More details about ROS , please check this link: wiki : http://wiki.ros.org/

1.Introduction of ROS

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS is similar in some respects to 'robot frameworks,' such
as Player, YARP, Orocos, CARMEN, Orca, MOOS, and Microsoft Robotics Studio.

2. Common commands and tools

2.1 Introduction to ROS turtles

The Chinese name of master is the node manager. The master is used to manage many processes in the system. When each node starts, it must register with the master to manage the communication between the nodes. After the master is started, register each node through the master.

●Start ROS master

```
roscore
```

After starting the master, we can start the node, a node is a process. When we run executable files, they are loaded into the process. A node is dedicated to a process.

Node startup command is rosrun+package name+node name

```
rosrun [--prefix cmd] [--debug] pkg_name node_name [ARGS]
```

●Start the small turtle simulator node

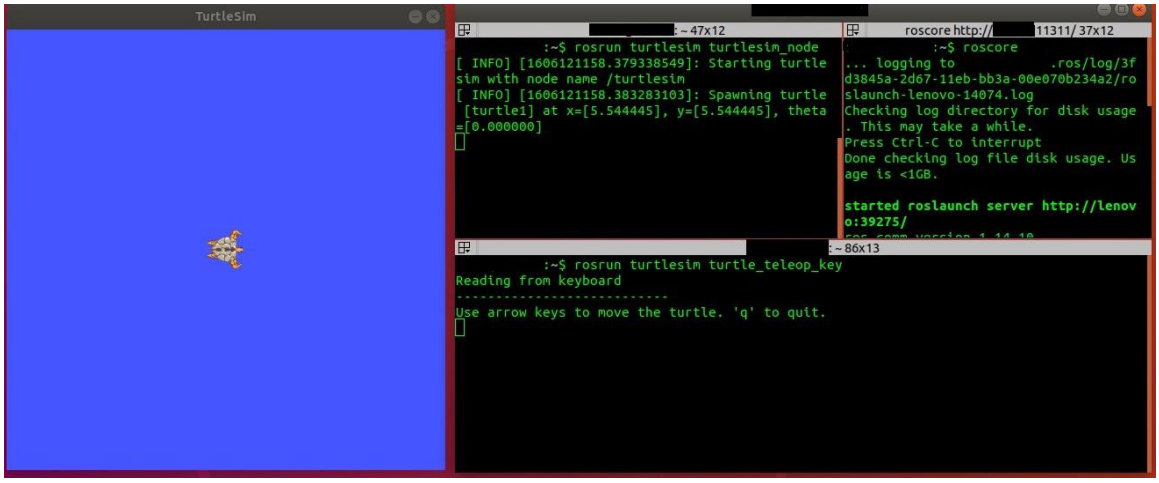
```
rosrun turtlesim turtlesim_node
```

● Start the small turtle keyboard controller node

```
rosrun turtlesim turtle_teleop_key
```

After the startup is complete, we can control the movement of the little turtle through the keyboard.

Keeping the cursor on the command line [rosrun turtlesim turtle_teleop_key], and then, we can control the movement of the turtle through the up, down, left, and right keyboards.

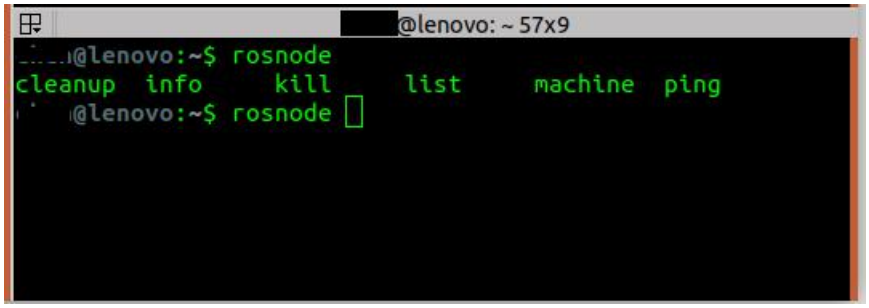


At this time, the system [rosrun turtlesim turtlesim_node] terminal will print some log information of small turtles

