全国大学生智能汽车竞赛
讯飞-智慧餐厅
技
术
报
告

参赛学校 北京邮电大学

队伍名称 SolveALL

领队老师 戴志涛

指导老师 戴志涛

参赛学生 司马宽宽、艾诺舟、刘柏川、李孟非

二零二一年

关于技术报告使用授权的说明

本人完全了解 2021 全国大学生智能汽车竞赛创意组分赛关于保留、使用技术报告的规定，即参赛作品著作权归参赛者本人所有，比赛组委会和科大讯飞（苏州）科技有限公司可以在相关主页上收录并公开参赛作品的技术报告以及参赛作品的视频、图像资料，并将相关内容进行编纂收录。

参赛队员签名：司马宽宽 艾浩舟 刘柏川 李亚非

指导老师签名：

日　　期：2021.3.20

目录

第 1 章 项目背景及意义

1.1 背景

全国大学生智能汽车竞赛由教育部高等学校自动化专业教学指导分委员会主办，旨在培养创新精神、动手能力、将科学理论运用到实际的，是以智能汽车为研究对象，面向全国大学生的具有探索性工程实践的创意科技竞赛。

1.2 项目意义

随着科学技术和互联网快速发展，智能科技应用广泛，其中智能移动服务机器人受到了人们的广泛关注，它所运用的无接触传播在疫情的情况下得到了大规模的推广，该竞赛以迅猛发展、前景广阔的汽车电子为背景，倡导求真务实的学风、脚踏实地的学习精神、勇于创新的动手能力、迎难而上的钻研精神，为优秀人才的培养奠定基础，从而进一步推动智能机器人行业的发展。

同时，当今机器人技术的发展日新月异，其应用于考古，探测，国防等众多领域。无人飞船，外星探测，智能化生产等等无不得益于机器人技术的发展。从某种意义上来说，机器人技术反映的是一个国家综合技术实力的高低，而智能小车是机器人的雏形，它的控制系统的研究与制作将有助于推动智能机器人控制系统的发展。

1.3 所要解决的问题

我们需要在线上仿真比赛 Gazebo 平台下，采用 ROS 构建比赛地图，通过自主导航算法实现无车人从起点到终点的比赛。考虑到比赛地图中的直道、弯道、障碍物的存在，在车模运行的过程中，我们需要避开在直道和弯道上的场地边框碰撞，避开随机位置的障碍物的碰撞，在较短的时间内到达终点结束比赛。

1.4 创新点

我们团队将仿真车的运动采用由局部规划器自动完成的分段限制速度的方式，在直道下仿真车以较快的速度运行，在弯道上以较快的加速度运行以实现在不容易发生碰撞道路速度的最大化，同时确保在狭窄道路下的驶出能够迅速加速，旨在保障不发生碰撞的情况下尽可能的提高速度。

第 2 章 项目方案设计

2.1 方案总体架构

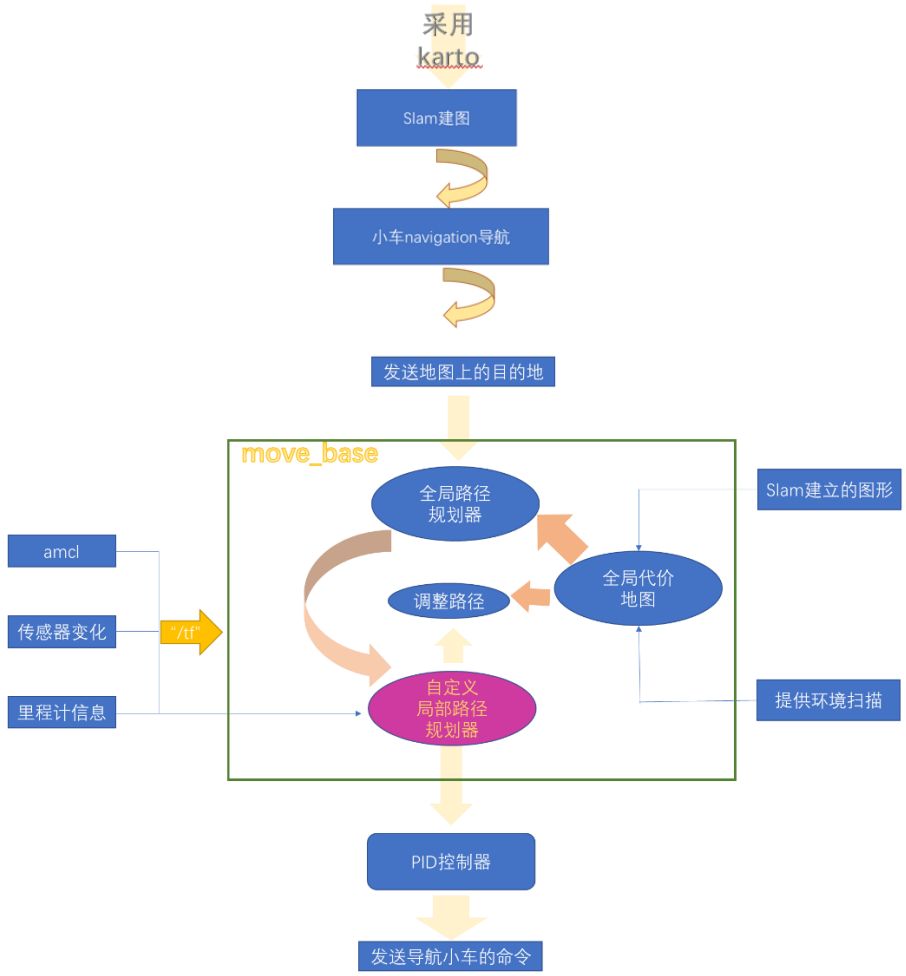


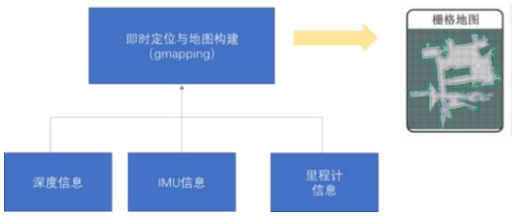
图 2-1　方案架构图

2.2 建图方案

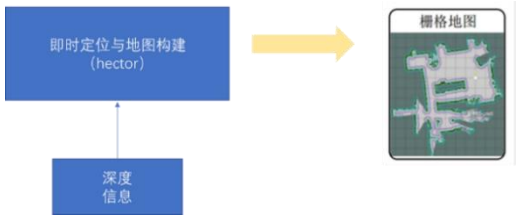
导航功能需要运行在已知静态地图上，地图的精度和准确度也将对导航产生直接影响。目前较为著名的激光雷达建图算法有 gmapping、hector、karto、cartography 等，其基本原理如图 2-2 所示，其中 karto 与 cartography 算法采取的都是图优化框架，但二者采取的优化库不一样， karto 采取的是 spa(karto_slam)或 g2o(nav2d), cartographer 采取的是 google 的 ceres 构建 problem 优化；karto 的前端与后端采取的是单线程进行，cartographer 采取的是 4 线程后端优化。

ROS 已经为这些算法提供了功能包，因此实验成本较低，我们对这四种算法

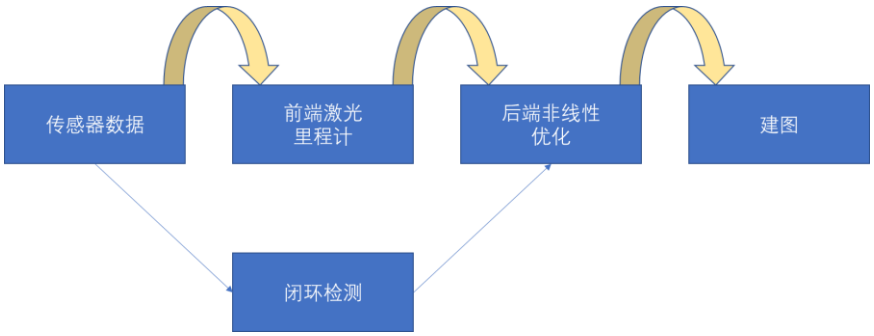
分别进行了测试，最终发现 karto 算法和 cartography 算法的建图效果最佳，但由于 cartography 算法需要调节阈值参数，实际应用时存在一些问题，故最终选择 karto 算法进行 SLAM 建图。



（a）gmapping 建图原理



（b）hector 建图原理



（c）Cartographer 建图原理

图 2-2 典型的 SLAM 建图算法原理

2.3 导航方案

2.3.1 概述

ROS 已经为导航功能开发了专用的功能包，用于实现移动机器人可靠移动，通过处理里程数据、传感器数据和环境地图数据，为机器人运动生成一条安全的路径。[1] 从概念层面上讲，导航功能包集是相当简单的。它从里程计和传感器数据流获取信息，并将速度命令发送给移动基站（比如你的机器人）。但是，想要在任意机器人上使用导航功能包集可能有点复杂。

使用导航功能包集的先决条件是，机器人必须运行 ROS，有一个 tf 变换树，使用正确的 ROS Message types 发布传感器数据。而且，我们需要在高层为一个具有一定形状和动力学特点的机器人配置导航功能包集。图 2-3 展示了运行 navigation 功能包所需的组件，白色的部分是必须且已实现的组件，灰色的部分是可选且已实现的组件，蓝色的部分是必须为每一个机器人平台创建的组件。

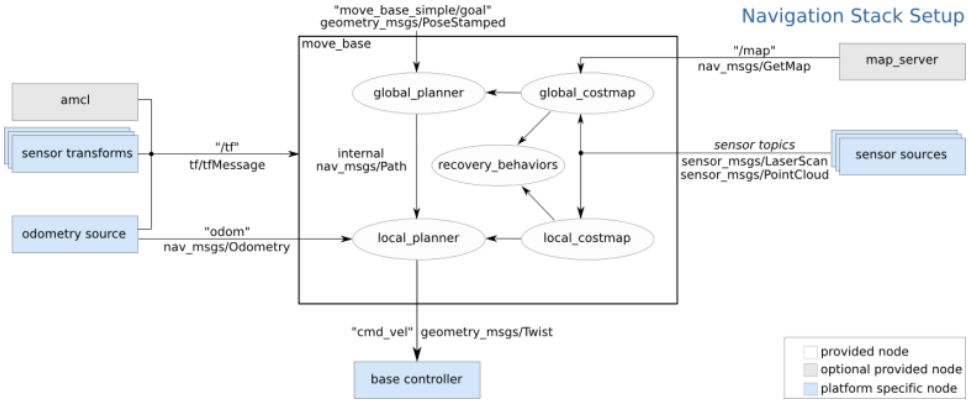


图 2-3 ROS Navigation 功能包结构图

2.3.2 定位

通过 SLAM 得到静态地图后车模需要知道自己在地图中的坐标，从而得出路径规划，我们采用 move_base 功能包中自带的 AMCL 算法作为定位算法。

配置 AMCL 包的相关参数以适应车模激光雷达的特性，即可实现在比赛地图中的高精度定位。

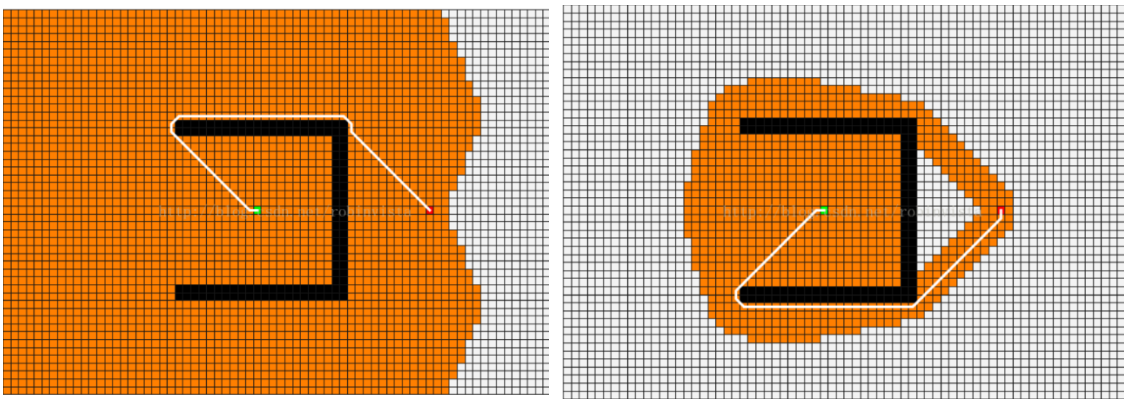
如图 2-3 所示，小车的路径与运动规划需要分为两个模块，即全局规划(global_planner)与局部规划(local_planner)。

2.3.3 全局规划

全局规划需要在地图上预先规划出一条路线，且需要知道小车的当前位置。我们采用 GlobalPlanner 规划器通过 A*最优路径的算法，计算 costmap 上的最小花费路径，作为机器人的全局路线。

A*算法的原理是它能够选择下一个被检查的节点时引入了已知的全局信息，对当前结点距离终点的距离作出估计，作为评价该节点处于最优路线上的可能性的量度，这样可以首先搜索可能性大的节点，从而提高了搜索过程的效率。

A*算法计算当前点与目标点和起始点的距离和，然后找出下一步如何走。Dijkstra 算法找出所有可能的路径，最后找出消耗最少的一条路径。A*算法使用了启发式信息（到目标的距离），而 Dijkstra 算法没有使用启发式信息，相比较而言 A*算法规划所需要的资源更少，效率更高。所以我们选择 A*算法作为全局路径规划。



(a)Dijkstra 算法　　(b)A*算法

图 2-4　Dijkstra 算法与 A*算法的路径规划效果对比

2.3.4 局部规划

环境发生变化，如出现未知障碍物时，全局规划就无能为力了。它是一种事前规划，因此对机器人系统的实时计算能力要求不高，虽然规划结果是全局的、较优的，但是对环境模型的错误及噪声鲁棒性差。而局部路径规划则是在环境信息完全未知或有部分可知的情况下，侧重于考虑机器人当前的局部环境信息，让机器人具有良好的避障能力，通过传感器对机器人的工作环境进行探测，以获取障碍物的位置和几何性质等信息，这种规划需要搜集环境数据，并且对该环境模型的动态更新能够随时进行校正，局部规划方法将对环境的建模与搜索融为一体，要求机器人系统具有高速的信息处理能力和计算能力，对环境误差和噪声有较高的鲁棒性，能对规划结果进行实时反馈和校正。

我们团队采用 teb_local_planner 进行局部路径规划。TEB 算法通过求解稀疏的标量化多目标优化问题，可以有效地获得最优轨迹。用户可以为优化问题提供权重，以便在目标冲突的情况下指定行为，实现了在线最佳本地轨迹规划器，它作为 ROS 导航软件包的插件来导航和控制移动机器人。由全局计划程序生成的初始轨迹在运行时实现时间最优目标，与障碍物的分离以及符合运动动力学约束（例如满足最大速度和加速度）的要求。

DWA 的原理主要是在速度空间（v,w）中采样多组速度，并模拟这些速度在一定时间内的运动轨迹，再通过一个评价函数对这些轨迹打分，最优的速度被选择出来发送给下位机。

