轮式机器人技术与强化实践报告（Week 7-8）

3210105327 万晨阳

实验要求：

1. 在非 ROS 的框架下完成任意路径规划算法（Astar 或 RRT）的实现。
2. 在非 ROS 的框架下完成任意避障规划算法（DWA）的实现。
3. 在 ROS 的框架下完成小车的自主导航任务：任意给定目标点，小车能够自主规划路径，并跟随路径到达目标点。

实验原理：

1. 目标规划：路径规划算法 RRT

RRT 是 Steven M. LaValle 和 James J. Kuffner Jr.提出的一种通过随机构建 Space Filling Tree 实现对非凸高维空间快速搜索的算法。该算法可以很容易的处理包含障碍物和差分运动约束的场景，因而广泛的被应用在各种机器人的运动规划场景中。

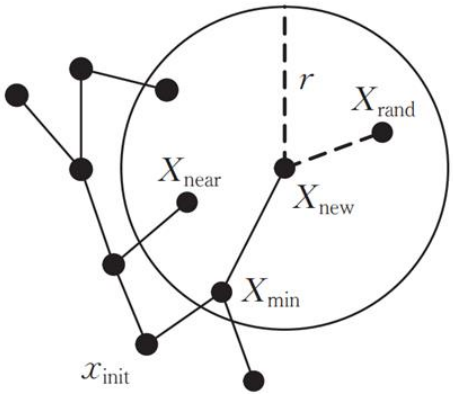
原始的 RRT 算法中将搜索的起点位置作为根节点，然后通过随机采样增加叶子节点的方式，生成一个随机扩展树，当随机树的叶子节点进入目标区域，就得到了从起点位置到目标位置的路径。

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  $E_i \leftarrow Edge(x_{new}, x_{near})$;
  if *CollisionFree*$(\mathcal{M}, E_i)$ **then**
    $\mathcal{T}.addNode(x_{new})$;
    $\mathcal{T}.addEdge(E_i)$;
  if $x_{new} = x_{goal}$ **then**
    Success();

```
RRT_BUILD(q_i,q_goal)
1       T.initial(q_i)
2       from k=1 to k=K
3           q_rand←RANDOM_STATE()
4           EXPAND(T,q_rand)
5       Give T
```

```
EXPAND(T,q)
1       q_near←NEAREST_NEIGHBOR(T,q);
2       if |q_near-q_rand|< ε then
3           q_new=q_rand
4       else
5           q_new=q_near+ ε
6       T.vertex_add(q_new);
7       T.edge_add(q_near,q_new);
8       if |q_goal-q_new|< μ then
9           Reached
10          Give Connectivity Path
11      else
12          Advanced
```



在基础的 RRT 算法上我们可以对其进行改进，通过概率控制使得某些时候树的生长是朝向目标点的（这样可以加快收敛速度）。基于概率的 RRT 算法在随机树的扩展的步骤中引入一个概率p，根据概率p的值来选择树的生长方向是随机生长还是朝向目标位置生长。引入向目

标生长的机制可以加速路径搜索的收敛速度。我在本次任务中实现的是基于概率的 RRT 算法。

2. 导航规划：避障算法 DWA

动态窗口法（Dynamic Window Approach， DWA）是一种避障规划方法，DWA 算法通过对速度空间施加约束以确保动力学模型和避障的要求，在速度空间中搜索机器人最优控制速度，最终实现快速安全地到达目的地。作者将速度空间约束在由机器人的平移速度和旋转速度(v,w)组成的二维速度搜索空间。

① 允许速度（Admissible velocities）

允许速度确保机器人可以在障碍前停下，最大的允许速度取决于当前轨迹距最近障碍的距离$dist(v,w)$，允许速度集V_a被定义为:

$$V_a = \{(v,w)|v \leq \sqrt{2 \cdot dist(v,w) \cdot \dot{v}_b} \wedge w \leq \sqrt{2 \cdot dist(v,w) \cdot \dot{w}_b}\}$$

