Aadiv Shah
WPI ID 413770228

# Assignment 4

RBE 550
Motion Planning

**Valet**

# Introduction

A common path planning problem for autonomous vehicles involves maneuvering in tight spaces and cluttered environments, particularly while parking. Create a simulated world and park three vehicles of increasing ridiculousness into a compact space. Planners must take into account vehicle kinematics and collision. The vehicles are -

1. Delivery Robot (differential drive robot)
2. Police Car (Ackerman steered)
3. Truck with trailer

# Implementation

## Environment Generation

Using **Pygame**, I simulated the continuous space of the environment with an obstacle and 2 other vehicles towards the bottom of the world as shown in Fig. 1.

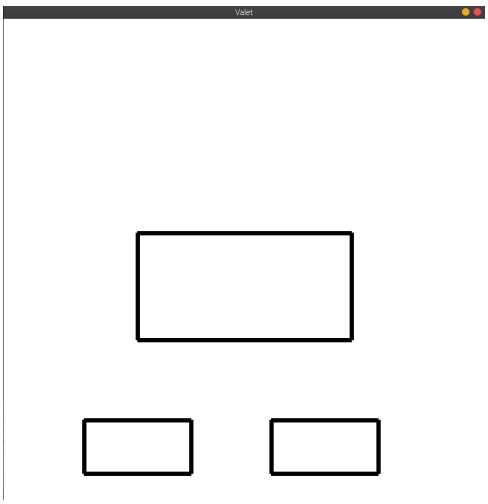


Figure 1: Continuous Space environment with Vehicles at the bottom and an obstacle in the middle.

Each of the obstacles were made using *pygame.draw.line()* for convenience while performing collision checking.

## Robot Framework

Keeping in mind modularity, I developed several classes keeping in mind the properties of the robot e.g *Robot* class, *Controller* class and *Planner* class. I also developed a *World* and *Visualiser* class to define the world and display the environment + robot on pygame. Finally all the classes and their respective methods are called sequentially in the main function.

## Planning Algorithms

The A* planning algorithms was inspired by Sakai, Atushi, et al. [1]. The robots may move to any $8-$*connected* neighbour. Obstacle avoidance is achieved by inflating the obstacles. *Collision checking* is performed by observing if any of the robot lines are intersecting with the lines of the obstacle. The code to perform line intersection is inspired from intersects [2].

Separate class files were created to keep the code organized. Inside a while loop, the controller is *stepped* and the position of the robots is updated. Finally the current state of the world and robots is updated and displayed.

All the 3 robots used the same underlying planner - A* Search algorithm. A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

```
    A* (start, goal)
1.  Closed set = the empty set
2.  Open set = includes start node
3.  G[start] = 0, H[start] = H_calc[start, goal]
4.  F[start] = H[start]
5.  While Open set ≠ ∅
6.      do CurNode ← EXTRACT-MIN- F(Open set)
7.      if ( CurNode  == goal ), then return BestPath
8.      For each Neighbor Node N of CurNode
9.          If ( N is in Closed set ), then Nothing
10.         else if ( N is in Open set ),
11.             calculate N's G, H, F
12.             If ( G[N on the Open set] > calculated G[N] )
13.                 RELAX(N, Neighbor in Open set, w)
14.                 N's parent=CurNode & add N to Open set
15.         else, then calculate N's G, H, F
16.                 N's parent = CurNode & add N to Open
```

Figure 1. A* Algorithm pseudo code

After the planner returns the trajectory, I fed these waypoints to the robot controller and this moved the robot to the specific waypoints.

There were some modifications that needed to be made for the robot to follow the planned trajectory correctly -

1. Differential drive - no modification needed as the robot can execute 0 radius turns
2. Ackerman and Ackerman + Trailer - if the next waypoint was lying within the minimum turning radius of the system, skip that waypoint otherwise the robot moves in circles indefinitely trying to reach that waypoint.

# Results

Apart from all the screenshots provided in this report, there are videos for all the 3 robots and their paths. I have implemented all **vehicle dynamics** by giving **control inputs as velocities**. The controller includes **time stepping** for more realistic results and making sure the motion on the animation is constant irrespective of the FPS.

The models of the following robots are synonymous with the velocity equations as described in Lavelle.

```
class Controller:
    def __init__(self, robot: Robot) -> None:
        self.robot = robot
    def step(self):
        # theta = pi - self.robot.angle * pi/180
        theta = self.robot.angle * pi/180
        self.robot.x_coord += ((self.robot.vel_l+self.robot.vel_r)/2)*cos(theta) *\
                              self.robot.dt
        self.robot.y_coord += ((self.robot.vel_l+self.robot.vel_r)/2)*sin(theta) *\
                              self.robot.dt
        self.robot.angle += atan2((self.robot.vel_r -\
                self.robot.vel_l),self.robot.height)*180/pi * self.robot.dt

# in the while True loop
    # dt
    self.robot.dt = (pygame.time.get_ticks() - lasttime)
    lasttime = pygame.time.get_ticks()
```

### The Delivery Robot

The planning for this type of differential drive robot is relatively simpler as it can execute zero-radius turns. The plot with the center of the rear axle is shown in Fig. 2. Clearly the robot moves avoiding all obstacles and orients itself in the correct direction at the end of

the path. I implemented a **simple P controller** to move the robot forward for smoother trajectory following.

