YUHESEN

FW-mid-OS-mate

Open Source ROS Robot

Development Platform

User manual V1.0

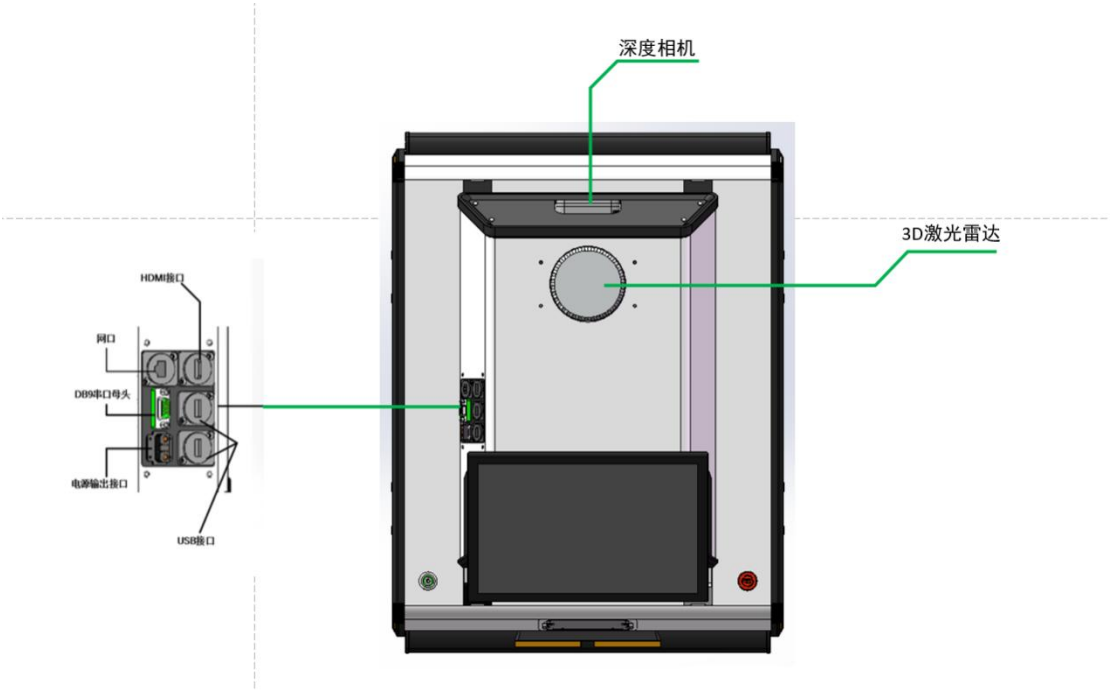
Ver. Ubuntu 20.04 Noetic

Shenzhen yuhesen Technology Co., Ltd.

CONTENTS

Hardware Introduction

- **Top view**



- **Loading capacity**

  The weight of FW-mid-OS-MATE is approx. 68kg with loading capacity of 80kg.

- **Working condition**

  FW-mid-OS-MATE can work outdoor and indoor application.

- **Protection**

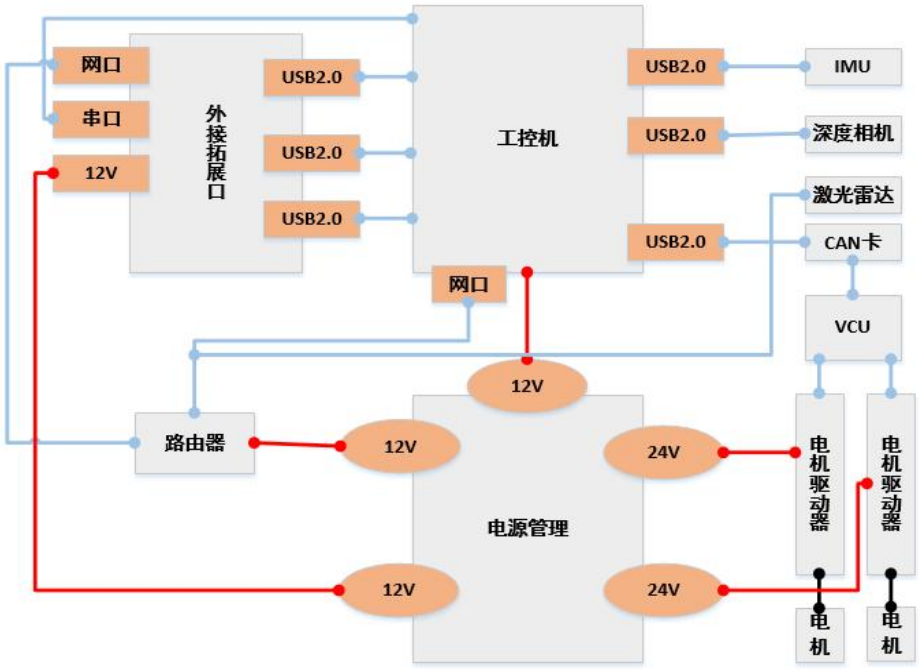
  The FW-mid-OS-MATE robot should avoid contact with water, mist, standing water on the ground, and any other liquids to prevent potential damage to its circuits and mechanisms.

- **Working Temperature and caution**

  The FW-mid-OS-MATE robot is designed to operate within a temperature range of -10°C to 50°C. During usage, it is important to keep the robot away from open flames and other heat sources.

- **Electrical description**

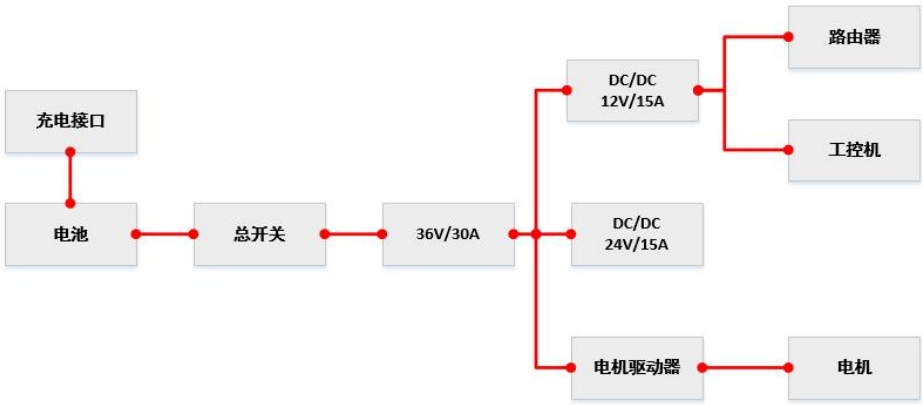
  The Yuhesen FW-mid-OS-MATE robot consists of the following internal components: 1 computer, 1*battery module, 1*VCU control board, 4*motor drivers, and 4*motors. The computer runs the ROS operating system and is connected to the CAN bus inside the robot chassis via a CAN adapter.



途中红色线为供电，蓝色线为控制信号，黑色先合并了供电与控制信号

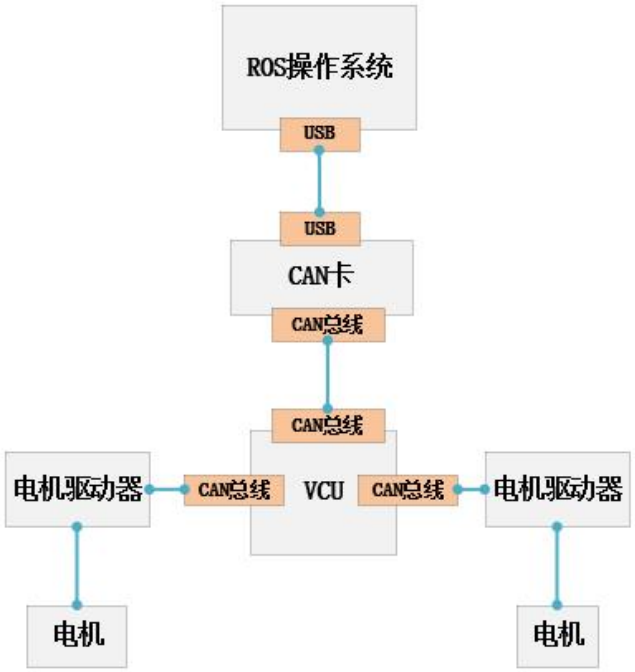
- **Power output**

  The power supply of the Yuhesen FW-mid-OS-MATE robot is provided by a battery module. The battery module consists of a 48V/20AH lithium-ion battery pack and is equipped with an integrated battery protection board. The output voltage of the battery module is dependent on the current remaining capacity, with the voltage decreasing as the remaining capacity decreases.



- **Communication link**

  The computer uses a CAN adapter to communicate with the VCU (Vehicle Control Unit) via the CAN bus. It sends speed and other control information to the VCU at a cycle of 20ms and receives real-time position feedback from the motors. Within the source code of the Yuhesen FW-mid-OS-MATE robot, there is a node named "yhs_can_control_node" that serves as the central communication hub for this purpose.



- **Motors**

Working voltage	24V
Rated power	350W*4

- **Sensors**

3D LiDAR	Measurement Range: 0.4m~150m
	FOV – Horizontal: 360° FOV – Vertical: -15~+15°
	Ranging Accuracy: +/-3cm
	Angle resolution: 0.18°（10HZ）
	Scanning frequency: 10HZ
	Communication: Ethernet port
IMU	Measurement Range: ±2000°/s Accelerometer: ±8G(1G = 1x gravitational acceleration)
	Gyro Resolution ratio: 0.01°/s Accelerometer: 1uG
	Gyro Zero-bias stability:8°/h Accelerometer: 60uG
	Communication: USB to serial port
Depth Camera	Range: 0.2~4m
	Working temperature: -10℃~50℃
	Depth FOV: 73.8°× 58.8° RGB FOV: 80.9°× 51.7°
	Resolution ratio: 1920×1080 640×480
	Communication: USB2.0
CAN Adapter	Communication: USB to CAN
Router	Network standards: 802.11ac
	Wireless network standard: 2.4G/5G Network interface: 1900M Gigabit Port*3
GNSS Receiver (Optional)	RTK accuracy(RMS): Horizontal ±(10+1x10-6XD)mm Vertical: ±(20+1x10 -6 XD)mm
	RTD accuracy(RMS): Horizontal ±0.25m(1σ) Vertical: ±0.50m(1σ)
	Single point positioning accuracy: single frequency H≤3m, V≤5m (1σ, PDOP≤4) Dual frequency H≤1.5m, V≤3m (1σ, PDOP≤4)）
	Pose accuracy heading angle: 0.2°/R（R refer to the length of dual antenna baseline） Roll/pitch angle: 0.4°/R（R refer to the length of dual antenna baseline）
	Communication: RS232、CAN

ROS Learning Introduction

ROS is a robot development system built on Ubuntu, consisting of a series of software packages and communication architecture. For Chinese students who may be more familiar with Windows, Ubuntu might be less familiar, which can present additional challenges when learning ROS. Fortunately, the rapid development of the internet industry in China has led to numerous books and online resources about Ubuntu and Linux, making self-learning much more accessible. When it comes to learning ROS, developers can choose suitable methods based on their own circumstances. Here are two suggestions that may be helpful:

For developers who have sufficient time and patience for learning, it is recommended to start by studying the usage of Ubuntu through books and online tutorials. Then, explore the ROS official wiki to gain a detailed understanding of ROS and its various concepts. Implement the tutorials and examples provided in the ROS official wiki to gain hands-on experience. Finally, return to this tutorial (note that some programming experiments in this tutorial may need modifications due to different navigation components, but they can provide programming insights).

For developers with tight project deadlines or less patience for theoretical learning, it is possible to start practicing with this tutorial directly. Quickly get the robot up and running to avoid extinguishing the enthusiasm for learning with tedious theory. After gaining a general experience with ROS through the exercises in this tutorial, go back and fill any gaps in theoretical knowledge.

If you want to quickly implement mapping and navigation functionalities, refer to **Experiment 6** and **Experiment 8**.

Computer, Router Setting Instruction

❖ User name and passwords

Ubuntu user name	yhs
Root passwords	123456
Source code content	/home/yhs/catkin_ws/src

❖ ROS version of Ubuntu

The Yuhesen FW-mid-OS-MATE robot's industrial control computer is installed with Ubuntu 20.04 and the Noetic version of ROS. (Note: If the robot's Ubuntu interface prompts a system upgrade popup, please close it and do not update the system version.)

❖ IDE

Visual Studio Code。The development environment is suggested by using Visual Studio Code.

❖ Software package

The computer included in the standard package of Yuhesen robots comes pre-installed with ROS, an IDE, and the Yuhesen robot's source code package, allowing you to use them directly.

❖ WiFi name and passwords of Router

Wifi name	YHS-ROS-XX (“XX” shall refer to chassis model number)
Wifi passwords, router’s login passwords	12345678

Before powering on the vehicle, please carefully read the user manual carried with the vehicle to understand the operation of the remote controller.

After powering on, drive the vehicle with the remote controller to a more open area to avoid collisions when not very skilled. It is recommended to adjust the control speed to the lowest gear.

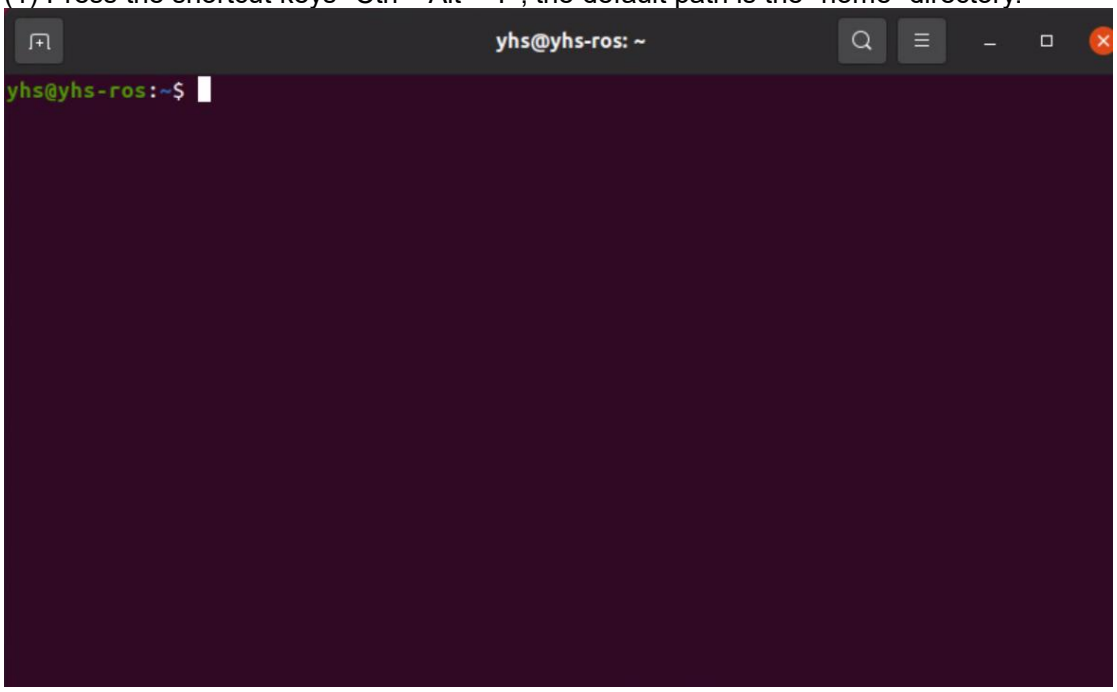
If you are not very familiar with the Ubuntu system, you can read the terminal operation introduction in the next section, follow the steps, become more familiar, and then let the vehicle automatically achieve autonomous driving functionality.

To enable the vehicle to achieve autonomous driving, if there is no screen connected to the vehicle, you need to prepare a computer with ROS system installed, connect to the vehicle's Wi-Fi, follow the operation instructions in Appendix One to remotely control the vehicle. Then, skip the preceding experimental sections and proceed directly with the instructions in Experiment Six and Experiment Eight.

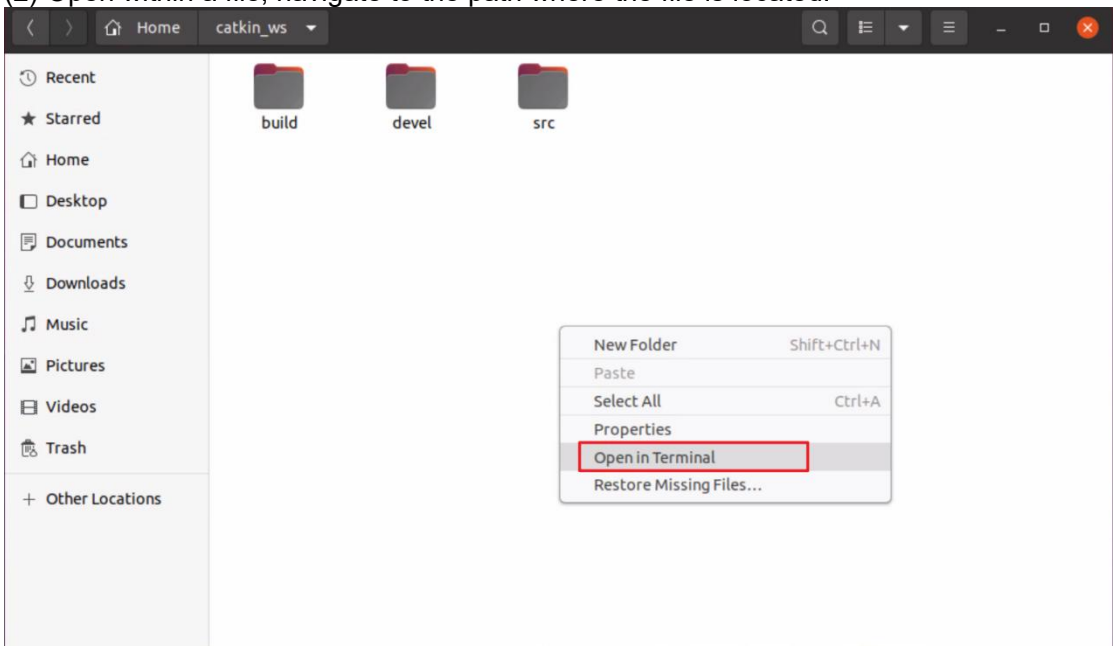
Terminal Command Operations

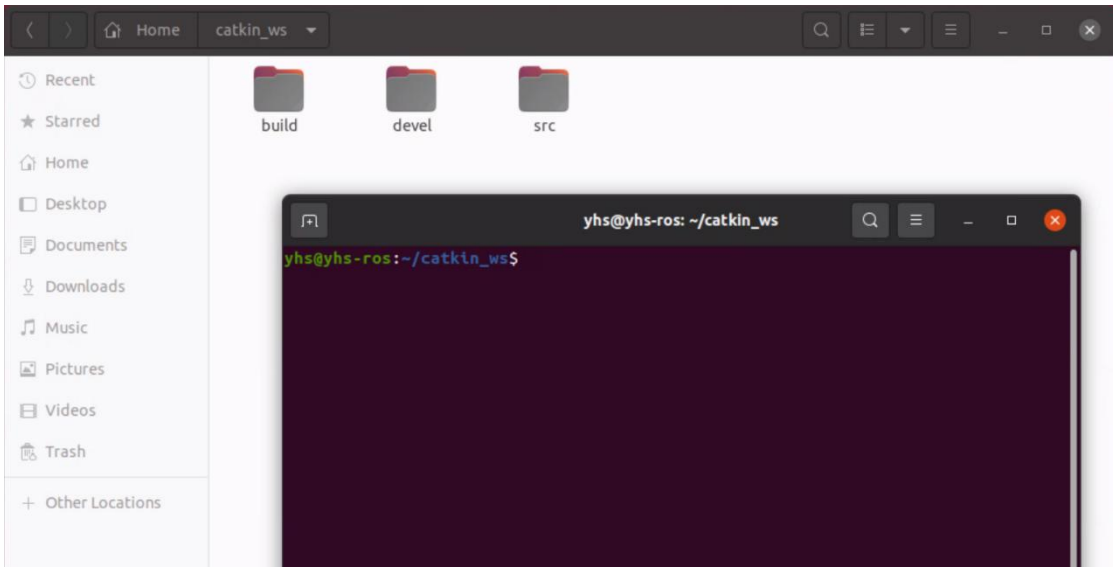
1. Open the Terminal

(1) Press the shortcut keys "Ctrl + Alt + T", the default path is the "home" directory.



(2) Open within a file, navigate to the path where the file is located.



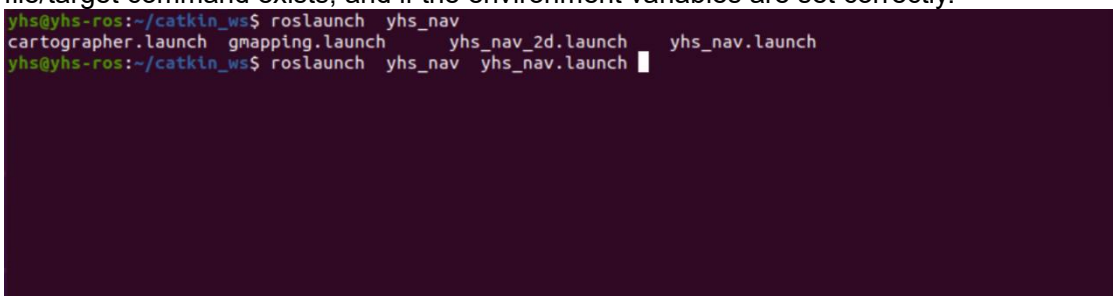


2. Command Completion with the "Tab" Key

(1) When running programs in the terminal, if some commands are long or complex, you can use the "Tab" key for completion. The complete command is as follows:

```
roslaunch  yhs_nav  yhs_nav.launch
```

(2) After entering "roslaunch yhs_nav" and pressing the "Tab" key, you may need to press it several times to see four options. If there are more options, all of them will be displayed. Therefore, it is advisable to input more information so that when you press "Tab," the options listed will be fewer. Based on the listed options, continue entering the command and press "Tab" to complete. If all the listed options do not include the target file/target command, check if the entered command is correct, if the target file/target command exists, and if the environment variables are set correctly.



(3) Enter the following command for completion options.

```
roslaunch  yhs_
```

(4) To quickly navigate to a folder within a package, enter "roscd yhs_nav" and continue pressing "Tab" a few times to display all the folders within the package.

```
roslaunch  yhs_nav2
```

```
yhs@yhs-ros:~/catkin_ws$ roscd  yhs_nav/
yhs_nav/include/  yhs_nav/launch/   yhs_nav/map/      yhs_nav/param/    yhs_nav/src/
yhs@yhs-ros:~/catkin_ws$ roscd  yhs_nav/
```

3. Terminal Program Termination Command

(1) Suppose a terminal is running a LiDAR driver program, and you intend to execute a mapping program command next. The mapping command will start the LiDAR, resulting in an error. Therefore, before executing the mapping command, you should terminate the LiDAR program. Press "Ctrl + C" in the terminal running the LiDAR driver program, wait for the program to terminate, and ensure to terminate all programs started in this experiment before proceeding to the next one.

```
^C[pointcloud_to_laserscan-4] killing on exit
[laser_link_to_base_link-3] killing on exit
[rslidar_sdk_node-2] killing on exit
RoboSense-LiDAR-Driver is stopping.....
terminate called after throwing an instance of 'std::system_error'
  what():  Invalid argument
[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

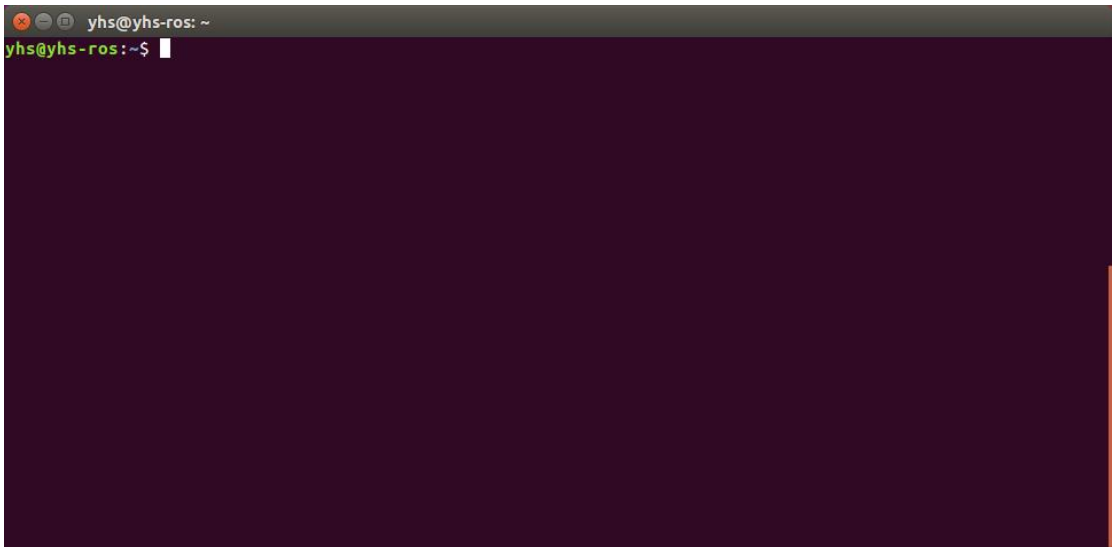
Experiment 1: Initial ROS

Note:

1. The following instructions are for ROS robots with a screen. If your robot doesn't have a screen, you can connect an external monitor or use SSH to remotely access the robot's terminal from a laptop. Refer to Appendix 1, section 3 on SSH remote control to learn how to access the robot's host terminal remotely from a laptop. (After remote access, note that only commands that open a visual interface, such as rviz, rqt, pcl_viewer, should be executed in the laptop terminal. All other commands should be executed in the robot's terminal.)

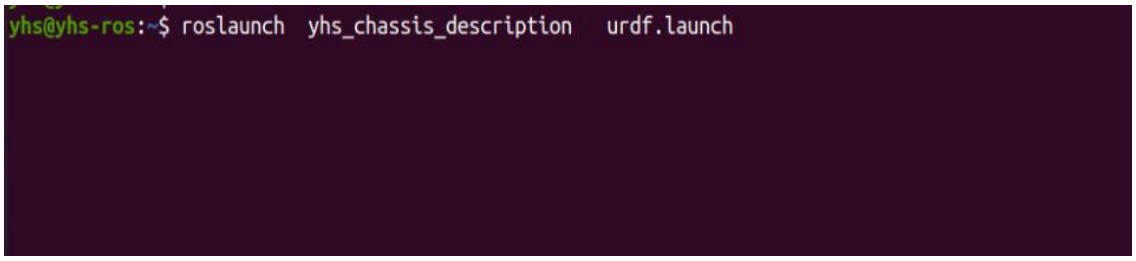
2. Before starting each experiment, make sure to close all programs from the previous experiment by pressing Ctrl+C in the terminal.

3. Right-click on the Ubuntu desktop and select "Open Terminal" to open the terminal. (Alternatively, you can start the terminal by simultaneously pressing the keyboard shortcut "Ctrl + Alt + T" or pressing "Ctrl + Shift + T" to open multiple terminals). Once the terminal program starts, a black text box interface with a blinking cursor will appear, indicating that we can enter commands.



Typing the following command in the terminal.

```
roslaunch yhs_chassis_description urdf.launch
```

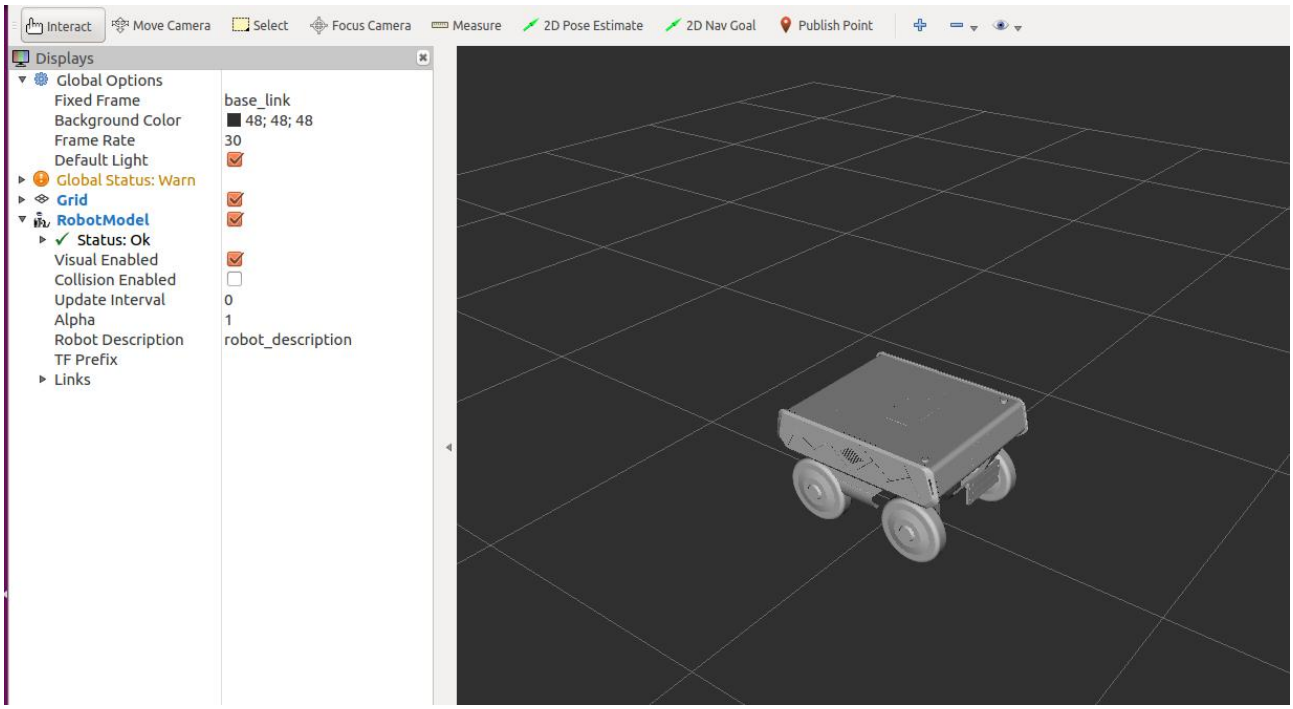


After entering the command, press Enter. The ROS core node will start running in the background, and you will see an "Rviz" startup logo on the desktop. After a moment of initialization, a graphical window will appear on the desktop. This is Rviz, the most commonly used graphical display interface in ROS. It provides a visual representation of sensors and algorithm results, making it convenient for programming and debugging of robots.

(**Note:** For robots without a screen, opening Rviz and other visual interfaces will result in an error message. To use Rviz and visual interfaces, either connect an external monitor to the ROS robot or open Rviz and visual interfaces on a remotely connected laptop. Refer to Appendix 1 for remote connection instructions.)

```
yhs@yhs-ros:~$ roslaunch  yhs_chassis_description  urdf.launch
... logging to /home/yhs/.ros/log/66a751d8-82e6-11ee-b7ef-68eda6082938/roslaunch-yhs-ros-16957.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://192.168.1.102:40955/

SUMMARY
========

PARAMETERS
 * /robot_description: <robot name="yhs_...
 * /rosdistro: kinetic
 * /rosversion: 1.12.17
 * /use_gui: False

NODES
  /
    joint_state_publisher (joint_state_publisher/joint_state_publisher)
    robot_state_publisher (robot_state_publisher/state_publisher)
    rviz (rviz/rviz)

auto-starting new master
process[master]: started with pid [16968]
ROS_MASTER_URI=http://192.168.1.102:11311

setting /run_id to 66a751d8-82e6-11ee-b7ef-68eda6082938
process[rosout-1]: started with pid [16981]
started core service [/rosout]
process[joint_state_publisher-2]: started with pid [16987]
process[robot_state_publisher-3]: started with pid [16999]
process[rviz-4]: started with pid [17000]
QXcbConnection: Could not connect to display
[rviz-4] process has died [pid 17000, exit code -6, cmd /opt/ros/kinetic/lib/rviz/rviz -d /home/yhs/catkin_ws/src/yhs_chassis_description/rviz/urdf
/log/66a751d8-82e6-11ee-b7ef-68eda6082938/rviz-4.log].
log file: /home/yhs/.ros/log/66a751d8-82e6-11ee-b7ef-68eda6082938/rviz-4*.log
```

Usually, the main interface of Rviz is divided into two columns: the left column is "Displays," and the right column is the main display area called "View." The "View" in the right column represents a 3D world display. By default, you can see a grid-like ground reference, and you can adjust the 3D perspective by dragging with the mouse. The "Displays" on the left side are used to configure the types and quantities of information displayed in the "View." The display of the ROS robot model mentioned below can only be viewed on the ROS robot's screen or an external monitor.



4. Going back to the command we entered earlier, roslaunch yhs_chassis_description urdf.launch, we used the roslaunch tool to launch the urdf.launch launch file from the package named "yhs_chassis_description". Launch files are text-based description files in ROS used to launch multiple program nodes (Nodes) collectively. Now, let's take a look at the contents of the urdf.launch file:

```
<launch>
<arg name="gui" default="false" />
<param name="robot_description" textfile="$(find yhs_chassis_description)/urdf/robot.urdf" />
<param name="use_gui" value="$(arg gui)"/>
<node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
<node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find yhs_chassis_description)/rviz/urdf.rviz" />
</launch>
```

This is an XML-formatted text file, and let's briefly understand its contents:

(1) Lines starting with <arg> are used to set static parameters. The gui parameter controls the display in the user interface.

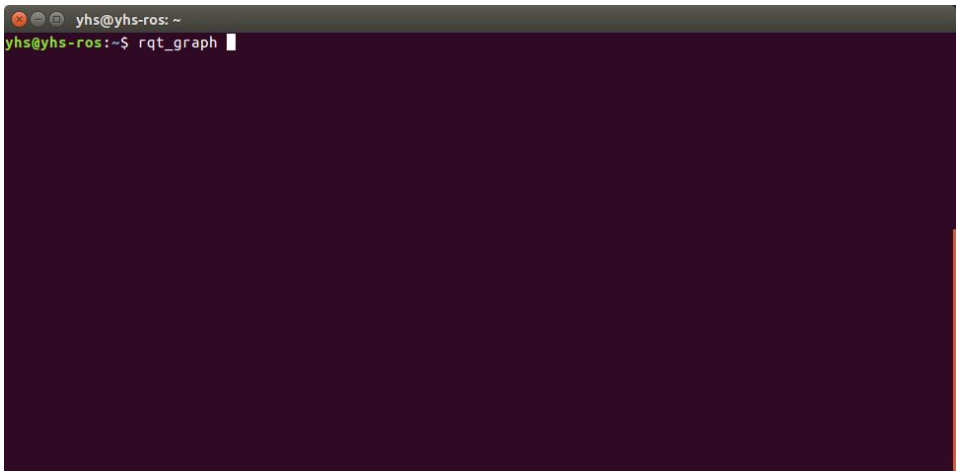
(2) Lines starting with <param> are used to set dynamic parameters. The robot_description parameter is the command for parsing the robot's URDF description, and the use_gui parameter controls the display in the user interface.

(3) Lines starting with <node> are the key components of the launch file. Each line launches a program node. The name parameter sets the display name of the node in ROS (you can choose any name you like). The pkg parameter specifies the name of the package to which the node belongs, helping to locate the node's program. The type parameter specifies the actual name of the node, and the args parameter specifies any additional startup arguments.

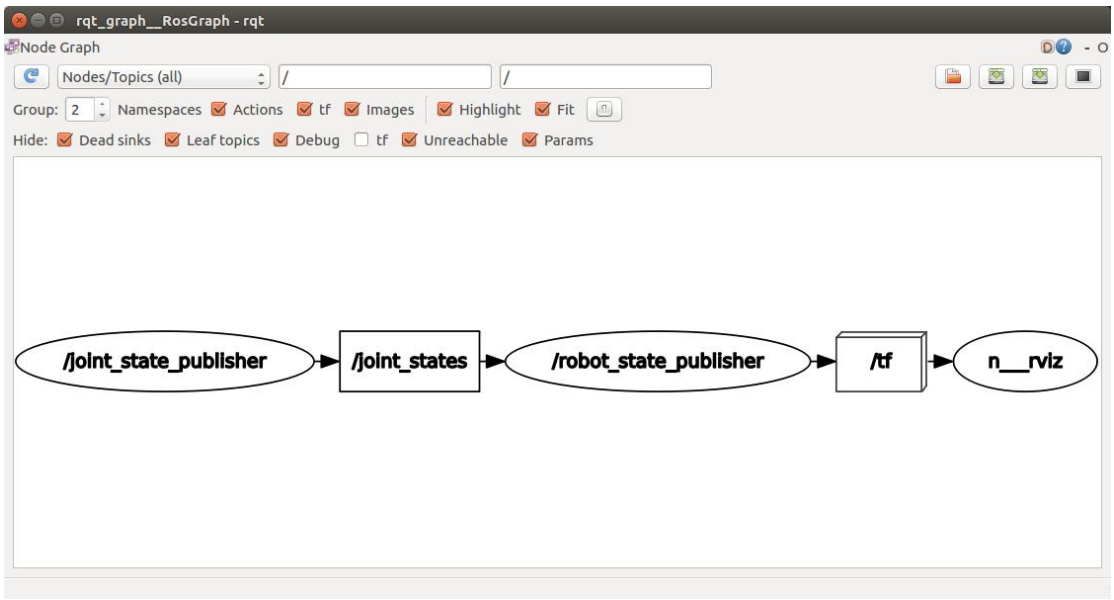
By using the <node> lines in the launch file, each specified node will be launched with the given parameters, contributing to the overall functionality of the robot system.

5. After understanding the contents of the launch file, let's take a look at what's happening behind the scenes. Please keep the Rviz interface launched from the urdf.launch file open. Right-click on the Ubuntu desktop and select "Open Terminal" to open a new terminal (you can also press the keyboard shortcut "Ctrl + Alt + T" to start the terminal). Enter the following command:

```
rqt_graph
```



After pressing Enter, a graphical window will appear displaying the output of the rostopic list command. This graphical window is called the rqt visualization interface. It provides a visual representation of the ROS topics and their connections in the system. Please note that this rqt visualization interface needs to be opened on an ROS robot with a screen.



The graphical window you see represents the currently running nodes and their data relationships. The ellipses represent nodes, the curved arrows represent data flow, and the rectangular boxes represent topics. Nodes communicate with each other through topics to exchange data. In the diagram, you can see that the three nodes correspond to the <node> entries in the launch file.

Here's a brief overview:

(1) The joint_state_publisher node reads the joint values of the robot from the robot_description parameter command, which parses the robot's URDF. It then publishes these joint values to the /joint_states topic.

(2) The robot_state_publisher node receives the joint values from the /joint_states topic and converts them into Transform (tf) format. It publishes the transformed data to the /tf topic.

(3) The rviz node (the graphical interface we see) reads the data from the /tf topic and updates the 3D model displayed in real-time.

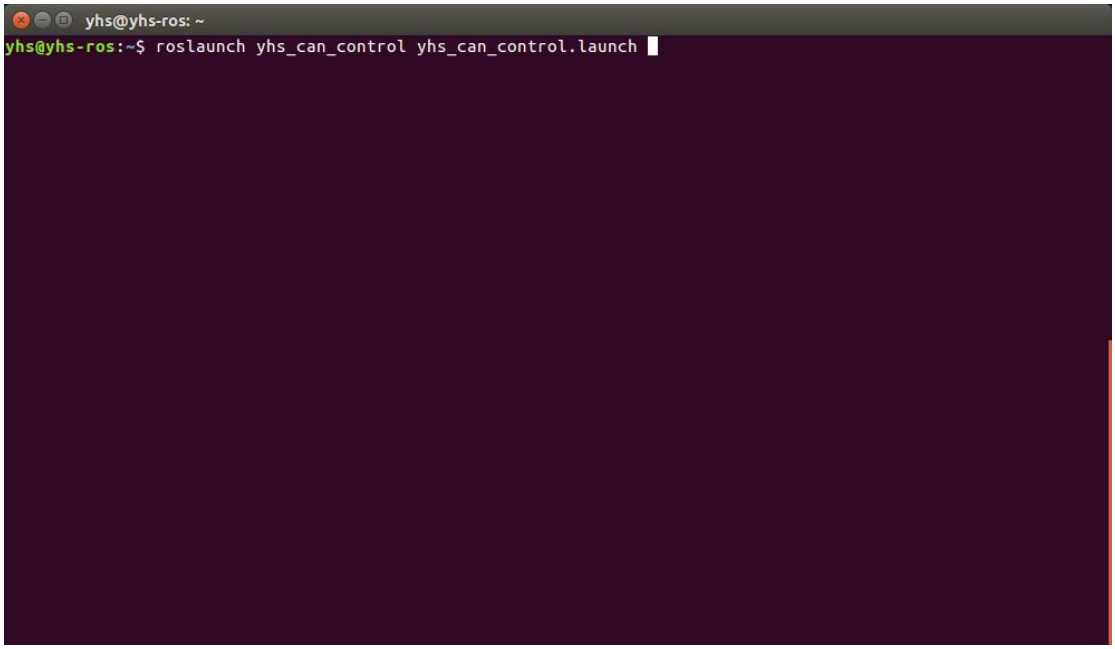
The launch file serves as the entry point for most ROS packages. When starting to learn ROS, it can be overwhelming to navigate through a package with numerous directories. In such cases, you can start by exploring the launch folder and reading each launch file. This will give you a general idea of which node from which package is being launched. You can then navigate to the respective package's src directory to find the corresponding .cpp or .py files for those nodes. This way, you can gradually understand the structure and functionality of the package.

Experiment 2: Chassis Drive and Control

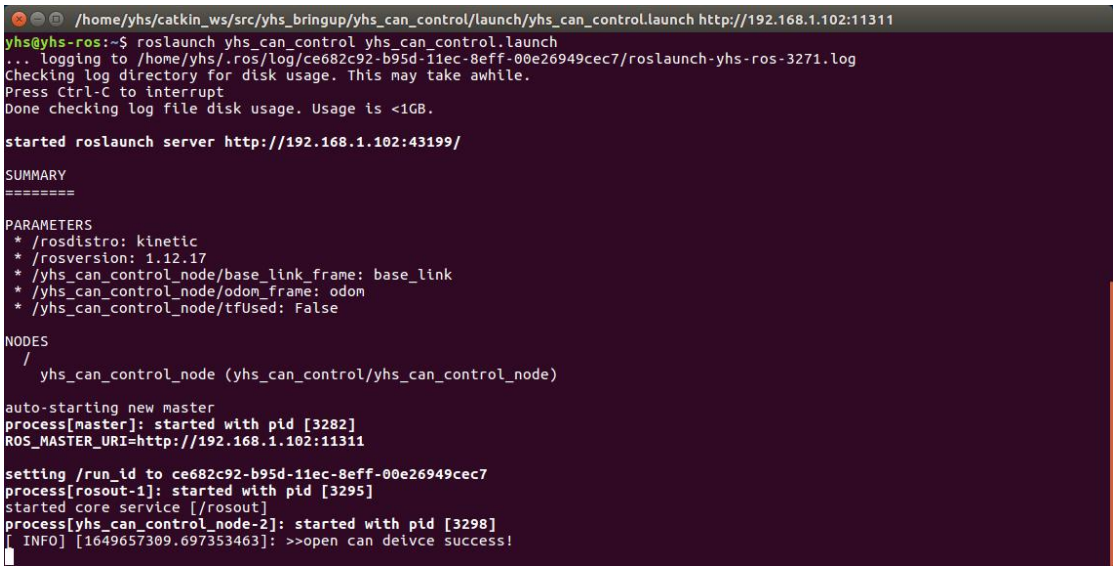
The Yuhesen robot chassis uses the CAN communication method, and direct communication between the computer and the chassis is not possible. Instead, a USB to CAN tool, which is also known as a CAN card, is used. To establish communication between the computer and the chassis, you will need a dedicated driver package that supports the chassis communication protocol and the CAN card's usage requirements.

1. To run the chassis driver program and open the terminal, enter the following command:

```
roslaunch yhs_can_control yhs_can_control.launch
```



After entering the command and pressing Enter, the terminal program will display a series of initialization messages, and eventually show a message indicating that the CAN card has been successfully opened.

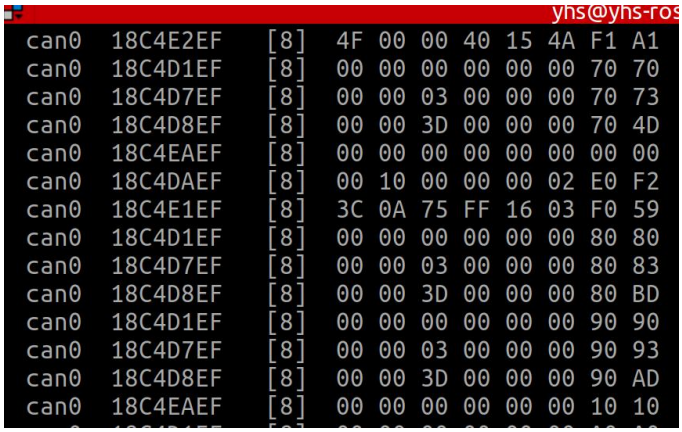


(If there is an error: Please open a new window and enter the command candump can0 to check if there is any data being transmitted from the chassis. If there is no data, please check if the blue light on the CAN card is flashing and if the CAN cable is properly connected.)

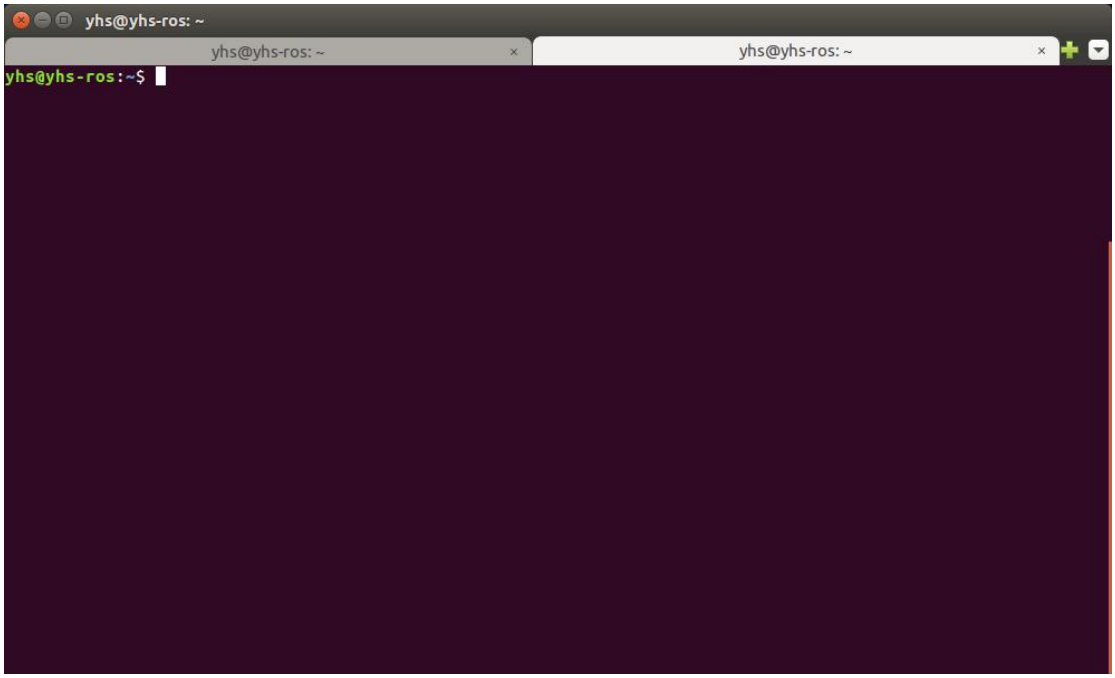
If the user wants to drive the car chassis through their own Ubuntu system by connecting the USB cable of the CAN card to their laptop, they need to perform some operations in the laptop terminal:

```
sudo ip link set can0 type can bitrate 500000
sudo ip link set can0 up
```

After the setup, if you see the blue light of the CAN card blinking, it indicates successful communication. You can use the command candump can0 to view the data transmitted from the chassis, as shown in the example below. If the candump command is not available, you can install it using the command sudo apt-get install can-utils.



2. After running a ROS package, it is common to check which topics are being published by the nodes and identify the topics that we need to use. To achieve this, you can open a new terminal alongside the current terminal by pressing "Ctrl + Shift + T" simultaneously. This allows you to have multiple terminals open for different commands or tasks.

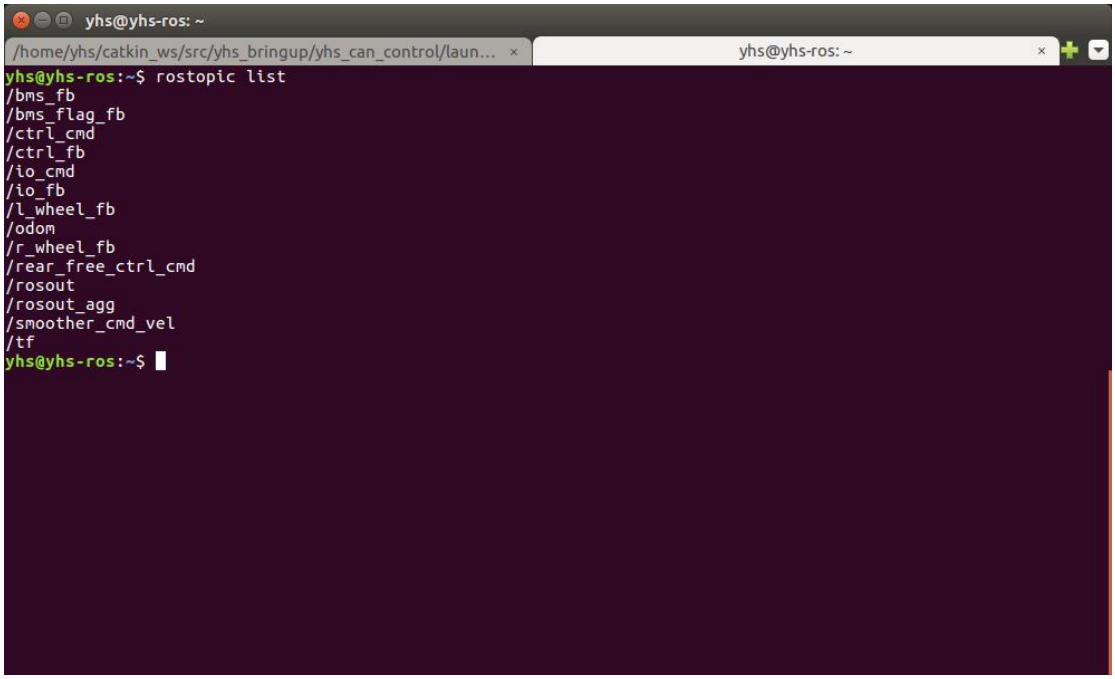


3. Then typing commands:

```
rostopic list
```

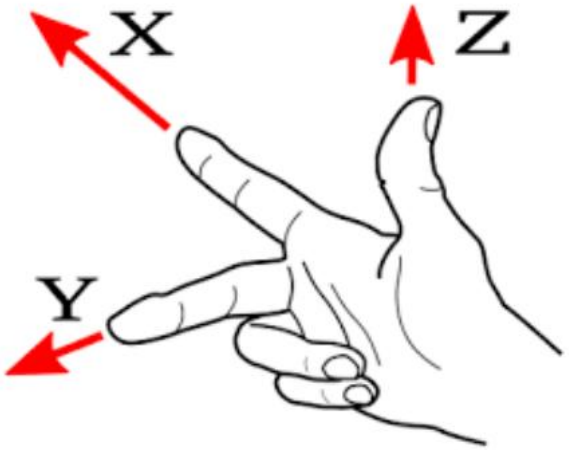
The command to view topics in ROS is rostopic list.

Upon hitting Enter, the following topics are displayed. The topic "smoother_cmd_vel" is used by the industrial control computer to send commands to the chassis for controlling its motion. **(Note: Some vehicle models may use "/cmd_vel" instead of "/smoother_cmd_vel". Please replace "/smoother_cmd_vel" with "/cmd_vel" in the following operations.)**



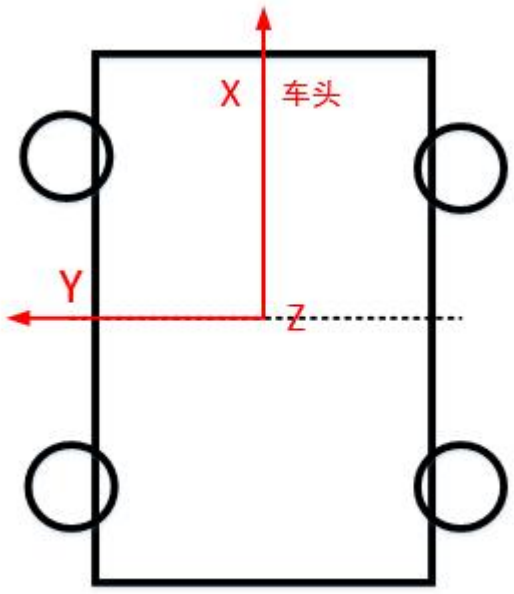
4. Before controlling the motion of the chassis, let's first understand the coordinate system used in the ROS system.

（1）In the ROS system, a right-handed coordinate system is used to represent the pose and motion of objects.

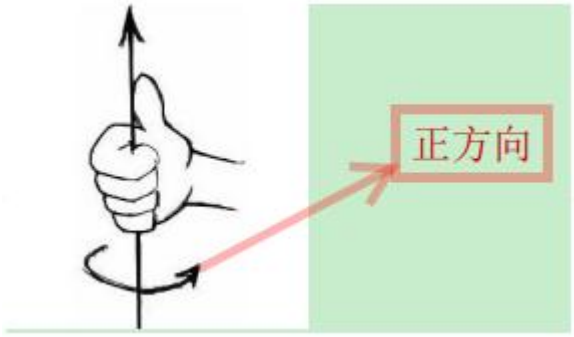


On a vehicle, this corresponds to the following:

The front of the vehicle is the positive X-axis direction.
The left side of the vehicle is the positive Y-axis direction.
The vertical upward direction is the positive Z-axis direction.



（2）The direction of rotation follows the right-hand rule. A counterclockwise rotation is considered the positive direction, while a clockwise rotation is considered the negative direction.



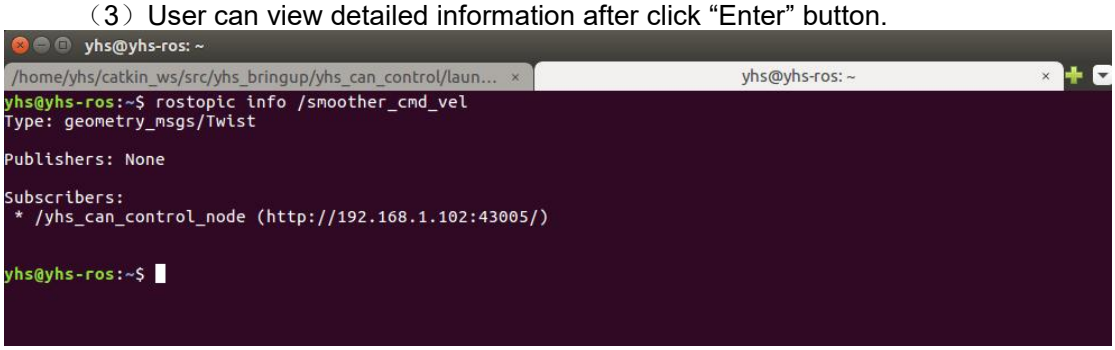
5. After understanding the coordinate system of the car chassis, let's proceed to understand the message type and content of the "smoother_cmd_vel" topic.

（1）Open terminal and typing following command.

