西北工業大學

本科毕业设计论文

题 目 集群无人机协同 SLAM

专业名称 飞行器控制与信息工程

学生姓名 刘昭宏

指导教师 布树辉

毕业时间 2022 年 6 月

摘　要

摘要正文

关键词：无人机集群，协同 SLAM

ABSTRACT

摘要正文-英文

Key Words: Flight control model, Quaternion, Three-dimensional visual simulation system, Loop simulation

目录

第一章 绪论

1.1 研究背景

当今阶段，无人机（Unmanned Aerial Vehicle）技术迅速发展，在单架无人机上可以集成更多的系统，意味着对于单机更强大的功能。单架无人机也已经被广泛应用于灾害救援、监控巡查、环保监测、电力巡检、交通监视、农业植保等领域。但是，面对复杂的应用环境和多样化的需求，单架无人机受自身软硬件条件的限制，仍然具有一些局限性；为了弥补单架无人机的局限性，由多架相同或不同型号的无人机组成多无人机系统，即无人机集群，协同定位，共同完成任务；

通过集群的方式，能最大地发挥无人机的优势，又能避免由于单架无人机执行任务不佳或失败造成的不良后果，提高任务执行效率，扩展新的任务执行方式，从而达到提高系统可靠性，增强任务执行效果的目的。对于无人机的自主导航，能够在进入未知的环境时掌握无人机的位置和姿态是使其成功的关键。尽管 GPS 对于掌握无人机的位置有巨大的帮助，但仍存在普适性有限和准确度不高的问题；在一些特定场景下，比如室内狭小空间，对定位精度要求很高，GPS 定位的局限性就被显露出来。而 SLAM 技术则可以仅通过自身携带的传感器，来完成这一任务，同时达到一定的精度；

SLAM（simultaneous localization and mapping）技术，即同时定位与建图，已有三十多年的研究；SLAM 最早由 Smith、Self 和 Cheeseman 于 1988 年提出。SLAM 指的是机器人在未知环境中从一个未知位置开始移动，在移动过程中根据位置和地图进行自身定位，同时在自身定位的基础上建造增量式地图，实现机器人的自主定位和导航。由于其重要的理论与应用价值，被很多学者认为是实现真正全自主移动机器人的关键 [1] 。

但是单平台 SLAM 受到传感器性能的限制，存在两点不足：一是测量距离受限，单平台常用的传感器如激光雷达，其最远有效距离为 200 米，不能够满足大场景定位建图的需要，任务效率比较低；二是单平台构建出的三维点云相对稀疏，不能表现出足够的场景信息 [2] 。

因此集群无人机协同 SLAM 方案有望解决单机存在的制约问题，主要表现在两个方面：一是多机意味着多传感器，能在大范围场景进行同时定位与建图；二是多机协同 SLAM 可以通过建图覆盖的检测，构建更加稠密和精细的点云地图。

目前单机同时定位与建图已经相对成熟，但是多机 SLAM 由于其控制复杂、数据传输量大、信息处理速度受限、关键数据融合效率低等问题，仍然需要大量的理论研究和实验。

1.2 研究内容及论文结构

本文研究目标旨在实现一套能够在室内高精度环境或 GPS 拒止环境下使用视觉进行多机定位和大范围建图的多无人机协同 SLAM 的方案；其中：

1. 在 SLAM 方面：掌握一些优秀的开源方案，选择各自优点做出一定的融合。并且有一套针对地图融合的方法。

2. 在仿真方面：在 ROS 的 gazebo 仿真平台中实现一定的集群控制方法，能够控制多个无人机协同完成同时定位与建图的任务。

3. 在真机方面：实现单机的视觉 SLAM；在安全的前提下实现双机协同 SLAM，将仿真环境下的协同 SLAM 算法在真机上完成验证，得到场景地图

本研究内容是多机协作进行定位与建图，多机协同 SLAM 能大大提高任务进行的效率，但由于无人机数量多，协同上存在一定困难；研究内容分为三个模块：SLAM 模块、仿真模块和真机模块。

SLAM 模块的主要内容是实现一套可协作的 SLAM 方案，实现的步骤有：

1. 研究传统的视觉 SLAM 的特征点提取、匹配、初始化、后端优化等技术，研究机器人的位姿估计技术；研究并了解 SLAM 技术的整体框架

2. 研究 CCM-SLAM 方案，重点研究其协同的机制和方法，服务端到子端的信息传递和接口设计等

3. 研究 VINS-Fusion 方案中的 VIO 方法，研究如何利用 IMU 与相机数据联合进行更加准确的位姿估计

仿真模块的主要内容是在 ROS 的 gazebo 中研究如何实现多机协同的同时定位与建图，实现的步骤有：

1. 首先研究 PX4 和 MAVROS 之间的通信方式，ROS 的话题发布和订阅方式，研究如何用程序解锁一架无人机、使其进入 Offboard 模式、起飞悬停并降落

2. 研究如何用程序发布话题，控制无人机按照航路点飞行

3. 研究如何构建多机的仿真环境，如何对多机进行控制，其控制策略的选择，即集中式或分布式的多机编队控制

4. 研究如何更改无人机的定位设置，将其从 GPS 定位改为视觉 SLAM 定位；并且完成单机的摄像头内容读取

5. 研究如何在 gazebo 中载入其他场景，在场景中控制无人机飞行，并且对拍摄到的画面进行建图，完成自身定位

6. 研究如何在 gazebo 中完成多机基于视觉的同时定位与建图，并且拼合地图，用第三方软件显示；研究多机的联合优化与协同方法

真机模块的主要内容是控制无人机的协同飞行及通信，实现的步骤有：

1. 研究无人机通过 MAVROS，MAVLINK 与地面站的通信方法，尤其是用于 SLAM 的关键数据的传输

2. 研究多无人机与地面站之间的、多无人机之间的数据传输

3. 研究多无人机之间的可变基线控制技术，如何设计一个详细的算法控制基线距离

基于研究内容的层级关系，本论文的结构主要由四层构成，如图1-1所示。

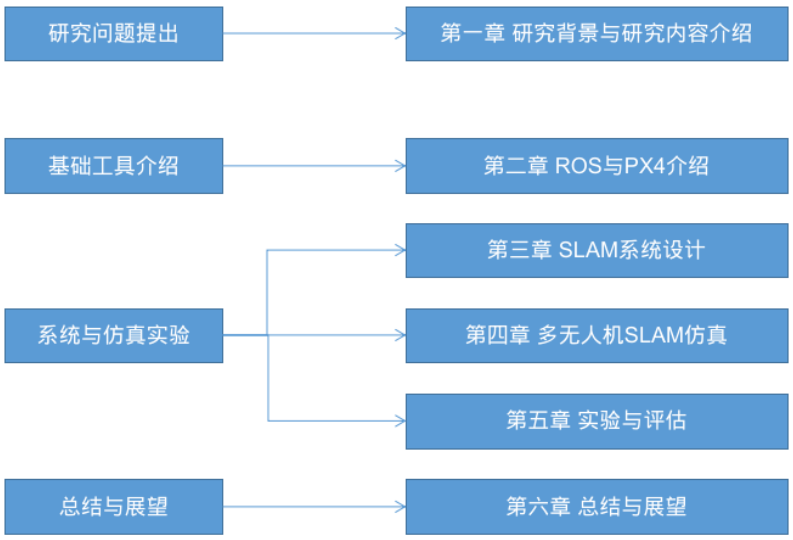

图 1-1 论文结构

第一章：主要介绍研究背景，进行问题的提出，从而引出本文的研究内容和研究目标；从宏观的角度概括本文的内容，并且对问题做出概述。

第二章：主要介绍研究中用到的 ROS（Robotics Operating System，机器人操作系统）和 PX4 AutoPilot 飞控软件系统。

第三章：主要介绍 SLAM 的原理和作用，SLAM 系统的基本流程，优秀的开源 SLAM 方案和地图融合的设计。

第四章：主要介绍仿真实验的情况，在 ROS 的 gazebo 环境中进行。

第五章：主要介绍真机实验的情况，并且做出相应的评估。

第六章：总结研究过程中的创新点和不足之处，提出进一步研究的大致方向，展望未来的研究工作。

第二章　ROS 与 PX4 介绍

2.1　ROS 介绍

本节主要对 ROS 平台进行介绍，包括 ROS 核心的消息机制和研究中要用到的 gazebo 仿真平台。

21 世纪开始，随着人工智能研究的发展，催生出了一批智能机器人的研究项目；ROS 诞生于 2007 年斯坦福大学 AI 实验室 Morgan Quigley 的 STAIR（Standford Artificial Intelligence Robot）项目，其期望构建一个基于移动机器人 + 机械臂的原型；该项目于 2008 年受到 Willow Garage 公司关注，其决定用商业化手段来推进机器人的发展，使机器人平台能够更快地走进人们的日常生活；Willow Garage 接手该项目后两年，2010 年第一代 ROS 即 ROS1.0 发布；2013 年，OSRF（Open Source Robotics Foundation）接管了 ROS 的维护工作和版本的升级工作，随后至 2018 年间，ROS 的 Indigo、Kinetic 和 Melodic 版本相继发布。

ROS 即 Robotics Operating System，是一个针对机器人的开源、元级操作系统，在某些方面，ROS 更像是一种机器人框架（robot framework）；它提供类似于操作系统的服务，包含底层的驱动程序管理、底层的硬件描述，随后上升到软件程序之间的消息传递、功能包的管理和发布、也提供用于获得、编译、编写和多设备跨计算机运行代码所需的库等。换言之，ROS 是由一套通信机制，开发工具，一系列应用功能和一个庞大的生态系统组成的集合，其目标为提高机器人研发中的软件复用率，不断完善他人的工作，进行更好的开发。

2.1.1　ROS 的消息机制

ROS 提供了一套松耦合分布式通信机制，这种分布式处理框架（又名 Nodes），是以多个节点及节点之间的通信组成的。其中，节点（Node）和节点管理器（ROS Master）是 ROS 的核心概念，若干个节点在节点管理器下构建起来，共同实现特定的功能。

每一个节点是一个独立的执行单元，由可执行文件构成，在程序中需要声明节点的名称；节点的名称必须唯一，否则 ROS 会舍弃掉时间节点靠前的节点；节点执行具体的任务进程，比如单目的 ORB-SLAM2 中，其节点为 Mono，SLAM 的任务仅靠一个节点完成。

