wheeledrobot 9-12

杨沛山　谢俊　邵可乐　万晨阳

June 2023

1　实验内容

本次实验我们需要在之前的实验基础上，将进行过仿真验证的路径规划算法移植到实物机器人上进行实验，使得实物机器人可以达到设置目标点自主导航前进的效果。

1.1　连接机器人

在用网线与机器人连接后，我们需要修改笔记本的 IP 地址，尝试使用 ssh 连接机器人。

修改机器人上的 `/etc/hosts` 文件，加入自己的笔记本的 IP 和机器名。同时也要修改自己笔记本上的 `/etc/hosts` 文件，加入机器人的 IP 和机器名。

修改自己和机器人的 `/.bashrc` 文件，设置 `ROS_MASTER_URI` 与 `ROS_IP` 环境变量。

最后在笔记本上查看机器人上的 topic 与 node，正确输出 topic，没有显示 error，说明我们组该部分实验成功。

1.2　建图 & 定位

在机器人上标定 IMU，转发激光雷达数据。由于我们使用的是新车，不需要打开 urg 节点。之后打开 tf 转发工具。

打开 `cartographer_occupancy_grid_node` 节点，之后在笔记本上打开网页，在网页上操作建图。此时可以控制小车扫描场地，尽可能覆盖全面。完成后先使用 `map_server` 保存地图，再在网页中保存后退出，并关闭 `cartographer_occupancy_grid_node` 节点。

启动 `map_server`，即可使用`/map` 这个 top 读取地图。

1.3　修改仿真代码并运行

由于仿真与实机使用时部分 topic 节点名字不同，因此我们需要对仿真代码进行一些修改。之后，在网站上完成手动定位，启动 `nav_real.launch`，在 RVIZ 中选定目标点，小车即可开始规划路径并前进。

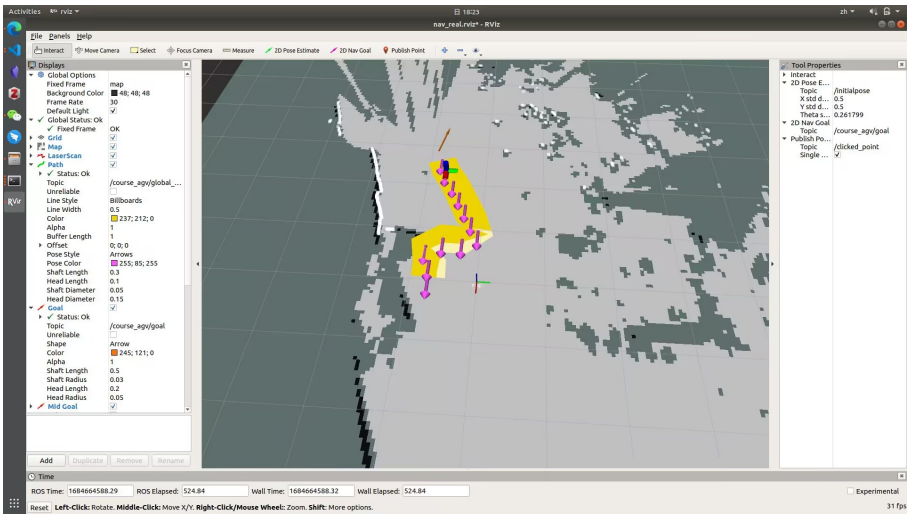


图 1: 选定目标点后显示 Global Planner 结果

2 程序框架和流程分析

启动 `nav_real.launch`，RVIZ 中会显示当前机器人的信息和地图信息，手动设置目标点后，规划开始。首先，执行 Global Planner，使用 RRT* 算法，规划出从起点至终点的全局路径。再根据全局路径上的点，作为 mid point，使用 Local Planner 的 DWA 算法实时规划出线速度与角速度。

我们采用的 RRT* 算法大致如下：

Algorithm 1: RRT* 算法

Data: feasiable region $\mathcal{M}$, initial state x_{init}, goal region x_{goal}, step size Δq, number of iterations K

