CalPolyPomona

College of Engineering

Truck Simulator

BY,

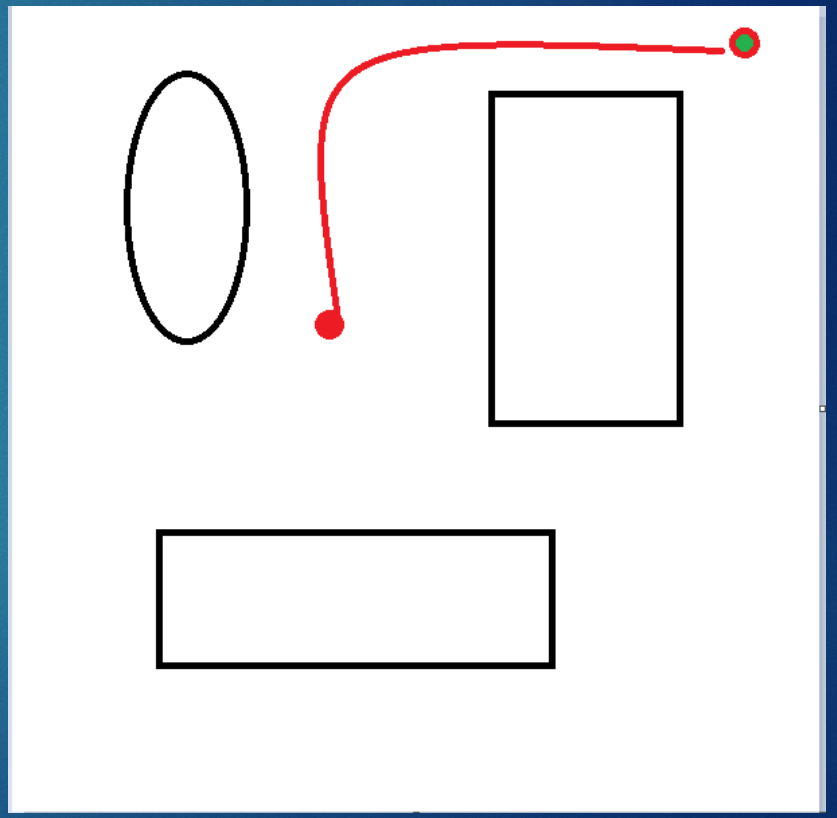
JONATHAN JENKINS

ME 5751: Robotic Motion Planning

Dec 8, 2022

Objective

- Develop a Truck Based motion planner
- Construct a collision - free path between some initial position to some goal position.
- Implement Truck kinematics to constrain the simulated robot motion

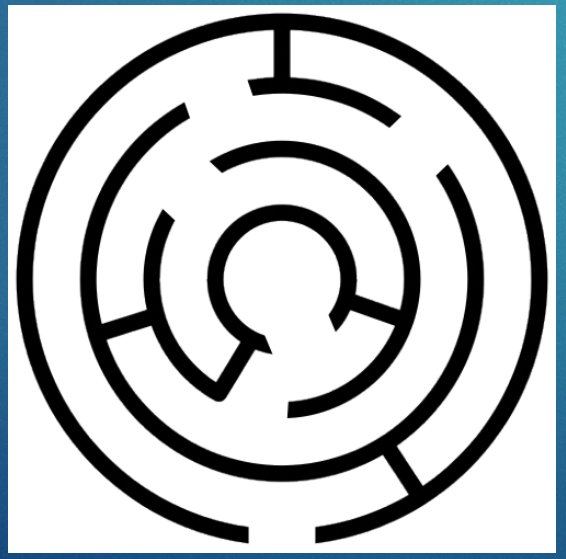


Requirements

- Rubric 1: Potential map generation
- Rubric 2: Path planner
- Rubric 3: P - controller
- Rubric 4: Wheel Kinematics of a truck

