Principles of Robot Autonomy: Homework 1

Wei-Lin Pai
10/08/24

Other students worked with:
Time spent on homework:

Problem 1:

1. Simple Environment Plot

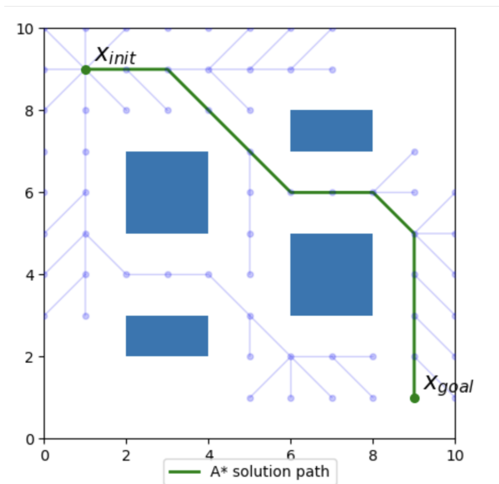

Figure 1: A*

2. Smooth Trajectory

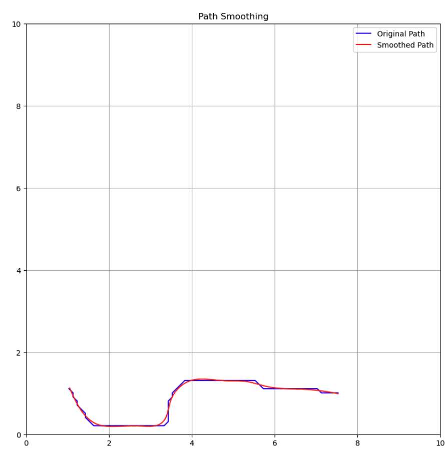

Figure 2: A* smooth

Problem 2:

1. Geometric Plotting

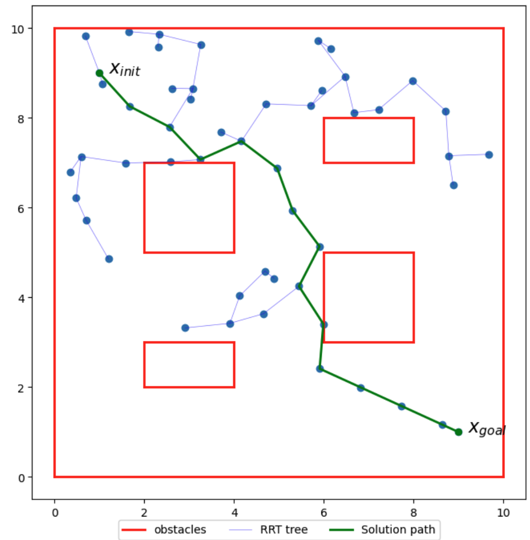

Figure 3: RRT

2. Adding Shortcutting

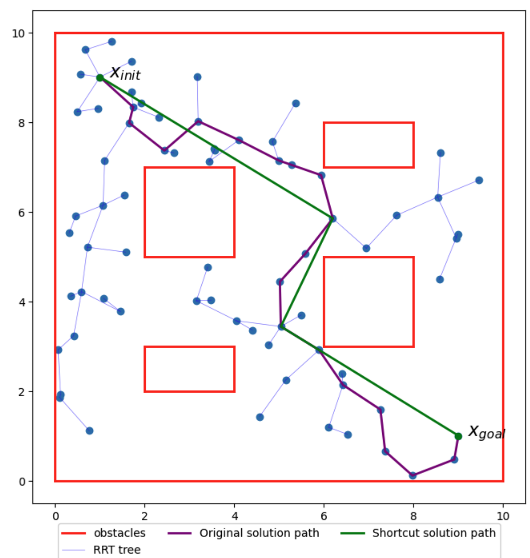

Figure 4: RRT adding shortcutting

Problem 3:

1. Turn Intergral J into: $\sum_{i=1}^{N} \alpha + v(t_i)^2 + w(t_i)^2 dt$, where N = Number of time discretization nodes

   Turn kinematic model constraint into:
   $x(t_{i+1}) = x(t_i) + v(t_i) * cos(\theta(t_i)) * dt$
   $y(t_{i+1}) = y(t_i) + v(t_i) * sin(\theta(t_i)) * dt$
   $\theta(t_{i+1}) = \theta(t_i) + w(t_i) * dt$

