# 电子系统导论实验报告

## 实验 0 【Python 编程作业】


指导教师： 王海鹏

学生姓名： 周训哲

学　　号： 20307110315

专　　业： 计算机科学与技术

日　　期： 2024.3.26

# 一、 实验目的：

本实验旨在探索自动导航小车的路径规划算法。如图 1 所示，两墙距离 2 m，根据以下条件实现模拟小车（可以用矩形表示，长宽自设）在两面墙中间运动（实线位置）的运动轨迹程序：

1. 两个障碍（形状颜色大小任意，但不能用点表示），随机放置，但都距离墙≥50cm;
2. 小车在沿两个障碍（车与障碍不能接触）画出一个类似 8 字的形状，方向如图所示

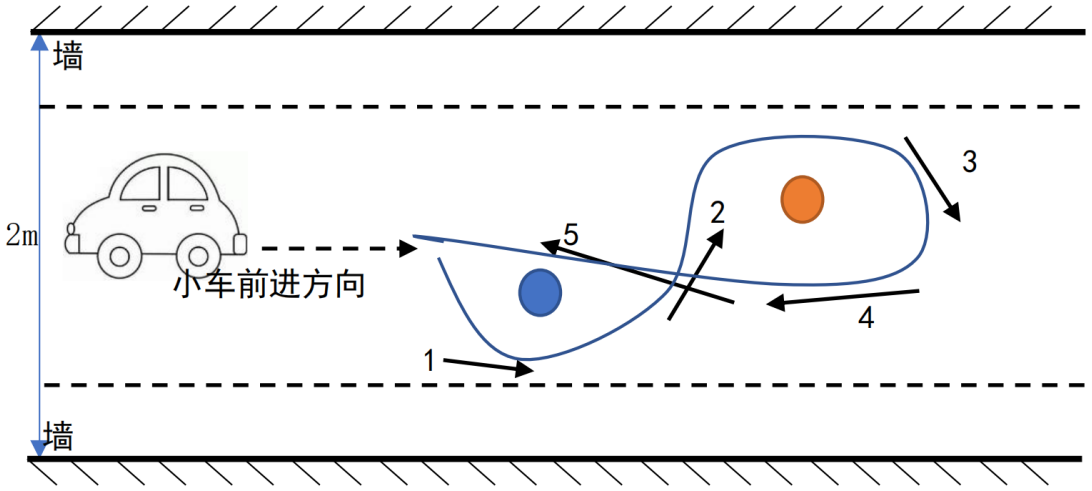



图 1. 小车模拟运动示意图

# 二、 实验原理：

实验基于计算机几何学与动态路径规划理论，采用Python编程语言和matplotlib库进行小车移动的模拟与可视化。小车的路径规划算法需要综合考虑障碍物的位置与尺寸，同时满足避免碰撞与路径最短化的双重约束：在本实验中，我采用了两种方法实现了目标绕桩的路径规划，并从数学计算出发，得到了路程最优的以两障碍物内公切线为轨迹的运动策略。

# 三、 实验内容：

## 3.1 场景构建

场景构建的初衷是创建一个规定边界内的小车行驶环境。通过设定上下边界与障碍物，确立了小车的运动场地。利用 matplotlib 的绘图功能，我使用 ax.plot()函数模拟出了场景的上下两面墙和障碍物的上下边界；

```
1.  # add the walls
2.  ax.plot([left_boundary, right_boundary], [wall_dist, wall_dist], 'k-
    ')  # Top wall
3.  ax.plot([left_boundary, right_boundary], [0, 0], 'k-')  # Bottom wall
4.  ax.plot([left_boundary, right_boundary], [wall_dist - dist_from_wall, wall_d
    ist - dist_from_wall], 'k--')  # Top dash
5.  ax.plot([left_boundary, right_boundary], [dist_from_wall, dist_from_wall], '
    k--')  # Bottom dash
```

使用 ax.add_patch()函数生成对应的矩形小车以及障碍物，我在实验中构建了一个红色的矩形小车和两个颜色为橙色和蓝色的障碍物（颜色也可以随机，考虑到演示效果，这里我限制其颜色较为醒目），形状随机为圆形，方形（可能有三种组合，也可以构造其他形状，由于考虑到仅作为演示便只实现了两种形状），坐标随机（为了方便计算，限制坐标必定为一左一右，详见源代码），大小随机（限制了最低和最高的半径或周长）；

```
# add obstacles
for obstacle in obstacles:
    ax.add_patch(obstacle)
# add the car
car = patches.Rectangle((car_pos[0], car_pos[1]), car_length, car_width, color='red', angle=0)
ax.add_patch(car)
```