```
rostopic  info  /smoother_cmd_vel
```

（2）To view detailed information about the "smoother_cmd_vel" topic, such as the data type, number of publishers, and number of subscribers, open a terminal and enter the command rostopic info smoother_cmd_vel.

（3）User can view detailed information after click “Enter” button.

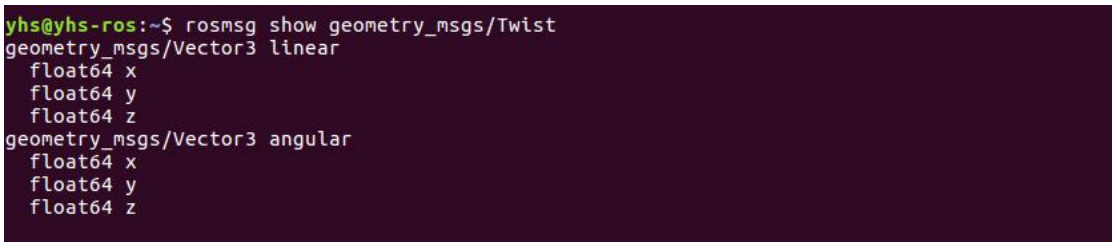


（4）Typing following command to view message contents.

```
rosmsg  show  geometry_msgs/Twist
```

The command "rosmsg show" can display detailed information about a given message type.

（5）Click “Enter” to view detailed contents.



The content of the message represents the following meanings:

```
geometry_msgs/Vector3 linear        Linear Velocity
  float64 x              Velocity in the X-axis direction, measured in m/s
  float64 y              Velocity in the Y-axis direction, set to 0 when sending velocity
commands
  float64 z              Velocity in the Z-axis direction, set to 0 when sending velocity
commands
geometry_msgs/Vector3 angular     Angular Velocity
  float64 x          Angular velocity around the X-axis, set to 0 when sending velocity
commands
  float64 y          Angular velocity around the Y-axis, set to 0 when sending velocity
commands   float64 z              Angular velocity around the Z-axis, measured in rad/s
```

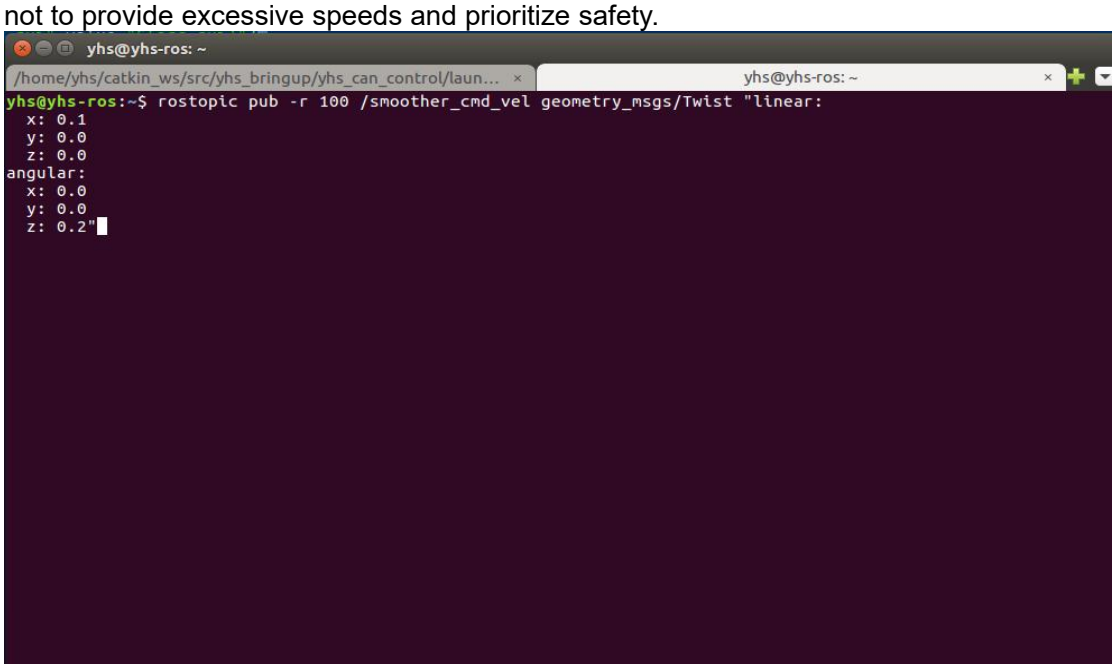
6. Issuing Velocity Commands

（1）Opening terminal and typing commands:

```
 rostopic pub -r 100 /smoother_cmd_vel geometry_msgs/Twist "linear:
  x: 0.1
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.2"
```

-r followed by the publishing frequency, where 100 represents 100 times per second. After entering "rostopic pub -r 100 /smoother_cmd_vel", you can press "Tab" to autocomplete the remaining content. In fact, the "Tab" key can be used not only in this case but also when entering any command. By typing a few letters of a command and pressing "Tab," it will automatically complete it. If the command cannot be automatically completed, check for input errors or the existence of the command.

Press Enter, switch the remote control to automatic mode, and at this point, the car will perform circular motion with a linear velocity of 0.1 m/s and an angular velocity of 0.2 rad/s. When testing, it is advisable not to provide excessive speeds and prioritize safety.

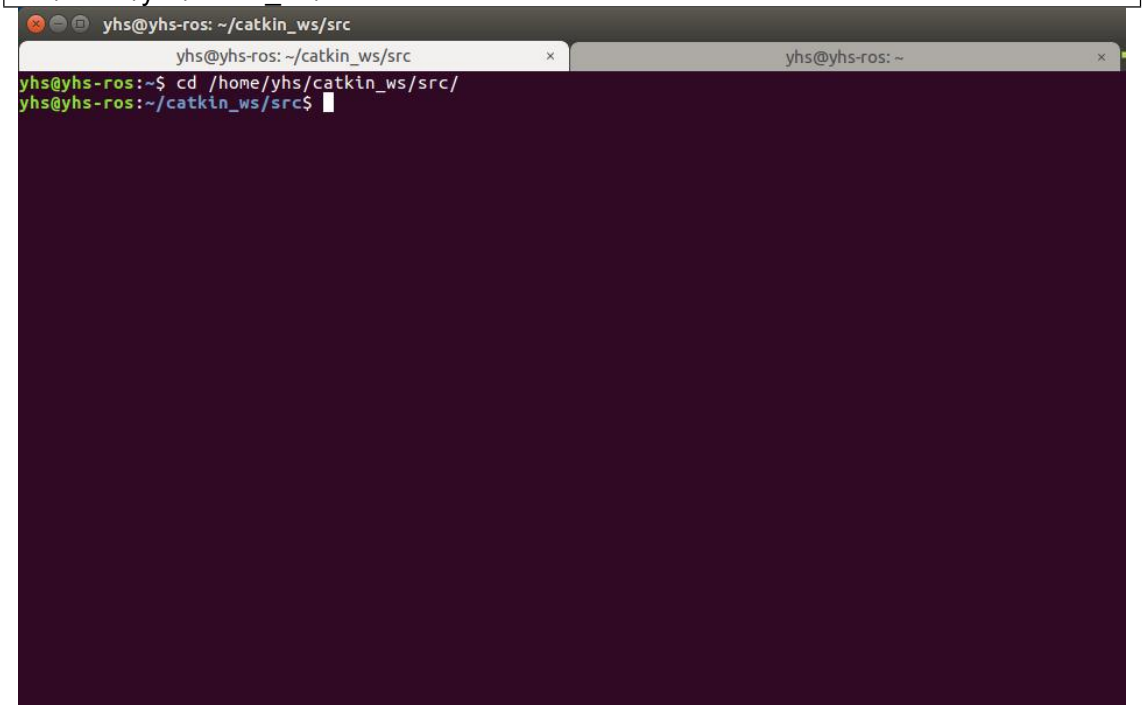


（2）Press "Ctrl + c" to stop the velocity publishing, and after it is stopped, the car will immediately come to a halt.

7. Issuing Velocity Through Program Node

（1）First, you need to create a new ROS source package. In Ubuntu, open a terminal program and enter the following command to enter the ROS workspace:

```
cd /home/yhs/catkin_ws/src
```

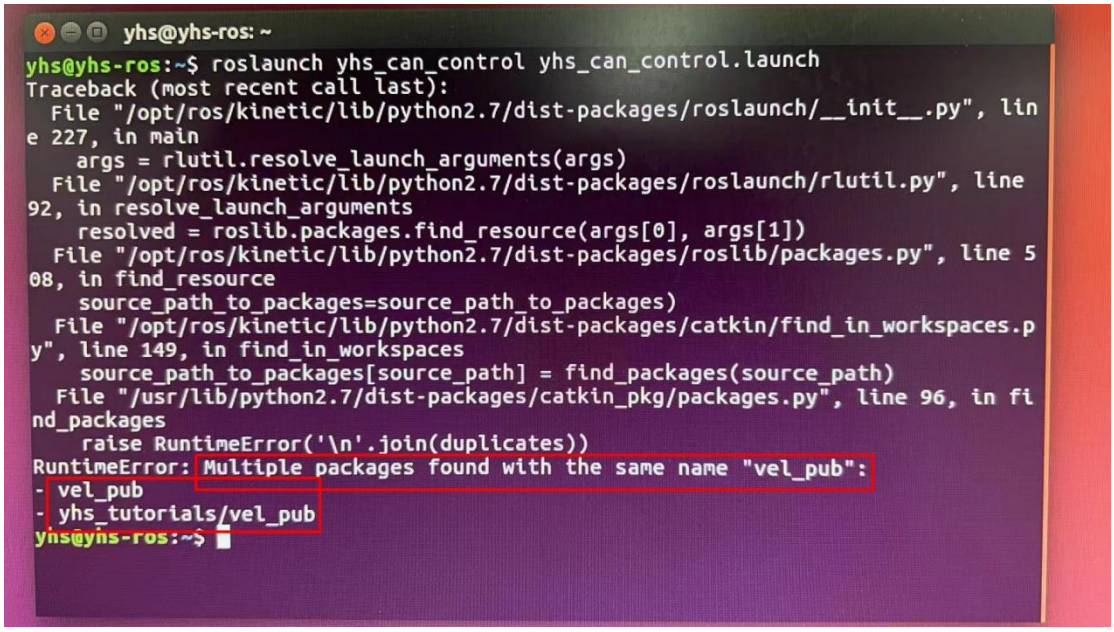


（2）After pressing Enter, you will enter the ROS workspace. Then, enter the following command to create a new ROS source package:

```
catkin_create_pkg  vel_pub  roscpp  geometry_msgs
```

（Note: All the source packages for the examples and experiments in this manual have already been created and can be found in the "yhs_tutorials" folder. Therefore, when you are creating your own packages, please make sure that the package names do not conflict with the examples. For example, if the package name is "vel_pub," you should not use the same name. Otherwise, you will encounter an error as shown in the following image. Simply delete the package you created to resolve the issue.

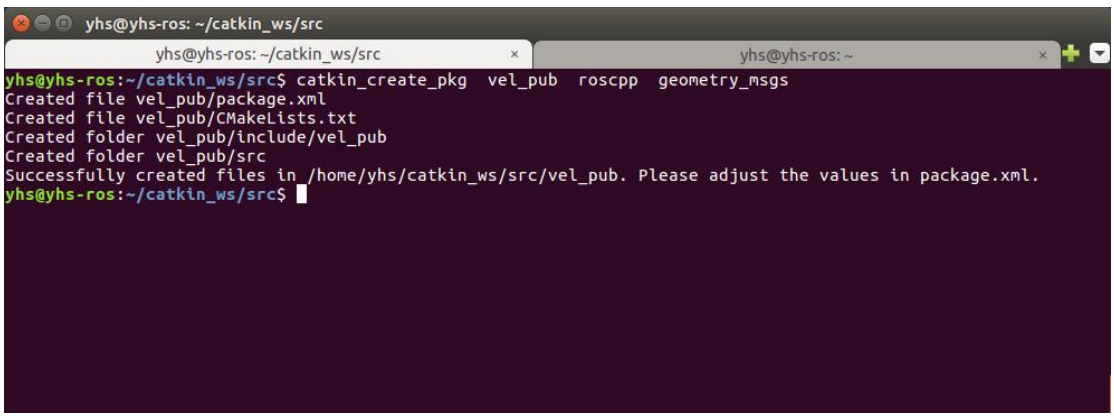
）



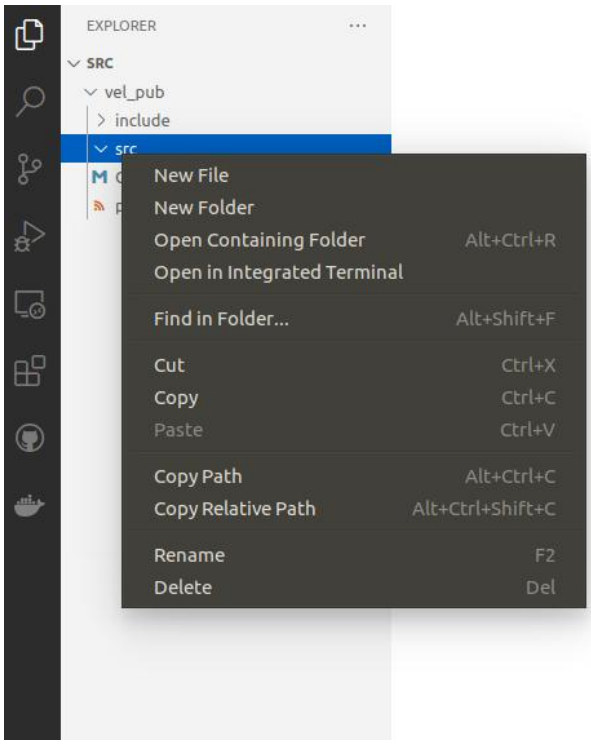
The specific meaning of this command is:

Command	Meaning
catkin_create_pkg	The command is used to create a ROS source package (package).
vel_pub	The name of new creating ROS source package.
roscpp	For this example, since it is written in C++, you will need the C++ dependencies.
geometry_msgs	The package that includes the message package format file for robot movement speed.

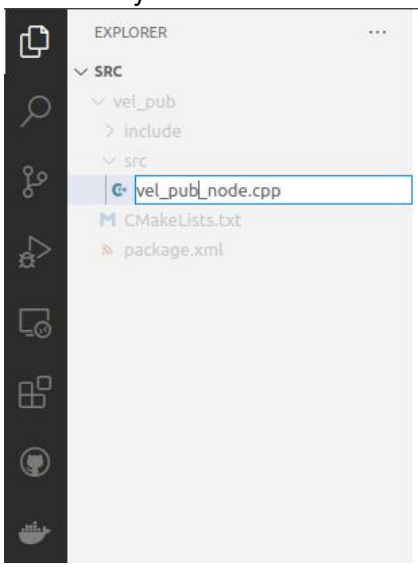
After pressing the Enter key, you will see the following information, indicating that the new ROS software package has been created successfully.



（3）In the IDE, you will see that the workspace now has a new folder named "vel_pub." Right-click on the "src" subfolder of the package, select "New File," and create a new code file. (Here, we are using Visual Studio Code, but you can also use other text editing tools such as Vim, Gedit, etc.)



（4）The newly created code file can be named as "vel_pub_node.cpp" in this case.



（5）Once the naming is complete, you can start writing the code for "vel_pub_node.cpp" on the right side of the IDE. The contents of the file are as follows:

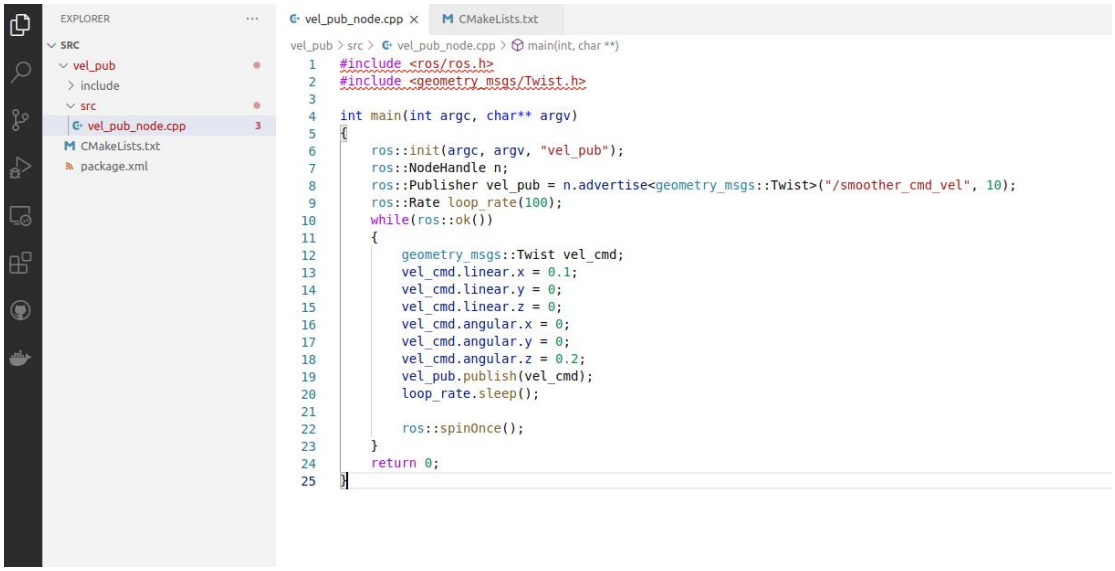
```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "vel_pub");
    ros::NodeHandle n;
    ros::Publisher vel_pub = n.advertise<geometry_msgs::Twist>("/smoother_cmd_vel",
```

```
    10);
    ros::Rate loop_rate(100);
    while(ros::ok())
    {
        geometry_msgs::Twist vel_cmd;
        vel_cmd.linear.x = 0.1;
        vel_cmd.linear.y = 0;
        vel_cmd.linear.z = 0;
        vel_cmd.angular.x = 0;
        vel_cmd.angular.y = 0;
        vel_cmd.angular.z = 0.2;
        vel_pub.publish(vel_cmd);
        loop_rate.sleep();
        ros::spinOnce();
    }
    return 0;
}
```

1) At the beginning of the code, two header files are included. One is the ROS system header file, and the other is the definition file for the motion speed structure type geometry_msgs::Twist.
2) The main function of a ROS node is defined as int main(int argc, char** argv), which follows the same parameter definition as in other C++ programs.
3) In the main function, the first step is to call ros::init(argc, argv, "vel_ctrl") to initialize the node. The third parameter of the function is the name of the node.
4) Next, a ros::NodeHandle object n is declared, and a publisher object vel_pub is created using n. The parameters passed to the vel_pub specify that it will broadcast data of type geometry_msgs::Twist on the topic "/smoother_cmd_vel". The control of the robot is achieved through this broadcasting mechanism. Regarding your question about why data is broadcasted on the topic "/smoother_cmd_vel" and how the robot knows which topic to listen to for executing the velocity, the answer lies in the conventions and common practices within ROS. In ROS, there are established conventions for topics, such as "/scan" for laser scanner data publication or "/tf" for coordinate frame transformation publication. Similarly, the choice of the topic "smoother_cmd_vel" for controlling robot velocity is a convention that has been commonly adopted. By subscribing to the "/smoother_cmd_vel" topic, the robot knows where to listen for velocity commands.
5) To continuously send velocities, you can use a while(ros::ok()) loop, where the ros::ok() return value serves as the loop termination condition. This allows the loop to exit gracefully when the program is being shut down.
6) To send velocity values, declare an object of type geometry_msgs::Twist named vel_cmd, and assign the velocity values to this object.
7) vel_cmd.linear.x represents the translational velocity of the robot for forward and backward motion, where positive values indicate forward motion and negative values indicate backward motion, the unit is meters per second (m/s).
8) vel_cmd.linear.y represents the translational velocity of the robot for left and right motion, where positive values indicate leftward motion and negative values indicate rightward motion, the unit is meters per second (m/s).
9) vel_cmd.angular.z (note the term "angular") represents the rotational velocity of the robot, where positive values indicate leftward rotation and negative values indicate rightward rotation, the unit is radians per second (rad/s).
10) Since the other values have no significance for the robot, you can assign them all to zero.
11) Once vel_cmd is assigned with the desired values, you can use the publisher object vel_pub to publish it to the topic "/smoother_cmd_vel". The chassis control node of the robot will receive the velocity values from this topic and forward them to the hardware chassis for execution.
12) To allow other callback functions to execute (though not used in this example), you can call the ros::spinOnce() function.

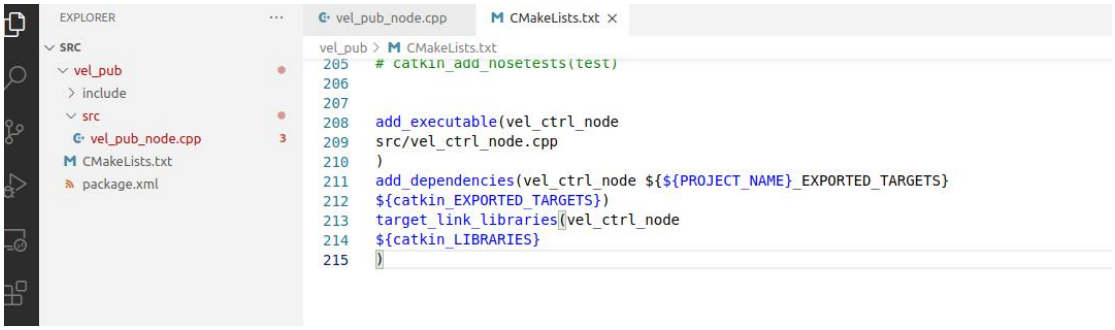
After finishing the code, if it hasn't been saved to the file yet, you will see a black dot on the right side of the file name "vel_pub_node.cpp" in the upper-right editing area. This indicates that the file has not been saved. To save the code file, press the keyboard shortcut "Ctrl + S". After saving, the

black dot next to the file name "vel_pub_node.cpp" in the upper-right editing area will change to a close button, indicating that the file has been successfully saved.



（6）Once the code is written, you need to add the file name to the build file to compile it. The build file is located in the directory of "vel_pub" package and named "CMakeLists.txt". In the IDE interface, click on the "CMakeLists.txt" file on the left side, and its contents will be displayed on the right side. At the end of the "CMakeLists.txt" file, add a new compilation rule for "vel_pub_node.cpp". The content should be as follows:

```
add_executable(vel_ctrl_node
src/vel_pub_node.cpp
)
add_dependencies(vel_ctrl_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(vel_pub_node
${catkin_LIBRARIES}
)
```



（7）After making the modifications, press the keyboard shortcut "Ctrl + S" to save the changes. The black dot on the right side of the file name, located above the code, will change to an "X," indicating that the file has been successfully saved. Now, let's proceed with the compilation of the code file.

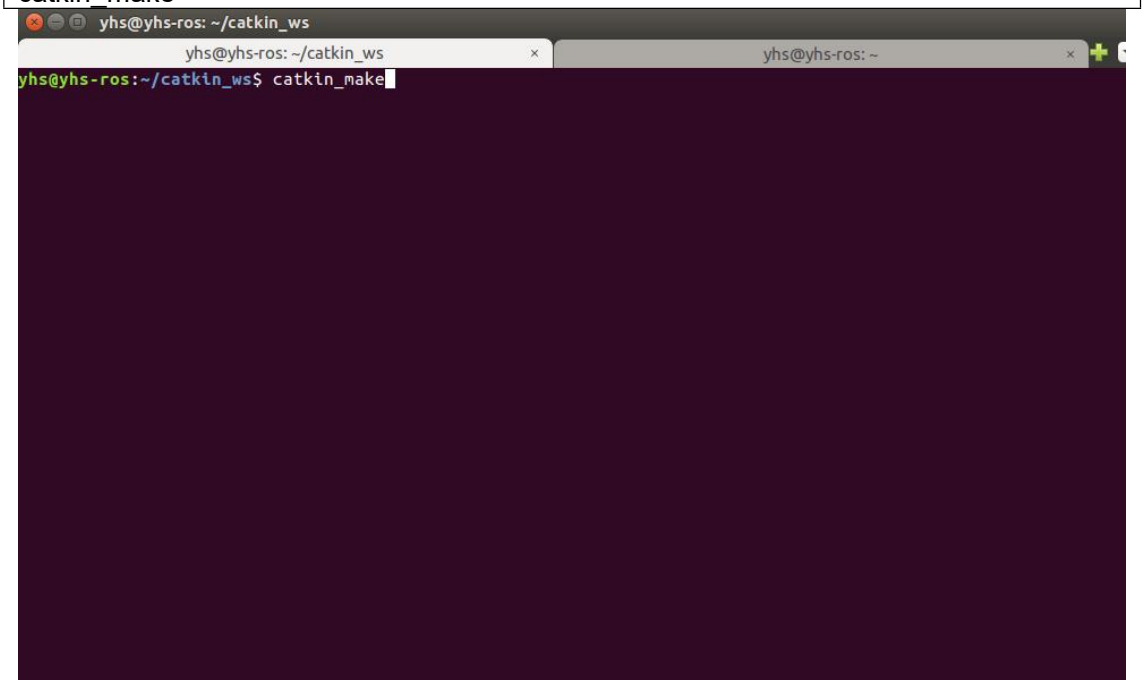
To start, open a terminal program and enter the following command to navigate to the ROS workspace:

```
cd /home/yhs/catkin_ws
```

```
yhs@yhs-ros:~$ cd /home/yhs/catkin_ws/
yhs@yhs-ros:~/catkin_ws$
```

（8）After navigating to the ROS workspace directory, you can execute the following command to start the compilation:

```
catkin_make
```



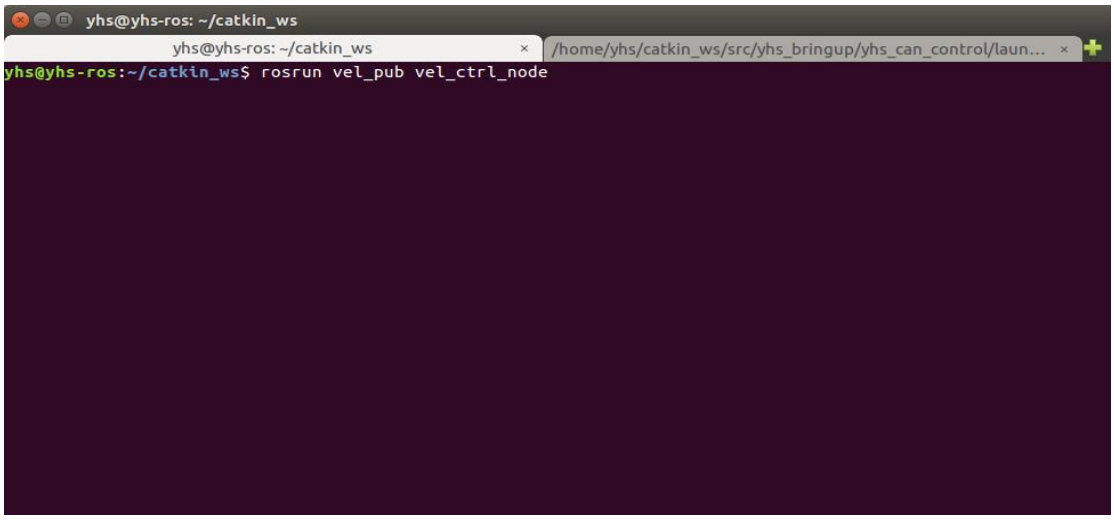
After executing the command catkin_make, you will see scrolling compilation information on the terminal. The compilation process will continue until you see the message "Linking CXX executable /home/yhs/catkin_ws/devel/lib/vel_pub/vel_ctrl_node," which indicates that the new vel_pub_node has been successfully compiled.

The path mentioned in the message may vary depending on your ROS workspace setup.

```
[ 42%] Built target lslidar_c16_msgs_generate_messages_cpp
[ 43%] Built target lslidar_c16_msgs_generate_messages_py
[ 43%] Linking CXX executable /home/yhs/catkin_ws/devel/lib/vel_pub/vel_ctrl_node
```

(9) Since you have already started the chassis CAN control node, you can directly launch the vel_ctrl_node node. To start a new terminal window, press the keyboard shortcut "Ctrl + Alt + T". Then, enter the following command:

```
rosrun vel_pub vel_ctrl_node
```

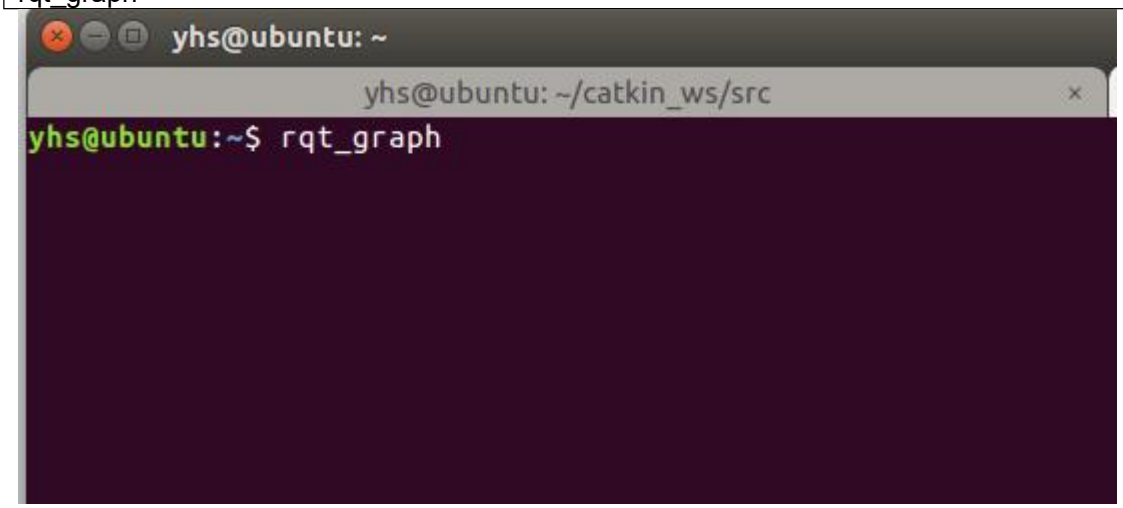


Indeed, in this case, we are using the rosrun command instead of roslaunch. rosrun is used to launch individual ROS nodes.

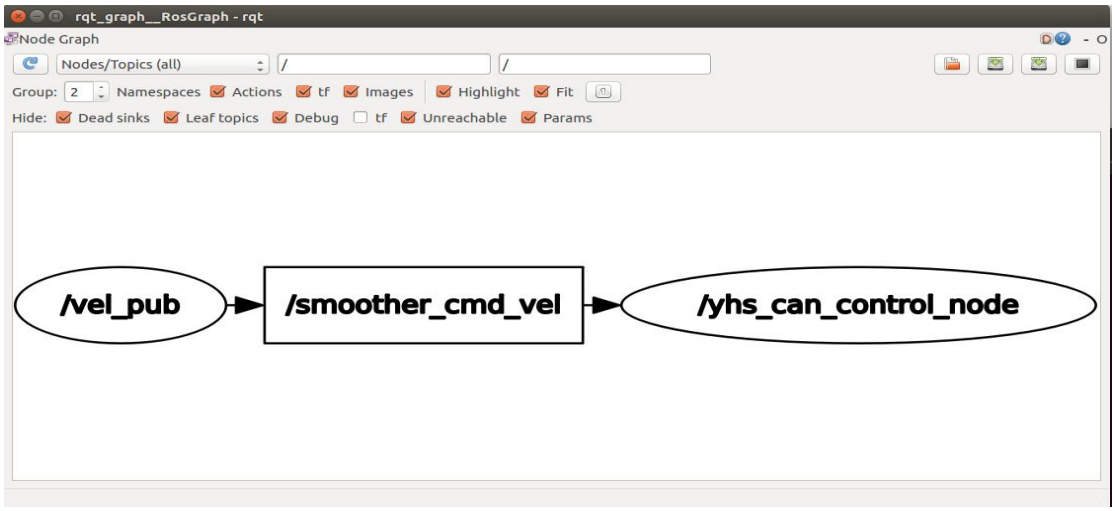
After pressing Enter, you should see the robot slowly performing circular motion with a linear velocity of 0.1 m/s and an angular velocity of 0.2 rad/s.

(10) We can use a command to view the ROS node network status. To do so, open a terminal and enter the following command:

```
rqt_graph
```



After pressing Enter, you should see a list of active ROS nodes printed directly in the terminal window.



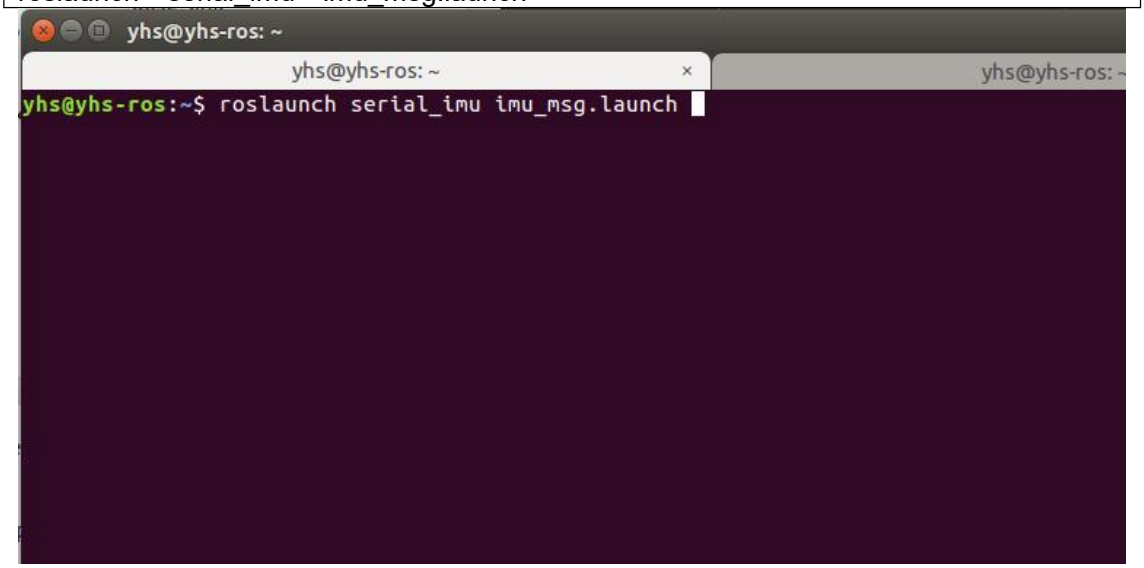
You can see that the vel_pub node we wrote sends velocity message packets to the chassis control node through the topic "/smoother_cmd_vel". Once the yhs_can_control_node receives the velocity message, it uses CAN communication to send it to the VCU (Vehicle Control Unit) of the chassis, thereby controlling the motion of the robot base.

Experiment 3: IMU Data Processing

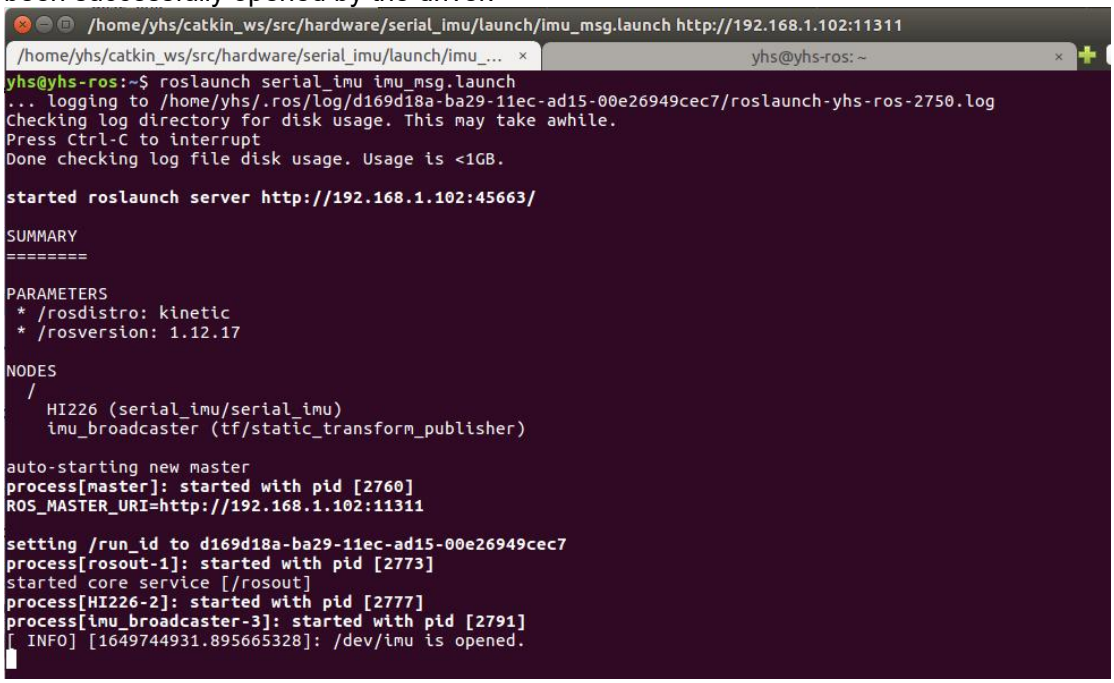
During the motion of the robot, relying solely on the data from the wheel encoders to calculate the rotation angle can lead to accumulating errors over time. Therefore, it is necessary to incorporate a high-precision IMU (Inertial Measurement Unit) to provide information such as angles and accelerations.

1. To run the IMU driver package, typically provided by the IMU manufacturer as a ROS driver package, typing commands:

```
roslaunch serial_imu imu_msg.launch
```



2. After pressing Enter, you see the message "/dev/imu is opened," it indicates that the IMU device has been successfully opened by the driver.

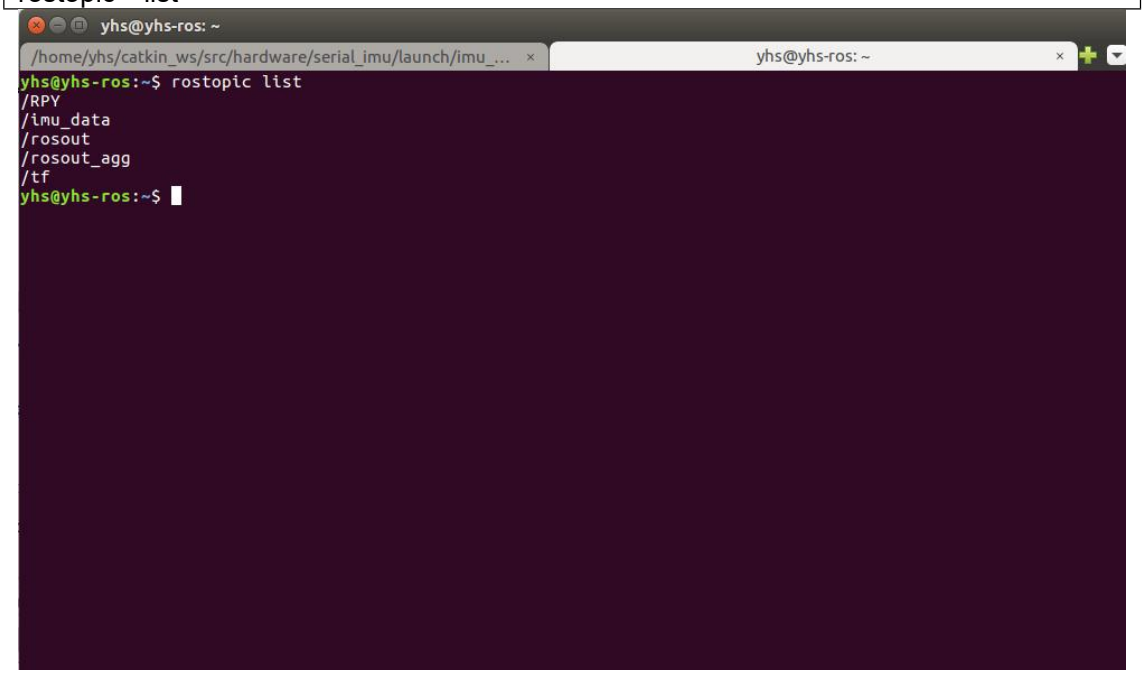


If you encounter a yellow warning message while running the IMU driver, it typically indicates a non-fatal issue or a warning condition. In most cases, these warnings do not affect the overall functionality of the IMU driver or its usability.

```
process[imu_broadcaster-3]: started with pid [6599]
[ INFO] [1691034119.768522870]: /dev/imu is opened.
[ WARN] [1691034119.778722314]: MSG to TF: Quaternion Not Properly Normalized
[ WARN] [1691034119.784706351]: MSG to TF: Quaternion Not Properly Normalized
[ WARN] [1691034119.790661995]: MSG to TF: Quaternion Not Properly Normalized
[ WARN] [1691034119.856731522]: MSG to TF: Quaternion Not Properly Normalized
```

3. To view the topics related to the IMU node, enter the following command and press Enter:

```
rostopic list
```

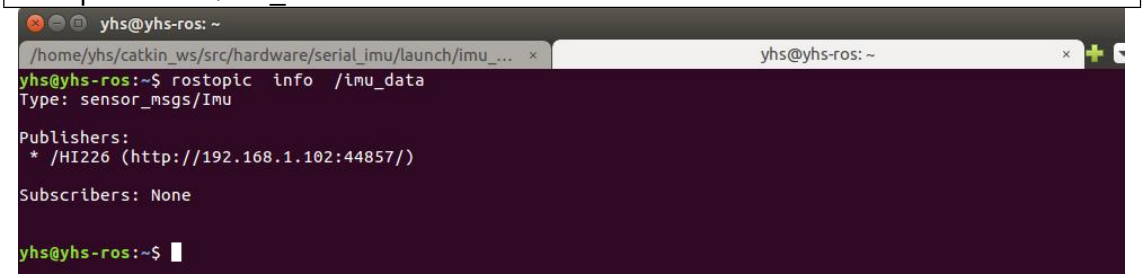


The topic name for the IMU is typically "imu_data".

4. To view the messages published on a specific topic, you can follow by below steps:

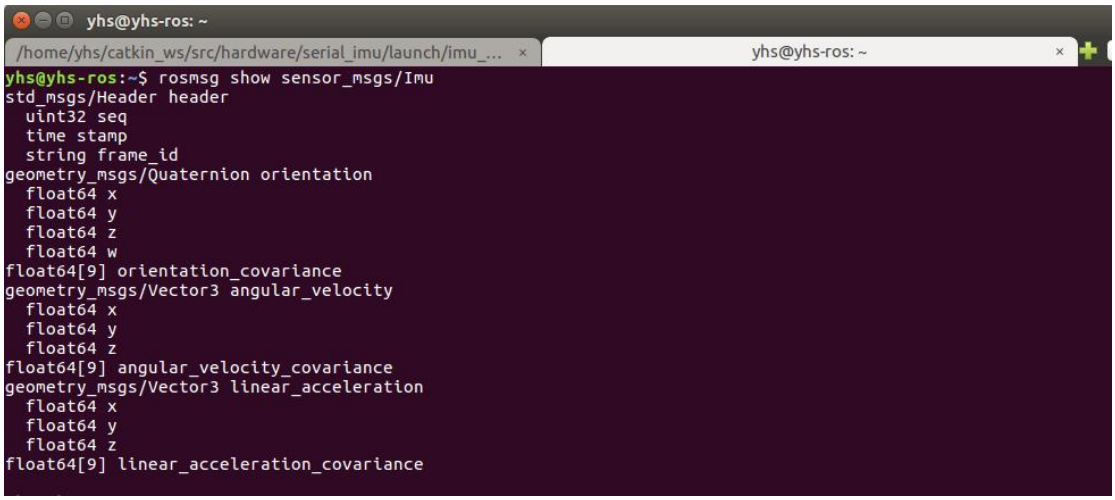
(1) To check the information of the IMU topic, enter the following command and press Enter:

```
rostopic info /imu_data
```



Having identified that the message type for the IMU topic is "sensor_msgs/Imu", you can proceed to view the content of the messages.

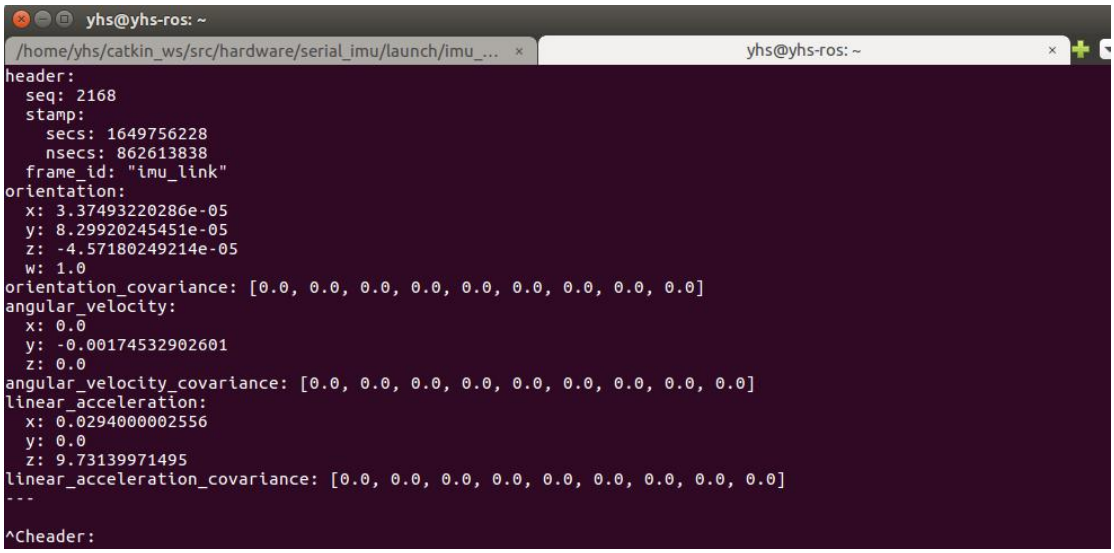
(2) To check the information of the IMU message, enter the following command and press Enter:



The detailed meanings as below:

std_msgs/Header header	**message header**
uint32 seq	
time stamp	
string frame_id	
geometry_msgs/Quaternion orientation	**Quaternions**
float64 x	
float64 y	
float64 z	
float64 w	
float64[9] orientation_covariance	**Quaternion Covariance**
geometry_msgs/Vector3 angular_velocity	**Angular velocity Unit: rad/s**
float64 x	
float64 y	
float64 z	
float64[9] angular_velocity_covariance	**Angular Velocity Covariance**
geometry_msgs/Vector3 linear_acceleration	**Linear acceleration Unit: m/s^2**
float64 x	
float64 y	
float64 z	
float64[9] linear_acceleration_covariance	**Linear acceleration Covariance**

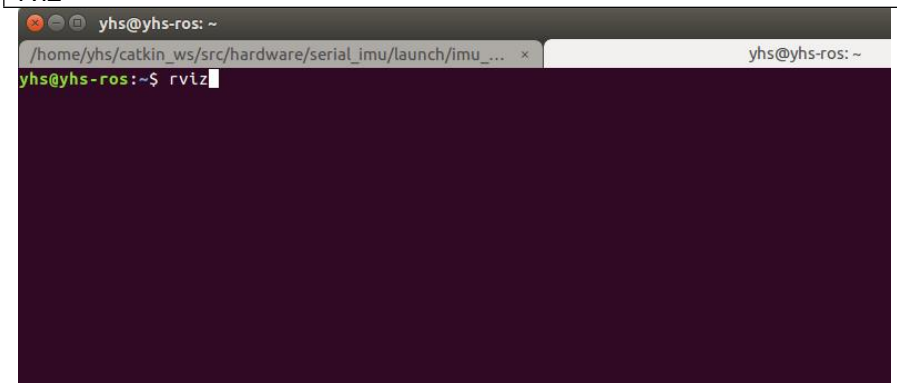
To view the IMU data when the robot is stationary, use the “rostopic echo /imu_data ” command:



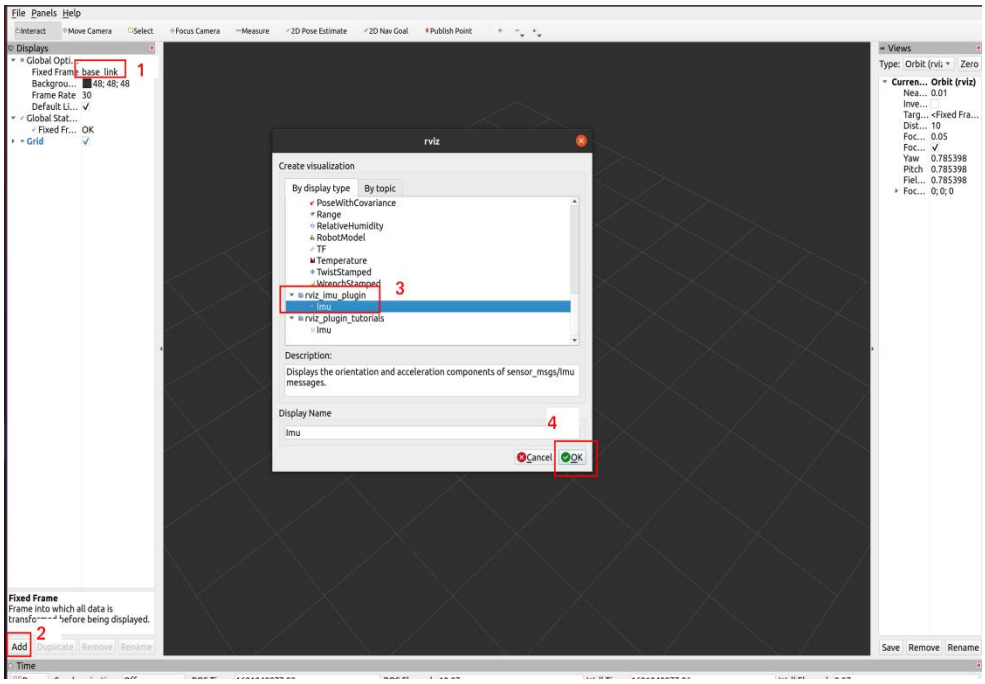
You can observe that if the IMU is placed horizontally, then in a stationary state, the angular velocities along all three axes should be close to zero. The accelerations along the x and y axes should be close to zero, while the acceleration along the z-axis should be close to the acceleration due to gravity.

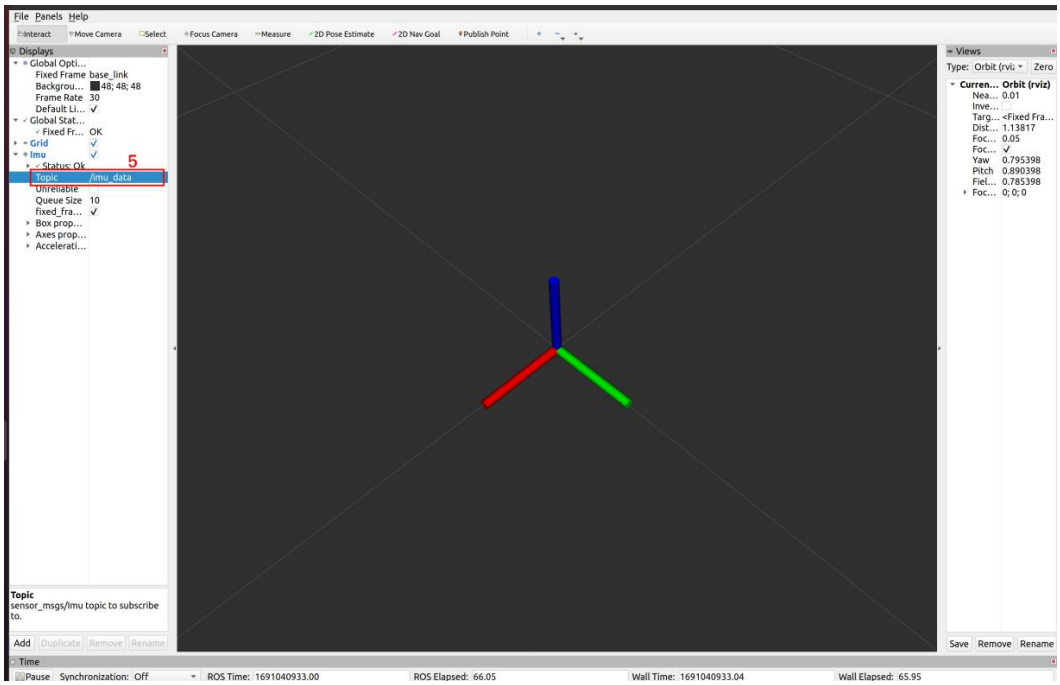
(3) To visualize the IMU rotation direction and rotation angle in RViz while the robot is rotating, please follow these steps:

rviz

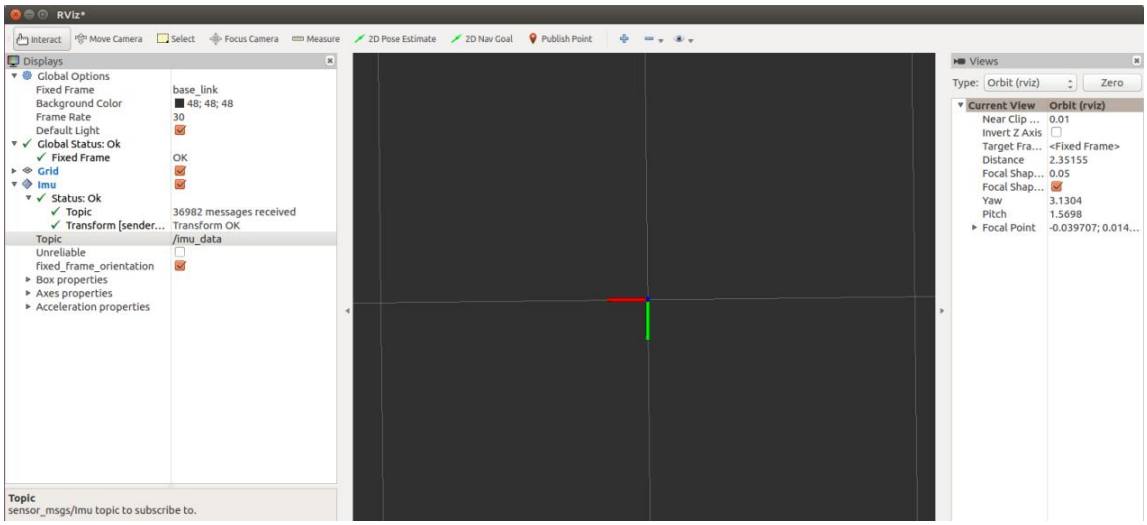


After opening rviz, continue to follow the steps on below images:





You can see that three axes appear on the black grid: the red axis represents the x-axis, the green axis represents the y-axis, and the blue axis represents the z-axis. Then, we will remotely control the car to rotate in place 90 degrees to the left and then 90 degrees to the right. Observe if the directions of the three axes remain the same as at the beginning. This will help determine the rotation accuracy of the IMU.

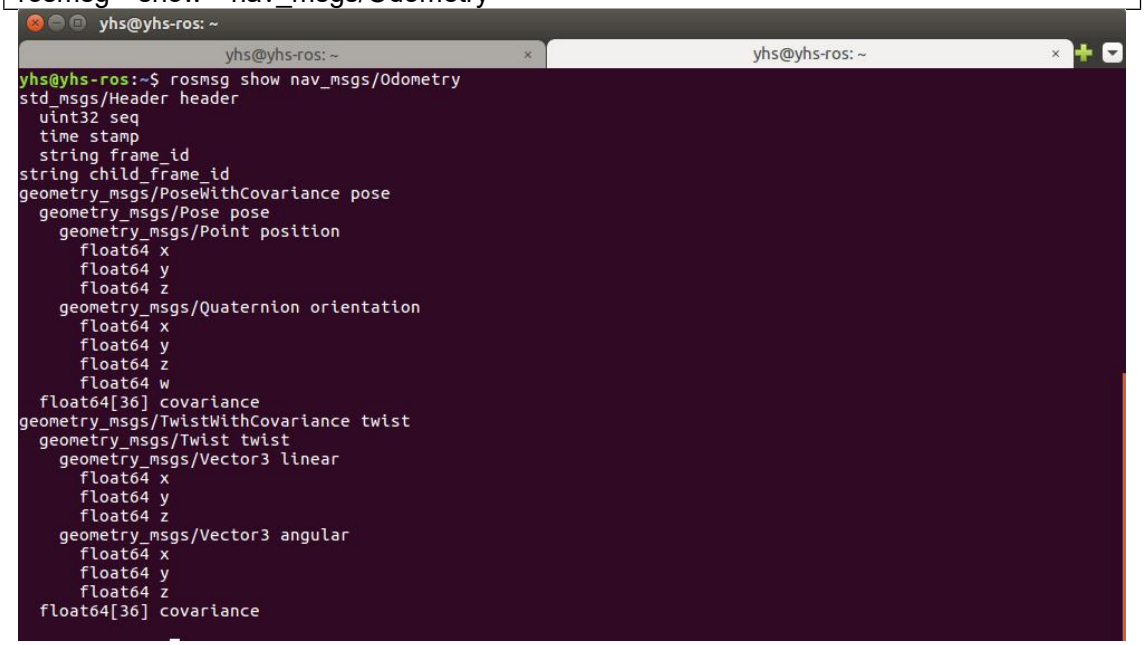

RViz*
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
base_link
Background Color
48; 48; 48
Frame Rate
30
Default Light
Global Status: Ok
Fixed Frame
OK
Grid
Imu
Status: Ok
Topic
36982 messages received
Transform [sender...
Transform OK
Topic
/imu_data
Unreliable
fixed_frame_orientation
Box properties
Axes properties
Acceleration properties
Topic
sensor_msgs/Imu topic to subscribe to.
Views
Type:
Orbit (rviz)
Zero
Current View
Orbit (rviz)
Near Clip ...
0.01
Invert Z Axis
Target Fra...
<Fixed Frame>
Distance
2.35155
Focal Shap...
0.05
Focal Shap...
Yaw
3.1304
Pitch
1.5698
Focal Point
-0.039707; 0.014...

Experiment 4: Odometry Fusion

Gmapping and Cartographer are mapping algorithms that heavily rely on accurate odometry information. Subscribing to high-precision odometry output is crucial for generating large-scale and loop-closure maps. It plays a key role in achieving high map accuracy.

1. To publish odometry data, the topic name is typically set as "odom" and the message type is "nav_msgs/Odometry". Please use the following command and press Enter:

