# 机器人导航

技术报告

学 院：电气与控制工程学院

班 级：ROS511858

姓 名：ROS511858

学 号：ROS51185

**目录**

**[实验一 slam建图实验（gmapping算法） 1](#_Toc30342)**

**[实验二 基于里程计数据的 PID 控制器设计实验 5](#_Toc9442)**

**[实验三 基于slam建图的 Move\_base 导航实验 8](#_Toc25867)**

**[实验四 基于雷达数据的锥桶障碍物停车实验 12](#_Toc12558)**

**[实验五 基于move\_base框架的多点导航实验 16](#_Toc13262)**

**实验一** **slam建图实验（gmapping算法）**

**实验目的：**

通过ROS框架在Gazebo仿真环境中实现基于激光雷达的二维地图构建，掌握gmapping算法参数调优方法，完成对各world环境的精确建模。

**实验内容：**

1、搭建slam建图环境

2、配置gmapping算法参数组

3、基于键盘控制实现环境探索并建图

**实验仪器：**

Gazebo 仿真平台、ROS Noetic、仿真小车、激光雷达

**实验原理：**

gmapping采用RBPF（Rao-Blackwellized Particle Filter）算法实现，通过粒子滤波维护多个候选轨迹假设，每个粒子携带独立地图副本。

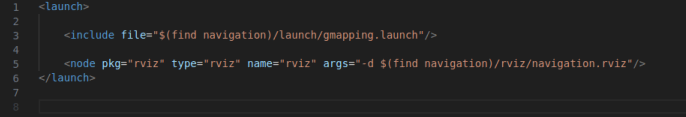
**实验步骤：**

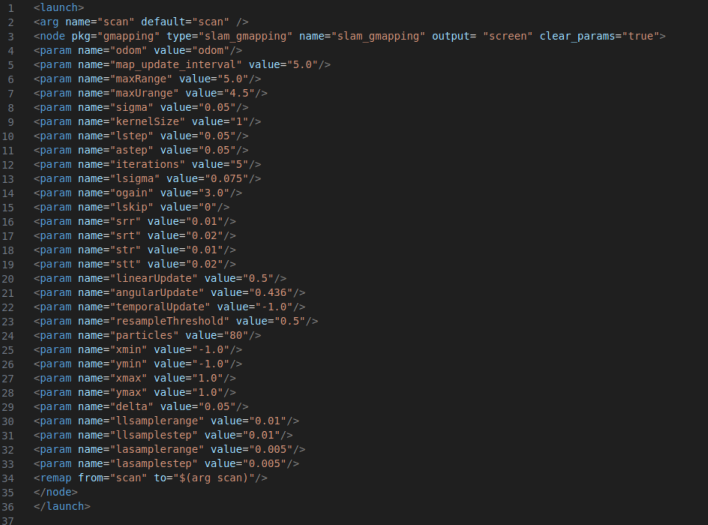
**<1> 环境初始化**

|  |
| --- |
| $ roslaunch robot\_task ucar\_robot.launch |

**<2> 参数化gmapping启动(新开一个终端输入下面命令)**

|  |
| --- |
| $ roslaunch navigation gmapping\_demo.launch |



下面我们来看一下Gmapping的launch文件各参数含义：

**其中比较重要的如下**

map\_update\_interval（float，默认值：5.0）

在更新地图之间的时间长度（以秒为单位）。降低该数量更频繁地更新占用网格，代价是更大的计算负荷。

minimumScore（float，default：0.0）

最小匹配得分，这个参数很重要，它决定了对激光的一个置信度，越高说明对激光匹配算法的要求越高，激光的匹配也越容易失败而转去使用里程计数据，而设的太低又会使地图中出现大量噪声，所以需要权衡调整。

linearUpdate（float，default：1.0）

每次机器人移动这么远时，处理一次扫描。

angularUpdate（float，default：0.5）

每次机器人旋转这么远时处理一次扫描。

particles (int, default: 30)

