Hybird A* algorithm part description

July 30, 2019

0.1 config.py

- This file is used to store known or artificially preset parameter variables.
- Here I preset two scenes, reverse parking and parallel parking. And assume that the map is a binary map containing obstacles
- Setting different resolutions in the setting is to achieve different accuracy requirements for different objects and thus reduce the computational cost.
  - Subsequent optimization direction: Set dynamic resolution based on obstacles

In [2]:
```
import numpy as np

# THE PARAMETERS OF VEHICLE
# all parameters are not based on real vehicles
parameters_vehicle = {
    'body_length': 4.7,                 # m
    'rear2back': 1.0,                   # m
    'rear2front': 3.7,                  # m
    'body_width': 2,                    # m
    'wheel_base': 2.7,                  # m
    'max_steering_angle': 0.6,          # rad
    'minimum_turning_radius': 1         # m
}

# KINEMATICS RESTRICTIONS
# all parameters are not based on real vehicles
constraints = {
    'velocity': [-1, 2],                # m/s
    'accekeration': [-1, 1],            # m/s^2
    'steering_angle_rate': [-0.6, 0.6], # rad/s
}

# PARAMETERS HYBIRD A*
setting = {
    'coordinate_resolution': 0.3,       # m
    'motion_resolution': 0.1,           # m
    'obstacle_resolution': 0.1,         # m
    'yaw_resolution': np.pi / 36,       # rad(= 5 deg)
```

```
    'num_steer': 5                      # number of optional steering angle
}

cost_weights = {
    'back_cost': 0.0,                   # backward penalty
    'switch_cost': 10.0,                # switch direction penalty
    'steer_cost': 0.0,                  # steer angle penalty
    'change_steer_cost': 10.0,          # change steer angle penalty
    'heuristic_cost': 1.0               # cost to come penalty
}

# PRE-DEFINED PARKING SCENARIO
# In order to simplify the problem,
# only polyhedra is considered here.
# Obstacle with vertices(clock-wise):
# [[[obst1_x1, obst1_y1],...,[obst1_xn, obst1_yn]],...,
#  [[obstm_x1, obstm_y1],...,[obstm_xn, obstm_yn]]]
list_scenario = ['Backwards', 'Parallel']
scenario = {
    'Backwards': {
        'number_obstacles': 3,
        'vertices_obstacles': [4, 4, 4],
        'coordinates_obstacles':
        np.array([[[-20, 5], [-1.3, 5], [-1.3, -5], [-20, -5]],
                  [[1.3, 5], [20, 5], [20, -5], [1.3, -5]],
                  [[-20, 15], [20, 15], [20, 11], [-20, 11]]],
                 dtype=np.float64),
        'start_point': np.array([[-6], [9.5], [0]]),
        'end_point': np.array([[0], [1.3], [np.pi / 2]])
    },
    'Parallel': {
        'number_obstacles': 4,
        'vertices_obstacles': [4, 4, 4, 4],
        'coordinates_obstacles':
        np.array([[[-15, 5], [-3, 5], [-3, 0], [-15, 0]],
                  [[3, 5], [15, 5], [15, 0], [3, 0]],
                  [[-3, 0], [-3, 2.5], [3, 2.5], [3, 0]],
                  [[-15, 15], [15, 15], [15, 11], [-15, 11]]],
                 dtype=np.float64),
        'start_point': np.array([[-6], [9.5], [0]]),
        'end_point': np.array([[parameters_vehicle['wheel_base'] / 2],
                               [4], [0]])
    }
}
```

0.2 main.py

- The main function contains the whole process of trajectory generation.