```
rosmsg show nav_msgs/Odometry
```



You can see the content of the message, and the meanings of each part are as follows:

```
std_msgs/Header header                          Message header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position      The real-time coordinates of the robot only
utilize the x and y axes.
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation     The real-time orientation of the robot
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance   When using the robot_pose_ekf package for sensor
fusion, it is not recommended to set the covariance values to zero.
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear              Linear velocity
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular             Angular velocity
      float64 x
      float64 y
```

```
        float64 z
    float64[36] covariance                    Covariance
```

2. Once you know the message type for odometry, which is "odom", you can publish it using the rostopic pub command. Referring to the example provided in "http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom", you can modify the linear and angular velocities to match your actual chassis speed. Here is the code:

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

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;

  double x = 0.0;
  double y = 0.0;
  double th = 0.0;

  double vx = 0.1;                    Linear velocity(m/s). Assign the linear
velocity of the chassis to vx.

  double vy = 0;
  double vth = 0.1;                   Angular velocity(rad/s). Assign the angular
velocity of the chassis to vth.

  ros::Time current_time, last_time;
  current_time = ros::Time::now();
  last_time = ros::Time::now();

  ros::Rate r(1.0);
  while(n.ok()){

    ros::spinOnce();                  // check for incoming messages
    current_time = ros::Time::now();

    //compute odometry in a typical way given the velocities of the robot
    double dt = (current_time - last_time).toSec();
    double delta_x = (vx * cos(th) - vy * sin(th)) * dt;      Calculate the unit time change
of robot x-axis coordinates
    double delta_y = (vx * sin(th) + vy * cos(th)) * dt;      Calculate the unit time change
of robot y-axis coordinates
    double delta_th = vth * dt;               Calculate the unit time variation of robot
angle

    x += delta_x;
    y += delta_y;
    th += delta_th;

    //since all odometry is 6DOF we'll need a quaternion created from yaw
    //Convert real-time angles of robots into the form of four elements
    geometry_msgs::Quaternion odom_quat = tf::createQuaternionMsgFromYaw(th);

    //first, we'll publish the transform over tf
    geometry_msgs::TransformStamped odom_trans;
    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;

    //The tf conversion from base_link to Odom is not published when using
robot_pose_ekf
    odom_broadcaster.sendTransform(odom_trans);

    //next, we'll publish the odometry message over ROS
    nav_msgs::Odometry odom;
    odom.header.stamp = current_time;
    odom.header.frame_id = "odom";

    //set the position
    odom.pose.pose.position.x = x;              Real time coordinates of robot x-axis
    odom.pose.pose.position.y = y;              Real time coordinates of robot y-axis
    odom.pose.pose.position.z = 0.0;
    odom.pose.pose.orientation = odom_quat;   Real time direction of robot y-axis

    //set the velocity
    odom.child_frame_id = "base_link";
    odom.twist.twist.linear.x = vx;
    odom.twist.twist.linear.y = vy;
    odom.twist.twist.angular.z = vth;

    //publish the message
    odom_pub.publish(odom);

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

3. In our chassis driver package, the CAN data parsing for the chassis has been implemented, and the odometry is being published. Please enter the command and press "Enter":

```
roslaunch  yhs_can_control  yhs_can_control.launch
```

```
yhs@yhs-ros:~$ roslaunch yhs_can_control yhs_can_control.launch
... logging to /home/yhs/.ros/log/0ce22c04-bac9-11ec-8e81-00e26949cec7/roslaunch-yhs-ros-2515.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://192.168.1.102:42013/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17
 * /yhs_can_control_node/base_link_frame: base_link
 * /yhs_can_control_node/odom_frame: odom
 * /yhs_can_control_node/tfUsed: False

NODES
  /
    yhs_can_control_node (yhs_can_control/yhs_can_control_node)

auto-starting new master
process[master]: started with pid [2526]
ROS_MASTER_URI=http://192.168.1.102:11311

setting /run_id to 0ce22c04-bac9-11ec-8e81-00e26949cec7
process[rosout-1]: started with pid [2539]
started core service [/rosout]
process[yhs_can_control_node-2]: started with pid [2542]
[ INFO] [1649813321.918615650]: >>open can deivce success!
```

When the chassis is in a stationary state, we will print the odometry. Please enter the command and press "Enter":

```
rostopic    echo    /odom
```

```
yhs@yhs-ros:~$ rostopic echo /odom
header:
  seq: 21408
  stamp:
    secs: 1649813535
    nsecs: 996825051
  frame_id: "odom"
child_frame_id: "base_link"
pose:
  pose:
    position:
      x: 0.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
  covariance: [0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10000000000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10000000000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000
0000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2]
twist:
  twist:
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.0
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0]
---
```

As the chassis remains stationary, both its position and orientation do not change. You can now control the robot to make it move and observe the changes.

4. Open a new terminal and enter the command to start the IMU node:

```
roslaunch    serial_imu    imu_msg.launch
```

```
/home/yhs/catkin_ws/src/hardware/serial_imu/launch/imu_msg.launch http://192.168.1.102:11311
/home/yhs/catkin_ws/src/yhs_bringup/yhs_can_control/laun...  ×   /home/yhs/catkin_ws/src/hardware/serial_imu/launch/imu_...  ×
yhs@yhs-ros:~$ roslaunch serial_imu imu_msg.launch
... logging to /home/yhs/.ros/log/0ce22c04-bac9-11ec-8e81-00e26949cec7/roslaunch-yhs-ros-2753.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://192.168.1.102:41169/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES
  /
    HI226 (serial_imu/serial_imu)
    imu_broadcaster (tf/static_transform_publisher)

ROS_MASTER_URI=http://192.168.1.102:11311

process[HI226-1]: started with pid [2770]
process[imu_broadcaster-2]: started with pid [2771]
[ INFO] [1649814084.291655360]: /dev/imu is opened.
```

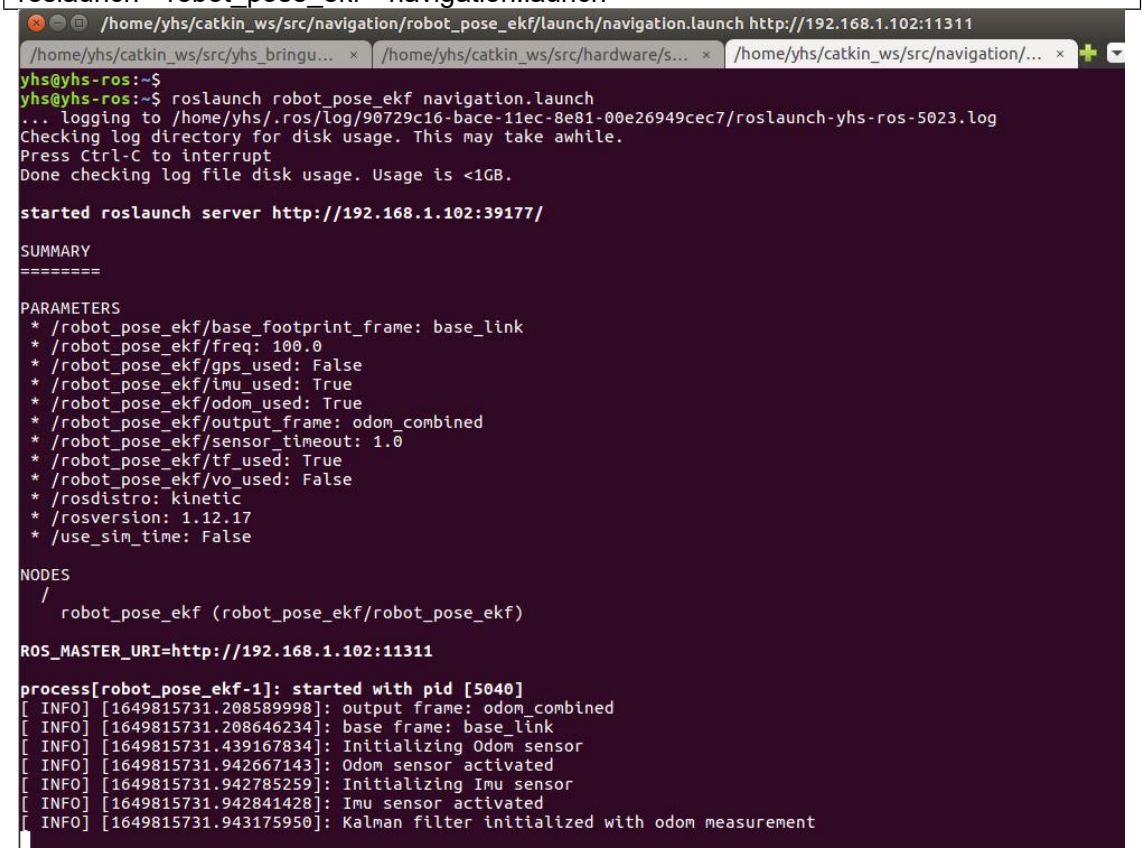
5. After starting the chassis control and IMU nodes, the final step is to launch the "robot_pose_ekf" node for fusing odometry and IMU data.

（1） First, let's take a look at the launch file. Locate the "navigation.launch" file within the "launch" folder of the "robot_pose_ekf" package at "/catkin_ws/src/navigation/robot_pose_ekf/launch/". Open it to proceed.

```
<launch>
  <arg name="set_odom_used" default="true"/>          used odom entering
  <arg name="set_imu_used" default="true"/>           used IMU entering
  <arg name="set_vo_used" default="false"/>
  <arg name="set_gps_used" default="false"/>

  <node pkg="robot_pose_ekf" type="robot_pose_ekf" name="robot_pose_ekf"
output="screen">
    <param name="output_frame" value="odom_combined"/>
    <param name="base_footprint_frame" value="base_link"/>
    <param name="freq" value="30.0"/>
    <param name="sensor_timeout" value="1.0"/>
    <param name="odom_used" value="$(arg set_odom_used)"/>
    <param name="imu_used" value="$(arg set_imu_used)"/>
    <param name="vo_used" value="$(arg set_vo_used)"/>
    <param name="gps_used" value="$(arg set_gps_used)"/>
    <param name="tf_used" value="true"/>
  </node>
</launch>
```

（2） Enter the command to start “robot_pose_ekf” node:

```
roslaunch   robot_pose_ekf   navigation.launch
```

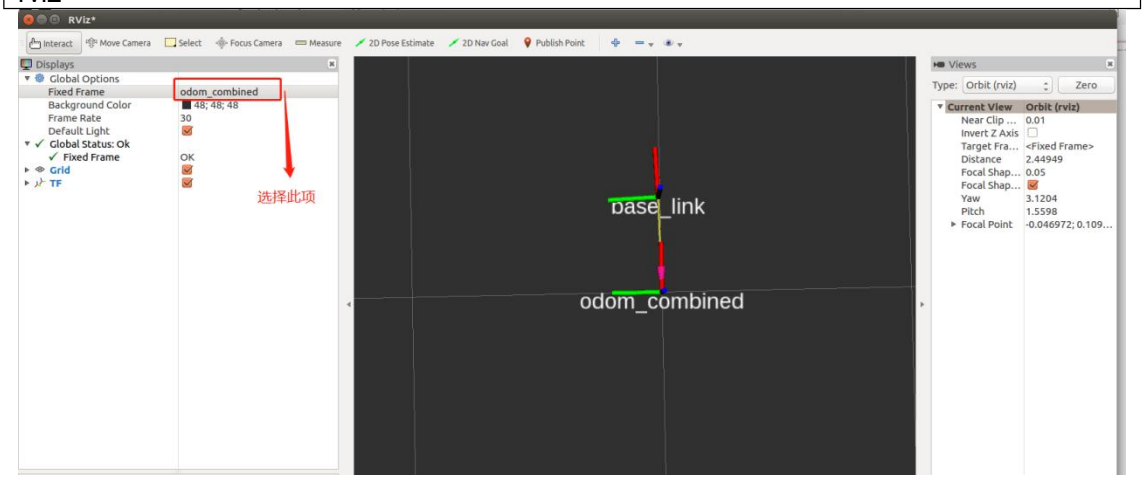


If you see the output message "Kalman filter initialized with odom measurement," it indicates that the

launch was successful, and the robot_pose_ekf node has been initialized. Additionally, it confirms that the node has successfully received odometry and IMU topics.

（3） To observe the robot's coordinate changes in RViz, please open a terminal and enter the command, then press "Enter".

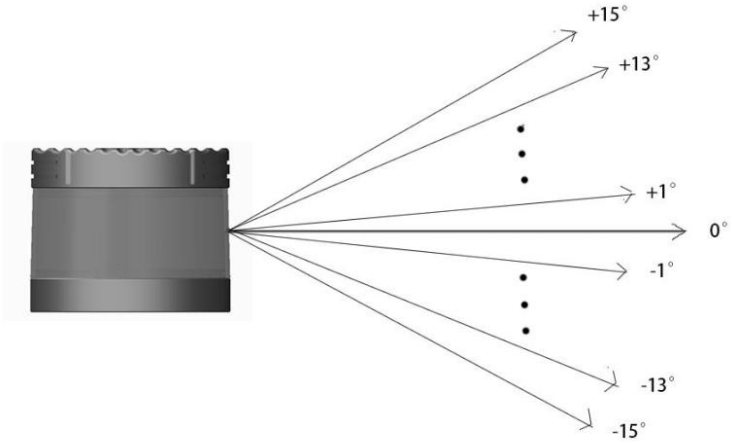
```
rviz
```



After remotely controlling the robot to move a certain distance and returning it to the starting point, if the "base_link" in RViz is close to the "odom_combined" frame, it indicates that the odometry measurements have minimal deviation. This suggests that the odometry can be reliably used for mapping and navigation purposes.

Experiment 5: 3D LiDAR

The 16-line laser radar you are using is a commonly used sensor for ground mobile robots. It utilizes the Pulsed Time-of-Flight (Pulsed ToF) method for distance measurement. The 16-line laser radar is equipped with 16 pairs of laser emitter-receiver modules. The motor driving the laser radar typically rotates at a speed of 10 Hz to perform a 360-degree scan.



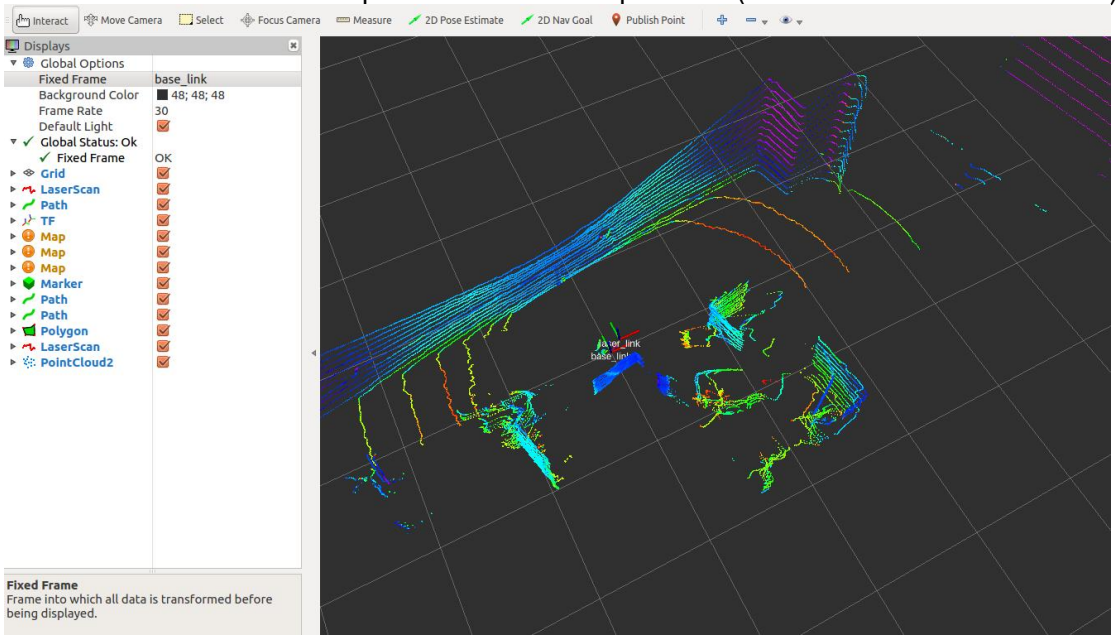
1. To get a basic experience of the characteristics of a laser radar sensor, let's start with a simple experiment.
Open a terminal and enter the command, then press "Enter":

```
roslaunch timoo_pointcloud TM16.launch
```

If adopt Robosense LiDAR, entering below command:

```
roslaunch rslidar_sdk start.launch
```

With that command, the nodes for the laser radar and RViz have been launched. Now you can visualize the laser data in RViz and explore the sensor's capabilities.(Need to connect with monitor)



In RViz, you can see a collection of points with different colors around the robot. These points represent the obstacle data collected by the laser radar. The ground reference in RViz is represented by a grid

pattern, where each grid represents a distance of 1 meter. You can try observing the relationship between these points and the surrounding obstacles. Due to measurement errors inherent in sensors, these points may exhibit slight fluctuations within a small range. You can have someone walk around the robot while observing the changes in the laser radar points.

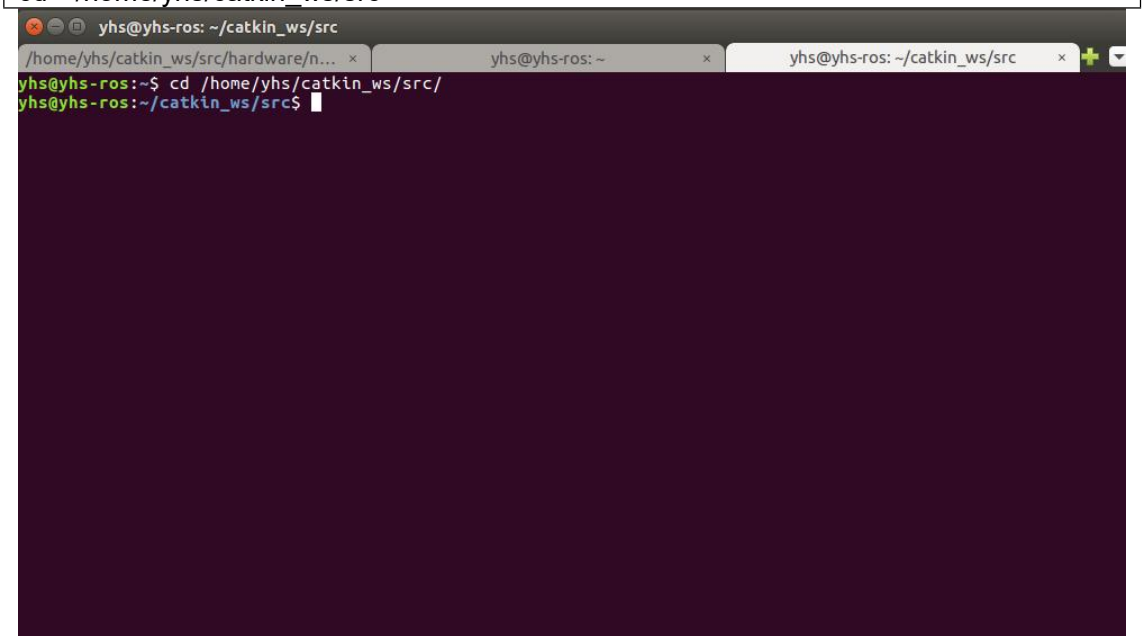
After observing the characteristics of the laser radar sensor, let's proceed to retrieve the specific measurement values from the laser radar by writing code. The general implementation approach is as follows:

1) Create a ROS node.
2) Subscribe to the laser radar message topic.
3) Parse the message packets from the laser radar and extract the distance values.
4) Use printf to output the retrieved distance values in the terminal program.

2. Here are the specific experimental steps:

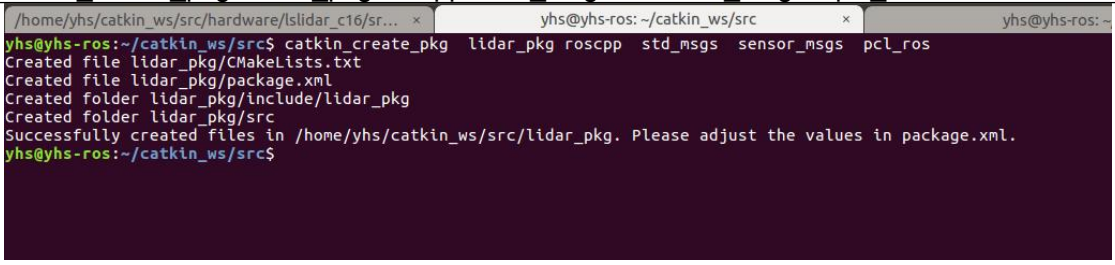
（1）Open a terminal program in Ubuntu. Enter the following command to navigate to your workspace:

```
cd  /home/yhs/catkin_ws/src
```



（2）Create a ROS package by below commands:

```
catkin_create_pkg  lidar_pkg roscpp  std_msgs  sensor_msgs  pcl_ros
```



Upon pressing the Enter key, if you see the above message, it indicates that the new ROS software package has been successfully created.

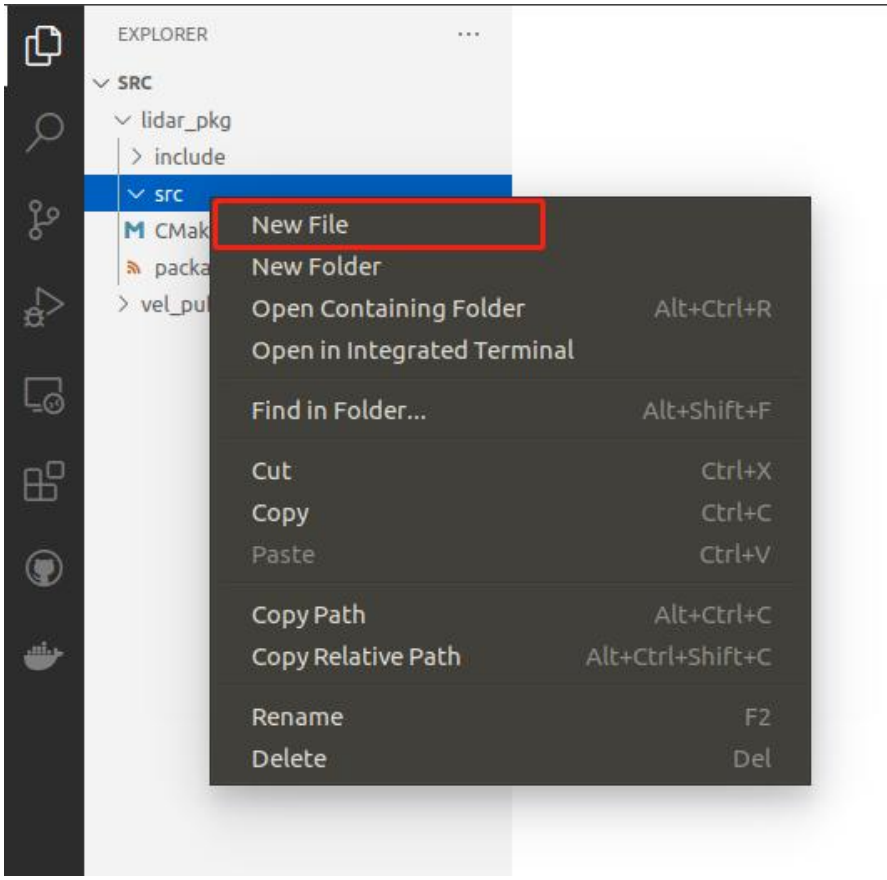
The meaning of this command:

Command	Meaning
catkin_create_pkg	creating a ROS package
lidar_pkg	name of new package
roscpp	C++ dependencies: This example is written in C++, so this

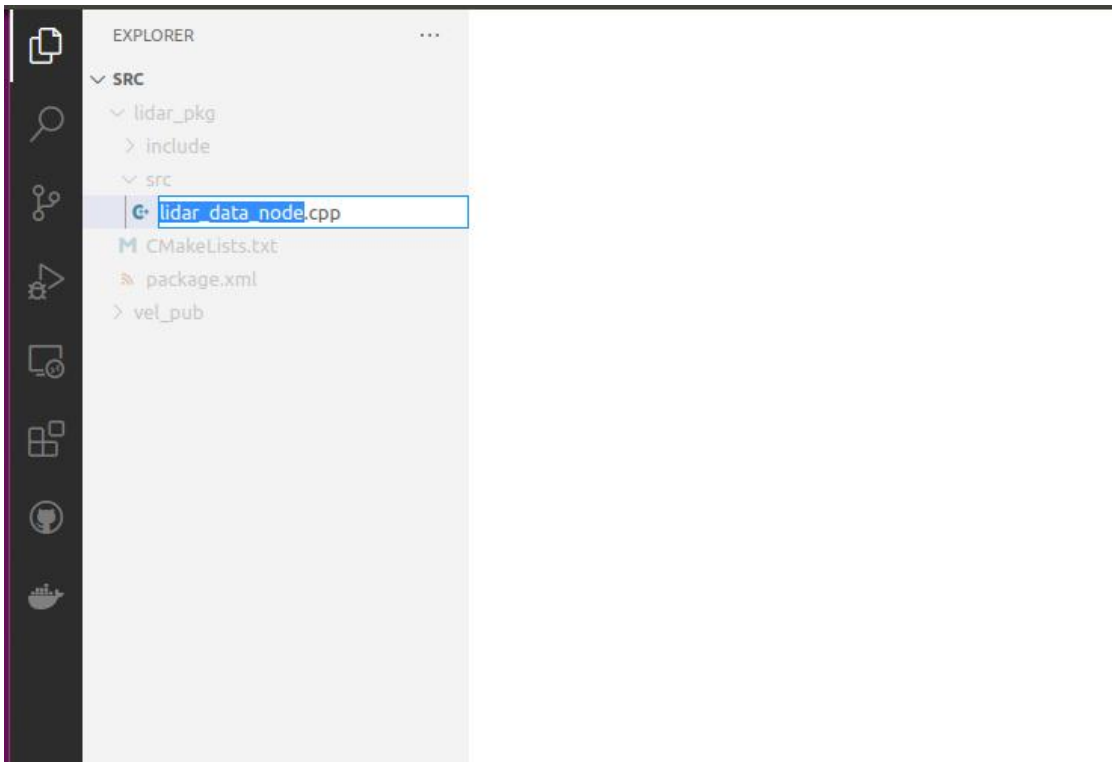
	dependency is required.
std_msgs	Standard message dependencies: This includes the String format needed for text output.
sensor_msgs	Sensor message dependencies: This is required for the laser radar data format.
pcl_ros	Dependencies for the open-source Point Cloud Library (PCL) in ROS.

3. Next, let's proceed with the IDE operations and file editing.

（1）In the IDE, you can see that the workspace now has a new folder called "lidar_pkg." Right-click on the "src" subfolder and select "New File" to create a new code file.



（2）The newly created code file can be named "lidar_data_node.cpp".



（3）Once you have named the file, you can start writing the code for "lidar_data_node.cpp" on the right side of the IDE. The content of the file can be as follows:

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_ros/point_cloud.h>

void callbackPC(const sensor_msgs::PointCloud2ConstPtr& msg)
{
    pcl::PointCloud<pcl::PointXYZ> pointCloudIn;
    pcl::fromROSMsg(*msg , pointCloudIn);
    int cloudSize = pointCloudIn.points.size();
    for(int i=0;i<cloudSize;i++)
    {
        ROS_INFO("[i= %d] ( %.2f , %.2f , %.2f)",
        i ,
        pointCloudIn.points[i].x,
        pointCloudIn.points[i].y,
        pointCloudIn.points[i].z);
    }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "lidar_data_node");
    ROS_INFO("lidar_node start");
    ros::NodeHandle nh;
    ros::Subscriber pc_sub = nh.subscribe("/timoo_points", 1 , callbackPC);
    ros::spin();
    return 0;
}
```

（4） Here's an explanation of the code:

1) Include necessary header files:

ros.h: This is the ROS system header file, which provides the necessary functions and definitions for working with ROS.

sensor_msgs/LaserScan.h: This is the ROS header file for the LaserScan message type.
pcl_ros/point_cloud.h: This header file belongs to the PCL (Point Cloud Library) package in ROS.

2) Define a callback function void callbackPC() to handle three-dimensional point cloud data. ROS automatically calls this callback function once for each received frame of point cloud data. The three-dimensional point cloud data is passed as a parameter to this callback function.

3) The parameter msg of the callback function void callbackPC() is a pointer to a memory region that stores the three-dimensional point cloud in the sensor_msgs::PointCloud2 format. In practical development, it is common to convert this point cloud format into the PCL (Point Cloud Library) point cloud format. By doing so, you can leverage the extensive functions provided by PCL to process the point cloud data effectively.

4) In the callback function void callbackPC(), define a point cloud container pointCloudIn of type pcl::PointXYZ. Use the pcl::fromROSMsg() function to convert the point cloud data in ROS format from the parameter into PCL format, and store it in the pointCloudIn container.

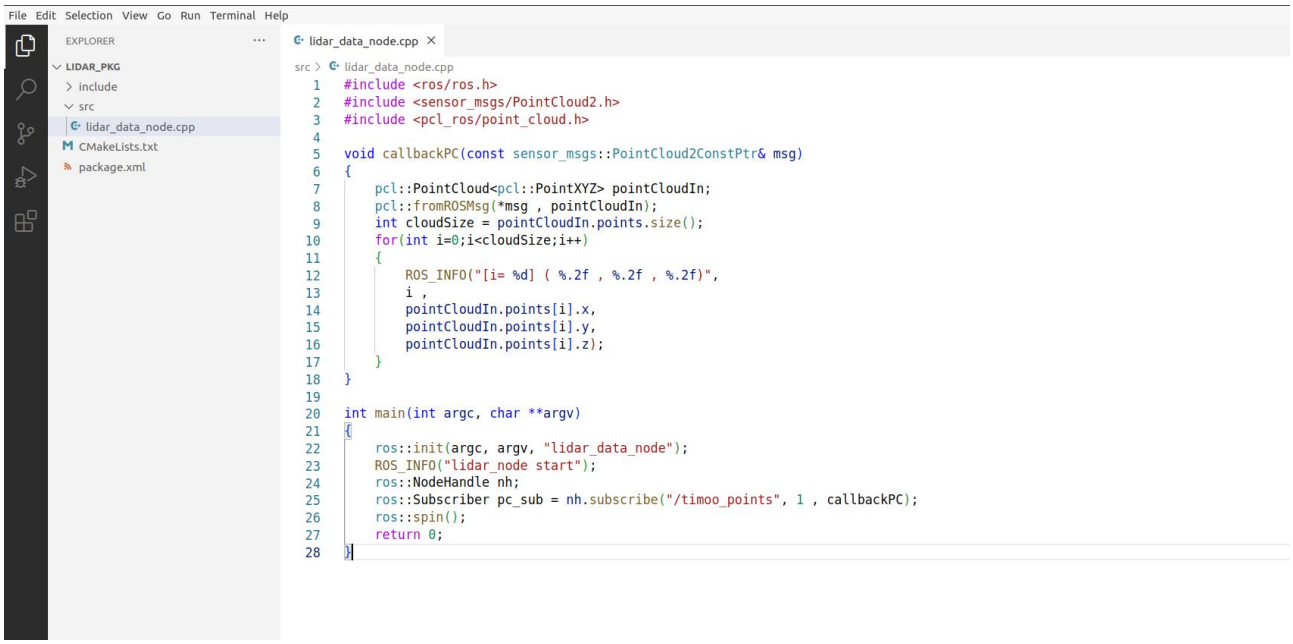
5) Get the number of three-dimensional points in the converted point cloud array pointCloudIn.points and store it in a variable cloudSize. Use a for loop to display the x, y, and z values of all points in pointCloudIn.points using ROS_INFO() in the terminal. Typically, the raw coordinate values in pointCloudIn.points are not directly used but need to be transformed into the robot's coordinate system before further processing using functions from the PCL point cloud library.

6) In the main() function, call ros::init() to initialize the node.

7) Call ROS_INFO() to output a string message to the terminal, indicating that the node has started successfully.
8) Define a ros::NodeHandle node handle named nh and use this handle to subscribe to the data from the topic "/lslidar_point_cloud" in the ROS core node. Set the callback function to the previously defined callbackPC(). The topic "/lslidar_point_cloud" is the name of the topic where the ROS node for the LiDAR publishes the three-dimensional point cloud. The LiDAR collects the three-dimensional point cloud and sends it in the form of ROS PointCloud2 messages to this topic. Our custom node, lidar_data_node, only needs to subscribe to this topic to receive the captured three-dimensional point cloud.

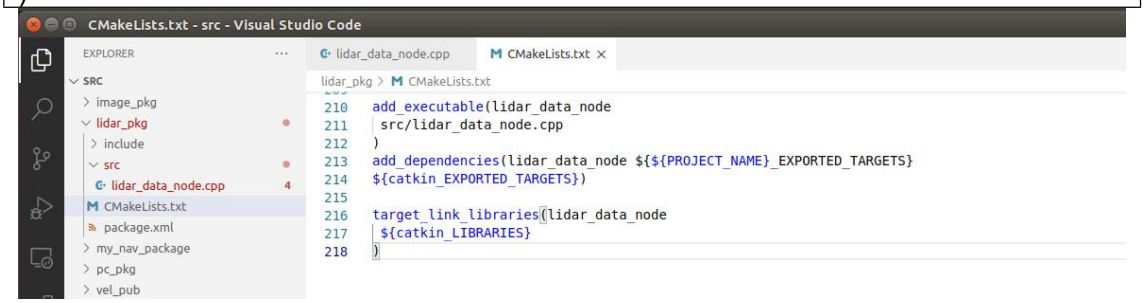
9) Call ros::spin() to block the main() function and keep the node program from exiting.

（5）After finishing writing the code, press the keyboard shortcut Ctrl+S to save the file. The small black dot on the right side of the filename at the top of the code will change to an "X," indicating that the file has been successfully saved.



（5） After finishing writing the code, you need to add the file name to the compilation file in order to compile it. The compilation file is located in the directory of lidar_pkg and named "CMakeLists.txt". In the IDE interface, click on that file on the left side, and the content of the file will be displayed on the right side. At the end of the "CMakeLists.txt" file in the lidar_pkg directory, add a new compilation rule for lidar_data_node.cpp. The content should be as follows:

```
add_executable(lidar_data_node
  src/lidar_data_node.cpp
)
add_dependencies(lidar_data_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

target_link_libraries(lidar_data_node
  ${catkin_LIBRARIES}
)
```

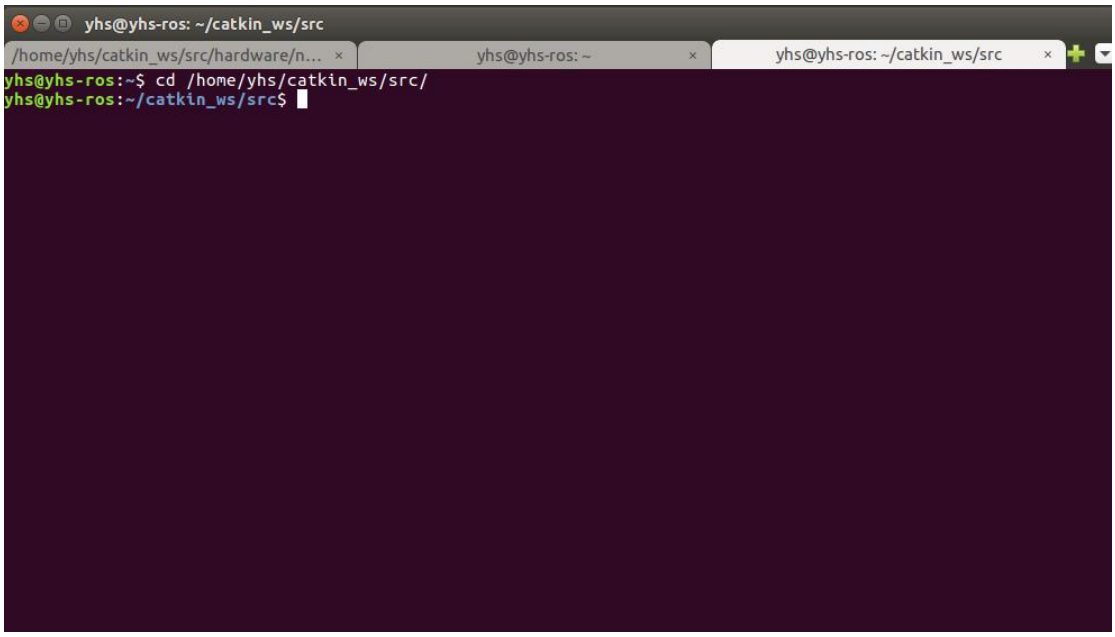


Similarly, after making the modifications, press the keyboard shortcut Ctrl+S to save the changes. The small white dot on the right side of the filename at the top of the code will change to an "X," indicating that the file has been successfully saved.

4. Let's proceed with the compilation of the code files:

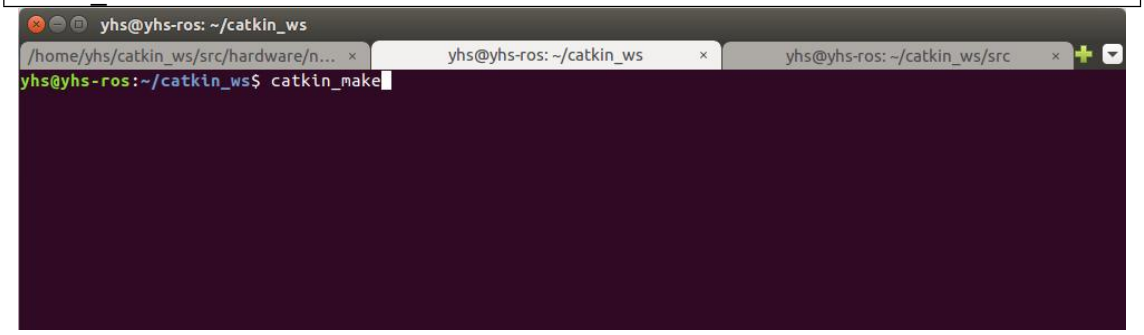
（1）Open a terminal program and enter the following command to navigate to the ROS workspace:

```
cd  /home/yhs/catkin_ws/src
```

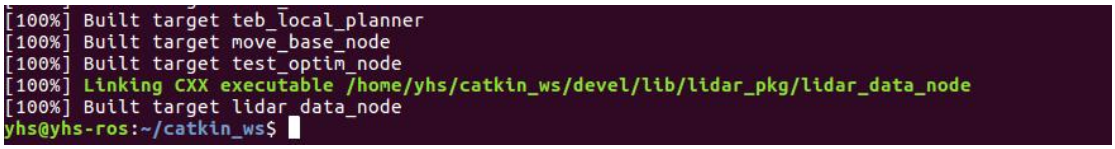


（2）Then, execute the following command to start the compilation.

```
catkin_make
```



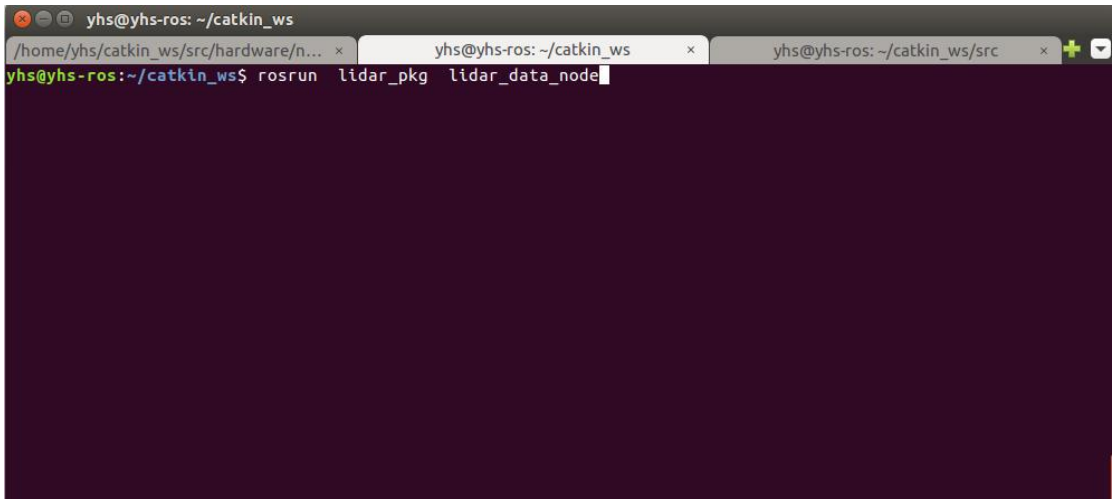
After executing this command, you will see scrolling compilation information until you see the message "[100%] Built target lidar_data_node," indicating that the new lidar_data_node node has been successfully compiled.



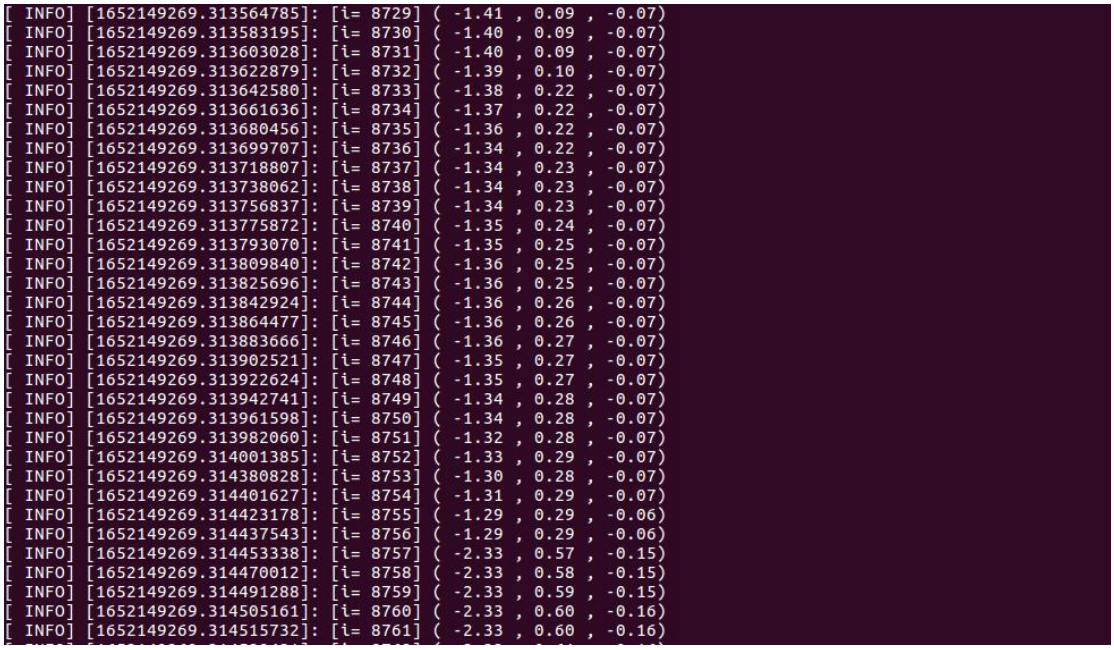
5. Now we can run the lidar_data_node that we just wrote to retrieve data from the already running LiDAR.

（1）Open a terminal program and enter the following command:

```
rosrun  lidar_pkg  lidar_data_node
```

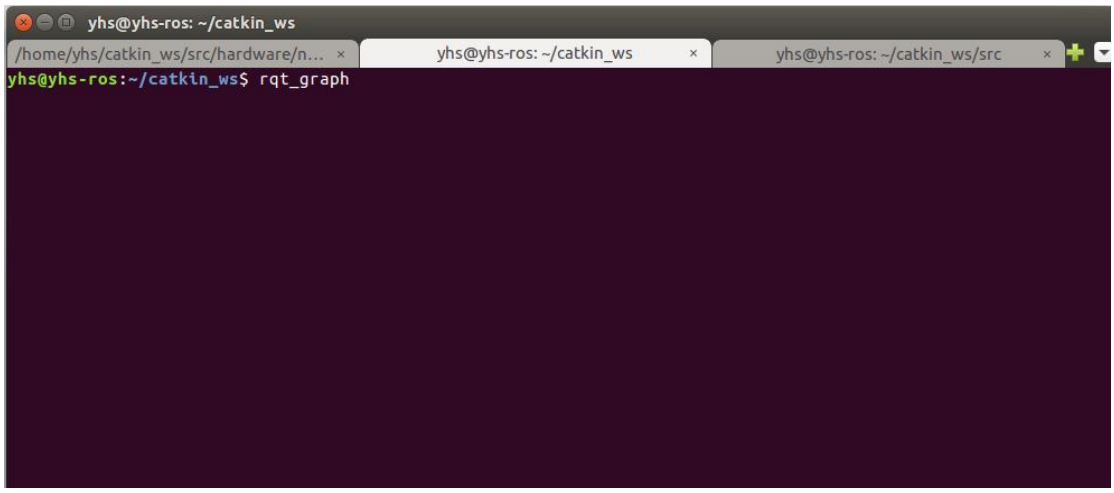


This command will start our lidar_data_node that we wrote. According to the program logic, it will continuously retrieve point cloud data from the LiDAR's topic "/lslidar_point_cloud". It will then convert the ROS-formatted three-dimensional point cloud into the PCL format and display the x, y, and z coordinates of all points in the terminal program.

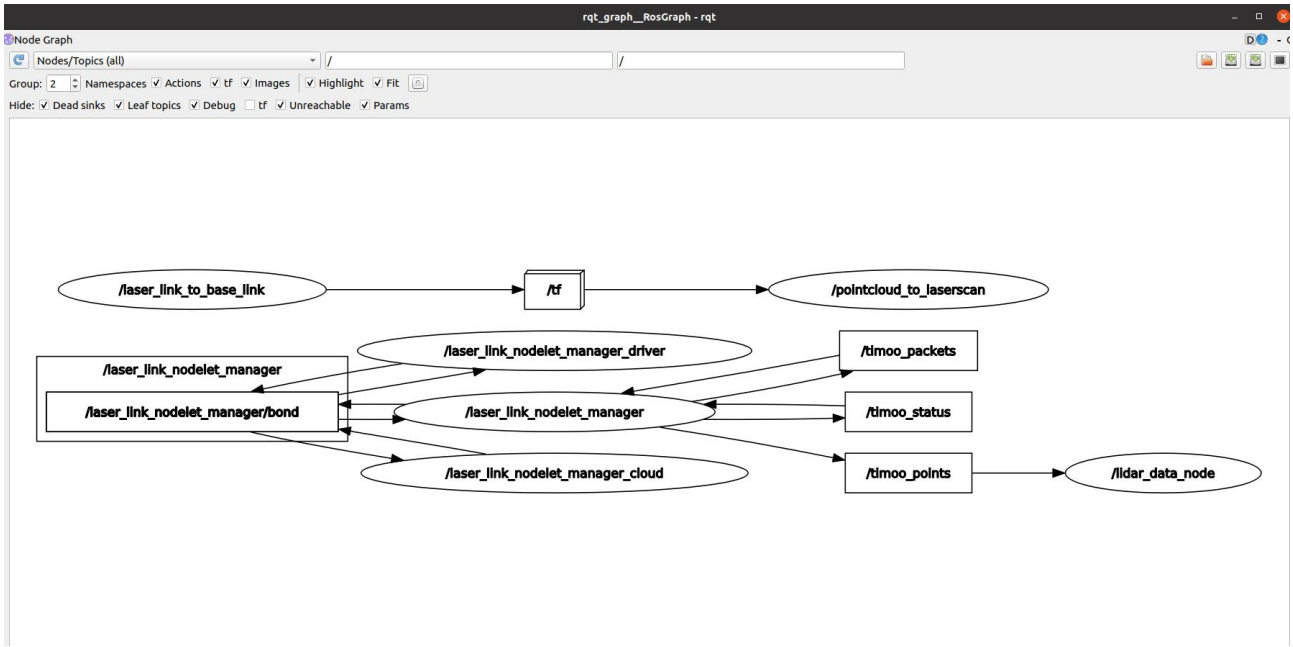


6. While keeping the above terminal program running, we can examine the node network relationships in the ROS system. Start another terminal program and enter the following command:

```
rqt_graph
```



After pressing the Enter key, a new window will pop up displaying a series of connected box diagrams, which represent the current node network relationships in the ROS system.



7. To convert point cloud data into LaserScan data

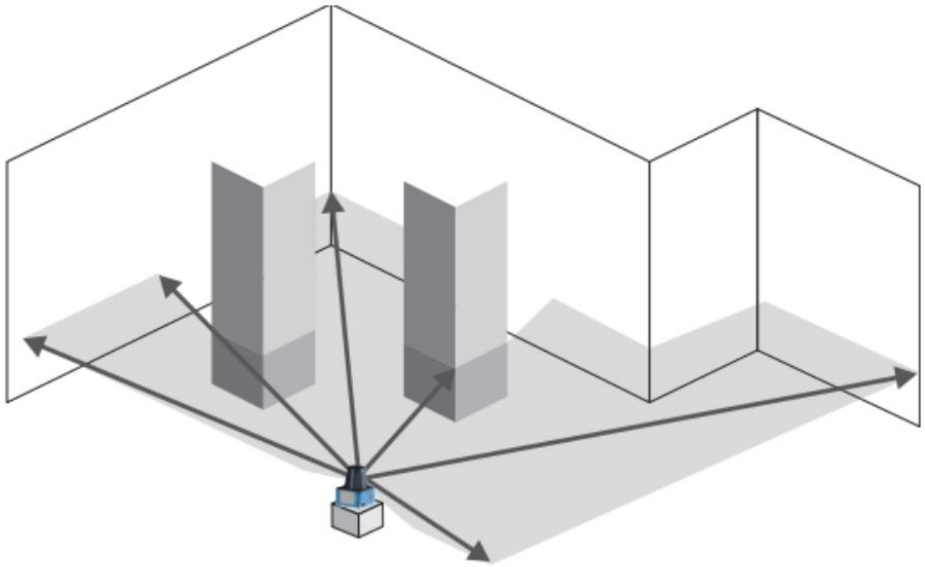
Alongside starting the LiDAR, the "pointcloud_to_laserscan" node is also launched to convert the laser point cloud data into LaserScan data. It publishes the converted data on the topic "scan". The subsequent navigation obstacle avoidance module will subscribe to this topic for further processing.

Experiment 6: SLAM Mapping

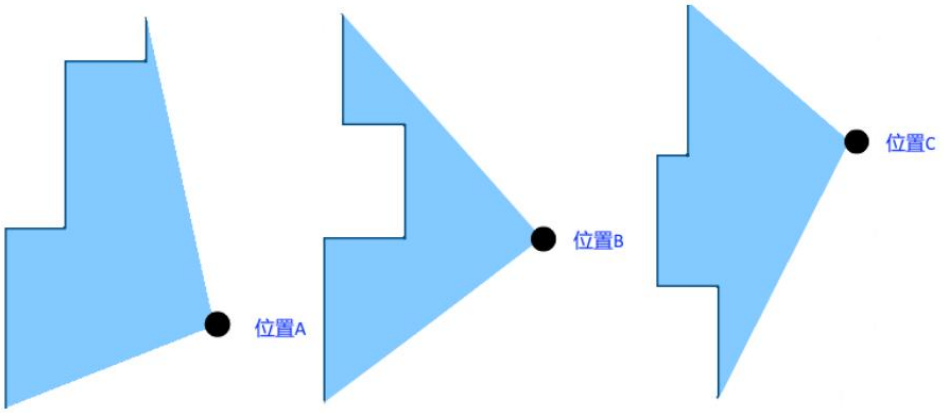
In the following section, we will have a preliminary understanding of mapping through the process of 2D mapping.

Having learned about the data format of LiDAR, we will now delve into the application of LiDAR data. Here, we come across a term called "SLAM," which stands for "Simultaneous Localization And Mapping." SLAM was first proposed by Smith, Self, and Cheeseman in 1988. Due to its significant theoretical and practical value, many researchers consider it a key technology for achieving true autonomy in mobile robotics.

To understand SLAM, it is crucial to grasp the characteristics of LiDAR data. The scan data from a LiDAR can be interpreted as a cross-sectional view of obstacle distribution. It reflects the shape and spatial distribution of the edges of obstacles facing the LiDAR at a specific height.

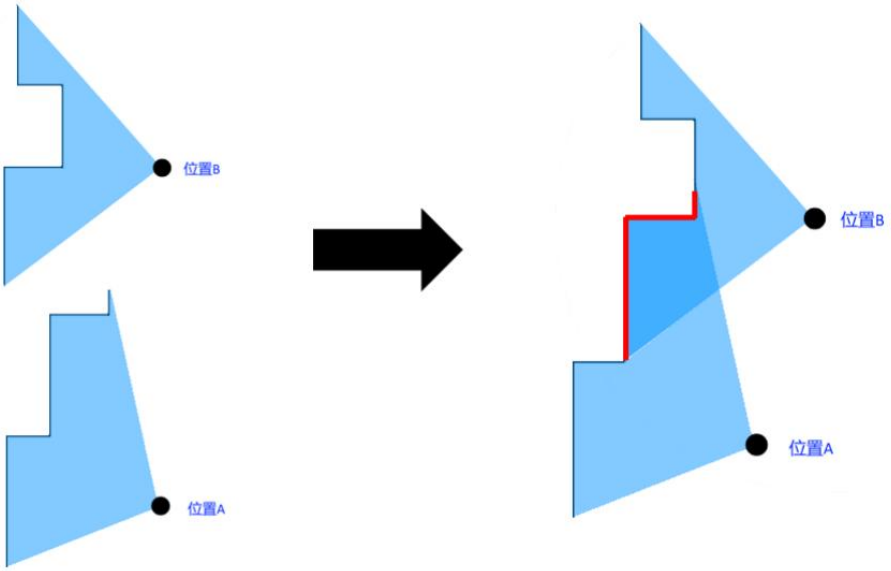


Therefore, when a robot equipped with a LiDAR moves in an environment, at any given moment, it can only perceive partial contours of obstacles within a limited range and their relative positions in the robot's coordinate system. For example, in the following figure, it illustrates the obstacle contours scanned by the LiDAR at three adjacent but relatively close positions, labeled as A, B, and C.