节点管理器是节点的控制中心，其作用是辅助节点的查找，帮助节点之间建立通信连接；还能提供节点的命名和注册等服务，以及提供了能够存储全局变量的配置的参数服务器。

如图2-1所示，节点在经过节点管理器注册后，可以建立节点之间的通信；常用的节点之间通信方式有两种，为话题（Topic）通信和服务（Service）通信：

1. 话题通信是异步通信机制，数据为单向传输；数据的流向为发布者（Publisher）到订阅者（Subscriber）；完成话题通信需要定义一个话题（Topic）及其消息（Message）的内容，之后通过发布者（Publisher）发布该话题，并且订阅者

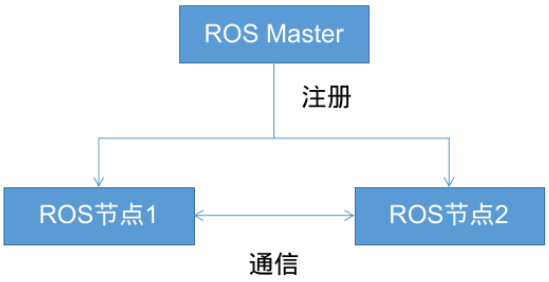

图 2-1 ROS 中的节点及通信

（Subscriber）订阅该话题的操作，完成数据的传输，消息的数据结构由.msg 文件定义；话题通信可以完成多对多的信息传递。

2. 服务的通信机制则为同步，数据为双向传输；数据的流向为客户端（Client）与服务器（Server）之间的交互；完成服务的通信需要客户端向服务器发送请求，服务器完成任务处理后，向客户端返回应答数据，表示请求和应答的数据结构定义在.srv 文件中；服务通信一般用于逻辑判断，比如询问一项任务是否执行完毕，是一对多的节点处理关系。

发布和订阅话题的方法，发布者和订阅者类似，以发布者为例：先实例化一个发布者对象，定义发布的话题名称、数据类型和队列长度，最后对消息进行定义并发送，简单的逻辑代码如下：

```
ros::NodeHandle n;  // define ros node handle
// define a publisher
ros::Publisher pub = n.advertise<'message type'>
        ("topic name", 'queue length');
// publish message
pub.publish(message);
```

需要注意的是，订阅者则需要声明并定义一个回调函数，在实例化 Subscriber 的对象后，通过 ROS 的 spin() 函数，循环等待回调函数获得话题消息。

客户端-服务器模型下的服务通信，则比话题的发布和订阅复杂；客户端的编程实现中，需要设置阻塞函数，其作用是直到发现对应的服务时才向下进行，否则程序被截止在该位置；如果对应的服务被发现，阻塞函数通过，之后创建客户端并且进行数据的设置，完成服务调用的请求，其代码实现如下：

```
// wait for right service
ros::service::waitForService("service name");
// create a client, connecting to service
ros::ServiceClient client = n.serviceClient
        <'data type'>("service name");
```

```
// call service
client.call(srv);
```

服务器的实现与订阅者类似，需要一个回调函数，如果收到了客户端发来的请求，则会触发回调函数，程序向下进行，否则将循环等待回调函数收到客户端发来的请求。

除此之外，ROS 中还有参数（Parameter）或参数服务器的概念，其作用类似全局共享字典，节点可以进行访问，适合存储一些和系统配置相关的静态非二进制的参数，以供节点读取。

2.1.2 gazebo 仿真

gazebo 是 ROS 自带的仿真软件，其功能有构建具有运动属性的机器人仿真模型，提供了一些针对 CAD 和 soildworks 等 2D、3D 设计软件的接口；gazebo 还具有构建现实世界的各种场景的仿真模型的功能，能够在 gazebo 环境中建立一个与现实十分相似的场景用于算法验证；在传感器的仿真上，gazebo 拥有一个强大的传感器模型库，比如单目相机、双目相机、深度相机等，还可以根据需求自行配置传感器的类型，实现多传感器融合；除此之外，gazebo 还引入了现实世界的物理性质，如重力的影响，使仿真环境更加贴近现实。

gazebo 的仿真环境中，其文件大致可以分为三种类型：model，world，launch 文件；同时，这三种文件也代表了不同的分级；

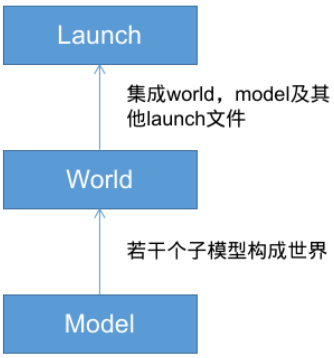

图 2-2 model，world，launch 层级关系

如图2-2所示：

1. model（模型）是 gazebo 环境中的元级元素，也就是最底层文件，比如环境中的树木、双目相机、无人机、墙壁、桌子等都是 model 级别的物体。model 由 sdf 文件和 config 文件构成；sdf 文件用 HTML 或 XML 标签语言描述了该模型的主要内容，包括模型的构建方法、模型中对其他模型的调用以及连接方式、模型的位姿等参数配置；而 config 文件中记录了模型的作者及联系方式、模型的版本、模型的命名和描述等信息。由于模型具有可拼接的属性，因此一个模型可以由若干个模型组成。

2. world（世界）文件将模型集成起来，包含模型和物理性质的设置，是 gazebo 环境中的中层文件。该层与 model 层相同，都无法完成代码对模型的直接控制。

3. launch 文件是集成了 model，world 以及其他 launch 文件的 gazebo 中最顶层的文件；launch 文件不同于 model 和 world 文件，其可以通过代码完成对模型的直接控制；launch 文件在 ROS 中拥有定义，其以 XML 标签语言书写，可以在 launch 文件中完成嵌套其他 launch 文件、命名重映射、设置参数、启动 ROS 节点等任务；在 ROS 中有与 launch 文件对应的指令 roslaunch，用于启动该 launch 文件。

2.2 PX4 AutoPilot

PX4 是一款专业级开源飞控，也可以称之为自动驾驶仪；因其应用的平台不局限于飞行器，在竞速和物流应用的地面车辆和潜水艇等载具上也可以用其进行控制。PX4 由来自学术界和业界的顶级开发商开发，并且配套有活跃的全球论坛社区，其 software 的源码在 github 上保持着 issue 和 pull request 的更新，是应用十分广泛的一款飞控软件。需要注意，PX4 Software 和 Pixhawk4 并不是同一概念，前者为飞控软件，而后者为飞控硬件。PX4 软件的内部包含了针对不同机型（包括多旋翼、固定翼和 VTOL 垂直起降固定翼等）的控制律设计，还包含了强大的飞行模式设计和安全设计。PX4 还可以作为核心应用在更广阔的平台，比如使用了 QGroundControl 地面站、Pixhawk 硬件、基于计算机、相机等的使用 MAVLink 协议的 MAVSDK 融合等。

2.2.1 FailSafe 机制

FailSafe 机制，即安全生效机制，其含义为：当错误发生时，对飞机进行保护或恢复到安全的状态，避免错误可能导致的不良后果。PX4 的 FailSafe 系统是可编辑的，意味着开发者可以根据自身的需求设置对 FailSafe 的触发，以保证在安全的情况下实验或完成任务。FailSafe 系统被触发后，一般有自动着陆、保持位置或返回特定的航路点几种反馈措施。

安全生效机制监控的主要情况有：

1. 低电量；该情况在仿真中影响较小，但在真机实验中，低电量可能意味着无法安全返航，因此必须由安全生效机制介入；

2. 远程控制信号丢失，如遥控器信号丢失；

3. 位置信息丢失，比如 GPS 信号弱，对位置的估计不够精确，可能会影响任务的完成情况，因此由安全生效机制介入；

4. 场外连接丢失；是指进入到 Offboard 模式后，丢失了与计算机之间的连接，导致计算机无法通过 Offboard 程序对无人机进行控制；

5. 数据链丢失；一般是指丢失了与 GCS（Ground Control Station，地面站）之间的数传连接；

6. 超出地理围栏；Geofence 即地理围栏，是执行任务前设置的无人机可活动区域，高度一般不设限；

7. 任务判断；防止在新的起飞位置上执行先前的任务；

PX4 版本更新后，如果直接输入 commander takeoff 指令，可能会遇到无法起飞的情况；同时在 PX4 终端中，会提示 FailSafe activated，即安全生效模式被激活；一般遇到这种情况，需要在 PX4 终端的字里行间和地面站 QGC 的信息提示中去分析问题原因。比如在仿真中遇到 No RC 的情况，RC 即 Remote Control，对于 SITL（软件在环仿真）是没有遥控器的信号输入的，因此需要在 QGC 中打开 virtual joystick （虚拟摇杆），否则 FailSafe 模式会由于没有 RC 保持给飞机上锁。

2.2.2 EKF 与飞行模式

一般 ECL 与 EKF 会同时出现，ECL 即 Estimation and Control Library，状态估计和控制库；EKF 即 Extended Kalman Filter，扩展卡尔曼滤波，是一种优化算法。两者的结合是使用扩展卡尔曼滤波方法的状态估计与控制库，其作用是加工传感器的数据，并对 IMU（惯性测量单元）加工的速度和位置、四元数表示的旋转矩阵、风速和磁罗盘得到的方向等信息进行加工处理和估计。EKF 使用 IMU、磁力计、高度计、GPS、测速仪等传感器。

为了将传统的气压计 +GPS 的高度与位置估计更改为使用视觉信息的高度与位置估计，需要修改 EKF 传感器的相关参数，这里指用于多旋翼和固定翼的基于扩展卡尔曼滤波的高度和位置估计。

1. *EKF2_AID_MASK* (Integer bitmask controlling data fusion and aiding methods)，该参数决定了 GPS 数据的融合方法；默认设置的参数为 0，其意为使用 GPS 数据作为定位；如果修改为视觉定位（vision position fusion），需要将该参数改为 3。

2. *EKF2_HGT_MODE* (Determines the primary source of height data used by the EKF)，该参数决定了 EKF 首选的高度信息传感器；默认参数设置为 0，使用气压器得到高度；如果修改为视觉定位，则需要将该参数改为 3。

在启动仿真之前，需要根据信息融合的类型更改以上两个参数，修改参数的方式有以下两种：

1. 更改 rcS 文件中的参数配置；rcS 文件属于脚本文件，用于配置系统，PX4 需要修改的 rcS 文件位于 *ROMFS/px4fmu_common/init.d-posix/rcS* 中，rcS 文件中的参数被修改后，需要删除 ROS 的 eeprom 中存储的参数文件（该文件用于快速预加载参数）；但该方法存在一些问题，需要在 launch 中特殊指明 PX4 使用的 rcS 文件地址，否则 PX4 会从默认的 build 文件夹中选择 rcS 文件进行参数配置。

