

第十八届全国大学生智能汽车竞赛

室外 ROS 无人车赛（高教组）

技 术 手 册

学 校：华南师范大学

队伍名称：Vanguard

目录

实验一	无人车底盘控制实验	3
实验二	雷达数据分析与锥桶坐标获取实验	7
实验三	激光雷达循迹实验	14
实验四	激光 SLAM 建图	26
实验五	导航点获取	31
实验六	循迹与导航的切换	34
实验七	自主导航	38
实验八	停车点识别	50
实验九	红绿灯识别	56
实验十	地图滤波和锥桶连线	60

... ..

实验一 无人车底盘控制实验

实验目的：

- 1、熟悉 ROS 话题通信。
- 2、熟悉 Twist 信息。
- 3、为后续激光雷达循迹和自主导航提供电控支持。

实验内容：

创建控制节点，向 ”~/car/cmd_vel” 话题发布 Twist 信息。

实验器材：

ROS 智能车、电脑。

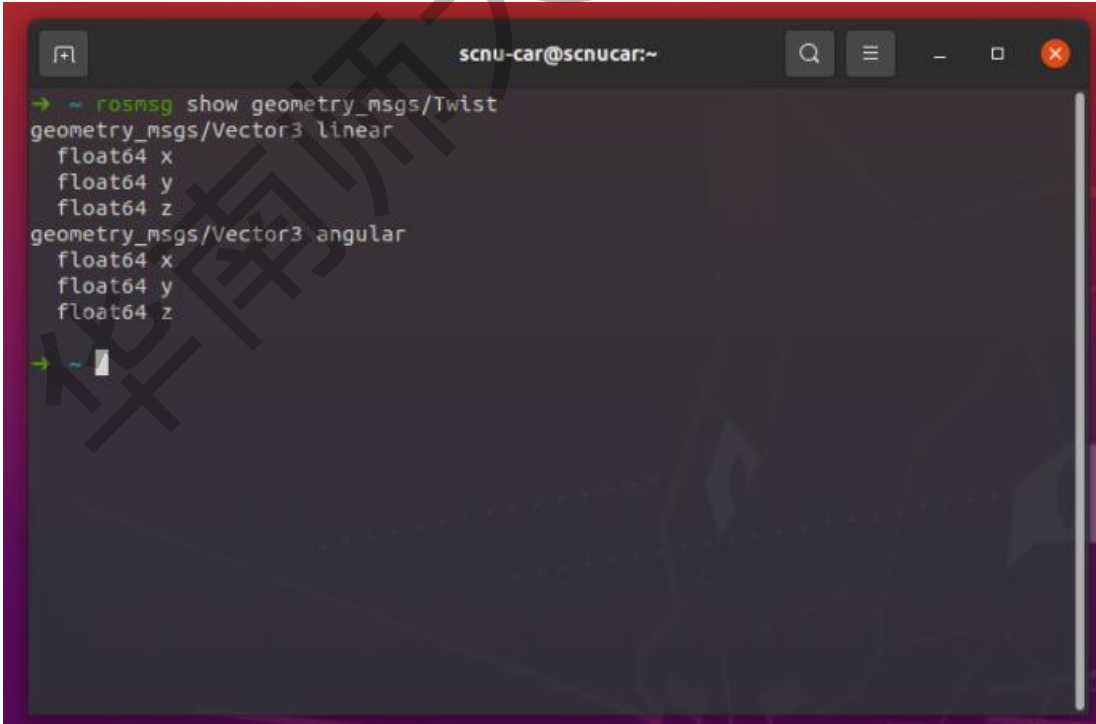
实验原理：

修改 Twist 信息，并将其发布至 ”~/car/cmd_vel” 话题。

实验步骤：

<1> 认识 Twist 信息。

打开一个终端，输入命令 `rosmmsg show geometry_msgs/Twist`。

A terminal window titled 'scnu-car@scnucar:~' with search, menu, and window control icons. The command `rosmmsg show geometry_msgs/Twist` has been executed. The output shows the structure of the Twist message, divided into linear and angular components, each with x, y, and z axes of type float64.

```
→ ~ rosmmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
→ ~
```

(图 1-1)

如图 1-1 所示，linear（线速度）下的 x、y、z 分别对应 x、y 和 z 方向

上的速度（单位是 m/s）；angular（角速度）下的 x、y、z 分别对应绕 x 轴的翻滚速度、绕 y 轴的俯仰速度和绕 z 轴的偏航速度（单位是 rad/s）。

在 ROS 智能车中，只需要用到线速度 x 分量和角速度 z 分量（偏航角）。

<2> Python 代码实现。

```
#导包
import rospy
#导入 Twist 信息
from geometry_msgs.msg import Twist

#初始化控制节点
rospy.init_node('racecar_control')

#创建发布者
pub = rospy.Publisher('~car/cmd_vel', Twist, queue_size=5)
twist = Twist()
```

在 Publisher 中，queue_size 参数用来指定队列大小，此处表示发布者可以在没有订阅者的情况下保留的未发布消息的最大数量为 5。

```
#设置线速度和角度
twist.linear.x = 1700
twist.linear.y = 0
twist.linear.z = 0
twist.angular.x = 0
twist.angular.y = 0
twist.angular.z = 90
while(True):
    #一直发布该信息，使智能小车以 1700 的线速度向前运动
    pub.publish(twist)
```

若要让 ROS 智能车实现直行，需要先测试出智能车的舵机中值，然后将该舵机中值赋给 twist.angular.z 变量即可。

- 代码文件：control_car.py
- 使用方法：
 - (1) 打开一个终端，先运行 `roslaunch racecar Run_car.launch`
 - (2) 再打开另一个终端，到代码文件夹下，运行 `python3 control_car.py`
 - (3) 打开电调，小车就会向前运动了

实验二 雷达数据分析与锥桶坐标获取实验

实验目的：

- 1、学会使用激光雷达测量数据。
- 2、为后续激光雷达循迹提供基础支持。

实验内容：

- 1、了解激光雷达测量数据。
- 2、处理获取到的雷达数据，获取锥桶坐标。

实验器材：

ROS 智能车、激光雷达、电脑。

实验原理：

激光雷达传入 LaserScan 消息，处理该消息得到锥桶坐标。

实验步骤：

<1> 了解激光雷达测量数据。

激光雷达传入的测量数据类型为 sensor_msgs 下的 LaserScan，打开一个终端，运行命令 `rostopic show sensor_msgs/LaserScan`。

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

(图 2-1)

如图 2-1 所示，其主要由以下几部分构成：

- Header header，包含 uint32 seq、time stamp、frame_id，为消息的头部信息，并非本实验重要内容，故不详述；

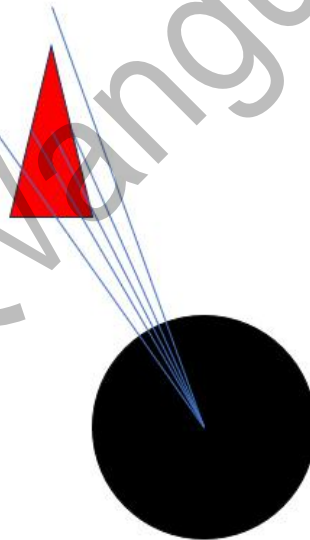
- float32 angle_min: 激光束的最小角度（以弧度表示）；
- float32 angle_max: 激光束的最大角度（以弧度表示）；
- float32 angle_increment: 激光束角度的增量（以弧度表示）；
- float32 time_increment: 每个激光束之间的时间增量（以秒为单位）；
- float32 scan_time: 一次完整扫描所需的时间（以秒为单位）；
- float32 range_min: 测量距离的最小值（以米为单位）；
- float32 range_max: 测量距离的最大值（以米为单位）；
- float32[] ranges: 激光束的距离值数组，按角度顺序排列；
- float32[] intensities: 激光束的强度值数组，按角度顺序排列；

上述数据中，我们需要使用到的是 angle_min、angle_increment 和 ranges。

<2> 处理雷达数据，获取锥桶基于小车的直角坐标系坐标。

1. 遍历雷达传入数据，找到距离小车 2 米范围内的所有锥桶。

激光雷达扫描锥桶示意图



（图 2-2）

如图 2-2 所示，激光雷达扫描到锥桶会有两个典型特征，一个是距离发生突变，一个是发生突变后的距离数据具有连续性。

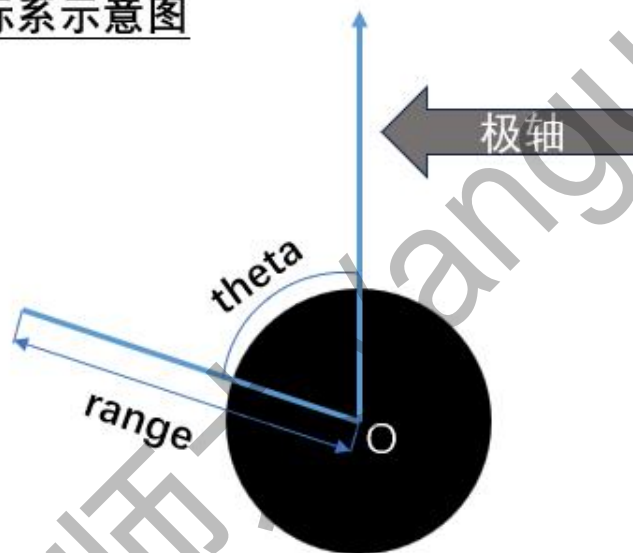
根据上述两个特性，我们遍历激光雷达传入的数据，找到符合 $\text{ranges}[i-1] - \text{ranges}[i] \geq 2.0$ 的数据（即距离减小超过 2.0 米就认为是距离突变）。注意，此时我们需要对 $\text{ranges}[i]$ 进行一定的限制，必然有的是 $\text{ranges}[i]$ 应该小于一定值。因为室外环境复杂，激光雷达扫描范围广，距离突变大于 2 米的情形很多，在遍历时雷达传入数据时限制 $\text{ranges}[i] < \text{ranges_maxvalue}$ 才取出做下一步处理，可以减小后续筛选处理的计算量。（经多次实验，确定 ranges_maxvalue 在 2.0 左右较为合适）。接下来，判断距离发生突变后的数据是否跟随着距离连续。我们定义的距离连续是指在一定范围内，激光雷达传入的距离数据在一定范围内浮动，在锥桶识别中，我们认为 $\text{ranges}[i]$ 与 $\text{ranges}[i+1]$ 的差的绝对值在 0.2 以内即满足一次连续。所以，从 i 开始，遍历 i 到 $i + \text{bucket_threshold}$ 的

ranges[]数据，计算 $\text{abs}(\text{ranges}[i] - \text{ranges}[i+\text{idx}])$ 的值，若该值小于 0.2 则认为是一次连续。（bucket_threshold 是连续性判断的遍历数据个数，经多次实验认为 bucket_threshold 的值为 10 较为合适）。在遍历十个数据的步骤中，最严格的要求当然是要求十个数据都能满足距离连续，但是这样往往会导致“丢桶”的情况发生，这是由于激光雷达传入的数据并不是每一个都肯定正确的，特别是受到强光或者高温影响的时候，所以往往我们只需要在 10 个数据中找到超过一定次数 valid_bucket_threshold 的连续就可以认为此处确实是锥桶了（经多次实验 valid_bucket_threshold 的值在 5-7 较为合适）。

2. 获取锥桶基于小车的极坐标系坐标。

设激光雷达所处位置为原点，由上往下看，激光雷达逆时针旋转，以小车正前方为极轴方向，建立极坐标系。

极坐标系示意图



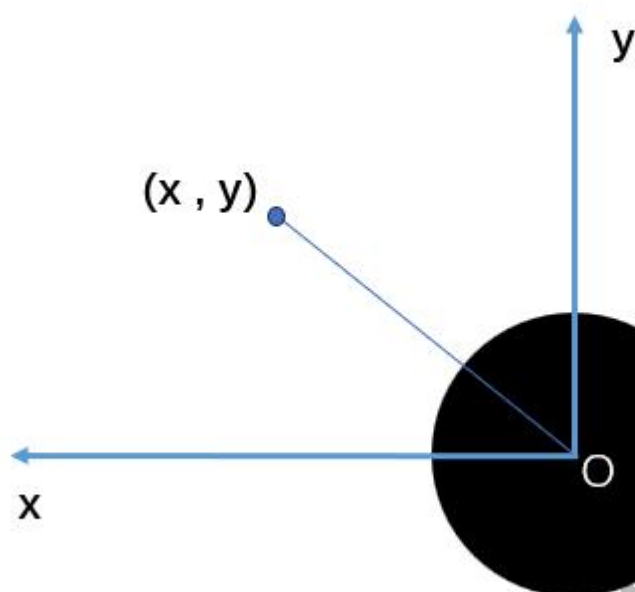
(图 2-3)

如图 2-3 所示，可知极角 $\theta = \text{angle_min} + i * \text{angle_increment}$ ，极径 $\text{range} = \text{ranges}[i]$ 。

3. 将极坐标系坐标转化为直角坐标系。

设激光雷达所处位置为原点，以小车正左边为 x 轴正方向、正前方为 y 轴正方向，建立平面直角坐标系。

直角坐标系示意图



(图 2-4)

如图 2-4 所示，可知 $x = \text{range} * \sin(\text{theta})$ ， $y = \text{range} * \cos(\text{theta})$ 。

<3> C++代码实现锥桶坐标获取。

- 首先输入以下命令，并输入密码，安装 sensor_msgs 依赖包：

`sudo apt install ros-sensor-msgs`

```
rosfang@rosfang-VirtualBox:~$ sudo apt install ros-sensor-msgs
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件：
  ros-geometry-msgs ros-std-msgs
下列【新】软件包将被安装：
  ros-geometry-msgs ros-sensor-msgs ros-std-msgs
升级了 0 个软件包，新安装了 3 个软件包，要卸载 0 个软件包，有 190 个软件包未被升
级。
需要下载 24.0 kB 的归档。
解压缩后会消耗 147 kB 的额外空间。
您希望继续执行吗？ [Y/n] Y
```

(图 2-5)

- 安装完成后，编写 C++代码
- 由于本实验中处理完激光雷达传入数据之后需实现在回调函数中发布数据，因此需要定义一个类如下：

```
class PubAndSub
```



```

{
private:
    ros::NodeHandle n_;
    ros::Publisher pub_;
    ros::Subscriber sub_;
public:
    PubAndSub()
    {
        pub_ = n_.advertise<geometry_msgs::Twist>("/car/cmd_vel",5);
        sub_ = n_.subscribe("/scan",5,&PubAndSub::callback,this);
    }
    void callback(const sensor_msgs::LaserScan::ConstPtr &laser);
};

```

- 其中，/scan 为雷达话题，/car/cmd_vel 为小车底盘控制话题
- 随后包含头文件、定义所需变量：

```

#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"
#include "geometry_msgs/Twist.h"
#include <math.h>

#define freq 1440//每转一圈雷达扫描次数

```

```

typedef struct //极坐标系下的点
{
    double range;
    double theta;
}Point_polar;

```

```

typedef struct //直角坐标系下的点
{
    double x;
    double y;
}Point_rectangular;

```

- 编写 main 函数如下：

```

int main(int argc, char *argv[])
{
    ros::init(argc,argv,"laser_go");
    PubAndSub PAS;
    ros::spin();
    return 0;
}

```

- 随后进行回调函数的内容编写，如下：

```
void PubAndSub::callback(const sensor_msgs::LaserScan::ConstPtr &laser)
{
    int i,j=0;
    geometry_msgs::Twist twist;
    Point_polar pp[30] = {0,0};
    Point_rectangular pr[30] = {0,0};

    for (i = 1; i < freq - bucket_threshold; i++)
    {
        if (laser->ranges[i - 1] - laser->ranges[i] >= 2.0 && laser->ranges[i] < 2.0 &&
laser->ranges[i] != 0)
        {
            int continue_ranges = 0; //距离连续次数
            int continue_dectect = 0; //距离连续标志位
            for (int idx = 1; idx < bucket_threshold; idx++)
            {
                if (abs(laser->ranges[i] - laser->ranges[i + idx]) < 0.2)
                {
                    continue_ranges++; //数据连续，参数自增
                    if (continue_ranges >= vaild_bucket_threshold)
                    {
                        //连续判断成功次数超过阈值，认为是锥桶
                        continue_dectect = 1;
                        break;
                    }
                }
            }
            if (continue_dectect == 1) //获得 2 米范围内各锥桶的极坐标
            {
                pp[j].range = laser->ranges[i];
                pp[j].theta = i * laser->angle_increment + laser->angle_min;
                j++;
            }
        }
        for(i = 0;i < 30;i++)
        {
            if(pp[i].range)
            {
                pr[i].x = pp[i].range*sin(pp[i].theta);
                pr[i].y = pp[i].range*cos(pp[i].theta); //获得 2 米范围内各锥桶的直角坐标
                ROS_INFO("found bucket point:(%.2f,%.2f)\n",pr[i].x,pr[i].y);
            }
        }
    }
}
```

