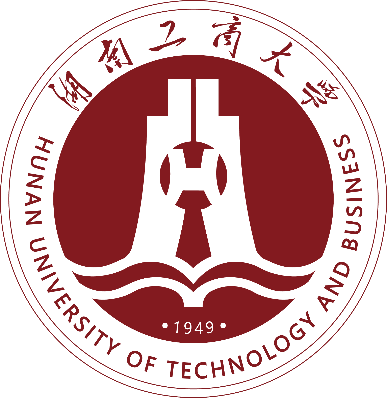
****

**湖南工商大学**

**《机器人操作系统及应用》课程报告**

**题　目： 机器人操作系统及应用课程设计报告**

|  |  |
| --- | --- |
| **姓 名:** | **邓嘉豪** |
| **学 号:** | **21023020011** |
| **专 业­­­­­­­­­­­­­­­­­­­:** | **人工智能** |
| **班 级:** | **2101** |
| **指导教师:** | **姜林** |
| **职 称­­­­­­­­­­­­­­­­­­­:** | **教授** |

**前沿交叉学院**

**2023年11月**

**课程报告评审表**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 等级  成绩  组成 | 优秀 | 良好 | 中等 | 及格 | 不及格 |
| 小车程序演示（50分） | 运动轨迹包含的基本功能为：前进、后退、左转向、右转向、速度设置、里程设置，每项5分。共30分  运动轨迹复杂性及技术难度。30分 | | | | |
|  |  |  |  |  |
| 报  告  撰  写  （50分） | 内容详实（10分）；  运动轨迹算法原理描述清晰，有原理图，有算法流程图，小车运动的演示图片  （20分）；  代码规范，有注释（10分）；  报告格式排版必须规范（10分）； | | | | |
|  |  |  |  |  |
| 综合成绩评定： 评阅老师（签章）：    年 月 日 | | | | | |

**目录**

[1.小车模型构建 1](#_Toc10949)

[1.1 模型整体介绍 1](#_Toc26291)

[1.2 创建Solidworks模型 1](#_Toc32484)

[1.3 构建传感器 1](#_Toc6286)

[1.3.1 激光雷达 1](#_Toc4682)

[1.3.2 摄像头 2](#_Toc18585)

[1.3.3 kinect摄像头 3](#_Toc5144)

[1.4 小车控制器 4](#_Toc7047)

[1.5 最终模型 5](#_Toc26595)

[1.6 通过键盘控制小车运动 7](#_Toc14728)

[2.图像控制 7](#_Toc17502)

[2.1 整体算法介绍 7](#_Toc30903)

[2.1.1 导入ROS消息类型和库 7](#_Toc8335)

[2.1.2 初始化 8](#_Toc20498)

[2.1.3 回调函数 9](#_Toc11095)

[2.1.4 PID控制器 11](#_Toc32586)

[2.1.5 OpenCV图像处理 10](#_Toc10394)

[2.1.6 障碍物避障 11](#_Toc31203)

[2.1.7 欧拉角处理 13](#_Toc10314)

[2.1.8 辅助函数 13](#_Toc22653)

[2.2 重要算法原理 14](#_Toc28447)

[2.2.1 PID算法 14](#_Toc26940)

[2.2.2 里程计获取位置信息 15](#_Toc26055)

[2.2.3 识别障碍物 17](#_Toc176)

[2.2.4 低通滤波器 18](#_Toc19148)

[2.3 算法流程图 19](#_Toc30372)

[2.4 小车运动轨迹图 20](#_Toc1162)

[3.语音控制 21](#_Toc22204)

[3.1 语音控制流程图 21](#_Toc7141)

[3.2 语音控制实例步骤 21](#_Toc19162)

[4.视频控制 23](#_Toc28065)

[4.1 自主巡线 23](#_Toc13704)

[5.建图与导航 28](#_Toc20363)

[5.1 建图 28](#_Toc25869)

[5.1.1 world环境建立 28](#_Toc4050)

[5.1.2 launch文件启动world环境并测试 29](#_Toc19415)

[5.1.3 键盘控制小车 31](#_Toc10303)

[5.2 导航 32](#_Toc20815)

[5.2.1 launch一键启动文件 32](#_Toc3127)

[5.2.2 代价地图配置 33](#_Toc9828)

[5.2.3 导航启动 34](#_Toc28578)

[6.应用案例开发 35](#_Toc9032)

[6.1 全局概括 35](#_Toc5675)

[6.2 Yolov3目标检测 36](#_Toc8241)

[6.2.1 图像话题发布 36](#_Toc30078)

[6.2.2 运行darknet\_ros 36](#_Toc2343)

[6.3 语音播报 37](#_Toc2666)

[6.3.1 yolo模块与语音模块的通信 37](#_Toc11548)

[总结 39](#_Toc10)

# 1.小车模型构建

## 1.1模型整体介绍

本次实验，我们使用的小车模型是一俩aodi牌长1.28m,宽0.56m,高0.37米的四轮驱动小车。并且搭配了摄像头和激光雷达。

## 1.2创建Solidworks模型

本次实验是通过Solidworks软件，来构建的非规则形状的物体模型。

1. 在网络中寻找合适的汽车装配体进行下载，其零件实体至少需包含四个车轮和一个车架，每个link对应一个实体。

2. 创建每一个link的点和坐标系。创建每一个link的点和坐标系，要与ROS中坐标系方向一致，机器人前进方向为X轴正方向，遵循右手定则。先创建base\_link的点和坐标系（一般以底盘为base\_link），再创建其他link的点和坐标系。

3. .创建continue类型joint的旋转轴

## 1.3构建传感器

### 1.3.1 激光雷达



图1：激光雷达模型代码

在本配置中，激光雷达传感器的更新率为 5.5 Hz。这意味着它每秒可以生成 5.5 次数据扫描。更新率越高，传感器提供的环境图像就越流畅。

扫描模式：

采样点之间的分辨率为 1 度。

扫描范围：-3 度到 3 度（可调整以获得更宽的覆盖范围）。

范围：

最小检测距离：0.1 米。

最大检测距离：30 米。

分辨率：0.01 米（距离测量精度）。

噪声：均值为 0.0，标准差为 0.01 的正态噪声（模拟真实世界传感器噪声）。

激光雷达传感器的数据以 /scan 主题发布到 ROS。这意味着其他 ROS 节点可以订阅此主题以获取传感器数据。

使用 laser 帧作为坐标引用。

### 1.3.2 摄像头



图2：摄像头模型代码

摄像头传感器是一种重要的传感器，可用于各种机器人应用。它可以提供周围环境的视觉图像，这对于计算机视觉、目标检测和导航至关重要。

在本配置中，摄像头传感器的更新率为 30 Hz。这意味着它每秒可以生成 30 帧图像。更新率越高，传感器提供的环境图像就越流畅。

摄像头传感器的视野为 114 度。这意味着它可以看到周围环境的 114 度范围。

摄像头传感器的分辨率为 1280 x 720 像素。这意味着它可以捕捉到高质量的图像。

摄像头传感器将图像数据发布到 ROS 的 /camera 主题。这意味着其他 ROS 节点可以订阅此主题以获取图像数据。

* 更新率：30.0 Hz（每秒 30 帧）
* 摄像头信息：
  + 名称：head
  + 水平视野：2.0 弧度（约为 114 度）
  + 图像分辨率：1280 x 720 像素
  + 图像格式：R8G8B8（红色、绿色、蓝色颜色模型）
  + 近裁剪平面：0.02 米
  + 远裁剪平面：300 米
  + 噪声模型：均值为 0.0，标准差为 0.007 的正态噪声

### 1.3.3kinect摄像头



图3：kinetic摄像头模型代码

深度传感器是一种重要的传感器，可用于各种机器人应用。它可以提供周围环境的三维视图，这对于导航、物体识别和 3D 建图至关重要。

在本配置中，深度传感器的更新率为 20 Hz。这意味着它每秒可以生成 20 帧深度图像。更新率越高，传感器提供的环境图像就越流畅。

深度传感器的视野为 60 度。这意味着它可以看到周围环境的 60 度范围。

深度传感器的分辨率为 640 x 480 像素。这意味着它可以捕捉到高质量的深度图像。

深度传感器将深度信息、RGB 图像和 3D 点云数据发布到 ROS 的以下主题：

* rgb/image\_raw：RGB 图像数据
* depth/image\_raw：深度图像数据
* depth/points：3D 点云数据

## **1.4小车控制器**

由于以前只使用过两驱因此我先对后轮进行了控制，将move.xacro集成进aodi\_union.xacro后进行键盘控制观察。



图4：小车驱动1

在运行过程中发现了xacro文件存在问题，由于碰撞属性部分设置不合理而导致汽车轮子转动但无法运动，比如车架触地而导致轮胎悬空，视觉上几乎无法发现，体现为虽然驱动轮能够按照按键控制转动，小车却无法运动。分别对visual标签的mesh文件和collision标签进行了修改后，才使汽车能够前进和后退。

