浙江大学本科生实验报告

课程名称:	机器人综合实践
姓名:	潘力豪
学号:	3210100385
指导教师:	周春琳
实验时间:	2024.7.7-7024.7.15

综合实验2 TB4小车路径规划

[TOC]

实验概述

了解Turtlebot4机器人的主要硬件构成及工作原理，使用上位机进行仿真和连接，运行ROS2系统进行自主导航规划和自主避障功能。

实验环境

Ubuntu22.04 (Jammy Jellyfish,Linux kernel=5.15) on x86_64 with RTX3060 Laptop

ROS2 Humble via fishROS with Navigation2

VSCode

Turtle Bot 4 教学移动机器人

实验原理

Turtlebot机器人基于树莓派平台开发，可以通过Wifi连接并使用ROS2平台控制，通过Rviz进行可视化工作。拥有较好的运动性能，可以实现原地转向和侧向直线行驶等功能。可以通过视觉信息和雷达进行SLAM建图，同时通过AMCL自定位方法可以实时根据局部信息校准在自身全局地图的位置，进而解决绑架问题等定位问题。

建图过程为增量式，根据设定的分辨率（单位/米）探索环境，输出定大小的全局代价地图（CostMap）和局部地图（Map）。地图以`nav_msgs::OccupancyGrid`格式发布为topic，可以通过订阅Topic获得或发布；或通过

```
costmap_ = costmap_ros->getCostmap();
mapWidth=costmap_->getSizeInCellsX();
unsigned int cost = static_cast<int>(costmap_->getCost(j, i));
```

获取。整张地图原点在左下角，以栅格化形式储存，可通过

```
index = (int)((x - origin_x) / resolution) + ((int)((y - origin_y) / resolution))
* width;
//x 世界坐标任一点系；origin_x 世界坐标系左下角点
```

计算地图索引，取值为[-1,100]的整数，0表示自由空间，100表示障碍物，-1表示位置，其他值表示有障碍物的概率。转化为图像格式储存后以0表示自由空间，255表示未知，254表示障碍物，其他值表示有障碍物的概率。整个规划算法基于自建的地图展开。

实验已经给了直线规划器的样例，也可以参考`nav2_Navfn`包的内容仿写相关函数

导航该过程需要解决如下问题

- 通过传感器获取机器人所在空间的位姿及所处环境信息

- 应用相关算法处理所获取的信息并建立环境模型
- 算寻找一条最优或近似最优的无碰撞路径，引导机器人安全到达目标点

仿真过程

配置环境的部分由于网络和设备硬件原因比较繁琐，具体放在附件中。我们建立了如下三张地图进行工作

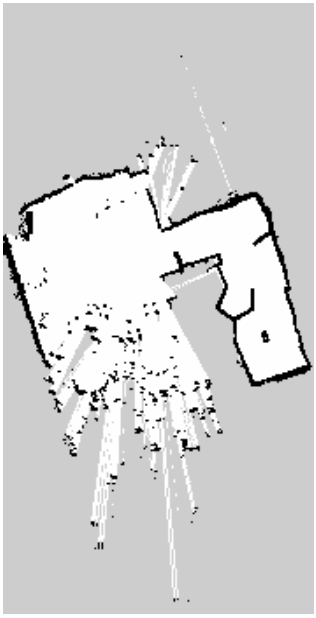	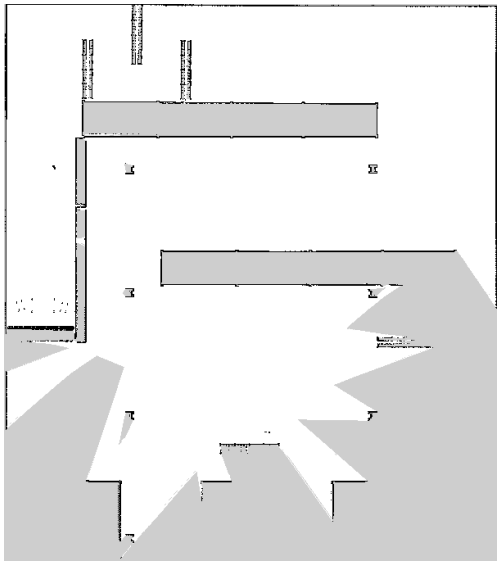	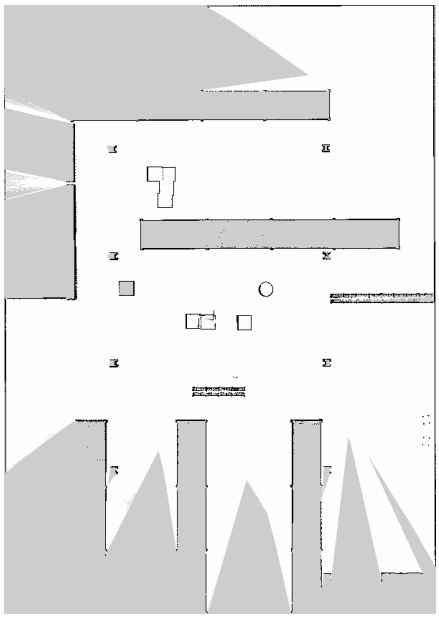
图1-1 实机运行地图	图1-2 仿真地图（无障碍）	图1-3 仿真地图（有障碍）

