Assignment 4
Valet

In this assignment, we moved from discrete path planning to continue path planning where the vehicle kinematics is to be considered for exploration. In this assignment I have created a simulated world and parked three vehicles of variable size into a compact space. Planners must take into account collisions. Given vehicles are: Police Car, Delivery Robot and Truck with trailer

Environment: I have created the environment and simulated it using pygame. The environment consists of an obstacle and 2 other vehicles at the bottom of the world as shown below. These obstacles were made by using pygame.draw.line() in order to check the collision easily while simulating in real time. I have created classes like World and Visualiser separately for the user to debug easily if something goes wrong with the environment or the agent. All the classes and their respective methods are called sequentially in the main function.

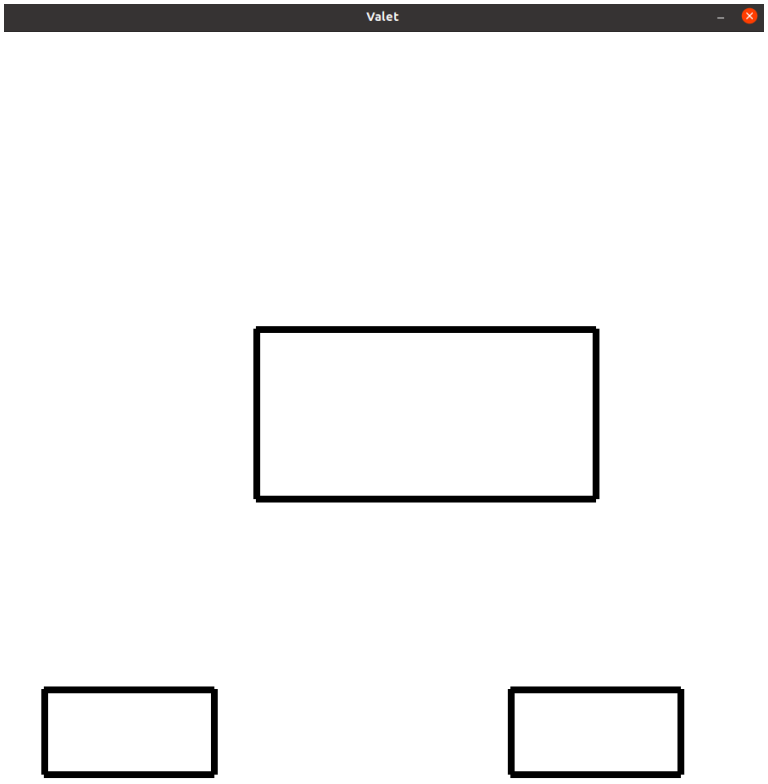


Figure 1: Environment Simulation

Robot: To make the agent/car more robust. I have created several classes that would take into consideration various parameters of the agent. Like, Controller class, Planner class and Robot class.

Algorithm:

The A* Algorithm[1]: The robot would move to any of the 8 - connected neighbour.

The collision checking is performed by observing if any of the robot/agent lines are intersecting/overlapping with the lines of the obstacle. (The pygame.draw.line function made this easily).[2]

To make the code more organized and neat for the user, separate classes were created. A controller class is stepped and the position of the robot/agent is updated in the while loop. Simultaneously, the state of the world and robots is updated and displayed.

For the given vehicles they used the same underlying planner i.e. A* 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 traveled, 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)
Closed set = the empty set
Open set = includes start node
G[start] = 0, H[start] = H_calc[start, goal]
F[start] = H[start]
While Open set ≠ Ø
    do CurNode ← EXTRACT-MIN- F(Open set)
    if ( CurNode  == goal ), then return BestPath
    For each Neighbor Node N of CurNode
        If ( N is in Closed set ), then Nothing
        else if ( N is in Open set ),
            calculate N's G, H, F
            If ( G[N on the Open set] > calculated G[N] )
                RELAX(N, Neighbor in Open set, w)
                N's parent=CurNode & add N to Open set
        else, then calculate N's G, H, F
                N's parent = CurNode & add N to Open
```

Figure 2: A* Search Algorithm Pseudocode

After the planner returns the trajectory, I feed these waypoints to the robot controller and this moves 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. 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.
2. Differential drive - no modification needed as the robot can execute 0 radius turns.

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

        self.robot.angle = self.robot.angle % 359
```

Figure 3: The Controller Class

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 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.

