CS 237A Homework 2

Kevin Tan

October 2019

Problem 1: A* Motion Planning

(i) Implement a number of functions that, combined, make up the A* motion planning algorithm.

```python
def is_free(self, x):
    """
    Checks if a give state is free, meaning it is inside the bounds of the map and
    is not inside any obstacle.
    Inputs:
        x: state tuple
    Output:
        Boolean True/False
    """
    x_pos, y_pos = x
    x_lo, y_lo = self.statespace_lo
    x_hi, y_hi = self.statespace_hi
    return x_lo <= x_pos <= x_hi and y_lo <= y_pos <= y_hi and self.occupancy.is_free(x)

def distance(self, x1, x2):
    """
    Computes the Euclidean distance between two states.
    Inputs:
        x1: First state tuple
        x2: Second state tuple
    Output:
        Float Euclidean distance

    HINT: This should take one line.
    """
    return ((x1[0] - x2[0]) ** 2 + (x1[1] - x2[1]) ** 2) ** 0.5

def get_neighbors(self, x):
    """
    Gets the FREE neighbor states of a given state. Assumes a motion model
    where we can move up, down, left, right, or along the diagonals by an
    amount equal to self.resolution.
    Input:
        x: tuple state
    Ouput:
        List of neighbors that are free, as a list of TUPLES
```

```
    """
    x_pos, y_pos = x
    neighbors = []
    for del_x, del_y in itertools.product((-1, 0, 1), (-1, 0, 1)):
        if del_x == 0 and del_y == 0:
            continue
        neighbor = self.snap_to_grid((x_pos + del_x, y_pos + del_y))
        if self.is_free(neighbor):
            neighbors.append(neighbor)
    return neighbors

def solve(self):
    """
    Solves the planning problem using the A* search algorithm. It places
    the solution as a list of tuples (each representing a state) that go
    from self.x_init to self.x_goal inside the variable self.path
    Input:
        None
    Output:
        Boolean, True if a solution from x_init to x_goal was found
    """
    while len(self.open_set) != 0:
        # get the lowest cost node from the frontier
        curr = self.find_best_est_cost_through()
        # check if the node is the end
        if curr == self.x_goal:
            self.path = self.reconstruct_path()
            return True
        self.open_set.remove(curr)
        self.closed_set.add(curr)
        for neighbor in self.get_neighbors(curr):
            if neighbor in self.closed_set:
                continue
            tentative_cost = self.cost_to_arrive[curr] + self.distance(curr, neighbor)
            if neighbor not in self.open_set:
                self.open_set.add(neighbor)
            elif tentative_cost > self.cost_to_arrive[neighbor]:
                continue
            self.came_from[neighbor] = curr
            self.cost_to_arrive[neighbor] = tentative_cost
            self.est_cost_through[neighbor] = tentative_cost + self.distance(neighbor, self.x_goal
    return False
```

(ii) The Jupyter Notebook has been included in another file in the submission.

Problem 2: Rapidly-Exploring Random Trees (RRTs)

(i) This part of the problem asks us to implement geometric RRT (RRT that doesn't take into account kinematic constraints, steers in straight lines, and makes infinitely sharp turns).

```
def solve(self, eps, max_iters=1000, goal_bias=0.05, shortcut=False):
    """
    Constructs an RRT rooted at self.x_init with the aim of producing a
    dynamically-feasible and obstacle-free trajectory from self.x_init
    to self.x_goal.

    Inputs:
        eps: maximum steering distance
        max_iters: maximum number of RRT iterations (early termination
            is possible when a feasible solution is found)
        goal_bias: probability during each iteration of setting
            x_rand = self.x_goal (instead of uniformly randomly sampling
            from the state space)
    Output:
        None officially (just plots), but see the "Intermediate Outputs"
        descriptions below
    """
    state_dim = len(self.x_init)
    V = np.zeros((max_iters, state_dim))
    V[0, :] = self.x_init    # RRT is rooted at self.x_init
    n = 1
    P = -np.ones(max_iters, dtype=int)
    success = False

    for _ in range(max_iters - 1):
        # randomly sample a point (or goal)
        z = random.uniform(0, 1)
        if z < goal_bias:
            sample = self.x_goal
        else:
            sample = self.random_state()
        # find the nearest point and steer it towards random sample
        nearest_index = self.find_nearest(V[range(n), :], sample)
        nearest = V[nearest_index]
        new = self.steer_towards(nearest, sample, eps)
        if self.is_free_motion(self.obstacles, nearest, new):
            V[n, :], P[n], n = new, nearest_index, n+1
            if np.array_equal(new, self.x_goal):
                self.path = self.reconstruct_path(V, P, n)
                success = True
                break

def find_nearest(self, V, x):
        """Returns the index of the nearest point in V to x."""
        min_index, min_value = 0, float('inf')
        for index, value in enumerate(V):
            distance = np.linalg.norm(value - x)
```

```
            if distance < min_value:
                min_index = index
                min_value = distance
        return min_index

def steer_towards(self, x1, x2, eps):
    angle = math.atan2(x2[1] - x1[1], x2[0] - x1[0])
    if if np.linalg.norm(x2 - x1) < eps:
        return x2
    else:
        return np.array([
            x1[0] + eps * math.cos(angle), x1[1] + eps * math.sin(angle)
        ])
```