RRT算法实现和仿真（鞠正阳）

> rrt规划.zip

算法流程

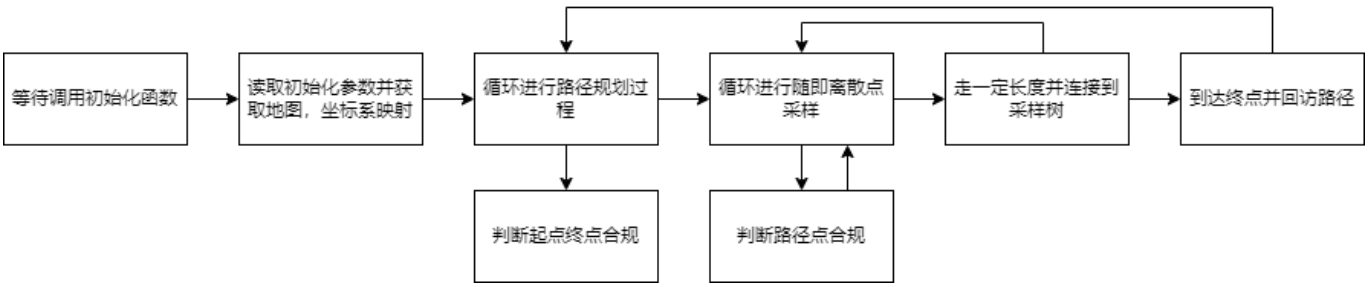

图2 基础RRT算法流程图

假设机器人为圆形并以半径进行膨胀，主要函数如下

```
bool isValid(double x0, double y0, const vector<vector<int>>& mapA) ;//判断点安全
Node* getNearestNode(const vector<Node*>& nodes, int x, int y);      //找树上最近节点
nav_msgs::msg::Path RRTLine::createPlan(...);                        //进行RRT和路径生成
void drawline(cv::Mat img,Node* goal);                               //利用cv2库可视化
struct Node {                                                        //节点类型定义
```

```
    double x, y;
    double costheta,sintheta;
    Node* parent;
};
```

算法细节

- 由于写代码时对Costmap格式不清楚，读取较为繁琐，故直接读取本地文件`ifstream mapFile("map.txt");... mapA[i][j] = number;`作为地图，正常运行。
- 由于写代码时对map2World用法不清楚，进行了坐标转换`int x=int((x0+15)/30*mapWidth);`
- 由于移植到ROS2插件的过程遇到了很多报错，使用`RCLCPP_ERROR()`进行调试信息输出，获得了较好的结果
- 为了方便使用push_back方法，故从goal向start遍历
- 对路径点进行了直线插值，路径更精细