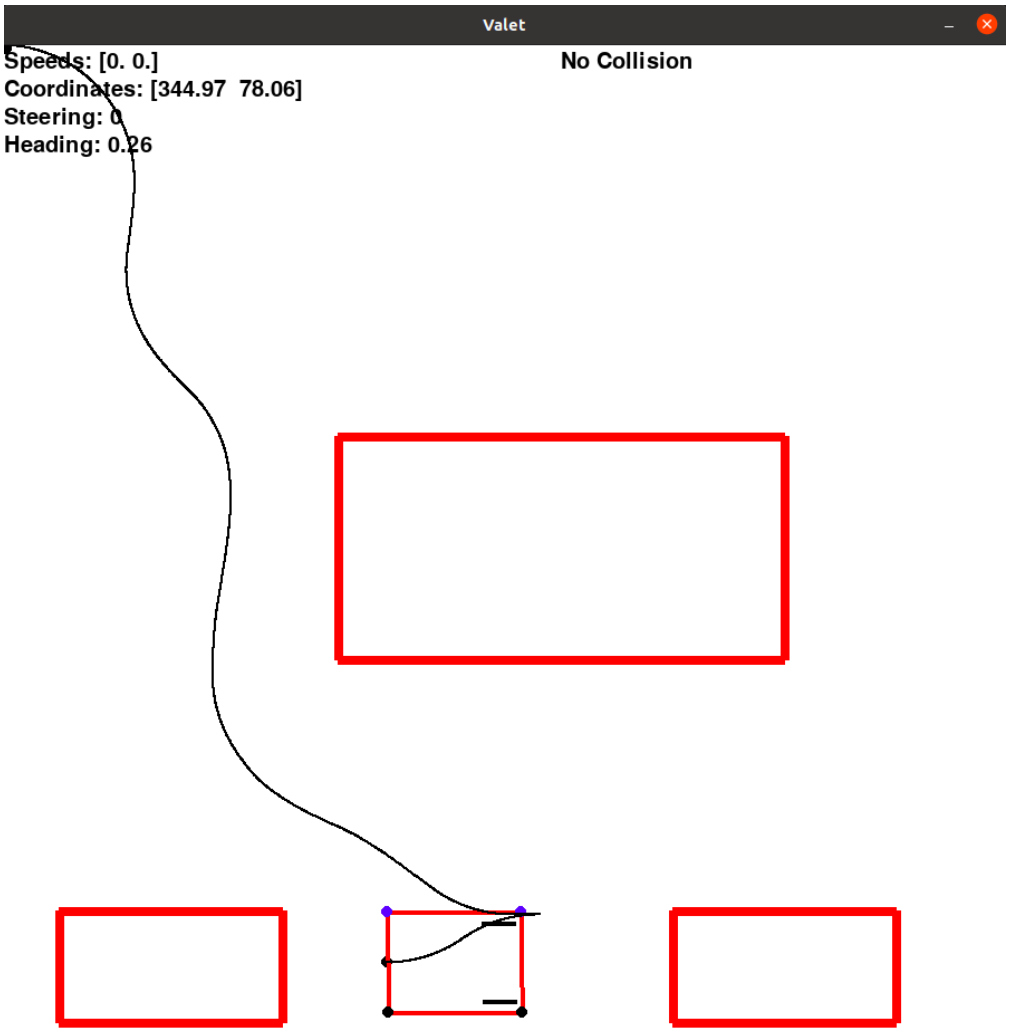


Figure 4: Path traced by Police Car using Ackerman

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 figure below