算法中的粒子数，因为gmapping使用的是粒子滤波算法，粒子在不断地迭代更新，所以选取一个合适的粒子数可以让算法在保证比较准确的同时有较高的速度。

|  |
| --- |
| <node pkg="gmapping" type="slam\_gmapping" name="slam\_gmapping" output="screen">//启动slam的节点  <param name="base\_frame" value="$(arg base\_frame)"/>  <param name="odom\_frame" value="$(arg odom\_frame)"/>  <param name="map\_update\_interval" value="0.01"/>//地图更新的一个间隔，两次scanmatch的间隔，地图更新也受scanmach的影响，如果scanmatch没有成功的话，是不会更新地图的  <param name="maxUrange" value="4.0"/>//set maxUrange < maximum range of the real sensor <= maxRange  <param name="maxRange" value="5.0"/>  <param name="sigma" value="0.05"/>  <param name="kernelSize" value="3"/>  <param name="lstep" value="0.05"/>optimize机器人移动的初始值（距离）  <param name="astep" value="0.05"/>//optimize机器人移动的初始值（角度）  <param name="iterations" value="5"/>//icp的迭代次数  <param name="lsigma" value="0.075"/>  <param name="ogain" value="3.0"/>  <param name="lskip" value="0"/>//为0,表示所有的激光都处理，尽可能为零，如果计算压力过大，可以改成1  <param name="minimumScore" value="30"/>//很重要，判断scanmatch是否成功的阈值，过高的话会使scanmatch失败，从而影响地图更新速率  <param name="srr" value="0.01"/>//以下四个参数是运动模型的噪声参数  <param name="srt" value="0.02"/>  <param name="str" value="0.01"/>  <param name="stt" value="0.02"/>  <param name="linearUpdate" value="0.05"/>//机器人移动linearUpdate距离，进行scanmatch  <param name="angularUpdate" value="0.0436"/>机器人选装angularUpdate角度，进行scanmatch  <param name="temporalUpdate" value="-1.0"/>  <param name="resampleThreshold" value="0.5"/>  <param name="particles" value="8"/>//很重要，粒子个数  <!--  <param name="xmin" value="-50.0"/>  <param name="ymin" value="-50.0"/>  <param name="xmax" value="50.0"/>  <param name="ymax" value="50.0"/>  make the starting size small for the benefit of the Android client's memory...  -->  <param name="xmin" value="-1.0"/>//map初始化的大小  <param name="ymin" value="-1.0"/>  <param name="xmax" value="1.0"/>  <param name="ymax" value="1.0"/>  <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"/>  <remap from="scan" to="$(arg scan\_topic)"/>  </node> |

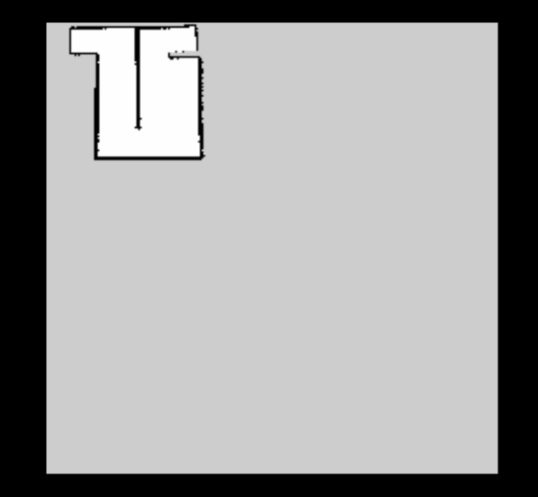
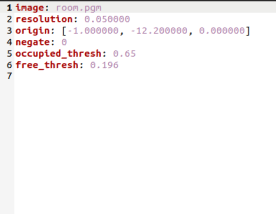
**<3>** **使用** **teleop\_twist\_keyboard 对小车进行控制**

|  |
| --- |
| $ rosrun turtlebot3\_teleop turtlebot3\_teleop\_key |

**<4>** **保存地图**

|  |
| --- |
| $ rosrun map\_server map\_saver -f room |

room.pgm room.yaml



**实验二** 基于里程计数据的 PID 控制器设计实验

**实验目的：**

通过编写PID控制算法，结合ROS框架，在Gazebo仿真环境中完成小车的闭环控制任务，使其从A点自动移动到B点，使用激光雷达或里程计数据实现路径规划与目标点追踪。

**实验内容：**

1. 搭建Task1仿真环境：加载task1.world，初始化机器人和传感器模型。
2. 基于PID算法完成运动控制：订阅（/odom）里程计判断机器人当前位置。计算误差并通过PID算法对小车的速度和角度进行反馈控制。
3. 从A点到B点的移动：实现路径闭环控制，从起点精准到达B点，实现任务目标。

