Principles of Robot Autonomy: Homework 1

[Chetan Reddy] [Narayanaswamy]

10/10/24

Other students worked with: None
Time spent on homework: 8 hours

Problem 1:

Part i

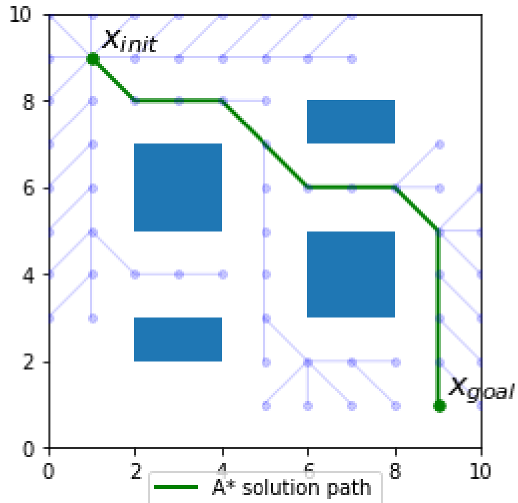

Figure 1: A* Algorithm (Simple Environment)

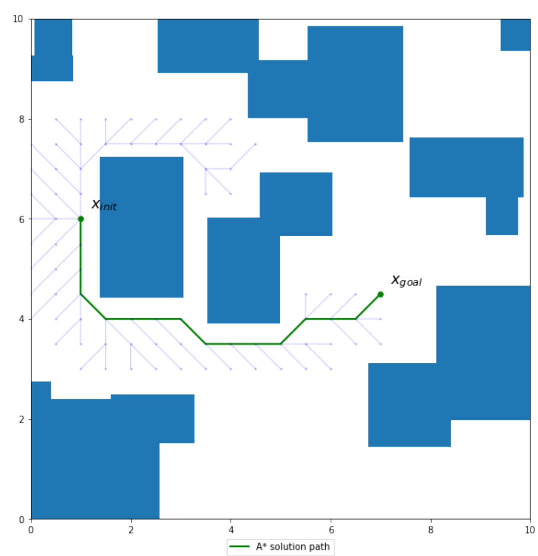

Figure 2: A* Algorithm (Cluttered Environment with num_obstacles= 20 and resolution = 0.5)

Part ii

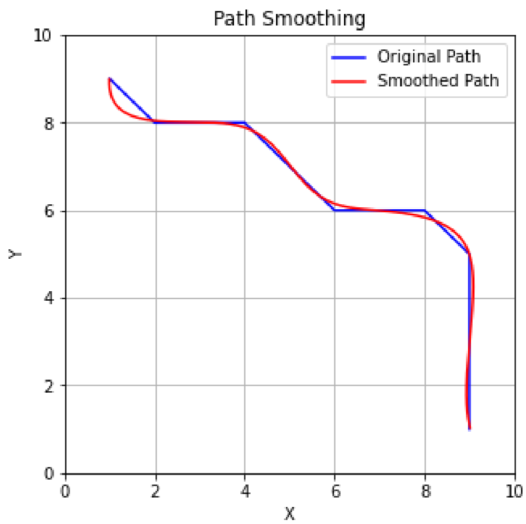

Figure 3: A* with Smoothening using Cubic Spline

Problem 2:

Part i

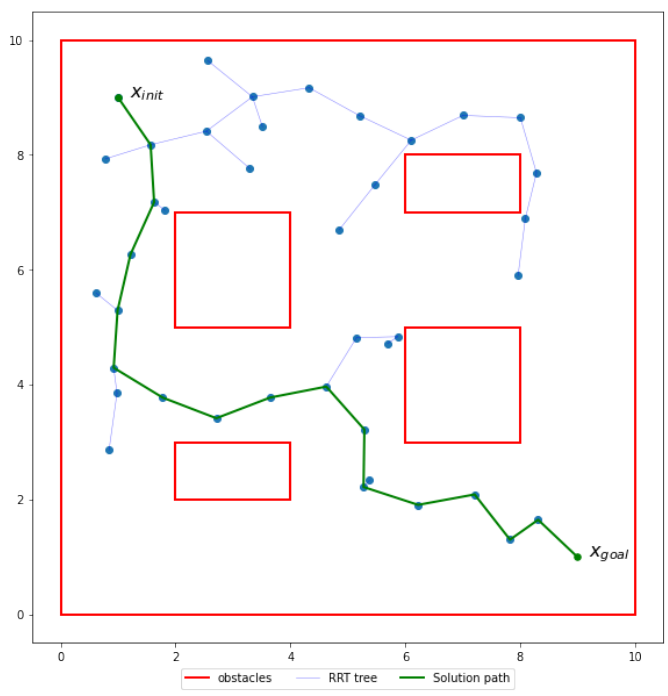

Figure 4: RRT

Part ii

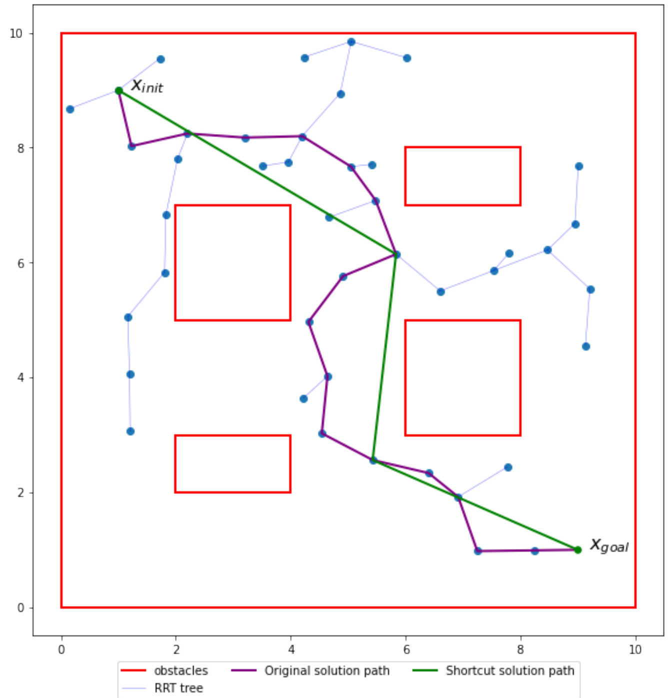

Figure 5: RRT with shortcutting

Problem 3:

Part i

By discretising time with some resolution $\Delta t = t_f/N$, we can obtain the following optimisation formulation.

$$\arg\min_{v_t, \omega_t} \sum_{k=0}^{N-1} \left(\alpha + v_t^2 + \omega_t^2\right) \Delta t$$

subject to **Equality Constraints**

$$\begin{aligned}
x_{k+1} - (x_k + v_k \cos(\theta_k)\Delta t) &= 0 \quad \forall\ k \in\ 0, 1, 2, ..., N-1 \\
y_{k+1} - (y_k + v_k \sin(\theta_k)\Delta t) &= 0 \quad \forall\ k \in\ 0, 1, 2, ..., N-1 \\
\theta_{k+1} - (\theta_k + \omega_k \Delta t) &= 0 \quad \forall\ k \in\ 0, 1, 2, ..., N-1 \\
x_0 = 0, y_0 = 0, \theta_0 - \pi/2 &= 0 \\
x_N - 5 = 0, y_N - 5 = 0, \theta_N - \pi/2 &= 0
\end{aligned}$$

and subject to **Inequality Constraints**

$$(x_k - x_{obs})^2 + (y_k - y_{obs})^2 - (r_{ego} + r_{obstacle})^2 \geq 0 \quad \forall\ k \in\ 0, 1, 2, ..., N$$

Part iii

We observe that the path becomes longer as alpha is reduced as it is a measure of the penalty given for increasing the time.

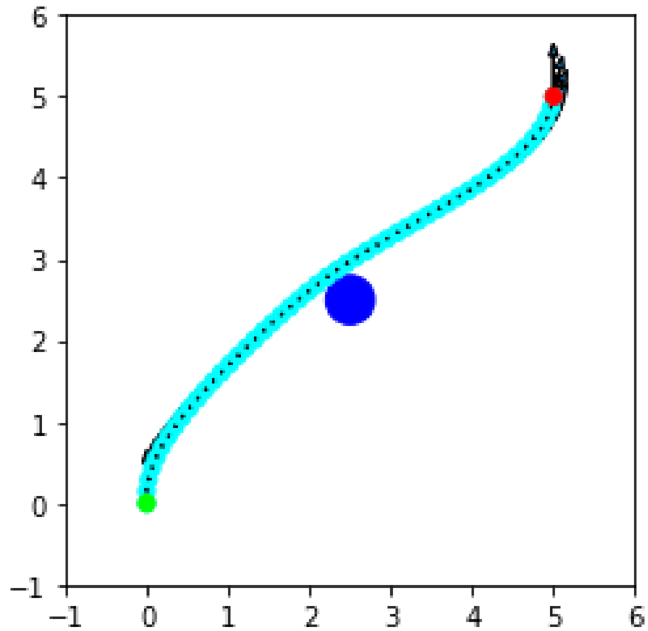

Figure 6: Path after Trajectory Optimisation

Problem 4:

We observe that the angle theta is heading towards the goal.

