# ME 5751 Finial: Truck Simulator

Cal Poly Pomona
Pomona, CA
Jonathan Jenkins
jjenkins@cpp.edu

***Abstract*— In this project we developed a truck simulator using a custom python algorithm. With a user suppled map, start point, and goal point our algorithm can generate an optimal travel path and motion control that will accurately simulate a truck driving. The methods used to test the controller involve a gui (graphic user interface) which simulates the robot and a custom python algorithm. The mathematics involved were sourced from "Introduction to autonomous mobile robots", 2nd Ed. by Siegwart, Nourbakhsh, and Scaramuzza. The findings in this project demonstrate our custom algorithm is very effectiveness in generating useable pathways from a wide variety of user supplied maps. Each test of our algorithm was a success, and the algorithm could develop a safe travel pathway and successfully control the robot motion.***

***Keywords*—Motion Planning, Grid Map, A* Algorithm map, Prioritization, Heuristic equation,**

## I. INTRODUCTION

Navigation and path planning is a major focus in modern robotic. The ability for a robot to transverse in an unknown environment relies on the ability to select a suitable path to travel. A motion planner is an algorithm that can derive a pathway from one position to another and controls a robot to execute the desired motion.

The goal of this lab is to develop an algorithm capable of generating an optimal path to take the robot from point A to point B and controlling a simulated robot to successfully execute the found path. Our motion planner consists of two key components, a path planner, and a controller. This path planner must develop the optimal route for any given map and do it as quickly as possible. The controller is responsible of controlling the robot kinematics and guide it to each goal position. After the development of our custom algorithm, we then conducted two experiments to determine the effectiveness of the code.

This paper will go into depth on different elements of this project including control theory, path planning algorithm, controller algorithm, python programing, and evaluation of the final algorithm's ability to develop optimal paths from user supplied environmental image, a starting position, and a goal position.

## II. BACKGROUND

Generating a good travel path is critical to the robot's ability to navigate within its environment. The objective of our algorithm is to generate a path between our truck's starting position and a goal position and control the truck down the path. As shown in Figure 1 an incorrect motion planner can make the difference between reaching your desired goal and crashing.

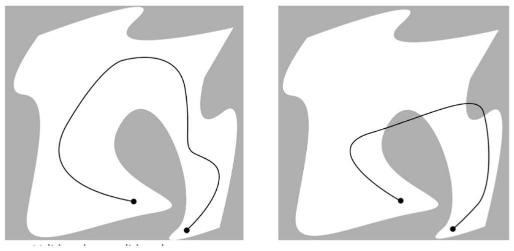


*Figure 1 Good vs Bad Path Panning*

To achieve this objective our algorithm needs to be meet three key requirements. First the algorithm needs to be able to develop a potential map. Second the algorithm needs to be able to implement a path planner to identify a safe route from the start position to the goal position. Third the algorithm needs to implement a p-controller that will control the motion of the truck while driving

A motion planner begins with generating a map. Maps can be used to represent a variety of information. The maps used in this project are a position map, cost map, and potential map. A position maps is a grid that shows occupied and unoccupied spaces. An example of a position map is shown in figure 2.

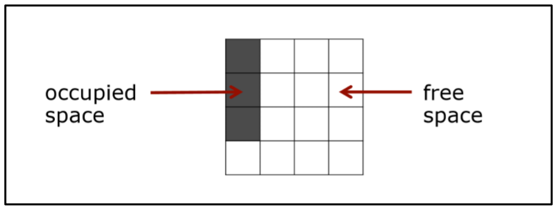


*Figure 2 Position Map*

A cost maps assigns each grid a value or "cost" depending on what that specific cost map is trying to represent. For example, a cost map can show the distance away from a known occupied space as shown in figure 3.

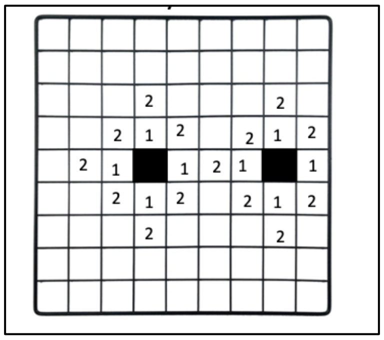


Figure 3 Distance  Cost map

In this project we will be implementing a distance cost map. In our cost map each occupied space is assigned a value based on the Manhattan distance formula as shown in equation 2.1.

$$d = |y_1 - y_0| + |x_1 - x_0| \tag{2.1}$$

The potential map is based on the cost map generated by the Manhattan distance. Our potential map represents a risk given to an unoccupied space based on the proximity that grid is to an occupied space. Spaces at a closer distance to an occupied space will be given a higher risk or cost and spaces that are farther away are assigned a lower risk.  An example of a potential cost map is shown in figure 4.

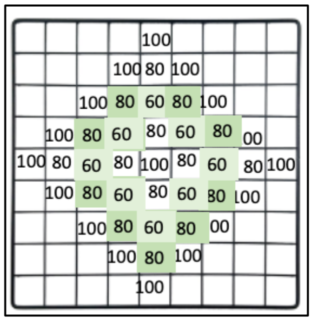


Figure 4 A Potential Cost Map

Our specific potential cost map assigns the highest risk to occupied spaces since it represents a physical object. Occupied spaces will have a risk value of 1000. Around all occupied spaces is an inflation zone or safety zone that is treated like an occupied space. We have this space to prevent accidental collisions with the wall. The safety zone will have a cost of 850. Lastly the unoccupied space outside the safety zone will have a gradient of cost based on the distance away each grid is from an occupied space.  The robot will prioritize traveling through grids with lower risk values. To assign the cost to each grid a linear equation shown in equation 2.2 was used. A visual representation of our potential cost map is shown in figure 5.

$$y = mx + b \quad m = \frac{y_2 - y_1}{x_2 - x_1} \tag{2.2}$$

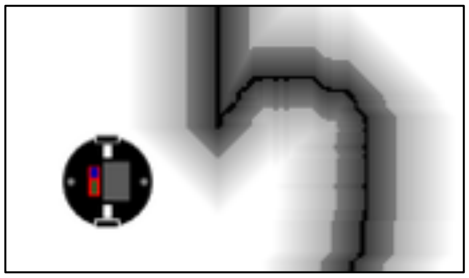


Figure 5 Our Potential  Cost map

With a Cost map in hand, we then use a user provided starting position and goal position as inputs to our path planner. The path planner will take these three inputs to calculate the optimal path between the start position and goal position.

Fundamentally, a path planner decides what grids should the robot move in order to reach the desired location. Each grid that can be moved to is called a node. Connections between nodes forms your path.  It is up to the path planner to choose the optimal path.   A visual example of nodes and pathways is shown in figure 6.

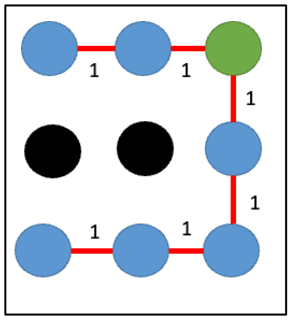


Figure 6 Nodes and Pathways

There are a variety of path planner algorithms that can be used.  Our path planner uses a dual search A* algorithm.