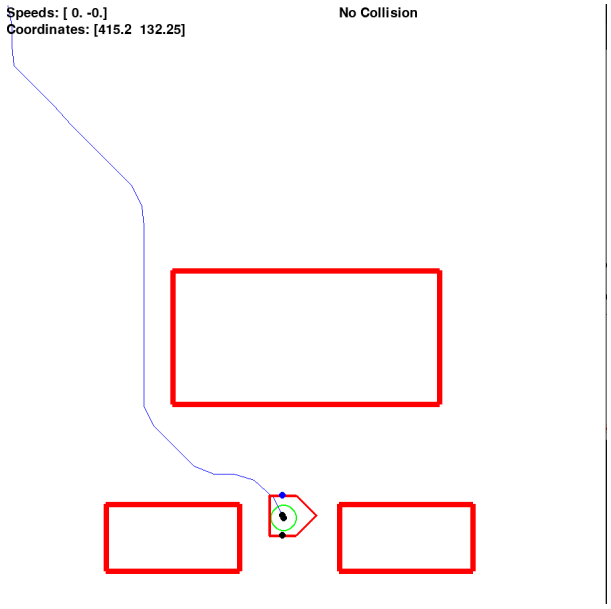


Figure 2: Path taken by the Delivery Robot

### The Police Car

The planning for this type of ackerman robot seemed difficult at first to model using kinematic velocity equations. My approach was to model the the robot using 2 control inputs - steering angle to determine turning radius and the speed of the vehicle. Then I adjusted the velocity of the left and right wheels such that they move along a particular radius as determined by the steering angle. The results of the path are shown in Fig. 3. For the steering I implemented a **simple P controller** to move smoothly towards the goal.

Note - the path is completely smooth and made of splines.

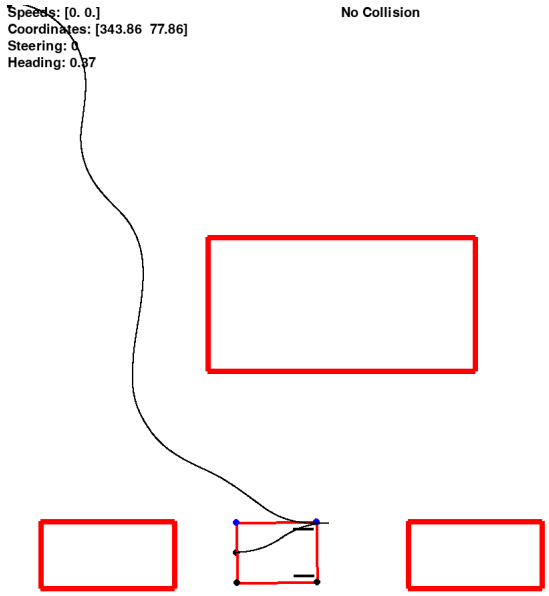


Figure 3: Path taken by the Police Car

### The Truck + Trailer

Building up on the ackerman robot, the trailer behaved as an independent robot with ackerman steering - the steering angle was decided by the heading of the robot it is connected to (truck). The results are shown in Fig. 4

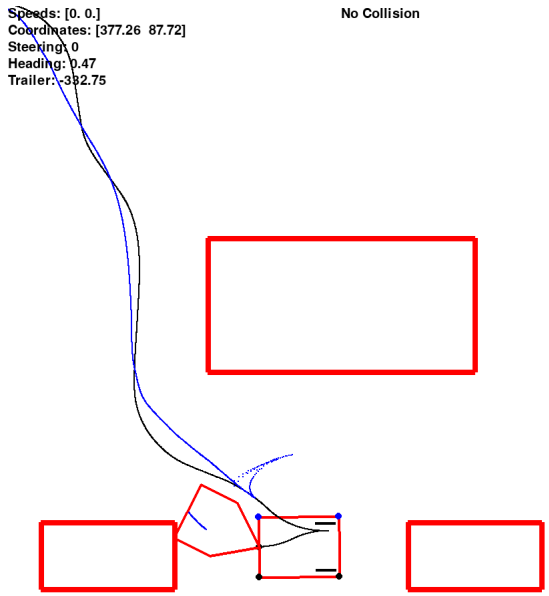


Figure 4: Path taken by the Truck + Trailer (black: Truck Axle, blue: Trailer Axle)

# Discussion

After discretizing continuous space - working with the discrete planners from Flatland was a breeze. Modelling the robot was a challenge but building up from the differential drive robot was effective. The trailer proved to be particularly difficult to handle while reversing due to its highly non-linear nature. In all the video submissions telemetry information of speed, coordinate, heading and steering angle is displayed on screen.