Potential Map Generation

- Takes 500*500-pixel map and converts it into a 500*500 matrix map
- Using a function to identify empty spaces (white) and occupied spaces (black).



```
#SET 2 A
# Using code created for hw2 . Run bushfire algorithu
# Make an array for Occupied and Unoccupied spaces
# # Initializing a array (queue)
self.land = []                                          #
self.water = []                                         #
# Identify Grid size
# A(i,j) = A(r,c)
self.row_length = len(grid)
self.col_length = len(grid[0])
 # Scan GRID Top Down -- LEFT to RIGHT
for r in range(self.row_length):
    for c in range(self.col_length):
        if grid[r,c] == 1:
            self.land.append((r,c))
        else:
            self.water.append((r,c))
 # Check to see if the grid contains atleast 1 land o
if  not self.land or not self.water:
    if  not self.land:
        Explination = "All cells are empty"
        print(Explination)                              #
    if  not self.water:
        Explination = "All cells are occupied"
        print(Explination)                              #
```

Potential Map Generation

- Assign a risk value to all empty spaces.
- Spaces closer to an occupied space will be given a higher risk, While spaces that are farther away are assigned a lower risk.
  - Black – 1000
  - Safety-zone – 850
  - Unoccupied space 800---- 0

```
#SET 3
# Convert the Bushfire map to a potential map / Cost map
# SPACES CLOSER TO AN OBJECT IS AT A HIGHER COST/VALUE
bushfire_costmap=Bushfire        # cost map ---- THIS MATRIX REPRESENTS THE COST MAP -
max_cost = 1000                                  # Specify the desired "cost" for an oc
max_cost_saftey_zone = 800                       # Specify the desired starting "cost"
min_cost_saftey_zone = 800                       # Specify the desired MINIMUM "cost" f
saftey_zone_distance = 18                        # Specify the desired saftey zone dist
cost_rate_safty=(min_cost_saftey_zone-max_cost_saftey_zone)/(saftey_zone_distance)
cost_rate = 15
```

Path Generation

- A planner decides what grids should the robot move to reach the desired location
- A* Algorithm
  - combines Breadth First Search (BFS) and Dijkstra algorithm
  - BFS algorithm assigns each grid a value corresponding to the distance from the starting position
  - Dijkstra's algorithm assigns each grid a value corresponding to the distance from the goal position

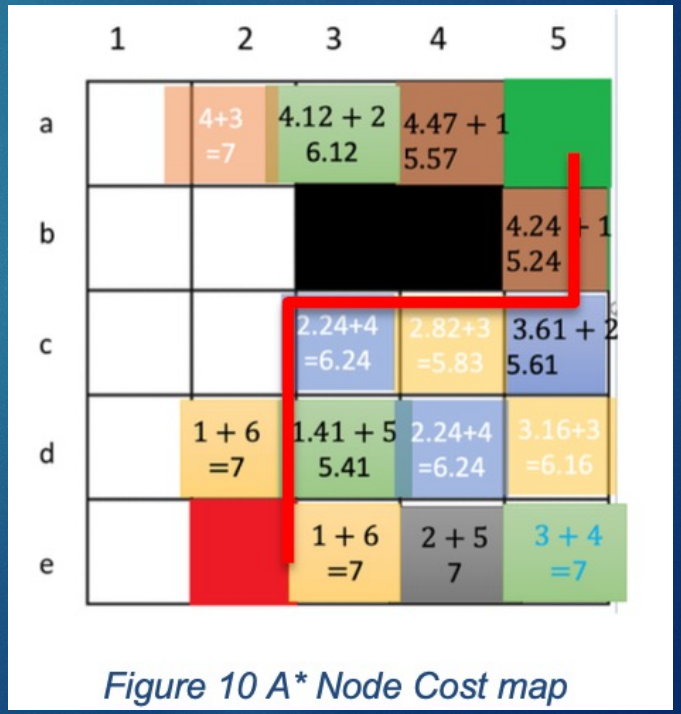


Figure 10 A Node Cost map*

Dual Search A* Algorithm

- The A* algorithm uses both BFS and Dijkstra's algorithm to find an efficient path between the start and the goal . A* algorithm priority search allows it to find a path without the need to search the entire map.

- We implement a dual A* search starting from both the goal position and the start position simultaneously.
- With a dual search the program can scan from two location congruently, ideally allowing the program to locate a final path faster.

