# 第十五届全国大学生智能汽车竞赛
# 室外光电组（预赛）

# 技 术 报 告

学　　校：浙江科技学院

队伍名称：浙科室外光电队

参赛队员：刘力铭，严一展，余婉婷

带队教师：孙勇智，李津蓉

# 关于技术报告和研究论文使用授权的说明

本人完全了解第十五届全国大学生智能汽车竞赛关保留、使用技术报告和研究论文的规定，即：参赛作品著作权归参赛者本人，比赛组委会和赞助公司可以在相关主页上收录并公开参赛作品的设计方案、技术报告以及参赛模型车的视频、图像资料，并将相关内容编纂收录在组委会出版论文集中。

参赛队员签名：刘力铭

余婉婷 李一辰

带队教师签名：张雷 李清蓉

日　　期：2020.7.30

# 目　录

# 第1章 方案概述

全国大学生智能汽车竞赛以其“公平、公正、公开”的特点享有极高的声誉，深受大学生的欢迎。由于疫情影响，第十五届竞赛室外光电组的预赛采用线上仿真比赛形式，要求利用组委会统一发布的带有激光雷达、惯性测量单元（IMU）以及深度摄像头等传感器的仿真机器人，在 Gazebo 仿真环境中在特定地图上实现自主导航，同时能够对非特定环境（如临时出现的障碍物）的识别和自主处理（如避碰等）。

本报告主要介绍了浙江科技学院室外光电组智能车团队在准备竞赛期间所作的一些工作。包括传感器数据处理以及融合算法、全局导航算法、局部避障算法和策略优化等多个方面。仿真机器人采集激光雷达和 IMU 的信息，通过定位算法和导航算法，计算输出控制指令，控制电机速度和舵机转角完成自主导航和避障。要实现比赛任务要求，核心问题是解决机器人的定位问题和导航问题。

在系统定位方面，本报告使用 RF2O（Range Flow-based 2D Odometry）算法从激光雷达的数据变化中计算得到机器的运动数据，并使用扩展卡尔曼滤波（Extended Kalman Filter）融合惯性测量单元数据和 RF2O 算法计算结果得到初步定位数据，最后在此基础上使用自适应蒙特卡洛定位算法（Adaptive Monte Carlo Localization, AMCL）对定位数据进行修正，以提高测量精度。

为了测试算法的性能，首先在 Gazebo 仿真软件中进行仿真实验，定量分析了传感器数据通过不同算法处理后得到的误差。然后使用四轮移动机器人实体，在实际环境中进行了定位实验。仿真和实验结果表明：本报告所使用的传感器数据处理方案具有比较高的精度和稳定性，并且避免了误差累积的问题。另外，该数据处理方案计算量较小，并且不依赖里程计传感器，可以较好地完成定位任务。

在导航方面，本报告使用 A*算法作为全局路径规划算法，该算法根据终点坐标、地图数据和实时定位信息，计算全局导航路径。根据激光雷达测量的数据，使用基于 TEB（Timed Elastic bands）的局部规划算法，对全局导航路径进行优化，并对未知的障碍物进行躲避。

为了测试和优化导航算法的性能，同样在 Gazebo 软件中进行仿真实验，测试不同参数对局部规划路径的影响。然后使用参数优化后的 TEB 局部路径规划算法进行导航实验，仿真结果表明：本报告使用的导航算法可以有效躲避机器人周围的障碍物，并使机器人在较高的速度下完成导航任务。

# 第2章 问题描述

根据室外光电 ROS 组的预赛要求，仿真机器人需要在 Gazebo 仿真环境中根据统一提供的地图自主导航，避开锥桶、墙壁等障碍物，从赛道起点跑到终点，记录比赛完成时间作为比赛成绩。

为完成该比赛任务，首先需要实时定位算法以得知机器人的瞬时位姿以及速度、角速度等运动数据。然后，机器人需要全局导航算法，根据提供的地图数据，规划一条可以达到终点的路径。其次，机器人需要局部规划算法来躲避地图上没有记录的障碍物。最后，机器人将导航算法规划的运动指令发送到电机、舵机以完成导航任务。

因此，为完成比赛任务，需要解决机器人的定位问题、路径规划问题以及运动控制问题，其中，运动控制问题可以直接使用 Gazebo 中控制插件 JointPositionController 和 JointVelocityController 完成。所以，实时定位和路径规划是完成预赛要解决的核心问题。

## 2.1 机器人的实时定位问题

稳定可靠的定位是实现移动机器人在未知环境中自主移动的首要条件。目前，即时定位与地图构建（Simultaneous Localization and Mapping, SLAM）技术是解决移动机器人定位与环境感知问题的有效方法。SLAM 技术是指机器人系统在运动的过程中，通过自身搭载的传感器，完成对机器人运动数据和定位信息的计算以及对周围环境的感知。目前，对 SLAM 的研究主要集中在传感器的搭配和选择以及多传感器数据融合算法方面。

在数据融合算法方面，数据融合算法的研究方向最早是基于卡尔曼滤波（Kalman Filter, KF）的数据融合算法[1]，比如 EKF-SLAM。这些算法将非线性系统通过泰勒展开后保留一部分低次项来近似原系统，由于该近似方法比较简单，因此这些算法计算量通常较小，但由于忽略了泰勒展开式中的高阶项，算法误差较大，并且对于复杂度较高的非线性系统，滤波算法容易发散。所以，该算法适用于处理器性能较差并且对系统实时性要求较高的简单机器人系统中。

在上个世纪 90 年代，基于粒子滤波的 SLAM 算法被提出并引起重视[2]，这些算法使用粒子来模拟非线性系统，通过将粒子输入到非线性系统中，根据粒子的输出来近似原系统，因此这些算法可以精确的还原非线性系统。通常这些算法计算量较大，但是在以激光雷达为传

感器的机器人系统中表现出了更良好的稳定性和可靠性，如 Fast-SLAM[3]、RBPF-SLAM[4]算法。

本报告所使用的机器人需要在高速运动中完成导航任务，因此，系统定位算法需要比较高的实时性。综合考虑传感器融合算法的精度、计算量等性能要求，本报告选用扩展卡尔曼滤波算法进行多传感器数据融合。

在传感器选择方面，目前机器人定位最常用的解决方案是使用车轮式里程计和惯性测量单元测量机器人的运动数据，使用激光雷达传感器或视觉传感器测量周围的障碍物信息，通过多传感器数据融合算法完成定位和导航[5]。

但是，本组别的难点在于所使用的机器人模型因为机械结构的限制，无法安装车轮式里程计，也就无法直接测量机器人的线速度数据。为了解决这个问题，有国内外学者提出使用视觉里程计，通过 ORB-SLAM2[6]、VINS-Fusion[7]等图像处理算法，从摄像头采集到的图像中得到机器人的线速度、角速度等数据，从而完成定位任务。但是，这些方案在高速行驶的机器人系统中计算得到的速度、角速度数据不稳定，并且算法计算量大，对系统硬件要求高。所以，尽管仿真机器人提供摄像头传感器，但是本报告并没有使用摄像头参与数据融合。

除了使用摄像头计算机器人定位数据以外，有的学者提出了激光雷达和惯性测量单元相融合的导航定位系统，使用激光雷达提取环境特征和构建地图，然后用惯性测量单元采集的姿态信息，并通过卡尔曼滤波补偿位置和姿态输出的误差[8]。但是因为惯性测量单元中的加速度传感器测量噪声大，该数据通过积分得到机器人的位置信息是不可靠的[9]。

文献[10]提出了从激光雷达的数据变化中计算出机器人运动数据的 RF2O（Range Flow-based 2D Odometry）算法，该算法计算量小，使得仅使用激光雷达和惯性测量单元完成精确定位成为可能。但是测量得到的运动数据仍然存在一定的误差，需要结合其它传感器数据处理算法以提升测量精度。

综上，在机器人的定位方面，本报告在 RF2O 算法的基础上，使用扩展卡尔曼滤波融合激光雷达和惯性测量单元数据，并通过自适应蒙特卡洛定位算法修正后，完成机器人的定位任务。

## 2.2 机器人的自主导航问题

机器人的自主导航技术主要包括全局路径规划和局部避障两个主要方面。一方面，机器人需要根据环境地图寻找到达目标点的最佳的可通过路径。另一方面，机器人还需要对于路

径上动态出现的障碍物做出一定的反应，使得其即能够有效的避开导航过程中突然出现的障碍物，又能够快速地到达目标位置。

在全局路径规划算法方面，最著名的算法是荷兰计算机科学家 Dijkstra E W.提出的 Dijkstra 寻路算法，该算法有效地解决了带权有向图的最短路径问题[11]。他利用广度优先搜索技术来解决路径最佳路径搜寻问题，被广泛应用于路由、机器人路径搜寻等计算机领域。但是在稠密图中，由于节点和边的数量巨大，Dijkstra 算法的效率将会变得很低，实时性得不到保证，并且由于在求解过程中会生成大量的临时候选路径，内存消耗较大。

1968 年，Hart P E 等人提出路径规划中最为重要也是目前最流行的 A*寻路算法。A*既保持了 Dijkstra 算法的优点，使得路径规划时不仅能够尽量靠近初始点，还采用启发式搜索算法的优点，利用启发式评估函数使得路径搜索能够快速趋向目标点附近[12]。它在求出最优路径的同时，也提高了路径搜索的效率，目前被广泛应用于网络游戏、机器人导航等领域。

在局部避障技术中用的最多的是基于动态窗口（Dynamic Window Approach, DWA）算法，该算法根据移动机器人的运动学模型中的速度模型，首先在机器人的速度空间中按规则采样若干组不同的速度，然后根据机器人的速度运动模型计算在该速度下机器人的运动轨迹，最后根据障碍物以及环境地图信息对每条轨迹进行评价，以选择最佳通过路径[13]。但是该算法在评价障碍物时并没有考虑全局路径对于局部轨迹选择的影响，因此规划的路径虽然很好地避开了障碍物，但是路径往往不是最优的[14]。并且，本报告所使用的机器人的运动模型属于阿克曼转向模型，在转弯的过程中受到最小转弯半径的限制，而 DWA 算法在进行路径规划时，并没有做这方面的考虑，因此，并不适合在本报告的机器人系统上使用。

时间弹性带（Timed Elastic bands, TEB）是 C. Rösmann 等人于 2012 年提出的一种在线轨迹优化避障算法[15]。TEB 规划器通过创建一组从起始点到目标点的机器人姿态序列来跟踪和优化全局路径，可以使机器人的运动轨迹执行时间最小，并且保证与障碍物之间的安全距离，同时可以满足最大速度和加速度等运动学和动力学约束[16]。TEB 规划算法可以限定规划路径的最小转弯半径，从而更加适合解决基于阿克曼转向模型车辆的导航问题。

综上，在机器人的自主导航方面，本报告的机器人系统使用 A*算法作为全局路径规划器，使用 TEB 规划器进行局部路径优化和避障。导航算法根据目标点和途径的障碍物数据规划出路径和机器人的运动指令，从而完成导航任务。

# 第3章 技术方案

## 3.1 定位算法的技术方案

机器人的定位数据可以分为相对定位和绝对定位。相对定位是指基于惯性传感器、里程计等可以获取机器人运动数据的传感器，通过积分获得的机器人相对初始状态的姿态数据。由于传感器测量总是存在误差，在积分的过程中，定位数据的误差会随时间累积逐渐增大，因此相对定位数据在长时间运行的机器人系统中并不可靠；绝对定位则是指根据一些已知的参照信息，如地图数据，通过激光雷达等能够获取周围环境信息的传感器，计算出机器人相对参照目标的位姿数据。绝对定位数据不连续，存在跳变，可能存在比较大的瞬时误差， 但是，绝对定位数据不会有误差的累积问题，故适合作为长期参考。

考虑到机器人的相对定位和绝对定位数据各有优劣势，因此，可以设计一个特点互补的传感器数据处理方案如图 3-1 所示，来弥补各自的缺点。

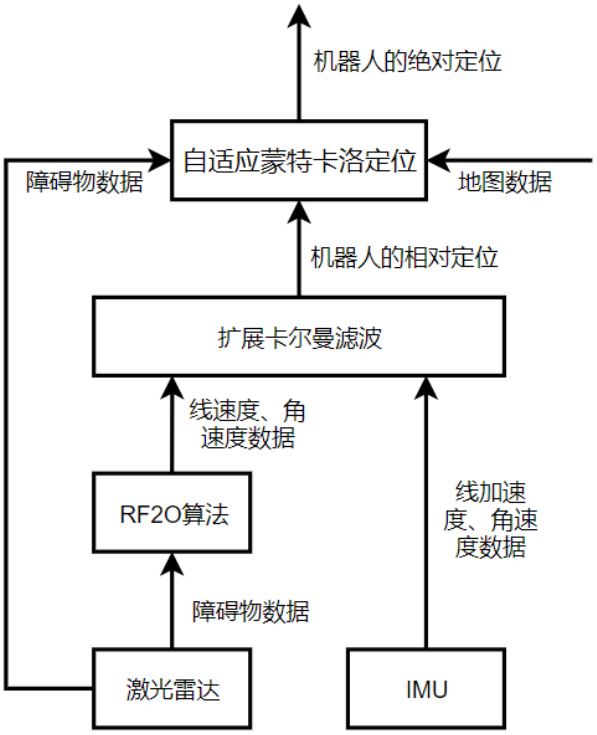


图 3-1 传感器数据处理方案框图

该方案使用惯性测量单元测量机器人的角速度，使用激光雷达获得机器人周围的障碍物信息。激光雷达信息通过 RF2O 算法处理得到机器人的线速度。使用扩展卡尔曼滤波将上述两种数据进行融合得到机器人的相对定位数据。但是，扩展卡尔曼滤波计算位姿数据仍然需

