TRAJECTORY PLANNING FOR MOBILE ROBOTS

MEAM 520 LAB-03

Dinesh Jagai
Department of Computer Science
University of Pennsylvania
Philadelphia, PA
dinesh97@seas.upenn.edu

Thomas Wang
Department of Computer Science
University of Pennsylvania
Philadelphia, PA
ttwang@seas.upenn.edu

Monday July 20^{th} 2020

1 Overview

2 Concepts

1. I'm going to implement A* algorithm because it's seems intuitive and similar to Dijkstra algorithm. The strategy here is to find the shortest path to the goal while considering the Euclidean distance to the goal. I plan in the configuration space.
2. A* algorithm will check each time if the next step is blocked by an obstacle. If the step is blocked, the algorithm won't take that step, and will consider the others around it. To detect collision, we can check if the next step is in the middle of any obstacle by checking the coordinates, and invalid steps are those that have coordinates inside an obstacle's space.

   Trajectory generating strategy: The code for the trajectory is shown in listing-2 The bulk of the code is in the update function. The trajectory strategy is essentially to follow the way-points using linear paths - namely x & y lines. In order for the robot to face the consecutive way points, it's turns/circle's in place. Since we need a really tight circle, we assume that there are no limits on turning radius.
   In a dynamic situation my trajectory might not converge to the destination. Firstly, it takes a while to turn move to move to each consecutive waypoint. Also, in a vastly changing environment, it can get the robot can get confused when turning between the way-points.

3. **Pseudocode for the planner**
   Start from the starting location by creating a Node there. Each Node contains their coordinate, their parent node, and their g, h, f values. Initialize a heapq for the ordering of nodes to visit based on their f value. The node with smallest f would be the first node. First put start node into the heapq. While the queue isn't empty, we will explore the map. At each loop, we will pop the first thing off the heapq, which has the smallest f value. For this node, we will explore the 8 nodes around it. For each of the 8 nodes around it, we will first check if it's within bounds and if it's collided with an obstacle. Add all of these nodes in a list called children. For each of the children, we will calculate their g, h, f values and record the parent. Then, add these children nodes into the heapq. The loop goes on, until the current node is the goal node. When it arrives at the goal node, follow the parents all the way back to the start node, and create an array. Reverse this array, and return this path.

3 Coding Assignment

Listing 1: Astar.py

```python
import heapq
import numpy as np
import math
from OccupancyMap import OccupancyMap
import matplotlib.pyplot as plt


class Node:
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position[0] == other.position[0] and self.position[1] == other.position[1]

    def __repr__(self):
      return f"{self.position} - g: {self.g} h: {self.h} f: {self.f}"

    def __lt__(self, other):
      return self.f < other.f

    def __gt__(self, other):
      return self.f > other.f


def return_path(current_node):
    path = []
    current = current_node
    while current is not None:
        path.append(current.position)
        current = current.parent
    path = path[::-1] # Reverse path
    return np.array(path)


def Astar(filepath, start, goal, radius):

    occ_map = OccupancyMap(filepath, radius)
    bounds = occ_map.get_bounds()
    x_min = bounds[0]
    x_max = bounds[1]
    y_min = bounds[2]
    y_max = bounds[3]

    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, goal)
    end_node.g = end_node.h = end_node.f = 0

    open_list = []
    closed_list = []

    heapq.heapify(open_list)
    heapq.heappush(open_list, start_node)

    adjacent_squares = ((0, -0.5), (0, 0.5), (-0.5, 0), (0.5, 0), (-0.5, -0.5), (-0.5, 0.5), (0.5,
        -0.5), (0.5, 0.5),)
    print(occ_map.get_blocks())
    while len(open_list) > 0:

        current_node = heapq.heappop(open_list)
        closed_list.append(current_node)

        print(current_node.position)
        if current_node == end_node:
            return return_path(current_node)

        children = []

        for new_position in adjacent_squares:

            node_position = [current_node.position[0] + new_position[0], current_node.position[1] +
                new_position[1]]

            if node_position[0] < x_min or node_position[0] > x_max or node_position[1] < y_min or
                node_position[1] > y_max:
                continue

            blocked = False
            for block in occ_map.get_blocks():
                if node_position[0] >= block[0] and node_position[0] <= block[1] and
                    node_position[1] >= block[2] and node_position[1] <= block[3]:
                    blocked = True
                    break

            if blocked:
                continue

            new_node = Node(current_node, node_position)
            children.append(new_node)

        for child in children:
            if len([closed_child for closed_child in closed_list if closed_child == child]) > 0:
                continue

            # check if this is node is diagonal
            if (abs(current_node.position[0] - child.position[0]) + abs(current_node.position[1] -
                child.position[1]) > 0.5):
                child.g = current_node.g + np.sqrt(2) / 2
            else:
                child.g = current_node.g + 0.5

            child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] -
                end_node.position[1]) ** 2)
            child.f = child.g + child.h

            if len([open_node for open_node in open_list if child.position == open_node.position
                and child.g > open_node.g]) > 0:
                continue

            heapq.heappush(open_list, child)

    return None
```

