query; Stanley controller; Dubin's curves; Ackermann steering; object window approach

I. INTRODUCTION

Robot motion planning consists of three major parts: costmap generation, path planning, and robot motion control. Costmap generation plays a pivotal role since it dictates the points the robot can travel in via assigned costs. To initialize this process, obstacle cells are inflated by the radius of the robot's body. Then a 4-connection brushfire algorithm is incorporated to assign costs ranging from 0 to 255 with lower costs representing free space and higher costs representing obstacles. A PRM-like method then populates a set of random points in a given free space which will be used to build a KD tree structure. We then establish a complete graph from the valid nodes and their respective edges, with each cost containing an associated cost of travel. Finally, we use Dijikstra's algorithm to search from a start point to a goal point on this graph, prioritizing the shortest, valid edges. From the path built, we apply Dubin's curve fitting to round sharp corners to smooth the planned path. The robot then incorporates the DARPA winning Stanley controller for Ackermann-steering robots to accurately follow the path. Finally, a double layer object detection window is utilized to avoid and react to obstacles.

II. BACKGROUND

i. Path Planning Methods

The A* search algorithm is a best-first search method which uses a heuristic function to determine the most optimal path. It presents a balance between the cost of the path from the start node to the next and the cheapest cost around the current node. Although A* demonstrates completeness and repeatability, it can become extremely time consuming depending on a given map's complexity and size. A* search is not utilized in this project.

Probabilistic roadmap planner is an alternative motion planning algorithm in robotics. It solves for the path of a robot from a starting point to goal point through the use of randomly sampled points in a given workspace. PRM planning consists of a construction phase in which a graph is built depending on the validity between points and a query phase in which the points are connected and a path is generated. Shortcomings for this methodology are due to its randomized approach to determining a path. However, because the opposite holds true, it can be an effective and efficient means of motion planning. For that, it is the motion planner for this project.

Rapidly-exploring random tree is an algorithm for path planning via a space-filling tree. This tree is built upon samples populated into a search space in which unsearched areas are prioritized. From a start configuration, a tree expands outward into free space and is built within set conditions. Drawbacks to this method are constructing a path that can be considered non-optimal and consuming an extensive amount of time. This method is not utilized in this project.

Dijkstra's shortest path algorithm is a method for determining the shortest path between nodes from start to goal. The algorithm initializes at the starting point and searches for the least cost edge. Once it has expanded to all points from the current, it moves on to the expanded points and iterates accordingly. If the path found prior is determined to be larger in cost, then the algorithm eliminates that edge. This continues until the goal has been reached.

ii. Controlling Methods

A proportional controller, p-controller for short, is a linear feedback control system which takes an input and outputs a corrective action. For motion planning, the steering angle acts as the input and affects the linear velocity to get back on track. For example, sharper turns would require slower speeds to achieve an appropriate turn motion. A main drawback to p-controllers is that they do not consider all variables that cause a robot to go off track. To improve this system, PI, PD, and PID controllers were developed to account for multiple variables. A p-controller did not serve as the main motion controller for this project. Although it did contribute to an active obstacle avoidance function.

The Stanley controller is a DARPA winning path tracking approach utilized by Stanford University. It utilizes the front axle as a reference point for detecting any heading or cross-track error. The controller emphasizes on eliminating these errors and then adjusts the robot's steering angle to be more inline with the planned path. Simultaneously, it acts within the bounds of the Ackermann-steering principles. The Stanley controller is utilized in this project.

iii. Dynamic Window Approach

The dynamic window approach is a velocity and acceleration-based method for avoiding obstacles. This is done by integrating the velocity and double integrating the acceleration to determine the displacements between the robot and its surroundings. If there is an obstacle, then the robot will output admissible velocities and steering angles within its own physical limitations. However, in order to achieve a good path, gains must be arranged for the target angle, distance, and velocity.

III. METHODS

Our methods can be split into three functional parts. Costmap generation, path planning, and robot control.

