
2018 控制工程专业学位研究生实验课程
-控制系统综合设计与实验

实 验 报 告

姓 名：_____

学 号：_____

年 级：_____2018 级硕_____

任课教师：_____

时 间：_____2019 年 1 月 18 日_____

目录

一、实验目的.....	3
二、实验内容.....	3
2.1 实验硬件设备	3
2.2 实验软件平台	3
2.3 实验主要内容	3
三、实验原理.....	4
3.1 ROS 操作系统.....	4
3.2 全向移动机器人平台	5
3.3 Kinect 视觉传感器.....	6
3.4 二维 SLAM 建图	7
四、实验设计.....	8
五、实验过程.....	8
5.1 ROS 操作系统的安装与环境搭建.....	8
5.2 全向移动平台的键盘操控	10
5.3 Kinect 深度信息转化为激光雷达数据.....	12
5.4 二维 SLAM 建图	13
六、实验结果.....	15
七、总结与展望.....	16
附录.....	17

一、实验目的

- 1) 了解并熟悉 Ubuntu 系统的使用与操作方法，掌握在 Ubuntu 系统下安装机器人操作系统（ROS）的方法；
- 2) 理解 ROS 的文件系统，掌握 ROS 架构组成及相关的概念，学会创建功能包、节点，发布与订阅话题等基本操作，并能够使用相关调试工具对系统进行调试；
- 3) 了解全向移动机器人平台的硬件构成，掌握机器人平台的串口通信协议并实现与机器人平台的通信；
- 4) 掌握 Kinect 视觉传感器的使用方法，包括驱动安装、数据获取与应用等；
- 5) 了解二维 SLAM 建图的原理，掌握相关 SLAM 算法及其应用；
- 6) 完成基于全向移动机器人平台和 Kinect 的二维 SLAM 建图综合实验；通过对实验环境搭建、编程、调试及实验结果分析等环节，加深对相关理论知识及应用技能的理解和掌握，锻炼学生独立解决技术难题的能力。

二、实验内容

2.1 实验硬件设备

实验过程中主要的硬件设备有：笔记本电脑、全向移动机器人平台、Kinect 视觉传感器及相关连接线材等。

2.2 实验软件平台

实验中使用 Ubuntu16.04 操作系统，在该系统下安装 kinetic 版本的 ROS 环境进行实验相关内容的开发。ROS 工程项目的开发使用 RoboWare Studio 这款开发软件工具。

2.3 实验主要内容

基于全向移动机器人平台和 Kinect 的二维 SLAM 建图综合实验主要包括以下四部分内容：

- 1) 安装 ROS 操作系统，搭建完整的软件实验环境；
- 2) 基于 ROS 操作系统编写节点程序，实现通过电脑键盘按键操控移动机器人平台运动的功能；
- 3) 获取 Kinect 视觉传感器的图像信息，并通过对图像的信息的处理实现

Kinect 模拟激光雷达的效果；

4) 利用全向移动机器人平台和 Kinect 对周围环境进行二维的 SLAM 建图。

三、实验原理

3.1 ROS 操作系统

ROS (Robot Operating System) 是一个广泛应用于机器人系统的软件框架，它包含了一系列的软件库和工具用于构建机器人应用。从驱动到最先进的算法，以及强大的开发者工具，ROS 包含了开发一个机器人项目所需要的所有东西，且都是开源的，这样无须改动就能够在不同的机器人上复用代码，避免了重复的劳动。

ROS 一般概念中的操作系统 (如 Windows、Linux 等) 并不相同。像 Windows 这类操作系统为我们管理计算机的物理硬件资源 (CPU、内存、磁盘等)，并提供如读文件、写文件、创建进程、创建线程及启动线程这样的操作。而 ROS 所工作的层级并没有这么低，它基于一般概念中的操作系统来运行，官方推荐基于 Ubuntu (Linux) 运行，并在 Ubuntu 操作系统提供的抽象和操作的基础之上，提供了更高层的抽象，如节点、服务、消息、主题等，以及更高层的操作，如硬件抽象、底层设备控制、通用功能的实现、进程间消息转发及使用 catkin 和 cmake 管理功能包等操作。ROS 系统的应用框架如图 1 所示。

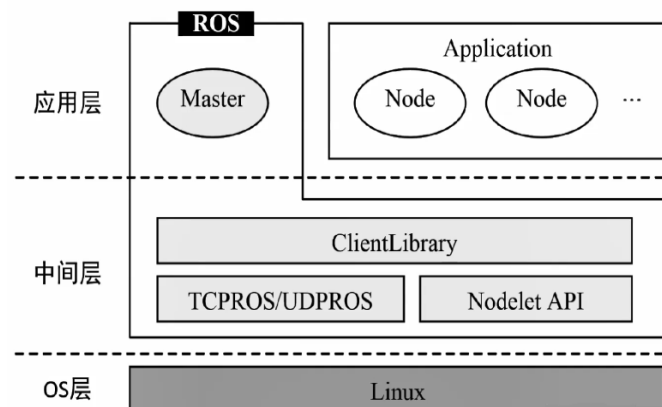


图 1 ROS 系统的应用框架

ROS 基于一个集中式拓扑的图结构来处理节点与其他节点之间在通信图网络上发布和接收的消息。节点是任意进程，它从传感器读取数据、控制执行器，或运用于在环境中自主映射或导航的高级复杂的机器人或视觉算法。ROS 还提供开发机器人项目所需的功能库和工具，如社区化的软件库*-ros-pkg，导航库和 rviz 可视化界面都是基于这个软件库；可视化工具、仿真环境和调试工具使得

