Name: Camilo Girgado & Jason Patel

Class: RBE 550

Assignment: Valet

Task

In this assignment, we were tasked with using kinematic planning to effectively park vehicles of various drivetrains and configurations. We were to create a world environment and implement a path planner to navigate the nonholonomic constraints and obstacles present.

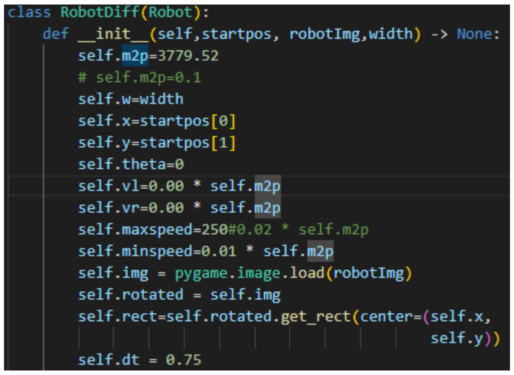
Approach

The first vehicle for this assignment incorporated skid steering (diwheel kinematics), the second vehicle incorporated Ackermann steering, and the third vehicle incorporated a trailer pulling application. We'll get into more detail about the scripting for those drivetrains below.

For the first differential robot we used the tutorial from Agrobotics (see Reference 1) to develop the base pygame setup with manual control of each wheel. We then expanded the tutorial to also include a manual drive Ackermann robot. We then added the lattice and state data structures and removed the manual drive for the A* search driven approach to moving the vehicle.

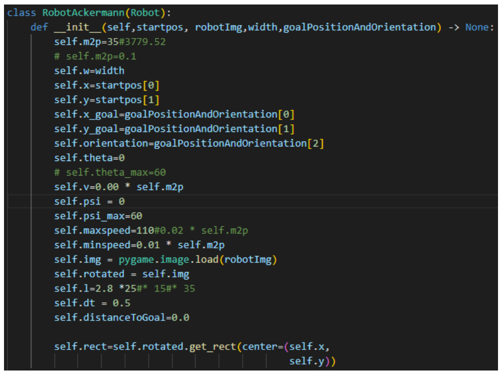
Our environment was built up using pygame and has the details of the environment stored in our environment.py class. We define a list of colors that can be used as tuples, as well as the overall map dimensions in width and height.

Our robots are initialized with their start positions, initial velocities, and orientations, as well as their image file in the classes shown below.



```python
class RobotDiff(Robot):
    def __init__(self,startpos, robotImg,width) -> None:
        self.m2p=3779.52
        # self.m2p=0.1
        self.w=width
        self.x=startpos[0]
        self.y=startpos[1]
        self.theta=0
        self.vl=0.00 * self.m2p
        self.vr=0.00 * self.m2p
        self.maxspeed=250#0.02 * self.m2p
        self.minspeed=0.01 * self.m2p
        self.img = pygame.image.load(robotImg)
        self.rotated = self.img
        self.rect=self.rotated.get_rect(center=(self.x,
                                                self.y))
        self.dt = 0.75
```

- Figure 1: Differential Drive -



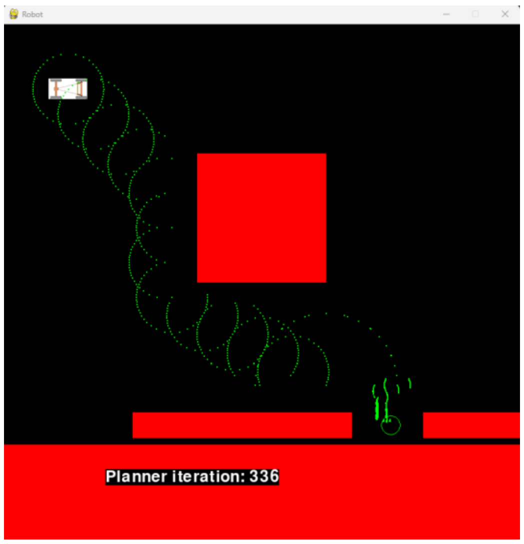
```python
class RobotAckermann(Robot):
    def __init__(self,startpos, robotImg,width,goalPositionAndOrientation) -> None:
        self.m2p=35#3779.52
        # self.m2p=0.1
        self.w=width
        self.x=startpos[0]
        self.y=startpos[1]
        self.x_goal=goalPositionAndOrientation[0]
        self.y_goal=goalPositionAndOrientation[1]
        self.orientation=goalPositionAndOrientation[2]
        self.theta=0
        # self.theta_max=60
        self.v=0.00 * self.m2p
        self.psi = 0
        self.psi_max=60
        self.maxspeed=110#0.02 * self.m2p
        self.minspeed=0.01 * self.m2p
        self.img = pygame.image.load(robotImg)
        self.rotated = self.img
        self.l=2.8 *25#* 15#* 35
        self.dt = 0.5
        self.distanceToGoal=0.0

        self.rect=self.rotated.get_rect(center=(self.x,
                                                self.y))
```

- Figure 2: Ackerman Drive -

Later while debugging, we found it easier to visualize our lattice search by printing a dot in green of each lattice state we explore.