使用 update()函数更新每一时刻小车的位置，并用 FuncAnimation()函数模拟小车的运动轨迹的动画。

```
# plot the animation
generate_path()
ani = FuncAnimation(fig, update, interval=40)
```

## 3.2 小车与障碍物模型

小车模型采用矩形形状，并以鲜明的红色进行标记，以便于观察其在模拟环境中的动态位置变化。障碍物则随机生成为圆形或方形，并以橙色和蓝色进行区分，确保视觉上的辨识度。模型参数如下：

```
wall_dist = 200
dist_from_wall = 50
car_length = 30
# must guarantee to go through obstacles
car_width = 20
left_boundary = -100
right_boundary = 250
margin = 5
r1 = random.uniform(5, 20)
r2 = random.uniform(5, 20)
```

## 3.3 轨迹规划与可视化

使用 matplotlib 的动画功能，通过连续帧的更新显示小车在规避障碍物时的轨迹变化。

动画展示了小车在每个时间段的确切位置，并以此来验证路径规划算法的有效性。在实现 update()函数过程前，首先考虑了如何使得小车能够行进到目标点的坐标，并将其可视化，最终的方案确定为使用一系列离散的坐标点进行表示，用 update()函数更新小车的状态。

```
def update(frame):
    global car_pos, targets, first_round
    speed = 2

    if (car_pos[0], car_pos[1]) == targets[0]:
        target = targets.pop(0)
        first_round += 1
        if first_round >= delete_line:
            targets.append(target)
    car_pos, car_angle = toward_target(car_pos, targets[0], speed)
    car.set_x(car_pos[0])
    car.set_y(car_pos[1])
    car.angle = car_angle
```

关于移动的轨迹构造，由于障碍物的位置是随机的但是又有限制的，最开始我是打算将小车的轨迹设置为一个尽可能大的 8 字上（即两个足够大的圆），这样可以保证小车不碰撞的要求。后续考虑到期末实验的目标是完成一个 8 字绕桩的真实小车的路径规划，并且有任务完成时间的限制。考虑到应该如何将小车移动的路径尽可能的缩短，我便想到了当小车在两个障碍物中间的时候沿着当前位置到障碍物的切线移动。

在实现过程中，由于我使用了 update 函数，所以实际上需要获得每个下一时刻小车的坐标，和朝向。所以我的首要目标就是实现一个小车移动到目标点的函数 toward_target()，每单位时刻移动 speed 参数的距离，也即将小车坐标更新为下一时刻的目标点，同时也将小车朝向目标移动方向。

```
def toward_target(car_pos, target, speed):
    vec = np.array([target[0], target[1]]) - np.array(car_pos)
    direction_angle = np.arctan2(vec[1], vec[0])
    car_angle = np.degrees(direction_angle)
    if np.linalg.norm(vec) > speed:
        car_pos += speed * vec / np.linalg.norm(vec)
    else:
        car_pos = np.array([target[0], target[1]])
    return car_pos, car_angle
```

直线的移动十分简单，直接设置目标点即可。而对于绕障碍物旋转的坐标，我构造了 generate_circle_path()函数，按顺时针和逆时针角度顺序生成若干个指定半径的坐标点，并将坐标点存到一个列表中，进行返回。在实现圆周运动的过程中，考虑到角度不同带来的坐标点数量不同，我实现了一个细节，即将运动的角度作为参数限制生成坐标点的数量，使得人工目测小车的速度是较为恒定的。

```python
def generate_circle_path(start, end, center, radius, direction):
    vec_start = np.array(start) - np.array(center)
    vec_end = np.array(end) - np.array(center)
    angle_start = np.arctan2(vec_start[1], vec_start[0])
    angle_end = np.arctan2(vec_end[1], vec_end[0])

    if direction == 'clockwise':
        if angle_start < angle_end:
            angle_start += 2 * np.pi
        theta = np.linspace(angle_start, angle_end, int(abs(angle_end - angle_start) / (2 * np.pi) * 100))
    else:
        if angle_end < angle_start:
            angle_end += 2 * np.pi
        theta = np.linspace(angle_start, angle_end, int(abs(angle_end - angle_start) / (2 * np.pi) * 150))
    path_x = center[0] + radius * np.cos(theta)
    path_y = center[1] + radius * np.sin(theta)
    path = np.array([path_x[1:-1], path_y[1:-1]]).T
    return [tuple(p) for p in path]
```