The A* algorithm combines two popular algorithms together, a breadth first search and a Dijkstra's algorithm. Breadth First Search (BFS) assigns a value to each grid corresponding to the distance that grid is to the starting location. The BFS algorithm implements the Manhattan distance formula as shown in equation 2.1. An example of this algorithm is shown in figure 7.

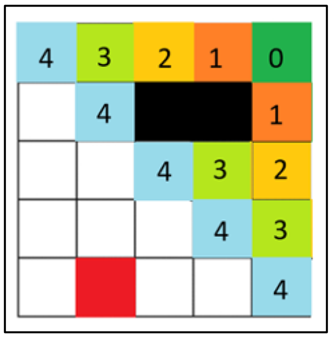


*Figure 7 Breadth First Search Cost Map*

After all grids are given a value, the algorithm will then test different path options. Each path is assigned a total cost value equal to the sum cost of all the grids/nodes that makes up that path. The pathway with the lowest cost is considered the most optimal path. An example of this process is shown if figure 8.

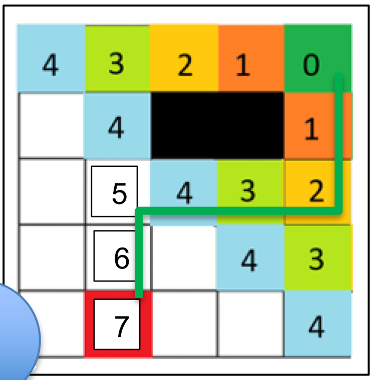


*Figure 8 BFS Path*

The Dijkstra's algorithm assigns each grid a value corresponding to the distance from the goal position plus the value assigned by the potential map. Instead of expanding evenly in all directions the Dijkstra's algorithm will expand the pathways closest to the goal position. An example of Dijkstra's algorithm is shown in figure 9.

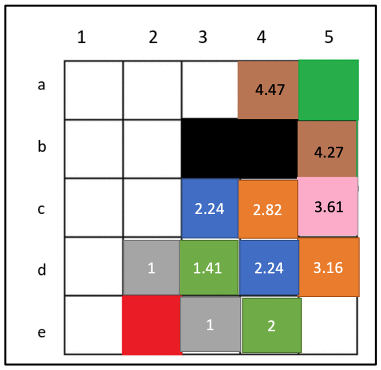


*Figure 9 Dijkstra's algorithm Cost map*

As mentioned, A* algorithm is design to combine the benefits of both methods. the A* algorithm uses the cost assigned by the potential map, and adds an additional cost based on each grids distance to the goal and start position. In the A* algorithm the node cost can be written as shown in equation 2.3. Where the node cost [ d (i,j) ] is equal to the known cost from the start point [ c(i,j) ] and the estimated cost from the goal point[ h(i,j) ]. An example of the A* grid map is shown in figure 10.

$$d(i,j) = c(i,j) + \mu H \qquad (2.3)$$

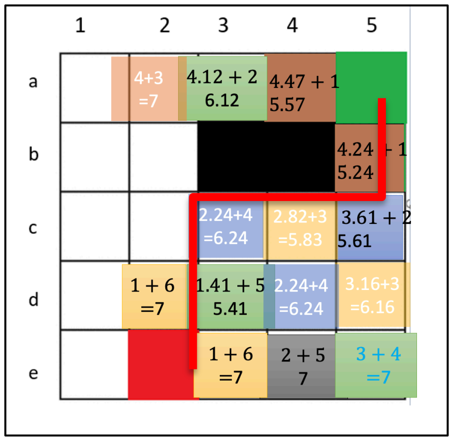


*Figure 10 A* Node Cost map*

With the A * algorithm we now have an efficient way to assign a cost to each grid, priorities grids that are both close to the goal position and the shortest path from the start position. Now with the optimal path found between the goal and start point the algorithm needs to reconstruct the discovered path into a clear list of instructions.

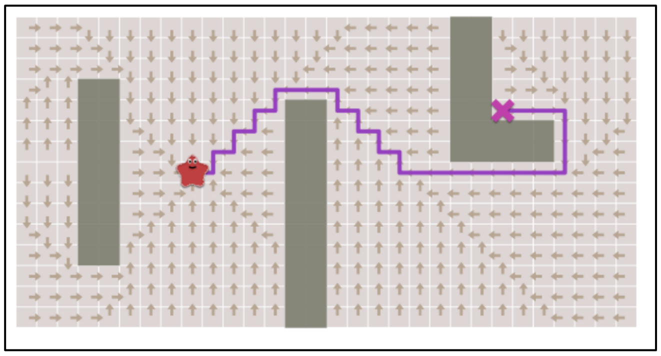


Figure 11 Path Reconstruction

As mentioned, the A* algorithm begins from the starting grid. At this starting point the program will index to all surrounding grids (neighbors) and assign then a node cost. We call the primary grid a parent grid, and all neighbors a child grid. Every node now has a parent and child relationship. This relationship for each grid is stored in a dictionary queue. When reconstruction is ready, we pull the list of each parent and each parent's parent beginning from the goal position. This list will eventually lead directly from the goal position to the start position forming the desired path the algorithm calculated. An example of this concept is shown in figure 12.

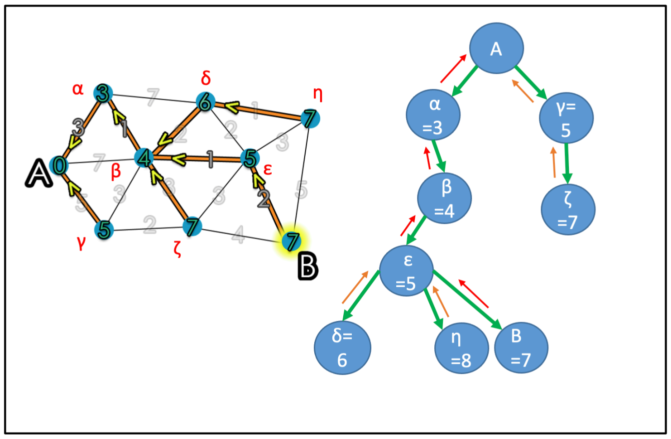


Figure 12 Parent / Child Grids

With a path formed now specific points of this path is converted to waypoints that the robot will drive towards. Each waypoint stores a (x,y) position and a travel direction that the robot is trying to meet. Figure 13 shows an example of waypoints assigned onto a path.

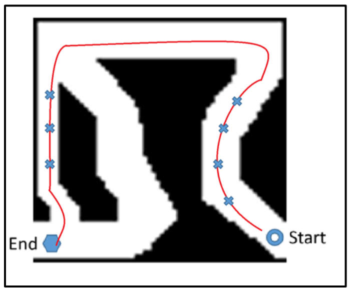


Figure 13 Waypoints

We begin by assigning waypoints every 10 grid spaces. Each assigned waypoint can be seen as arrows in figure 14. After a waypoint is assign the travel direction needs to be calculated. We do this by forming a triangle between each two waypoint and calculate the distance between the points and the angle. We convert this angle to radians with respect to the global frame. We then store this direction angle for each waypoint.