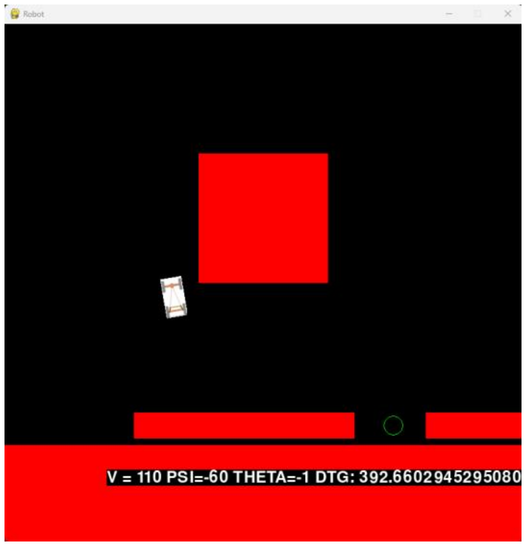
Ackermann

In this configuration, the front of our robot has steerable wheels whose pose is shown with the variable psi. Because this robot has steering independent from its drivetrain, it is no able to make turns of small radius like the differential drive robot can.



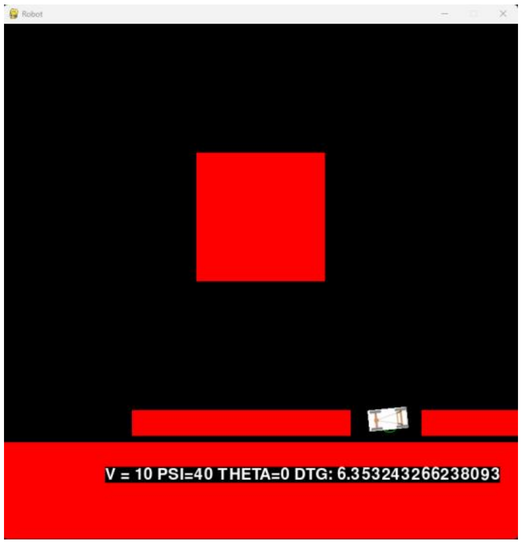
Planner iteration: 336

- Figure 3: Planner through iteration 336 -



V = 110 PSI=-60 THETA=-1 DTG: 392.6602945295080

- Figure 4: Approaching the goal (green circle) -

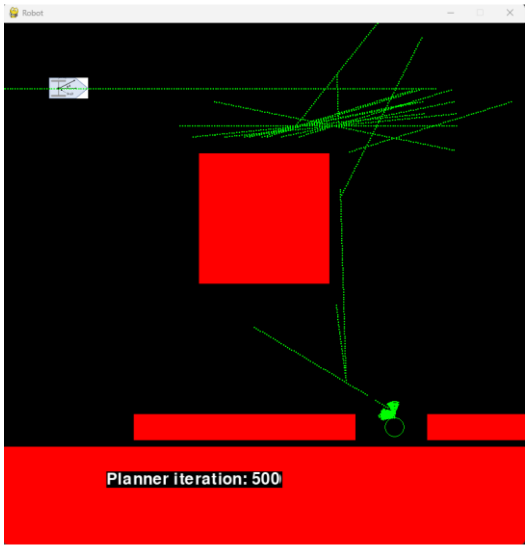


V = 10 PSI=40 THETA=0 DTG: 6.353243266238093

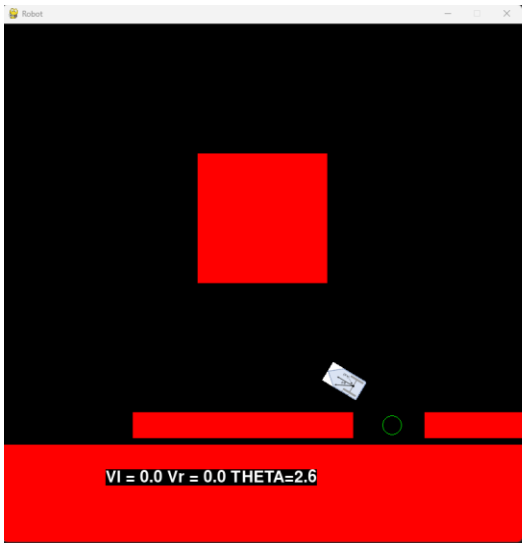
- Figure 5: Final parked state –

Differential Drive

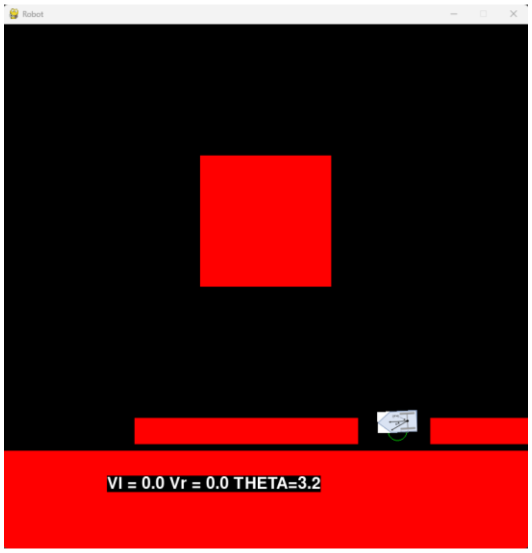
