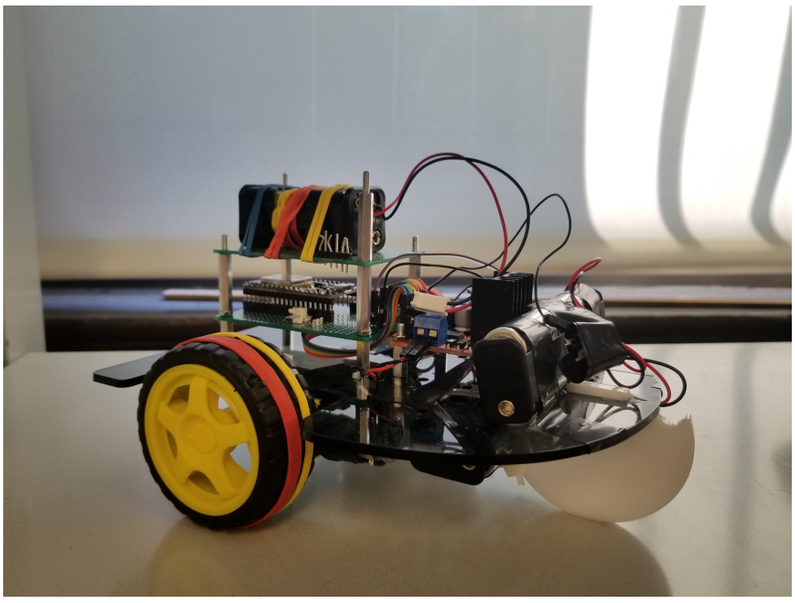
Differential Drive Kinematics and Path Planning for Two-Wheeled Mobile Robot

MEAM 520: Introduction to Robotics

Hyun Kim, Xiran Xu, Woohyeok Yang

December 2019



Contents

1 Introduction

In this semester, all of our team members are taking MEAM 510 and we are asked to build the race car for the class assignment. This encouraged us the final project idea for understanding the kinematics of the two-wheeled mobile robot and building the trajectory planning for the mobile robot. In this paper, we will introduce the differential drive mechanism for a two-wheeled mobile robot. Then we will derive the forward and inverse kinematics of the mobile robot using the simplified geometry model. Finally, we will develop the trajectory planning for the mobile robot using a Rapidly-exploring Random Tree* (RRT*) planner and introduce an example of the real-life path planning for two-wheeled robot using the vive pulse and Arduino, a computing platform.

2 Literature Review

2.1 Differential Drive Kinematics

Many of mobile robots use a differential drive mechanism, where the direction of motion is controlled by two drive wheels with controlling speeds of the left and right wheels. The kinematics model of a mobile robot is shown in Figure 1 [6]. For two-wheeled mobile robot, each wheel is mounted on a common axis. Let us assume the robot move on the 2D planer surface without slipping. The center of the robot is denoted by (x, y, θ) and is located in the midpoint between the left and right wheels where (x, y) indicates the coordinate position of robot on the planer axis and θ is the heading angle of the robot. The width of the robot is denoted as l and the linear velocities of each left and right wheels are denoted as v_l and v_r respectively.

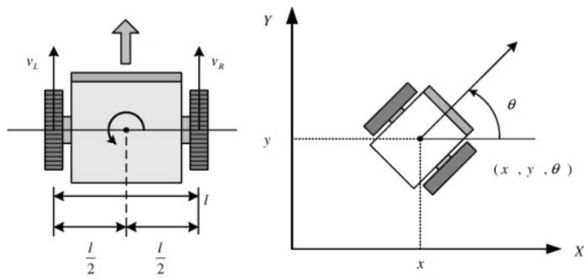


Figure 1: A kinematics model of two-wheeled mobile robot.

When the robot rotates, the common axis must rotate about a point that is a center of that curvature at that moment. This point is known as the Instantaneous Center of Curvature (ICC) and is shown in Figure 2 [7][6].

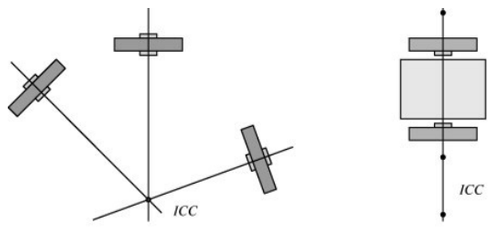


Figure 2: Instantaneous center of curvature (ICC) for a differential-driving mobile robot.

