Principles of Robot Autonomy: Homework 01

Purushotham Mani
10/08/24

Other students worked with: None
Time spent on homework: approx. 5 Hours

Problem 1:

1. Simple Environment A* plot:

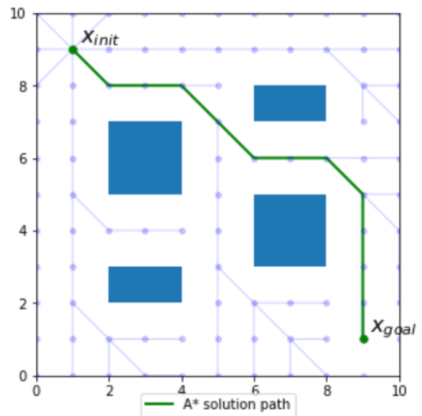

Figure 1: Solution

2. Smooth Trajectory:

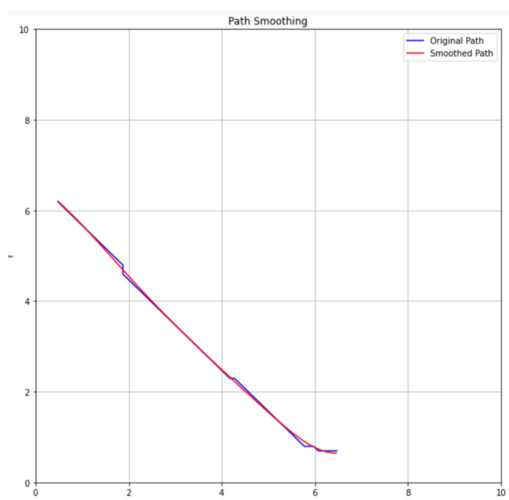

Figure 2: Smooth Trajectory using Cubic spline

Problem 2:

1. Geometric Planning (RRT):

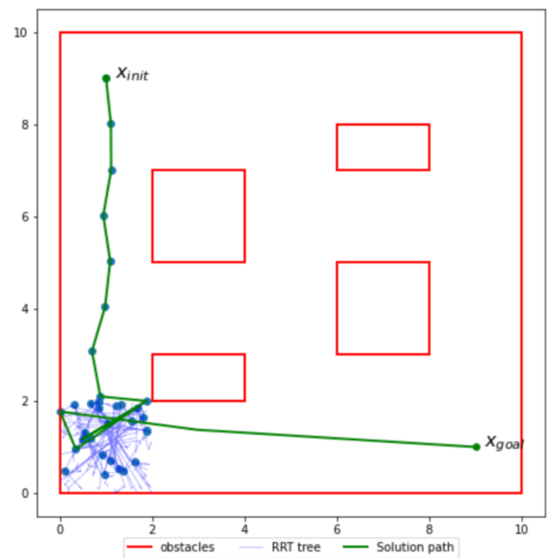

Figure 3: RRT Solution

2. Shortcut Path:

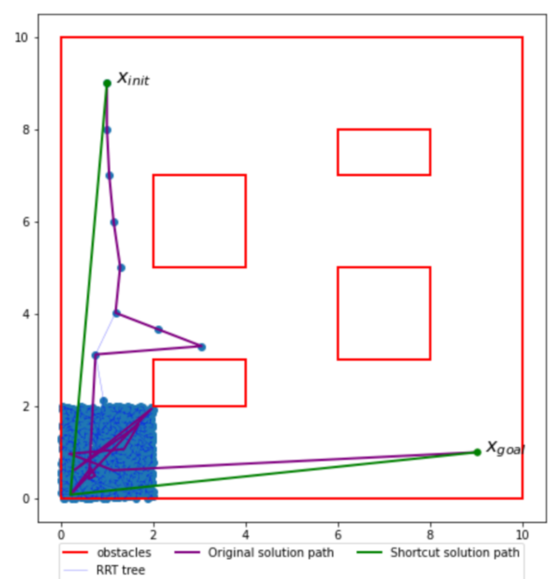

Figure 4: Shortcut Path

Problem 3:

1. Trajectory Optimization:
   Objective function:

$$J = \sum_{t=0}^{T_f} (\alpha + v_t^2 + w_t^2) * \Delta t$$

   Constraints:

$$\begin{aligned} x_{t+1} &= x_t + v_t * \cos\theta_t * \Delta t, \\ y_{t+1} &= y_t + v_t * \sin\theta_t * \Delta t, \\ \theta_{t+1} &= \theta_t + w_t * \Delta t, \\ S_0 &= S_{initial}, \\ S_{T_f} &= S_{final}, \end{aligned}$$

$$\sqrt{(x_t - x_{obstacle})^2 + (y_t - y_{obstacle})^2} - (r_{ego} + r_{obstacle}) \geq 0$$

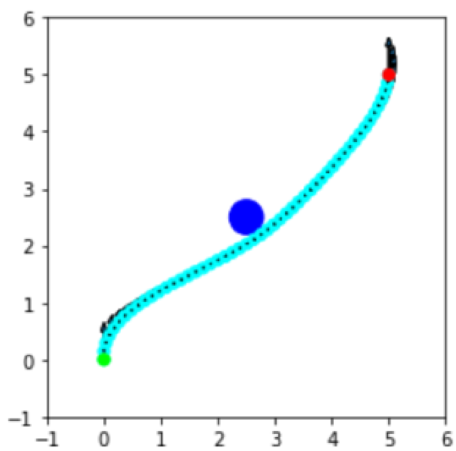

Figure 5: Trajectory Optimization Result

2. Explain the differences that you see with the different choices of α.
   As α increases, the relative weightage given to time taken to reach the goal increases (this is because you are integrating from 0 to T_f). Therefore, with increase in α, we notice that we get shorter path length in euclidean sense.

Problem 4:

1. Heading Controller

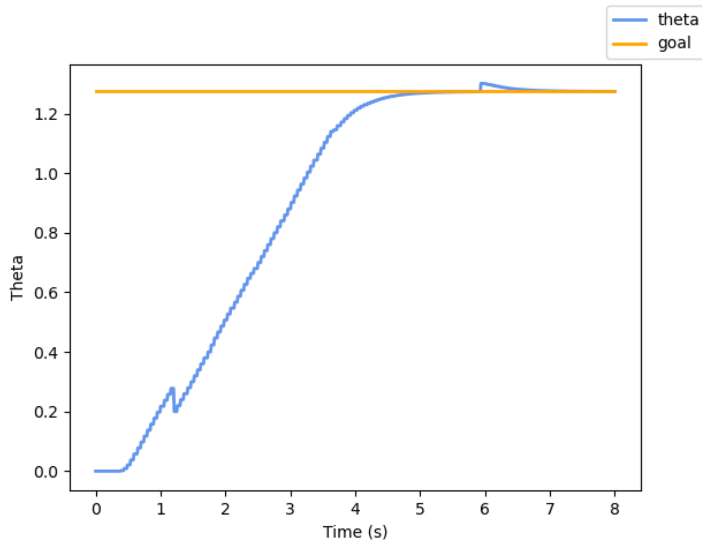

Figure 6: Theta Tracking

Appendix A: Code Submission

0.1 Problem 1: A* motion planning

```python

def is_free(self, x):
    ########## Code starts here ##########
    isfree = self.occupancy.is_free(x)
    isin = (x[0]>=self.statespace_lo[0] and x[0]<=self.statespace_hi[0]) and (x[1]>=self.statespace_lo[1] and x[1]<=self.statespace_hi[1])
    return (isfree and isin)
    ########## 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 ##########
    for i in range(-1,2):
        for j in range(-1,2):
            if (i,j)!=(0,0):
                if i!=0 and j!=0:
                    candidate = self.snap_to_grid((x[0]+(self.resolution*i/1.414213),x[1]+(self.resolution*j/1.414213)))
                    if self.is_free(candidate):
                        neighbors.append(candidate)
                else:
                    candidate = self.snap_to_grid((x[0]+(self.resolution*i),x[1]+(self.resolution*j)))
                    if self.is_free(candidate):
                        neighbors.append(candidate)
    ########## Code ends here ##########
    return neighbors

def solve(self):
    ########## Code starts here ##########
    while len(self.open_set)>0:
        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 neighbor in self.get_neighbors(x_curr):
            if neighbor in self.closed_set:
                continue
            tentative_cost_to_arrive = self.cost_to_arrive[x_curr]+self.distance(x_curr,neighbor)
            if neighbor not in self.open_set:
                self.open_set.add(neighbor)
            elif tentative_cost_to_arrive>self.cost_to_arrive[neighbor]:
                continue
            self.came_from[neighbor]=x_curr
```

```
                self.cost_to_arrive[neighbor]=tentative_cost_to_arrive
                self.est_cost_through[neighbor]=tentative_cost_to_arrive+self.distance(neighbor,x_curr)

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

0.2 Problem 1: Fitting Cubic Spline A*