DWA 的优势在于反应速度较快,计算不复杂,通过速度组合(线速度与角速度)可以快速得出下一时刻规划轨迹的最优解。可以将优化由横向与纵向两个维度向一个维度优化。缺点在于，此算法主要体现在较高的灵活性和其静态环境,而较高的灵活性会极大的降低行驶的平稳性。而小车的惯性较大，再加上可调参数比较

少，不能较好地保障小车行驶过程中的顺滑平稳。

而 TEB 的创新之处在于利用时间信息增强经典的弹性带。因此，不仅可以考虑路径的几何约束和运动学约束，还可以同时考虑移动机器人的动态约束。g2o 为稀疏系统结构提供了算法和求解器。TEB 能够解决动态障碍和运动约束的问题，将“TEB”表述为一个多目标优化问题，将约束条件描述成超图的形式，然后用图优化，能够更直观地解决小车避障的问题，再加上 TEB 可调参数较多，可以更好地多角度的去调节小车的局部路径，使得轨迹更加平滑。

全局路径规划和局部路径规划并没有本质上的区别，很多适用于全局路径规划的方法经过改进也可以用于局部路径规划，而适用于局部路径规划的方法同样经过改进后也可适用于全局路径规划。两者协同工作，机器人可更好的规划从起始点到终点的行走路径。

2.3.5 总结

我们所采用的导航方案可以概括为：使用 AMCL 算法实现高度定位；使用 move_base 默认的全局规划器 NavFn 实现全局路径规划，并设置使用 A*算法；使用 dwa_local_planner 和 teb_local_planner 作为局部规划器来实现更好的避障与运动控制。

2.4 控制方案

2.4.1 单独使用局部规划器

在练习阶段中，组委会提供的车模物理属性较为友好，对速度及方向的控制基本没有难度，直接使用 teb_local_planner 发布期望速度命令小车的差分控制器便能迅速调整到目标状态。使用初代车模时只需要略微调整机器人速度规划参数，如 max_vel_x、max_vel_theta、acc_lim_theta 等即可跑出不错的成绩。

2.4.2 动态速度规划

进一步调试局部规划器后我们发现其规划的速度经常过大或过小，导致撞墙和转向过慢的问题。对此，我们决定采用动态速度规划的策略，即在地图中规定若干个关键点。由实验数据得到这些关键点之间的最佳运动参数，小车经过这些关键点后自动改变 max_vel_x、max_vel_theta 等参数为最佳值，实现对局部规划器期望速度参数的强制限制。经过实验，动态速度规划的控制方案优化效果十分

明显，能够将比赛用时从 28~30s 提升至 21~23s。

2.4.3 自定义规划器

为了提升比赛难度，组委会发布了“赛事模型包 v3.5”。经过我们的分析，发现该版本车模的惯性矩相较此前增大了 10 倍，如图 2-6 所示。虽然改动处相比整个 urdf 文件而言几乎微不足道，但其对车模的运行状态造成了巨大影响。使用相同的参数运行新版赛事包车模，我们的主观感受是“车变重了”、“更难控制了”，具体表现为在入弯时刹不住、“推头”或过度转向，出弯时无法及时加速，轨迹跟踪效果差。

```
<inertial>
        <mass value="${mass}" />
        <inertia ixx="0.01" ixy="0.0"
                 ixz="0.0"
                 iyy="0.01" iyz="0.0"
                 izz="0.01" />
</inertial>
```

(a)urdf 文件修改前参数

```
<inertial>
        <mass value="${mass}" />
        <inertia ixx="0.1" ixy="0.0"
                 ixz="0.0"
                 iyy="0.1" iyz="0.0"
                 izz="0.1" />
</inertial>
```

(b)urdf 文件修改后参数

图 2-6 urdf 文件变化对比

我们尝试继续通过调整 teb_local_planner 的参数优化寻迹，代价是 max_vel_x 和 max_vel_theta 都必须降到足够低才能保证小车顺利通过弯道，最终调参得到的极限比赛时间为 27~30s。这种结果显然无法取得较好名次，因此，我们必须另辟蹊径。

经过分析，我们发现 teb_local_planner 之所以无法对小车实现良好的控制，核心原因在于整个控制流程中缺少“反馈”。由于 move_base 插件通过 cmd_vel 话题发布速度指令并被 gazebo 的差分驱动插件接收，因此 cmd_vel 中的速度命令相当于是“理想命令”。即假如 cmd_vel 中此时包含的信息为”linear.x=1.5,angular.z=2.0”，则这条命令只是希望小车满足的速度，而由于小车的惯性矩增大，其速度并不会立即到达期望值，而是有一个加减速过程，宏观上表现为小车的状态总是“慢半拍”。由经典控制理论，此时的控制方式属于开环控制，如图 2-7 所示。

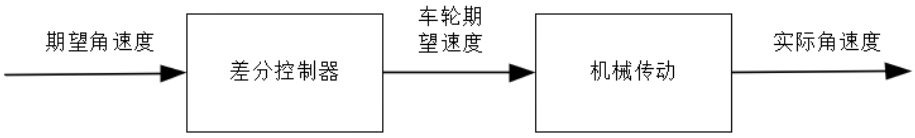


图 2-7 角速度开环控制系统方框图

对于这种情况，我们可以加入反馈回路，将小车此时的速度与位姿信息接入控制流程的输入端，构成闭环控制回路。当小车此时的实际位姿与期望位姿偏差较大时，可以给定较大控制量，而在接近期望位姿时再减小控制量，实现以偏差纠正偏差的负反馈控制过程，结构图如图 2-8 所示。

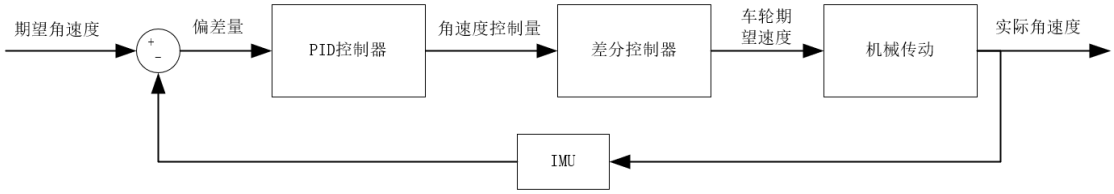


图 2-8 带有反馈回路的角速度控制系统方框图

在得到小车实际位姿的反馈值后，我们使用 PID 算法对其进行控制，通过调整 Kp、Ki、Kd 等参数让系统输出最优控制量。

经过权衡后我们决定不修改 teb_local_planner 的源代码，而是用 Python 手写一个全新的控制器，否则需要详细了解 teb_local_planner 的代码结构，在调试 c++代码的过程中也可能出现很多问题。

有关自定义规划器原理的详细论述见 3.2 节。

2.4.4 方案总结

为了解决 move_base 默认局部规划器的缺点与新版车模惯性矩增大的问题，我们自己开发了新的算法，虽然并没有从导航包的底层出发进行修改，但这种“缝合怪”式的解决方案在一定程度上确实起到了应有的效果。

第 3 章 项目算法功能

3.1 AMCL 算法原理

3.1.1AMCL 原理流程

（1）粒子滤波和蒙特卡洛

图 3-1 AMCL 流程图

蒙特卡洛分析是一种统计模拟方法,主要用于电子技术 EDA 上。它是在给定电路元器件参数容差的统计分布规律的情况下,用一组组伪随机数求得电路元器件参数的随机抽样序列,对这些随机抽样序列进行直流、交流小信号和瞬态分析,并通过多次分析结果估算出电路性能的统计分布规律。

粒子滤波的思想基于蒙特卡洛方法，它是利用粒子集来表示概率，可以用在任何形式的状态空间模型上。其核心思想是通过从后验概率中抽取的随机状态粒子来表达其分布，是一种顺序重要性采样法。简单来说，粒子滤波法是指通过寻找一组在状态空间传播的随机样本对概率密度函数进行近似，以样本均值代替积分运算，从而获得状态最小方差分布的过程。

（2）重要性采样

重要性采样算法就是在有限的采样次数内，尽量让采样点覆盖对积分贡献很大的点。

要估计的期望值可改写为：

$$\mathrm{E[X;P]} = \int_{\mathrm{a}=-\infty}^{+\infty} aP(X \in [a; a+da]) \ (1)$$

注意到，更优（即让估计值方差更小）的 P 会使得样本分布的频率与其在 **E**[*X;P*]计算中的权重更加相关。

（3）自适应蒙特卡洛

自适应体现在：

(a)解决了机器人绑架问题，它会在发现粒子们的平均分数突然降低了（意

味着正确的粒子在某次迭代中被抛弃了）的时候，在全局再重新的撒一些粒子。

(b)解决了粒子数固定的问题，因为有时候当机器人定位差不多得到了的时候，比如这些粒子都集中在一块了，还要维持这么多的粒子没必要，这个时候粒子数可以少一点了。

（4）KLD 采样

为了控制上述粒子数冗余，在栅格地图中，根据粒子在栅格的占比确定粒子的分散程度。在每次迭代重采样的时候，增加允许粒子数量。选取粒子占比较少的数值，说明粒子已经集中，就将上限设低，采样到该数值。

3.1.2 订阅主题与发布主题

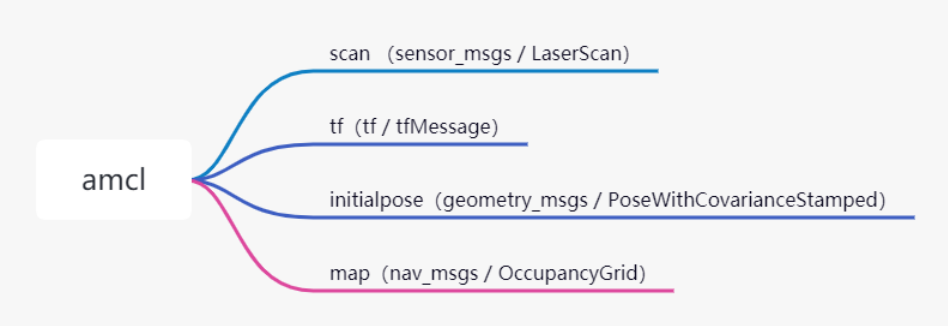


图 3-2 订阅主题

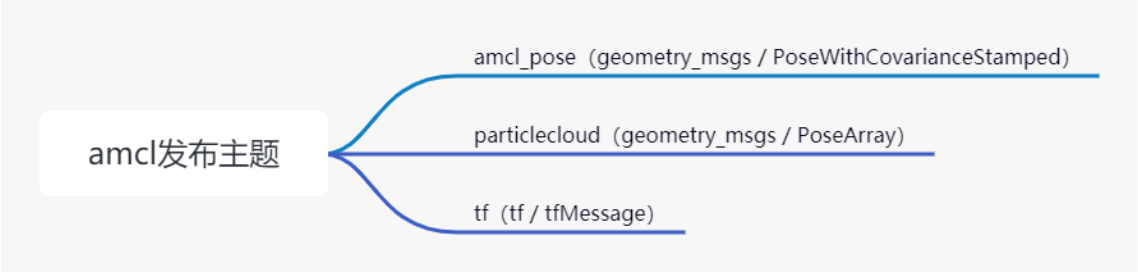


图 3-3 发布主题

3.1.3 转换

amcl 将输入的 laser scans 转换为 odometry 帧（/odom_frame_id）。因此，必须存在从 tf 树的路径，其中将激光扫描从帧发布到里程表帧。实现细节：在接收到第一激光扫描时，amcl 查找激光器的帧和基本帧（/base_frame_id）之间的变换，并且将其永久锁存。因此 amcl 不能处理相对于基座移动的激光器。

下图显示了使用 odometry 和 amcl 的本地化之间的差异。在操作期间，amcl 估计基本帧（/base_frame_id）相对于全局帧（/global_frame_id）的变换，但是它仅公开全局帧和里程表帧（/odom_frame_id）之间的变换。本质上，这种变换解释了使用航位推算发生的漂移。发布的变换是未来的日期。

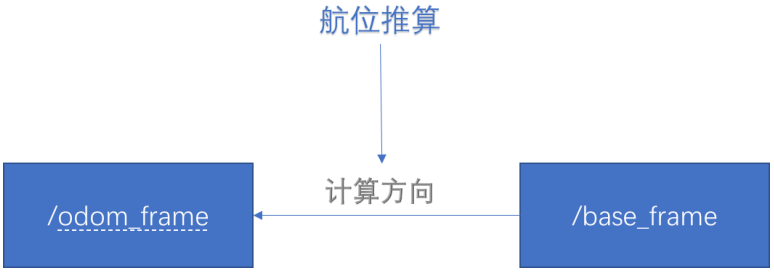


图 3-4 里程计本地化

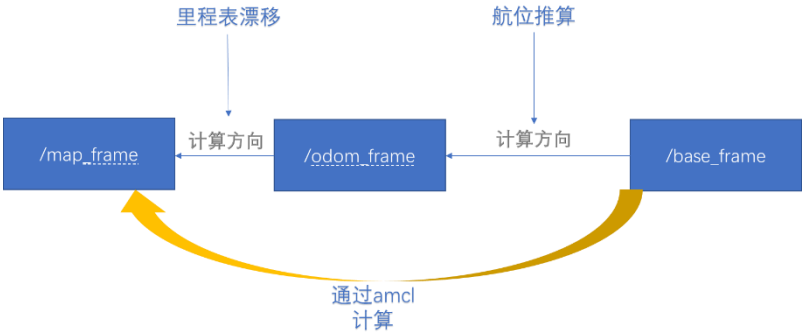


图 3-5　amcl 地图本地化

3.2 Karto SLAM 建图算法

3.2.1 算法概述

建图方面我们使用 Karto 算法，该算法基于位姿图（Pose Graph）优化的思想。有关 Karto 的详细论述见原文 *Efficient sparse pose adjustment for 2D mapping* [2]，本节只对该算法的核心内容进行阐述。

位姿图是一组机器人姿态，这些姿势由非线性约束连接，这些约束从对附近姿势共有的特征的观察中获得。

优化大型位姿图一直是移动机器人的瓶颈，因为直接非线性优化的计算时间会随着图的大小三次增长。 在 Karto 算法中提出了一种构造和求解线性子问题的有效方法，作者称为稀疏姿态调整（Spare Pose Adjustment，SPA）。

3.2.2 稀疏姿态调整(SPA)

为了优化一组姿势和约束，Karto 使用已知的 Levenberg-Marquardt（LM）方法为框架，其特定的实现方式使其对于 2D 地图构建中遇到的稀疏系统非常有效。 与计算机视觉的稀疏捆绑调整类似，这是相机和功能部件的 LM 的类似有效实现，称为系统稀疏姿势调整（SPA）。

