1.实验名称及目的

1.1实验名称

基于Python和C++的跨平台分布式传感器数据采集与ROS发布实验

1.2实验目的

本实验旨在构建一个完整的分布式机器人仿真传感器数据处理系统，主要目标包括：

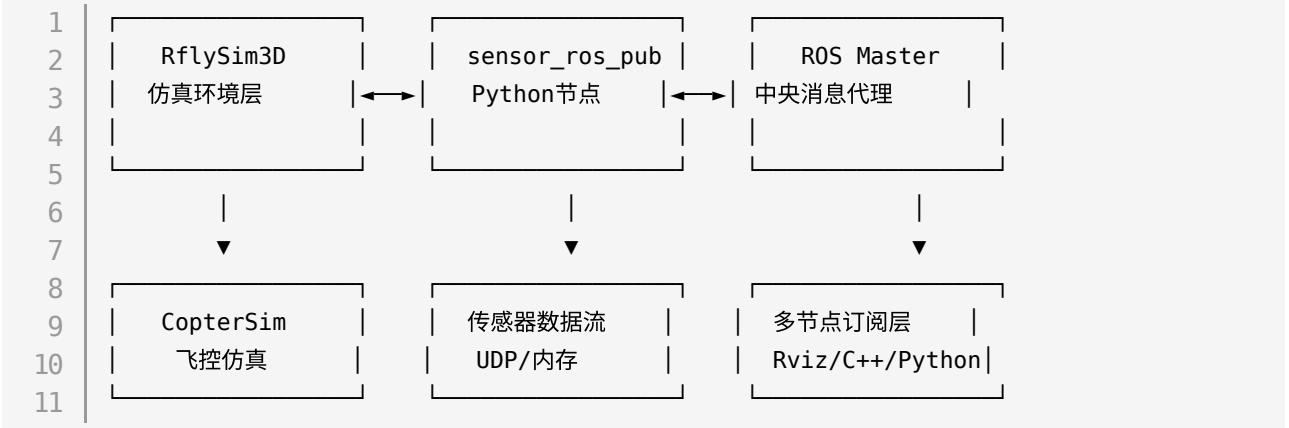
1. **分布式仿真架构**：通过Python接口在Linux远端系统中请求RflySim3D发送多种传感器数据
2. **跨平台数据采集**：接收RGB图像、深度图像、激光雷达点云等多模态传感器数据
3. **ROS生态系统集成**：将传感器数据无缝转发到ROS话题空间，支持ROS1和ROS2双版本
4. **C++高性能处理**：使用C++实现传感器数据的高效接收、处理和显示
5. **跨ROS版本兼容**：C++程序支持ROS1和ROS2的统一代码库实现
6. **实时可视化**：使用OpenCV进行传感器数据的高性能实时显示

1.3关键知识点

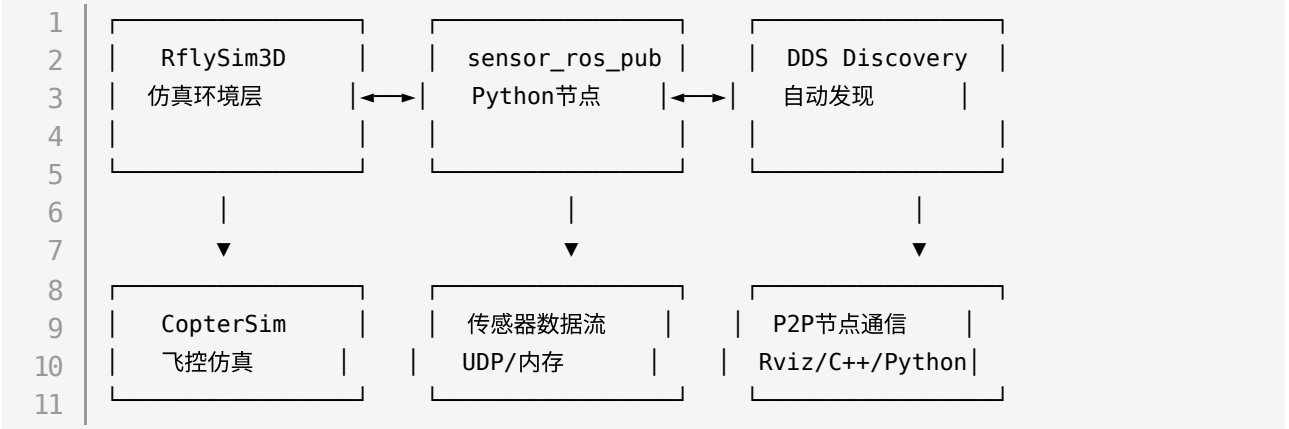
系统架构概述

本实验构建了一个多层次的分布式传感器数据处理架构，支持ROS1和ROS2两种不同的通信模式：

ROS1架构（中央式）：



ROS2架构（分布式DDS）：



核心技术知识点

知识点0：C++传感器数据处理架构

C++程序核心组件：

- **sensor_show_node**：主程序入口，负责自动发现和订阅传感器ROS话题
- **SensorShowNode类**：核心数据处理类，实现跨ROS版本的统一接口
- **智能类型识别**：根据话题名称自动判断传感器类型和数据格式

核心类设计架构：

```cpp
class SensorShowNode {
public:
    // 传感器数据类型枚举
    enum class DataType: uint8_t {
        RGB = 0, DEPTH = 1, GRAY = 2,
        INFRARED_RGB = 3, INFRARED_GRAY = 4,
        PointCloud2 = 5
    };

    // 构造函数：根据ROS版本创建不同的订阅者
    SensorShowNode(const std::vector<std::pair<std::string, DataType>> topic_list);

    // 回调函数：处理不同类型的传感器数据
    void ImageCallback(...);
    void PointCloudCallback(...);

    // 运行主循环
    void Run();
};
```

跨ROS版本编译策略：

```cpp
#ifdef USE_ROS1
    ros::NodeHandle nh_;
    std::vector<ros::Subscriber> image_subs;
#else
    rclcpp::Node::SharedPtr nh_;
    std::vector<rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr>
image_subs;
#endif
```

知识点1：分布式网络通信架构

自动IP发现机制：

- 使用`ReqCopterSim.py`接口实现局域网内仿真节点的自动发现
- 支持本地（127.0.0.1）和远程（192.x.x.x）两种运行模式
- 通过UDP广播实现零配置网络连接

网络拓扑优势：

```python
# 自动获取仿真电脑IP，实现网络透明
req = ReqCopterSim.ReqCopterSim()
TargetIP = req.getSimIpID(StartCopterID)  # 自动发现目标IP
req.sendReSimIP(CopterID)  # 请求数据回传
```

注意事项：

- 多实验并行时可能产生IP冲突，建议使用手动IP配置
- 需要确保防火墙允许相关端口通信

知识点2：多协议传感器数据传输

SendProtocol传输协议详解：

- `SendProtocol[0] = 0`：共享内存模式（仅Windows，最高性能）
- `SendProtocol[0] = 1`：UDP + PNG压缩（跨平台，平衡性能）
- `SendProtocol[0] = 2`：UDP无压缩（最低延迟，高带宽需求）
- `SendProtocol[0] = 3`：UDP + JPEG压缩（最低带宽，适度质量损失）

传输性能对比：

协议模式	延迟	带宽占用	图像质量	跨平台性
共享内存	极低	无网络	无损	仅Windows
UDP+PNG	低	中等	无损	全平台
UDP原始	极低	极高	无损	全平台
UDP+JPEG	中等	极低	有损	全平台

激光雷达数据特殊处理：

- 仅支持共享内存（0）和UDP直传（1）两种模式
- 点云数据无法压缩，建议使用千兆网络

知识点3：跨ROS版本兼容性设计与架构差异

ROS1 vs ROS2 核心架构对比：

特性	ROS1 (Noetic)	ROS2 (Humble)
通信架构	中央式 (ROS Master)	分布式 (DDS)
服务发现	依赖Master节点	自动P2P发现
消息传输	TCP/UDP	DDS (RTPS)
实时性	有限支持	原生实时支持
容错性	单点故障风险	高可用性
安全性	基础认证	内置安全机制

架构启动差异：

- **ROS1**：必须先启动 `roscore` 作为消息代理和服务注册中心
- **ROS2**：节点直接通过DDS自动发现和通信，无需中央节点

编译时版本选择：

```
#ifdef USE_ROS1
    // ROS1 Noetic代码路径 - 需要NodeHandle
    #include <ros/ros.h>
    ros::NodeHandle nh;  // 连接到ROS Master
#else
    // ROS2 Humble代码路径 - 直接创建节点
    #include "rclcpp/rclcpp.hpp"
    auto node = rclcpp::Node::make_shared("sensor_node");  // DDS自动发现
#endif
```

运行时环境适配：

```
ros_version = os.getenv("ROS_VERSION")
if ros_version == "1":
    import rospy
    rospy.init_node("rflysim_subscriber")  # 注册到ROS Master
else:
    import rclpy
    rclpy.init()  # 初始化DDS环境
    node = rclpy.create_node("rflysim_subscriber")  # 创建DDS节点
```

通信机制差异：

```
# ROS1: 通过Master节点中转
rospy.Publisher('/topic', String, queue_size=10)

# ROS2: 直接DDS点对点通信
node.create_publisher(String, '/topic', 10)
```

知识点4：传感器数据类型与话题映射

完整传感器支持矩阵：

```python
sensor_type_mapping = {
    # 图像传感器类别
    1: "img_rgb",            # RGB彩色相机
    2: "img_gray",           # 单色灰度相机
    3: "img_depth",          # 深度相机
    4: "img_Segmentation",   # 语义分割相机
    8: "fisheye",            # 鱼眼相机
    9: "img_cine",           # 吊舱相机
    40: "img_Infrared_Gray", # 红外灰度相机
    41: "img_Infrared",      # 红外彩色相机

    # 点云传感器类别
    20: "vehicle_lidar",     # 车载激光雷达
    21: "global_lidar",      # 全局激光雷达
    22: "livox_lidar",       # Livox激光雷达
    23: "mid360_lidar",      # Mid360激光雷达
    7:  "Depth_Cloud",       # 深度点云
}
```

ROS话题命名规范：

```
/rflysim/sensor{SeqID}/{sensor_type}
例如：
/rflysim/sensor1/img_rgb       # 1号RGB相机
/rflysim/sensor2/mid360_lidar  # 2号Mid360激光雷达
/rflysim/sensor1/camera_info   # 相机内参信息
```

知识点5：实时性能优化策略

数据流水线优化：

1. **异步数据采集**：传感器数据采集与ROS发布分离
2. **缓存机制**：实现帧缓存避免数据丢失
3. **线程隔离**：图像处理与网络传输使用独立线程
4. **内存池**：预分配内存减少动态分配开销

性能调优参数：

```jsonc
{
    "DataCheckFreq": 30,     // 采集频率：平衡实时性与性能
    "DataWidth": 640,        // 分辨率：根据计算能力调整
    "DataHeight": 480,       // 分辨率：根据计算能力调整
    "SendProtocol": [1,0,0,0,0,0,0,0]  // 传输协议选择
}
```

核心接口使用说明

1）Python传感器数据获取接口

VisionCaptureApi核心功能：

```python
# 传感器初始化和配置
vis.jsonLoad()  # 加载Config.json中的传感器配置
isSuss = vis.sendReqToUE4(0, TargetIP)  # 向RflySim3D发送取图请求
vis.startImgCap()  # 开启传感器数据采集循环

# 数据状态检查和获取
vis.hasData[i]  # 检查第i号传感器数据是否更新
vis.Img[i]      # 获取第i号传感器的像素数据矩阵

# 直接显示（可选，ROS发布模式下一般不使用）
cv2.imshow('Img'+str(i), vis.Img[i])  # 使用OpenCV显示图像
```

2）C++传感器数据处理接口

SensorShowNode核心功能：

```cpp
// 自动传感器话题发现和订阅
std::vector<std::string> topics = getROSTopics();
for (const auto& topic : topics) {
    if (topic.find("/sensor") != std::string::npos) {
        // 根据话题名称判断传感器类型
        auto sensor_type = identifySensorType(topic);
        createSubscriber(topic, sensor_type);
    }
}

// 图像数据处理回调
void ImageCallback(const sensor_msgs::Image::ConstPtr& msg,
                   const DataType type, const int index) {
    cv_bridge::CvImagePtr cv_ptr = cv_bridge::toCvCopy(msg, getImgCodec(type));
    cv::imshow("Image" + std::to_string(index), cv_ptr->image);
    cv::waitKey(1);
}

// 点云数据处理回调
void PointCloudCallback(const sensor_msgs::PointCloud2::ConstPtr& msg) {
    std::cout << "Received PointCloud2: " << msg->width * msg->height
              << " points." << std::endl;
}
```

2） ReqCopterSim接口使用（自动获取ip接口）

```
req = ReqCopterSim.ReqCopterSim() \# 获取局域网内所有CopterSim程序的电脑IP列表

TargetIP = req.getSimIpID(StartCopterID) \#自动获取CopterSim的StartCopterID号程序所在电脑的IP，作为目标IP。这里获取CopterSim所在仿真电脑的IP。

vis = VisionCaptureApi.VisionCaptureApi(TargetIP) \#创建一个视觉传感器实例，这个实例对应的ip号为TargetIP

req.sendReSimIP(CopterID) \# 请求mavlink数据到本电脑
```

3）相机数量和参数配置

其中，视觉传感器的初始状态由本文件夹下的Config.json决定，主要包含以下配置项：

```
"SeqID":0：使用自动更新ID的方式，创建了SeqID为0的视觉传感器

"TypeID":20：传感器类型为激光雷达

"TargetCopter":1：相机绑定在1号飞机上

"SendProtocol":[1,0,0,0,0,0,0,0]：传输模式为1：UDP网络传输模式（图片使用jpeg压缩，点云直传）。

"SensorPosXYZ":[0,0,-0.3]：相机分布位置。
```

4）飞机控制指令

```
MavList = MavList+[PX4MavCtrl.PX4MavCtrler(CopterID,TargetIP)] \#初始化并建立i号飞机的
MAVLink通信连接，连接上远端的电脑飞机，创建一个飞机控制实例列表mavlist

MavList[i].InitMavLoop() \# 初始化Mavlink监听程序，读取第i个飞机数据

MavList[i].initOffboard() \# 第i个飞机进入Offboard模式

Error2UE4Map = Error2UE4Map+[-np.array([mav.uavGlobalPos[0]-
mav.uavPosNED[0],mav.uavGlobalPos[1]-mav.uavPosNED[1],mav.uavGlobalPos[2]-
mav.uavPosNED[2]])# 用于获取车辆的全局位置，同时一个 Python 列表转换为一个 NumPy
数组，便于进行计算处理

MavList[i].SendPosNED(0, 0, -8, 0) \# 对第i个飞机发送8米高的位置控制指令

mav=MavList[j] \# 从列表中取第i个实例作为当下处理的飞机实例

MavList[i].endOffboard() \# 对第i个飞机进入endoffboard模式

MavList[i].stopRun() \# 终止第i个飞机
```

2.实验效果

本实验实现了完整的分布式传感器数据处理流水线：

- **C++高性能处理**：实现了跨ROS版本的传感器数据实时处理和显示
- **智能话题发现**：C++程序能够自动识别和订阅传感器相关的ROS话题
- **多模态数据支持**：同时处理RGB、深度图像和激光雷达点云数据
- **实时可视化**：使用OpenCV实现传感器数据的高效实时显示

3.文件目录

例程目录：
[安装目录]\RflySimAPIs\2.RflySimUsage\0.ApiExps\e13_VisAPIPyCpp\1.rosCppExps

文件夹/文件名称	类型	说明
核心python模块		
sensor_ros_pub.py	辅助	Python传感器数据获取及ROS转发程序

文件夹/文件名称	类型	说明
C++传感器处理模块		
sensor_show/	功能包	C++传感器数据显示模块（跨ROS版本兼容）
├── src/main.cpp	源码	主程序入口，自动发现和订阅传感器话题
├── src/sensor_show.cpp	源码	传感器数据处理核心逻辑
├── src/sensor_show.h	头文件	类定义和接口声明
├── CMakeLists.txt	构建	跨ROS版本的CMake构建配置
├── build.sh	脚本	智能构建脚本，自动检测ROS版本
├── package_ros1.xml	配置	ROS1版本的包描述文件
└── package_ros2.xml	配置	ROS2版本的包描述文件
自动化脚本		
RunRflysim3DSITL.bat	启动	一键启动RflySim3D仿真环境
RunWslRosCore.bat	启动	WSL环境下启动ROS核心服务
RunWslRosTrans.bat	启动	WSL环境下运行Python传感器发布程序
WinWslCppBuild.bat	构建	一键编译C++传感器显示模块
WinWslCppSensorShow.bat	启动	智能启动C++传感器显示程序（自动检测编译状态）

文件夹/文件名称	类型	说明
WinWSL.bat	环境	启动WSL环境和X11图形转发服务
配置文件		
Config.json	配置	传感器参数配置（分辨率、频率、传输协议等）

4.运行环境

4.1 软件要求

Windows 10及以上版本；RflySim工具链；Visual Studio Code；Linux（Ubuntu 20.04）；Linux（Ubuntu 20.04）。

> ①：若使用Pixhawk 6X飞控，平台安装时的编译命令为：px4_fmu-v6x_default，推荐PX4固件版本为：1.12.3。其他配套飞控及编译命令请见：https://rflysim.com/doc/zh/1/Hardware.html

4.2 硬件要求

笔记本/台式电脑① 1台；WinWSL 1台；虚拟机/视觉盒子/其他板卡 可选台。

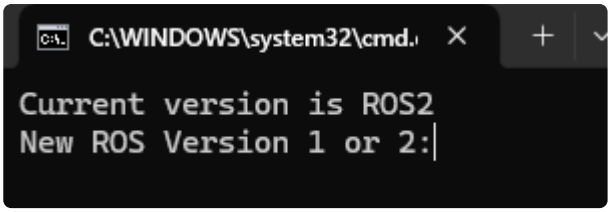
> ①：推荐配置请见：https://rflysim.com/

5.实验步骤

5.1前置准备步骤

Step 0：配置ROS环境

在`\Desktop\RflyTools`中找到双击运行“RosSwitch.bat”，切换需要的ROS版本



ROS版本选择说明：

- **选择ROS1**：需要双击运行`RunWslRosCore.bat`脚本启动ROS Master节点
- **选择ROS2**：无需启动核心服务，DDS会自动处理节点发现和通信

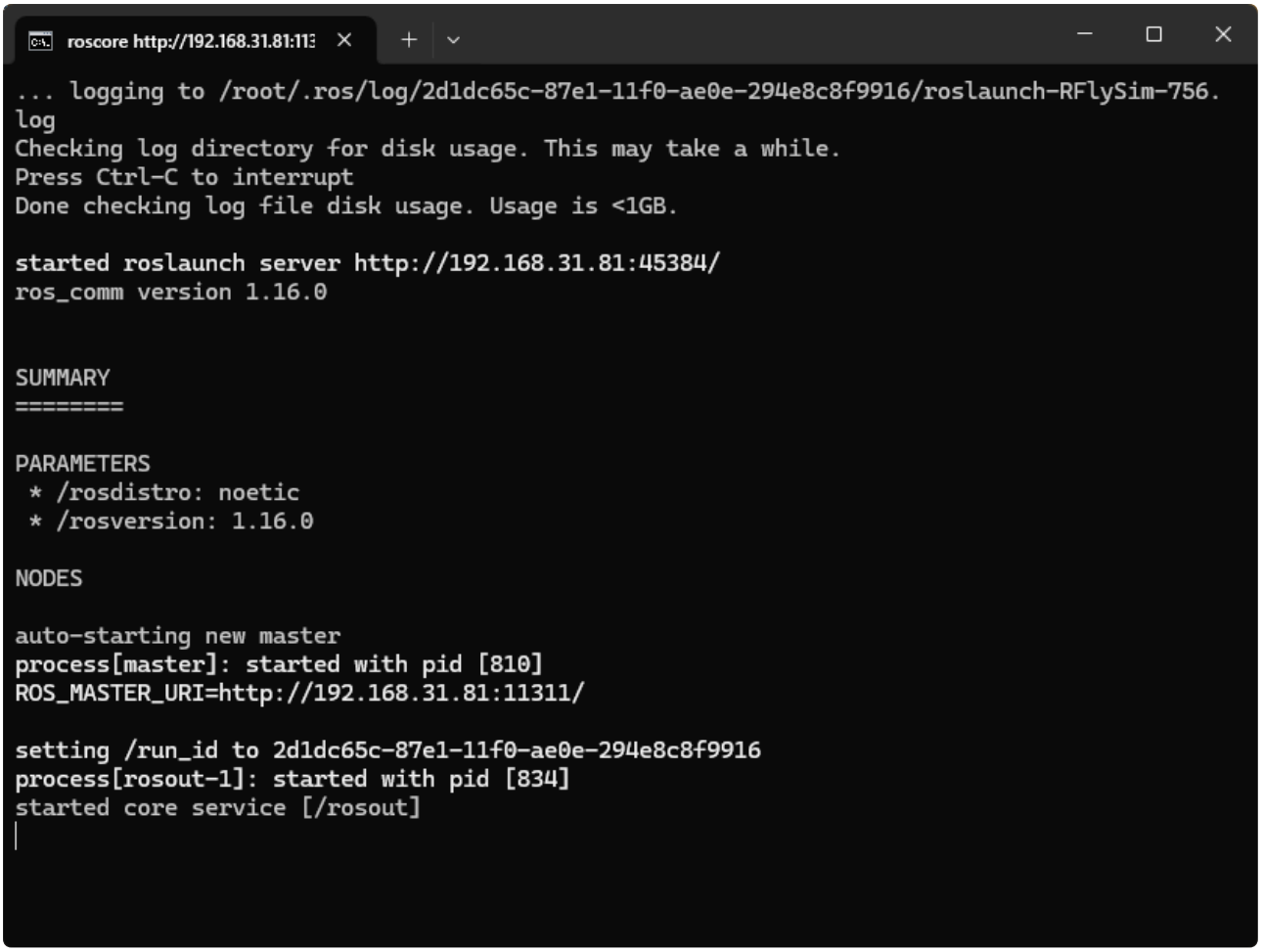
ROS1模式 - 启动核心服务：

功能说明：
在WSL环境中启动ROS Master作为中央消息代理和服务注册中心

备选方法：
也可双击`WinWSL.bat`进入WSL环境，然后手动输入`roscore`命令

预期结果：
ROS Master成功启动，终端显示节点注册信息和参数服务器状态

ROS2模式说明：
ROS2使用DDS (Data Distribution Service) 实现分布式通信，节点间可直接发现和通信，无需中央Master节点。系统启动后，各节点会自动通过DDS协议进行服务发现和建立通信链路。



Step 1：编译C++传感器显示模块

在开始实验前，需要先编译C++传感器显示功能包。直接双击启动WinWslCppBuild.bat完成编译或者启动WinWSL按如下步骤手动配置

构建ROS工作空间：

```
mkdir -p ws/src
```

拷贝sensor_show功能包：

将 `sensor_show` 功能包拷贝到 `ws/src/` 目录下

编译功能包：

在工作空间根目录下打开终端，执行编译脚本：

```
./src/sensor_show/build.sh
```

> **注意**：编译脚本会根据当前ROS环境自动选择ROS1或ROS2的编译方式。

5.2ROS图像转发步骤

Step 2：启动RflySim3D仿真环境

操作方法：

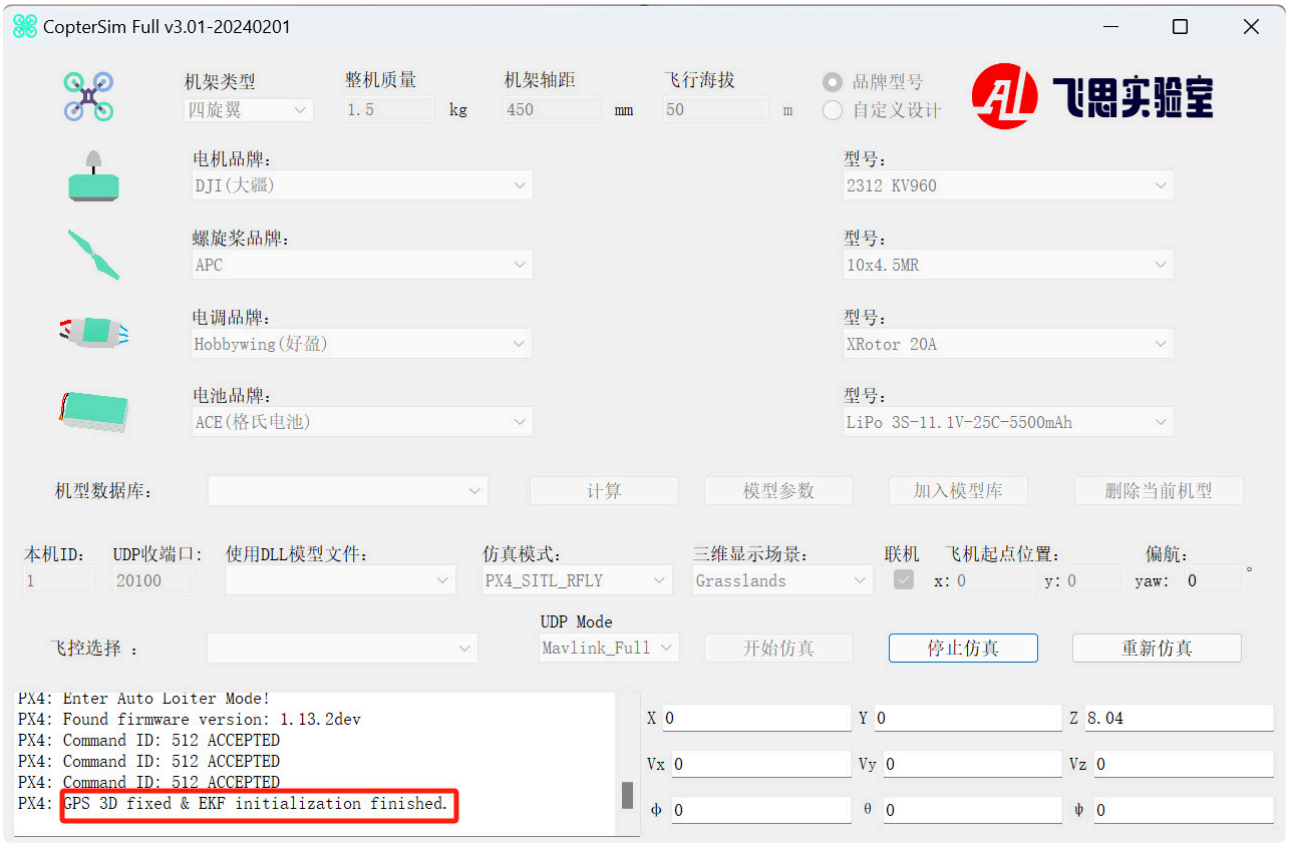
双击运行`RunRflysim3DSITL.bat`脚本

预期结果：

- 启动1个QGC地面站
- 启动1个CopterSim仿真软件
- 启动1个RflySim3D三维仿真环境
- CopterSim软件下侧日志栏显示`GPS 3D fixed & EKF initialization finished`
- RflySim3D软件内显示1架无人机模型

状态检查：

确保所有仿真组件正常运行，无人机初始化完成



Step 3：启动传感器数据发布节点

操作方法：

双击运行`RunWslRosTrans.bat`脚本

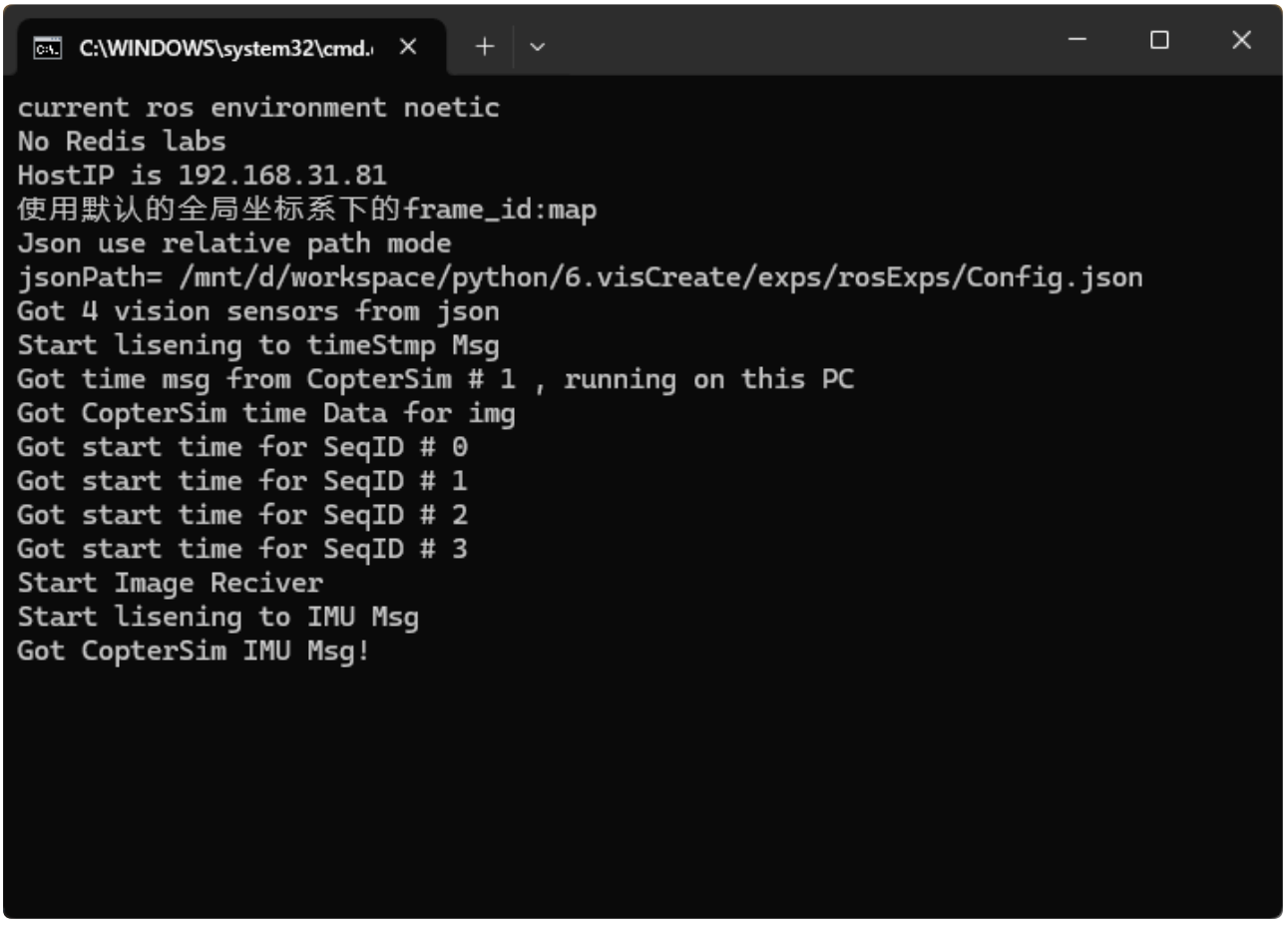
功能说明：

- 运行`sensor_ros_pub.py`Python程序
- 从RflySim3D获取传感器数据（RGB图像、激光雷达点云等）
- 控制无人机起飞到指定高度
- 将传感器数据发布到ROS话题

核心流程：

1. 自动获取仿真电脑IP地址
2. 建立与RflySim3D的通信连接
3. 加载传感器配置（Config.json）
4. 启动传感器数据采集循环
5. 控制无人机执行起飞任务
6. 持续发布传感器数据到ROS话题

备选方法：
双击`WinWSL.bat`进入WSL环境，手动执行`python3 sensor_ros_pub.py`



5.3C++核心传感器处理步骤

Step 4：启动C++传感器数据处理节点

一键启动方式：
双击启动`WinWslCppSensorShow.bat`脚本自动处理传感器数据

手动启动方式：
按照如下步骤手动启动C++传感器处理程序

核心功能说明：

- **高性能处理**：C++实现的传感器数据实时处理和显示
- **跨ROS版本**：单一代码库同时支持ROS1和ROS2
- **智能识别**：自动发现传感器话题并识别数据类型
- **实时可视化**：使用OpenCV定实时显示图像数据

环境配置：
根据使用的ROS版本配置环境变量：

```
# ========== ROS1环境配置 ==========
source devel/setup.bash

# ========== ROS2环境配置 ==========
source install/setup.bash
```

程序运行：
根据ROS版本选择对应的运行命令：

```
# ========== ROS1运行命令 ==========
rosrun sensor_show sensor_show_node

# ========== ROS2运行命令 ==========
ros2 run sensor_show sensor_show_node
```

预期功能：

- 自动发现并订阅传感器相关ROS话题
- 使用OpenCV显示接收到的图像数据(本例中包含RGB图像和深度图)
- 在终端输出点云数据统计信息（本例中包含360雷达数据）
- 验证传感器数据的完整性和实时性

实验技巧与注意事项

启动顺序建议

建议按照以下顺序启动各个组件，以确保系统稳定运行：

ROS1启动序列：

1. 首先启动ROS Master：运行`RunWslRosCore.bat`
2. 启动RflySim3D仿真环境
3. 等待无人机完全初始化后，启动传感器数据发布节点
4. 最后启动可视化工具和接收节点

ROS2启动序列：

1. 直接启动RflySim3D仿真环境（无需ROS Master）
2. 等待无人机完全初始化后，启动传感器数据发布节点
3. 启动可视化工具和接收节点（DDS自动处理节点发现）

注意： ROS2的分布式架构使得节点启动顺序更加灵活，但建议仍按上述顺序以确保数据流的连续性。

网络连接提示

- 如果使用多台电脑进行分布式仿真，请确保所有设备在同一局域网内
- 检查防火墙设置，确保相关端口未被阻塞
- 如果IP自动获取失败，可以手动配置目标IP地址

C++程序性能优化建议

编译优化：

- 使用Release模式编译：`catkin_make -DCMAKE_BUILD_TYPE=Release`
- 启用编译器优化：`-O3 -march=native`参数
- 使用多线程编译：`catkin_make -j$(nproc)`

运行时优化：

- **内存管理**：预分配图像缓冲区，减少动态内存分配
- **CPU亲和性**：设置进程优先级和CPU亲和性绑定
- **显示优化**：调整OpenCV显示窗口大小和刷新率

系统调优：

- 对于配置较低的电脑，可以降低传感器的分辨率和帧率
- 在Config.json中调整`DataCheckFreq`参数来控制传感器数据更新频率
- 使用合适的传输协议（UDP压缩vs直传）平衡质量与性能
- 调整ROS话题队列大小和订阅者的QoS设置

6.参考资料

1. RflySim官方文档： https://rflysim.com
2. ROS官方教程： http://wiki.ros.org
3. OpenCV-Python教程： https://opencv-python-tutroals.readthedocs.io
4. PX4飞控固件： https://px4.io

7.常见问题

Q1：启动仿真后，无人机无法正常起飞怎么办？

A1：检查以下几点：

- 确保CopterSim日志显示"GPS 3D fixed & EKF initialization finished"
- 检查QGC地面站中的传感器状态是否正常
- 确认MAVLink连接建立成功
- 检查无人机是否已解锁（armed）

Q2：终端中看不到传感器数据显示怎么办？

A2：请按ROS版本分别排查：

ROS1排查步骤：

- 确认ROS Master正在运行：检查`roscore`进程
- 确认传感器数据发布节点正常运行
- 检查ROS话题是否正确发布：`rostopic list`
- 验证话题数据：`rostopic echo /rflysim/sensor1/img_rgb`
- 检查节点连接状态：`rosnode list`和`rosnode info /sensor_node`

ROS2排查步骤：

- 确认DDS环境正常：`ros2 daemon status`
- 检查可用话题：`ros2 topic list`
- 验证话题数据：`ros2 topic echo /rflysim/sensor1/img_rgb`
- 检查节点状态：`ros2 node list`和`ros2 node info /sensor_node`
- 验证QoS兼容性：`ros2 topic info /rflysim/sensor1/img_rgb --verbose`

Q3：编译C++传感器处理程序时出现依赖错误怎么办？

A3：请按以下步骤检查和解决：

基础依赖检查：

- 确保已安装ROS开发环境：`sudo apt install ros-$ROS_DISTRO-desktop-full`
- 安装OpenCV开发库：`sudo apt install libopencv-dev libopencv-contrib-dev`
- 安装cv_bridge：`sudo apt install ros-$ROS_DISTRO-cv-bridge`
- 安装传感器消息库：`sudo apt install ros-$ROS_DISTRO-sensor-msgs`

编译环境配置：

- 设置环境变量：`source /opt/ros/$ROS_DISTRO/setup.bash`
- 检查ROS版本：`echo $ROS_VERSION`
- 清理旧的编译文件：`rm -rf build/ devel/ install/`

高级问题解决：

- CMake版本问题：`sudo apt install cmake build-essential`
- 编译器问题：`sudo apt install gcc-9 g++-9`
- 如果使用自定义OpenCV：设置`OpenCV_DIR`环境变量

Q4：WSL环境中无法显示图形界面怎么办？

A4：需要配置X11转发：

- 确保VcXsrv X Server正在运行
- 在WSL中设置DISPLAY环境变量：`export DISPLAY=:0`
- 检查防火墙是否允许VcXsrv通信
- 重启WSL后重新尝试

Q5：多台电脑联机仿真时连接失败怎么办？

A5：请检查网络配置：

- 确保所有电脑在同一局域网内
- 检查IP地址获取是否正确
- 关闭Windows防火墙或添加例外
- 使用ping命令测试网络连通性
- 手动指定IP地址而不使用自动获取