Apollo: Autonomous Parking, (Motion Planning and Control)

Amith Ramdas Achari, Peng Han, Yuehui Yu, Xiaoxu Sun

amithr3, penghan2, yuehuiy2, xiaoxus2

Department of Mechanical Engineering, UIUC

Goal

Goal:

Park a car from any position in a parking lot to a specific parking space.

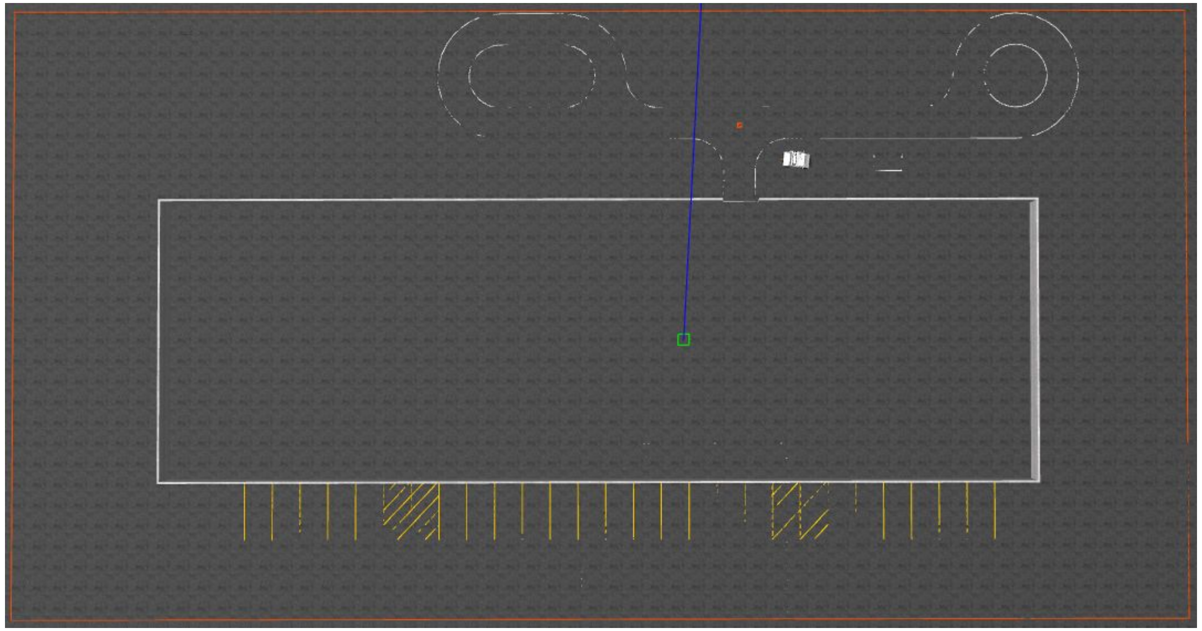
Result of simulation:

Control the car from the start position to the parking space in the highbay environment

Implement the results of simulation on GEM.

Environment (Highbay)

- We use a Gazebo environment to simulate the Highbay experiment site.
- Simulated the environment in ROS Noetic for both planning and control



Demo

https://youtu.be/q6XLgg58LSw

Controller(PID)

```python
def update(self, feedback_val):


    curr_time = time.time()
    del_time = curr_time - self.last_time

    curr_err= self.set_point- feedback_val
    del_err = curr_err - self.last_err
    err_dot = del_err/ del_time

    self.p_term = curr_err
    self.d_term = err_dot
    self.i_term += curr_err* del_time
```

```python
    self.last_time = curr_time
    self.last_err = curr_err

    out=  self.Kp * self.p_term+ self.Ki * self.i_term + self.Kd * self.d_term
    print(out)

    out_limit=self.out_limits

    if out>np.max(out_limit):
        out=np.max(out_limit)
    if out<np.min(out_limit):
        out=np.min(out_limit)

    #print(out)

    self.output = out
```