2. 更改地面站中的参数配置；使用 QGC 地面站，直接在参数设置中更改位置和高度估计的传感器，这种方式免去了每次更改参数后需要删除 ROS 参数缓存文件的麻烦。

飞行模式决定了飞行器对 RC 远程控制输入的回应，以及在全自主的飞行过程中飞行器如何控制自身的运动；飞行模式为操作者主要提供了不同种类和程度的自动控制协助，飞行欧式的切换可以由遥控器和地面站完成。下面介绍几种主要的飞行模式（对于多旋翼无人机）：

1. Manual/Stabilized Mode，手动/增稳飞行模式；是最常用的飞行模式，手动模式即操纵手通过手中的遥控器（RC）控制飞机的滚转、俯仰、油门和前后左右的移动；增稳模式也是由操纵手操纵，但引入了内部控制律，使得遥控器输入到飞机运动的表现更加平滑、易控，这是由其内部的控制律决定的；一般情况下为了防止飞机过于剧烈的运动，都采用增稳模式进行飞行。

2. Position Mode，定点模式；其主要特点是当操纵杆释放或回归中心时，飞机会保持在 3D 空间中的一个定点位置处，并且自动解算出相应的力去补偿风和其他的干扰力。该模式是对于新手最安全的模式。

3. Altitude Mode，定高模式；其主要特点是当操纵杆释放或回归中心时，飞机会保持固定高度，但不会去平衡风等其他干扰所造成的水平位置的漂移。

4. Offboard Mode，场外模式；指通过电脑连接或地面站连接，使飞机按照设定的位置、速度或高度等参数飞行，该模式通过 MAVLink 与场外设备传递信息，是仿真中使用的主要模式。

5. 其他模式，如圆轨迹模式、起飞降落模式、跟随模式、返回模式等。

2.2.3 联合 MAVROS 的 Offboard 模式

本小节主要介绍在 Offboard 模式下，使用 ROS 节点通过 MAVROS 向 PX4 发送信息，使无人机起飞降落、按航路点移动等。

Offboard 模式下的无人机正常起飞，需要首先解锁，然后切换飞行模式（默认手动模式）到 Offboard 模式，但是需要在切换模式前，以不低于 $2Hz$ 的频率发布一些设定点（setpoints），具体实现起飞的步骤如下：

1. 连接 fcu（MAVROS），判断的标准是 MAVROS 消息类中的 state 是否表示为连接；如果连接上则可以继续执行，未连接上则通过 spin() 函数循环等待。

2. 设定 setpoints 的坐标值，并以不低于 $2Hz$ 的频率发送一些点。

3. 解锁，解锁成功后切换到 Ofboard 模式并起飞。

其程序的实现如下：

首先是需要的头文件定义，其包括了 C++ 的基本库，MAVROS 的消息相关库，和 ROS 库；

```
//
// Created by hazyparker on 2022/1/11.
// realize mode switching and anto takeoff and landing
```

```
#include <iostream>
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
#include <mavros_msgs/PositionTarget.h>
```

之后需要在 main 函数之前，声明定义回调函数，其中包括对当前状态和当前位置的回调函数和消息定义：

```
// record current state
mavros_msgs::State current_state; /* NOLINT */

// callback function for Subscriber stats_sub
void state_cb(const mavros_msgs::State::ConstPtr& msg){
current_state = *msg;
}

// record current pose
geometry_msgs::PoseStamped current_pose; /* NOLINT */

// callback function for Subscriber for local_pos_sub
void local_cb(const geometry_msgs::PoseStamped::ConstPtr& msg){
current_pose = *msg;
}
```

在 main 函数中，首先要定义 ROS 节点和句柄，实例化飞行模式和当前位置信息的订阅者和发布者，然后以一定的频率发送一些点，以便切换到 Offboard 模式：

```
// init ros node
ros::init(argc, argv, "offb_node");

// create node handle
ros::NodeHandle nh;

// define subscribers and clients
ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>
("mavros/state", 10, state_cb);
```

```
ros::Subscriber local_pos_sub = nh.subscribe<geometry_msgs::PoseStamped>
("mavros/local_position/pose",10,local_cb);
ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>
("mavros/setpoint_position/local", 10);
ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>
("mavros/cmd/arming");
ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>
("mavros/set_mode");

//the set-point publishing rate MUST be faster than 2Hz
ros::Rate rate(20.0);

// wait for FCU connection
while(ros::ok() && !current_state.connected){
ros::spinOnce();
rate.sleep();
ROS_INFO("wait for fcu connecting...");
}
ROS_INFO("fcu connected successfully");

// set pose
geometry_msgs::PoseStamped pose;
pose.pose.position.x = 0;
pose.pose.position.y = 0;
pose.pose.position.z = 2;

//send a few set-points before starting
for(int i = 100; ros::ok() && i > 0; --i){
local_pos_pub.publish(pose);
ros::spinOnce();
rate.sleep();
}
```

最后首先发送指令，使无人机解锁，之后使其切换到 Offboard 模式，完成起飞并到达目标点的指令：

```
int main(int argc, char **argv){
```

```
mavros_msgs::SetMode offb_set_mode;
offb_set_mode.request.custom_mode = "OFFBOARD";

mavros_msgs::CommandBool arm_cmd;
arm_cmd.request.value = true;

ros::Time last_request = ros::Time::now();
ROS_INFO("Off boarding");
while(ros::ok()){
if( current_state.mode != "OFFBOARD" &&
(ros::Time::now() - last_request > ros::Duration(5.0))){
if( set_mode_client.call(offb_set_mode) &&
offb_set_mode.response.mode_sent){
ROS_INFO("Off-board mode enabling...");
}
last_request = ros::Time::now();

} else {
if( !current_state.armed &&
(ros::Time::now() - last_request > ros::Duration(5.0))){
if (current_state.mode != "OFFBOARD") ROS_INFO("Off board mode was shut unexpecte
if( arming_client.call(arm_cmd) &&
arm_cmd.response.success){
ROS_INFO("Vehicle armed");
}
last_request = ros::Time::now();
}
}

local_pos_pub.publish(pose);

// wait until reach set point
ros::spinOnce();
```

```
// define Point: current position and set point position (expected)
geometry_msgs::Point curr,aim;
curr = current_pose.pose.position;
aim = pose.pose.position;
double dist = sqrt(pow((curr.x - aim.x), 2) +
pow((curr.y - aim.y), 2) + pow((curr.z - aim.z), 2));
if(dist < 0.1){
ROS_INFO("reached the goal...");
break;
}
rate.sleep();
}

}

return 0;
}
```

第三章　SLAM 系统设计

本章节主要介绍 SLAM 的概念和原理、一个基本 SLAM 系统的框架、两个优秀的开源 SLAM 框架和地图融合算法。

3.1　SLAM 系统

同时定位与建图（SLAM，simultaneous localization and mapping）技术，其希望是机器人在对环境和自身所处在环境中的位置未知的情况下，在反复的运动过程中不断观测到的地图特征完成自身位置的定位和姿态的确定，之后再根据自身位置对环境构建增量式的地图，从而达到同时定位与建图的目的。

3.1.1　SLAM 的分类

SLAM 主要分为视觉 SLAM、激光 SLAM、融合 SLAM 和新颖 SLAM。

对于视觉 SLAM，即用相机完成同时定位与建图的任务。由于相机造价相对较低、电量消耗相对较少、能够获取环境的大量信息，因此相机成为了完成定位与建图任务常用的传感器。视觉 SLAM 主要有五个步骤，传感器信息读取、视觉里程计（Visual Odometry）、后端优化（Optimization）、回环检测（Loop Closing）、建图（Mapping）[6] 。对于静态、刚体、光照变化不明显、且没有过多人为干扰的场景，视觉 SLAM 技术已经十分成熟。当前比较好的方案有 ORB-SLAM；其在对特征点的描述上做了很大创新，相比于 SIFT（尺度不变特征变换，Scale-invariant feature transform）的大计算量和对 GPU 的特殊需求、FAST 关键点描述没有描述子的缺点，ORB 改进了 FAST 的检测子，为其增加了方向性，并且采用了二进制描述子 BRIEF（Binary Robust Independent Elementary Feature）[5] 。

对于激光 SLAM，主要有两种传感器，单线束激光雷达和多线束激光雷达；单线束激光雷达即 2D 雷达，2D 激光雷达的扫描范围比较单一，角度有限，因此比较适合仅平面运动的机器人的定位与建图，对应的经典算法如 GMapping；多线束雷达即 3D 雷达，其获取的信息包含距离和角度，能够还原出目标的三维点云，且不受光照影响，缺点是造价比较昂贵且易受不良天气影响 [7] ，对应的经典算法如谷歌提出的 Cartographer。

对于融合 SLAM，常见的有视觉和惯性的融合，即相机 +IMU（inertial measurement unit，惯性测量单元，包含加速度计和角加速度计）等的多传感器融合；IMU 的工作原理是对加速度的积分、初始速度和起始位置进行混合运算，得到运动轨迹和位姿。但是其容易产生漂移（Drift），并且这种累积误差会随时间增加 [8] 。

对于 VIO（视觉惯性里程计），即上文提到的由相机和惯性测量单元组成的融合传感器，根据融合的框架可以分为松耦合和紧耦合两种。松耦合中对相机关键帧数据的视觉运动估计和对 IMU 测量数据的运动估计是两个独立的模块，计算时互不干涉；计算完成后将其轨迹结果按一定的方法进行融合。紧耦合则是共同使

用相机视觉数据和惯导运动估计数据，共同完成对一组变量的估计；因此其算法更加复杂，且传感器之间的噪声也会相互影响，但是具有更好的效果，也是目前阶段研究的重点方向。这方向上好的方案有 VINS-fusion[9]。

对于新颖 SLAM，比如语义 SLAM；使用神经网络的语义分割、目标检测方法，从图像到位姿，使用其语义分割的结果来完成点云地图的建立和场景识别。语义 SLAM 能够探测出图像中的各个物体，并且能得到在图像中的位置，可以节省传统的人工标定物品的成本，方便机器人的自主理解能力和简便的人机交互 [11] 。

3.1.2 成像原理及相机参数

在各种 SLAM 中，视觉 SLAM 由于其传感器（光学相机）造价较低的原因，成为了 SLAM 中最常用的方式，要了解使用相机的视觉 SLAM 的原理，首先需要了解相机的成像原理及其参数。