但后轮驱动时前轮作为固定部分与地面接触，在汽车转向时具有较大的侧向力阻值汽车顺利转向，需修改move.xacro为四驱。



图5：小车驱动2

## **1.5最终模型**

Rviz中的模型：



图6：最终小车模型（rviz）

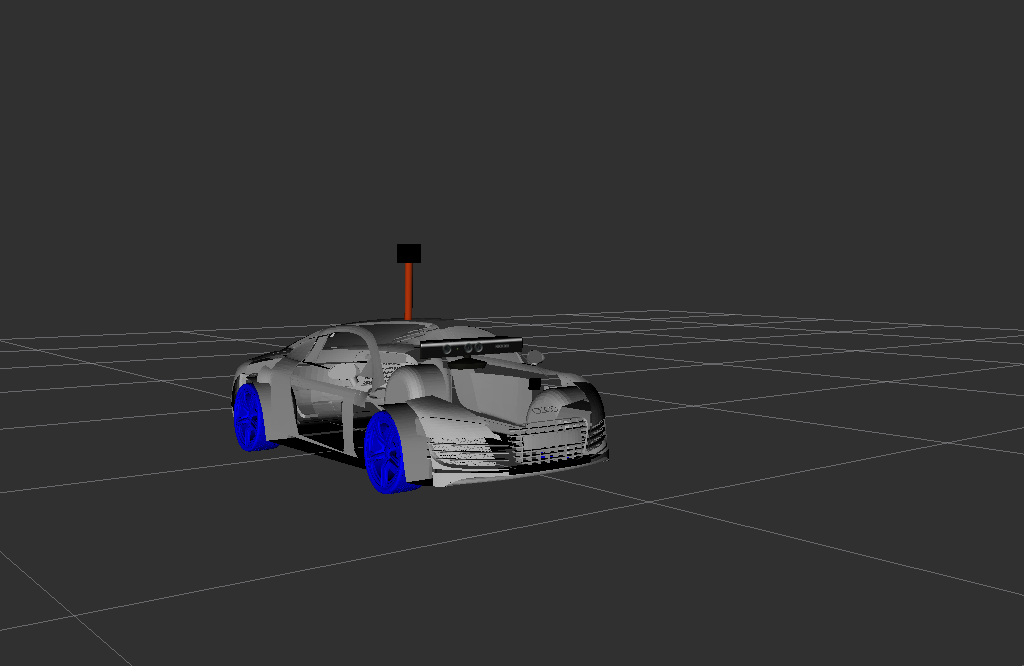


图7：最终小车模型（rviz）

Gazebo中的小车模型

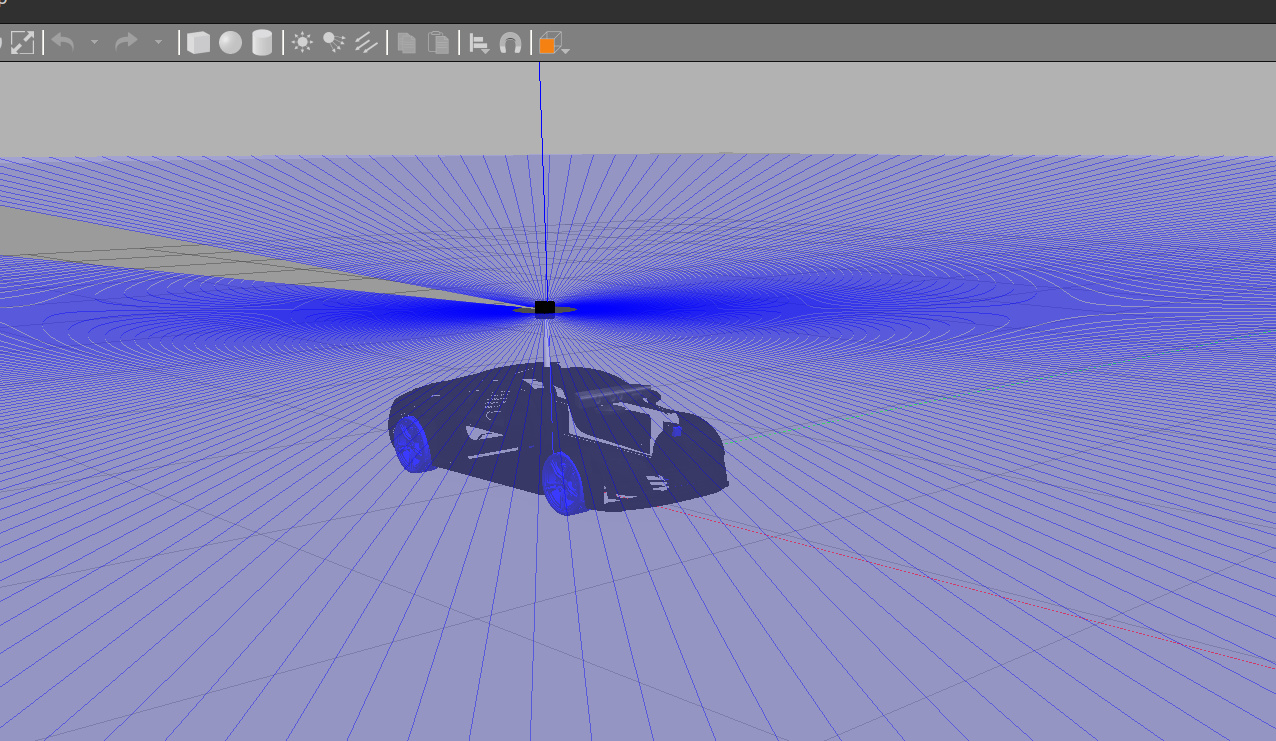


图8：最终小车模型（gazebo）

## **1.6通过键盘控制小车运动**

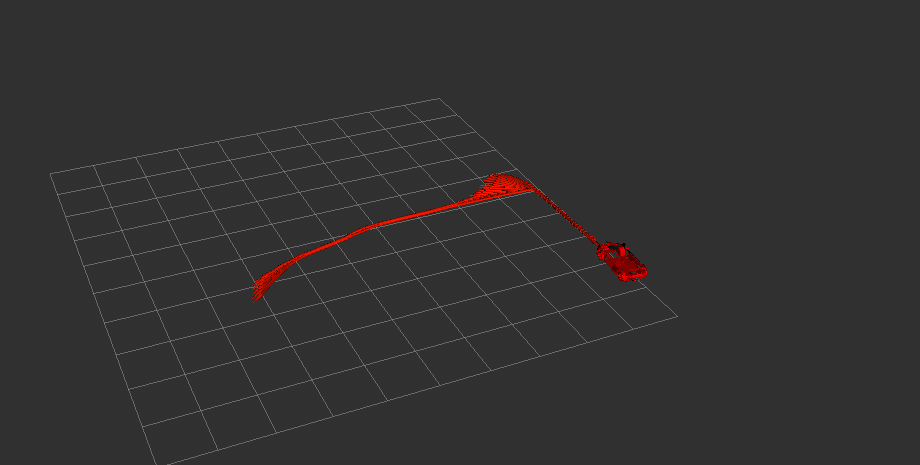


图9：键盘控制小车运动

# 2.图像控制

## **2.1 整体算法介绍**

### 2.1.1 导入ROS消息类型和库

from geometry\_msgs.msg import Twist

导入了ROS中用于表示机器人速度的消息类型Twist。

from nav\_msgs.msg import Odometry

导入ROS中用于表示里程计信息的消息类型Odometry。

from sensor\_msgs.msg import LaserScan

导入ROS中用于表示激光扫描数据的消息类型LaserScan。

from tf.transformations import euler\_from\_quaternion

导入了ROS中的tf库，其中的transformations模块包含了一些用于处理四元数到欧拉角的转换函数，此处使用了euler\_from\_quaternion函数。

from math import atan2, pi

导入Python的math库，其中包含了计算反正切和圆周率π的函数。

### 2.1.2 初始化

class LineFollower:

def \_\_init\_\_(self):

# 初始化ROS节点

rospy.init\_node('line\_follower', anonymous=True)

这段代码定义了一个名为LineFollower的类，代表了巡线机器人的行为。在类的构造函数\_\_init\_\_中，进行了以下初始化操作：

1.ROS节点的初始化：

rospy.init\_node('line\_follower', anonymous=True)：这一行代码初始化了ROS节点。'line\_follower'是节点的名称，anonymous=True表示如果在同一系统中启动多个具有相同名称的节点，它们会被区分开，以避免冲突。

2. 订阅激光扫描和里程计主题：

rospy.Subscriber('/scan', LaserScan, self.scan\_callback)：订阅名为'/scan'的激光扫描主题，并指定了回调函数self.scan\_callback，该函数会在接收到激光扫描数据时被调用。

rospy.Subscriber('/odom', Odometry, self.odom\_callback)：订阅名为'/odom'的里程计主题，并指定了回调函数self.odom\_callback，该函数会在接收到里程计信息时被调用。

3. 发布速度控制命令：

self.cmd\_vel\_pub = rospy.Publisher('/cmd\_vel', Twist, queue\_size=10)：创建了一个发布器，用于向'/cmd\_vel'主题发布机器人速度控制命令。这里使用的消息类型是Twist，它包含了机器人的线速度和角速度信息。queue\_size=10表示在发布速度控制命令时最多存储10个消息，超过这个数量的消息会被丢弃。

4. 控制相关变量的初始化：

self.cmd\_vel = Twist()：创建了一个Twist类型的实例，用于存储机器人的速度控制命令。

self.Kp、self.Ki、self.Kd：设置了PID控制器的比例、积分和微分系数。

self.initial\_orientation = None：用于存储机器人的初始方向，初始值为None。

self.last\_yaw\_error = 0.0：存储上一次的偏航角误差，初始值为0.0。

self.laser\_ranges = []：用于存储激光扫描数据。

self.obstacle\_detected = False：用于标识是否检测到障碍物，初始值为False。

### 2.1.3 回调函数

1. scan\_callback函数：

当激光扫描主题（'/scan'）接收到新的消息时，该回调函数被调用。

self.laser\_ranges = scan\_msg.ranges：将激光扫描数据存储在self.laser\_ranges中，以便后续使用。

if min(scan\_msg.ranges) < 0.5:：检查激光扫描中是否存在距离小于0.5的点，即是否存在潜在的障碍物。

self.obstacle\_detected = True：如果存在障碍物，将self.obstacle\_detected标志设置为True；否则，设置为False。

2. odom\_callback函数：

当里程计主题（'/odom'）接收到新的消息时，该回调函数被调用。

if self.initial\_orientation is None:：检查是否已获取初始方向，如果尚未获取，则将当前里程计消息的方向作为初始方向。

current\_yaw = self.get\_yaw\_from\_quaternion(odom\_msg.pose.pose.orientation)：从当前里程计消息中获取机器人的当前偏航角。

initial\_yaw = self.get\_yaw\_from\_quaternion(self.initial\_orientation)：获取机器人的初始偏航角。

yaw\_error = self.normalize\_angle(initial\_yaw - current\_yaw)：计算偏航角误差，即机器人当前偏航角与初始偏航角之差，并进行角度规范化。

if self.obstacle\_detected:：如果存在障碍物，调用self.stop\_robot()停止机器人。

这两个回调函数主要负责处理激光扫描和里程计信息，计算偏航角误差，并在检测到障碍物时采取相应的措施，例如停止机器人。

### 2.1.5 OpenCV图像处理

通过读取一张图像，对其进行灰度化、平滑处理、边缘检测、轮廓提取等图像处理操作，并提取出其中最大的轮廓，并将该轮廓上的像素点坐标保存，以作为小车移动的参考点。

1. 获取图像文件的绝对路径：通过os模块中的os.path.abspath()函数获取当前脚本文件所在目录的绝对路径，再通过os.path.join()函数将图像文件名拼接到路径后面，得到图像文件的绝对路径。

2. 读取图像：使用cv2.imread()函数读取图像文件，如果读取成功，则将图像数据存储在image变量中；否则输出日志信息并退出程序。

3. 将图像转换为灰度图像：使用cv2.cvtColor()函数将彩色图像转换为灰度图像。

4. 进行图像平滑处理：使用cv2.GaussianBlur()函数对灰度图像进行高斯平滑处理，以减少图像中的噪声。

5. 进行边缘检测：使用cv2.Canny()函数对平滑后的灰度图像进行边缘检测，得到图像中的边缘信息。

6. 进行轮廓提取：使用cv2.findContours()函数对边缘图像进行轮廓提取，得到图像中的所有轮廓。其中cv2.RETR\_EXTERNAL表示只提取最外层的轮廓，cv2.CHAIN\_APPROX\_SIMPLE表示只保留轮廓的端点坐标。

7. 找到最大的轮廓：使用max()函数根据轮廓的面积大小找到最大的轮廓，并将其赋值给max\_contour变量。

8. 获取轮廓上的像素点坐标：使用for循环遍历max\_contour中的每个点，并将其x、y坐标存储到points列表中。

9. 对点坐标进行缩小和平移：将points中的每个点的坐标值都分别除以100，然后将第一个点的坐标值作为基准点，将points中每个点的坐标值都减去基准点的坐标值，得到waypoints列表，即为轮廓上的所有点相对于基准点的坐标值。

### 2.1.4 PID控制器

代码中，PID控制器用于控制机器人的方向，以使其沿着预定的路径移动。PID控制器的参数有三个部分：比例系数（Proportional, P）、积分系数（Integral, I）、微分系数（Derivative, D）。这些参数对控制器的行为有重要影响。

# PID 控制器参数

self.Kp = 3 # 比例系数

self.Ki = 6 # 积分系数

self.Kd = 6 # 微分系数

1．比例系数 self.Kp（Proportional）:

比例控制的作用是根据当前误差的大小调整系统的输出。如果误差较大，比例控制会产生较大的控制输出，以加速系统的响应。较大的比例系数会导致系统更快地接近期望值，但可能引起过冲，即系统超过了目标值。因此，在选择比例系数时需要权衡系统的响应速度和稳定性。

2.积分系数 self.Ki（Integral）:

积分控制的作用是根据误差随时间的积累来进行调整。它主要用于消除系统的稳态误差，即长时间内的误差积累。如果系统存在稳态误差，增大积分系数可以减小这一误差。然而，过大的积分系数可能导致系统超调或产生振荡，因此需要谨慎选择。

3.微分系数 self.Kd（Derivative）:

微分控制的作用是根据误差变化率的大小来进行调整。它通过考虑误差的变化趋势来抑制系统的过冲和振荡。较大的微分系数有助于提高系统的稳定性，但可能引入噪声。因此，在选择微分系数时需要平衡系统的稳定性和抗干扰能力。

在实际应用中，PID参数的选择通常需要通过试验和调试来完成。一种常见的调试方法是从小的数值开始，逐渐增大参数，并观察系统的响应。通过实验和观察，可以找到适合系统的PID参数组合，以满足特定应用场景的性能要求。

总的来说，PID控制器是一种灵活且广泛应用的控制方法，它可以通过调整比例、积分和微分参数来适应不同的系统和应用场景。调试PID参数是一个经验性的过程，需要深入理解系统的特性和动态响应，以达到稳定、快速响应且尽量减小超调的效果。

### 2.1.6 障碍物避障

通过激光扫描数据来检测是否有障碍物存在，如果检测到障碍物，则停止机器人的运动。以下是相关的代码部分：

def scan\_callback(self, scan\_msg):

self.laser\_ranges = scan\_msg.ranges

if min(scan\_msg.ranges) < 0.5:

self.obstacle\_detected = True

else:

self.obstacle\_detected = False

在上述代码中，scan\_callback是激光扫描信息的回调函数。首先，通过scan\_msg.ranges获取到激光扫描数据，这是一系列距离数值，表示机器人周围的障碍物距离。接着，通过判断最小距离是否小于设定的阈值（这里是0.5，可根据实际情况调整），来判断是否检测到障碍物。

def odom\_callback(self, odom\_msg):

# 如果有障碍物，则停止机器人

if self.obstacle\_detected:

self.stop\_robot()

return

在odom\_callback回调函数中，当检测到障碍物时，会调用stop\_robot函数停止机器人的运动。这是通过将机器人的线速度和角速度都设置为零来实现的，从而停止机器人的运动。

### 2.1.7 欧拉角处理

在提供的代码中，有一个函数用于将四元数表示的姿态信息转换为欧拉角。这个函数是 get\_yaw\_from\_quaternion，下面是对这个函数的详细介绍：

def get\_yaw\_from\_quaternion(self, quaternion):

# 将四元数转换为欧拉角表示，返回偏航角

\_, \_, yaw = euler\_from\_quaternion([quaternion.x, quaternion.y, quaternion.z, quaternion.w])

return yaw

这个函数接受一个四元数（Quaternion）作为输入，并返回欧拉角中的偏航角（yaw）。在机器人运动中，姿态通常以四元数的形式表示，但在某些情况下，欧拉角更容易理解和使用。

函数的主要部分是调用 euler\_from\_quaternion 函数，该函数来自于 tf.transformations 模块。它将四元数转换为对应的欧拉角，返回的是一个包含三个角度的元组，分别对应着滚动角（roll）、俯仰角（pitch）和偏航角（yaw）。在这里，由于我们只关注偏航角，因此使用下划线来占位不需要的滚动角和俯仰角。

使用get\_yaw\_from\_quaternion函数将四元数表示的机器人方向转换为欧拉角，以便进行角度的计算。

具体而言，函数的流程如下：首先通过 euler\_from\_quaternion 将四元数转换为欧拉角。其次从返回的欧拉角元组中提取偏航角，并将其返回。

这个函数在机器人运动控制中经常用于获取机器人当前的偏航角，以便进行姿态调整和导航。在代码中，它被用于计算偏航角误差，这是实现巡线小车运动控制的重要一环。

### 2.1.8 辅助函数

normalize\_angle 函数用于将角度规范化到区间 [-π, π] 之间，以确保角度在合理的范围内，防止超出规定的角度范围。以下是对这个函数的详细介绍：

def normalize\_angle(self, angle):