Karto 中假定系统变量是机器人的全局姿态c的集合，以平移和角度为参数: $c_i = [t_i, \theta_i] = [x_i, y_i, \theta_i]^T$。约束（constraint）是从另一个（$c_i$）位置对一个节点$c_j$的测量。在$c_i$的帧中，$c_i$和$c_j$之间测得的偏移量是 $\bar{z}_{ij}$。对于c_i和c_j的任何实际姿势，可以将它们的偏移量计算为

$$h(c_i, c_j) \equiv \begin{cases} R_i^T(t_j - t_i) \\ \theta_j - \theta_i \end{cases} \quad (2)$$

其中R_i是θ_i的 2×2 旋转矩阵。$h(c_i, c_j)$被称为测量方程（measurement equation）。与约束相关的误差函数以及总误差为：

$$\begin{aligned} e_{ij} &\equiv \bar{z}_{ij} - h(c_i, c_j) \\ \chi^2(\mathbb{c}, \mathbb{p}) &\equiv \sum_{ij} e_{ij}^T \Lambda_{ij} e_{ij} \end{aligned} \quad (3)$$

有关线性系统、误差雅可比矩阵及稀疏性的讨论参见原文，这里不做展开。Karto 对系数矩阵的存储使用了 CCS 数据格式（compressed column storage），建立稀疏矩阵**H**的伪代码如代码段 3-1 所示。

代码段 3-1　创建稀疏矩阵**H**的伪代码

`H=CreateSpare(e,cf)`

输入：约束e_{ij}的集合，一系列空节点

输出：使用 CCS 数据格式的上三角稀疏矩阵 H

1) 初始化一个 c_f 大小的 C++ std::map 类型向量；每个 map 都与 H 的列容量匹配。map 的键是行索引，数据是空的 3×3 矩阵。令 map[i,j]代表 map 的第 i 行第 j 列入口。

2) 对于每个约束e_{ij}，假定$i < j$：

在接下来的步骤中，如果 map 的入口不存在就创建。

如果c_i为空，map[i,i]+=$J_i^T \Lambda_{ij} J_i$。

如果c_j为空，map[j,j]+=$J_j^T \Lambda_{ij} J_i$。

如果c_i, c_j都为空，map[j,i]+=$J_j^T \Lambda_{ij} J_i$

3) 设置上三角稀疏矩阵 H

a) 在下列步骤中，忽略 3x3 映射[i, i]中的元素对角线以下的元素

b) 按列依次遍历 map[i,i]，然后按行依次遍历列和行通过确定的数量每个列中的元素及其行位置。

c) 按列然后按行顺序再次遍历 map[]按顺序将项插入到 val 中。

3.3 导航算法

3.3.1 A*算法

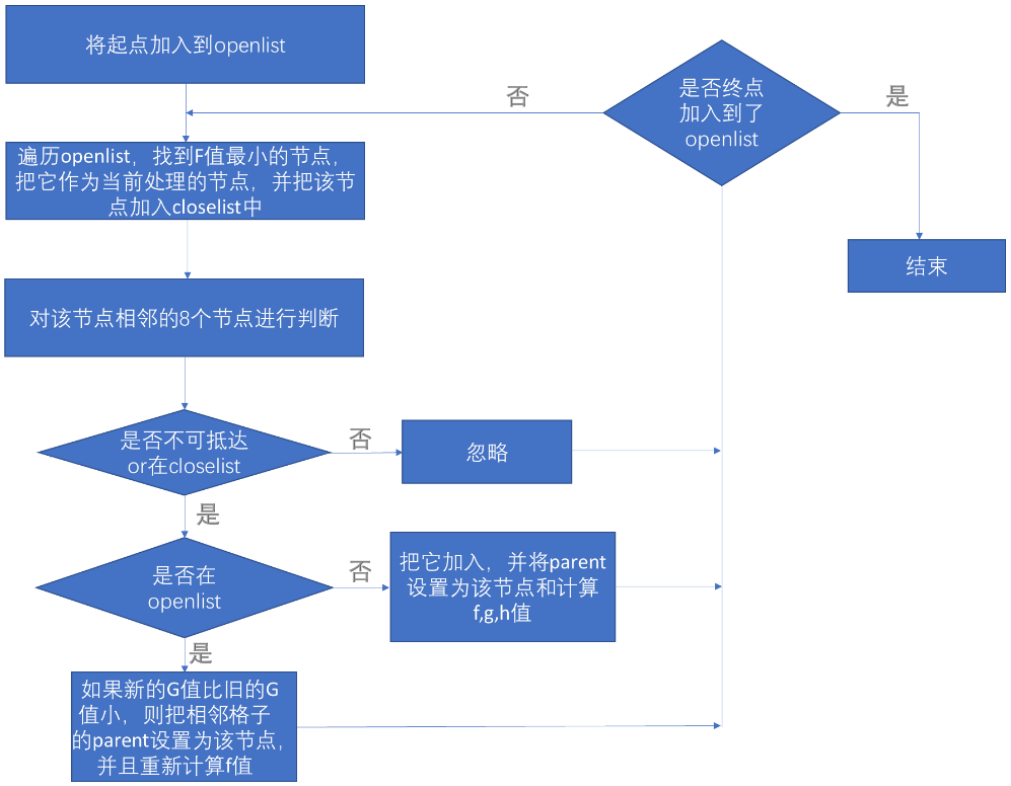


图 3-6 A*算法流程图

1. 算法概述

A*算法 A*(A Star）算法是一种启发式（heuristic）算法，他是静态路网中求解最短路最有效的方法。A*算法的代价公式表示为

$$f(n) = g(n) + h(n) \quad （4）$$

其中，f(n) 是节点 n 从初始点到目标点的估价函数，g(n) 是在状态空间中从初始节点到 n 节点的实际代价，即起始节点到当前节点的实际代价。h(n)是从 n 到目标节点最佳路径的估计代价，即当前节点到目标节点的估计代价，称为启发函数。

2. 算法伪代码

代码段 3-2　A*算法伪代码

```
1   {
2     Open = [起始节点];
3     Closed = [];
```

```
4     while (Open 表非空)
5     {
6     从 Open 中取得一个节点 X，并从 OPEN 表中删除。
7       if (X 是目标节点)
8       {
9         求得路径 PATH;
10        返回路径 PATH;
11      }
12      for (每一个 X 的子节点 Y)
13      {
14        if (Y 不在 OPEN 表和 CLOSE 表中)
15        {
16           求 Y 的估价值;
17           并将 Y 插入 OPEN 表中;
18        }
19        //还没有排序
20        else if (Y 在 OPEN 表中)
21        {
22          if (Y 的估价值小于 OPEN 表的估价值)
23          更新 OPEN 表中的估价值;
24        }
25        else //Y 在 CLOSE 表中
26        {
27           if (Y 的估价值小于 CLOSE 表的估价值)
28           {
29             更新 CLOSE 表中的估价值;
30             从 CLOSE 表中移出节点，并放入 OPEN 表中;
31           }
32        }
33      将 X 节点插入 CLOSE 表中;
34      按照估价值将 OPEN 表中的节点排序;
35      }//end for
36    }   //end while
37  }
```

3.3.2 TEB 算法

由于 ROS 已经为 TEB 算法封装好了功能包，我们并不需要深入研究其原理，本节仅对论文 *Trajectory modification considering dynamic constraints of autonomous robots*[3]中的相关内容作简要解释。

3.3.2.1 Timed Elastic Band 概述

定义机器人位姿为$x_i = [x_i, y_i, \beta_i]^T \in \mathbb{R} \times S^1$，其中$x_i, y_i$是机器人位置，$\beta_i$被定义为全局中机器人的方向，论文原文中称其为 configuration。一段空间内的 configuration 序列可以记为

$$Q = \{x_i\}_{i=0\ldots n} \quad n \in \mathbb{N} \quad (5)$$

如图 3-7 所示。

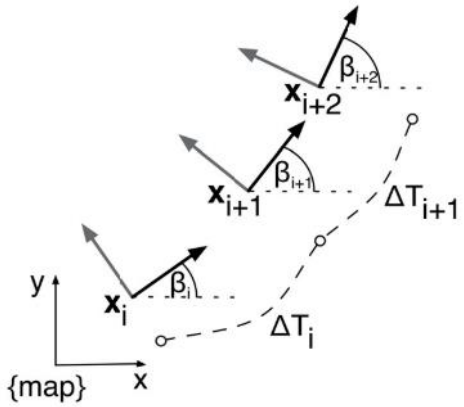


图 3-7 configurations 和时间差的序列

两个 configuration 间的时间间隔定义为ΔT_i，表示机器人由一个 configuration 运动到另一个 configuration 所需时间。

$$\tau = \{\Delta T_i\}_{i=0\ldots n-1} \quad (6)$$

每个时差表示机器人依次从一个 configuration 过渡到下一个 configuration 所需的时间。TEB 被定义为两个序列的元组：

$$B \coloneqq (Q, \tau) \quad (7)$$

其核心思想是通过实时加权多目标优化方法，从结构和时间间隔两方面对 TEB 进行自适应优化。

$$f(B) = \sum_k \gamma_k f_k(B) \quad (8)$$

$$B^* = argmin f(B) \quad (9)$$

其中 B^*表示被优化的 TEB 序列结果，$f(B)$表示全局目标函数，f_k目标函数的权值。作者在撰写论文时 ROS 对稀疏矩阵优化问题的支持并未十分完善，故采用分段连续，可微分的代价函数计算破坏约束的惩罚值。

$$e_{\mathrm{T}}(x, x_r, \epsilon, n) \simeq \begin{cases} \left(\frac{x - x_r - \epsilon}{S}\right)^n, if\ x > x_r - \epsilon \\ 0, \quad otherwise \end{cases} \quad (10)$$

其中x_r为界限值，S, n, ϵ影响近似的准确度。具体地说，S 表示缩放，n 表示多项

式阶数，ϵ表示限界值附近一个小位移。

3.3.2.2 约束目标函数

1. 跟随轨迹和避障约束

约束主要有两个目标：跟随已知的全局规划路径和避障。两个目标函数均十分相似，跟随路径施力将 elastic bands 拉向全局路径，而避障约束施力使得 elastic bands 远离障碍物。configuration 序列与全局路径点序列或障碍物z_j的最近距离记为$d_{min,j}$，如图 3-8 所示。跟随路径目标以 configuration 距全局路径的允许最大距离$r_{p_{max}}$作为约束，避障目标以 configuration 距障碍物的允许最小距离$r_{o_{\min}}$作为约束。

$$f_{path} = e_\tau\left(d_{min,j}, r_{p_{max}}, \epsilon, S, n\right)$$

$$f_{ob} = e_\tau\left(-d_{min,j}, -r_{O_{min}}, \epsilon, S, n\right) \quad (11)$$

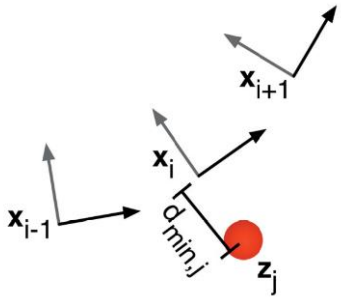


图 3-8

2. 速度和加速度约束

由速度和加速度组成的动力学约束可以用类似 1 中运动学约束的惩罚函数表示。以图 3-8 为例，机器人运动的平均线速度和角速度可以通过相邻的 configuration x_i, x_{i+1}和时间间隔ΔT计算得到。

$$v_i \simeq \frac{1}{\Delta T} \left\| \begin{matrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{matrix} \right\| \quad (12)$$

$$\omega_i = \frac{\beta_{i+1} - \beta_i}{\Delta T} \quad (13)$$

类似的，线速度约束可以表示为：

$$f_{v_i} = e_\tau(v_i, v_{max}, \epsilon, S, n) \quad (14)$$

机器人的平均线加速度用同样的近似计算得到：

$$a_i = \frac{2(v_{i+1} - v_i)}{\Delta T_i + \Delta T_{i+1}} \quad (15)$$

角加速度同理可计算得到。

以差动机器人为运动模型，两轮的转速可通过以下计算得到：

$$v_{\omega_r,i} = v_i + L\omega_i (16)$$

$$v_{\omega_t,i} = v_i - L\omega_i (17)$$

其中 L 为两轮轴距的一半。

对（16）（17）式作关于时间的微分得到两轮的加速度。

则可结合机器人设计的机械参数得到的轮子速度和加速度的上下界确定约束上下界。

3. Non-Holonomic 运动学约束

差动机器人在平面运动只有两个自由度，其只能以朝向的方向直线运动或旋转。这种运动学约束使得机器人以有若干弧段组成的平滑的轨迹运动。相邻的两个 configuration 应在弧段的两端。

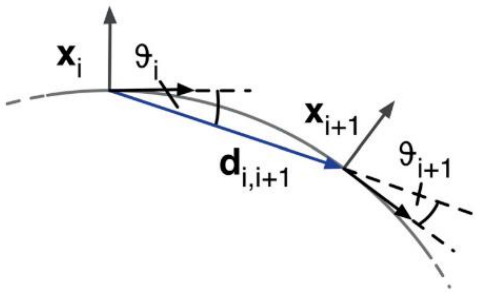


图 3-9　非完整运动学圆上构型之间的关系

初始 configuration x_i与运动方向$d_{i,i+1}$的夹角θ_i应与结束 configuration 对应的夹角θ_{i+1}相等。

若β_i为机器人在第 i 段弧段相对于世界坐标系的绝对姿态，则有：

$$\theta_i = \theta_{i+1} (18)$$

$$\Leftrightarrow \begin{pmatrix} \cos\beta_i \\ \sin\beta_i \\ 0 \end{pmatrix} \times d_{i,i+1} = d_{i,i+1} \times \begin{pmatrix} \cos\beta_{i+1} \\ \sin\beta_{i+1} \\ 0 \end{pmatrix} (19)$$

其中，运动方向向量：

$$d_{i,i+1} := \begin{pmatrix} x_{i+1} - xi \\ y_{i+1} - y_i \\ 0 \end{pmatrix} (20)$$

相应的目标函数：

$$f_k(x_i, x_{i+1}) = \left\| \left[\begin{pmatrix} \cos\beta_i \\ \sin\beta_i \\ 0 \end{pmatrix} + \begin{pmatrix} \cos\beta_{i+1} \\ \sin\beta_{i+1} \\ 0 \end{pmatrix} \right] \times d_{i,i+1} \right\|^2 (21)$$