Result: a path Γ from x_{init} to x_{goal}

Initialize $\mathcal{T}$;
Add x_{init} to $\mathcal{T}$;
for $k = 1$ *to* K **do**
    $x_{rand} \leftarrow \text{Sample}(\mathcal{M})$;
    $x_{nearest} \leftarrow \text{Nearest}(\mathcal{T}, x_{rand})$;
    $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand}, \Delta q)$;
    if *ObstacleFree*$(x_{nearest}, x_{new})$ **then**
        $\mathcal{T} \leftarrow \text{AddVertex}(\mathcal{T}, x_{new})$;
        $\mathcal{T} \leftarrow \text{AddEdge}(\mathcal{T}, x_{nearest}, x_{new})$;
        $X_{near} \leftarrow \text{Near}(\mathcal{T}, x_{new}, \Delta q)$;
        $dis(x_{new}) \leftarrow \infty$;
        foreach $x_{near} \in X_{near}$ **do**
            if *ObstacleFree*(x_{new}, x_{near}) *and* $dis(x_{new}) + dis(x_{near}) < dis(x_{near})$ **then**
                $\mathcal{T} \leftarrow \text{Rewire}(\mathcal{T}, x_{new}, x_{near})$;
                $dis(x_{near}) \leftarrow dis(x_{new}) + dis(x_{near})$;
            end
        end
    end
end
$\Gamma \leftarrow \text{ExtractPath}(\mathcal{T}, x_{init}, x_{goal})$;
return Γ;

DWA 算法过程大致如下：

首先，根据机器人当前的速度 (v_a, ω_a) 计算 $V_d = \{(v,\omega)|v \in [v_l, v_h] \wedge \omega \in [\omega_l, \omega_h]\}$，其中

$$\begin{cases} v_l = v_a - a_{max} \cdot \Delta t \\ v_h = v_a + a_{max} \cdot \Delta t \\ \omega_l = \omega_a - \alpha_{max} \cdot \Delta t \\ \omega_l = \omega_a + \alpha_{max} \cdot \Delta t \end{cases}$$

以及 $V_s = \{(v,\omega)|v \in [v_{min}, v_{max}] \wedge \omega \in [\omega_{min}, \omega_{max}]\}$。取窗口 $V_r = V_s \cap V_d$。之后，按照速度和角速度的分辨率大小遍历所有可能的速度 (v,ω)，并根据

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x + \frac{v}{\omega}(\sin(\theta + \omega\Delta t) - \sin\theta) \\ y + \frac{v}{\omega}(-\cos(\theta + \omega\Delta t) + \cos\theta) \\ \theta + \omega\Delta t \end{pmatrix}$$

来计算每个 (v,ω) 对应的新位置 (x', y', θ')。如果计算出的新位置落在可行区域内，即可根据

$evaluation(v,\omega) = \alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot velocity(v,\omega)$ 来计算其 $cost$，并选取最小的 $cost$ 的 (v,ω) 前进。

具体地，$heading(v,\omega)$ 为新的前进方向 θ' 与当前机器人位置 (x',y') 和目标点之间连线的夹角，$dist(v,\omega)$ 新的位置 (x',y') 与最近的障碍物之间的距离的倒数，$velocity(v,\omega)$ 为机器人的最大速度减去新速度，即 $v_{max} - v$。

除此之外，我们还需要根据给定的机器人参数设置代码中各项参数的值，例如底盘宽度 0.4m，轮子半径 0.09m，最大线速度 1m/s 等。

3 对于 RRT* 算法的改进

通过上述 RRT* 伪代码中不难看出, 其每生成一个新的点时是完全采用随机的落点, 而这种随机会对整个算法的效率造成极大的影响, 比如对于同一段路径, 也许几十次迭代就能生成, 但是在较劣的情况下可能会出现迭代成百上千个点都无法成功的情况.

由此, 我们将 RRT* 的代码加上了一个概率 P, 每次生成最新点的时候, 有 P 的概率是直接指向终点, 而 $1-P$ 的概率随机生成.

Algorithm 2: RRT* 算法生成点

Data: possibility P, current dot x_{now}, target dot x_{tar}, step $length$

Result: new dot x_{next}

if $Rand() \leq P$ **then**
  $x_{next} \leftarrow (x_{tar} - x_{now})/length$
else
  $x_{next} \leftarrow x_{Rand}$
end
return x_{next}

以下为添加改算法并通过修改概率 P 之后获得的算法结果.

注: 测试时采用同一张地图, 同一个起点终点

3.1 起点终点间有障碍物

以下的表格中,1-5 为 5 次测试,null 表示迭代次数超过了 max(此处我们设为 3000 次迭代).