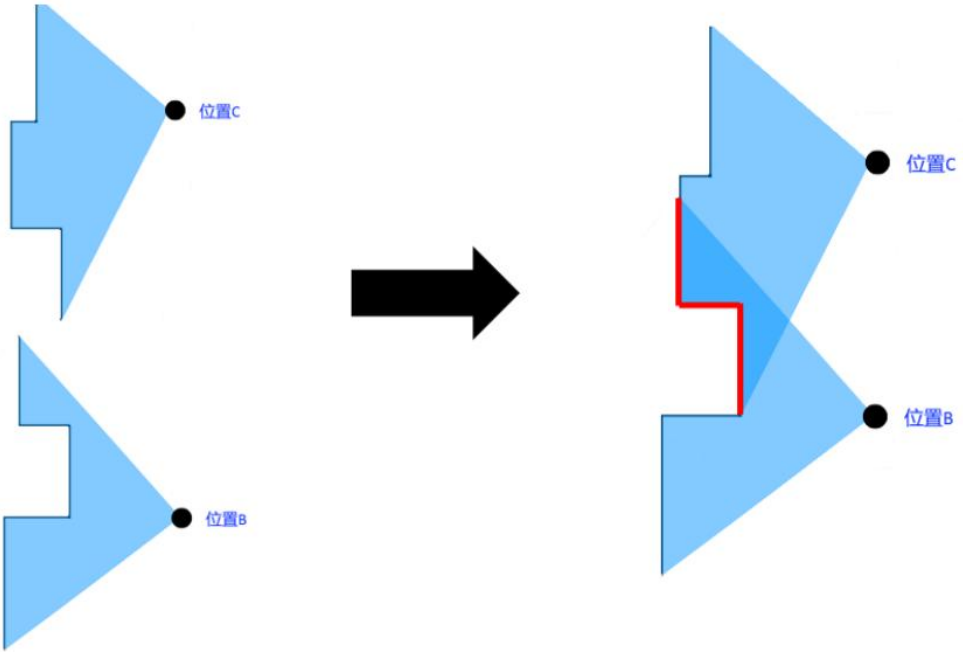


Although we don't know the exact relationship between positions A, B, and C yet, through careful observation, we can notice that some parts of the obstacle contours scanned at positions A, B, and C can be matched and overlapped. Since these three positions are relatively close to each other, we can assume that the similar parts of the scanned obstacle contours belong to the same obstacle. By overlaying and aligning these similar parts, we can obtain a larger pattern of the obstacle contour.

For example, when we overlay the obstacle contours from position A and position B, we get the following result:

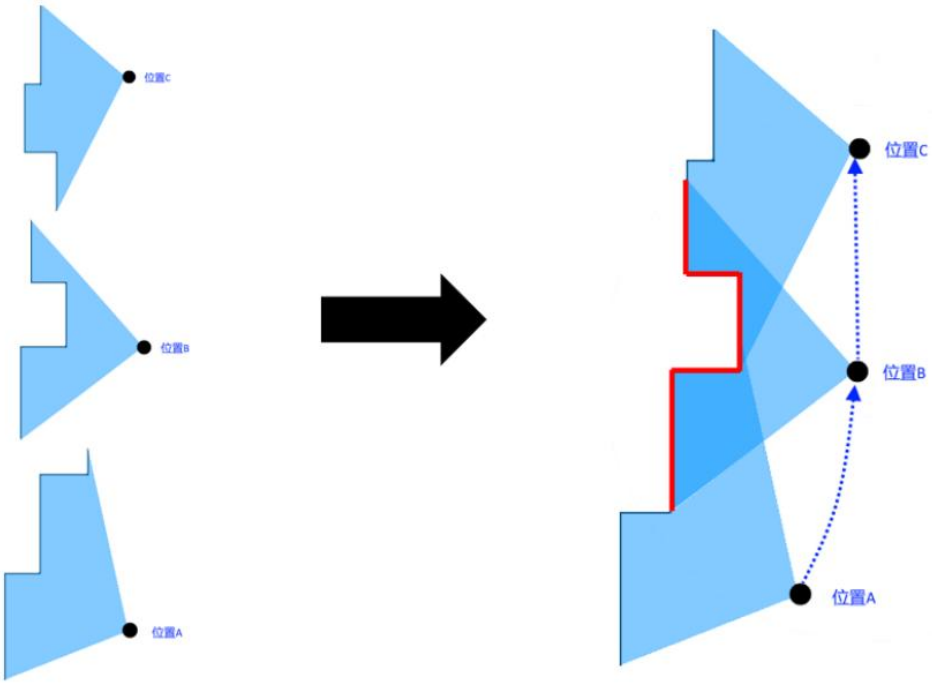


Similarly, when we overlay the obstacle contours from position B and position C, we get the following result:



By following the above method and combining the obstacle contours scanned at consecutive positions of the LiDAR, we can create a relatively complete 2D map. This map is a representation of the obstacle contours and their distribution in the environment, as observed from the LiDAR's scanning plane. During the map building process, we can also infer the relationships between the robot's positions and its location within the map based on the overlapping relationships of the obstacle contours. This simultaneous achievement of map building and real-time self-localization of the robot gives rise to the term "SLAM," which stands for "Simultaneous Localization And Mapping."

Taking the previous example of positions A, B, and C, by combining the LiDAR scan contours from these three positions, we can obtain a relatively complete 2D map and also determine the positions of A, B, and C within this map:



This chassis is equipped with a 3D LiDAR, and for 3D mapping, the lio-sam mapping algorithm is used(https://github.com/TixiaoShan/LIO-SAM). For 2D mapping, you can choose between gmapping or cartographer. Below are detailed instructions for each:

3D Mapping:

1. Recording Data:

(1) Open a terminal and enter the command, then press Enter:

```
roslaunch yhs_can_control yhs_can_control_bringup.launch
```

```
[ INFO] [1652163197.188928146]: start angle and end angle select feature activated.
[ INFO] [1652163197.188992040]: start_angle: 0 end_angle: 360 angle_flag: 1
[ INFO] [1652163197.189038178]: >>open can deivce success!
[ INFO] [1652163197.190131928]: publishing 84 packets per scan
[ INFO] [1652163197.225482712]: Only accepting packets from IP address: 192.168.1.200
[ INFO] [1652163197.225532773]: Opening UDP socket: port 2368
[ INFO] [1652163197.246496027]: Only accepting packets from IP address: 192.168.1.200
[ INFO] [1652163197.246532189]: Opening UDP socket: port 2369
[ INFO] [1652163197.247303149]: distance threshlod, max: 150, min: 0.15
[ INFO] [1652163197.247340948]: return mode : 1
[ INFO] [1652163197.247361777]: vertical angle resolution: 2 degree
[ INFO] [1652163197.257829287]: output frame: odom_combined
[ INFO] [1652163197.257898278]: base frame: base_link
[ INFO] [1652163197.492871951]: Initializing Odom sensor
[ INFO] [1652163197.502880413]: Odom sensor activated
[ INFO] [1652163197.503859697]: Kalman filter initialized with odom measurement
[ INFO] [1652163197.783457209]: Initializing Imu sensor
[ INFO] [1652163197.813408815]: Imu sensor activated
```

If "Imu sensor activated" appears, it indicates successful initialization.

(2) Press "ctrl + shift + t" to open another terminal side by side, enter the command and hit Enter.:

```
rosbag record /imu_data / timoo_points -O test.bag
```

If the ROS car is equipped with a Robosense LiDAR, enter:

```
rosbag record /imu_data /rslidar_points -O test.bag
```

```
yhs@yhs-ros:~$ rosbag record /imu_data /timoo_points -O test.bag
[ INFO] [1699927870.080106309]: Subscribing to /imu_data
[ INFO] [1699927870.083243841]: Subscribing to /timoo_points
[ INFO] [1699927870.086385237]: Recording to test.bag.
```

At this point, the IMU data topic ("/imu_data") and LiDAR data topic ("/timoo_points") will start recording. Specify the bag name after "-O", it can be any name, but if it matches the name of the previous recording, it will overwrite it.

(3) You can now remotely control the robot's movement. **Be careful not to perform in-place rotations or sudden stops. It is best to return to the starting point before stopping the recording. A loop closure check will be performed when returning to the starting point.**

(4) Press "ctrl + c" in the terminal recording the bag file to stop the recording and save the bag file in the current directory. If you want to save the bag file in a different directory, navigate to that directory using "cd" before recording.

(5) After saving the bag file, check if the bag file is valid or contains the desired topics. Open a terminal in the directory where the bag file is saved and enter the command, then press Enter:

```
rosbag info test.bag
```

```
yhs@yhs-ros:~$ rosbag  info   test.bag
path:        test.bag
version:     2.0
duration:    34.8s
start:       Nov 14 2023 10:11:10.34 (1699927870.34)
end:         Nov 14 2023 10:11:45.17 (1699927905.17)
size:        109.1 MB
messages:    3829
compression: none [116/116 chunks]
types:       sensor_msgs/Imu         [6a62c6daae103f4ff57a132d6f95cec2]
             sensor_msgs/PointCloud2 [1158d486dd51d683ce2f1be655c3c181]
topics:      /imu_data       3483 msgs    : sensor_msgs/Imu
             /timoo_points    346 msgs    : sensor_msgs/PointCloud2
```

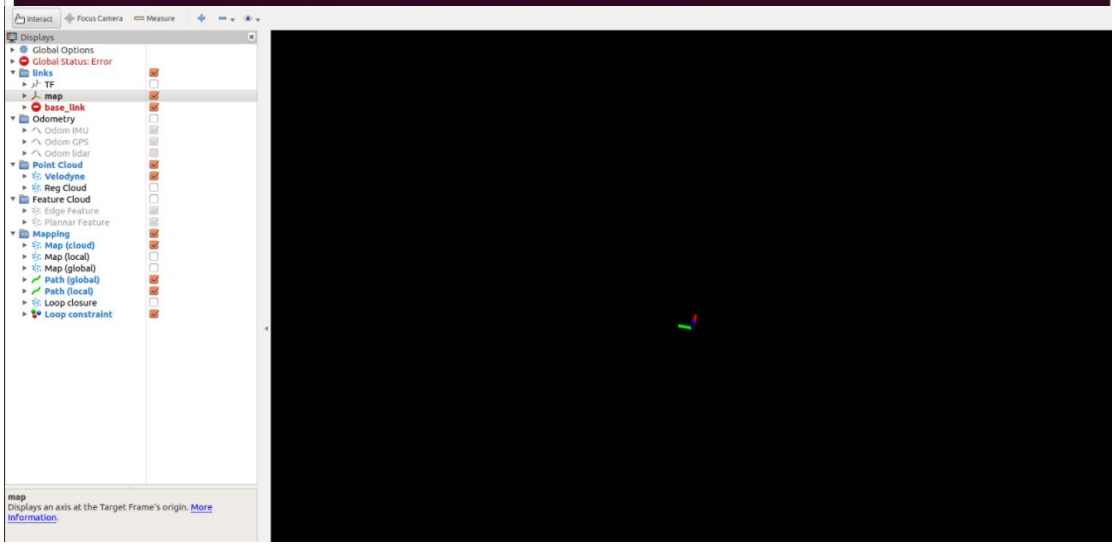
You will be able to see the size of the "test.bag" file and the recorded IMU and LiDAR data. If any data is missing, you will need to re-record. Therefore, it is recommended to record a short bag file of a few seconds, verify its correctness, and then proceed with recording the larger scene.

2. **Mapping**

(1) Close all previously running programs, open a terminal, enter the command, and press Enter:

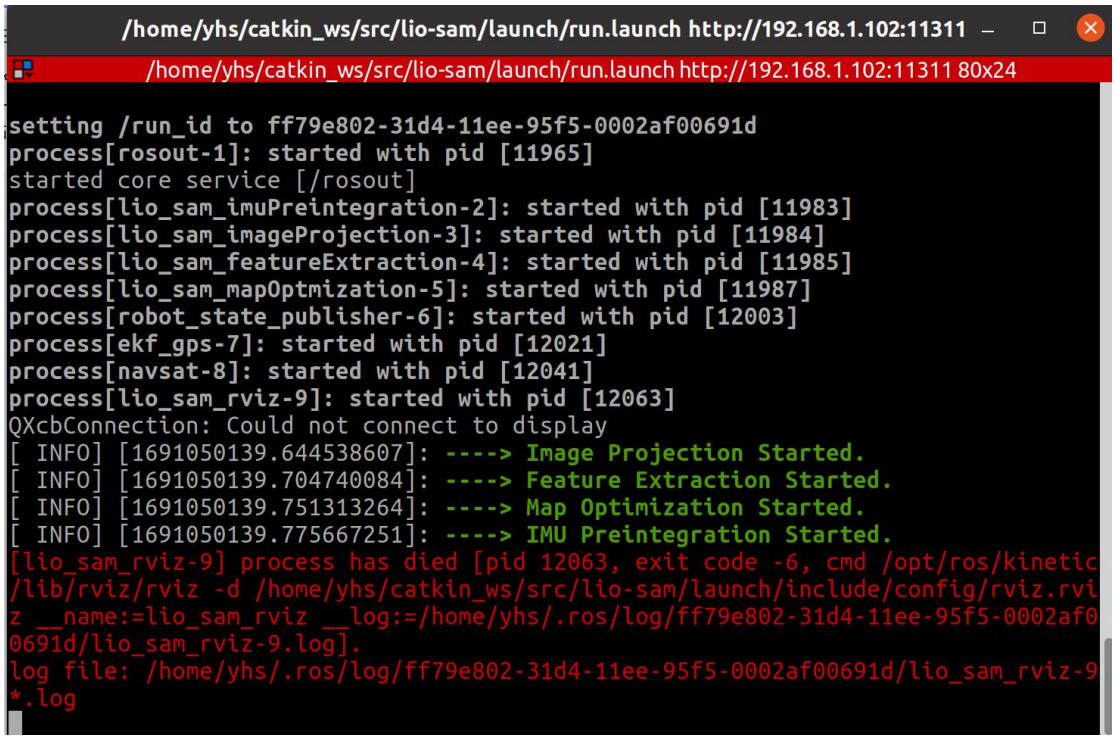
```
roslaunch  lio_sam  run.launch
```

```
setting /run_id to ed7fc994-d02f-11ec-90cf-347df6c3e4cf
process[rosout-1]: started with pid [6591]
started core service [/rosout]
process[lio_sam_imuPreintegration-2]: started with pid [6603]
process[lio_sam_imageProjection-3]: started with pid [6610]
process[lio_sam_featureExtraction-4]: started with pid [6611]
process[lio_sam_mapOptmization-5]: started with pid [6612]
process[robot_state_publisher-6]: started with pid [6623]
process[ekf_gps-7]: started with pid [6640]
process[navsat-8]: started with pid [6656]
process[lio_sam_rviz-9]: started with pid [6683]
[ INFO] [1652166481.758168902]: ----> Image Projection Started.
[ INFO] [1652166481.789191546]: ----> Feature Extraction Started.
[ INFO] [1652166481.813919265]: ----> Map Optimization Started.
[ INFO] [1652166481.846049880]: ----> IMU Preintegration Started.
```



You will see four lines of green text, and Rviz will also be opened.

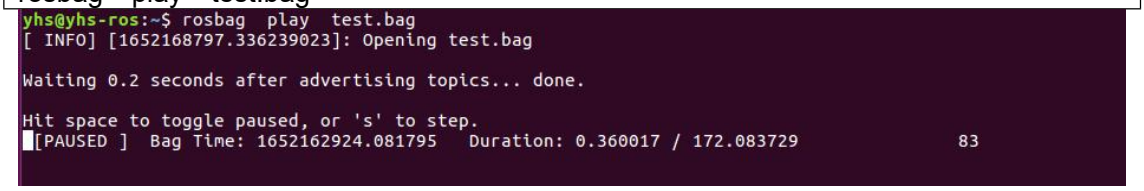
If you are running this mapping program remotely through a laptop, you may encounter a situation where Rviz cannot be opened, as shown in the following image. In that case, simply open Rviz in the terminal of your remote laptop.



This program is responsible for converting the format of the LiDAR data. You can ignore the yellow text messages printed during its execution.

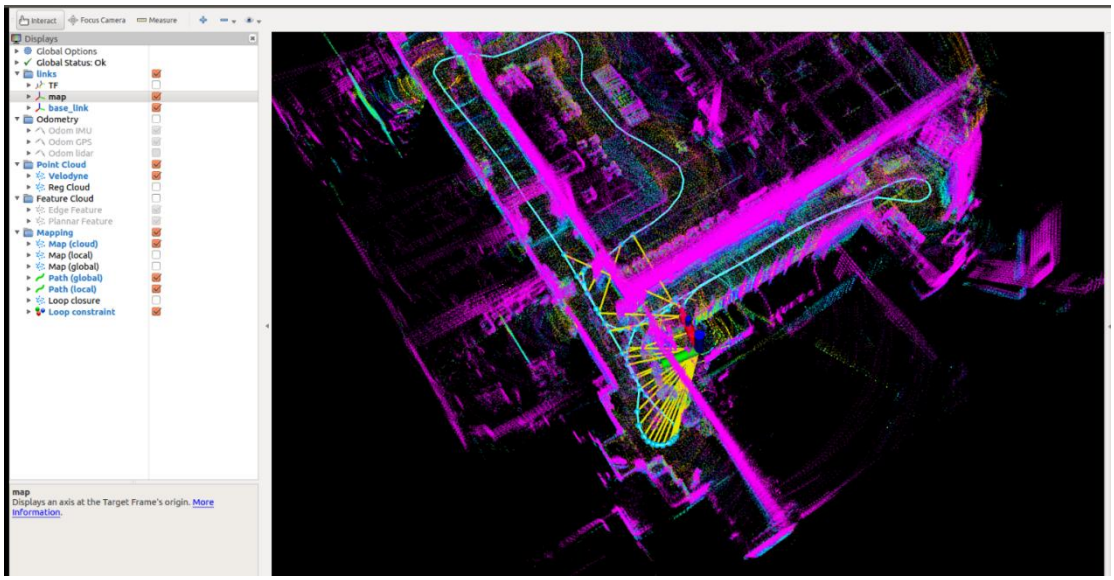
(2) Open another terminal, enter the command, and press Enter.

```
rosbag play test.bag
```



This will start playing the previously recorded bag file. Press the spacebar to pause and resume playback. **Please note that the bag file is located in the home directory, so you can play it by opening a terminal. If it is located in a different directory, you need to use the "cd" command to navigate to that directory before playing the bag file, otherwise it won't be found.**

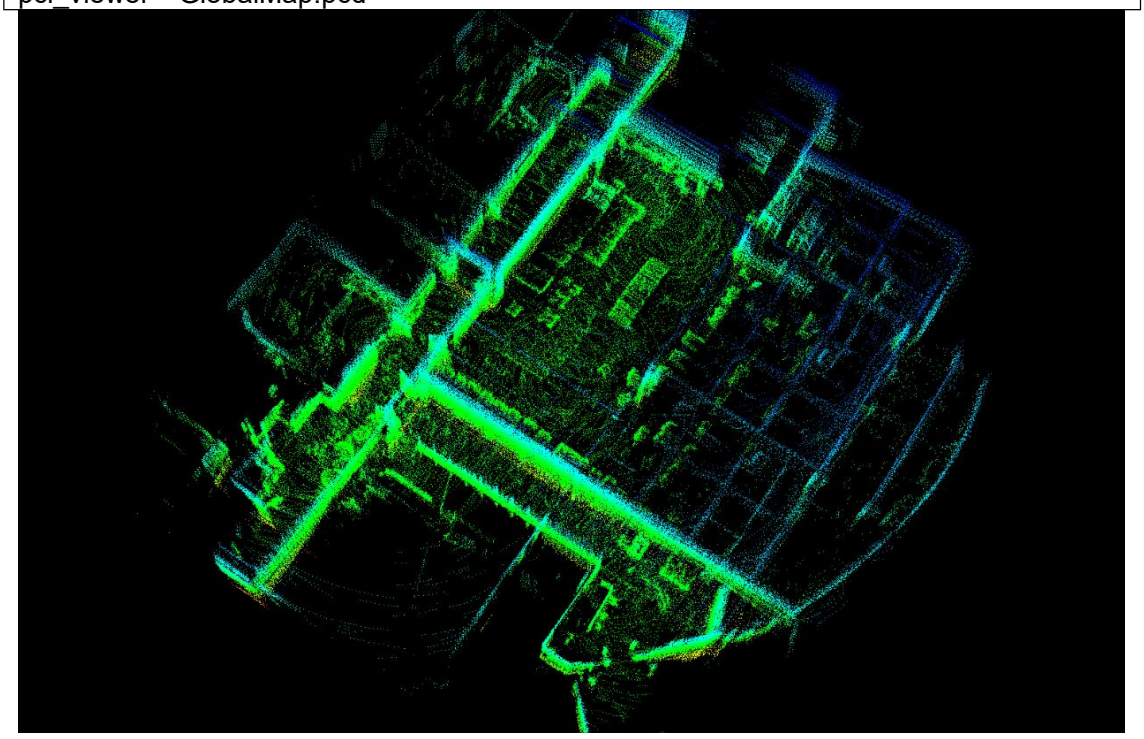
(3) During the mapping process, you can observe the real-time mapping effect in Rviz. If there is significant drift along the Z-axis, close the program and make adjustments to the positive parameters before restarting the mapping process.



After the bag file finishes playing, press "ctrl + c" in the terminal running the lio-sam mapping program. The program will automatically save the point cloud data as PCD files before exiting. The default saving directory is "/home/yhs/Downloads/LOAM". Note that each mapping session will overwrite the previous files, so it's important to make backups.

(4) To view the point cloud map, open a terminal, enter the command, and press Enter.

```
cd  /home/yhs/Downloads/LOAM
pcl_viewer  GlobalMap.pcd
```



You will see the generated point cloud map. In the point cloud map interface, you can input numbers 1 to 4 to change the point cloud's color. Place the mouse cursor over a point in the point cloud and press the "f" key to zoom in. Holding the middle mouse button allows you to drag the point cloud map, and the left mouse button allows you to change the viewing angle. (**Note: You need a screen-equipped vehicle to view this point cloud map.**)

(5) To adjust the parameters, open the "params.yaml" file located in the "/home/yhs/catkin_ws/src/lio-sam/config" directory. Make modifications based on the previously saved

backup. If the generated point cloud map appears normal after saving, there is no need to make further modifications.

Parameter	Modification
savePCDDirectory	Point Cloud Saving Directory
odometrySurfLeafSize	Default: 0.2. If there is drifting in outdoor mapping, it can be changed to 0.4
mappingCornerLeafSize	Default: 0.1. If there is drifting in outdoor mapping, it can be changed to 0.2
mappingSurfLeafSize	Default: 0.2. If there is drifting in outdoor mapping, it can be changed to 0.4
z_tollerance	Default: 1000. If the ground has minimal fluctuations indoors, it can be changed to 0

3. **Generating Grid Map from Point Cloud**

(1) Close all previously running programs, open a terminal, and enter one of the following commands, then press Enter:

```
roslaunch pcl_csf run.launch
```

```
yhs@yhs-ros:~$ roslaunch pcl_csf run.launch
... logging to /home/yhs/.ros/log/fd22339c-d0c9-11ec-b58c-00e26949cec7/roslaunch-yhs-ros-5691.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://192.168.1.102:44515/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES
  /
    pcl_csf_node (pcl_csf/pcl_csf_node)

auto-starting new master
process[master]: started with pid [5701]
ROS_MASTER_URI=http://192.168.1.102:11311

setting /run_id to fd22339c-d0c9-11ec-b58c-00e26949cec7
process[rosout-1]: started with pid [5714]
started core service [/rosout]
process[pcl_csf_node-2]: started with pid [5720]
[0] Configuring terrain...
[0] Configuring cloth...
[0]   - width: 85 height: 76
[0] Rasterizing...
[0] Simulating...
[0]   - post handle...
[ INFO] [1652232654.187794659]: can save map......
```

After running the program, it will load the "SurfMap.pcd" point cloud file located in the "/home/yhs/Downloads/LOAM" directory. Make sure that this file exists. Once the loading is complete, the program will filter out the ground data from the point cloud. The larger the point cloud file, the longer it will take. When you see the output "can save map.....," you can proceed to save the map. Please do not close this program at this stage.

(2) Open another terminal and enter the following commands separately, then press Enter:

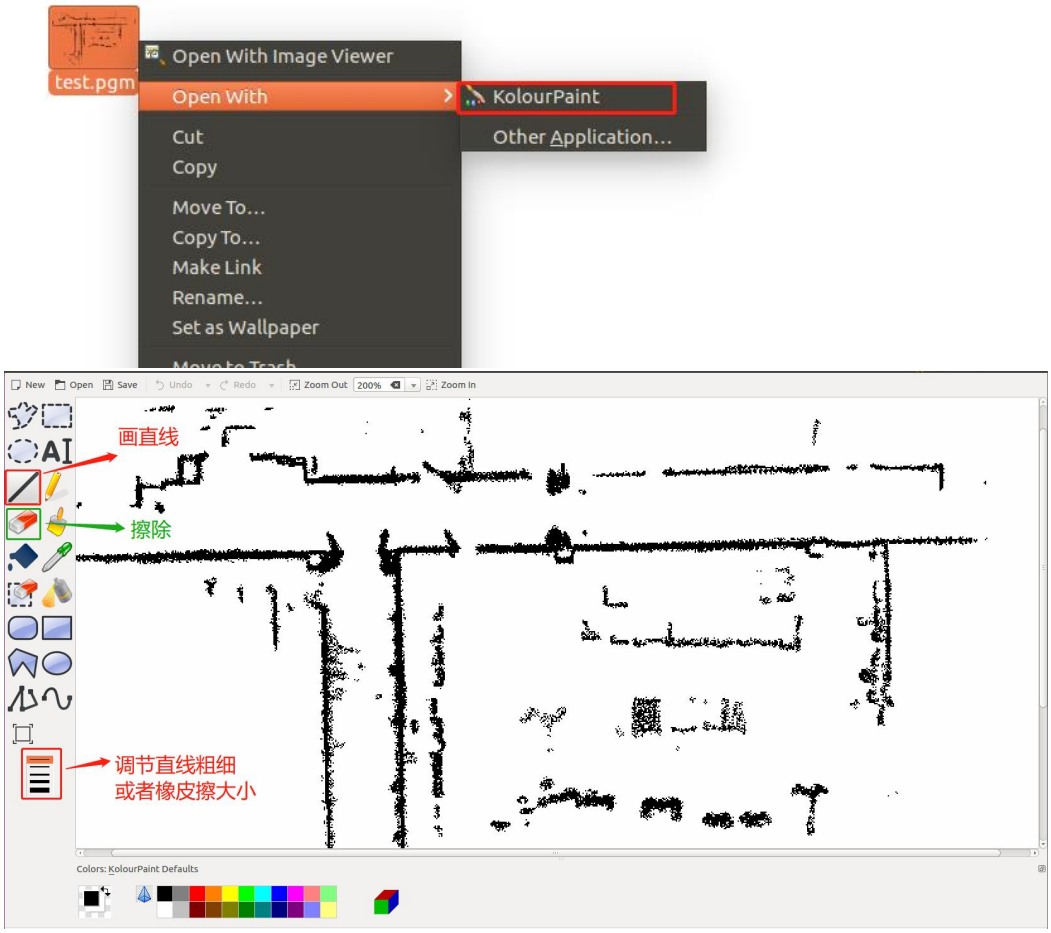
```
roscd yhs_nav/map
```

```
rosrun map_server map_saver -f test
```

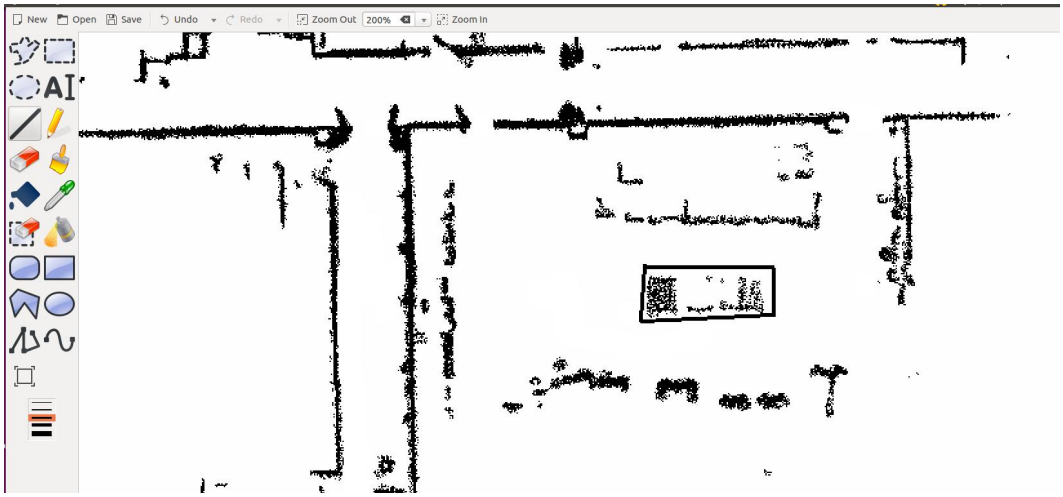
```
yhs@yhs-ros:~/catkin_ws/src/yhs_nav/map$ rosrun map_server map_saver -f test
[ INFO] [1652233363.294073735]: Waiting for the map
[ INFO] [1652233364.188694251]: Received a 810 X 553 map @ 0.050 m/pix
[ INFO] [1652233364.188742812]: Writing map occupancy data to test.pgm
[ INFO] [1652233364.198693538]: Writing map occupancy data to test.yaml
[ INFO] [1652233364.198991652]: Done
```

After the saving process is complete, you will find two new files in the directory "/home/yhs/catkin_ws/src/yhs_nav/map": one named "test.pgm" and the other named "test.yaml". The "test.pgm" file is in image format and can be opened and viewed by double-clicking on it. Please note that if the name of the map saved in the subsequent session is the same as the previous one, it will overwrite the previous map.

(3) Map editing: Sometimes, we may need to set forbidden areas on the map or remove certain obstacles. For this purpose, we can use an image editing software. Locate the image you want to edit, right-click on it, and select "KolourPaint" to open it.



Below are the results of erasing certain static obstacles and adding forbidden areas. After editing, click on "Save" and close the image to complete the process.



3D Mapping with RTK:

If the navigation suite includes RTK, you can use RTK data for mapping.

1. Recording the Bag File:
(1) Open a terminal and enter the following command to start the LiDAR and IMU nodes:

```
roslaunch yhs_can_control yhs_can_control_bringup.launch
```

```
yhs@yhs-ros:~$ roslaunch yhs_can_control yhs_can_control_bringup.launch
... logging to /home/yhs/.ros/log/7b8338dc-5b9c-11ed-bf8d-042b580bf0d1/roslaunch-yhs-ros-9639.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://192.168.1.102:39093/
```

(2)Open another terminal and enter the following command

```
roslaunch nmea_navsat_driver nmea_serial_driver.launch
```

```
yhs@yhs-ros:~$ roslaunch nmea_navsat_driver nmea_serial_driver.launch
... logging to /home/yhs/.ros/log/d81b8aea-5b9c-11ed-bf8d-042b580bf0d1/roslaunch-yhs-ros-10410.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://192.168.1.102:40081/
```

Start the Sino integrated navigation node. Then, check the /fix topic data by entering:

```
rostopic echo /fix
```

```
header:
  seq: 16
  stamp:
    secs: 1667496757
    nsecs: 606498956
  frame_id: "navsat_link"
status:
  status: -1
  service: 1
latitude: nan
longitude: nan
altitude: nan
position_covariance: [nan, 0.0, 0.0, 0.0, nan, 0.0, 0.0, 0.0, nan]
position_covariance_type: 1
---
```

Make sure that the status is 2 before starting the recording. If the status remains non-2 for a long time outdoors, check if the RTK account is valid or if there are any environmental factors affecting it.

(3) Configure the package file for robot_localization. Open the file by entering:

```
vi /home/yhs/catkin_ws/src/robot_localization/params/navsat_transform_template.yaml
```

These two parameters correspond to latitude and longitude. Enter the following command in the terminal: rostopic echo /fix

```
---
header:
  seq: 108
  stamp:
    secs: 1667490671
    nsecs: 822699069
  frame_id: "navsat_link"
status:
  status: 2
  service: 1
latitude: 22.6903566683
longitude: 114.301032572
altitude: 56.2624
position_covariance: [0.25, 0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 1.0]
position_covariance_type: 1
---
```

Note: The latitude and longitude values to be filled in should correspond to the coordinates of the starting point (the location where the recording of the bag file was initiated).

(4) Open another terminal and enter the following command:

```
rosbag record /imu_data /odom /fix / timoo_points -O bagname.bag
```

If you are using the Robosense LiDAR, use the following command instead:

```
rosbag record /imu_data /odom /fix /rslidar_points -O bagname.bag
```

```
yhs@yhs-ros:~$ rosbag record /imu_data /odom /fix /timoo_points -O bagname.bag
[ INFO] [1699928057.550255878]: Subscribing to /fix
[ INFO] [1699928057.553472142]: Subscribing to /imu_data
[ INFO] [1699928057.556244793]: Subscribing to /odom
[ INFO] [1699928057.558978217]: Subscribing to /timoo_points
[ INFO] [1699928057.561931271]: Recording to bagname.bag.
```

At this point, the data recording will start. Control the ROS-enabled vehicle to perform the mapping scan. To stop and save the recording, press "ctrl + c" in the terminal running the rosbag command.

2. Mapping

(1) Close all previously running programs, open a terminal, and enter the following command, then press Enter:

```
roslaunch lio_sam run.launch
```

(2) Open another terminal, enter the following command, and press Enter:

```
rosbag play bagname.bag
```

3. Generating Grid Map from Point Cloud:

(1) Open a terminal and enter one of the following commands, then press Enter:

```
roslaunch pcl_csf run.launch
```

When you see the output "can save map....," it means you can save the map. Please do not close this program for now.

(2) Open another terminal and enter the following commands separately, then press Enter:

```
roscd yhs_nav/map
rosrun map_server map_saver -f test
```

After the saving process is complete, you will find two new files in the directory "/home/yhs/catkin_ws/src/yhs_nav/map": one named "test.pgm" and the other named "test.yaml". The "test.pgm" file is in image format and can be opened and viewed by double-clicking on it. Please note that if you use the same name for saving the map in subsequent sessions, it will overwrite the previous map.

2D Mapping:

1.2D Mapping Method 1: Gmapping Mapping

(1) Open a terminal, enter the following command, and press Enter:

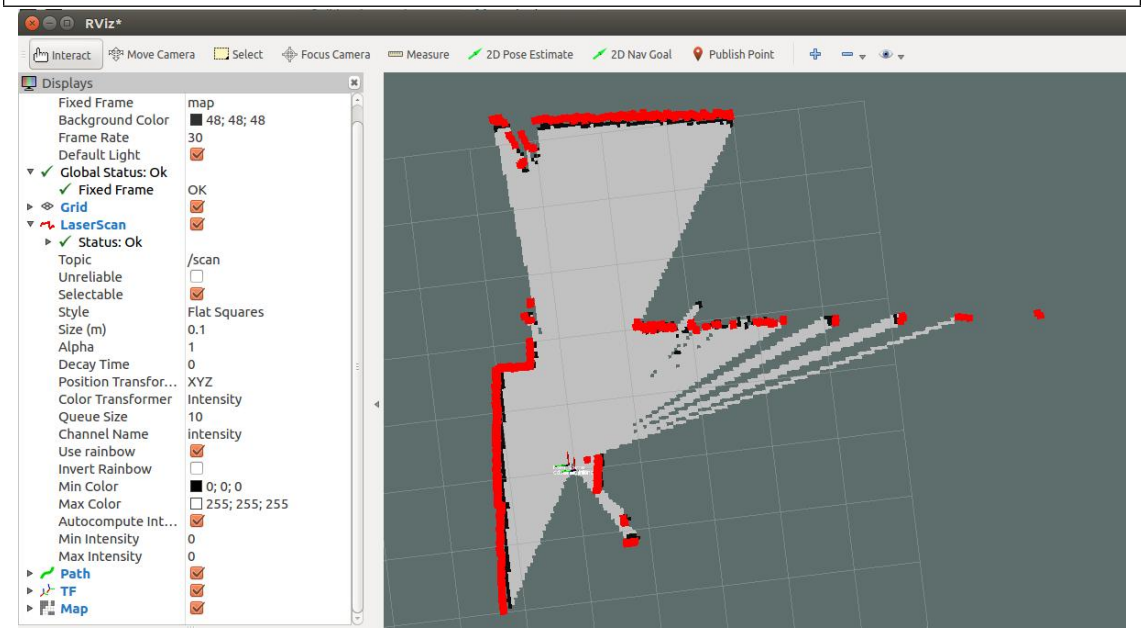
```
roslaunch yhs_nav gmapping.launch
```

```
update frame 0
update ld=0 ad=0
Laser Pose= 0.0999897 1.44477e-05 -2.39423e-05
m_count 0
Registering First Scan
update frame 2280
update ld=0.0500084 ad=0.0600143
Laser Pose= 0.0998256 -0.000277561 -0.00181417
m_count 1
Average Scan Matching Score=583.609
neff= 8
Registering Scans:Done
```

If you see the message "Registering Scans: Done," it indicates that Gmapping has been successfully launched.

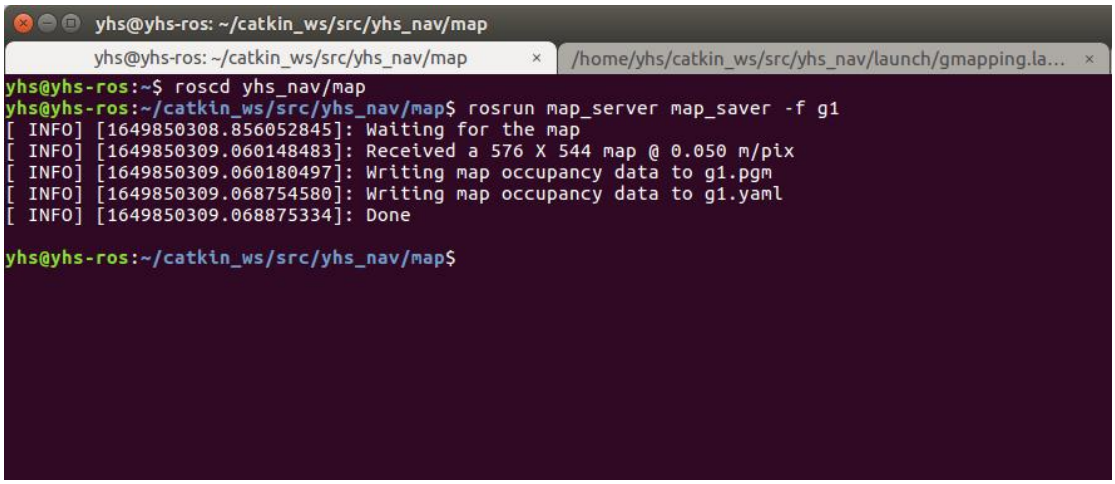
(2) Open another terminal, enter the following command, and press Enter:

```
rviz
```



(3) After controlling the robot to complete a full circle around the environment, you can save the map. Please note that Gmapping mapping is more suitable for small-scale mapping. Large-scale mapping can consume significant memory resources. **During the robot's scanning process, avoid rotating too quickly to prevent map drift.** When saving the map, make sure the Gmapping program is still running and do not close the Rviz interface displaying the generated map. To save the map, return to the terminal where Gmapping was launched and enter the following command:

```
roscd yhs_nav/map
rosrun map_server map_saver -f g1
```

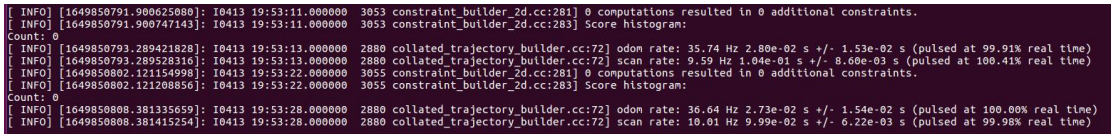


After saving, you will find two new files in the directory "/home/yhs/catkin_ws/src/yhs_nav/map": one named "g1.pgm" and the other named "g1.yaml". The "g1.pgm" file is in image format and can be opened by double-clicking to view the map that has been generated. If you need to make modifications to the map, you can use a drawing tool to open and edit it. Please note that if you use the same name for saving the map in subsequent sessions, it will overwrite the previous map.

2.2D Mapping Method 2: Cartographer Mapping

(1) Open a terminal, enter the following command, and press Enter:

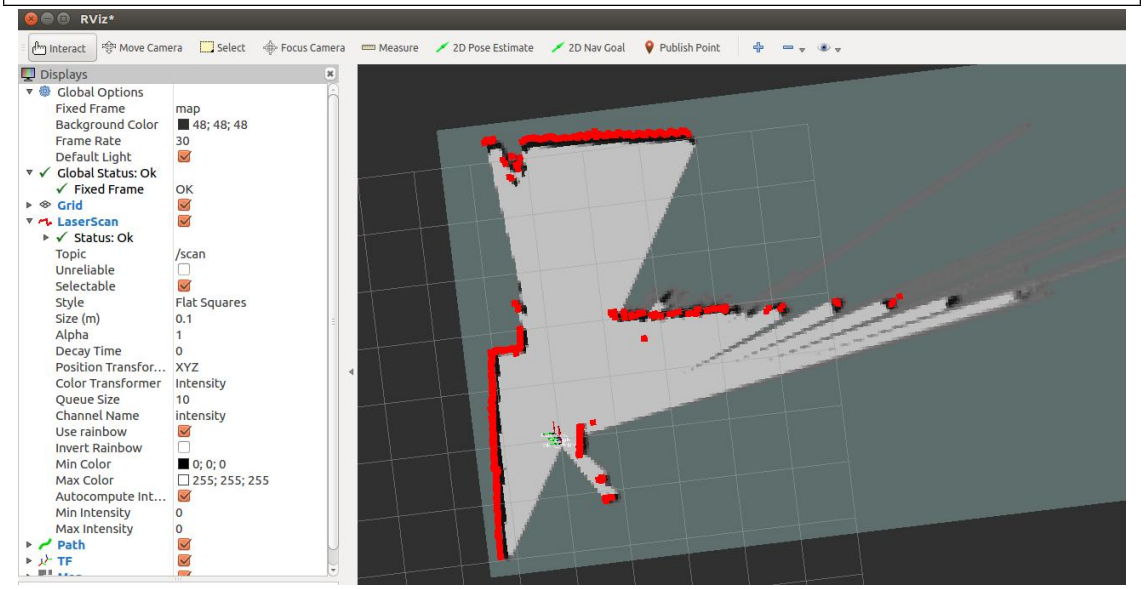
```
roslaunch yhs_nav cartographer.launch
```



When you see the odometry and laser frequency being printed, it indicates that Cartographer mapping has been successfully launched.

(2) Open another terminal, enter the following command, and press Enter:

```
rviz
```

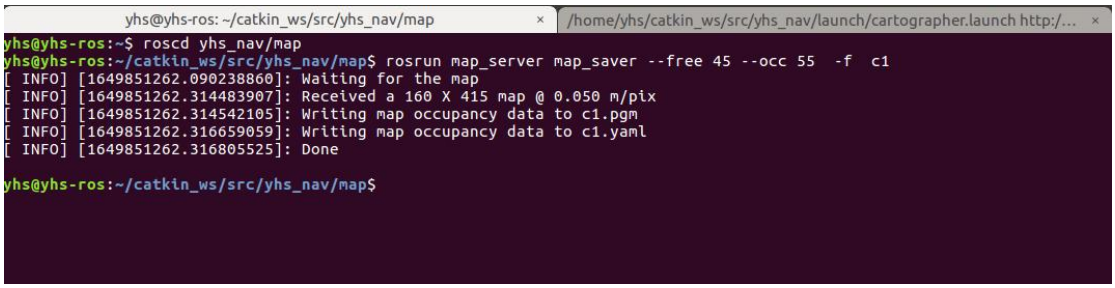


Rviz will open, and you will see the ground around the robot turning into a dark gray color, while a white pattern appears beneath the robot's feet. This pattern is formed by multiple line segments, representing the projections of the robot's center on the ground and the lines connecting each red obstacle point detected by the laser. These line segments represent the trajectory of the laser measurements, indicating that there are no obstacles within these segments.

(3) After controlling the robot to complete a full circle around the environment, you can save the map. **During the scanning process, ensure that the rotation speed of the robot is not too fast to avoid map drift. Ideally, it is recommended to return the robot to its initial starting point to form a loop at the end of mapping. If there are still some areas with drift after returning to the starting point, you can drive the robot to those areas and wait for the loop to close. If the drift is too severe, it may not be possible to close the loop, and it is advisable to restart the mapping process.** When saving the map, ensure that the Cartographer program is still running and do not

close the Rviz interface displaying the generated map. Return to the terminal where Cartographer was launched and enter the following command:

roscd yhs_nav/map
rosrun map_server map_saver --free 45 --occ 55 -f c1



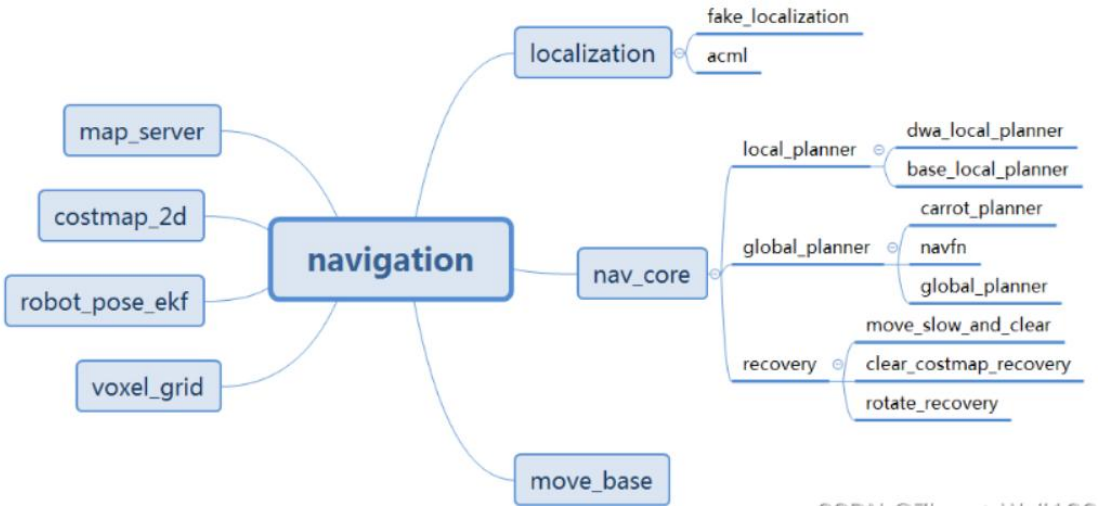
After saving, you will find two new files in the directory "/home/yhs/catkin_ws/src/yhs_nav/map": one named "c1.pgm" and the other named "c1.yaml". The "c1.pgm" file is in image format and can be opened by double-clicking to view the generated map. Please note that if you use the same name for saving the map in subsequent sessions, it will overwrite the previous map. Once the map is saved, you can close all the previous mapping programs by pressing Ctrl+C to terminate the programs.

Experiment 7: Navigation Framework

Before proceeding with robot navigation using the generated map, let's first understand the principles of robot navigation. The navigation module is primarily responsible for path planning, which is a relatively complex module as it requires interaction with the localization, mapping, and control modules. Therefore, it involves multiple modules.

Navigation provides a framework, which can be found at this link:(https://github.com/ros-planning/navigation) Let's explore the structure of this framework.

1.The functionality packages within Navigation are as follows:



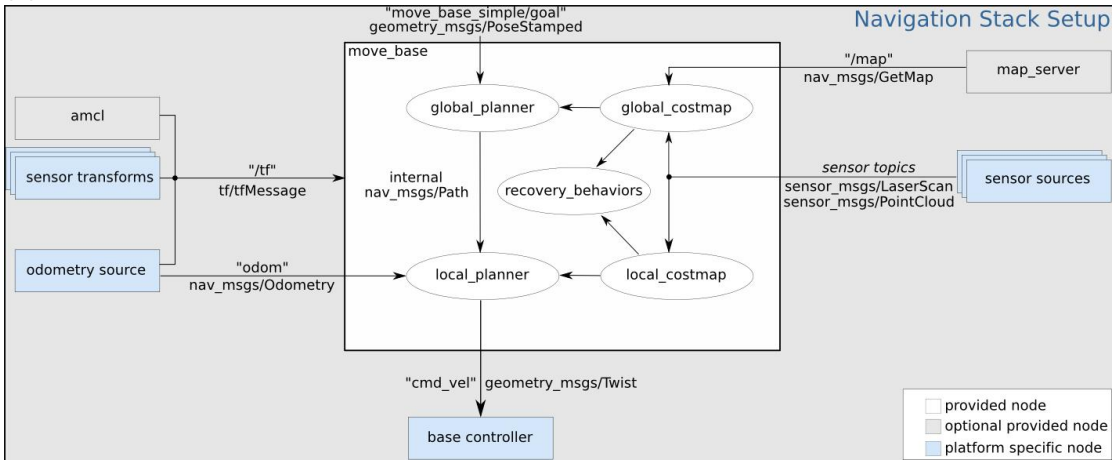
Each package's function indicates as per below:

Packages	Function
amcl	amcl: Corrects the robot's position within a known map using particle filtering based on the robot's odometry and map features.
base_local_planner	Local Path Planner.
dwa_local_planner	Local Path Planner: Utilizes the Dynamic Window Approach for local path planning.
carrot_planner	Global Path Planner: Generates a path from the current position of the robot to the target point along a straight line.
clear_costmap_recovery	Recovery Behaviors for Unplanned Paths: Implements algorithms for recovering from situations where a path cannot be planned.
costmap_2d	Costmap: Implements cost maps for navigation purposes.
fake_localization	Localization Simulation: Primarily used for localization simulation.
global_planner	Global Path Planning Algorithms: Provides various algorithms for global path planning.

map_server	Map Server: Offers map management services.
move_base	Robot Movement Navigation Framework: The main logical framework for navigation.
move_slow_and_clear	Recovery Strategies: Implements a recovery strategy.
nav_core	Main Package for Algorithm Replacement through Interfaces: Provides interfaces for algorithm replacement.
nav_fn	Global Path Planning Algorithms: Various algorithms for global path planning.
robot_pose_ekf	Integrated Odometry, GPS, and IMU-based Position Estimation using Extended Kalman Filtering.
rotate_recovery	Rotation Recovery Strategy Implementation Package.
voxel_grid	Three-Dimensional Cost Maps.

2.Connecting the functionality packages within Navigation is achieved through the "move_base" package.

(1)The framework is as follows:



(2)Based on the framework diagram above, let's take a look at the data that needs to be provided and the hardware sensors that are required.

map_server	Pre-built Map: The previously generated map using Cartographer will be utilized.
odometry source	Fused Odometry with IMU: The robot_pose_ekf package provides the functionality to fuse odometry data with IMU measurements, resulting in more accurate robot localization.
Sensor sources	Laser Range Finder: The use of a laser rangefinder, such as a LiDAR sensor, is essential for accurate obstacle detection and mapping. Additionally, camera data can be optionally incorporated into the navigation system for visual perception tasks.

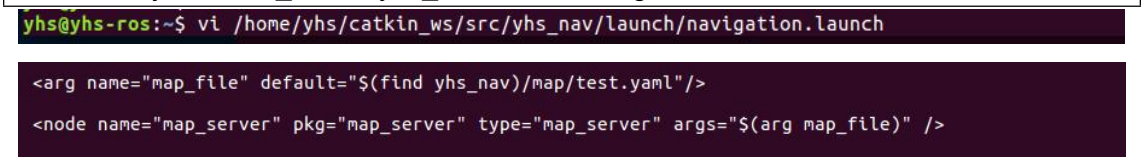
Once all the data is prepared, you can proceed with navigation testing.

Experiment 8: Point-Based Autonomous Navigation in Rviz

Once the map is created, we can proceed with autonomous navigation on this map.
Note: For 3D navigation, use a 3D map, and for 2D navigation, use a 2D map.

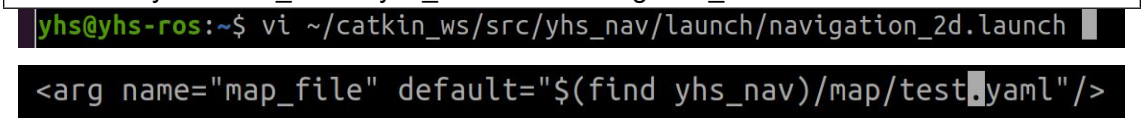
1.Selecting and Switching Maps
(1) If you have created a new map at a different location and saved it, and now you want to use this map for navigation, you need to make modifications in the launch file. If you have used a 3D mapping method, open a terminal and enter the following command:

```
vi /home/yhs/catkin_ws/src/yhs_nav/launch/navigation.launch
```



If you have used a 2D mapping method, open a terminal and enter the following command:

```
vi /home/yhs/catkin_ws/src/yhs_nav/launch/navigation_2d.launch
```



"test" is the configuration file generated after saving the new map, and it has the same name as the map. If you want to switch the navigation map, you need to modify this file. To do so, follow these steps:

a. Enter the terminal and execute the command to open the configuration file.
b. Once inside the file, press the "i" key on the keyboard. You will see "INSERT" appear in the bottom left corner of the terminal. Now you can use the arrow keys (↑↓←→) to move the cursor and make the necessary modifications.
c. After completing the modifications, press the "Esc" key.
d. While holding down the "Shift" key, press ":". This will bring up a colon (:) at the bottom left corner of the terminal.
e. Type "wq" (without quotes) after the colon and press Enter to save the changes.

(Note: When using a 3D map for navigation, make sure that the 2D grid map "test.yaml" in the navigation.launch file corresponds to the point cloud map located in /home/yhs/Downloads/LOAM. Failure to do so may result in the robot being unable to localize properly.)