4. 最快路径约束

目标函数即为最小化时间间隔序列的二次方。

$$f_k = \left(\sum_{i=1}^{n} \Delta T_i \right)^2 \quad (22)$$

目标函数使得机器人获得最快路径，路径上的各 configuration 点在时间上均匀分开，而非传统的空间上求最短路径。

3.3.2.3 TEB 算法实现

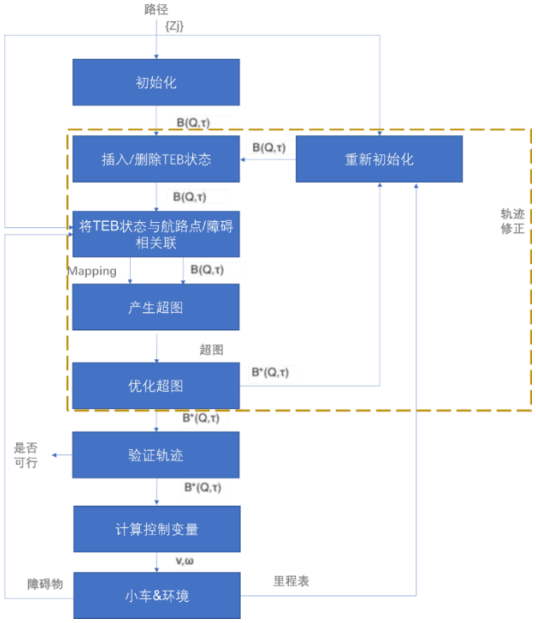


图 3-10 TEB 控制流程

每次迭代过程中，算法动态添加新的 configuration，删除旧的 configuration 以将关于空间和时间的分辨率调整至余下的轨迹长度和规划的范围内。添加一个 hysteresis 实现避障。优化问题转化为一个超图问题，通过 g2o 中关于大规模稀疏矩阵的优化算法解决。

TEB 优化问题（公式 8）可以转化为 hyper-graph 问题。configurations 和以及时间间隔作为 nodes，目标函数f_k以及其他约束函数为 edges，各 nodes 由 edges 连接起来构成 hyper-graph。该 graph 中，每一个约束都为一条 edge，并且每条 edge 允许连接的 nodes 的数目是不受限制的。

图 3-11 中展示了两个 configurations (s_0, s_1)，一个时间间隔ΔT_0，一个障碍点o_1组成的 hyper-graph。速度约束需要计算平均速度，则该约束与$s_0, s_1, \Delta T_0$相关，该 edge 将三个 nodes 连接起来。障碍物约束将障碍物和离障碍物最近的

configuration 连接起来。

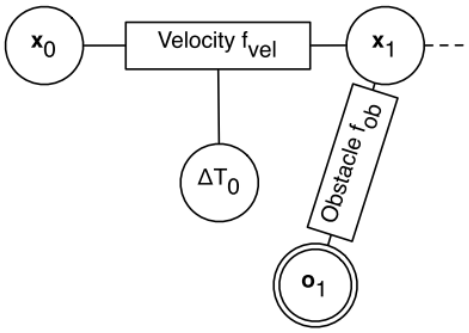


图 3-11　速度和障碍物目标函数被公式化为 hyper-graph

在确认能进行 TEB 优化后，将计算控制量v和ω直接用于机器人系统。在每次新迭代之前，重新初始化阶段都会检查新的和更改中的路径点，如果在分析短程相机或激光扫描数据后收到航路点，这将很有用。

3.3.2.4 算法核心代码

代码段 3-3　teb_local_planner 算法核心代码

```
// 截取全局路径的一部分作为局部规划的初始轨迹，主要是裁切掉机器人后方的一部分路径
pruneGlobalPlan(*tf_, robot_pose, global_plan_,
cfg_.trajectory.global_plan_prune_distance);
// 将初始的全局路径转换到局部坐标系下，便于后续进行局部规划
if (!transformGlobalPlan(*tf_, global_plan_, robot_pose, *costmap_,
global_frame_, cfg_.trajectory.max_global_plan_lookahead_dist,
transformed_plan, &goal_idx, &tf_plan_to_global))
// 更新路径上的航迹点
if (!custom_via_points_active_)
   pdateViaPointsContainer(transformed_plan,
cfg_.trajectory.global_plan_viapoint_sep);
// 更新障碍物，选择是否使用 costmap_converter 插件转换障碍物信息
if (costmap_converter_)
  updateObstacleContainerWithCostmapConverter();
else
  updateObstacleContainerWithCostmap();
// 开始执行局部轨迹规划
  bool success = planner_->plan(transformed_plan, &robot_vel_,
cfg_.goal_tolerance.free_goal_vel);
// 检查轨迹的冲突情况
bool feasible = planner_->isTrajectoryFeasible(costmap_model_.get(),
footprint_spec_, robot_inscribed_radius_, robot_circumscribed_radius,
cfg_.trajectory.feasibility_check_no_poses);
// 获取机器人需要执行的速度指令
```

```
if (!planner_->getVelocityCommand(cmd_vel.linear.x, cmd_vel.linear.y,
cmd_vel.angular.z, cfg_.trajectory.control_look_ahead_poses))
```

3.4 自定义规划器原理

3.4.1 概述

为了优化新版车模惯性矩过大的问题，我们使用 Python 语言手写了新的控制器来替换 teb_local_planner。包含该控制器的节点关系图如图 3-12 所示。

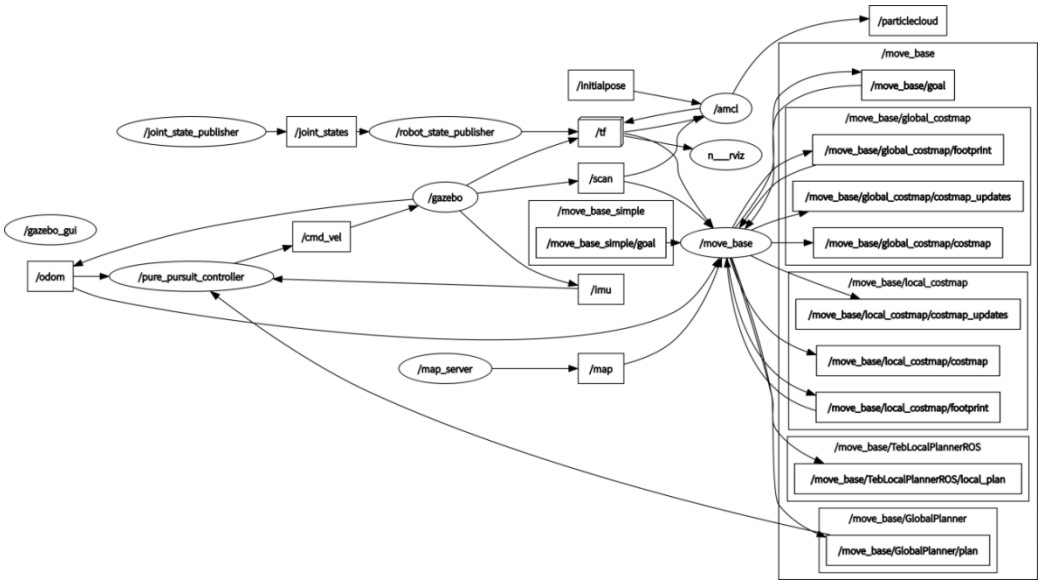


图 3-12 包含自定义规划器(pure_pursuit_controller)的 ros 节点图

从节点图中可以看出，我们订阅了/odom 话题来获取小车的坐标，订阅了/imu 话题来获取小车的位姿及加速度，订阅了/move_base/GlobalPlanner/plan 话题来获取全局规划路径，并把最终的速度控制命令发布于 cmd_vel 话题中。

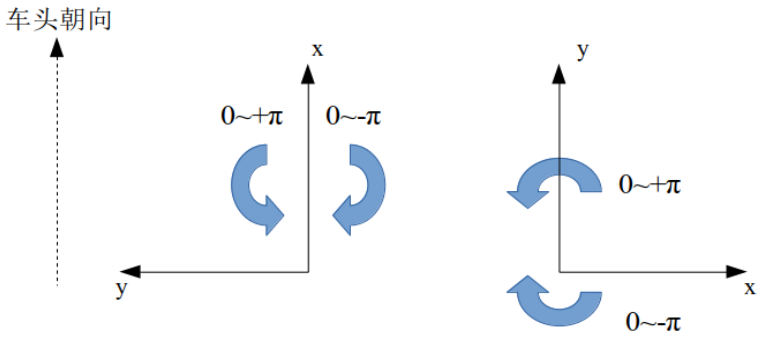
该控制器追踪的轨迹是由 GlobalPlanner 发布的，因为根据实验发现全局规划器的路径完全能够满足避障要求，跟随局部规划器的路径反而可能造成振荡、撞墙、倒车等问题。由于 GlobalPlanner 和 LocalPlanner 发布的 plan 都为 nav_msgs/Path 消息类型，若障碍环境较为复杂需要跟踪局部规划器路径，则只需要将控制器订阅的话题改为/move_base/TebLocalPlannerROS/local_plan 即可，不需要对控制算法本身做出修改。

3.4.2 位姿信息获取

小车的 x、y 坐标直接通过订阅/odom 话题即可，拥有了实时坐标我们还可以通过定周期取差值的方法获取 x 方向线速度。

通过订阅/imu 话题可以获取小车的偏航角及线加速度，查阅相关资料得知/imu 中发布的位姿信息是以四元数的形式发布的，我们是对小车的偏航角进行控制，因此还需要将位姿四元数转为欧拉角，通过 tf 库中的 euler_from_quaternion 函数即可实现转换。

需要注意的是 gazebo 中小车的位置坐标系及偏航角坐标系与常见的坐标系不同，如图 3-13 所示。为了便于计算，在获取小车的原始位姿后我们将其换算为常见坐标系下的数值。



（a）gazebo 坐标系　　（b）常见坐标系

图 3-13 坐标系

3.4.3 轨迹跟踪

1. 目标点的选择

全局规划器发布的路径为 nav_msgs/Path 型数组，查阅相关资料得知该消息类型的数据结构为：

```
Header header
geometry_msgs/PoseStamped[] poses
```

其中 geometry_msgs/PoseStamped 类消息的数据结构为：

```
Header header
Pose pose
```

从中可以看出，全局路径实际上包含了从小车当前位置出发直到目标点的路

程中所有中间点的坐标信息。通过 rqt_topic 插件观察发现，随着小车移动，poses 数组的每一个元素（PoseStamped 类中的 pose 域）只有 position 数据域在不断更新，而 orientation 域始终不变，因此全局规划器实际上只发布了路径点的 x、y 坐标。

对于全局路径跟踪，应当给予一定的前瞻距离，否则当小车高速运行时其避障能力和切弯能力会受到影响。由于/move_base/GlobalPlanner/plan 发布的路径点都是等距离的，且 poses 数组随着小车的运动随时更新，固定数组下标获得的路径点就是距小车相等距离的路径点。因此我们只需要加入一个数组前瞻索引即可实现轨迹跟踪的前瞻距离选择，而无需进行额外计算。

2. 跟踪方式

由于小车采用差速驱动方式，我们借鉴在阿克曼模型中适用的 pure_pursuit 算法来实现轨迹跟踪，即不断调整小车偏航角使其正对下一个目标点。只需要计算小车当前坐标到目标点位置的向量，再求向量 y 方向分量对 x 方向分量的反正切值即可获得偏航角数值。然而，由于向量计算法只取锐角，为了将其转为在图 3-13 所示的坐标系下，还需对数值进行分类讨论。

如图 3-14 所示，根据目标向量在 x、y 方向上投影向量的正负共有四种情况，可以进行分类讨论从而得到计算偏航角α与实际偏航角θ的映射关系。

$$\theta = \begin{cases} \alpha, \vec{m}_x > 0, \vec{m}_y > 0 \\ \pi - \alpha, \vec{m}_x < 0, \vec{m}_y > 0 \\ -(\pi - \alpha), \vec{m}_x < 0, \vec{m}_y < 0 \\ -\alpha, \vec{m}_x > 0, \vec{m}_y < 0 \end{cases} \tag{23}$$

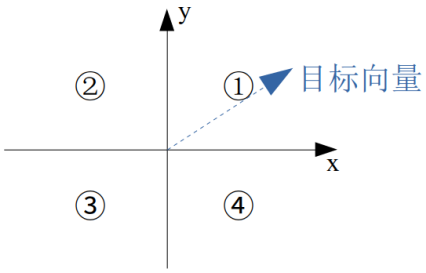


图 3-14 小车距离目标点的向量

3.4.4 PID 控制器

1. 简介

在获取目标偏航角后，我们需要根据小车当前偏航角与目标偏航角的偏差得

到合适的控制量。

PID 是比例(Proportion)积分，(Integral)微分，(Differential coefficient)的缩写，分别代表了三种控制环节。通过这三个控制环节的组合可有效地纠正被控制对象的偏差，从而使其达到一个稳定的状态。其系统方框图如图 3-15 所示。

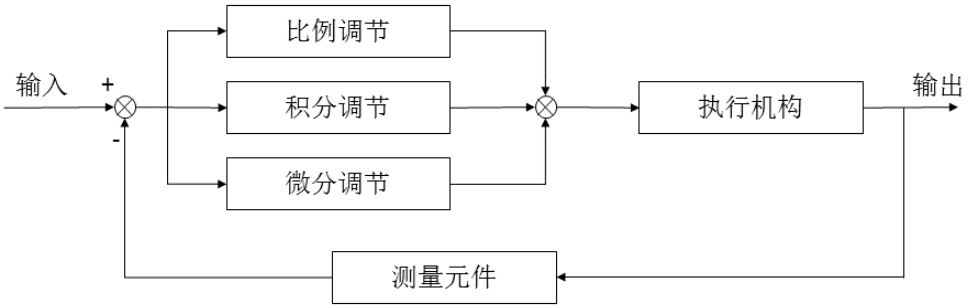


图 3-15 PID 控制器方框图

在工业过程中，连续控制系统的理想 PID 控制规律为：

$$u(t) = K_p\left(err(t) + \frac{1}{T_i}\int err(t)dt + T_d\frac{\mathrm{d}err(t)}{dt}\right) \quad (24)$$

式中，K_p——比例增益；

T_i——积分时间常数；

T_d——微分时间常数；

$u(t)$—PID 控制器的输出信号；

$err(t)$——输入值与期望值之差。

PID 的比例项用来对系统的偏差进行反应，所以只要存在偏差，比例项就会起作用；积分项通过偏差的累计来抵消系统静差；微分项对偏差的变化趋势做出反应，根据偏差的变化趋势实现超前调节，提高反应速度。

将 PID 控制器应用于计算机时需要将其离散化，PID 算法的离散化表达共有两种形式：位置式 PID 和增量式 PID。设采样周期为 T，系统正在进行第 k 次采样，则位置式 PID 的表达式为：

$$u(k) = K_p err(k) + K_i\sum err(k) + K_d\big(err(k) - err(k-1)\big) \quad (25)$$

增量式的 PID 的表达式为：

$$\Delta u(k) = K_p\big(err(k) - err(k-1)\big) + K_i\sum err(k) + K_d(err(k) - 2err(k-1) + err(k-2)) \quad (26)$$

2. PID 控制器在仿真赛中的应用

（1）PID 类的构建

为了灵活的应用 PID 控制器，我们将该算法封装为一个 Python 类，给定设置

比例系数、积分系数、微分系数等参数的接口，可以方便的初始化 PID 实例，该类的初始化函数如代码段 3-3 所示。

代码段 3-4　PID 类初始化函数

```
def __init__(self, kp, ki, kd, int_duty=0.01, int_max=100,
output_max=10, sub_ctrl=False):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.now_value = 0  # 实际值
        self.now_err = 0  # 当前偏差
        self.last_err = 0  # 上一轮偏差
        self.last_last_err = 0  # 上上轮偏差
        self.delta_value = 0  # 增量式式 PID 的控制增量
        self.int_count = 0  # 积分量
        self.expect_value = 0  # 期望值
        self.int_duty = int_duty  # 积分周期/s
        self.curr_time = rospy.get_time()
        self.last_time = 0
        self.int_max = int_max  # 积分限额
        self.output_max = output_max  # 控制量限额
        self.sub_ctrl = sub_ctrl  # 是否开启分段 pid
```