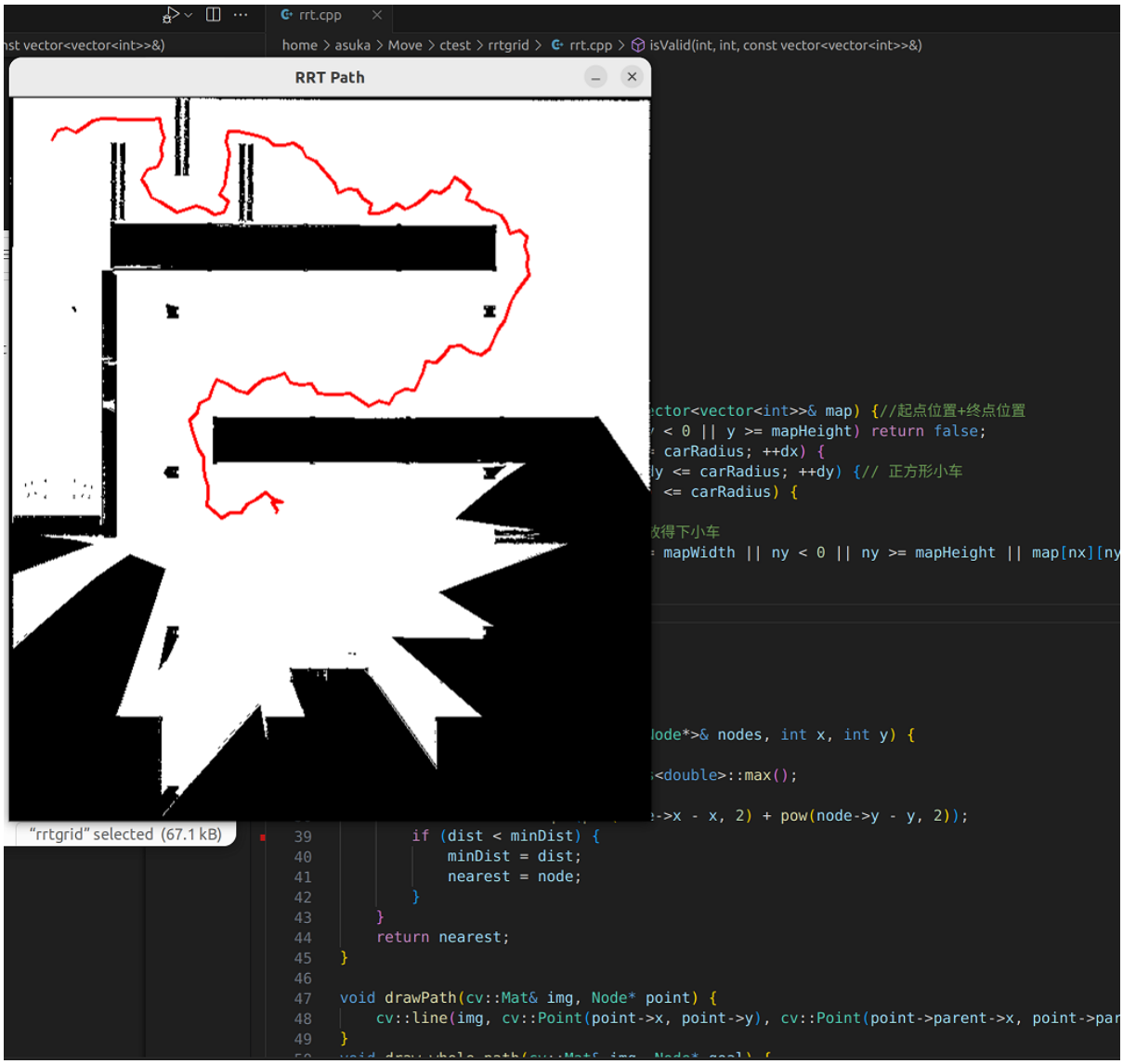

图4 单次RRT规划路径

算法性能

由于基础RRT算法的随机性，系统不得不在全局随机取点，且方向不确定，理论上时间最优时时间复杂度是$O(M\cdot N)$，即正比于地图规模；否则时间复杂度是$O(n)$，即正比于采样点数。空间复杂度也是