如图3-1，可以用小孔成像的原理简单地解释针孔相机的模型：

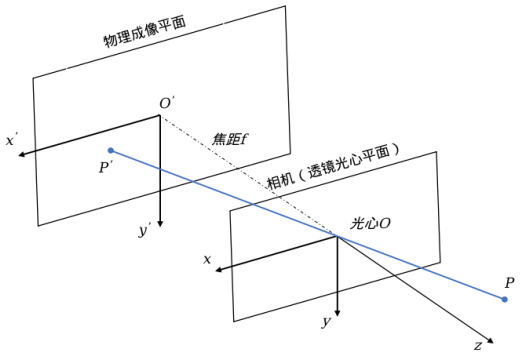

图 3-1 针孔相机模型

Oxy 平面为相机光心（垂直主光轴）所在的平面，称其为相机平面，对应的 $O-x-y-z$ 坐标系即为相机坐标系；$O'x'y'$ 平面为物理成像平面，$\overline{OO'}$ 的长度为焦距 f；在现实世界中有一点 P，设其在相机坐标系下的坐标为 $[X,Y,Z]^T$，其经过小孔 O 投影后，在相机坐标系下落在像素平面上的坐标为 $[X',Y',Z']^T$。

理论下，小孔成像为倒立的实像，但在实际的相机中，成像被人为旋转，成正立的像，因此不考虑坐标系正负号的影响，由相似三角形关系，有：

$$\frac{Z}{f}=\frac{X}{X'}=\frac{Y}{Y'}$$

在此基础上，定义像素坐标系。像素坐标系为二维坐标系，在物理成像平面上；像素坐标系的原点位于图像的左上角，横轴为 u 轴，向右与 x 轴平行，纵轴为 v 轴，向下与 y 轴平行，则可以得到像素坐标与 P' 坐标的关系为：

$$\begin{cases}u=\alpha X'+c_x=\alpha f\dfrac{X}{Z}+c_x\\ v=\beta Y'+c_y=\beta f\dfrac{Y}{Z}+c_y\end{cases}$$

其中，α 和 β 为横轴和纵轴的缩放倍数，c_x 为图像横向像素的一半，c_y 为图像纵向像素的一半。令 $f_x = \alpha f$，$f_y = \beta f$，将像素坐标系下的坐标转换为齐次坐标：

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{Z}\mathbf{KP}$$

则得到了相机的内参矩阵（Camera Intrinsics）**K**。通常情况下，对于焦距固定的相机（或定焦镜头），其出厂之后内参矩阵是固定的；如果无法从厂家得到相机的内参，可以使用标定的方法获得相机的内参矩阵，常见的标定算法有张正友标定法。

除相机的内参外，还有相机外参（Camera Extrinsics）的定义；相机的外参由其旋转矩阵 **R** 和平移向量 **t** 构成。对于 P 点而言，其在相机坐标系（像素坐标平面）下的坐标应为其在世界坐标系下的坐标根据相机相对于世界坐标系的位姿所变换得到的，相机的位姿即由其外参决定，则世界坐标系下 P 点的坐标 $\mathbf{P_w}$ 在相机坐标系下的坐标为：

$$Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}(\mathbf{RP_w} + \mathbf{t})$$

3.1.3 视觉 SLAM 的基本步骤

经典的视觉 SLAM 框架如图3-2所示：

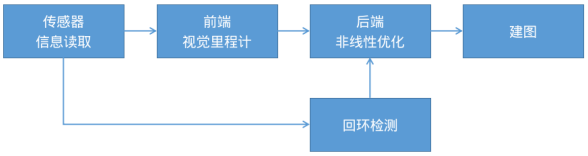

图 3-2 经典视觉 SLAM 框架

经典视觉 SLAM 流程包括以下基本步骤：

1. 传感器信息读取。主要为相机图像的读取以及一些预处理步骤，在不同的视觉 SLAM 算法中，可能还涉及惯性测量元件信息的读取和预处理。

2. 前端视觉里程计（Visual Odometry）。视觉里程计的功能是从相邻的几帧图像之中，根据几何约束，得到相机的运动；并且通过记录地图点（路标）与相机的相对位置，构建局部地图。

3. 后端（非线性）优化（Optimization）。后端主要涉及滤波与非线性优化算法，其目的是减少传感器的误差，完成对运动的相机和周围环境的不确定性的估计。

4. 回环检测（Loop Closure Detection）。回环检测的作用是使机器人能够分辨出当前面临的场景是否曾经来到过，解决位置估计的误差随时间累计，发生漂移的问题；并且消除累计误差，最终得到全局一致的轨迹和地图。

5. 建图（Mapping）。建图即根据前端里程计和后端优化得到的地图点（路标点），构建地图。

视觉里程计是 SLAM 的关键，其基本完成了同时定位与建图的任务。视觉里程计的实现主要有两种方法：特征点法和直接法。

特征点法作为很长时间以来视觉里程计的主要方法，其特点是比较稳定，在光照差异大和画面中有动态物体的情况下也能较好地完成任务。特征点法的核心在特征点的提取与匹配；提取即从每帧图像中找到可以代表图像特征的点，辨识度更高的点。

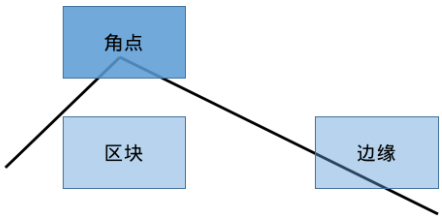

图 3-3 角点、边缘、区块

特征点的一种方法是使用角点。如图3-3，可以把一张图像中的内容分为角点、边缘、区块三种类型。可以发现，指出某两张图像中出现的统一区块是最难以实现的，因为存在大面积相同的色块，无法确定具体点的匹配；其次是边缘，其具有一定的特征，但沿边缘行进，仍可能出现相同的局部特征，造成误匹配；因此选择其中最具有特征角点作为特征。性能较好、比较稳定的角点有 SIFT、SURF，ORB 等。

在完成特征点提取与匹配后，根据不同的传感器类型，有不同的得到相机位姿的方法。对于传统的单目相机传感器，可以使用对极几何约束，得到相机的运动，并通过三角测量的方法恢复地图点相对于每时刻相机的位置。

前端视觉里程计的另一种方法是直接法，由光流法演变而来。光流法与特征点法不同，特征点法使用特征点的描述子来完成特征匹配，而光流法可以跟踪特征点的运动，这样就无需进行大量的描述子匹配运算。直接法可以弥补特征点法的一些缺陷。

后端优化主要有两种方法，滤波和非线性优化。拓展卡尔曼滤波 EKF 及其演变出的粒子滤波方法等，在早期的 SLAM 设计中应用十分广泛。但随着非线性优化方法的普及，现在的 SLAM 方案多用 BA 图优化及位姿图优化的方法，并且有可以使用的 ceres 和 g2o 库。

回环检测在判断场景是否曾经来过时，一般用的是词袋模型（Bag of Words），根据图像中是否存在同样的几种相似特征来判断是否在外观上相似。

建图根据地图的需求，可以分为用于定位的稀疏地图，用于导航、重建的稠密地图和用于交互的语义地图。按照地图的分类，可以分为拓扑地图和度量地图

两种；拓扑地图着重与图节点之间的连通性，而度量地图则能够精确地表示出地图点相对于相机的位置。

3.1.4 对极几何约束与三角测量

3.2 ORB-SLAM2

ORB-SLAM2 是由萨拉戈萨大学的 Raúl Mur-Artal 开发，可以用在单目、双目、RGB-D 深度相机的视觉 SLAM 系统。

3.2.1 ORB 特征点及描述子

ORB 特征点是上文3.1.3提到过的特征点的一种，它的全称是 Oriented FAST and Rotated BRIEF；其中，FAST 是角点的一种，BRIEF 是 Binary Robust Independent Elementary Feature，是一种二进制描述子；ORB 特征点即由改进的 FAST 关键点和带旋转的 BRIEF 描述子组成。

FAST 角点选取的核心思想是，选取的点和周边像素点亮度差别很大，则该点可能是 FAST 角点。如图3-4所示，对于可能被选取为 FAST 角点的像素点 p，其亮度为 I_p；设置亮度差异的阈值 T，可以为 $0.3I_p$；选取半径为 3 的圆上的 16 个点，对应图中的 16 个深色点，记录各点的像素值 I_i，如果 16 个点中有连续 N 个点满足 $|I_i - I_p| > T$，则像素点 p 可以被认为是 FAST 角点，该标准为 FAST-N，N 一般取 9、11、12。

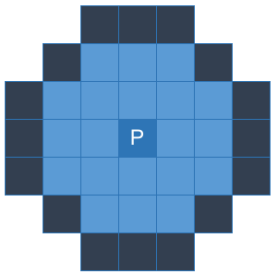

图 3-4 FAST 角点

单纯的 FAST 关键点是不带方向性的，为了准确性，ORB 的 Oriented FAST 关键点给 FAST 角点添加了方向和尺度的描述。其尺度的描述是由计算机视觉中构建金字塔模型的方法，在不同分辨率的图像下都能够提取到特征点，而旋转的描述则是由灰度质心法实现的，如图3-5所示：

设 O 为角点像素，取一个最小的图像块，只有四个像素构成，建立 $O-x-y$ 坐标系，原点的像素用 $I(0,0)$ 表示，则可以找到该图像块的质心为：

$$C = (\frac{I(1,0)}{I(0,0)}, \frac{I(0,1)}{I(0,0)})$$

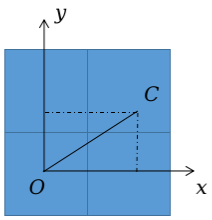

图 3-5 灰度质心法

则可以得到方向向量 $\vec{OC}$，该关键点的方向定义为：

$$\theta = \arctan \frac{I(0,1)}{I(1,0)}$$

BRIEF 描述子为二进制编码，反映了关键点附近 128 个像素对的大小关系，最终由 0 和 1 构成，其具有旋转不变性、选点速度快且易于存储。

3.2.2 ORB-SLAM2 的主要进程

ORB-SLAM2 是一套规范完整的视觉 SLAM 系统，其主要进程可以分为四个：跟踪进程、本地地图进程、回环检测进程、可视化进程。如图3-6所示，从以下五个主要函数中分析。

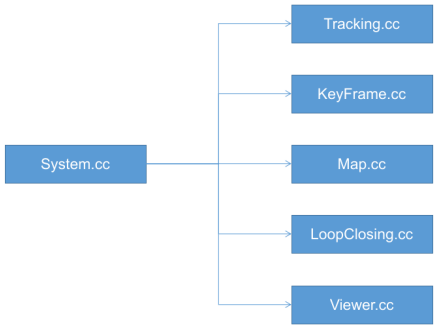

图 3-6 ORB-SLAM2 框架

System.cc，SLAM 系统顶端控制：

首先是读取 ORB 词典，该词典是配合 DoW2 库训练得到的，用于回环检测时判断图像的外观相似度。

其次是创建 ORB 特有的 KeyFrameDatabase，该类用于处理关键帧数据；创建地图，实例化 Map 对象；创建画笔，为可视化做准备。

```
//Create KeyFrame Database
mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary);
```

```
//Create the Map
mpMap = new Map();

//Create Drawers. These are used by the Viewer
mpFrameDrawer = new FrameDrawer(mpMap);
mpMapDrawer = new MapDrawer(mpMap, strSettingsFile);
```