In this configuration, we control both rear wheel velocities independently with our vL and vR variables (these velocities are then converted into pixels). Because each wheel is able to be independently driven, the robot can make zero radius turns, a fault that the Ackermann robot needed to overcome above.



- Figure 6: Planner through iteration 500 -



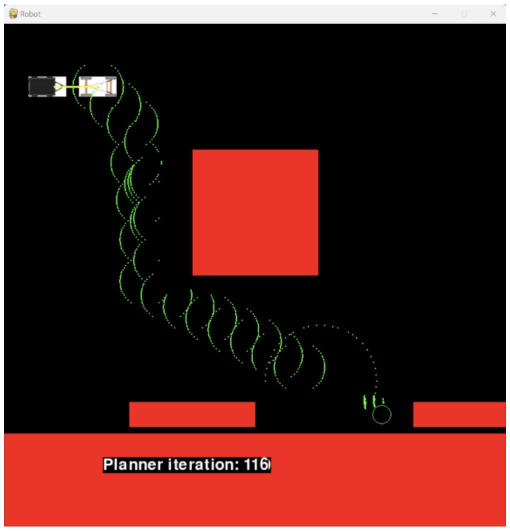
- Figure 7: Approaching the goal (green circle) -



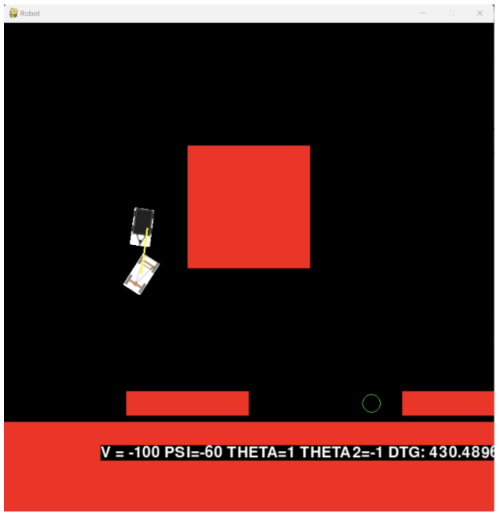
- Figure 8: Final parked state –

Trailer

The third configuration compounds on the Ackermann robot with the addition of a trailer. The trailer position is calculated with differential x and y terms and displays its pull angle with the variable theta2. These differential positions are calculated using the theta2 angle.

