RBE 550: Assignment 3 - Valet

Mandeep Singh
M.S. Robotics Engineering
Worcester Polytechnic Institute (WPI)
Email: msingh2@wpi.edu

***Abstract*—A common path planning problem for autonomous vehicles involves maneuvering in tight spaces and cluttered environments, particularly while parking. In this assignment we will do path planning for for three different type of vehicles while taking into account the vehicle kinematics and obstacles present in the environment. The three different type of vehicles are diwheel (Differential drive) robot, car (ackerman drive) and a trailer. We have to plan path for parking of these vehicles in the given environment.**

***Index Terms*—Motion planning, Search algorithms.**

I. Environment

To simulate the given environment I have made a grid of size 200x200 in which there is one main obstacle (black) in the middle of the grid and two parked cars (red color) on the south end of the grid. Our task is to park all type of cars in between the two parked cars (red boxes). The black obstacle is marked as (0,0,0) on the grid and the red obstacles as (254,0,0).

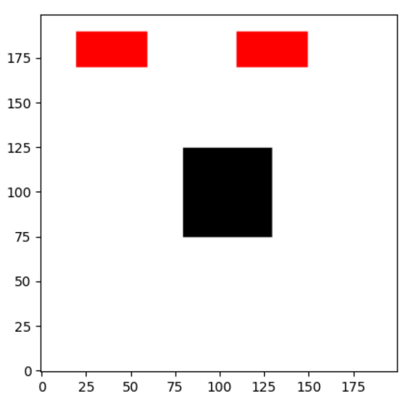
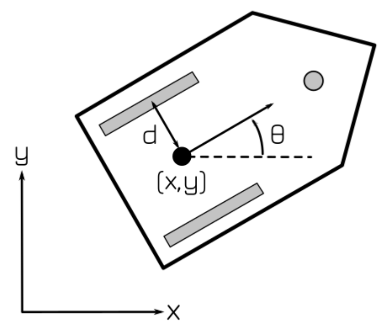


Fig. 1: The given environment

II. The Delivery Robot

First up is the delivery robot. Diwheel kinematics has been assumed for the delivery robot which means we only have to give left and right wheel velocities as input to the robot and it should accordingly determine the heading angle as well as the position co-ordinates from the the given velocities. As the diwheel robot allows us to take even zero radius turns by just varying the right and left wheel velocities, we don't need to exactly do parallel parking. The robot will just go to its desired position and adjust its heading angle. The kinematics equations for the diwheel robot can be seen in Fig. 2.



$$\dot{x} = \frac{r}{2}(u_l + u_r)\cos\theta$$
$$\dot{y} = \frac{r}{2}(u_l + u_r)\sin\theta$$
$$\dot{\theta} = \frac{r}{L}(u_r - u_l).$$

Fig. 2: Diwheel robot kinematics

A. Implementation

The starting position for the robot is marked as (20,25) and goal position as (180,85) which is the position between the two parked cars. The length and width of the car is taken as 25 and 12 respectively.

The method used to plan the path for diwheel robot is Hybrid A*. We can't use regular A* because the kinematics of the robot is also included in the system. The code for A* is almost same as Djikstra as done in last assignment only difference will be we have to define a heuristic in A* algorithm. In our case our heuristic is the Eucledian distance between the current position and desired goal position. And at the end we have to minimise the sum of cost and heuristic value. So, I will not discuss the basic code to run a A* algorithm but I will be discussing how the child nodes are being generated in this Hybrid A* algorithm where we have use the kinematic constraints for diwheel robot. Please see the below written pseudo-code followed for child node generation for the Hybrid A* method.

Possible wheel velocities are taken as -1 and 1 for both right and left wheel.

```
#Initialisations
R = 5                    #Radius of the wheel
L = 12                   # Distance between wheels
Ur = [-1,0,1]            # Possible wheel velocities
Children = [ ]           # list to collect all child nodes
#Possible velocity combinations
Inputs = [(-1,1), (-1,0), (-1,1), (0,1), (1,1), (1,0), (1,-1), (0,-1)]
dt = 1

for input in inputs:
        #heading direction and positions
        theta_new = current_node.theta + (R/L) * input[0] - input[1] * dt
        x_new = current_node.pos[0] + (R/2) * cos(theta_new) * dt
        y_new = current_node.pos[1] + (R/2) * sin(theta_new) * dt

        #get corner positions of the diwheel robot
        corners = outline(x_new, y_new, theta_new)

        for corner in corners:
        #collision check with grid boundary
                if corner[0]> sh-1 or corner[0] < 0 or corner[1] > sh-1 or corner[1] < 0:
                        continue
        #collision check with obstacles
                if grid[corner[0]][corner[1]][0] == 0 or grid[corner[0]][corner[1]][0] == 254:
                        continue
        #if no collision then make new child node
        node_position = (x_new, y_new)
        new_node = Node(current_node, node_position)
        #update heuristic (h), cost(g) and overall cost(f = h+g) of A*
        new_node.h = heuristic(new_node.pos, end_pos, theta_new)
        new_node.g = current_node.g + cost(new_node.pos, current_node.pos, theta_new)
        new_node.f = new_node.g + new_node.h
        new_node.theta = theta_new
        children.append(new_node)
```