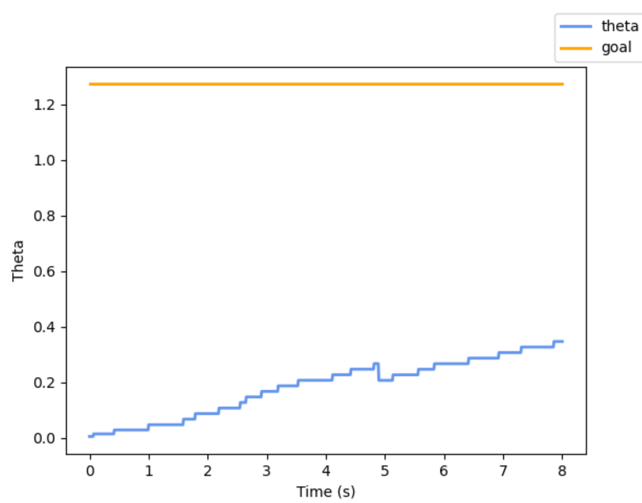

Figure 7: Plot of P Controller Response (Theta)

Appendix A: Code Submission

The code snippets of functions that were required to be filled have been included. In particular, the code written is located between "Code starts here" and "Code ends here"

Problem 1 (i)

```

    def is_free(self, x):

        # ########## Code starts here ##########
        # Checking if the state is within the map bounds
        if (x[0]>=self.statespace_lo[0] and
            x[1]>= self.statespace_lo[1] and
            x[0]<=self.statespace_hi[0] and
            x[1]<=self.statespace_hi[1]):
            pass
        else:
            return False

        # Checking Collision with the obstacles using
        # the method inside the DetOccupancyGrid2D object
        if self.occupancy.is_free(x):
            return True
        else:
            return False
        # ########## Code ends here ##########

    def distance(self, x1, x2):

        # ########## Code starts here ##########
        return (((np.array(x1)-np.array(x2))**2).sum())**0.5
        # ########## Code ends here ##########


    def get_neighbors(self, x):
        # ########## Code starts here ##########
        neighbors = []

        # There are 8 possible directions that are captured by i and j

        for i in [-self.resolution,0,self.resolution]:
            for j in [-self.resolution,0,self.resolution]:
                if i==j==0:
                    continue
                neighbor = (x[0]+i,x[1]+j)
                if self.is_free(neighbor):
                    neighbors.append(self.snap_to_grid(neighbor))
        # ########## Code ends here ##########
        return neighbors

    def solve(self):
```

```python
        # ########## Code starts here ##########
        while self.open_set:
            x_current = self.find_best_est_cost_through()

            if x_current == self.x_goal:
            # Assuming that the points are already snapped to grid
                self.path = self.reconstruct_path()
                return True

            # Move the current node from open_set to closed_set
            self.open_set.remove(x_current)
            self.closed_set.add(x_current)

            for neighbor in self.get_neighbors(x_current):
                if neighbor in self.closed_set:
                    continue
                else:
                    tentative_cost_to_arrive = self.cost_to_arrive[x_current] +
                    self.distance(x_current, neighbor)

                    if neighbor not in self.open_set:
                        self.open_set.add(neighbor)

                    elif (tentative_cost_to_arrive >=
                        self.cost_to_arrive.get(neighbor, float('inf'))):
                        continue

                    self.came_from[neighbor] = x_current
                    self.cost_to_arrive[neighbor] = tentative_cost_to_arrive
                    self.est_cost_through[neighbor] = tentative_cost_to_arrive +
                                                self.distance(neighbor,self.x_goal)

        # Is the open set is empty and x_goal has not been reached,
        # the function returns False
        else:
            return False
        # ########## Code ends here ##########
```

Problem 1 (ii)

```python
def compute_smooth_plan(path, v_desired=0.15, spline_alpha=0.05)
    -> TrajectoryPlan:
    # Ensure path is a numpy array
    path = np.asarray(astar.path)

    ##### YOUR CODE STARTS HERE #####
    n_timesteps = len(astar.path)
    ts = np.zeros(n_timesteps)

    distances = ((path[1:]-path[:-1])**2).sum(axis=1) # Shape is n_timesteps-1
    durations = distances/v_desired
```

```

    i = 1
    for duration in durations:
        ts[i] = ts[i-1]+duration
        i+=1

    path_x_spline = scipy.interpolate.splrep(ts, path[:, 0], k=3, s=spline_alpha)
    path_y_spline = scipy.interpolate.splrep(ts, path[:, 1], k=3, s=spline_alpha)
    ###### YOUR CODE END HERE ######

    return TrajectoryPlan(
        path=path,
        path_x_spline=path_x_spline,
        path_y_spline=path_y_spline,
        duration=ts[-1],
    )
```