[INFO] [1607648666.226328691]: Starting turtlesim with node name /turtlesim [INFO] [1607648666.229275030]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]

2.2 [rosnode] node information

Input [rosnode] in the command line, and then double-click the Tab key, we can see the words as shown below



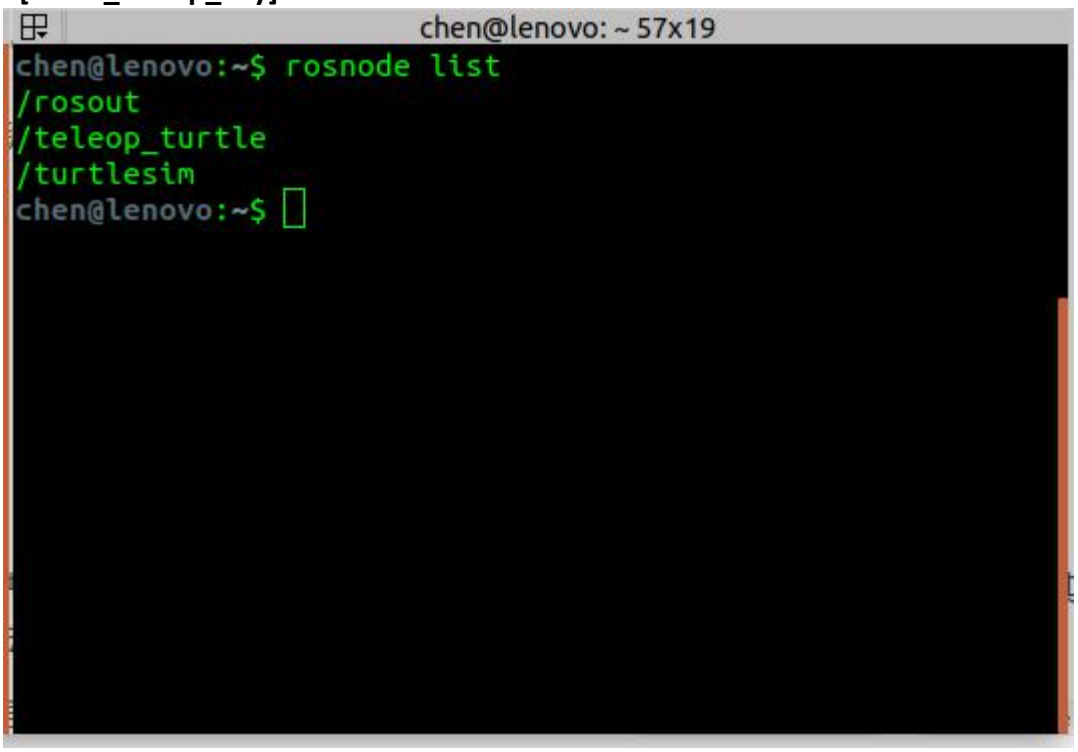
The detailed function list of the **[rosnode]** command is as follows:

Rosnode command	Function
rosnode list	Query all nodes currently running
rosnode info node_name	Display the detailed information of the node
rosnode kill node_name	End a node
rosnode ping	Test whether the node is alive
rosnode machine	List nodes running on a specific machine or list machine
rosnode cleanup	Clear the registration information of inoperable nodes