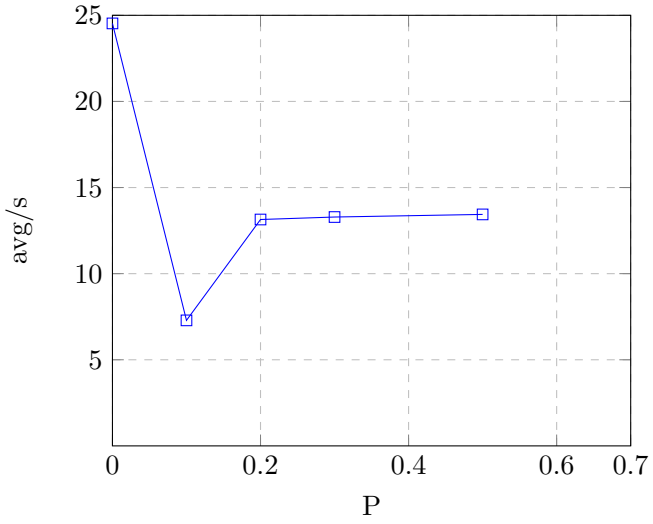
P	0	0.1	0.2	0.3	0.5	0.7
1/s	null	7.7	7.61	14.17	18.06	null
2/s	14.88	8.49	10.42	17.51	null	null
3/s	36.17	4.51	15.41	9.76	10.35	null
4/s	22.31	10.31	16.5	12.73	13.12	null
5/s	null	5.43	15.78	12.27	12.24	null
avg/s	24.45333333	7.288	13.144	13.288	13.4425	null

表 1: 起点终点间有障碍物

从表格中的时间体现 RRT* 的高随机性, 即使是完全相同的情况, 时间差距也非常大. 同时我们也不难看出在添加了 P 之后, 总体时间有着较为优秀的提升.

并且可以看到, 随着 P 的上升, 时间不一定是下降的, 因为此处我在起点与中间之间有一个直角障碍, 也就是说, 当 P 值过大时, 小车可能会在那个角里面无限循环, 达到迭代上限.

下图为 P 与 $average\_time$ 的折线图.



3.2 起点终点间无障碍物

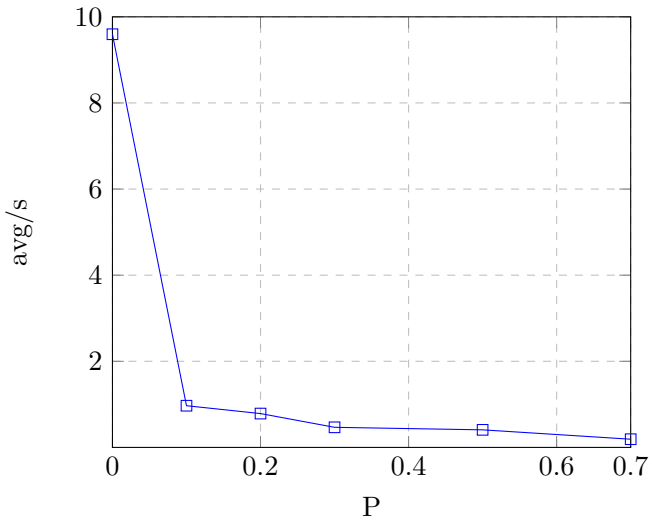
同时我们也测试了若起点与终点之间没有障碍物时, 其运行时间:

P	0	0.1	0.2	0.3	0.5	0.7
1/s	1.9	0.736	0.7	0.47	0.267	0.247
2/s	14.65	1.24	1.12	0.359	0.294	0.172
3/s	12.69	0.89	0.68	0.69	0.618	0.128
4/s	9.144	1.04	0.77	0.384	0.435	0.206
5/s	null	0.935	0.66	0.431	0.426	0.199
avg	9.596	0.9682	0.786	0.4668	0.408	0.1904

表 2: 起点终点间无障碍物

可以看到在这种情况下用时下降是十分稳定且明显的.

下图为 P 与 $average\_time$ 的折线图.