要积分，无法避免误差累积的问题。为了克服相对定位数据的误差累积问题，减少测量误差，进一步将激光雷达数据通过自适应蒙特卡洛定位算法来修正机器人的相对定位，得到系统的绝对定位数据。通过绝对定位数据可以得到比较准确并且没有误差累积的机器人位移、偏航角数据，而相对定位可以得到准确并且连续的机器人线速度、角速度、加速度数据。

### 3.1.1 扩展卡尔曼滤波的基本原理

扩展卡尔曼滤波是一种根据系统的状态估计和状态测量，对系统状态进行最优估计的算法[1]。

在机器人模型中，假设系统的真实状态与上一时刻的状态和当前系统输入有关，系统的测量状态与当前系统的真实状态有关，则建立系统的状态转换模型和传感器观测模型如下：

$$\begin{cases} x_k = f(x_{k-1}, u_k) + w_k \\ z_k = h(x_k) + v_k \end{cases} \tag{3-1}$$

式中，$k$ 表示离散时间；$x_k$ 表示在 $k$ 时刻下系统的真实状态；$u_k$ 表示输入矩阵；$w_k$ 表示过程噪声，假定符合 $w_k \sim N(0, Q_k)$ 的多元高斯噪声；$z_k$ 表示在 $k$ 时刻下的测量；$v_k$ 表示测量噪声，假定符合 $v_k \sim N(0, R_k)$ 的多元高斯噪声；$f(x,u)$、$h(x)$ 分别为非线性可微的状态转换函数和测量函数。

基于以上模型，扩展卡尔曼滤波求解分为两个阶段，分别为预测阶段和更新阶段。

(1)预测阶段：上一时刻的最佳估计 $\hat{x}_{k-1|k-1}$ 和协方差矩阵 $P_{k-1|k-1}$ 被用来计算当前时间的预测 $\hat{x}_{k|k-1}$ 和预测协方差矩阵 $P_{k|k-1}$。由于状态换函数 $f(x,u)$ 为非线性函数，需要将该函数线性化处理，得到雅克比矩阵 $F_k$。预测阶段求解公式如下：

$$\begin{cases} \hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k) \\ P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \end{cases} \tag{3-2}$$

(2)更新阶段：同样将测量函数 $h(x)$ 线性化得到测量矩阵 $H_k$。随后根据卡尔曼滤波的更新公式计算得到卡尔曼增益 $K_k$、当前时刻的最佳估计 $\hat{x}_{k|k}$ 和当前时刻的协方差矩阵 $P_{k|k}$，更新阶段公式如下：

$$\begin{cases} K_k = P_{k|k-1}H_k^T\left(H_k P_{k|k-1}H_k^T + R_k\right)^{-1} \\ \hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\left(z_k - h\left(\hat{x}_{k|k-1}\right)\right) \\ P_{k|k} = \left(I - K_k H_k\right)P_{k|k-1} \end{cases} \tag{3-3}$$

更新阶段计算出的 $\hat{x}_{k|k}$ 就是在 $k$ 时刻扩展卡尔曼滤波的估计结果。在 $k+1$ 时刻，该结果又是扩展卡尔曼滤波的输入，通过迭代，扩展卡尔曼滤波逐渐收敛，逐渐准确地估计动态系统的状态。

### 3.1.2 基于扩展卡尔曼滤波的多传感器融合

测量机器人的定位数据首先需要获得机器人的速度、角速度数据。本报告所使用的机器人系统安装了一个惯性测量单元和激光雷达传感器。惯性测量单元能够测量机器人的加速度和角速度，但是，测量得到的加速度数据噪声很大，通过加速度积分得到速度、位移数据是不可靠的，所以仅仅使用惯性测量单元进行定位是不可靠的。

为了解决该问题，本报告采用将 RF2O 算法和惯性测量单元数据相融合的方案。RF2O 算法能够基于激光雷达测量的环境数据近似地计算出激光雷达的速度、角速度数据[10]，该算法得到的激光雷达速度约束方程如下所示：

$$\begin{aligned} &\left(\cos\theta + \frac{R_\alpha k_\alpha \sin\theta}{r}\right)v_{x,s} + \left(\sin\theta - \frac{R_\alpha k_\alpha \cos\theta}{r}\right)v_{y,s} \\ &+\left(x\sin\theta - y\cos\theta - R_\alpha k_\alpha\right)\omega_s + R_t = 0 \end{aligned} \tag{3-4}$$

式中，$v_{x,s}$、$v_{y,s}$、$\omega_s$ 分别为机器人的 X 轴线速度、Y 轴线速度和角速度；$x$、$y$ 为机器人在 X 轴、Y 轴上的位移；$\theta$ 为激光雷达扫描到的某点与激光雷达的夹角，$R_\alpha$ 表示该点与激光雷达的距离变化对夹角的偏导，$R_t$ 表示该距离对时间的偏导；$k_\alpha$ 为常数，与激光雷达的性能有关。

根据公式(3-4)，可以直接从激光雷达的数据变化中计算出机器人的速度、角速度数据，并且速度数据没有误差累积问题。但是该公式的推导过程中舍去了泰勒展开后的高次项，并且公式推导建立在周围环境是静止的假设之上，因此需要将该公式计算出来的数据融合惯性测量单元的测量数据以得到更精准的结果。

基于扩展卡尔曼滤波的设计原理，需要建立移动机器人系统的状态转换模型和传感器观测模型。在状态转换模型方面，由于本报告的机器人系统仅考虑 2D 平面上的运动。根据状态矩阵中各个状态的物理意义，可以推导出各个状态的转换关系，式(3-1)中系统状态转换模型可以实现为式(3-5)所示：

$$x_k=\begin{bmatrix}X_k\\Y_k\\\theta_k\\\nu_k\\\varphi_k\\\omega_k\end{bmatrix}=\begin{bmatrix}0&0&0&\cos(\theta_{k-1})\cdot\Delta t&-\sin(\theta_{k-1})\cdot\Delta t&0\\0&0&0&\sin(\theta_{k-1})\cdot\Delta t&\cos(\theta_{k-1})\cdot\Delta t&0\\&&&0&0&\Delta t\\\vdots&\vdots&\vdots&1&0&0\\&&&0&1&0\\0&0&0&0&0&1\end{bmatrix}\begin{bmatrix}X_{k-1}\\Y_{k-1}\\\theta_{k-1}\\\nu_{k-1}\\\varphi_{k-1}\\\omega_{k-1}\end{bmatrix}+w_{k\text{-}1} \tag{3-5}$$

式中，$X$ 、$Y$ 分别为机器人在 X、Y 轴上的位移；$\theta$ 为机器人的偏航角；$\nu$ 、$\varphi$ 分别为机器人在 X、Y 轴上的速度分量；$\omega$ 为机器人的旋转角速度；

在传感器观测模型方面，方案使用惯性测量单元测量机器人的角速度，使用激光雷达通过 RF2O 算法计算出机器人的 X、Y 轴线速度。式(3-1)中的传感器观测模型可以表达为：

$$z_k=\begin{bmatrix}0&&&&&\\&0&&&&\\&&0&&&\\&&&1&&\\&&&&1&\\&&&&&1\end{bmatrix}\begin{bmatrix}X\\Y\\\theta\\\nu\\\varphi\\\omega\end{bmatrix}+v_k \tag{3-6}$$

本系统的状态转换模型和传感器测量模型都已经被推导出来。在机器人系统运行的过程中，传感器每次测量得到数据后，通过扩展卡尔曼滤波的预测公式(3-2)和更新公式(3-3)的不断迭代，逐渐准确地估计机器人的位姿。

### 3.1.3 定位结果的修正

由于扩展卡尔曼滤波本质上仍然是通过积分速度、角速度得到机器人的位移、偏航角数据，无法彻底解决误差累积问题，因此随着系统运行时间增加，误差累积增加，系统无法长期保持稳定。

自适应蒙特卡洛定位算法是一种估计机器人在地图中的姿态的定位方法[17]。AMCL 算法使用粒子来表示机器人可能的姿态，通过粒子集采样、权值运算以及重采样步骤逐渐丢弃与观测不符的粒子。最终，粒子会不断收敛，从而得到机器人在地图中实际的位姿数据。

AMCL 算法粒子群逐渐收敛的过程如图 3-2 所示。

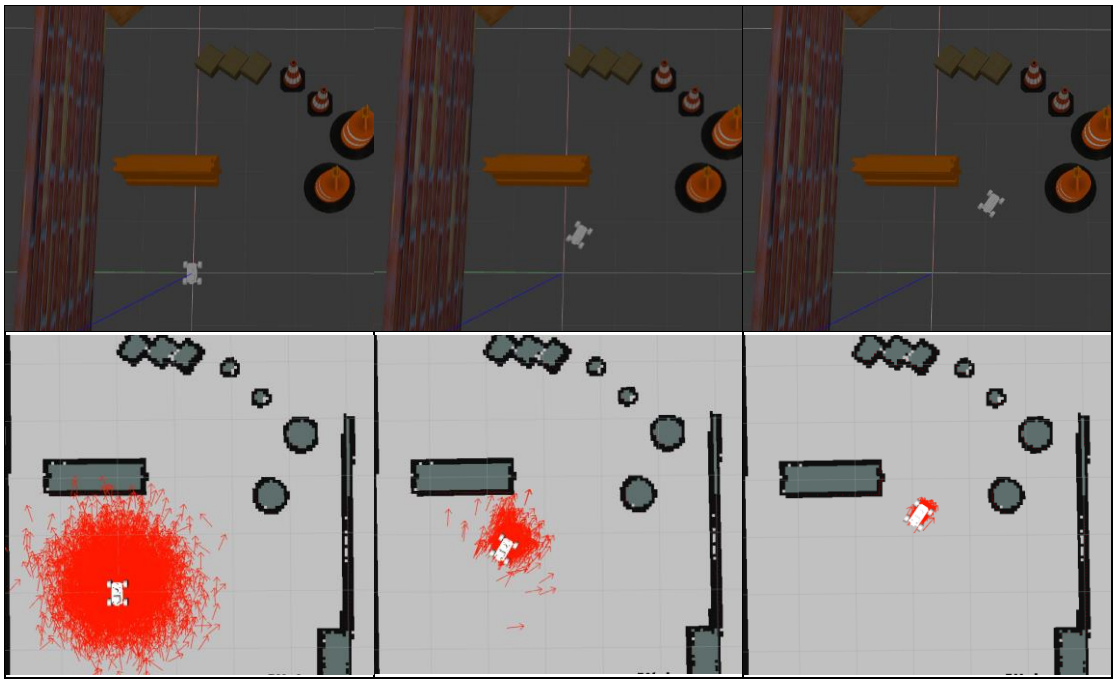

图 3-2 自适应蒙特卡洛定位算法粒子群逐渐收敛的过程图

使用 AMCL 算法计算得到的机器人定位数据是以地图数据为参考的，因此没有误差累积问题。但是，AMCL 算法会遇到绑架问题，即当机器人丢失了先前的位姿信息或者得到了一个错误的位姿信息，机器人就无法依靠 AMCL 算法继续计算当前自己在地图上的定位。

为了解决该问题，本报告使用扩展卡尔曼滤波和 AMCL 算法相结合的方案，即使用扩展卡尔曼滤波计算出机器人的定位数据后，再使用 AMCL 算法测量该定位数据的累积误差得到修正值，最后使用扩展卡尔曼滤波的计算结果减去修正值，即可以得到精度更高并且没有误差累积问题的定位数据。

扩展卡尔曼滤波可以减少误差累积对系统的影响，从而解决 AMCL 算法的绑架问题。此外，由于 AMCL 算法仅起到修正累积误差的作用，而累积误差在短时间内变化较小，因此修正值的更新频率可以远远低于扩展卡尔曼滤波算法，以此来减少系统定位的总体计算量。

## 3.2 系统导航的技术方案

根据比赛要求，本报告设计的系统导航方案如下：仿真机器人使用 A*全局规划算法在已知的地图上规划从起点到终点的可到达路径。然后使用基于 TEB 算法的局部规划算法完成局部导航路径的优化以及对未知障碍物的躲避。

### 3.2.1 基于 A*算法的全局路径规划

A*算法是一种启发式搜索算法，是在 Dijkstra 算法上的改进版本，它根据设置启发函数来对下一个可能的结点评估其代价，对代价高的结点进行过滤剔除，使得会朝着目标点靠近的方向进行搜索并且距离起点代价更低。