(ii) This part asks us to implement a basic post-processing algorithm "Shortcut" that removes extraneous nodes between nodes for which there exists a straight line path that does not violate state space constraints. It, however, does not fix the fact that the resulting trajectory still does not obey kinematic constraints.

```
def shortcut_path(self):
    """
    Iteratively removes nodes from solution path to find a shorter path
    which is still collision-free.
    Input:
        None
    Output:
        None, but should modify self.path
    """
    success = False
    while not success:
        success = True
        for i in range(1, len(self.path) - 1):
            if self.is_free_motion(self.obstacles, self.path[i-1], self.path[i+1]):
                self.path.pop(i)
                success = False
                break
```

(iii) This question asks us to implement RRT where we are no longer generating straight-line trajectories between points in our tree and sampled points, but, rather, trajectories that satisfy the kinematic constraints of Dubin's car (a car that moves at a fixed velocity).

```
def find_nearest(self, V, x):
    """Returns the index of the nearest point in V to x."""
    min_index, min_value = 0, float('inf')
    for index, value in enumerate(V):
        distance = dubins.path_length(value, x, self.turning_radius)
        if distance < min_value:
            min_index = index
            min_value = distance
    return min_index

```

```python
def steer_towards(self, x1, x2, eps):
    """
    A subtle issue: if you use dubins.path_sample to return the point
    at distance eps along the path from x to y, use a turning radius
    slightly larger than self.turning_radius
    (i.e., 1.001*self.turning_radius). Without this hack,
    dubins.path_sample might return a point that can't quite get to in
    distance eps (using self.turning_radius) due to numerical precision
    issues.
    """
    if dubins.path_length(x1, x2, self.turning_radius) < eps:
        return x2
    else:
        return dubins.path_sample(x1, x2, 1.001*self.turning_radius, eps)[0][1]
```

(iv) The PDF of the Jupyter notebook has been included separately.

Problem 3: Geometric Planning to Trajectories and Control

(i) This problem asks us to use spline interpolation on a rough piecewise trajectory to get a smoother trajectory.

```python
def compute_smoothed_traj(path, V_des, alpha, dt):
    """
    Fit cubic spline to a path and generate a resulting trajectory for our
    wheeled robot.

    Inputs:
        path (np.array [N,2]): Initial path
        V_des (float): Desired nominal velocity, used as a heuristic to assign nominal
            times to points in the initial path
        alpha (float): Smoothing parameter (see documentation for
            scipy.interpolate.splrep)
        dt (float): Timestep used in final smooth trajectory
    Outputs:
        traj_smoothed (np.array [N,7]): Smoothed trajectory
        t_smoothed (np.array [N]): Associated trajectory times
    Hint: Use splrep and splev from scipy.interpolate
    """
    distances = np.zeros(len(path)-1)
    for i in range(len(path)-1):
        x1, y1 = path[i]
        x2, y2 = path[i+1]
        distances[i] = ((x2-x1)**2 + (y2-y1)**2)**0.5
    timesteps = map(lambda distance: round((distance / V_des) + 0.5), distances)
    intervals = np.zeros(len(path))
    for i in range(1, len(intervals)):
        intervals[i] = intervals[i-1] + timesteps[i-1] * dt
    # interpolate the x and y values
    old_x = np.array([tup[0] for tup in path])
```

```
    old_y = np.array([tup[1] for tup in path])
    tck_x = splrep(intervals, old_x, s=alpha)
    tck_y = splrep(intervals, old_y, s=alpha)
    total_time = intervals[-1]
    times = np.linspace(0, total_time, total_time/dt)
    smooth_traj = np.zeros((len(times), 7))
    smooth_traj[:,0], smooth_traj[:,1] = splev(times, tck_x), splev(times, tck_y)
    smooth_traj[:,3], smooth_traj[:,4] = splev(times, tck_x, der=1), splev(times, tck_y, der=1)
    smooth_traj[:,2] = np.vectorize(math.atan2)(smooth_traj[:,4], smooth_traj[:,3])
    smooth_traj[:,5], smooth_traj[:,6] = splev(times, tck_x, der=2), splev(times, tck_y, der=2)
    return smooth_traj, times
```