Once each waypoint is assigned potion position and an angle value the path is technically completed. We however implement a reduction function to minimize the number of waypoints in the path to maximize the path speed and efficiency. We do this by filtering out unnecessary waypoints if they are a certain distance apart and if they are within a limited theta angle range. Our requirements were that waypoints must be no more than 70 units apart and up to 45 degrees travel change. Because of our requirements sharp turns will have more waypoints for higher resolution and strait paths will have less points for faster travel. An example of our final path is shown in figure 14.

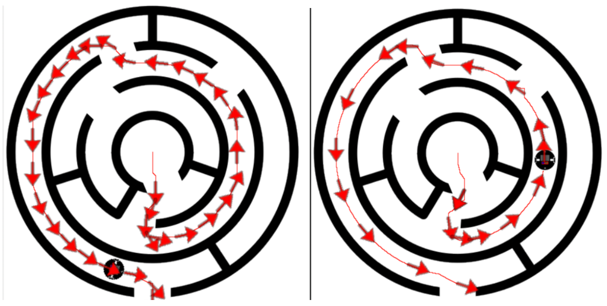


Figure 14 Waypoints and Reduction Algorithm

With the list of waypoints, we now send the information to the Proportional control (P control). The P-Control was used to adjust the robot's linear and angular velocity in order to control it to reach each waypoint.

Proportional control (P control) is a type of linear feedback control system in which a correction is applied to the controlled variable. The size of the correction is proportional to the current error of the system. Error (e) is the difference between the desired value and the measured value. Proportional gain (k) is the size of the correction that will be implemented. Figure 15 shows a standard proportional control logic diagram.

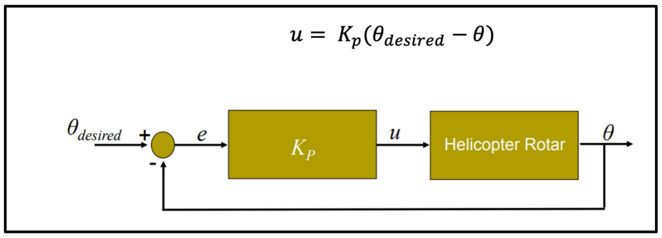


Figure 15 Proportional Control Diagram

There are many ways to apply P control to a system. In this experiment P Control will be used to adjust the robot's linear and angular velocity in order to control it to reach a desired point (goal point). Our error will be calculated from the current position and orientation of the robot and measured and compared to a goal position and orientation every 0.1 seconds. The kinematics between these two points are shown in figure 16.

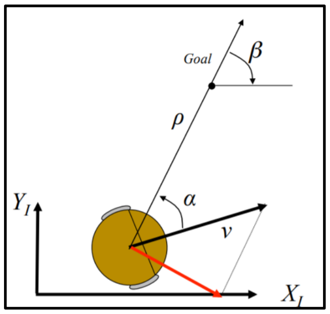


Figure 16 . Robot and Goal Point Kinematics

The value rho (ρ) is the positional vector between the current and desired point in space. Alpha ($\alpha$) is the angle between the robot direction of travel (Xr) and the rho position vector. Lastly beta (β) is the angle between the x-axis of the goal point and the rho position vector. These variables can be re-written as the following equations.

$$\rho = \sqrt{\Delta x^2 + \Delta y^2} \quad (2.4)$$

$$\alpha = -\theta + atan2(-\Delta y, -\Delta x) \quad (2.5)$$

$$\beta = -\theta - \alpha \quad (2.6)$$

Alpha, beta, and rho make up the error components of the proportional controller. A fixed proportional gain variable (k) is used for each error variable. Lastly the proportional controller equations to control the robot linear velocity (v) and angular velocity (w) can be written as shown in equation 2.7 & 2.8.

$$v = k_\rho \rho \quad (2.7)$$

$$w = k_\alpha \alpha + k_\beta \beta \quad (2.8)$$

## III. KINEMATIC MODEL

The robot used in our simulations is based on a truck and implements truck kinematics with steering. Truck kinematics needs to consider the physical frame of the truck width, axile distance, wheel diameter, and limitation of steering angle and wheel speed. For our truck the physical characteristics was assigned to our in our project outline and is listed below.

- Car width: 2L = 16
- Car axile distance: d = 20
- Wheel diameter: r = 3
- Steering angle: b < 40 Degrees
- Wheel speed: rho < 20 rad/s

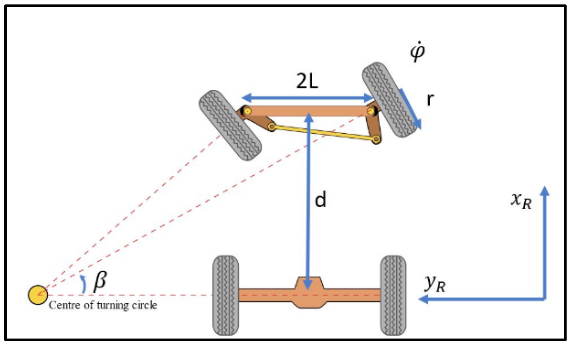


Figure 17 Truck Physical Characteristics

To implement an accurate simulation, we must use the physical characteristics of the truck and constrain the desired linear and angular velocity generated from the p-controller to the truck's kinematics equations. The truck kinematics steams from Ackerman steering equation. Ackerman steering specify the wheel angle to allow the truck to turn without wheel slippage. Equation 3.1 shows the Ackerman steering equation for both tires.

$$\delta_{f,in} = \tan^{-1}\left(\frac{L}{R - \frac{T}{2}}\right) \quad \delta_{f,out} = \tan^{-1}\left(\frac{L}{R + \frac{T}{2}}\right) \quad (3.1)$$

Using the anchorman steering equation, we can then derive the vehicle speed and turning radius based on the linear kinematics of the truck shown in equations 3.2 – 3.4.

$$\dot{x} = s\cos\theta \tag{3.2}$$

$$\dot{y} = s\sin\theta \tag{3.3}$$

$$\dot{\theta} = \frac{s}{l}\tan\phi \approx \frac{s}{l}\phi \tag{3.4}$$

After deriving the vehicle linear and angular velocity based on the p controller desired speeds, we derive the theoretical steering angle, wheel speed, and linear velocity that the robot would need to perform to follow the controller. Next, if the calculated steering angle or wheel speed exceeds our physical limit, we will limit the vehicle kinematics to the constraints of the truck. Finally, we have the filtered steering angle, and wheel speed that we will use to control the simulated truck. We use the same truck kinematics and calculate the final truck linear velocity and angular velocity. The truck will now execute the motion planner while using truck kinematics.

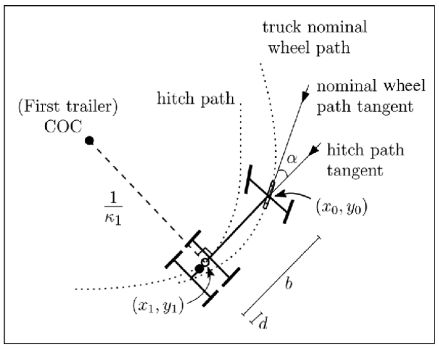


Figure 18 Truck Kinematics

## III CONTROL DESIGN

To implement our algorithm, we developed three custom python script. The first script called *cost_map.py* was for the potential map generation. A second script called *path_planner.py* focused on the path planner. The last script called *p_controller.py* was for the p-controller.