2.2 Forward Kinematics

Consider the two-wheeled mobile robot is located in the some arbitrary position A with (x, y, θ). By controlling the v_r and v_l, the new position B of the robot at time $t + \delta t$ is shown in Figure 3 [6]. The new position of the robot is denote as (x', y', θ') and we can find (x', y', θ') in two ways.

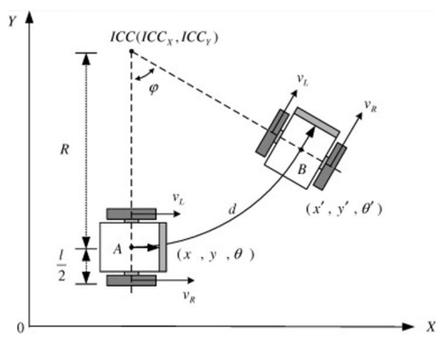


Figure 3: The location of ICC along with the robot's motion from position A to B at time $t + \delta t$.

First we will find the equations for the angular ω and R, the radius of the instantaneous curvature of the robot trajectory relative to the center of the motion. We know for a circular motion at radius R, the angular velocity ω is

$$\omega R = v. \tag{1}$$

Using equation (1), we can write the velocity for right and left wheel as

$$v_r = \omega(R + l/2) \tag{2}$$

$$v_l = \omega(R - l/2). \tag{3}$$

Re-arranging the equation, we can solve for R and ω as

$$R = \frac{l}{2}\frac{v_l + v_r}{v_r - v_l} \tag{4}$$

$$\omega = \frac{v_r - v_l}{l} \tag{5}$$

For the first method, we can find the new position using transformation matrix. When the mobile robot is moving from position A to B, using the basic trigonometry, the coordinates of the ICC can be written as

$$ICC = \begin{bmatrix} ICC_x \\ ICC_y \end{bmatrix} = \begin{bmatrix} x - Rsin\theta \\ y + Rsin\theta \end{bmatrix} \tag{6}$$

Then using the rotational matrix, the new position of the robot (x', y', θ') at time $t + \delta t$ can be computed as following

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} cos(\omega\delta t) & -sin(\omega\delta t) & 0 \\ sin(\omega\delta t) & cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix} \tag{7}$$

where the radius R and the angular velocity ω can be calculated using equation (4) and (5).

The second method, which is more common method is using the integration and differential equations. In this method, we will break the robot's motion into small time segment dt where v_r and v_l are constant at dt. Then perform the integration to obtain the position. The new position of the robot (x', y', θ') at time $t + \delta t$ can written as following

$$x' = x(t) = \int v(t)cos(\theta(t))dt \tag{8}$$

$$y' = y(t) = \int v(t)sin(\theta(t))dt \tag{9}$$

$$\theta' = \theta(t) = \int \omega(t)dt \tag{10}$$

where $v(t)$ and the angular velocity ω can be calculated using equation (1) and (5).

2.3 Inverse Kinematics

We know from forward kinematics there are many numerical solutions. For example, as shown in the Figure 4 [5], when the arbitrary starting and final position are given, there are numerous path that the mobile robot can take. The goal of inverse kinematics is to find the best solution that gives the quickest time and most energy efficient solution. To do so, we will decompose the problem and control only a few degrees of freedom (DOF) at a time.

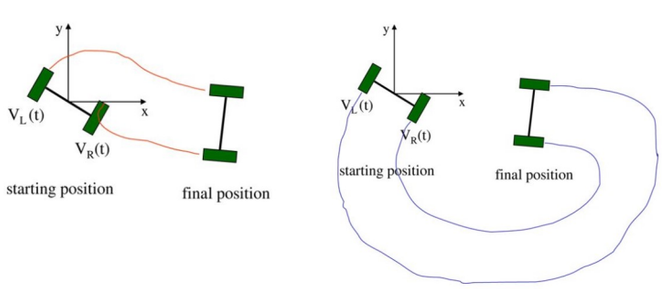


Figure 4: Two example solutions for the given arbitrary starting position and final position.

Consider the basic driving motion of the two-wheeled mobile robot. The equation (4) and (5) demonstrate the basic drive motion of the mobile robot in four cases.