下图就展示了某一场景下的速度空间。

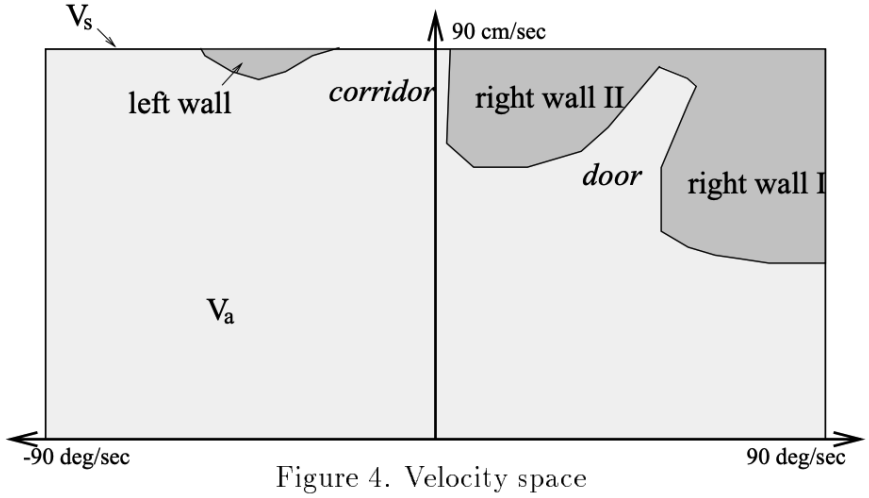


Figure 4. Velocity space

② 滑动窗口（Dynamic window）

考虑到机器人存在加速度限制，搜索空间被限定动态窗口中（在下一个规划间隔可达到的速度），具体如下:

$$V_d = \{(v,w)|v \in [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t] \wedge w \in [w_a - \dot{w} \cdot t, w_a + \dot{w} \cdot t]\}$$