Our algorithm begins by importing a user supplied map and sending it to our previously made code *cost_map.py*. This script was developed during the second module lab. This script will take the image and convert it to a matrix. Every pixel is converted to an element in the matrix. Using this matrix, a potential map is generated by identifying all unoccupied spaces and assigning every grid a risk value. An initial value is assigned by the distance each grid is from the nearest occupied grid. After each grid is assigned a distance value the final potential map risk value is given by a linear equation. Occupied spaces are assigned a value of 1000. While unoccupied spaces are assigned a value of 0-850. Unoccupied spaces with a cost of more than 850 are within a “no-go” zone we called safety zone. This safety zone represents our inflation radius and spans 18 grids out from every occupied space. All unoccupied space outside the inflation zone will begin with a value of 800 and decrease from the distance each individual grid is from the edge of the safety zone. The lowest assign risk a grid can have is 0. Figure 19 shows an example of a potential map generated by our *cost_map.py* script.

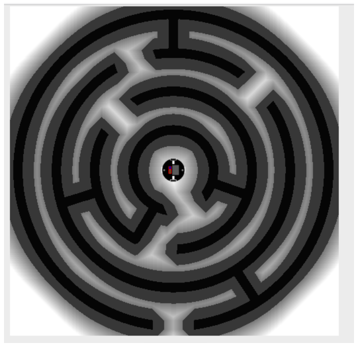
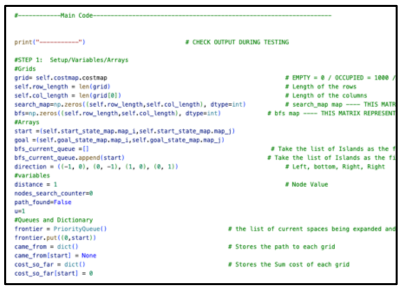


Figure 19 Cost Map’s Potential Map Output

Next our algorithm sends the cost map to the path planner script called code *path_planner_final.py*. This script was based on the scripted developed during the third module lab. The objective of the path planner is to identify a safe route between the starting position and the goal position. To implement this, search the A* algorithm was used. Our A* algorithm was set up similarly as in module 3. We began by creating three priority queue that will store the list of grids being search, a list that will contain the parent child relationship, and a list to store the current cost acuminated for each path. The code used for step one can be seen in figure 20.



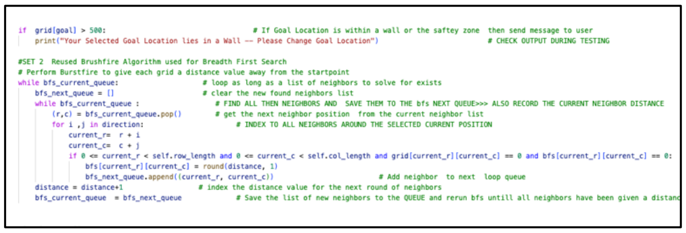
```
#------------Main Code-----------------------------------------------------------------


print("-----------")                              # CHECK OUTPUT DURING TESTING

#STEP 1:  Setup/Variables/Arrays
#Grids
grid= self.costmap.costmap                                                # EMPTY = 0 / OCCUPIED = 1000 /
self.row_length = len(grid)                                               # Length of the rows
self.col_length = len(grid[0])                                            # Length of the columns
search_map=np.zeros((self.row_length,self.col_length), dtype=int)         # search_map map ---- THIS MATR
bfs=np.zeros((self.row_length,self.col_length), dtype=int)            # bfs map ---- THIS MATRIX REPRESENT
#Arrays
start =(self.start_state_map.map_i,self.start_state_map.map_j)
goal =(self.goal_state_map.map_i,self.goal_state_map.map_j)
bfs_current_queue =[]                                                  # Take the list of Islands as the f
bfs_current_queue.append(start)                                       # Take the list of Islands as the fi
direction = ((-1, 0), (0, -1), (1, 0), (0, 1))                            # Left, bottom, Right, Right
#variables
distance = 1                                                              # Node Value
nodes_search_counter=0
path_found=False
u=1
#Queues and Dictionary
frontier = PriorityQueue()                                   # the list of current spaces being expanded and
frontier.put((0,start))
came_from = dict()                                           # Stores the path to each grid
came_from[start] = None
cost_so_far = dict()                                         # Stores the Sum cost of each grid
cost_so_far[start] = 0
```

Figure 20 A* Algorithm Setup Code

As mentioned before A* algorithm assigns cost to each cell based on the sum cost of the potential map, the breath-first-search, and the heuristic function. The next phase of the A* algorithm indexes to the grids and calculated the cost for each inspected grid. The BFS cost value is given by the distance the grid is from the starting location. This process is like the brushfire algorithm we developed before so that code was reused. The Manhattan distance equation was used to calculate each grid BFS cost. The code used for this step is found in figure 21.

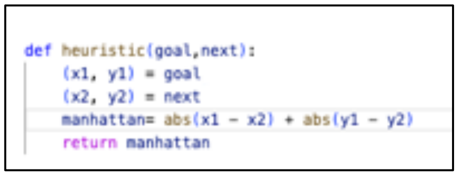


```
if  grid[goal] > 500:                              # If Goal Location is within a wall or the saftey zone  then send message to user
    print("Your Selected Goal Location lies in a Wall -- Please Change Goal Location")                  # CHECK OUTPUT DURING TESTING

#SET 2  Reused Brushfire Algorithm used for Breadth First Search
# Perform Burstfire to give each grid a distance value away from the startpoint
while bfs_current_queue:                       # loop as long as a list of neighbors to solve for exists
    bfs_next_queue = []                        # clear the new found neighbors list
    while bfs_current_queue :                     # FIND ALL THEN NEIGHBORS AND  SAVE THEM TO THE bfs NEXT QUEUE>>> ALSO RECORD THE CURRENT NEIGHBOR DISTANCE
        (r,c) = bfs_current_queue.pop()           # get the next neighbor position  from the current neighbor list
        for i ,j in direction:                        # INDEX TO ALL NEIGHBORS AROUND THE SELECTED CURRENT POSITION
            current_r=  r + i
            current_c=  c + j
            if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[current_r][current_c] == 0 and bfs[current_r][current_c] == 0:
                bfs[current_r][current_c] = round(distance, 1)
                bfs_next_queue.append((current_r, current_c))                        # Add neighbor  to next  loop queue
    distance = distance+1                     # index the distance value for the next round of neighbors
    bfs_current_queue  = bfs_next_queue               # Save the list of new neighbors to the QUEUE and rerun bfs untill all neighbors have been given a distanc
```

Figure 21 A* Algorithm BFS Code

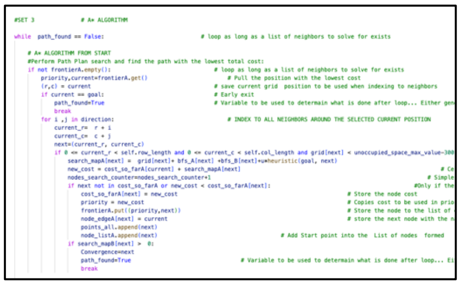
The heuristic cost was calculated by the grids distance from the goal position, as shown in figure 22. The heuristic cost also is given a scalar to adjust the level of influence this variable has on the overall A* algorithm.



```
def heuristic(goal,next):
    (x1, y1) = goal
    (x2, y2) = next
    manhattan= abs(x1 - x2) + abs(y1 - y2)
    return manhattan
```

Figure 22 Heuristic Cost

As mentioned, the A* algorithm will search from the start position to the goal position, prioritizing searching paths with the lowest cost value. To improve search efficiency, we implemented a dual A* algorithm where two separate searches occur. One search from the start position to the goal position. The second search from the goal position to the start position. The code for the dual search is shown figure 23 and figure 24.