1. If $v_l = v_r$, then R becomes infinite and $\omega = 0$.
   In this case, the robot will move in straight line with no rotation.
2. If $v_l = -v_r$, then $R = 0$ and $\omega = -2v_l/l$ or $\omega = 2v_r/l$.
   In this case, the robot will rotate about the midpoint of the wheel axis and the robot just rotate in its place.

3. If $v_l = 0$, then$R$ becomes $1/2$
   In this case, the robot will rotate along the left wheel.

4. If $v_r = 0$, then$R$ becomes $-1/2$
   In this case, the robot will rotate along the right wheel.

Based of the drive motion of the robot, note that a differential drive robot cannot move in the direction along the common axis. In other words, the robot cannot move sideways. This is a singularity of the differential drive robot [1].

To find the best solution, we can decouple the motion into the basic driving motion listed above. For example, for the given starting position and final position shown in Figure 4, the robot path can be decomposed into 3 basic motions shown in the Figure 5 [5].

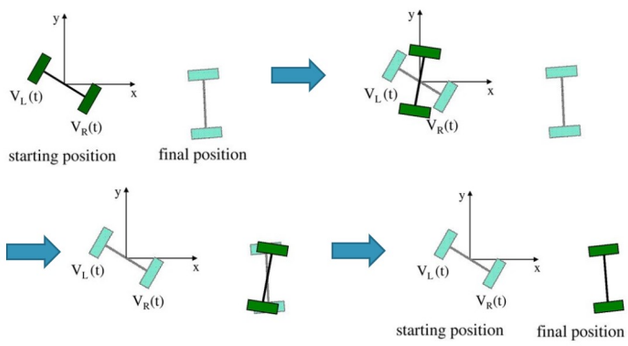


Figure 5: The inverse kinematics of the robot' motion at given start and final position using the basic motion of the mobile robot.

3 Path Planning for Two-wheeled Mobile Robot

For the final game of MEAM 510, we are asked to do a path planning for our mobile robot for the 60 seconds autonomous mode at the beginning of the competition. The arena for the game is shown in Figure 6.

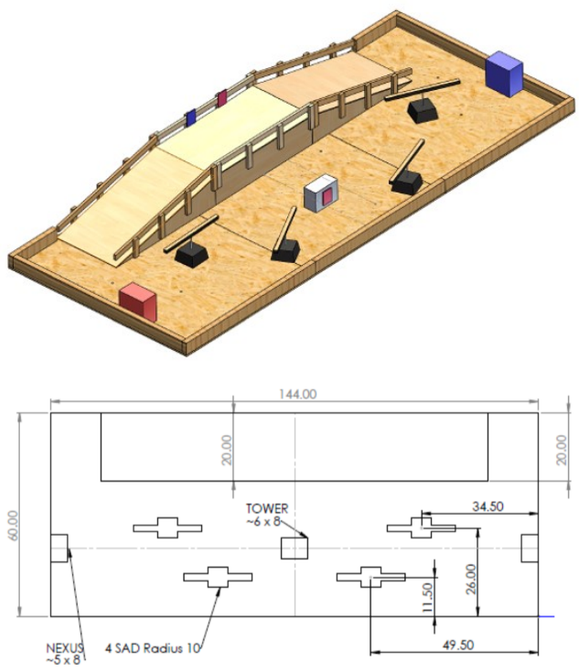


Figure 6: The MEAM 510 final project game arena.

The goal of our robot is to attack the opposing team's nexus (home base shown on two sides, red and blue), go up the bridge(shown on the top) to hit the buttons, or take over the tower (in the middle) to deplete the health of the opposing team. Four obstacles with rotating arms are located as shown in the picture. However, in autonomous mode, the four moving obstacles were not moving. The robot will be randomly placed on the arena and there is a 60 seconds time limit. Due to above facts, we seek to find a path planning algorithm that allow our robot to move to the desired location as fast as possible without colliding with or hit by any of the obstacles.

In LAB3 we implemented the Rapidly-exploring Random Tree(RRT). This time we choose to use RRT* since it provides a significant improvement on the quality of the path planned. The path planned using RRT* will

have lower cost and will be more smooth.