A* Algorithm Improvements

Single Search

```
2000 iterations completed
5000 iterations completed
PATH TO GOAL FOUND
the number of nodes searched
123425
the number of iterations
32561
the number of points in path
```

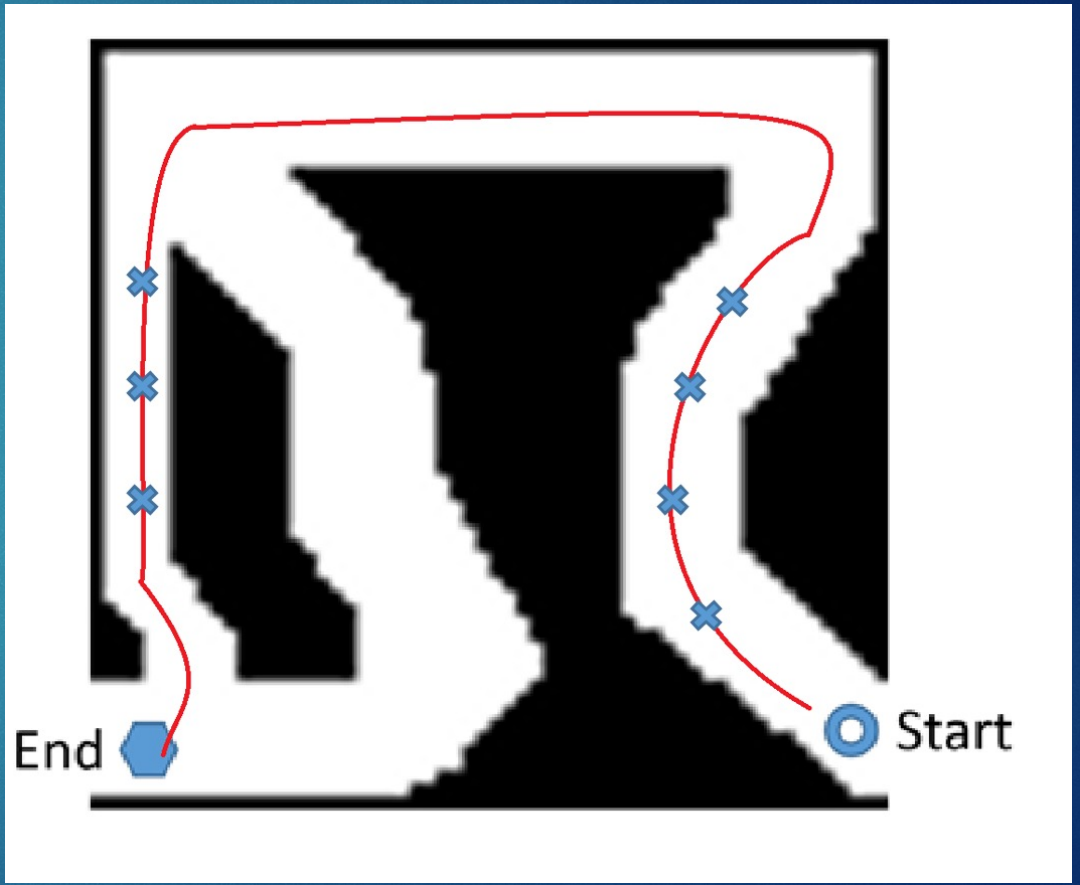
56%
Iterations
Reduction

Dual Search

```
2000 iterations completed
5000 iterations completed
PATH TO GOAL FOUND
the number of nodes searched
104126
the number of iterations
13658
```