**实验仪器：**

Gazebo 仿真平台、ROS Noetic、仿真小车、激光雷达、里程计

**实验原理：**

本实验采用PID控制器完成小车从A到B的运动控制任务。其设计要点如下：

误差分析：通过Odom坐标获取实时位姿 (x, y, θ)，计算当前位置与目标点的线性误差（距离）和角度误差。

PID控制器设计：

比例项（P）：根据实时误差生成控制量，快速逼近目标点。

积分项（I）：累积误差消除系统稳态误差。

微分项（D）：预测误差变化趋势，抑制超调和震荡。

控制量计算与发布：

输出线速度和角速度：线速度与角速度（Twist消息）控制小车的前进与转向。

闭环实时控制：通过订阅Odom获取反馈数据进行实时更新，直到误差足够小。

**实验步骤：**

**<1> 环境初始化**

|  |
| --- |
| $ roslaunch robot\_task ucar\_robot.launch |

**<2> 运行程序控制机器人运动到目标点(新开一个终端输入下面命令)**

|  |
| --- |
| $ rosrun robot\_task task2 |

**核心代码与函数分析**

（1）核心逻辑：PID控制运动闭环

控制过程概述，利用Odom实时获取机器人位姿 (x, y, θ)。

计算目标点与当前位置的误差：

目标距离 (linear error)：

目标角度 (angular error)：

通过PID控制器不断迭代调整线速度和角速度，逐步逼近目标点。

（2）核心函数

moveForward()：

|  |
| --- |
| // 前进指定距离  void moveForward(double distance) {  geometry\_msgs::Twist cmd\_vel;  double start\_x = current\_x\_;  double start\_y = current\_y\_;  while (ros::ok()) {  ros::spinOnce(); // \*\*确保订阅回调生效\*\*    double dx = current\_x\_ - start\_x;  double dy = current\_y\_ - start\_y;  double traveled = std::sqrt(dx \* dx + dy \* dy);  if (traveled >= distance - 0.03) {  cmd\_vel.linear.x = 0;  cmd\_vel\_pub\_.publish(cmd\_vel);  ROS\_INFO("前进 %.2f 米完成！", distance);  break;  }  cmd\_vel.linear.x = std::min(kp\_linear\_ \* (distance - traveled), 0.3);  cmd\_vel\_pub\_.publish(cmd\_vel);  rate\_.sleep();  }  } |

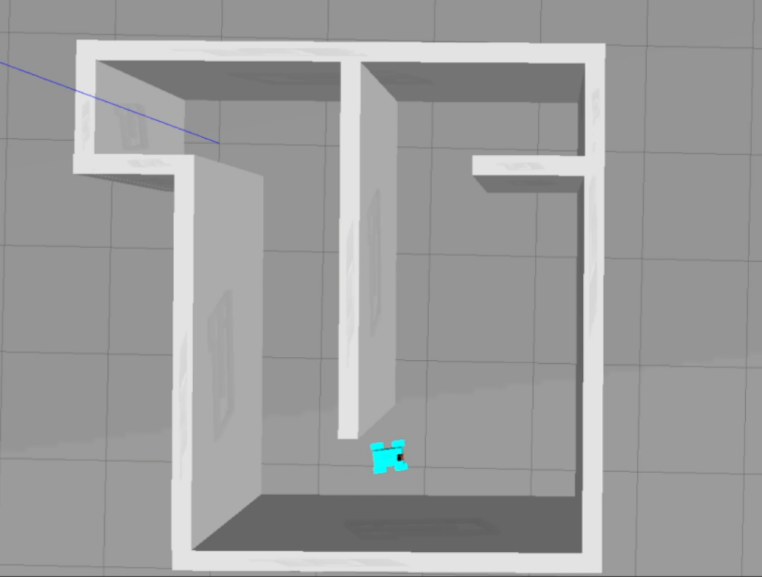
rotate()：

|  |
| --- |
| void rotate(double target\_angle) {  geometry\_msgs::Twist cmd\_vel;  while (ros::ok()) {  ros::spinOnce(); // \*\*确保订阅回调生效\*\*  double angle\_diff = normalizeAngle(target\_angle - current\_theta\_);  if (std::abs(angle\_diff) < 0.03) {  cmd\_vel.angular.z = 0;  cmd\_vel\_pub\_.publish(cmd\_vel);  ROS\_INFO("旋转到 %.2f 弧度完成！", target\_angle);  break;  } else {  cmd\_vel.angular.z = std::min(0.5 \* angle\_diff, 0.5);  }  cmd\_vel\_pub\_.publish(cmd\_vel);  rate\_.sleep();  }  } |

moveForward()：计算机器人的当前位置与起始位置之间的直线距离误差，并通过一个简单的比例控制（P控制）动态调整线速度 ，引导机器人以平滑且逐渐减速的方式向前移动指定距离。当误差小于阈值（距离目标剩余 0.03 米以内）时，停止前进。

rotate()：计算当前机器人姿态角与目标姿态角之间的角度误差，通过一个比例控制器调整角速度 ，使机器人以平滑的方式逐步旋转至目标角度。函数会持续更新角度误差，直至其小于阈值（角度误差小于 0.03 弧度）时停止旋转。