## 3.4 路径规划

在 main_v1.py 实现中，小小车首先从初始点找到与左边（x 坐标小的）障碍物的切点，设置将下切点（y 坐标小的）作为移动的目标，让小车沿着切线移动到目标点。在到达下切点之后，对于左边障碍物一律采用逆时针旋转的方式，之后小车会沿障碍物移动到两障碍物中心的中点位置，记做 midpoint，与当前左障碍物的下切点的位置。之后小车又会沿直线运动到 midpoint 的位置，并从 midpoint 与右边障碍物上切点位置进入到绕右边障碍物旋转的轨迹中，顺时针旋转到 midpoint 与右障碍物下切点的位置。之后类比之前的方法，移动到 midpoint，之后又到左上切点（左障碍物上切点）的位置，逆时针旋转到左下切点。不断重复上述过程，便可生成一个形状类似 8 的轨迹。我构造了 v1 的 generate_path()函数用于生成对应的轨迹的目标点序列。

```python
def generate_path():
    global targets, first_round, delete_line
    global targets, first_round
    if t1[0] > t3[0]:
        targets.append(t3)
    else:
        targets.append(t1)
        targets.extend(generate_circle_path(t1, t3, (x1, y1), r1 + car_width + margin, 'counter'))
```

```
            delete_line = len(targets) + 1

    targets.extend([mid_point, t6])
    targets.extend(generate_circle_path(t6, t5, (x2, y2), r2 + margin, 'clockwise'))
    targets.extend([mid_point, t4])
    targets.extend(generate_circle_path(t4, t3, (x1, y1), r1 + car_width + margin, 'counter'))
```

关于生成切点的函数，我构造了一个 get_tangent_points()函数，按照上下顺序返回一个点到某个圆的切线的两个切点。

```
def get_tangent_points(current_pos, obstacle_center, obstacle_radius):
    # calculate distances
    x1, y1 = current_pos
    x2, y2 = obstacle_center
    d = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
    L = math.sqrt(d ** 2 - obstacle_radius ** 2)
    # calculate angles
    alpha = math.atan2((y2 - y1), (x2 - x1))
    theta = math.asin(obstacle_radius / d)
    # calculate tangent points
    t1 = (x1 + L * math.cos(alpha + theta), y1 + L * math.sin(alpha + theta))
    t2 = (x1 + L * math.cos(alpha - theta), y1 + L * math.sin(alpha - theta))
    # guarantee t1 always on the top of t2
    if t1[1] > t2[1]:
        t3 = t1
        t1 = t2
        t2 = t3
    return t1, t2
```

而在 main_v2 中，考虑到轨迹路径的长度问题，两圆之间的由于障碍物大小随机，所以两障碍物之间的内公切线不一定会经过两圆的中点。并且内公切线的距离会小于经过中点的行进距离。所以在 v2 版本中，我做出以下改进：将小车每次从两障碍物中间经过改成沿内公切线移动。理论上来说这样绕 8 字的**距离是最短**的。

```
def generate_path():
    global targets, first_round, delete_line
    if t1[0] > t3[0]:
        targets.append(t3)
    else:
        targets.append(t1)
```

```
            targets.extend(generate_circle_path(t1, t3, (x1, y1), r1 + car_width + margin, 'counter'))
            delete_line = len(targets) + 1

        targets.append(t6)
        targets.extend(generate_circle_path(t6, t5, (x2, y2), r2 + margin, 'clockwise'))
        targets.append(t4)
        targets.extend(generate_circle_path(t4, t3, (x1, y1), r1 + car_width + margin, 'counter'))
```