之后是主要线程的初始化，跟踪线程、本地地图线程和回环检测线程：

```
//Initialize the Tracking thread
//(it will live in the main thread of execution, the one that called this constru
mpTracker = new Tracking(this, mpVocabulary, mpFrameDrawer, mpMapDrawer,
mpMap, mpKeyFrameDatabase, strSettingsFile, mSensor);

//Initialize the Local Mapping thread and launch
mpLocalMapper = new LocalMapping(mpMap, mSensor==MONOCULAR);
mptLocalMapping = new thread(&ORB_SLAM2::LocalMapping::Run,mpLocalMapper);

//Initialize the Loop Closing thread and launch
mpLoopCloser = new LoopClosing(mpMap, mpKeyFrameDatabase, mpVocabulary, mSensor!=
mptLoopClosing = new thread(&ORB_SLAM2::LoopClosing::Run, mpLoopCloser);
```

System 相当于工程中的 Main 函数，汇集了实现一个 SLAM 系统所需的全部流程，使用顶层的代码进行控制，具体功能则由底层的类及其中的函数去实现。

Tracking.cc，关键的跟踪进程，其主要函数为图像的转换函数和 Track 主函数：

1. 图像转换函数是针对不同传感器设置的，以弹幕相机为例；该函数需要判断图像是否为灰度图，判断图像为 RGB 或 BGR 编码，并最后将图像转为系统使用的灰度 3 通道图像，赋予其时间戳和 ORB 字典、相机的内参矩阵和畸变参数等信息，构成当前帧的对象。

2. Track 函数，该函数是 Tracking.cc 中的主要函数，或称之为顶端函数。该函数首先判断系统状态，之后根据状态选择使用运动模型或里程计模型得到当前位姿，以及是否需要重定位；如果得到相机位姿和正确匹配，则依次在建立的 Map 中跟踪路标、将路标更新到画笔中、判断是否将该帧加入关键帧、清除里程计的匹配和未跟踪的 MP。

KeyFrame.cc，该类的操作都基于关键帧进行：

1. 共视图；确定关键帧之间的共视关系，该内容将被用于优化及回环检测，最终建立一个关键帧之间的共视图关系。

2. 地图点；还包括 MP 地图点的添加与移除、跟踪与匹配等；

3. 位姿；包括获取位姿、旋转矩阵 **R**，平移向量 **t** 等；

LoopClosing.cc，回环检测的主要函数：

1. 检测是否有新的关键帧；在此之前要给回环的队列加互斥锁，之后返回逻辑变量，表示是否有新的关键帧；
2. 检测是否有闭环；提取出一个关键帧，并用互斥锁加锁，防止被擦除；之后对地图进行判断，如果地图包含少于 10 个关键帧，则直接判定为没有闭环；使用具有共视关系的关键帧基于词袋模型进行评分，将共视关键帧得到的最高分作为最低分；最后删除其他的闭环候选帧，暂时找到与当前关键帧一致的关键帧。
3. 计算相似变换；Sim3 算法，其结果是得到两帧之间的相对位姿，只有 Sim3 求解器得到足够多的匹配点，才能接受该闭环帧。
4. 融合位姿图。

3.2.3 ORB-SLAM2 的配置及使用

ORB-SLAM2 是标准的 CMake 工程，这就意味着其可以使用 CMake 进行配置并且进行 make 编译；其虽然支持 ROS，但并不是 ROS 的 catkin 工程，使用的是 rosbuild 相关命令，而不是 ROS 常见的 catkin make 编译指令。

ORB-SLAM2 的 github 主页上，详细介绍了其编译的方法，官方将编译代码集成到了 build.sh 文件中；正常情况下，可以通过 chmod +x 命令给该 sh 文件赋予可执行权限后，运行 sh 命令，直接编译。如图3-7所示，需要编译的内容为两个第三方库和 ORB-SLAM2 主要函数；需要注意的是，make -j 命令指的是只用全部线程进行编译，这样做可能会造成电脑资源锁死，可以根据电脑的配置自行修改所用线程数。

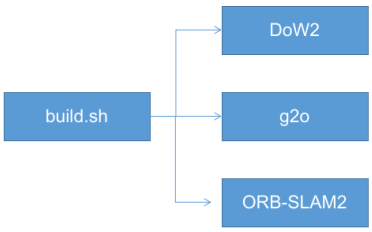

图 3-7 配置 ORB-SLAM2

ORB-SLAM2 在普通编译过程中，可能会遇到一些问题：

1. System.cc 中，usleep 未定义；这是由于 System.h 文件中缺少头文件 unistd.h 导致的，加上该头文件即可。

2. 找不到 Eigen3 库；大概率是因为 Pangolin 版本为 0.6，但 ORB-SLAM2 使用的 Pangolin 版本为 0.5；一种方法是将 Pangolin 版本退回 0.5，另一种方法是将 CMakeList.txt 中的 Eigen3 REQUIRED 改为 REQUIRED NO_MODULE。如果仍然找不到 Eigen3 库，如果是用 Eigen3 源代码编译安装的非模板类 Eigen3 库，则可以通过 ln -s 建立软链接的方式，在 ORB 需要找到 Eigen3 的位置添加上 Eigen3 的库。

3. Eigen3 的问题，在 ros 编译的 CMakeList 中也需要修改。

普通编译完成后，使用 TUM 数据集进行测试，从 System.cc 中可以得出，在终端中需要三个参数：ORB 字典的路径、相机配置文件的路径、数据集的路径。

由于需要在 ROS 环境下使用，首先执行 build_ros.sh。之后需要添加环境变量给 ROS 工作空间，即在.bashrc 文件中，添加上 ROS 功能包的路径，指向 Examples/ROS 文件夹。至此，ORB-SLAM2 在普通环境和 ROS 环境下均配置完成。

3.3 CCM-SLAM

CCM-SLAM 是由苏黎世联邦理工大学 Patrik Schmuck 开发，其含义是中心式协同单目视觉 SLAM。其特点是适用于多个设备（最多四个）同时运行，并且有一个终端负责管理控制和处理数据，是一种联合协同式的 SLAM 方案。

3.3.1 CCM-SLAM 的结构

CCM 在算法上使用了 ORB-SLAM2 的方法，保留了其跟踪、关键帧、关键帧数据库、Sim3 求解等类；与此同时，更加明确了自己的 Client+Server 机制，其核心是中心式的协同结构和各设备之间的通信方法。

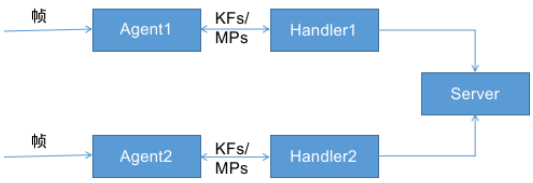

图 3-8 CCM-SLAM 框架

如图3-8所示，CCM 的核心是其将整个系统的 SLAM 进程分为了 Client 和 Server 两个模块处理。应用了网络中的客户端与服务器模型，这里的客户端指的是携带相机传感器的各设备终端，设备的种类可以是无人机、无人车、无人船甚至改进的手持相机，而服务器一般是一台中央处理电脑。

设备作为客户端，其并不是一个简单的相机加图片数据传输，而是一个有一定复杂的的运算系统。在客户端中，需要接收传感器得到的帧信息，并且做里程计运算，得到关键帧，通过通信模块将数据发送给服务器中负责接收信息的处理

单元；除此之外，在客户端中还存储有建立的局部地图，地图点保持与服务器的地图数据库更新。

关键帧和地图点的数据进入服务器的信息处理单元后，地图点直接进入服务器的地图存储库中；对于新的关键帧，如果能够检测到地图场景重叠，则直接进入到优化过程，并将优化后的结果存储到地图存储库中；该关键帧正常情况下会进入地图匹配和地图融合模块，之后再进入优化环节，随后存储到库中；如果所有客户端都到达相同的场景，则最后地图存储库只会有一张地图。

3.3.2 Client 与 Server 机制

CCM-SLAM 主要有两大板块的内容，一部分是地图的处理，另一部分是其客户端和服务器的机制及通讯方式。

客户端的设计可以分为三步，如图3-9所示，整体的流程是由 ClientNode 到 ClientSystem 再到 ClientHandler，其中的引用关系即为该流程。

图 3-9 Client 设计

ClientNode 负责的内容包括：ROS 节点初始化，句柄的设计，使用指针创建 ClientSystem。在 ROS 节点的初始化过程中，使用的节点名称是 CSLAM client node；值得注意的是其句柄的设计，一般的 ROS 句柄都实例化 NodeHandler，但其又实例化了 NodeHandler(" ")，该操作旨在给相机和地图等话题之前加上节点的名称，以于不同的终端作区分；最后使用了智能指针，创建了 ClientSystem 对象，意味着对该客户端创建了属于其自己的客户端系统。

```
ros::init(argc, argv, "CSLAM client node");

if(argc != 3)
{
cerr << endl << "Usage: rosrun cslam clientnode path_to_vocabulary path_to_cam_pa
ros::shutdown();
return 1;
}

ros::NodeHandle Nh;  // topic name will be: node name(only), like "/image_raw"
ros::NodeHandle NhPrivate("~");  // topic node will be: node name + topic name, l
// reference: <https://blog.csdn.net/weixin_44401286/article/details/112204903>
```

```
boost::shared_ptr<cslam::ClientSystem> pCSys{new cslam::ClientSystem(Nh,NhPrivate
```

ClientSystem 负责的内容包括：读取 ID，SLAM 初始化及预先数据准备。首先从 ROS 的参数服务器读取客户端 ID，使用 ROS 的 param 方法读取该数据；之后进行标准的 SLAM 数据预先准备流程，即加载词典、创建关键帧数据库、创建地图；最后进行初始化步骤，在这里引入了 ClientHandler。

```
int ClientId;
// get ClientID from launch file, actually ROS parameter server
mNhPrivate.param("ClientId",ClientId,-1);
mClientId = static_cast<size_t>(ClientId);  // assign ClientID to member of class

// load vocabulary
this->LoadVocabulary(strVocFile);

// Create KeyFrame Database
mpKFDB.reset(new KeyFrameDatabase(mpVoc));

// Create the Map
mpMap.reset(new Map(mNh,mNhPrivate,mClientId,eSystemState::CLIENT));
usleep(10000); //wait to avoid race conditions

// Initialize Agent
mpAgent.reset(new ClientHandler(mNh,mNhPrivate,mpVoc,mpKFDB,mpMap,
mClientId,mpUID,eSystemState::CLIENT,strCamFile,nullptr));
usleep(10000); //wait to avoid race conditions
mpAgent->InitializeThreads();
usleep(10000); //wait to avoid race conditions
```