2.Navigation
①3D Navigation: Navigation using a 3D map:

(1) Start NDT localization by opening a terminal and entering the following command:

```
roslaunch ndt_localizer ndt _localizer.launch
```

```
yhs@yhs-ros:~$ roslaunch ndt_localizer ndt_localizer.launch
```

(2) Start navigation by opening another terminal and entering the following command:

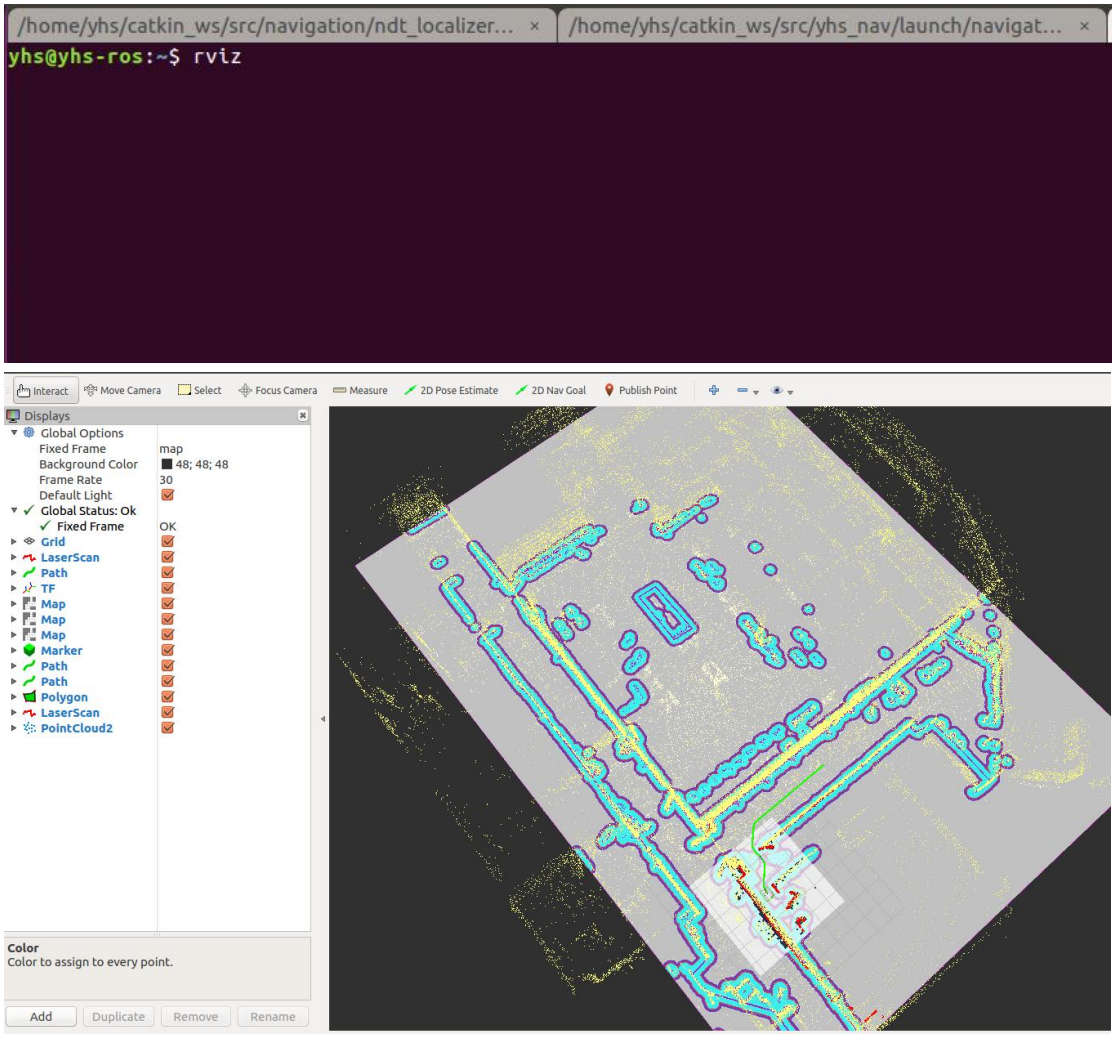
```
roslaunch yhs_nav navigation.launch
```

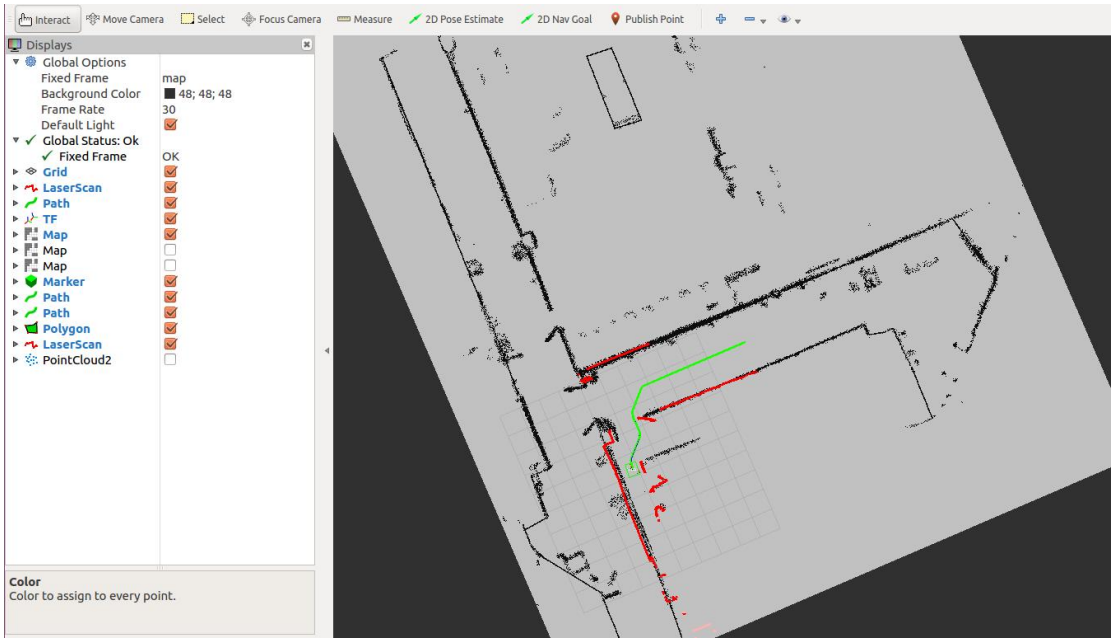
```
yhs@yhs-ros:~$ roslaunch yhs_nav navigation.launch
[ INFO] [1649907161.301759121]: Recovery behavior will clear layer obstacles
[ INFO] [1649907161.317118598]: Recovery behavior will clear layer obstacles
[ INFO] [1649907161.372144141]: odom received!
```

When you see the message "odom received!" printed in the output, it indicates that there are no errors, and the navigation has been successfully initiated.

(3) Open another new terminal and enter the following command:

```
rviz
```





②RTK 3D Navigation: Navigation using a 3D map with RTK data:

When starting the navigation, initialize using GPS data:

(1) Start NDT localization

```
roslaunch ndt_localizer ndt_localizer.launch
```

(2) Start navigation

```
roslaunch yhs_nav navigation.launch
```

(3) Start robot localization node

```
roslaunch robot_localization navsat_transform_template.launch
```

(4) Open RViz

```
rviz
```

After starting the robot_localization node, it will continuously publish the transformed GPS odometry coordinates as /geometry/gps. The ndt_localizer node subscribes to the coordinates published by robot_localization. Since the /fix topic data does not include the heading angle, the published coordinates do not have direction information. In the ndt_localizer node program, an incremental direction of 45 degrees is used for initialization. The key to successful initialization is determining if the localization is accurate. If there is a misjudgment of successful localization, manual initialization is required in RViz.

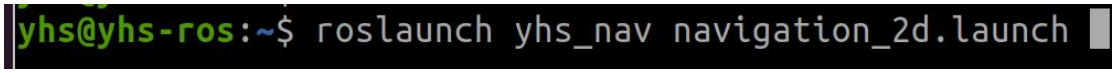
Note: RTK (Real-Time Kinematic) localization can be affected by environmental

conditions. If initialization is not possible, manual initialization in RViz is necessary.

③2D Navigation: Navigation using a 2D map:

(1) Start navigation by opening a terminal and entering the following command:

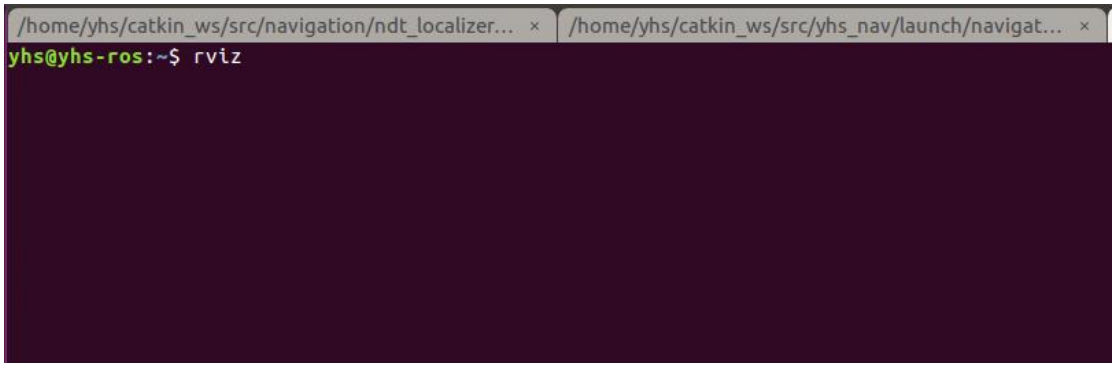
```
roslaunch yhs_nav navigation_2d.launch
```



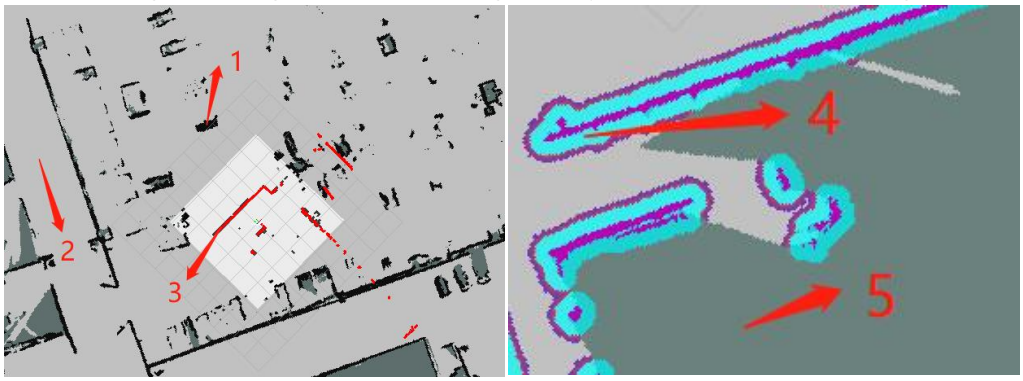
When you see the message "odom received!" printed in the output, it means that there are no errors, and the navigation has been successfully initiated.

(2) Open another new terminal and enter the following command:

```
rviz
```



After starting the navigation and opening RViz, you will enter the following interface:



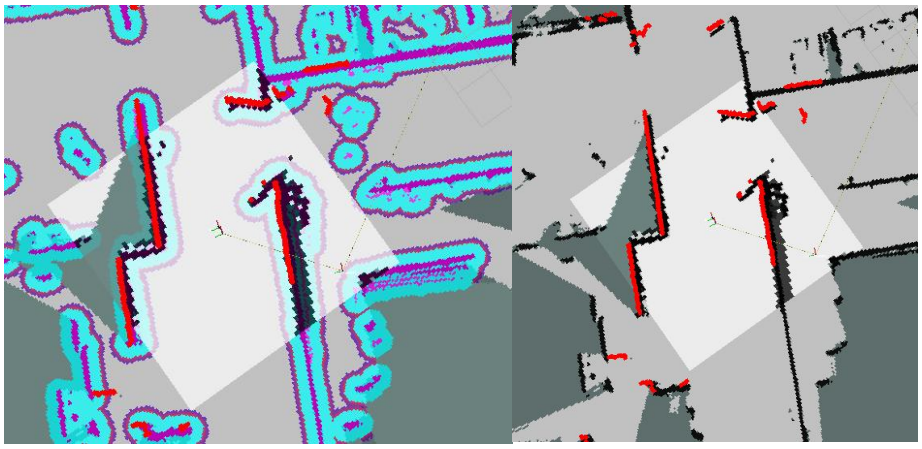
As you can see, the constructed map has been loaded into RViz. Now let's analyze the meaning of various colors represented in RViz:

1	Black areas represent static obstacles.
2	Light gray areas indicate the traversable regions.
3	Red color represents laser data.
4	Blue areas represent the inflated or expanded regions.
5	Dark gray denotes unknown areas.

(3) At this point, if you notice that the position of the robot on the map does not match its actual position, you need to provide an approximate initial position for successful

localization. Click on the "2D Pose Estimate" button in the toolbar of RViz. Then, move the mouse cursor to the desired location on the map, which will be indicated by a green arrow.

According to the actual position of the robot, locate an approximate point on the RViz map. Then, while holding down the left mouse button, drag the arrow in the desired direction. The direction of the arrow represents the orientation of the robot. Release the left mouse button, and the robot will be repositioned on the map. Observe if the repositioned point and orientation provided by us differ significantly from the actual position. If there is a significant difference, repeat the process until the position and orientation of the robot on the map closely match the actual position. Finally, control the robot remotely and align the red laser with the black static obstacles. In the following two images, the localization is successful.



(4) Navigating with waypoints in RViz:

Step 1: Enter the desired number of navigation waypoints in the "Max Goal Count" input field.

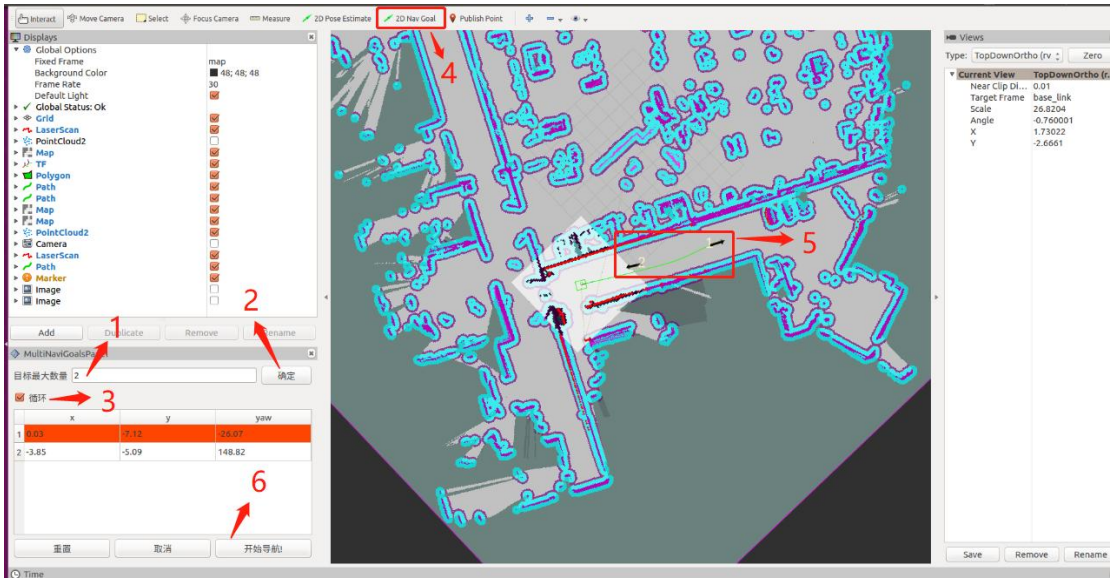
Step 2: Click "OK" to confirm.

Step 3: Choose whether to enable waypoint looping.

Step 4: Select "2D Nav Goal" in the toolbar.

Step 5: Within the accessible area of the map, click and hold the left mouse button to set the navigation waypoints.

Step 6: Click "Start Navigation" to begin the navigation process.



Now, switch the remote control to automatic mode, and the robot will follow the planned path to reach the target point.

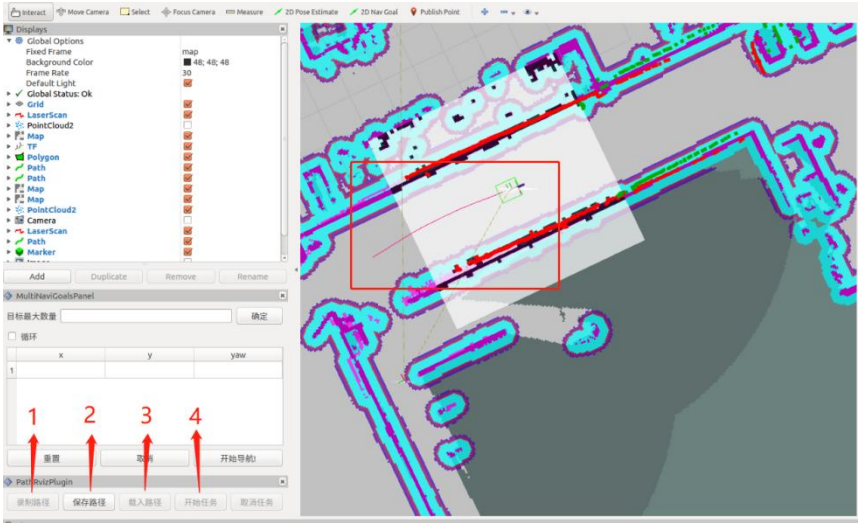
(5) Recording a path in RViz:

Step 1: Click on "Record Path" and control the robot remotely. The path taken by the robot will be displayed in real-time within the red box.

Step 2: After recording is complete, click "Save Path."

Step 3: Click "Load Path."

Step 4: Click "Start Mission" to switch the remote control to automatic mode.

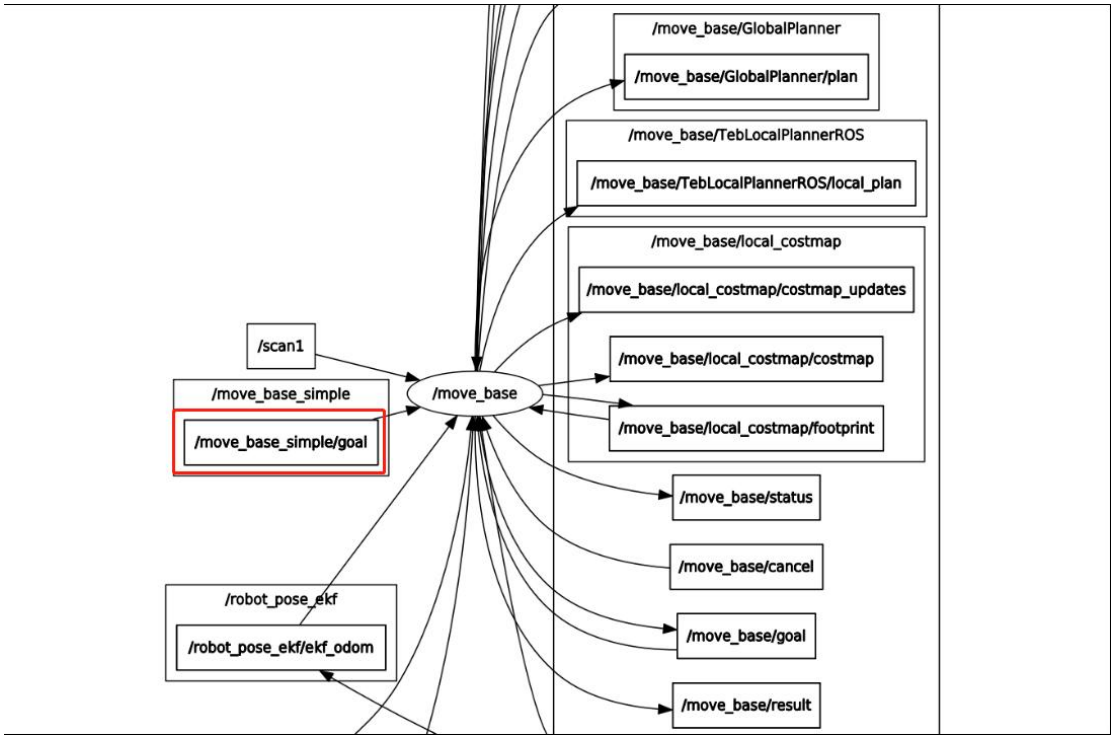


Now, the robot will return to the starting point of the recorded path and follow the recorded path to reach the endpoint. It will continue this cyclic motion, and you can cancel the

mission midway. Each time you record a new path, it will overwrite the previously recorded path.

Experiment 9: Programming Autonomous Navigation Nodes

In the previous section, we used graphical operations in RViz to navigate the robot. In this section, we will write code to accomplish the same task. From the previous two sections, we learned that the "move_base" node connects the functionalities of the navigation package. Therefore, let's focus on the "move_base" node and check if it subscribes to any topics related to navigation waypoints. Please refer to the following diagram:



Intuitively, the term "goal" implies a target point. Let's examine the information about the "/move_base_simple/goal" topic to gain more insights:

```
yhs@yhs-ros:~$ rostopic info /move_base_simple/goal
Type: geometry_msgs/PoseStamped

Publishers:
 * /rviz_1649920486324058429 (http://192.168.1.10:37569/)

Subscribers:
 * /move_base (http://192.168.1.102:35007/)
```

From here, we can reasonably conclude that this is the topic for sending navigation waypoints. Before sending navigation waypoints from RViz, you can print the information from this topic. After sending the navigation waypoints, you should see information being outputted here.

```
yhs@yhs-ros:~$ rostopic echo /move_base_simple/goal
header:
  seq: 1
  stamp:
    secs: 0
    nsecs:         0
  frame_id: "map"
pose:
  position:
    x: -3.85247850418
    y: -5.09073925018
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.963029816592
    w: 0.269394826146
---
```

1. Creating a ROS package.

(1)Open a terminal in Ubuntu and enter the following command to navigate to your ROS workspace:

```
cd   /home/yhs/catkin_ws/src
```

```
yhs@yhs-ros:~$ cd   /home/yhs/catkin_ws/src
yhs@yhs-ros:~/catkin_ws/src$
```

(2)After pressing Enter, you will enter the ROS workspace. Then, enter the following command to create a new ROS package:

```
catkin_create_pkg   my_nav_package   roscpp   move_base_msgs   actionlib
```

```
yhs@yhs-ros:~/catkin_ws/src$ catkin_create_pkg my_nav_package roscpp move_base_msgs actionlib
Created file my_nav_package/CMakeLists.txt
Created file my_nav_package/package.xml
Created folder my_nav_package/include/my_nav_package
Created folder my_nav_package/src
Successfully created files in /home/yhs/catkin_ws/src/my_nav_package. Please adjust the values in package.xml.
yhs@yhs-ros:~/catkin_ws/src$
```

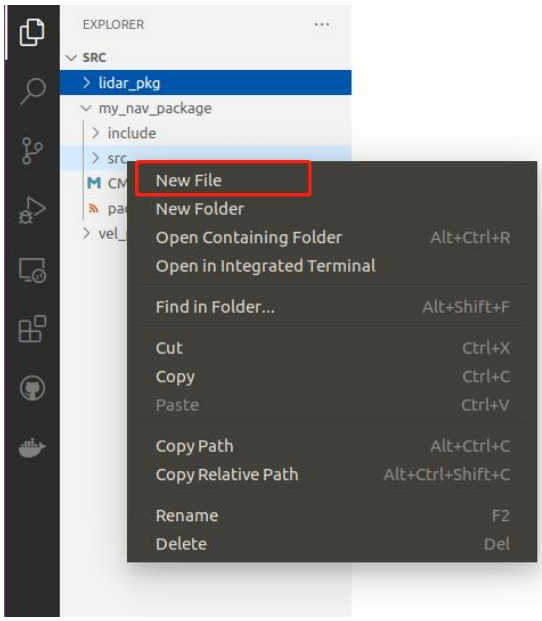
After pressing Enter, you will see the information displayed indicating the successful creation of the new ROS package.

This command has the following specific meaning:

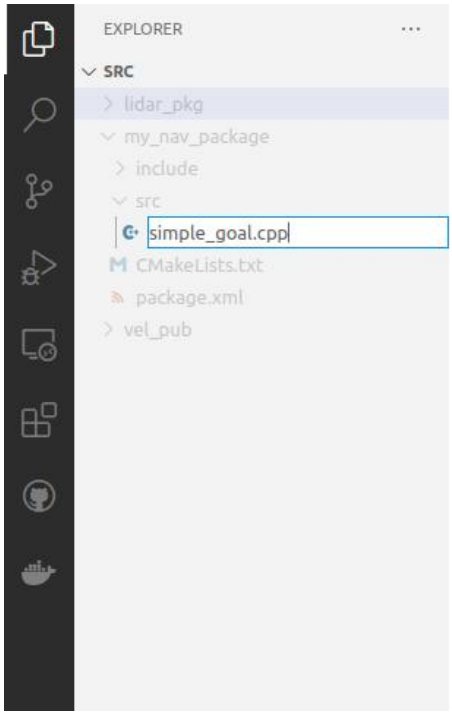
Command	Meaning
catkin_create_pkg	It is a command-line tool used to create a new ROS package.
my_nav_package	It is the name of the package you are creating. You can replace it with your desired package name.
roscpp	It is a dependency indicating that the package will use the roscpp library, which is the C++ client library for ROS.
move_base_msgs	The navigation messages dependencies for move_base
actionlib	It is a dependency that provides the programming interface for Navigation in the actionlib format.

2. Editing in an IDE

(1) In the IDE, you will see a new folder named "my_nav_package" in your workspace. Right-click on the "src" subfolder, and select "New File" to create a new code file.



(2) The newly created code file should be named "simple_goal.cpp".



(3) Once you have named the file, you can start writing the code for "simple_goal.cpp" on the right-hand side of the IDE.

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;
```

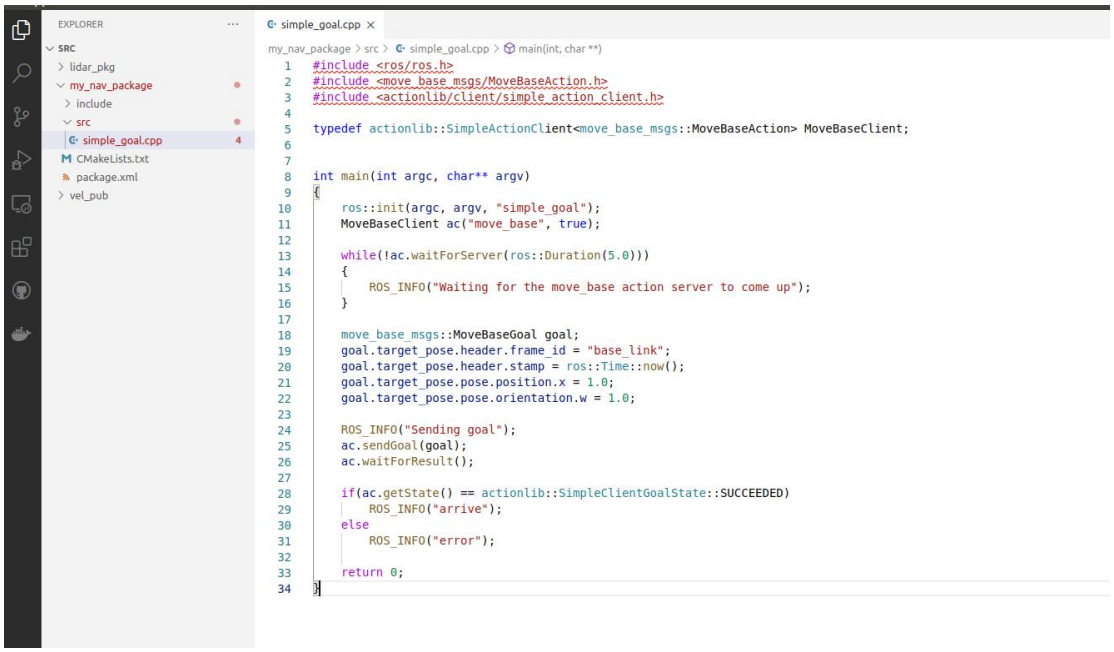
```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "simple_goal");
    MoveBaseClient ac("move_base", true);

    while(!ac.waitForServer(ros::Duration(5.0)))
    {
        ROS_INFO("Waiting for the move_base action server to come up");
    }

    move_base_msgs::MoveBaseGoal goal;
    goal.target_pose.header.frame_id = "base_link";
    goal.target_pose.header.stamp = ros::Time::now();
    goal.target_pose.pose.position.x = 1.0;
    goal.target_pose.pose.orientation.w = 1.0;

    ROS_INFO("Sending goal");
    ac.sendGoal(goal);
    ac.waitForResult();

    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
        ROS_INFO("arrive");
    else
        ROS_INFO("error");

    return 0;
}
```

1) In the beginning of the code, three header files are included. The first one is the system header file for ROS, the second one is the definition file for the navigation goal structure move_base_msgs::MoveBaseGoal, and the third one is the definition file for actionlib::SimpleActionClient.
2) The main function of the ROS node is int main(int argc, char** argv), which follows the same parameter definitions as other C++ programs.
3) Inside the main function, the node is initialized using ros::init(argc, argv, "simple_goal"), where the third parameter is the node name.
4) Next, a MoveBaseClient object named ac is declared. It is used to call and monitor the navigation service.
5) Before requesting the navigation service, it is necessary to ensure that the navigation service is up and running. Therefore, ac.waitForServer() is called to check the status of the navigation service. ros::Duration() is a sleep function, where the parameter represents the sleep duration in seconds. If the sleep is interrupted by a signal (in this

case, the signal indicating the navigation service has started), the sleep is interrupted. So, ac.waitForServer(ros::Duration(5.0)) means to sleep for 5 seconds and interrupt the sleep if the navigation service starts. By nesting it in a while loop, the program waits for 5 seconds and continues to sleep for another 5 seconds if the navigation service is not yet started, until the sleep is interrupted when the navigation service starts.

6) After confirming that the navigation service has started, a structure object of type move_base_msgs::MoveBaseGoal named goal is declared. It is used to pass the target information for navigation. goal.target_pose.header.frame_id indicates the coordinate frame in which the target position is specified. In the example, it is set to "base_link," indicating that the target position is defined in the robot's base coordinate system. goal.target_pose.header.stamp is set to the current timestamp. goal.target_pose.pose.position.x is set to 1.0, which means the destination for this navigation is 1.0 meter forward in the robot's base coordinate system along the X-axis (the robot's forward direction). The y and z components of goal.target_pose.pose.position are not assigned, so they default to 0. goal.target_pose.pose.orientation.w is set to 1.0, indicating that the target orientation for navigation is facing the positive direction of the X-axis (forward direction).
7) ac.sendGoal(goal) passes the navigation target information to the client ac of the navigation service, which monitors the subsequent navigation process.
8) ac.waitForResult() waits for the navigation result from MoveBase. This function blocks and waits until the entire navigation process is completed or interrupted due to some other reasons.
9) After ac.waitForResult() unblocks, ac.getState() is called to obtain the result of the navigation service. If it returns "SUCCEEDED," it means the navigation successfully reached the destination, and you can celebrate! If it returns any other result, it indicates that the navigation service was interrupted due to some reasons.

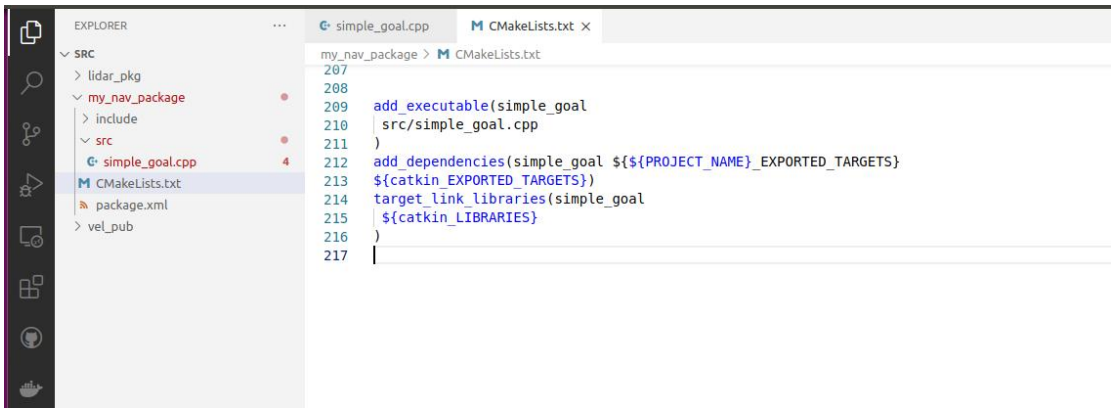
(4) After finishing the code, it is not immediately saved to the file. If you observe carefully, you will notice a small black dot to the right of the file name title "simple_goal.cpp" in the upper-right editing area. This dot indicates that the file has not been saved. Press the keyboard shortcut "Ctrl + S" to save the code file. The black dot next to the file name "simple_goal.cpp" in the upper-right editing area will change to a white close button. Now you can proceed with compiling the code.



(5) Once the code is written, the file name needs to be added to the compilation file in order to proceed with the compilation. The compilation file is located in the directory of "my_nav_package" and is named "CMakeLists.txt". Click on that file on the left side of the IDE, and the file content will be displayed on the right side. At the end of the "CMakeLists.txt" file, add a new compilation rule for "simple_goal.cpp".

Here is an example of the content:

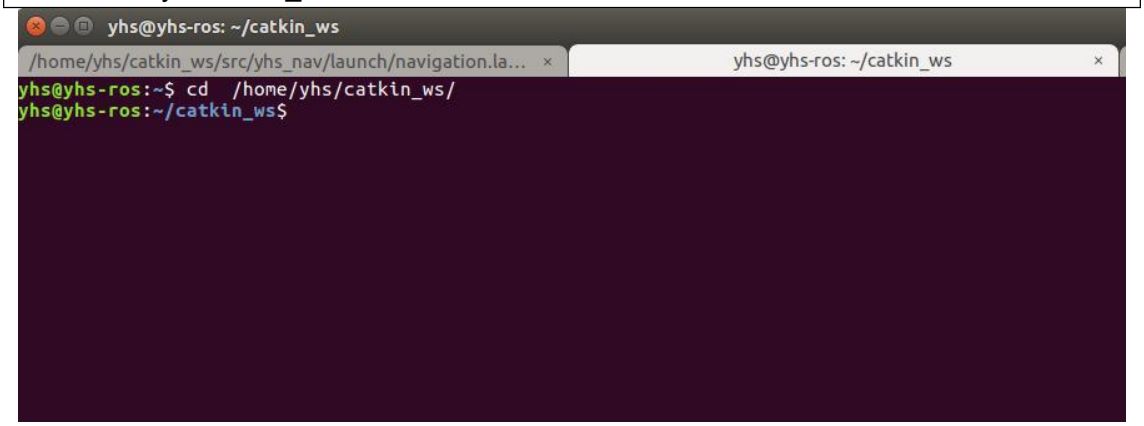
```
add_executable(simple_goal
  src/simple_goal.cpp
)
add_dependencies(simple_goal ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(simple_goal
  ${catkin_LIBRARIES}
)
```



3. Compilation

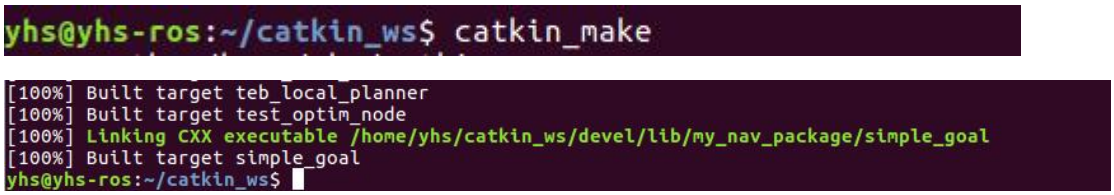
(1) Now let's proceed with the compilation of the code file. Start a terminal program and enter the following command to navigate to your ROS workspace:

```
cd  /home/yhs/catkin_ws/
```



(2) Once inside the ROS workspace directory, execute the following command to begin the compilation:

```
catkin_make
```

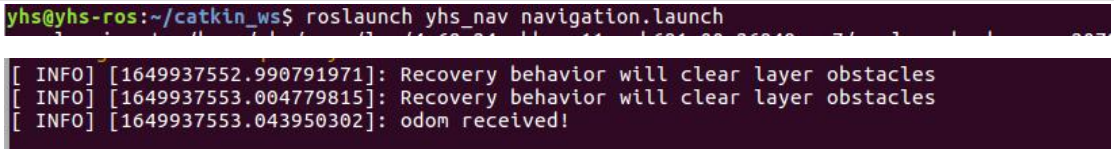


After executing this command, you will see scrolling compilation information until you encounter the message "[100%] Built target simple_goal," indicating that the new simple_goal node has been successfully compiled.

4. Run the Navigation Request Node.

(1) Before starting our navigation request node, we need to start the navigation service for the Yuhesen robot. Launch a terminal program and enter the following command:

```
roslaunch  ndt_localizer  ndt_localizer.launch
roslaunch  yhs_nav  navigation.launch
```



Alternatively, you can also start 2D navigation by following command:

```
roslaunch  yhs_nav navigation_2d.launch
```

(1) After successfully starting the navigation, follow the steps from the previous experiment to ensure successful localization of the robot.

(2) Once the localization is successful, open another terminal and enter the following command, then press Enter:

```
rosrun  my_nav_package  simple_goal
```

```
yhs@yhs-ros:~/catkin_ws$ rosrun my_nav_package simple_goal
```

```
yhs@yhs-ros:~/catkin_ws$ rosrun my_nav_package simple_goal
[ INFO] [1649937953.213240327]: Sending goal
[ INFO] [1649937958.815019803]: arrive
yhs@yhs-ros:~/catkin_ws$
```

At this point, switch the remote controller to the automatic control mode, and the robot will move forward for 1 meter before stopping.

5. Try assigning different values to goal.target_pose.pose.position.x and goal.target_pose.pose.position.y in the code to observe the robot's navigation behavior.

6. Attempt changing goal.target_pose.header.frame_id to a different coordinate system, such as "map," in the code to observe if the robot's navigation destination changes.

Experiment 10: Autonomous Charging

If the navigation suite includes a charging station, the ROS robot can achieve autonomous docking for recharging.
(Autonomous docking for recharging requires high localization accuracy for ROS robots and is currently suitable only for indoor environments with precise maps and unobstructed LiDAR sensors.)

To enable autonomous docking for recharging, you need to have a pre-built map.

Please note that the map must be created using the 3D LIO-SAM mapping method, as the 3D navigation system will be used for autonomous docking.

1. Open a terminal and start the 3D navigation by entering the following command:

```
roslaunch ndt_localizer ndt_localizer.launch
roslaunch yhs_nav navigation.launch
```

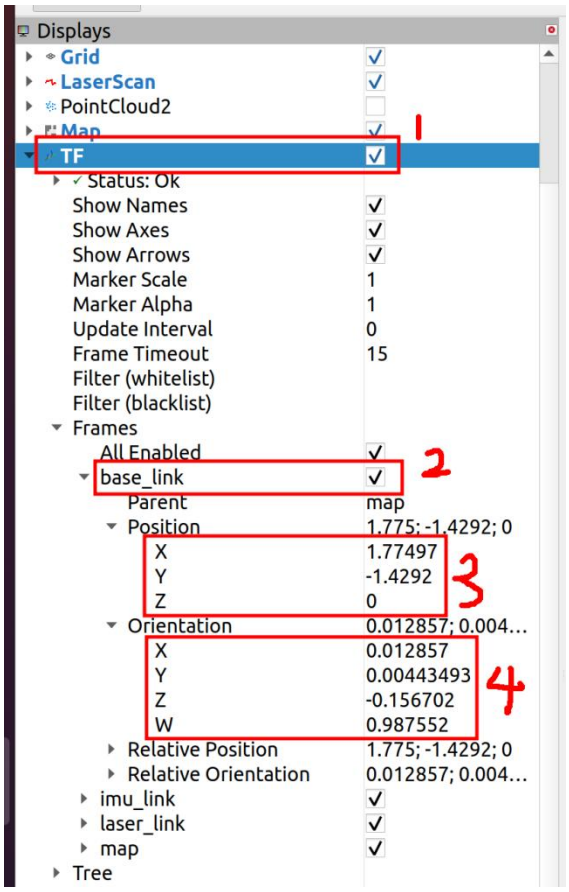
2. Open RVIZ, and initialize the pose of the robot by following these steps:

```
rviz
```

- Launch another terminal.
- Enter the following command to start RVIZ:
- Once RVIZ is launched, set up the necessary visualization and configuration for your robot's navigation system.
- Locate the "2D Pose Estimate" button in RVIZ's toolbar and click it.
- In the RVIZ window, click on the map where the robot's current position is located.
- Drag the arrow to indicate the robot's orientation.
- RVIZ will update the estimated pose of the robot based on your input.

3. Manually control the robot to align the charging electrodes on the power-receiving end with the charging electrodes on the charging station and successfully initiate the charging process. **(Note: After docking with the charging station, ensure that the charging electrodes are inserted to an appropriate depth and preferably in the center.)**

4. Record the current charging point's pose:
In RVIZ, observe the current pose of the robot.



Look for the Position and Orientation data of the robot.
Take note of these values, precision up to two decimal places is sufficient.

5. Open a terminal and start the docking node for autonomous recharging by entering the following command:

```
roslaunch recharge_controller recharge_controller.launch
```

```
yhs@yhs-ros:~$ roslaunch recharge_controller recharge_controller.launch
... logging to /home/yhs/.ros/log/5ae4dbee-355d-11ee-ab73-042b58098e2d/roslaunch-yhs-ros-2
361.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://192.168.1.102:39971/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES
  /
    recharge_controller (recharge_controller/recharge_controller_node)

auto-starting new master
process[master]: started with pid [2372]
ROS_MASTER_URI=http://192.168.1.102:11311

setting /run_id to 5ae4dbee-355d-11ee-ab73-042b58098e2d
process[rosout-1]: started with pid [2386]
started core service [/rosout]
process[recharge_controller-2]: started with pid [2389]
```

6. To view the services for docking and undocking from the charging station:

```
rosservice list
```

```
yhs@yhs-ros:~$ rosservice list
/dis_recharge
/recharge
/recharge_controller/get_loggers
/recharge_controller/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
```

Among them, /recharge is the service for initiating the autonomous recharging process, and /dis_recharge is the service for undocking from the charging station. In the following steps, we will use these two services to perform autonomous charging and undocking operations.

7. Initiate autonomous recharging by executing the following command.

```
rosservice call /recharge
```

After entering the command, press the Tab key, and the remaining content will be displayed:

```
yhs@yhs-ros:~$ rosservice call /recharge "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
recharge_goal:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: ''
  pose:
    position:
      x: 0.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 0.0"
```

Then, fill in the recorded Position and Orientation data **using the ← and → keys to move the cursor.**

Once you have completed filling in the data, manually move the robot to a different location, and then press Enter. Switch the remote controller to the autonomous mode, and the robot will begin the autonomous recharging process. **(Note: The accuracy of the map and the point cloud from the LiDAR sensor can affect the robot's localization, and consequently, the docking accuracy. If significant changes occur in the environment, it is recommended to rebuild the map promptly and avoid placing obstacles around the LiDAR sensor.)**

Once the recharging process is completed, it will return "1".

```
yhs@yhs-ros:~$ rosservice call /recharge "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
recharge_goal:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: ''
  pose:
    position:
      x: 1.87
      y: -0.86
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.97
      w: -0.21"
result: 1
```

If an exceptional situation occurs during the recharging process, it will return "0" instead of "1".

8. Undocking from the charging station.

After successfully charging the robot, initiate the undocking process by executing the following command:

```
rosservice call /dis_recharge
```

After entering the command, press the Tab key, and the remaining content will be displayed:

```
yhs@yhs-ros:~$ rosservice call /dis_recharge "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
dis_recharge:
  data: ''"
```

Then, simply press Enter, switch the remote controller to the autonomous mode, and the robot will begin the undocking process from the charging station.

Once the undocking is completed successfully, it will return "1". In case of any exceptional situation, it will return "0".

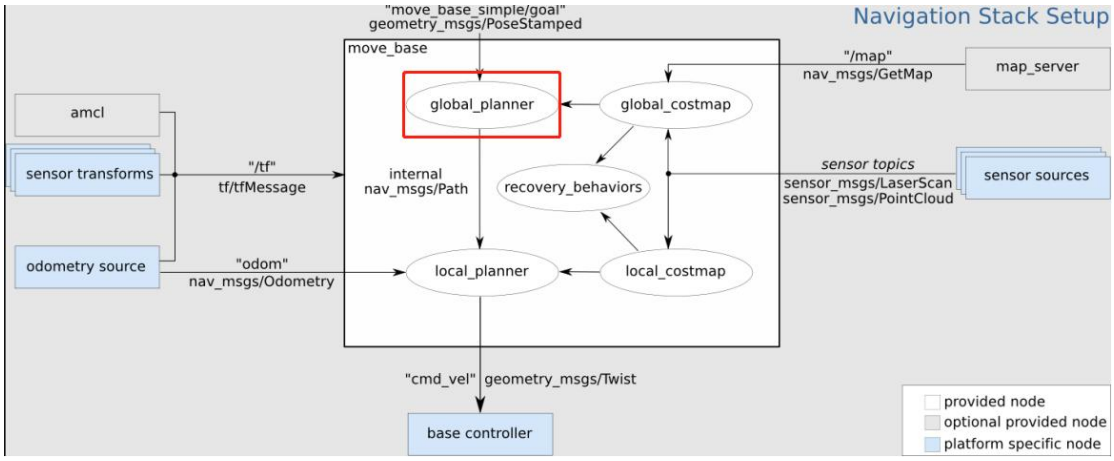
```
yhs@yhs-ros:~$ rosservice call /dis_rech
arge "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
dis_recharge:
  data: ''"
result: 1
```

Experiment 11: Global Path Planning

In autonomous driving, global path planning algorithms commonly use Dijkstra's algorithm or A* algorithm. For a detailed understanding of the algorithm principles and implementation, you can refer to the following resource: https://www.redblobgames.com/pathfinding/a-star/introduction.html.

Let's compare the effectiveness of these two algorithms in navigation.。

1. First, let's examine the position of global path planning in the navigation system:

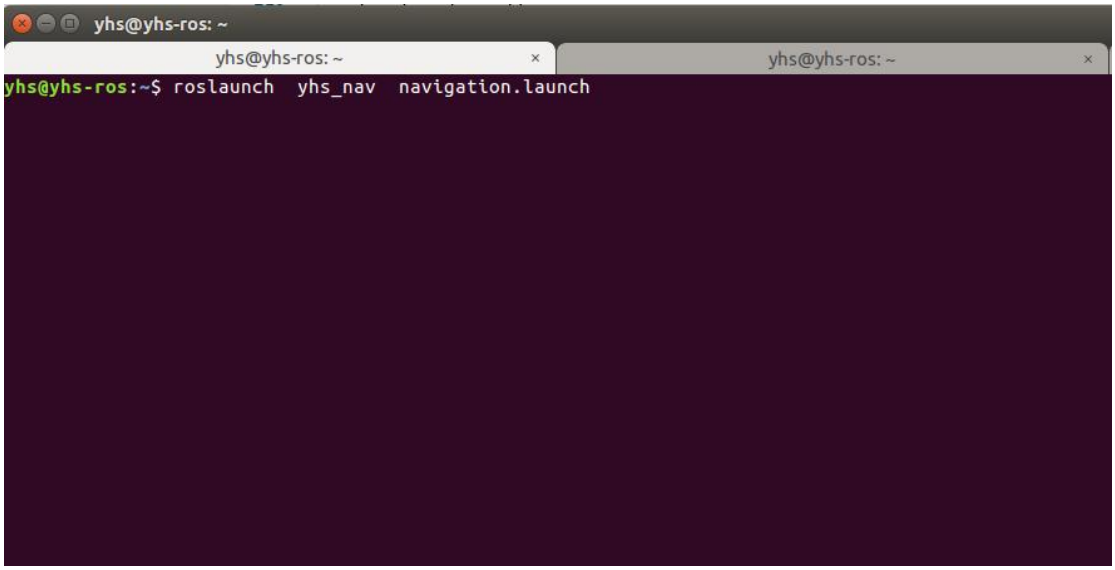


Looking at the image, the global_planner is the plugin package responsible for global path planning.

2. Start the navigation and add global path visualization in RVIZ.

(1) Open a terminal and enter the following command, then press Enter to start the 3D navigation:

roslaunch	ndt_localizer ndt_localizer.launch
roslaunch	yhs_nav navigation.launch



```
[ INFO] [1649993367.565004732]: Recovery behavior will clear layer obstacles
[ INFO] [1649993367.578541383]: Recovery behavior will clear layer obstacles
[ INFO] [1649993367.627263782]: odom received!
```

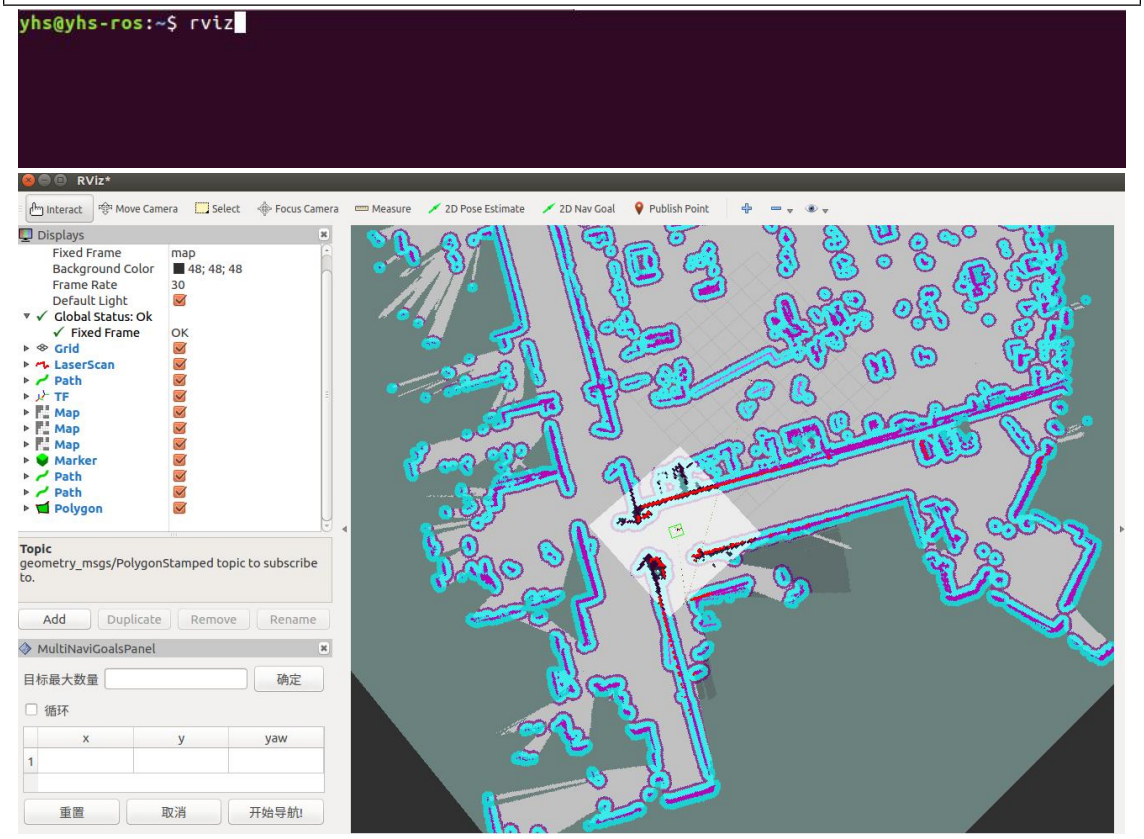
When you see the "odom received!" output, it indicates that the navigation has been successfully started.

Alternatively, you can start 2D navigation by following these steps:

```
roslaunch yhs_nav navigation_2d.launch
```

(1) Open another terminal, enter the following command, and press Enter:

```
rviz
```



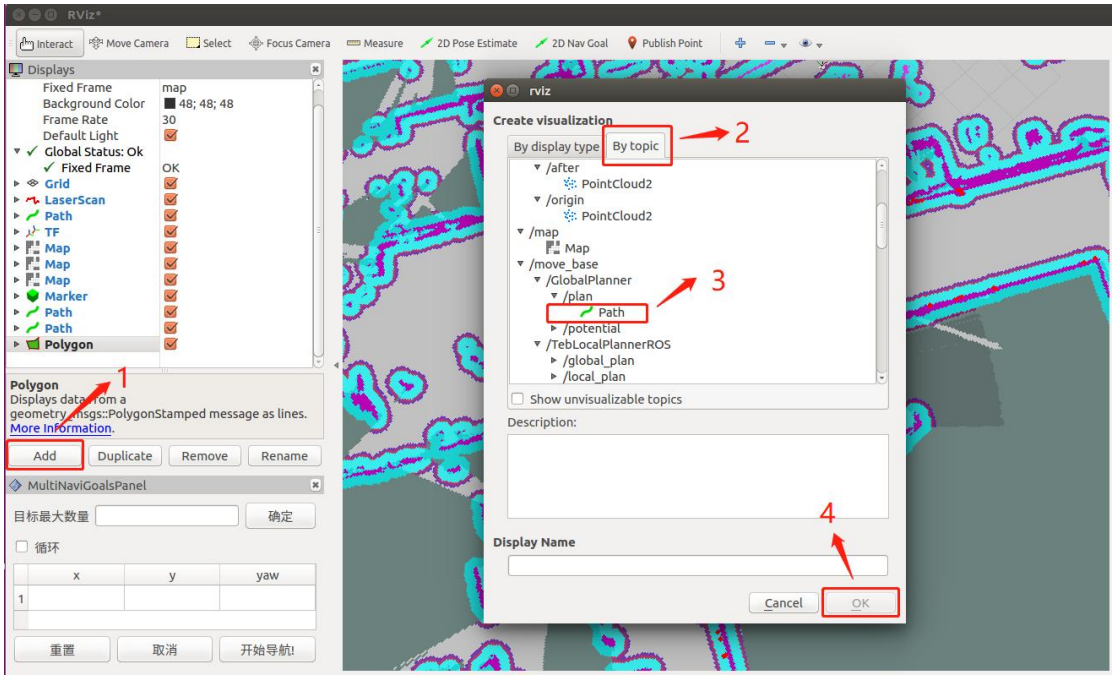
Following the methods mentioned in the previous sections, after sending the initial pose, you can manually control the car to move and ensure successful localization.

(2)To add global path visualization in RVIZ, follow these steps:

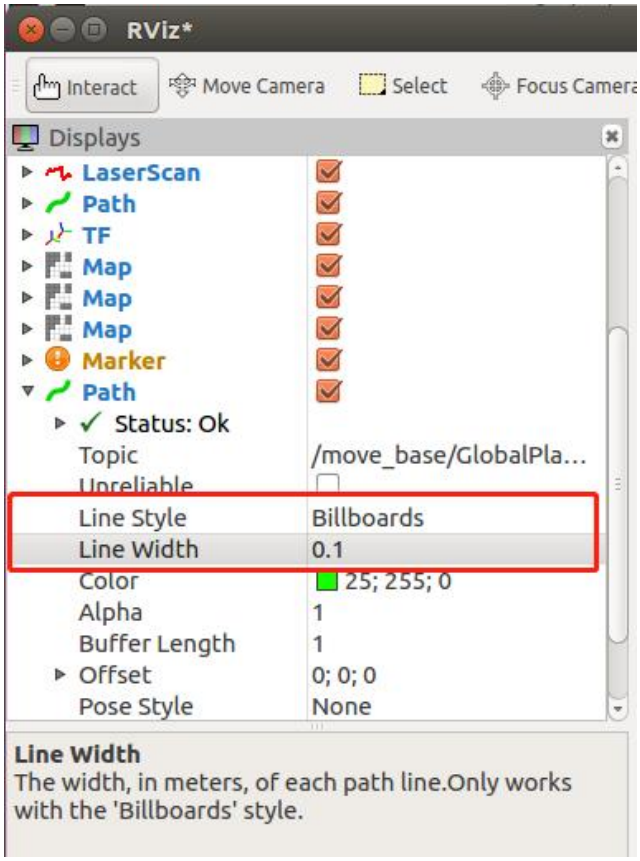
- Make sure RVIZ is running in the terminal you opened earlier.
- In the RVIZ window, click on the "Add" button.
- From the dropdown menu, select "Path" under the "By Topic" category.
- In the "Path" section, set the "Topic" field to the appropriate topic that provides the global path information. Typically, it is something like "/global_path" or "/move_base/GlobalPlanner/plan".

- Adjust any other settings or display preferences as needed.

Once configured, the RVIZ window will display the global path, providing a visual representation of the planned trajectory for the robot.



To add global path visualization in RVIZ, follow these steps as shown in the image (please note that this is a step-by-step guide, assuming you have already added the global path display and saved the configuration):



After expanding the "Path" option, you can choose the line style and width. As shown in the image, let's set the global path to be displayed in green.

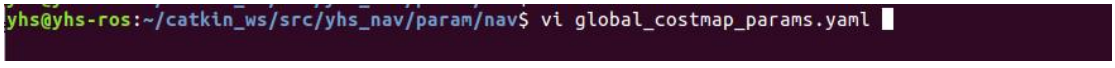
3. Using Dijkstra's Algorithm

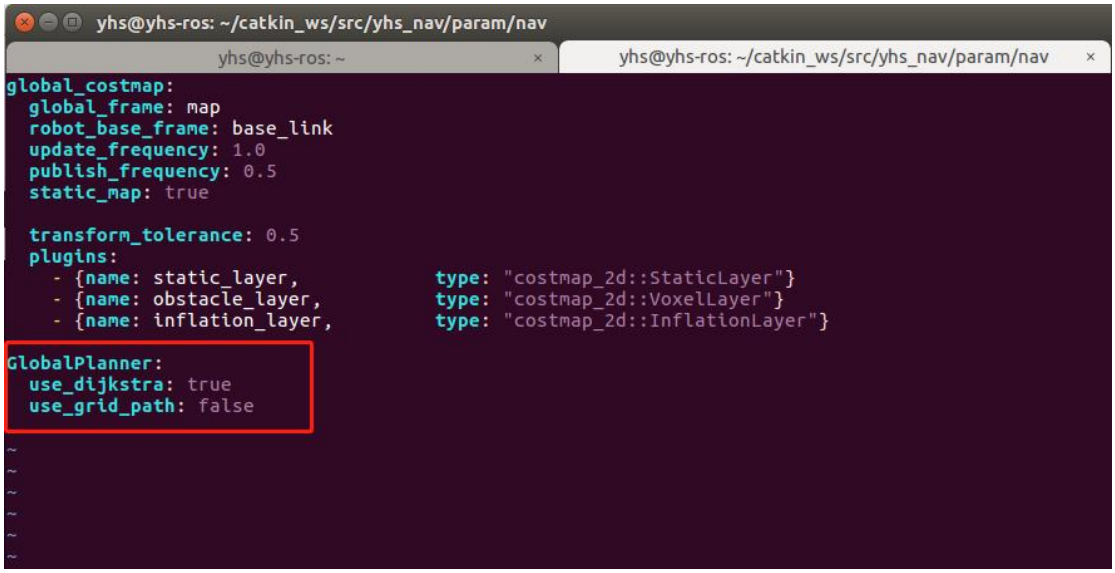
(1) Open the configuration file and enter the following command, then press Enter:

```
roscd  yhs_nav/param/nav
```

(2)Typing follow commands and press Enter:

```
vi  global_costmap_params.yaml
```

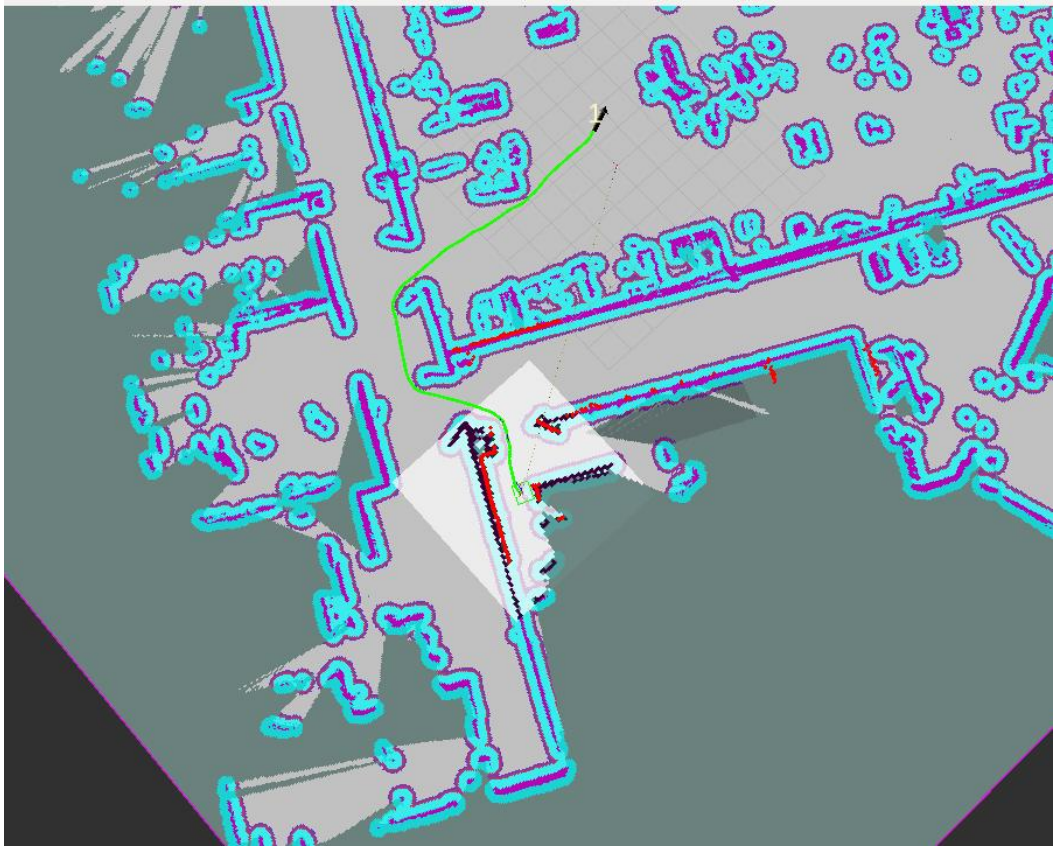




As shown in the image, make the necessary changes in the configuration file and save your modifications before exiting.

use_dijkstra	true
use_grid_path	false

(3) It is crucial to restart the navigation system after modifying the configuration. Then, follow the steps mentioned in the previous two sections to set the navigation target point. Check the planned global path by following these steps:



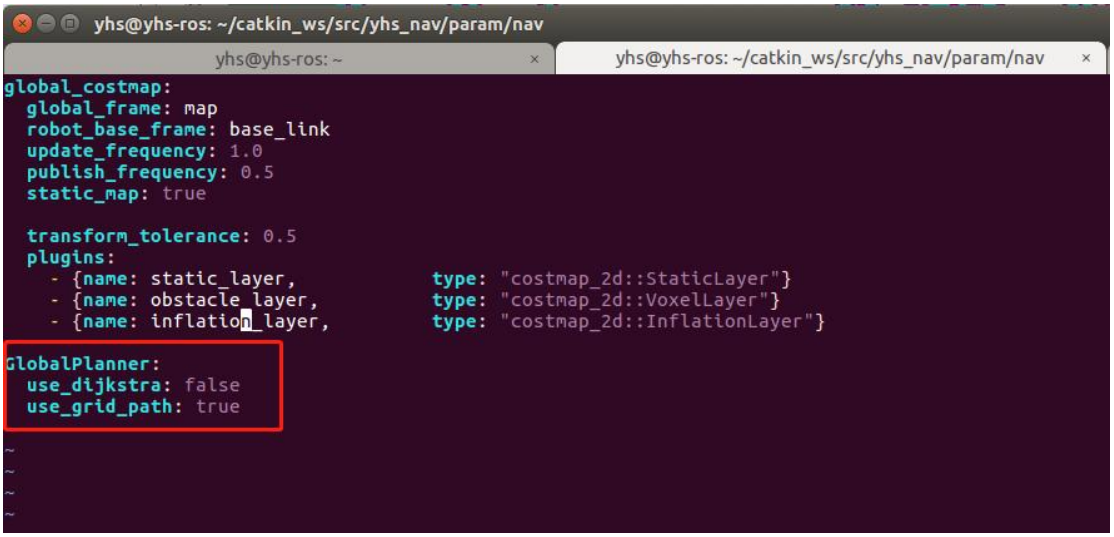
4. Using A* Algorithm

(1) Open the configuration file by entering the following command and pressing Enter:

```
roscd  yhs_nav/param/nav
```

(2) Typing below command and press Enter:

```
vi  global_costmap_params.yaml
```



As shown in the image, make the necessary changes in the configuration file and save your modifications before exiting.

use_dijkstra	false
use_grid_path	true

（3）一定要重新启动导航，然后按照上两节的方法下导航目标点。查看规划出来的全局路径：

5、我们可以综合考虑规划的时长和使用的场景来采用哪一种全局路径规划算法。

Experiment 12: Local Path Planning

In the previous experiment, we selected a global path planning algorithm and sent the navigation point to generate the global path. Then, we used the local path planner to generate specific action strategies for the robot based on this path and the information from the costmap. The commonly used local path planning algorithms are the Dynamic Window Approach (DWA) and the Time Elastic Band (TEB) algorithm.

Dynamic Window Approach (DWA): The DWA algorithm samples multiple sets of velocities in the velocity space (v, w) and simulates the motion trajectories of these velocities over a certain period of time. It evaluates these trajectories using a cost function and selects the optimal trajectory to drive the robot's movement.

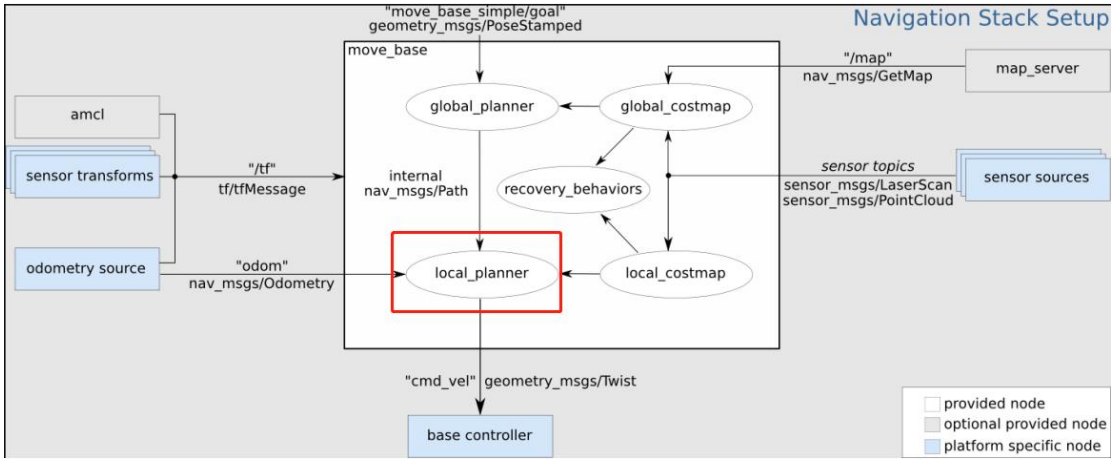
Time Elastic Band (TEB): The TEB algorithm is a nonlinear optimization algorithm. It utilizes the global path as the initial trajectory to obtain a series of discrete robot poses (pose) and corresponding time differences (timediff) within a local range. The poses and time differences on the trajectory need to satisfy constraints such as velocity, acceleration, robot kinematics, and distance to obstacles. These constraints can be modeled as a nonlinear optimization problem to obtain an optimized trajectory. The velocity can be derived from the poses and time differences, which is then sent to the robot for control.

Comparison of advantages and disadvantages:

	Advantages	Disadvantages
DWA	Simple and computationally efficient, suitable for real-time applications. Can handle dynamic obstacles and robot constraints.	May result in suboptimal trajectories in complex environments. Limited ability to handle long-term planning.
	suitable for differential and omnidirectional chassis	Not suitable for Ackermann steering chassis
TEB	Suitable for most chassis	Requires more computational resources compared to DWA.
	Provides smooth and dynamically feasible trajectories.	Significant fluctuations in speed and angle, as well as unstable control.
	It has good obstacle avoidance capabilities for dynamic obstacles.	Suboptimal (non-global) solution.

In the navigation system, the TEB (Timed Elastic Band) local planning algorithm is commonly used. Next, we will proceed with the parameter tuning for TEB local planning.

1.Let's start by examining the position of the local path in the navigation system.

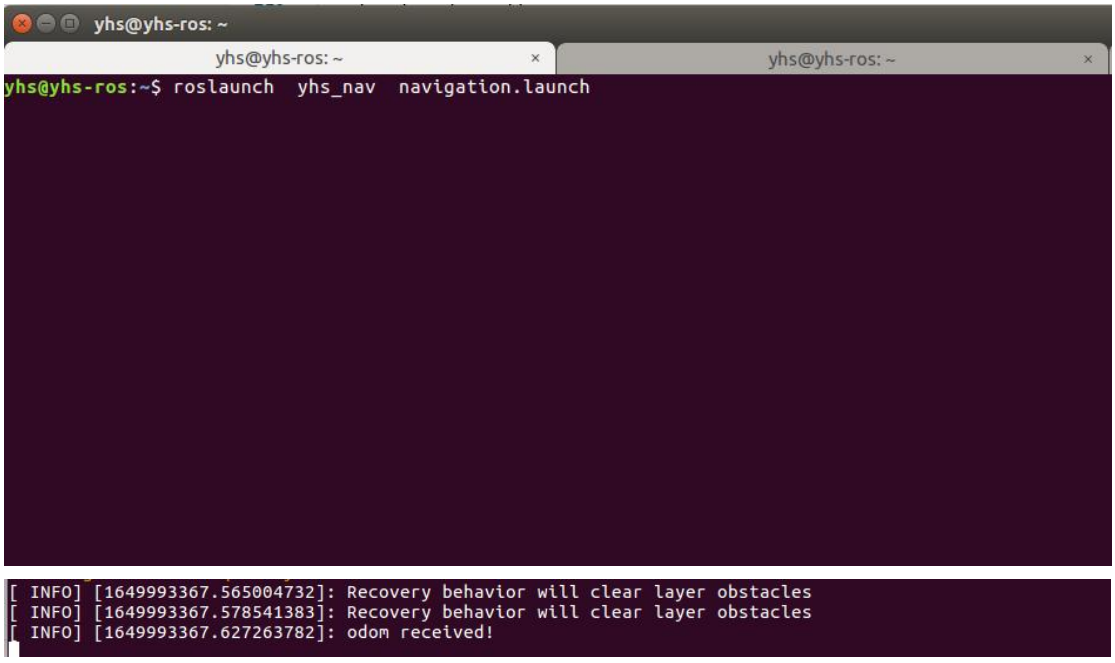


As shown in the image, the "local_planner" refers to the plugin package responsible for local path planning in the navigation system.

2.Start the navigation and add local path visualization in RVIZ.

(1)Open a terminal, enter the following command and press Enter to start the 3D navigation:

roslaunch ndt_localizer ndt_localizer.launch
roslaunch yhs_nav navigation.launch



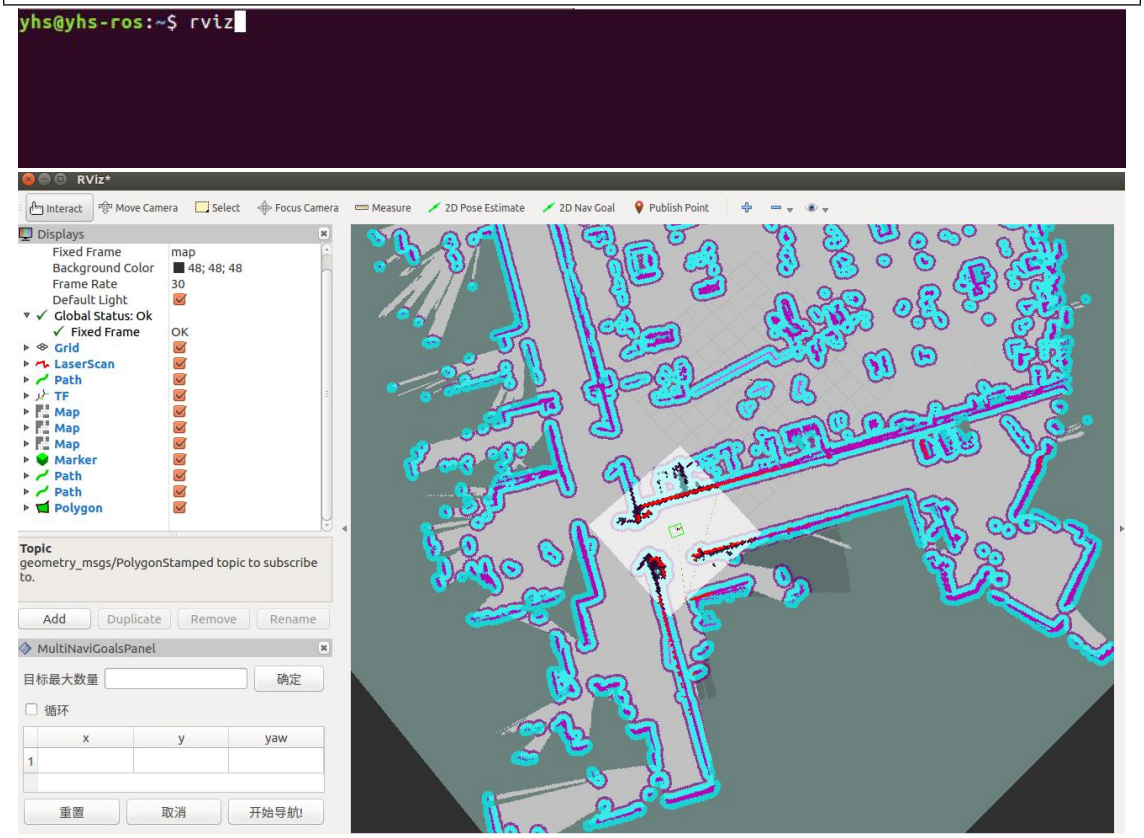
Once you see the output message "odom received!" in the terminal, it indicates that the navigation system has been successfully launched.

Alternatively, to start the 2D navigation, follow these steps:

roslaunch yhs_nav navigation_2d.launch

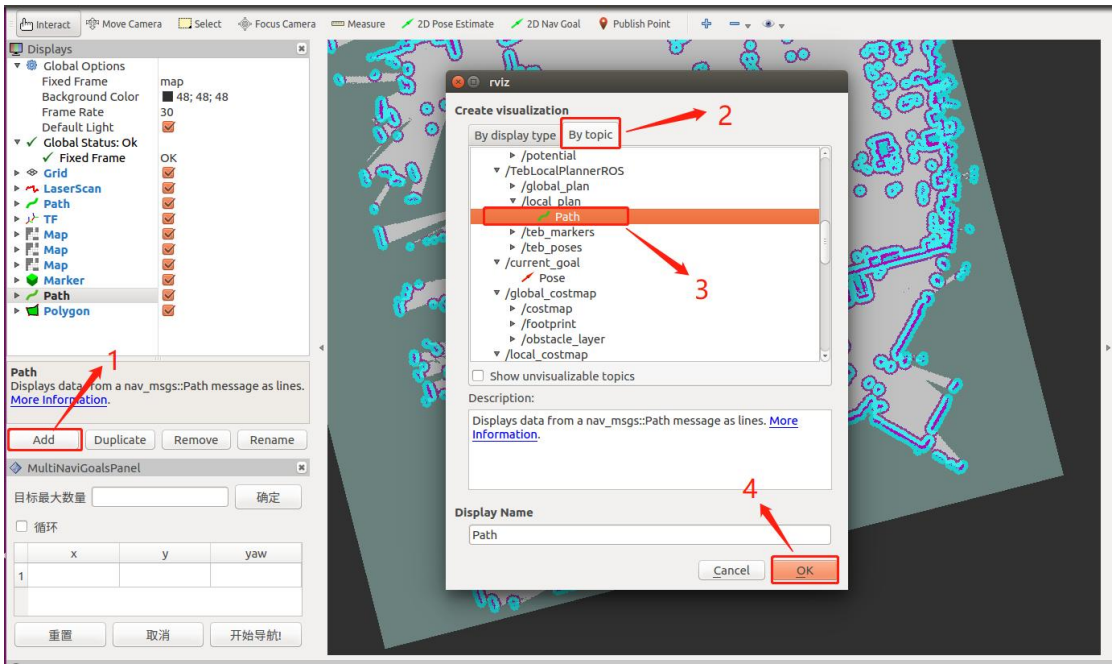
(2)Open another terminal, enter the following command and press Enter to start 2D navigation:

```
rviz
```

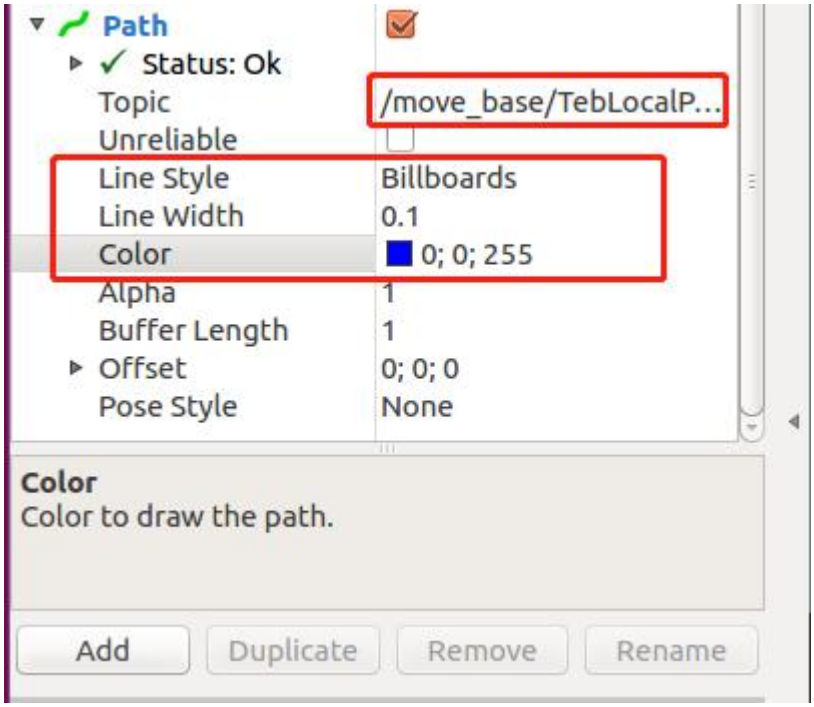


Following the methods provided in the previous two sections, after sending the initial pose, control the movement of the remote-controlled car to achieve successful localization.

(3) Add local path visualization in RVIZ.



Follow the steps shown in the image to add local path visualization (assuming local path visualization has already been added and the configuration has been saved).



After expanding the "Path" option, you can choose the style and width of the lines. As shown in the image, we set the local path to be displayed in blue.

3.Adjusting TEB Parameters

(1) Let's first understand the parameters of TEB. You can visit the following link: http://wiki.ros.org/teb_local_planner to view detailed parameter descriptions. Here, we will only introduce the parameters that we may need to adjust:

Trajectory	**Parameters related to the trajectory:**
dt_ref	Desired time resolution of the trajectory.
dt_hysteresis	Hysteresis effect for automatic size adjustment based on the current time resolution. Typically set to around 10% of dt_ref.
feasibility_check_no_poses	Number of poses analyzed for feasibility checks per sampling interval.

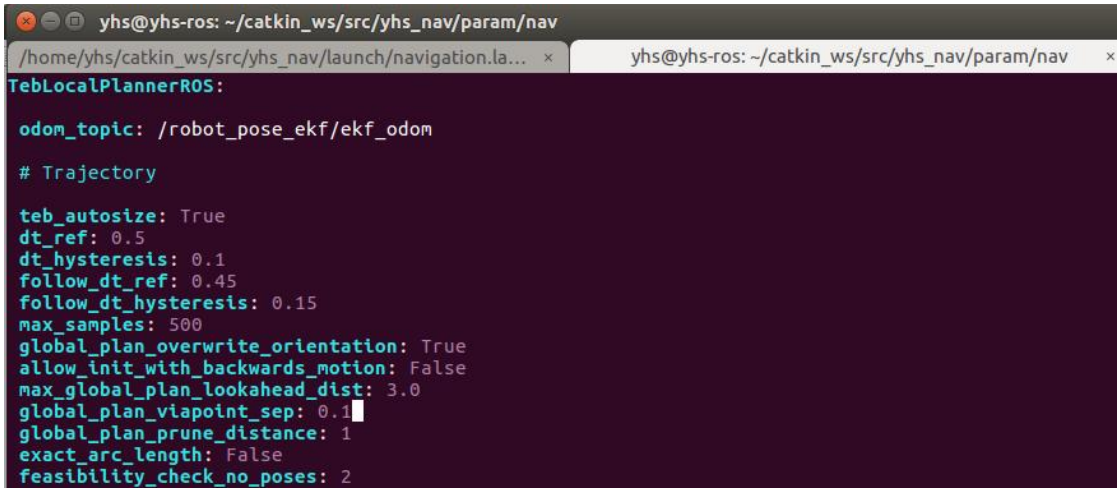
Robot	**Parameters related to robot velocity:**
max_vel_x	Maximum linear velocity of the robot.
max_vel_x_backwards	Maximum linear velocity in backward direction for the robot.
acc_lim_x	Linear acceleration limit for the robot.
max_vel_theta	Maximum angular velocity of the robot.
acc_lim_theta	Angular acceleration limit for the robot.

Optimization	**Parameters related to optimization:**
weight_kinematics_forward_drive	Weight for enforcing forward drive in the optimization process.
weight_viapoint	Assign a weight to viapoints in the optimization process of the TEB local planner.

Obstacles	**Parameters related to obstacles:**
min_obstacle_dist	Minimum desired distance to obstacles.
inflation_dist	Buffer zone around obstacles (should be greater than min_obstacle_dist to take effect).

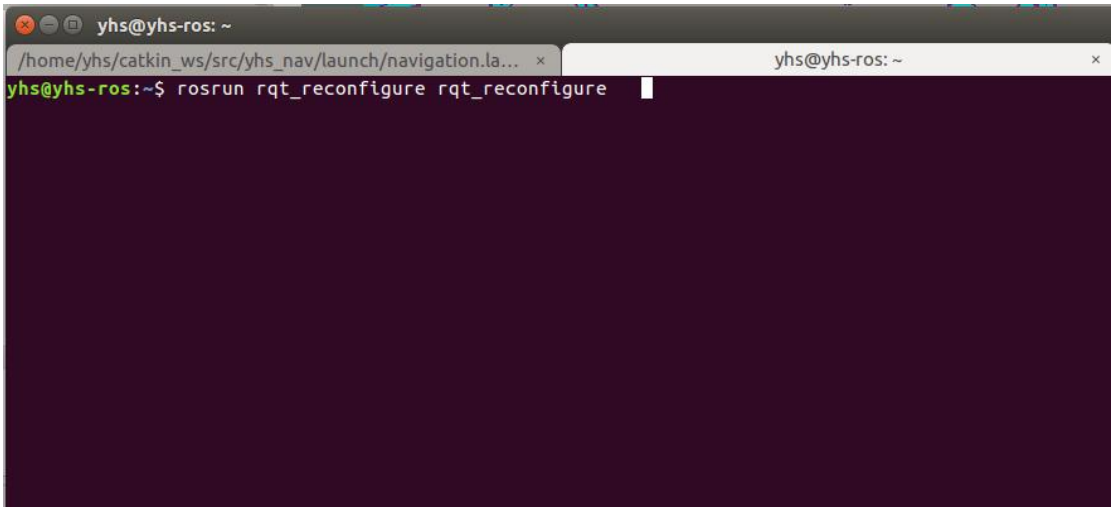
(2) All TEB parameters are located in the teb_local_planner_params.yaml configuration file, which can be modified using the following command:

```
vi /home/yhs/catkin_ws/src/yhs_nav/param/nav/teb_local_planner_params.yaml
```

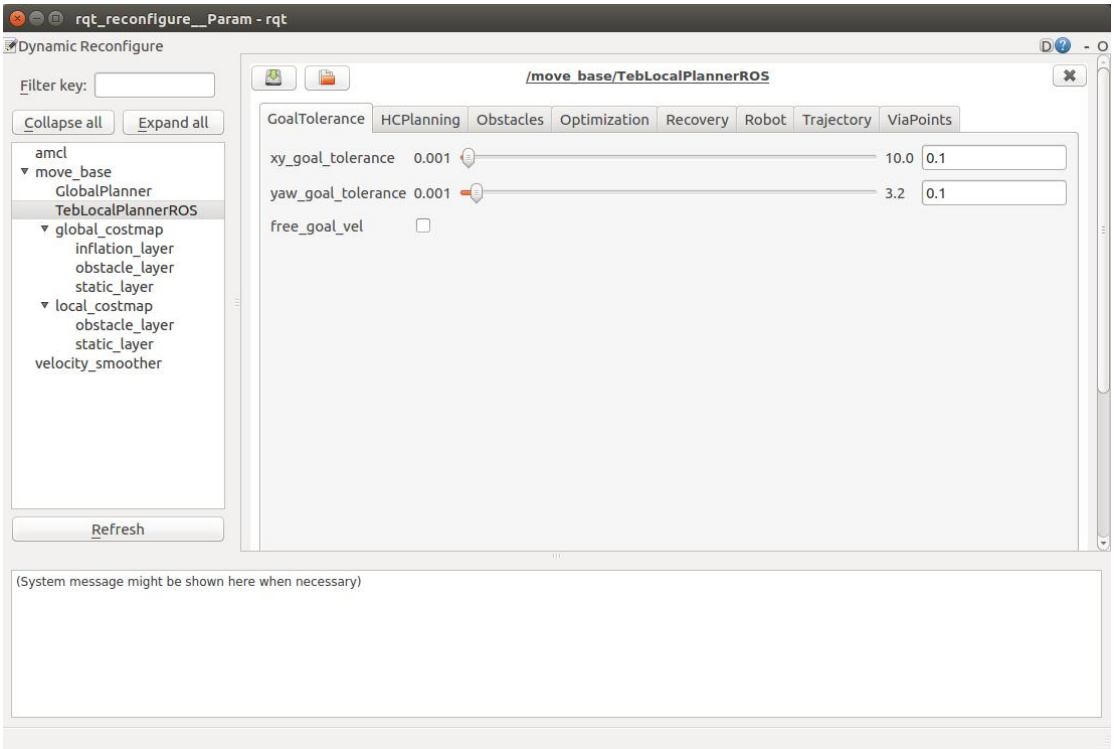


(3) Modifying the configuration file directly to debug local path planning is inefficient as it requires restarting navigation each time after modification. ROS provides a software interface for dynamically adjusting parameters. Additionally, you can open a new terminal and use the following command to launch the dynamic reconfigure interface:

```
rosrun rqt_reconfigure rqt_reconfigure
```



Below window will be show after press Enter:

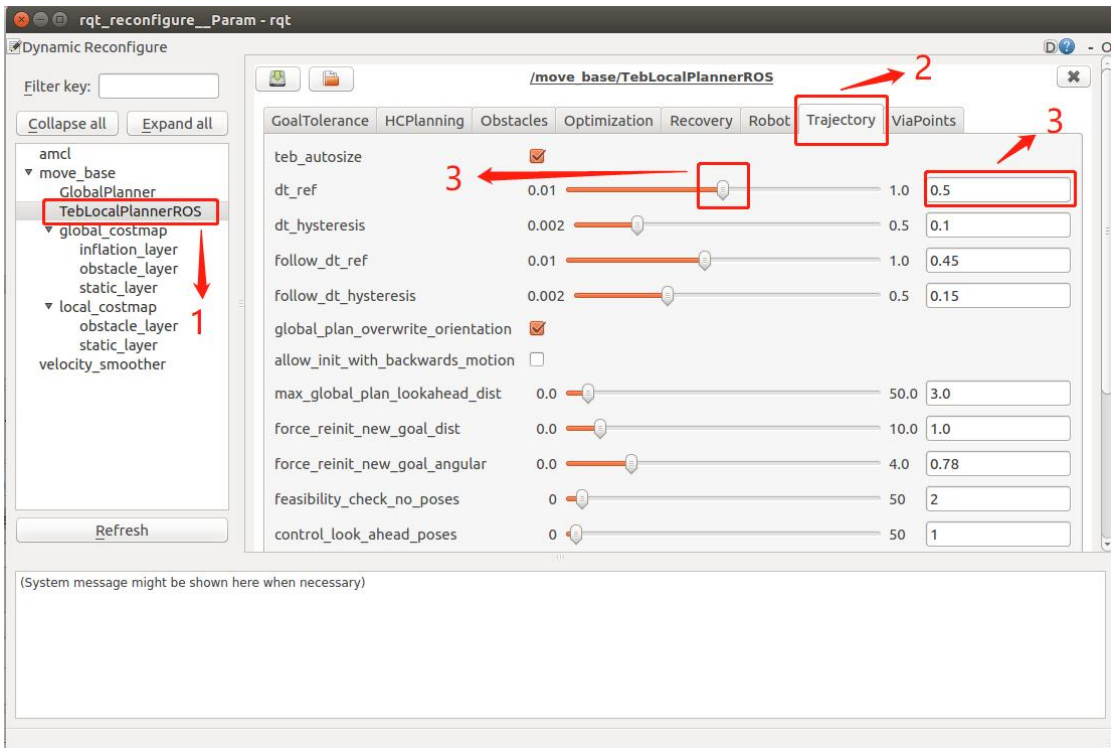


(4) Instructions:

Step 1: Click on "TebLocalPlannerROS" under the "move_base" section on the left side.

Step 2: On the top-right panel, click to select a specific category of parameters, such as "Trajectory" parameters.

Step 3: Drag the slider of a parameter or enter an exact numerical value in the input field, then press Enter.



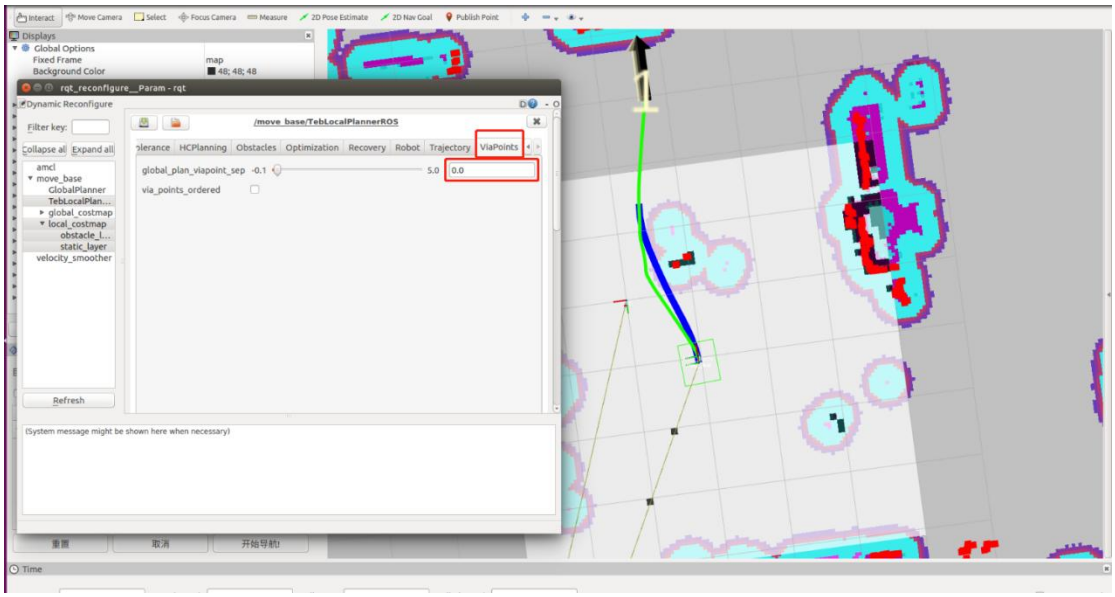
(5) Debugging Process:

Before switching the controller to automatic control, follow the procedure from Experiment 8 to send a navigation goal.

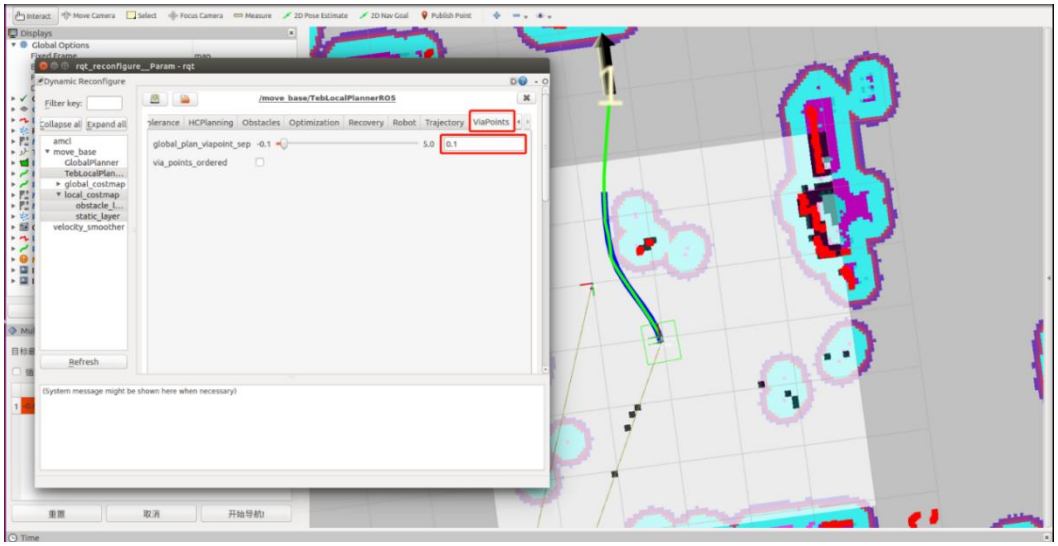
Keep the dynamic configuration interface at the top layer.

Adjust a specific parameter under a category, such as "Global_plan_viapoint_sep" under the "ViaPoints" category:

When the value of "Global_plan_viapoint_sep" is set to 0, the blue local path is closer to the obstacles.



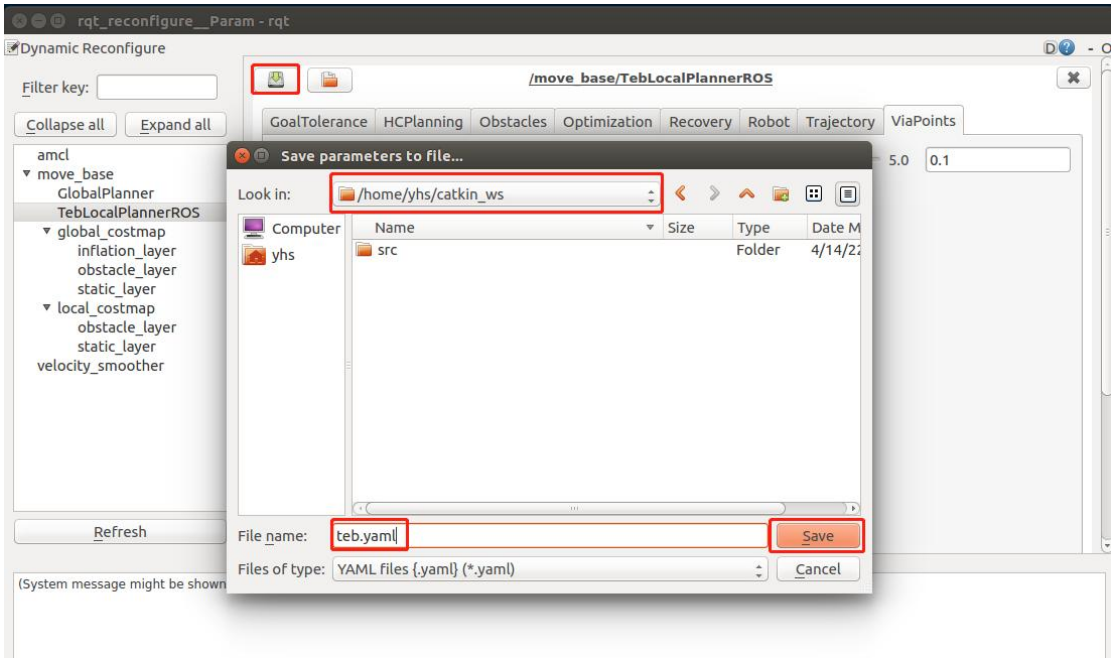
When the value of "Global_plan_viapoint_sep" is set to 0, the blue local path is closer to the obstacles.



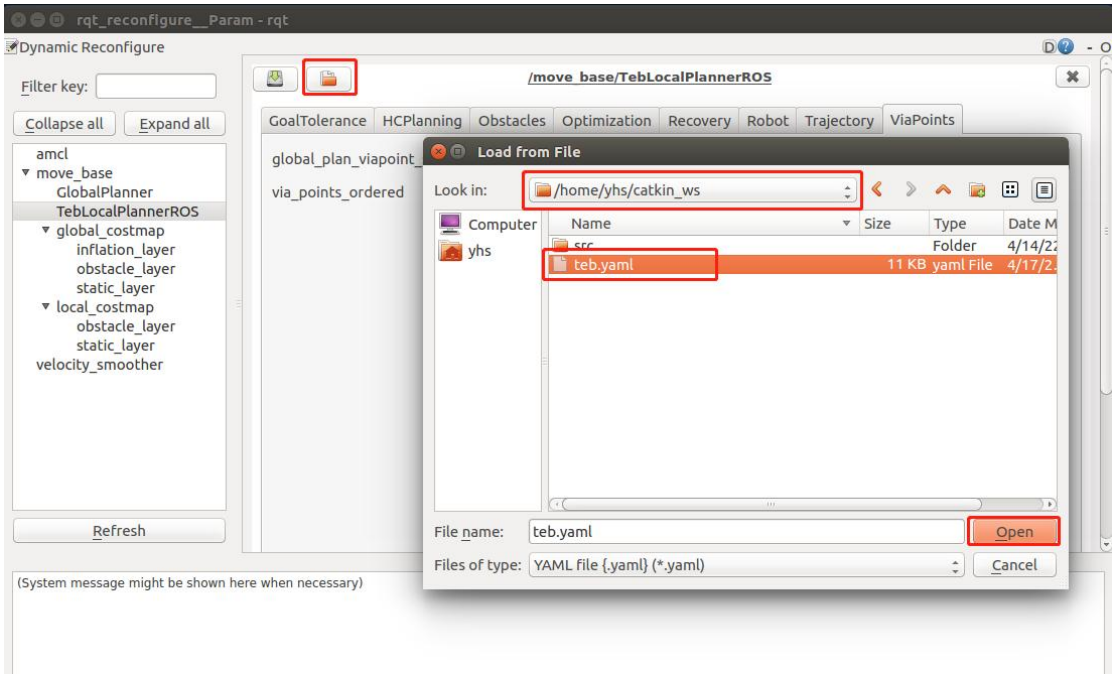
When the planned local path aligns well with our expectations, you can switch the controller to automatic control mode. This allows the vehicle to follow the local path and observe the actual performance.

(6) When the actual performance of the vehicle meets our requirements, we can save the parameters as follows:

Click on the save icon in the top-left corner, select a save path in the pop-up dialog, enter a name for the saved parameters, and click the save button to save all the parameters.

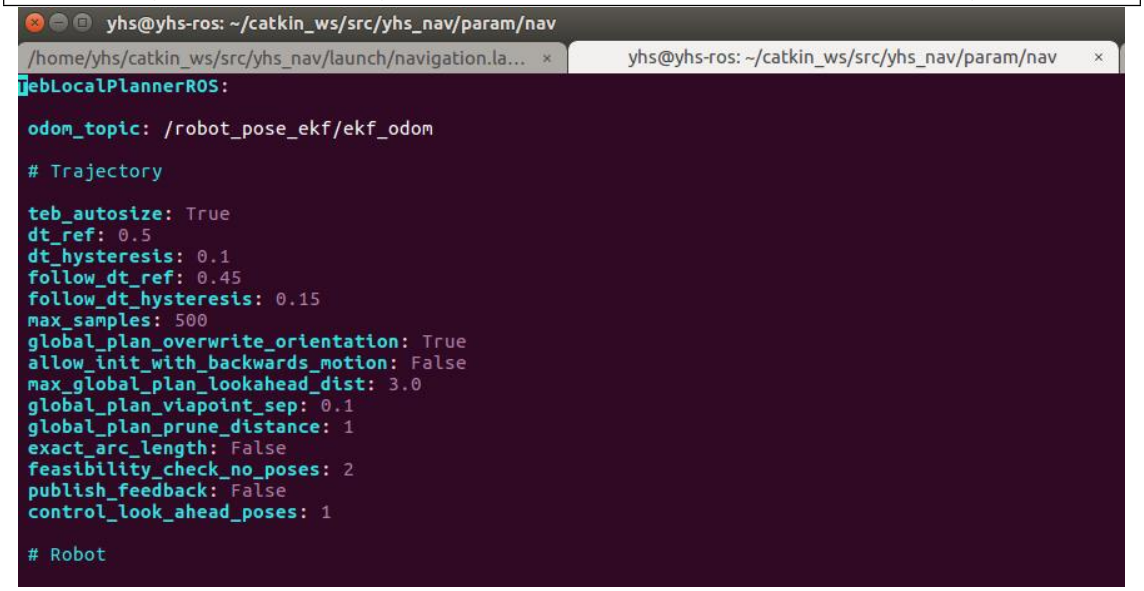


After saving, click on the open icon, select the file, and you can load the configuration.



(7) Finally, if we decide to change certain parameters, we need to open the "teb_local_planner_params.yaml" file, modify the parameters, and enter the command followed by pressing Enter:

```
vi /home/yhs/catkin_ws/src/yhs_nav/param/nav/teb_local_planner_params.yaml
```



After making the changes, save and exit the file. Then, restart the navigation system to apply the modifications.

(8) We have adjusted the parameters of the TEB local planner to a relatively suitable configuration. After following the above method to adjust the parameters, there is no need to modify the "teb_local_planner_params.yaml" file. However, if you decide to make further changes, it is recommended to create a backup of the file before modifying it.

Experiment 13: RGB-D Camera

The Yuhesen ROS robot is equipped with a vision sensor on its front end, which is a typical RGB-D camera. RGB-D stands for "RGB-Depth," where "RGB" refers to the colors red, green, and blue, representing the color image, and "Depth" refers to the depth (distance) image. Therefore, an RGB-D camera is a composite sensor that can perceive object color and detect object distance. In terms of data acquisition, it provides a color image and a point cloud containing distance information. In this experiment, we will have a general understanding of the camera's principle of operation and the data it captures.

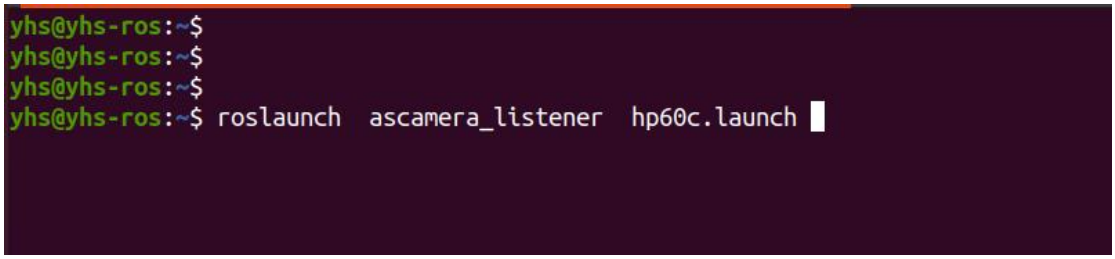
First, let's take a look at the external components of the depth camera. The main body is in the shape of a rectangular prism. On the front panel of the main body, several key sensors are arranged:

In this experiment, we primarily utilize the color camera.

1.Start the camera

(1) Open a terminal and enter the command, then press Enter to start the camera node.

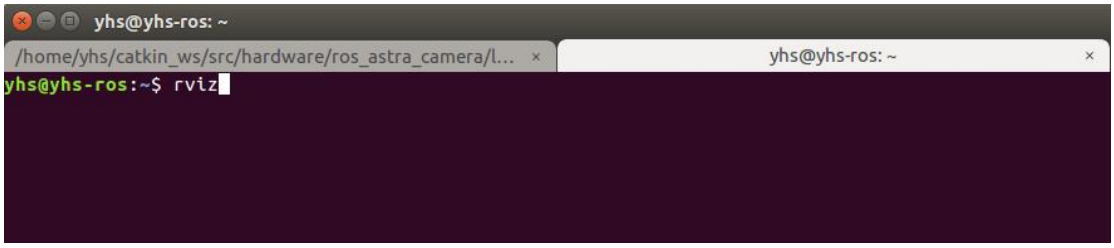
Single camera start: roslaunch ascamera_listener hp60c.launch
Multiple cameras start: roslaunch ascamera_listener hp60c_multiple.launch



```
2023-11-14 13:56:27[INFO] [CameraPublisher.cpp] [1174] [setResolution] set depth resolution: 320 x 240 @ 15fps
2023-11-14 13:56:27[INFO] [CameraPublisher.cpp] [1188] [setResolution] set rgb resolution: 640 x 480 @ 15fps
2023-11-14 13:56:28[INFO] [CameraHp60c.cpp] [899] [setInternalParameter] mjpeg info: size(640x480)
2023-11-14 13:56:28[INFO] [CameraHp60c.cpp] [250] [startStreaming] start streaming
2023-11-14 13:56:28[INFO] [CameraSrv.cpp] [175] [onAttached] attached end
2023-11-14 13:56:28[INFO] [Camera.cpp] [120] [backgroundThread] SN [ ASC60CD21000190 ]'s parameter:
2023-11-14 13:56:28[INFO] [Camera.cpp] [121] [backgroundThread] irfx: 214.343
2023-11-14 13:56:28[INFO] [Camera.cpp] [122] [backgroundThread] irfy: 214.254
2023-11-14 13:56:28[INFO] [Camera.cpp] [123] [backgroundThread] ircx: 157.263
2023-11-14 13:56:28[INFO] [Camera.cpp] [124] [backgroundThread] ircy: 119.298
2023-11-14 13:56:28[INFO] [Camera.cpp] [125] [backgroundThread] rgbfx: 287.19
2023-11-14 13:56:28[INFO] [Camera.cpp] [126] [backgroundThread] rgbfy: 287.027
2023-11-14 13:56:28[INFO] [Camera.cpp] [127] [backgroundThread] rgbcx: 161.588
2023-11-14 13:56:28[INFO] [Camera.cpp] [128] [backgroundThread] rgbcy: 119.314
```

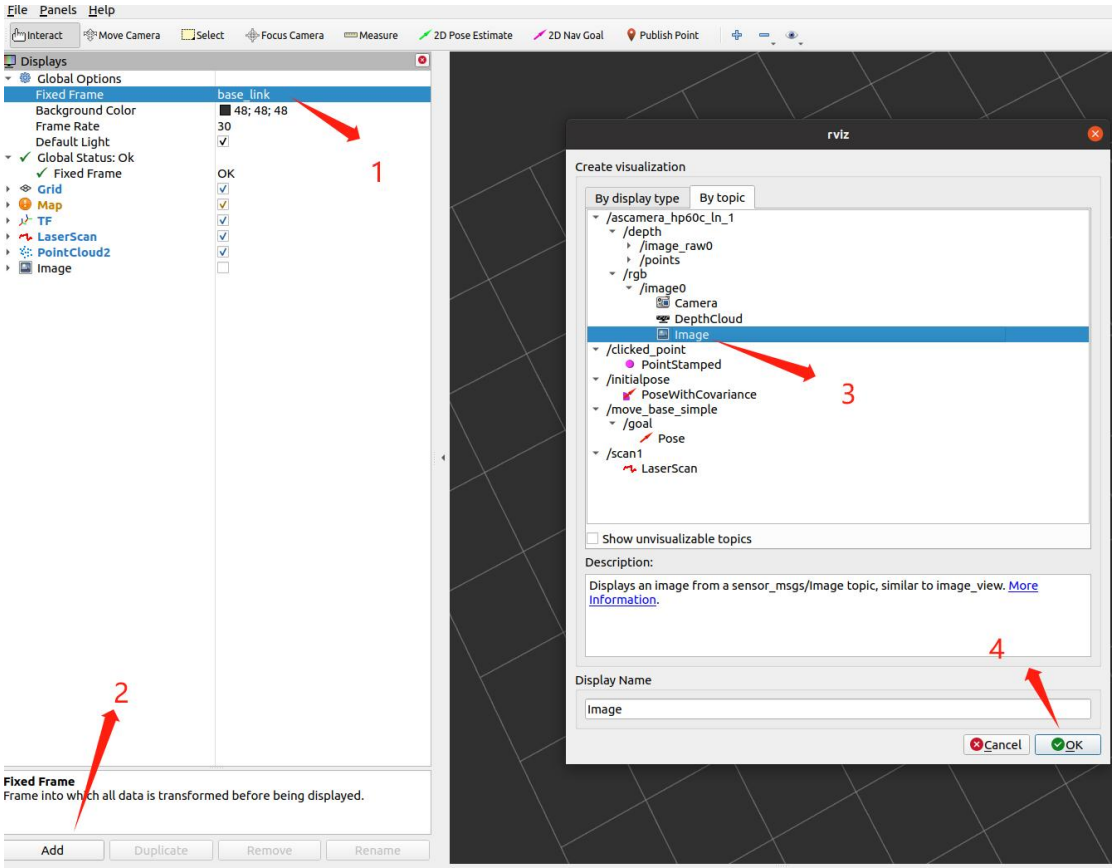
(2) Open another terminal and enter the command to launch RViz.

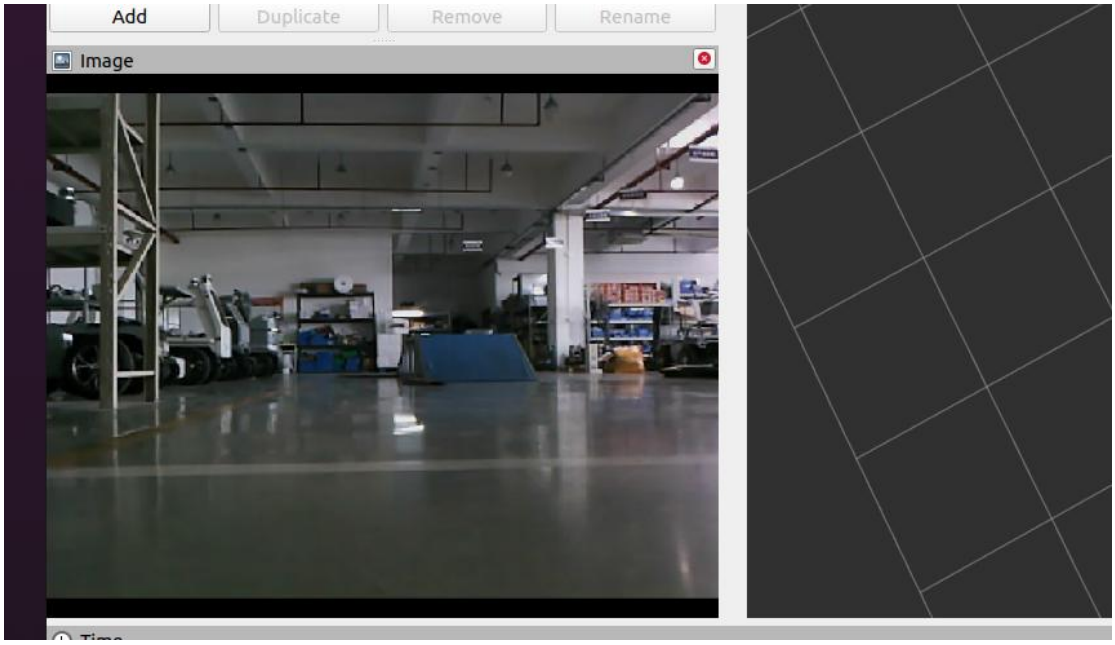
```
rviz
```



2.Adding color image display

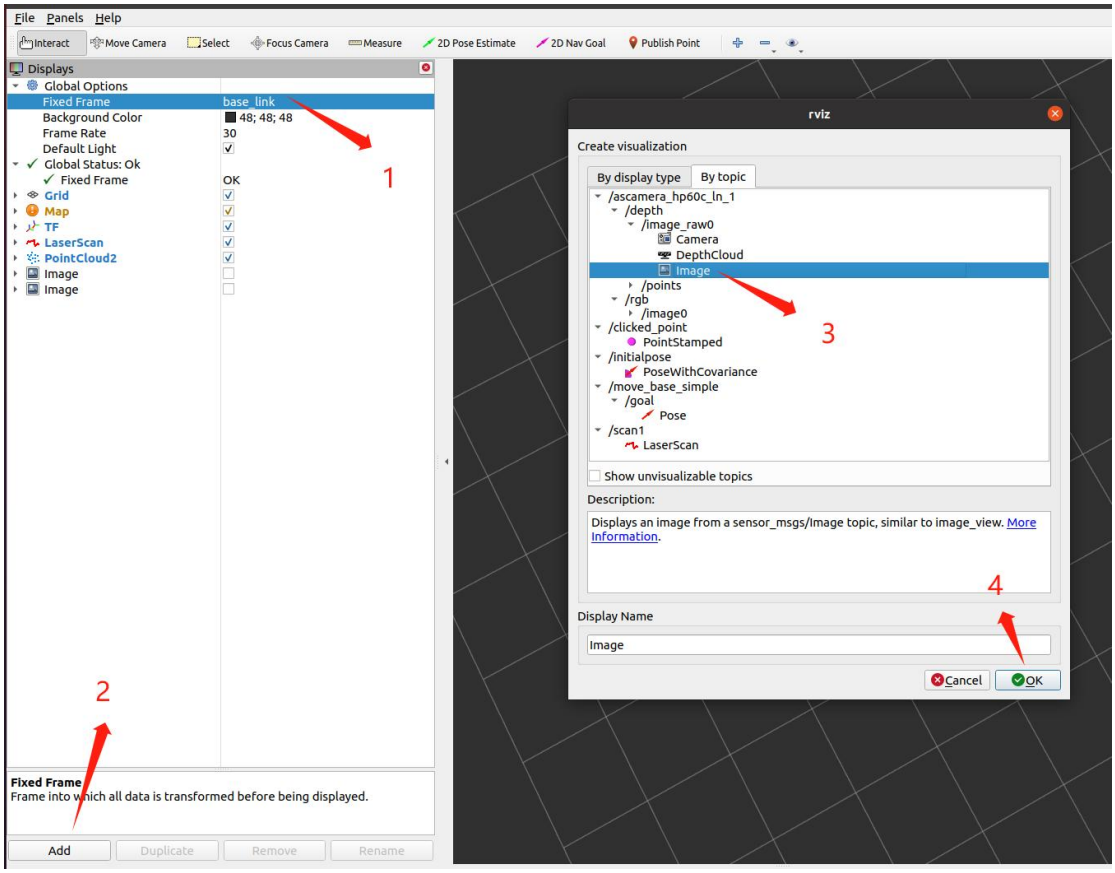
Follow the steps below to add the display of the color image in RViz:

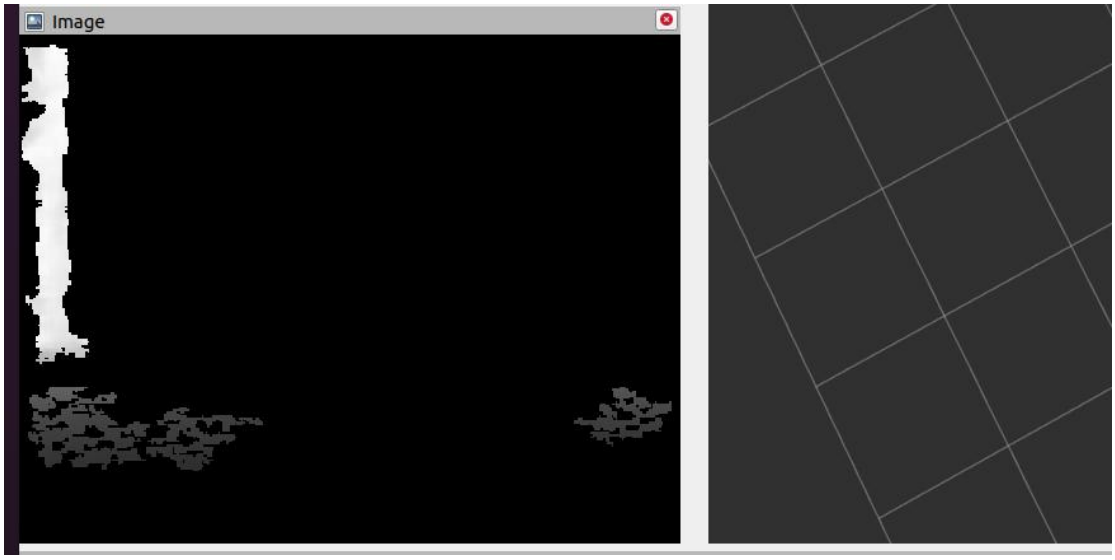




3.Adding depth image display

Follow the steps below to add the display of the depth image in RViz:





Image

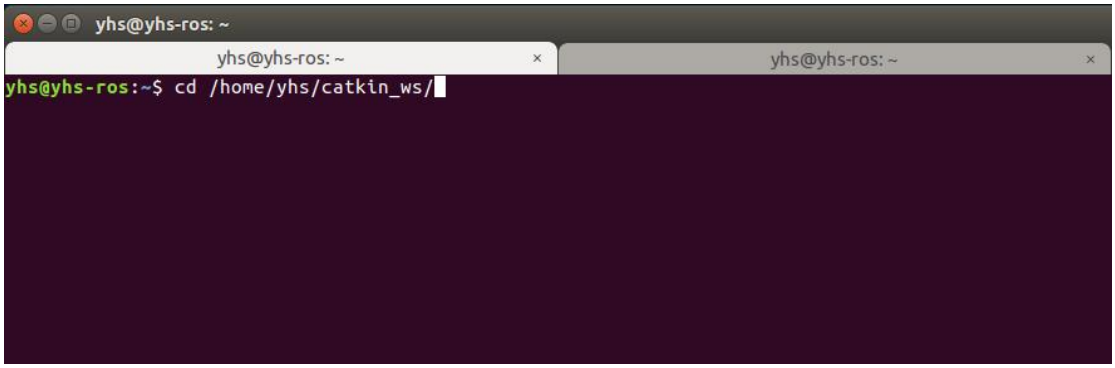
Experiment 14: Color Image from RGB-D Camera

In the previous experiment, we experienced the effect of camera imaging. In this experiment, we will learn how to retrieve color images captured by the camera in code.

1. Create a ROS source code package.

(1) Open a terminal and enter the following command to navigate to the ROS workspace:

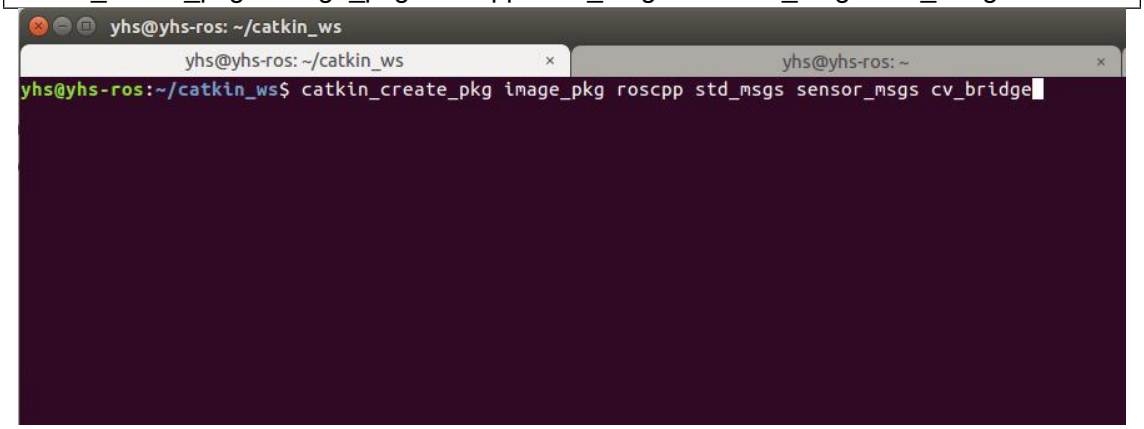
```
cd  /home/yhs/catkin_ws/
```



Press Enter to ROS workspace:

(2)Type the following command to create new ROS source package:

```
catkin_create_pkg  image_pkg  roscpp  std_msgs  sensor_msgs  cv_bridge
```

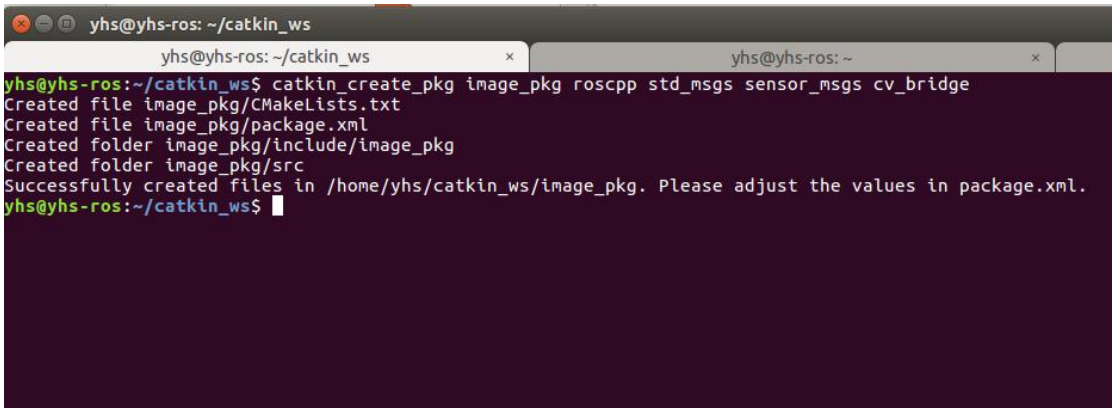


The meaning of this command:

Command	Meaning
catkin_create_pkg	To create a ROS source code package
image_pkg	This is the name you choose for the newly created ROS source code package. You can replace it with your desired package name.
roscpp	This dependency is required because this example code is written in C++.

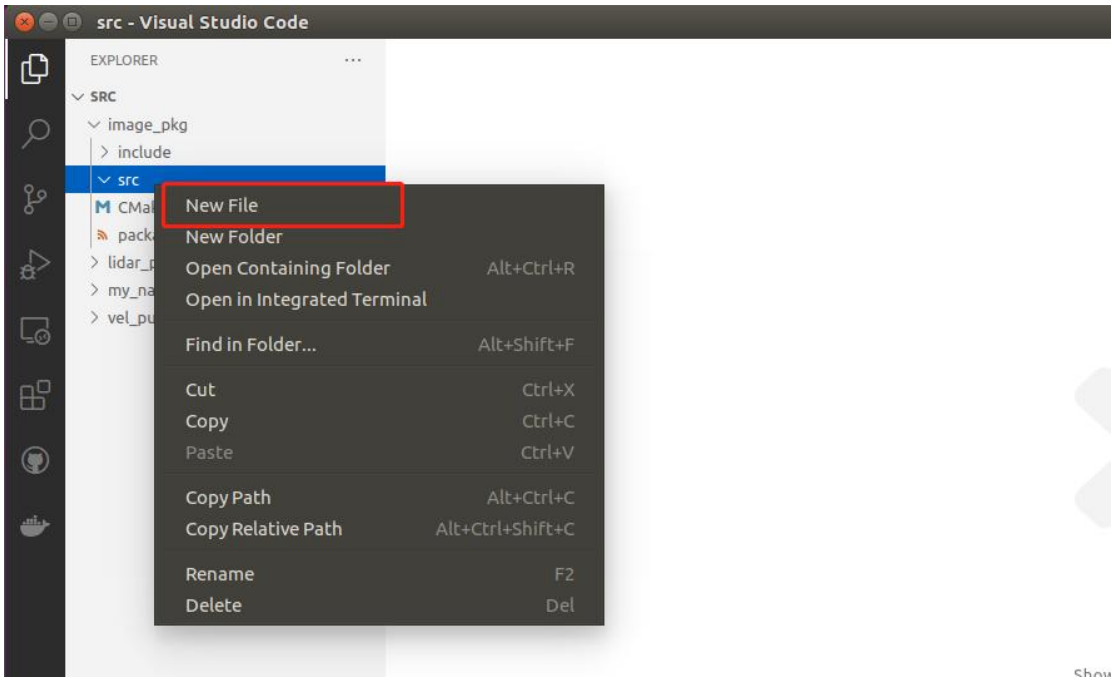
std_msgs	This dependency provides the standard message types, including the String format used for text output.
sensor_msgs	This dependency provides the message types for sensor data, including image data formats.
cv_bridge	This dependency is used for converting image formats between ROS and OpenCV.

After pressing Enter, you will see the following information, indicating that the new ROS package has been successfully created:

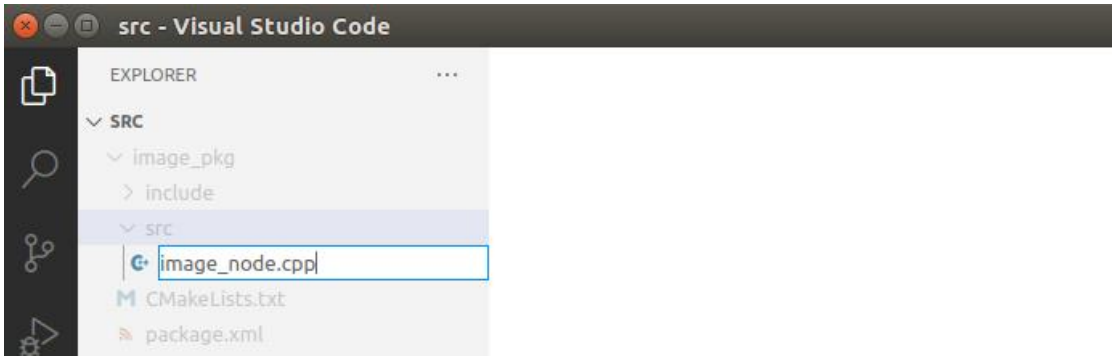


2. Editing in the IDE:

(1) You will notice that a new folder named image_pkg has been added to your workspace. Right-click on the src subfolder of image_pkg and select "New File" to create a new code file.



Name the newly created code file as "image_node.cpp".



(2) After naming the file, you can start writing the code for image_node.cpp on the right side of the IDE. The content of the file is as follows:

```
#include <ros/ros.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

bool bCaptrueOneFrame = true;

void callbackRGB(const sensor_msgs::ImageConstPtr& msg)
{
    if(bCaptrueOneFrame == true)
    {
        cv_bridge::CvImagePtr cv_ptr;
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
        imwrite("/home/yhs/1.jpg",cv_ptr->image);
        ROS_WARN("captrue image");
        bCaptrueOneFrame = false;
    }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_node");
    ROS_WARN("image_node start");
    ros::NodeHandle nh;
    ros::Subscriber rgb_sub =
nh.subscribe("/ascamera_hp60c_ln_1/rgb/image0",1,callbackRGB);
    ros::spin();
    return 0;
}
```

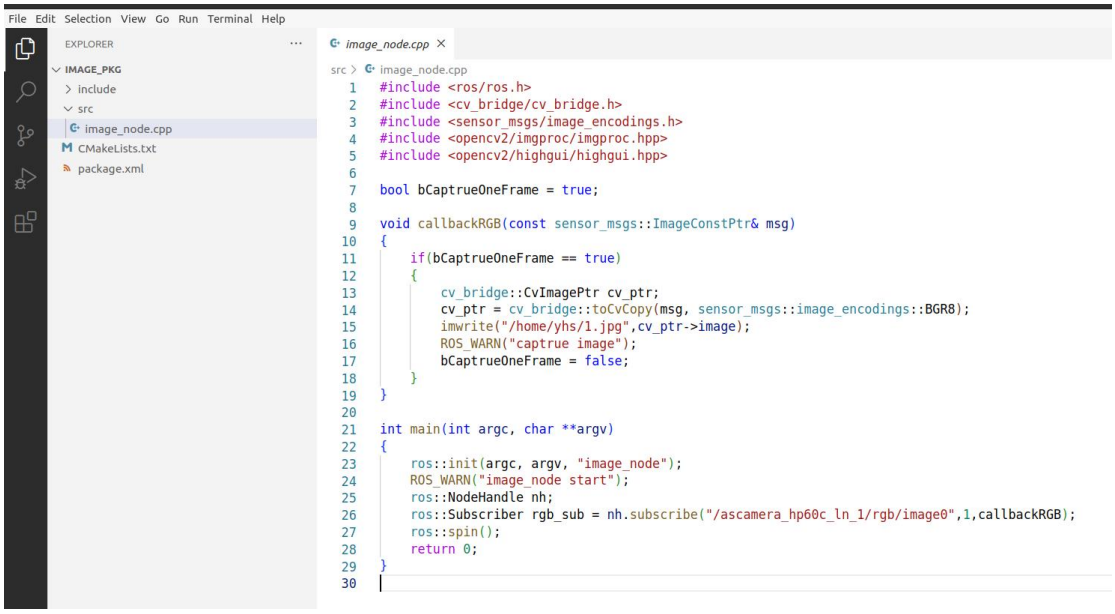
1) The code begins by including five header files: ros.h is the system header file for ROS, cv_bridge.h is the header file needed for converting ROS image data to OpenCV

image data, image_encodings.h is the header file for color image formats, imgproc.hpp is the header file for image processing functions in OpenCV, and highgui.hpp is the header file for image display and saving functions in OpenCV.。

2) Next, the program defines a boolean variable bCaptrueOneFrame to indicate whether to capture and save a single frame image. This variable is initialized to true, and after saving a frame image, it is set to false to ensure that only the first frame is saved and subsequent frames are not overwritten by frequent saving operations.
3) The program defines a callback function named void callbackRGB() to handle color image data. ROS automatically calls this callback function each time a frame of color image is received. The color image data is passed to this callback function as a parameter.
4) The parameter msg of the callback function void callbackRGB() is a pointer to sensor_msgs::Image format, which represents the memory area where the color image is stored. In practical development, the image is usually converted to OpenCV format for further processing, enabling the use of rich OpenCV functions for image manipulation.
5) Inside the callback function void callbackRGB(), the value of bCaptrueOneFrame is checked. If it is true, the image saving operation is performed, and bCaptrueOneFrame is set to false. This ensures that the image is saved only once, and subsequent invocations of this callback function will skip the image saving operation due to the value of bCaptrueOneFrame being false.
6) To save the image data, it needs to be converted to the OpenCV format. A pointer cv_ptr of type cv_bridge::CvImagePtr is defined, and the function cv_bridge::toCvCopy is called to convert the image data from msg to the OpenCV format. The converted image data in the OpenCV format is stored in the memory space pointed to by cv_ptr, named cv_ptr->image, which is of type cv::Mat. We can use the imwrite() function to save this image as a file, placed in the home directory of Ubuntu with the file name "1.jpg". Note that "robot" in "/home/yhs/1.jpg" represents the current username on Ubuntu. If the username is not "yhs", it should be replaced with the actual username, all lowercase. After completing the image operation, the ROS_WARN() function is called to display the save message "capture image" in the terminal program.
7) In the main function main(), ros::init() is called to initialize the node.
8) ROS_WARN() is used to output a string message to the terminal program, indicating that the node has started successfully.
9) A ros::NodeHandle node handle nh is defined in the main function, and this handle is used to subscribe to the data of the topic "/ascamera_hp60c_ln_1/rgb/image0" from the ROS core node. The callback function is set to the previously defined callbackRGB(). "/ascamera_hp60c_ln_1/rgb/image0" is the topic name where the camera's ROS node publishes color data. The camera captures color images and sends them in the form of ROS image data messages to this topic. Our own node, image_node, only needs to subscribe to this topic to receive the captured image data from the camera.

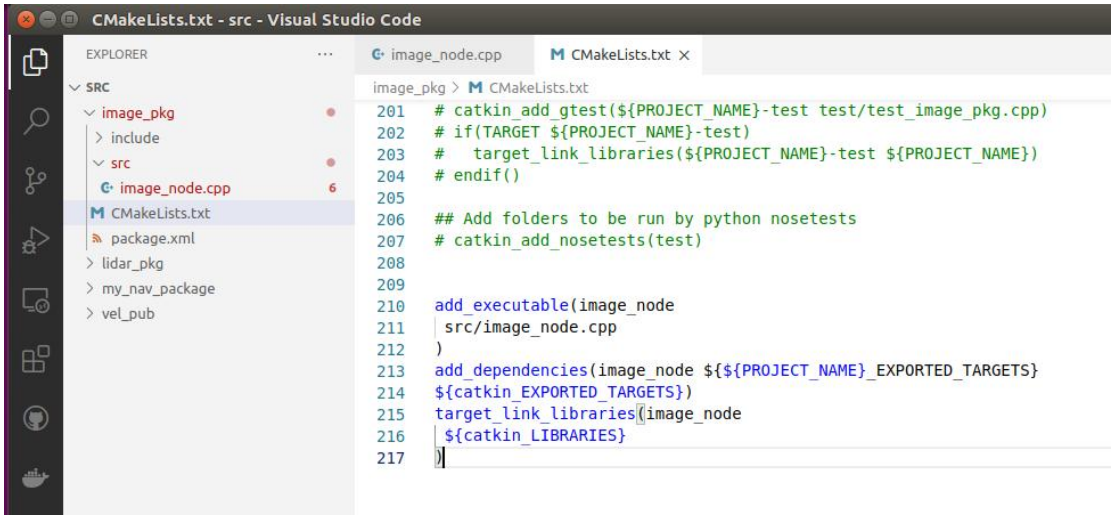
10) ros::spin() is called to block the main function and keep the node program from exiting.。

(3) After completing the code writing, press the keyboard shortcut Ctrl+S to save the file. The small black dot on the right side of the file name above the code will change to "X", indicating that the file has been successfully saved.



(4) After completing the code writing, the file name needs to be added to the compilation file in order to compile it. The compilation file is located in the directory of image_pkg and is named "CMakeLists.txt". In the IDE interface, click on that file on the left side, and the content of the file will be displayed on the right side. At the end of the "CMakeLists.txt" file, add a new compilation rule for image_node.cpp. The content is as follows:

```
add_executable(image_node
 src/image_node.cpp
)
add_dependencies(image_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(image_node
 ${catkin_LIBRARIES}
)
```

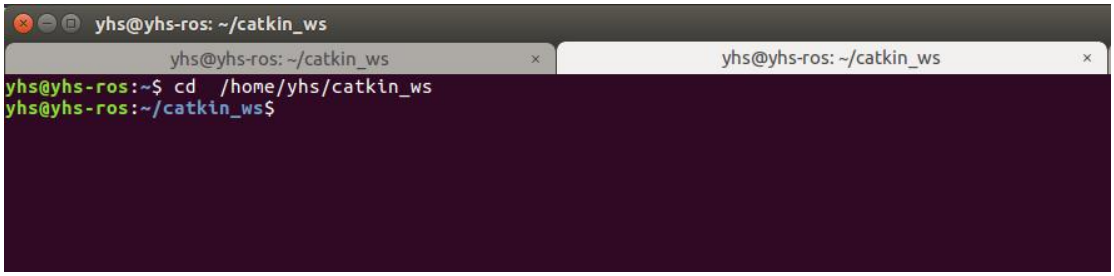


Similarly, after making the modifications, press the keyboard shortcut Ctrl+S to save the changes. The small black dot on the right side of the file name above the code will change to "X", indicating that the file has been successfully saved.

3. Compilation

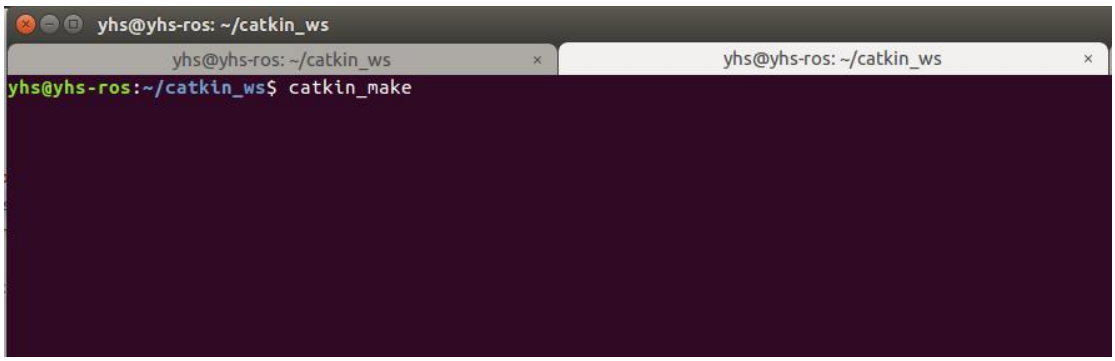
(1) Launch a terminal program and enter the following command to access the ROS workspace.

```
cd  /home/yhs/catkin_ws
```



(2) Execute the following command to start the compilation.

```
catkin_make
```



After executing this command, you will see scrolling compilation information until you encounter the message "Built target image_node". This indicates that the new image_node node has been successfully compiled.

```
[ 82%] Linking CXX executable /home/yhs/catkin_ws/devel/lib/image_pkg/image_node
[ 83%] Built target cartographer_rosbag_validate
[ 84%] Built target cartographer_pbstream_to_ros_map
[ 84%] Built target cartographer_dev_trajectory_comparison
[ 84%] Built target cartographer_rviz
[ 84%] Built target image_node
```

4. Run the node

(1) Open a terminal, enter the command, and press Enter to start the camera node (if it is already running, there is no need to restart it).

roslaunch ascamera_listener hp60c.launch

```
yhs@yhs-ros:~$ roslaunch  ascamera_listener  hp60c.launch
```

```
2023-11-14 14:32:20[INFO] [CameraPublisher.cpp] [1174] [setResolution] set depth resolution: 320 x 240 @ 15fps
2023-11-14 14:32:20[INFO] [CameraPublisher.cpp] [1188] [setResolution] set rgb resolution: 640 x 480 @ 15fps
2023-11-14 14:32:22[INFO] [CameraHp60c.cpp] [250] [startStreaming] start streaming
2023-11-14 14:32:22[INFO] [CameraSrv.cpp] [175] [onAttached] attached end
2023-11-14 14:32:22[INFO] [CameraHp60c.cpp] [899] [setInternalParameter] mjpeg info: size(640x480)
2023-11-14 14:32:23[INFO] [Camera.cpp] [120] [backgroundThread] SN [ ASC60CD21000190 ]'s parameter:
2023-11-14 14:32:23[INFO] [Camera.cpp] [121] [backgroundThread] irfx: 214.343
2023-11-14 14:32:23[INFO] [Camera.cpp] [122] [backgroundThread] irfy: 214.254
2023-11-14 14:32:23[INFO] [Camera.cpp] [123] [backgroundThread] ircx: 157.263
2023-11-14 14:32:23[INFO] [Camera.cpp] [124] [backgroundThread] ircy: 119.298
2023-11-14 14:32:23[INFO] [Camera.cpp] [125] [backgroundThread] rgbfx: 287.19
2023-11-14 14:32:23[INFO] [Camera.cpp] [126] [backgroundThread] rgbfy: 287.027
2023-11-14 14:32:23[INFO] [Camera.cpp] [127] [backgroundThread] rgbcx: 161.588
2023-11-14 14:32:23[INFO] [Camera.cpp] [128] [backgroundThread] rgbcy: 119.314
```

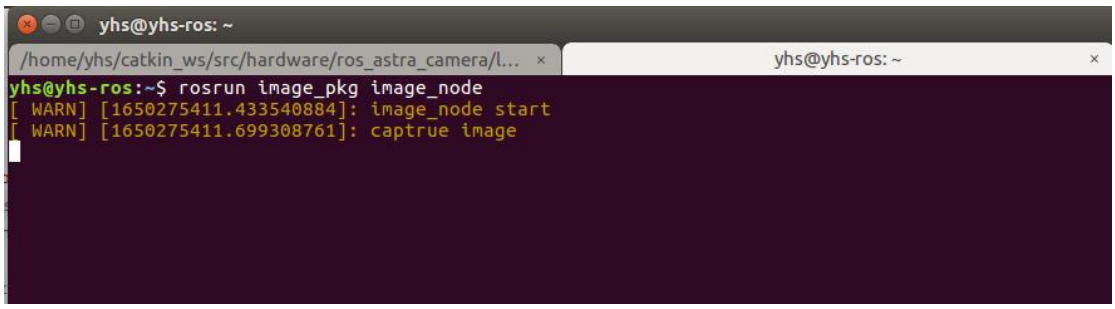
(1) Now we can run the image_node that we just wrote. Open another terminal, enter the command, and press Enter:

rosrun image_pkg image_node

```
yhs@yhs-ros: ~
/home/yhs/catkin_ws/src/hardware/ros_astra_camera/l...  ×   yhs@yhs-ros: ~  ×
yhs@yhs-ros:~$ rosrun image_pkg image_node
```

This command will start the image_node that we wrote. According to the program logic, it will continuously retrieve color image data packets from the camera's color image topic "/ascamera_hp60c_ln_1/rgb/image0". It will then convert the ROS-format image data into OpenCV format and save it as an image file named "1.jpg" in the home directory. In the

terminal, you will see the message "image_node start" indicating that the image_node has started successfully. After saving the image, it will display "capture image" in the terminal.



5. In Ubuntu, open the file manager, and you will find a file named "1.jpg" in the "Home" directory. Double-click to open it, and you will see the image captured by the image_node that we wrote, which is the color image data obtained from the camera and saved as an image.

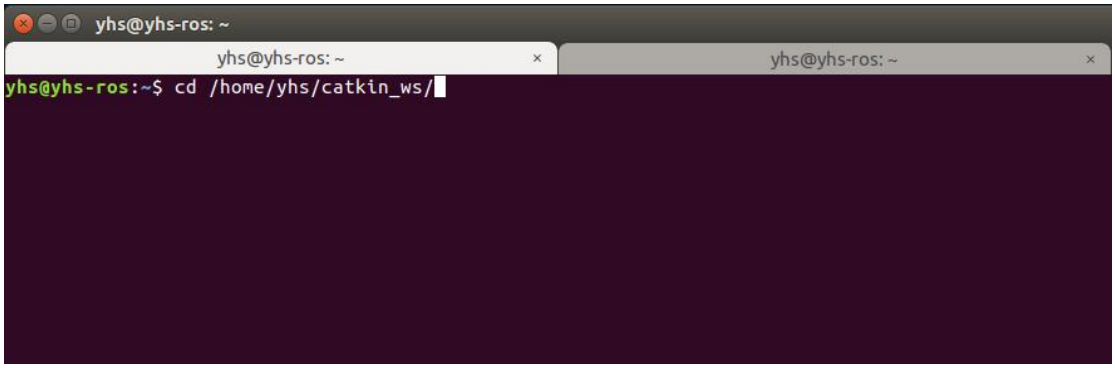
Experiment 15: 3D Point Cloud from RGB-D Camera

In the previous experiment, we have already obtained the color image captured by the camera. In this experiment, we will learn how to retrieve the 3D point cloud captured by the RGB-D camera in the code.

1. Create a ROS package

(1) Open a terminal and enter the command to ROS workspace.

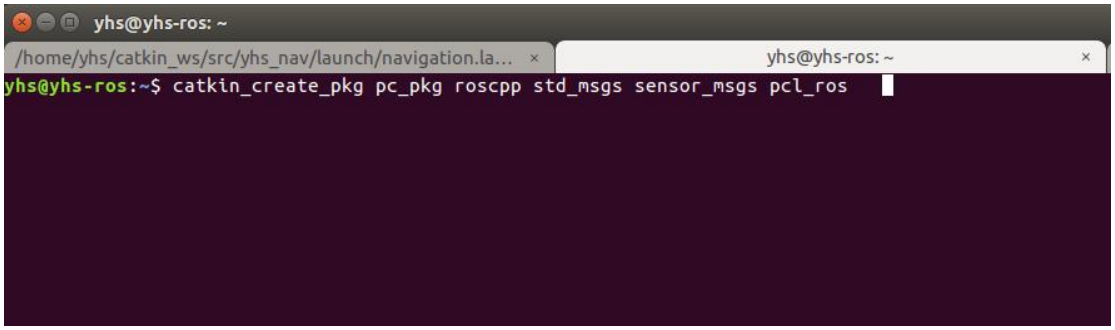
```
cd  /home/yhs/catkin_ws/
```



Press Enter to ROS workspace:

(2)Type the following command to create a new ROS source code package:

```
catkin_create_pkg  pc_pkg roscpp  std_msgs  sensor_msgs  pcl_ros
```

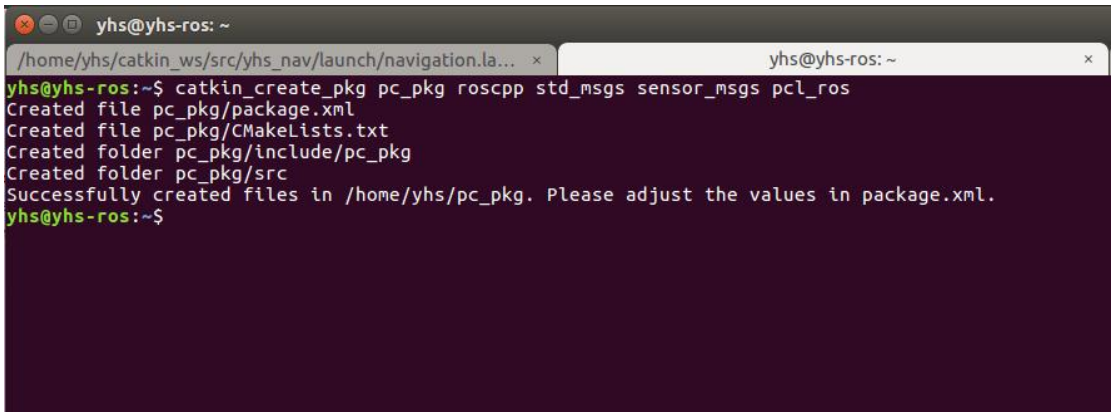


The meaning of this command:

Command	Meaning
catkin_create_pkg	To create a ROS source code package
pc_pkg	This is the name you choose for the newly created ROS source code package. You can replace it with your desired package name.
roscpp	This dependency is required because this example code is written in C++.
std_msgs	This dependency provides the standard message types,

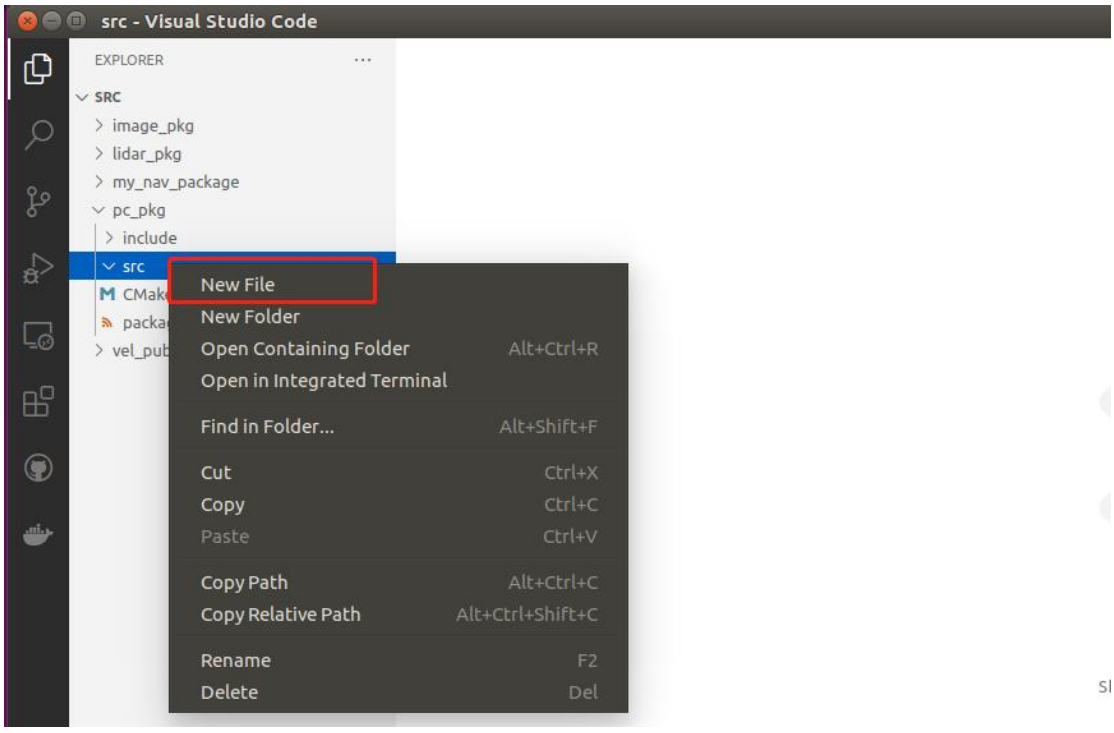
	including the String format used for text output.
sensor_msgs	This dependency provides the message types for sensor data, including image data formats.
pcl_ros	dependency for the open-source point cloud library (PCL) in ROS.

After pressing the Enter key, you will see the following information, indicating that the new ROS package has been successfully created.

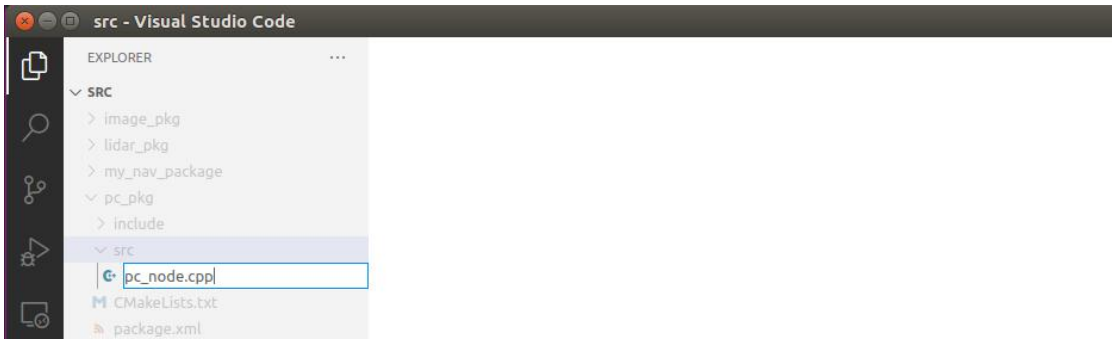


2. Editing in the IDE

(1) You will notice that a new folder named "pc_pkg" has been added to the workspace. Right-click on the "src" subfolder of the "pc_pkg" folder and select "New File" to create a new code file.



Name the newly created code file as "pc_node.cpp".



(2) After naming it, you can start writing the code for pc_node.cpp on the right side of the IDE. The content of the code is as follows:

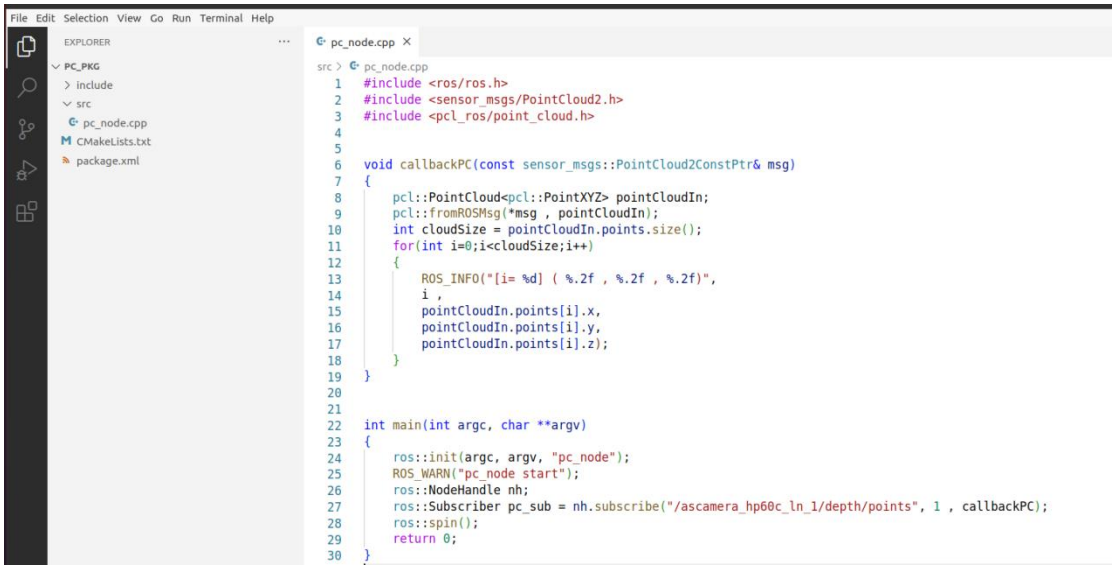
```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_ros/point_cloud.h>

void callbackPC(const sensor_msgs::PointCloud2ConstPtr& msg)
{
    pcl::PointCloud<pcl::PointXYZ> pointCloudIn;
    pcl::fromROSMsg(*msg , pointCloudIn);
    int cloudSize = pointCloudIn.points.size();
    for(int i=0;i<cloudSize;i++)
    {
        ROS_INFO("[i= %d] ( %.2f , %.2f , %.2f)",
        i ,
        pointCloudIn.points[i].x,
        pointCloudIn.points[i].y,
        pointCloudIn.points[i].z);
    }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "pc_node");
    ROS_WARN("pc_node start");
    ros::NodeHandle nh;
    ros::Subscriber pc_sub = nh.subscribe("/ascamera_hp60c_ln_1/depth/points", 1 ,
callbackPC);
    ros::spin();
    return 0;
}
```

1) At the beginning of the code, three header files are included: ros.h is the ROS system header file, sensor_msgs/PointCloud2.h is the header file for point cloud types in

ROS, and pcl_ros/point_cloud.h is the header file for point cloud types in PCL (Point Cloud Library).

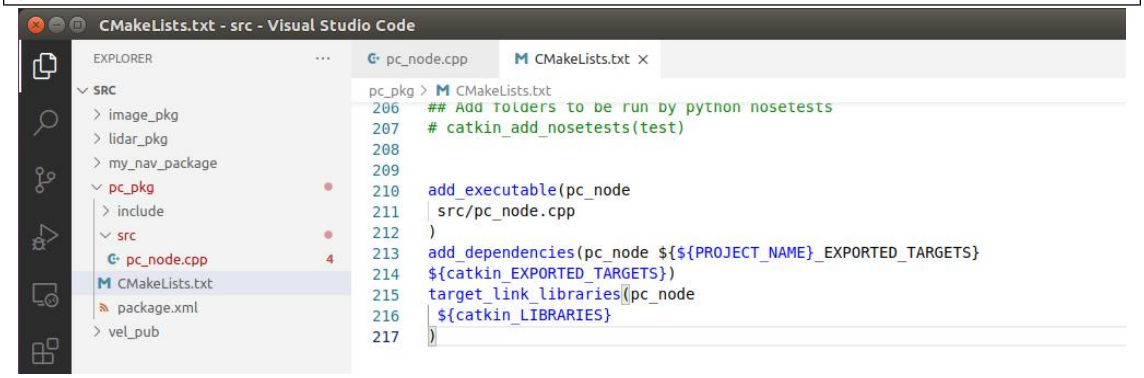
2) A callback function void callbackPC() is defined to handle the three-dimensional point cloud data. This callback function is automatically called by ROS whenever a depth image is received and converted into a three-dimensional point cloud. The three-dimensional point cloud data is passed as a parameter to this callback function.
3) The parameter msg of the callback function void callbackPC() is a pointer to the memory region storing the three-dimensional point cloud in the sensor_msgs::PointCloud2 format. In practical development, it is common to convert this point cloud to the PCL point cloud format, which allows the use of various PCL functions for point cloud data processing.
4) Within the callback function void callbackPC(), a point cloud container pcl::PointXYZ named pointCloudIn is defined. The pcl::fromROSMsg() function is called to convert the point cloud data in ROS format from the parameter into the PCL format, which is then stored in the pointCloudIn container.
5) The number of three-dimensional points in the converted point cloud array pointCloudIn.points is obtained and stored in a variable cloudSize. A for loop is used to display the x, y, and z values of all points in pointCloudIn.points using ROS_INFO(). Typically, the raw coordinate values in pointCloudIn.points are not used directly but need to be transformed to the robot coordinate system before being processed using PCL point cloud library functions.
6) In the main() function, ros::init() is called to initialize the node.
7) ROS_WARN() is called to output a string message to the terminal, indicating that the node has started successfully.
8) A ros::NodeHandle named nh is defined in the main() function. Using this handle, the node subscribes to the data of the topic "/ascamera_hp60c_ln_1/depth/points" published by the ROS core node. The callback function is set to the previously defined callbackPC(). This "/ascamera_hp60c_ln_1/depth/points" is the topic name where the camera's ROS node publishes the three-dimensional point cloud. The three-dimensional point cloud captured by the camera is sent to this topic in the form of ROS message packets. Our own node pc_node can receive the three-dimensional point cloud captured by the Kinect2 by subscribing to this topic.
9) ros::spin() is called to block the main() function, ensuring that the node program does not end or exit.

(4) After finishing writing the code, press the keyboard shortcut Ctrl+S to save the file. The small black dot on the right side of the file name above the code will turn into an "X," indicating that the file has been successfully saved.



(5) After finishing the code, you need to add the file name to the compilation file in order to compile it. The compilation file is located in the directory of pc_pkg and is named "CMakeLists.txt". Click on that file on the left side of the IDE, and the content of the file will be displayed on the right side. At the end of the "CMakeLists.txt" file, add a new compilation rule for pc_node.cpp. The content is as follows:

```
add_executable(pc_node
  src/pc_node.cpp
)
add_dependencies(pc_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(pc_node
  ${catkin_LIBRARIES}
)
```

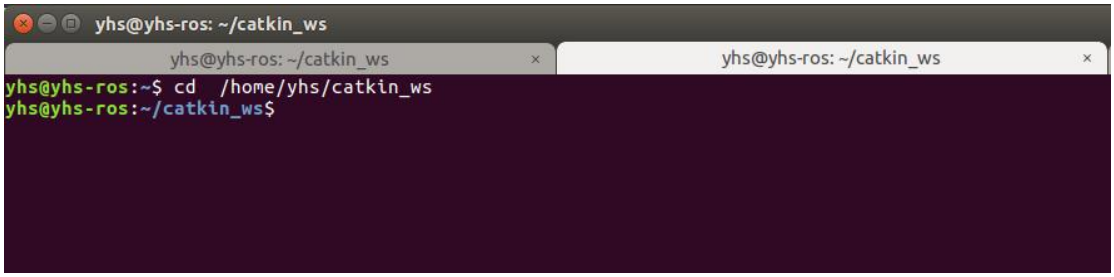


Similarly, after making the modifications, press the keyboard shortcut Ctrl+S to save the changes. The small black dot on the right side of the file name above the code will turn into an "X," indicating that the file has been successfully saved.

3. Compilation

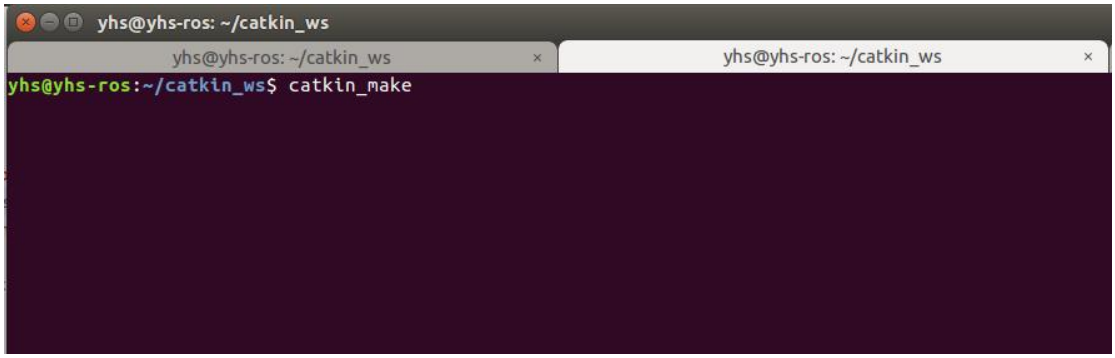
(1) Launch a terminal program and enter the following command to navigate to the ROS workspace:

```
cd  /home/yhs/catkin_ws
```

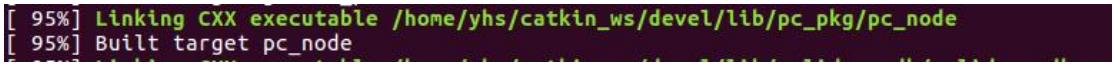


(2) Execute the following command to start the compilation:

```
catkin_make
```



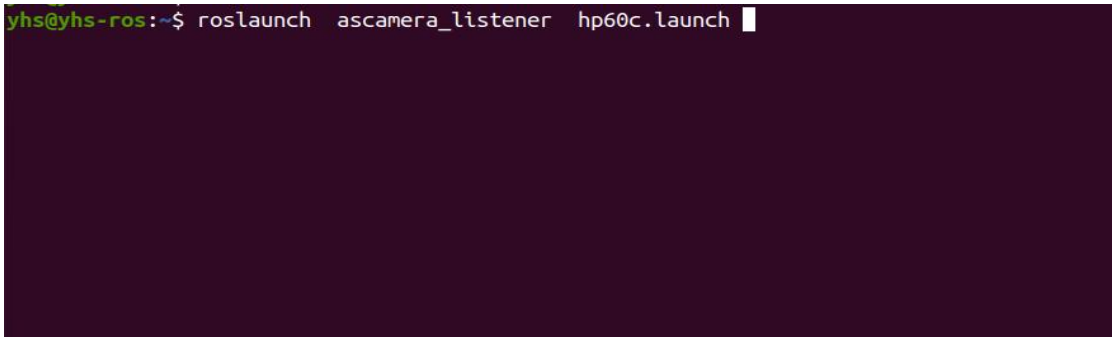
After executing this command, you will see scrolling compilation information until you see the message "Built target pc_node," indicating that the new pc_node node has been successfully compiled.



4. Run Node

(1) Open a terminal, enter the command, and press Enter to start the camera node (if it is already running, there is no need to start it again):

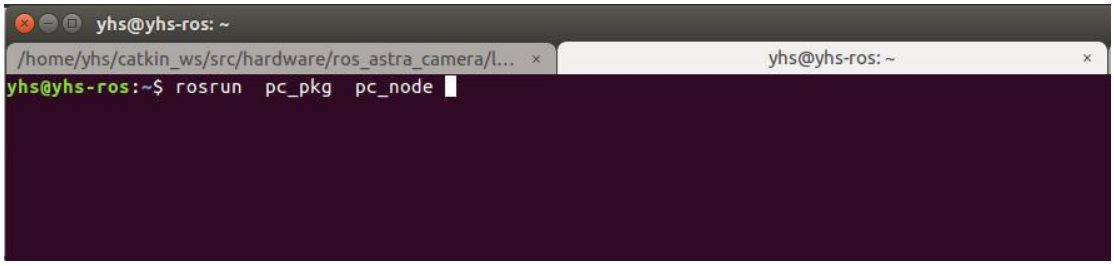
```
roslaunch  ascamera_listener  hp60c.launch
```



```
2023-11-14 14:32:20[INFO] [CameraPublisher.cpp] [1174] [setResolution] set depth resolution: 320 x 240 @ 15fps
2023-11-14 14:32:20[INFO] [CameraPublisher.cpp] [1188] [setResolution] set rgb resolution: 640 x 480 @ 15fps
2023-11-14 14:32:22[INFO] [CameraHp60c.cpp] [250] [startStreaming] start streaming
2023-11-14 14:32:22[INFO] [CameraSrv.cpp] [175] [onAttached] attached end
2023-11-14 14:32:22[INFO] [CameraHp60c.cpp] [899] [setInternalParameter] mjpeg info: size(640x480)
2023-11-14 14:32:23[INFO] [Camera.cpp] [120] [backgroundThread] SN [ ASC60CD21000190 ]'s parameter:
2023-11-14 14:32:23[INFO] [Camera.cpp] [121] [backgroundThread] irfx: 214.343
2023-11-14 14:32:23[INFO] [Camera.cpp] [122] [backgroundThread] irfy: 214.254
2023-11-14 14:32:23[INFO] [Camera.cpp] [123] [backgroundThread] ircx: 157.263
2023-11-14 14:32:23[INFO] [Camera.cpp] [124] [backgroundThread] ircy: 119.298
2023-11-14 14:32:23[INFO] [Camera.cpp] [125] [backgroundThread] rgbfx: 287.19
2023-11-14 14:32:23[INFO] [Camera.cpp] [126] [backgroundThread] rgbfy: 287.027
2023-11-14 14:32:23[INFO] [Camera.cpp] [127] [backgroundThread] rgbcx: 161.588
2023-11-14 14:32:23[INFO] [Camera.cpp] [128] [backgroundThread] rgbcy: 119.314
```

(2) Now we can run the pc_node that we just wrote. Open another terminal, enter the command, and press Enter.

```
rosrun  pc_pkg  pc_node
```



This command will start the pc_node that we have written. Following the program logic, it will continuously retrieve point cloud data from the topic "/ascamera_hp60c_ln_1/depth/points" published by the camera. It will then convert the ROS-formatted three-dimensional point cloud into PCL format and display the x, y, and z coordinate values of all points in the terminal program.

```
[ INFO] [1650332104.351277242]: [i= 166753] ( 0.33 , 0.38 , 5.29)
[ INFO] [1650332104.351294097]: [i= 166754] ( 0.34 , 0.38 , 5.29)
[ INFO] [1650332104.351306993]: [i= 166755] ( 0.36 , 0.39 , 5.37)
[ INFO] [1650332104.351318623]: [i= 166756] ( 0.37 , 0.39 , 5.37)
[ INFO] [1650332104.351333737]: [i= 166757] ( 0.38 , 0.39 , 5.37)
[ INFO] [1650332104.351347978]: [i= 166758] ( 0.39 , 0.39 , 5.37)
[ INFO] [1650332104.351377028]: [i= 166759] ( 0.39 , 0.38 , 5.29)
[ INFO] [1650332104.351390744]: [i= 166760] ( 0.40 , 0.38 , 5.29)
[ INFO] [1650332104.351403879]: [i= 166761] ( 0.41 , 0.38 , 5.29)
[ INFO] [1650332104.351417198]: [i= 166762] ( 0.42 , 0.38 , 5.29)
[ INFO] [1650332104.351433817]: [i= 166763] ( 0.43 , 0.38 , 5.29)
[ INFO] [1650332104.351446754]: [i= 166764] ( 0.43 , 0.38 , 5.21)
[ INFO] [1650332104.351458177]: [i= 166765] ( 0.44 , 0.38 , 5.21)
[ INFO] [1650332104.351471088]: [i= 166766] ( 0.45 , 0.38 , 5.21)
[ INFO] [1650332104.351484760]: [i= 166767] ( 0.46 , 0.38 , 5.21)
[ INFO] [1650332104.351500110]: [i= 166768] ( 0.47 , 0.38 , 5.21)
[ INFO] [1650332104.351512947]: [i= 166769] ( 0.48 , 0.38 , 5.21)
[ INFO] [1650332104.351525847]: [i= 166770] ( 0.48 , 0.37 , 5.13)
[ INFO] [1650332104.351539336]: [i= 166771] ( 0.49 , 0.37 , 5.13)
```

In the terminal, you will see the message "pc_node start" indicating that pc_node has started successfully. After that, it will continuously refresh and display the coordinate values of all points in the point cloud.

Experiment 16: Incorporating Obstacle Layer with Camera LaserScan Data

Note: The following steps have already been configured, so users do not need to configure them again. These steps are provided for informational purposes only.

During robot navigation, 2D laser data is used for dynamic global and local path planning to avoid obstacles along the path. However, the scanning range of a 2D laser is limited to a plane, and it cannot scan areas that are higher or lower than its mounting position. Therefore, we can utilize point cloud data from a depth camera to perform obstacle avoidance during the navigation process.

1. Converting Point Cloud Data to LaserScan Data

(1) To convert point cloud data to LaserScan data, we use the "pointcloud_to_laserscan" package. Detailed information about this package can be found at "http://wiki.ros.org/pointcloud_to_laserscan". Let's take a look at the launch file used for running this package, located at:

```
/home/yhs/catkin_ws/src/hardware/ascamera_listener/launch/pointcloud_to_laserscan.launch
```

The detailed contents as below:

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

<launch>

    <node pkg="pointcloud_to_laserscan" type="pointcloud_to_laserscan_node" name="pointcloud_to_laserscan1">

        <remap from="cloud_in" to="/ascamera_hp60c_ln_1/depth/points"/>
        <remap from="scan" to="scan1"/>
        <rosparam>
            target_frame: ascamera_hp60c_ln_1 # Leave disabled to output scan in pointcloud frame
            transform_tolerance: 0.001
            min_height: -0.1
            max_height: 1.5

            angle_min: -3.1415926
            angle_max: 3.1415926
            angle_increment: 0.006
            scan_time: 0.1
            range_min: 0.1
            range_max: 2.0
            use_inf: false
```

```
            inf_epsilon: 1.0

            concurrency_level: 1
        </rosparam>
    </node>
</launch>
```

Here are the parts that we may need to modify:

/ascamera_hp60c_ln_1/depth/points	which is the topic where the camera publishes point cloud data. If you have a different topic, you can modify this parameter accordingly.
scan1	The converted LaserScan data will be published on the topic "scan1". Since the 2D laser topic is usually named "scan," it's important to avoid using the same topic name. Here, it has been changed to "scan1".
min_height	This parameter specifies the minimum height for truncating the point cloud data. If the minimum height is set too low, it may include ground points in the converted LaserScan data, affecting path planning. You can modify this value and observe the results repeatedly until you find a suitable value.
max_height	This parameter specifies the maximum height for truncating the point cloud data. If the maximum height is set too high, it may include points above door frames in the converted LaserScan data, affecting path planning. You can modify this value and observe the results repeatedly until you find a suitable value.