1. Preprocess it according to the input map
   - Because the map is my own preset, this part of the code also needs to adapt the map.
   - ***TODO***: increase the readability of the code
2. Call the Hybird A* algorithm to get the initial path
   - The four-wheel steering model is still based on the bicycle model
   - ***TODO***: need to determine a more rigorous model to match the motion state of the four-wheel steering vehicle
   - Feasible space for the next node(control inputs)
     * move: forwards(1), backwards(0)
     * steer_f δ_f[rad]: {-0.6, -0.48, -0.36, -0.24, -0.12, 0, 0.12, 0.24, 0.36, 0.48, 0.6}
     * steer_r δ_r[rad]: {-0.6, -0.48, -0.36, -0.24, -0.12, 0, 0.12, 0.24, 0.36, 0.48, 0.6}
   - cost so far:
     * ***TODO***: may need to change the cost function according to the motion model, use grid search or other methods to get better weight coefficients
   - cost to go: distance to the end point obtained by the A* algorithm as a heuristic
     * ***TODO***: use the distance obtained by the reeds shepp method as heuristic(not necessary, because as long as an initial feasible solution is obtained, a better solution can be obtained by the nonlinear slover.)
   - Use the reeds shepp method to get the curve when approaching the end point
     * **Not finished**
   - Collision check: use a two-step collision detection
     * The first step is a rough check: the car is treated as a circle and the diameter of the circle is the length of the car
     * The second step is stricter: Think of the car itself as a tight rectangle
       · Point in Polygon: If the point is inside, the sign of the outer product between the measured point and each vertex is the same, otherwise it is different.
3. Calculate the velocity using the path obtained above and smooth it
   - Preprocess the state and control variables to feed into the nonlinear solver
   - **Not done**
4. Using a nonlinear optimizer to get a better solution
   - The goal of using this method is to achieve a safe and stable path in a narrow environment.
   - **Not done**

In [ ]:
```
from Config import config
from utilities.preprocessed_map import calc_obst_map
from utilities.hybrid_a_star import calc_path
import time
# import matplotlib.pyplot as plt

for scenario in config.list_scenario:
    ####################################
    # 1. Preprocess                    #
    ####################################
```

```python
binary_map = config.scenario[scenario]
ob_x, ob_y, ob_KDTree, sampled_ob_map_ind, sampled_ob_KDTree, map_info = \
    calc_obst_map(binary_map['coordinates_obstacles'], config.setting,
                  config.parameters_vehicle['minimum turning radius'])

obstacle = {
    'obst_x': ob_x,
    'obst_y': ob_y,
    'obst_KDTree': ob_KDTree,
    'sampled_obst_map_ind': sampled_ob_map_ind,
    'sampled_obst_KDTree': sampled_ob_KDTree,
    'map_info': map_info
}

# code for test
# plt.figure()
# plt.scatter(obst_x, obst_y)
# plt.show()
# plt.imshow(sampled_obst_map_ind.T[::-1, :])
# plt.show()

#####################################
# 2. Call Hybrid A*                 #
#####################################
start_a_star = time.time()
# TODO: Hybrid A* algorithm, generate initial path
init_x, init_y, init_theta = calc_path(
    binary_map['start_point'], binary_map[
        'end_point'], obstacle, config)
time_a_star = (time.time() - start_a_star) * 1000

#####################################
# 3. Call smoother                  #
#####################################
# TODO: calculate the corresponding control variable from the initial path

#####################################
# 4. Call nonlinear solver          #
#####################################
exist_path = 0
start_solver = time.time()
# TODO: nonlinear optimization algorithm, optimize the initial path

time_solver = (time.time() - start_solver) * 1000

if exist_path:
    print('The optimized path has been found\n')
    # TODO: visualize the path
```

```
        else:
            # TODO: randomly change several state values
            #       in the initial path, then let the solver resolve
            print('Path not found yet')

    print('-' * 18 + 'summary' + '-' * 18)
    print('Scenario: %s Parking' % scenario)
    print('Time spent by Hybrid A*: %d ms' % time_a_star)
    print('Time spent by nonlinear solver: %d ms' % time_solver)
    print('-' * 20 + 'END' + '-' * 20)
```

0.2.1 The following are specific code. I have commented on each function. I have also marked some of them that have not yet been completed.