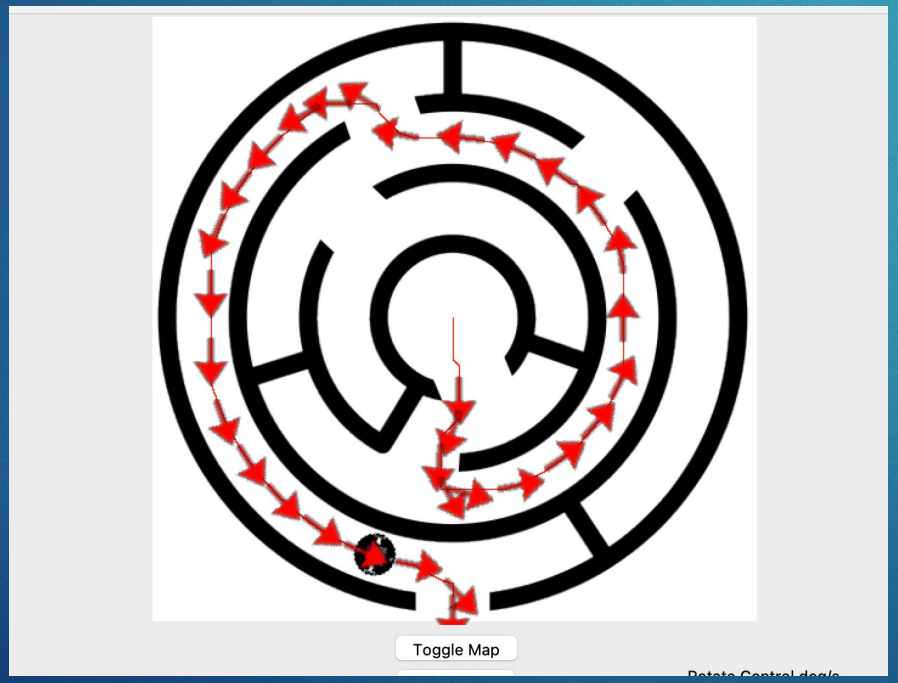
Point Tracking

- Converting our found path to robot waypoints.
- Point tracking
  - We discretize the paths into a set of waypoints (positions)
  - Assign a direction value to each waypoint
  - Reduce the number of waypoints in a path to maximize efficiency



Point tracking

- Create a list of waypoints from a fraction of points in the found path
- Each Waypoint is assigned a theta value based on the travel direction from the previous waypoint

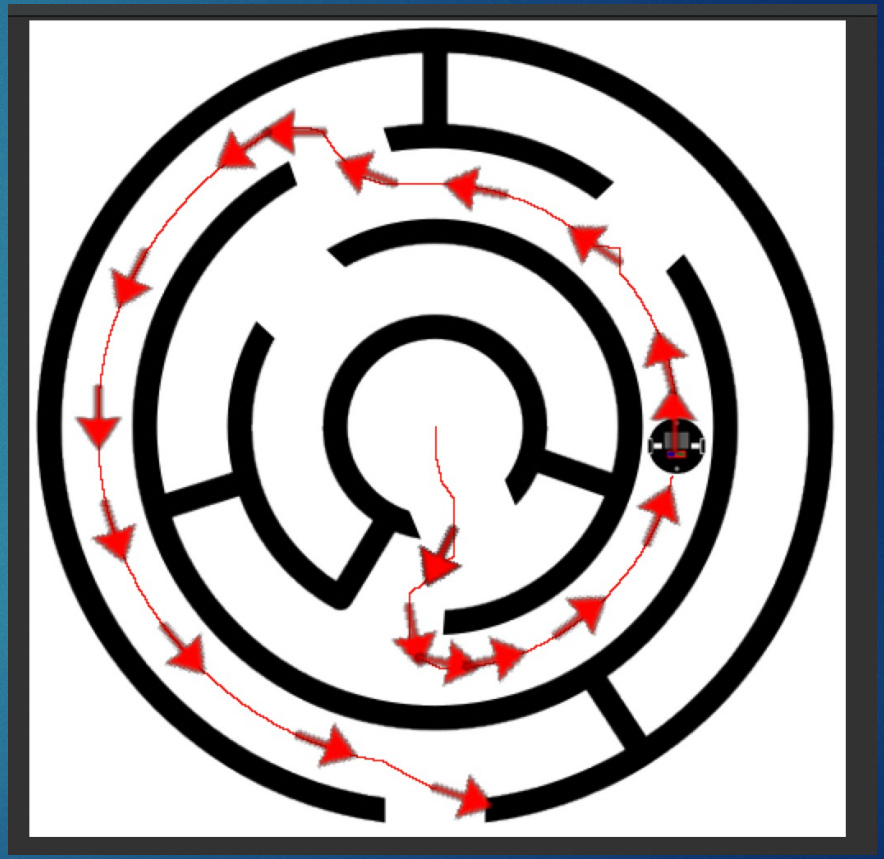



```
def find_theta(dx,dy):
    #Set singularities
    if dx == 0 and dy > 0:
        angle = 0
    elif dx < 0 and dy == 0:
        angle = math.pi/2
    elif dx == 0 and dy < 0:
        angle = math.pi
    elif dx > 0 and dy == 0:
        angle = -math.pi/2
        #angle = 1.5*math.pi


    #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
```