●Query all nodes currently running

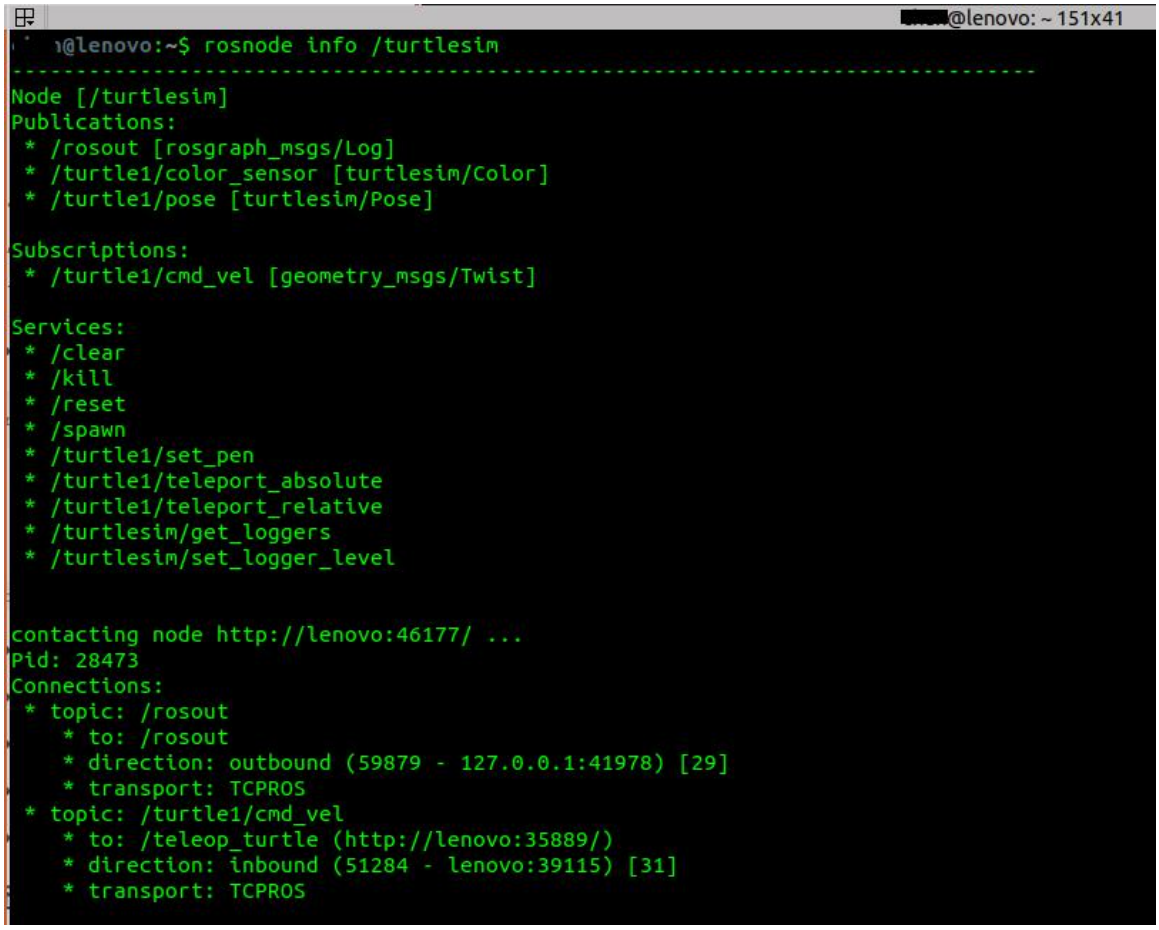
rosnode list

With the start of Little Turtle, we launched two executable programs: **[turtlesim_node]** and **[turtle_teleop_key]**. As shown below.



●View **[/turtlesim]** node

```
rosnode info /turtlesim
```



Some information about the current node:

- Publications: The publishers defined on this node
- Subscriptions: Subscribers defined on this node
- Services: Services defined on this node
- Process id, network port occupied
- Connections: connection information between this node and other nodes

●View **[/teleop_turtle]** node

```
rosnode info /teleop_turtle
```

```
[redacted]@lenovo: ~ 151x41
@lenovo:~$ rosnode info /teleop_turtle
--------------------------------------------------------------------------------
Node [/teleop_turtle]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/cmd_vel [geometry_msgs/Twist]

Subscriptions: None

Services:
 * /teleop_turtle/get_loggers
 * /teleop_turtle/set_logger_level


contacting node http://lenovo:35889/ ...
Pid: 28555
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound (39115 - 127.0.0.1:51282) [12]
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /turtlesim
    * direction: outbound (39115 - 127.0.0.1:59700) [11]
    * transport: TCPROS

@lenovo:~$
```

2.3 ROS-rqt(QT Tool)

Open the command line window and input **[rosrun rqt]**, then double-click the **[Tab]** key to view the contents of the QT tool in ROS, as shown below.