Our costmap generation is a simple inflation of obstacles based on the robot's geometry. We find the minimum circumscribed circle (MCC) that totally encloses the robot's profile. From there, we inflate obstacles by the radius of our MCC, the reason for this approach is that all measurements from our robot are in reference to the center of the robot. Obstacles are defined as pixel values of 255 in our costmap, all pixels inflated are assigned 255 values using a 4-neighbor connection method. We also perform a gradient cost value based on distance to the nearest obstacle pixel (closer to the obstacle, higher cost), but this is not used in this final implementation since we use a PRM-based planner for our path planner, versus a method like

Dijikstra's or A* which uses the cost moves of pixels to calculate an optimal path.

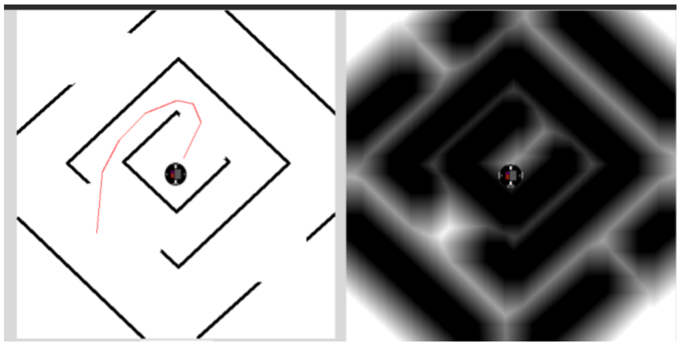


Fig 1: Map and Costmap side by side. Only black pixels are used in our path planner.

Our path planner is akin to a PRM planner but with the addition of a Dubin's curve fitting algorithm to smooth the sharp corners in the path. We start by randomly sampling *n* points on our map in free space. We then construct a KDTree of the sampled points. From that KDTree, we query the nodes to find possible connections between nodes. We then utilize the Bresenham line generation algorithm to check for obstacles when connecting a straight line between two nodes. If there is an obstacle, we do not include that in our graph structure. If there is a valid path from one node to another, we add that node and edge to our graph structure. Our edge has two attributes, the connection from node to node and an associated distance (think cost) between those two nodes.

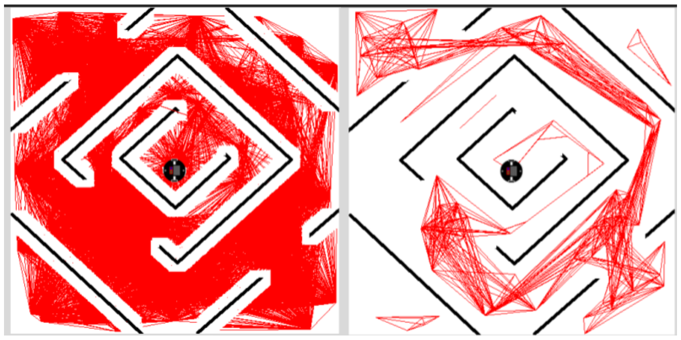


Fig 2: PRM-like planner without pruning (Left) vs the final graph structure computed on (Right)

Once we have processed the full tree, we use Dijikstra's algorithm to traverse from the start node to the end node. One issue with this method is that the edges become very sharp, for this reason, we utilize Dubin's curve method in an attempt to smooth the corners.

We utilize Dubin's algorithm by first removing a set number of pixels from both line segments that connect to a node. We then apply Dubin's algorithm to fit an arc between those now disconnected line segments in order to remove the sharp corner. The Dubin's algorithm finds the optimal (shortest) path between two vectors (x_1, y_1, θ_1) and (x_2, y_2, θ_2) using curve segments and straight segments to connect between the two poses. Those paths are defined as "RSR", "RSL", "LSR", "LSL", "RLR", "LRL". For example, "RSR" means a right turn followed by a straight segment, followed by a right turn. Whereas "LRL" means a left turn followed by a right turn, followed by a left turn.