A*算法的评价函数如式(3-7)所示，使该算法既能够求得最优路径，又可通过启发式搜索的方式来节省时间，进而提高搜索的效率。

$$f(n)=g(n)+h(n) \tag{3-7}$$

该估值函数 $f(n)$ 由 $g(n)$ 和 $h(n)$ 两部分组成。其中 $g(n)$ 是起始点到当前位置的最小路径代价；从起始点 $S$ 到当前位置的最小路径代价由公式(3-8)给出：

$$g(n)=\begin{cases} dist(S,n) & parent(n)=S \\ g\left(parent(n)+dist\left(parent(n),n\right)\right) & parent(n)!=S \end{cases} \tag{3-8}$$

该公式的含义是：若起始位置 $S$ 是当前位置 $n$ 的父节点，则从起始位置移动到该位置的最小路径代价为起始位置到该节点的距离；若起始点 $S$ 不与当前节点相邻，则从起始点到该节点的路径代价为起始点到其父节点的路径代价与其父节点到该节点的路径代价之和。

$h(n)$ 代表目标点到当前位置的最小路径代价，$h(n)$ 可以通过多种距离度量方法计算，如哈曼顿距离，切尔夫距离以及欧几里得距离。

A*算法在执行中需要维护两个表，一个是 OpenList 表，代表可能会被考虑的方块；一个是 CloseList 表示将不会被考虑的方块；具体步骤如下：

1.从起始点 $S$ 开始，首先将起始点看作是待处理的方格加入到 OpenList 中，OpenList 列表中的格子是待检查的方格。

2.检查与起点 $S$ 相邻的所有方格，忽略代表障碍物的方格，将与起始点相邻的方格且可以到达的方格加入到 OpenList 表中，将起始点 $S$ 从 OpenList 删除加入到 CloseList 中。设置起始点 $S$ 为与其相邻节点的父节点。然后计算新加入 OpenList 表中每个节点的评估值。

3.选择具有最小评估值得节点记做 $A$，将节点 $A$ 添加到 CloseList 表中，同时从 OpenList 表中删除节点 $A$。

4.检查与 $A$ 相邻的所有方格节点，不考虑障碍物方格。如果该方格已经在 CloseList 中，则不考虑。如果不在 OpenList 中，则将该节点添加到 OpenList 中，并计算该方格评估值以

及其父节点评估值；如果该点已经存在 OpenList 中，则检查当前的评估值是否小于旧值，如果小于则更新其父节点及其评估值，如果不小于，则不更新。

5.当 OpenList 表中包含目标方格时，最短路径就被找到；如果没有遍历完所有方格，跳转到第三步，否则目标不可达。

### 3.2.2 TEB 局部路径规划原理

本报告采用局部路径规划算法为 TEB 算法，TEB 是基于图优化的局部路径规划算法，根据动态约束，如机器人的速度和加速度，将时间最小化来得到最短路径。

定义机器人的姿态为 $X_i=\left(x_i,y_i,\beta_i\right)^T$，$\left(x_i,y_i\right)$ 为机器人的位置坐标，$\beta_i$ 为方向信息。那么，机器人在时间 0 到 n 时的轨迹姿态可以表示为：

$$Q=\left\{X_i\right\}_{i=0\cdots n} \tag{3-9}$$

并且将两个姿态之间的关系用时间间隔 $\Delta T_i$ 来连接，可表示为：

$$\tau=\left\{\Delta T_i\right\}_{i=0\cdots n-1} \tag{3-10}$$

TEB 算法的关键思想是，将机器人姿态以及时间间隔定义为两个序列元组，根据实时加权的多个目标优化来对 TEB 进行调整与优化，从而求得最优的轨迹。首先，将各种约束，譬如速度与加速度、全局路径与障碍物、非完整约束的运动学以及运动时间，通过式(3-11)进行加权的多目标优化求和得到总体约束；由于 TEB 局部性，系统矩阵为稀疏矩阵，可将整体的目标函数转变为超图，以约束条件为边，构造图优化问题，采用的 g2o 求解库进行优化，计算出最优解 $B^*$。

$$\begin{cases} B:=\left(Q,\tau\right) \\ f\left(B\right)=\sum_k \gamma_k f_k\left(B\right) \\ B^*=\min_B f\left(B\right) \end{cases} \tag{3-11}$$

式中，$B$ 为时间弹性带的描述，包含了导航过程中机器人姿态以及姿态之间的关系；$\gamma_k$ 为多个目标函数的权值；$f_k\left(B\right)$ 是考虑多个约束条件的目标函数；$B^*$ 为解算得到的最优轨迹。得到最优轨迹后，算法根据两个连续组态 $x_i$，$x_{i+1}$ 的欧式距离以及两个位姿间平移时

间间隔 $\Delta T_i$，求出系统的平均线速度以及角速度，并把其当作运动控制命令下发至移动机器人。

$$\begin{cases} v_i \approx \dfrac{1}{\Delta T_i}\left\| \begin{pmatrix} x_{i+1} & -x_i \\ y_{i+1} & -y_i \end{pmatrix} \right\| \\ \omega_i \approx \dfrac{\beta_{i+1}-\beta_i}{\Delta T_i} \end{cases} \tag{3-12}$$

进行新一轮迭代时，每次初始化时再次对全局的路径和周围环境新的障碍物的信息进行检测，确保该算法的实时性。

在本报告所使用的机器人系统中，公式(3-11)的约束条件还需要考虑到机器人的最大速度、不同速度下的最大转弯半径、过弯角速度等运动模型的限制，而这些限制是由仿真机器人的模型参数决定的。通过仿真实验得到仿真机器人的各项基本运动参数如表 3-1 所示。

表 3-1 仿真机器人的各项基本运动参数

| 速度(m/s) | 1.0 | 1.5 | 2.0 | 2.5 |
|---|---|---|---|---|
| 转弯半径(m) | 0.65 | 0.85 | 1.05 | 1.125 |
| 角速度(r/s) | 1~1.7 | 1~2.4 | 1~2.9 | 1~3.5 |
| 理论角速度(r/s) | 1.538 | 1.765 | 1.905 | 2.222 |
| 轴距(m) | 0.335 | | | |
| 最大舵机转角 | 45° | | | |

### 3.2.3 机器人运动学模型

本报告所使用的机器人符合阿克曼转向模型。机器人运动时，角速度受到舵机转角和电机速度的影响，因此，导航算法输出的运动指令需要经过阿克曼转向模型换算后才能得到相应的舵机转角和电机速度指令。

阿克曼转向模型如图 3-3 所示，$\delta_o$ 和 $\delta_i$ 分别为外侧前轮和内侧前轮偏角，当车辆右转时，右前轮为内侧轮胎，其转角 $\delta_i$ 较左前轮胎转角 $\delta_o$ 更大。$l_w$ 为轮距，$L$ 为轴距，后轮两轮胎转角始终为 0。

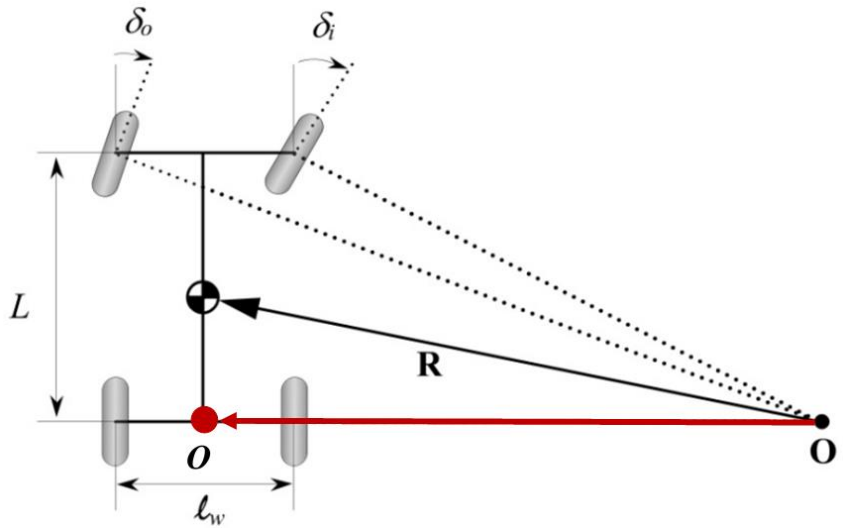


图 3-3 阿克曼转向模型示意图

根据几何关系，机器人运动的转向半径 $R$ 和机器人的线速度 $v$ 、角速度 $\omega$ 的关系为：

$$R = \frac{v}{\omega} \tag{3-13}$$

根据阿克曼转向模型，机器人的舵机转角 $\delta$ 与转向半径的关系可以表示为：

$$\delta = \frac{\delta_o + \delta_i}{2} \cong \frac{L}{R} \tag{3-14}$$

根据以上两式子可以得到舵机转角与机器人线速度、角度的关系式如式(3-15)所示。

$$\delta \cong \frac{L\omega}{v} \tag{3-15}$$

同时，为帮助车模过弯，机器人的左右轮应该具有一定的差速，左轮速度 $v_l$ 和右轮速度 $v_r$ 的推导式子如下所示：

$$\begin{cases} v = \dfrac{v_l + v_r}{2} \\ v_l = v\left(1 + \dfrac{\tan\delta \cdot l_w}{2L}\right) \\ v_l = v\left(1 - \dfrac{\tan\delta \cdot l_w}{2L}\right) \end{cases} \tag{3-16}$$

根据式(3-15)和式(3-16)，可以将导航算法规划得到的期望速度、角速度指令换算为舵机转角、左右电机速度指令，通过 Gazebo 控制插件，控制仿真机器人的电机和舵机，从而完成运动控制和导航。

# 第4章 方案实现

机器人操作系统（Robot Operating System, ROS）是机器人项目开发的开源软件平台。ROS 本身并不是实时操作系统，但它可以提供许多类似操作系统的服务，例如进程之间的通讯，常用传感器的硬件驱动以及程序包管理等功能。

ROS 的设计理念是减少机器人系统开发过程中的代码重复使用、重复开发的现象。在此理念之上，ROS 的功能模块都是以功能包（Packages）的形式存在，开发者通过下载需要的开源功能包可以有效减少代码复用率，降低开发成本。

ROS 同时还集成了很多工具，这些工具可以使得开发工作变得更加简单。其中，Gazebo 是基于强大物理引擎的仿真软件，可以精确的模拟出复杂的机器人工作环境并测试机器人系统算法的可靠性；Rviz 是数据可视化工具，它通过订阅机器人系统的实时数据，使得开发者能够及时观察机器人系统的运行状态；rqt 上集成了多种开发工具，包括数据示波器 rqt_plot，负责消息记录的 console 等等；

在机器人的定位设计上，ROS 提供了很多已经完成的软件包，如 tf 软件包提供了机器人系统的坐标变换、slam_gmapping 软件包提供了基于激光雷达的开源 SLAM 算法实现、navigation 软件包提供了定位、导航等功能。合理使用这些软件包可以大大减少开发难度。

## 4.1 ROS 节点整体设计

分布式计算是 ROS 操作系统的一个最显著的特征。在 ROS 中，节点是表示完成一定功能的进程。在此基础上，ROS 提供了话题（Topics）和服务（Services）使得节点与节点之间能够互相通讯。

### 4.1.1 定位算法的 ROS 节点设计

本报告的定位部分 ROS 节点通讯简化图如图 4-1 所示，其中，使用椭圆形代表运行的节点，使用矩形代表话题。/gazebo 为 Gazebo 仿真软件发布的节点，Gazebo 根据机器人的 urdf 文件中的相关描述在仿真环境中加载机器人模型和传感器插件，并根据仿真机器人的实时状态和仿真环境输出惯性测量单元数据话题/imu 和激光雷达数据话题/scan。/map_server 为地图数据节点，该节点通过加载现有的地图文件，发布地图数据话题/map。 /rf2o_node、/ekf_node 和/amcl_node 则分别是第 3 章中提到的 RF2O 算法、扩展卡尔曼滤波算法以及自

适应蒙特卡洛定位算法的软件实现。它们订阅传感器数据的话题，通过程序计算，得到机器人系统的定位数据。

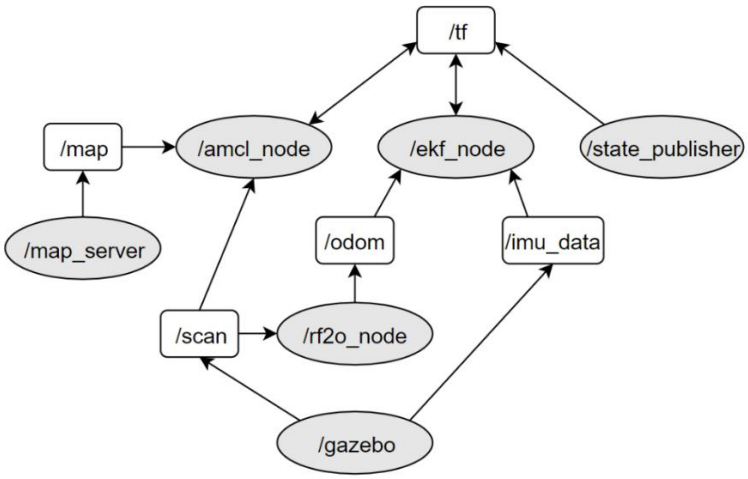


图 4-1 定位部分 ROS 节点与话题通讯关系简化图

### 4.1.2 导航部分的 ROS 节点设计

本报告的导航部分 ROS 节点通讯简化图如图 4-2 所示。/move_base 为机器人系统的导航算法节点，该节点会订阅/map 话题以得到先验的地图数据和/tf 话题以得到 EKF 算法和 AMCL 算法计算得到的机器人定位数据，然后根据 TEB 算法规划机器人的运动路径并发布机器人的运动控制指令。/cmd_to_ackermann 节点会订阅/cmd_vel 并根据机器人的阿克曼转向模型计算得到机器人的期望舵机转角和期望线速度。最后，/servo_commands 通过订阅/cmd_ackermann 控制 Gazebo 软件中的仿真机器人完成相应的运动指令，从而完成运动控制和导航。同时，为了方便调试，/keyboard_teleop 节点会根据键盘指令直接控制机器人运动。