The idea of the algorithm is here and it works for a lot of the maps and routes, but we did encounter a few bugs along the way. For example, since the step size was set as 0.5, if there are gaps that are less than 0.5, this algorithm wouldn't be able to find that path. Also, there isn't a 'max iteration' set, so in some cases it might take a long time to output None.

Listing 2: mobile_path.py

```
import numpy as np

class Path:
    """
    a class defining a path for a mobile robot

    ...

    Attributes
    ----------
    v :
        desired forward velocity to follow the path with


    Methods
    ----------
    gamma :
        function with arguments x, y that evaluates to 0 when robot is
        on the path
    gamma_jacobian :
        function with arguments x, y that returns and evaluates the
        jacobian of the gamma function at the point (x, y)
    """
    def __init__(self, v):
        self.v_des = v

    def circle(self, x_c, y_c, r):

        """
        set the classes attributes such that the path is a circle

        Parameters:
            x_c :
                desired x coordinate for circle center
            y_c :
                desired y coordinate for circle center
            r :
                desired radius of circle
        """
        def gamma(x,y):
            return (y-y_c)**2 + (x - x_c)**2 - r**2

        def gamma_jacobian(x,y):
            return np.array([2*(x - x_c), 2*(y-y_c)])

        self.gamma = gamma
        self.gamma_jacobian = gamma_jacobian


    def y_line(self, m, b):
        """
        set the classes attributes such that the path is a line in form y(x)

        Parameters:
            m :
                the slope of the line
            b :
                the y intercept of the line
        """
        def gamma(x,y):
            return y - m*x - b

        def gamma_jacobian(x,y):
            return np.array([-m, 1])

        self.gamma = gamma
        self.gamma_jacobian = gamma_jacobian


    def x_line(self, m, b):
        """
        set the classes attributes such that the path is a line in form x(y)

        Parameters:
            m :
                the slope of the line
            b :
                the x intercept of the line
        """
        def gamma(x,y):
            return x - m*y - b

        def gamma_jacobian(x,y):
            return np.array([1, -m])

        self.gamma = gamma
        self.gamma_jacobian = gamma_jacobian


class Trajectory:
    """
    a class defining a trajectory for a mobile robot, composed of many paths

    ...

    Attributes
    ----------
    waypoints :
        a numpy array of waypoints of size (N, 2)
    v :
        desired forward velocity to follow the trajectory
    l :
        tracking point distance from robot's axle
    goal_r :
        radius of the goal region for a waypoint (if robot is in this circle
        it is considered to be at the waypoint)
    goal_theta :
        window of acceptable angle error between robot and trajectory (if
        robot is outside this window it should try to get point itself back
        towards the path)
    seg_num :
        index of current segment that the robot is following. Segment n is
        defined as the path between waypoint n and waypoint n+1
    segment :
        an instance of the Path class defining the path to follow for the
        current segment

    Methods
    ----------
    update :
        using the most recent state information, evaluate which segment the
        robot should be on and construct the corresponding Path
    """
    def __init__(self, x0, waypoints, v, l, goal_r =0.1, goal_theta=np.pi/2):
        self.waypoints = waypoints
        self.v = v
        self.l = l
        self.goal_r = goal_r
        self.goal_theta = goal_theta

        # initialize segment
        self.seg_num = 0
        self.update(x0)

    def update(self, x):
        """
        using the most recent state information, evaluate which segment the
        robot should be on and construct the corresponding Path

        Parameters:
            x :
                current state information of the robot (x, y, theta)
        """


################Start your implementation here#########################
        self.segment = []
        path = Path(self.v)
        # check to see if a path exists (more segments than waypoints \implies no path exists)
        if ((self.waypoints.shape[0] - 1) <= self.seg_num):
            self.segment = None
            return
        next_waypoint = self.waypoints[self.seg_num + 1, :]
        # Find the l2 norm between the two consecutive waypoints
        euclid_dist_wps = np.sqrt(np.sum(np.power((x[0:2]- next_waypoint),2)))
        #  Update the segment number if the distance is close ebough to the goal
        if (euclid_dist_wps < self.goal_r):
            self.seg_num = self.seg_num + 1
        # If that's not the case, we need to adjust the path so that the robot can move towards the
            waypoint
        # We shall do this by considering the angles between the robot and the anglers between the
            next waypoint.
        # In general, we want the robot to Circle for when the angle error is too high such that
            it's facing the
        # next waypoint. Then the robot can move in a straight line to go to the next waypoint.
        else:
            current_waypoint = self.waypoints[self.seg_num, :]
            next_waypoint = self.waypoints[self.seg_num + 1, :]
            diff = next_waypoint - current_waypoint
            waypoints_angle = math.atan2(diff[1], diff[0])
            angle_error = abs(waypoints_angle - x[2])
            if (abs(angle_error % 2*np.pi) > self.goal_theta):
                path.circle(x[0], x[1], self.l)
                self.segment = path
            else:
                # x line
                if (diff[0] == 0):
                    m = 0
                    b = current_waypoint[0]
                    path.x_line(m, b)
                    self.segment = path
                else:
                    m = diff[1] / diff[0]
                    b = current_waypoint[1] - m * current_waypoint[0]
                    path.y_line(m, b)
                    self.segment = path
```