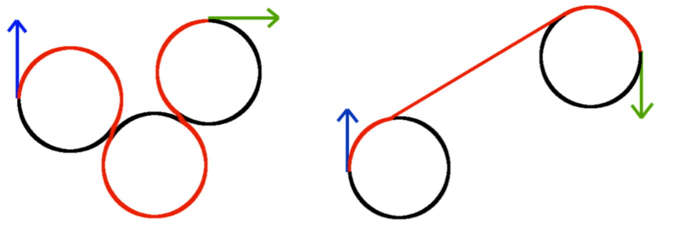
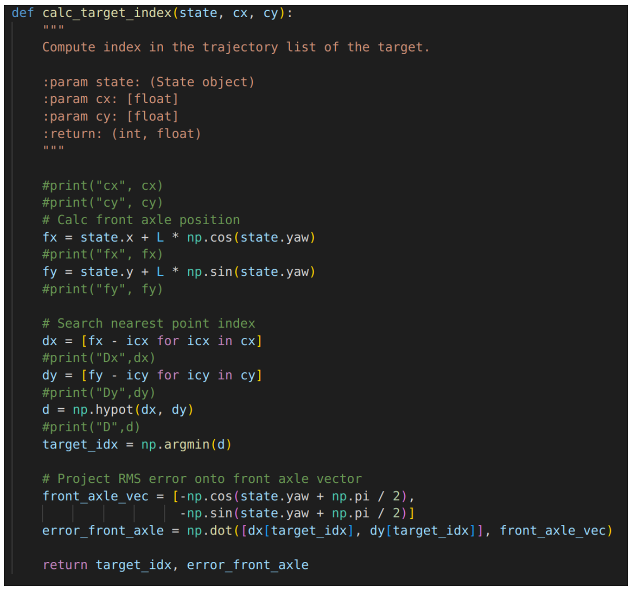


Fig 3: "RLR" path (Left) and "RSR" path (Right) generated by Dubin's algorithm, courtesy of [5]

Additionally, each trajectory point within our path planner has an associated theta value that is calculated based on the arctangent of the current and next trajectory point. These theta values are critical for our robot controller.

We utilize the Stanley controller developed by Stanford's autonomous racing team to control our robot with ackermann-style steering. The Stanley controller uses the current heading of the vehicle, the current steering angle of the vehicle, and the trajectory point's location and yaw value in order to converge to the correct path. This is done by minimizing the cross-track error, which is the error of the vehicle's heading in relation to the trajectory point's yaw.

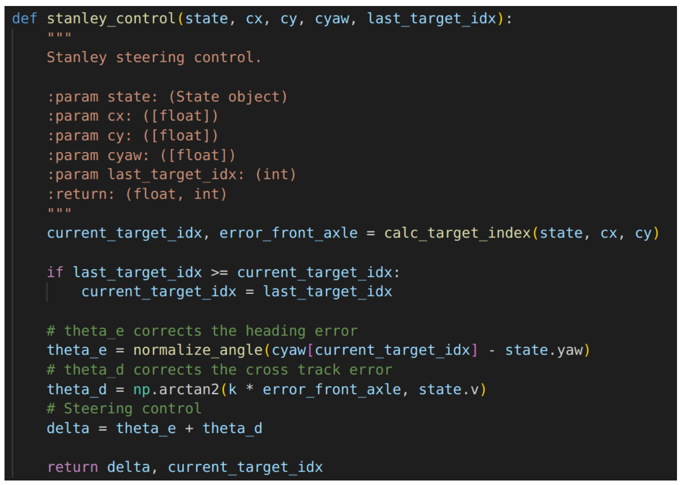
First, the closest target index needs to be calculated. The target index is the closest trajectory point to the vehicle's front axle, this is found by calculating the minimum hypotenuse between the vehicle and the trajectory points. Using the known target trajectory point, the front axle error is calculated through the dot product of the trajectory and the current front axle vector. This can be seen in the helper function below.



```python
def calc_target_index(state, cx, cy):
    """
    Compute index in the trajectory list of the target.

    :param state: (State object)
    :param cx: [float]
    :param cy: [float]
    :return: (int, float)
    """

    #print("cx", cx)
    #print("cy", cy)
    # Calc front axle position
    fx = state.x + L * np.cos(state.yaw)
    #print("fx", fx)
    fy = state.y + L * np.sin(state.yaw)
    #print("fy", fy)

    # Search nearest point index
    dx = [fx - icx for icx in cx]
    #print("Dx",dx)
    dy = [fy - icy for icy in cy]
    #print("Dy",dy)
    d = np.hypot(dx, dy)
    #print("D",d)
    target_idx = np.argmin(d)

    # Project RMS error onto front axle vector
    front_axle_vec = [-np.cos(state.yaw + np.pi / 2),
                      -np.sin(state.yaw + np.pi / 2)]
    error_front_axle = np.dot([dx[target_idx], dy[target_idx]], front_axle_vec)

    return target_idx, error_front_axle
```