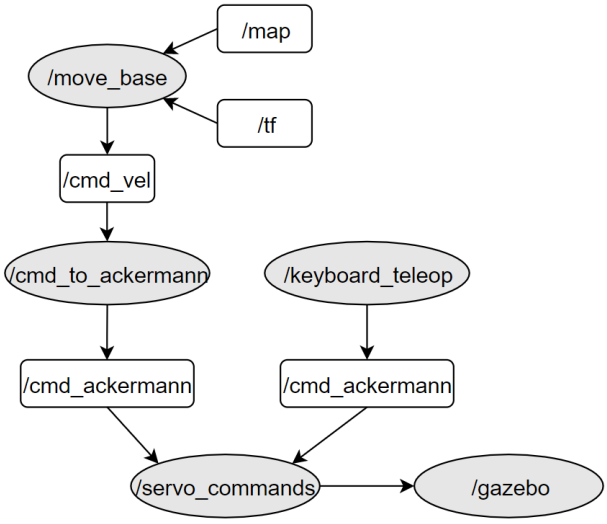


图 4-2 导航部分 ROS 节点与话题通讯关系简化图

/tf 话题则是用来保存机器人系统中的各个坐标结构。/ekf_node 和/amcl_node 节点通过该话题更新机器人系统的相对定位以及绝对定位数据。下面具体介绍本报告的坐标系统设置。

## 4.2 坐标节点设计

在复杂的机器人系统中，会涉及到的复杂的坐标系转化关系。ROS 提供了 TF 功能来高效管理各个坐标系。在本报告中，移动机器人简化模型以及各个坐标系定义如图 4-3 所示。

base_link 为机器人坐标系，该坐标系一般以机器人系统的重心为原点。

laser_link 和 imu_link 分别为激光雷达坐标系和惯性测量单元坐标系，它们也都是以传感器的重心为原点，并且由于机器人的机械结构固定，它们与 base_link 坐标系之间的转换关系是固定不变的。激光雷达和惯性测量单元测量到的传感器数据需要通过坐标轴变换转换到 base_link 上才可以计算使用。

odom 坐标系为里程计坐标系，base_link 坐标系相对于 odom 坐标系的姿态即为机器人的相对定位。该坐标通常由线速度以及角速度传感器测量，通过积分计算得到，会存在误差累积的问题，故不能成为长时间的参考。但是 odom 坐标系变化连续，在短时间内准度可以保证，故该坐标系下的速度、角速度数据可以作为运动控制的反馈数据。

map 坐标系为世界坐标系，base_link 坐标系相对于 odom 坐标系的姿态即为机器人的绝对定位。该坐标通常是 odom 坐标系通过定位算法计算得到。map 坐标系会随时间变化发生跳变，但是不存在误差累积问题，故适合作为长期参考。解决移动机器人定位问题本质上就是得到 map 坐标系与 base_link 坐标系之间的变换关系。

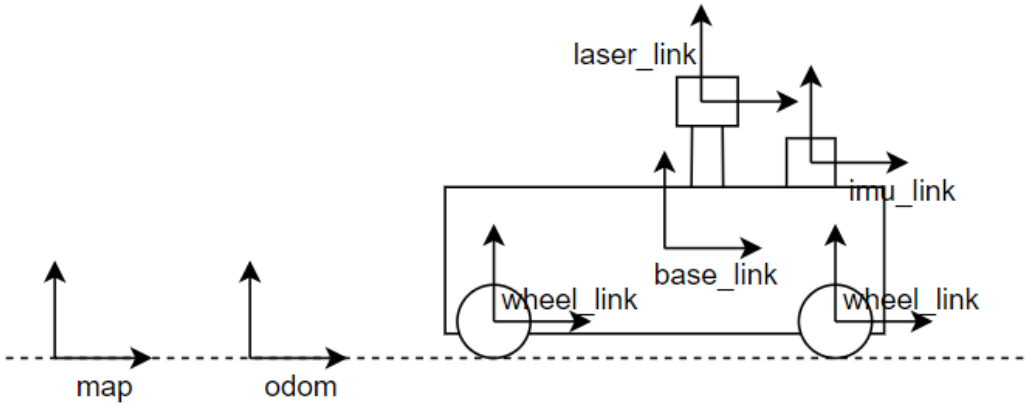


图 4-3 机器人坐标系框架图

在 TF 功能包中，各个坐标系都按照树形结构连接。本报告的各个坐标系之间的简化关

系如图 4-4 所示。其中，各个坐标系的转换关系如下：由于机械结构的关系，base_link、imu_link 以及 laser_link 坐标系之间的转换关系固定不变，传感器测量的数据都会转换到 base_link 坐标系下，以方便统一计算；odom 坐标系由惯性测量单元和激光雷达数据通过卡尔曼滤波算法计算得到；map 坐标系在 odom 坐标系的基础上通过自适应蒙特卡洛定位算法计算得到。得益于该树形连接关系，任意两个坐标系的转换关系可以根据其在树形图上的连接关系迭代计算得出。

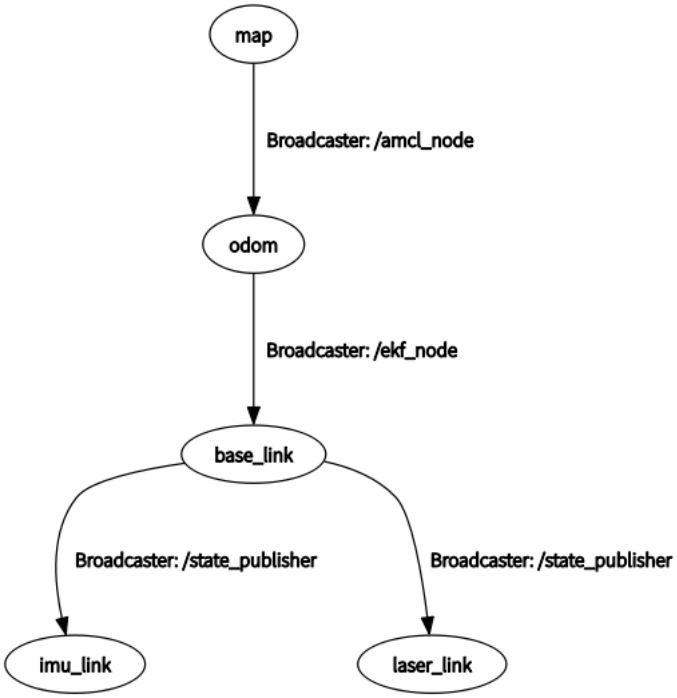


图 4-4 各个坐标系的树形简化关系图

# 第5章 测试分析

## 5.1 定位算法的测试分析

### 5.1.1 RF2O 算法的仿真实验

在现实生活中，由于传感器的误差总是存在的，测量出机器人运动数据的真实值是不可能实现的，因此，也就无法在现实的环境下定量地测量算法的性能。本报告使用机器人仿真平台 Gazebo 搭建仿真环境，并在该仿真软件中测试 RF2O 的性能。

在 RF2O 算法仿真中，给仿真机器人的激光雷达传感器配置和 LS01G 激光雷达传感器一样的参数，并在激光雷达的每一个样本的测量上增加一个服从 $N(0,0.01)$ 的高斯噪声。分别在机器人静止、直线和旋转三种情况下，计算测量的均方根误差（Root Mean Squared Error, RMSE）来体现测量的精度。其中，RMSE 的计算公式为：

$$RMSE(X,h)=\sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(x_i)-y_i\right)^2} \tag{5-1}$$

在静止实验中，控制机器人的线速度与角速度为零，保持 30 秒，记录机器人的运动数据。由于机械结构上的限制，机器人只能产生 X 轴的线速度以及 Z 轴的角速度，因此仅记录这两组数据。静态误差测量数据如图 5-1 和图 5-2 所示，算法在 X 轴线速度的误差平均值为-0.000134，均方根误差为 0.01384；在 Z 轴角速度的静态误差平均值为 0.000157，均方根误差为 0.0165。、

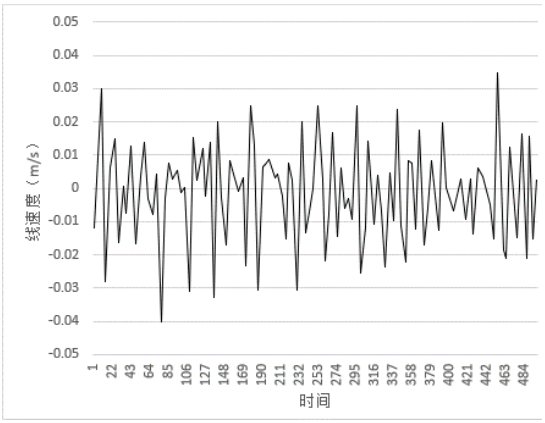


图 5-1 X 轴线速度静态噪声测量图

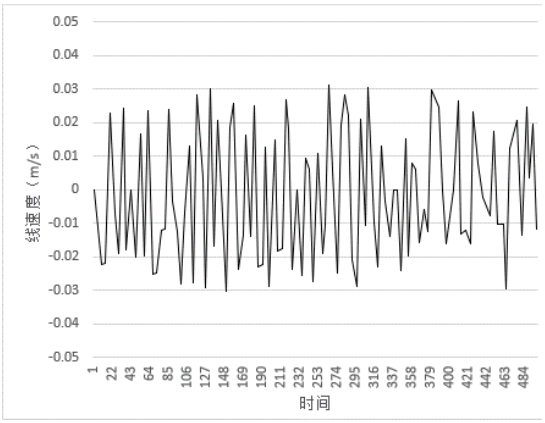


图 5-2 Z 轴角速度静态噪声测量图

在直行实验中，仅仅控制机器人缓慢地加速直行，保持 30 秒，记录机器人在 X 轴上的

线速度并通过积分得到位移数据。线速度测量数据如图 5-3 所示，积分后得到的位移数据如图 5-4 所示。算法在 X 轴线速度的动态误差平均值为-0.00092，均方根误差为 0.04。从测量结果可以看出，RF2O 算法在机器人运动速度较小时，测量误差比较小，当机器人运动线速度大于 0.25m/s 时，测量的误差变大。对于位移的计算来说，线速度计算位移是积分的过程，虽然大致上测量值与真实值保持一致，但误差随时间逐渐增大。

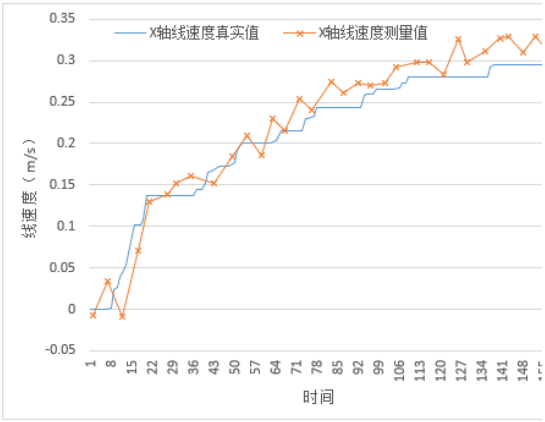


图 5-3 X 轴线速度动态测量图

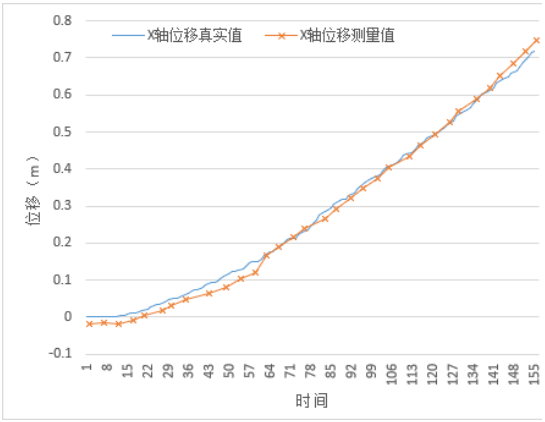


图 5-4 积分得到 X 轴位移动态测量图

在旋转实验中，仅仅控制机器人定轴加速旋转，保持 30 秒，记录机器人在 Z 轴上的角速度并通过积分得到偏航角数据。角速度测量数据如图 5-5 所示，积分后得到的偏航角数据如图 5-6 所示。算法在 Z 轴角速度的动态误差平均值为-0.0028，均方根误差为 0.039。从测量结果可以看出，RF2O 算法在机器人角速度较小时，测量误差比较小，当机器人运动线速度大于 0.25r/s 时，测量的误差变大。对于偏航角的计算来说，同样误差随时间逐渐增大。