通过前进、转向、前进...相结合，实现从A点到B点的移动



**实验三** 基于slam建图的 Move\_base 导航实验

**实验目的：**

通过构建基于move\_base框架的路径规划系统，并结合Gazebo仿真平台，完成小车从A点到B点的导航任务。选择合适的局部路径规划算法（TEB或DWA），并评估路径规划与导航算法的性能。

**实验内容：**

1、 搭建move\_base导航环境，包括参数配置、代价地图。

2、使用TEB或DWA局部路径规划算法，处理复杂环境中的小车避障与路径规划能力。

3、实现小车从指定位置A点导航到目标位置B点。验证路径规划有效性。

**实验仪器：**

Gazebo 仿真平台、ROS Noetic、仿真小车模型、RViz

**实验原理：**

1、机器人自主导航：

全局路径规划：从静态地图中规划出一条从起始到目标的最优路径（基于A\*/Dijkstra算法）。

局部路径规划：根据全局路径和传感器数据，实时规划无碰撞路径，使小车在动态环境中执行导航任务（如TEB规划器）。

代价地图管理：基于激光雷达数据和里程计实时更新local\_costmap和global\_costmap，分别处理局部障碍避障与全局路径优化。

2、TEB（时间弹性带）算法原理：

将机器人规划问题描述为一条优化弹性路线。考虑时间和环境约束条件，生成动态响应的局部规划路径。

具有路径平滑化、动态障碍物规避、高效短路径优化的特点。

**实验步骤：**

**<1> 环境初始化**

|  |
| --- |
| $ roslaunch robot\_task ucar\_robot.launch |

**<2> 启动导航系统控制(新开一个终端输入下面命令)**

|  |
| --- |
| $ roslaunch navigation navigation.launch |

**<3> 设置目标点**

启动RViz，在RViz中，利用“2D Nav Goal”工具，设置目标点为B点。

**核心代码分析**

navigation.launch是导航系统的总体启动文件，负责加载各组件：

|  |
| --- |
| <launch>  <!-- 地图服务器 -->  <node name="map\_server" pkg="map\_server" type="map\_server" args="$(find navigation)/maps/room3.yaml"/>  <!-- 自适应蒙特卡洛粒子滤波器（AMCL） -->  <include file="$(find navigation)/launch/amcl.launch"></include>  <!-- move\_base导航 -->  <node pkg="move\_base" type="move\_base" respawn="false" name="move\_base" output="screen">  <rosparam file="$(find navigation)/param/costmap\_params.yaml" command="load" />  <rosparam file="$(find navigation)/param/move\_base\_params.yaml" command="load" />  <rosparam file="$(find navigation)/param/teb\_local\_planner\_params.yaml" command="load" />  <rosparam file="$(find navigation)/param/costmap\_converter\_params.yaml" command="load" />  </node>  <!-- RViz导航可视化 -->  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find navigation)/rviz/navigation.rviz"/>  </launch> |

AMCL算法:

负责加载定位算法（自适应蒙特卡洛算法），生成机器人在地图中的位姿估计：

|  |
| --- |
| <launch>  <node pkg="amcl" type="amcl" name="amcl">  <param name="min\_particles" value="1000"/>  <param name="max\_particles" value="10000"/>  <param name="gui\_publish\_rate" value="10.0"/>  <param name="odom\_frame\_id" value="odom"/>  <param name="global\_frame\_id" value="map"/> <!-- tf发布层次 -->  <param name="laser\_model\_type" value="beam"/>  <param name="recovery\_alpha\_slow" value="0.0"/>  <param name="recovery\_alpha\_fast" value="0.0"/>  </node>  </launch> |