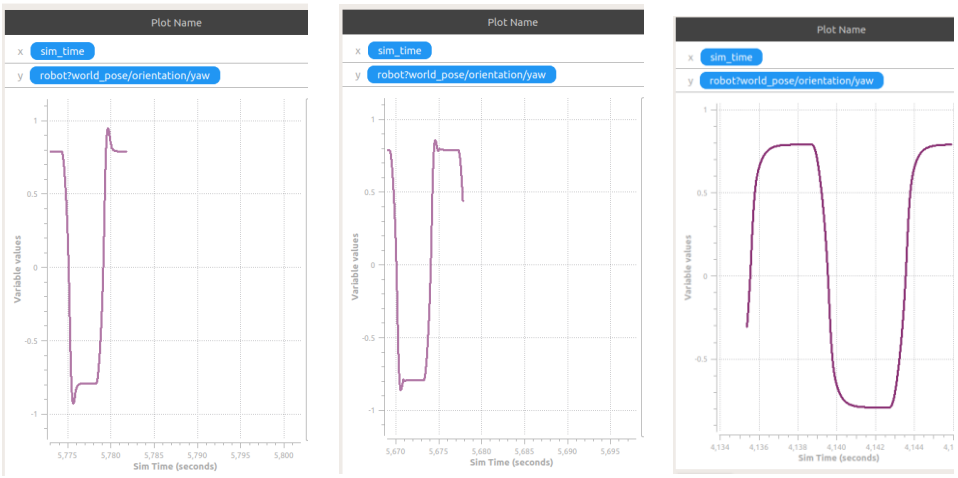
（2）偏航角的 PID 控制

我们将小车当前偏航角与目标偏航角的偏差作为控制器输入量，将输出量作为小车角速度的期望值。

严格意义上对于角度的控制应当使用*串级 PID*，即采用两个 PID 控制器串联工作，外环角度 PID 控制器的输出作为内环角速度 PID 控制器的设定值，由内环控制器的输出去操纵控制阀，从而对外环被控量具有更好的控制效果。由于仿真环境较为理想，只采用一级 PID 控制即可满足要求，且通常不会产生静态误差，故积分系数可以置 0 而转化为 PD 控制。图 3-16 展示了偏航角 PD 控制器的调节效果。

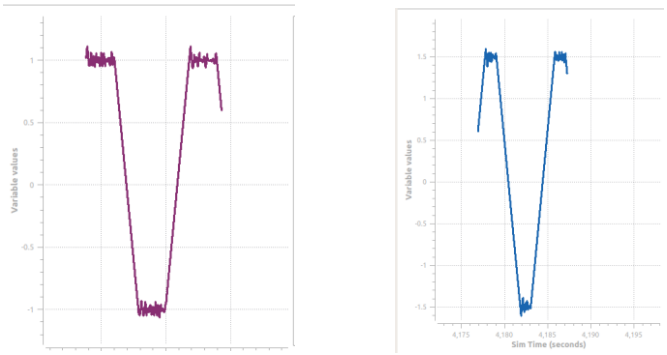
（3）线速度 PID 控制

将小车当前的线速度作为输入量，控制速度作为输出量即可完成线速度的 PID 控制模型。在调试的过程中我们发现，在比例系数与积分系数设置恰当的情况下，无论怎么调试积分系数最终的输出量依然会高频振荡。对于该问题，我们决定采用分段 PID 控制，对于不同的偏差量大小使用不同的 PID 参数，当偏差量小于一定阈值后采用更小的比例系数与积分系数，减缓调节时间并缓解积分系数对高频噪声的放大作用。线速度的 PID 调试过程如图 3-17 所示。

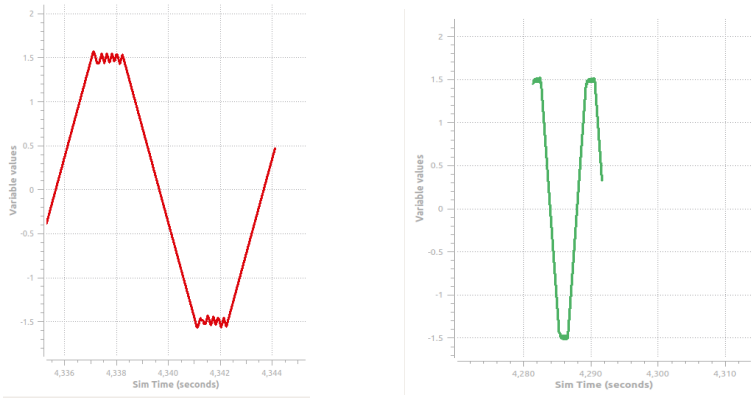


（a）纯 P 控制　　（b）PD 控制（参数调试中）　　（c）PD 控制（参数调试完成）

图 3-16 偏航角 PD 控制器调节效果



（a）纯 P 控制（期望值±1.5）　（b）加入积分环节



（c）加入微分环节　　（d）分段 PID

图 3-17 线速度 PID 控制器调节效果

进一步调试发现，在仿真环境中对线速度使用 PID 控制没有必要，因为在 gazebo 差分控制器收到速度命令后便会以最大加速度调整到期望值，且不会出现振荡与超调，故在最终的代码版本中已取消线速度的 PID 控制，但这一算法在线下赛中或许会派上用场。

（4）控制量限额

在调节比例系数的过程中我们发现，不论对于线速度还是角速度，其控制量（发布到 cmd_vel 中的速度命令）不能超过某个值，否则小车会失控，控制过程无法收敛。经测试，线速度的最大控制量为 6m/s，角速度的最大控制量为 20rad/s 左右。至于为什么存在这种阈值我们暂无从得知，姑且认为是 gazebo 差分控制器的默认值，而官方提供的 urdf 文件中并未对速度最大值进行限制。

第 4 章 项目实施过程

4.1 Gazebo 环境部署

开始 gazebo 仿真的第一步是配置 ROS 工作空间，新建一个工作空间的具体步骤如代码段 4-1 所示。

代码段 4-1 新建 ROS 工作空间

```
$ mkdir -p ~/ifly_race_ws/src
$ cd ~/ifly_race_ws/src
$ catkin_init_workspace
$ cd ..
$ catkin_make
```

组委会已经为我们提供了加载 gazebo 世界及车模的功能包，我们只需将 gazebo_pkg 文件夹其复制到~/ifly_race_ws/src 目录下，将模型文件复制到 ~/.gazebo/models 目录下即可。

由于新增了功能包，我们需要让 ROS 知道它的位置，具体方法是执行工作空间下 devel/setup.sh 文件即可。但这种做法只会在当前终端中生效，为了使每次启动终端时都能执行该文件，我们需要修改终端的自动文件.bashrc，具体步骤如代码段 4-2 所示。

代码段 4-2 修改.bashrc 文件

```
$ sudo echo source ~/ifly_race_ws/devel/setup.sh >> ~/.bashrc
```

首次启动 gazebo 时，我们遇到了加载时间过长的问题。查阅资料得知这是由于 gazebo 在线服务器下行速度太慢，一些模型文件无法成功下载。于是我们从国内网站上下载了 gazebo 所需的基本模型文件并将其复制到~/.gazebo/models 路径下，最终成功解决问题。

4.2 SLAM 建图

4.2.1 建图功能包构建

由于小车模型配有激光雷达、IMU 和里程计，因此我们可以选择常见的 SLAM 算法如 gmapping、hector、karto、cartographer 等。ROS 已经为这些导航算法提供了功能包，我们只需要在 roslaunch 文件中声明该算法对应的节点并为其配置相关参数即可。以 gmapping 算法为例，由于需要激光雷达和里程计的信息，因此我们需要让 gmapping 功能包知道接收这些数据的话题是什么，具体配置见代码段 4-3。

代码段 4-3 ifly_gmapping.launch 文件

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="robot" doc=""/>
  <arg name="configuration_basename" default="ifly_lds_2d.lua"/>
  <arg name="set_base_frame" default="base_footprint"/>
  <arg name="set_odom_frame" default="odom"/>
  <arg name="set_map_frame"  default="map"/>

  <!-- Gmapping -->
  <node pkg="gmapping" type="slam_gmapping" name="ifly_slam_gmapping"
output="screen">
    <param name="base_frame" value="$(arg set_base_frame)"/>
    <param name="odom_frame" value="$(arg set_odom_frame)"/>
    <param name="map_frame"  value="$(arg set_map_frame)"/>
    <rosparam command="load" file="$(find
ifly_slam)/config/gmapping_params.yaml" />
  </node>
</launch>
```

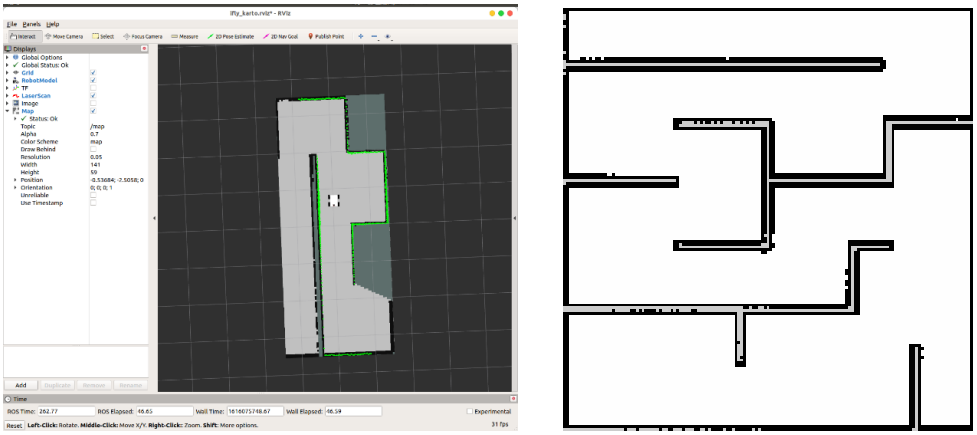
最初我们使用 *Turtlebot3* 中默认的 gmapping 算法进行建图，但发现由于该算法高度依赖里程计的精度，导致在建图过程中一旦小车旋转，地图就会发生重叠的现象，最终生成的地图存在边界模糊、精度差、大小不一致等问题。

经过测试，我们发现 karto 算法的建图效果最好，因此最终使用该算法进行建图。由于我们分别为三种建图算法写了单独的 roslaunch 文件，因此从 gmapping 切换到 karto 很简单，只需要在启动 SLAM 的 roslaunch 文件中更改“slam_methods”参数即可。启动 SLAM 的 roslaunch 文件如代码段 4-4 所示。

代码段 4-4 ifly_slam.launch 文件

```
<launch>
    <!-- Arguments -->
    <arg name="model" default="robot" />
    <arg name="slam_methods" default="gmapping" doc="slam type [gmapping,
cartographer, hector, karto, frontier_exploration]"/>  <!--建图算法-->>
    <arg name="configuration_basename" default="ifly_lds_2d.lua"/>
    <arg name="open_rviz" default="true"/>
    <!-- SLAM: Gmapping, Cartographer, Hector, Karto, Frontier_exploration,
RTAB-Map -->
    <include file="$(find ifly_slam)/launch/ifly_$(arg
slam_methods).launch">
        <arg name="model" value="$(arg model)"/>
        <arg name="configuration_basename" value="$(arg
configuration_basename)"/>
    </include>
    <!-- rviz -->
    <group if="$(arg open_rviz)">
        <node pkg="rviz" type="rviz" name="rviz" required="true"
            args="-d $(find ifly_slam)/rviz/ifly_$(arg slam_methods).rviz"/>
    </group>
</launch>
```

使用 karto 算法进行 SLAM 建图的运行效果如图 4-1 所示。



（a）建图过程　　（b）生成的地图

图 4-1 karto 算法建图效果

4.3 导航

导航功能包是本次线上赛的关键，导航包的正确构建是完成比赛的基础，而导航算法的实现和导航参数的调试也将直接影响到完赛时间。

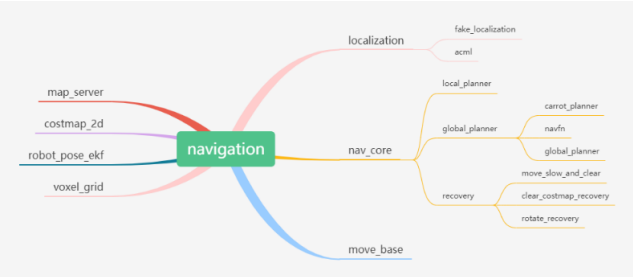


图 4-2　导航功能包集

4.3.1 从 Turtlebot3 开始

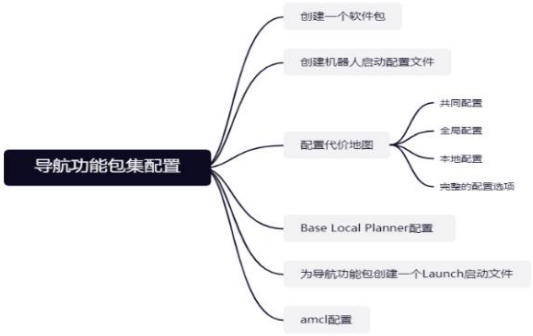


图 4-3　导航功能包结构

由于我们开始备赛的时间较早，当时官方还没有推出导航包构建教程，因此我们只能自己探索。对于繁杂的配置文件，从头开始手写显然是不现实的，对此我们决定从著名的 ROS 开源项目 *Turtlebot3* 中移植代码。该项目涵盖了本次线上赛需要用到的 navigation 功能包配置，运行效果见图 4-4 与图 4-5。

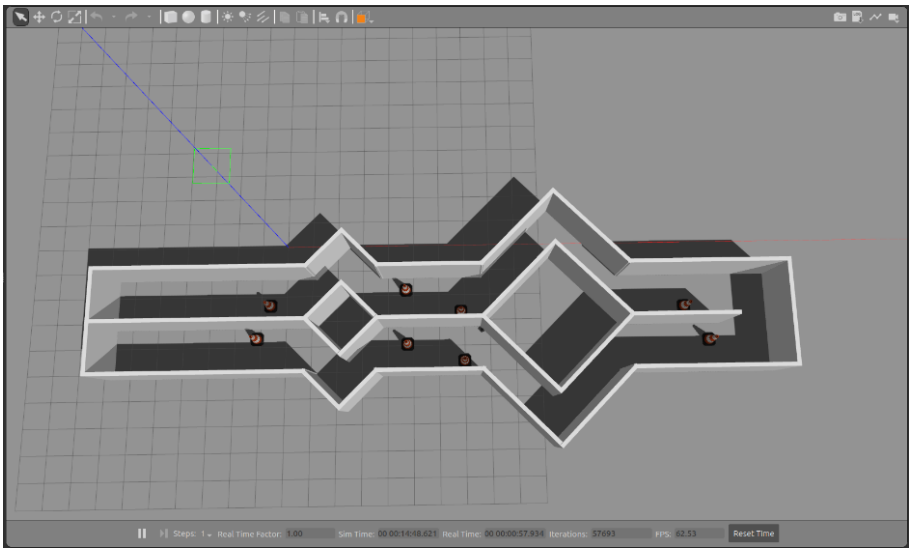


图 4-4turtlebot3_runway 地图

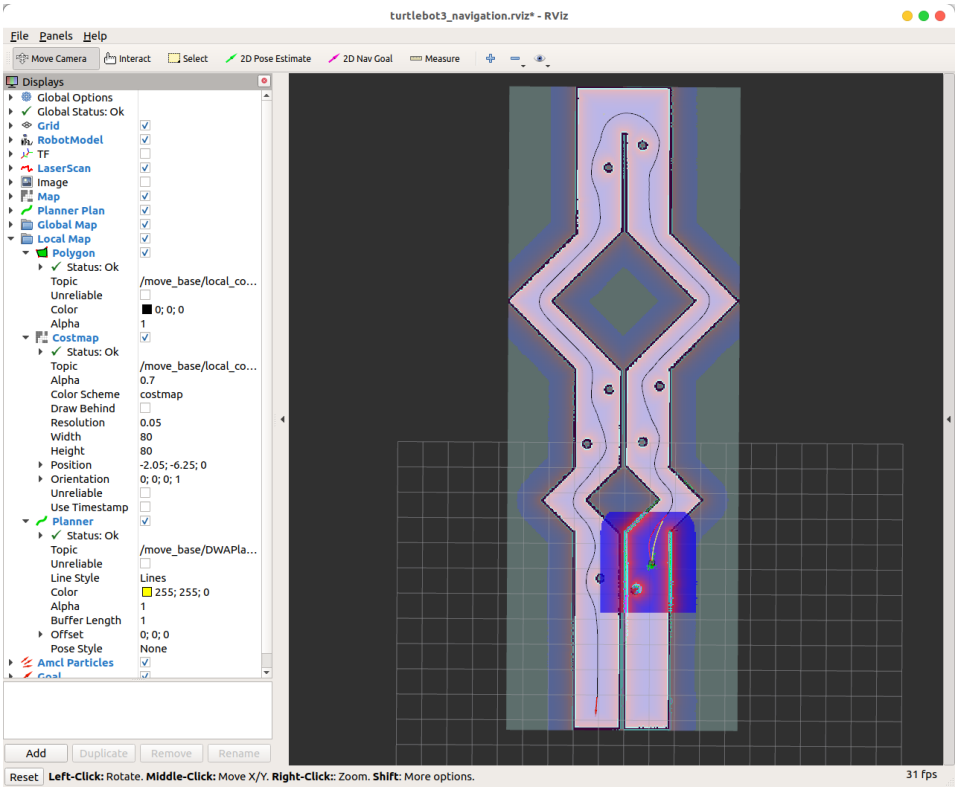


图 4-5 turtlebot3 导航包运行效果

我们在官方模型与地图的基础上以 *Turtlebot3* 为模板构建了比赛所用的 navigation 功能包，对 roslaunch 文件中的模型加载路径、参数等做了针对性修改，实现了线上赛功能包的构建，即源代码附件中的 ifly_navigation 文件夹。

4.3.2 配置导航功能包启动文件

为了运行 ROS navigation 功能包集，我们需要指定一些加载参数并一次运行多个节点，我们使用 roslaunch 文件来实现上述任务。

代码段 4-5 ifly_navigation.launch 文件

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="robot"/>
  <arg name="map_file" default="$(find ifly_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="false"/>
  <arg name="multi_robot_name" default="false"/>
  <arg name="local_planner_method" default="dwa" doc="teb"/>

  <!-- 加载模型 -->
  <arg name="urdf_file" default="$(find xacro)/xacro --inorder '$(find
gazebo_pkg)/urdf/waking_robot.xacro'" />
  <!-- <param name="robot_description" command="$(arg urdf_file)" /> -->

  <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg
map_file)"/>

  <!-- AMCL -->
  <include file="$(find ifly_navigation)/launch/amcl.launch"/>
  <!-- <include file="$(find amcl)/examples/amcl_diff.launch" /> -->

  <!-- move_base -->
  <include file="$(find ifly_navigation)/launch/move_base.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find
ifly_navigation)/rviz/ifly_navigation_dwa_rviz.rviz"/>
  </group>
</launch>
```

如代码段 4-5 所示，该文件中指定了地图文件和模型参数文件的路径，加载了 map_server 节点，并包含了 move_base 和 amcl 的 roslaunch 文件，是一个顶层

启动文件。

在 amcl.launch 文件中我们加载了节点，并为 AMCL 算法指定了必要的参数，例如粒子数量限制、机器人初始坐标、激光雷达特性、驱动方式等。在 move_base.launch 文件中我们为 move_base 功能包指定了全局规划器和局部规划器及相应的参数配置文件所在目录。以上两个文件的具体内容见源代码，在此不做赘述。

4.3.3 配置代价地图

代价地图参数对于本地规划器是至关重要的。在 ROS 中，代价地图由静态地图层、障碍物图层和膨胀层组成。静态地图层直接给导航堆栈提供静态 SLAM 地图解释。障碍物图层包含 2D 障碍物和 3D 障碍物（体素层）。膨胀层是将障碍物膨胀来计算每个 2D 代价地图单元 的代价。

静态地图层对提供给导航功能包的静态 SLAM 地图直接进行解释。障碍层包括有 2D 的障碍物和 3D 的障碍物（体素层）。膨胀层将障碍物进行膨胀以计算每个 2D 代价单元 (cell) 的通行代价。此外，还存在全局代价地图 (global costmap)，和局部代价地图 (local costmap)。全局代价地图通过膨胀导航功能包的地图上已存在的障碍物产生。局部代价地图通过将机器人传感器检测到的障碍物进行膨胀产生。[1]

为了对代价地图的参数进行管理，我们创建 costmap_common_params.yaml、global_costmap_params.yaml、local_costmap_params.yaml 三个文件，用于指定膨胀半径、缩放因子、机器人投影尺寸等参数。

4.3.4 配置全局规划器

全局规划器在运动之前，根据接收到的目标位姿基于 global_costmap 提供的全局地图调用对应的路径规划算法生成一条从当前位置到目标位置的路径，这里的 global_costmap 是根据之前用 karto 算法创建的地图 map.pgm 而生成的，它区别于局部规划器，在规划的过程中并未获取当前传感器的数据。全局规划器根据预先采集的地图数据，计算一条从当前位置到目标位置的可行的最优路径，但它不会考虑路况，也不会避开障碍物，因为它的数据来源只有地图，并没有实时路况数据。

ROS 提供的局规划器的插件：navfn，carrot_planner 以及 global_planner，我们选择 global_planner 做为全局规划器，之后创建了 global_planner_params.yaml

文件，用于管理对于 global_planner 的具体参数。

4.3.5 配置局部规划器

起初我们使用 *Turtlebot3* 默认的 dwa_local_planner 规划器，经过一系列配置，我们成功在 2 月末让小车顺利跑完全程，而此时很多队伍仍然处于配置 ROS 开发环境的阶段，这对我们是一种极大的鼓舞。

由于 navigation 功能包集支持多种局部规划器，为了更好的发挥团队作用，我们决定“兵分三路”，即队内三个人分别负责调试一种局部规划器，选出实际效果最好的一种。

我们使用 Git 来进行分支管理和版本控制。如图 4-6 所示，仓库分别创建了三种局部规划期的调试分支，分别由一名队友负责，当出现重大突破的时候便提交一次 commit 并 push 到远程仓库，这样大家就可以知道彼此的调试进度。

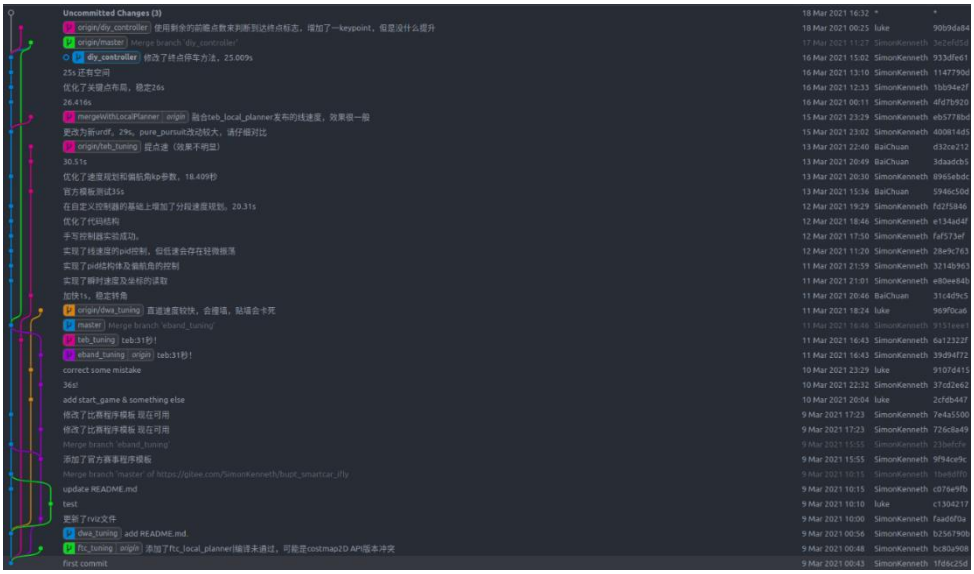


图 4-6 Git 开发树

对于 dwa_local_planner，在进一步调试的过程中我们发现 DWA 规划器在 U 型弯附近经常会突然减速，然后执行原地旋转的操作，无法流畅的过弯，非常影响比赛用时。随后，我们尝试通过降低前向模拟时间(sim_time)，并增大全局路径跟随权重（path_distance_bias），企图让小车更贴合全局路径来避免陷入局部最优解。经过测试，这种方法确实有效，但依然无法完全避免弯道处停车转向的问题，而 DWA 也没有提供针对该问题的参数接口，鉴于此，我们决定弃用 DWA 规划器。

对于 eband_local_planner，由于该规划器提供的可调试参数较少，因此其调

试上限是很容易达到的。即使 eband_local_planner 提供了一些专门用于优化运动控制的参数，例如 PID 控制器的 k_prop、k_damp 等，但经过调试，实际比赛用时并没有给人惊喜，最终的极限时间在 32~38 秒之间。鉴于此，eband 规划器同样被弃用。

而对于 teb_local_planner，该规划器的可调式参数最多，尤其是有大量专门用于优化的参数，而且调试过程中发现这些参数对车模的运行都会起作用，因此可以预见，teb_local_planner 的调试上限最高。

综上，我们决定采用 teb_local_planner，而此时团队合作策略也随之发生变化。鉴于官方讨论群中绝大多数队伍都开始采用 teb_local_planner，考虑到不同学校的课业负担不同，如果单纯比拼调参，则一定不会拉开太大差距，因此我们决定让一名队员继续 teb_local_planner 的调试，抽出两名队员进行自定义控制器的开发，其开发进展也可从图 4-6 中看到，反馈量的实现、PID 控制器的实现、最新比赛成绩等都展示在 Git Graph 中，给团队合作带来了极大便利。

4.3.6 动态速度规划

有关动态速度规划的方案思想已由 2.4.2 节叙述，本节阐述具体的实现方法。首先，我们需要规定出应当设置哪些关键点，因为小车最需要限速的路段是弯道，因此我们的基本选取原则是入弯点和出弯点，即小车在入弯点处减速，过弯后经过出弯点时再加速。除此之外，如果有长直道，则可以先让小车达到很高的速度，再用剩下的距离全力减速入弯。在终点附近我们设置了较多关键点，来确保到达终点时可以可靠地停车。最终选取的关键点如图 4-7 所示。

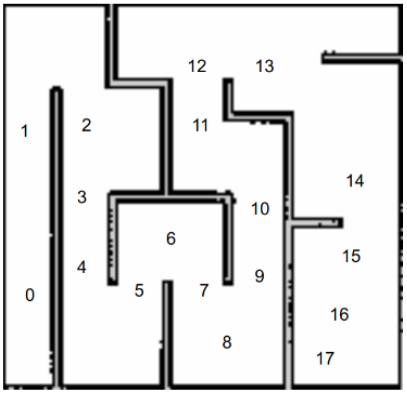


图 4-7 地图关键点标记

得到关键点后，我们需要为其指定速度和加速度限制，这可以通过 python list

数据类型实现，如代码段 4-6 所示，每一行代表一个关键点，其中的元素分别代表该关键点的 x、y 坐标、偏航角、线速度限制、角加速度限制。

为了能让小车在通过关键点后改变局部规划器参数，首先要获取小车的瞬时坐标，这可以通过订阅/odom 话题实现。当小车前进时，不断计算当前坐标与关键点坐标的欧氏距离，当距离小于一定阈值后便认为小车已经通过关键点，此时通过 ROS 的 dynamic_reconfigure 机制将期望参数上载到 ROS 参数服务器中，局部规划器将自动从参数服务器下载参数，从而实现数值的更新。最后，再将关键点数组的下标+1，开始计算小车与下一个关键点的距离。

代码段 4-6 关键点 List 与经过点判定方式

```
key_points = [
    [2.402670, -0.000090, 0.000, 1.8, 1.5],  # 0
    [3.75, -0.000090, 0.000, 0.75, 1.5],  # 1
    [4.702972, -1.080354, 3.082496, 1.25, 1.5],  # 2
    [2.936867, -1.032038, -3.14, 1.43, 1.5],  # 3
    [1.995490, -1.044512, -2.983582, 0.8, 1.5],  # 4
    [1.339966, -2.061234, -2.934374, 1.0, 1.5],  # 5
    [1.741479, -2.556157, -1.542630, 0.85, 1.5],  # 6
    # [0.661139, -3.180960, -1.812663,1.3],  # 7
    # [4.646070, -1.056370, -2.934374,1.3],  # 8
    [1.504895, -4.168083, 0.000, 1.35, 1.5],  # 9
    [2.956114, -4.201780, 0.000, 1.1, 1.5],  # 10
    [4.295692, -3.125136, 0.000, 1.3, 1.5],  # 11
    [5.027173, -3.123793, 0.000, 1.1, 1.5],  # 12
    [5.194778, -4.593526, -2.392219, 2.0, 2.0],  # 13
    [2.902147, -5.983389, 0, 1.85, 2],  # 14
    [1.717385, -5.971060, 0, 1.55, 2],  # 15
    [0.836030, -5.705181, 2.962307, 0.00, 1.0],  # 16
    [-0.2504603767395, -5.2709980011, 2.446521, 0.0, 0]  # 17 终点
]
PASS_THRES_RADIUS = 0.5
def is_passed(now_pos, next_waypoint):
    dis_to_next_point = math.sqrt(
        (now_pos[0]-next_waypoint[0])**2+(now_pos[1]-next_waypoint[1])**2)
    # print('--dis:%.2f %.2f'%(dis_to_next_point[0],dis_to_next_point[1]))
    if dis_to_next_point <= PASS_THRES_RADIUS:
        return True
    else:
        return False
```

dynamic_reconfigure 的 Python API 使用字典作为数据结构，更新参数的方式如代码段 4-7 所示。

代码段 4-7　使用 dynamic_reconfigure Pyhon API 动态更新局部规划器参数

```
reconfig_client = dynamic_reconfigure.client.Client(
          '/move_base/TebLocalPlannerROS')
params = {'max_vel_x': key_points[i][3],'acc_lim_theta':key_points[i][4]}
config = reconfig_client.update_configuration(params)
```

4.3.7 实现自定义局部规划器

3.3 节中已经详细叙述了自定义局部规划器的原理，本节对具体实现方法进行叙述。

1. 局部规划器的实现

为了替换掉 move_base 中的局部规划器，首先要解决的问题是切断其控制小车的途径。通过 rqt_graph 工具我们可以得知，move_base 通过/cmd_vel 话题控制小车移动，如图 4-8 所示。故只需要让 move_base 将速度命令发送至其他话题，或让 gazebo 差分控制器接收其他话题的速度命令即可，由于组委会要求不能修改 urdf 文件，故选择第一种方式。

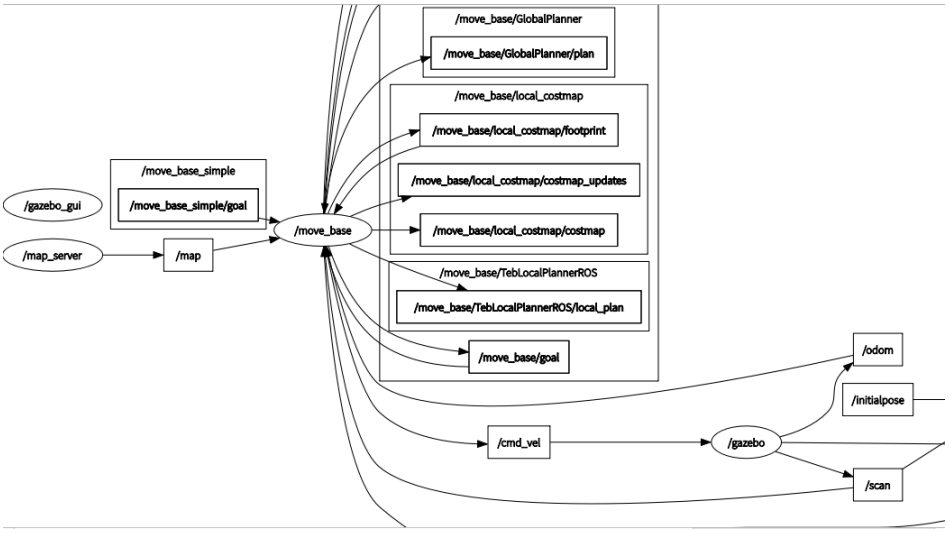


图 4-8　move_base 节点图（部分）

由于 ROS Python API 要求订阅话题时提供回调函数，因此更新位姿必须通过某个函数实现。如果将位姿更新函数作为 Python 类的成员函数，位姿信息作为成员变量，那么就可以轻松实现多个函数间的信息共享。因此我们将整个局部规划器封装为一个 Python 类，声明实例后将指定的成员函数关联到 Subscriber，如代码段 4-8 所示。

代码段 4-8 将 PathFollower 类的成员函数作为订阅器的回调函数

```
path_follower = PathFollower(forehead_index=44)  # 轨迹跟踪器实例
    path_sub = rospy.Subscriber(
        "/move_base/GlobalPlanner/plan", Path,
path_follower.update_globle_path)  # 订阅全局规划器发布的路径
    imu_sub = rospy.Subscriber("/imu", Imu, path_follower.update_posture)
    odom_sub = rospy.Subscriber(
        "/odom", Odometry, path_follower.update_position)
```

我们将自定义局部规划器对小车的实际控制分为两层，顶层控制函数（名为 follow）监控轨迹跟踪状态，并获取下一个目标点的目标偏航角以及目标线速度，将其传递给中层控制。中层控制函数（control）根据小车当前偏航角与目标偏航角的偏差通过 PID 控制器计算得到角速度控制量，根据当前线速度与目标线速度的偏差计算得到线速度控制量，并将控制量发布到/cmd_vel 话题，从而实现实际控制。具体函数实现见源代码。

在调试的过程中我们发现，新版车模的线速度和角速度不能同时变大，否则小车的转弯半径会非常大，导致过弯时撞墙。经过分析，我们认为是由于角加速度有上限，因为

$$a = \omega v \quad (27)$$

故线速度和角速度的乘积有一最大值。因此我们需要在弯道处（角速度大的地方）限制线速度，$v = a_{max}/\omega$，a_{max}需要通过实验测得。

对动态速度规划的调试趋于极限后，我们发现动态调整前瞻距离也非常有利于提高小车的运行速度。在某些弯道处增大前瞻距离将有助于小车提前产生转向动作，从而顺利入弯；在曲率较小的弯道处增大前瞻距离能让小车轨迹更接近直线的，增大线速度。

最后需要注意的是，由于计算目标偏航角时取用路径规划数组，因此控制器的调用频率一定要低于路径的发布频率，否则控制器会接收到空数组，小车会保持上一次的偏航角继续前进，而一旦接近障碍物到一定程度全局规划器也无法产生新的路径，从而导致小车失控。

2. PID 控制器的实现

我们将 PID 控制器单独写在一个.py 文件中，并将其封装为一个 Python 类，以便于复用和维护。PID 类的初始化函数中可以声明 Kp、Ki、Kd、积分限额、控制量限额、是否启动分段 pid 等，可以根据不同的控制对象进行具体配置。

PID 控制器的计算部分被封装为 get_output 函数，由于 PID 参数在类内共享，因此计算函数只需给定期望值和当前值两个参数即可。

在调节比例系数的过程中我们发现，不论对于线速度还是角速度，其控制量（发

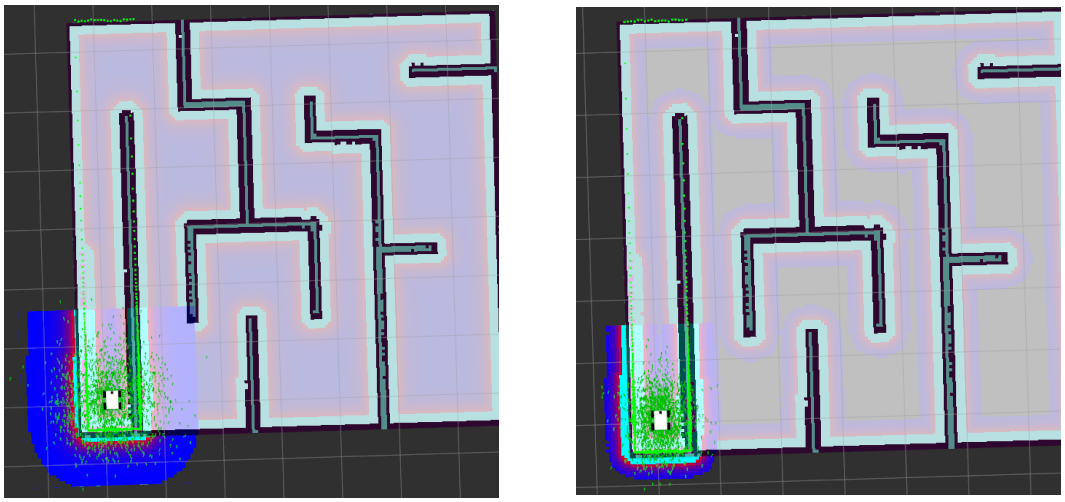
布到 cmd_vel 中的速度命令）不能超过某个阈值，否则小车会失控，控制过程无法收敛。这里所指的阈值与角加速度无关，只要线速度或角速度其中一个超过阈值小车就会失控。经测试，线速度的最大控制量为 6m/s，角速度的最大控制量为 20rad/s 左右。至于为什么存在这种阈值我们暂无从得知，姑且认为是 gazebo 差分控制器的默认值，而官方提供的 urdf 文件中并未对速度最大值进行限制。因此，我们在 PID 类的初始化函数中增加了控制量输出上限，若计算值大于上限则使用上限值。

第 5 章 项目数据分析

5.1 代价地图参数分析

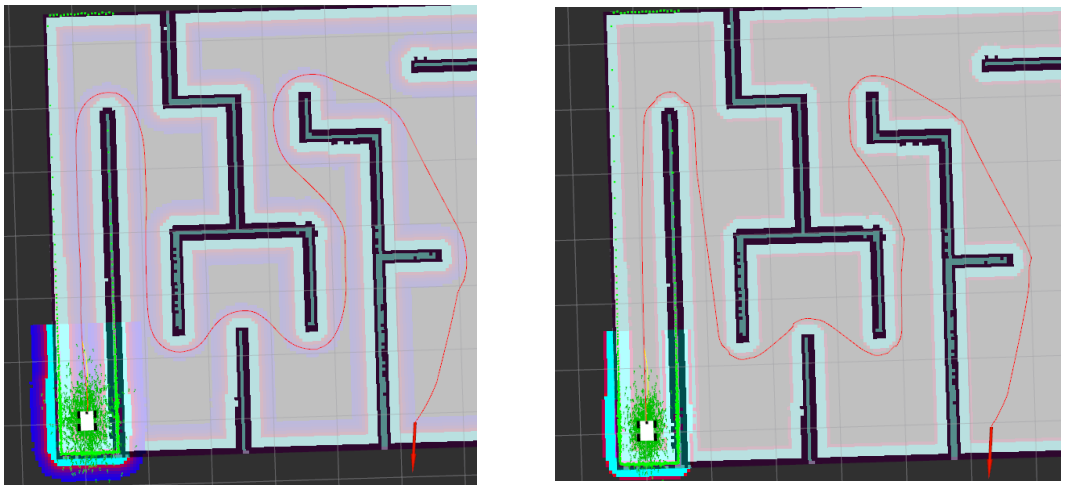
代价地图的可调试参数中 inflation_radius 和 cost_scaling_factor 两个参数最为重要。inflation_radius 控制零代价点距离障碍物有多远，即决定了障碍物周围的膨胀半径，该值越小规划路径将越贴近于障碍物；cost_scaling_factor 是膨胀的衰减系数，设置高值将使衰减曲线更为陡峭[1] 。

在图 5-1（a）中，inflation_radius=1，cost_scaling_factor=6；图 5-1（b）中 inflation_radius=0.4，cost_scaling_factor=10。



(a)温和的膨胀曲线　　(b)陡峭的膨胀曲线

图 5-1 不同参数下地图膨胀情况



(a)中膨胀半径对应的轨迹　　(b)低膨胀半径对应的轨迹

图 5-2 不同膨胀半径对应的全局路径

如图 5-2 所示，中膨胀半径生成的轨迹更为保守，而低膨胀半径生成的轨迹将更为极限，小车在寻迹的过程中保守的曲线将有更好的避障能力，而贴近障碍物的轨迹因为的曲率降低在某种程度上可能会加速小车运行。

因此，我们需要权衡 inflation_radius 和 cost_scaling_factor 的相对大小以在保守与激进的路径规划中取得最优结果。

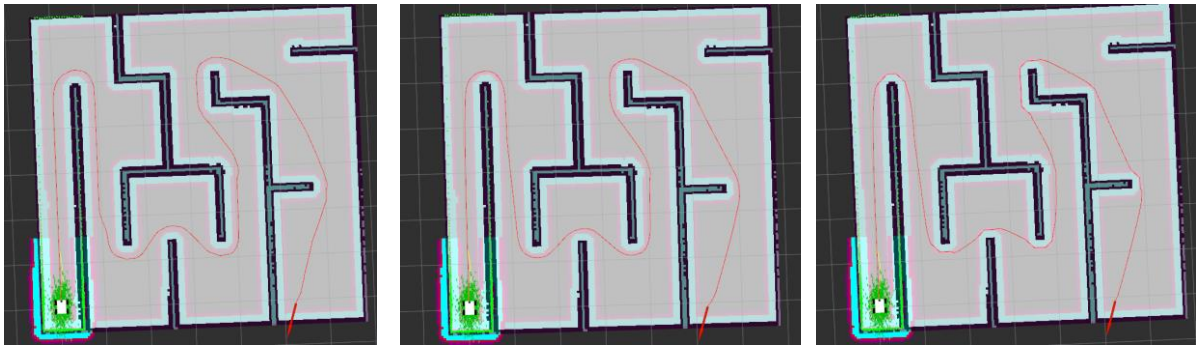
5.2 全局规划器数据分析

我们使用 GlobalPlanner 替代 navfn，因为其支持 A*算法、支持二次近似切换、支持切换风格路径，我们将 use_dijkstra 参数置为 true 以启动 A*算法。经过实验，cost_factor , neutral_cost , lethal_cost 这三个参数会对轨迹生成造成较大影响。根据源代码[1]，在 navfn 中代价值计算方式为：

$$cost = COST\_NEUTRAL + COST\_FACTOR * costmap\_cost\_value \quad (28)$$

传入 costmap 的代价值的范围为 0 到 255。源代码的注解中提到，当 COST_NEUTRAL 为 50 时， COST_FACTOR 需要设置为大约 0.8 以确保输入值均匀分布到 50 到 253 的输出范围内。如果 COST_FACTOR 设置为更高的值，则代价值将稳定到障碍物的高度，规划器会不重视狭窄走廊的宽度，不会沿着中心规划路径。

[1] https://github.com/ros-planning/navigation/blob/indigo-devel/navfn/include/navfn/navfn.h



(a)neutral_cost=50　　(b)neutral_cost=170　　(c)neutral_cost=240

图 5-3 不同 neutral_cost 对轨迹的影响

经过调试我们发现，cost_factor 设置得太低或太高都会降低路径的质量，neutral_cost 同理，如果值非常极端全局规划器甚至不会生成路径。图 5-3 展示了在 cost_factor 固定时，不同 neutral_cost 所造成的影响，可见 neutral_cost=170 左右时轨迹最佳。

5.3 局部规划器数据分析

5.3.1 dwa_local_planner 调试结果分析

dwa_local_planner 使用动态窗口 (Dynamic Window Approach, DWA) 算法。根据论文描述，DWA 的目标是产生一对 (v, ω) 对，表示最适合机器人局部条件的圆形轨迹。DWA 通过在下一个时间间隔中搜索速度空间来达到此目标。该空间中的速度被限制为允许的，这意味着机器人必须能够在到达由这些允许的速度决定的圆形轨迹上的最近障碍物之前停止。此外，DWA 将仅考虑动态窗口内的速度，动态窗口被定义为在给定当前平移和旋转速度和加速度的情况下在下一时间间隔内可到达的速度对的集合。DWA 最大化目标函数，其取决于（1）到目标的进展，（2）从障碍物的清除，以及（3）前向速度以产生最佳速度对[1] 。