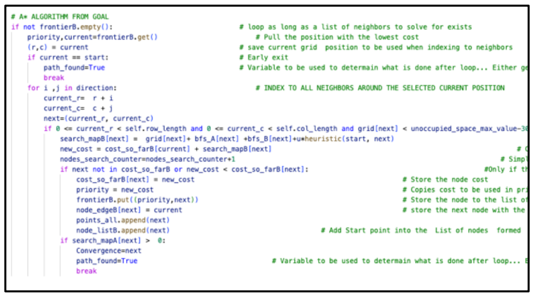


```
#SET 3              # A* ALGORITHM

while  path_found == False:                         # loop as long as a list of neighbors to solve for exists

    # A* ALGORITHM FROM START
    #Perform Path Plan search and find the path with the lowest total cost:
    if not frontierA.empty():                         # loop as long as a list of neighbors to solve for exists
        priority,current=frontierA.get()                 # Pull the position with the lowest cost
        (r,c) = current                               # save current grid  position to be used when indexing to neighbors
        if current == goal:                           # Early exit
            path_found=True                           # Variable to be used to determain what is done after loop... Either gen
            break
        for i ,j in direction:                           # INDEX TO ALL NEIGHBORS AROUND THE SELECTED CURRENT POSITION
            current_r=  r + i
            current_c=  c + j
            next=(current_r, current_c)
            if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[next] < unoccupied_space_max_value-300
                search_mapA[next] =  grid[next]+ bfs_A[next] +bfs_B[next]+u*heuristic(goal, next)
                new_cost = cost_so_farA[current] + search_mapA[next]                                          # Ce
                nodes_search_counter=nodes_search_counter+1                                                   # Simple
                if next not in cost_so_farA or new_cost < cost_so_farA[next]:                              #Only if the
                    cost_so_farA[next] = new_cost                                       # Store the node cost
                    priority = new_cost                                                 # Copies cost to be used in prio
                    frontierA.put((priority,next))                                      # Store the node to the list of
                    node_edgeA[next] = current                                          # store the next node with the n
                    points_all.append(next)
                    node_listA.append(next)                          # Add Start point into the  List of nodes  formed
                if search_mapB[next] >  0:
                    Convergence=next
                    path_found=True                      # Variable to be used to determain what is done after loop... Ei
                    break
```

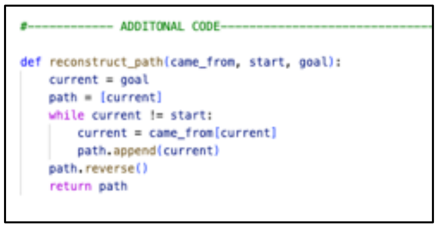
Figure 23 A* Search From Start



```
# A* ALGORITHM FROM GOAL
if not frontierB.empty():                         # loop as long as a list of neighbors to solve for exists
    priority,current=frontierB.get()                 # Pull the position with the lowest cost
    (r,c) = current                               # save current grid  position to be used when indexing to neighbors
    if current == start:                          # Early exit
        path_found=True                           # Variable to be used to determain what is done after loop... Either ge
        break
    for i ,j in direction:                           # INDEX TO ALL NEIGHBORS AROUND THE SELECTED CURRENT POSITION
        current_r=  r + i
        current_c=  c + j
        next=(current_r, current_c)
        if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[next] < unoccupied_space_max_value-30
            search_mapB[next] =  grid[next]+ bfs_A[next] +bfs_B[next]+u*heuristic(start, next)
            new_cost = cost_so_farB[current] + search_mapB[next]                                          # C
            nodes_search_counter=nodes_search_counter+1                                                   # Simpl
            if next not in cost_so_farB or new_cost < cost_so_farB[next]:                              #Only if th
                cost_so_farB[next] = new_cost                                       # Store the node cost
                priority = new_cost                                                 # Copies cost to be used in pri
                frontierB.put((priority,next))                                      # Store the node to the list of
                node_edgeB[next] = current                                          # store the next node with the
                points_all.append(next)
                node_listB.append(next)                          # Add Start point into the  List of nodes  formed
            if search_mapA[next] >  0:
                Convergence=next
                path_found=True                      # Variable to be used to determain what is done after loop... E
                break
```

Figure 24 A* Search From Goal

Now that a path from the start position and goal position is found the next step is to reconstruct the path. As mentioned before the algorithm stores the parent node and child node for each grid in the dictionary queue called *came_from*. Now with the final position found (goal) we can search who the parent grid was for each instance all the way back to the starting position. This reconstruction is saved in an array called '*points*'. The code used for this section is shown in figure 19.

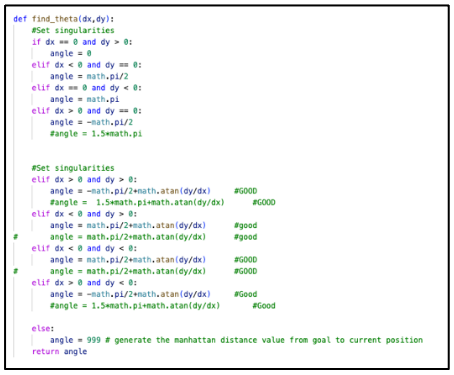


```
#----------- ADDITONAL CODE-----------------------------

def reconstruct_path(came_from, start, goal):
    current = goal
    path = [current]
    while current != start:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

Figure 25 Path Reconstruction

Next the A* algorithm will choose waypoints from the list of points in the path to convert into waypoints. Waypoints are intermediate goal positions that the robot will travel to in order to get to the final goal position. Each waypoints contains a position value (x,y) and a rotation value (theta). The position value is pulled from the path reconstruction. The rotation for each waypoint is assigned by the slope angle between two waypoints. This angle will then be converted to radians. The code used to calculate a waypoint rotation is shown in figure 26.



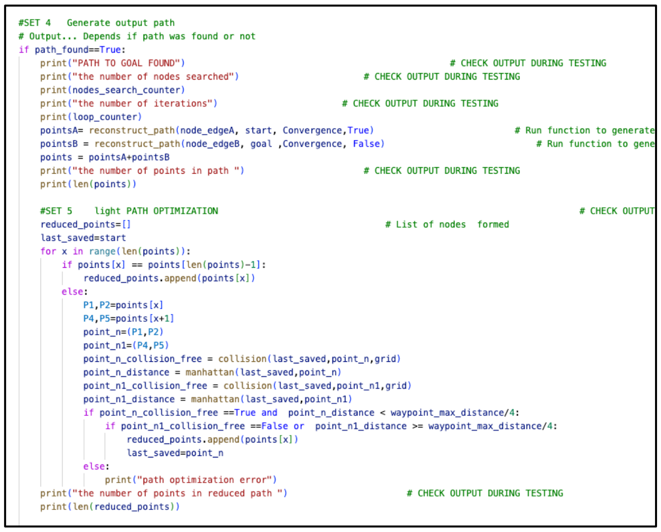
```
def find_theta(dx,dy):
    #Set singularities
    if dx == 0 and dy > 0:
        angle = 0
    elif dx < 0 and dy == 0:
        angle = math.pi/2
    elif dx == 0 and dy < 0:
        angle = math.pi
    elif dx > 0 and dy == 0:
        angle = -math.pi/2
        #angle = 1.5*math.pi