0.3 preprocessed_map.py

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KDTree


class map_boundary():
    '''
    Includes information such as map size and resolution of
    each dimension(xy coor, yaw, obstacle)
    '''

    def __init__(self, obst_coordinates, setting):

        self.xy_reso = setting['coordinate_resolution']
        self.min_sampled_x = np.round(
            obst_coordinates[:, :, 0].min() / self.xy_reso)
        self.max_sampled_x = np.round(
            obst_coordinates[:, :, 0].max() / self.xy_reso)
        self.sampled_x_width = (self.max_sampled_x -
                                self.min_sampled_x).astype(np.int64)
        self.min_sampled_y = np.round(
            obst_coordinates[:, :, 1].min() / self.xy_reso)
        self.max_sampled_y = np.round(
            obst_coordinates[:, :, 1].max() / self.xy_reso)
        self.sampled_y_width = (self.max_sampled_y -
                                self.min_sampled_y).astype(np.int64)

        self.yaw_reso = setting['yaw_resolution']
        self.min_sampled_theta = np.round(-np.pi / self.yaw_reso)
        self.max_sampled_theta = np.round(np.pi / self.yaw_reso)
        self.sampled_theta_width = (self.max_sampled_theta -
```

```python
                                    self.min_sampled_theta).astype(np.int64)

        self.obst_reso = setting['obstacle_resolution']
        self.min_sampled_obst_x = np.round(
            obst_coordinates[:, :, 0].min() / self.obst_reso)
        self.max_sampled_obst_x = np.round(
            obst_coordinates[:, :, 0].max() / self.obst_reso)
        self.sampled_obst_x_width = (self.max_sampled_obst_x -
                                     self.min_sampled_obst_x).astype(np.int64)
        self.min_sampled_obst_y = np.round(
            obst_coordinates[:, :, 1].min() / self.obst_reso)
        self.max_sampled_obst_y = np.round(
            obst_coordinates[:, :, 1].max() / self.obst_reso)
        self.sampled_obst_y_width = (self.max_sampled_obst_y -
                                     self.min_sampled_obst_y).astype(np.int64)

        self.motion_reso = setting['motion_resolution']


def calc_obst_map(obst_coordinates, setting, vehicle_radius):
    '''
    PARAMETERS:
    obst_coordinates: [n_obst, m_vertices, p_dimensions]-narray
    setting: resolution in different dimensions, i.e. xy, theta, ob_xy
    vehicle_radius: minimum turning radius
    RETURN:
        obst_x: not sampled obstacle in x
        obst_y: not sampled obstacle in y
        sampled_obst_map_ind: binary obstacle map index
        sampled_obst_KDTree: later use for collision detection
                             shape: [n_x, n_y]
        map_info[instance]: contains the boundaries of the map
                            and resolution information
    '''
    obst_x = []
    obst_y = []
    map_info = map_boundary(obst_coordinates, setting)

    for i in range(obst_coordinates.shape[0]):
        for j in range(obst_coordinates.shape[1]):
            next_ind = j + 1 if j != obst_coordinates.shape[1] - 1 else 0
            tmp_coor = np.array([obst_coordinates[i, j, :],
                                 obst_coordinates[i, next_ind, :]])
            n = abs(tmp_coor[0, :] - tmp_coor[1, :]) / 0.1
            obst_x.extend(np.linspace(tmp_coor[0, 0], tmp_coor[1, 0], sum(n)))
            obst_y.extend(np.linspace(tmp_coor[0, 1], tmp_coor[1, 1], sum(n)))

    obst_x = np.array(obst_x)
```

```
    obst_y = np.array(obst_y)
    obst_KDTree = KDTree(np.vstack((obst_x, obst_y)).T)
    sampled_obst_x = np.array(
        [i / map_info.obst_reso for i in obst_x])
    sampled_obst_y = np.array(
        [i / map_info.obst_reso for i in obst_y])

    sampled_obst_map_ind = np.zeros(
        (map_info.sampled_obst_x_width,
            map_info.sampled_obst_y_width), dtype=np.int64)