```
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 = [astar.resolution/v_desired*i for i in range(0,len(path))]
    path_x_spline = scipy.interpolate.splrep(x=ts,y=path[:,0],s=spline_alpha)
    path_y_spline = scipy.interpolate.splrep(x=ts,y=path[:,1],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],
    )

```

0.3 Problem 2: RRT with goal biasing

```
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()
        if z<goal_bias:
            x_rand = self.x_goal
```

```python
        else:
            x_rand = np.random.uniform(self.statespace_lo,state_dim)
        x_near = V[self.find_nearest(V[:n], x_rand)]
        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] = self.find_nearest(V[:n], x_rand)
            if np.linalg.norm(x_new-self.x_goal)<0.01:
                self.path=[self.x_goal]
                while n!=0:
                    self.path.append(V[P[n]])
                    n = P[n]
                self.path.reverse()
                success=True

            n = n + 1
    ########## Code ends here ##########

    plt.figure()
    self.plot_problem()
    self.plot_tree(V, P, color="blue", linewidth=.5, label="RRT tree", alpha=0.5)
    if success:
        if shortcut:
            self.plot_path(color="purple", linewidth=2, label="Original solution path")
            self.shortcut_path()
            self.plot_path(color="green", linewidth=2, label="Shortcut solution path")
        else:
            self.plot_path(color="green", linewidth=2, label="Solution path")
        plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.03), fancybox=True, ncol=3)
        plt.scatter(V[:n,0], V[:n,1])
    else:
        print("Solution not found!")

    return success

def find_nearest(self, V, x):
    ########## 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):
    ########## Code starts here ##########
    # Hint: This should take 1-4 line.
    return (eps * (x2 - x1) / np.linalg.norm(x2 - x1) + x1) if np.linalg.norm(x2 - x1) > eps else x2
    ########## Code ends here ##########
    pass
```

0.4 Problem 2: Shortcut

```python
def shortcut_path(self):
    ########## Code starts here ##########
```

```python
    success=False
    while not success:
        success=True
        i=1
        while (i<len(self.path)-1):
            if self.is_free_motion(self.obstacles, self.path[i-1],self.path[i+1]):
                del self.path[i]
                i=i-1
                success=False
            i+=1
    ########## Code ends here ##########
```

0.5 Problem 3: Trajectory Optimization

```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[1],EGO_FINAL_GOAL_POS[1],np.pi/2])  # Final state.
############################## Code ends here ##############################

def optimize_trajectory(time_weight: float = 1.0, verbose: bool = True):

    def cost(z):
        ############################## Code starts here ##############################
        t_f, s, u = unpack_decision_variables(z)
        dt = t_f / N
        J=0
        for i in range(N):
            J+= (time_weight+(u[i][0])**2+(u[i][1])**2)*dt

        return J
        ############################## 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))

    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
            constraint_list.append(np.array([s[i + 1][0] - (x + V * np.cos(th) * dt)]))
            constraint_list.append(np.array([s[i + 1][1] - (y + V * np.sin(th) * dt)]))
            constraint_list.append(np.array([s[i + 1][2] - (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[N] - 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) -
            ↪  (OBS_RADIUS + EGO_RADIUS))
            ############################## 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
                      }])
    if verbose:
        print(result)
    return unpack_decision_variables(result.x)
```

0.6 Problem 4: Heading Controller

```
#!/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
```

```python
from asl_tb3_msgs.msg import TurtleBotControl, TurtleBotState  # Corrected import statement

class HeadingController(BaseHeadingController):
    def __init__(self):
        super().__init__('HeadingController')
        self.kp = 2.0

    def compute_control_with_goal(self, h_curr: TurtleBotState, h_des: TurtleBotState) -> TurtleBotControl:
        """
        Takes in the current and desired state of type TurtleBotState,
        and returns control message of type TurtleBotControl.
        """
        # print(h_curr)
        err = wrap_angle(h_des.theta - h_curr.theta)

        msg = TurtleBotControl()
        msg.omega = self.kp * err
        return msg

def main():
    rclpy.init(args=None)
    heading_controller = HeadingController()
    rclpy.spin(heading_controller)
    rclpy.shutdown()

if __name__ == "__main__":
    main()
```