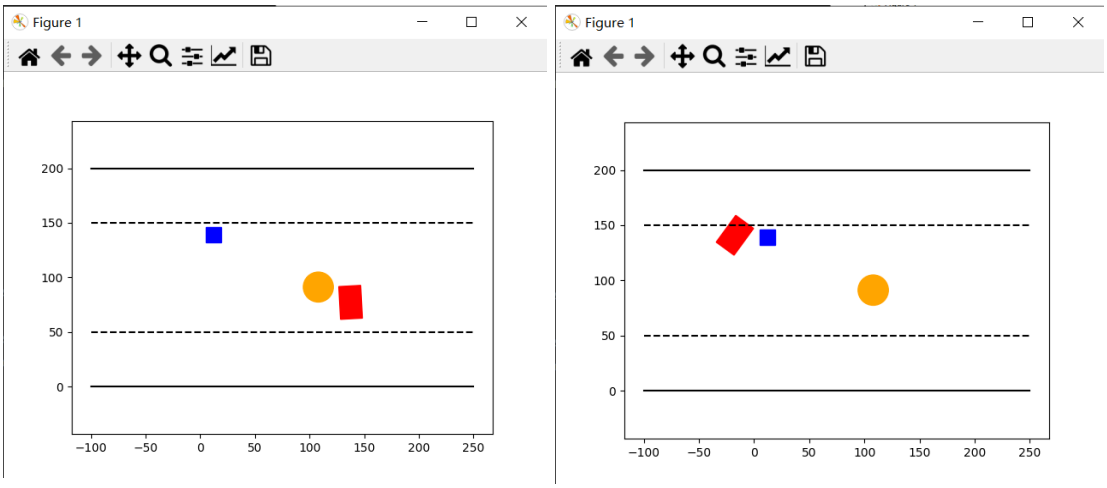
关于内公切线的切点的生成，我构造了如下函数，总共生成四个切点，首先生成左边障碍物的上下切点，然后生成右边障碍物的上下两个切点：

```
def get_internal_tangent_points(x1, y1, r1, x2, y2, r2):
    d = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
    theta = math.atan2(y2 - y1, x2 - x1)
    phi = math.acos((r1 + r2) / d)
    # left circle
    t1 = (x1 + r1 * math.cos(theta + phi), y1 + r1 * math.sin(theta + phi))

    t2 = (x1 + r1 * math.cos(theta - phi), y1 + r1 * math.sin(theta - phi))

    # right circle
    t3 = (x2 - r2 * math.cos(theta + phi), y2 - r2 * math.sin(theta + phi))

    t4 = (x2 - r2 * math.cos(theta - phi), y2 - r2 * math.sin(theta - phi))

    if t1[1] > t2[1]:
        tt = t1
        t1 = t2
        t2 = tt
    if t3[1] > t4[1]:
        tt = t3
        t3 = t4
        t4 = tt
    return t1, t2, t3, t4
```

# 四、 实验分析：

通过两种实验实现，我不仅成功地模拟了小车在含障碍环境中的运动，而且在 main_v2.py 版本中，通过优化行驶路径至内公切线，显著减少了行驶距离，证实了路径最

短化的理论假设。实验的效果展示如图：




# 五、 总结与思考：

## 5.1 实验过程中的问题及解决方案

**初始路径规划**：由于障碍物位置和大小随机，每次小车从初始点到开始绕 8 字的路径起始点的路径并不是固定的，本次实验中采用的方法是先让小车移动到左边障碍物的下切点，但是此方法并不能保证总的路径长度是最短的。

**路径方向调整**：最初路径的设计未能准确复现与实验要求相同的驾驶方向（顺逆时针的区别）。经过经助教提醒后调整顺逆时针参数后，路径得以正确模拟。

**避免碰撞**：一开始我将小车路径设置的完全贴合障碍物，以获得最短的运动路径。但是在助教提醒下，考虑到现实世界中，为防止小车在行驶过程中与障碍物接触，增加了一个边缘距离（margin），从而确保了小车在绕障碍物行驶时，总是保持一定的安全距离。

**不规则形状障碍物处理**：在处理方形障碍物时，我直接采取的办法是让小车绕方形障碍物的外切圆进行运动，此方法亦适用于处理其他非圆形障碍物，保证了行驶的连续性和安全性。本次实验中仅为了作为演示，只使用了方形和圆形两种形状。

## 5.2 思考与反思

在小车路径规划的实验过程中，我深刻体会到理论与实际应用之间的差异。虽然理论上最短路径的规划是清晰的，但在实际模拟中，往往需要考虑到更多的变量，如障碍物的形状、大小和随机位置。此外，考虑到真实世界中的动态环境因素，如障碍物可能的移动，需要更加复杂的算法来适应这些变化。

实验表明，优化算法在实现最短路径规划方面的重要性。main_v2.py 中沿内公切线移动的策略，显著提高了路径规划的效率，减少了小车行驶的时间。这为后续现实树莓派控制的小车自动导航的实际应用提供了有价值的参考。

本实验成功实现了小车在包含障碍物的环境中的路径规划和模拟。main_v1.py 和 main_v2.py 两个版本的实现，通过人眼直观感受证实了内公切线路径规划在确保安全的前提下有效缩短了小车的行驶距离。