其中$\dot{v}$和$\dot{w}$表示机器人的加速度。最终的速度搜索空间为$V_r = V_s \cap V_a \cap V_d$，如下图。

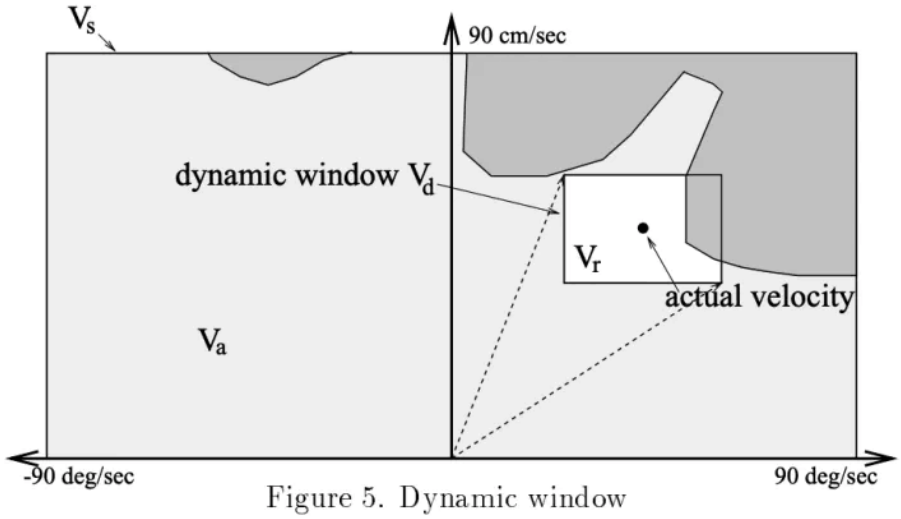


Figure 5. Dynamic window

③ 优化过程

目标函数考虑了方位角、安全距离和速度：

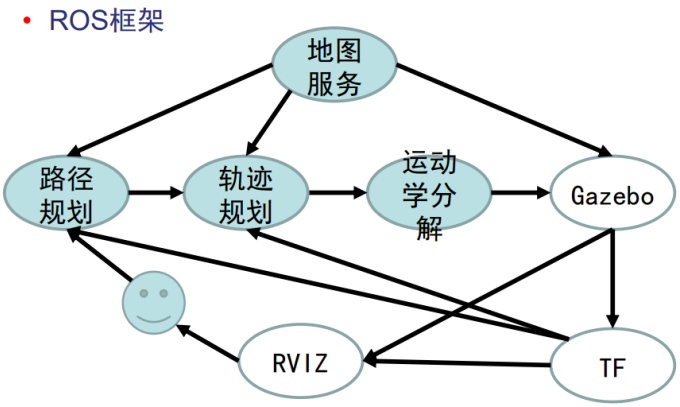
$$G(v,w)=\sigma(\alpha\cdot heading(v,w)+\beta\cdot dist(v,w)+\gamma\cdot velocity(v,w))$$

其中，方位角项$heading(v,w)$保证了机器人在运动过程中可以快速对准目标点。安全距离项$dist(v,w)$保证了机器人不会发生碰撞，速度项$velocity(v,w)$确保了尽快到达目标点。

参数α,β,γ可以根据需求调整。这三个指标是目标函数的重要组成部分，缺一不可。仅使$clearance$和$velocity$最大化，机器人始终在无障碍空间运动，但不会有向目标位置移动的趋势。单独最大化$heading$，机器人很快就会被阻碍其前进的第一个障碍所阻挡，无法在其周围移动。通过组合三个指标，机器人在上述限制条件下能够快速地绕过碰撞，同时朝着目标方向运动。

实验过程与实验结果：

任务的完成基于以下的 ROS 框架：



首先，在第五、六周的实验中我通过 python 实现了 RRT 与 DWA 的算法类，实现的基本如下：

- **对于 RRT 算法类**

类接受障碍点、起始点、终止点、地图图幅、网格长度以及 dist 阈值作为初始化需要的参数：

```
1. class RRT:
2.     def __init__(self, obstacles_point, start_point, end_point, p
   lanning_minx, planning_miny, planning_maxx, planning_maxy, grid,
   dist):
3. ……
```

而后在该类中中包含了名为 RRT_Tree 的子类，以完成 RRT 树的构建和生长；同时在这个算法类中包含了终点判断、寻找下一个点、最小距离判断、RRT 树生长的功能的实现。最终结构如下：

```
1. class RRT:
2.     def __init__(self, obstacles_point, start_point, end_point, p
   lanning_minx, planning_miny, planning_maxx, planning_maxy, grid,
   dist):
```

```
            ...
    class RRT_Tree:
        def __init__(self, point, father, child, endpoint, startpoint):
        ...
    def Process(self):
        ...
        return best_path_X, best_path_Y

    def search_next(self):
        ...
    def min_distance_node(self, temp_point):
        ...
    def if_near_destination(self, node):
        ...
```