    # [n_samples, n_dimensions]
    sampled_obst_KDTree = KDTree(np.vstack((sampled_obst_x, sampled_obst_y)).T)
    for i in range(map_info.sampled_obst_x_width):
        x = i + map_info.min_sampled_obst_x
        for j in range(map_info.sampled_obst_y_width):
            y = j + map_info.min_sampled_obst_y
            dist, ind = sampled_obst_KDTree.query([[x, y]], k=1)
            if dist <= vehicle_radius / map_info.obst_reso:
                sampled_obst_map_ind[i, j] = 1

    return obst_x, obst_y, obst_KDTree, sampled_obst_map_ind,\
        sampled_obst_KDTree, map_info


if __name__ == '__main__':
    obst_coordinates = np.array([[[-20, 5], [-1.3, 5], [-1.3, -5], [-20, -5]],
                                 [[1.3, 5], [20, 5], [20, -5], [1.3, -5]],
                                 [[-20, 15], [20, 15], [20, 11], [-20, 11]]])
    setting = {
        'coordinate_resolution': 0.3,        # m
        'motion_resolution': 0.1,            # m
        'obstacle_resolution': 0.1,          # m
        'yaw_resolution': np.pi / 36,        # rad(= 5 deg)
    }
    obst_x, obst_y, sampled_obst_map_ind, sampled_obst_KDTree, map_info = \
        calc_obst_map(obst_coordinates, setting, 1)
    plt.figure()
    plt.scatter(obst_x, obst_y)
    plt.show()
    plt.imshow(sampled_obst_map_ind.T[::-1, :])
    plt.show()
    print(map_info)
```

0.4 hybird_a_star.py

In [ ]:
```
import numpy as np
from queue import PriorityQueue as PQ
```

```python
import itertools
from collision_check import is_collision
import a_star


class Node():
    '''
    PARAMETERS:
    move: forward[1]/backward[0]
    state:
        x: x pisition
        y: y position
        theta: the yaw angle
        shape: [3, n_state]
    steer: the steer angle
    cost: select nodes based on this
    prevNode_ind: Parent node index
    '''

    def __init__(self, move, state, steer_f, steer_r, cost, prevNode_ind):
        self.move = move
        self.state = state
        self.steer_f = steer_f
        self.steer_r = steer_r
        self.cost = cost
        self.prevNode_ind = prevNode_ind


def calc_path(start_point, end_point, obstacle, global_config):
    '''
    calculate the initial path by Hybird A*
    PARAMETERS:
    start_point[narray]: [x, y, theta] - [x position, y position, steer angle]
    end_point[narray]: [x, y, theta]
    obstacle[dict]: include all obstacle information
        obst_x: the position of x
        obst_y: the position of y
        sampled_obst_map_ind: sampled obstacle index map
        sampled_obst_KDTree: KDTree generated based on sampled obstacle
        map_info: includes information such as map size and resolution of
                  each dimension(xy coor, yaw, obstacle)
    global_config: global setting parameters

    RETURN:
    '''

    if len(start_point) != 3 or len(end_point) != 3:
        print('The starting point or ending point must be three-dimensional')
```

```python
        return None, None, None

    start_point, end_point = map(convert2pi, [start_point, end_point])

    map_info = obstacle['map_info']

    start_node = Node(1, start_point, 0.0, 0.0, 0.0, -1)
    end_node = Node(1, end_point, 0.0, 0.0, 0.0, -2)
    # TODO: design heuristic functions to calculate cost to go
    # temporarily using A* to calculate the distance to the end point as a
    # heuristic
    # dist_heuristic_map: consider the obstacle, the distance from the
    #           point to the end point on the map
    heuristic_cost = global_config.cost_weights['heuristic_cost']
    distance_heuristic_map = a_star.get_dist_map(
        end_node, obstacle['obst_x'], obstacle['obst_y'],
        map_info, global_config.parameters_vehicle)

    open_dict, close_dict = {}, {}
    open_dict[calc_index(start_node, map_info)] = start_node