Fig. 3: Pseudo-code for Hybrid A* child nodes generation - Diwheel Robot

B. Results

The plot of position of the center of the robot over the course of time can be seen in Fig. 4 and 5. Also, for visual results, please see the video submission posted alongside the assignment submission.

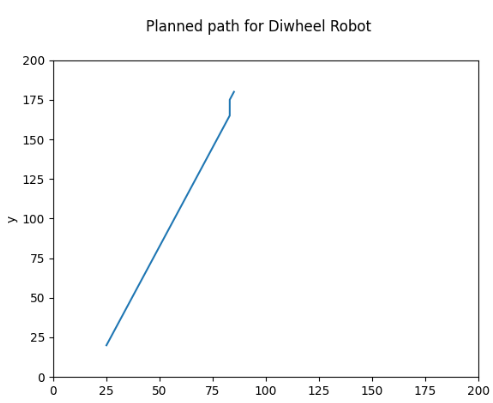


Fig. 4: Plot of the center of the axle

III. The Car

Next up is planning for a car. Standard ackerman steering kinematics constraints have been applied for the car to maneu-

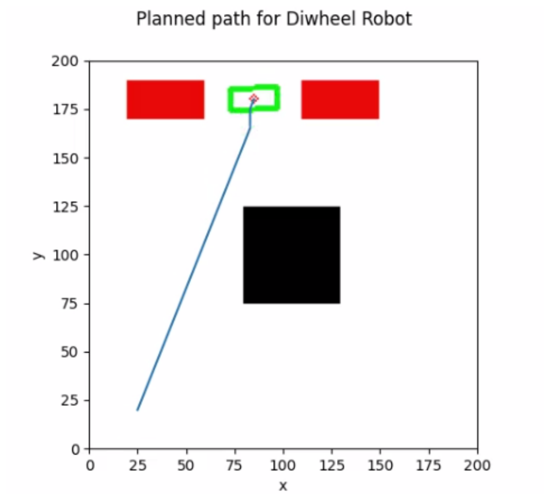
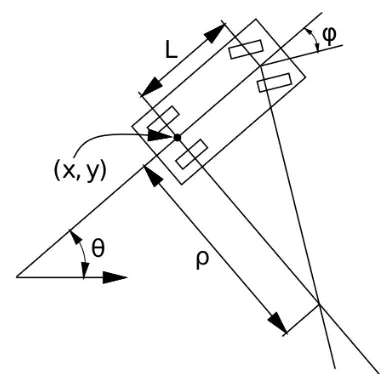


Fig. 5: Plot of the center of the axle

ver. In ackerman geometry we have to give both the vehicle velocity as well as heading angle as input and then only we can determine the final position co-ordinates of the center of the axle. Please note that in case of ackerman geometry zero radius turns are not possible and thus proper parallel parking has to be executed in the environment to park the car properly. The kinematics for the ackerman geometry in a car are depicted in the Fig. 6.



$$\dot{x} = u_s \cos\theta$$
$$\dot{y} = u_s \sin\theta$$
$$\dot{\theta} = \frac{u_s}{L}\tan u_\phi.$$

Fig. 6: Standard Ackerman steering car kinematics

A. Implementation

The method used to plan the path for car is Hybrid A*. We can't use regular A* because the kinematics of the robot is also included in the system. The code for A* is almost same as Djikstra as done in last assignment only difference will be we have to define a heuristic in A* algorithm. In our case our heuristic s the Eucledian distance between the current position and desired goal position. And at the end we have to minimise the sum of cost and heuristic value. So, I will not discuss the basic code to run a A* algorithm but I will be discussing how the child nodes are being generated in this Hybrid A* algorithm where we have use the kinematic constraints for car. Please see the below written pseudo-code followed for child node generation for the Hybrid A* method.

Implementing parallel parking was difficult as the normal Hybrid A* algorithm was taking way more time. So, in order to reduce time one solution was to use Reeds-Shepps curves but I used even simpler idea i.e. to divide the parallel parking path into 3 intermediate waypoints which were then fed into the planner to find the path between them.