- 代码文件: laser_go.cpp
- 使用方法
 - (1) 打开一个终端, 先运行 `roslaunch racecar Run_car.launch` 运行节点。
 - (2) 再打开另一个终端, 到代码文件夹下, 运行 `roslaunch laser_go.launch`, 就能够正常输出 2 米内的锥桶在所建平面直角坐标系中的坐标了。

华南师大Vanguard

实验三 激光雷达循迹实验

实验目的：

- 1、掌握采用激光雷达识别锥桶并循迹的方法。
- 2、ROS 无人车完成比赛第一圈的循迹任务。

实验内容：

- 1、处理锥桶坐标信息并对锥桶进行分组。
- 2、根据分好组的锥桶数据进行循迹操作。

实验器材：

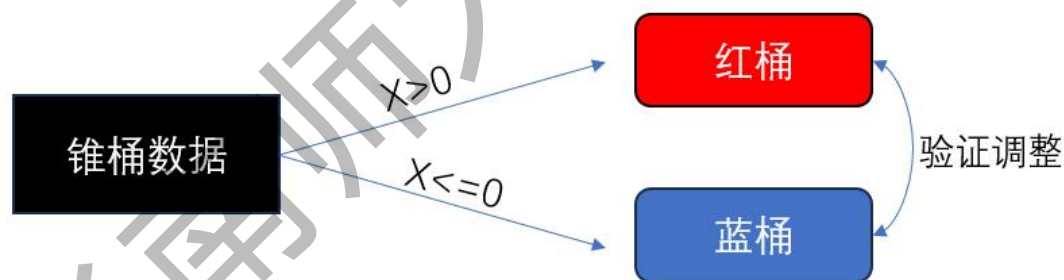
ROS 智能车、激光雷达、电脑。

实验原理：

处理锥桶坐标信息，判断锥桶所处的位置，并通过锥桶位置进行循迹。

实验步骤：

<1> 将锥桶分为红、蓝锥桶两组。

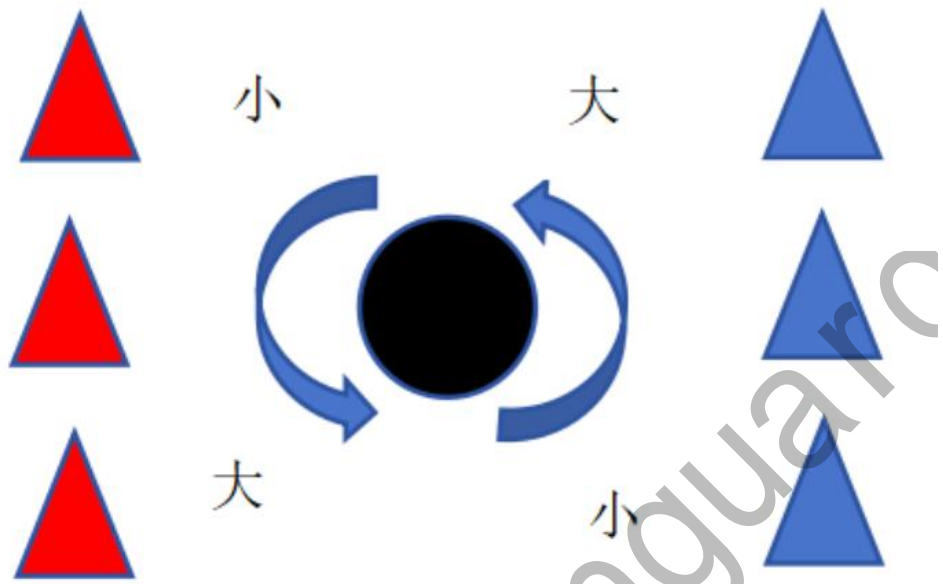


(图 3-1)

如图 3-1 所示，在一般情况下，蓝色锥桶在小车的右边，而红色锥桶在小车的左边，所以我们对所有锥桶数据先进行一次初筛，以 $x=0$ 为基准，把所有 $x>0$ 的锥桶数据存放入红色锥桶数组 `red_p[]`，其余的锥桶数据存放入蓝色锥桶数组 `blue_p[]`。

我们知道，存在一些情况，如直道入弯、蛇形弯道中，红蓝锥桶并不完全符合上述分布情况，即存在一些蓝色锥桶的 x 坐标大于 0，或存在一些红色锥桶的 x 坐标小于 0，所以我们需要对初筛后的红、蓝锥桶数组进行验证调整。可以确定的一点是，当蓝色锥桶出现在 $x>0$ 的一侧时，证明小车正前方指向外边界；反之，当红色锥桶出现在 $x<0$ 的一侧时，证明小车正前方指向内边界，而这两种情

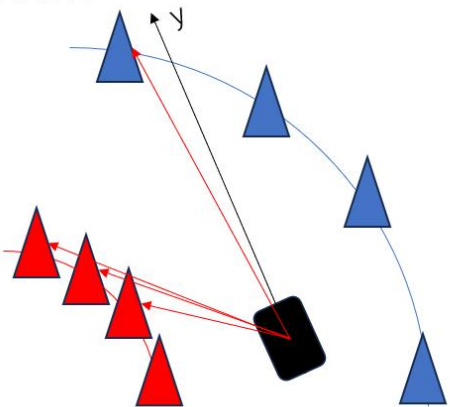
形不会同时发生，所以我们只需要对数组元素较多的锥桶数组进行验证。验证的思路有很多，我们采用取“标准锥桶”+距离判断的方式寻找是否存在误分组的锥桶，标准锥桶即为可信度最高的锥桶，即在这个数组中误分组概率最低的元素。



(图 3-2)

如图 3-2 所示，我们只取小车后方一定距离（这个距离很小）开始往前的锥桶数据，根据激光雷达扫描顺序，可知从近到远，红色锥桶下标从大到小，而蓝色锥桶下标从小到大。一般情况下，离小车更近（此处含义是 y 的绝对值最小，而不是实际距离）的锥桶更不容易出错，所以当我们选取标准锥桶时，默认选取下标为 $j-1$ （下标最大）的锥桶为红色标准锥桶，而选取下标为 0（下标最小）的锥桶为蓝色标准锥桶。

小车直道入弯示意图



(图 3-3)

如图 3-3 所示，这是一个直道入弯的情形，是一个典型的容易出现误分组

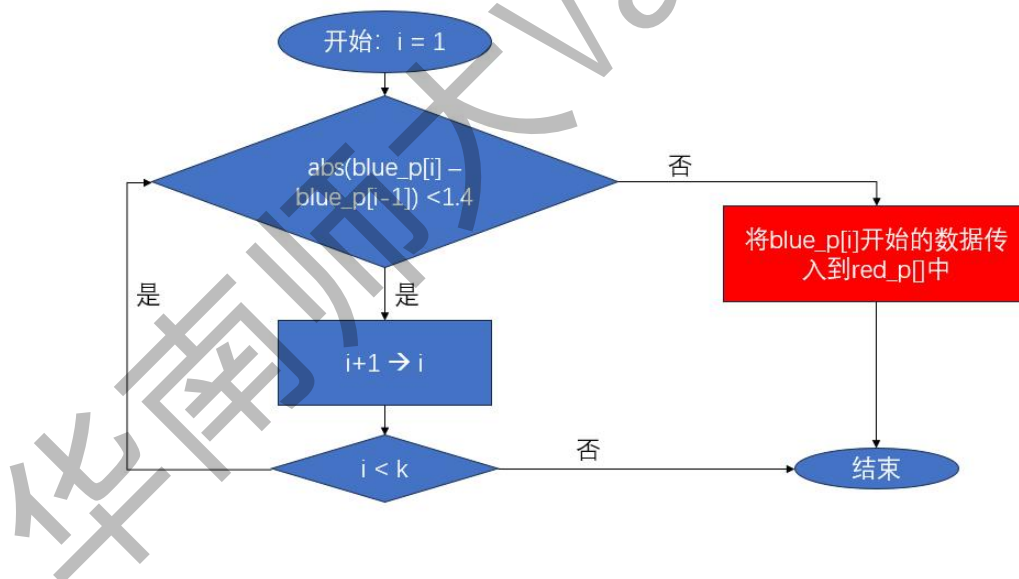
的情况，此时初筛后的红色锥桶数组有 4 个元素，然而我们知道下标为 0 的元素实际上是一个蓝色锥桶。

根据上述标准锥桶的选取原则，我们会选择下标为 3 的元素作为红色标准锥桶，计算 red_p[2] 与 red_p[3] 的距离（计算数组名 a[下标 m] 与数组名 b[下标 n] 的距离，含义为 $\sqrt{\text{pow}((a[m].x - b[n].x), 2) + \text{pow}((a[m].y - b[n].y), 2)}$ ，为简单表述，下同），若距离超过 1.4 米，则记录 m = 2（变量 m 用来记录第一个大于距离阈值的锥桶下标），显然上面这个情形不会超过 1.4 米，那我们认为 red_p[2] 也是一个正确的锥桶，可以用于下一个锥桶的判断。

接着计算 red_p[1] 与 red_p[2] 的距离（实际上这是一种递推），判断距离；计算 red_p[0] 与 red_p[1] 的距离，此时距离超过 1.4 米，则说明 red_p[0] 不是红色锥桶，而是蓝色锥桶。实际上因为激光雷达的扫描方向不变，并且实际的赛道边界是连续的，所以当 red_p[i] 是误分组的锥桶，那 red_p[0]~red_p[i] 均为被误分组的锥桶。类似地，当 blue_p[i] 是误分组的锥桶，可以说明 blue_p[i]~blue_p[k-1] 均为被误分组的锥桶。

至此，我们找到了被误分组的锥桶。接下来，就是把被误分组的锥桶归还至另一个锥桶数组。要注意的是，为了保持“从近到远，红色锥桶下标从大到小，而蓝色锥桶下标从小到大”这个特性，更新锥桶数组时需要遵循一定的规则，被误分组的红色锥桶应该补充在红色锥桶数组的前边，被误分组的蓝色锥桶应该补充在蓝色锥桶数组的后边...

蓝桶数组验证调整流程图



(图 3-4)

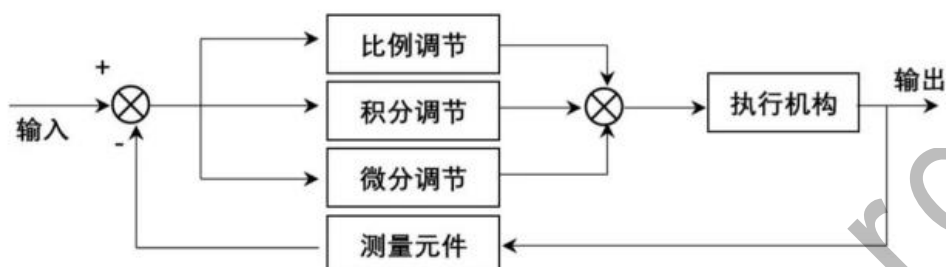
如图 3-4 所示，这是一个误分组锥桶判断和调整的基本流程（以蓝桶数组元素多的情况为例，当红桶数组元素多时处理过程类似）。

至此，我们完成了将锥桶分成红、蓝锥桶两组，可以认为把内、外边界的锥桶坐标分别存在了两个数组中，用于服务后续循迹。

〈2〉 熟悉 PID 控制和理解 PID 控制调参思路。

PID 控制是一种常用的反馈控制算法，用于调节和稳定系统的输出。PID 代表比例（Proportional）、积分（Integral）和微分（Derivative）。它能够控制系统达到目标状态，并使系统在目标状态附近稳定运行。

常见的 PID 控制有位置式 PID 和增量式 PID，本实验将 PID 控制应用于舵机控制，采用更适合的位置式 PID，PID 输出量为三个部分的输出量总和。



(图 3-5)

如图 3-5 所示，为一个典型的 PID 闭环控制示意图。

PID 算法的公式：

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

将其离散化，得到：

$$\begin{aligned} \bullet \text{ 比例项: } K_p e(t) &\xrightarrow{\text{离散化}} K_p e_k \\ \bullet \text{ 积分项: } K_i \int_0^{t_k} e(\tau) d\tau &\xrightarrow{\text{离散化}} K_i \sum_{i=1}^k e(i) \Delta t \\ \bullet \text{ 微分项: } K_d \frac{de(t_k)}{dt} &\xrightarrow{\text{离散化}} K_d \frac{e(k) - e(k-1)}{\Delta t} \end{aligned}$$

(图 3-6)

如图 3-6 所示，离散化后的位置式 PID 公式为 $u(k) = K_p * e_k + K_i * \text{err_sum} + K_d * (e_k - e(k-1))$ 。

下面简单说明一下实际应用时 PID 参数的基本调节方法：

- K_p ：该参数在 PID 算法中的作用最主要体现在反应速度方面，提高该参数的值可以显著提高小车的反应速度，同时也能够起到消除静态误差的作用。

然而，该参数过大时，将会造成小车在寻路时明显摆动。当发现小车反应速度过慢时，应增大该参数；当发现小车左右摆动明显时，应减小该参数。

- Ki: 该参数在 PID 算法中起到消除静态误差的作用，主要在转弯时体现。但在实际应用中，累积的误差可能导致小车反应速度大幅降低，因此，在实际应用中，必须对积分值进行限幅，减小累积误差对反应速度的影响。当发现小车转弯时容易碰撞外圈时，应增大该参数；转弯时碰撞内圈，或是转弯后反应速度明显降低，则应减小该参数。
- Kd: 该参数在 PID 算法中主要起到减少抖动的作用，在小车左右晃动明显时，可以适当增大该参数；当小车出现转弯困难等现象时，可以适当减小该参数。

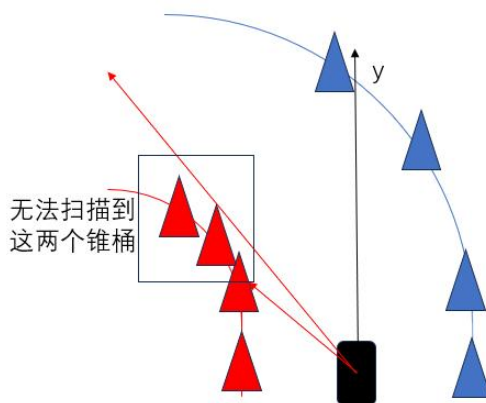
对小车前进方向的控制，实际就是利用当前位置与目标位置的偏差通过 PID 控制器输出给舵机。

〈3〉 掌握获取位置偏差的方法。

在步骤〈1〉中，我们已经将锥桶坐标分为了红、蓝锥桶两组。除了在直道行驶时，一般情况下，扫描到的红、蓝锥桶个数不相等，为了计算误差方便，我们需要取等量的红、蓝锥桶用来计算位置偏差。

对于等量锥桶的选取，有两个直接的方案可供选择，一个是舍弃过多的锥桶，即只在两个锥桶数组中取出等量的锥桶，取的锥桶数目由数量较少的锥桶决定，如果某种锥桶只取其中一部分来计算误差，优先取靠近小车的锥桶（即红色锥桶优先取下标大的、蓝色锥桶优先取下标小的）。另一个方案是对锥桶较少的数组进行补充，即将个数少的锥桶数组元素补充到和个数多的一样多，可行的补充方案是将个数少的锥桶数组中距离小车最远的那个锥桶数据，多复制几份，直到两个数组元素个数相等。（之所以复制最远的数据，是因为大部分情况被一边个数少是因为存在锥桶被遮挡了（如图 3-7），离得近的锥桶挡住了离得较远的锥桶，在激光雷达的扫描过程中，并不会扫描到离得远的锥桶，所以我们要补充的实际上是较远的锥桶数据，使用最远的数据进行复制，能减小补充过程带来的数据误差。）

锥桶被遮挡示意图



(图 3-7)

比较上述两个方案，不难得知，方案 1 的优点在于锥桶的坐标准确性高，计算结果更可靠，缺点则在于没有充分利用所有扫描到的锥桶数据；而方案 2 的优点在于充分利用了锥桶数据，缺点则是在计算过程中引入了更多误差。综上，我们综合两个方案的优点，尽可能利用多的锥桶数据，又要尽可能少引入新误差，制定了一个更为合理锥桶选取方案：

- (1) 当有一种锥桶个数大于 3 个时，只保留 3 个距离较近的。
- (2) 当一种锥桶比另一种锥桶个数大 2 时，个数多的锥桶将最远的锥桶数据舍弃。
- (3) 当一种锥桶比另一种锥桶个数大 1 时，个数少的锥桶将最远的锥桶复制一份一起参与计算误差。