$O(n)$。此算法并不保证有限时间内有解，也不保证解最优，但是在小规模栅格地图（此例）中比较有效。经过100轮随机采样测试，发现其用时不稳定，且探索也较多（数万），探索点利用率仅为2%~4%。

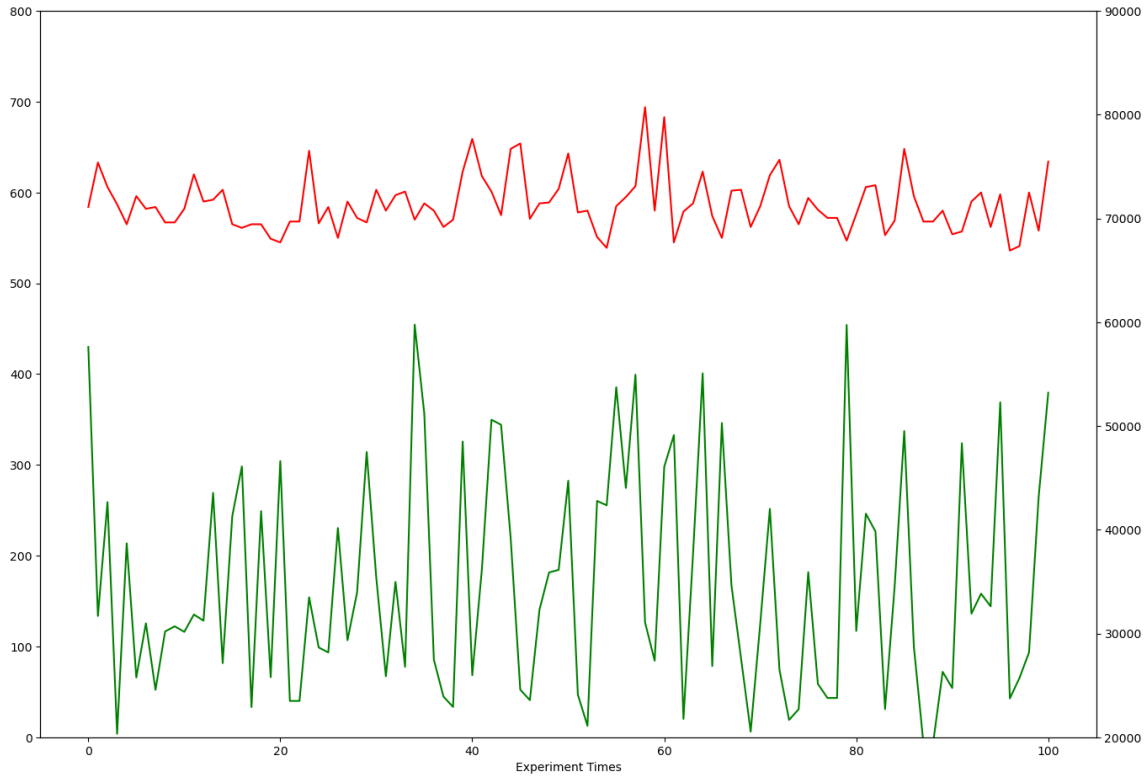

图5 上曲线：100次测试路径点数（左）；下曲线：总遍历点数（右）

将本算法移植到ROS2::NAV2平台后明显可以看出时间不稳定造成的负面影响，即长时间无响应，直接跳到下一个点，几乎无法在单次访问中生成可行路径。

A*算法实现和仿真（季书航）

> 仿真Astar算法（基于BaseGlobalPlanner修改）.zip

算法流程

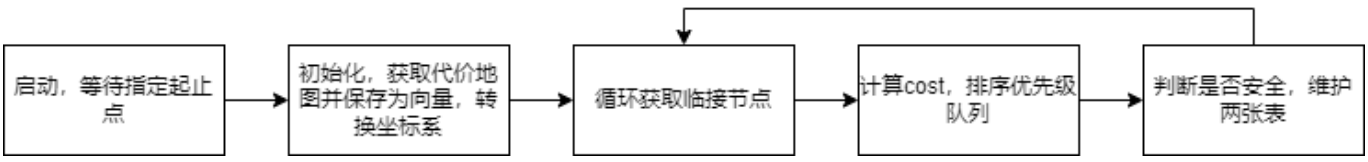

图6 A*算法流程图

算法细节

- 算法基于`BaseGlobalPlanner`实现，采用曼哈顿距离作为启发函数
- 使用点序号代替坐标，使用`costmap_->indexToCells(cell_index, x, y);`实现转化

```
struct Node{//节点定义
  float cost;
  int index;
};
bool AstarPlanner::makePlan(...);//A*过程
double AstarPlanner::getMoveCost(int firstIndex, int secondIndex);//计算实际消耗
double AstarPlanner::getHeuristic(int cell_index, int goal_index);//计算启发函数
vector<int> AstarPlanner::get_neighbors(int current_cell)//临接点
```

算法仿真

无

实机实验

另一版A*算法实现和实机（潘力豪）

> 实机运行的Astar规划器.zip

算法细节

- 启发式函数 $h(n)$选择直线距离
- 使用库函数`std::unordered_map<unsigned int, Node*>`减轻工作量
- 调用了`worldToMap`功能实现现实坐标到地图坐标的转换，使用`costmap_->getCost(...)`构建可调膨胀系数

```
struct Node//节点定义
{
    unsigned int x, y;
    float g_cost, h_cost;
    Node *parent;

    Node(unsigned int x, unsigned int y, float g_cost, float h_cost, Node *parent)
        : x(x), y(y), g_cost(g_cost), h_cost(h_cost), parent(parent) {}

    float f_cost() const { return g_cost + h_cost; }
};
struct CompareNode;//比大小
nav_msgs::msg::Path StraightLine::createPlan(...);
```

实验解读

算法在示例规划器模板上修改。路径生成搜索了A*算法的路径点。

地图读取和转化过程，由于有关障碍物的代价等信息在地图坐标下，因此我们