Motion Planning(A-star Algorithm with Euler Distance)

```python
def find_path(self, start, end):
    ##TODO
    frontier = []
    visited = {}
    cost = {}
    heapq.heappush(frontier,(0.0 + self.eulerdist(end,start),start))
    visited[start] = None
    cost[start] = 0
    path = deque()
    while len(frontier) > 0:
        curr_point = heapq.heappop(frontier)[1]
        # print(curr_point)
        if curr_point == end:
            break

        for neighbor in self.neighbors(curr_point):
            new_cost = cost[curr_point] + self.resolution + self.eulerdist(neighbor, end)
            if neighbor not in cost or new_cost < cost[neighbor]:
                heapq.heappush(frontier, (new_cost, neighbor))
                visited[neighbor] = curr_point  ## track path
                cost[neighbor] = cost[curr_point] + self.resolution

    curr_point = end
    while curr_point != None:
        path.append(curr_point)
        curr_point = visited[curr_point]
    path.reverse()

    print(path)
    return path
```

Motion Planning(Hybrid A-star Algorithm)

```
def find_path(self, start, end):
    steering_inputs = [-40,0,40]
    cost_steering_inputs= [0.1,0,0.1]

    speed_inputs = [-1,1]
    cost_speed_inputs = [1,0]

    start = (float(start[0]), float(start[1]), float(start[2]))
    end = (float(end[0]), float(end[1]), float(end[2]))
    # The above 2 are in discrete coordinates

    open_heap = [] # element of this list is like (cost,node_d)
    open_diction={} #  element of this is like node_d:(cost,node_c,(parent_d,parent_c))

    visited_diction={} #  element of this is like node_d:(cost,node_c,(parent_d,parent_c))

    obstacles = set(self.obstacle)
    cost_to_neighbour_from_start = 0

    hq.heappush(open_heap,(cost_to_neighbour_from_start + self.euc_dist(start, end),start))

    open_diction[start]=(cost_to_neighbour_from_start + self.euc_dist(start, end), start,(start,start))

    while len(open_heap)>0:
        chosen_d_node =  open_heap[0][1]
        chosen_node_total_cost=open_heap[0][0]
        chosen_c_node=open_diction[chosen_d_node][1]
        visited_diction[chosen_d_node]=open_diction[chosen_d_node]

        if self.euc_dist(chosen_d_node,end)<1:

            rev_final_path=[end] # reverse of final path
            node=chosen_d_node
            m=1
            while m==1:
                visited_diction
                open_node_contents=visited_diction[node] # (cost,node_c,(parent_d,parent_c))
                parent_of_node=open_node_contents[2][1]
```

Motion Planning(Hybrid A-star Algorithm)

```
                rev_final_path.append(parent_of_node)
                node=open_node_contents[2][0]
                if node==start:
                    rev_final_path.append(start)
                    break
            final_path=[]
            for p in rev_final_path:
                final_path.append(p)
            return final_path

        hq.heappop(open_heap)

        for i in range(0,3) :
            for j in range(0,2):

                delta=steering_inputs[i]
                velocity=speed_inputs[j]


                cost_to_neighbour_from_start =  chosen_node_total_cost-self.euc_dist(chosen_d_node, end)

                neighbour_x_cts = chosen_c_node[0] + (velocity * math.cos(math.radians(chosen_c_node[2])))
                neighbour_y_cts = chosen_c_node[1]  + (velocity * math.sin(math.radians(chosen_c_node[2])))
                neighbour_theta_cts = math.radians(chosen_c_node[2]) + (velocity * math.tan(math.radians(delta))/(float(self.vehicle_length)))

                neighbour_theta_cts=math.degrees(neighbour_theta_cts)

                neighbour_x_d = round(neighbour_x_cts)
                neighbour_y_d = round(neighbour_y_cts)
                neighbour_theta_d = round(neighbour_theta_cts)


                neighbour = ((neighbour_x_d,neighbour_y_d,neighbour_theta_d),(neighbour_x_cts,neighbour_y_cts,neighbour_theta_cts))

                if (((neighbour_x_d,neighbour_y_d) not in obstacles) and \
                     (neighbour_x_d >= self.min_x) and (neighbour_x_d <= self.max_x) and \
                     (neighbour_y_d >= self.min_y) and (neighbour_y_d <= self.max_y)) :
```