代价地图配置：

静态层：加载地图

障碍层：利用激光雷达数据实时更新动态障碍物。

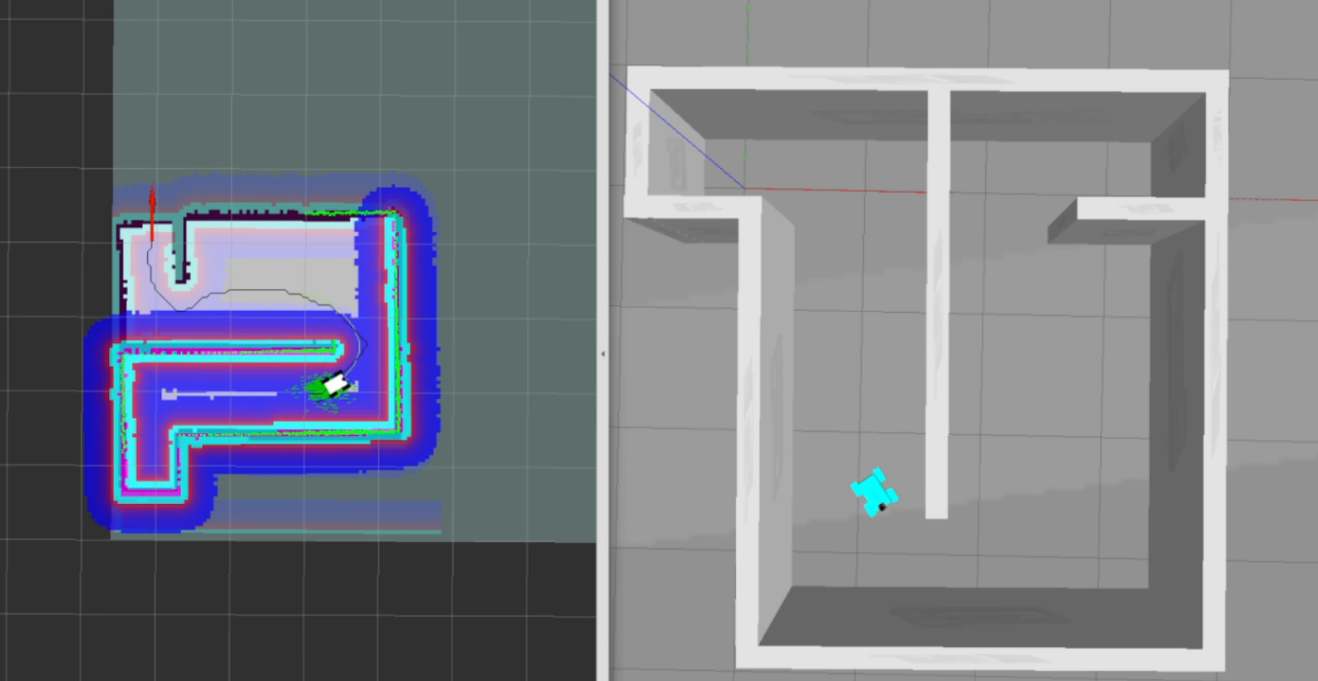
膨胀层：为障碍物扩展额外的安全空间。

|  |
| --- |
| global\_costmap:  global\_frame: map  robot\_base\_frame: base\_link  plugins:  - {name: static\_layer, type: "costmap\_2d::StaticLayer"}  - {name: obstacle\_layer, type: "costmap\_2d::ObstacleLayer"}  - {name: inflation\_layer, type: "costmap\_2d::InflationLayer"}  local\_costmap:  global\_frame: odom  robot\_base\_frame: base\_link  rolling\_window: true  plugins:  - {name: obstacle\_layer, type: "costmap\_2d::ObstacleLayer"}  - {name: inflation\_layer, type: "costmap\_2d::InflationLayer"} |

局部路径规划算法：TEB(Timed Elastic Band)负责生成局部路径并优化动态导航性能。

核心参数解析：

|  |
| --- |
| TebLocalPlannerROS:  max\_vel\_x: 1.0 # 最大线速度  max\_vel\_x\_backwards: 0.2 # 最大后退速度  max\_vel\_theta: 1.0 # 最大角速度  xy\_goal\_tolerance: 0.2 # x/y轴终点到目标容忍误差  yaw\_goal\_tolerance: 0.1 # 角度容忍误差  enable\_homotopy\_class\_planning: true # 是否启用异类规划路径 |



**实验四** 基于雷达数据的锥桶障碍物停车实验

**实验目的：**

通过激光雷达数据处理，编写ROS节点程序使得仿真小车在运行过程中识别特定雷达数据模式（锥桶障碍物的左右距离），并按照要求：

在双锥桶障碍物区域停车 3 秒。在单锥桶障碍物区域停车 5 秒。在最后一个锥桶处停止运动。

**实验内容：**

1、搭建仿真环境：

使用Gazebo启动task2.world仿真环境。

2、实现锥桶检测与停车策略：

通过解析激光雷达 /scan 数据，根据雷达距离与对应的角度判断障碍物分布。检测左右锥桶距离范围，判断是单锥桶还是双锥桶障碍物。在双锥桶障碍物处停车 3 秒，在单锥桶障碍物处停车 5 秒，最后一个锥桶完全停止运动。