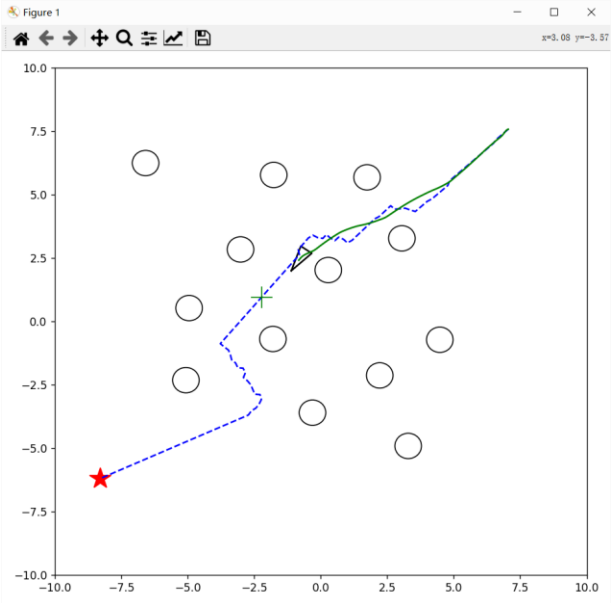
- **对于 DWA 算法类**

首先在 dwaconfig 中定义了进行 DWA 算法的各种所需参数。在 DWA 算法类中首先完成了对于三种代价（heading_cost， dist_cost， velocity_cost）的计算函数的实现，在规划（plan）部分对速度空间进行了遍历搜索，假想其在某一速度点下运动的情况进行了总代价的计算，然后不断比较，基于此完成了在速度空间中的规划：

```
class DWA:
    #计算方向角代价
    def heading_cost(self, from_x, from_y, to_x, to_y, robot_info):
        ...
        return np.abs(angleCost)+now_dist

    #计算距离代价
    def dist_cost(self,dwaconfig,robot_info, planning_obs, radius):
        ...
        return 1 / dist

    #计算速度代价
    def velocity_cost(self, now_v, max_v):
        ...
        return delta

    #计算运动
    def motion(self, robot_info, dwaconfig):
        ...
        return new_robot_info

```

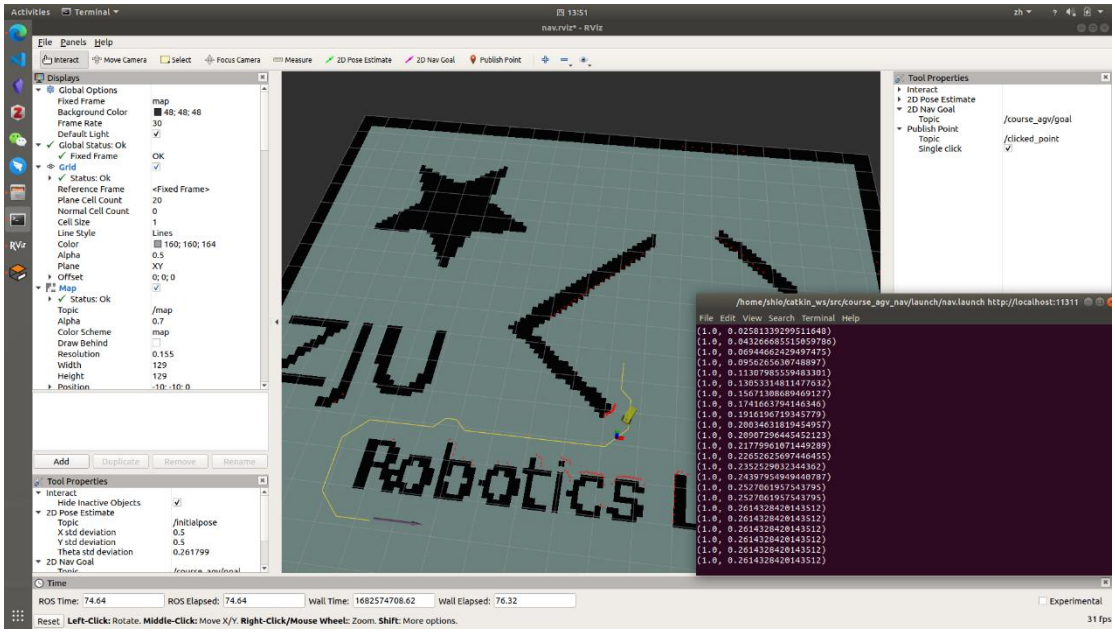
```
    #  robot info : [x, y, theta, vx, vw]
    #  dwaconfig : {robot_radius, obs_radius, dt, max_speed, min_speed, max_accel, v_reso, max_yawrate, max_dyawrate, yawrate_reso, predict_time, to_goal_cost_gain, speed_cost_gain, obstacle_cost_gain, tracking_dist, arrive_dist}

    #遍历搜索完成规划
    def plan(self, robot_info, dwaconfig, midpos, planning_obs):
        ...
        return nvx, nvw
```