ClientHandler 负责的内容包括：向总地图中添加该 ID 客户端的地图，定义 Sim3 转换，将该 ID 客户端的相机话题名加载到 ROS 的参数服务器，最后开始了具体的初始化。

3.3.3 CCM-SLAM 的配置

与 ORB-SLAM2 相同，CCM-SLAM 在其 github 上有详细的编译方法；与 ORB-SLAM2 不同的是，CCM-SLAM 是完全的 catkin 架构，完全基于 ROS，所以其必须被放置在 ROS 工作空间的 src 文件夹下，并且使用 catkin make 命令编译 CCM-SLAM 工程。需要注意的是最好不要使用全部线程进行编译。

3.4 多机协同及地图融合方案

地图的融合原理与回环检测类似，但不同于回环检测，地图融合在识别出场景重叠后，不止需要完成定位，还要对重叠场景的地图点完成归类和补充。

3.4.1 算法原理

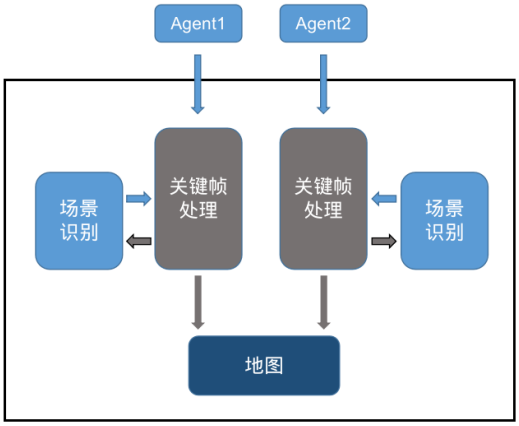

图 3-10 系统框架

地图融合系统的框架如图3-10所示：无人机 1 和无人机 2 将帧信息传递给系统的关键帧处理单元，关键帧处理单元对帧做出处理，首先筛选出其中的关键帧，没有入选的帧则被舍弃。

筛选出的关键帧随后首先进入到场景识别模块，该处的作用类似回环检测，即识别出相似的场景，这一点仍然使用基于外观的相似检测，即使用 BoW 的词袋模型。如果没有探测到场景重叠（注意，相邻的关键帧不会被并入考虑是否重叠，除了很大概率存在重叠外，在代码的逻辑中还需剔除共视关键帧的影响），则进入正常的优化和回环检测、建图等环节；如果探测到场景重叠，则需要视情况而定：

1. 若该场景与本机已经经历过的场景重合，则不进入地图拼合模块，直接进入优化和回环检测模块；对重叠的定义需要判断是完全重叠或部分重叠，若完全重叠则也会进入到回环检测模块；

2. 若该场景与本机的场景无重合，在与另一架无人机的场景识别中探测到重叠，则会进入到地图融合进程；

当确定需要地图拼合时，如图3-11所示，将两张图像中的地图点进行特征点匹配操作；对于图中的地图点，其在两个坐标系下，通过三角测量得到的坐标是不一致的，由此推得两个坐标系的转换关系，从而以一个无人机坐标系为主，将另一个坐标系中的点进行坐标变换后，重复的点舍弃，新增的点则按照坐标系转换关系补充到地图中，从而完成地图融合。

3.4.2 编程实现

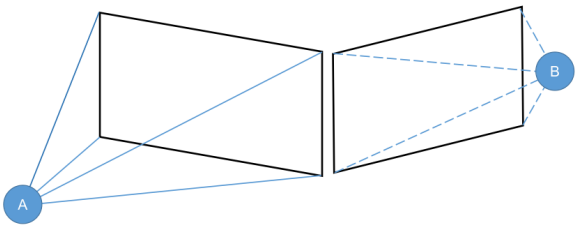

图 3-11 地图点匹配

第四章　多无人机 SLAM 仿真

本章主要多无人机 SLAM 的仿真；为了循序渐进，首先介绍仿真环境的配置，其次是单机的 SLAM 仿真，最后过渡到多机的 SLAM 仿真。

4.1　gazebo 仿真环境配置

在进行仿真之前，首先要对场景进行搭建，对 launch 文件进行配置。

4.1.1　场景

仿真环境中，场景的设计不能过于简单，墙壁、地面等大面积重复物体应该具有纹理，否则 ORB-SLAM2 的特征点提取会十分困难；除了纹理的设计，还应尽可能多地提供物体，使场景丰富。如图**??**所示：

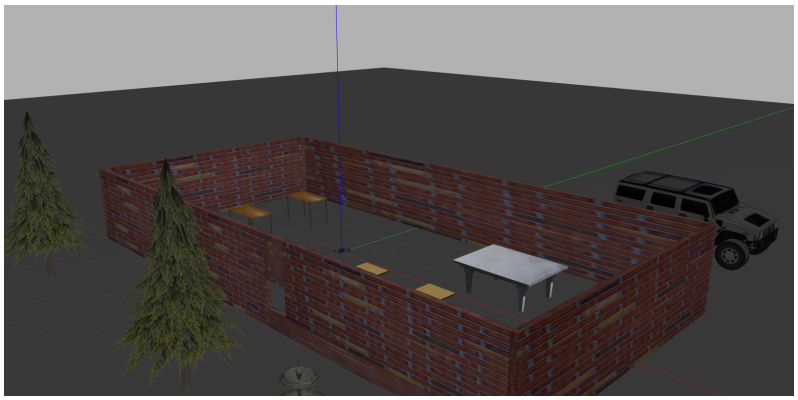

图 4-1 gazebo 场景示例

进入 gazebo 界面后，使用 Control+B 进入其编辑界面；之后有两种选择：

1. 建立基础模型，一般在这里会绘制上地面及其纹理；如果是室内场景，还会绘制一个大致的墙壁结构，墙壁的纹理，其上的门窗等；最后将该模型保存到.gazebo/model 文件中，这是 gazebo 模型的默认文件。

2. 直接选择插入模型；可以在此直接插入上文建立的基础模型，也可以插入其他模型库已有的模型，默认存储在/.gazebo/models 文件夹下

最后需要将自建场景存储为一个.world 文件，gazebo 不会给文件后缀的提示，需要自己输入后缀，这一点需要特别注意。在此之后，就获得了自建的 world 场景，下一步需要在 launch 文件中更改该项，完成调用。

4.1.2 launch 文件

2.1.2节中，曾简单介绍 launch 文件的功能。作为整个仿真环境的配置文件，launch 文件中基本包含了仿真所需的参数。

launch 文件使用 XML 标签语言书写，主要为确定启动的节点和加载参数使用，简单的开始节点、加载参数的方法如下：

```
<node pkg="your package name" name="your node name" type="your node type"/>
<param name="param name" value="param value"/>
<arg name="arg name" default="arg value"/>
```

在一个简单的 launch 文件中，主要会包括设备的信息设置、PX4 配置和 gazebo 仿真配置。

1. 设备的信息设置；这一部分包括设备的位姿，通过 x, y, z, R, P, Y 6 个变量表示三维位置坐标和滚转、俯仰、偏航姿态角；还包括设置设备的类型和名称，在四旋翼无人机仿真中使用 iris 作为参数 vehicle 的值，并且在 sdf 参数中，将 sdf 参数的值指向 iris 的 sdf 文件（一般在该位置，都使用默认参数加 find 指令去寻找软件在环仿真的 gazebo 模型路径）；最后是一些 gui 等参数的设置，默认使用官方设置的即可。

2. PX4 软件在环仿真的参数；在这里启动了名为 sitl 的节点，其参数指向了 EKF 设置的 rcS 文件；

3. gazebo 仿真参数配置；在这里需要修改 world_name 参数，其默认是 world 参数的值，所以实际上也可以直接修改 world 参数值，将其指向自建的 world 文件。

自此，launch 文件配置完成，可以使用 PX4 启动 launch 文件来进行简单的仿真。

4.2 单机 SLAM 仿真

在进行多机仿真前，首先要进行单机的 SLAM 仿真，为多机仿真做铺垫。单机 SLAM 仿真分为单机 Offboard 模式起降和航路点飞行，以及 SLAM 下的起降和航路点飞行四步。

4.2.1 launch 文件配置

4.1.2节中介绍了 launch 文件的详细配置方法，对于单机的 SLAM 仿真，配置方法基本与其相同，但是需要给 iris 无人机假装上双目相机。以下介绍给无人机加装相机的方法：

选择 mavros_posix_sitl.launch 文件，找到设备模型和世界配置区块，原始设置如下：

```
<!-- vehicle model and world -->
<arg name="est" default="ekf2"/>
<arg name="vehicle" default="iris"/>
<arg name="world" default="$(find mavlink_sitl_gazebo)/worlds/empty.world"/>
<arg name="sdf" default="$(find mavlink_sitl_gazebo)/models/$(arg vehicle)/$(arg
```

加载双目相机，也就是给 iris 无人机装上相机，需要在设备配置处，加上其附加配置的 sdf 文件；由于在这里只添加了相机的 sdf 文件，所以该附加 sdf 文件不需要手动修改，直接链接到相机上即可。因此需要新建 camera 参数，其值为 iris_stereo_camera，是 gazebo 自带的可以添加到 iris 无人机上的双目相机，修改后的 launch 文件部分如下：

```
<!-- vehicle model and world -->
<arg name="est" default="ekf2"/>
<arg name="vehicle" default="iris"/>
<!-- add stereo camera for iris -->
<arg name="my_camera" default="iris_stereo_camera"/>
<arg name="world" default="$(find mavlink_sitl_gazebo)/worlds/empty.world"/>
<!-- also need to revise sdf -->
<arg name="sdf" default="$(find mavlink_sitl_gazebo)/models/$(arg my_camera)/$(ar
<!-- <arg name="sdf" default="$(find mavlink_sitl_gazebo)/models/$(arg vehicle)/$
```