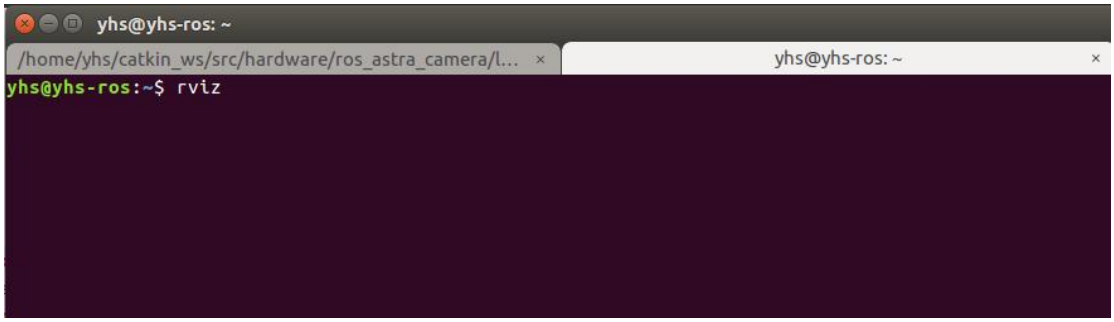
(2) Since we started the camera launch file along with the "pointcloud_to_laserscan" node, open a terminal and enter the command, then press Enter:

```
roslaunch ascamera_listener hp60c.launch
```

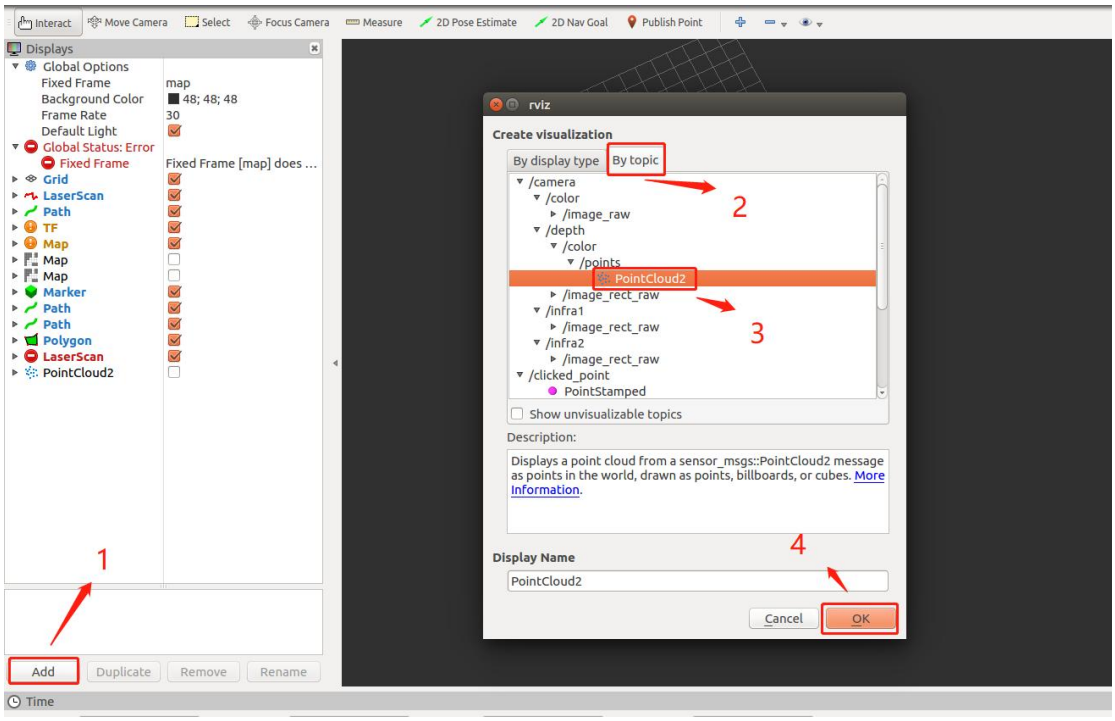
```
2023-11-14 14:32:20[INFO] [CameraPublisher.cpp] [1174] [setResolution] set depth resolution: 320 x 240 @ 15fps
2023-11-14 14:32:20[INFO] [CameraPublisher.cpp] [1188] [setResolution] set rgb resolution: 640 x 480 @ 15fps
2023-11-14 14:32:22[INFO] [CameraHp60c.cpp] [250] [startStreaming] start streaming
2023-11-14 14:32:22[INFO] [CameraSrv.cpp] [175] [onAttached] attached end
2023-11-14 14:32:22[INFO] [CameraHp60c.cpp] [899] [setInternalParameter] mjpeg info: size(640x480)
2023-11-14 14:32:23[INFO] [Camera.cpp] [120] [backgroundThread] SN [ ASC60CD21000190 ]'s parameter:
2023-11-14 14:32:23[INFO] [Camera.cpp] [121] [backgroundThread] irfx: 214.343
2023-11-14 14:32:23[INFO] [Camera.cpp] [122] [backgroundThread] irfy: 214.254
2023-11-14 14:32:23[INFO] [Camera.cpp] [123] [backgroundThread] ircx: 157.263
2023-11-14 14:32:23[INFO] [Camera.cpp] [124] [backgroundThread] ircy: 119.298
2023-11-14 14:32:23[INFO] [Camera.cpp] [125] [backgroundThread] rgbfx: 287.19
2023-11-14 14:32:23[INFO] [Camera.cpp] [126] [backgroundThread] rgbfy: 287.027
2023-11-14 14:32:23[INFO] [Camera.cpp] [127] [backgroundThread] rgbcx: 161.588
2023-11-14 14:32:23[INFO] [Camera.cpp] [128] [backgroundThread] rgbcy: 119.314
```

(3) Open another terminal, enter the command, and press Enter:

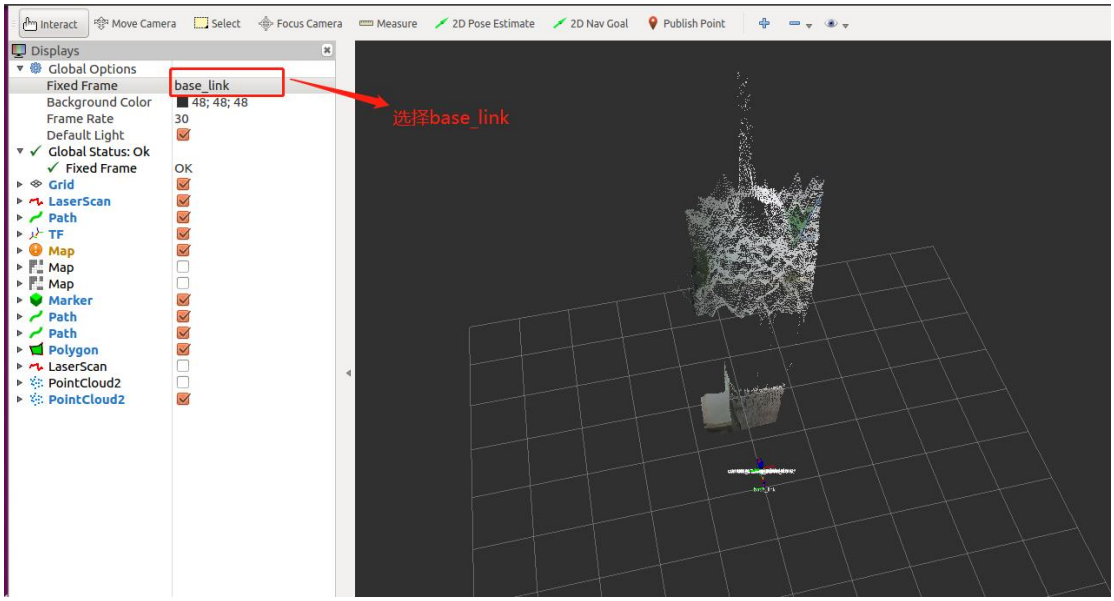
rviz



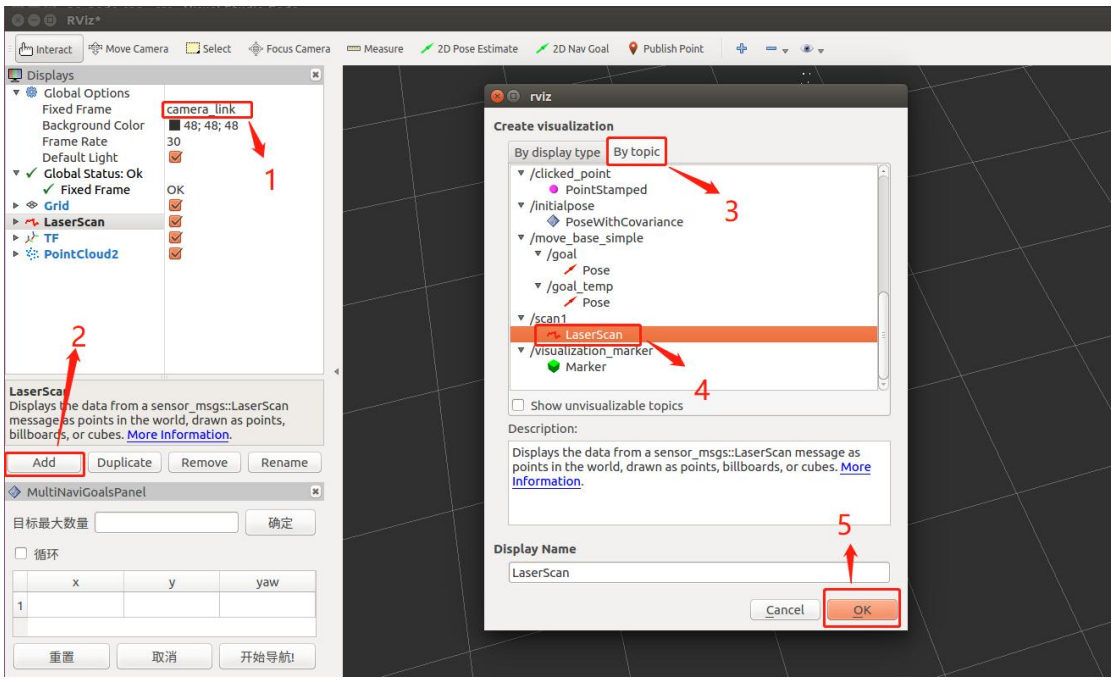
After opening Rviz, follow the steps below to add the display of camera point cloud data: **(Note: After adding, make sure to select the corresponding topic in the "Topic" field to display the data. The camera topic may vary, so the following steps are for reference only. You can customize them based on your specific topic.)**

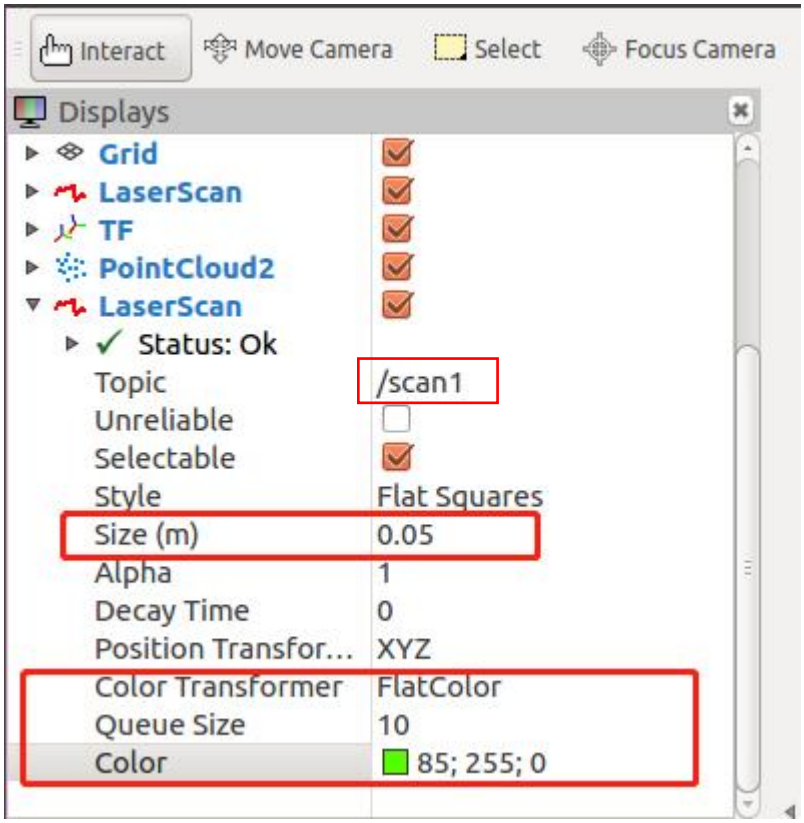


The white area represents the point cloud data.

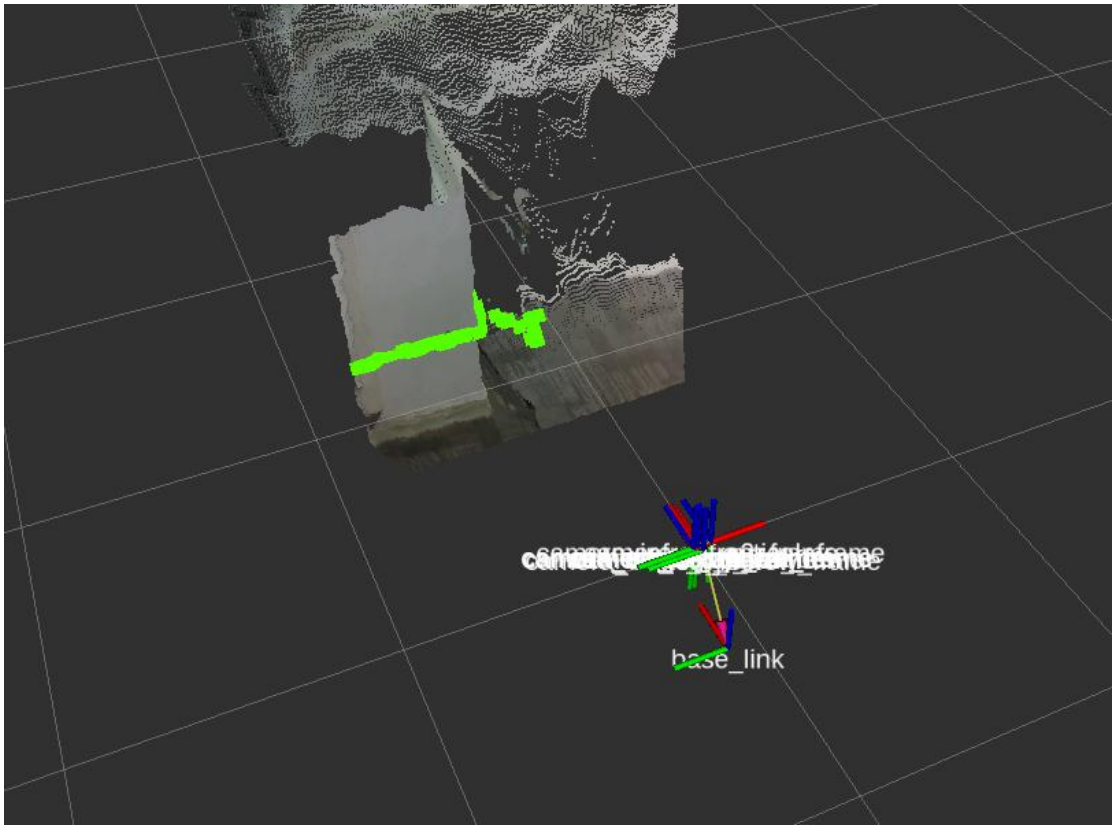


To add the display of camera LaserScan data, follow the steps below:





The green points represent the data obtained after converting the point cloud to LaserScan.



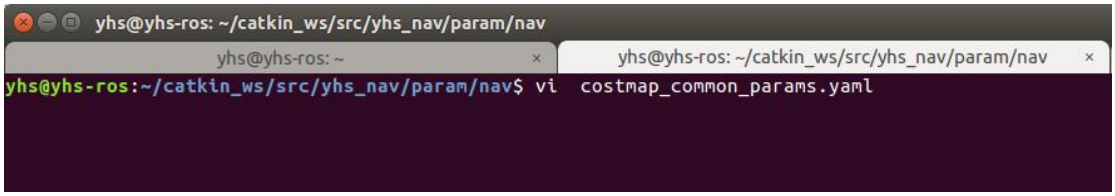
2. Adding LaserScan Data to the obstacle layer

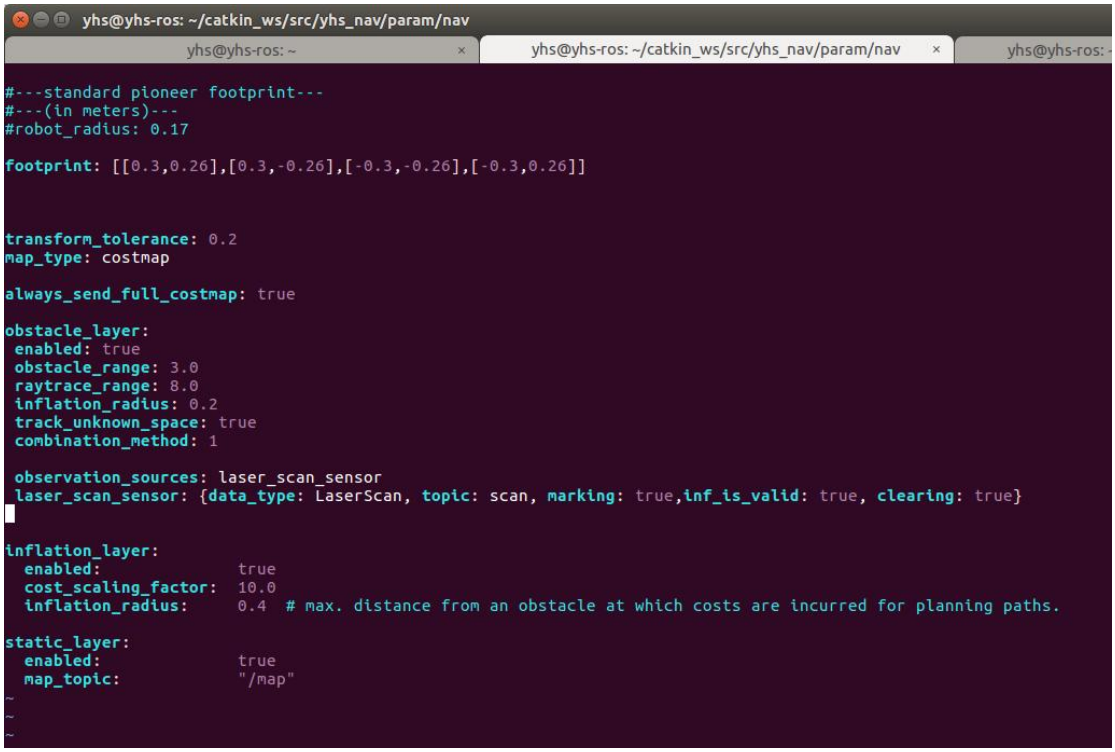
(1) Open a terminal and enter the command, then press Enter:

```
roscd  yhs_nav/param/nav/
```

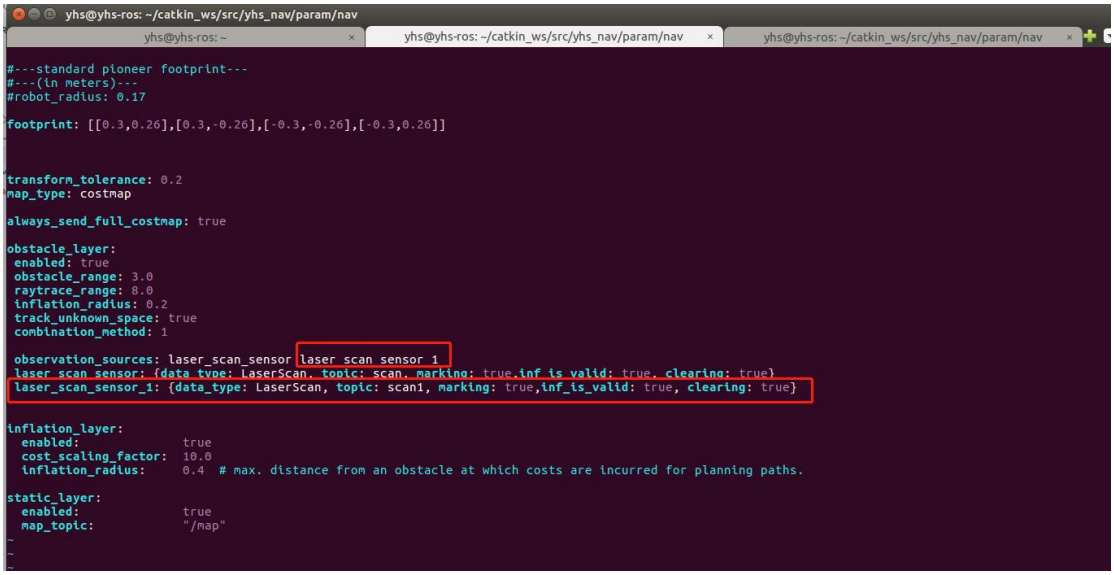
(2) Type the following command and press Enter:

```
vi  costmap_common_params.yaml
```





(2) The contents to add after obtaining camera LaserScan data are as follows:



laser_scan_sensor_1	Name of the obstacle source, which should be different from the previous one.
scan1	LaserScan topic, where "scan1" is the topic on which the camera's point cloud data is converted and published as LaserScan.

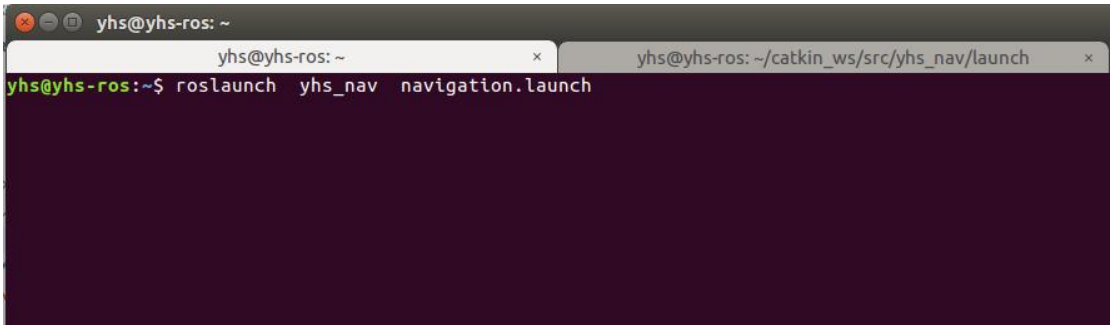
3. Starting Navigation

(1) Close all open programs. Open a terminal and start the NDT localization by entering the following command:

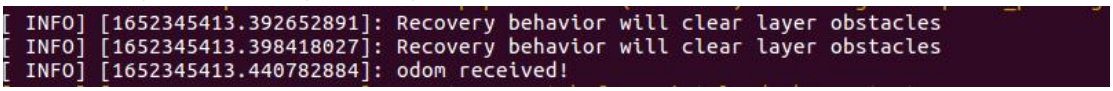
```
Roslaunch ndt_localizer ndt_localizer.launch
```

(2) Type the following command and press Enter to start navigation:

```
roslaunch yhs_nav navigation.launch
```



After seeing the following message, it indicates a successful startup:

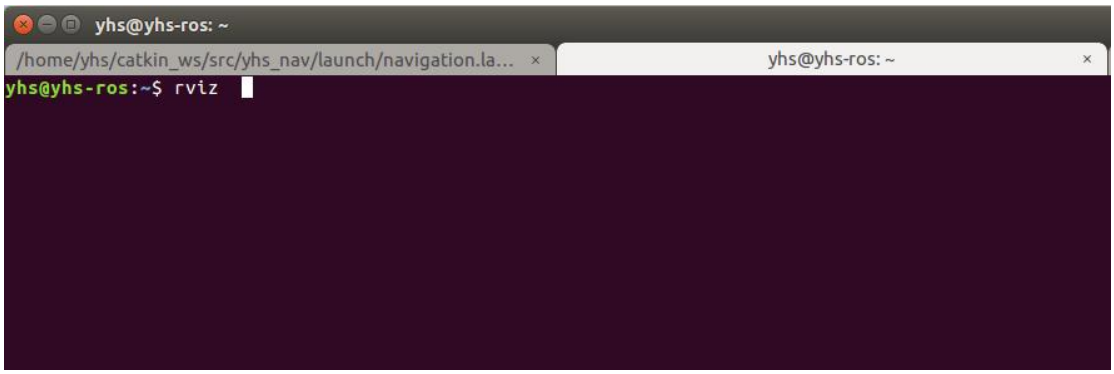


Alternatively, start 2D navigation.

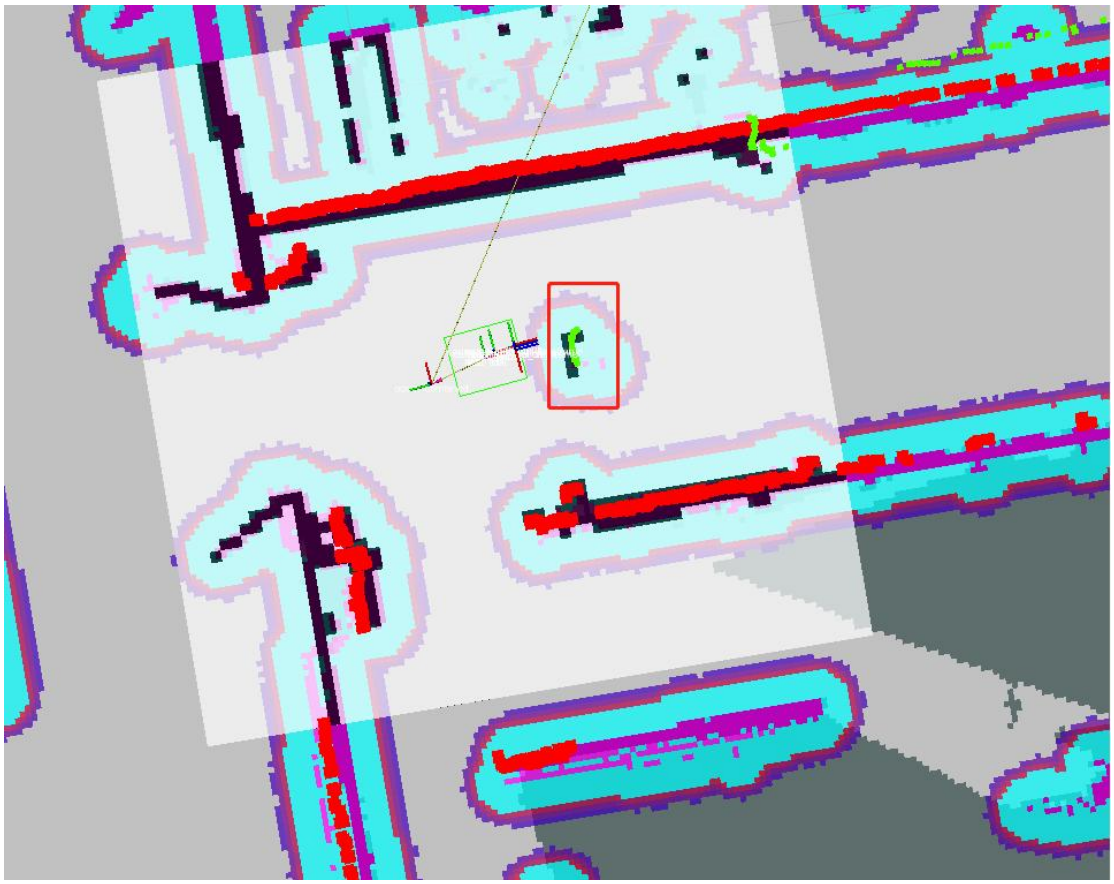
```
roslaunch yhs_nav navigation_2d.launch
```

Open another terminal, type the following command and press Enter:

```
rviz
```



As shown in the image below, the laser is mounted higher than the camera. The camera can detect relatively low objects, while the laser cannot.



4. If you have additional LaserScan data, you can add it following the same procedure mentioned above. To remove the added data, simply delete or comment out the corresponding content.

Appendix 1: Remote Control

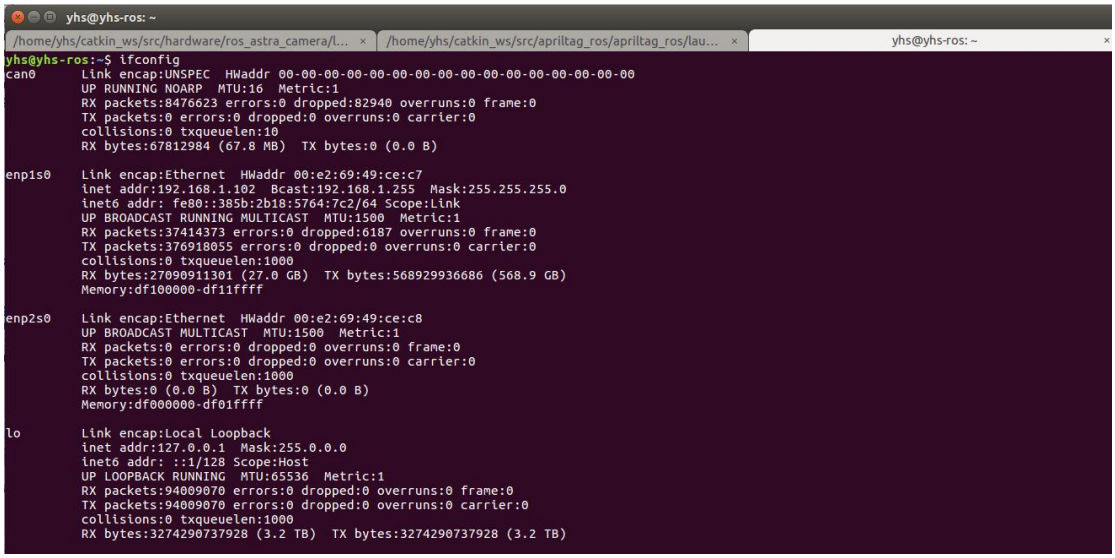
When there is no screen installed on the robot, we can use another computer to remotely control the robot by opening the Rviz interface.

1. Robot Host Setup **(Robot host setup, here we are just demonstrating the steps, assuming the ROS robot is already configured and doesn't require any modifications)**

(1) Temporarily connect a screen to the robot and connect the keyboard and mouse to the USB ports of the industrial computer.

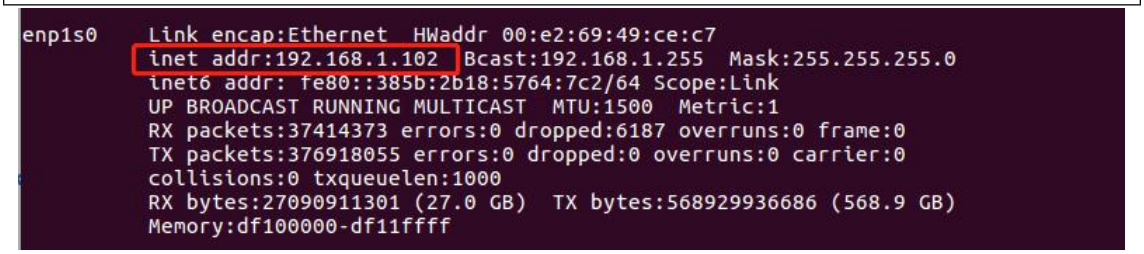
(2) Open a terminal and enter the command, then press Enter:

```
ifconfig
```



You can see the IP address of the robot itself as:

```
192.168.1.102
```



(3) Type the following command and press Enter:

```
vi /home/yhs/.bashrc
```

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
source /opt/ros/kinetic/setup.bash
source /home/yhs/catkin_ws/devel/setup.bash

export ROS_HOSTNAME=192.168.1.102
export ROS_MASTER_URI=http://192.168.1.102:11311
```

Locate to the end of the file and add the following two lines:

```
export ROS_HOSTNAME=192.168.1.102
export ROS_MASTER_URI=http://192.168.1.102:11311
```

"192.168.1.102" is the IP address of the industrial computer. If it is not "192.168.1.102", please change it to the actual IP address. After making the changes, save and exit the file, and then restart the industrial computer.

2. Computer Slave Setup **(Here, the user needs to configure their own laptop)**）

(1) Ensure that your computer is running Ubuntu and has ROS installed. Connect your computer to the same Wi-Fi network as the robot's router.

(2) Open a terminal and enter the command, then press Enter:

```
ifconfig
```

```
wlp109s0  Link encap:Ethernet  HWaddr 34:7d:f6:c3:e4:cf
          inet addr:192.168.1.10  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::bc98:7ecc:c4b7:8210/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:362908421 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13910281 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:548667728055 (548.6 GB)  TX bytes:1308214955 (1.3 GB)
```

As shown in the image above, the IP address of the computer slave is "192.168.1.10".

(3) Open a terminal on the computer slave and enter the command, then press Enter:

```
export ROS_HOSTNAME=192.168.1.10
export ROS_MASTER_URI=http://192.168.1.102:11311
```

"192.168.1.10" is the IP address of the computer slave, and "192.168.1.102" is the IP address of the robot's industrial computer. If the addresses have changed, please enter the actual IP addresses. Alternatively, you can add these two lines of code at the bottom of the .bashrc file, following the steps for the host configuration. Then, run the command

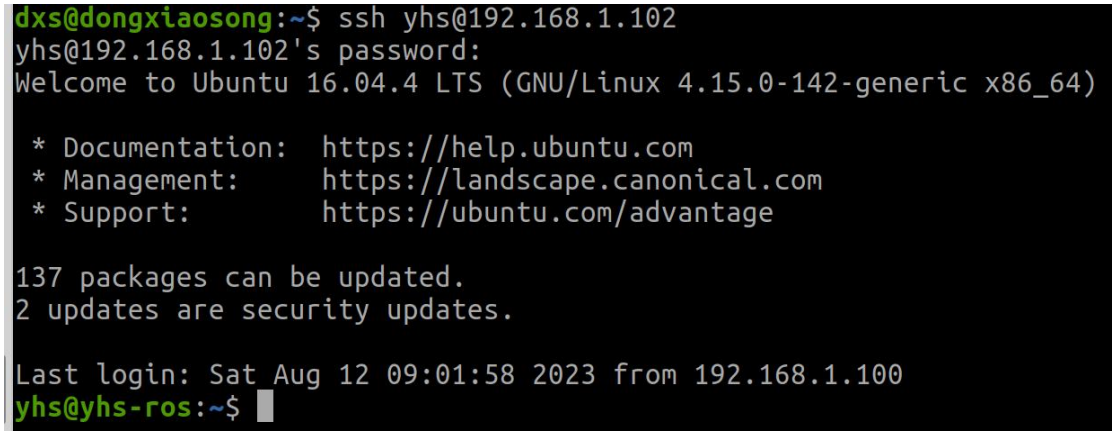
"source ~/.bashrc". Please refer to the image below for better understanding.

3.SSH Remote Control

After configuring the laptop slave according to the above steps, you can proceed with SSH remote control. Open a terminal and enter the following command:

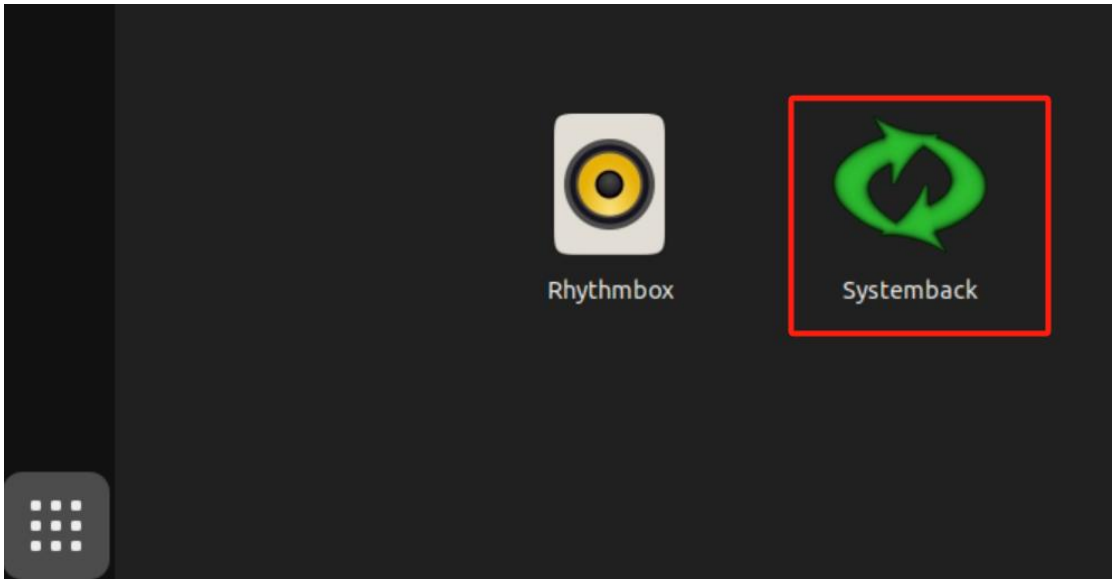
```
ssh yhs@192.168.1.102
```

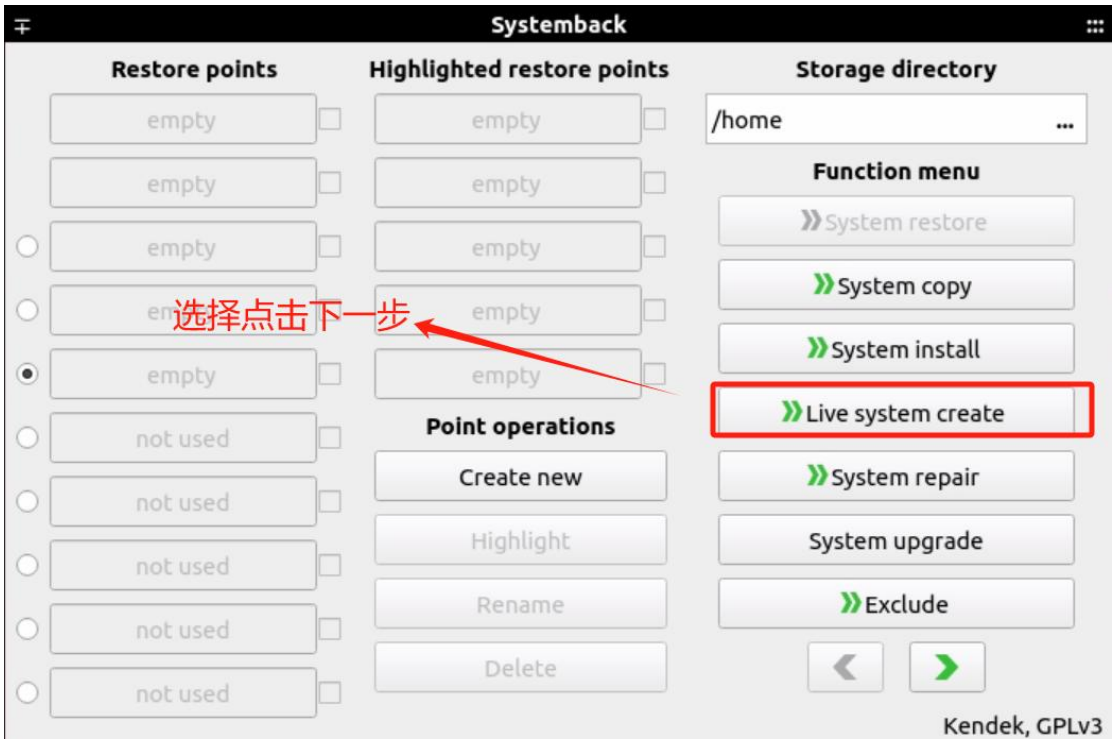
Press Enter, and then enter the password. If the following image appears, it indicates a successful remote connection. If your Ubuntu system does not have SSH installed, you can search online for instructions on how to install it. "yhs" is the username of the host, which is the ROS robot, and "192.168.1.102" is the IP address of the host. **(Note: When entering the password, it will not be displayed on the screen.)**



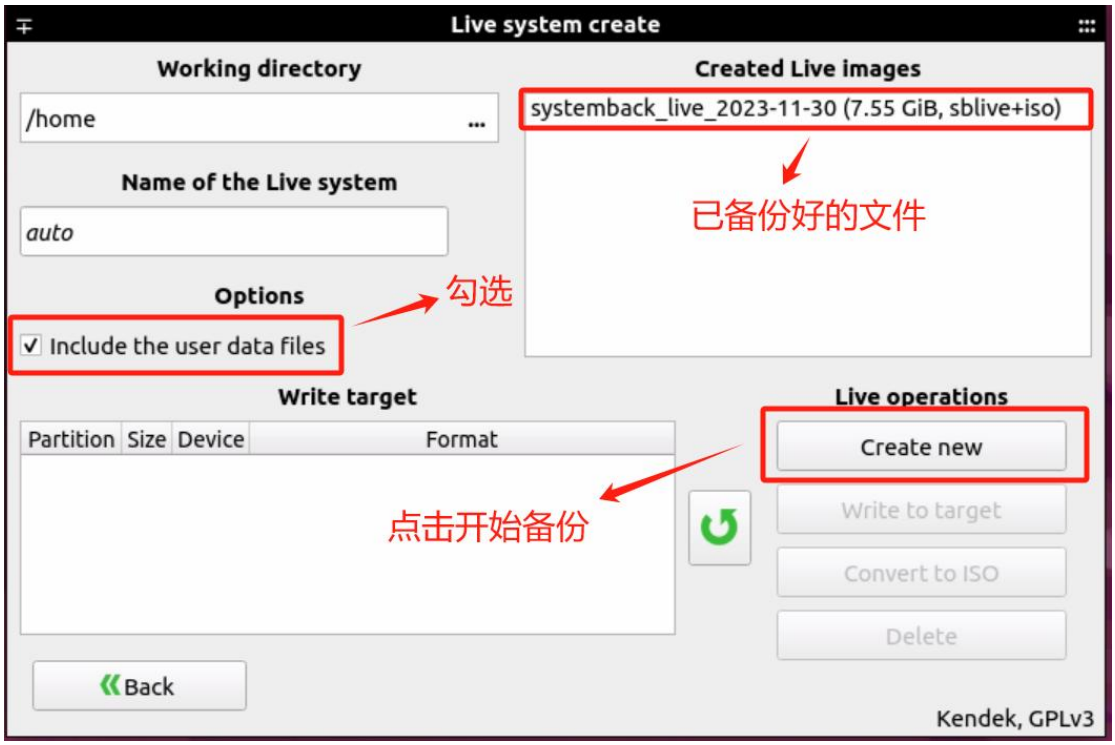
Appendix 2: System Backup

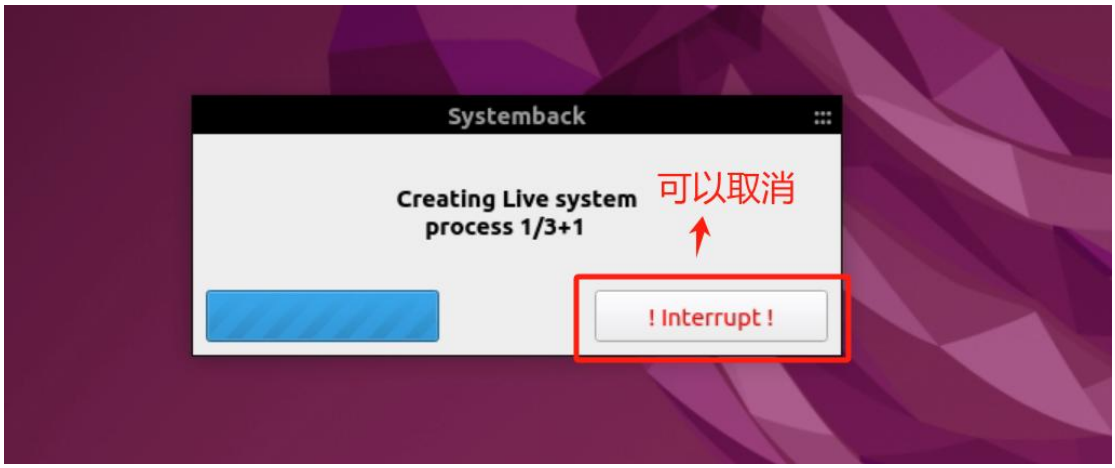
(1) On the desktop, search and select "Systemback" in the top left corner. Enter the password and choose "Live system create" to proceed.



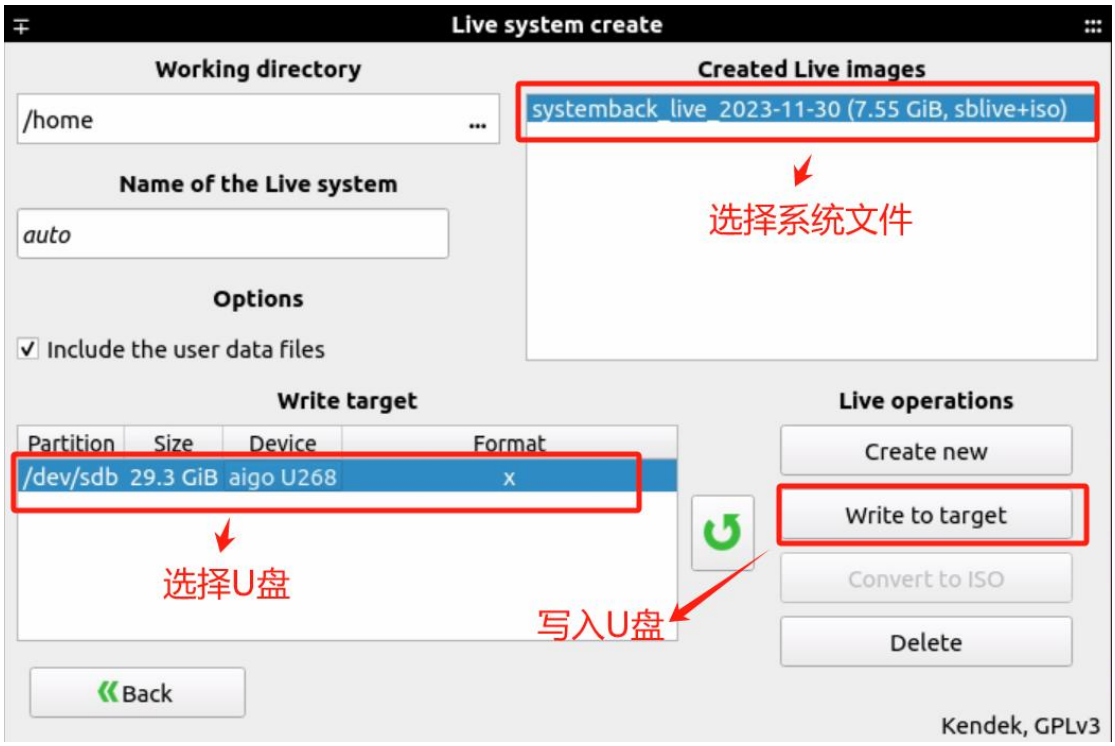


(2) Start the system backup. The backup process may take a while, so please wait for it to complete. During the backup process, avoid running programs or performing other operations.



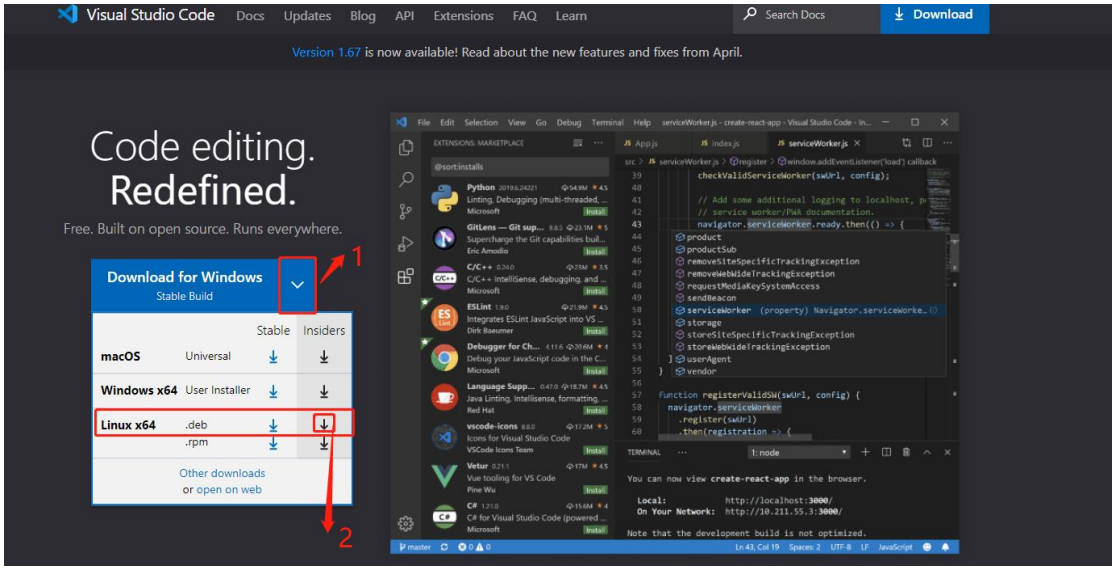


(3) After the backup is complete, insert the USB drive into the industrial computer and write the system to the USB drive (Note: the USB drive will be cleared before writing). The writing process may take a while, so please wait for it to complete.

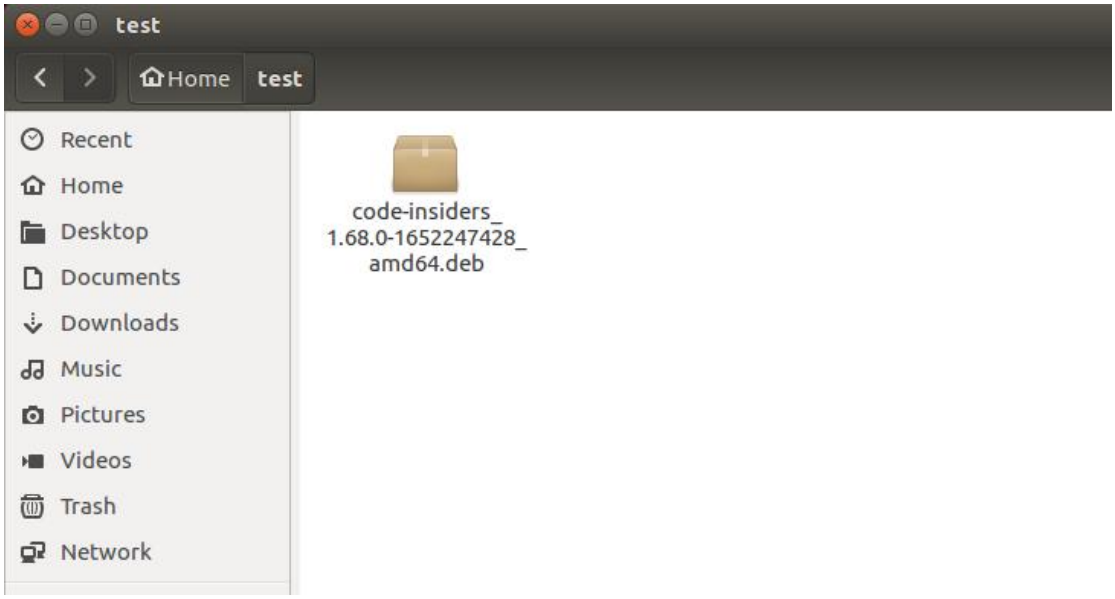


Appendix 3: Installing VSCode

1. Go to the official website (https://code.visualstudio.com/) and download the Linux x64 deb installation package.

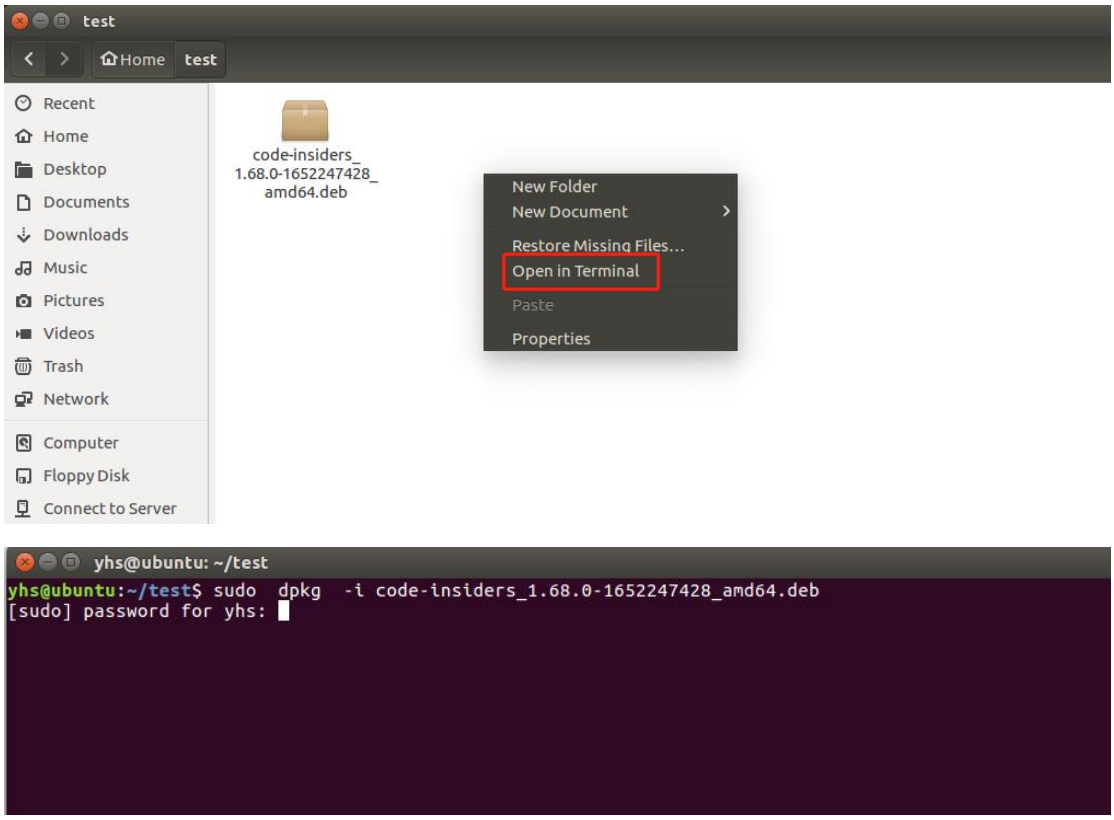


2. Let's assume that the downloaded file is "code-insiders_1.68.0-1652247428_amd64.deb".



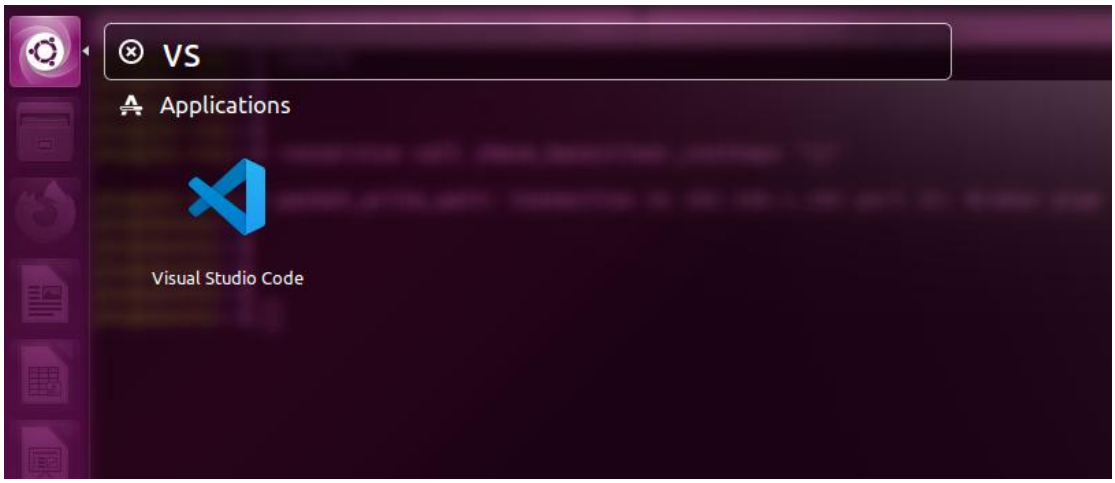
3. Right-click on the installation package in the directory, and choose "Open in Terminal" from the context menu. Enter the command and press Enter.

```
sudo  dpkg  -i code-insiders_1.68.0-1652247428_amd64.deb
```



4. Enter the root password and wait for the installation to complete.

5. Open VS Code and click on the search bar in the top left corner. Type "vs" and you will see the software icon. Click on the icon to open the software.
With this, the software installation is complete.



Tel: 0755-28468956

Fax: 0755-28468956

Website: www.yuhesen.com

Email: sales01@yuhesen.com

Add: Room 501, Building 1, Chuangwei Qunxin Technology Park, Baolong 6th Rd, No.1, Longgang District, Shenzhen City, Guangdong Province, China 518116