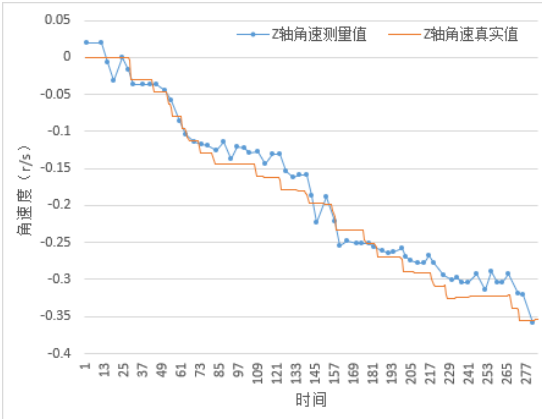


图 5-5 Z 轴角速度动态测量图

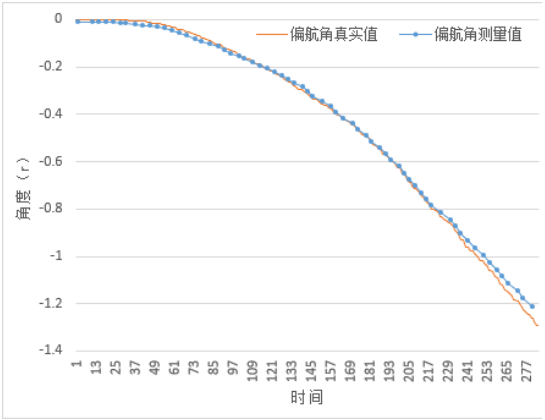


图 5-6 积分得到 Z 轴偏航角动态测量图

综上，得出 RF2O 算法在不同实验中的测量误差如表 5-1 所示。RF2O 算法在三个场景下的误差平均值都无限接近于零，说明测量漂移较小。RF2O 算法在直行和旋转的情况下测量噪声相差不大，但都远远大于静态时的测量噪声。并且，随着运动速度的增大，测量误差

呈现增加的趋势。尽管如此，RF2O 的测量噪声依然在可以接受的范围之内。由于偏航角和位移的计算都是积分的过程，误差随时间逐渐增大，故仅使用该算法无法完成机器人准确定位任务。

表 5-1 RF2O 算法在不同场景下的测量误差表

| | 静态 | 直行 | 旋转 |
|---|---|---|---|
| 误差平均值 | -0.000134 | 0.00092 | -0.0028 |
| 均方根误差 | 0.01384 | 0.04 | 0.039 |

### 5.1.2 扩展卡尔曼滤波的仿真实验

为了定性比较扩展卡尔曼滤波算法的效果，在 Gazebo 仿真软件中，建立室内仿真场景并在室内安放多种不同形状的障碍物。仿真场景图和该仿真场景的栅格地图如图 5-7 和图 5-8 所示。为了保证仿真的结果更加符合现实，RF2O 算法和扩展卡尔曼滤波算法的工作频率都设置为 10Hz，同时，在惯性测量单元和激光雷达的每一次样本的测量上都增加一个服从正态分布的噪声。

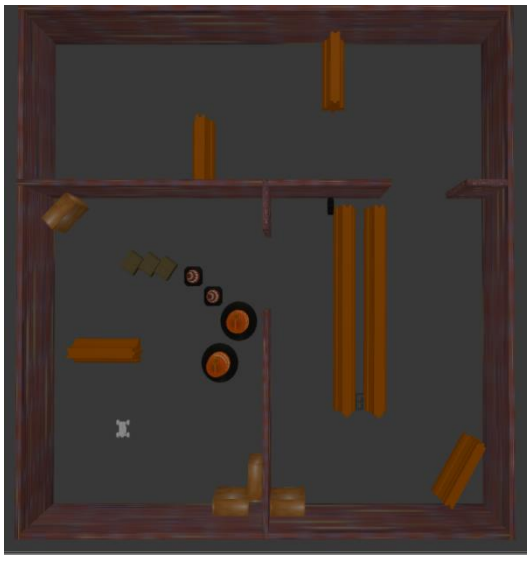

图 5-7 室内仿真场景图

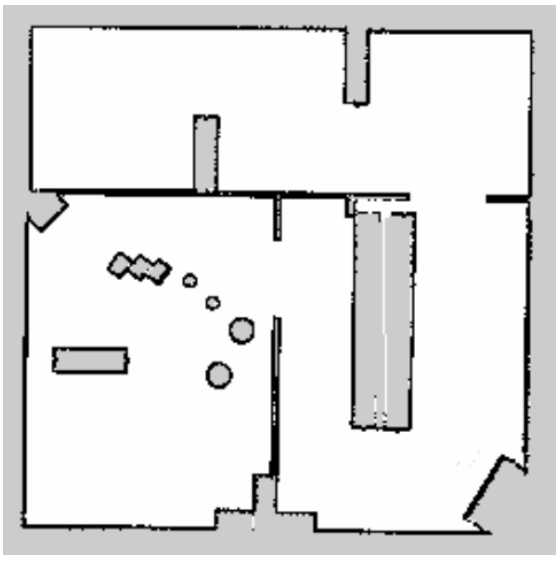

图 5-8 仿真场景的栅格地图

为测试扩展卡尔曼滤波的效果，机器人以场景的左下角作为出发点，并运动到场景的左上角作为终点。在机器人系统运动的过程中，分别记录机器人位姿的真实值（红色曲线）、RF2O 算法的计算值（蓝色曲线）以及扩展卡尔曼滤波对多传感器数据融合后的姿态数据（绿色曲线）后描绘轨迹如图 5-9 所示，并进行比对。为方便描述，将场景房间分为一号，二号，

三号房间。

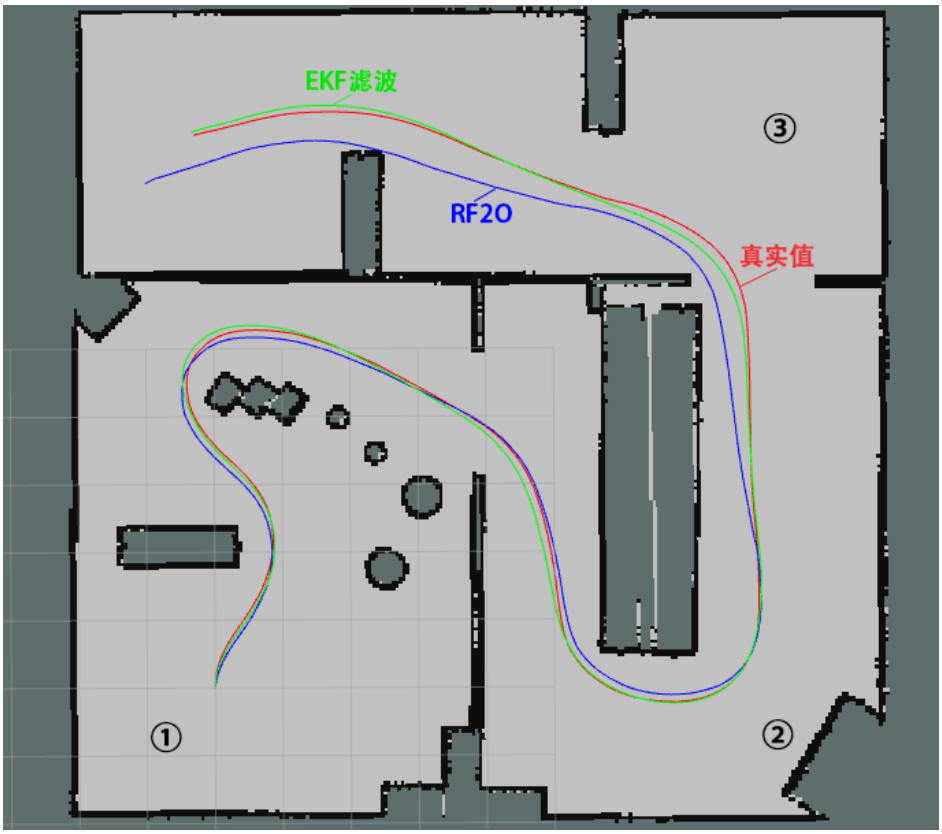


图 5-9 单次仿真中，不同数据处理方法得到的机器人运动轨迹图

从轨迹图中，可以看出：通过 RF2O 算法计算的轨迹曲线（蓝色曲线）在二号房间的后半段以及三号房间由于误差的累积逐渐偏离实际曲线（红色曲线）。在机器人运动到终点后，RF2O 计算得到的 X 轴位移测量误差为 0.84 米，Y 轴位移测量误差为-0.74 米。因此，单独使用 RF2O 算法无法完成定位任务。而扩展卡尔曼滤波算法通过融合 RF2O 算法的计算结果以及惯性测量单元的测量数据，轨迹曲线（绿色曲线）能够较好地贴合实际曲线。尽管如此，扩展卡尔曼滤波数据在二号房间和三号房间入口的转弯处都与真实数据存在明显偏差。在机器人运动的终点，扩展卡尔曼滤波计算得到的 X 轴位移测量误差仅为-0.07 米，Y 轴位移测量误差为 0.1 米。综上，扩展卡尔曼滤波能够减少误差累积对系统的影响，提高测量精度。

### 5.1.3 自适应蒙特卡洛定位算的仿真实验

为了定性比较自适应蒙特卡洛定位算法的效果，测试实验与扩展卡尔曼滤波的测试实验步骤相同。在程序上，增加 AMCL 算法，该算法通过修正扩展卡尔曼滤波算法计算得到的相对定位数据计算出机器人的绝对定位。扩展卡尔曼滤波算法的工作频率设置为 10Hz，AMCL 算法的工作频率设置为 0.2Hz。在运行的过程中分别记录扩展卡尔曼滤波与 AMCL

算法得到的定位误差。

本报告采用均方根误差（RMSE）来描述位姿数据测量的误差[18]。假设 $P_1,\cdots,P_n$ 为预测的姿态数据序列；$Q_i,\cdots,Q_n$ 为真实的姿态数据序列；$\Delta$ 为每秒的数据间隔，即每秒钟预测的姿态数据个数，本次实验 $\Delta$=10 。则均方根误差的计算公式如下所示：

$$RMSE\left(E_{1:n},\Delta\right):=\left(\frac{1}{n-\Delta}\sum_{i=1}^{n-\Delta}\left\|\left(Q_i^{-1}Q_{i+\Delta}\right)^{-1}\left(P_i^{-1}P_{i+\Delta}\right)\right\|\right)^{1/2} \tag{5-2}$$

采集到的定位数据所计算得到的误差如表 5-2 所示。可以得出：本报告使用的传感器数据处理方案可以有效降低 X 轴和 Y 轴的位移测量误差，最大测量偏差不超过 7.4 厘米。但是由于仿真机器人系统使用的 IMU 传感器测量噪声较小，使得扩展卡尔曼滤波算法融合 RF2O 算法计算结果得到的偏航角误差较小，引入 AMCL 算法反而增大了偏航角的测量误差，但是该误差仍然较小，最大测量偏差不超过 1.6° 。

表 5-2 不同定位方案的测量误差表

| | 仅使用 RF2O 算法 | 使用 EKF 融合多传感器数据 | 使用 AMCL 修正 EKF 计算结果 |
|---|---|---|---|
| 均方根误差（X 轴位移） | 0.315175295 | 0.372654996 | 0.140620729 |
| 均方根误差（Y 轴位移） | 4.695963809 | 0.490947403 | 0.29658343 |
| 均方根误差（偏航角） | 20.3348107 | 0.117389029 | 0.307936455 |
| 最大测量偏差（X 轴位移） | 0.845049 米 | 0.21802 米 | 0.057 米 |
| 最大测量偏差（Y 轴位移） | 0.814297111 米 | 0.185093 米 | 0.074 米 |
| 最大测量偏差（偏航角） | 3.11243124° | 0.884777255° | -1.59° |

### 5.1.4 实际环境下的定位实验

为了验证在仿真实验中得到的结果，本节使用真正的机器人在现实环境中完成定位任务。但由于无法测量出定位数据的真实值，也就无法对测量结果进行定量比较。因此，在本节通过定位数据以及激光雷达测量的障碍物数据绘制地图来定性地比较不同定位方案的性能。同时，由于机器人每次运行的运动轨迹、以及传感器数据都不相同，为避免这些差异对实验结果的影响，本节使用 rosbag 工具记录机器人某次运动时的传感器数据以及数据发布的时间。在运动完成后，依据时间顺序对数据进行回放，再根据各个定位方案，计算出定位数据并完

成建图。

在本次实验中，机器人途经的路径大约有 60 米长，平均速度为 0.5 米/秒，并且除自适应蒙特卡洛定位算法的工作频率为 0.2Hz 以外，其他各算法的工作频率都设置为 10Hz。不同定位方案得到的机器人运动轨迹以及建立的地图如图 5-10 所示。

左侧地图为参考建图。图中的定位数据是由目前普遍使用的基于粒子滤波的 SLAM 算法融合 RF2O 算法结果和惯性测量单元数据产生，最符合真实情况，但该定位方案的计算量最大，算法运行一次的平均耗时为 0.305 秒。

中间地图的定位数据源自于 RF2O 算法，可见 RF2O 算法计算得到的角速度误差较大，导致转弯处的地图出现弯曲、形变，算法单次平均耗时为 0.018 秒。

右侧地图中的定位数据则是来自于本报告使用的传感器数据处理方案，得到的定位数据所建立的地图非常接近参考地图，证明数据误差较小。此外，该方案的计算量也远远小于基于粒子滤波的参考定位方案。程序运行初期，方案平均耗时为 0.068 秒。随着 AMCL 算法逐渐收敛，方案的总计算量逐渐减少，在系统运行 10 秒后，方案平均耗时逐渐稳定，为 0.025 秒。

实验结果表明，本报告使用的传感器数据处理方案具有测量误差较小、计算量较小的优点，较好地解决了机器人的定位问题。

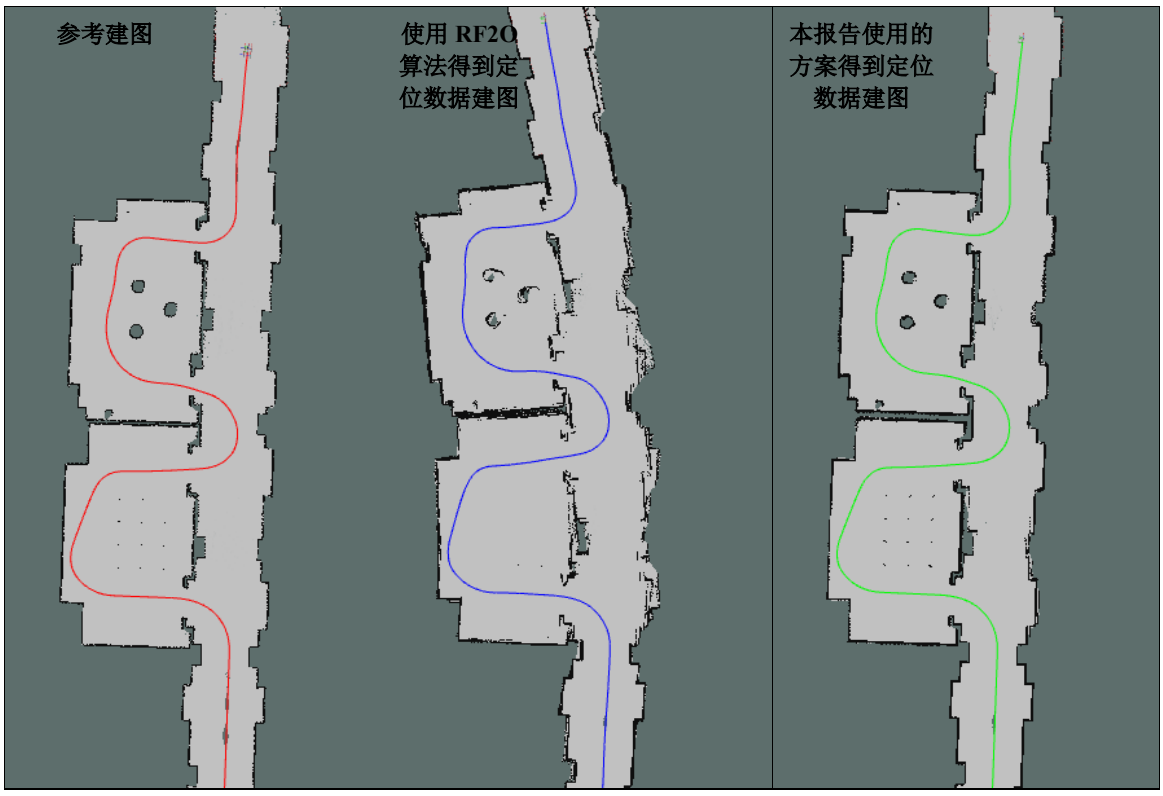


图 5-10 不同定位方案估计轨迹以及地图对比图

## 5.2 导航算法的测试分析

### 5.2.1 全局路径规划测试

为了验证 A*算法的可靠性，在图 5-7 所示的仿真场景下，将仿真机器人放置于场景的左下角，并设置左上角为导航终点。A*导航算法会首先接收地图数据，并对地图上障碍物进行膨胀，即以地图上的每个障碍物为圆心，膨胀一个固定半径。然后，将机器人模型简化为一个二维的长方形，那么，要使机器人不要接触障碍物只需要保证该二维长方形不会接触地图上的障碍物区域以及膨胀区域。最后导航算法接收导航终点数据和机器人的定位数据，使用 A*算法得到全局路径规划如下图 5-11 所示。

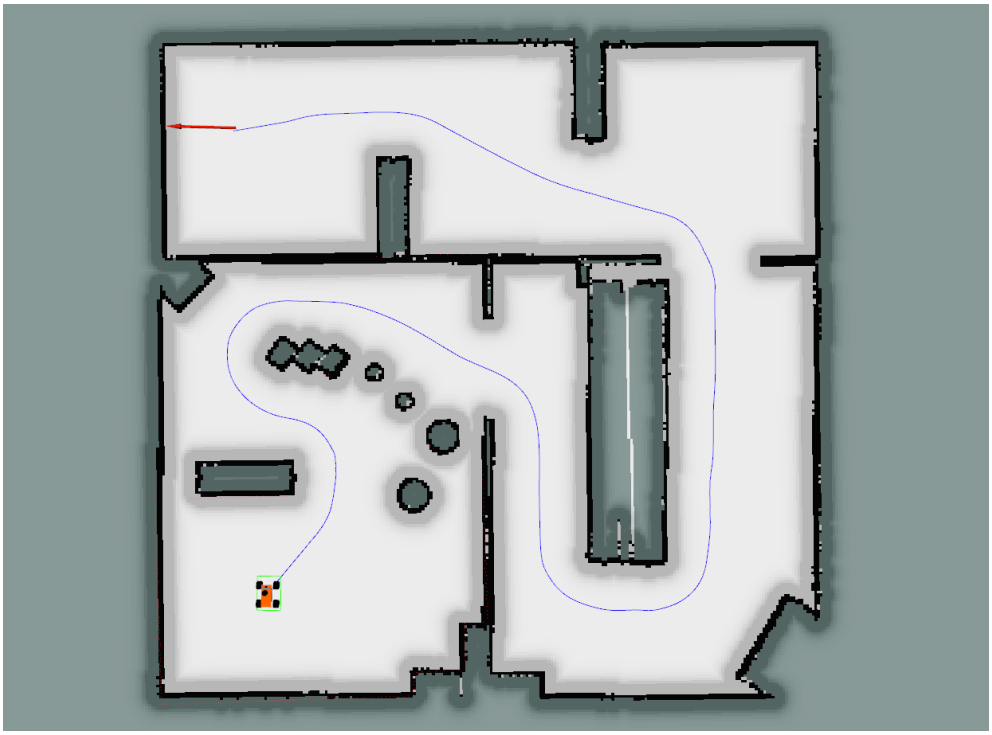

图 5-11 基于 A*算法的全局路径规划演示图

### 5.2.2 TEB 算法的参数调整

为了更为清晰地了解 TEB 算法工作的原理和参数优化，首先在 Rviz 可视化工具中设定三个小正方形为障碍物，围绕障碍物的深色圆圈是移动机器人到障碍物的安全距离。利用 TEB 算法实现起点到目标点的局部路径规划。同时，使用 rqt 工具箱中的 rqt_reconfigure 工具可以实现相应参数的动态调整和优化。在默认参数下，TEB 算法规划得到的路径如图 5-12 所示。

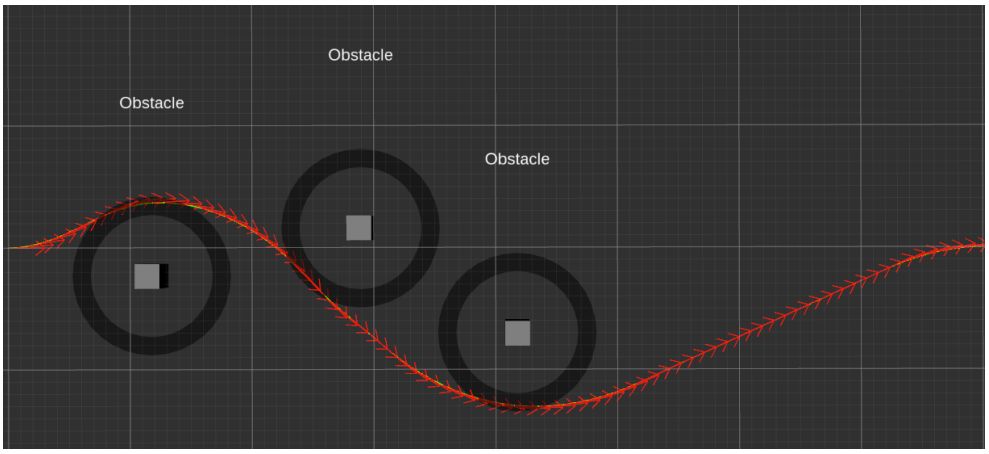


图 5-12 默认参数下，TEB 算法的规划结果示意图

增大 TEB 算法的 min_obstacle_dist 参数，即增大机器人模型与地图障碍物的最小距离，得到规划轨迹如图 5-13 所示。对比图 5-12 可以看出，增大移动机器人与障碍物之间的最小距离，可以提高移动机器人运行的安全性，但是也会导致移动机器人移动速度降低和旋转角度增大，对于具有转弯半径限制的阿克曼转向移动机器人而言会增加规划的执行时间，且在狭窄通道环境情况下导致陷入局部最小。

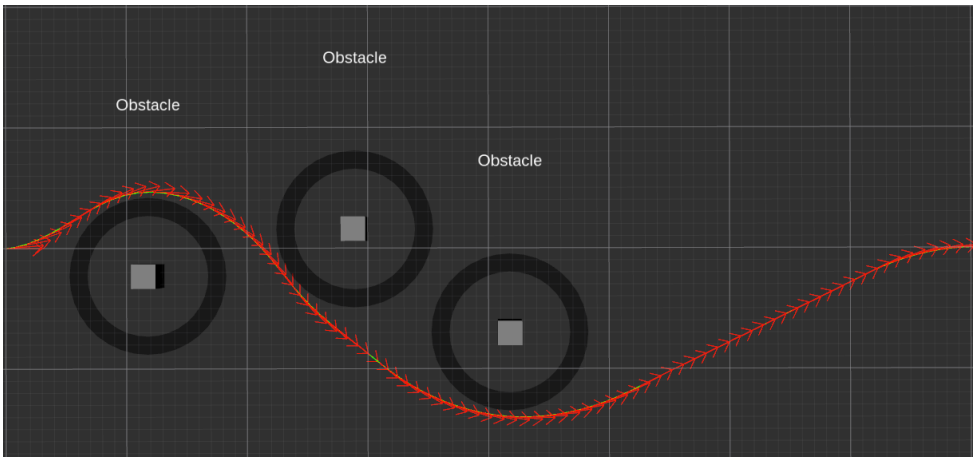


图 5-13 增大 min_obstacle_dist 后，TEB 算法的规划结果示意图

增加 TEB 算法的 weight_shortest_path 参数，即增加 TEB 在路径规划时对规划路径的最短距离的权重。参数调整前与调整后的弯道路径规划曲线如图 5-14 和图 5-15 所示。对比可以看出，增加最小路径的权重可以优化机器人过弯的路径，减小过弯的路程。同时，优化的过弯曲线可以增大过弯的旋转半径，机器人可以以更快的速度过弯。但是，该参数增大了移动机器人与障碍物碰撞的概率。

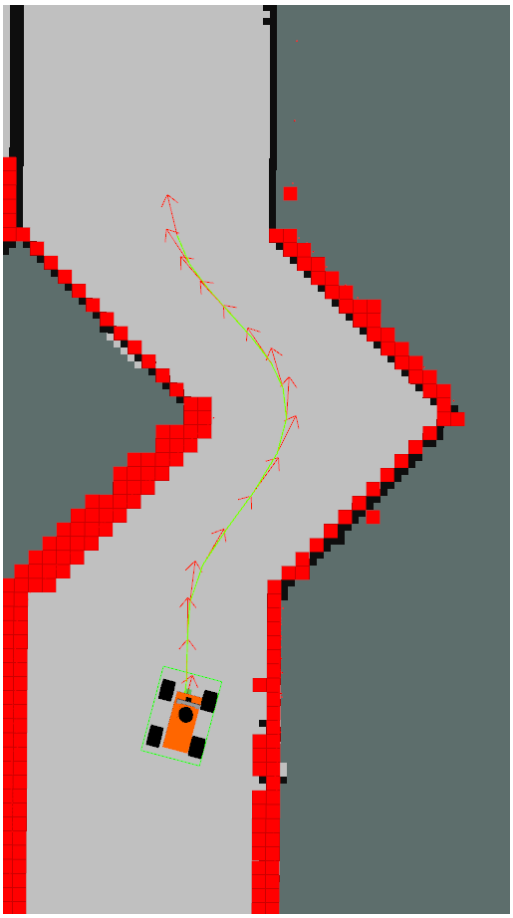

图 5-14 参数优化前的转弯路径规划图

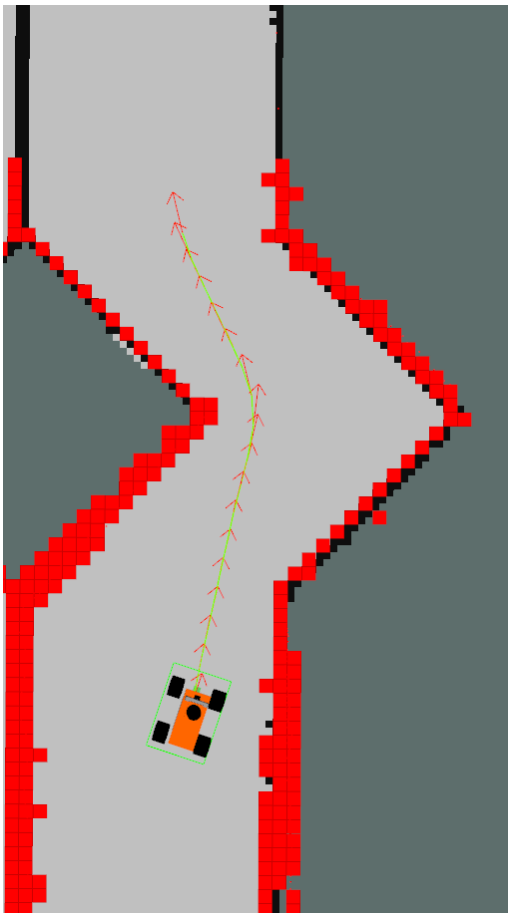

图 5-15 参数优化后的转弯路径规划图

### 5.2.3 导航算法测试

本节实验使用 TEB 局部路径规划对机器人进行导航配置，通过 Gazebo 仿真建立地图模型，优化固定开始点与目标点姿态之间的轨迹，通过 rqt_reconfigure 确定轨迹的最优参数。其中，仿真环境和定位算法配置与定位算法的上文仿真实验相同。并且，在仿真环境中加入地图上没有记录的临时障碍物。

本实验对 TEB 局部该规划的参数设置有，期望的轨迹时间分辨率 0.3，指定考虑优化的全局计划子集的最大长度为 0.3，每个采样间隔的姿态可行性分析数为 5，与障碍的最小期望距离为 0.4m，障碍物周围缓冲区的距离为 0.6m，在每个内循环迭代中调用的实际求解器迭代次数为 5，在每个外循环迭代中调用的实际求解器迭代次数 4，为硬约束近似的惩罚函数添加一个小的安全范围是 0.1m，满足最大允许平移速度的优化权重为 2，满足最大允许角速度的优化权重为 1，满足最大允许平移加速度与角速度的优化权重均为 1，保持与障碍物的最小距离的优化权重为 50，考虑到的不同轨迹的最大数量为 4，目标位置的允许距离误差为 0.2m，目标位置的允许角度误差为 0.1。

通过以上参数进行配置，在仿真环境中设立目标点，其规划过程轨迹如图 5-16 所示，图中，蓝色曲线为 A*算法规划的全局路径，红色的方向线段为 TEB 算法规划的局部路径。可以看出，局部规划路径大致上与全局规划一致，但是，当激光雷达扫描到地图上没有记录的障碍物时，局部规划器会做出避障动作。

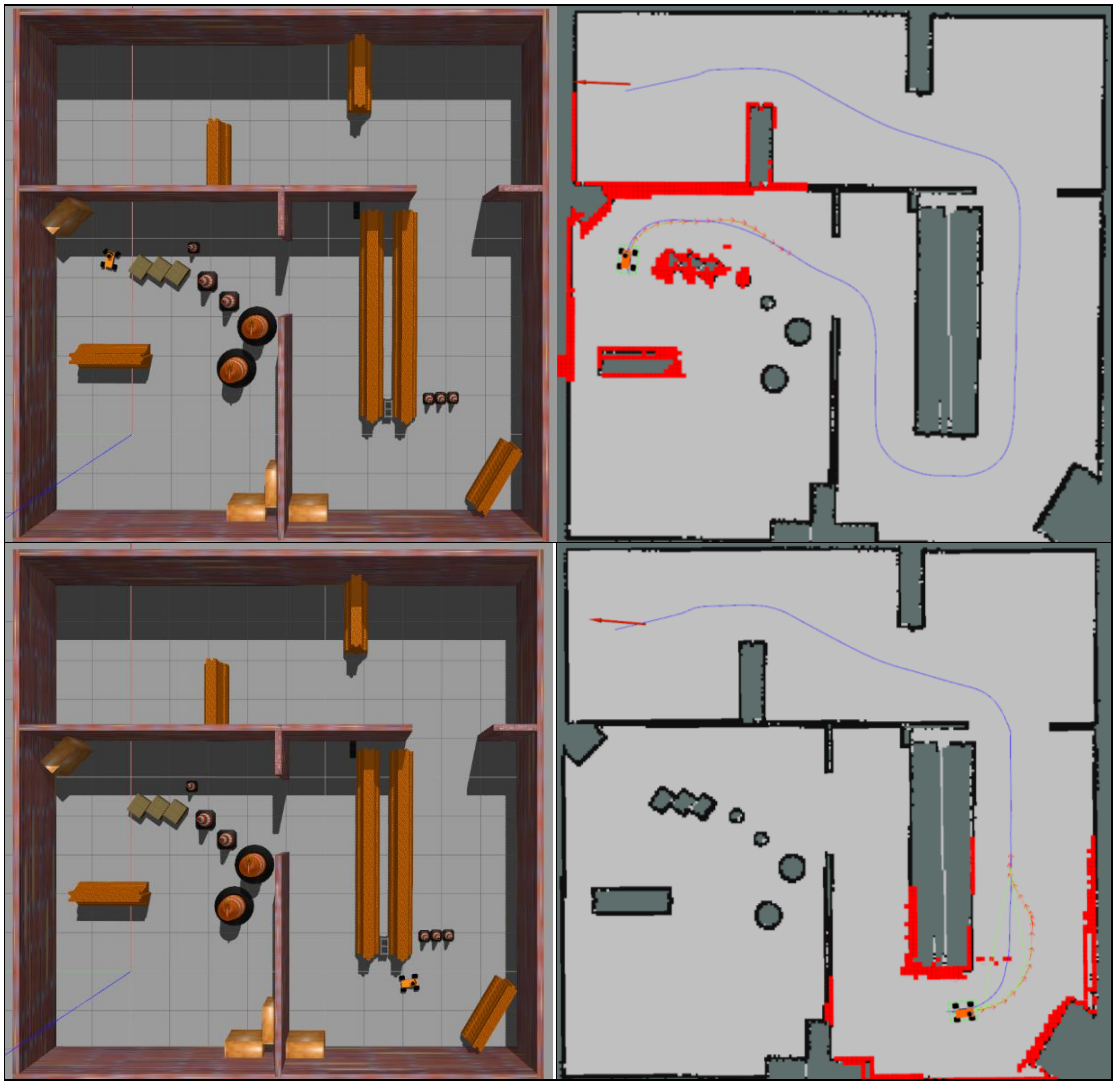

图 5-16 仿真路径规划过程图

仿真结果表明，TEB 算法能够实时规划局部机器人运动路径轨迹，同时也能够对全局路径进行跟踪，所获得的轨迹适合基于阿克曼转向模型的移动机器人运动。

# 第6章 作品总结

本报告详细介绍了本团队为第十五届全国大学生智能汽车竞赛而准备的仿真智能车系统方案，涉及传感器数据处理以及融合算法、全局导航算法、局部避障算法和策略优化等多个方面。分析整个机器人系统，系统主要有以下特色：

1）针对传感器的数据处理问题，本报告阐述了所使用的 RF2O 算法、扩展卡尔曼滤波算法以及自适应蒙特卡洛定位算法的基本原理。根据机器人模型和传感器数据特点决定采用以下方案进行传感器数据处理：使用 RF2O 算法从激光雷达的数据变化中计算得到机器的运动数据，并使用扩展卡尔曼滤波融合惯性测量单元数据和 RF2O 算法计算结果得到机器人系统的相对定位数据，最后在此基础上使用自适应蒙特卡洛定位算法对定位数据进行修正，以得到精度更高的绝对定位数据。

为测试定位算法的性能，首先在 Gazebo 仿真软件中进行仿真实验，定量计算了传感器数据通过不同算法处理后得到的误差。然后设计了室内四轮移动机器人实体，并在实际环境中进行了定位实验。实验数据表明：1.算法得到的定位数据精度较高，在仿真实验中位移以及偏航角误差控制在 7.4cm 以及 1.6° 之内，并且方案得到的定位数据没有误差累积问题，稳定性高。该仿真结果与实际环境下的定位实验结果一致。2.定位方案总体计算量较小，单次运行所消耗的时间远远小于目前普遍使用的基于粒子滤波的 SLAM 算法。

2）在导航算法方面，本报告使用 A*全局规划算法在已知的地图上规划从起点到终点的可到达路径。然后使用基于 TEB 算法的局部规划算法完成局部导航路径的优化以及对未知障碍物的躲避。

为了测试和优化导航算法的性能，同样在 Gazebo 软件中进行仿真导航实验，仿真结果表明：本报告使用的导航算法可以有效躲避机器人周围的障碍物，并使机器人在较高的速度下完成导航任务。

但是纵观本报告的整个系统方案设计，还是有几个可以改进的地方：

1）定位数据不够准确，通过 RF2O 算法计算机器人的速度数据存在一定的近似和误差。因此，可以通过安装车轮式编码器得到更加准确的速度数据。

2）局部路径规划算法仍然存在缺陷，TEB 局部规划算法为了保证导航的安全性，算法规划出来的过弯速度较慢，不够极限。因此，可以自己编写路径跟踪算法，从而以最大的速度完成导航。

# 参考文献


[1] Kalman R E. A new approach to linear filtering and prediction problems[J]. 1960.

[2] 李辉. 基于激光雷达的 2D-SLAM 的研究[D].浙江工业大学,2017.

[3] 王古超. 基于 ROS 的全向移动机器人系统设计与研究[D].安徽理工大学,2019.

[4] He H, Jia Y H, Sun L. Simultaneous Location and Map Construction Based on RBPF-SLAM Algorithm[C]//2018 Chinese Control And Decision Conference (CCDC). IEEE, 2018: 4907-4910.

[5] Zhi L, Xuesong M. Navigation and control system of mobile robot based on ROS[C]//2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC). IEEE, 2018: 368-372.

[6] Mur-Artal R, Tardós J D. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras[J]. IEEE Transactions on Robotics, 2017, 33(5): 1255-1262.

[7] Qin T, Shen S. Online temporal calibration for monocular visual-inertial systems[C]//2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2018: 3662-3669.

[8] 严小意,郭杭. 激光 SLAM 移动机器人室内定位研究[J]. 测绘通报,2019(12):8-11.

[9] 杜少林,陈书钊,陈鹏光,田亚芳,陈剑鸣. MEMS 加速度计噪声分析与降噪方法研究[J]. 传感器与微系统,2018,37(07):45-48+55.

[10] Jaimez M, Monroy J G, Gonzalez-Jimenez J. Planar odometry from a radial laser scanner. A range flow-based approach[C]//2016 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2016: 4479-4485.

[11] Dijkstra E W. A note on two problems in connexion with graphs[J]. Numerische mathematik, 1959, 1(1): 269-271.

[12] Hart P E, Nilsson N J, Raphael B. A formal basis for the heuristic determination of minimum cost paths[J]. IEEE transactions on Systems Science and Cybernetics, 1968, 4(2): 100-107.

[13] Fox D, Burgard W, Thrun S. The dynamic window approach to collision avoidance[J]. IEEE Robotics & Automation Magazine, 1997, 4(1): 23-33.

[14] 熊安. 基于 SLAM 的轮式机器人定位与导航技术研究[D].中国科学院大学(中国科学院国家空间科学中心),2018.

[15] Rösmann C, Hoffmann F, Bertram T. Kinodynamic trajectory optimization and control for car-like robots[C]//2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2017: 5681-5686.

[16] 高佳佳. 基于全局地图的移动机器人路径规划研究[D].西安工业大学,2019.

[17] Dellaert F, Fox D, Burgard W, et al. Monte carlo localization for mobile robots[C]//Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C). IEEE, 1999, 2: 1322-1328.

[18] Sturm J, Engelhard N, Endres F, et al. A benchmark for the evaluation of RGB-D SLAM

systems[C]//2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2012: 573-580.
[19] He H, Jia Y H, Sun L. Simultaneous Location and Map Construction Based on RBPF-SLAM Algorithm[C]//2018 Chinese Control And Decision Conference (CCDC). IEEE, 2018: 4907-4910.
[20] Censi A. An ICP variant using a point-to-line metric[C]//2008 IEEE International Conference on Robotics and Automation. Ieee, 2008: 19-25.