    # used to store the cost of each node
    pq = PQ()
    pq.put((calc_cost(start_node, distance_heuristic_map,
                      map_info, heuristic_cost),
            calc_index(start_node, map_info)))

    # control inputs space: includes all possible input combinations
    max_steer = global_config.parameters_vehicle['max_steering_angle']
    num_steer = global_config.setting['num_steer']
    delta_space = np.arange(-max_steer, max_steer +
                            max_steer / num_steer, max_steer / num_steer)
    direction_space = [0, 1]
    inputs_space = np.array(list(itertools.product(
        delta_space, delta_space, direction_space)))

    while True:
        if not len(open_dict):
            print('ERROR: cannot find path, no elements in the open dict')
            return None, None, None

        cur_node_ind = pq.get()[1]
        cur_node = open_dict[cur_node_ind]

        # TODO: when the current point is near the end point,
        #       use the reeds shepp curve to approach the end point.
        flag, cur_node = is_reach(cur_node, end_point, obstacle)
        if flag:
```

```
                close_dict[calc_index(end_node, map_info)] = cur_node
                break

            del open_dict[cur_node_ind]
            close_dict[cur_node_ind] = cur_node

            for i in range(len(inputs_space)):
                # get the next node
                next_node = get_next_node(cur_node_ind, cur_node, inputs_space[
                                          i, :], map_info, global_config)

                # check if the next node meets the constraint,
                # otherwise exit the loop
                if not is_feasible(next_node, obstacle,
                                   global_config.parameters_vehicle):
                    continue

                next_node_ind = calc_index(next_node, map_info)
                if close_dict.get(next_node_ind):
                    continue

                if not open_dict.get(next_node_ind):
                    open_dict[next_node_ind] = next_node
                    pq.put((calc_cost(next_node, distance_heuristic_map,
                                      map_info, heuristic_cost), next_node_ind))

    # get the init path
    init_path = get_init_path(end_node, close_dict, map_info)

    return init_path


def convert2pi(point):
    '''
    Make sure the steering angle is between [-pi, pi]
    '''
    point[2] %= 2 * np.pi if point[2] >= 0 else -2 * np.pi
    if point[2] > np.pi:
        point[2] -= 2 * np.pi
    elif point[2] < - np.pi:
        point[2] += 2 * np.pi

    return point


def calc_index(Node, map_info):
    '''
    Get the index of the current state in the state space
```

```python
    '''
    x_ind = np.round(Node.state[0, -1] /
                     map_info.xy_reso) - map_info.min_sampled_x
    y_ind = (np.round(Node.state[1, -1] / map_info.xy_reso) -
             map_info.min_sampled_y) * map_info.sampled_x_width
    theta_ind = (np.round(Node.state[2, -1] / map_info.xy_reso) -
                 map_info.min_sampled_theta) * map_info.sampled_x_width * \
        map_info.sampled_y_width
    ind = (theta_ind + y_ind + x_ind).astype(np.int64)

    return ind


def get_next_node(cur_node_ind, cur_node, control_input,
                  map_info, global_config):
    '''
    Get the next node based on the control input(steer_f, steer_r, move)
    '''

    steer_f, steer_r, move = control_input
    arc_length = map_info.xy_reso
    num_segment = int(np.round(arc_length / map_info.motion_reso) + 1)
    next_state_segment = np.zeros((len(cur_node.state), num_segment))

    for i in range(num_segment):
        prev_segment = next_state_segment[
            :, i - 1] if i else cur_node.state[:, -1]