因为最终误差计算用的是锥桶的 x 坐标，所以复制的一份锥桶并不会引入太大误差，因为遮挡的情况发生时，两个锥桶 x 值必然很接近。

计算误差采用加权求和的方式。首先要确定有几对锥桶参与误差计算，这一步由等量锥桶的选取结果可得出；接着，从近到远累加每对锥桶数据的 x 值之和与对应权重的乘积（具体查看代码相关部分即可，不再详述）；最后，将误差传入给 PID 控制器，用于舵机控制。

<4> C++代码实现激光雷达循迹。

本实验代码建立在《实验二 雷达数据分析与锥桶坐标获取实验》的基础上，部分相同代码不再赘述，只介绍本实验新增部分代码。

- 首先，编写 Control.h 头文件用于位置式 PID 控制：

```
namespace Control
{
class PID
{
public:
double kp;
double ki;
double kd;
double error_out;
double last_error;
double integral;
double inte_max;//积分限副
double last_diff;//上一次微分值

double PIDPositional(double error);//位置式 PID 控制器
double PIDIncremental(double error);
void Init();
};
double PID::PIDPositional(double error)//位置式 PID 控制器
```

```
{
integral += error;
```

```
    if(integral>inte_max)
        integral = inte_max;
error_out = (kp*error) + (ki*integral) + (kd*(error-last_error));
last_error = error;
    return error_out;
}

double PID::PIDIncremental(double error)//增量式 PID 控制器
{
error_out = kp*(error-last_error) + ki*error + kd*((error-last_error)-last_diff);
    last_diff = error - last_error;
    last_error = error;
    return error_out;
}
```

```
void PID::Init()//PID 初始化,参数在此调节
{
    kp = 3.5; //1830 3.5
    ki = 0;
    kd = 4.5; //1830 4.5
    error_out = 0.0;
    last_error = 0.0;
    integral = 0.0;
    inte_max = 8.0;
    last_diff = 0.0;
}
}
```

- 接着，补充回调函数：

```
void PubAndSub::callback(const sensor_msgs::LaserScan::ConstPtr &laser)
{
    int i=0,j=0,k=0,flag_jk=0;
    double range = 0,error = 0,theta=0,tmp_x=0,tmp_y=0,angle=0,rate_sum = 0,error_sum = 0;
    Point_rectangular red_p[30] = {0,0};
    Point_rectangular blue_p[30] = {0,0};
    for (i = 1;i < freq - bucket_threshold;i++)
    {
        if (laser->ranges[i - 1] - laser->ranges[i] >= 2.0 && laser->ranges[i] < 2.0 && laser->ranges[i] !=
0 && laser->ranges[i - 1] - laser->ranges[i+1] >= 2.0)
        {
            int continue_ranges = 0;
            int continue_dectect = 0;
```

```

//距离突然减小，并且较小距离小于 2 米且不为 0
for(int idx = 1; idx < bucket_threshold; idx++)
{
    if(abs(laser->ranges[i] - laser->ranges[i+idx]) < 0.2)
    {
        continue_ranges++;
        if(continue_ranges >= vaild_bucket_threshold)
        {
            continue_dectect = 1;
            break;
        }
    }
}

if(continue_dectect == 1)
{
    theta=i*laser->angle_increment + laser->angle_min;
    cal_point(theta,laser->ranges[i],tmp_x,tmp_y);
    if((tmp_x <=max_left_dis && tmp_x>=-max_right_dis) && tmp_y < 2&& tmp_y>-0.2 &&
    tmp_y!= 0 )
    {
        if(tmp_x>0)
        {
            if(laser->ranges[i]<2)
            {
                red_p[j].x = tmp_x;
                red_p[j].y = tmp_y;
                ROS_INFO("_____found red bucket point:(%.2f,%.2f)\n",red_p[j].x,red_p[j].y);
                j++;
            }
        }
        else
        {
            blue_p[k].x = tmp_x;
            blue_p[k].y = tmp_y;
            ROS_INFO("_____found blue bucket point:(%.2f,%.2f)\n",blue_p[k].x,blue_p[k].y);
            k++;
        }
    }
}

ROS_INFO("j=%d,k=%d\n",j,k);

```

```

if(k >= j)

```

```

{
int m = 100;
for(int i = 1; i < k; i++)
{
if(sqrt(pow((blue_p[i-1].x-blue_p[i].x),2)+pow((blue_p[i-1].y-blue_p[i].y),2)) > 1.4)
{
m = i;
break;
}
}
if(m != 100)
{
for(int i = j - 1; i >= 0; i--)
{
red_p[i + k - m] = red_p[i];
}
for(int i = 0; i < k - m; i++)
{
red_p[i] = blue_p[m + i];
}
j += (k - m);
k = m;
}
else
{
int m = 100;
for(int i = j-2; i >= 0; i--)
{
if(sqrt(pow((red_p[i].x-red_p[i+1].x),2)+pow((red_p[i].y-red_p[i+1].y),2)) > 1.4)
{
m = i;
}
}
if(m != 100)
{
for(int i = 0; i < m + 1; i++)
{
blue_p[k + i] = red_p[i];
}
for(int i = 0; i < j - m - 1; i++)
{
red_p[i] = red_p[i + m + 1];
}
}
}
}

```

```

}
j -= (m+1);
k += (m + 1);
}
}

```

```

//锥桶数据标准化：等量且小于 4 个
if(j>3)
{
    //红桶大于 3 个，舍弃最远的也就是最小号元素
    for(int i = 0; i < 3; i++)
    {
        red_p[i] = red_p[i+1];
    }
    j = 3; //调整后，红桶有 3 个
}
if(k>3)
{
    //蓝桶大于 3 个，舍弃最远的也就是最大号元素
    k = 3; //调整后，蓝桶有 3 个
}
if(j>k && k!=0)
{
    //红桶比蓝桶多且蓝桶不为零的情况
    if(k == 1)
    {
        //蓝桶只有一个，那只用两对锥桶计算误差
        if(j > 2)
        {
            for(int i = 0; i < 2; i++)
            {
                red_p[i] = red_p[i+1];
            }
            j = 2;
        }
    }
    blue_p[1] = blue_p[0]; //复制一份蓝桶
    k = 2;
}
else if(k == 2)
{
    blue_p[2] = blue_p[1];
    k = 3;
}
}

```

```

else if(k > j && j != 0)
{
    if(j == 1)
    {
        k = 2;
        red_p[1] = red_p[0];
        j = 2;
    }
    else if(j == 2)
    {
        red_p[2] = red_p[1];
        red_p[1] = red_p[0];
        j = 3;
    }
}

```

```

Point_rectangular* point_to_red = &red_p[j-1];
int error_count = j < k ? j : k;

if (error_count)
{
    for (i = 0; i < error_count; i++)
    {
        error_sum += (blue_p[i].x + point_to_red->x) * rate_p[i];
        rate_sum += rate_p[i];
        point_to_red--;
    }
    error = error_sum / rate_sum * 53.33;
}
angle = pid.PIDPositional(error);
twist.angular.z = middle_angle + angle;
twist.linear.x = speed;
if (abs(angle) >= 20)
{
    twist.linear.x -= 25;
}
if (twist.angular.z > 180)
    twist.angular.z = 180;
if (twist.angular.z < 0)
    twist.angular.z = 0;
pub_.publish(twist);

```

- 最后，补充 main 函数：

```

int main(int argc, char *argv[])

```

```
{  
pid.Init();  
ros::init(argc,argv,"laser_go");  
PubAndSub PAS;  
ros::spin();  
return 0;  
}
```

- 至此，代码编写完成，将小车放在赛道中，打开 Run_car.launch 和本实验编写的节点，即可实现激光雷达自主循迹。

华南师大 Vanguard

实验四 激光 SLAM 建图

实验目的：

- 1、了解 gmapping 算法的原理。
- 2、熟悉在 ROS 环境下快速配置并使用 gmapping 算法的流程。
- 3、了解 gmapping 的关键参数。

实验内容：

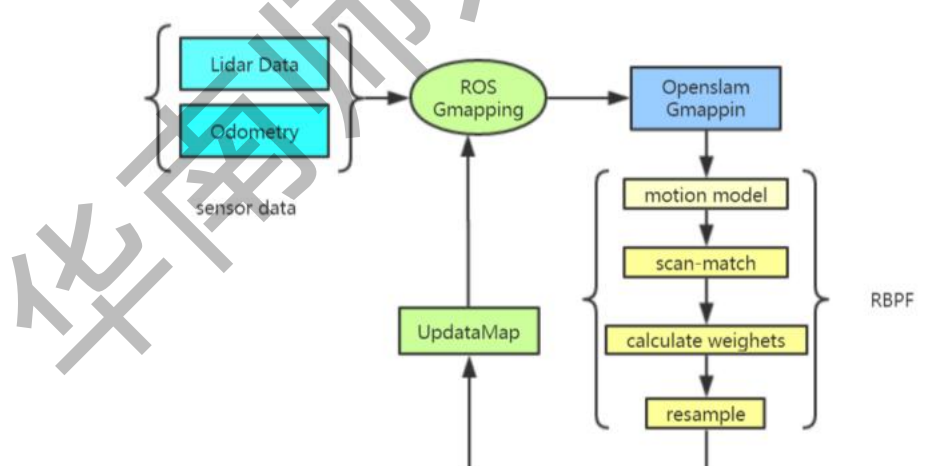
- 1、获取 gmapping 功能包。
- 2、在配置好各种传感器的基础上，运行 gmapping 节点进行建图。
- 3、了解 gmapping 算法各参数的作用。

实验器材：

ROS 智能车、激光雷达、IMU、编码器、电脑。

实验原理：

本实验我们采用 gmapping 算法进行建图，gmapping 是一个基于 2D 激光雷达使用 RBPF（Rao-Blackwellized Particle Filters）算法完成二维栅格地图构建的 SLAM 算法。gmapping 算法可以实时构建室内环境地图，在小场景中计算量少，且地图精度较高，对激光雷达扫描频率要求较低。



(图 4-1)

以下是对其各方面的简要介绍：

①基本原理：gmapping 算法基于概率滤波原理，使用粒子滤波器（Particle Filter）来估计机器人的位姿和地图。它通过结合机器人的运动信息和传感器数

据，如激光雷达数据，来进行位姿估计和地图建立。

②建图过程：gmapping 算法将环境表示为一个栅格地图（Occupancy Grid Map），其中每个栅格表示一个空间单元的状态（占用或空闲）。算法通过将激光雷达数据与当前地图进行匹配，更新栅格地图的状态。

③定位过程：gmapping 算法使用粒子滤波器来估计机器人的位姿。初始时，通过均匀采样生成一组粒子，每个粒子代表一个可能的位姿。随着机器人移动和传感器观测的更新，粒子的权重会进行调整，以反映位姿的可信度。然后，根据粒子的权重重新采样，使得位姿估计更加准确。

④数据关联：gmapping 算法使用激光雷达数据与地图进行数据关联，以确定每个粒子的权重。它使用一种称为匹配度计算（Matching Measure）的方法，通过比较激光雷达测量值与地图上相应位置的期望测量值的相似度，来计算权重。

⑤地图优化：为了提高地图的精度和准确性，gmapping 算法使用了一个称为 Rao-Blackwellized 粒子滤波（Rao-Blackwellized Particle Filter）的技术。该技术将地图表示与机器人位姿分离，使得可以针对不同的机器人位姿进行地图更新和优化。

实验步骤：

<1> 获取 gmapping 功能包。

获取 gmapping 功能包有从包安装和从源码编译两种方式，在本实验中，我们采用的方式是从包安装。

连接互联网，并在终端中输入以下命令：

```
sudo apt install ros-noetic-gmapping
```

随后输入密码，安装 gmapping 功能包。

<2> 编写 launch 用于启动 gmapping 节点。

编写 launch 文件，内容如下：

```
<?xml version="1.0"?>
<launch>
  <arg name="use_rviz" default="true" />
  <!-- gmapping -->
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
    <remap from="scan" to="scan"/>
    <param name="map_update_interval" value="0.4"/>
    <param name="maxUrange" value="3"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
  </node>
</launch>
```

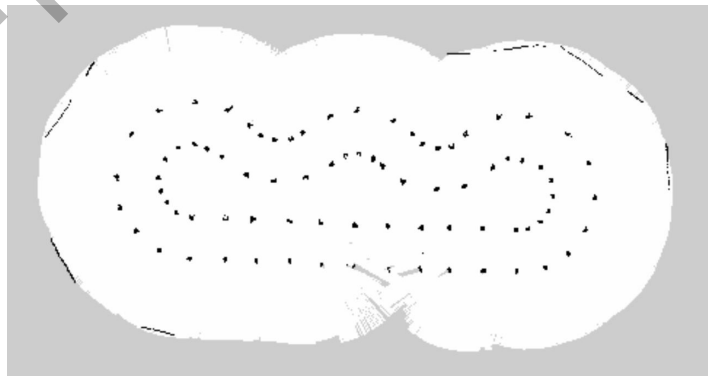
```

<param name="iterations" value="5"/>
<param name="lsigma" value="0.075"/>
<param name="ogain" value="3.0"/>
<param name="lskip" value="0"/>
<param name="srr" value="0.1"/>
<param name="srt" value="0.2"/>
<param name="str" value="0.1"/>
<param name="stt" value="0.2"/>
<param name="linearUpdate" value="0.11"/>
<param name="angularUpdate" value="0.25"/>
<param name="temporalUpdate" value="1.0"/>
<param name="resampleThreshold" value="0.25"/>
<param name="particles" value="8"/>
<param name="xmin" value="-10.0"/>
<param name="ymin" value="-10.0"/>
<param name="xmax" value="10.0"/>
<param name="ymax" value="10.0"/>
<param name="delta" value="0.05"/>
<param name="llsamplerange" value="0.01"/>
<param name="llsamplestep" value="0.01"/>
<param name="lasamplerange" value="0.005"/>
<param name="lasamplestep" value="0.005"/>
<param name="odom_frame" value="odom"/>
<param name="base_frame" value="base_footprint"/>
<param name="minimumScore" value="0"/>
</node>
</launch>

```

该文件的作用为启动 gmapping 节点。注意，若未配置好雷达，建图将无法进行，若未配置好里程计，则建图效果可能会受影响。

当我们配置好各传感器，可以获取雷达点云、里程计等信息后，启动该文件即可开始建图，我们可以使用 rviz 查看建图情况。建图较好的情况如下：



(图 4-2)

我们可以使用以下命令将地图保存到本地：

```
rosrun map_server map_saver -f ~/car2023_ws/src/racecar/map/mymap
```

并且可以自由调整保存的路径

<3> 了解 gmapping 的各参数作用。

gmapping 具有以下参数:

```
·throttle_scans (int, default: 1), 处理的扫描数据门限, 默认每次处理 1 个扫描数据
·base_frame (string, default:"base_link"), 机器人底盘坐标系
·map_frame (string, default:"map"), 地图坐标系
·odom_frame (string, default:"odom"), 里程计坐标系
·maxRange (float), 传感器有效范围
·maxUrange(float), 传感器最大范围
·map_update_interval (float, default: 5.0), 地图更新频率
·maxUrange (float, default: 80.0), 探测最大可用范围
·sigma (float, default: 0.05), endpoint 匹配标准差
·kernelSize (int, default: 1), 用于查找对应的 kernel size
·kernel_size_lstep (float, default: 0.05), 平移优化步长
·astep (float, default: 0.05), 旋转优化步长
·iterations (int, default: 5), 扫描匹配迭代步数
·lsigma (float, default: 0.075), 用于扫描匹配概率的激光标准差
·ogain (float, default: 3.0), 似然估计为平滑重采样影响使用的 gain
·lskip (int, default: 0), 每次扫描跳过的光束数.
·minimumScore (float, default: 0.0), 激光雷达信息置信度, 越高则越依赖激光雷达进行建图
·srr (float, default: 0.1), 平移时里程误差作为平移函数
·srt (float, default: 0.2), 平移时的里程误差作为旋转函数
·str (float, default: 0.1), 旋转时的里程误差作为平移函数
·stt (float, default: 0.2), 旋转时的里程误差作为旋转函数
·linearUpdate (float, default: 1.0), 机器人移动某个距离处理一次扫描数据
·angularUpdate (float, default: 0.5), 机器人每旋转某个角度处理一次扫描数据
·temporalUpdate (float, default: -1.0), 如果最新扫描处理比更新慢则处理 1 次扫描, 为负数时候关闭基于时间的更新
·resampleThreshold (float, default: 0.5), 基于重采样门限的 Neff
·particles (int, default: 30), 滤波器中粒子数目
·xmin (float, default: -100.0), 地图初始尺寸
·ymin (float, default: -100.0), 地图初始尺寸
·xmax (float, default: 100.0), 地图初始尺寸
·ymax (float, default: 100.0), 地图初始尺寸
·delta (float, default: 0.05), 地图分辨
·llsamplerange (float, default: 0.01), 于似然计算的平移采样距离
·llsamplestep (float, default: 0.01), 用于似然计算的平移采样步长
·lasamplerange (float, default: 0.005), 用于似然计算的角度采样距离
```

```
·lasamplestep (float, default: 0.005), 用于似然计算的角度采样步长  
·transform_publish_period (float, default: 0.05), 变换发布时间间隔  
·occ_thresh (float, default: 0.25), 栅格地图栅格值
```