RRT* is an optimized version of RRT. The basic principle of RRT* is the same as RRT, but two key features of the algorithm result in significantly improved result. First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the cost of the vertex. After the nearest vertex is found, vertices in a fixed radius from the new vertex are examined. Instead of choosing the nearest vertex as the parent of the newly added vertex, RRT* chooses the best neighbor(vertex with lowest cost) as the parent vertex. The second difference is an added step to change the tree in order to reduce path cost using the newly added vertex. If any neighbor vertex's cost decreases after being rewired to the newly added vertex, we rewire them.[4]

4 RRT* Pseudocode and MATLAB Code

4.1 Pseudocode

1. Initialize tree *vertices*.
2. Insert vertex v_{start} into *vertices*.
3. For $i \leftarrow 1$ to number of vertices
   (a) Pick a random vertex v_{rand}.
   (b) Find the nearest vertex $v_{nearest}$ to v_{rand}.
   (c) Set $v_{nearest}$ as v_{min}, $cost(v_{nearest}) + cost(v_{rand})$ as $cost_{min}$.
   (d) for all vertices *neighbors* in the neighborhood with given radius, do
      i. Steer from all *neighbors* toward v_{rand} with step size, use interpolation to get new vertex v_{new} if the distance between two vertices is greater than the step size. Otherwise v_{new} is v_{rand}
      ii. Check if there is any collision and the cost for each path. If there is no collision and the path has lower cost than $cost_{min}$, update that $v_{neighbor}$ as v_{min} and update $cost_{min}$ as $cost(v_{neighbor}) + cost(path)$.
   (e) Insert v_{new} to *vertices*
   (f) Find the parent vertex for the goal, wire them if no collision found on the path

4.2 Brief description of RRT* Algorithm

1. Define the size of the map and the location of obstacles, plot the map.
2. Initialize the coordinate, cost and parent of start and goal.
3. Insert first vertex v_{start} in the list *vertices*.
4. Pick a random vertex v_{random}.
5. Find the nearest vertex $v_{nearest}$ to the random vertex v_{random}.
6. Steer from $v_{nearest}$ toward v_{random}. If two vertices are too far away(greater than the step size), use interpolation to reach new vertex v_{new}.
7. Check if there is any collision with the obstacles.
8. Use $v_{nearest}$ as the parent vertex of v_{new}. Update the cost of going from $v_{nearest}$ to v_{new} and set it as the minimum cost $cost_{min}$.
9. Check for all vertices within a given radius *neighborhood*, insert them into a list *neighbors*.
10. In all members of *neighbors*, check if v_{new} can be reached from a different parent vertex with cost lower than $cost_{min}$ without colliding with the obstacles. Update the parent vertex of v_{new} to the vertex with the least cost.
11. Add v_{new} to the list *vertices*.
12. Repeat above process until goal is reached.
13. Search backwards from goal to start to find the optimal least cost path.

4.3 Functions

1. calcDistance(v_1, v_2): calculate the distance between two vertices.
2. noCollision(v_1, v_2, obstacle): check if path from v_1 to v_2 intersects any of the four edges of a rectangular obstacle.
3. notCollided(v_1, v_2, obstacles): check if path from v_1 to v_2 intersects any of the obstacles in the map.
4. collisionCheck: cross product for collision check

4.4 Results and Analysis

We choose two cases to do the simulation to test the performance of the planner. The first case is to let the robot go up the ramp from a bottom corner of the arena. We ran the simulation multiple times and it always takes less than 60 seconds to find the path. The three simulations results for case 1 is shown in Figure 7.