        # TODO: design a car model that fits four-wheel steering.
        #       at this stage, a simplified model is used to approximate
        #       the four-wheel steering model.
        increments = move * map_info.motion_reso * \
            np.array([[np.cos(prev_segment[2])], [np.sin(prev_segment[2])],
                      [(np.tan(steer_f) + np.tan(steer_r)) /
                       global_config.parameters_vehicle['wheel_base']]])

        next_state_segment[:, i] = convert2pi(prev_segment + increments)

    # TODO: try to gain better weight through apprenticeship learning
    # calculate the cost so far
    cost_so_far = 0
    cost_so_far += (move > 0) * abs(arc_length) + \
        (move < 0) * abs(arc_length) * global_config.cost_weights['back_cost']
    cost_so_far += (cur_node.move != move) * \
        global_config.cost_weights['switch_cost']
    cost_so_far += (abs(steer_f) + abs(steer_r)) * \
        global_config.cost_weights['steer_cost']
    cost_so_far += (abs(cur_node.steer_f - steer_f) +
```

```python
                abs(cur_node.steer_r - steer_r)) * \
            global_config.cost_weights['change_steer_cost']

    next_node = Node(move, next_state_segment, steer_f, steer_r,
                     cur_node.cost + cost_so_far, cur_node_ind)

    return next_node


def is_feasible(node, obstacle, parameters_vehicle):
    '''
    Check if the node meets the constraint(no collision etc.)
    '''

    # check if overflow the map boundary
    map_info = obstacle['map_info']
    reso = np.array([1 / map_info.xy_reso, 1 /
                     map_info.xy_reso, 1 / map_info.yaw_reso])
    min_boundary = np.array(
        [map_info.min_sampled_x, map_info.min_sampled_y,
         map_info.min_sampled_theta]).reshape(3, 1)
    max_boundary = np.array(
        [map_info.max_sampled_x, map_info.max_sampled_y,
         map_info.max_sampled_theta]).reshape(3, 1)
    # shape: (3,)
    sampled_state = node.state[:, -1] * reso
    is_overflow = ((sampled_state - min_boundary) < 0).sum() + \
        ((sampled_state - max_boundary) > 0).sum()

    if is_overflow:
        return False

    # cheack if collision
    if is_collision(node.state, obstacle['obst_x'],
                    obstacle['obst_y'], obstacle['obst_KDTree'],
                    parameters_vehicle):
        return False

    return True


def is_reach(node, end_point, obstacle):
    '''
    Check if the current node is close to or reach the end point
    '''
    # TODO: complete function
    # whether to reach the end
    flag = False
```

```
    # TODO
    pass
    return flag, node


def get_init_path(end_node, close_dict, map_info):
    '''
    Reverse traversing close_dict to get init path
    '''
    # complete function
    init_path = end_node.state
    prev_ind = calc_index(end_node, map_info)
    while True:
        prev_node = close_dict[prev_ind]
        init_path = np.hstack((prev_node, init_path))
        prev_ind = prev_node.prevNode_ind

        if prev_ind == -1:
            break

    return init_path


# TODO: using A* to calculate the distance to the end point as a heuristic
# def distance_heuristic(end_node, obst_x, obst_y,
#                        map_info, parameters_vehicle):
#     '''
#     Consider the obstacles, and obtain the distance from each point
#     on the map to the end point by the A* algorithm.
#     '''
#     return a_star.get_dist_map(end_node, obst_x, obst_y,
#                                map_info, parameters_vehicle)

# TODO: design new heuristics distance
def calc_cost(node, distance_heuristic_map, map_info, heuristic_cost):
    return node.cost + heuristic_cost * distance_heuristic_map[
        node.state[0, -1] / map_info.xy_reso - map_info.min_sampled_x,
        node.state[1, -1] / map_info.xy_reso - map_info.min_sampled_y]


if __name__ == '__main__':
    calc_path(np.array([1, 2, 1.5 * np.pi]), np.array([10, 10, -1.5 * np.pi]))
```

0.5 collision_check.py