# 将角度规范化到 -π 到 π 之间

while angle > pi:

angle -= 2.0 \* pi

while angle < -pi:

angle += 2.0 \* pi

return angle

这个函数接受一个角度作为输入，并返回规范化后的角度。具体而言，它通过不断减去或加上 2π，将输入的角度调整到 [-π, π] 区间内。

函数的主要流程如下：

如果角度 angle 大于 π，就从中减去 2π，一直减到 angle 小于等于 π。

如果角度 angle 小于 -π，就从中加上 2π，一直加到 angle 大于等于 -π。

返回规范化后的角度。

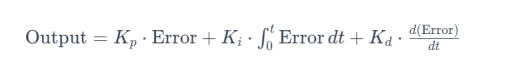
这样，无论输入的角度是多少圈，都会被规范到 [-π, π] 的范围内。这在处理角度差异时是非常有用的，尤其是在涉及到机器人姿态控制、导航或其他需要考虑角度范围的应用场景中。

## **2.2 重要算法原理**

### 2.2.1 PID算法

原理：PID控制器用于根据机器人当前状态和目标状态的差异，调整机器人的控制输入。在这里，主要使用了比例项（Proportional，self.Kp）、积分项（Integral，self.Ki）、和微分项（Derivative，self.Kd）。

算法的数学表达式如下：



Error表示期望设定值与当前状态之间的差异（偏差）；

Kp​ 是比例系数，表示根据当前误差的大小进行调整。较大的 Kp​ 会导致更快的响应，但可能引起过冲（系统超调）；

Ki​ 是积分系数，表示根据误差随时间的积累来进行调整。 Ki​ 的作用是消除系统的稳态误差。过大的 Ki​ 可能导致系统超调或产生振荡；在控制系统中，稳态误差是指系统在达到稳定状态时与目标值之间的残差或偏差。简而言之，稳态误差表示系统无法完全达到预期的目标值。积分系数Ki​ 在PID控制中的作用是通过积分项来消除系统的稳态误差。积分的过程可以理解为对误差随时间的累积。如果系统存在稳态误差，积分项将随时间增加，对控制器的输出产生调整，逐渐减小稳态误差，直至系统收敛到期望值。

因此，通过调整Ki​ 的值，可以影响积分项的贡献，从而使系统更好地消除稳态误差。然而，过大的Ki​ 可能导致系统超调或振荡，因此通常需要仔细调整这个参数以实现系统的良好性能。

Kd​ 是微分系数，表示根据误差变化率的大小进行调整。Kd​ 的作用是减小系统的超调和振荡。较大的Kd​ 有助于抑制系统的振荡，但可能引入更多的噪声。

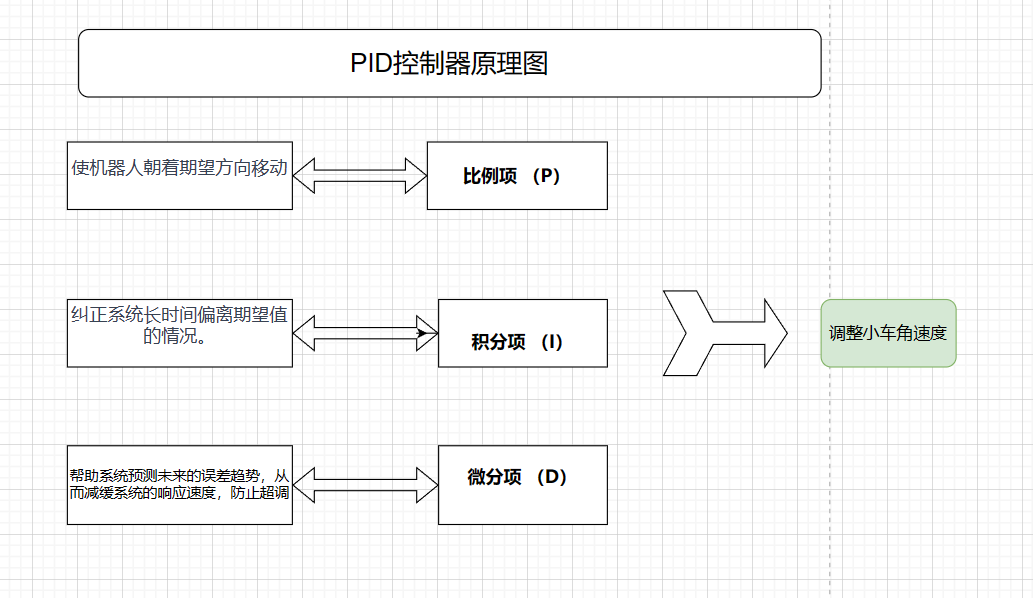


图10：PID控制原理图

### 2.2.2 里程计获取位置信息

里程计主要是为上面的PID算法做基础的，通过调用里程计信息，来获取小车的初始方向，同时也查看小车之后的方向。小车获取了这些信息。结合PID算法，控制小车的角速度，使其走在正确的路径上。

里程计（Odometry）是一种测量和估计机器人运动的方法，通常通过测量轮子旋转的角度来推断机器人的位姿（位置和方向）。在你的代码中，通过订阅/odom话题获取里程计信息，然后使用这些信息来估计小车的位姿。

以下是里程计获取小车位姿信息的算法原理：

获取里程计信息 (`rospy.Subscriber('/odom', Odometry, self.odom\_callback)'):

在ROS中，rospy.Subscriber用于订阅话题，你订阅了/odom话题，该话题发布机器人的里程计信息。

self.odom\_callback是回调函数，每当接收到里程计信息时都会调用该函数。

里程计信息中的位姿数据 (odom\_msg.pose.pose.orientation, odom\_msg.pose.pose.position):

里程计信息通常包括机器人的位姿，其中包括位置（position）和方向（orientation）。

你的代码中使用了odom\_msg.pose.pose.orientation获取四元数表示的方向信息。

将四元数转换为欧拉角 (self.get\_yaw\_from\_quaternion函数):

机器人的方向通常以四元数形式给出，因此需要将其转换为更直观的欧拉角（例如偏航角）。

self.get\_yaw\_from\_quaternion函数通过euler\_from\_quaternion函数从四元数中提取偏航角。

更新偏航角误差 (yaw\_error):

计算偏航角误差，即期望的偏航角与当前实际偏航角之间的差异。

用于后续的PID控制，以调整机器人的方向。

应用PID控制 (angular\_z = self.Kp \* yaw\_error + self.Ki \* (yaw\_error + self.last\_yaw\_error) + self.Kd \* (yaw\_error - self.last\_yaw\_error)):

使用PID算法来计算角速度，以使机器人朝向期望方向调整。

这是通过比例（P）、积分（I）和微分（D）项的加权和来实现的，从而对偏航角误差进行调整。

通过这个过程，你的代码实现了通过里程计信息获取小车的位姿，特别是偏航角，并使用PID控制来调整机器人的方向，使其能够沿着期望的轨迹行驶。

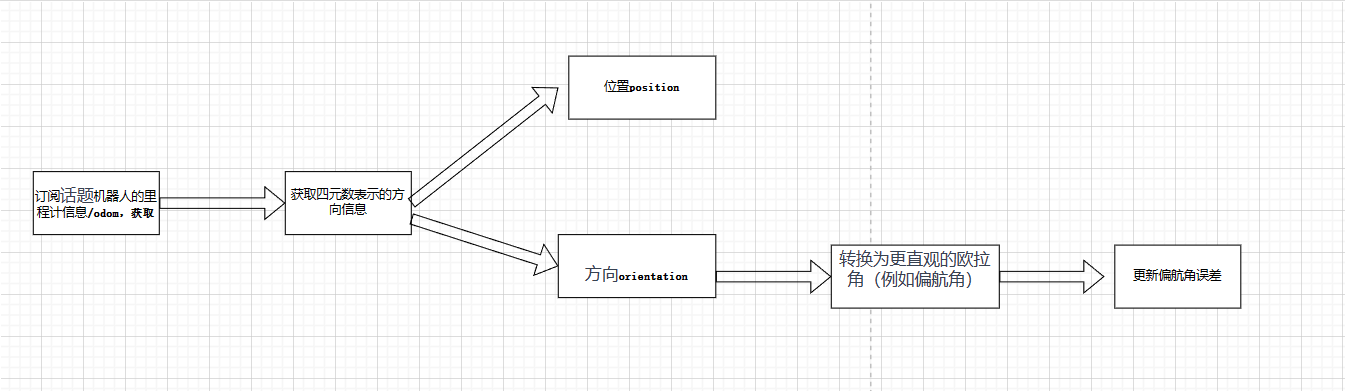


图11：航线角纠正原理图

### 2.2.3 识别障碍物

激光雷达（Lidar）通过发射激光束并测量其返回时间来获取环境中物体的距离，从而生成点云数据。在机器人导航中，激光雷达常用于检测障碍物和构建环境地图。以下是激光雷达检测障碍物的算法原理：

数据获取：

激光雷达以其位置为中心发射激光束，测量激光束与物体表面的交点。返回的数据通常以角度和距离的形式组成。

点云生成：

通过激光雷达多个角度的测量，可以得到一个点云，其中每个点的坐标表示一个物体表面上的点。

障碍物检测：

对点云数据进行处理，通常通过聚类、滤波等方法，将点云中的点分组，形成代表不同物体的簇（cluster）。

对每个簇进行分析，判断其是否代表障碍物。这通常包括考虑簇的密度、形状等特征。

阈值设置：

根据激光雷达的性能和环境特点，设置适当的阈值，以判断哪些点或簇被认为是障碍物。

例如，可以设置距离阈值，当某个点的距离小于该阈值时，将其认为是障碍物。

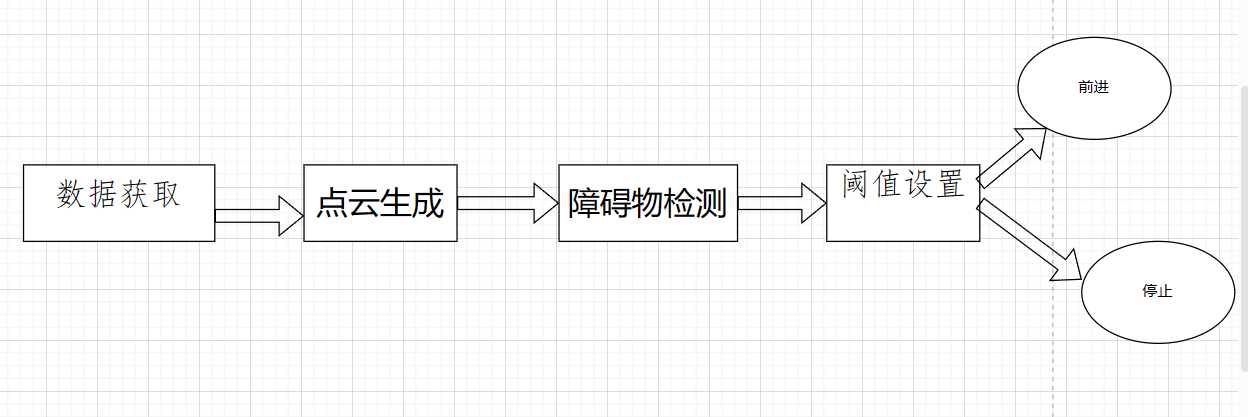


图12：障碍物识别原理图

### 2.2.4 低通滤波器

低通滤波器的作用是通过去除高频分量，使得信号变化更加平滑。这在传感器数据处理中常用于减少噪声的影响。当处理传感器数据时，特别是在实时系统中，噪声和突发的测量值可能导致不稳定的行为。低通滤波器用于消除这些高频噪声，从而保持平滑的输出。

以下是更详细的算法原理：

1.初始化：

在初始化阶段，通过构造函数 \_\_init\_\_ 设置滤波器的衰减系数 alpha，决定了新值对滤波输出的影响程度。衰减系数通常在0到1之间，越接近1，新值的影响越小。

初始时，filtered\_value 被设置为 None，以便在第一次更新时直接赋值。

2.更新滤波器状态 (update 方法)：

update 方法用于接收新的测量值 new\_value，并根据低通滤波器的更新规则计算出新的滤波后的值。

如果 filtered\_value 为 None，说明是第一次更新，直接将其设置为 new\_value。

否则，使用以下公式进行滤波器更新：

filtered\_value = alpha \* new\_value + (1 - alpha) \* filtered\_value

这个公式表示新的滤波值是当前输入值 new\_value 与上一时刻滤波值 filtered\_value 的加权和，其中 alpha 表示新值的权重，(1 - alpha) 表示上一时刻滤波值的权重。