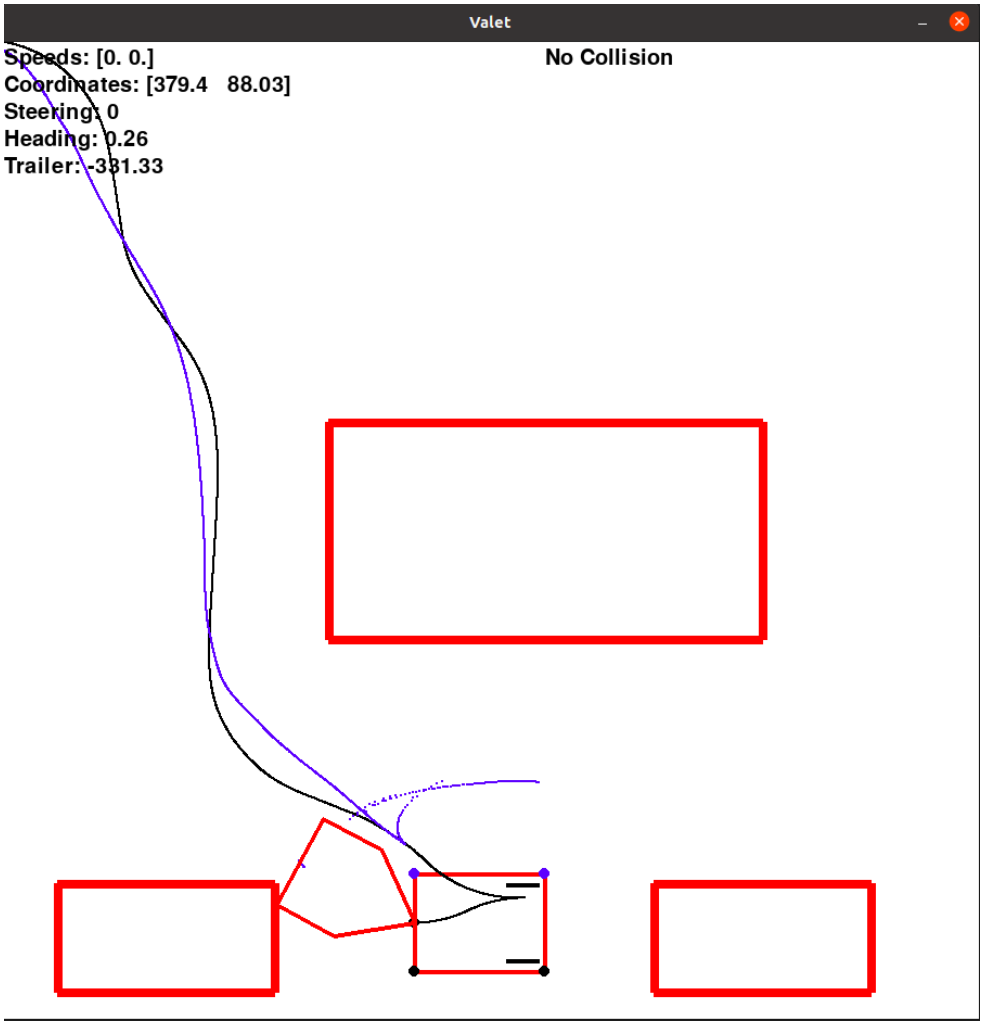


Figure 5: Path traced by Trailer using Ackerman

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.