3、使用ROS程序控制机器人：

使用 /cmd\_vel 控制机器人速度。根据雷达数据更新策略，动态调整前进或停止。

**实验仪器：**

Gazebo 仿真平台、ROS Noetic、仿真小车模型

**实验原理：**

1. 激光雷达数据格式解析

LaserScan消息格式：

angle\_min 和 angle\_max：激光雷达扫描的最小和最大角度（单位：弧度）。

angle\_increment：每个角度之间的增量（弧度）。

ranges：每个角度对应的物体距离（单位：米）。

通过以下步骤提取特定角度范围的物体距离：

将角度（左右锥桶分别为90°和270°）转换为弧度。

根据弧度计算ranges数组中的索引值。

提取并分析对应距离值，根据双锥桶和单锥桶的具体特征逻辑，判断停车策略。

2. 停车逻辑设计

双锥桶判断：左右障碍物距离均在(0.9, 1.2)范围内（验证两侧均有锥桶）。

单锥桶判断：

一侧距离为无穷大（inf），另一侧在(1.1, 1.2)范围内。

如左侧锥桶距离为 inf 且右侧距离处于限定范围。

开始停车：以3秒或5秒为停车时长后，忽略雷达检测一段时间，继续运动。

3. 运动控制

基于/cmd\_vel话题发布速度指令控制机器人：

停车时发布线速度=0指令。

向前运动时发布线速度=0.5指令。

**实验步骤：**

**<1> 环境初始化**

|  |
| --- |
| $ roslaunch robot\_task ucar\_robot.launch |

**<2> 运行停车监控程序(新开一个终端输入下面命令)**

|  |
| --- |
| $ rosrun robot\_task task4 |

**<3> 验证停车行为**

激光雷达检测到双锥桶（左右障碍物均距离小于 1.2m）时，停车 3 秒。

激光雷达检测到单锥桶（左侧或右侧距离为inf，另一侧为<1.2m）时，停车 5 秒。

在最后一个锥桶处，停车并停止导航。

**核心代码分析**

1. 激光雷达数据提取

函数：getRangeAtAngle() 从指定角度（度）转换到索引值，提取雷达距离：

输入：激光雷达消息、查询角度。

输出：指定角度的物体距离。

|  |
| --- |
| double getRangeAtAngle(const sensor\_msgs::LaserScan& scan\_msg, double angle\_deg) {  double angle\_rad = angle\_deg \* M\_PI / 180.0; // 将角度转换为弧度  if (angle\_rad < scan\_msg.angle\_min || angle\_rad > scan\_msg.angle\_max) {  return std::numeric\_limits<double>::infinity(); // 返回无穷大  }  int index = static\_cast<int>((angle\_rad - scan\_msg.angle\_min) / scan\_msg.angle\_increment);  index = std::max(0, std::min(index, static\_cast<int>(scan\_msg.ranges.size()) - 1));  return scan\_msg.ranges[index];  } |

2. 雷达数据停车逻辑

函数：run() 解析激光数据，判断停车条件并调整小车状态：

左右两侧锥桶的距离通过getRangeAtAngle()分别在90°和270°角获取。

停车条件解析：

双锥桶障碍：左、右均在 (0.9, 1.2m)。

单锥桶障碍：一侧距离为inf，另一侧在 (1.1, 1.2m)。

|  |
| --- |
| if ((0.9 < left\_dist && left\_dist < stop\_threshold\_) &&  (0.9 < right\_dist && right\_dist < stop\_threshold\_)) {  ROS\_INFO("双锥桶检测，停车3秒 (左=%.2f, 右=%.2f)", left\_dist, right\_dist);  doStopAndIgnore(3.0, 2.0);  } else if (std::isinf(left\_dist) && (1.1 < right\_dist && right\_dist < stop\_threshold\_)) {  ROS\_INFO("单锥桶右侧检测，停车5秒 (左=inf, 右=%.2f)", right\_dist);  doStopAndIgnore(5.0, 2.0);  } else {  geometry\_msgs::Twist cmd\_vel;  cmd\_vel.linear.x = forward\_speed\_;  cmd\_vel.angular.z = 0.0;  cmd\_vel\_pub\_.publish(cmd\_vel);  } |