    #Set singularities
    elif dx > 0 and dy > 0:
        angle = -math.pi/2+math.atan(dy/dx)      #GOOD
        #angle =  1.5*math.pi+math.atan(dy/dx)      #GOOD
    elif dx < 0 and dy > 0:
        angle = math.pi/2+math.atan(dy/dx)       #good
#       angle = math.pi/2+math.atan(dy/dx)       #good
    elif dx < 0 and dy < 0:
        angle = math.pi/2+math.atan(dy/dx)       #GOOD
#       angle = math.pi/2+math.atan(dy/dx)       #GOOD
    elif dx > 0 and dy < 0:
        angle = -math.pi/2+math.atan(dy/dx)      #Good
        #angle = 1.5*math.pi+math.atan(dy/dx)      #Good

    else:
        angle = 999 # generate the manhattan distance value from goal to current position
    return angle
```

Figure 26 Waypoint Theta Function

Each point in the path can be converted into a waypoint, however this will cause the robot to move very slowly and inefficiently. Therefore, after each waypoint has been generated, we then optimize the path to limit the number of waypoints in the path. The density of waypoints will be depended on two variables, the distance between each waypoint and the angular change between each waypoint. We allow a maximum distance between waypoints to be 70 grids apart. We also limit the maximum rotation change between waypoints to be 45 degrees. Based on the distance between points and the change of angle between waypoints a final list of waypoints is produced. This code is shown in figure 27.

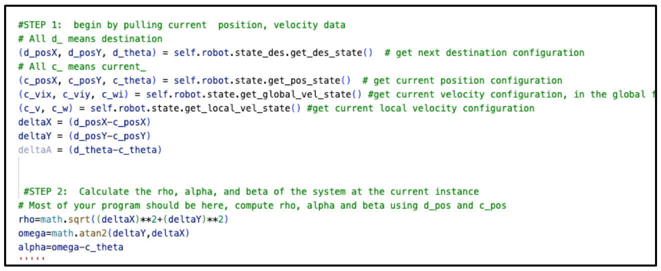


```
#SET 4   Generate output path
# Output... Depends if path was found or not
if path_found==True:
    print("PATH TO GOAL FOUND")                                          # CHECK OUTPUT DURING TESTING
    print("the number of nodes searched")                        # CHECK OUTPUT DURING TESTING
    print(nodes_search_counter)
    print("the number of iterations")                        # CHECK OUTPUT DURING TESTING
    print(loop_counter)
    pointsA= reconstruct_path(node_edgeA, start, Convergence,True)                 # Run function to generate
    pointsB = reconstruct_path(node_edgeB, goal ,Convergence, False)                 # Run function to gene
    points = pointsA+pointsB
    print("the number of points in path ")                       # CHECK OUTPUT DURING TESTING
    print(len(points))

    #SET 5    light PATH OPTIMIZATION                                                      # CHECK OUTPUT
    reduced_points=[]                                        # List of nodes  formed
    last_saved=start
    for x in range(len(points)):
        if points[x] == points[len(points)-1]:
            reduced_points.append(points[x])
        else:
            P1,P2=points[x]
            P4,P5=points[x+1]
            point_n=(P1,P2)
            point_n1=(P4,P5)
            point_n_collision_free = collision(last_saved,point_n,grid)
            point_n_distance = manhattan(last_saved,point_n)
            point_n1_collision_free = collision(last_saved,point_n1,grid)
            point_n1_distance = manhattan(last_saved,point_n1)
            if point_n_collision_free ==True and  point_n_distance < waypoint_max_distance/4:
                if point_n1_collision_free ==False or  point_n1_distance >= waypoint_max_distance/4:
                    reduced_points.append(points[x])
                    last_saved=point_n
            else:
                print("path optimization error")
    print("the number of points in reduced path ")                    # CHECK OUTPUT DURING TESTING
    print(len(reduced_points))
```

Figure 27 Path Optimimization

Finally, the A* algorithm will export the calculated waypoint list and send it to the next python script called *P_controller.py*. This script is task with controlling the virtual robot and implementing truck kinematics to allow an accurate motion between each waypoint. The first step in the p_controller is to calculate the rho, omega, and alpha variable. These variables represent the current error of the robot to the desired values. The code used for this step is shown in figure 28.

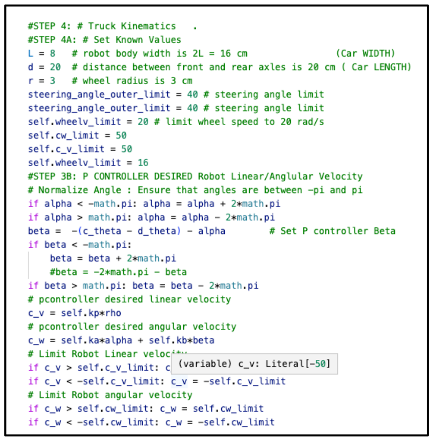


```
#STEP 1:  begin by pulling current  position, velocity data
# All d_ means destination
(d_posX, d_posY, d_theta) = self.robot.state_des.get_des_state()  # get next destination configuration
# All c_ means current_
(c_posX, c_posY, c_theta) = self.robot.state.get_pos_state()  # get current position configuration
(c_vix, c_viy, c_wi) = self.robot.state.get_global_vel_state() #get current velocity configuration, in the global f
(c_v, c_w) = self.robot.state.get_local_vel_state() #get current local velocity configuration
deltaX = (d_posX-c_posX)
deltaY = (d_posY-c_posY)
deltaA = (d_theta-c_theta)


 #STEP 2:  Calculate the rho, alpha, and beta of the system at the current instance
# Most of your program should be here, compute rho, alpha and beta using d_pos and c_pos
rho=math.sqrt((deltaX)**2+(deltaY)**2)
omega=math.atan2(deltaY,deltaX)
alpha=omega-c_theta
'''''
```

Figure 28 P_Controller Error

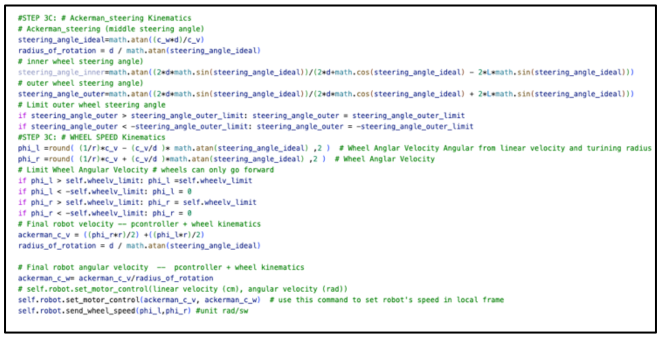
The next step of the controller is to take the calculated error and calculate the theoretical linear velocity and angular velocity of the system. We do this by using the truck kinematic equations highlighted in section 3.