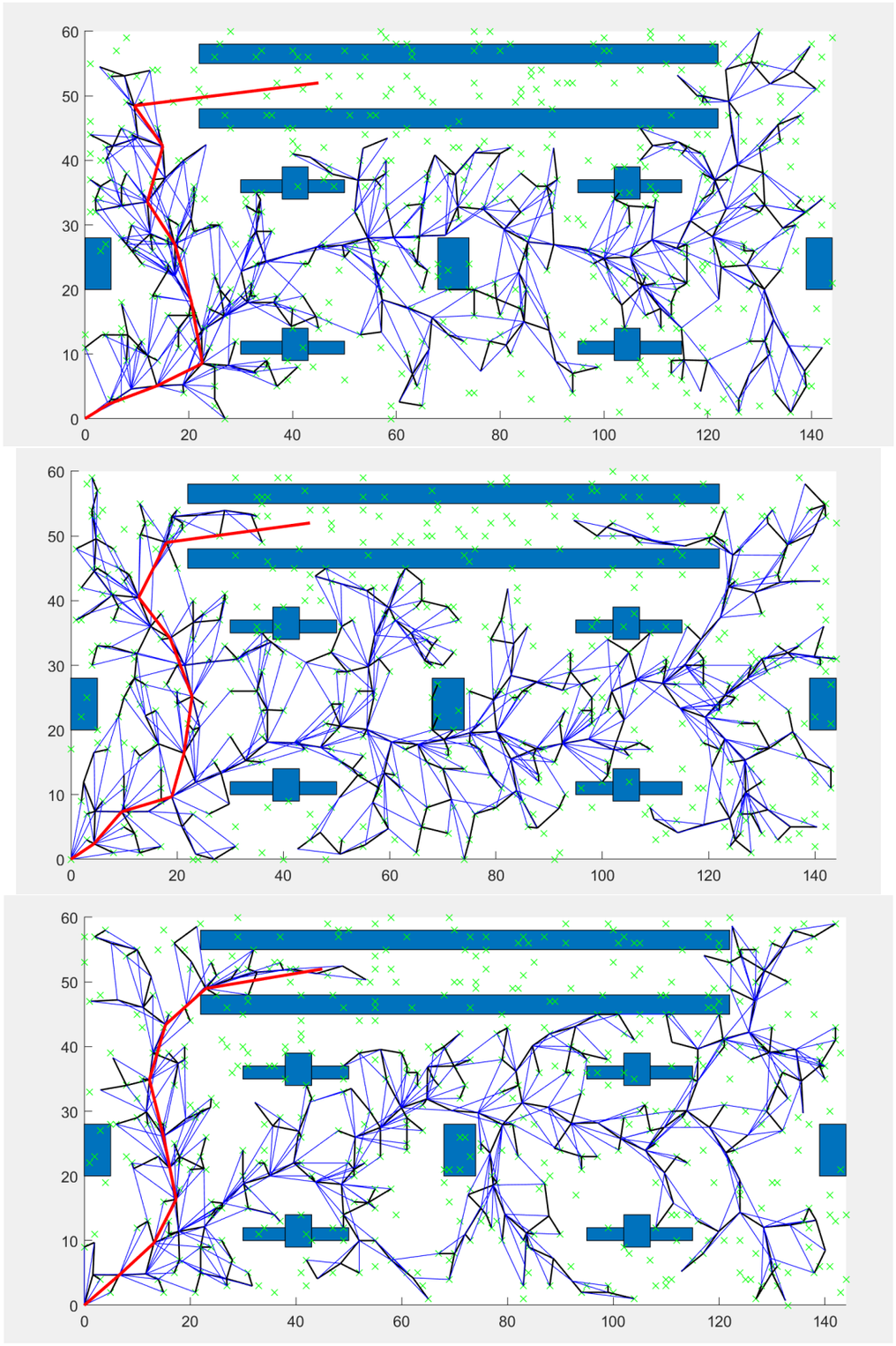


Figure 7: Path simulation going up the ramp from a bottom corner of the arena.

The second case is to let the robot go to the opposing team's home base from our own home base. Again, we ran the simulation multiple times and it always takes less than 60 seconds to find the path. The three simulations results for case 1 is shown in Figure 8.

As shown in Figure 8 and 7, the paths founded are very similar except some paths are more smooth than others. This is due the positions of the randomly generated vertices. We use 500 randomly generated vertices when doing the simulation. More vertices will give more smooth path, but it will need longer execution time. Although we only use 500 vertices, since we are always trying to choose the best possible parent vertex for each newly added vertex in the tree, the paths found are reasonably short and smooth.

However, there are several problems regarding our RRT* planner. The first problem is that we didn't