更改完 launch 文件的无人机配置后，可以试运行来检测，即 roslaunch 该 launch 文件。之后有两种方法可以检查，一是使用 rostopic list 命令，查找当前活跃的话题，如果 MAVROS 能顺利连接上双目相机，则会出现 image_raw 话题（对于双目分为左右，但对于单目仅有一个话题），这种情况下一般是成功的；另一种方法是使用 ROS 自带的 rqt_graph 命令，该命令可以以图的形式展示出，这种方式用途更广。装配成功后，仿真加载时应如图4-2所示：

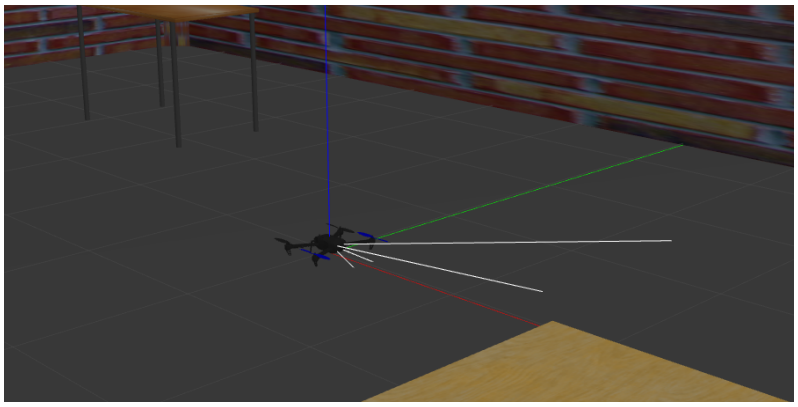

图 4-2 装配相机的 iris 无人机

需要注意，在需要与 MAVROS 建立通信的 launch 文件中，需要设置 fcu 端口号，具体表示为 udp+ 本地端口号，在多机的 launch 文件中，每一架飞机的端口号

都是不同的，但单机不需要作出修改。

如果设置双目相机参数后加载失败或加载到一个正方体模型，则是双目相机模型缺失导致的。一种方法是删除 gazebo 自带的 models 中的双目相机，并用 PX4 中的双目相机模型代替；另一种方法则是直接将 sdf 的值指向 PX4 的双目相机，即其值为 PX4 双目相机的路径。

4.2.2 Offboard 程序

完成双目相机的配置后，第二步是进入 Offboard 程序。与2.2.3的 Offboard 模式类似，首先目标是完成一套完整的起飞降落。需要注意，Offboard 模式中，所有消息指令都需要以大于 $2Hz$ 的频率发送，否则会激活安全生效机制，飞机返航。

1. 起飞的方式。动力学模型上的起飞即给四旋翼增加动力，保持平衡并提供向上的大于重力的升力,在程序中表示为发布位置指令,该消息类型为 MAVROS 的 geometry_msgs 数据类型。

2. 降落的方式。一般使用切换飞行模式的方法，将飞行模式切换到降落模式，选择降落模式中的自动降落即可。

降落的程序如下:

```
// proceed landing process
ROS_INFO("landing");
mavros_msgs::SetMode land_set_mode;
land_set_mode.request.custom_mode = "AUTO.LAND";
while (ros::ok()){
if( current_state.mode != "AUTO.LAND" &&
(ros::Time::now() - last_request > ros::Duration(5.0))){
if( set_mode_client.call(land_set_mode) &&
land_set_mode.response.mode_sent){
ROS_INFO("Land enabled");
}
last_request = ros::Time::now();
}
if(!current_state.armed){
break;
}

ros::spinOnce();
rate.sleep();
}
```

航路点飞行的简单实现方法为，依次发布各航路点的位置信息，但需要一个函数去判断飞机是否到达了航路点。函数的实现可以利用预计坐标和现处位置之间的欧式距离作评判标准，小于某值则认为到达航路点，实现的关键在现处位置消息的订阅。首先需要定义 current_pose 作为现处位置的变量，之后定义回调函数，获得该信息，实现的代码如下：

```
// record current pose
geometry_msgs::PoseStamped current_pose; /* NOLINT */

// callback function for Subscriber for local_pos_sub
void local_cb(const geometry_msgs::PoseStamped::ConstPtr& msg){
current_pose = *msg;
}
```

检查是否到达航路点的函数如下：

```
// check if reached a waypoint
bool check_waypoint(const geometry_msgs::PoseStamped &now_pose, const geometry_ms
// define Point to hold current position and aim position
geometry_msgs::Point curr, aim;
curr = now_pose.pose.position;
aim = aim_pose.pose.position;
double precision = 0.1;

// define return value
bool reach = false;

// calculate distance
double dist = sqrt(pow((curr.x - aim.x), 2) +
pow((curr.y - aim.y), 2) + pow((curr.z - aim.z), 2));
if(dist < precision){
reach = true;
ROS_INFO("reached waypoint!");
}

return reach;
}
```

前往下一个航路点的方法如下：

```
// set second waypoint
geometry_msgs::PoseStamped pose2;
pose2.pose.position.x = 2;
pose2.pose.position.y = 2;
pose2.pose.position.z = 2;

// heading for waypoint 2
while(ros::ok()){
// publish pose1 information
local_pos_pub.publish(pose2);
ros::spinOnce();

// check if reached a waypoint
if (check_waypoint(current_pose, pose2)) break;
rate.sleep();
}
```

按 2 个航路点飞行的地面站轨迹示意图如图**??**所示：

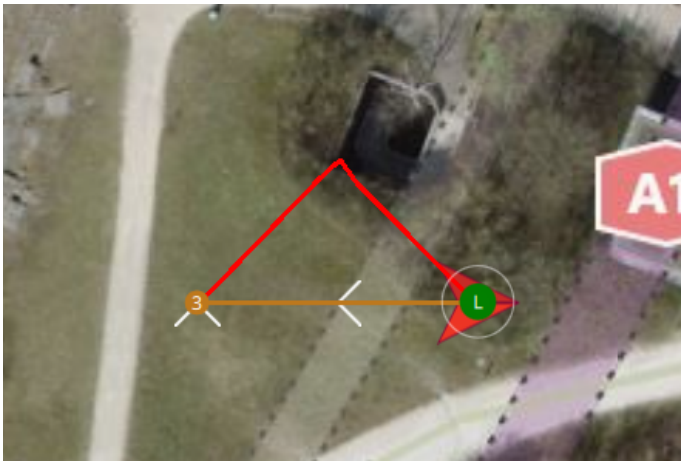

图 4-3 地面站的航路点飞行轨迹

4.2.3 视觉定位的坐标变换

在正常 GPS 定位的 Offboard 模式下，飞机的位置在 MAVROS 中作为已知量存在。但在视觉定位模式下，MAVROS 需要通过 vision_pose/pose 话题获得飞机的位姿信息。而 ORB-SLAM2 解算出的位姿并不是 MAVROS 的地理位置消息格式，因此需要做一些转换。

ORB-SLAM2 得到的外参 **Tcw** 矩阵为 4×4 矩阵：

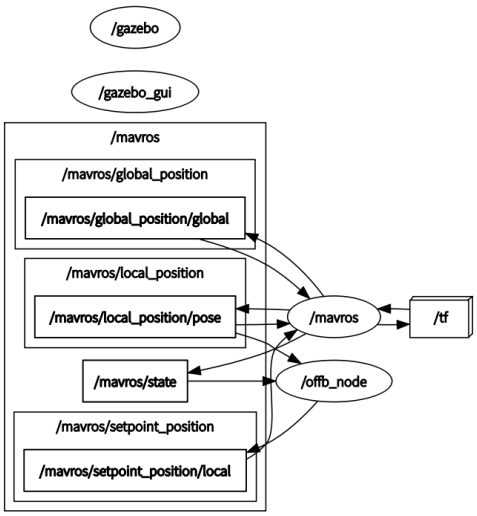

图 4-4 Offboard 节点话题关系

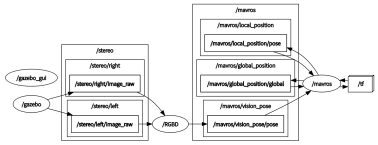

图 4-5 视觉 SLAM 节点话题关系

$$\mathbf{Tcw} = \begin{bmatrix} \mathbf{R_{cw}} & \mathbf{t_{cw}} \\ 0 & 1 \end{bmatrix}$$

其中，$\mathbf{R_{cw}}$ 为旋转矩阵，$\mathbf{t_{cw}}$ 为平移向量。

可以证明，从相机的外参矩阵得到相机位姿：

$$\mathbf{R_{wc}} = \mathbf{R_{cw}}^{T}$$
$$\mathbf{t_{wc}} = -\mathbf{R_{wc}} \cdot \mathbf{t_{cw}}$$

在此之后，还需要将 $\mathbf{R_{wc}}$ 和 $\mathbf{t_{wc}}$ 赋值给 ROS 中 tf 坐标系类型的 Transform 变量，之后通过调用 ROS 的 poseTFToMsg 函数，将 tf 类型的变量转为 MAVROS 的地理位置消息类型变量。实现的代码如下：

```
Rwc = Tcw.rowRange(0,3).colRange(0,3).t(); // Rotation information
twc = -Rwc*Tcw.rowRange(0,3).col(3); // translation information
vector<float> q = ORB_SLAM2::Converter::toQuaternion(Rwc);

tf::Transform new_transform;
new_transform.setOrigin(tf::Vector3(twc.at<float>(0, 0), twc.at<float>(0, 1), twc
```

```
tf::Quaternion quaternion(q[0], q[1], q[2], q[3]);
new_transform.setRotation(quaternion);

tf::poseTFToMsg(new_transform, pose.pose);
x = pose.pose.position.x;
y = pose.pose.position.y;
z = pose.pose.position.z;
pose.pose.position.x = z;
pose.pose.position.y = -x;
pose.pose.position.z = -y;
orb_pub->publish(pose);
```

4.2.4 单机仿真实验结果

进行起飞后 SLAM 建图的结果：

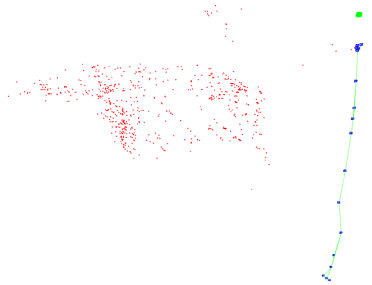

图 4-6 单机 SLAM 的效果

4.3 多机 SLAM 仿真

4.3.1 launch 文件配置

多机配置与单机配置最大的区别是，引用的子文件不同；单机引用 posix_sitl 文件，而多机则引用 vehicle_spawn 文件，而 spawn 文件是多机所独有的。