(ii) This problem asks us to use time rescaling to revise our smoothed trajectory into one that respects our control input constraints.

```
def modify_traj_with_limits(traj, t, V_max, om_max, dt):
    """
    Modifies an existing trajectory to satisfy control limits and
    interpolates for desired timestep.

    Inputs:
        traj (np.array [N,7]): original trajectory
        t (np.array [N]): original trajectory times
        V_max, om_max (float): control limits
        dt (float): desired timestep
    Outputs:
        t_new (np.array [N_new]) new timepoints spaced dt apart
        V_scaled (np.array [N_new])
        om_scaled (np.array [N_new])
        traj_scaled (np.array [N_new, 7]) new rescaled traj at these timepoints
    Hint: This should almost entirely consist of calling functions from Problem Set 1
    """
    velocity, omega = P1.compute_controls(traj)
    s = P1.compute_arc_length(velocity, t)
    V_tilde = P1.rescale_V(velocity, omega, V_max, om_max)
    tau = P1.compute_tau(V_tilde, s)
    om_tilde = P1.rescale_om(velocity, omega, V_tilde)
    x, y, th, _, _, _, _ = traj[-1]
    s_f = P1.State(x, y, V_tilde[-1], th)
    return P1.interpolate_traj(traj, tau, V_tilde, om_tilde, dt, s_f)
```

(iii) This problem asks us to switch from a trajectory tracking controller to a pose stabilization controller when we are close to our goal.

(iv) Jupyter notebook attached separately.

Problem 4: Bidirectional RRT (RRTConnect)