其中，对建图效果影响较大的参数有：

- maxRange, 该值越小时，建图处理的信息量越少，建图过程越流畅，但越容易发生定位漂移；
- particles , 根据粒子滤波的原理，该值增大时会显著增加算法的运算量， 但可以提高建图的准确性。

一般来讲，在算力有限的情况下，我们需要使 maxRange 和 particles 都尽量小，以获得较高的建图流畅性。值得注意的是，这两个参数过小时也会导致建图失败，需要依据实际情况进行调整。

实验五 导航点获取

实验目的:

- 1、学习 ROS 中 tf 包
- 2、掌握 gmapping 建图发布的坐标变换信息
- 3、获取 map 坐标系下 base_link 所在位置的位姿

实验内容:

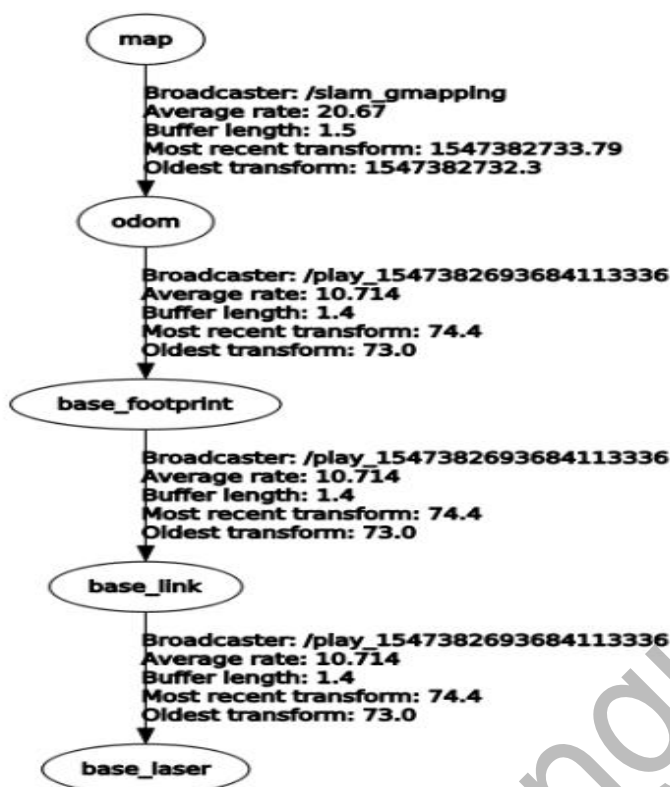
- 1、查找 gmapping 发布的坐标变换信息
- 2、利用 tf 包实现监听 base_link 坐标系与 map 坐标系变换
- 3、存储获取的 map 坐标系下 base_link 所在位置的位姿

实验器材:

ROS 智能车、激光雷达、电脑。

实验原理:

tf 包是一个坐标变换的工具包，提供了坐标转换等方面的功能。在 gmapping 建图时，会有五个坐标系，分别为雷达坐标系，base_footprint 坐标系（智能车底盘为原点的坐标系），base_link 坐标系（智能车中心为原点的坐标系），odom（里程计）坐标系和 map 坐标系（建图起始位置为原点）。当我们知道这五个坐标系之间的变换关系后，如图 5-1 所示，就能通过 tf 坐标变换获得 map 坐标系下 base_link 所在位置的坐标。



(图 5-1)

实验步骤:

<1> 查找 gmapping 发布的坐标变换信息

可以上 <http://wiki.ros.org/gmapping> 网站了解 gmapping 节点所发布的坐标变换。如图 5-2 所示，gmapping 发布 map 坐标系到 odom 坐标系的坐标变换，再加上 car_tf.launch 文件发布的 base_footprint 和 base_link 的坐标变换，base_link 和雷达的坐标变换，里程计发布的 base_footprint 和 odom 的坐标变换，我们可以利用 tf 包获取 map 和 base_link 的坐标变换。

4.1.6 Provided tf Transforms

map → odom

the current estimate of the robot's pose within the map frame

(图 5-2)

<2> 利用 tf 包实现监听 base_link 坐标系与 map 坐标系变换

首先创建 tf 监听器，如图 5-3 所示

```
try:
    self.tf_listener.waitForTransform('/map', '/base_link', rospy.Time(), rospy.Duration(1.0))

except (tf.Exception, tf.ConnectivityException, tf.LookupException):
    return
```

(图 5-3)

其中 tf 监听器监听 base_link 到 map 坐标系的坐标变换

然后编写函数，通过 tf 监听器获取四元数与坐标，如图 5-4 所示

```
def get_pos(self):
    try:
        (trans, rot) = self.tf_listener.lookupTransform('/map', '/base_link', rospy.Time(0))
        # rospy.Time(0)
    except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
        rospy.loginfo("tf Error")
        return None
    return trans[0],trans[1],rot[2],rot[3]
```

(图 5-4)

其中 trans 为 tf 监听器监听 base_link 到 map 坐标系的坐标变换返回的坐标，rot 为返回的四元数。智能车在赛道行驶，只有 x, y 坐标发生变化，四元数 z, w 发生变化，所以我们只用记录 x, y, z, w 就能确定智能车的位姿。该函数的返回值 trans[0], trans[1], rot[2], rot[3] 便为 x, y, z, w。

<3> 存储获取的 map 坐标系下 base_link 所在位置的位姿

在第一圈时，需要获取智能车在赛道中的位姿作为第二圈的导航点，因此需要记录获取的位姿。如图 5-5 所示，我们用 csv 文件进行记录，并用欧式距离每隔 L 米记录一次位姿

```
if __name__ == "__main__":
    csv_dir = "nav_point.csv" #路径
    with open(csv_dir, 'w', encoding='utf8', newline='') as f:
        csv_write = csv.writer(f, delimiter=',', quoting=csv.QUOTE_MINIMAL)
        nav_point = Get_Point()
        r = rospy.Rate(10)
        last_x=0
        last_y=0
        flg=0 #记录智能车在原点的位姿标志
        l=1.5
        while not rospy.is_shutdown():
            if ((x-last_x)**2+(y-last_y)**2)**(0.5)>=l or flg==0: #欧式距离记录点
                csv_write.writerow([x,y,z,w])
                flg=1
                last_x=x
                last_y=y
            r.sleep()
```

(图 5-5)

<4> 运行

以上代码编写在 get_point.py 中，要运行该脚本，首先运行 Run_car.launch 文件和 Run_gmapping.launch 文件，然后运行雷达寻路代码或者用键盘控制智能车运动，再运行该脚本便可以实现记录导航点功能。

实验六 循迹与导航的切换

实验目的:

- 1、掌握用代码运行 launch 文件和关闭节点
- 2、掌握线程的创建方法
- 3、实现循迹和导航的自动切换

实验内容:

- 1、使用 roslaunch 模块和 rosnode 模块运行 launch 文件和关闭节点
- 2、使用 threading 模块开启线程
- 3、编写代码控制循迹和导航的自动切换

实验器材:

ROS 智能车、激光雷达、电脑。

实验原理:

第一圈主要需要运行 gmapping 和雷达循迹，而导航主要需要开启 amcl 和 move base 等节点构成的 launch 文件。考虑到智能车算力资源，不能同时开启建图和导航，因此需要编写脚本实现循迹和导航切换。launch 文件中包含多个节点，一个节点运行相当开启一个进程，因此运行 launch 文件相当开多个进程。可以在脚本中使用 roslaunch 和 rosnode 模块实现对 launch 文件和节点的开启和关闭。同时，该脚本需要处理好雷达循迹和导航发布的控制信息。这要求在脚本进程中开一个子线程订阅雷达循迹还有导航发布控制信息的话题，再由该线程直接发最终的控制信息。

实验步骤:

<1> 使用 roslaunch 模块和 rosnode 模块运行 launch 文件和关闭节点

roslaunch 模块可以实现 launch 文件的运行和关闭，代码如下图 6-1 所示，为了方便代码复用，可以写成类的形式。

```
class Run_launch():
    def __init__(self, launch_dir):
        self.uid=roslaunch.rlutil.get_or_generate_uid(None, False)
        roslaunch.configure_logging(self.uid)
        self.tracking_launch = roslaunch.parent.ROSLaunchParent(self.uid, launch_dir) # launch 文件路径

    def start(self): # 运行launch文件
        self.tracking_launch.start()

    def shutdown(self): # 关闭launch文件
        self.tracking_launch.shutdown()
```

(图 6-1)

在实例化 Run_launch 类时，需要传入 launch 文件的路径，使用 start() 方法运行 launch 文件，shutdown 方法关闭文件。

rostopic 模块可以用于关闭在运行的节点。导航开启后，为了减少资源开销，可以用 rostopic.killnode(["节点名"]) 关闭节点

<2> 协调雷达循迹和导航发布的控制信息

遇到红绿灯需要停车或者第二圈遇到停车点需要停车，因此需要协调好雷达循迹和导航发布的控制信息，需要停车时，要立刻发停车控制信息，在切换时，需要使用导航发布的控制信息。考虑到上述要求，可以在脚本进程订阅雷达循迹还有导航发布控制信息的话题，再发最终的控制信息。如图 6-2 所示，可以创建类来实现

```
class Judgement():
    def __init__(self):
        self.laser_controller=rospy.Subscriber(
            "laser_control", Twist , self.laser_control_info, queue_size=10) # 订阅雷达控制信息
        self.nav_controller=rospy.Subscriber(
            "nav_control", Twist , self.nav_control_info, queue_size=10) # 订阅导航控制信息
        self.pub = rospy.Publisher(
            "/car/cmd_vel", Twist, queue_size=10) # 发布最终的控制信息
        self.control_info=Twist()
        self.state="laser"

    def laser_control_info(self,data):
        if self.state=="laser":
            self.control_info.linear.x=data.linear.x
            self.control_info.angular.z=data.angular.z
            self.pub.publish(self.control_info)
        elif self.state=="stop":
            self.control_info.linear.x=1500
            self.control_info.angular.z=90
            self.pub.publish(self.control_info)

    def nav_control_info(self,data):
        if self.state=="nav":
            self.control_info.linear.x = data.linear.x
            self.control_info.angular.z = data.angular.z
            self.pub.publish(self.control_info)
        elif self.state=="stop":
            self.control_info.linear.x=1500
            self.control_info.angular.z=90
            self.pub.publish(self.control_info)
```

(图 6-2)

在 Judgement 类中，用有限状态机的思想，通过定义 state 来说明当前是使用雷达控制信息，导航控制信息还是停车，state 的变化通过红绿灯和停车点来决定。

<3> 创建线程

实验步骤二中的 state 的改变依赖于红绿灯和停车点，因此需要再脚本中增加 state 转移的代码，而订阅话题需要一直接收并用回调函数处理，这两段程序需要并发运行。为了便于信息共享，可以开子线程用于处理订阅话题的回调函数。首先，现在 Judgement 中增加一个实例方法，如图 6-3 所示

```
def spin(self):
    rospy.spin()
```

(图 6-3)

其中 `rospy.spin()` 函数用于处理订阅话题的回调函数。

在 python 中, 可以使用 `threading` 模块创建线程, 具体代码如图 6-4 所示

```
judge=Judgement() # 实例化对象
t1=threading.Thread(target = judge.spin) # 创建线程, 该线程执行judge.spin()
t1.start() # 开启线程
```

(图 6-4)

<4> 编写代码控制循迹和导航的自动切换

```
rospy.init_node('main')
rviz=Run_launch(["/home/scnu-car/vanguardcar/src/racecar/launch/rviz.launch"])
gmapping=Run_launch(["/home/scnu-car/vanguardcar/src/racecar/launch/Run_gmapping.launch"])
amcl=Run_launch(["/home/scnu-car/vanguardcar/src/racecar/launch/amcl_nav_teb.launch"])
map_save=Run_launch(["/home/scnu-car/vanguardcar/src/racecar/launch/map_save.launch"])
map_adv=Run_launch(["/home/scnu-car/vanguardcar/src/racecar/launch/map_adv.launch"])
gmapping.start() # 开建图
rviz.start() # 开rviz
initial_pose_pub = rospy.Publisher('initialpose', PoseWithCovarianceStamped, queue_size=1) # 开导航后用于发布初始位置
pose_msg = PoseWithCovarianceStamped()
judge=Judgement() # 实例化对象
t1=threading.Thread(target = judge.spin) # 创建线程, 该线程执行judge.spin()
t1.start() # 开启线程
t2=threading.Thread(target = loop_run) # 创建线程, 该线程执行judge.spin()
t2.start() # 开启线程
csv_dir = "nav_point.csv"
with open(csv_dir,'w',encoding='utf8',newline='') as f :
    csv_write = csv.writer(f, delimiter=',', quoting=csv.QUOTE_MINIMAL)
    nav_point = Get_Point(csv_write)
    r= rospy.Rate(10)
    last_x=0
    last_y=0
    flg=0
    while not rospy.is_shutdown():
        if judge.state=="laser":
            x,y,z,w=nav_point.get_pos()
            if judge.is_red_green_light==1: # 红绿灯停车
                judge.state="stop"
                flg=0
                f.close()
                map_save.start()
                rospy.sleep(1)
                amcl.start()
```

(图 6-5)

```

    rospy.sleep(1)
    pose_msg.header.frame_id = "map"
    pose_msg.pose.pose.position.x = x
    pose_msg.pose.pose.position.y = y
    pose_msg.pose.covariance[0] = 0.25
    pose_msg.pose.covariance[6 * 1 + 1] = 0.25
    pose_msg.pose.covariance[6 * 5 + 5] = 0.06853892326654787
    pose_msg.pose.pose.orientation.z = z
    pose_msg.pose.pose.orientation.w = w
    initial_pose_pub.publish(pose_msg)
    t2.start()
    judger.state="nav"
    rosnode.kill_nodes(["slam_gmapping"]) # 关闭建图
    rosnode.kill_nodes(["laser_go"]) # close laser_control

    elif ((x-last_x)**2+(y-last_y)**2)**(0.5)>=1 or flg==0: #欧式距离采点值
        csv_write.writerow([x,y,z,w])
        flg=1
        last_x=x
        last_y=y
    r.sleep()
    amcl.shutdown()
    map_save.shutdown()

```

(图 6-6)

当识别到红绿灯时，就停下，保存建好的地图，运行导航 launch 文件，发布停下时车的位姿，最后切换到导航状态，开启导航。该脚本命名为 auto_transform.py, 运行该脚本时需要打开红绿灯识别，雷达循迹。

实验七 自主导航

实验目的:

- 1、掌握 move_base 算法导航的基本使用方法
- 2、了解 DWA 和 TEB 局部运动规划的优缺点
- 3、熟悉导航中的各种参数及调参方法

实验内容:

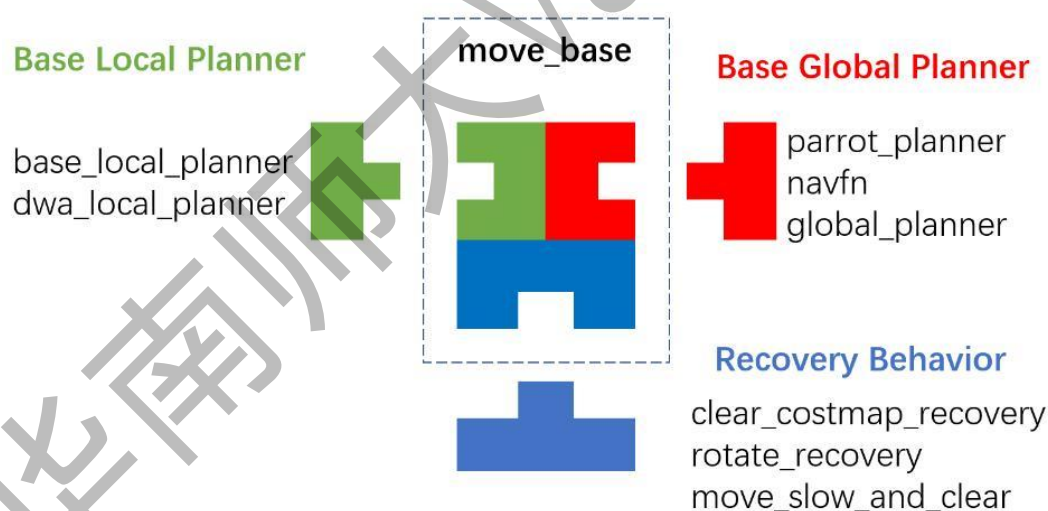
- 1、切换为 Teb 局部运动规划算法
- 2、多点导航

实验器材:

ROS 车、电脑

实验原理:

move_base 算法是 ROS (Robot Operating System) 中常用的路径规划和导航算法之一。它基于全局路径规划和局部避障两个核心模块。



move_base 组成 图(7-1)

①全局路径规划: move_base 使用代价地图 (costmap) 来表示环境中的障碍物信息, 通过算法 (如 Dijkstra、A*等) 在这个代价地图上计算出全局路径, 以使机器人从起点移动到目标点。

②局部避障: move_base 还包含一个局部避障模块, 该模块使用代价地图中的障碍物信息, 结合机器人当前状态 (位置、姿态等) 和传感器数据, 计算出机器人在局部区域内的最优运动指令, 以避免障碍物。

move_base 算法的优点是可以处理复杂的环境和全局路径规划, 同时提供局部避障能力。它适用于需要长时间导航和全局路径规划的场景, 如移动机器人在大型室内环境中的导航任务。

DWA (Dynamic Window Approach):动态窗口法是一种在机器人路径规划中常用的方法。它基于机器人在速度和转向速度空间中的“动态窗口”，通过考虑机器人的运动动力学和环境的障碍物来选择最佳的速度和转向速度组合。DWA 旨在快速生成机器人的安全、平滑的运动轨迹。

优势:在速度和转向速度空间中进行搜索,适用于具有较低运动约束的情况。实时性较好,适用于快速移动的场景。

TEB (Timed Elastic Band):时间弹性带法是另一种用于路径规划的方法。TEB 方法将路径规划问题建模为一个在时间上具有弹性的带状结构。它考虑了机器人的动力学约束、避障和全局路径规划,以生成时间上连续和光滑的运动轨迹。TEB 的目标是兼顾路径的质量和执行的时间。

优势:能够生成时间上连续和平滑的运动轨迹。考虑了更复杂的动力学约束,适用于具有较高运动约束和需要更精细控制的场景。

TEB 在运动过程中会调整自己的位姿朝向,当到达目标点时,通常机器人的朝向也是目标朝向而不需要旋转。DWA 则是先到达目标坐标点,然后原地旋转到目标朝向。在我们这个赛项上,车的底盘结构为阿克曼底盘,很显然使用 TEB 更合适。

导航流程:

①自动发布导航点:在先前建好地图的基础上,提前设置导航点,并在导航时通过一个节点进行发布。

②全局路径规划:基于地图和机器人当前位置,使用路径规划算法(如 A* 算法)计算出一条从起点到目标点的全局路径。

③局部路径规划:基于机器人当前位置和局部感知信息,使用 DWA 算法、TEB 算法或其他规划算法,生成机器人的局部路径,并考虑避障。

④运动控制:根据当前位置和局部路径,通过简单的 PID 控制,向下位机发送指令,控制机器人的执行器(电机和舵机),实现机器人的自主导航。

实验步骤:

<1> 安装 navigation 功能包

连接互联网,在终端输入以下命令:

```
sudo apt install ros-noetic-navigation
```

输入密码,即可安装 navigation 功能包。

<2> 配置 move_base 参数

此处我们使用 NavfnROS 作为全局路径规划算法,使用 DWAPlanerROS 作为局部路径规划算法,参数配置需要编写五个文件,其内容如下:

- global_costmap_params.yaml:

```
global_costmap:
  footprint: [[-0.305, -0.18], [-0.305, 0.18], [0.305, 0.18],
[0.305,-0.18]]
  footprint_padding: 0.01
  transform_tolerance: 0.5
```



```

update_frequency: 5.0
publish_frequency: 5.0

global_frame: map
robot_base_frame: base_footprint
resolution: 0.1
rolling_window: true
width: 15.0
height: 15.0
track_unknown_space: false
plugins:
  - {name: static, type:
"costmap_2d::StaticLayer"}
  - {name: sensor, type: "costmap_2d::ObstacleLayer"}
  - {name: inflation, type: "costmap_2d::InflationLayer"}
static:
  map_topic: /map
  subscribe_to_updates: true
sensor:
  observation_sources: laser_scan_sensor
  laser_scan_sensor: {sensor_frame: laser_link, data_type:
LaserScan, topic: scan, marking: true, clearing: true}
inflation:
  inflation_radius: 0.4
  cost_scaling_factor: 20.0

```

• local_costmap_params.yaml:

```

local_costmap:
  footprint: [[-0.305, -0.18], [-0.305, 0.18], [0.305, 0.18],
[0.305,-0.18]]
  footprint_padding: 0.01
  transform_tolerance: 0.5
  update_frequency: 10.0
  publish_frequency: 10.0

  global_frame: map
  robot_base_frame: base_footprint
  resolution: 0.1
  rolling_window: true
  width: 10
  height: 10
  resolution: 0.05
  track_unknown_space: false
  plugins:
    - {name: sensor, type: "costmap_2d::ObstacleLayer"}

```

```

- {name: inflation, type: "costmap_2d::InflationLayer"}
  sensor:
    observation_sources: laser_scan_sensor
    laser_scan_sensor: {sensor_frame: laser_link, data_type:
LaserScan, topic: scan, marking: true, clearing: true}
    inflation:
      inflation_radius: 0.35
      cost_scaling_factor: 16.0

```

• base_global_planner_params.yaml:

```

GlobalPlanner:
  allow_unknown: false
  default_tolerance: 0.5      #路径规划器目标点的公差范围
# visualize_potential: true  #指定是否通过可视化 PointCloud2 计算的潜
在区域
  use_dijkstra: false        #如果为 true, 则使用 dijkstra 算法。 否
则使用 A *算法
  use_quadratic: true         #二次逼近
  use_grid_path: false        #如果为 true, 沿着栅格边界创建路径。 否则,
使用梯度下降的方法。
#   old_navfn_behavior: true   #如果你想要 global_planner 准确反映
navfn 的行为, 此项设置为 true。

```

```

  # lethal_cost: 253
  # neutral_cost: 66
  # cost_factor: 0.55
#   publish_potential: True

```

```

NavfnROS:
  allow_unknown: false

```

• dwa_local_planner_params.yaml:

```

DWAPlannerROS:

  acc_lim_th: 1
  acc_lim_x: 1
  acc_lim_y: 1
  max_vel_x: 1
  min_vel_x: 1
  max_vel_y: 0
  min_vel_y: 0
  max_vel_trans: 1
  min_vel_trans: 1
  # max_rot_vel: 1
  # min_rot_vel: 1
  max_vel_theta: 1
  min_vel_theta: 1

```

```

sim_time: 3
sim_granularity: 0.1
goal_distance_bias: 45.0
path_distance_bias: 30.0
occdist_scale: 0.1
forward_point_distance: 0.5
stop_time_buffer: 0.4
#####
  scaling_speed: 0.5
#####
  max_scaling_factor: 0.2
  oscillation_reset_dist: 0.05

  #
  vx_samples: 40
  vy_samples: 0
  vtheta_samples: 10

  theta_stopped_vel: 0.01
  trans_stopped_vel: 0.01
  #
  xy_goal_tolerance: 0.15
  yaw_goal_tolerance: 0.07
  latch_xy_goal_tolerance: true #如果目标误差被锁定，若机器人达到目标 xy
位置，它将旋转到位，即使误差没有达到，也会做旋转

```

• move_base_params.yaml:

```

# Move base node parameters. For full documentation of the parameters
in this file, please see
#
# http://www.ros.org/wiki/move\_base
#
shutdown_costmaps: false

controller_frequency: 10.0
controller_patience: 3.0
planner_frequency: 10.0
planner_patience: 10.0
oscillation_timeout: 10.0
oscillation_distance: 0.2
clearing_rotation_allowed: false

```

<3> 编写 launch 文件开启导航算法

编写 launch 文件如下:

```
<?xml version="1.0"?>
```



```

<launch>
  <arg name="use_rviz" default="true" />
  <!-- for amcl -->
  <arg name="init_x" default="0.0" />
  <arg name="init_y" default="0.0" />
  <arg name="init_a" default="0.0" />
  <!-- Map server -->
  <node name="map_server" pkg="map_server" type="map_server"
args="$(find racecar)/map/mymap.yaml"/>
  <!-- wheel odometry -->
  <include file="$(find
encoder_driver)/launch/wheel_odom.launch"/>
  <!-- Localization -->
  <node pkg="robot_localization" type="ekf_localization_node"
name="ekf_se" clear_params="true">
    <rosparam command="load" file="$(find
racecar)/param/ekf_params.yaml" />
  </node>
  <!-- AMCL -->
  <include file="$(find
racecar)/launch/includes/amcl.launch.xml">
    <arg name="init_x" value="$(arg init_x)"/>
    <arg name="init_y" value="$(arg init_y)"/>
    <arg name="init_a" value="$(arg init_a)"/>
  </include>
  <!-- Navstack -->
  <node pkg="move_base" type="move_base" respawn="false"
name="move_base">
    <!-- local planner -->

    <param name="base_global_planner"
value="navfn/NavfnROS"/>
    <param name="base_local_planner"
value="dwa_local_planner/DWAPlannerROS"/>
    <rosparam file="$(find
racecar)/param/dwa_local_planner_params.yaml" command="load"/>

    <!-- costmap layers -->
    <rosparam file="$(find
racecar)/param/local_costmap_params.yaml" command="load"/>
    <rosparam file="$(find
racecar)/param/global_costmap_params.yaml" command="load"/>
    <!-- move_base params -->

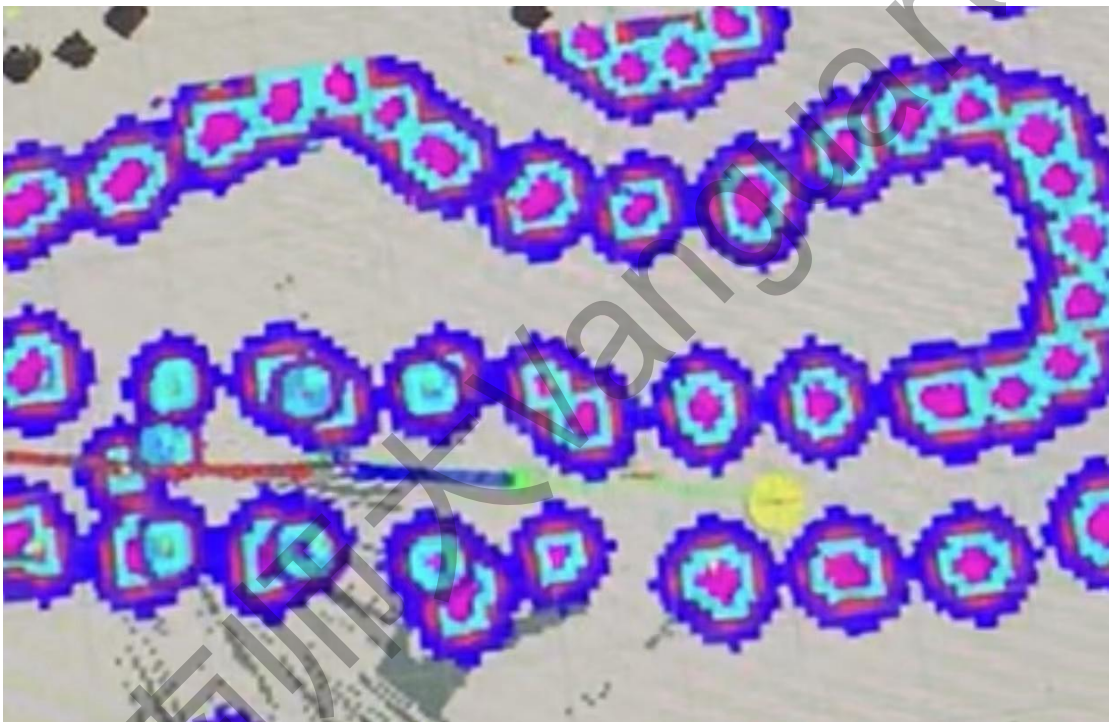
```

```

    <rosparam file="$(find
racecar)/param/base_global_planner_params.yaml" command="load"/>
    <rosparam file="$(find
racecar)/param/move_base_params.yaml" command="load"/>
  </node>
  <!-- Rviz -->
  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
racecar)/rviz/amcl.rviz" if="$(arg use_rviz)" />
</launch>

```

其中，需注意各文件的路径问题。运行该文件，在 rviz 界面修正小车的起始位置后发布导航点，即开始进行路径规划。如图：



(图 7-2)

上图中，绿色细线即为规划出的路径。我们可以选择使用 rviz 中的 2D goal point (紫色箭头) 发布导航点，也可以选择使用程序进行自动发送。

在导航时，提前开启控制器，可以实现按照规划路径的导航。我们可以在编写的 launch 文件中加入以下代码：

```

<node pkg="racecar" type="car_controller_new" respawn="false"
name="car_controller">

  <param name="Vcmd" value="1.5" /> <!--speed of car
m/s    1.5    -->
  <!-- ESC -->
  <param name="baseSpeed" value="200"/>
  <param name="baseAngle" value="0.0"/>
  <param name="Angle_gain_p" value="-5.5"/>

```

```

<param name="Angle_gain_d" value="-3.0"/>
<param name="Lfw" value="1.8"/>
<param name="vp_max_base" value="400"/>
<param name="vp_min" value="400"/>
<param name="goalRadius" value="0.35"/>

</node>

```

随后，我们便可以实现自主导航。

值得注意的是，有许多参数会影响导航效果，需要读者自行调整。

<4> 局部路径规划更换为 TEB

上面所使用的 DWA 算法并不适用于我们所使用的阿克曼底盘 ROS 智能车，故需要将其更换为可以适用于阿克曼底盘的 TEB 算法。

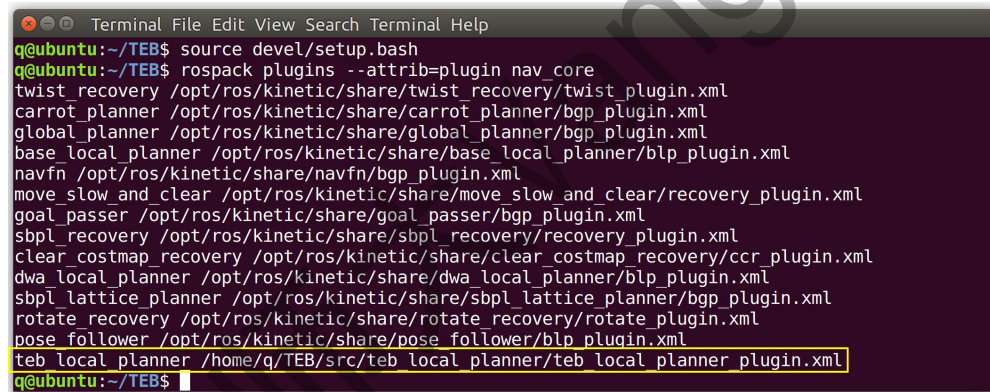
由于 ROS 的 navigation 不自带 TEB 算法，需要自己安装相关的功能包。连接互联网，在终端中输入下面这条命令：

```
sudo apt install ros-noetic-teb-local-planner
```

安装完后重新 source 一遍工作空间，通过下面命令检查是否安装成功：

```
rospack plugins --attrib=plugin nav_core
```

若出现下列框中字样，则安装成功



```

q@ubuntu:~/TEB$ source devel/setup.bash
q@ubuntu:~/TEB$ rospack plugins --attrib=plugin nav_core
twist_recovery /opt/ros/kinetic/share/twist_recovery/twist_plugin.xml
carrot_planner /opt/ros/kinetic/share/carrot_planner/bgp_plugin.xml
global_planner /opt/ros/kinetic/share/global_planner/bgp_plugin.xml
base_local_planner /opt/ros/kinetic/share/base_local_planner/blk_plugin.xml
navfn /opt/ros/kinetic/share/navfn/bgp_plugin.xml
move_slow_and_clear /opt/ros/kinetic/share/move_slow_and_clear/recovery_plugin.xml
goal_passer /opt/ros/kinetic/share/goal_passer/bgp_plugin.xml
sbpl_recovery /opt/ros/kinetic/share/sbpl_recovery/recovery_plugin.xml
clear_costmap_recovery /opt/ros/kinetic/share/clear_costmap_recovery/ccr_plugin.xml
dwa_local_planner /opt/ros/kinetic/share/dwa_local_planner/blk_plugin.xml
sbpl_lattice_planner /opt/ros/kinetic/share/sbpl_lattice_planner/bgp_plugin.xml
rotate_recovery /opt/ros/kinetic/share/rotate_recovery/rotate_plugin.xml
pose_follower /opt/ros/kinetic/share/pose_follower/blk_plugin.xml
teb_local_planner /home/q/TEB/src/teb_local_planner/teb_local_planner plugin.xml
q@ubuntu:~/TEB$