```
#STEP 4: # Truck Kinematics   .
#STEP 4A: # Set Known Values
L = 8   # robot body width is 2L = 16 cm                (Car WIDTH)
d = 20  # distance between front and rear axles is 20 cm ( Car LENGTH)
r = 3   # wheel radius is 3 cm
steering_angle_outer_limit = 40 # steering angle limit
steering_angle_outer_limit = 40 # steering angle limit
self.wheelv_limit = 20 # limit wheel speed to 20 rad/s
self.cw_limit = 50
self.c_v_limit = 50
self.wheelv_limit = 16
#STEP 3B: P CONTROLLER DESIRED Robot Linear/Anglular Velocity
# Normalize Angle : Ensure that angles are between -pi and pi
if alpha < -math.pi: alpha = alpha + 2*math.pi
if alpha > math.pi: alpha = alpha - 2*math.pi
beta =  -(c_theta - d_theta) - alpha        # Set P controller Beta
if beta < -math.pi:
    beta = beta + 2*math.pi
    #beta = -2*math.pi - beta
if beta > math.pi: beta = beta - 2*math.pi
# pcontroller desired linear velocity
c_v = self.kp*rho
# pcontroller desired angular velocity
c_w = self.ka*alpha + self.kb*beta
# Limit Robot Linear veloc
if c_v > self.c_v_limit: c (variable) c_v: Literal[-50]
if c_v < -self.c_v_limit: c_v = -self.c_v_limit
# Limit Robot angular velocity
if c_w > self.cw_limit: c_w = self.cw_limit
if c_w < -self.cw_limit: c_w = -self.cw_limit
```

Figure 29 Truck Kinematics

Next, we will use the theoretical linear velocity and angular velocity to calculate the theoretical wheel speed, and steering angle necessary. We will then limit the final wheel speed and steering angle based on the truck constants highlighted in section three. The code used in this step is shown in figure 30.

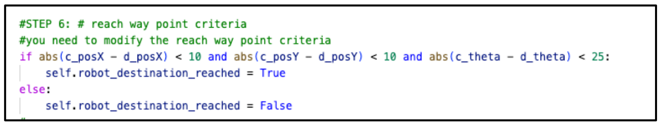


```
#STEP 3C: # Ackerman_steering Kinematics
# Ackerman_steering (middle steering angle)
steering_angle_ideal=math.atan((c_w*d)/c_v)
radius_of_rotation = d / math.atan(steering_angle_ideal)
# inner wheel steering angle)
steering_angle_inner=math.atan((2*d*math.sin(steering_angle_ideal))/(2*d+math.cos(steering_angle_ideal) - 2*L*math.sin(steering_angle_ideal)))
# outer wheel steering angle)
steering_angle_outer=math.atan((2*d*math.sin(steering_angle_ideal))/(2*d*math.cos(steering_angle_ideal) + 2*L*math.sin(steering_angle_ideal)))
# Limit outer wheel steering angle
if steering_angle_outer > steering_angle_outer_limit: steering_angle_outer = steering_angle_outer_limit
if steering_angle_outer < -steering_angle_outer_limit: steering_angle_outer = -steering_angle_outer_limit
#STEP 3C: # WHEEL SPEED Kinematics
phi_l =round( (1/r)*c_v - (c_v/d )* math.atan(steering_angle_ideal) ,2 )  # Wheel Anglar Velocity Angular from linear velocity and turining radius
phi_r =round( (1/r)*c_v + (c_v/d )*math.atan(steering_angle_ideal) ,2 )  # Wheel Anglar Velocity
# Limit Wheel Angular Velocity # wheels can only go forward
if phi_l > self.wheelv_limit: phi_l =self.wheelv_limit
if phi_l < -self.wheelv_limit: phi_l = 0
if phi_r > self.wheelv_limit: phi_r = self.wheelv_limit
if phi_r < -self.wheelv_limit: phi_r = 0
# Final robot velocity -- pcontroller + wheel kinematics
ackerman_c_v = ((phi_r*r)/2) +((phi_l*r)/2)
radius_of_rotation = d / math.atan(steering_angle_ideal)

# Final robot angular velocity  --  pcontroller + wheel kinematics
ackerman_c_w= ackerman_c_v/radius_of_rotation
# self.robot.set_motor_control(linear velocity (cm), angular velocity (rad))
self.robot.set_motor_control(ackerman_c_v, ackerman_c_w)  # use this command to set robot's speed in local frame
self.robot.send_wheel_speed(phi_l,phi_r) #unit rad/sw
```

Figure 30 Ackerman Kinematics and Filtering

Finally, we take the filtered wheel speed and steering angle and use that to calculate the flited linear velocity and angular velocity. The filtered speed will be used to control the actual robot simulation. As the robot moves between waypoints, we will use a simple function to identify when the robot has successfully reached each waypoint. The reach criteria are based on the robot position and rotation being within a small deviation to the waypoint position and rotation. The code used in this final step is show in figure 31.



```
#STEP 6: # reach way point criteria
#you need to modify the reach way point criteria
if abs(c_posX - d_posX) < 10 and abs(c_posY - d_posY) < 10 and abs(c_theta - d_theta) < 25:
    self.robot_destination_reached = True
else:
    self.robot_destination_reached = False
```

Figure 31 Reach Criteria

## IV. METHOD

After completing the development of our motion planner, we then moved to develop suitable challenges to test the code. The algorithm will be tested on two different maps. On each map the algorithm will be evaluated by three metrics. First, we will evaluate the successful creation of the generated path without any collision with occupied spaces. A successful map generation is shown in figure 32.

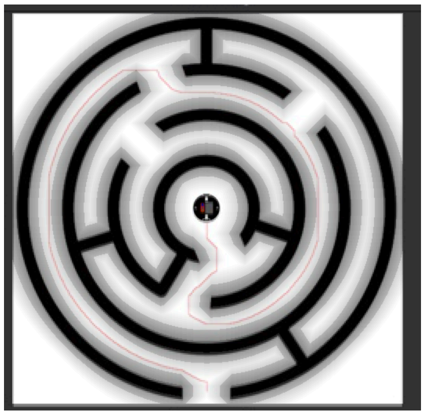
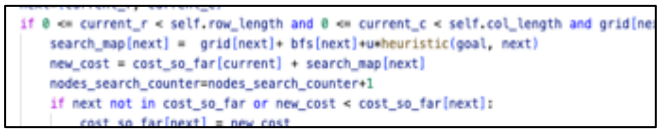


Figure 32 Exported Cost Map Overlay

Second, we will evaluate the A* algorithm statistics. We will evaluate the number of nodes inspected, the number of iterations needed, and the total time required to execute the motion planner. We will also evaluate the number of waypoints generated for the path and the number of waypoints the algorithm was able to reduce. The node counter and iteration counter are shown in figure 33.