Problem 2 (i)

```

    def solve(self, eps, max_iters=1000, goal_bias=0.05, shortcut=False):


        state_dim = len(self.x_init)

        # V stores the states that have been added to the RRT
        # (pre-allocated at its maximum size
        # since numpy doesn't play that well with appending/extending)
        V = np.zeros((max_iters + 1, state_dim))
        V[0,:] = self.x_init
        # RRT is rooted at self.x_init
        n = 1
        # the current size of the RRT (states accessible as V[range(n),:])

        # P stores the parent of each state in the RRT. P[0] = -1 since the root
        # has no parent, P[1] = 0 since the parent of the first additional state
        # added to the RRT must have been extended from the root, in general
        # 0 <= P[i] < i for all i < n P = -np.ones(max_iters + 1, dtype=int)


        success = False


        ########## Code starts here ##########
        for k in range(max_iters):
            z = np.random.uniform(0,1)
            if z<goal_bias:
                x_rand = self.x_goal # x_rand is shape (2,)
            else:
                x_rand = np.random.uniform(self.statespace_lo,self.statespace_hi)

```

```
            x_near_idx = self.find_nearest(V[:n],x_rand)
            x_near = V[x_near_idx] # Shape (2,)

            x_new = self.steer_towards(x_near,x_rand,eps)

            if self.is_free_motion(self.obstacles,x_near,x_new):
                V[n,:] = x_new
                P[n] = x_near_idx
                n += 1

                if np.linalg.norm(x_new-self.x_goal)<=eps:
                    V[n,:] = self.x_goal
                    P[n] = n-1
                    success = True
                    break


        if success:
            p_idx = n
            path = []

            while True: # We backtrack until we hit the root
                path.append(V[p_idx,:])

                if p_idx==0:
                    break
                p_idx = P[p_idx]
                # The present index is set to the parent of the present idx

            self.path = np.array(path[::-1])



        ########## Code ends here ##########

        ### Code to Plot ##
        return success


class GeometricRRT(RRT):

    def find_nearest(self, V, x):
        # Consult function specification in parent (RRT) class.
        ########## Code starts here ##########
        # Hint: This should take 1-3 line.

        # Shape of V = (n,2) and x = (2,)
        distances = ((V-x)**2).sum(axis=1)**0.5
        #Array of euclidean distance of every node from x


        return np.argmin(distances)
```

```python
        # Returning the integer index of the node in the tree

        ########## Code ends here ##########


    def steer_towards(self, x1, x2, eps):
        # Consult function specification in parent (RRT) class.
        ########## Code starts here ##########
        # Hint: This should take 1-4 line.
        direction_vector = x2-x1 # Shape = (2,)
        distance = (direction_vector**2).sum()**0.5 # Scalar
        if distance<=eps:
            return x2
        else:
            normalised_direction_vector = direction_vector/distance
            x_new = x1 + normalised_direction_vector*eps
            return x_new #Shape = (2,)

        ########## Code ends here ##########
```

Problem 2 (ii)

```python
    def shortcut_path(self):
        ########## Code starts here ##########
        if self.path is None:
            return "No Path Exists"

        ans = [self.path[0]]

        for i in range(1,self.path.shape[0]-1):
            parent = ans[-1]
            node = self.path[i]
            child = self.path[i+1]
            if self.is_free_motion(self.obstacles,parent,child):
                pass
            else:
                parent = node
                ans.append(node)

        ans.append(self.path[-1])
        self.path = np.array(ans)
        ########## Code ends here ##########
```

Problem 3 (ii)

```python
############################## Code starts here ##############################
s_0=np.array([EGO_START_POS[0],EGO_START_POS[1],np.pi/2])  # Initial state.
s_f=np.array([EGO_FINAL_GOAL_POS[0],EGO_FINAL_GOAL_POS[1],np.pi/2])# Final state.
############################## Code ends here ##############################