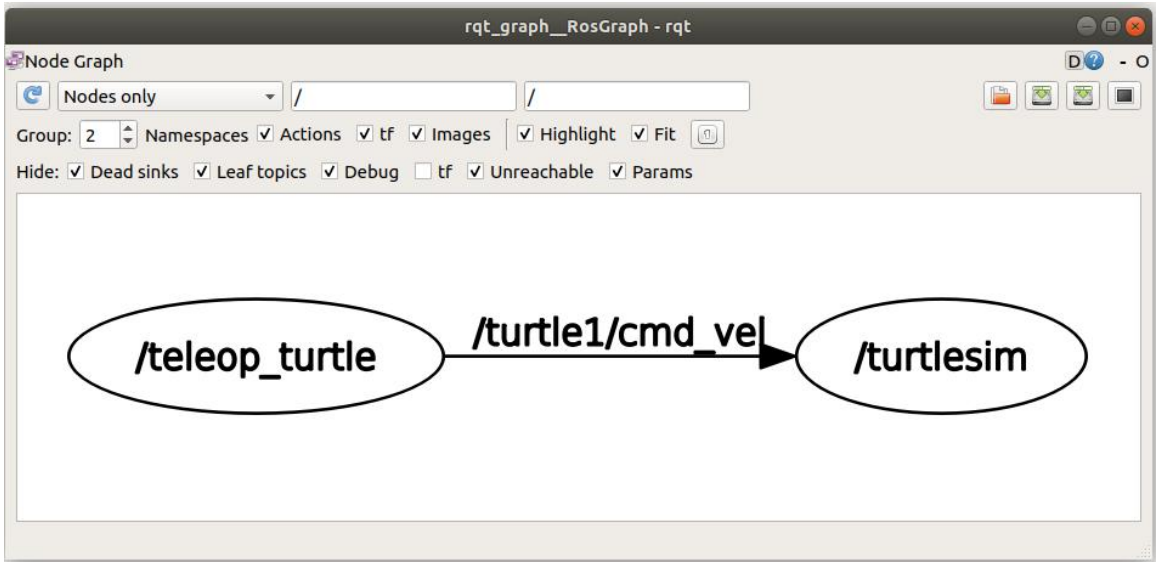
```
[redacted]@lenovo: ~ 147x39
@lenovo:~$ rosrun rqt_
rqt_action          rqt_gui             rqt_moveit          rqt_py_common       rqt_runtime_monitor  rqt_top
rqt_bag             rqt_gui_cpp         rqt_msg             rqt_py_console      rqt_rviz             rqt_topic
rqt_bag_plugins     rqt_gui_py          rqt_nav_view        rqt_reconfigure     rqt_service_caller   rqt_web
rqt_console         rqt_image_view      rqt_plot            rqt_robot_dashboard rqt_shell
rqt_dep             rqt_launch          rqt_pose_view       rqt_robot_monitor   rqt_srv
rqt_graph           rqt_logger_level    rqt_publisher       rqt_robot_steering  rqt_tf_tree
```

We briefly introduce several commonly used QT tools.