The trajectory code and strategy mentions the one talked about in section 2. That is, a trajectory that consists of linear paths between the point and rotates to face consecutive waypoints.

All other modified code is included in the appendix.

4 Simulation

5 Questions To Think About

1. The A* and RRT successfully found the path every time.

2. A* takes about 0.5 seconds, but RRT can take very different times. Going across map1, one time it took 3 seconds and a different time it took 49 seconds.

3. A* has the same path every time, but RRT has different paths each time.

4. The following maps are for A* For map 1 The time for planning for world 1 was 7.66 seconds and the trajectory is inserted below:

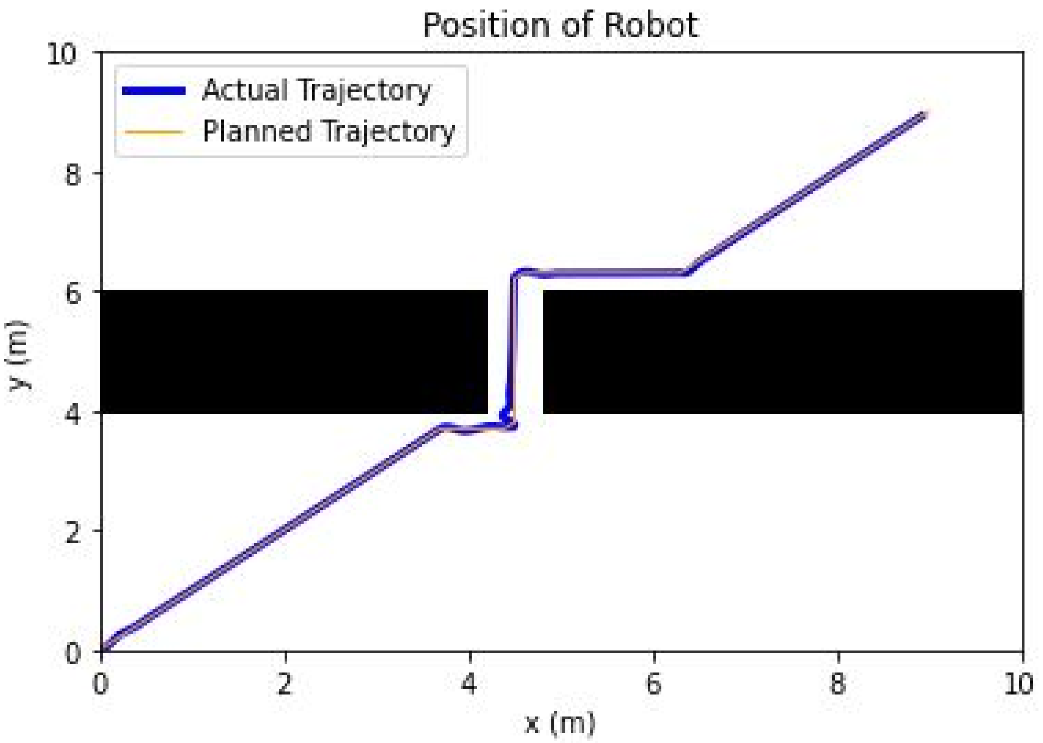

Figure 5.1.4.1: Figure Showing path planning for world 1

The time for planning for world 2 was 4.97 seconds and the trajectory is inserted below:

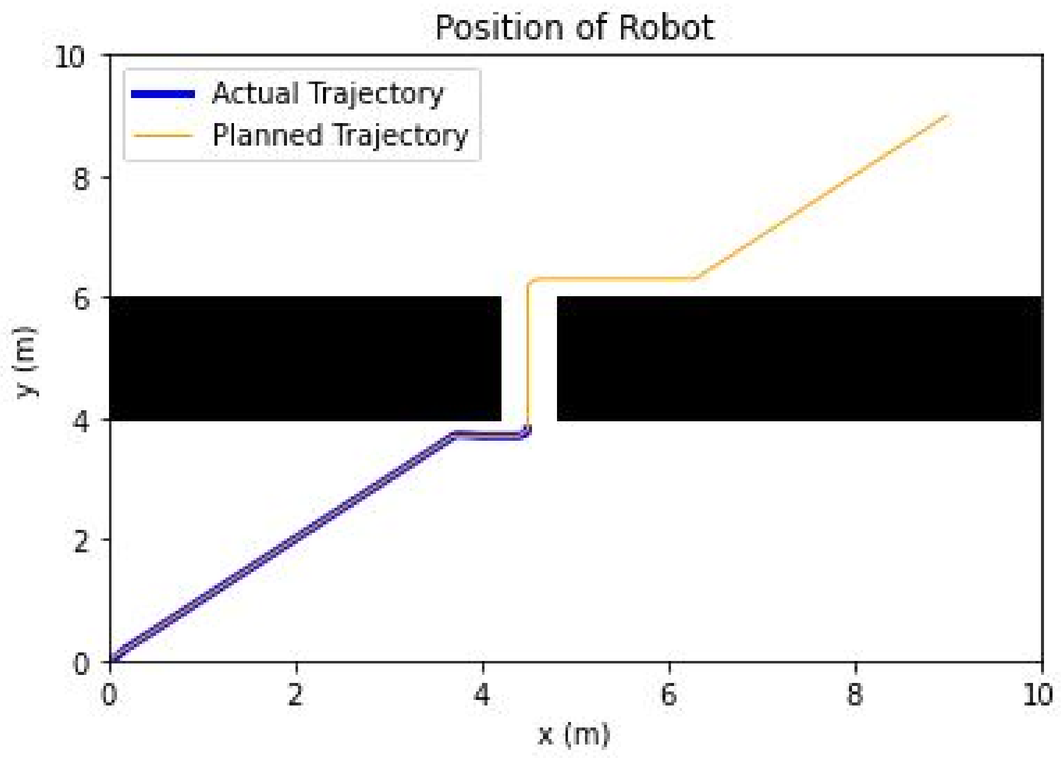

Figure 5.1.4.2: Figure Showing path planning for world 2

The time for planning for world 3 was 7.6 seconds and the trajectory is inserted below:

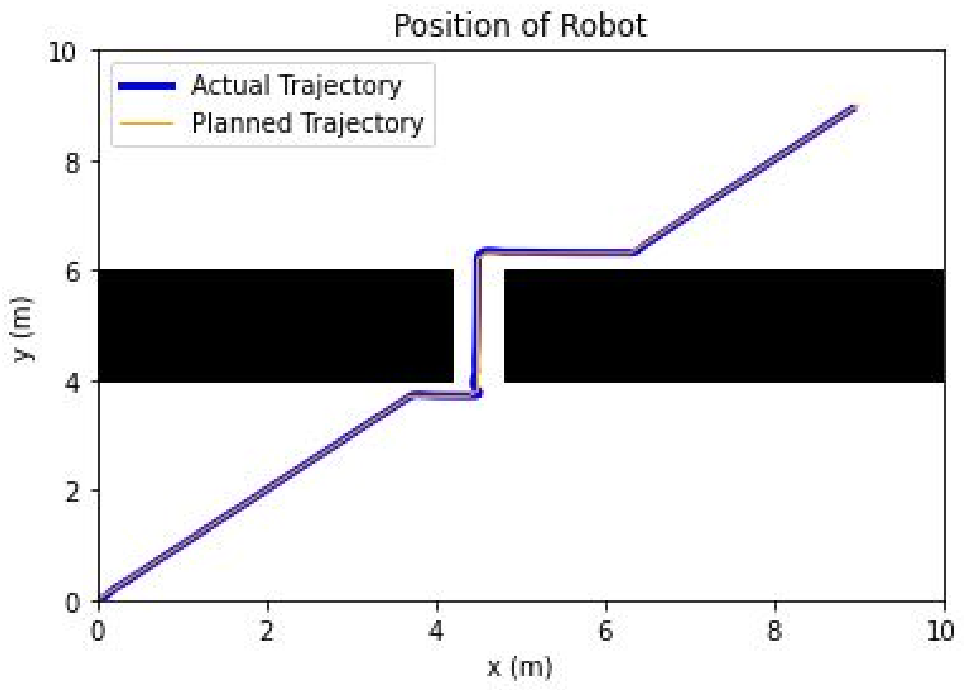

Figure 5.1.4.3: Figure Showing path planning for world 3

The time for planning for world 4 was 6.37 seconds and the trajectory is inserted below:

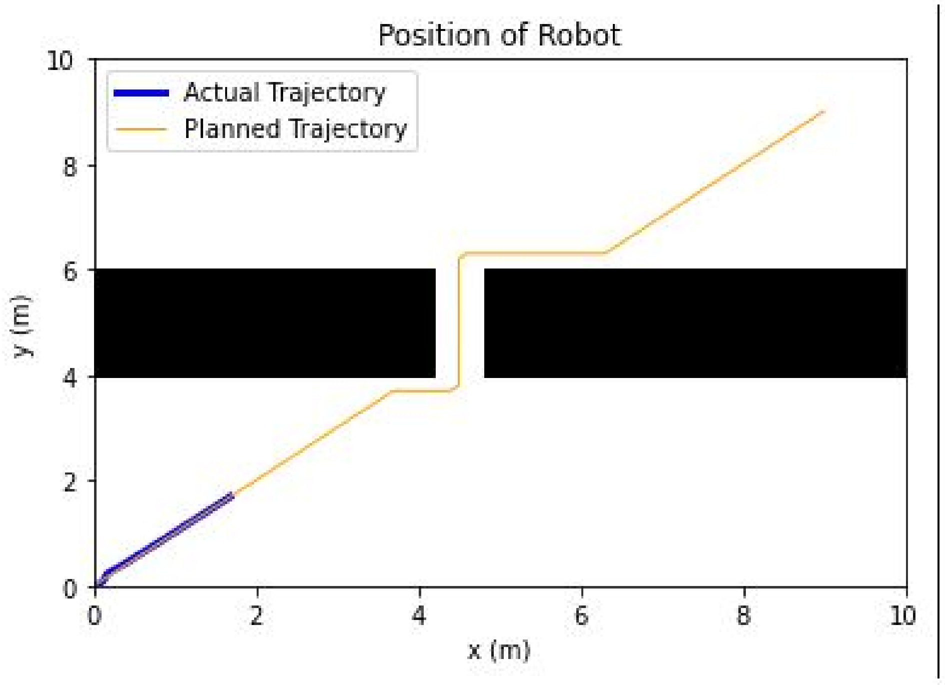

Figure 5.1.4.4: Figure Showing path planning for world 4

The time for planning for the empty world was 6.27 seconds and the trajectory is inserted below:

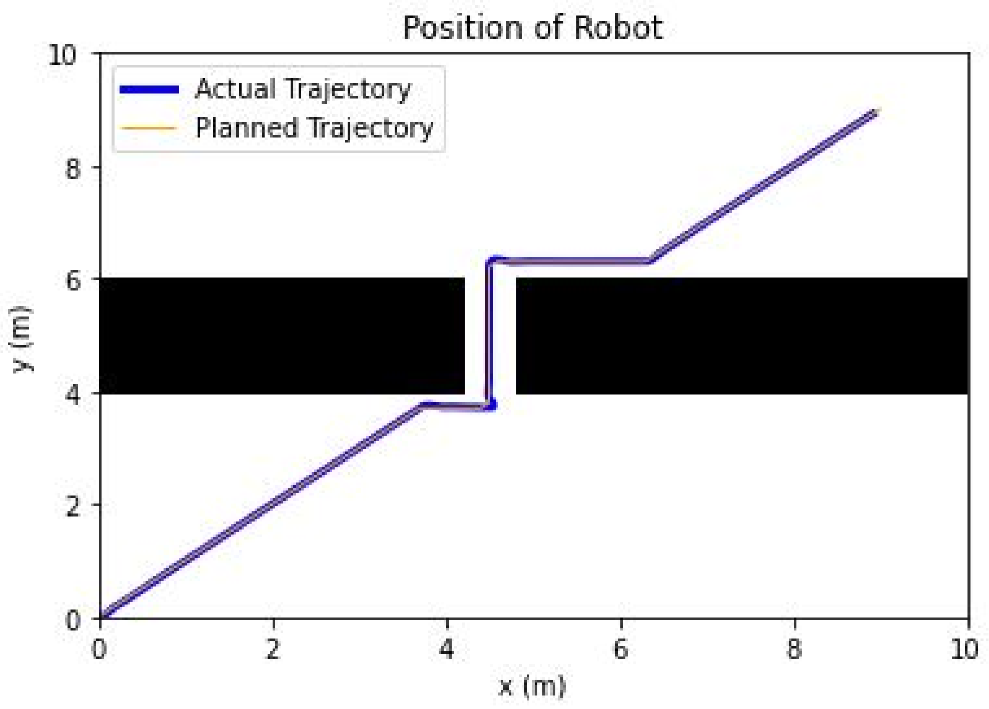

Figure 5.1.4.5: Figure Showing path planning for empty world

Clearly the A* planner works well where segments were calculated as straight line paths

5. Astar: A* would have an advantage because it's fast, and if there's a simple solution, A* can find it really easily and quickly. Some disadvantages would be how to find a good heuristic that incorporates all the joints. Also it would be hard to map out the 3D world and identify the obstacles and check for collision. It should also consider joint limits and the '8 connected' model might not work for all the joints. It's hard to work with a mix of R and P joints.

   RRT: The disadvantage would be it might take a very long time to come up with a solution. Also, it's possible that it can come up with a solution that's too complicated and has many small twists and turns. An advantage would be that if a joint moves in a weird/particular way, the algorithm can easily move accordingly. It can work with R and P joints and consider how each of them move separately.

6. The software only succeeds sometimes. There are a lot of times where the trajectory is planned out, but the robot doesn't follow it or fails in the middle of following.

7. The difficult plans are ones where there are small and complicated turns. Sometimes especially RRT gives trajectory that makes small turns and twitches, and in those situations the robot would fail.

8. The planned trajectory is not always the simulation result. Most of the times the robot follow the trajectory closely. However, there are times where the robot completely goes off the trajectory and fails. There are other times the robot skew away from the trajectory a little bit, but then goes back on the right track. There are times where the robot seems like it's following the trajectory well, but then stops in the middle. The error between these two varies dramatically.

9. In a physical setting, the planner would work similarly. The input is the map of the setting, with the obstacles clearly labeled. This might be difficult in a dynamic situation or an unclear new environment. If the robot doesn't know the environment parameters, then the algorithm wouldn't be effective. As for the trajectory planning, the robot will face real world problems like slippery floor and hitting itself while turning. It also

can't have dramatic turns that could cause it to topple. It should be careful to not go too fast as it might topple when changing directions.