3. 返回滤波后的值：

update 方法返回更新后的 filtered\_value。

4. 应用：

在实际应用中，这个滤波器可以应用于各种传感器数据，比如温度、光照、距离等。通过调整 alpha 的值，可以平衡对新值的响应速度和对噪声的抑制程度。

## **2.3 算法流程图**

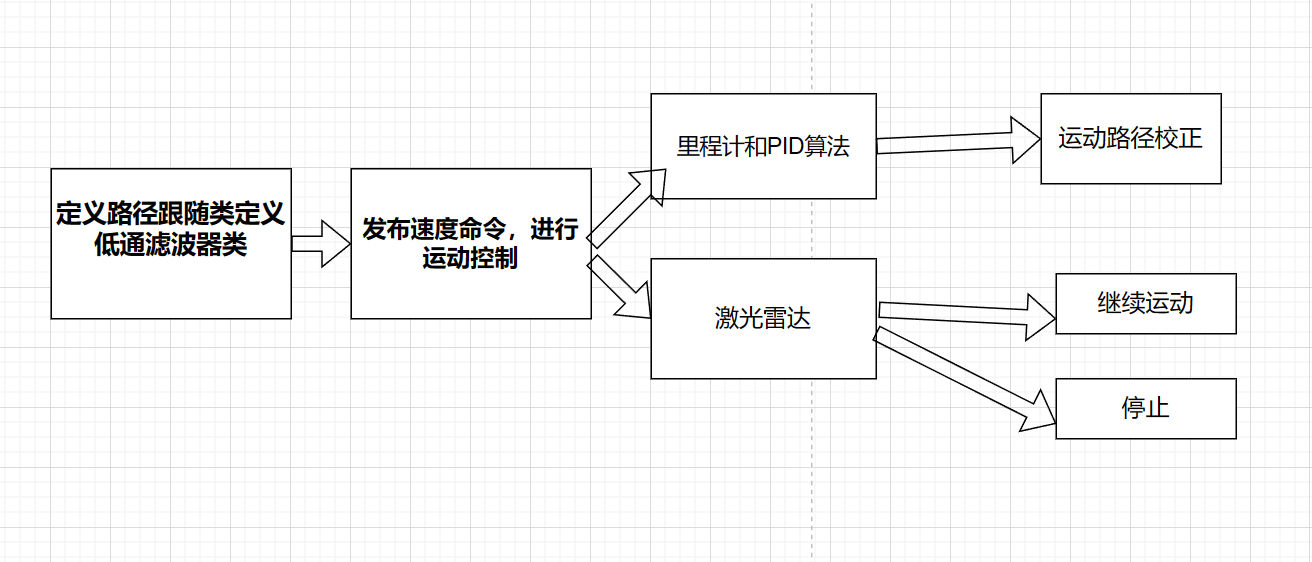


图13：整体算法流程图

## **2.4 小车运动轨迹图**

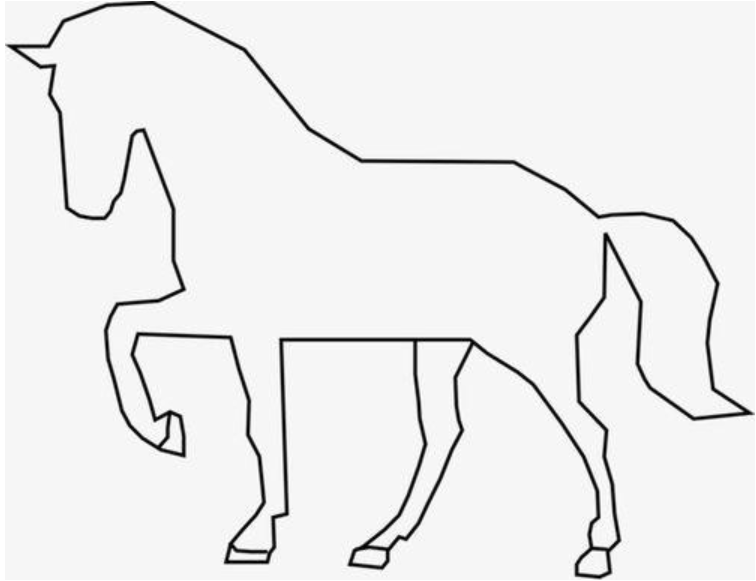


图14：输入小车图像

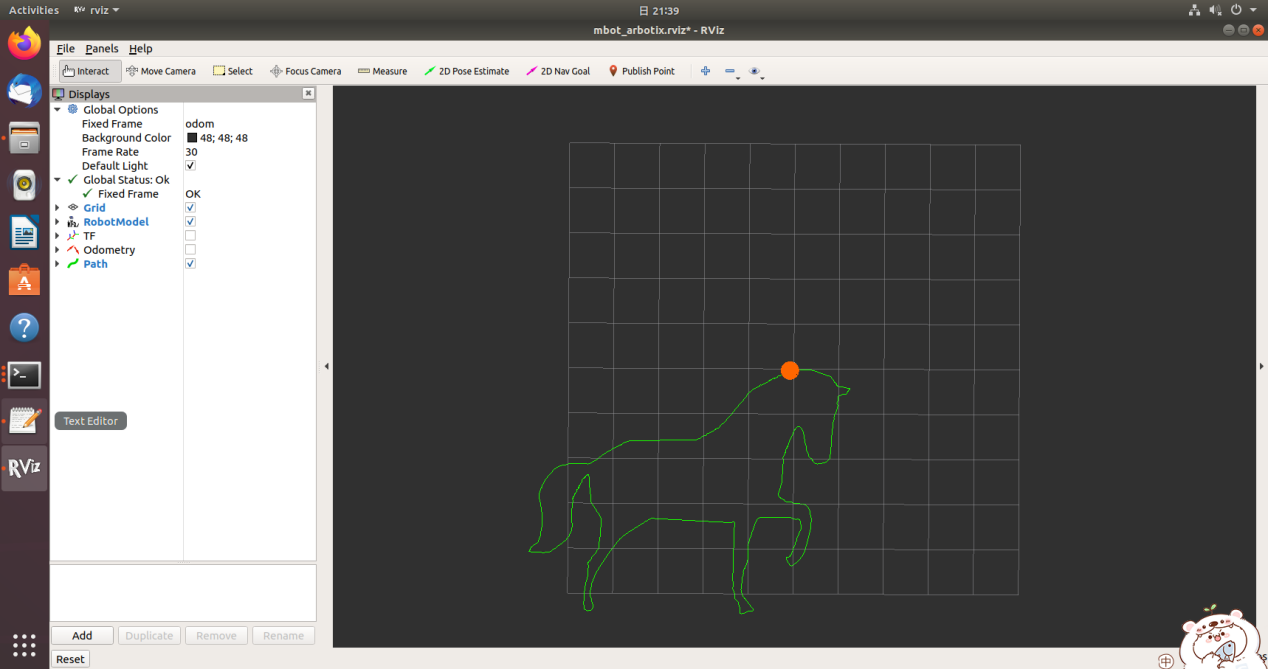


图15：小车运动轨迹图

# 3.语音控制

## **3.1 语音控制流程图**

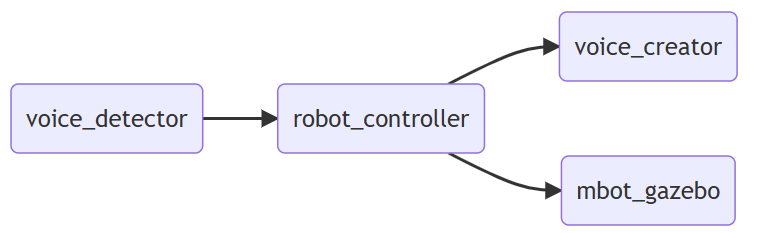


图16：语音控制流程图

voice\_detector：负责语音识别，将语音转换为文字，并作为 client，通过 human\_chatter 服务，发给 robot\_controller 。

robot\_controller：作为 human\_chatter 服务 server，接收 voice\_detector 发来的文字化的指令，并生成对应的语音播报文字和控车命令。前者通过 str2voice 服务，发给 voice\_creator，后者通过 /cmd\_vel topic，发给 mbot\_gazebo。