Point tracking

- Further reduction of waypoints based on linear distance and theta value.
- Sharp turns will have more waypoints for higher resolution
- Strait paths will have less points



```
#SET 5    PATH OPTIMIZATION                                                    # CH
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:
            if point_n1_collision_free ==False or  point_n1_distance >= waypoint_max_distance:
                reduced_points.append(points[x])
                last_saved=point_n
        else:
            print("path optimization error")
```

Controller

- Proportional control (P control) is a type of linear feedback control system in which a correction is applied to the controlled variable.
  - Error (e) : the difference between the desired value and the measured value.
  - Proportional gain (k) : the size of the correction that will be implemented
- A P-Control was used to adjust the robot's linear and angular velocity in order to control it to reach a desired point (goal point)

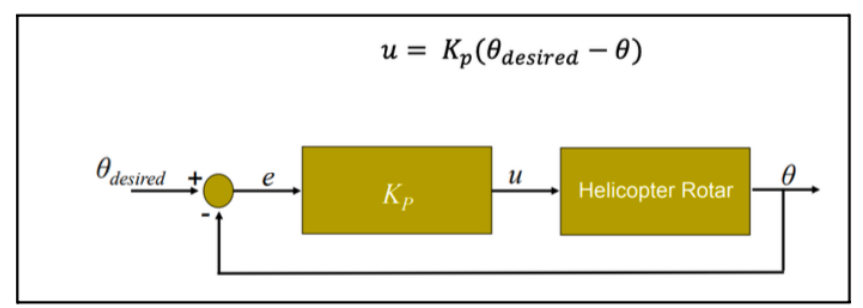


Figure 2. Proportional Control Diagram

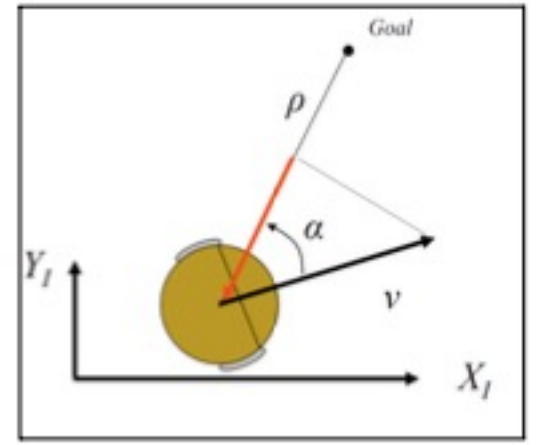
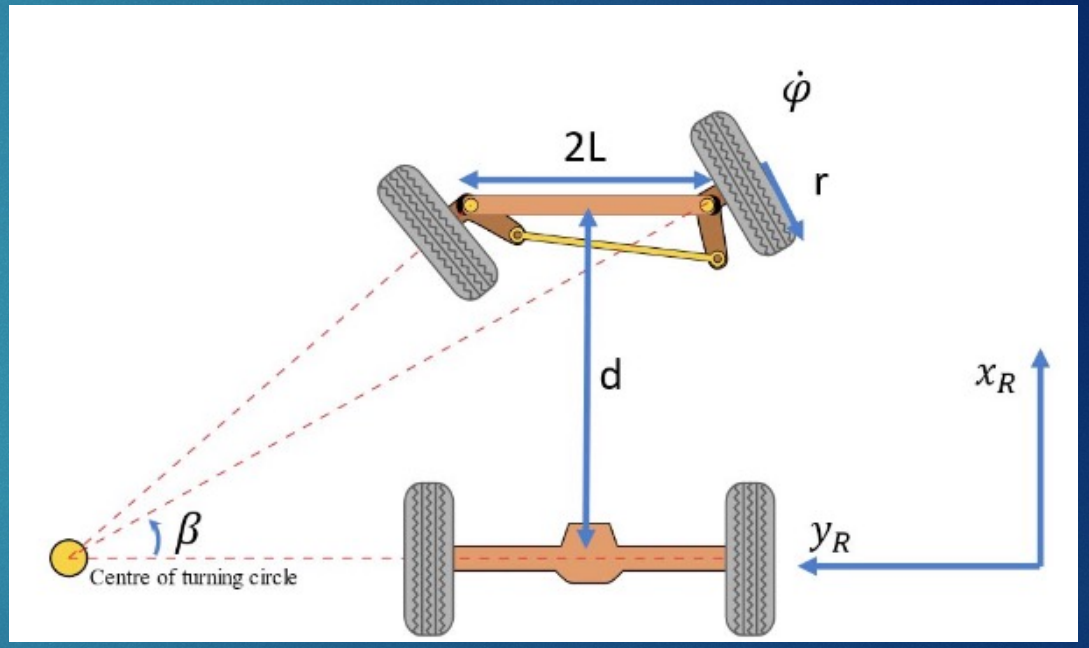