(i) Implement GeometricRRTConnect.

```python
def find_nearest_forward(self, V, x):
    min_index, min_value = 0, float('inf')
    for index, value in enumerate(V):
        distance = np.linalg.norm(value - x)
        if distance < min_value:
            min_index = index
            min_value = distance
    return min_index

def steer_towards_forward(self, x1, x2, eps):
    angle = math.atan2(x2[1] - x1[1], x2[0] - x1[0])
    return x2 if np.linalg.norm(x2 - x1) < eps else np.array([x1[0] + eps * math.cos(angle), x1[1]

def solve(self, eps, max_iters = 1000):
    """
    Uses RRT-Connect to perform bidirectional RRT, with a forward tree
    rooted at self.x_init and a backward tree rooted at self.x_goal, with
    the aim of producing a dynamically-feasible and obstacle-free trajectory
    from self.x_init to self.x_goal.

    Inputs:
        eps: maximum steering distance
        max_iters: maximum number of RRT iterations (early termination
            is possible when a feasible solution is found)

    Output:
        None officially (just plots), but see the "Intermediate Outputs"
        descriptions below
    """

    state_dim = len(self.x_init)

    # represent the forward tree
    V_fw = np.zeros((max_iters, state_dim))     # nodes
    V_fw[0,:] = self.x_init
    n_fw = 1                                    # number of nodes
    P_fw = -np.ones(max_iters, dtype=int)       # nodal relationships
    fw_tree = [V_fw, P_fw, n_fw]


    # represent the backward tree
    V_bw = np.zeros((max_iters, state_dim))     # nodes
    V_bw[0,:] = self.x_goal
    n_bw = 1                                    # number of nodes
    P_bw = -np.ones(max_iters, dtype=int)       # nodal relationships
    bw_tree = [V_bw, P_bw, n_bw]

    # whether we were able to find a collision-free path
    success = False

    for _ in range(max_iters - 1):
```

```
        if not success:
            success = self.grow_fw_tree(fw_tree, bw_tree, eps)
        if not success:
            success = self.grow_bw_tree(fw_tree, bw_tree, eps)

    # update n_fw and n_bw (the grow_*_tree methods do not)
    n_fw = fw_tree[2]
    n_bw = bw_tree[2]

def grow_bw_tree(self, fw_tree, bw_tree, eps):
    """Samples a random point in the state space, grows the backward tree towards
    the random point, and then tries best to connect the forward tree to the state
    newly added to the backward tree.
    """
    V_bw, _, n_bw = bw_tree
    V_fw, _, n_fw = fw_tree
    # sample point (x_rand) and steer towards it (x_near)
    x_rand = self.random_state()
    x_near_index = self.find_nearest_backward(V_bw[range(n_bw),:], x_rand)
    x_near = V_bw[x_near_index]
    x_new = self.steer_towards_backward(x_rand, x_near, eps)
    # check if new path violates state space constraints
    if self.is_free_motion(self.obstacles, x_new, x_near):
        # add vertex and associated edge
        self.add_to_tree(bw_tree, x_new, x_near_index)
        # find nearest point in backward tree (x_connect) to the (x_new)
        x_connect_index = self.find_nearest_forward(V_fw[range(n_fw),:], x_new)
        x_connect = V_fw[x_connect_index]
        while True:
            # repeatedly try to extend (x_connect) towards (x_new)
            x_newconnect = self.steer_towards_forward(x_connect, x_new, eps)
            if self.is_free_motion(self.obstacles, x_connect, x_newconnect):
                self.add_to_tree(fw_tree, x_newconnect, x_connect_index)
                if np.array_equal(x_newconnect, x_new):
                    self.path = self.reconstruct_path(bw_tree, fw_tree, x_newconnect
                    return True
                x_connect = x_newconnect
            else:
                break
    # new path violates state space constraints or unable to connect backward tree
    return False

def grow_fw_tree(self, fw_tree, bw_tree, eps):
    """Samples a random point in the state space, grows the forward tree towards
    the random point, and then tries best to connect the backward tree to the state
    newly added to the forward tree.
    """
    V_fw, _, n_fw = fw_tree
    V_bw, _, n_bw = bw_tree
    # sample point (x_rand) and steer towards it (x_near)
    x_rand = self.random_state()
```

```
    x_near_index = self.find_nearest_forward(V_fw[range(n_fw),:], x_rand)
    x_near = V_fw[x_near_index]
    x_new = self.steer_towards_forward(x_near, x_rand, eps)
    # check if new path violates state space constraints
    if self.is_free_motion(self.obstacles, x_near, x_new):
        # add vertex and associated edge
        self.add_to_tree(fw_tree, x_new, x_near_index)
        # find nearest point in backward tree (x_connect) to the (x_new)
        x_connect_index = self.find_nearest_backward(V_bw[range(n_bw),:], x_new)
        x_connect = V_bw[x_connect_index]
        while True:
            # repeatedly try to extend (x_connect) towards (x_new)
            x_newconnect = self.steer_towards_backward(x_new, x_connect, eps)
            if self.is_free_motion(self.obstacles, x_newconnect, x_connect):
                self.add_to_tree(bw_tree, x_newconnect, x_connect_index)
                if np.array_equal(x_newconnect, x_new):
                    self.path = self.reconstruct_path(fw_tree, bw_tree, x_newconnect
                    return True
                x_connect = x_newconnect
            else:
                break
    # new path violates state space constraints or unable to connect backward tree
    return False

def reconstruct_path(self, fw_tree, bw_tree, intersection):
    """Reconstructs the bidirectional RRT given the forward and backward trees."""
    V_fw, P_fw, n_fw = fw_tree
    V_bw, P_bw, n_bw = bw_tree
    path_fw, path_bw = [], []
    # find the subpath in the forward tree
    curr_fw = self.find_nearest_forward(V_fw[range(n_fw), :], intersection)
    while curr_fw != -1:
        path_fw.append(V_fw[curr_fw])
        curr_fw = P_fw[curr_fw]
    path_fw.reverse()
    # find the subpath in the backward tree
    curr_bw = self.find_nearest_backward(V_bw[range(n_bw), :], intersection)
    while curr_bw != -1:
        path_bw.append(V_bw[curr_bw])
        curr_bw = P_bw[curr_bw]
    return path_fw + path_bw

def add_to_tree(self, tree, node, parent_index):
    """Adds a new node to a tree, given its parent's index in the tree."""
    n = tree[2]
    tree[0][n,:], tree[1][n] = node, parent_index
    tree[2] += 1


def random_state(self):
    """Returns a random state in the free space."""
```

```python
    return [
        random.uniform(self.statespace_lo[dim], self.statespace_hi[dim])
        for dim in range(len(self.statespace_lo))
    ]
```

(ii) Implement DubinsRRTConnect.

```python
def find_nearest_forward(self, V, x):
    """Returns the index of the nearest point in V to x."""
    min_index, min_value = 0, float('inf')
    for index, value in enumerate(V):
        distance = path_length(value, x, self.turning_radius)
        if distance < min_value:
            min_index = index
            min_value = distance
    return min_index

def find_nearest_backward(self, V, x):
    """Returns the index of the nearest point in V to x."""
    min_index, min_value = 0, float('inf')
    for index, value in enumerate(V):
        distance = path_length(x, value, self.turning_radius)
        if distance < min_value:
            min_index = index
            min_value = distance
    return min_index

def steer_towards_forward(self, x1, x2, eps):
    if path_length(x1, x2, self.turning_radius) < eps:
        return x2
    else:
        return path_sample(x1, x2, 1.001*self.turning_radius, eps)[0][1]

def steer_towards_backward(self, x1, x2, eps):
    if path_length(x1, x2, self.turning_radius) < eps:
        return x1
    else:
        return path_sample(x1, x2, 1.001*self.turning_radius, eps)[0][-1]
```