```
In [ ]: import numpy as np
        from sklearn.neighbors import KDTree
```

```python
def is_collision(state_mat, obst_x, obst_y, obst_KDTree,
                 parameters_vehicle):
    '''
    Check if the current node has collided
    PARAMETERS:
        state: current node's state, shape: [3, n_state]
        obst_x: position of x
        obst_y: position of y
        vehicle_parameters: includes all informations about the vehicle

    RETURN:
        bool: True-collision, False-no collision
    '''

    # here using hierarchical method for collision detection
    # the first step is: a rough test with a circle of
    #                    [center=(cent_x, cent_y), d=body_length]
    # the second step is: to detect the collision with the rectangle
    #                     close to the body.
    body_L = parameters_vehicle['body_length']
    rear2back = parameters_vehicle['rear2back']
    body_width = parameters_vehicle['body_width']
    dist2cent = body_L / 2.0 - rear2back
    R = body_L / 2.0
    # the coordinates of each vertex of the rectangle
    # when the center of the axis is the origin
    rectangle_car = np.array([[body_L - rear2back, - body_width / 2.0],
                              [body_L - rear2back, body_width / 2.0],
                              [- rear2back, body_width / 2.0],
                              [- rear2back, - body_width / 2.0]])

    for i in range(state_mat.shape[1]):
        x, y, theta = state_mat[:, i]
        cent_x = x + dist2cent * np.cos(theta)
        cent_y = y + dist2cent * np.sin(theta)

        # rough detection by circle
        ids = obst_KDTree.query_radius([[cent_x, cent_y]], R)
        if not ids[0].size:
            continue

        # fine detection with a rectangle
        #   use the outer product to judge whether the
        #   obstacle point is inside the car rectangle
        for ob_x, ob_y in zip(obst_x[ids[0]], obst_y[ids[0]]):
            # use the center of the rear axle as the coordinate origin
            rotate_mat = np.array([[np.cos(theta), -np.sin(theta)],
```

```python
                                [np.sin(theta), np.cos(theta)]])
            rotate_ob_x, rotate_ob_y = np.dot(
                rotate_mat, np.array([[ob_x - x], [ob_y - y]]))

            # the sign of the outer product, 1 is positive and -1 is negative
            sign = []
            for j in range(len(rectangle_car)):
                next_j = j + 1 if j < len(rectangle_car) - 1 else 0

                cross = (rectangle_car[j, 0] - rotate_ob_x) * (
                    rectangle_car[next_j, 1] - rotate_ob_y) - (
                    rectangle_car[next_j, 0] - rotate_ob_x) * (
                    rectangle_car[j, 1] - rotate_ob_y)

                if cross == 0:
                    continue
                if cross > 0:
                    sign.append(1)
                elif cross < 0:
                    sign.append(-1)

            if len(set(sign)) == 1:
                return True

    return False


if __name__ == '__main__':
    state_mat = np.array([[5.0, 10.0], [5.0, 10.0], [10 * np.pi / 180.0, 0.0]])
    obst_x = np.random.randn(5) + 10.0
    obst_y = np.random.randn(5) + 10.0
    obst_KDTree = KDTree(np.vstack((obst_x, obst_y)).T)

    parameters_vehicle = {
        'body_length': 4.7,                 # m
        'rear2back': 1.0,                   # m
        'rear2front': 3.7,                  # m
        'body_width': 2,                    # m
        'wheel_base': 2.7,                  # m
        'max_steering angle': 0.6,          # rad
        'minimum turning radius': 1         # m
    }

    flag = is_collision(state_mat, obst_x, obst_y,
                        obst_KDTree, parameters_vehicle)

    if flag:
        print('Collision!!!')
```

```
        else:
            print('free!!!')
```

0.6 a_star.py

In [ ]:
```
import numpy as np
from sklearn.neighbors import KDTree
from queue import PriorityQueue as PQ


class Node2d():

    def __init__(self, position, cost, prev_ind):
        self.position = position
        self.cost = cost
        self.prev_ind = prev_ind