   Initial Condition:
   $x(0) = 0, y(0) = 0, \theta(0) = \pi/2$
   $x(t_N) = 0, y(t_N) = 0, \theta(t_N) = \pi/2$

   Collision Avoidance:
   $\sqrt{(x(t) - x_{obstacle})^2 + (y(t) - y_{obstacle})^2} - (r_{ego} + r_{obstacle}) >= 0$

2. Open-loop trajectory ($\alpha = 1$)

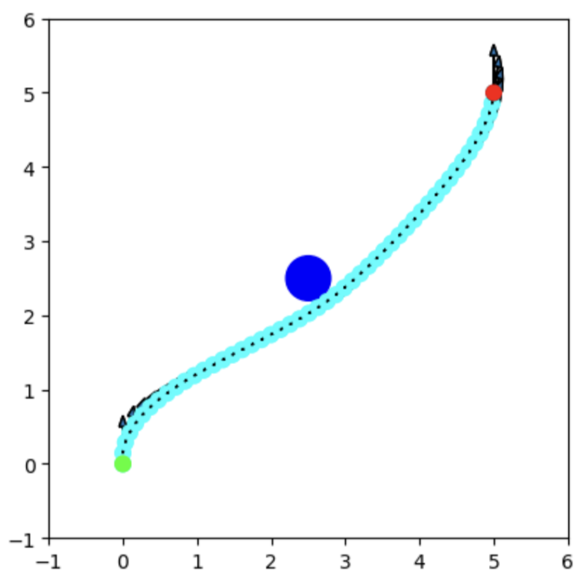

Figure 5: Open Loop Trajectory

3. Increasing α to 20, the energy and time comsumption increase, we can observed from plot that the trajectory planned in the way change less direction, which infers lower control energy required.

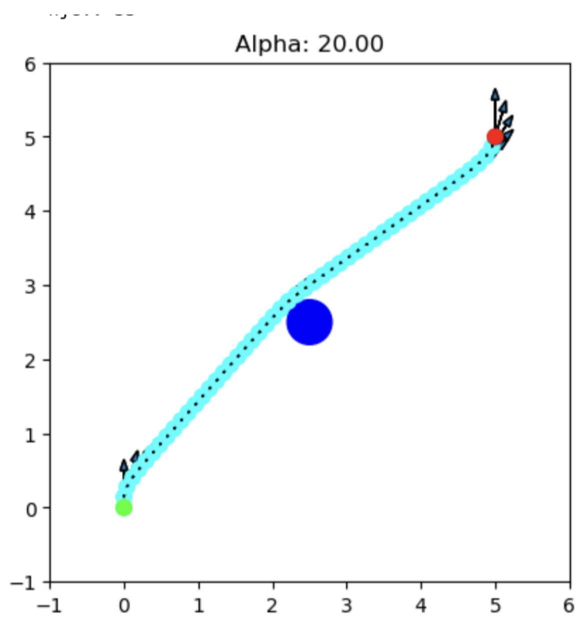

Figure 6: Trajectory with $\alpha = 20$

When increasing α to 100, we can found out that the trajectory gradually deviates from the restriction of dynamic feasibility. This is because α stands for too high portion of minimized function, which decrease the importance of dynamic feasibility.

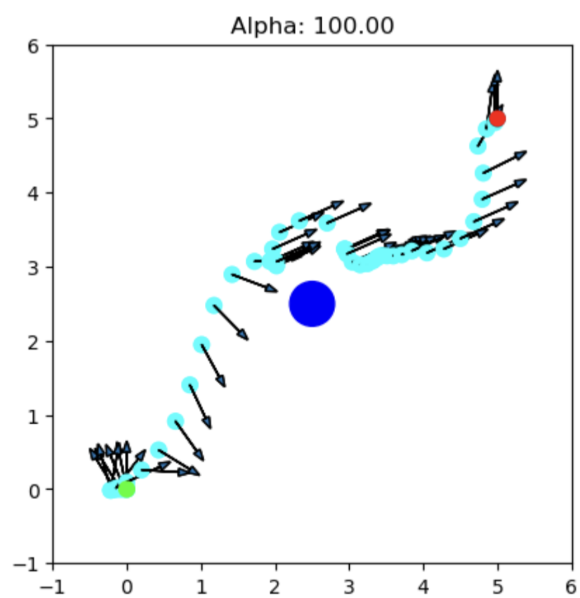

Figure 7: Trajectory with $\alpha = 100$

Problem 4:

1. p3 output

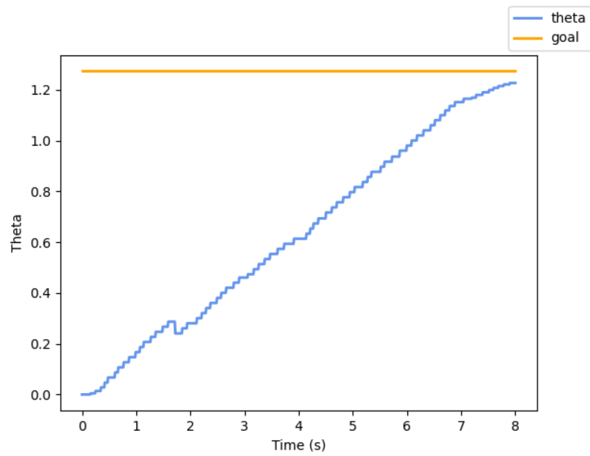

Figure 8: P3 plot

Appendix A: Code Submission

P1 astar.py

```
class AStar(object):
    def is_free(self, x):
    ########## Code starts here ##########
    return self.occupancy.is_free(x)
    ########## Code ends here ##########

    def distance(self, x1, x2):
    ########## Code starts here ##########
    return np.linalg.norm(np.array(x1)-np.array(x2))
    ########## Code ends here ##########

    def get_neighbors(self, x):
    neighbors = []
    ########## Code starts here ##########
    if(self.is_free(self.snap_to_grid((x[0]+self.resolution,x[1])))):
        neighbors.append(self.snap_to_grid((x[0]+self.resolution,x[1])))
    if(self.is_free(self.snap_to_grid((x[0]-self.resolution,x[1])))):
        neighbors.append(self.snap_to_grid((x[0]-self.resolution,x[1])))
    if(self.is_free(self.snap_to_grid((x[0],x[1]+self.resolution)))):
        neighbors.append(self.snap_to_grid((x[0],x[1]+self.resolution)))
    if(self.is_free(self.snap_to_grid((x[0],x[1]-self.resolution)))):
        neighbors.append(self.snap_to_grid((x[0],x[1]-self.resolution)))
    if(self.is_free(self.snap_to_grid((x[0]+self.resolution/(2**(1/2)),
    x[1]+self.resolution/(2**(1/2)))))):
        neighbors.append(self.snap_to_grid((x[0]+self.resolution/(2**(1/2)),
        x[1]+self.resolution/(2**(1/2)))))
    if(self.is_free(self.snap_to_grid((x[0]+self.resolution/(2**(1/2)),
    x[1]-self.resolution/(2**(1/2)))))):
        neighbors.append(self.snap_to_grid((x[0]+self.resolution/(2**(1/2)),
        x[1]-self.resolution/(2**(1/2)))))
    if(self.is_free(self.snap_to_grid((x[0]-self.resolution/(2**(1/2)),
    x[1]+self.resolution/(2**(1/2)))))):
        neighbors.append(self.snap_to_grid((x[0]-self.resolution/(2**(1/2)),
        x[1]+self.resolution/(2**(1/2)))))
    if(self.is_free(self.snap_to_grid((x[0]-self.resolution/(2**(1/2)),
    x[1]-self.resolution/(2**(1/2)))))):
        neighbors.append(self.snap_to_grid((x[0]-self.resolution/(2**(1/2)),
        x[1]-self.resolution/(2**(1/2)))))
    ########## Code ends here ##########
    return neighbors

    def solve(self):
    ########## Code starts here ##########
    while self.open_set:
        x_curr = self.find_best_est_cost_through()
        if x_curr == self.x_goal:
            self.path = self.reconstruct_path()
            return True
```

```
            self.open_set.remove(x_curr)
            self.closed_set.add(x_curr)
            for x in self.get_neighbors(x_curr):
                if x in self.closed_set:
                    continue
                tent_cost_to_arrive = self.cost_to_arrive[x_curr] + self.distance(x,x_curr)
                if x not in self.open_set:
                    self.open_set.add(x)
                elif tent_cost_to_arrive > self.cost_to_arrive[x]:
                    continue
                self.came_from[x] = x_curr
                self.cost_to_arrive[x] = tent_cost_to_arrive
                self.est_cost_through[x] = tent_cost_to_arrive + self.distance(x,self.x_goal)

        return False
        ########## Code ends here ##########
```

sim astar.ipynb

```
    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 #####
        ts = np.cumsum(np.linalg.norm(np.diff(path,axis = 0),axis = -1)/v_desired)
        ts = np.insert(ts,0,0.0)
        path_x_spline = scipy.interpolate.splrep(ts,path[:,0],s = spline_alpha)
        path_y_spline = scipy.interpolate.splrep(ts,path[:,1],s = spline_alpha)
        ###### YOUR CODE END HERE ######
```

P2 rrt.py

```
    class RRT(object):
        def solve(self, eps, max_iters=1000, goal_bias=0.05, shortcut=False):
        ########## Code starts here ##########

        for i in range(max_iters):
            if np.random.uniform(0,1) < goal_bias:
                x_rand = self.x_goal
            else:
                x_rand = np.array((np.random.uniform(self.statespace_lo,self.statespace_hi)))
            near_indx = self.find_nearest(V[:n],x_rand)
            x_near = V[near_indx,:]
            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] = near_indx
                n += 1
                if np.array_equal(self.x_goal,x_new):
                    success = True
                    V[n] = self.x_goal
                    P[n] = n-1
```

```
                break

        self.path = [V[n]]
        current_index = n
        while current_index != 0:
            current_index = P[current_index]
            self.path.append(V[current_index])
        self.path.reverse()

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

    def shortcut_path(self):
        ########## Code starts here ##########
        success = False
        while not success:
            success = True
            i = 0
            while i < len(self.path) - 2:
                if self.is_free_motion(self.obstacles, self.path[i], self.path[i+2]):
                    self.path.pop(i+1)
                    success = False
                else:
                    i += 1
        ########## Code ends here ##########

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.

        return np.argmin(np.linalg.norm(V-x,axis=1))

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

    def steer_towards(self, x1, x2, eps):
        # Consult function specification in parent (RRT) class.
        ########## Code starts here ##########
        # Hint: This should take 1-4 line.
        if np.linalg.norm(x1-x2) < eps:
            return x2
        else:
            return x1+(x2-x1)/np.linalg.norm(x1-x2)*eps
        ########## Code ends here ##########
        pass
```

P3 trajectory optimization.ipynb

```
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
        # HINT: you may find `unpack_decision_variables` useful here. z is the packed 1D
            representation of t,s and u. Return the value of the cost.
        t,s,u = unpack_decision_variables(z)
        return time_weight* t + np.sum(np.square(u))*t/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 dynamics constraints
            x_next, y_next, th_next = s[i+1]
            constraint_list.append([x_next - x - V * np.cos(th) * dt,
                                    y_next - y - V * np.sin(th) * dt,
                                    th_next - th - om * dt])
```

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

        ############################## Code starts here ##############################
        # TODO: Append to ‘constraint_list‘ with initial and final state constraints
        constraint_list.append(s[0]-s_0)
        constraint_list.append(s[-1]-s_f)
        ############################## Code ends here ##############################
        return np.concatenate(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(np.sqrt((x - OBSTACLE_POS[0])**2 + (y - OBSTACLE_POS[1])**2) -
              (EGO_RADIUS + OBS_RADIUS))
          ############################## Code ends here ################################
      return np.array(constraint_list)
```

head controller.py

```
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__('HeadingController')
        self.kp = 2.0
        self.get_logger().info("HeadingController Created")

    def compute_control_with_goal(self,state,goal):
        msgs = TurtleBotControl()
        msgs.omega = self.kp*wrap_angle(goal.theta-state.theta)
        return msgs

if __name__ == "__main__":
    rclpy.init()
    headingcontroller = HeadingController()
    rclpy.spin(headingcontroller)
    # headingcontroller.destroy_node()
    rclpy.shutdown()
```