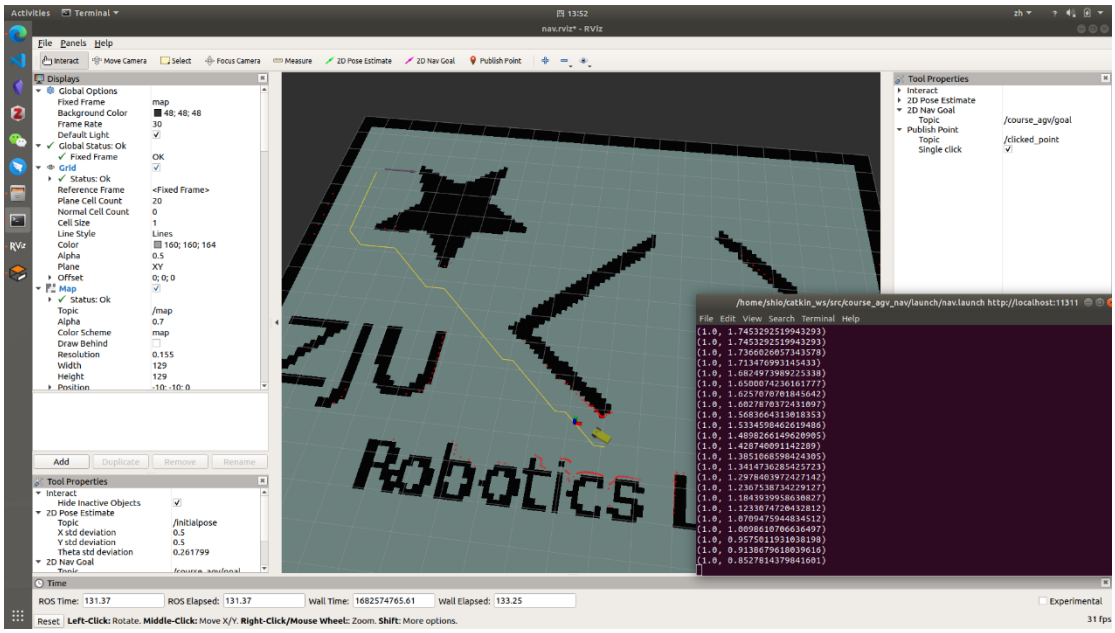
基于给定的非 ROS 框架，实现了使用 matplotlib 进行绘图模拟小车进行规划和运动。效果如下：



然后基于给定的 ROS 框架，我对代码进行了略微的修改以适合框架的接口，主要是把 dwaconfig 的内容集成到了 DWA 算法类中，并对两个算法类的返回内容进行了略微的修改。同时在完成指定的任务之后，我注意到小车在运动过程中总是发生翻车的情况，所以对于之前运动控制的代码部分我也进行了部分修改，使得速度解算得到的值相对小一点，避免翻车的发生。

rviz 中的最终实验结果如下：





实验总结：

通过完成第七、八周的任务，我了解了经典的路径规划和避障规划算法，如 PRM、RRT、VFH、DWA 等，并对 RRT 和 DWA 进行了具体的实现。通过完成小车的自主导航，我认识到了小车的实际运动是相当复杂的，不仅要考虑全局规划出来的理想运动路径，对于局部的物理运动限制也要有充分的考虑。因为后者是即时的，所以这一过程中很多时候我们对算法速度有比较高的要求。一开始由于参数原因算法运行的特别慢，后来经过调整，我的小车能够比较流畅地完成运动。

同时，我进一步熟悉了 Linux 系统的操作和使用，也进一步了解了 ROS 项目的建构和 debug 过程。