voice\_creator：作为 str2voice 服务server，接收 robot\_controller 发来的语音播报文字，合成语音文件并播放。

mbot\_gazebo：作为机器人小车，接收 /cmd\_vel topic，并调整运动状态。

## **3.2 语音控制实例步骤**

启动小车模型

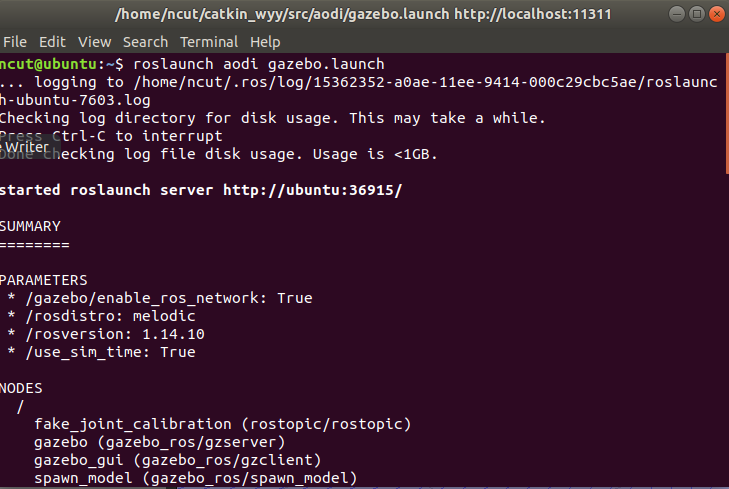


图17：小车模型启动命令

启动语音控制launch文件

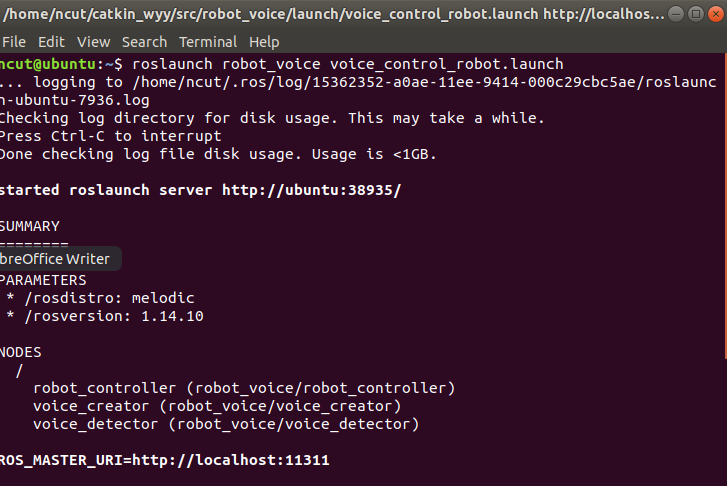


图18：语音控制启动命令

当通过麦克风说出“向后”时，小车接受到指令，进而向后运动

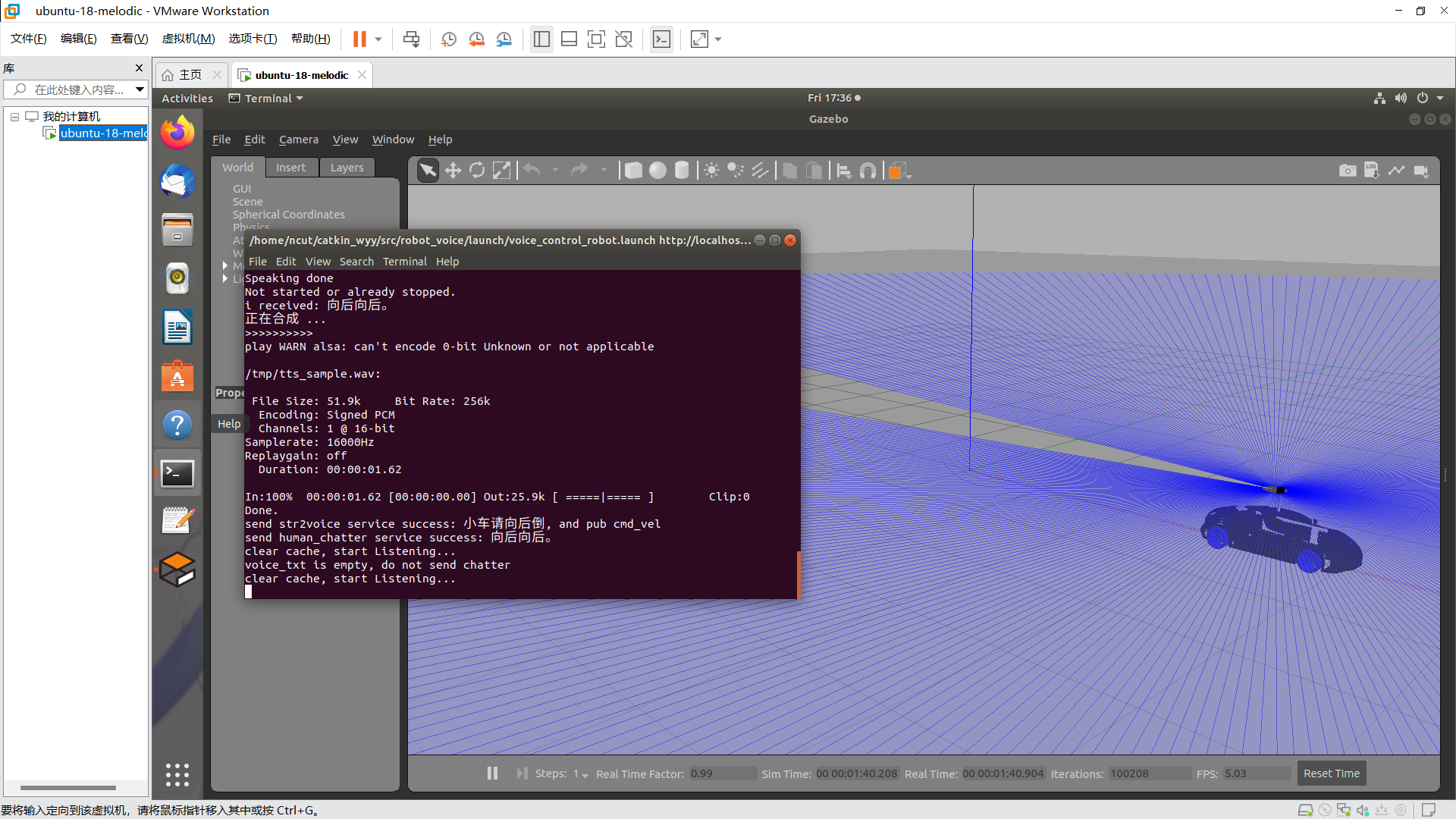


图19：小车接受语音指令运动图

当通过麦克风说出“向前”时，小车接受到指令，进而向前运动

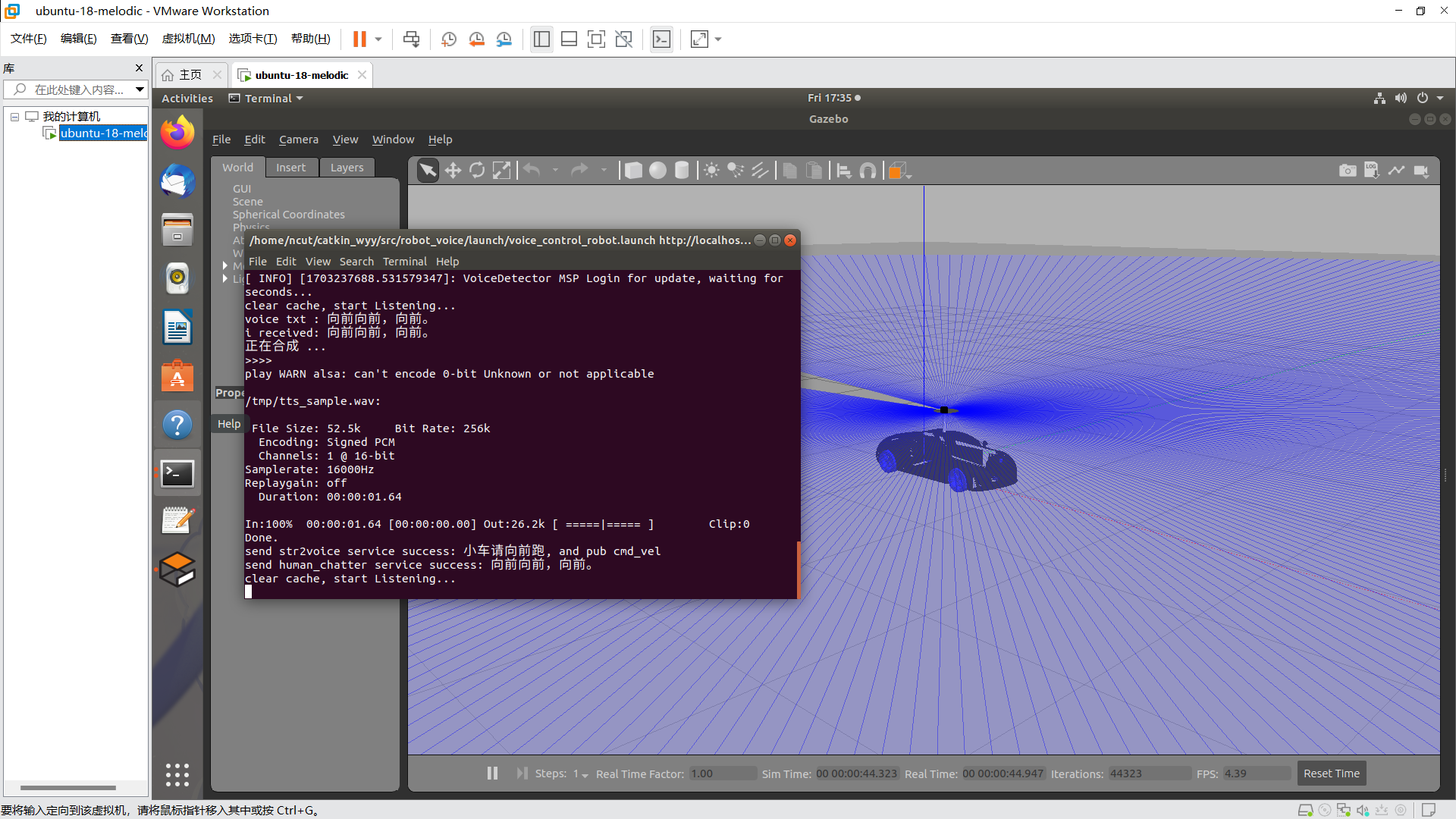


图20：小车接受语音指令运动图

# 4.视频控制

## **4.1 自主巡线**

计算原理公式

颜色阈值分割：

mask = cv2.inRange(hsv, lower\_yellow, upper\_yellow)

其中，hsv 是图像的 HSV 颜色空间表示，lower\_yellow 和 upper\_yellow 是黄色像素的 HSV 阈值。

矩阵计算：

M = cv2.moments(mask)

其中，M 是矩阵对象，包含图像的矩和重心信息。

比例控制：

err = cx - w/2

self.twist.linear.x = 0.2

self.twist.angular.z = -float(err) / 100

其中，err 是黄色目标中心点与图像中心点的水平距离，self.twist 是机器人的速度指令。

算法流程

图像获取：订阅相机图像话题，并将其转换为 NumPy 数组。

颜色阈值分割：将图像转换为 HSV 颜色空间，并创建黄色像素的二值掩码。

感兴趣区域 (ROI)：将黄色目标的搜索区域缩小到图像中的特定区域。

矩阵计算：计算掩码图像的矩以找到黄色区域的重心。

误差计算：确定重心与图像中心的水平误差。

速度控制：将线速度设置为常数值，并将角速度按比例设置为误差，以将机器人导向黄色目标。

命令发布：将速度指令发布到机器人的控制话题。

可视化：显示处理后的图像，其中包含检测到的黄色目标及其重心。

源代码解析

导入语句

rospy：用于 ROS 通信和节点设置。

cv2：用于图像处理和计算机视觉。

cv\_bridge：用于将 ROS 图像消息转换为 OpenCV 格式。

numpy：用于数值运算和数组操作。

sensor\_msgs.msg：用于处理图像消息。

geometry\_msgs.msg：用于处理速度指令。

Follower 类

\_\_init\_\_() 方法

\_\_init\_\_() 方法在节点初始化时调用。它负责创建 ROS 订阅者和发布者，并设置图像处理变量。

self.bridge：用于将 ROS 图像消息转换为 OpenCV 格式。

self.image\_sub：订阅相机图像话题。

self.cmd\_vel\_pub：发布速度指令话题。

self.twist：包含机器人速度指令的变量。

image\_callback() 方法

image\_callback() 方法在接收到新的图像消息时调用。它负责处理图像并发布速度指令。

image：接收到的图像消息。

hsv：将图像转换为 HSV 颜色空间后的结果。

mask：包含黄色像素的二值掩码。

M：包含图像矩和重心信息的矩阵对象。

cx：黄色区域重心点的水平坐标。

err：黄色区域重心点与图像中心点的水平距离。

def image\_callback(self, msg):

image = self.bridge.imgmsg\_to\_cv2(msg,desired\_encoding='bgr8')

hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

mask = cv2.inRange(hsv, lower\_yellow, upper\_yellow)

M = cv2.moments(mask)

if M['m00'] > 0:

cx = int(M['m10']/M['m00'])

err = cx - w/2

self.twist.linear.x = 0.2

self.twist.angular.z = -float(err) / 100

self.cmd\_vel\_pub.publish(self.twist)

cv2.circle(image, (cx, cy), 20, (0,0,255), -1)

cv2.imshow("window", image)

cv2.waitKey(3)

将图像转换为 OpenCV 格式：

image = self.bridge.imgmsg\_to\_cv2(msg,desired\_encoding='bgr8')

将图像转换为 HSV 颜色空间：

hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

应用颜色阈值分割以隔离黄色像素：

mask = cv2.inRange(hsv, lower\_yellow, upper\_yellow)

计算黄色区域的重心：

M = cv2.moments(mask)

计算重心与图像中心的误差：

cx = int(M['m10']/M['m00'])

err = cx - w/2

基于误差设置线速度和角速度：

self.twist.linear.x = 0.2

self.twist.angular.z = -float(err) / 100

发布速度指令：

self.cmd\_vel\_pub.publish(self.twist)

显示处理后的图像：

cv2.circle(image, (cx, cy), 20, (0,0,255), -1)

cv2.imshow("window", image)

cv2.waitKey(3)

主执行

初始化名为 follower 的 ROS 节点。

创建 Follower 类的实例。

运行 ROS 事件循环以处理传入的消息并持续处理图像。

在gazebo仿真环境下的实验

在进行自主巡线前可以使用rqt\_image\_view命令查看摄像头画面，发现摄像头画面不合适则需要修改xacro文件中camera和support的位置进行调整。如图分别为调整前后的画面。