ROS 的开发变得更为高效便捷，如使用 `rqt_plot` 可以在线绘制获得的数据，使用 `rviz` 可以看到真实机器人的三维显示，等等。

ROS 已经发布了多个版本，在本实验中，我们采用的是基于 Ubuntu16.04 系统下的 ROS-Kinetic 版本。

3.2 全向移动机器人平台

该实验所使用的是如图 2 所示的三轮全向移动平台，该平台主体由三个全向轮径向对称安装组成，各轮互成 120 度夹角，这种结构使得它可以在平面内向任意方向平移；同时平台机身采用全铝合金结构，使用空心杯直流无刷伺服电机和大功率伺服驱动器驱动，保证平台具有充沛的动力。

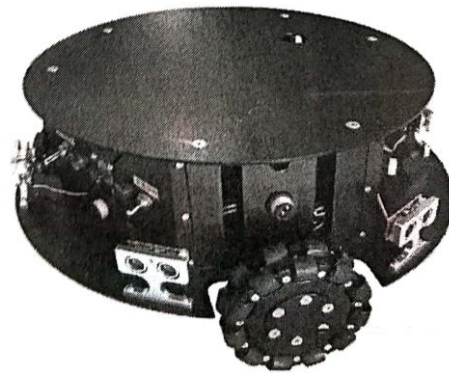


图 2 三轮全向移动平台

该平台使用 STM32F407 控制芯片，外围板载 Flash 存储器构成多用途的运动控制卡。为了方便用户使用和开发，该平台将底层运动控制模型和算法的程序封装成“程序指令”的形式，通过串口通信的方式向平台发送“运动程序指令”就能够控制平台做指定的运动。串口通信的协议内容如下：波特率 115200bps，8 位数据位，1 位停止位，无奇偶校验。通信指令采用 ASCII 码进行传输，所有指令使用相同的格式如下所示：

`<HEAD><CMDTYPE><CMDID>[<SEPRAT><Param1>...<SEPRAT><ParamN>]<END>`其中，方括号[]中间的段落为一个或多个可选择的参数；`<HEAD>`是指令头标识符，是一个 ASCII 字符‘<’；`<END>`是指令尾标识符，是一个 ASCII 字符‘>’；`<CMDTYPE>`是指令种类标识符；`<CMDID>`是指令编号标识符，`<SEPRAT>`是指令头标识符，是一个 ASCII 字符‘#’；`<ParamN>`是由指令决定的参数，根据指令的不同所使用的参数和数量各不相同。

根据实验过程所需要的运动形式，用到的具体程序指令有以下三种：

(1) L 指令：该指令可以实现平台沿直线轨迹运动，指令的具体内容为：

<L0#SPD#ACC#ALPH#W>，即<指令标识符#运动线速度#运动线加速度#平台系 X 轴与地面系的初始夹角#运动角速度>。例如当要求不涉及转动的纯粹的直线运动：可以使用指令<L0#0.2#0.5#0#0>让平台以 0.2m/s 的速度，0.5m/s² 的加速度向前运动；使用<L0#0.2#0.5#3.14159#0>向后运动。

(2) M 指令：该指令使可以直接控制各个电机的速度，实现平台以一定的角速度原地旋转。指令的具体内容为：<M0#SPD#ACC>，即<指令标识符 # 转动速度 #转动加速度>。在实际应用时，使用指令<M0#0.05#0.1>控制平台以 0.05rad/s 的角速度 0.1rad/s² 的角加速度逆时针旋转；同样地，使用指令<M0#0.05#0.1>，使平台顺时针旋转。

(3) S 指令：该指令使平台停止一切运动。指令的具体内容为<S0>，该指令只有指令标识符，没有其它参数。

3.3 Kinect 视觉传感器

Kinect 是由微软开发，可应用于 Xbox360 主机的周边设备，它是一款 3D 体感摄影机。Kinect 有三个镜头，中间的镜头是 RGB 彩色摄影机，用来采集彩色图像。左右两边镜头则分别为红外线发射器和红外线 CMOS 摄影机所构成的 3D 结构光深度感应器，用来采集深度数据（场景中物体到摄像头的距离）。彩色摄像头最大支持 1280*960 分辨率成像，红外摄像头最大支持 640*480 成像。Kinect 还搭配了追焦技术，底座马达会随着对焦物体移动跟着转动。Kinect 也内建阵列式麦克风，由四个麦克风同一时候收音，比对后消除杂音，并通过其采集声音进行语音识别和声源定位。如图 3 所示为实验时所用到的 Kinect 一代产品，该版本 Kinect 检测范围在 0.8m~4m 之间，角度范围为水平方向 57 度，垂直方向 43 度。



图 3 Kinect V1

Kinect 可同时获取深度图像和可见光彩色图像，广泛应用于视觉图像处理任务中。使用时先安装相应版本的功能包和驱动程序，然后通过运行 Kinect 节点启动它。通过使用 Kinect 节点发布的各种主题便可以查看传感器获得的相应的图像，例如使用主题/camera/rgb/image_color，可以查看 RGB 传感器采集的彩色

图像；使用主题/camera/depth/image，可以查看深度传感器采集的深度信息；使用主题/camera/depth/point，可以以点云的形式查看深度的 3D 展示。通过在可视化工具 rviz 界面中添加订阅话题，可以方便直观地查看各类图像信息。

在良好的工作条件下，Kinect 获得的深度数据精度可以达到毫米级。在实验过程中，我们使用 Kinect 模拟激光雷达的效果，即需要将 Kinect 的深度数据实时转换为相对应的激光雷达数据，其转换原理为：首先建立 Kinect 深度数据几何模型，实时读取 Kinect 深度数据，经初始矫正后，对深度图像做滤波与类似平滑处理，减小深度信息噪声，并恢复丢失的深度信息，以获取较为准确的深度图像，各像素点的像素代表了该点处的深度值，可根据几何模型换算获取测距数据，实现二维激光雷达的功能。相较于成本昂贵的激光雷达，这种方法可以有效地降低应用成本。

3.4 二维 SLAM 建图

SLAM (simultaneous localization and mapping),即时定位与地图构建，SLAM 问题可以描述为：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估计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。SLAM 方法可大致分为两大类:一种是基于滤波器的方法，另一种是基于图优化的方法。目前 ROS 系统中有 5 种常用的 SLAM 构建方法：HectorSLAM、Gmapping、KartoSLAM、CoreSLAM 和 LagoSLAM。我们通过对这 5 种方法的优缺点和精确的定量比较，定义一个通用的准则，选取一种适用于 Kinect 的二维地图建模方法。

HectorSLAM 对扫描的频率有较高的要求，当扫描频率较低时无法取得良好的结果，该方法不需要里程计的数据，适用于飞行器和在地形起伏较大的地面行驶的机器人。HectorSLAM 是结合已经构建的地图对激光束点阵进行优化，以此来预测激光点在实际地图中的位置以及占据栅格的概率，从而更新地图。Gmapping 采用基于 RBPF (Rao-Blackwellized particle filter) 粒子滤波的 Fast SLAM2.0 算法，其利用扩展卡尔曼滤波将当前机器人的环境观测信息融入粒子滤波器的提议分布设计之中，使得粒子集中地分布于高观测似然区域，自适应重采样技术引入减少了粒子耗散问题，计算粒子分布的时候不单单仅依靠机器人的运动，同时将当前观测考虑进去，减少了机器人位置在粒子滤波步骤中的不确定性。Gmapping 可以实时构建室内地图，在构建小场景地图所需的计算量较小且

精度较高。相比 Hector SLAM 对激光雷达频率要求低、鲁棒性高（Hector 在机器人快速转向时很容易发生错误匹配，建出的地图发生错位，原因主要是优化算法容易陷入局部最小值）；而相比 KartoSLAM 在构建小场景地图时，Gmapping 不需要太多的粒子并且没有回环检测因此计算量小于 KartoSLAM 而精度并没有差太多。Gmapping 有效利用了车轮里程计信息，这也是 Gmapping 对激光雷达频率要求低的原因：里程计可以提供机器人的位姿先验。而 Hector 和 KartoSLAM 的设计初衷不是为了解决平面移动机器人定位和建图，Hector 主要用于救灾等地面不平坦的情况，因此无法使用里程计。而 KartoSLAM 是用于手持激光雷达完成 SLAM 过程，也就没有里程计可以用。

四、实验设计

本实验要求利用全向移动机器人平台和 Kinect 视觉传感器实现对周围环境的二维 SLAM 建图，根据实验内容的具体要求，设计实验步骤如下：

1) 在个人笔记本电脑上安装 Ubuntu16.04 系统，并在该系统上安装 ROS 操作系统，搭建完整的软件实验环境。

2) 基于 ROS 操作系统编写节点程序，分别实现电脑与全向移动机器人平台之间的串口通信、将电脑键盘按键值转化为移动平台的运动指令以及通过键盘按键操控移动平台运动的功能。

3) 在 Ubuntu 系统中安装 Kinect 视觉传感器的驱动程序，通过查看 Kinect 节点发布图像信息话题，获取相应的图像信息；创建功能包，将 Kinect 获取的深度信息转化为激光雷达数据，从而实现 Kinect 模拟激光雷达的效果。

4) 利用移动机器人平台和 Kinect 视觉传感器搭建实验平台，创建 SLAM 算法功能包，并整合各独立实验的实验结果，搭建起整体的软件系统，通过对系统进行联合调试，最终实现对周围环境的 2D-SLAM 建图。

五、实验过程

5.1 ROS 操作系统的安装与环境搭建

整个实验是基于 Ubuntu 系统下的 ROS 操作系统完成。为了方便使用，在外置移动硬盘上安装 Ubuntu16.04 系统，然后安装 Kinetic 版本的 ROS 系统，并配置相关环境。下面详细介绍 ROS 系统的安装与配置过程。

（1）安装 ROS

在 Ubuntu 系统下打开一个终端，输入如下指令添加源文件：

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

再输入如下指令设置秘钥：

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
0xB01FA116
```

然后输入指令更新系统软件使之处于最新版：`$ sudo apt-get update`

最后获取安装包安装全功能版 ROS：`$ sudo apt-get install
ros-kinetic-desktop-full`

（2）配置 ROS 工作环境

首先初始化 rosdep，使用指令如下：

```
$ sudo rosdep init
```

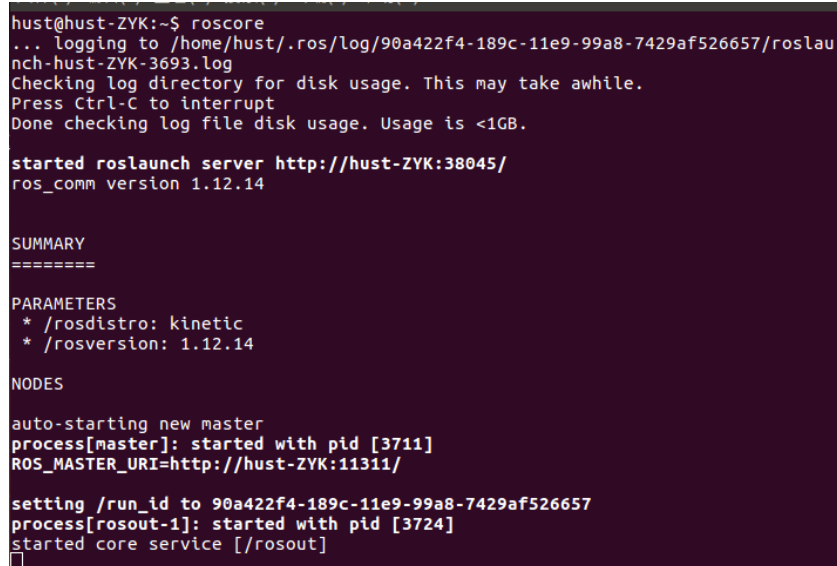
```
$ rosdep update
```

然后初始化环境变量

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

```
$ source ~/.bashrc
```

最后测试 ROS 是否安装成功：使用指令`$ roscore`启动 ROS 环境，终端中显示图 4 所示的信息，表明 ROS 安装成功并能够顺利启动。



```
hust@hust-ZYK:~$ roscore
... logging to /home/hust/.ros/log/90a422f4-189c-11e9-99a8-7429af526657/roslau
nch-hust-ZYK-3693.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://hust-ZYK:38045/
ros_comm version 1.12.14

SUMMARY
=====

PARAMETERS
 * /rostdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [3711]
ROS_MASTER_URI=http://hust-ZYK:11311/

setting /run_id to 90a422f4-189c-11e9-99a8-7429af526657
process[rosout-1]: started with pid [3724]
started core service [/rosout]
```

图 4 ROS 系统启动测试

最后通过运行官方的小海龟测试程序熟悉 ROS 系统的操作。使用指令获取小海龟测试程序：`$ sudo apt-get install ros-kinetic-turtlesim`。然后，在三个不同的终端分别执行以下两条指令：

```
$ rosrn turtlesim turtlesim_node
```

```
$ rosrn turtlesim turtle_teleop_key
```

通过键盘控制界面中出现的小海龟向前、向后以及旋转的运动，得到测试界面如图 5 所示。

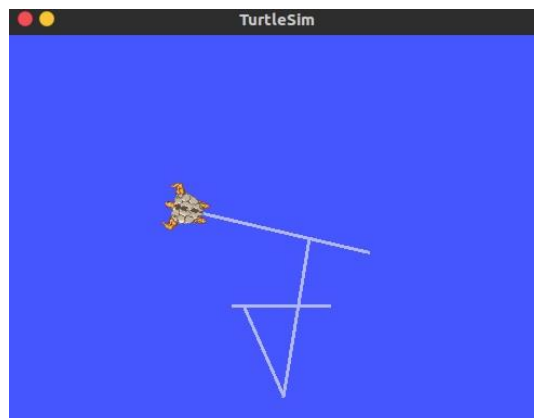


图 5 ROS 系统下运行小海龟的测试程序

5.2 全向移动平台的键盘操控

由于要借助全向移动平台的运动对周围环境进行地图的构建，因此需要实现平台的键盘操控功能，按照实验原理部分的分析，首先要读取电脑键盘的键值，这一子功能作为一个节点；然后将键盘键值转化为移动平台的运动指令，并借助串口通信协议发送给移动平台，使之做相应的运动，这些子功能作为另一个节点。

（1）读取键盘键值的节点创建

为了使平台具有更多的运动形式，下载功能完备的键盘控制的 ROS 包，并将该功能包放置到当前工作空间中，使用如下指令操作：

```
$ git clone https://github.com/Forrest-Z/teleop_twist_keyboard.git
```

由于该程序使用 python 语言编写，故要将相应的源代码 teleop_twist_keyboard.py 设置为“允许作为程序执行文件”。然后编译当前工作空间，然后通过使用如下指令启动键盘节点：`$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py`。

```
hust@hust-ZYK:~/zykctk_ws$ rosrn teleop_twist_keyboard teleop_twist_keyboard.py
Reading from the keyboard and Publishing to Twist!
-----
Moving around:
   u   i   o
   j   k   l
   m   ,   .

For Holonomic mode (strafing), hold down the shift key:
-----
   U   I   O
   J   K   L
   M   <   >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5      turn 1.0
```

图 6 节点 teleop_twist_keyboard 的运行界面

当我们按下键盘时，teleop_twist_keyboard 节点会发布/cmd_vel 主题发布速度信息，通过指令\$ rostopic echo /cmd_vel，可查看速度信息的具体情况如图 7 所示，可知，该速度信息包含三轴的位移速率和三轴的角速率。

```
hust@hust-ZYK:~/zykctk_ws$ source ~/zykctk_ws/dev
el/setup.bash
hust@hust-ZYK:~/zykctk_ws$ rostopic echo /cmd_vel
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0
--
```

图 7 /cmd_vel 话题内容

(2) 控制平台运动节点的创建

由于移动平台使用串口通信方式进行指令的传输，因此首先要在当前工作空间下安装 ROS 系统的串口通信包，使用如下指令：

```
$ git clone https://github.com/Forrest-Z/serial.git
```

由于 ROS 系统串口的开启需要高级权限，因此每次连接串口线进行串口通信时要使用如下指令允许权限：\$ sudo chmod 666 /dev/ttyUSB0。

新建 base_controller 功能包，通过如下指令创建 base_controller 节点源文件：

```
$ catkin_create_pkg base_controller roscpp
```

在 base_controller 节点中订阅 teleop_twist_keyboard 节点发布的/cmd_vel 话题，接收到速度指令，然后按照通信协议转换指令格式并写入到串口发送函数中，完成指令的传输。根据实际的运动情况，只用到 X 轴向的线速度和 Z 轴向的角速度信息。base_controller 节点源码在附录中给出。

平台运动控制的节点的启动可通过如下指令操作：

```
$ rosrn base_controller base_controller
```

5.3 Kinect 深度信息转化为激光雷达数据

(1) 安装 Kinect 传感器驱动

在安装驱动之前要先配置驱动的运行环境: OpenNI 和 Kinect Sensor Module, 通过如下方式获取运行环境的源文件, 并编译安装:

```
$ git clone https://github.com/OpenNI/OpenNI.git
```

```
$ git clone https://github.com/avin2/SensorKinect.git
```

然后, 使用如下指令安装 Kinect 传感器的驱动程序:

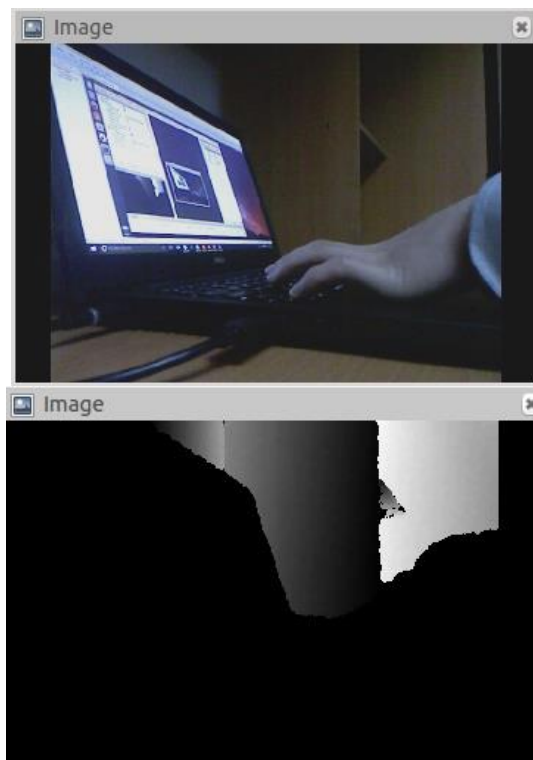
```
$ sudo apt-get install ros-kinetic-freenect-*
```

```
$ rospack profile
```

成功安装 Kinect 驱动后, 连接实验所用的 Kinect 一代硬件设备, 然后使用如下指令启动 Kinect 硬件: `$ roslaunch freenect_launch freenect.launch`。根据实验原理部分的内容, 我们可以通过订阅 Kinect 节点发布的话题获得图像信息, 使用下面两条指令分别获得 RGB 图像和深度图像如图 8 中的(a)和(b)所示。