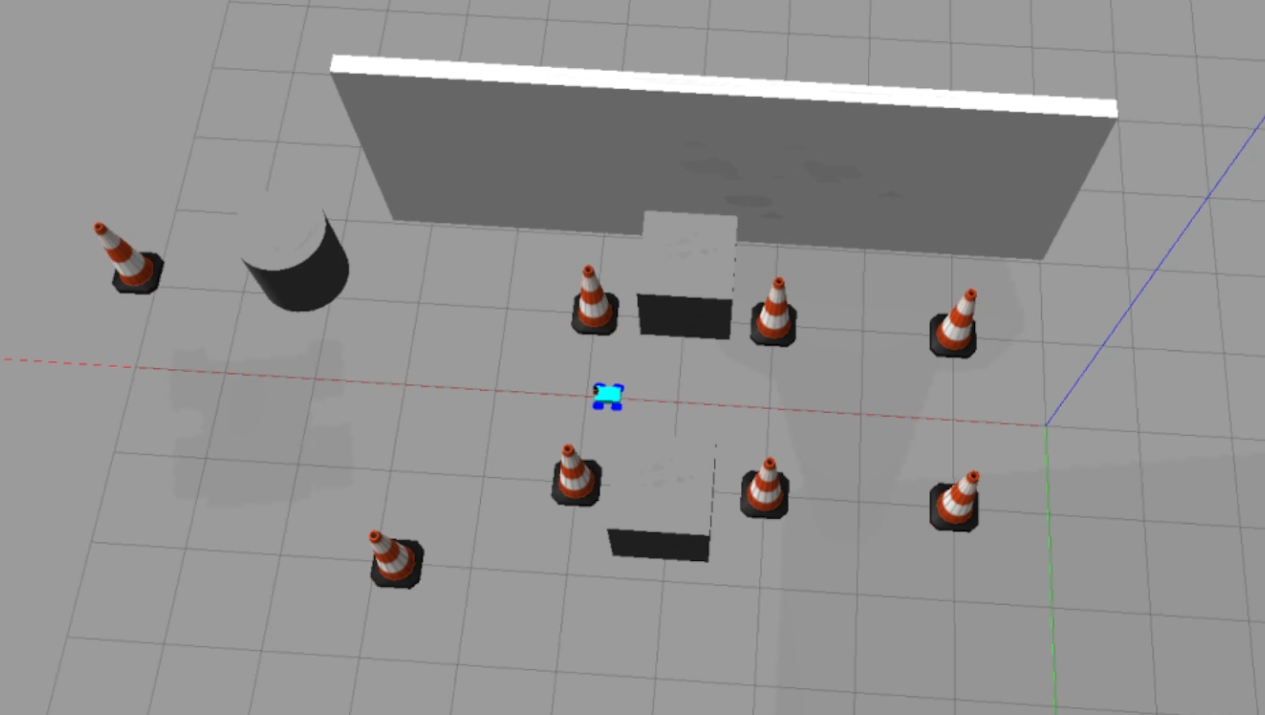
3. 停车与恢复

函数：doStopAndIgnore() 实现停车与恢复运动：

停车时发布线速度为0的/cmd\_vel消息。

停车后允许忽略激光雷达临时干扰（ignore\_duration）。

|  |
| --- |
| void doStopAndIgnore(double stop\_duration, double ignore\_time) {  geometry\_msgs::Twist stop\_cmd;  stop\_cmd.linear.x = 0.0;  stop\_cmd.angular.z = 0.0;  cmd\_vel\_pub\_.publish(stop\_cmd);  ros::Duration(stop\_duration).sleep();  ROS\_INFO("忽略雷达%.1f秒，恢复运动...", ignore\_time);  ignoring\_ = true;  ignore\_end\_time\_ = ros::Time::now() + ros::Duration(ignore\_time);  } |