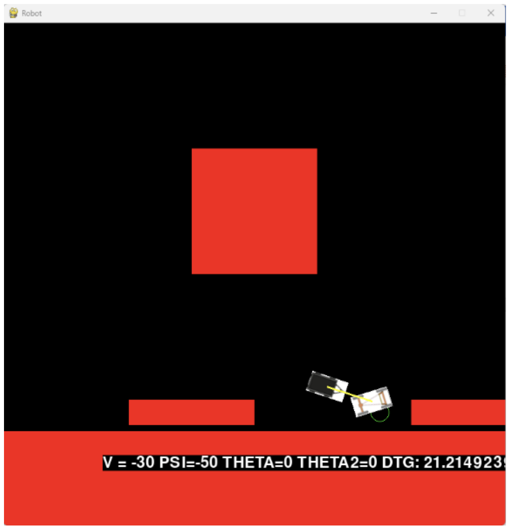


- Figure 9: Planner through iteration 500 -



V = -100 PSI=-60 THETA=1 THETA2=-1 DTG: 430.4896

- Figure 10: Approaching the goal (green circle) -



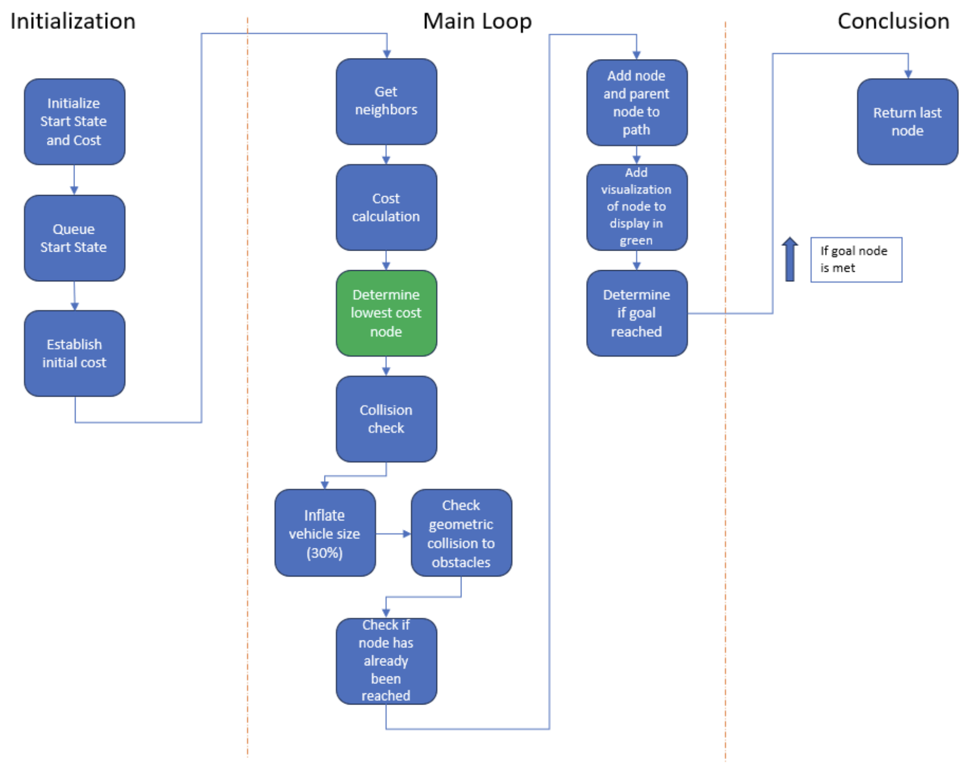
V = -30 PSI=-50 THETA=0 THETA2=0 DTG: 21.214923

- Figure 11: Final parked state -

Lattice Planner

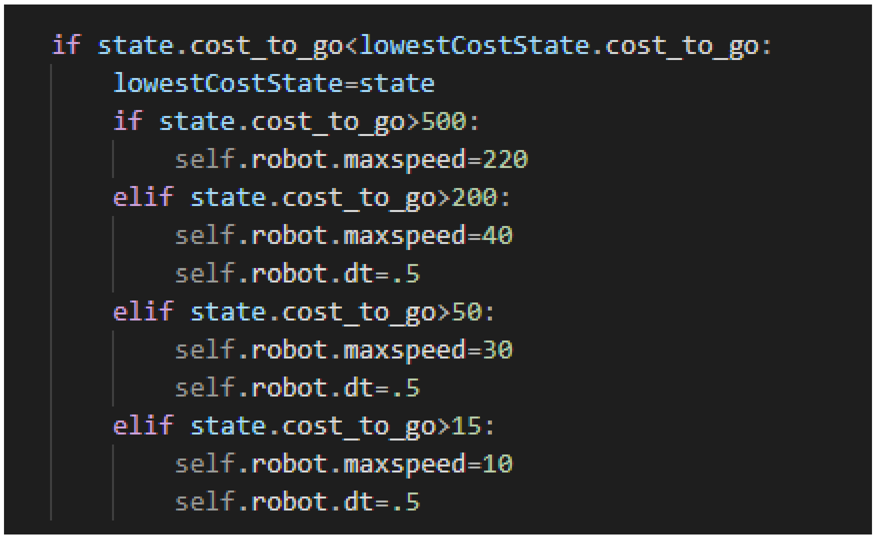
Our planner consists of the A* search algorithm and a exploring state lattice. The nodes of the lattice are placed into a priority queue, with a heuristic cost function that calculates the distance between the start state and each state plus a weighted current state to the goal. The time increment of 0.5 is used to calculate the x-dot, y-dot, and theta-dot calculations representing the kinematics of each vehicle type.