经过实验我们发现，在其他参数合适的情况下，表 5-1 中所示 6 个参数对小车运行时间的影响最大，其影响过程如表中所述。

ROS Navigation Tuning Guide[1] 指出，如果将仿真时间 sim_time 的值设置为非常低（例如 ≤ 2.0）将导致性能下降，尤其是当机器人需要通过狭窄的门口、或家具之间的间隙时，这是因为没有足够多的时间进行计算以获得最佳轨迹以便通过狭窄的通道。另一方面，由于使用了 DWA 规划器，所有的轨迹都是简

单的圆弧，如果将仿真时间 sim_time 设置的非常高（≥5.0），将得到一条很长的缺乏灵活性能的曲线。而对于本次仿真赛，我们发现 sim_time 只能维持在 1s 以内，否则小车在过 U 型弯时将原地转向，造成时间的大量浪费。其原因可能是前向仿真时间过长导致 DWA 认为此处弯道的曲率很大，从而不断降低线速度加大角速度来“适应”此处的弯道。而当 sim_time 较小时，DWA 将无法“看到”完整的弯道，其规划出的圆弧曲率会比较大，让小车以更大的线速度运行。

表 5-1　dwa_local_planner 调试记录

max_vel_x	max_vel_theta	min_vel_theta	a_x	a_theta	sim_time	比赛用时
0.8	5.0	3.37	10.0	10.0	0.8	32.48
~	~	4.37	~	~	0.5	31.32
1.5	8.0	~	~	~	~	
2.5	~	5.5	~	~	~	撞墙
2.0	~	5.0	~	~	~	中途无法继续规划
1.8	~	4.5	~	~	~	无法继续规划
~	~	~	6.0	10.0	~	30.088
		~	4.0	12.0		28.032

注：~表示与数值上一行相同。

此外，path_distance_bias 与 goal_distance_bias 将极大影响轨迹的跟随效果，在 ROS 中，目标函数的计算公式如下：

$$\begin{aligned} cost \;=\; & path\_distance\_bias \\ & * (distance(m)\ to\ path\ from\ the\ endpoint\ of\ the\ trajectory) \\ & + goal\_distance\_bias \\ & * (distance(m)\ to\ local\ goal\ from\ the\ endpoint\ of\ the\ trajectory) \\ & + occdist\_scale \\ & * (maximum\ obstacle\ cost\ along\ the\ trajectory\ in\ obstacle\ cost \end{aligned} \quad (29)$$

path_distance_bias 决定局部规划器以多大权重与全局路径保持一致，较大值将使局部规划器倾向于跟踪全局路径。goal_distance_bias 衡量机器人无论走哪条路径应该以多大权重尝试到达目标点，决定跟踪局部路径的程度。occdist_scale

权衡机器人以多大的权重躲避障碍物，该值过大会导致机器人原地不动，因为静止将永远不会接触障碍物。由于本次线上赛的障碍物环境并不复杂，因此我们给与 path_distance_bias 更高的权重来让小车贴近于全局路径，从而获得更高的运行速度。

5.3.2 teb_local_planner 调试结果分析

如 4.3.7 中所述，新版车模的角加速度有限制，故在配置 Teb 的运动学参数时需要格外注意。我们假定此时已经拥有了合适的动态速度规划，仅调整 Teb 的参数。

在调试小车的过程中，调试教程我们主要参考了古月居的帖子[2]，文章中提到 weight_kinematics_turning_radius 是机器人最小转向半径的权重，越大则越容易达到最小转向半径的情况；weight_optimaltime 是最优时间权重，值越大那么车会在直道上快速加速，并且路径规划的也会切内道。此前 DWA 规划器中小车在 U 型弯处容易停车转向的问题通过这两个参数可以很好的解决。

但在实验的过程中，仍然会偶尔出现这种问题，为了从根本上解决，我们将 min_turning_radius，即最小转弯半径置为非零值，让小车拥有转弯半径，同时增大 weight_kinematics_turning_radius，可以解决小车停车转向的问题。但这也对速度有了较大的限制，倘若转弯速度调大，小车转弯灵活度下降容易接触外侧墙壁导致小车卡住。所以为了进一步提高转弯速度和灵活度，我们将转弯半径设置为较小的值，同时适当降低转向半径权重以减少转弯停车转向卡顿现象的发生。关于时间权重，增大该值小车加速更快，但也会导致小车速度变化幅度大，转弯时失误的概率增大，并且我们发现小车运行速度变化较大情况下的用时并不一定比平稳略低速度情况下的用时少，所以我们在保证小车速度不发生剧烈变化的情况下逐渐提高该值。同理，我们用类似的方式对 weight_max_vel_x 最大速度权重和 weight_max_vel_theta 最大角速度权重进行调试。

Teb 局部规划器的另一个问题是容易倒车，经过不断的调试和改进，我们直接将 weight_kinematics_forward_drive 调整到 1000 仍无法完全避免。但经过分析，倒车的根本原因是没有正确寻迹，如果小车的轨迹足够顺滑，则 Teb 就不会产生倒车行为。

为了提高小车运行顺滑度，我们适当减小了 min_obstacle_dist 与障碍物的最小期望距离。min_obstacle_dist 较大时会限制小车规划路径的范围，同时也极大

---

[2] https://www.guyuehome.com/9811

限制了小车的速度和灵活度。若该值较小，小车避障能力下降，再加上 amcl 定位存在的误差，小车容易撞墙卡住。所以我们适当减小该值，并且适当地增大 inflation_dist 障碍物周围缓冲区，不仅减缓转弯时速度冲击而且提高转弯灵活度。

我们考虑在最后 1/3 段路程以较大的速度冲向终点，但在经过最后一个转角时往往会冲向最后一个障碍物导致小车减速和方向调整。为了优化这一段的时间损耗，我们稍微减小 dynamic_obstacle_inflation_dist 动态障碍物的膨胀范围和略微增大 weight_viapoint 轨迹跟随力度，以此来减小最后一个障碍物对小车的影响，使小车能够在最后小弯道较好地转向冲向终点目标。

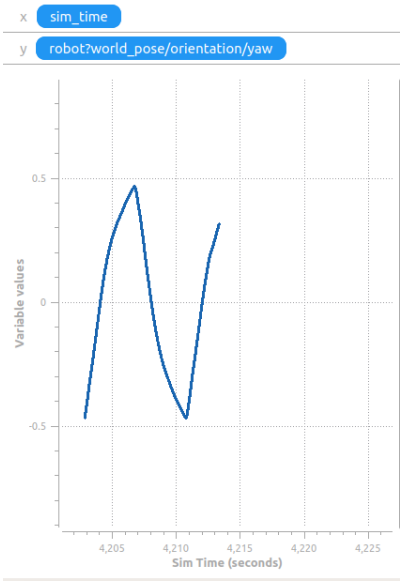
5.4 PID 控制器调试结果分析

众所周知，影响 PID 控制器性能最明显的三个参数是比例增益 Kp，积分增益 Ki、微分增益 Kd。相关资料表明[3]，比例增益的大小应视具体情况而定，比例增益太小，控制作用太弱，不利于系统克服扰动，余差太大，控制质量差；比例增益太大，控制作用太强，容易导致系统的稳定性变差，引发振荡；积分控制虽然能消除余差，但它存在着控制不及时的缺点。因为积分输出的累积是渐进的，其产生的控制作用总是落后于偏差的变化，不能及时有效地克服干扰的影响，难以使控制系统稳定下来；微分控制作用的特点是：动作迅速，具有超前调节功能，可有效改善被控对象有较大时间滞后的控制品质；但是它不能消除余差，尤其是对于恒定偏差输入时，根本就没有控制作用。根据 PID 控制器三种参数的特点，我们决定对 x 方向线速度作 PID 控制，对偏航角作 PD 控制。

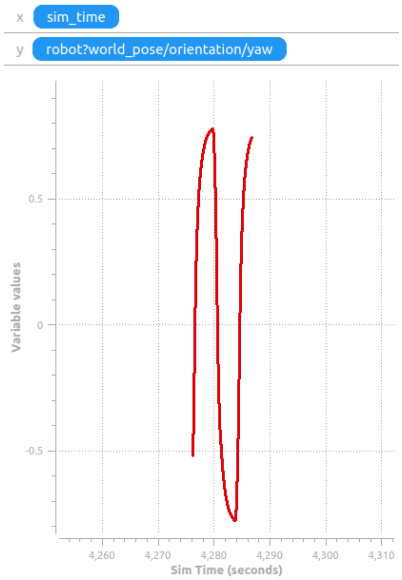
以偏航角调试为例，我们用 gazebo 自带的示波器观察了调节效果，如图 5-4 所示，不同的比例增益对控制过程有显著影响。当 Kp 过小时，调节时间很长，无法及时达到期望值；当 Kp 增大，调节过程开始加快；但如果 Kp 过大，系统则会出现振荡，如 5-4（d）所示；只有当参数适中时，才会拥有最佳的控制作用，能让系统即快速又平稳。

---

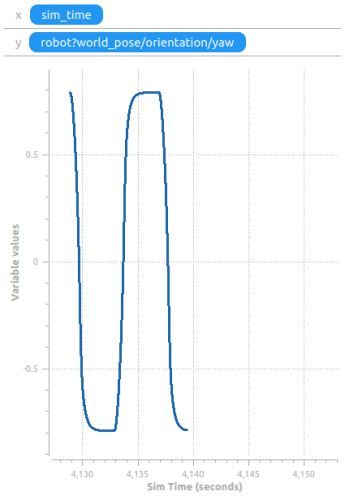
[3] https://baike.baidu.com/item/PID%E6%8E%A7%E5%88%B6%E5%99%A8/1888589?fr=aladdin



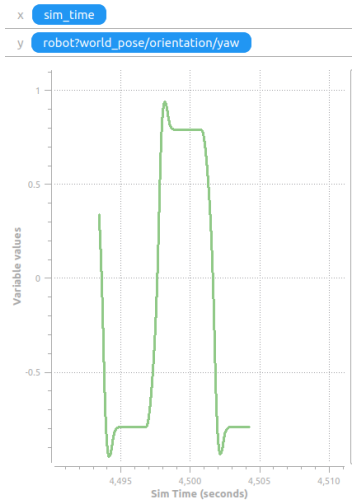
(a)Kp=0.5



(b)Kp=2



(c)Kp=4.5



(d)Kp=8

图 5-4 偏航角 PD 控制中比例系数的作用

第 6 章 项目作品总结

6.1 比赛感想

经过准备了两个月的智能车讯飞赛道的研究学习过程中，从比赛地图的构建再到小车导航的算法设计；从小车可以勉强跑进 30s，再到分析其可以改进的参数和优化的算法让小车得到提速。我们一步一步地走过来，从中学到了不少的知识，总结了不少的经验。

在每个阶段中，我们都尽量地将小车改进和完善，无论是组委会给的练习车模还是比赛的正规小车，我们都尽力将参数调到极致，不断分析思考可以改进的算法，努力达到最好效果。

在调试仿真车模的过程中，我们的小车曾经出现过很多问题，如小车在行驶的中途突然掉头向回走，终点处不能立刻减速，小车碰墙导致翻车，转弯出线速度和角速度参数的不合适导致转不过弯，或多或少拖慢了我们的进度，但经队员们不断的努力，和指导老师戴志涛老师的帮助，我们从控制算法和速度规划方案上做了一步步的改良和升级，渐渐对各模块有了更加深入的认识，对整体有了清晰的把握，我们在不断的前进的同时，小车也不断的加速前进。

6.2 不足与改进

从小车总体性能各因素考虑，小车改进分为建图、导航、控制三部分。

对于建图部分：我们小组分别比较了 gmapping、hector、cartographer 建出来的地图模型的对比，通过不断实践，选出在本次环境中构建出的相对准确性高的 hector。

关于导航部分：导航算法我们在 Dijkstra 和 A*算法中比较，由于它能够选择下一个被检查的节点时引入了已知的全局信息，对当前结点距离终点的距离作出估计，作为评价该节点处于最优路线上的可能性的量度，从而提高了搜索过程的效率，我们选定了 A*算法。而我们在单独使用局部规划器进行控制的后期，我们发现其规划的速度经常过大或过小，导致撞墙和转向过慢的问题。对此，我们决定采用动态速度规划的策略。动态速度规划的控制方案优化效果十分明显，能够将比赛用时从 28~30s 提升至 21~23s。

关于控制部分：当我们在使用相同的参数运行新版赛事包车模，出现了“在入弯时刹不住”、“推头”以及过度转向，轨迹跟踪效果差等很多问题。对于这系列问题，我们决定用 Python 手写一个全新的控制器。在得到小车实际位姿的反馈值后，使用 PID 算法对其进行控制，通过调整 Kp、Ki、Kd 等参数让系统输出最优控制量，使得问题迎刃而解。

在控制算法上，由于 teb 中的参数改进始终不能取得令人满意的成绩，我们决定另辟蹊径，实现自定义规划器。而方案二的 teb 算法中参数的改动有待进一步研究，突破瓶颈。

6.3 总结

本报告详细介绍了我们为智能车比赛讯飞赛道而准备为智能车系统方案。涉及导航、建图、控制等方案的设计，自定义规划器以及动态速度规划算法的实现。分析整个调试过程，我们在速度规划算法和 pid 控制上面有许多改进与创新。主要由以下几个方面：在地图中规定若干个关键点，小车经过这些关键点后自动改变 max_vel_x、max_vel_theta 等参数为最佳值，实现对局部规划器期望速度参数的强制限制。

使用 Python 语言手写了新的控制器来替换 teb_local_planner 使得新版车模惯性矩过大的问题得到优化，实现对偏航角、线速度的 PID 控制、并采用控制量额。

6.4 展望

随着互联网技术的进步和移动智能终端的普及，现今客人的用餐触点、方式等发生了很大的改变。于此同时，以“互联网+”模式的智慧餐饮逐渐崛起。通过自助点餐结账系统、服务呼叫系统、后厨互动系统、前台收银系统、预定排号系统以及信息管理系统等的智能化系统的应用，智慧餐厅可实现节约用工数量、降低经营成本、提升管理绩效等经营目标，是未来餐饮行业发展的重点和方向。

通过这个创意科技竞赛，它培养了我们勇于创新的动手能力、迎难而上的钻研精神，更激发了我们对智能机器人的兴趣，我们将会继续钻研机器人相关知识，励志为祖国智能机器人行业的发展贡献自己的力量。

在线下赛中，我们同样会用尽所学知识，继续保持这份追求知识的热情与执着，全力以赴、演绎精彩！

参考文献

[1] C. Parameters, G. Planner, and L. P. Selection, “Ros 导航功能调优指南 *,” pp. 1–18, 2019.

[2] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent, “Efficient sparse pose adjustment for 2D mapping,” *IEEE/RSJ 2010 Int. Conf. Intell. Robot. Syst. IROS 2010 - Conf. Proc.*, pp. 22–29, 2010, doi: 10.1109/IROS.2010.5649043.

[3] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram, “Trajectory modification considering dynamic constraints of autonomous robots,” *7th Ger. Conf. Robot. Robot. 2012*, pp. 74–79, 2012