Possible wheel velocities are taken as -1 and 1 and angles are taken as -30 to 30 in the increment of 15.

```
#Initialisations
L = 20                    # Wheel base
U = [-1,1]                # Possible wheel velocities
Children = [ ]            # list to collect all child nodes
#Possible velocity combinations
Inputs = [(-1,-30),(-1,-15), (-1,0),(-1,15),(-1,30),\
          (1,-30),(1,-15), (1,0),(1,15),(1,30)]
dt = 6
for input in inputs:
        #heading direction and positions
        beta = (input[0]/L)*np.tan(np.deg2rad(input[1]))*dt
        theta_new = (current_node.pos[2]) + beta
        x_new = round(current_node.pos[0] + input[0] * np.cos(theta_new)*dt)
        y_new = round(current_node.pos[1] + input[0] * np.sin(theta_new)*dt)

        #get boundary positions of the car
        car_pos = outline(grid, node_position, 12,25)
        #collision check
        check = True
        for a in range(0,7):
        #collision check with grid boundary
                if car_pos[a,0] > sh-1 or car_pos[a,0] < 0 or car_pos[a,1] > sh-1 or \
                   car_pos[a,1] < 0:
                      check = False
                      break
        #collision check with obstacles
                if grid[car_pos[a][1]][car_pos[a][0]][0] == 0 or \
                  grid[car_pos[a][1]][car_pos[a][0]][0] == 254:
                      check = False
                      break
        if check == False:
                continue

        #if no collision then make new child node
        node_position = (x_new, y_new, theta_new)
        new_node = Node(current_node, node_position)
        #update heuristic (h), cost(g) and overall cost(f = h+g) of A*
        new_node.h = heuristic(new_node.pos, end_pos, theta_new)
        new_node.g = current_node.g + \
                        cost(new_node.pos, current_node.pos, theta_new)
        new_node.f = new_node.g + new_node.h
        new_node.theta = theta_new
        children.append(new_node)
```

Fig. 7: Pseudo-code for Hybrid A* child nodes generation - Car

B. Results

The plot of position of the center of the rear axle of the car over the course of time can be seen in Fig. 8 and 9. Also, for visual results, please see the video submission posted alongside the assignment submission.

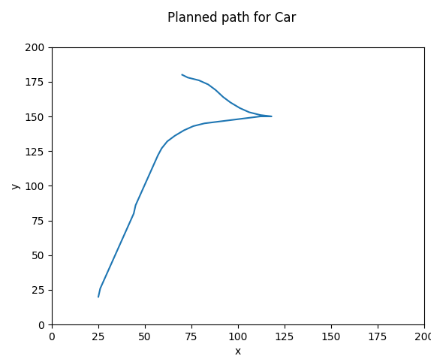


Fig. 8: Plot of the center of the axle

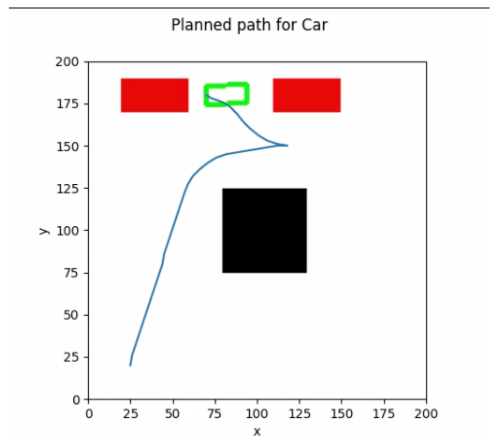
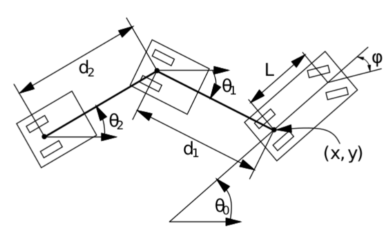


Fig. 9: Plot of the center of the axle

IV. The Truck

Next up is a planner for for parking of a Truck which is carrying a trailer in the back. Here, also we have to follow standard ackerman geometry for the truck in the front and only one more constraint has be applied for the heading direction of the trailer in the back. Similar to the car, we can't make zero radius turns with the truck and trailer. The kinematics constraints for the truck and trailer combination has been shown in the below Fig. 10.



$$\dot{x} = s\cos\theta_0$$
$$\dot{y} = s\sin\theta_0$$
$$\dot{\theta}_0 = \frac{s}{L}\tan\phi$$
$$\dot{\theta}_1 = \frac{s}{d_1}\sin(\theta_0 - \theta_1)$$