Flow Diagram


Initialization
Main Loop
Conclusion
Initialize Start State and Cost
Queue Start State
Establish initial cost
Get neighbors
Cost calculation
Determine lowest cost node
Collision check
Inflate vehicle size (30%)
Check geometric collision to obstacles
Check if node has already been reached
Add node and parent node to path
Add visualization of node to display in green
Determine if goal reached
If goal node is met
Return last node

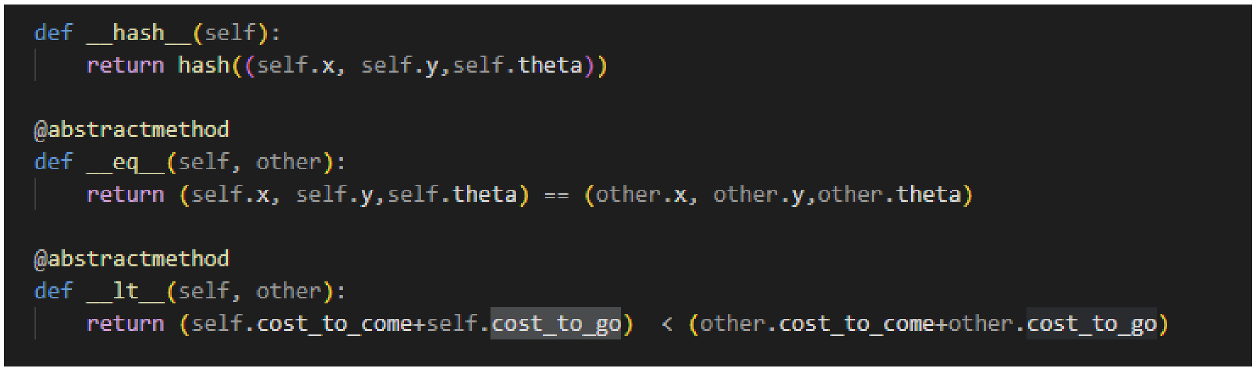
Challenges

One of the challenges was converging on the goal with the correct theta as it was proving difficult to find an exact solution within a reasonable time. To overcome this, we start with a high velocity and then reduce the velocity as we get closer to the goal. We tried changing the time increment, but this did not yield the benefits from the velocity change alone. With this change utilized we were able to limit the iteration count to 500 iterations so that the time waiting for the result was reasonable.



```
if state.cost_to_go<lowestCostState.cost_to_go:
    lowestCostState=state
    if state.cost_to_go>500:
        self.robot.maxspeed=220
    elif state.cost_to_go>200:
        self.robot.maxspeed=40
        self.robot.dt=.5
    elif state.cost_to_go>50:
        self.robot.maxspeed=30
        self.robot.dt=.5
    elif state.cost_to_go>15:
        self.robot.maxspeed=10
        self.robot.dt=.5
```

Another problem encountered was using the PriorityQueue mechanism in Python and checking for duplicate entries in the path and cost data structures. We simply overridden the hash, lt, and eq methods in Python on the state to satisfy the PriorityQueue and dictionary requirements.



```
def __hash__(self):
    return hash((self.x, self.y,self.theta))

@abstractmethod
def __eq__(self, other):
    return (self.x, self.y,self.theta) == (other.x, other.y,other.theta)

@abstractmethod
def __lt__(self, other):
    return (self.cost_to_come+self.cost_to_go)  < (other.cost_to_come+other.cost_to_go)
```

Another issue to overcome was to adequately use the pygame environment with regards to pixel to meters conversion, and secondly, that the y axis is in opposite direction of the screen when displayed.

Conclusions

Overall, this was a very rewarding assignment because of the excitement of when the agent robot can reach and park in the spot as intended. The resulting motion was deterministic as well, meaning that when the same parameters were used to run multiple episodes the resulting motion trajectory was the same.

References

https://www.youtube.com/watch?v=ZekupxukiOM&list=PL9RPomGb9IpTOGS6xjuIb8WdwmmDQOt6L&index=1