def get_index(state, map_info):
    return (state[1, -1] / map_info.xy_reso - map_info.min_sampled_y) * \
        map_info.sampled_x_width + state[0, -1] / \
        map_info.xy_reso - map_info.min_sampled_x


def get_dist_map(end_node, obst_x, obst_y, map_info, parameters_vehicle):
    '''
    Use the A* algorithm to get the distance from the end point
    to any point in the map at one time.
    '''
    end_node = Node2d(np.round(end_node.state[:1, :] / map_info.xy_reso),
                      end_node.cost, end_node.prevNode_ind)
    end_node_ind = get_index(end_node.state, map_info)
    # sample based on xy-coordinate resolution
    obst_x = [x / map_info.xy_reso for x in obst_x]
    obst_y = [y / map_info.xy_reso for y in obst_y]

    # get the obstacle index map
    obst_map_ind = np.zeros(
        (map_info.sampled_x_width, map_info.smapled_y_width))
    sampled_obst_kdtree = KDTree(np.vstack((obst_x, obst_y)).T)
    for ix in range(map_info.sampled_x_width):
        x = ix + map_info.min_sampled_x
        for iy in range(map_info.smapled_y_width):
            y = iy + map_info.min_sampled_y
            dist, ind = sampled_obst_kdtree.query([[x, y]], k=1)
            if dist <= parameters_vehicle['minimum_turning_radius']:
                obst_map_ind[ix, iy] = 1
```

```python
open_dict, close_dict = {}, {}
open_dict[end_node_ind] = end_node

#                   dx  dy cost
motion = np.array([[-1, 0, 1],
                   [1, 0, 1],
                   [0, -1, 1],
                   [0, 1, 1],
                   [-1, -1, np.sqrt(2)],
                   [-1, 1, np.sqrt(2)],
                   [1, -1, np.sqrt(2)],
                   [1, 1, np.sqrt(2)]])
n_motion = len(motion)
pq = PQ()
pq.put((end_node.cost, end_node_ind))

while True:

    if not len(open_dict):
        # finish search
        print('Done!')
        break

    cur_node_ind = pq.get()[1]
    cur_node = open_dict[cur_node_ind]
    del open_dict[cur_node_ind]
    close_dict[cur_node_ind] = cur_node

    for i in range(n_motion):
        next_node = Node2d(cur_node.state + np.array([motion[i, :2]]).T,
                           cur_node.cost + motion[i, 2], cur_node_ind)

        # overflow and collision detection
        min_boundary = np.array(
            [map_info.min_sampled_x, map_info.min_sampled_y]).reshape(2, 1)
        max_boundary = np.array([map_info.max_sampled_x,
                                 obst_map_ind.max_sampled_y]).reshape(2, 1)
        if ((next_node.state - min_boundary) < 0).sum():
            continue
        if ((next_node.state - max_boundary) > 0).sum():
            continue

        # collision detection
        ix = next_node.state[0, -1] - map_info.min_sampled_x
        iy = next_node.state[1, -1] - map_info.min_sampled_y
        if obst_map_ind[ix, iy]:
            continue
```

```
            next_node_ind = get_index(next_node.state, map_info)
            if close_dict.get(next_node_ind):
                continue
            if open_dict.get(next_node_ind):
                if open_dict[next_node_ind].cost > next_node.cost:
                    open_dict[next_node_ind].cost = next_node.cost
                    open_dict[next_node_ind].prev_ind = next_node.prev_ind
            else:
                open_dict[next_node_ind] = next_node
                pq.put((next_node.cost, next_node_ind))

    # get distance map based on the end point
    dist_map = np.full(
        (map_info.sampled_x_width, map_info.smapled_y_width), np.inf)
    for node in close_dict.values():
        dist_map[node.state[0, -1] - map_info.min_sampled_x,
                 node.state[1, -1] - map_info.min_sampled_y] = node.cost

    return dist_map
```

In [ ]: