实验12：pixhawk软件仿真

**一、环境配置**

本实验基于Linux系统下进行，那么首先需要在电脑上配置linux环境。linux系统有很多中不同的版本，在此推荐使用Ubuntu 16.04版本的系统。在Windows系统上配置linux环境一般有两种途径，其一是通过vmware软件在虚拟机上安装了Ubuntu，可以参考如下链接，进行相关的配置：

<https://blog.csdn.net/stpeace/article/details/78598333>

其二是通过安装windows和Ubuntu双系统的方式，配置链接可以参考如下的链接：

<https://blog.csdn.net/qq_31442743/article/details/80859237>

这两种方式都可以，在此推荐采用安装双系统的方式。在对Ubuntu系统进行配置的时候需要需要满足如下的要求：

/home ：不少于20G的空间。

/swap：Ubuntu系统的内存大小，不小于4G，建议给8G，否则编译的时候会非常耗时，也不能顺畅运行软件仿真。

**二、pixhawk开发环境配置**

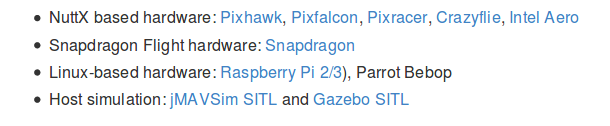
这部分内容在pixhawk的官网上的描述很清楚，原链接：<http://dev.px4.io/starting-installing-linux.html>

中文翻译的网站链接：<https://fantasyjxf.gitbooks.io/px4-wiki/content/index.html>

其他参考网站链接：<https://www.ncnynl.com/category/uav-pixhawk-quadrotor/>

Pixhawk开发环境配置，可参考的资源很多，在此仅对关键步骤，以及可能出现的问题进行简单阐述：

1. PX4的软件架构可以在多种硬件开发板上运行，如图1所示：

图1. PX4支持的硬件

我们使用的是Pixhawk硬件开发板，在图1中很明确指出，它是NuttX based hardware，所谓的NuttX为一种嵌入式操作系统。在配置的时候，要注意这一点。

2. 在进行基于NuttX的硬件配置的时候。如果出现gcc-arm-none-eabi版本导致的PX4/Firmware编译错误，请参考[the bare metal installation instructions](https://fantasyjxf.gitbooks.io/px4-wiki/content/1_Getting-Started/adcanced_linux.html" \l "toolchain-installation) 手动安装4.9或者5.4版本的arm-none-eabi工具链。

3. 代码下载的步骤，严格按照使用：git clone 的命令进行。不能够使用浏览器下载，亦不能通过拷贝下载好的软件包的方式取代下载过程。否则会出现编译错误。

4. 在下载安装包的过程中，要保持网络的顺畅。

**三、四旋翼飞机的软件仿真**

**1、相关的说明**

**官网链接：**[**https://dev.px4.io/v1.9.0/en/simulation/**](https://dev.px4.io/v1.9.0/en/simulation/)

PX4支持两种软件模拟的方式，其一为：PX4支持**软件在环路（SIT​​L）模拟**，其中飞控代码完全在一台计算机上运行（或者在同一台计算机上或在同一网络上的另一台计算机）；其二为：PX4支持**硬件在环（HITL）仿真**，即使用使用一个真正的飞行控制器板仿真固件。

PX4支持的软件模拟器有很多种，例如： Gazebo，jMAVsim，AirSim，XPlane，Simulation-In-Hardware。

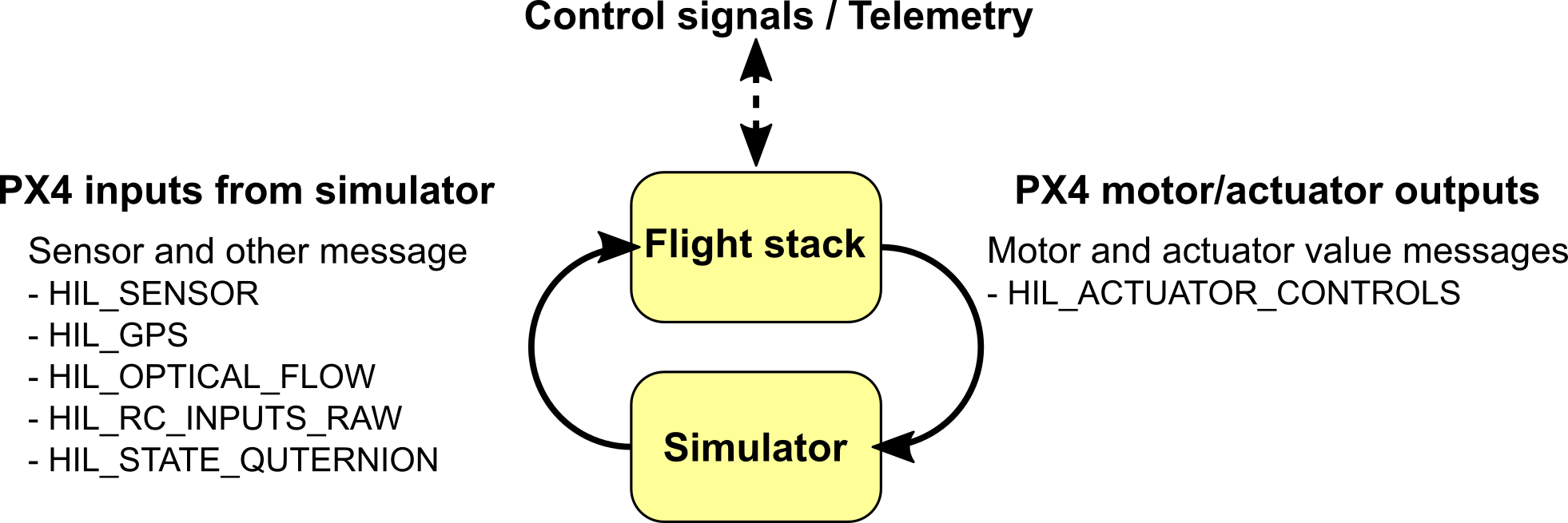
 在本实验种，使用的是Gazebo，这是目前最重用的软件模拟器。Gazebo是自主机器人强大的3D仿真环境，特别适用于对测试机器人的壁障算法和计算机视觉算法的仿真。在本实验中，需要基于Gazebo，完成通过上位机PC给下位机pixhawk发送命令，使得四旋翼飞机完成一定的动作。模拟器将所有的这些动作，在上位机通过GUI显示出来。这样做的目的在于，能够通过软件模拟的方式提前验证代码的效果和可靠性，否则如果直接在硬件上执行，很危险，而且不便捷。在官网上可以找到图2所示的模拟软件架构图。

图2. 软件模拟的架构图

在本实验中，Control signals即为上位机PC发送的命令，Telemetry为数控电台的信号，在本实验中不作介绍，有兴趣的可自己查阅文献实践。

**2.安装**MAVROS

通过PC机或者其他嵌入式开发板子给pixhawk发送命令的开发叫做对飞控板的外部控制(offboard control)。PX4提供了一套基于ROS的mavlink开发包，即为mavros。

上文中提到的MAVLink（Micro Air Vehicle Link，微型空中飞行器链路通讯协议）是无人飞行器与地面站（Ground Control Station ，GCS）之间通讯，以及无人飞行器之间通讯最常用的协议。它已经在APM、PIXHAWK等飞控平台上进行了大量测试。mavros可以看作是对mavlink的封装，使得后者可以更加方便地被使用，省去了开发这对于mavlink协议的理解然后再编程实现的过程。

安装mavros的步骤，可以参考以下链接：

<https://dev.px4.io/v1.9.0/en/ros/mavros_installation.html>

安装完成mavros之后，参考以下链接，构建一个最简单的外部控制程序：

<https://dev.px4.io/v1.9.0/en/ros/mavros_offboard.html>

对外部控制程序进行一些解释：

|  |
| --- |
| #include <ros/ros.h>  #include <geometry\_msgs/PoseStamped.h>  #include <mavros\_msgs/CommandBool.h>  #include <mavros\_msgs/SetMode.h>  #include <mavros\_msgs/State.h> |

.mavros\_msgs包含MAVROS包中提供的服务（service）和话题（topic）所需的一切自定义消息。所有服务和话题以及相应的消息类型可参照文档[mavros wiki](http://wiki.ros.org/mavros)。

.我们创建一个简单的回调函数来保存飞控的当前状态。我们可以用它检查连接状态，解锁状态以及外部控制标志。

|  |
| --- |
| ros::Subscriber state\_sub = nh.subscribe<mavros\_msgs::State>("mavros/state", 10, state\_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"); |

.实例化一个用来发布被控制的本地位置的发布器，以及适当的客户端来请求解锁和模式更改。注意，对你自己的系统，"mavros"前缀部分会有所不同，它依赖于对应节点的launch文件中定义的名字。

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

.px4飞行栈的两个机外（offboard）控制指令之间有500ms的时限。如果超过了时限，commander指令将会切换回进入机外控制模式前的上一个模式。这正是为什么发布频率必须高于2Hz的原因，并且还要考虑可能的延迟。这也是我们推荐从位置控制（POSCTL）模式进入机外控制模式的原因。这样一来，如果飞机意外脱离了机外控制模式，飞机将会停在当前轨道并悬停。

|  |
| --- |
| // wait for FCU conn\*\*\*\*\*\*\*\*/ection  while(ros::ok() && current\_state.connected){  ros::spinOnce();  rate.sleep();  } |

.在发布任何东西之前，我们需要等待MAVROS和飞控建立连接。一旦接收到心跳消息[heartbeat message](https://en.wikipedia.org/wiki/Heartbeat_message)，该循环就会立即退出。以上代码是以一定频率（20Hz）来执行ROS消息回调函数，即[ros::spinOnce()](http://wiki.ros.org/roscpp/Overview/Callbacks and Spinning).

|  |
| --- |
| geometry\_msgs::PoseStamped pose;  pose.pose.position.x = 0;  pose.pose.position.y = 0;  pose.pose.position.z = 2; |

.即使px4飞行栈在航空NED坐标系中运行，MAVROS仍然会将这些坐标转换到标准的ENU坐标系，反之亦然。这是我们将Z设置为+2的原因。

|  |
| --- |
| //send a few setpoints before starting  for(int i = 100; ros::ok() && i > 0; --i){  local\_pos\_pub.publish(pose);  ros::spinOnce();  rate.sleep();  } |

.在进入机外控制模式之前，就必须开始发送设定值（这里是指pose），否则模式切换会被拒绝。这里的100是一个随意选取的值。

|  |
| --- |
| mavros\_msgs::SetMode offb\_set\_mode;  offb\_set\_mode.request.custom\_mode = "OFFBOARD"; |

.设置自定义模式为OFFBOARD

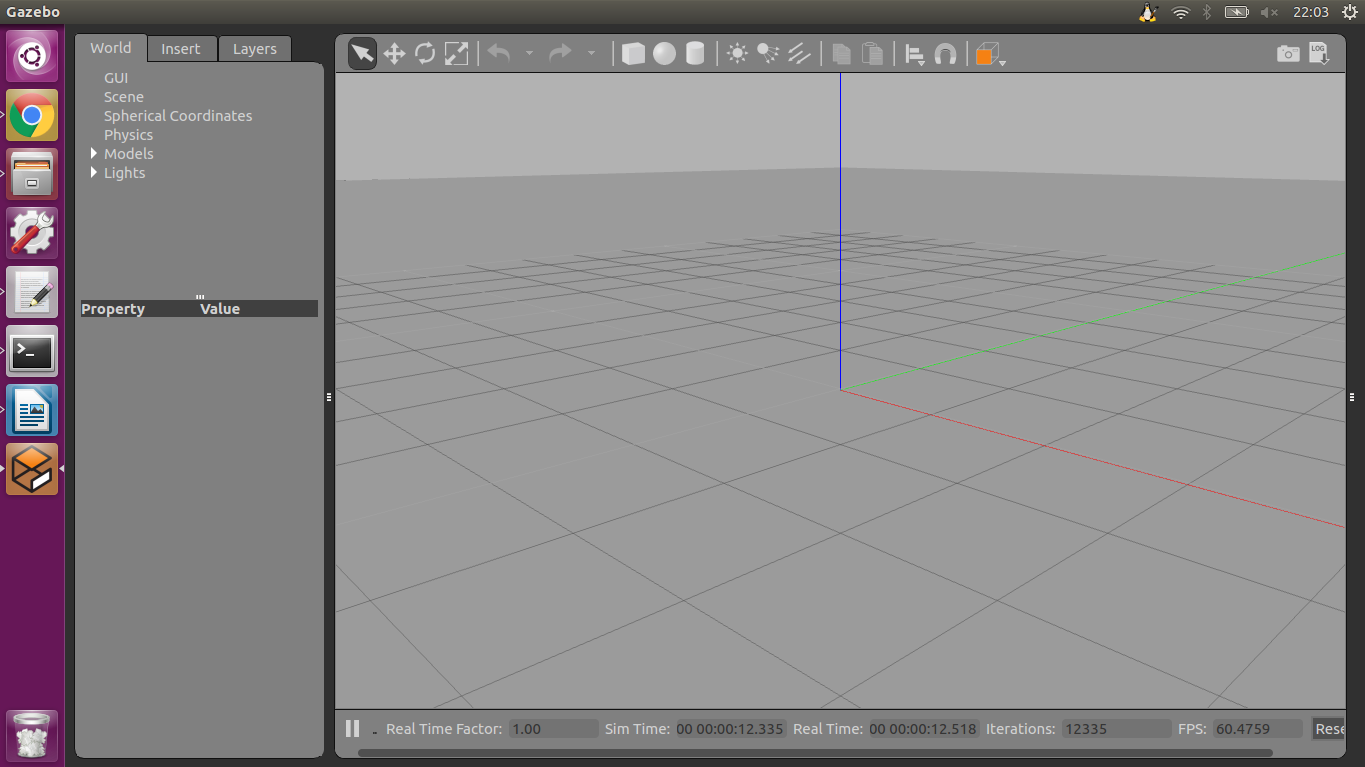
|  |
| --- |
| mavros\_msgs::CommandBool arm\_cmd;  arm\_cmd.request.value = true;  ros::Time last\_request = ros::Time::now();  while(ros::ok()){  if( current\_state.mode != "OFFBOARD" &&  (ros::Time::now() - last\_request > ros::Duration(5.0))){  if( set\_mode\_client.call(offb\_set\_mode) &&  offb\_set\_mode.response.mode\_sent){  ROS\_INFO("Offboard enabled");  }  last\_request = ros::Time::now();  } else {  if( !current\_state.armed &&  (ros::Time::now() - last\_request > ros::Duration(5.0))){  if( arming\_client.call(arm\_cmd) &&  arm\_cmd.response.success){  ROS\_INFO("Vehicle armed");  }  last\_request = ros::Time::now();  }  }  local\_pos\_pub.publish(pose);  ros::spinOnce();  rate.sleep();  } |

.剩下的代码比较好理解。我们试图在解锁旋翼允许它起飞后，将它切换至机外控制模式。为了避免大量请求堵塞飞控，我们设置服务调用间隔时间为5秒。在同一个循环里，继续以合适的频率持续发布设定的pose。

**3.安装gazebo**

一般安装的ROS就已经自己安装了gazebo，如果已经安装了ROS，可在terminal中输入

gazebo即可出现图3所示的效果：



**图3. Gazebo的可视化界面**

若没有安装gazebo，则按下面方法安装即可：

|  |
| --- |
| sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb\_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'  wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -  # 首先更新源  sudo apt-get update  # 安装Gazebo7  sudo apt-get install gazebo7  # 开发者需要安装额外的包  sudo apt-get install libgazebo7-dev |

**4.源码编译gazebo仿真**

因为我们的飞控用的是pixhawk，它的开源代码是自带仿真的，所以我们可以在其仿真环境的基础上修改，以满足我们的需求。

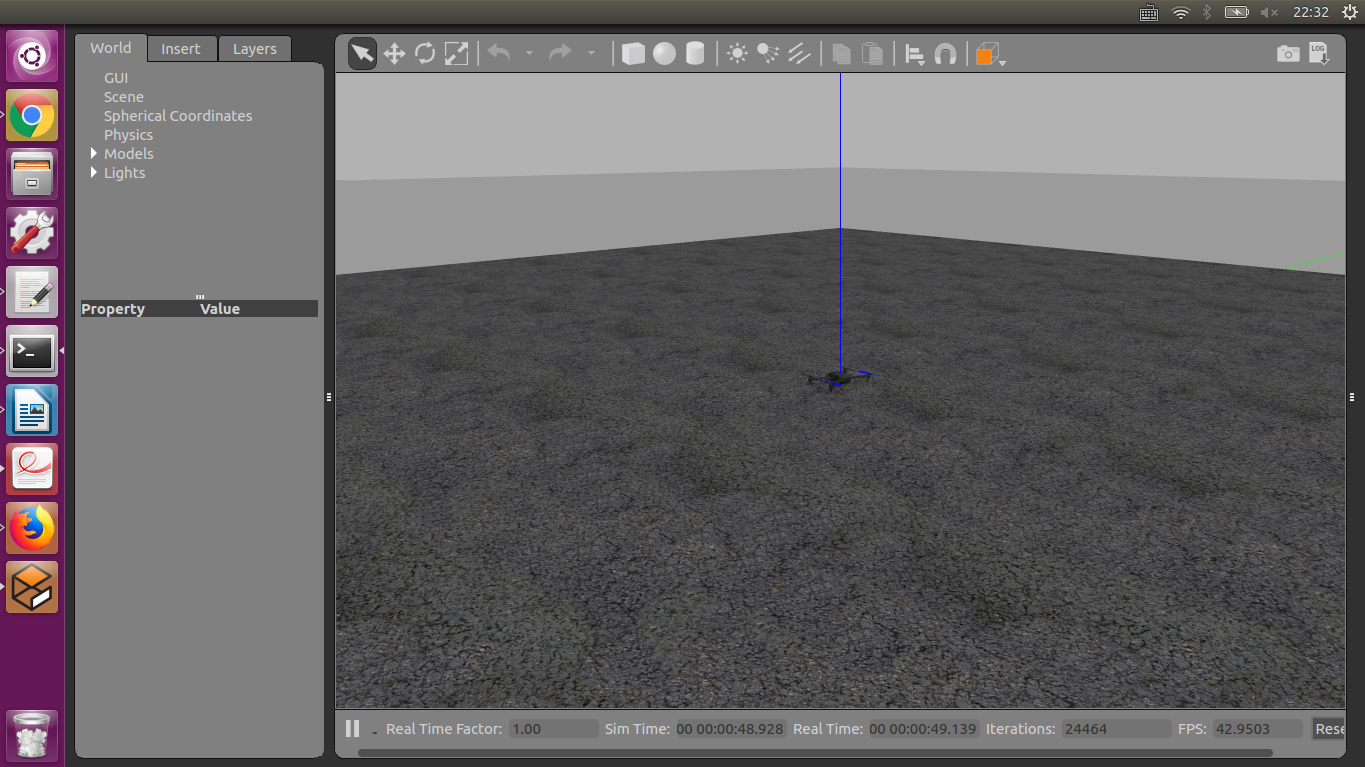
|  |
| --- |
| # 源码下载  mkdir -p ~/src  cd ~/src  git clone https://github.com/PX4/Firmware.git  cd Firmware  git submodule update --init --recursive  cd ..    # 安装交叉编译器  arm-none-eabi-gcc --version  sudo apt install gcc-arm-none-eabi    # 编译源码  cd Firmware  make px4fmu-v2\_default    # 配置gazebo  ## 安装库  sudo apt-get install libprotobuf-dev libprotoc-dev protobuf-compiler libeigen3-dev  ## 在源码gazebo仿真文件夹中创建build文件夹  cd ~/src/Firmware/Tools/sitl\_gazebo  mkdir Build  ## 添加路径  ### 设置插件的路径以便 Gazebo 能找到模型文件  export GAZEBO\_PLUGIN\_PATH=${GAZEBO\_PLUGIN\_PATH}:$HOME/src/Firmware/Tools/sitl\_gazebo/Build  ### 设置模型路径以便 Gazebo 能找到机型  export GAZEBO\_MODEL\_PATH=${GAZEBO\_MODEL\_PATH}:$HOME/src/Firmware/Tools/sitl\_gazebo/models |

|  |
| --- |
| ### 禁用在线模型查找  export GAZEBO\_MODEL\_DATABASE\_URI=""  ### 设置 sitl\_gazebo 文件夹的路径  export SITL\_GAZEBO\_PATH=$HOME/src/Firmware/Tools/sitl\_gazebo  ### 编译gazebo仿真  cd Build  cmake ..  make |

**5.运行gazebo**

**打开一个terminal，依次执行以下语句：**

|  |
| --- |
| cd /PX4代码包的路径  make posix\_sitl\_default  source ~/catkin\_ws/devel/setup.bash // 只有mavros是用源码编译的才需要  source Tools/setup\_gazebo.bash $(pwd) $(pwd)/build\_posix\_sitl\_default  export ROS\_PACKAGE\_PATH=$ROS\_PACKAGE\_PATH:$(pwd)  export ROS\_PACKAGE\_PATH=$ROS\_PACKAGE\_PATH:$(pwd)/Tools/sitl\_gazebo  roslaunch px4 mavros\_posix\_sitl.launch |

出现下图所示的界面，即为成功：

**6.通过mavros控制飞机**

**在一个新的terminal中运行上一节中建立好的offboard\_node.cpp**

|  |
| --- |
| cd catkin\_ws  catkin\_make  source devel/setup.bash //每新开一个terminal都要有这个，否则就会出错  rosrun offboard offboard\_node |