**实验五** 基于move\_base框架的多点导航实验

**实验目的：**

熟悉 ROS move\_base 框架与 Action Server 工作机制。

通过多点导航程序，实现机器人在不同点位的精准到位巡航控制。

掌握自主巡航程序的编程和调试方法，了解移动机器人路径规划规划与控制原理。

**实验内容：**

1、搭建Gazebo仿真环境（task3.world场景），加载巡检小车。

2、编写代码实现基于MoveBase框架的多点导航程序，让小车按顺序完成点位1->2->3->4->5->6->7的巡航。

每次到达点位后记录日志，提示是否到达目标点。

借助ROS的Action机制处理多点巡航。

3、小车循环执行多点导航，并记录巡航路径运行日志。

**实验仪器：**

Gazebo 仿真平台、ROS Noetic、仿真小车模型

**实验原理：**

1. Move\_base 框架与导航架构

move\_base 是 ROS 中用于导航的核心组件，提供路径规划与执行的功能。其架构：

全局路径规划器：基于已知全局地图，计算起点到目标点的整体路径。

局部路径规划器：动态调整路径，避开障碍物确保本地避障。

Action Server: 接收目标点（Pose），提供任务执行的功能。

2. 多点导航工作流程

多点导航任务的核心：

按目标点顺序，向 move\_base发送导航目标点。

move\_base 接收目标点请求，利用路径规划器计算路径并驱动小车运动。

为实现循环巡航，需要依次发送所有目标点目标。

3. TF坐标系与目标点定义

通过RViz查看各目标点的位姿：

位置表示为(x, y, z)，四元数 (qx, qy, qz, qw) 。

**实验步骤：**

**<1> 环境初始化**

|  |
| --- |
| $ roslaunch robot\_task ucar\_robot.launch |

**<2> 启动导航系统控制(新开一个终端输入下面命令)**

|  |
| --- |
| $ roslaunch navigation navigation.launch |

**<3> 运行巡航代码实现机器人多点导航，完成点位依次巡航(新开一个终端输入下面命令)**

|  |
| --- |
| $ rosrun robot\_task task5 |

**核心代码分析**

1. 添加目标点：addGoal()

目标点定义：

ROS的导航目标点由位置和朝向两个部分组成。

通过函数传递 (x, y, z, ox, oy, oz, ow) 参数，定义目标点的坐标与朝向。

队列管理：

所有目标点存储于 std::vector<goals\_> 队列中，方便在巡航过程中按顺序处理。

|  |
| --- |
| void addGoal(double x, double y, double z, double ox, double oy, double oz, double ow) {  move\_base\_msgs::MoveBaseGoal goal;    goal.target\_pose.header.frame\_id = "map"; // 使用全局地图坐标系  goal.target\_pose.header.stamp = ros::Time::now(); // 时间戳  goal.target\_pose.pose.position.x = x; // 目标点 x 坐标  goal.target\_pose.pose.position.y = y; // 目标点 y 坐标  goal.target\_pose.pose.position.z = z; // 目标点 z 坐标  goal.target\_pose.pose.orientation.x = ox; // 四元数的 x  goal.target\_pose.pose.orientation.y = oy; // 四元数的 y  goal.target\_pose.pose.orientation.z = oz; // 四元数的 z  goal.target\_pose.pose.orientation.w = ow; // 四元数的 w    goals\_.push\_back(goal); // 将目标添加至目标点队列  } |

2. 目标点发送与目标达成：sendGoal()

目标点发送：

通过 ac\_.sendGoal(goal)，将目标点发送至 move\_base 的 Action Server。

ROS 的 move\_base 开始规划路径，通过路径规划器与运动控制模块驱动机器人向目标点移动。

等待结果：

ac\_.waitForResult()：阻塞当前程序线程，直到目标点任务完成或超时。

|  |
| --- |
| if ((0.9 < left\_dist && left\_dist < stop\_threshold\_) &&  (0.9 < right\_dist && right\_dist < stop\_threshold\_)) {  ROS\_INFO("双锥桶检测，停车3秒 (左=%.2f, 右=%.2f)", left\_dist, right\_dist);  doStopAndIgnore(3.0, 2.0);  } else if (std::isinf(left\_dist) && (1.1 < right\_dist && right\_dist < stop\_threshold\_)) {  ROS\_INFO("单锥桶右侧检测，停车5秒 (左=inf, 右=%.2f)", right\_dist);  doStopAndIgnore(5.0, 2.0);  } else {  geometry\_msgs::Twist cmd\_vel;  cmd\_vel.linear.x = forward\_speed\_;  cmd\_vel.angular.z = 0.0;  cmd\_vel\_pub\_.publish(cmd\_vel);  } |

3. 巡航流程控制：run()

主线程循环：

使用 while (ros::ok()) 作为主程序循环，保证程序在 ROS 运行状态下持续执行。每次完成所有 goals\_ 中的目标点任务后，重新开始下一次巡逻。

多点巡逻：

循环目标点队列 goals\_，调用 sendGoal() 将每个目标点逐一发送至 move\_base，等待任务完成。

|  |
| --- |
| void run() {  ros::Rate loop\_rate(1); // 设置循环频率  while (ros::ok()) { // 主循环  for (size\_t i = 0; i < goals\_.size(); ++i) {  sendGoal(goals\_[i]); // 逐点发送巡航目标  }  ROS\_INFO("完成一轮巡航，重新开始...");  loop\_rate.sleep(); // 每次巡逻间隔  }  } |