Fig 4: Helper function for finding the target index and front axle error used in the Stanley controller. Courtesy of [3] for the base implementation

The output of this function is used in the final Stanley controller function to output our current closest trajectory index, and the steering angle required to converge to the trajectory path.



```python
def stanley_control(state, cx, cy, cyaw, last_target_idx):
    """
    Stanley steering control.

    :param state: (State object)
    :param cx: ([float])
    :param cy: ([float])
    :param cyaw: ([float])
    :param last_target_idx: (int)
    :return: (float, int)
    """
    current_target_idx, error_front_axle = calc_target_index(state, cx, cy)

    if last_target_idx >= current_target_idx:
        current_target_idx = last_target_idx

    # theta_e corrects the heading error
    theta_e = normalize_angle(cyaw[current_target_idx] - state.yaw)
    # theta_d corrects the cross track error
    theta_d = np.arctan2(k * error_front_axle, state.v)
    # Steering control
    delta = theta_e + theta_d

    return delta, current_target_idx
```

Fig 5: Stanley controller main function, outputs the desired steering angle and the current target trajectory index. Courtesy of [3] for the base implementation

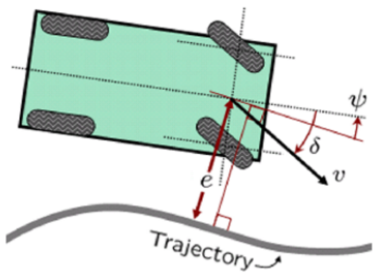


Fig. 2. Kinematic model of an automobile.

Fig 6: From [4], the kinematic model of a vehicle with cross-track error based trajectory following

Steering angle is adjusted to minimize this cross-track error in each time step. Limitations are built into the steering angle in order to simulate a vehicle's physical limitations. We use a simple proportional velocity controller based on steering angle. Our velocities are calculated as follows.

$$c_v = k_1 \cdot cos^2(\delta_{steering})$$

$$c_\omega = \frac{c_v}{L \cdot tan(\delta_{steering})}$$

where,

L	*: wheelbase of the vehicle*
$\delta_{steering}$	*: output of the Stanley controller*
k_1	*: velocity gain*

This ensures a slower velocity on tight turns and a near-maximum velocity on straightaways. This allows for a steering controller that can track its trajectory points accurately for our timestep.

Finally, to add some robustness to our robot, we implement a two-layer obstacle detection and avoidance algorithm. The outer layer in our two layer system checks pixels a certain radius R_{outer} such that;

$$R_{outer} > R_{MCC}$$

We then check those pixels for obstacle values on each time step. If an obstacle is found, it is determined to be on the left or the right side of the vehicle. If it is on the left side, we turn right at max steering angle for one time step and then check again and vice-verse for the if an obstacle is detected on the right side. This ensures that our robot can avoid obstacles if they are found on only one side of the vehicle. In the case of the robot moving into a corner and detecting obstacles on both the left and right side, our second layer handles these cases. The second layer has radius R_{inner} such that;

$$R_{MCC} < R_{inner} < R_{outer}$$

If an obstacle is detected in this layer, the robot is instructed to reverse with zero steering angle for one time step. The two layers working in tandem perform a n-point turn in corners in order to move out of them. Visualization of the front-only double layer obstacle detection below.