4 实验结果

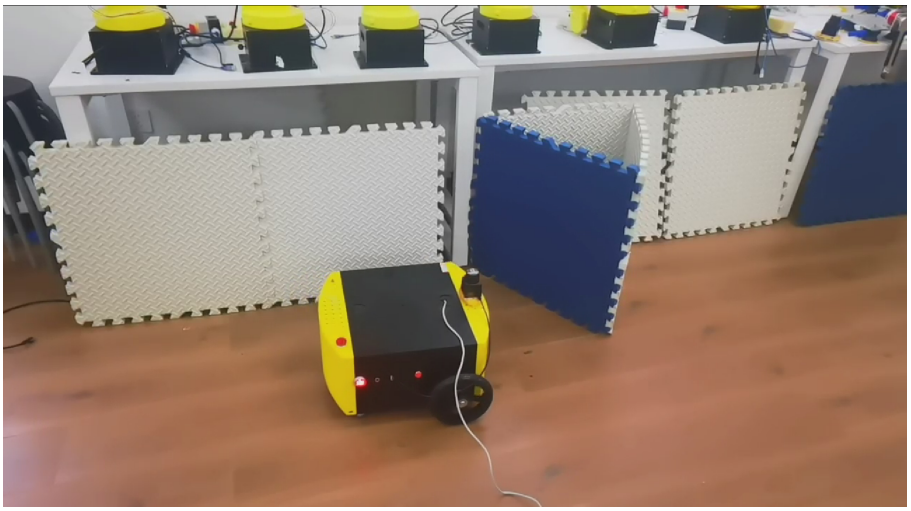


图 2: 小车正常前行

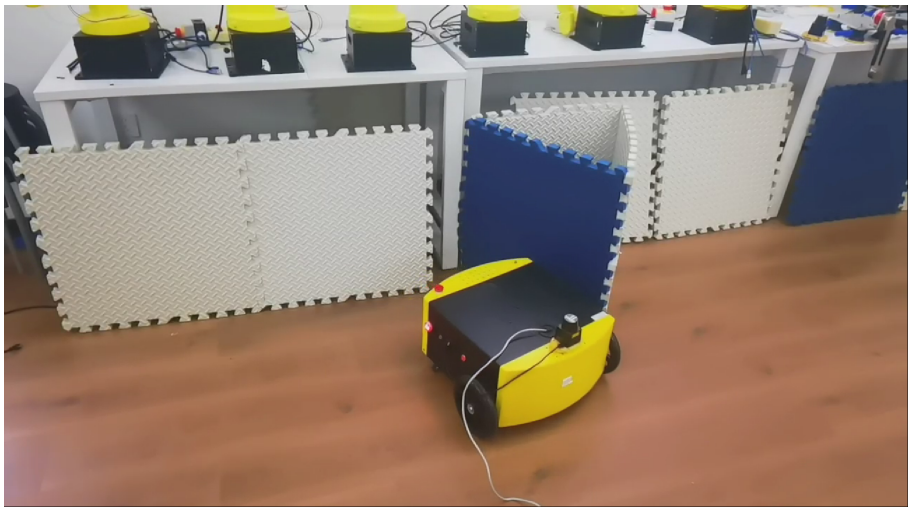


图 3: 小车主动避障

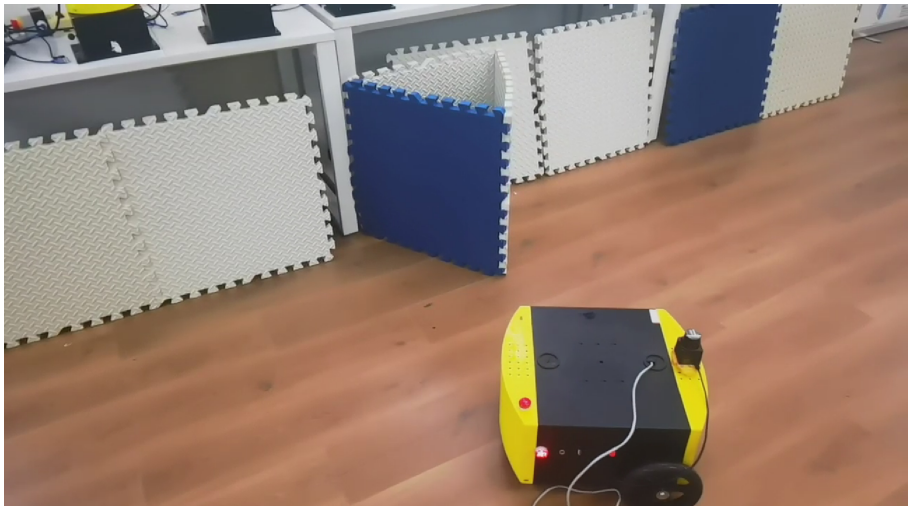


图 4: 小车绕过障碍后继续朝目标点前进