代价计算过程，我们主要使用机器人位置到终点的曼哈顿距离来表示代价，同时考虑到与障碍物的碰撞，我们采用了costmap_下的getCost功能来对代价进行修正，使之能够完成避障功能。

通过自定义启发式函数并在头文件中声明，实现了A*的寻路过程。

```
// 修改部分：增加代价权重
float inflation_cost = costmap_->getCost(neighbor_x, neighbor_y);
float cost_weight = (inflation_cost == nav2_costmap_2d::FREE_SPACE) ? 1.0 : inflation_cost / 100.0;

// 计算邻居节点的代价
float g_cost = current_node->g_cost + std::hypot(neighbor_world_x - cur_world_x, neighbor_world_y - cur_world_y) * cost_weight;
float h_cost = heuristic(neighbor_world_x, neighbor_world_y, goal.pose.position.x, goal.pose.position.y);
```

图7 修改权重代价实现避障的相关代码

```
// 执行A*搜索算法找到路径
std::priority_queue<Node*, std::vector<Node*>, CompareNode> open_set;
std::unordered_map<unsigned int, Node*> all_nodes;
std::unordered_map<unsigned int, bool> closed_set;

// 初始化起始节点
Node* start_node = new Node(start_x, start_y, 0.0, heuristic(start_x, start_y, goal_x, goal_y), nullptr);
open_set.push(start_node);
all_nodes[start_y * costmap_->getSizeInCellsX() + start_x] = start_node;

Node* goal_node = nullptr;

// A*搜索的主循环
while (!open_set.empty()) {
  Node* current_node = open_set.top();
  open_set.pop();

  // 检查是否到达终点
  if (current_node->x == goal_x && current_node->y == goal_y) {
    goal_node = current_node;
    break;
  }

  closed_set[current_node->y * costmap_->getSizeInCellsX() + current_node->x] = true;

  // 遍历当前节点的所有邻居节点
  for (int dx = -1; dx <= 1; ++dx) {
    for (int dy = -1; dy <= 1; ++dy) {
      if (dx == 0 && dy == 0) continue;
```