Motion Planning(Hybrid A-star Algorithm)

```
                neighbour = ((neighbour_x_d,neighbour_y_d,neighbour_theta_d),(neighbour_x_cts,neighbour_y_cts,neighbour_theta_cts))

                if (((neighbour_x_d,neighbour_y_d) not in obstacles) and \
                        (neighbour_x_d >= self.min_x) and (neighbour_x_d <= self.max_x) and \
                        (neighbour_y_d >= self.min_y) and (neighbour_y_d <= self.max_y)) :

                        heurestic = self.euc_dist((neighbour_x_d,neighbour_y_d,neighbour_theta_d),end)
                        cost_to_neighbour_from_start = abs(velocity)+ cost_to_neighbour_from_start +\
                                                                    cost_steering_inputs[i] + cost_speed_inputs[j]

                        total_cost = heurestic+cost_to_neighbour_from_start

                        skip=0

                        found_lower_cost_path_in_open=0

                        if neighbour[0] in open_diction:

                            if total_cost>open_diction[neighbour[0]][0]:
                                skip=1

                            elif neighbour[0] in visited_diction:

                                if total_cost>visited_diction[neighbour[0]][0]:
                                    found_lower_cost_path_in_open=1


                        if skip==0 and found_lower_cost_path_in_open==0:

                            hq.heappush(open_heap,(total_cost,neighbour[0]))
                            open_diction[neighbour[0]]=(total_cost,neighbour[1],(chosen_d_node,chosen_c_node))

print("Did not find the goal - it's unattainable.")
return []
```

Motion Planning(RRT Algorithm)

```python
def step(self, nnear, nrand, dmax = 5):
    d = self.distance(nnear, nrand)
    if d > dmax:
        u = dmax/d
        (xnear, ynear) = (self.x[nnear], self.y[nnear])
        (xrand, yrand) = (self.x[nrand], self.y[nrand])

        (px, py) = (xrand - xnear, yrand - ynear)
        theta = math.atan2(py, px)
        (x, y) = (int(xnear + dmax * math.cos(theta)),
                  int(ynear + dmax * math.sin(theta)))
        self.remove_node(nrand)
        if abs(x - self.goal[0]) < dmax and abs(y - self.goal[1]) < dmax:
            self.add_node(nrand, self.goal[0], self.goal[1])
            self.goalstate = nrand
            self.goalFlag = True
    else:
        self.add_node(nrand, x, y)
```

Motion Planning(RRT Algorithm)

```
t1 = time.time()
while (not graph.path_to_goal()):
    time.sleep(0.002)
    elapsed = time.time() - t1
    t1 = time.time()
    if elapsed > 10:
        raise

    if iteration % 10 == 0:
        X, Y , Parent = graph.bias(goal)
        pygame.draw.circle(map.map, map.grey, (X[-1], Y[-1]), map.nodeRad + 2, 0)
        pygame.draw.line(map.map, map.Blue, (X[-1], Y[-1]), (X[Parent[-1]],Y[Parent[-1]]),map.edgeThickness)
    else:
        X, Y, Parent = graph.expand()
        pygame.draw.circle(map.map, map.grey, (X[-1], Y[-1]), map.nodeRad + 2, 0)
        pygame.draw.line(map.map, map.Blue, (X[-1], Y[-1]), (X[Parent[-1]], Y[Parent[-1]]), map.edgeThickness)

    if iteration % 10 == 0:
        pygame.display.update()
    iteration += 1
```

Challenges

- We have successfully implemented A* algorithm to find a path to park the car. Hybrid A* star works most of the time, in some cases it gives weird trajectories
- We have implemented PID control to move the car, and we are still working on implementing MPC control (we might consider dropping it because of the complexity), instead just implement pure pursuit controller.

Next Step

- To make our path more smooth the next step is to transform the waypoints to a polynomial, so that we can control the lateral errors and steering angle error using controller.
- Now that we have Hybrid A star, we want to implement RRT to the GEM environment.
    - Hurdle: Adding dynamic constraints and setting obstacles