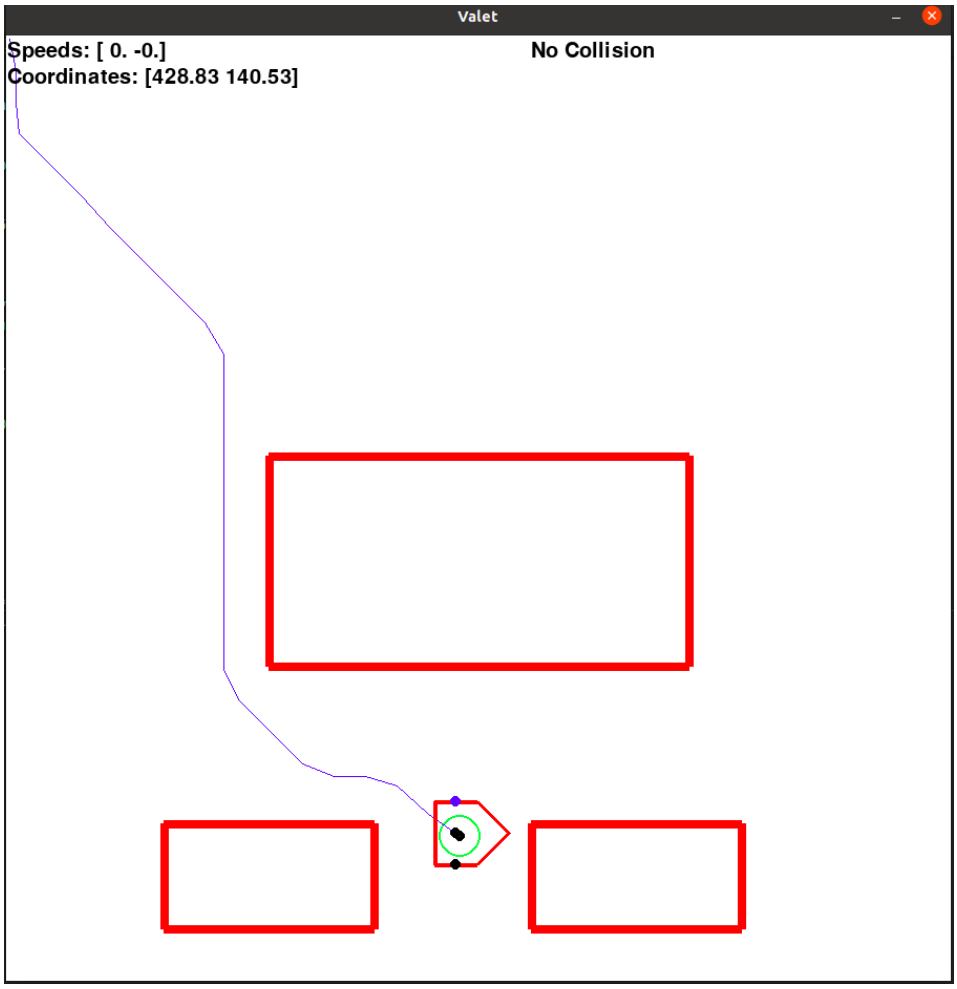


Figure 6: Path traced by Delivery Robot using differential

Discussion:

After discretizing continuous space - working with the discrete planners from Flatland was a breeze. Modeling 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?

[3] https://github.com/RitikJain12/Parking_different_vehcles

Appendix:
Some rough work

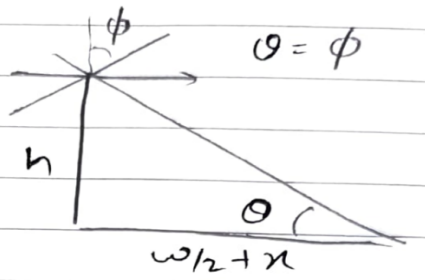
1) for Radius

2) Set V_1

3) Set V_2

4) V_{avg} V_1

5) V_2 changes as V_1



$$\tan\phi = \frac{h}{w/2 + x}$$

$$h\cot\phi = w/2 + x$$

So, $$\frac{h\cot\phi + w}{h\cot\phi - w} = \frac{V_1}{V_2}$$

$$\frac{2h}{w}\cot\phi = \frac{V_1 + V_2}{V_1 - V_2}$$

$$V_2 = V_1\left(\frac{2h\cot\phi - w}{2h\cot\phi + w}\right)$$

eq. $$\frac{w}{2} + x = \left(\frac{V_1 + V_2}{V_1 - V_2}\right)\frac{w}{2}$$

$$\frac{w}{2} + \frac{V_2\, w}{V_1 - V_2}$$

$$\frac{x}{V_2} = \frac{w + x}{V_1}$$

On solving

$$x = \frac{V_2\, w}{(V_1 - V_2)}$$