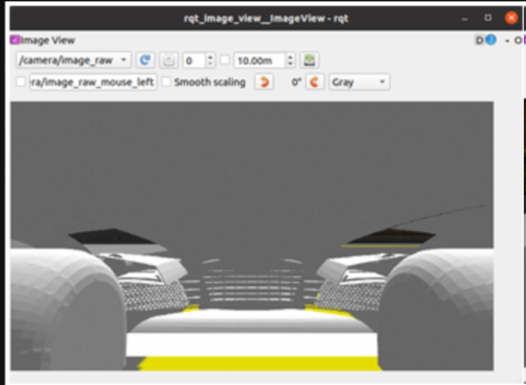


图21：摄像头调整前画面

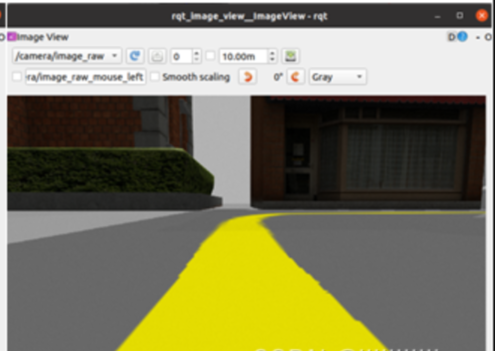


图22：摄像头调整后画面

下面是小车在黄色线圈上运动的图

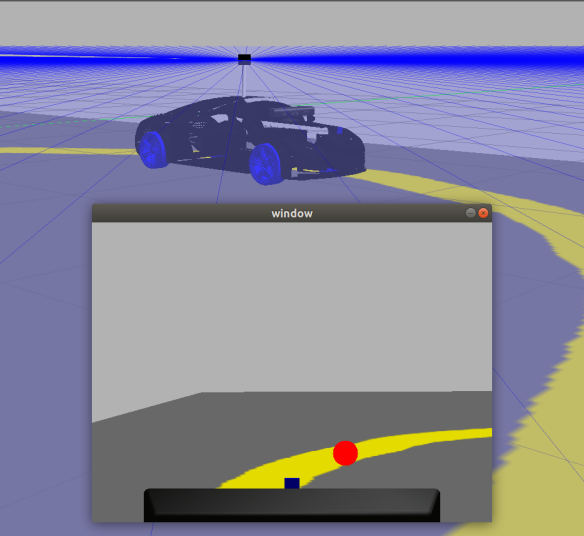


图23：小车沿黄线运动图（1）

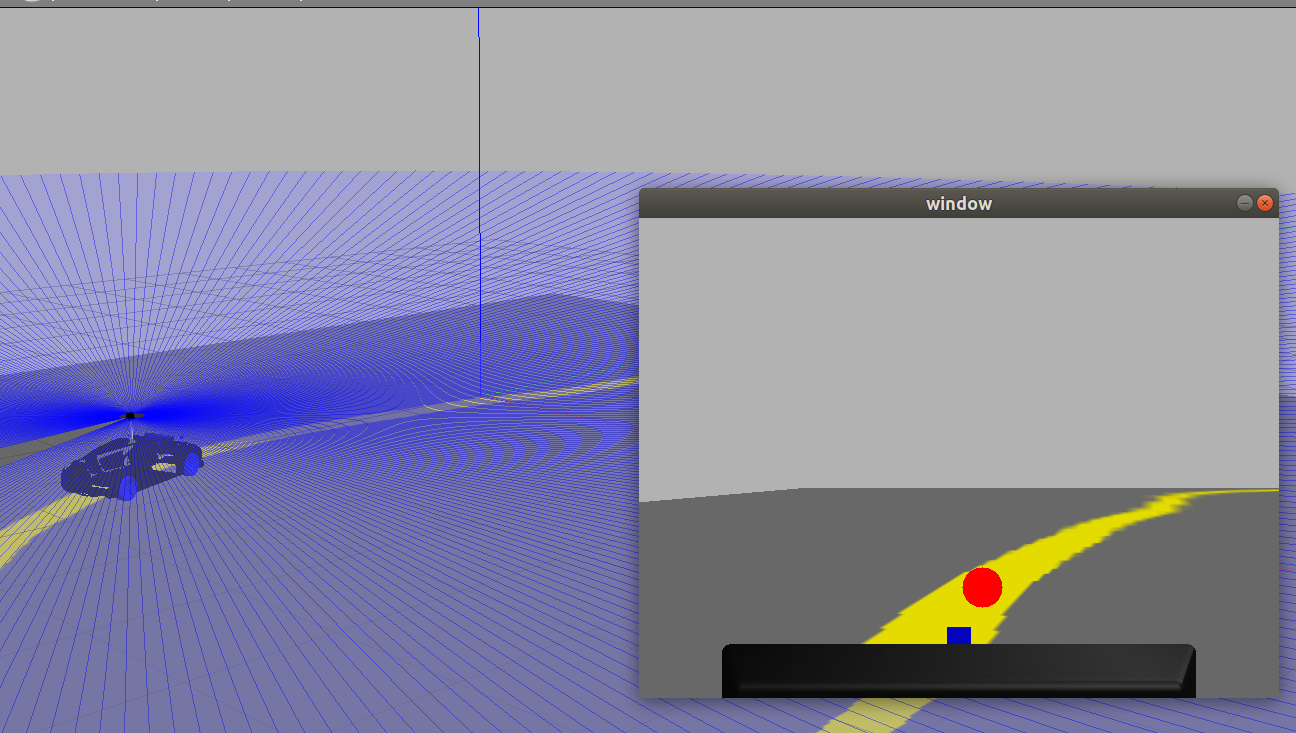


图24：小车沿黄线运动图（2）

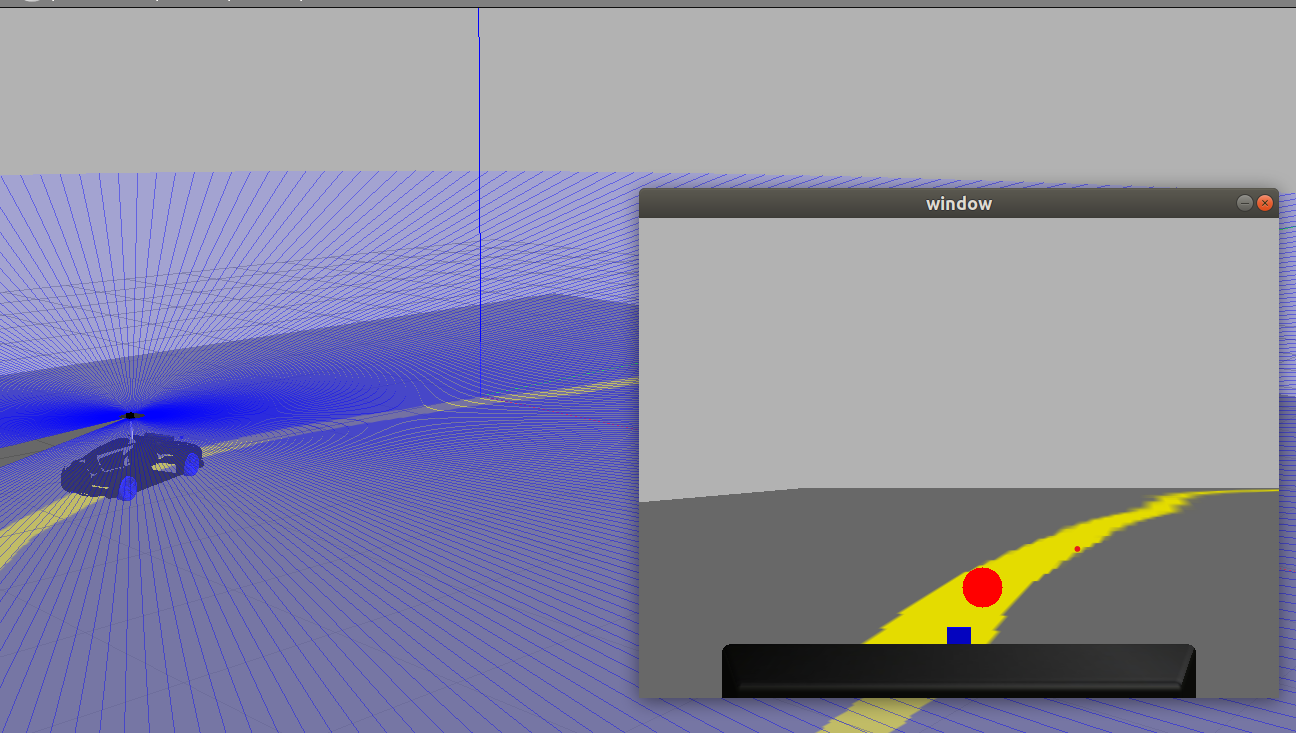


图25：小车沿黄线运动图（3）

# 5.建图与导航

## **5.1建图**

### 5.1.1world环境建立

world的创建方法有很多了，你可以自己画一个世界，也可以导入，这里我们是自己手动建立的模型，在gihub中下载的gazebo模型，可以在osrf/gazebo\_models: Gazebo database of SDF models. This is a predecessor to https://app.gazebosim.org (github.com)此网址下载各种各样的模型。

接下来我们使用roslaunch mbot\_gazebo model.launch打开查看世界：

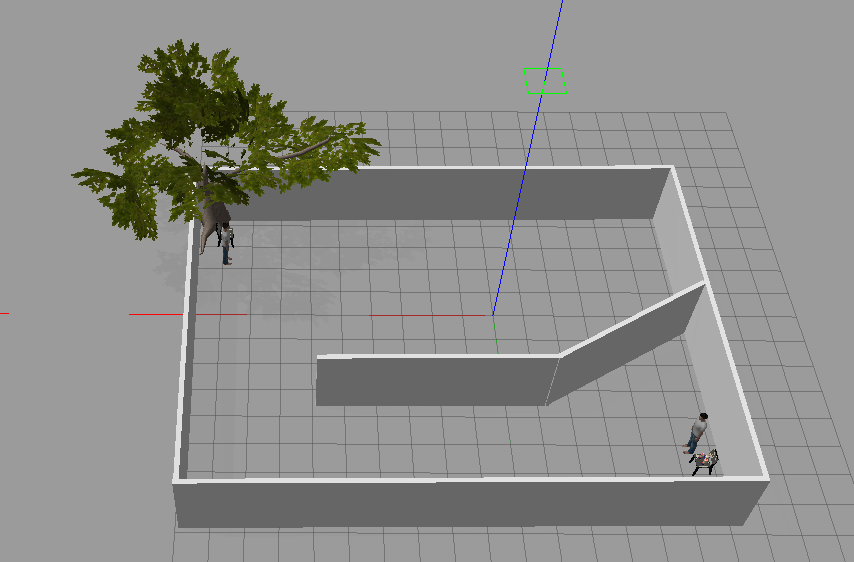


图26：建模环境

### 5.1.2 launch文件启动world环境并测试

创建一个启动文件mbot\_gazebo/launch/mbot\_laser\_nav\_gazebo.launch,运行Gazebo,加载机器人模型,并且启动一些必要的节点:

<launch>

<!-- 设置launch文件的参数 -->

<arg name="world\_name" value="$(find mbot\_gazebo)/worlds/citizen.world"/>

<arg name="paused" default="false"/>

<arg name="use\_sim\_time" default="true"/>

<arg name="gui" default="true"/>

<arg name="headless" default="false"/>

<arg name="debug" default="false"/>

<!-- 运行gazebo仿真环境 -->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<arg name="world\_name" value="$(arg world\_name)" />

<arg name="debug" value="$(arg debug)" />

<arg name="gui" value="$(arg gui)" />

<arg name="paused" value="$(arg paused)"/>

<arg name="use\_sim\_time" value="$(arg use\_sim\_time)"/>

<arg name="headless" value="$(arg headless)"/>

</include>

<node

name="xacro\_to\_urdf"

pkg="xacro"

type="xacro"

args="$(find aodi)/urdf/aodi\_union.xacro -o $(find aodi)/urdf/aodi.urdf" />

<node

name="spawn\_model"

pkg="gazebo\_ros"

type="spawn\_model"

args="-file $(find aodi)/urdf/aodi.urdf -urdf -model aodi"

output="screen" />

<!-- 运行joint\_state\_publisher节点，发布机器人的关节状态 -->

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher" ></node>

<!-- 运行robot\_state\_publisher节点，发布tf -->

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="robot\_state\_publisher" output="screen" >

<param name="publish\_frequency" type="double" value="50.0" />

</node>

</launch>

然后一键启动launch文件：

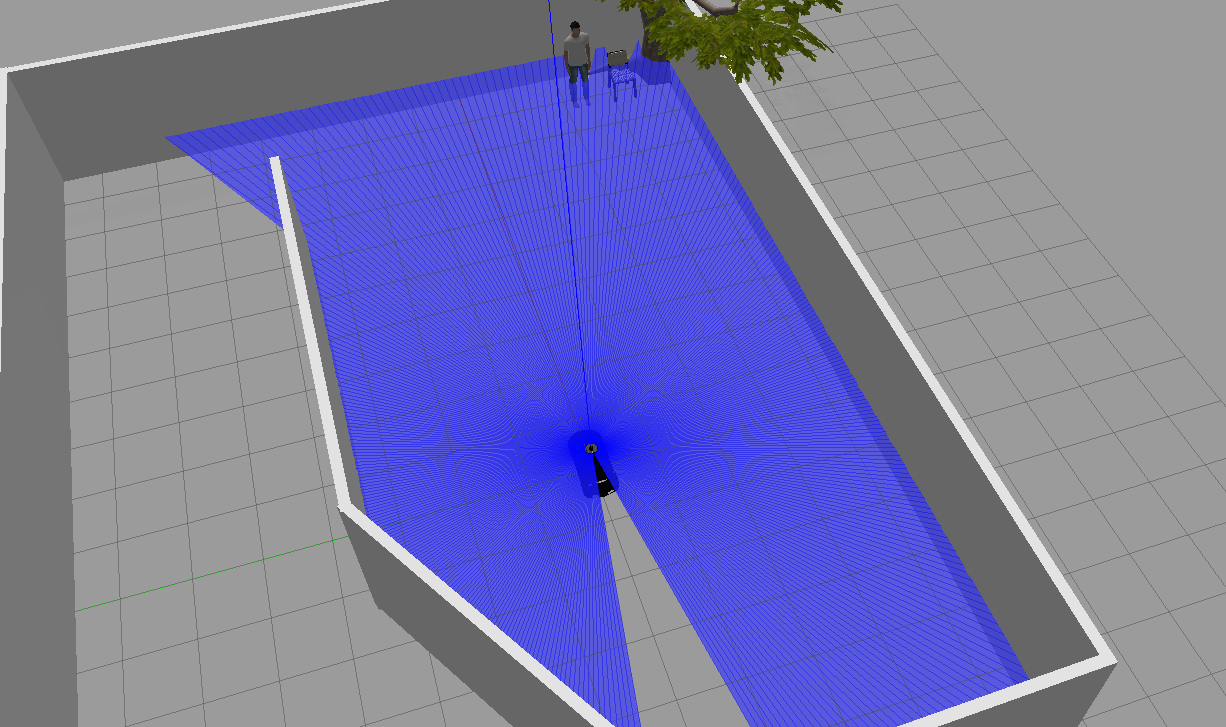


图27：小车载入环境

### 5.1.3 键盘控制小车

我们先通过roslaunch mbot\_gazebo mbot\_laser\_nav\_gazebo.launch, 来打开小车所在的仿真环境,然后再启动rosrun mbot\_teleop mbot\_teleop.py通过键盘控制小车。