10. If we had more time, we would also write our own RRT algorithm just to practice! For the A* algorithm, one problem we have is that the step size is 0.5 right now, meaning each turn we check the 8 steps that are 0.5 away from it $(0, -0.5), (0, 0.5), (-0.5, 0), (0.5, 0), (-0.5, -0.5), (-0.5, 0.5), (0.5, -0.5), (0.5, 0.5))$. If there is a really narrow pathway that is less than 0.5 wide, the algorithm wouldn't be able to find it. If we had more time, we would implement something that can adjust the step size according to the size of map and radius of the robot. I would also add a maximum iteration field to it, in order to prevent situations where it take way too long to output a 'None' answer. An alternative is to add warnings that pop up so that the user know how much the algorithm has explored. I think the algorithm could be sped up through vectorizing some of the functions and removing unnecessary loops. Also, a good thing to do would be to write code to visualize the results and the map each time.

6 Appendix

Listing 3: mobile_controller.py

```
import numpy as np

class Controller:
    """
    a class implementing path following controller for a differential
    drive robot

    ...

    Attributes
    ----------
    path :
        the desired path to be followed by the robot, an instantiation
        of the Path class
    robot_params :
        a dictionary of intrinsic robot parameters to be used by
        the controller
    frequency :
        the frequency that the controller loop runs at, measured in Hz

    Methods
    -------
    update(x, x_dot):
        calculate the wheel speeds needed to follow the desired path using
        the latest state information
    """

    def __init__(self, path, robot_params, frequency):
        '''
        initialize the Controller class

        Parameters:
            path :
                the desired path for the robot to follow
                an instance of the Path class
            robot_params:
                a dictionary of intrinsic robot parameters to be used by
                the controller
            frequency :
                the frequency that the controller loop runs at, measured in Hz
        '''

        # intrinsics
        self.path = path
        self.frequency = frequency # in Hz
        self.r = robot_params['wheel-radius']
        self.d = robot_params['axle-width']
        self.l = robot_params['tracking-point-axle-offset']
        self.integral_e2 = 0



    def set_path(self, path):
        self.path = path

    def update(self, x, x_dot):
        '''
        calculate the wheel speeds needed to follow the desired path using
        the latest state information

        Parameters:
            x :
                numpy array of length 3 representing state of robot
                array elements correspond to [x, y, theta]
            x_dot :
                numpy array of length 3 representing derivative of robot state
                array elements correspond to [x_dot, y_dot, theta_dot]

        Return:
            q_dot :
                numpy array of length 2 representing desired wheel speeds
                array elements correspond to [phi_dot_right, phi_dot_left]
            e1 :
                a measure of position error, a scalar
            e2 :
                a measure of velocity error, a scalar
        '''


############### Start your implementation here ##########################

        q_dot = np.array([0, 0])
        e1 = 0
        e2 = 0
        ### STUDENT CODE HERE ###
        kp = 1
        ki = 1
        xp = x[0] + self.l*np.cos(x[-1])
        yp = x[1] + self.l*np.sin(x[-1])
        e1 = self.path.gamma(xp, yp)
        xp_dot = np.array([[1, 0, -self.l*np.sin(x[-1])], [0, 1, self.l*np.cos(x[-1])]])@(x_dot)
        e2 = self.path.v_des - np.sqrt(xp_dot[0]**2 + xp_dot[1]**2)
        delta_t = 1/(self.frequency)
        self.integral_e2 = self.integral_e2 + delta_t*e2
        m_2 = np.array([[-kp*e1], [self.path.v_des + ki*self.integral_e2]])
        jacobian = np.array([[(self.r*np.cos(x[-1])/2 + self.l*self.r*np.sin(x[-1])/self.d),
            (self.r*np.cos(x[-1])/2 - self.l*self.r*np.sin(x[-1])/self.d)], [
                          (self.r*np.sin(x[-1])/2 - self.l*self.r*np.cos(x[-1])/self.d),
                              (self.r*np.sin(x[-1])/2 + self.l*self.r*np.cos(x[-1])/self.d)]])
        #m_1 = np.array([[self.path.gamma_jacobian(xp, yp)@jacobian], [(self.r/2)*np.array([1,
            1])]])
        m_1 = np.vstack((self.path.gamma_jacobian(xp, yp)@jacobian,((self.r/2)*np.array([1, 1]))))
#        print(m_1)
#        print(((self.r/2)*np.array([1, 1])).shape)
#        print((self.path.gamma_jacobian(xp, yp)@jacobian).shape)
        q_dot = np.linalg.inv(m_1)@m_2
        return q_dot, e1, e2
```