Figure 3. Robot and Goal Point

Truck Dynamics

- A truck has certain kinematic constraints . Based on the vehicle design and limitations we need to implement limitations to allow for accurate simulation.
  - Car width : 2L = 16
  - Car axile distance : d = 20
  - Wheel diameter : r = 3
  - Steering angle : b < 40 Degrees
  - Wheel speed : rho < 20 rad/s

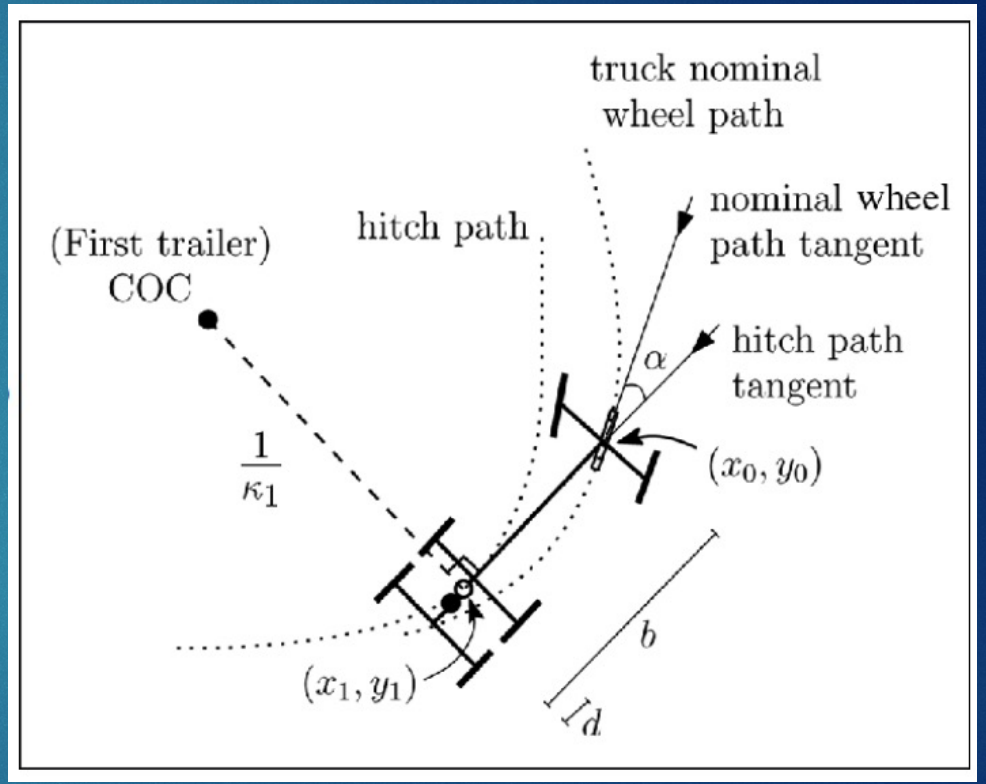


$$\dot{x} = s\cos\theta$$
$$\dot{y} = s\sin\theta$$
$$\dot{\theta} = \frac{s}{l}\tan\phi \approx \frac{s}{l}\phi$$

$$\phi_i = \tan^{-1}\left(\frac{2l\sin\phi}{2l\cos\phi - w\sin\phi}\right)$$
$$\phi_o = \tan^{-1}\left(\frac{2l\sin\phi}{2l\cos\phi + w\sin\phi}\right)$$