- [rqt_graph] calculation graph visualization tool

Open the command line window and input the following command, a dialog window will pop up.

```
rosrun rqt_graph rqt_graph
```

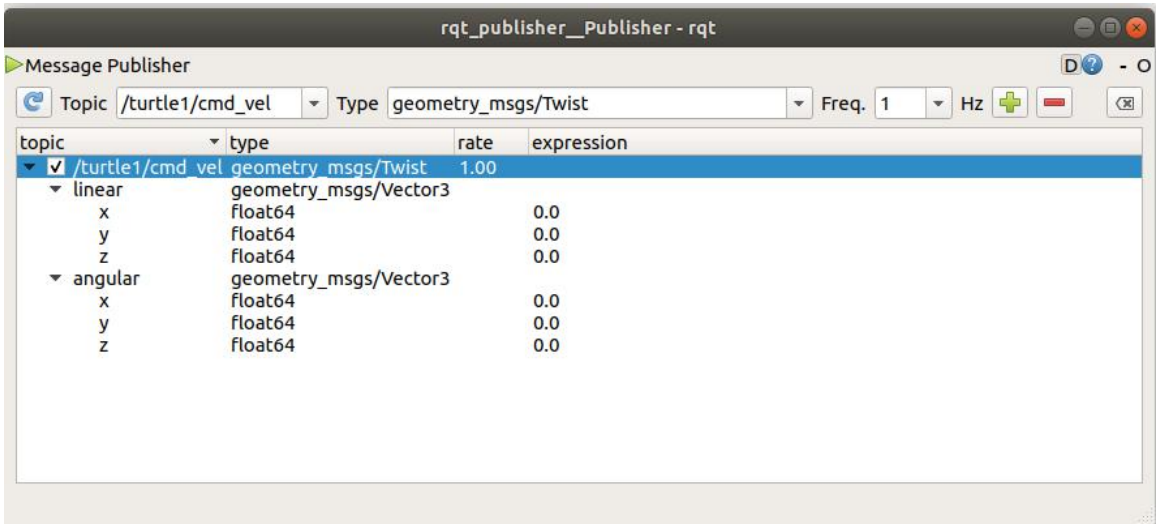


From the above image, we can see that the **[/teleop_turtle]** node transmits data to the **[/turtlesim]** node through the topic **[/turtle1/cmd_vel]**.

[/Teleop_turtle] is a node with Publisher (publishing) function.
[/Turtlesim] is a node with Subscriber (subscription) function.
[/Turtle1/cmd_vel] is the topic of communication between publisher and subscriber.