进入多机的 launch 文件，则需要引入组的概念。一个组使用一个命名空间，具有一套 MAVROS 的配置信息，多个组可以并行存在。因此在多机的 launch 文件设计中，一个组就是一架飞机，需要设置该组的命名空间，该组下的所有话题将共同使用该命名空间开头的话题。除此之外，还需要修改无人机的 ID，ID 默认从 0 到 10；之后修改 fcu 地址和 MAVLINK 的 udp 端口，一般在末尾加上该无人机的 ID 即可。完整的双机 launch 配置代码如下：

```
<!-- UAV0 -->
```

```
<group ns="uav0">
<!-- MAVROS and vehicle configs -->
<arg name="ID" value="0"/>
<arg name="fcu_url" default="udp://:14540@localhost:14580"/>
<!-- PX4 SITL and vehicle spawn -->
<include file="$(find px4)/launch/single_vehicle_spawn_rcs.launch">
<arg name="x" value="0"/>
<arg name="y" value="0"/>
<arg name="z" value="0"/>
<arg name="R" value="0"/>
<arg name="P" value="0"/>
<arg name="Y" value="0"/>
<arg name="vehicle" value="$(arg vehicle)"/>

<arg name="my_camera" value="iris_fpv_cam"/>

<arg name="mavlink_udp_port" value="14560"/>
<arg name="mavlink_tcp_port" value="4560"/>
<arg name="ID" value="$(arg ID)"/>
<arg name="gst_udp_port" value="$(eval 5600 + arg('ID'))"/>
<arg name="video_uri" value="$(eval 5600 + arg('ID'))"/>
<arg name="mavlink_cam_udp_port" value="$(eval 14530 + arg('ID'))"/>



</include>
<!-- MAVROS -->
<include file="$(find mavros)/launch/px4.launch">
<arg name="fcu_url" value="$(arg fcu_url)"/>
<arg name="gcs_url" value=""/>
<arg name="tgt_system" value="$(eval 1 + arg('ID'))"/>
<arg name="tgt_component" value="1"/>
</include>
</group>

<!-- UAV1 -->
```

```
<group ns="uav1">
<!-- MAVROS and vehicle configs -->
<arg name="ID" value="1"/>
<arg name="fcu_url" default="udp://:14541@localhost:14581"/>
<!-- PX4 SITL and vehicle spawn -->
<include file="$(find px4)/launch/single_vehicle_spawn_rcs.launch">
<arg name="x" value="2"/>
<arg name="y" value="0"/>
<arg name="z" value="0"/>
<arg name="R" value="0"/>
<arg name="P" value="0"/>
<arg name="Y" value="0"/>
<arg name="vehicle" value="$(arg vehicle)"/>

<arg name="my_camera" value="iris_fpv_cam"/>

<arg name="mavlink_udp_port" value="14561"/>
<arg name="mavlink_tcp_port" value="4560"/>
<arg name="ID" value="$(arg ID)"/>
<arg name="gst_udp_port" value="$(eval 5600 + arg('ID'))"/>
<arg name="video_uri" value="$(eval 5600 + arg('ID'))"/>
<arg name="mavlink_cam_udp_port" value="$(eval 14530 + arg('ID'))"/>



</include>
<!-- MAVROS -->
<include file="$(find mavros)/launch/px4.launch">
<arg name="fcu_url" value="$(arg fcu_url)"/>
<arg name="gcs_url" value=""/>
<arg name="tgt_system" value="$(eval 1 + arg('ID'))"/>
<arg name="tgt_component" value="1"/>
</include>
</group>
```

其默认每架无人机调用的是 spawn 文件，其中关键的模型组合有两种方式，一种是使用 urdf 进行模型配置，一种是使用新的 sdf。具体使用哪种视 PX4 的版本而定，由于 urdf 模型的配制方法要旧于 sdf，新版的 PX4 统一使用 sdf 进行 gazebo

仿真的模型配置，并且使用 jinja.py 脚本生成 sdf 模型配置。

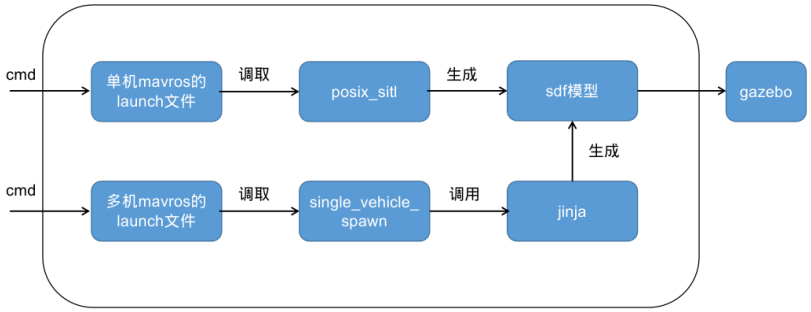

图 4-7 仿真模型生成流程

图4-7展示了 PX4 中使用 gazebo 生成模型的方法，决定了修改模型的方法。可以看出，launch 文件分为顶层和底层文件；平常直接使用 roslaunch 命令启动的一般为顶层文件，而顶层文件中调取的一般为底层文件。对于单机和多机，其调用的底层文件不同；单机调取 posix_sitl.launch 文件，而多机则调取 single_vehicle_spawn 文件。二者的区别是单机的底层文件直接使用了即有的 sdf 模型，但多机的底层文件基于 spawn 繁殖机制，重新调用 jinja 生成了新的 sdf 模型，并且在该过程中对 MAVLINK 的许多参数作了配置，这一点是单机所没有的。最终都生成 gazebo 可以加载出的模型文件。

旧版本使用 urdf 版本的 PX4，如果在 launch 文件中直接将其修改为 sdf 格式生成的模型，一般会报错 jinja.py 文件中有很多参数未定义，一种简单的处理方式是直接将新版本的 janja 替换到旧版本中去（为什么要使用旧版本，因为新版本的稳定性可能存在一些问题），但是这样做可能会连带产生一些附加问题，尽量选择不修改 jinja 这种较为底层的文件。

由于 ROS 的机制，话题如果重复，则会报错，并用时间戳最新的话题替换掉时间戳更旧的话题；如果多架无人机的话题不加以区分，可能会导致最终话题重复，从而只有一架无人机可以使用。

4.3.2 多机编队处理

多机的控制可以使用编队控制，控制方法可以分为两种：集中式和分布式。

1. 集中式即集群中存在领机，其他飞机需要跟随领机的运动轨迹。这种方式的实现相对简单，需要各机实时订阅领机的位置，并且始终和领机保持设定好的队形，也就是相对位置。

2. 分布式则没有领机的概念，各无人机按照各自设定好的航路点行进，该方式的实现方法主要需要依靠多线程的设计。

当无人机数量较多时，可以采用分级跟随的办法，假设有 N 架无人机，则构建一个 $N \times N$ 邻接矩阵 T，这里以 $N = 5$ 为例：

$$T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

若 $N_{i,j} > 0$，则 i, j 之间会建立通信，如图4-8所示：

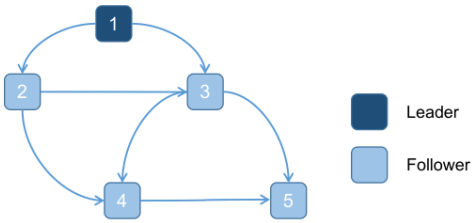

图 4-8 编队通信方法

有 $N_{2,1}, N_{3,1}, N_{3,2}, N_{4,2}, N_{4,3}, N_{5,3}, N_{5,4}$ 均不为 0，则在这些对之间，按由小到大的顺序建立通信，传输长机的位置信息，保持相对位置的跟随。

在无人机发生编队队形变换时，则需要考虑最小总路径，可以用 KM 匈牙利算法解得每一架飞机对应的最小路径，并且在队形变换过程中需要考虑碰撞的影响。

图4-9展示了三机编队的仿真结果：

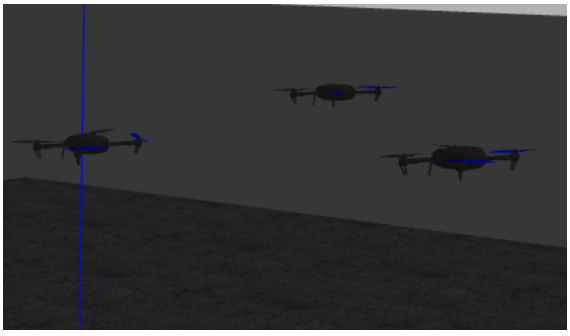

图 4-9 三机编队仿真

4.3.3 多机 SLAM 仿真

多机 SLAM 仿真除 PX4 的配置外，还要对 SLAM 的 launch 文件进行配置。其中包含了设置相机的参数文件，更重要的是配置各无人机所接受的相机话题。

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

```
<arg name="dist" default="0"/>
<arg name="cam" default="$(find ccmslam)/conf/px4_sitl.yaml"/>

<group ns="ccmslam">

<node pkg="tf" type="static_transform_publisher" name="linkC0_broadcaster" args="

<node pkg="ccmslam" type="ccmslamClientNode" name="ccmslamClientNode0" args="$(fi

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++ -->
<!-- Agent Specific Params - !!!MUST BE ADJUSTED!!! -->

<param name="~FrameId" type="string" value="odomC0" />
<param name="~ClientId" type="int" value="0" />

<param name="~TopicNameCamSub" type="string" value="/iris0/usb_cam/image_raw" />

<param name="~MapInTopicName" type="string" value="MapOutServer0" unless="$(arg
<param name="~MapInTopicName" type="string" value="MapOutServer0Disturbed" if="$(

</node>

</group>
</launch>
```

启动时需要启动服务器和客户端，然后打开 ORB-SLAM 节点，最终结果如图：

第五章　实验与评估

5.1　真机配置与实验

5.2　地图融合实验

第六章　总结与展望

6.1　全文总结

6.2　对未来工作的展望

参考文献

[1] 于琰平. 基于 FlightGear 的四旋翼无人机三维可视仿真系统研究 [J]. 天津大学, 2010.

[2] 刘鹏. 基于 FlightGear 的无人直升机飞行仿真技术研究 [J]. 南京航空航天大学, 2011.

[3] 李海泉. 小型无人机飞行力学建模及虚拟训练平台的建立 [J]. 南京航空航天大学, 2011.

致　谢

另外要感谢 Curt Olson 等飞行爱好者们，是他们创造了 FlightGear 这个功能强大的开源的飞行模拟软件；感谢为 Linux 贡献代码的程序员们，这个自由免费的平台为我完成毕设提供了不少便利；感谢清华大学王磊博士，他创作的 LaTeX 模板使我的论文的排版得以顺利完成。

毕业设计小结