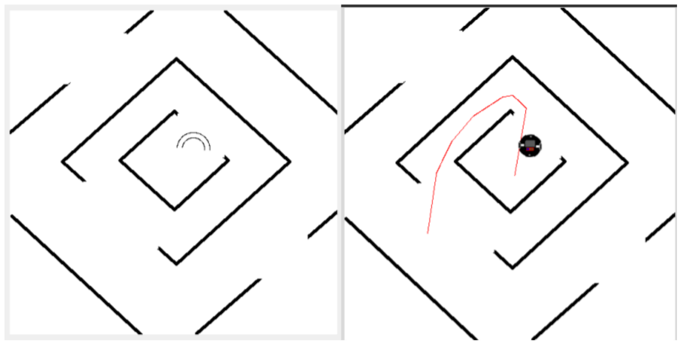


Fig 7: The two layers of obstacle detection visualized (Left), the robot's current position (Right)

IV. RESULTS AND DISCUSSION

The results and discussion of the project will consist of testing a single custom-built map to demonstrate the complexities and considerations required of our motion planner. Metrics to be analyzed will be path planning, track efficiency, and obstacle avoidance over several trials.

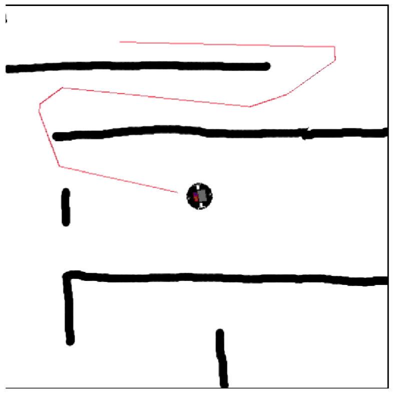


Fig 8. First iteration generated path.

In the first iteration, the planner generates a relatively good path. One issue the motion planner runs into is the number of randomly sampled points ends up not being enough to generate a valid path which results in multiple reattempts. From the path shown in figure 8, there is a questionable edge due to its close proximity to the obstacle in the beginning. However, this is resolved through the object detection window algorithm incorporated in the motion planner. Following this correction, the robot offshoots from the track to which the Stanley controller corrects the robot's steering so that it is able to go back on track. This is done so in the third and fourth frames. The time required to complete this path is 28 seconds.

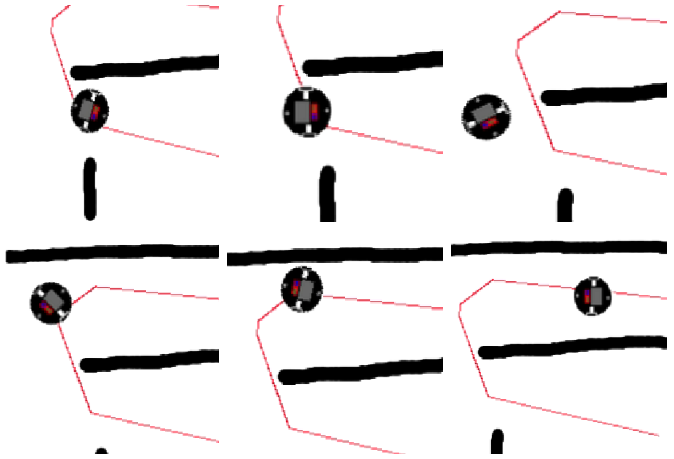


Fig 9. Sequential order of obstacle avoidance and Stanley controller implementation.

In the second iteration, the planner generates a less efficient path than the first as seen in Figure XX. It unnecessarily navigates under the wall to the left of the starting point. This is due to the limitations of Dijkstra's algorithm as it only selects the shortest path or lowest cost edge between the current point and the valid edges to it. When moving upward from this point, the robot detects that it is within the map's outer edge so it adjusts by steering to the right. This is observed since it steers right before reaching its next destination point as seen in the first and second frames.

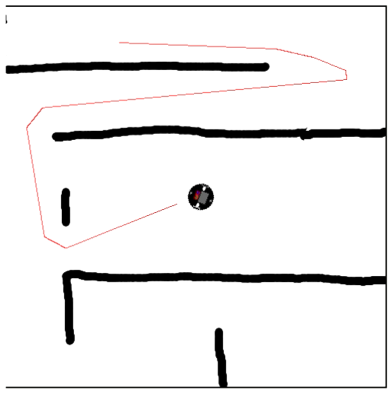


Figure 10. Second iteration generated path.

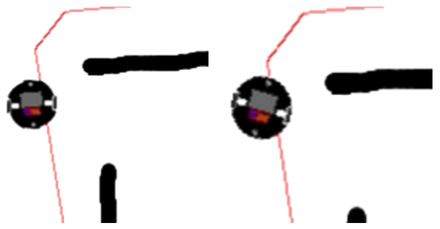


Fig 11. Example of obstacle avoidance from second iteration.