```
$ rosrn image_view image_view image:=/camera/rgb/image_color
```

```
$ rosrn image_view image_view image:=/camera/depth/image
```



(a) RGB 图像

(b) 深度图像

图 8 Kinect 传感器获得的图像信息

(2) 深度信息转化为激光雷达数据

根据实验原理部分的分析,将深度信息转化为激光雷达数据的算法实现过程复杂,因此使用已有的功能包 `depthimage_to_laserscan` 实现,该功能包可通过如下指令获取:

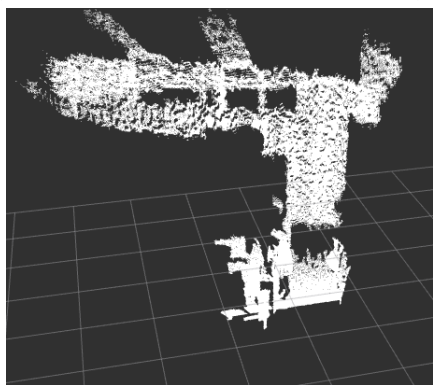
```
$ git clone https://github.com/ros-perception/depthimage_to_laserscan.git
```

`Depthimage_to_laserscan` 节点通过订阅 Kinect 节点发布的 `Image(sensor_msgs/Image)`和 `Camera_info(sensor_msgs/CameraInfo)`话题,将其中的深度信息转化为激光数据后,通过发布 `Scan(sensor_msgs/LaserScan)`话题给出伪激光数据。

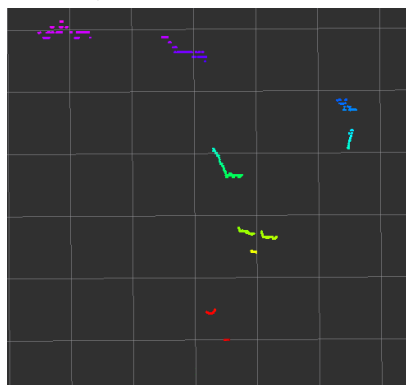
在启动 Kinect 硬件后,通过执行如下指令运行节点,实现数据转化功能:

```
$roslaunch depthimage_to_laserscan depthimage_to_laserscan  
image:=/camera/depth/image_raw
```

然后调用 `rviz` 工具可以查看转化得到的激光数据信息,使用指令: `$ roslaunch rviz rviz` 打开 `rviz` 工具。在 `rviz` 中 `Fixed Frame` 一栏选择 `camera_depth_optical_frame`,通过添加 `PointCloud2`,可得到点云图如图 9 中的(a)所示,通过添加 `LaserScan` 话题,调出 `LaserScan` 显示界面,并在 `LaserScan` 中的 `Color Transformer` 一栏中选择 `AxisColor`,得到如图 9 中(b)所示的伪激光点,其中距离近的物体的点是红色的,距离远物体的点是紫色的。



(a) 点云图像



(b) 伪激光点

图 9 Kinect 深度信息转化得到的伪激光点

5.4 二维 SLAM 建图

如图 10 所示将 Kinect 传感器固定在全向移动平台上,通过键盘操控平台移动对周围环境进行地图创建。分别使用了两种 SLAM 算法来创建地图。



图 10 SLAM 实验硬件平台

1) Hector_slam 算法

通过如下指令获取 hector_slam 算法的功能包：

```
$ git clone https://github.com/DaikiMaekawa/hector_slam_example
```

通过以下指令依次启动运行各个功能节点，便可控制平台移动开始建图：

```
$ roscore
```

```
$ roslaunch teleop_twist_keyboard teleop_twist_keyboard.py
```

```
$ sudo chmod 666 /dev/ttyUSB0
```

```
$ roslaunch base_controller base_controller
```

```
$ roslaunch freenect_launch freenect.launch
```

```
$ roslaunch hector_slam_example hector_openni.launch
```

2) Gmapping 算法

通过如下指令获取 Gmapping 算法的功能包：