# 附录

**扩展卡尔曼滤波更新阶段实现程序片段：**

```
void Ekf::correct(const Measurement &measurement)
{
  // First, determine how many state vector values we're updating
  std::vector<size_t> updateIndices;
  for (size_t i = 0; i < measurement.updateVector_.size(); ++i)
  {
    updateIndices.push_back(i);
  }
  size_t updateSize = updateIndices.size();
  // Now set up the relevant matrices
  Eigen::VectorXd stateSubset(updateSize);                    // x (in most literature)
  Eigen::VectorXd measurementSubset(updateSize);              // z
  Eigen::MatrixXd measurementCovarianceSubset(updateSize, updateSize);  // R
  Eigen::MatrixXd stateToMeasurementSubset(updateSize, state_.rows());  // H
  Eigen::MatrixXd kalmanGainSubset(state_.rows(), updateSize);  // K
  Eigen::VectorXd innovationSubset(updateSize);               // z - Hx
  stateSubset.setZero();
  measurementSubset.setZero();
  measurementCovarianceSubset.setZero();
  stateToMeasurementSubset.setZero();
  kalmanGainSubset.setZero();
  innovationSubset.setZero();
  // Now build the sub-matrices from the full-sized matrices
  for (size_t i = 0; i < updateSize; ++i)
  {
    measurementSubset(i) = measurement.measurement_(updateIndices[i]);
    stateSubset(i) = state_(updateIndices[i]);
    for (size_t j = 0; j < updateSize; ++j)
    {
      measurementCovarianceSubset(i, j) = measurement.covariance_(updateIndices[i],
updateIndices[j]);
    }
  // The state-to-measurement function, h, will now be a measurement_size x full_state_size
  // matrix, with ones in the (i, i) locations of the values to be updated
  for (size_t i = 0; i < updateSize; ++i)
  {
    stateToMeasurementSubset(i, updateIndices[i]) = 1;
  }
  // (1) Compute the Kalman gain: K = (PH') / (HPH' + R)
```

```
    Eigen::MatrixXd pht = estimateErrorCovariance_ * stateToMeasurementSubset.transpose();
    Eigen::MatrixXd hphrInv = (stateToMeasurementSubset * pht + measurementCovarianceSubset).inverse();
    kalmanGainSubset.noalias() = pht * hphrInv;
    innovationSubset = (measurementSubset - stateSubset);

    // (2) Check Mahalanobis distance between mapped measurement and state.
    if (checkMahalanobisThreshold(innovationSubset, hphrInv, measurement.mahalanobisThresh_))
    {
      // (3) Apply the gain to the difference between the state and measurement: x = x + K(z - Hx)
      state_.noalias() += kalmanGainSubset * innovationSubset;

      // (4) Update the estimate error covariance using the Joseph form: (I - KH)P(I - KH)' + KRK'
      Eigen::MatrixXd gainResidual = identity_;
      gainResidual.noalias() -= kalmanGainSubset * stateToMeasurementSubset;
      estimateErrorCovariance_ = gainResidual * estimateErrorCovariance_ * gainResidual.transpose();
      estimateErrorCovariance_.noalias() += kalmanGainSubset *
                                            measurementCovarianceSubset *
                                            kalmanGainSubset.transpose();
    }
  }
```

**扩展卡尔曼滤波预测阶段实现程序片段：**

```
  void Ekf::predict(const double referenceTime, const double delta)
  {
    double roll = state_(StateMemberRoll);
    double pitch = state_(StateMemberPitch);
    double yaw = state_(StateMemberYaw);
    double xVel = state_(StateMemberVx);
    double yVel = state_(StateMemberVy);
    double zVel = state_(StateMemberVz);
    double pitchVel = state_(StateMemberVpitch);
    double yawVel = state_(StateMemberVyaw);
    double xAcc = state_(StateMemberAx);
    double yAcc = state_(StateMemberAy);
    double zAcc = state_(StateMemberAz);
    // We'll need these trig calculations a lot.
    double sp = ::sin(pitch);
    double cp = ::cos(pitch);
    double cpi = 1.0 / cp;
    double tp = sp * cpi;
    double sr = ::sin(roll);
```

```
        double cr = ::cos(roll);
        double sy = ::sin(yaw);
        double cy = ::cos(yaw);
        prepareControl(referenceTime, delta);
        // Prepare the transfer function
        transferFunction_(StateMemberX, StateMemberVx) = cy * cp * delta;
        transferFunction_(StateMemberX, StateMemberVy) = (cy * sp * sr - sy * cr) * delta;
        transferFunction_(StateMemberX, StateMemberVz) = (cy * sp * cr + sy * sr) * delta;
        transferFunction_(StateMemberX, StateMemberAx) = 0.5 * transferFunction_(StateMemberX,
StateMemberVx) * delta;
        transferFunction_(StateMemberX, StateMemberAy) = 0.5 * transferFunction_(StateMemberX,
StateMemberVy) * delta;
        transferFunction_(StateMemberX, StateMemberAz) = 0.5 * transferFunction_(StateMemberX,
StateMemberVz) * delta;
        transferFunction_(StateMemberY, StateMemberVx) = sy * cp * delta;
        transferFunction_(StateMemberY, StateMemberVy) = (sy * sp * sr + cy * cr) * delta;
        transferFunction_(StateMemberY, StateMemberVz) = (sy * sp * cr - cy * sr) * delta;
        transferFunction_(StateMemberY, StateMemberAx) = 0.5 * transferFunction_(StateMemberY,
StateMemberVx) * delta;
        transferFunction_(StateMemberY, StateMemberAy) = 0.5 * transferFunction_(StateMemberY,
StateMemberVy) * delta;
        transferFunction_(StateMemberY, StateMemberAz) = 0.5 * transferFunction_(StateMemberY,
StateMemberVz) * delta;
        transferFunction_(StateMemberZ, StateMemberVx) = -sp * delta;
        transferFunction_(StateMemberZ, StateMemberVy) = cp * sr * delta;
        transferFunction_(StateMemberZ, StateMemberVz) = cp * cr * delta;
        transferFunction_(StateMemberZ, StateMemberAx) = 0.5 * transferFunction_(StateMemberZ,
StateMemberVx) * delta;
        transferFunction_(StateMemberZ, StateMemberAy) = 0.5 * transferFunction_(StateMemberZ,
StateMemberVy) * delta;
        transferFunction_(StateMemberZ, StateMemberAz) = 0.5 * transferFunction_(StateMemberZ,
StateMemberVz) * delta;
        transferFunction_(StateMemberRoll, StateMemberVroll) = delta;
        transferFunction_(StateMemberRoll, StateMemberVpitch) = sr * tp * delta;
        transferFunction_(StateMemberRoll, StateMemberVyaw) = cr * tp * delta;
        transferFunction_(StateMemberPitch, StateMemberVpitch) = cr * delta;
        transferFunction_(StateMemberPitch, StateMemberVyaw) = -sr * delta;
        transferFunction_(StateMemberYaw, StateMemberVpitch) = sr * cpi * delta;
        transferFunction_(StateMemberYaw, StateMemberVyaw) = cr * cpi * delta;
        transferFunction_(StateMemberVx, StateMemberAx) = delta;
        transferFunction_(StateMemberVy, StateMemberAy) = delta;
        transferFunction_(StateMemberVz, StateMemberAz) = delta;
        // Prepare the transfer function Jacobian. This function is analytically derived from the
```

```
// transfer function.
double xCoeff = 0.0;
double yCoeff = 0.0;
double zCoeff = 0.0;
double oneHalfATSquared = 0.5 * delta * delta;
yCoeff = cy * sp * cr + sy * sr;
zCoeff = -cy * sp * sr + sy * cr;
double dFx_dR = (yCoeff * yVel + zCoeff * zVel) * delta +
                (yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
double dFR_dR = 1.0 + (cr * tp * pitchVel - sr * tp * yawVel) * delta;
xCoeff = -cy * sp;
yCoeff = cy * cp * sr;
zCoeff = cy * cp * cr;
double dFx_dP = (xCoeff * xVel + yCoeff * yVel + zCoeff * zVel) * delta +
                (xCoeff * xAcc + yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
double dFR_dP = (cpi * cpi * sr * pitchVel + cpi * cpi * cr * yawVel) * delta;
xCoeff = -sy * cp;
yCoeff = -sy * sp * sr - cy * cr;
zCoeff = -sy * sp * cr + cy * sr;
double dFx_dY = (xCoeff * xVel + yCoeff * yVel + zCoeff * zVel) * delta +
                (xCoeff * xAcc + yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
yCoeff = sy * sp * cr - cy * sr;
zCoeff = -sy * sp * sr - cy * cr;
double dFy_dR = (yCoeff * yVel + zCoeff * zVel) * delta +
                (yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
double dFP_dR = (-sr * pitchVel - cr * yawVel) * delta;
xCoeff = -sy * sp;
yCoeff = sy * cp * sr;
zCoeff = sy * cp * cr;
double dFy_dP = (xCoeff * xVel + yCoeff * yVel + zCoeff * zVel) * delta +
                (xCoeff * xAcc + yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
xCoeff = cy * cp;
yCoeff = cy * sp * sr - sy * cr;
zCoeff = cy * sp * cr + sy * sr;
double dFy_dY = (xCoeff * xVel + yCoeff * yVel + zCoeff * zVel) * delta +
                (xCoeff * xAcc + yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
yCoeff = cp * cr;
zCoeff = -cp * sr;
double dFz_dR = (yCoeff * yVel + zCoeff * zVel) * delta +
                (yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
double dFY_dR = (cr * cpi * pitchVel - sr * cpi * yawVel) * delta;
xCoeff = -cp;
yCoeff = -sp * sr;
```

```
  zCoeff = -sp * cr;
  double dFz_dP = (xCoeff * xVel + yCoeff * yVel + zCoeff * zVel) * delta +
                  (xCoeff * xAcc + yCoeff * yAcc + zCoeff * zAcc) * oneHalfATSquared;
  double dFY_dP = (sr * tp * cpi * pitchVel - cr * tp * cpi * yawVel) * delta;
  // Much of the transfer function Jacobian is identical to the transfer function
  transferFunctionJacobian_ = transferFunction_;
  transferFunctionJacobian_(StateMemberX, StateMemberRoll) = dFx_dR;
  transferFunctionJacobian_(StateMemberX, StateMemberPitch) = dFx_dP;
  transferFunctionJacobian_(StateMemberX, StateMemberYaw) = dFx_dY;
  transferFunctionJacobian_(StateMemberY, StateMemberRoll) = dFy_dR;
  transferFunctionJacobian_(StateMemberY, StateMemberPitch) = dFy_dP;
  transferFunctionJacobian_(StateMemberY, StateMemberYaw) = dFy_dY;
  transferFunctionJacobian_(StateMemberZ, StateMemberRoll) = dFz_dR;
  transferFunctionJacobian_(StateMemberZ, StateMemberPitch) = dFz_dP;
  transferFunctionJacobian_(StateMemberRoll, StateMemberRoll) = dFR_dR;
  transferFunctionJacobian_(StateMemberRoll, StateMemberPitch) = dFR_dP;
  transferFunctionJacobian_(StateMemberPitch, StateMemberRoll) = dFP_dR;
  transferFunctionJacobian_(StateMemberYaw, StateMemberRoll) = dFY_dR;
  transferFunctionJacobian_(StateMemberYaw, StateMemberPitch) = dFY_dP;
  Eigen::MatrixXd *processNoiseCovariance = &processNoiseCovariance_;
  if (useDynamicProcessNoiseCovariance_)
  {
    computeDynamicProcessNoiseCovariance(state_, delta);
    processNoiseCovariance = &dynamicProcessNoiseCovariance_;
  }
  // (1) Apply control terms, which are actually accelerations
  state_(StateMemberVroll) += controlAcceleration_(ControlMemberVroll) * delta;
  state_(StateMemberVpitch) += controlAcceleration_(ControlMemberVpitch) * delta;
  state_(StateMemberVyaw) += controlAcceleration_(ControlMemberVyaw) * delta;
  state_(StateMemberAx) = (controlUpdateVector_[ControlMemberVx] ?
    controlAcceleration_(ControlMemberVx) : state_(StateMemberAx));
  state_(StateMemberAy) = (controlUpdateVector_[ControlMemberVy] ?
    controlAcceleration_(ControlMemberVy) : state_(StateMemberAy));
  state_(StateMemberAz) = (controlUpdateVector_[ControlMemberVz] ?
    controlAcceleration_(ControlMemberVz) : state_(StateMemberAz));
  // (2) Project the state forward: x = Ax + Bu (really, x = f(x, u))
  state_ = transferFunction_ * state_;
  // (3) Project the error forward: P = J * P * J' + Q
  estimateErrorCovariance_ = (transferFunctionJacobian_ *
                              estimateErrorCovariance_ *
                              transferFunctionJacobian_.transpose());
  estimateErrorCovariance_.noalias() += delta * (*processNoiseCovariance);
}
```