One issue between the motion controller and object detection window occurs at the end of the longest edge in between the two horizontal obstacle walls. Since the robot is moving at its maximum speed, it detects an object to its front left and reacts by steering right when the proper action is to detect a wall that is to its front right and steer left. The robot starts attempting to make many turns until it is able to get back on track. This is due to a yaw angle issue with the Stanley controller caused by the obstacle avoidance method. The reaction causes the robot to steer off track, but this is again, corrected by the Stanley controller. This path is completed in 40 seconds demonstrating the importance of generating an efficient path.

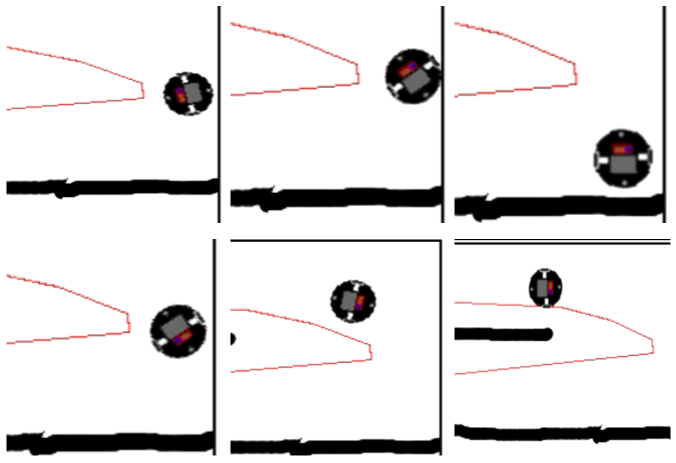


Fig 12. Frames of improper obstacle reaction followed by Stanley controller correction.

From the trials, the PRM path planner requires a much longer time than expected to determine a path. It also generates very sharp corners which are not within the limits of the vehicle's steering. Since our solution to resolving sharp corners is to simply subtract a certain amount of points from each edge, it does not solve the problem with extremely sharp corners. Thus a method for eliminating points from the next edge within a radius of the end point of the previous edge is necessary. With this, a smoother Dubin's curve can be generated to allow the robot to navigate corners more smoothly.

Improvement with the Stanley controller is also necessary as the trajectories are not smoothly transitioning due to the yaw angle. This issue typically occurs when the obstacle correction occurs which causes the robot to keep moving perpendicular to the path until it is within the yaw angle limits of the Stanley controller. It would then attempt to reduce the cross track error until it is within tolerance i.e. alignment with the path.

V. *CONCLUSION*

The combination of a PRM path planner, a Stanley controller, and an object detection window resulted in an effective motion planner. From the test map and its multiple iterations, the planner is able to produce relatively nice paths that the robot tracks adequately. However, if the map had enough complexity, it would try to reiterate until a valid path was determined or terminate the trial. There are plenty of improvements that could be incorporated to improve its path generation quality and time. Restricting the KD tree query to a limited radius would improve time significantly since it currently queries every sample point to one another. Additionally, the Dubin's curve planner could be improved by reducing the length of edges within the steering angle radius of the end point of the previous edge. Adjusting the Stanley controller to recognize an increasing cross track error would assist in quicker corrections. Upon making these improvements, the motion planner would achieve much better handling and run times.

VI. *REFERENCES*

[1] R. Siegwart, I. Nourbakhsh, D. Scaramuzza, Introduction to Autonomous Mobile Robots, 2nd ed., The MIT Press : Cambridge, 2011.

[2] Y Chang, California Polytechnic University, ME5751 Lecture Slides, Fall 2022

[3] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, Alexis Paques, PythonRobotics: a Python code collection of robotics algorithms https://doi.org/10.48550/arXiv.1808.10703

[4] Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A. and Mahoney, P. (2006), Stanley: The robot that won the DARPA Grand Challenge. J. Field Robotics, 23: 661-692. https://doi.org/10.1002/rob.20147

[5] Andy G, A Comprehensive, Step-by-Step Tutorial to Computing Dubin's Paths