```
$ sudo apt-get install ros-kinetic-slam-gmapping
```

由于 Gmapping 算法需要的激光雷达提供的距离信息和里程计信息两方面的数据才能够运行创建地图。由于移动平台中没有硬件里程计模块，因此无法从移动平台得到里程计信息，所以需要创建一个节点，用来发布虚拟的 tf 树的形式生成里程计信息。故创建 myodom 功能包和 myodom 节点源文件，在源文件中订阅速度指令/cmd_vel 话题，在回调函数中根据控制指令更新移动平台当前的速度和角速度信息。然后主函数中根据已更新的速度信息计算里程计位移和角度，最后分别向 tf 树和 ROS 发布坐标变换信息和里程计数据。该 cpp 源文件的具体内容在附录中给出。

由于 Gmapping 算法中要启动的节点很多，为了能够方便同时启动之前的各个功能节点，在当前工作空间的 base_controller 功能包下创建一个 Launch 文件，内容如下：

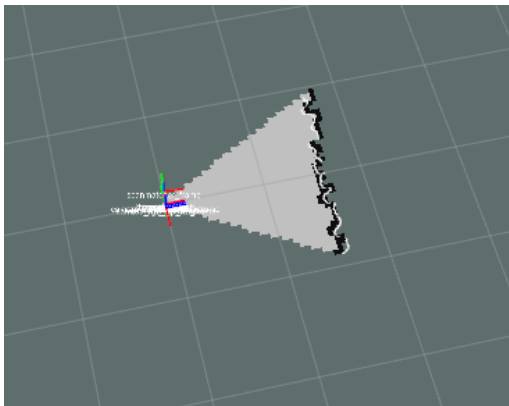
```
<launch>
<node pkg="base_controller" type="base_controller" name="base_controller"/>
<node pkg="myodom" type="myodom" name="myodom"/>
<node pkg="settf" type="tf_broadcaster" name="tf_broadcaster"/>
<node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan"
      name="depthimage_2_laserscan"/>
<remap from="image" to="/camera/depth/image_raw"/>
<node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"/>
<remap from="scan" to="scan"/>
<node pkg="rviz" type="rviz" name="rviz" />
</launch>
```

在启动 Kinect 驱动和键盘节点后，通过指令 `$ roslaunch base_controller zykslaunch.launch`，运行该 Launch 文件，同时启动其余各个功能节点，便可控制平台移动进行地图的创建。

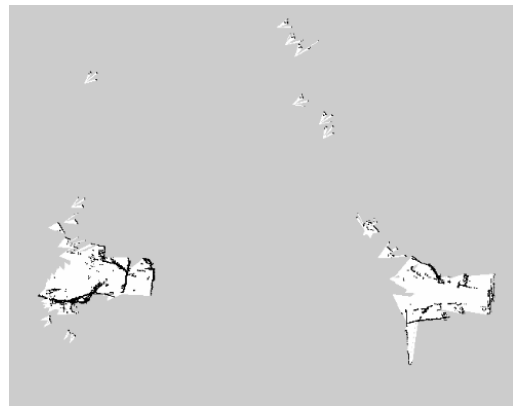
六、实验结果

运行 Launch 文件后，使用键盘控制平台移动，在 rviz 界面中可以得到实时显示的周围环境的二维地图。

当使用 Hector_slam 算法时，得到实验结果如图 11 所示，其中(a)图为未移动平台时创建的单帧地图，其中(b)图为在房间中移动平台时创建的地图。



(a) 未移动平台时创建的地图



(b) 移动平台时创建的地图

图 11 使用 Hector_slam 算法的建图结果

从图中可见，单帧地图效果很好，但移动后发现，创建的各帧地图无法拼接在一起，呈现混乱的状态。通过查阅资料可知 Hector_slam 算法通过最小二乘法匹配扫描点，且依赖高精度的数据，对于扫描角很小且噪声较大的 Kinect，匹配扫

描点的时候会陷入局部点，导致建立的地图非常混乱。因此当使用 Kinect 进行二维地图创建时，不能使用 Hector_slam 算法。

当使用 Gmapping 算法时，得到实验结果如图 12 所示，可以看到，相比于使用 Hector_slam 算法创建的地图，Gmapping 算法能够创建出连续的地图，但是该地图和周围环境的实际情况有较大差异。分析原因后发现，由于我们使用自己编写的虚拟里程计节点发布里程计数据，而依赖里程计数据更新地图中移动平台的位置，由于虚拟里程计节点中平台速度和位置的计算方法和所使用的移动平台的运动学模型不相符，所以导致所建地图的和实际环境差异巨大。

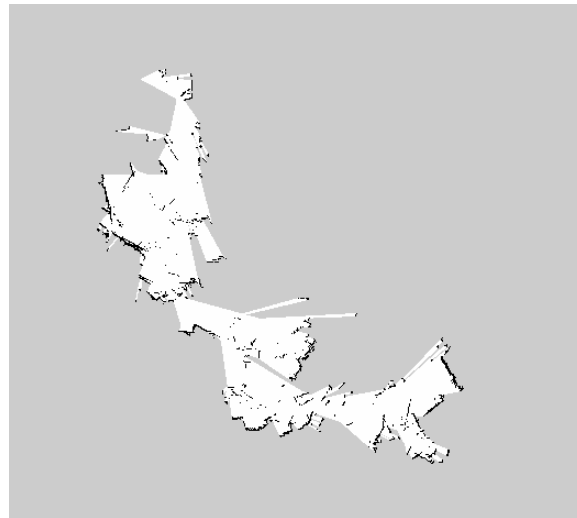


图 12 使用 Gmapping 算法的建图结果

七、总结与展望

基于全向移动机器人平台和 Kinect 的二维 SLAM 建图综合实验中，通过实验环境搭建、软件编程、调试及实验结果分析等环节，我们掌握了在 Ubuntu 系统开发与调试 ROS 系统的基本思路，学会创建功能包、节点，发布与订阅话题等 ROS 系统的操作；同时对全向移动机器人平台和 Kinect 视觉传感器的调试使用，也为我们提供了一个硬件系统实践的机会，学习的很多实践技能，如硬件设备之间的串口通信的调试技巧等；此外我们还了解了二维 SLAM 建图的基本原理，通过使用不同的算法创建地图，验证了 Hector_slam 算法不适用于基于 kinect 硬件的二维建图情况。但由于课程时间有限，未对所使用的全向移动平台的运动学模型进行分析，使得最终创建的地图和周围环境的实际情况有较大差异，以后有机会要在这方面作出改进。

附录

(1) 移动平台运动控制节点 cpp 源码

```
#include "ros/ros.h" //ros 需要的头文件
#include <string>
#include <iostream>
#include <cstdio>
#include <unistd.h>
#include <math.h>
#include "serial/serial.h"
using std::string;

float linear_temp=0,angular_temp=0;//暂存的线速度和角速度
unsigned char Stop_data[4]={0x3c,0X53,0X30,0x3E}; //<S0>指令
string send_buffer; //串口数据发送变量
/*****订阅/cmd_vel 主题回调函数*****/
void callback(const geometry_msgs::Twist & cmd_input)//
{
    string port("/dev/ttyUSB0"); //串口号
    unsigned long baud = 115200; //串口波特率
    serial::Serial my_serial(port,baud,
                             serial::Timeout::simpleTimeout(1000)); //配置串口
    angular_temp = cmd_input.angular.z ;//获取/cmd_vel 的角速度
    linear_temp = cmd_input.linear.x ;//获取/cmd_vel 的线速度
    if(linear_temp !=0) //只要线速度不为零，就使用 L 指令控制
    {
        //L 指令的格式 <L0#SPD#ACC#ALPH#W>
        if(linear_temp < 0)
        {
            if(angular_temp == 0)
            {
                send_buffer = "<L0#0.05#0.05#3.14159#0> ";//向 X 轴的负方向运动
                my_serial.write (send_buffer);//串口数据发送变量
            }
            else
                my_serial.write(Stop_data,4); //串口发送停止指令
        }
        else
        {
            if(angular_temp == 0)
            {
                send_buffer = "<L0#0.05#0.05#0#0>";//向 X 轴的正方向运动
```

