HW2

March 7, 2022

This uses a Toroidal A* algorithm to find a path between two points in the link arm's configuration space. I wanted to use my previous A* algorithm to do this originally, but ended up using Daniel Ingram's instead as converting mine from use on a Euclidean grid to a Toroidal grid proved too difficult and time consuming.

To find the path, the start and goal need to first be converted from (x, y) coordinates in the workspace into (θ_1, θ_2) coordinates in the configuration space. This is done with inverse kinematics.

1 Two-Link Manip

main.py

[ ]:
```
import math
from math import pi
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import from_levels_and_colors

plt.ion()

# Simulation parameters
resolution = 100
obstacles = [[11, 11, 4]]   # x, y, r
start = (9, 15)  # x, y
goal = (15, 9)
a1 = 10
a2 = 10


def main():
    arm = NLinkArm([a1, a2], [0, 0])
    grid = get_occupancy_grid(arm)

    start_angles = get_angles(grid, start)
    goal_angles = get_angles(grid, goal)
    if start_angles == (-1, -1) or goal_angles == (-1, -1):
        print("No path exists")
        return
```

```
    plt.imshow(grid)
    plt.show()
    route = astar_torus(grid, start_angles, goal_angles)
    for node in route:
        theta1 = 2 * pi * node[0] / resolution - pi
        theta2 = 2 * pi * node[1] / resolution - pi
        arm.update_joints([theta1, theta2])
        arm.plot(obstacles)


def get_angles(grid, node):
    D = (node[0] * node[0] + node[1] * node[1] - a1 * a1 - a2 * a2) / (2 * a1 *␣
 ↪a2)
    t2 = math.atan2(math.sqrt(1 - D * D), D)
    t1 = math.atan2(node[1], node[0]) - math.atan2(a1 * math.sin(t2), a1 + a2 *␣
 ↪math.cos(t2))
    t1 = round(t1 * resolution / 2 / pi) + resolution // 2
    t2 = round(t2 * resolution / 2 / pi) + resolution // 2

    if grid[t1][t2] == 1:
        t2 = -math.atan2(math.sqrt(1 - D * D), D)
        t1 = math.atan2(node[1], node[0]) - math.atan2(a1 * math.sin(t2), a1 +␣
 ↪a2 * math.cos(t2))
        t1 = round(t1 * resolution / 2 / pi) + resolution // 2
        t2 = round(t2 * resolution / 2 / pi) + resolution // 2

    if grid[t1][t2] == 1:
        return -1, -1

    return t1, t2


def detect_collision(line_seg, circle):
    p0 = np.array([circle[0], circle[1]])  # x0, y0
    p1 = np.array([line_seg[0][0], line_seg[0][1]])  # x1, y1
    p2 = np.array([line_seg[1][0], line_seg[1][1]])  # x2, y2
    radius2 = circle[2] * circle[2]
    linkLen2 = line_seg[2] * line_seg[2]

    d10 = p1 - p0
    d20 = p2 - p0
    d21 = p2 - p1
    c = d21[0] * d10[1] - d10[0] * d21[1]
    dist2 = c * c / linkLen2

    if dist2 <= radius2:
```

```
        if d10[0] * d10[0] + d10[1] * d10[1] <= radius2:
            return True
        if d20[0] * d20[0] + d20[1] * d20[1] <= radius2:
            return True
        dMid0 = p1 + d21 / 2 - p0
        hypot2 = dMid0[0] * dMid0[0] + dMid0[1] * dMid0[1]
        if hypot2 - dist2 <= linkLen2 / 4:
            return True
    return False


def get_occupancy_grid(arm):
    grid = [[0 for _ in range(resolution)] for _ in range(resolution)]
    theta_list = [2 * i * pi / resolution for i in range(-resolution // 2,␣
 ↪resolution // 2 + 1)]
    for i in range(resolution):
        for j in range(resolution):
            arm.update_joints([theta_list[i], theta_list[j]])
            points = arm.points
            collision_detected = False
            for k in range(len(points) - 1):
                for obstacle in obstacles:
                    line_seg = [points[k], points[k + 1], arm.link_lengths[k]]
                    collision_detected = detect_collision(line_seg, obstacle)
                    if collision_detected:
                        break
                if collision_detected:
                    break
            grid[i][j] = int(collision_detected)
    return np.array(grid)


def astar_torus(grid, start_node, goal_node):
    """
    Author: Daniel Ingram (daniel-s-ingram)
    https://github.com/AtsushiSakai/PythonRobotics/blob/master/ArmNavigation/
 ↪arm_obstacle_navigation/arm_obstacle_navigation.py
    Finds a path between an initial and goal joint configuration using
    the A* Algorithm on a toroidal grid.
    Args:
        grid: An occupancy grid (ndarray)
        start_node: Initial joint configuration (tuple)
        goal_node: Goal joint configuration (tuple)
    Returns:
        Obstacle-free route in joint space from start_node to goal_node
    """
    colors = ['white', 'black', 'red', 'pink', 'yellow', 'green', 'orange']
```

```
    levels = [0, 1, 2, 3, 4, 5, 6, 7]
    cmap, norm = from_levels_and_colors(levels, colors)

    grid[start_node] = 4
    grid[goal_node] = 5

    parent_map = [[() for _ in range(resolution)] for _ in range(resolution)]

    heuristic_map = calc_heuristic_map(goal_node)

    explored_heuristic_map = np.full((resolution, resolution), np.inf)
    distance_map = np.full((resolution, resolution), np.inf)