Truck Dynamics

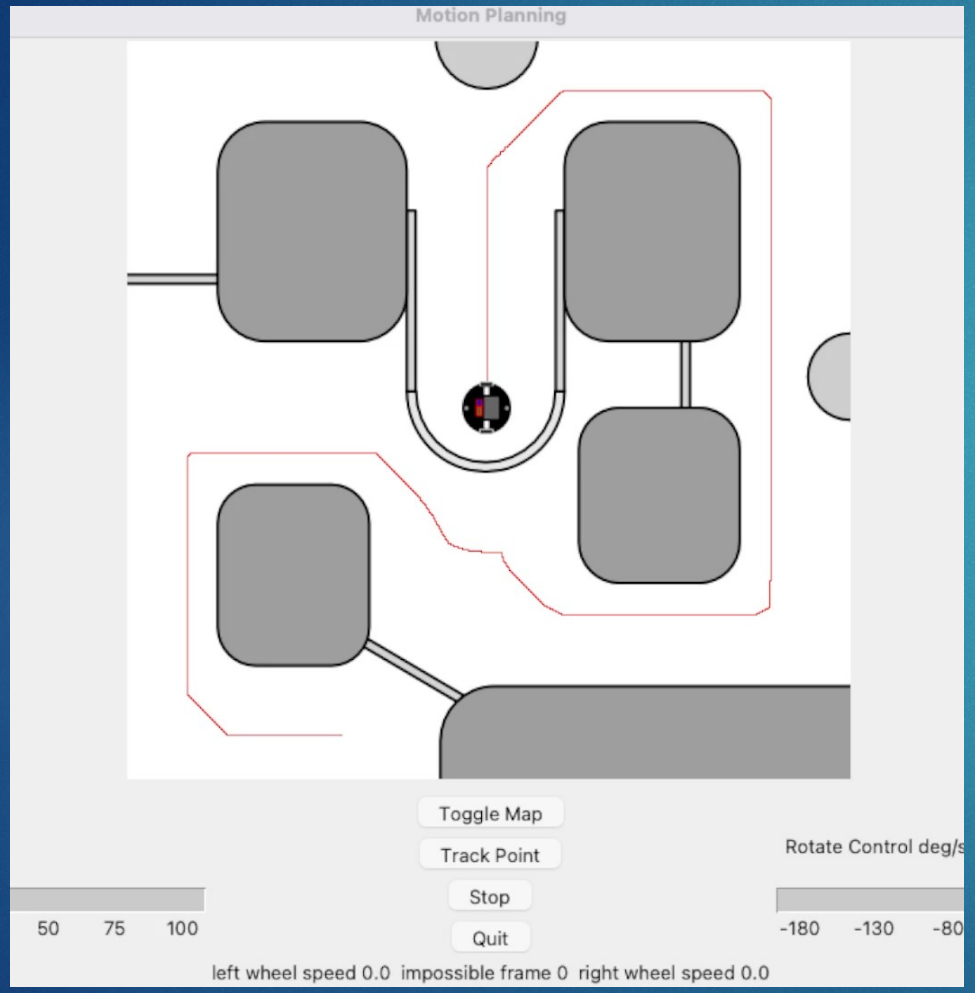
- Incorporating the desired velocity and rotation from the p-controller and limiting the output based on the truck kinematics
  - P Controller + Truck Kinematics
  - Limit outputs based on known constraints
  - Outputs the final vehicle linear and angular velocity that will be executed.



$$\begin{aligned} \dot{x} &= s\cos\theta \\ \dot{y} &= s\sin\theta \\ \dot{\theta} &= \frac{s}{l}\tan\phi \approx \frac{s}{l}\phi \end{aligned}$$

$$\begin{aligned} \phi_i &= \tan^{-1}\left(\frac{2l\sin\phi}{2l\cos\phi - w\sin\phi}\right) \\ \phi_o &= \tan^{-1}\left(\frac{2l\sin\phi}{2l\cos\phi + w\sin\phi}\right) \end{aligned}$$

Demonstration




```
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
93
the number of points in Final path
25
```

```
--- 0.9919109344482422 seconds ---
```

Link To Online Video

- https://youtu.be/2k3NQvptN9E