图8 A*搜索实现的部分代码

```
private:
  // TF buffer
  std::shared_ptr<tf2_ros::Buffer> tf_;

  // node ptr
  nav2_util::LifecycleNode::SharedPtr node_;

  // Global Costmap
  nav2_costmap_2d::Costmap2D * costmap_;

  // The global frame of the costmap
  std::string global_frame_, name_;

  double interpolation_resolution_;

  // Heuristic function to estimate cost from current node to goal
  float heuristic(double x1, double y1, double x2, double y2);
};

}  // namespace nav2_straightline_planner
```

图9 A* 头文件增加的函数与变量声明

环境和搜索参数对规划时间的影响

在我们的A*算法中，由于搜索得到的路径通过A*算法搜索得到，因此路径长度理论上是最优的。（不考虑地图膨胀后对代价函数的影响）实际上，由于地图膨胀后的代价因素，机器人在路径规划中往往会贴着粉色边缘。但是考虑到碰撞后，路径的最优性没有改变。

地图栅格化（默认分辨率）后，我们对地图搜索的间隔定为1。即对于下个节点的搜索，搜索的空间考虑为相邻的8个距离为1的方格点。计算其代价并采用A*算法，选择未被搜索过的最好的点，得到完整路径后录入`global_path`中输出。

由于A*的代价函数，属于启发式搜索，因此搜索节点与可行路径、障碍物有关，与地图尺寸大小无关。路径越长，往往搜索节点数越多，搜索时间就越长。但由于A*的启发性，往往时间不会很长。

A*算法的时间复杂度和空间复杂度

A*算法的时间复杂度主要取决于启发式函数的效率和优先队列的操作。假设可搜索的节点（地图信息量，空间分支因子为N，深度为M）为$N\cdot M$，则时间复杂度为$O(N\cdot Mlog(N\cdot M))$,其中$log(N\cdot M))$来源于优先序列的搜索操作；空间复杂度为$O(N\cdot M)$，因为最坏情况下，需要储存所有节点的信息。

通过多次仿真，发现对于不同长度路径的搜索花费的时间差别并不大，因为A*算法为深度优先搜索，而一般来说不会出现近路为“死胡同”而需要特别去绕远路的情况。因此A*算法规划时间非常短。仿真结果如下

表：A*规划时间与路径长度、搜索节点数的关系

路径长度	搜索节点数	规划时间（10 次平均）
短路径（2m-4m）	40-80	几乎没有
中路径（5m-15m）	100-300	0.42s
长路径（20m 以上）	400 以上	0.58s