```

TEB 成功安装 图(7-3)

要使用 TEB 算法，还需要配置参数、更改 launch 文件，流程如下：

- 编写 TEB 相关参数文件内容如下：

```

TebLocalPlannerROS:

odom_topic: odom

# Trajectory

teb_autosize: True
dt_ref: 0.3
dt_hysteresis: 0.1
max_samples: 500
global_plan_overwrite_orientation: True
#allow_init_with_backwards_motion: True

```

```

allow_init_with_backwards_motion: True
#max_global_plan_lookahead_dist: 3.0
max_global_plan_lookahead_dist: 2.0 #向前规划
global_plan_viapoint_sep: -1
global_plan_prune_distance: 1
exact_arc_length: False
feasibility_check_no_poses: 2
publish_feedback: False

# Robot

max_vel_x: 0.4 #速度约束 racecar/src/multi_goal/src
max_vel_x_backwards: 0.2 #最大倒车速度不能设置为 0
max_vel_y: 0
max_vel_theta: 0.6 #角速度约束 the angular velocity is also
boundedbymin_turning_radius in case of a carlike robot ( $r = v / \omega$ )
acc_lim_x: 0.5 #加速度约束
acc_lim_theta: 0.6 #角加速度约束
# ***** Carlike robot parameters
*****
min_turning_radius: 0.3 #最小转向半径 Min turning radius of the
carlike robot (compute value using a model or adjust with
rqt_reconfigure manually)
#wheelbase: 0.555 # Wheelbase of our robot
wheelbase: 0.4
#cmd_angle_instead_rotvel: True # stage simulator takes the
angleinstead of the rotvel as input (twist message)
cmd_angle_instead_rotvel: True
#
*****
***
footprint_model: # types: "point", "circular", "two_circles",
"line", "polygon"
  type: "line"
  radius: 0.3 # for type "circular"
  line_start: [0.0, 0.0] # for type "line"
  line_end: [0.5, 0.0] # for type "line"
  front_offset: 0.2 # for type "two_circles"
  front_radius: 0.2 # for type "two_circles"
  rear_offset: 0.2 # for type "two_circles"
  rear_radius: 0.2 # for type "two_circles"
  vertices: [ [0.25, -0.05], [0.18, -0.05], [0.18, -0.18],
[-0.19, -0.18], [-0.25, 0], [-0.19, 0.18], [0.18, 0.18], [0.18,
0.05], [0.25, 0.05] ] # for type "polygon"

```

```

# GoalTolerance

xy_goal_tolerance: 0.2
yaw_goal_tolerance: 0.1
free_goal_vel: False
complete_global_plan: True

# Obstacles

#min_obstacle_dist: 0.45 # This value must also include our
robot's expansion, since footprint_model is set to "line".
min_obstacle_dist: 0.27 #与障碍物期望最小距离
# inflation_dist: 1.0
inflation_dist: 0.6 #障碍物周围的缓冲区 要比 60 行大才有用
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.0
obstacle_poses_affected: 15
dynamic_obstacle_inflation_dist: 0.6
include_dynamic_obstacles: True
costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5
# Optimization

no_inner_iterations: 5
no_outer_iterations: 4
optimization_activate: True
optimization_verbose: False
penalty_epsilon: 0.1 #为速度约束提供缓冲效果
obstacle_cost_exponent: 4
# weight_max_vel_x: 2
weight_max_vel_x: 2 #最大速度权重
weight_max_vel_theta: 1
weight_acc_lim_x: 1
weight_acc_lim_theta: 1
weight_kinematics_nh: 1000
weight_kinematics_forward_drive: 1 #倒车惩罚
#weight_kinematics_forward_drive: 1000
#weight_kinematics_turning_radius: 5
weight_kinematics_turning_radius: 1 #最小转向半径权重
#weight_optimaltime: 0.5 # must be > 0
weight_optimaltime: 1 #直道加速
weight_shortest_path: 0
#weight_obstacle: 270

```

```

weight_obstacle: 100
#weight_inflation: 0.1
weight_inflation: 0.2
weight_dynamic_obstacle: 10 # not in use yet
weight_dynamic_obstacle_inflation: 0.2
#weight_viapoint: 1000
weight_viapoint: 1
weight_adapt_factor: 2
# Homotopy Class Planner
#enable_homotopy_class_planning: True
enable_homotopy_class_planning: False
#enable_multithreading: True
enable_multithreading: False
max_number_classes: 1

selection_cost_hysteresis: 1.0
selection_prefer_initial_plan: 0.95
selection_obst_cost_scale: 1.0
selection_alternative_time_cost: False
roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
roadmap_graph_area_length_scale: 1.0
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_heading_threshold: 0.45
switching_blocking_period: 0.0
viapoints_all_candidates: True
delete_detours_backwards: True
max_ratio_detours_duration_best_duration: 3.0
visualize_hc_graph: False
visualize_with_time_as_z_axis_scale: False
# Recovery

shrink_horizon_backup: True
shrink_horizon_min_duration: 10
oscillation_recovery: False
oscillation_v_eps: 0.1
oscillation_omega_eps: 0.1
oscillation_recovery_min_duration: 10
oscillation_filter_duration: 10

```

- 更改 launch 文件中 move_base 部分，需要注意文件路径，具体如下：

```

<node pkg="move_base" type="move_base" respawn="false"
name="move_base">

    <param name="base_global_planner"
value="navfn/NavfnROS"/>
    <param name="base_local_planner"
value="teb_local_planner/TebLocalPlannerROS"/>
    <rosparam file="$(find
racecar)/param/teb/teb/teb_local_planner_params.yaml"
command="load"/>

    <rosparam file="$(find
racecar)/param/teb/local_costmap_params.yaml" command="load"/>
    <rosparam file="$(find
racecar)/param/teb/global_costmap_params.yaml" command="load"/>

    <rosparam file="$(find
racecar)/param/teb/base_global_planner_params.yaml"
command="load"/>
    <rosparam file="$(find
racecar)/param/teb/move_base_params.yaml" command="load"/>
</node>

```

再次运行 launch 文件，此时便可使用 TEB 算法进行局部路径规划。

实验八 停车点识别

实验目的:

1、理解图像处理和计算机视觉的基本概念，例如颜色空间、边缘检测、轮廓检测等。

2、熟悉 OpenCV 库，包括其功能和模块。了解如何使用 OpenCV 进行图像处理、对象检测和特征提取。

3、掌握图像处理技术，例如直方图均衡化、滤波、形态学操作等，以提高图像质量和检测效果。

实验内容:

- 1、读取智能车摄像头
- 2、识别停车点标志

实验器材:

ROS 智能车、电脑

实验原理:

读取智能车摄像头并转化为 HSV 色彩空间、二值化、腐蚀、膨胀等操作，根据颜色和停车点标志大小及长宽比筛选，进行停车点标志的识别。

实验步骤:

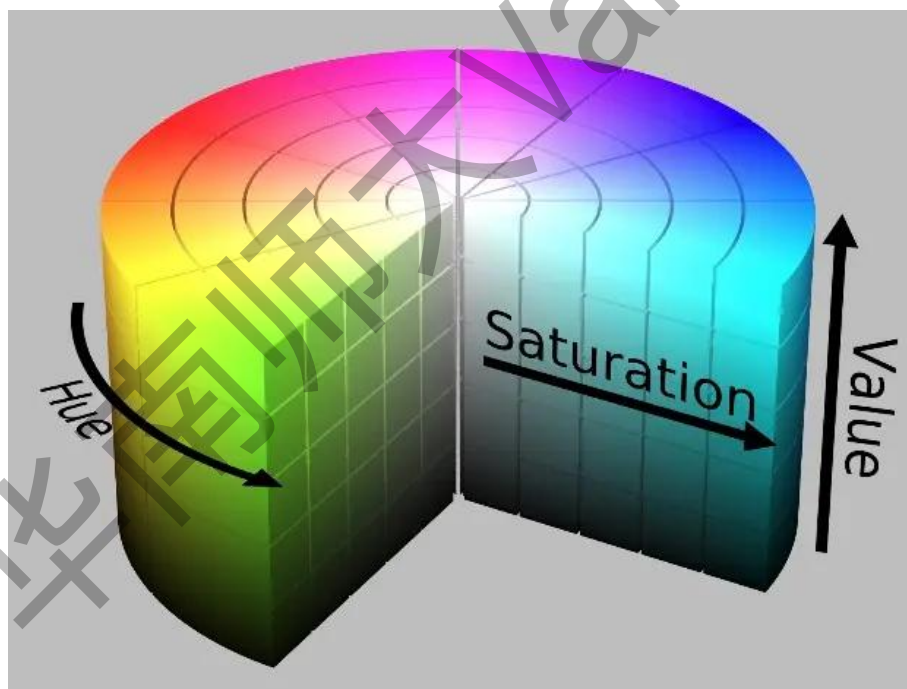
<1> 转化 HSV 颜色空间

摄像头获取到的图像是 RGB 图像。由图 8-1 所示，停车标志最明显的特征就是其大片的蓝色，显然它的蓝色分量要比绿色分量和红色分量大。如果用拆分通道后的蓝色通道去其他颜色通道还有比较大的余量，说明该物体在图像中足够“蓝”。白色的物体三个通道分量都很大，显然相减之后会得到接近零的值。但是 RGB 空间分离效果不佳，受光线影响大。更好的方法是将图片转换到 HSV 颜色空间。



(图 8-1)

如图 8-2 所示，其中 H 代表色调、色相。取值范围 $0-360^\circ$ Hue = 0 表示红色 H = 120 表示绿色 H= 240 表示蓝色，连续性很好。S 表示饱和度、色彩纯净度饱和度越高，颜色越深 0 表示纯白色值越大，越饱和。V 表示明度。取值范围 0-100% 明度越高，颜色越亮，0 表示纯黑色。



(图 8-2)

其代码如下：

```
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
imshow("hsv", hsv)
```

其识别结果如下：



(图 8-3)

〈2〉 提取蓝色区域

inRange()函数

函数作用：根据设定的阈值去除阈值之外的背景部分具体参数： mask = cv2.inRange (hsv, lower_red, upper_red)

代码如下：

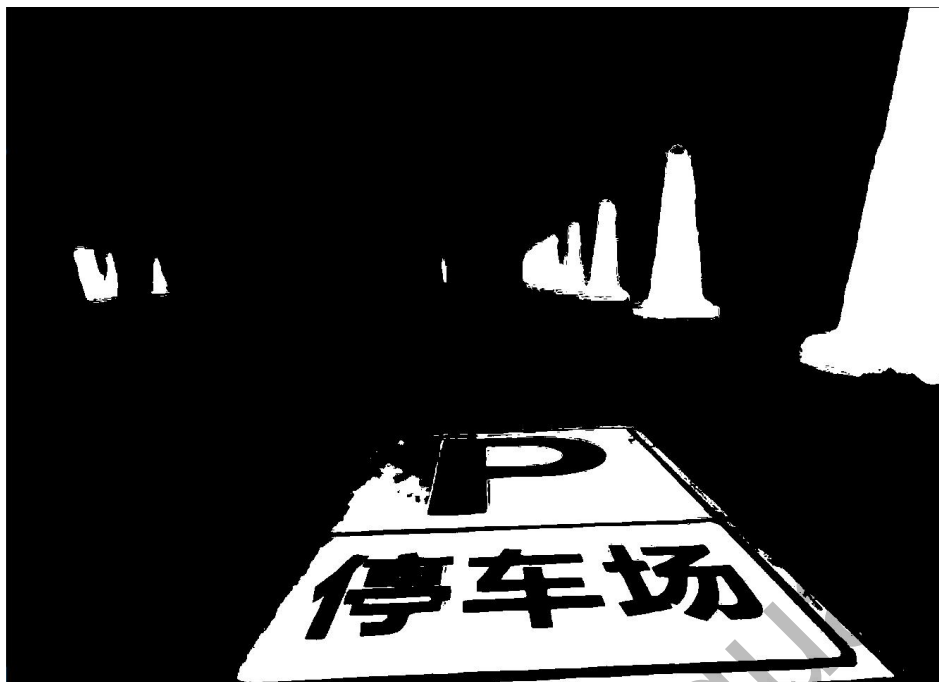
```
blue_lower = np.array([100, 100, 100], dtype=np.uint8)
blue_upper = np.array([124, 255, 255], dtype=np.uint8)
mask_blue = cv2.inRange(hsv, blue_lower, blue_upper)
cv2.imshow("mask_blue", mask_blue)
```

第一个参数：hsv 指的是 hsv 色彩空间的原图

第二个参数：blue_lower 指的是图像中低于这个 blue_lower 的值，图像值变为 0

第三个参数：blue_upper 指的是图像中高于这个 blue_upper 的值，图像值变为 0，图像值变为 0，即变为黑色

如下图是将蓝色以外的颜色变为黑色：

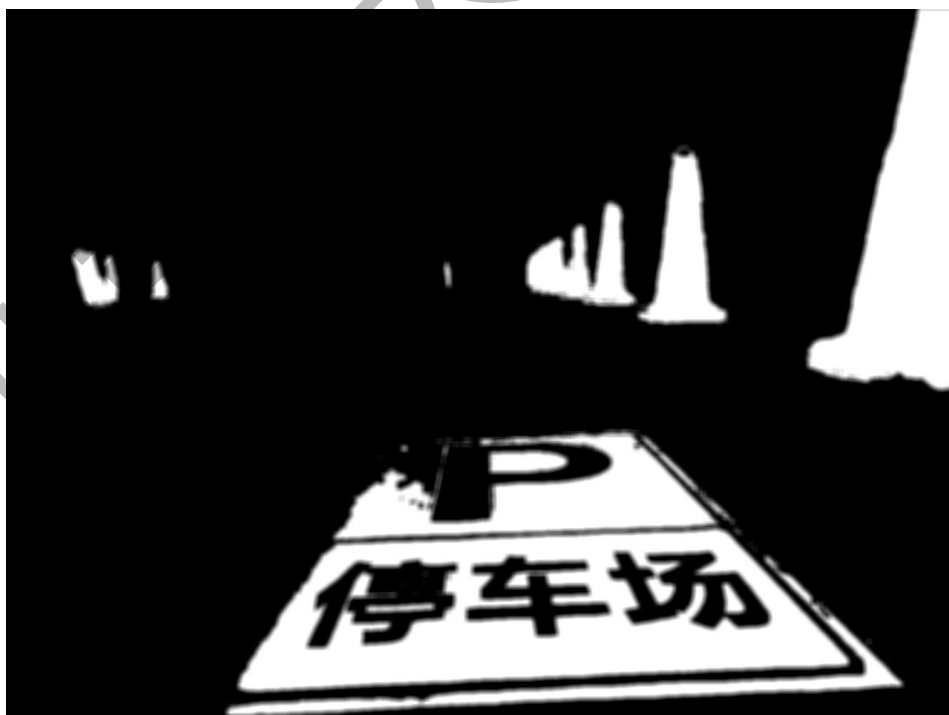


(图 8-4)