# References

[1] Sakai, Atsushi, et al. "Pythonrobotics: a python code collection of robotics algorithms." arXiv preprint arXiv:1808.10703 (2018).

[2] How to check if two given line segments intersect?

# Appendix

Some diagrams + rough work

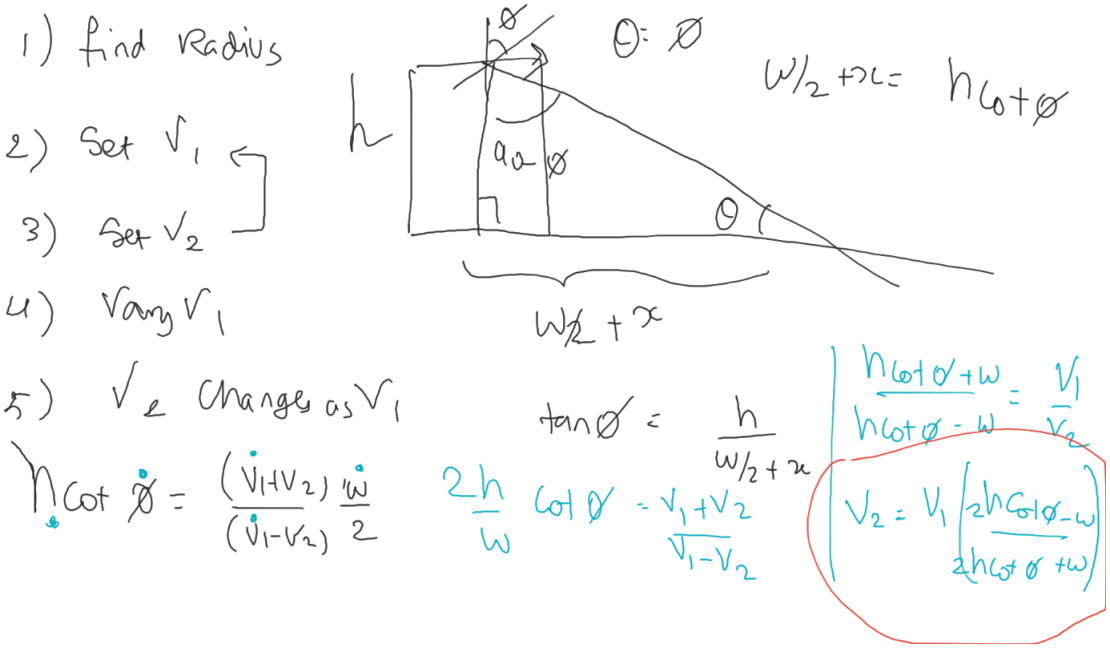


Figure 5: Calculations to relate steering angle and turning radius

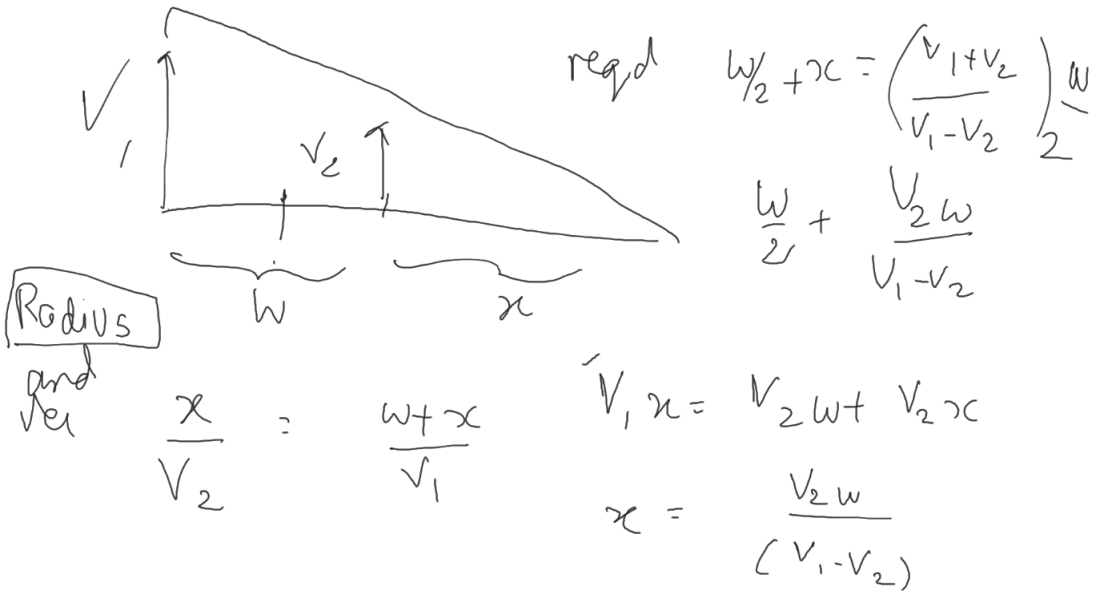


Figure 6: Calculations for adjusting $V1$ and $V2$ for ackerman robot