A*算法修改参数仿真结果

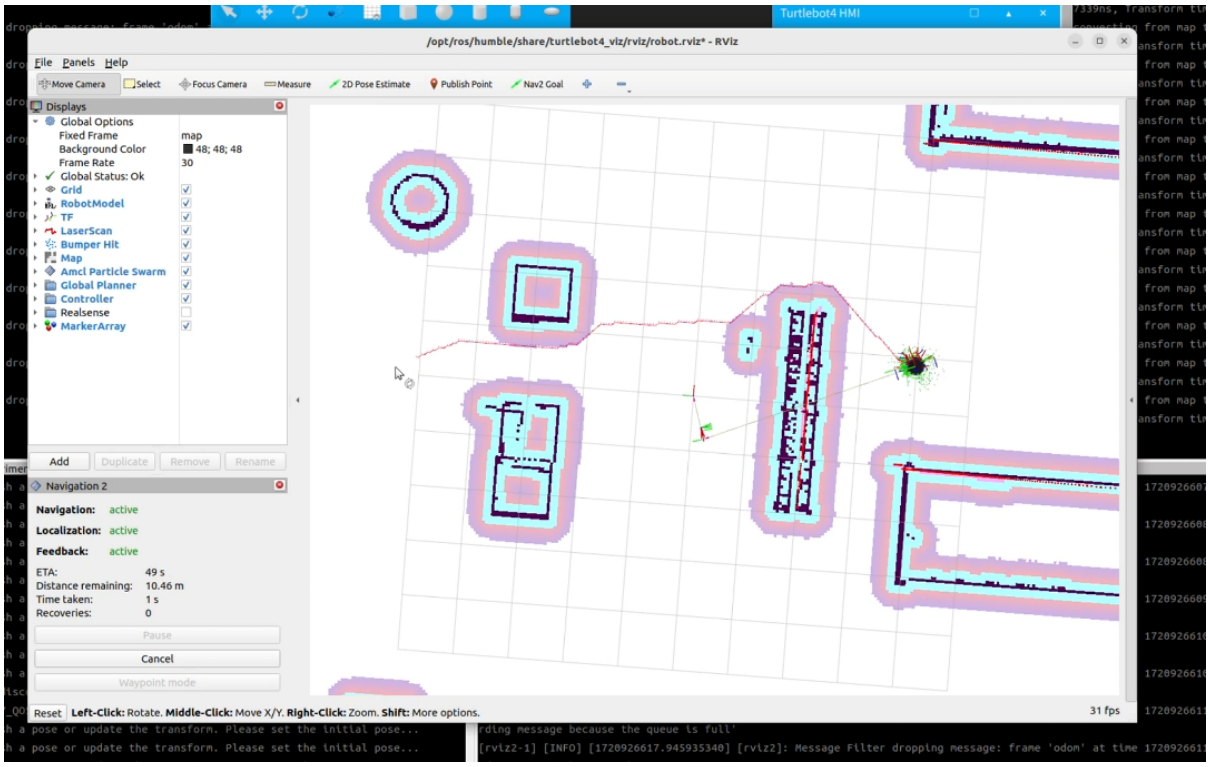

图10 A*算法仿真界面截图

实机演示和对应的Rviz可视化

见附件

> 自带Navfn规划器演示.mp4
>
> 实机运行Astar.mp4
>
> 实机运行Astar算法路径可视化.mkv

实验总结

A*算法具有较快的运行速度和较高的质量，在选用两类启发函数的条件下都能在非常短的时间内完成计算和寻路，没有出现卡在墙里的情况，且对障碍物膨胀的适应性较强（贴着墙壁走）。缺点是需要SLAM提前建图，算法对建图的准确度和分辨率要求较高。

RRT算法没有明确的指向性，不能在全局规划下找到更优解，不适合栅格地图环境，更适合探索类任务。

讨论心得

潘力豪

> 这次仿真实验和实物实验我收获了很多，我负责了A*算法的改写与仿真的运行，并参与了实物的实验验收。
>
> 初次接触ubuntu22.04系统，安装ROS2和配仿真环境就花费了我非常多的时间。而在运行仿真的过程中，由于需要改bashrc文件来进行实物与仿真的切换，经常会发生因为没有修改文件导致仿真中没有小车的情况；同时在写A*算法时，由于对ROS2的C++程序格式了解不足，导致很多结构错误，例如在hpp文件中改写函数声明，否则会导致编译失败；还有A*障碍物检测与膨胀系数的修改需要用mapToWorld

> 功能实现，因此实现障碍物的避障……这些都花费了我非常多的时间。在助教老师的帮助下，我们最终完整的实现了小车的路径避障，完成了相应的任务，我感到收获了很多。

鞠正阳

> 本次实验熟悉了Ubuntu的环境和ROS2系统的使用，也捡起了C++的代码编写能力，回顾了A*和RRT算法的基本流程，复习的意味大于学习的部分。我负责RRT算法书写和接口阅读，提供其它技术支持。代码能力是限制我本次实践的重要问题，需要熟悉C++各类函数和常用容器的使用，能够帮助写出更高效的代码，更帮助进行查错和调试。我也学会了阅读官方文档找到解决方法。
>
> 同时，小车性能非常优秀，希望可以再次使用。基于实验指导书的不清楚之处，我总结了一些个人经验附在文件中。nav2_tb4_course.md