```

```
def optimize_trajectory(
    time_weight: float = 1.0,
    verbose: bool = True
):
    """Computes the optimal trajectory as a function of `time_weight`.

    Args:
        time_weight: \alpha in the HW writeup.

    Returns:
        t_f_opt: Final time, a scalar.
        s_opt: States, an array of shape (N + 1, s_dim).
        u_opt: Controls, an array of shape (N, u_dim).
    """

    def cost(z):
        ############################## Code starts here #####################
        # TODO: Define a cost function here

        t_f, s, u = unpack_decision_variables(z)
        return time_weight*t_f + (u**2).sum()*t_f/N
        ############################## Code ends here #######################

    # Initialize the trajectory with a straight line
    z_guess = pack_decision_variables(
        20, s_0 + np.linspace(0, 1, N + 1)[:, np.newaxis] * (s_f - s_0),
        np.ones(N * u_dim))

    # Minimum and Maximum bounds on states and controls
    # This is because we would want to include safety limits
    # for omega (steering) and velocity (speed limit)
    bounds = Bounds(
        pack_decision_variables(
            0., -np.inf * np.ones((N + 1, s_dim)),
            np.array([0.01, -om_max]) * np.ones((N, u_dim))),
        pack_decision_variables(
            np.inf, np.inf * np.ones((N + 1, s_dim)),
            np.array([v_max, om_max]) * np.ones((N, u_dim)))
    )

    # Define the equality constraints
    def eq_constraints(z):
        t_f, s, u = unpack_decision_variables(z)
        dt = t_f / N
        constraint_list = []
        for i in range(N):
            V, om = u[i]
            x, y, th = s[i]
            ############################## Code starts here #################
            # TODO: Append to `constraint_list` with dynanics constraints
            x_next,y_next,th_next = s[i+1]
            constraint_list.append(x_next - x - V*np.cos(th)*dt)
```

```python
            constraint_list.append(y_next - y - V*np.sin(th)*dt)
            constraint_list.append(th_next - th - om*dt)
            ############################## Code ends here ###################

        ############################## Code starts here ###################
        # TODO:
        # Append to `constraint_list` with initial and final state constraints
        xN,yN,thetaN = s[N]
        constraint_list.append(xN-s_f[0])
        constraint_list.append(yN-s_f[1])
        constraint_list.append(thetaN-s_f[2])

        x0,y0,theta0 = s[0]
        constraint_list.append(x0 - s_0[0])
        constraint_list.append(y0 - s_0[1])
        constraint_list.append(theta0-s_0[2])


        ############################## Code ends here ########################
        return np.array(constraint_list)

    # Define the inequality constraints
    def ineq_constraints(z):
        t_f, s, u = unpack_decision_variables(z)
        dt = t_f / N
        constraint_list = []
        for i in range(N):
            V, om = u[i]
            x, y, th = s[i]
            ############################## Code starts here ###################
            # TODO:
            # Append to `constraint_list` with collision avoidance constraint
            constraint_list.append((x-OBSTACLE_POS[0])**2 +
                                   (y-OBSTACLE_POS[1])**2 -
                                   (EGO_RADIUS+OBS_RADIUS)**2)
            ############################## Code ends here ####################
        return np.array(constraint_list)

    result = minimize(cost,
                      z_guess,
                      bounds=bounds,
                      constraints=[{
                          'type': 'eq',
                          'fun': eq_constraints
                      },
                      {
                          'type': 'ineq',
                          'fun': ineq_constraints
                      }],
                      options={'maxiter': 300, 'disp': True})
    if verbose:
        print(result)
```

```

    return unpack_decision_variables(result.x)
```

Problem 4

```
#!/usr/bin/env python3

import numpy as np
import rclpy
from asl_tb3_lib.control import BaseHeadingController
from asl_tb3_lib.math_utils import wrap_angle
from asl_tb3_msgs.msg import TurtleBotControl, TurtleBotState

class HeadingController(BaseHeadingController):
    def __init__(self):
        super().__init__()

        # Define the proportional control gain
        self.kp = 2.0

    def compute_control_with_goal(self,
                                  state: TurtleBotState,
                                  goal: TurtleBotState) -> TurtleBotControl:
        """
        Compute the control command to reach the desired heading.

        :param state: Current state of the TurtleBot
        :param goal: Desired state of the TurtleBot
        :return: Control command for the TurtleBot
        """
        # Calculate the heading error (    [     ,   ])
        err = wrap_angle(goal.theta - state.theta)

        # Compute the angular velocity using the proportional control formula
        omega = self.kp * err
        print(omega)
        # Create a new TurtleBotControl message
        control_msg = TurtleBotControl()
        control_msg.omega = omega

        return control_msg

if __name__ == "__main__":
    rclpy.init()

    # Create an instance of the HeadingController
    heading_controller = HeadingController()

    # Spin the node to keep it running and listening for messages
    rclpy.spin(heading_controller)

```

```
    # Shut down the ROS2 system
    rclpy.shutdown()
```