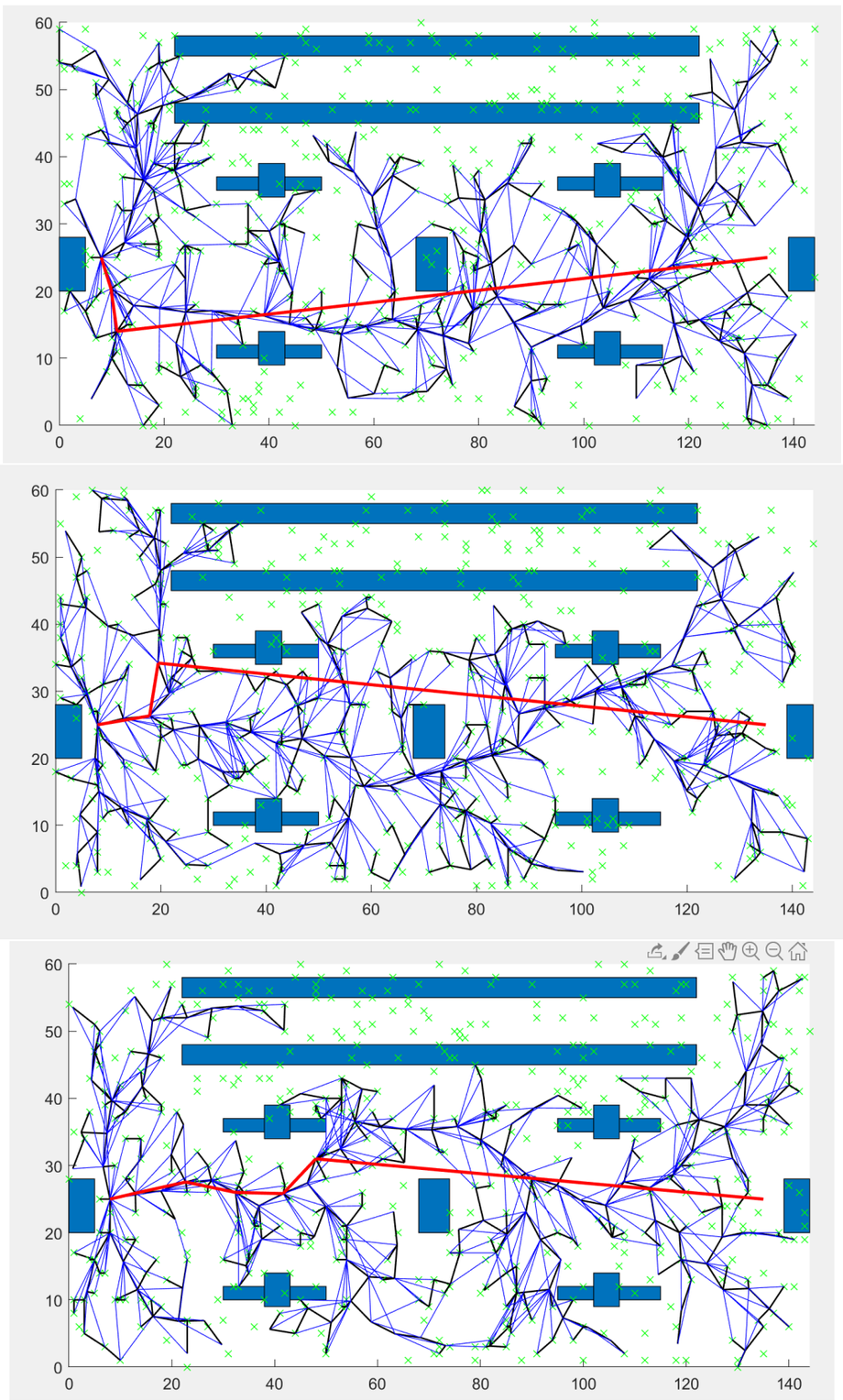


Figure 8: Path simulation going from its own home base to enemy's home base.

manage to accomplish the rewire process perfectly in our code. If we were able to finish the rewire process, the paths are expected to be more smooth and efficient. The second problem is that we didn't manage to take into consideration the moving obstacles. But we have some ideas inspired by papers and online resources. The first method is to implement potential filed planner when moving obstacles are detected. The second method is using dynamic path replanning. We will further modify the current planner, making it possible to be implemented in dynamic environments with moving obstacles.

5 Real-Life Application

5.1 Set Up

To implement the path planning plan to robot that we made for MEAM510, we used nodeMCU-ESP32 chip to make wireless connection and to operate. In order to locate the position, we used vive sensors. HTC vive uses so called lighthouse tracking system, this system utilize two infrared transmitter for x axis and y axis [2]. There are few operating modes in vive, but we used mode b for our settings. In b mode vive transmit two empty sync pulse and one sync pulse with x pulse and one sync pulse with y pulse. The sync pulse is

60Hz. Two photosensors were attach on top of the robot to detect the direction of the robot and get the robot's position (x, y) at the center of two wheels. In order to receive better signal, we used op amp to amplify the signal and converted analog signal to digital signal using comparator. Then we used Arduino, an open-source computing platform, to implement the path planning for the robot.