```
if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[ne
    search_map[next] =  grid[next]+ bfs[next]+u*heuristic(goal, next)
    new_cost = cost_so_far[current] + search_map[next]
    nodes_search_counter=nodes_search_counter+1
    if next not in cost_so_far or new_cost < cost_so_far[next]:
        cost_so_far[next] = new_cost
```

Figure 33 Node Counter Variable

Lastly, we will evaluate the how well the controller moves the robot between each waypoint. Ideally, we should see a smooth and steady motion without any collation with the walls of the map. An example of a good motion is shown in figure 34

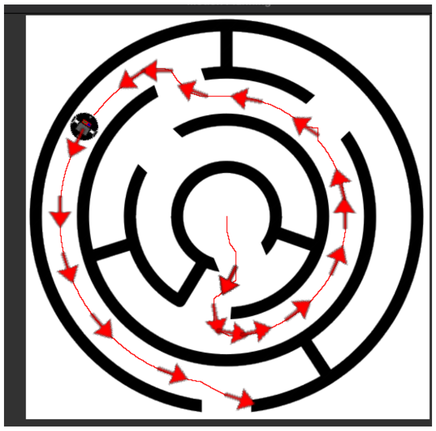


Figure 34 Robot Motion

## V. EXPERIMENT RESULTS AND DISCUSSION

As mentioned, to determine the effectiveness of our Truck algorithm we tested the program on two custom maps. Each test will require the program to successfully develop the potential map, find a safe path using the path planner, and drive the truck using the p-controller with truck kinematics.

The first experiment uses a custom maze map. This map represents complex obstacle with many false paths. Figure 35 shows the found path and the robots' traversing between the start position and goal position. The algorithm began by implementing the A* algorithm to find a path from start to goal point. It took 13658 iterations for the A* to find a successful path. Next the path panner takes the points in the found path and converts a portion of them into waypoints. Out of the 1636 points in the path 45 were converted into waypoints. The algorithm then assigned theta values for each point. After each theta angle has been assigned a further reduction of waypoints are performed. The final number of waypoints in the finish path was 20. Figure 36 shows the output statistics for this experiment. The entire algorithm took 0.3 seconds to execute and complete.

Next, we initiated the truck simulation and monitored how well the p-controller navigated through the map. Overall, the program was successfully able to control the truck through the pathway without any visual errors or collisions.

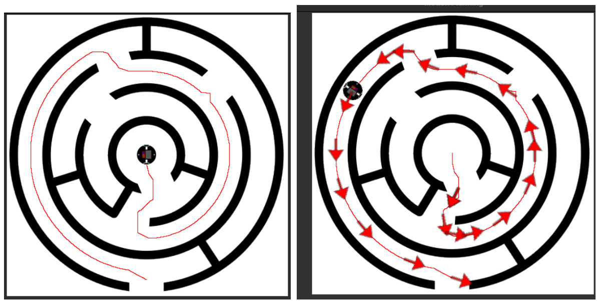


Figure 35 Experiment 1 Map

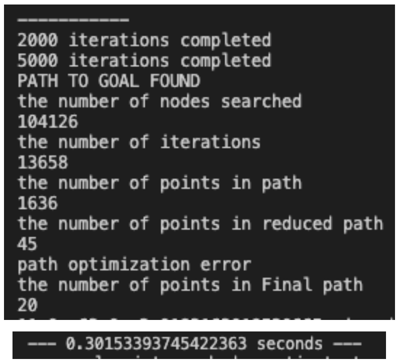

```
-----------
2000 iterations completed
5000 iterations completed
PATH TO GOAL FOUND
the number of nodes searched
104126
the number of iterations
13658
the number of points in path
1636
the number of points in reduced path
45
path optimization error
the number of points in Final path
20
```
```
--- 0.30153393745422363 seconds ---
```

Figure 36 Experiment 1 Statistics

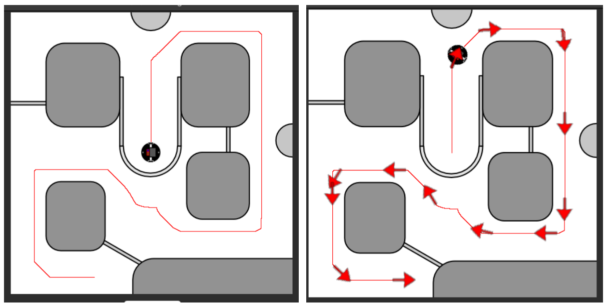


Figure 37 Experiment 2 Map

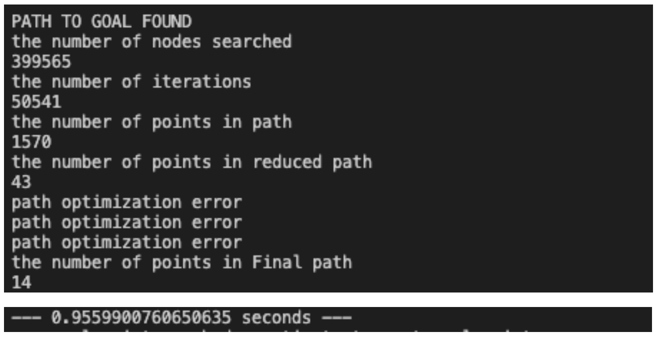

```
PATH TO GOAL FOUND
the number of nodes searched
399565
the number of iterations
50541
the number of points in path
1570
the number of points in reduced path
43
path optimization error
path optimization error
path optimization error
the number of points in Final path
14
```
```
--- 0.9559900760650635 seconds ---
```

Figure 38 Experiment 2 Statistics

The second experiment uses a custom street map. This map represents a city street where a truck would need to navigate 90 degree turns while moving multiple directions. Figure 37 shows the found path and the robots' traversing between the start position and goal position. The algorithm began by implementing the A* algorithm to find a path from start to goal point. It took 50,541 iterations for the A* to find a successful path. Next the path panner takes the points in the found path and converts a portion of them into waypoints. Out of the 1570 points in the path 43 were converted into waypoints. The algorithm then assigned theta values for each point. After each theta angle has been assigned a further reduction of waypoints are performed. The final number of waypoints in the finish path was 14. Figure 38 shows the output statistics for this experiment. The entire algorithm took 0.96 seconds to execute and complete.

Next, we initiated the truck simulation and monitored how well the p-controller navigated through the map. Overall, the program was successfully able to control the truck through the pathway without any visual errors or collisions.

## VI. CONCLUSION

In modern robotics good path planning is critical for effective navigation. A robot following a poor path can cause unnecessary wasted travel time or even cause the robot to collide with its environment. The goal of this lab is to develop an algorithm capable of generating an optimal path to take the robot from point A to point B.

We implemented a custom A* algorithm to generate our paths. After the development of our custom algorithm, we then conducted three experiments to determine the effectiveness of the code. Each map tests the programs' with progressively more difficult environments. Each test was a success, and the algorithm was capable of correctly generate an effective pathway from start to goal that would allow the robot to transverse the space. Depending on the map we had up to 50% reduction of waypoints while maintaining accurate robot motion down the desired pathway.

Overall, we are satisfied with the effectiveness and the level of versatility shown by our custom algorithm. The program has been tested and we believe it produces suitable data for use in future robot navigation tasks.

## VII. ACKNOWLEDGEMENTS

None

## VIII. REFERENCES

Jenkins, Jonathan. "ME 5751 Module 1: Proportional Control for Point Tracking." 2022. *ME5751_Module1.* <https://github.com/jonjenkins31/ME5751_Module1>.

Siegwart, Roland, Illah R Nourbakhsh and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. Second Edition. Massachusetts Institute of Technology, 2011.