    explored_heuristic_map[start_node] = heuristic_map[start_node]
    distance_map[start_node] = 0
    while True:
        grid[start_node] = 4
        grid[goal_node] = 5

        current_node = np.unravel_index(
            np.argmin(explored_heuristic_map, axis=None),␣
↪explored_heuristic_map.shape)
        min_distance = np.min(explored_heuristic_map)
        if (current_node == goal_node) or np.isinf(min_distance):
            break

        grid[current_node] = 2
        explored_heuristic_map[current_node] = np.inf

        i, j = current_node[0], current_node[1]

        neighbors = find_neighbors(i, j)

        for neighbor in neighbors:
            if grid[neighbor] == 0 or grid[neighbor] == 5:
                distance_map[neighbor] = distance_map[current_node] + 1
                explored_heuristic_map[neighbor] = heuristic_map[neighbor]
                parent_map[neighbor[0]][neighbor[1]] = current_node
                grid[neighbor] = 3

    if np.isinf(explored_heuristic_map[goal_node]):
        route = []
        print("No route found.")
    else:
        route = [goal_node]
        while parent_map[route[0][0]][route[0][1]] != ():
            route.insert(0, parent_map[route[0][0]][route[0][1]])
```

```
        print("The route found covers %d grid cells." % len(route))
        for i in range(1, len(route)):
            grid[route[i]] = 6
            plt.cla()
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect('key_release_event',
                                         lambda event: [exit(0) if event.key ==␣
↪'escape' else None])
            plt.imshow(grid, cmap=cmap, norm=norm, interpolation=None)
            plt.show()
            plt.pause(0.001)

    return route


def find_neighbors(i, j):
    neighbors = []
    if i - 1 >= 0:
        neighbors.append((i - 1, j))
    else:
        neighbors.append((resolution - 1, j))

    if i + 1 < resolution:
        neighbors.append((i + 1, j))
    else:
        neighbors.append((0, j))

    if j - 1 >= 0:
        neighbors.append((i, j - 1))
    else:
        neighbors.append((i, resolution - 1))

    if j + 1 < resolution:
        neighbors.append((i, j + 1))
    else:
        neighbors.append((i, 0))

    return neighbors


def calc_heuristic_map(goal_node):
    X, Y = np.meshgrid([i for i in range(resolution)], [i for i in␣
↪range(resolution)])
    heuristic_map = np.abs(X - goal_node[1]) + np.abs(Y - goal_node[0])
    for i in range(heuristic_map.shape[0]):
        for j in range(heuristic_map.shape[1]):
            heuristic_map[i, j] = min(heuristic_map[i, j],
```

```
                                    i + 1 + heuristic_map[resolution - 1, j],
                                    resolution - i + heuristic_map[0, j],
                                    j + 1 + heuristic_map[i, resolution - 1],
                                    resolution - j + heuristic_map[i, 0])
    return heuristic_map


class NLinkArm:
    """
    Author: Daniel Ingram (daniel-s-ingram)
    Class for controlling and plotting a planar arm with an arbitrary number of␣
 ↪links.
    """
    def __init__(self, link_lengths, joint_angles):
        self.n_links = len(link_lengths)
        if self.n_links != len(joint_angles):
            raise ValueError()

        self.link_lengths = np.array(link_lengths)
        self.joint_angles = np.array(joint_angles)
        self.points = [[0, 0] for _ in range(self.n_links + 1)]
        self.end_effector = np.array(self.points[self.n_links])
        self.lim = sum(link_lengths)
        self.update_points()

    def update_joints(self, joint_angles):
        self.joint_angles = joint_angles
        self.update_points()

    def update_points(self):
        for i in range(1, self.n_links + 1):
            self.points[i][0] = self.points[i - 1][0] + \
                                self.link_lengths[i - 1] * \
                                np.cos(np.sum(self.joint_angles[:i]))
            self.points[i][1] = self.points[i - 1][1] + \
                                self.link_lengths[i - 1] * \
                                np.sin(np.sum(self.joint_angles[:i]))
        self.end_effector = np.array(self.points[self.n_links]).T

    def plot(self, obs=None):  # pragma: no cover
        if obs is None:
            obs = []
        plt.cla()

        for obstacle in obs:
            circle = plt.Circle(
                (obstacle[0], obstacle[1]), radius=obstacle[2], fc='k')
```

```
            plt.gca().add_patch(circle)

        for i in range(self.n_links + 1):
            if i is not self.n_links:
                plt.plot([self.points[i][0], self.points[i + 1][0]],
                         [self.points[i][1], self.points[i + 1][1]], 'r-')
            plt.plot(self.points[i][0], self.points[i][1], 'k.')

        plt.xlim([-self.lim, self.lim])
        plt.ylim([-self.lim, self.lim])
        plt.draw()
        plt.pause(0.001)


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