<3> 图像处理

`cv2.blur(src, ksize)`:采用均值滤波，它只取内核区域下所有像素的平均值并替换中心元素。我们使用 9x9 的盒式过滤器

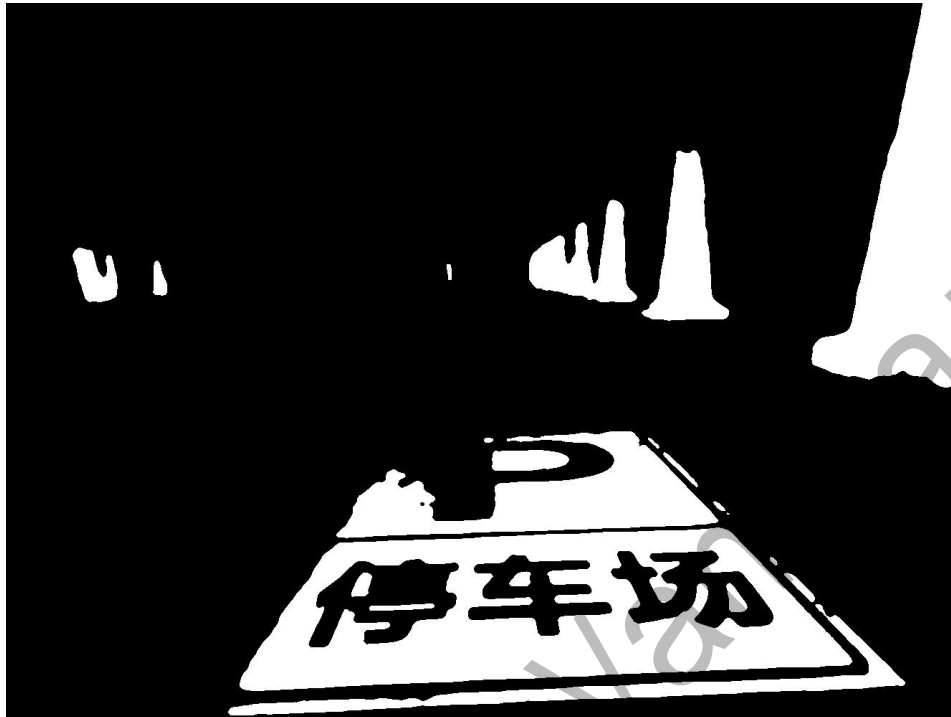
```
# 模糊处理
blurred = cv2.blur(mask_blue, (9, 9))
cv2.imshow("blurred", blurred)
```



(图 8-5)

`cv2.threshold(src, thresh, maxval, type)`: 对图像进行阈值化处理。在这里用于将模糊后的图像转换为二值图像。

```
# 二值化
_, binary = cv2.threshold(blurred, 127, 255, cv2.THRESH_BINARY)
cv2.imshow("binary", binary)
```

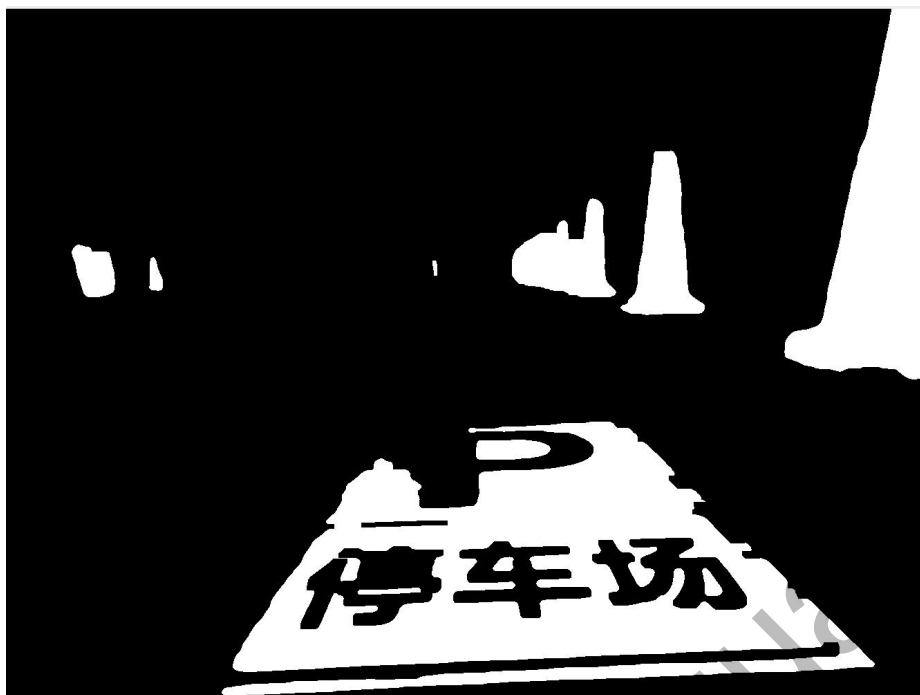


(图 8-6)

`cv2.getStructuringElement(shape, ksize)`: 创建一个形状为 `shape`, 大小为 `ksize` 的结构元素。在这里用于生成闭合图像的结构元素。

`cv2.morphologyEx(src, op, kernel)`: 对图像进行形态学操作, 如腐蚀和膨胀。在这里用于对二值图像进行闭合操作。

```
# 使区域闭合无空隙
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))
closed = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)
cv2.imshow("closed", closed)
```



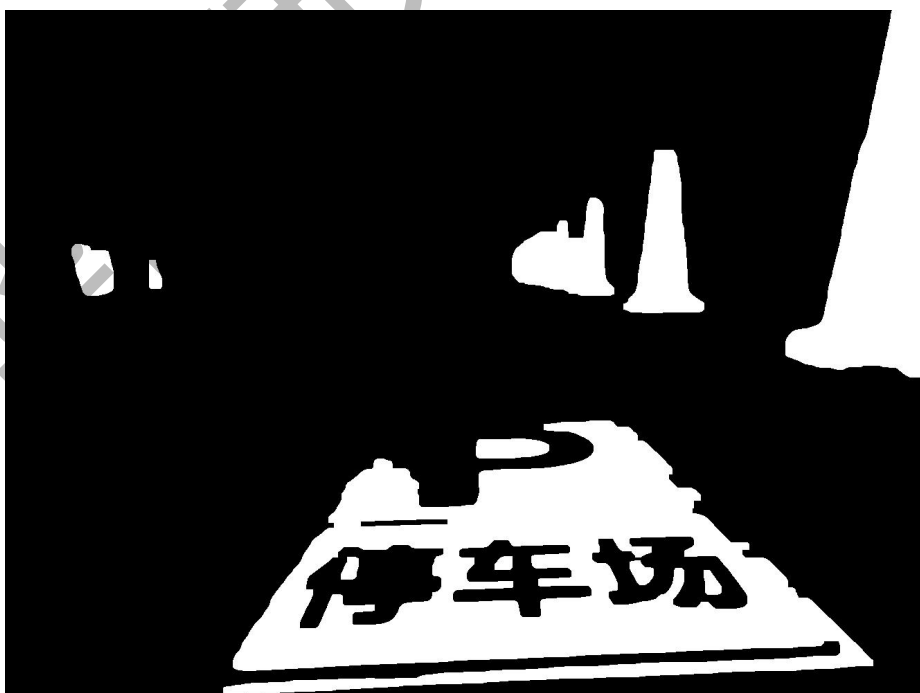
(图 8-7)

`cv2.erode(src, kernel, iterations)`: 对图像进行腐蚀操作。在这里用于对闭合后的图像进行腐蚀。

`cv2.dilate(src, kernel, iterations)`: 对图像进行膨胀操作。在这里用于对腐蚀后的图像进行膨胀。

腐蚀和膨胀

```
erode = cv2.erode(closed, None, iterations=4)
dilate = cv2.dilate(erode, None, iterations=4)
cv2.imshow("dilate", dilate)
```



(图 8-8)

<4>查找轮廓

`cv2.findContours(image, mode, method)`: 查找图像中的轮廓。在这里用于找到二值图像中的外部轮廓。

查找轮廓

```
contours, _ = cv2.findContours(dilate, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

<5>遍历并筛选轮廓

由于右边的锥桶也是蓝色,所以我们需要根据停车点标志的尺寸大小,筛选出蓝色部分中真正的标志,并在原图中绘制出来。

`cv2.minAreaRect(points)`: 根据一组点拟合最小外接矩形。在这里用于找到外接矩形。

`cv2.boxPoints(rect)`: 获取最小外接矩形的四个顶点坐标。

`cv2.drawContours(image, contours, contourIdx, color, thickness)`: 在图像上绘制轮廓。在这里用于在原图像上绘制找到的外接矩形。

遍历轮廓

```
for con in contours:
    rect = cv2.minAreaRect(con)
    box = cv2.boxPoints(rect).astype(int)

    h1 = max(box[0, 1], box[1, 1], box[2, 1], box[3, 1])
    h2 = min(box[0, 1], box[1, 1], box[2, 1], box[3, 1])
    l1 = max(box[0, 0], box[1, 0], box[2, 0], box[3, 0])
    l2 = min(box[0, 0], box[1, 0], box[2, 0], box[3, 0])

    h = h1 - h2
    l = l1 - l2

    # 筛选出高大于等于150像素点,宽大于等于300像素点
    if h >= 150 and l >= 300:
        for i in range(4):
            cv2.line(res, tuple(box[i]), tuple(box[(i + 1) % 4]), (0, 0, 255), 2)
```



(图 8-9)

实验九 红绿灯识别

实验目的：

1、理解图像处理和计算机视觉的基本概念，例如颜色空间、边缘检测、轮廓检测等。

2、熟悉 OpenCV 库，包括其功能和模块。了解如何使用 OpenCV 进行图像处理、对象检测和特征提取。

3、掌握图像处理技术，例如直方图均衡化、滤波、形态学操作等，以提高图像质量和检测效果。

实验内容：

- 1、读取智能车摄像头
- 2、识别红绿灯

实验器材：

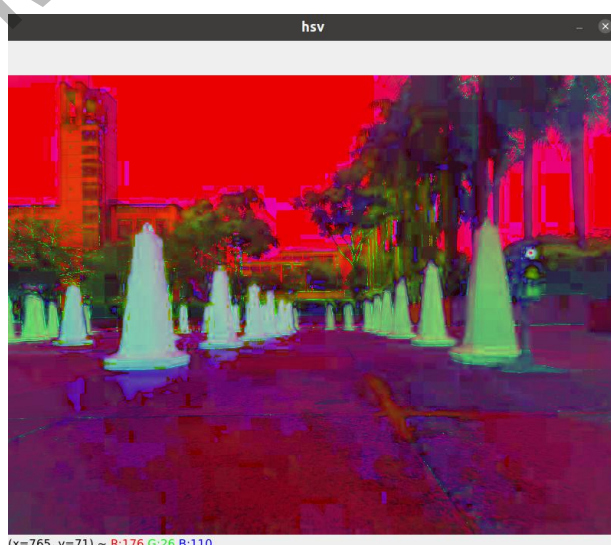
ROS 智能车、电脑

实验原理：

读取智能车摄像头并转化为 HSV 色彩空间、二值化、腐蚀、膨胀等操作，根据颜色和红绿灯的大小及长宽比筛选，进行红绿灯的识别。由于实验原理与上一个实验类似，故以下只展示每一步提取和识别的效果。

实验步骤：

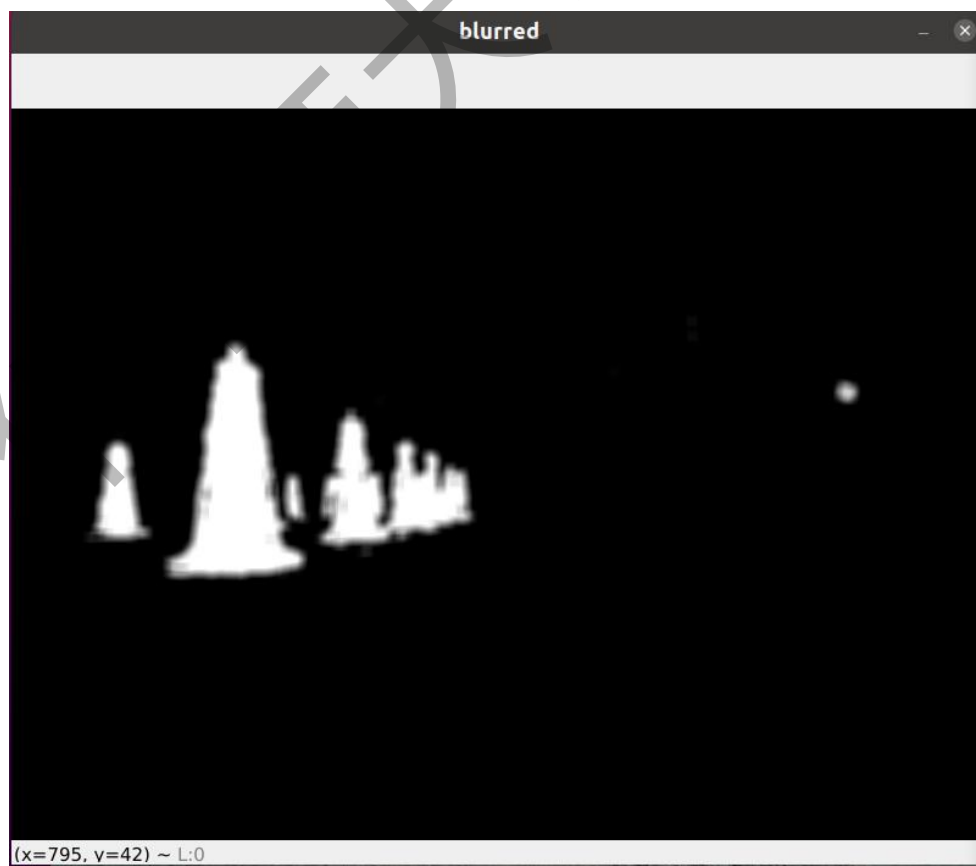
<1> HSV 空间转化



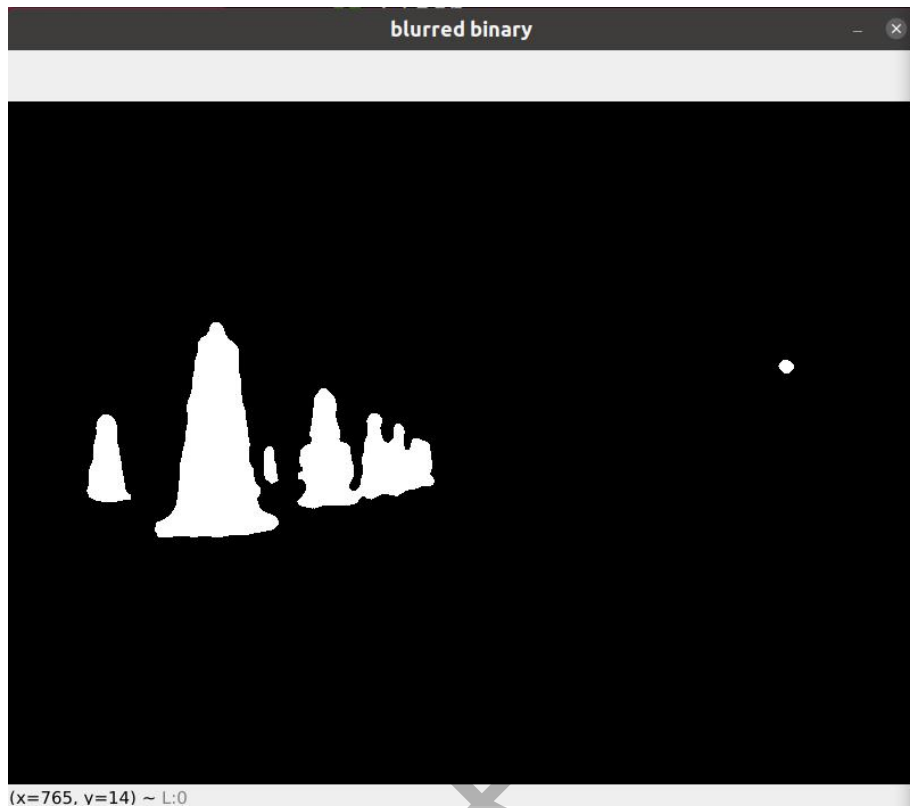
〈2〉 提取红色区域



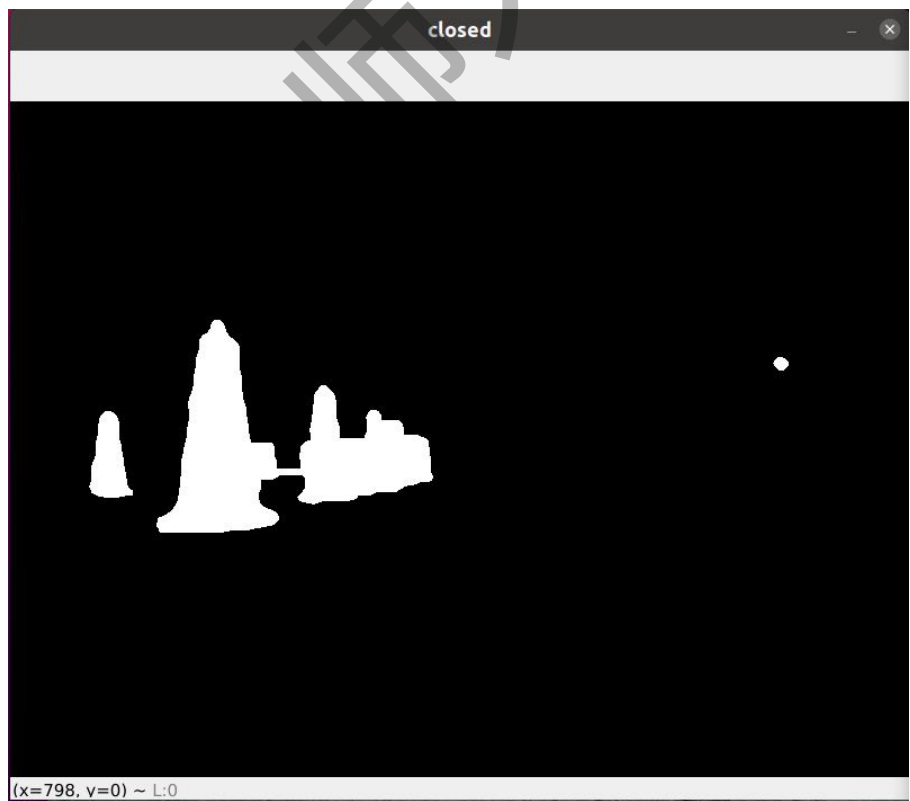
〈3〉 高斯模糊



<4>二值化



<5>闭运算



<6>膨胀



<7>筛选红绿灯



实验十 地图滤波和锥桶连线

实验目的：

- 1、掌握简单的图像处理技能
- 2、通过图像处理算法增强地图可用性
- 3、锥桶连线以获得更好的导航效果

实验内容：

- 1、地图滤波实验
- 2、锥桶连线实验

实验器材：

ROS 智能车、电脑

实验原理：

在建图时，由于各种原因，可能出现以下问题：

- 地图中存在噪点
- 锥桶信息不明显

我们可以通过滤波的方式解决这两个问题。

我们将 gmapping 建完图后的地图数据进行保存，得到以 pgm 格式保存的图像数据。因此我们可以使用图像处理的方法对地图进行滤波操作，去除噪点。滤波完毕后我们可以得到清晰的锥桶数据，使用凸包连线等方法可将锥桶进行连线操作。

实验步骤：

<1> 滤波方案论证



原始地图数据（图 10-1）

滤波方法 1：中值滤波

中值滤波是一种常用的数字图像处理方法，用于去除图像中的噪声。其优点是它能够有效地去除图像中的椒盐噪声等离群值，而不会引入太多的模糊效果。原理很简单：

- 选择滤波器大小： 中值滤波器通常是一个正方形的窗口，其大小是一个奇数，例如 3x3、5x5 等。选择窗口大小的目的是为了能够找到中间值。
- 窗口移动： 将窗口按照一定的步长在图像上滑动。对于图像中的每个像素，将窗口中包含的像素值进行排序。
- 取中值： 选取排序后的像素值中间的值作为当前像素的新值。因为中值是有序集合中的中间值，它对异常值更具有鲁棒性，能够有效地去除噪声。

滤波方法 2：面积过滤法

观察图像我们可以发现，锥桶是由一堆黑点组成的，且面积比噪点的大，所以根据这一特征我们可以先将所有的轮廓找出来，然后算面积，只要面积小于一定阈值的就过滤掉。这个方法还有一个好处就是它不仅可以过滤，还可以加强锥桶的轮廓特征，得到更清晰的锥桶图像。

我们选第二种滤波方法。

<2> 地图滤波的实现

安装库

在小车的终端上依次输入下列命令