```

        my_serial.write (send_buffer); //串口数据发送变量
    }
    else
        my_serial.write(Stop_data,4); //串口发送停止指令
    }
}
else if (angular_temp !=0) //线速度为零，角速度不为零，使用 M 指令控制
{
    if(angular_temp < 0)    //M 指令<M0#SPD#ACC>
        send_buffer = "<M0#-0.05#0.1>"; //以 0.1 米每秒的速度顺时针旋转
    else
        send_buffer = "<M0#0.05#0.1>"; //以 0.1 米每秒的速度逆时针旋转
    my_serial.write (send_buffer); //串口数据发送变量
}
else //线速度和角速度均为零，使用 S 指令，停止
    my_serial.write(Stop_data,4); //串口发送停止指令
}
/*****主函数*****/
int main(int argc, char **argv)
{
    string port("/dev/ttyUSB0"); //串口号
    unsigned long baud = 115200; //串口波特率
    serial::Serial my_serial(port, baud,
                             serial::Timeout::simpleTimeout(1000)); //配置串口
    ros::init(argc, argv, "base_controller"); //初始化串口节点
    ros::NodeHandle n; //定义节点进程句柄
    ros::Subscriber sub = n.subscribe("cmd_vel", 20, callback); //订阅/cmd_vel 主题
    ros::spin();
    return 0;
}

```

(2) 模拟发布里程计信息节点 cpp 源码

```

#include "ros/ros.h"
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>

float linear_temp=0, angular_temp=0; //暂存的线速度和角速度
double x = 0.0, y = 0.0, th = 0.0;
double vx = 0.0, vy = -0.0, vth = 0.0;
/*****订阅/cmd_vel 主题回调函数*****/
void callback(const geometry_msgs::Twist & cmd_input)
{
    double delta_x = 0.0, delta_y = 0.0, delta_th = 0.0;

```

```

angular_temp = cmd_input.angular.z ;//获取/cmd_vel 的角速度,rad/s
linear_temp = cmd_input.linear.x ;//获取/cmd_vel 的线速度.m/s

if(linear_temp !=0) //只要线速度不为零,
{
    if(linear_temp < 0)
    { vx = -0.05; vy = 0.0; vth = 0.0; }
    else
    { vx = 0.05; vy = 0.0; vth = 0.0; }
}
else if (angular_temp !=0) //线速度为零, 角速度不为零,
{
    if(angular_temp < 0)
    { vx = 0.0; vy = 0.0; vth = -0.1; }
    else
    { vx = 0.0; vy =0.0; vth = 0.1; }
}
else //线速度和角速度均为零, 使用 S 指令, 停止
{ vx = 0.0; vy = 0.0; vth = 0.0; }
}
/*****主函数*****/
int main(int argc, char** argv)
{
    ros::init(argc, argv, "odometry_publisher");
    ros::NodeHandle n;
    ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);
    tf::TransformBroadcaster odom_broadcaster;
    ros::Subscriber sub = n.subscribe("cmd_vel", 20, callback); //订阅/cmd_vel 主题
    ros::Time current_time, last_time;
    current_time = ros::Time::now();
    last_time = ros::Time::now();
    ros::Rate r(2.0);

    while(n.ok())
    {
        ros::spinOnce(); // check for incoming messages
        current_time = ros::Time::now();
        /**通过回调函数中更新的速度信息计算里程计位移和角度**/
        double dt = (current_time - last_time).toSec();
        double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
        double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
        double delta_th = vth * dt;
        x += delta_x;

```

```
y += delta_y;
th += delta_th;

geometry_msgs::Quaternion odom_quat =
    tf::createQuaternionMsgFromYaw(th); //将角度表示为四元数
的形式

geometry_msgs::TransformStamped odom_trans; //坐标变换的具体信息
odom_trans.header.stamp = current_time;
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_link";
odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = y;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation = odom_quat;
odom_broadcaster.sendTransform(odom_trans); //向 tf 树广播坐标变换
信息

nav_msgs::Odometry odom;
odom.header.stamp = current_time;
odom.header.frame_id = "odom";
odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.position.z = 0.0;
odom.pose.pose.orientation = odom_quat; //位置信息
odom.child_frame_id = "base_link";
odom.twist.twist.linear.x = vx;
odom.twist.twist.linear.y = vy;
odom.twist.twist.angular.z = vth; //速度信息
odom_pub.publish(odom); //向 ROS 发布里程计信息

last_time = current_time;
r.sleep();
}
}
```