5.2 Path Planning

For this project, our goal was to go up the hill and conquer it no matter where the robot starts. We divided map into few areas and wrote scenarios for each case. When the autonomous mode starts, robot will gather its location from the vive and check if it is in the vicinity of goal. Then to decide whether the robot is on the bridge or not. If robot is on bridge just turn and will tries to go up to goal. Same with near the entrance of the bridge. If robot's initial location does not meet any of these cases then the robot will first try to move the entrance of the bridge and then move up to hill.

To determine if the robot have reached the goal, we used distance formula as shown below

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{11}$$

If the distance from goal to center of robot is within 100 unit of vive which is about 5 in, then the robot has reach the goal position.

5.3 Pseudocode

1. Get current location x and y.
2. While it reaches the goal($xgoal$, $ygoal$)
   (a) Determine which side it started (red or blue)
   (b) If it is on the bridge line
      i. Move to the goal
   (c) It is at the entrance of hill
      i. Go up the hill and to the goal
3. If it is in the vicinity of goal($xgoal$, $ygoal$)
   (a) Declare it is reached the goal($xgoal$, $ygoal$)

5.4 Functions

1. getrising(int *inputpin*): check if the pulse width is bigger than 6us. less than is noise. if it is above return the rising edge time.
2. calcul(int *inputpin*): check for x, y pulse and return the x, y coordinate in time unit.
3. getxy() : get coordinates for (x, y) and the direction of the robot from the vive.
4. motorcontrol(int a, int b, int c, int e, int d): decides the direction of wheel rotation to move robot forward or backward.
5. motorhalt(): stop the motor of the robot.
6. forward(): move robot forward for small increment.
7. right(): turn robot right for small increment.
8. left(): turn robot left for small increment.
9. moveto(int $pos1$,int $pos2$) make robot to move toward the goal postion ($xgoal$, $ygoal$)

5.5 Results and Analysis

Unfortunately, our robot's essential part broke down during the competition. Therefore, we couldn't check able to check the autonomous code. For future development, we wanted to adopt obstacle detection using an ultrasonic sensor. Right now, our robot could plan a path from position A to B. However, this is not considering any other moving objects in the field such as other robots and moving obstacles. If there are moving obstacles or moving robots, our robot will simply bump into it. Thus, we can add ultrasonic sensors to detect any obstacles that is near our robot. If the robot detects any obstacles are close enough to cause the collision, we can simply move the robot to move away from the obstacles first and find a path to reach a goal position.

References

[1] P. Allen, "Differential drive robot." Columbia University Department of Computing Science, 2015.

[2] S. Buckley. (2015) This is how valve's amazing lighthouse tracking technology works. [Online]. Available: https://gizmodo.com/this-is-how-valve-s-amazing-lighthouse-tracking-technol-1705356768

[3] T. Chin. (2019) Robotic path planning: Rrt and rrt*]. [Online]. Available: https://medium.com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378

[4] D. Connell and H. La, "Dynamic path planning and replanning for mobile robots using rrt*," *IEEE International Conference on Systems, Man, and Cybernetics (SMC).*, 2017.

[5] R. Ellis. (2019) Kinematics of mobile robots [powerpoint slides]. [Online]. Available: https://slideplayer.com/slide/15952324/

[6] S. Han, B. Choi, and J. Lee, "A precise curved motion planning for a differential driving mobile robot," *Elsevier*, vol. 18, pp. 486–494, 2008.

[7] T. Hellström, "Kinematics equations for differential drive and articulated steering." Department of Computing Science Umeå, 2011.

[8] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning." *Robotics Science and Systems VI 104*.

[9] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," pp. TR 98–11, 1998.

[10] A. Malventano. (2016) Steamvr htc vive in-depth – lighthouse tracking system dissected and explored. [Online]. Available: https://pcper.com/2016/04/steamvr-htc-vive-in-depth-lighthouse-tracking-system-dissected-and-explored/

[11] me-no dev. (2018) ledc.h. [Online]. Available: https://github.com/espressif/arduino-esp32/blob/master/tools/sdk/include/driver/driver/ledc.h