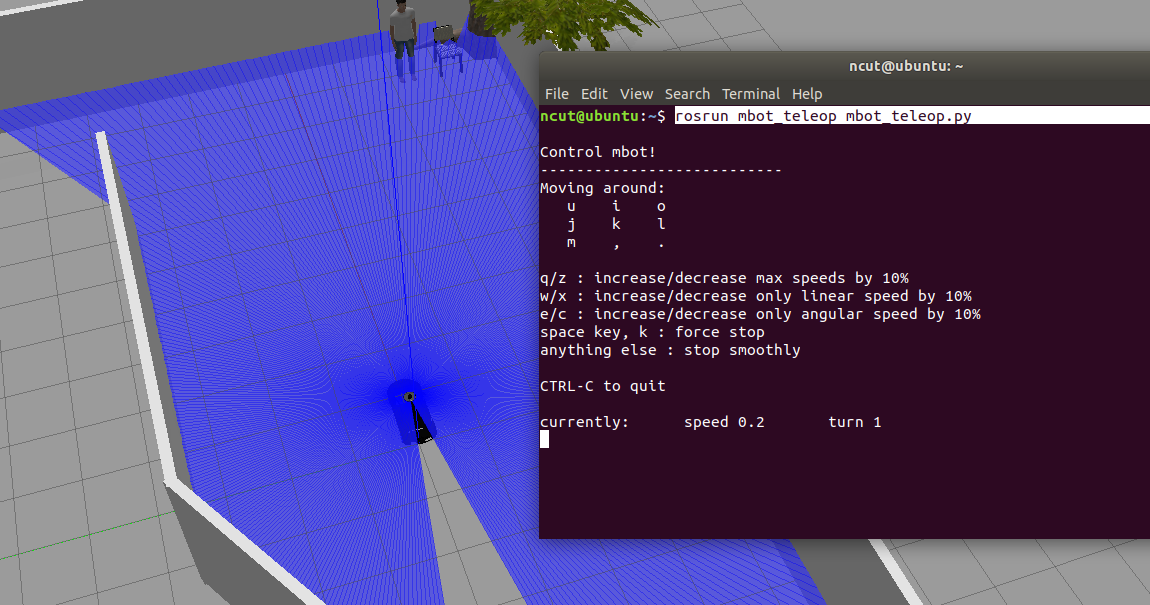


图28：用键盘控制该环境下的小车

3.1.4 launch文件启动并建图

启动rviz开始使用键盘控制小车建图，建图过程需要有耐心，最好速度不要太快，当然不同的算法适应性也不太一样，我们可以看看建图效果还是可以的：

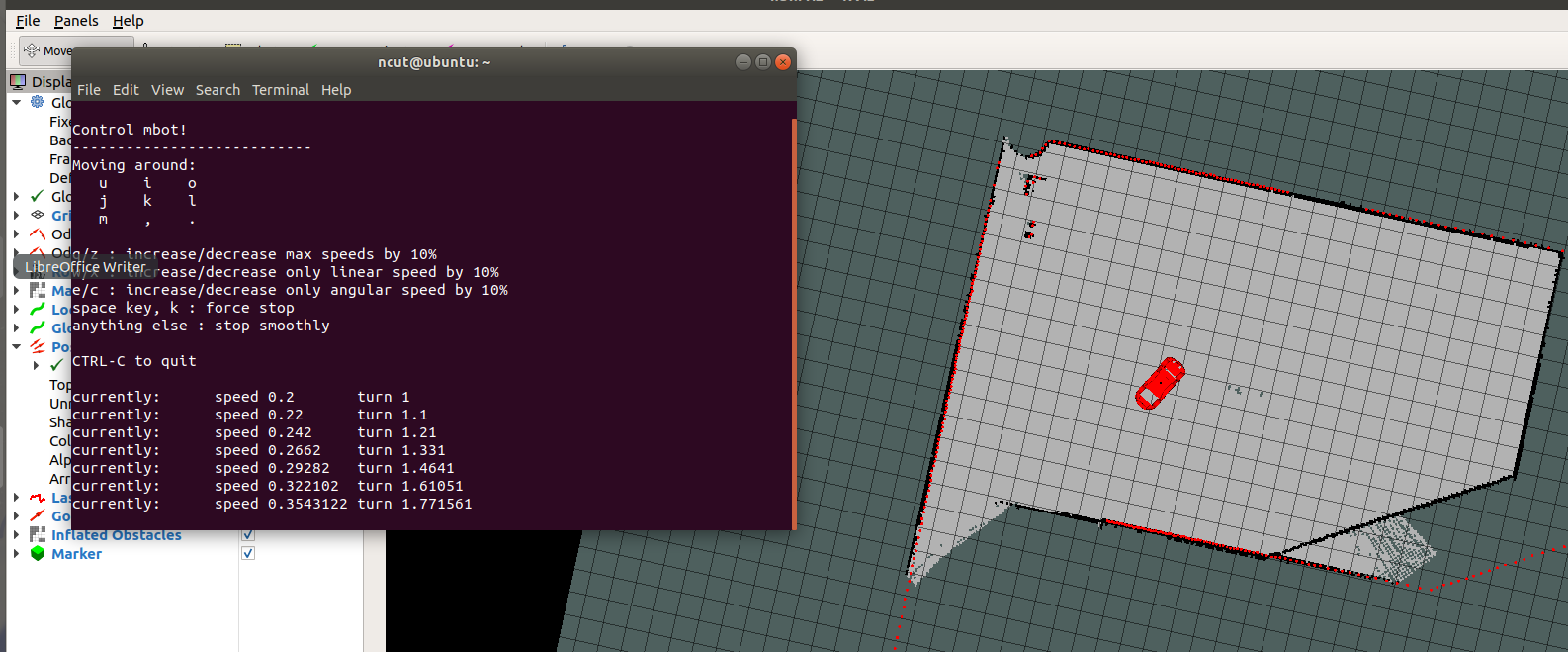


图29：小车SLAM建图

最后建成的地图我们需要及时保存。保存的地图一共有两个文件，map.pgm和map.yaml。让我们看一下建好的地图效果还是蛮不错的：

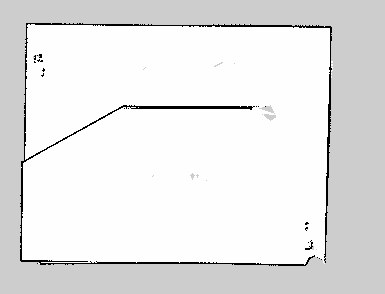


图30：保存建图结果

## **5.2导航**

### 5.2.1 launch一键启动文件

我们先来看一下导航的launch启动文件，第2到12行是加载小车模型，第13行到15行就是启动导航launch文件；第16～18行是加载配置好的rviz；

<launch>

<arg name="model" default="$(find xacro)/xacro --inorder '$(find aodi)/urdf/aodi\_union2.xacro'" />

<param name="robot\_description" command="$(arg model)" />

<node

name="joint\_state\_publisher\_gui"

pkg="joint\_state\_publisher\_gui"

type="joint\_state\_publisher\_gui" />

<node

name="robot\_state\_publisher"

pkg="robot\_state\_publisher"

type="robot\_state\_publisher" />

<include file="$(find mrobot\_navigation)/launch/gmapping.launch"/>

<!-- 运行move\_base节点 -->

<include file="$(find mrobot\_navigation)/launch/move\_base.launch" />

<!-- 运行rviz -->

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find mrobot\_navigation)/rviz/nav.rviz"/>

</launch>

5.2.2 代价地图配置

导航功能包使用两种代价地图存储周围环境中的障碍信息：一种用于全局路径规划(global\_costmap)，一种用于本地路径规划和实时避障(local\_costmap)。两种代价地图需要使用一些共用的或独立的配置文件：通用配置文件、全局规划配置文件和本地规划配置文件。config/navigation下这三个文件分别与之对应。

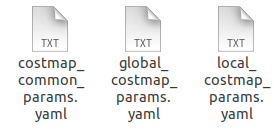


图31：配置文件

代价地图用来存储周围环境的障碍信息，其中需要声明地图关注的机器人传感器消息，以便于地图信息的更新。针对两种代价地图通用的配置选项，创建名为costmap\_common\_params.yaml的配置文件。全局规划配置文件用于存储配置全局代价地图的参数，命名为global\_costmap\_params.yaml，本地规划配置文件用来存储本地代价地图的配置参数，命名为local\_costmap\_params.yaml。

### 5.2.3 导航启动

在上面gazebo启动后，我们通过roslaunch mrobot\_navigation exploring\_slam\_demo.launch来开始导航前所有准备工作，然后开始导航。