Fig. 10: Kinematic constraints for the Trailer

A. Implementation

The method used to plan the path for trailer is Hybrid A*. We can't use regular A* because the kinematics of the robot is also included in the system. The code for A* is almost same as Djikstra as done in last assignment only difference will be we have to define a heuristic in A* algorithm. In our case our heuristic s the Eucledian distance between the current position and desired goal position. And at the end we have to minimise the sum of cost and heuristic value. So, I will not discuss the basic code to run a A* algorithm but I will be discussing how the child nodes are being generated in this Hybrid A* algorithm where we have use the kinematic constraints for trailer. Please see the below written pseudo-code followed child node generation for the Hybrid A* method.

Implementing parallel parking was difficult as the normal Hybrid A* algorithm was taking way more time. So, in order to reduce time one solution was to use Reeds-Shepps curves but I used even simpler idea i.e. to divide the parallel parking path into 3 intermediate waypoints which were then fed into the planner to find the path between them.

Possible wheel velocities are taken as -1 and 1 and angles are taken as -30 to 30 in the increment of 15.

B. Results

The plot of position of the center of the rear axle of the truck over the course of time can be seen in Fig. 11 and 12. Also, for visual results, please see the video submission posted alongside the assignment submission.

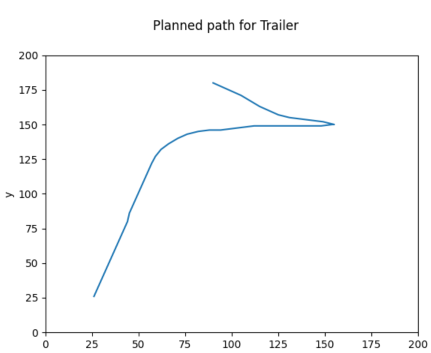


Fig. 11: Plot of the center of the rear axle of Trailer

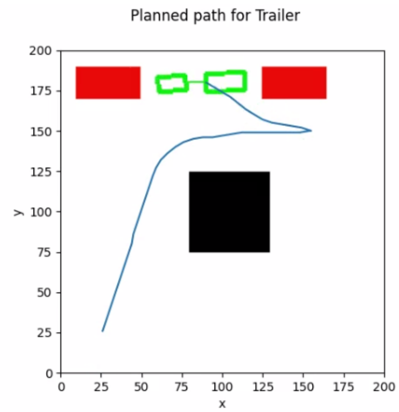


Fig. 12: Plot of the center of the rear axle of Trailer

```
#Initialisations
L = 20                  # Wheel base
d = 12                  # Distance of trailer
U = [-1,1]              # Possible wheel velocities
Children = [ ]          # list to collect all child nodes
#Possible velocity combinations
Inputs = [(-1,-30),(-1,-15), (-1,0),(-1,15),(-1,30),\
          (1,-30),(1,-15), (1,0),(1,15),(1,30)]
dt = 6
for input in inputs:
        #heading direction and positions
        beta = (input[0]/L)*np.tan(np.deg2rad(input[1]))*dt
        theta_new = (current_node.pos[2]) + beta
        x_new = round(current_node.pos[0] + input[0] * np.cos(theta_new)*dt)
        y_new = round(current_node.pos[1] + input[0] * np.sin(theta_new)*dt)
        #heading direction of trailer
        theta_trail_new = (current_node.trail_theta) + \
                    (input[0]/d)*np.sin(theta_new - current_node.trail_theta)*dt
        node_position = (x_new, y_new, theta_new)
        #get boundary positions of the truck and trailer
        pos = outline(grid, node_position)
        #collision check
        check = True
        for a in range(0,7):
        #collision check with grid boundary
                if pos[a,0] > sh-1 or pos[a,0] < 0 or pos[a,1] > sh-1 \
                or pos[a,1] < 0:
                        check = False
                        break
        #collision check with obstacles
                if grid[pos[a][1]][pos[a][0]][0] == 0 or \
                  grid[pos[a][1]][pos[a][0]][0] == 254:
                        check = False
                        break
        if check == False:
                continue
        #if no collision then make new child node
        new_node = Node(current_node, node_position)
        #update heuristic (h), cost(g) and overall cost(f = h+g) of A*
        new_node.h = heuristic(new_node.pos, end_pos, theta_new)
        new_node.g = current_node.g + \
                    cost(new_node.pos, current_node.pos, theta_new)
        new_node.f = new_node.g + new_node.h
        new_node.theta = theta_new
```

Fig. 13: Pseudo-code for Hybrid A* child nodes generation - Trailer