```
pip3 install opencv-python  
pip3 install numpy
```

代码实现：

```

import cv2
import numpy as np

class mapBlur:
    def __init__(self) -> None:
        pass

    def mapBlur(self):
        # 读取图像并转换为灰度图像
        img = cv2.imread('originMap.pgm')

        # img_cut = img[325:390 + 1, 435:720 + 1] #outer
        # img_cut = img[280:430 + 1, 390:760 + 1] #inner

        # 获取灰度图像
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # 获取二值化图像
        _, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)
        # 查找轮廓
        contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        blk_contour = [] #保留的黑色区域, 即障碍物
        filtered = [] #需要被滤除的区域
        # 迭代所有的轮廓, 计算其面积
        for contour in contours:
            area = cv2.contourArea(contour) #计算黑点区域面积
            if area >= 0.30: #面积>=0.3 的保留
                blk_contour.append(contour)
            else:
                filtered.append(contour)
            # print("黑点面积:", area)
        # 绘制黑点区域的轮廓
        blurred_img = np.copy(img)
        # print(blk_contour)
        #用白色填充
        cv2.drawContours(blurred_img, filtered, -1, (255, 255, 255),
cv2.FILLED)

        #标志被保留的部分, 即增强
        cv2.drawContours(blurred_img, blk_contour, -1, (0, 0, 0), 3)

```

```

#保存图片，在实车上跑时使用
# saved_img = cv2.cvtColor(blured_img, cv2.COLOR_BGR2GRAY)
# cv2.imwrite('map/mymap.pgm', saved_img)
# img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# cv2.imwrite('map/originMap.pgm', img)
# print('Blured Map Finshed!')
# 显示图像
result = cv2.hconcat([img, blured_img])
#result = cv2.resize(result, (result.shape[1]//2,
result.shape[0]//2))
# cv2.imshow('originMap', img)
# cv2.imshow('bluredMap', blured_img)
cv2.imshow('result', result)
cv2.waitKey(0)
cv2.destroyAllWindows()
if __name__ == '__main__':
    mapBlur().mapBlur()

```

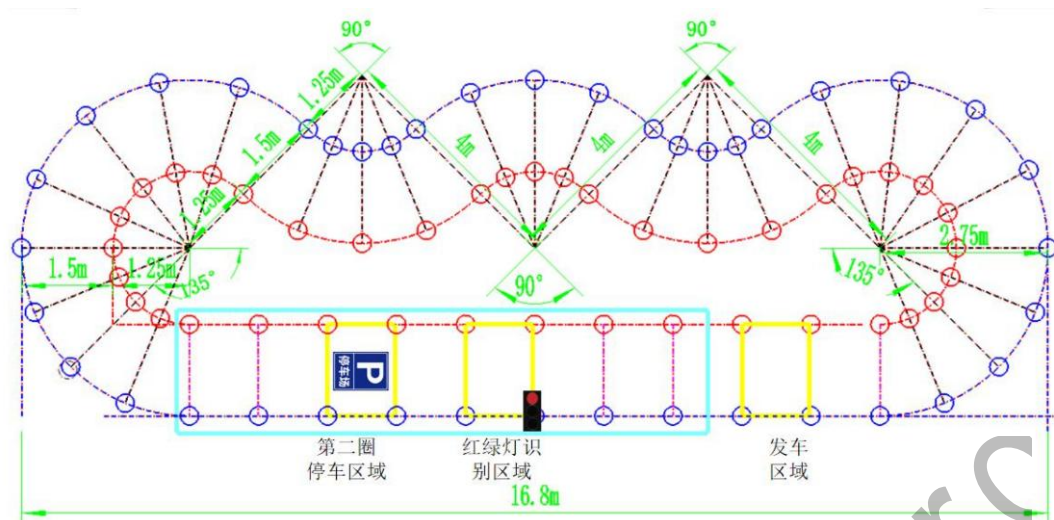
滤波效果：

对比原图可以看到，图中的噪点基本去除了，并且锥桶经过增强后清晰可见。



滤波后地图数据（图 10-2）

〈3〉 地图连线思路及代码实现



思路：观察地图，不难发现其特征，锥桶都是分布在多个同心圆上，同时相邻锥桶与内外圈的锥桶距离不一样。根据这两个特征我们可以写出相应的代码。

首先，我们读取一个名为'originMap.pgm'的图像，并将其转换为灰度图像。然后，我们对图像进行裁剪，只保留图像的一部分。接着，我们将灰度图像二值化，使得图像只包含黑色和白色两种颜色。

然后，我们使用 `cv2.findContours` 函数找出二值化图像中的所有轮廓。这些轮廓代表了图像中的黑色区域，即障碍物。我们将面积大于等于 0.3 的轮廓添加到 `blk contour` 列表中，将其他轮廓添加到 `filtered` 列表中。作用是滤波去噪。

接着，我们计算每个黑色区域的中心点，并将这些点添加到 `points` 列表中。

然后，我们定义了三个圆心，每个圆心都有一个大圆和一个小圆。我们检查每个点是否在大圆和小圆之间，并且是否在特定的角度范围内。如果满足条件，我们就将这个点添加到 `cir points big` 列表中。

然后，我们检查 `cir_points_big` 列表中的每对点之间的距离。如果距离小于 34 的平方，我们就将这对点添加到 `line_points` 列表中。

接下来，我们检查每个点是否在小圆内。如果满足条件，我们就将这个点添加到 `cir_points_small` 列表中。然后，我们检查 `cir_points_small` 列表中的每对点之间的距离。如果距离小于 25 的平方，我们就将这对点添加到 `line_points` 列表中。

最后，我们在图像上画出所有的线段，并用白色填充所有需要被滤除的区域，用黑色标记所有被保留的区域。

```
import cv2
import numpy as np
import time

# 不开方，减少运算量，计算两点之间的距离的平方。
def distance_2(p1, p2):
    return (p1[0] - p2[0])**2 + (p1[1] - p2[1])**2

# 计算斜率
def tan_2(p1, p2):
    if p1[0] - p2[0] != 0:
        return (p1[1] - p2[1]) / (p1[0] - p2[0])
    else:
        return 0

class mapBlur:
    def __init__(self) -> None:
        pass

    def mapBlur(self):
        # 计算处理耗时
        start_time = time.perf_counter()
        # 读取图像并转换为灰度图像
        img = cv2.imread('originMap.pgm')
        # 外圈轮廓
        img_cut = img[280:430 + 1, 390:760 + 1]

        # 灰度图
        gray = cv2.cvtColor(img_cut, cv2.COLOR_BGR2GRAY)
        # 进行二值化处理
```



```

_, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)
# 查找轮廓

contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
#坐标转换
for i in range(len(contours)):
    for j in range(len(contours[i])):
        contours[i][j][0][0] += 390
        contours[i][j][0][1] += 280
blk_contour = [] #保留的黑色区域, 即障碍物
filtered = []    #需要被滤除的区域
for contour in contours:
    area = cv2.contourArea(contour) #计算黑点区域面积
    if area >= 0.3:                 #面积>=0.3 的保留
        blk_contour.append(contour)
    else:
        filtered.append(contour)
    # print("黑点面积:", area)
# 绘制黑点区域的轮廓
blured_img = np.copy(img)
# print(blk_contour)
#计算黑点区域的中心点
points = []
for i in blk_contour:
    points.append(i[int(i.shape[0]/2)][0])
# print(points)
line_points=[]

cir_center = [
    (467,357), #第一个圆心
    (582,357), #第二个圆心
    (692,357) #第三个圆心
]

# cv2.circle(blured_img, (467,357), 3, (0, 0, 255), -1) #第一个
圆心
# cv2.circle(blured_img, (582,357), 3, (0, 0, 255), -1) #第二个
圆心

```

```

# cv2.circle(blured_img, (692,357), 3, (0, 0, 255), -1) #第三个
圆心

# cv2.circle(blured_img, (415,357), 3, (0, 0, 255), -1)
# cv2.circle(blured_img, (442,357), 3, (0, 0, 255), -1)
#大同心圆半径 = 476-415=61
#小同心圆半径 = 476-442=34
cir_points_big = [] #大同心圆上的点
for i in range(len(points)):
    #第1个圆的筛选条件:
    if distance_2(points[i], cir_center[0]) < 62**2 and
distance_2(points[i], cir_center[0]) > 40**2:
        if tan_2(points[i], cir_center[0]) > 0.8 or
tan_2(points[i], cir_center[0]) < -0.8 or cir_center[0][0] -
points[i][0] > 0:
            # print(tan_2(points[i], cir_center[0]))
            # cv2.circle(blured_img, points[i], 3, (0, 0, 255),
-1)

            cir_points_big.append(points[i])

    #第2个圆的筛选条件:
    if distance_2(points[i], cir_center[1]) < 62**2 and
distance_2(points[i], cir_center[1]) > 40**2:
        if tan_2(points[i], cir_center[1]) > 1.1 or
tan_2(points[i], cir_center[1]) < -1.1:
            # print(tan_2(points[i], cir_center[1]))
            # cv2.circle(blured_img, points[i], 3, (255, 255, 0),
-1)

            cir_points_big.append(points[i])

    #第3个圆的筛选条件:
    if distance_2(points[i], cir_center[2]) < 62**2 and
distance_2(points[i], cir_center[2]) > 40**2:
        if tan_2(points[i], cir_center[2]) > 0.8 or
tan_2(points[i], cir_center[2]) < -0.8 or cir_center[2][0] - points[i][0]
< 0:
            # print(tan_2(points[i], cir_center[2]))
            # cv2.circle(blured_img, points[i], 3, (0, 255, 0),
-1)

            cir_points_big.append(points[i])

```

```

# 大圆匹配
for i in range(len(cir_points_big)):
    for j in range(i+1, len(cir_points_big)): #减少重复计算
        if i != j:
            dis = distance_2(cir_points_big[i],
cir_points_big[j])
            if dis < 34**2:
                line_points.append([cir_points_big[i],
cir_points_big[j]])
# # #大圆连线
# for n in range(len(line_points)):
#     cv2.line(blured_img, line_points[n][0], line_points[n][1],
(0, 0, 255), 2)
cir_points_small = [] #小同心圆上的点
for i in range(len(points)):
    #第1个圆的筛选条件:
    if distance_2(points[i], cir_center[0]) < 37**2:
        # cv2.circle(blured_img, points[i], 3, (0, 0, 255), -1)
        cir_points_small.append(points[i])

    #第2个圆的筛选条件:
    if distance_2(points[i], cir_center[1]) < 37**2:
        # if tan_2(points[i], cir_center[1]) > 1.1 or
tan_2(points[i], cir_center[1]) < -1.1:
            # # print(tan_2(points[i], cir_center[1]))
            # cv2.circle(blured_img, points[i], 3, (255, 255, 0), -1)
            cir_points_small.append(points[i])
    #第3个圆的筛选条件:
    if distance_2(points[i], cir_center[2]) < 37**2:
        # cv2.circle(blured_img, points[i], 3, (0, 255, 0), -1)
        cir_points_small.append(points[i])
# 小圆匹配
for i in range(len(cir_points_small)):
    for j in range(i+1, len(cir_points_small)): #减少重复计算
        if i != j:

```

```

        dis = distance_2(cir_points_small[i],
cir_points_small[j])
        if dis < 25**2:
            line_points.append([cir_points_small[i],
cir_points_small[j]])
    # # #小圆连线
    # for n in range(len(line_points)):
    #     cv2.line(blured_img, line_points[n][0], line_points[n][1],
(0, 0, 255), 2)
    # time = [] * len(points)
    # print(time)
    # #计算同一个点是否被使用了两次
    # for i in range(len(points)):
    #     if line_points[i][0] == points[i]:
    #         time[i] += 1
    #     time[line_points[i][1]] += 1
    # line_points.clear()
    # 按距离筛选
    for i in range(len(points)):
        for j in range(i+1, len(points)): #减少重复计算
            if i != j:
                dis = distance_2(points[i], points[j])
                if dis < 24.5**2:
                    line_points.append([points[i], points[j]])
    #连线
    for n in range(len(line_points)):
        cv2.line(blured_img, line_points[n][0], line_points[n][1],
(0, 0, 0), 1)
    #用白色填充, 即滤除
    cv2.drawContours(blured_img, filtered, -1, (255, 255, 255),
cv2.FILLED)
    #标志被保留的部分, 即增强
    cv2.drawContours(blured_img, blk_contour, -1, (0, 0, 0), 3)
    # cv2.drawContours(blured_img, points, -1, (0, 0, 255), 2)
    #保存图片
    # saved_img = cv2.cvtColor(blured_img, cv2.COLOR_BGR2GRAY)
    # # cv2.imwrite('bluredMap.pgm', saved_img)
    # saved_img = cv2.cvtColor(blured_img, cv2.COLOR_BGR2GRAY)
    # cv2.imwrite('map/mymap.pgm', saved_img)

```

```

# img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# cv2.imwrite('map/originMap.pgm', img)
# print('Blured Map Finshed!')
end_time = time.perf_counter()
execution_time = (end_time - start_time) * 1000
print("程序运行时间: ", execution_time, "毫秒")
# 显示图像
cv2.imshow('originMap', img)
cv2.imshow('bluredMap', blured_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
if __name__ == '__main__':
    map = mapBlur()
    map.mapBlur()

```

滤波及连线效果:

可以看到, 锥桶基本被连接起来, 足够用于导航。



滤波及连线效果 (图 10-4)

<4> 上机复现

终端进入本实验代码目录

复现滤波:

python3 mapBlur.py

复现连线:

python3 bucketConnection.py