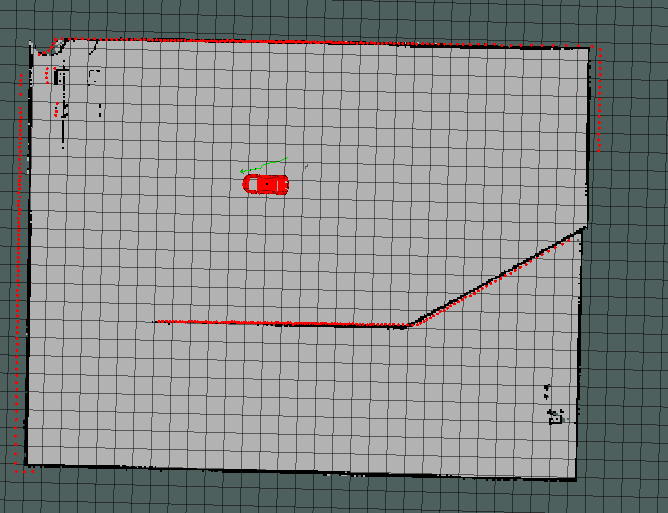


图32：小车在建好的图自动导航

我们可以通过rosrun rqt\_tf\_tree rqt\_tf\_tree来查看导航过程中的TF树：

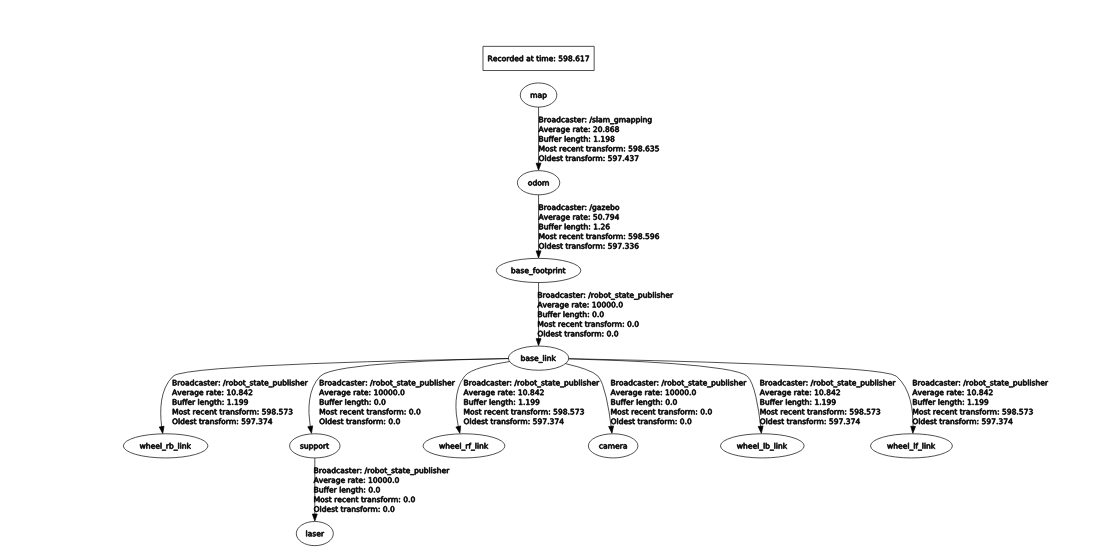


图33：导航过程TF树

# 6.应用案例开发

## **6.1 全局概括**

实现实时目标检测及语音播报：

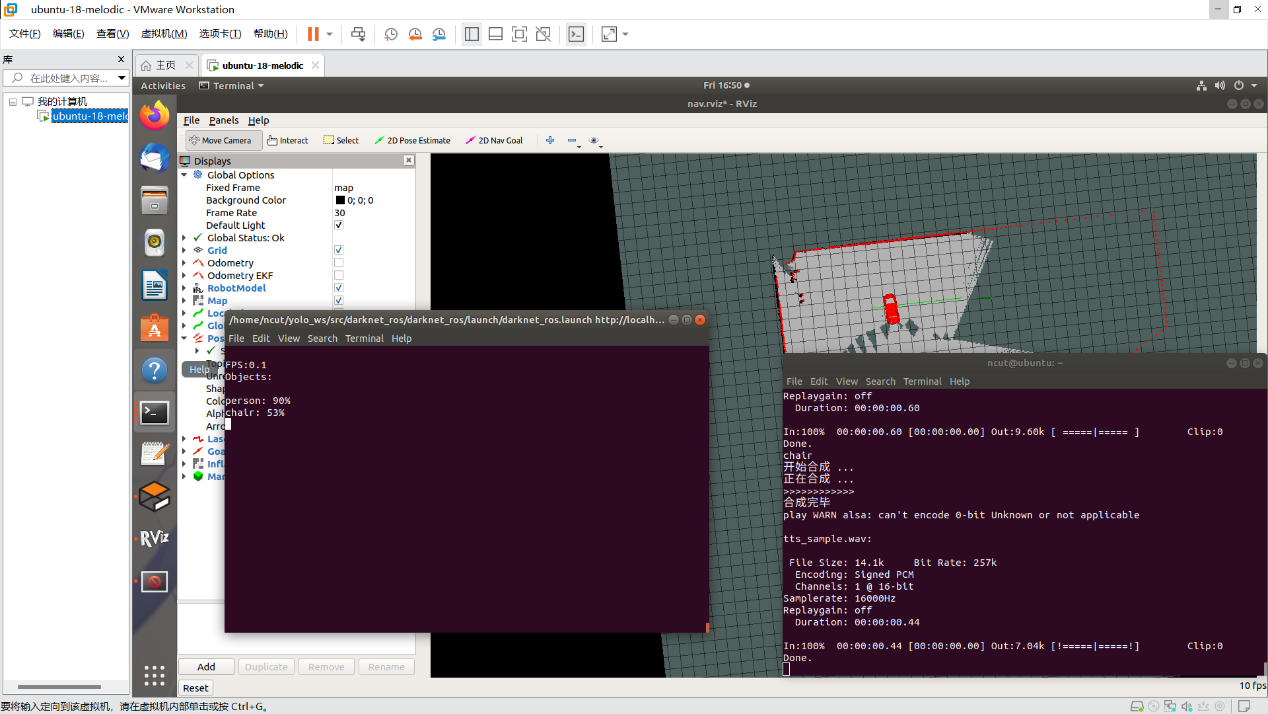


图34：目标检测与语音播报

各节点的通信情况：

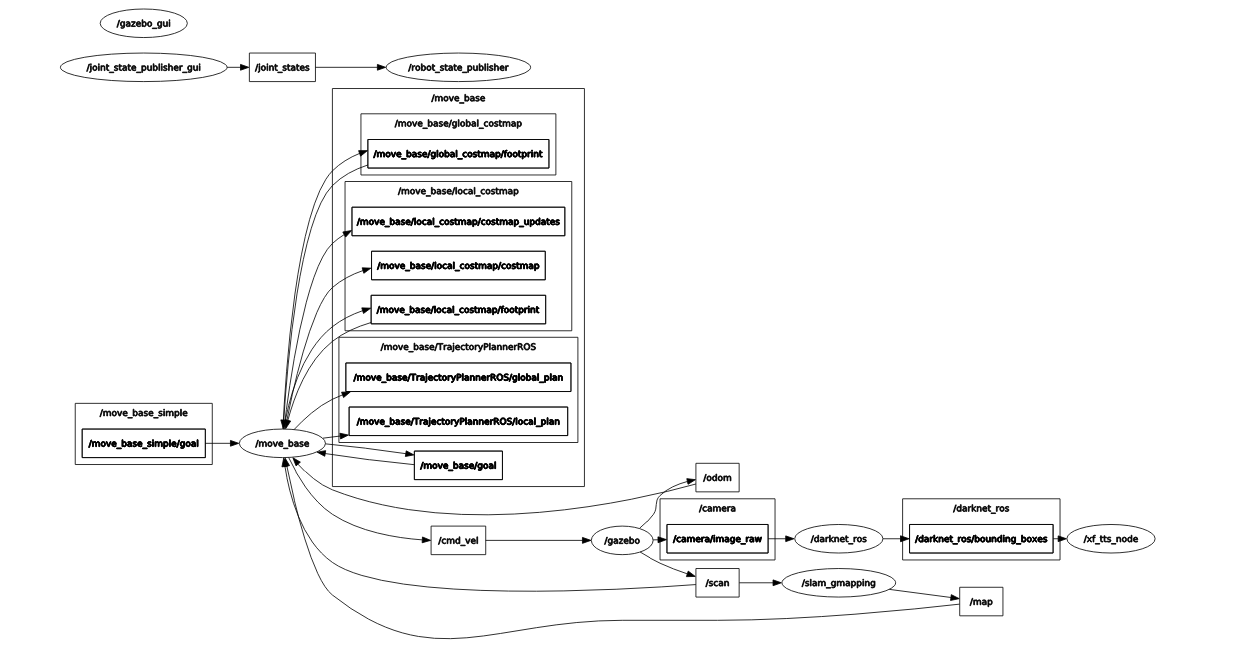


图35：节点通信情况

## **6.2 Yolov3目标检测**

### 6.2.1 图像话题发布

使用roslaunch usb\_cam usb\_cam-test.launch一键发布摄像头图像话题

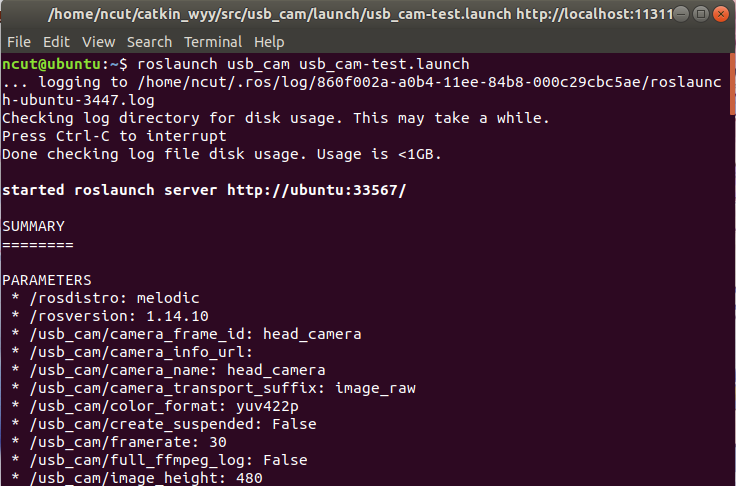


图36：发布摄像头图像话题

### 6.2.2 运行darknet\_ros

执行darknet\_ros进行检测，在运行检测之前需要更改一下配置文件，使得darknet\_ros订阅的话题与usb\_cam发布的图片话题对应。打开darknet\_ros/config/ros.yaml文件，找到：

subscribers:

camera\_reading:

topic: /camera/image\_raw

queue\_size: 1

将camera改为usb\_cam，然后roslaunch darknet\_ros darknet\_ros.launch将会看到以下效果：

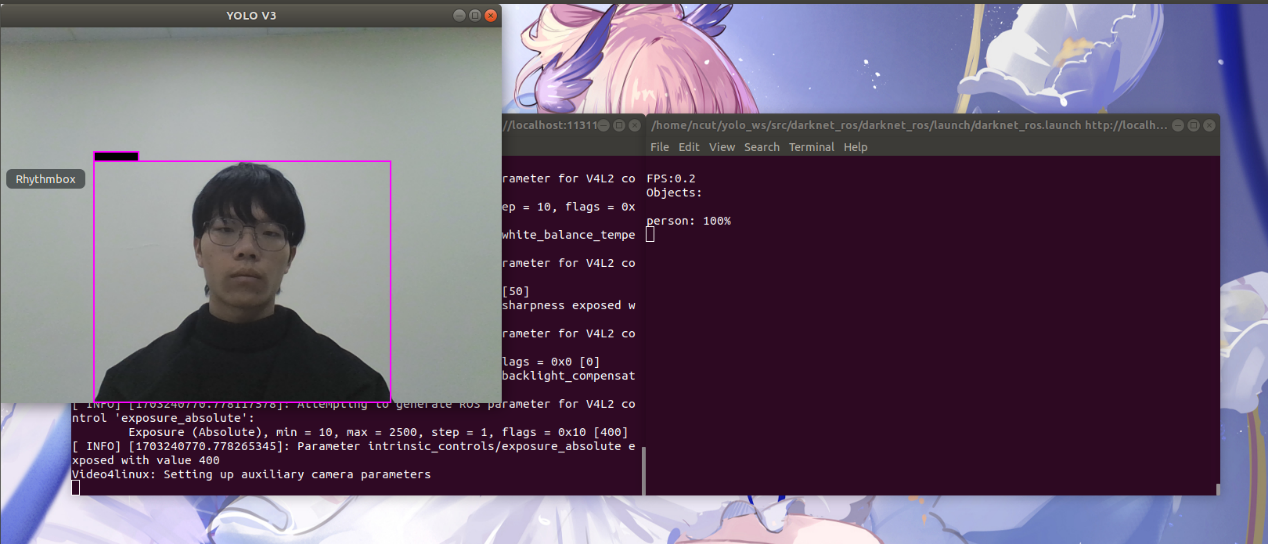


图37：usb摄像头的目标检测结果

接下来我们在小车摄像头上进行目标检测，将之前改动的地方还原。然后启动小车再建图的过程，再roslaunch darknet\_ros darknet\_ros.launch，其识别效果如下：

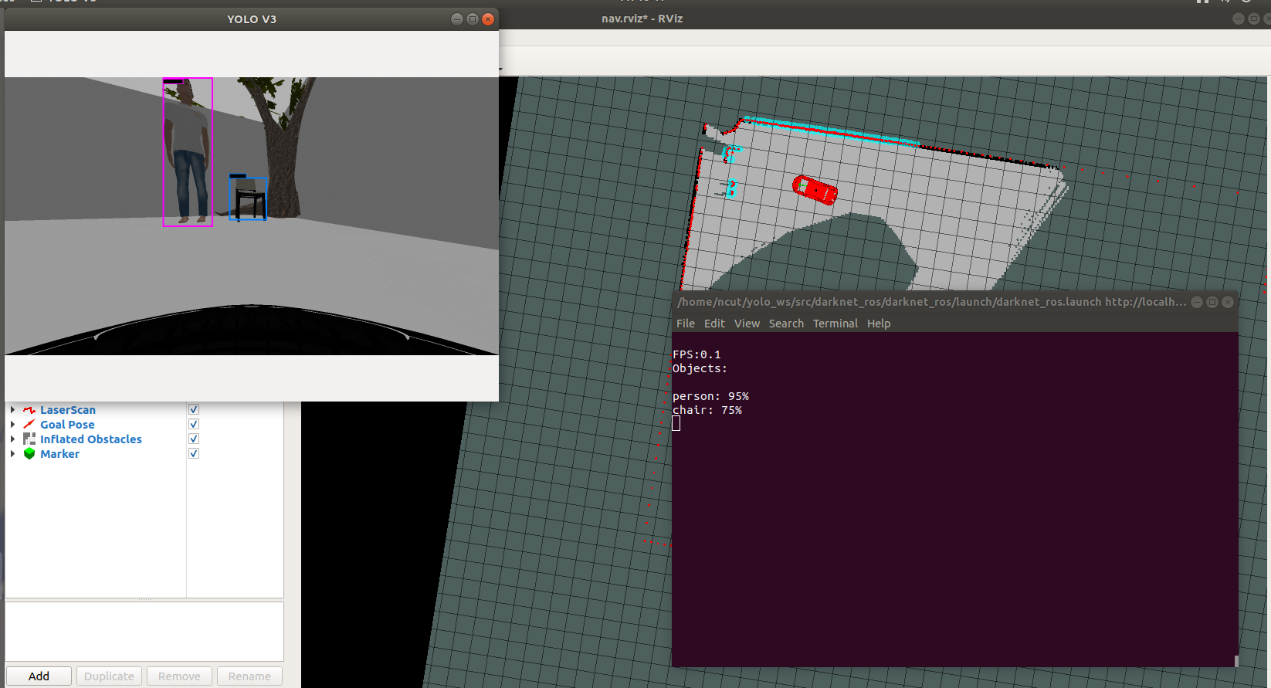


图38：小车摄像头的目标检测结果

## **6.3 语音播报**

### 6.3.1 yolo模块与语音模块的通信

首先yolo模块发布话题消息。

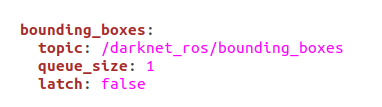


图39：yolo发布话题消息

然后再语音合成模块订阅话题消息，语音合成模块节点接收到darknet\_ros/bounding\_boxes的话题消息。



图40：订阅话题消息

最后回调函数调用darknet\_ros/bounding\_boxes的话题消息，生成音频语音对识别的物体进行播报。

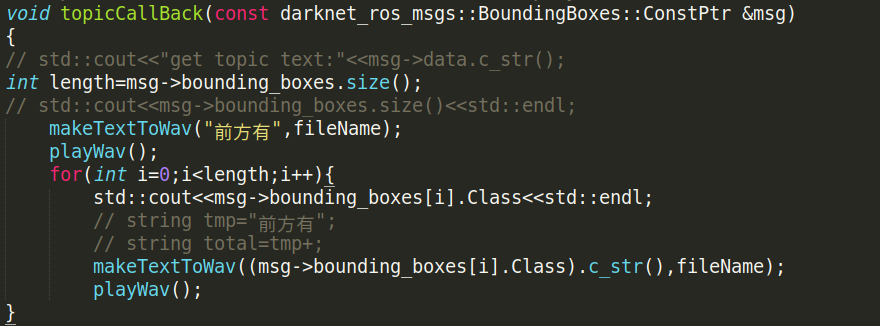


图41：回调函数

接下来我们看一下语音播报的效果：

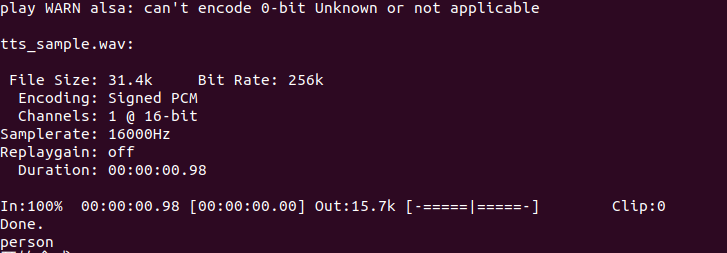


图42：语音播报结果（1）

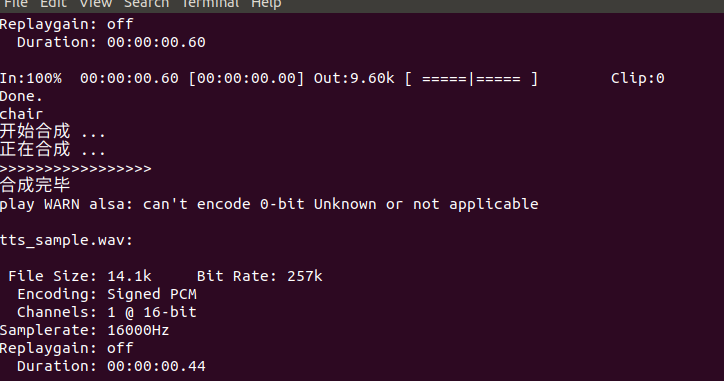


图43：语音播报结果（2）

# 总结

为了实现本次实验内容，我们自主设计了机器人URDF模型，结合科大讯飞对小车实现了语音控制和语音播报还有视频控制，使用yolo3实现了目标检测，自主研究算法结合PID控制和OpenCV实现了图像控制小车运动与避障功能和自主巡线功能。我们还自主搭建了gazebo仿真环境，实现了小车自主导航与SLAM建图功能。在自主搭建的gazebo物理仿真环境下运用yolo3和科大讯飞实现的智能应用场景能够让小车在自主导航或者语音控制运动过程中实时构建地图场景，并且对遇到的障碍物进行识别并进行语音播报，并将障碍物类型显示在终端上。