●**[rqt_topic]** view topic tool

```
rosrun rqt_topic rqt_topic
```

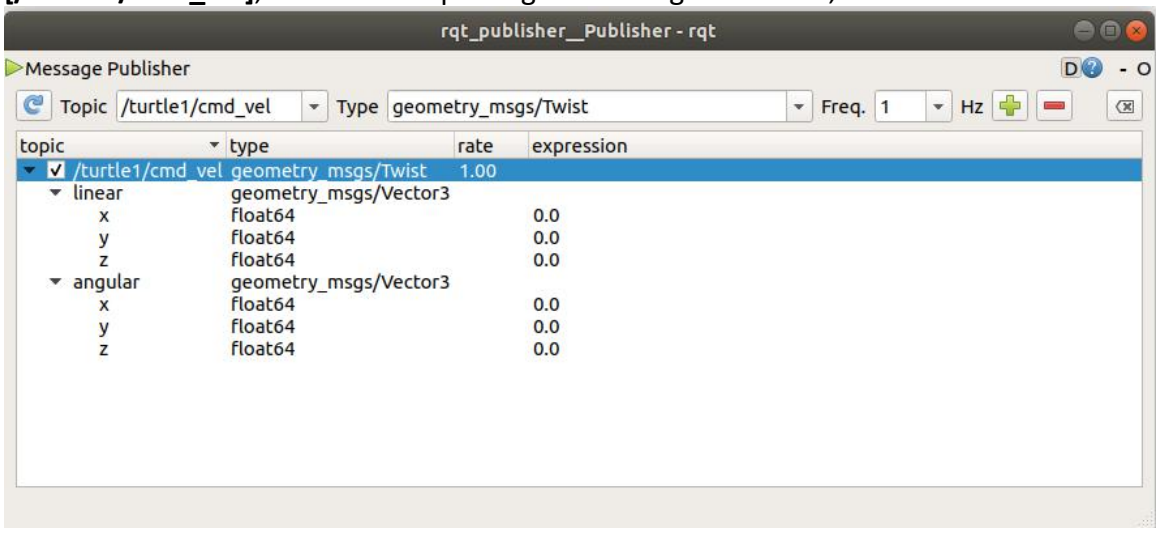


Through this tool, we can see some real-time changes of the little turtles.
[rqt_publisher] tool
[Rqt_publisher] provides a GUI plug-in for publishing arbitrary messages with fixed or calculated field values.

Input following command, a dialog window will pop up.

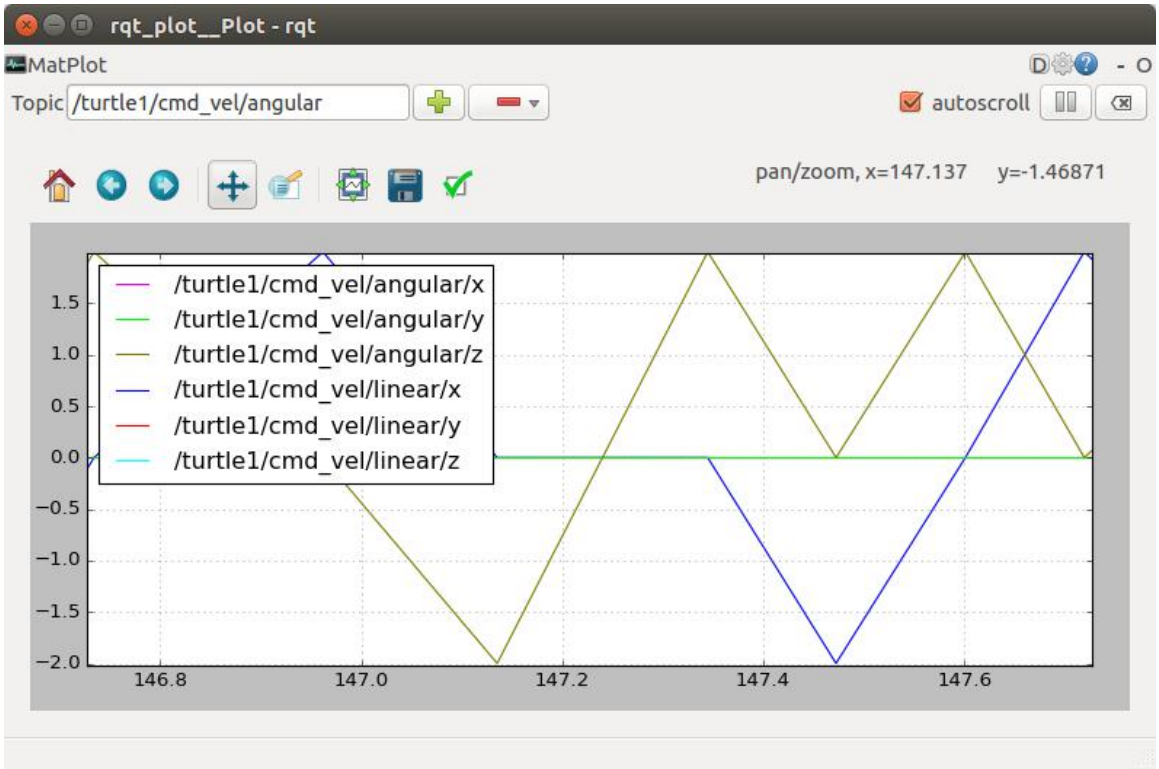
```
rosrun rqt_publisher rqt_publisher
```

Click the selection box on the right of **[Topic]** to find the topic we need **[/turtle1/cmd_vel]**, and click the plus sign on the right to add it, as shown below.



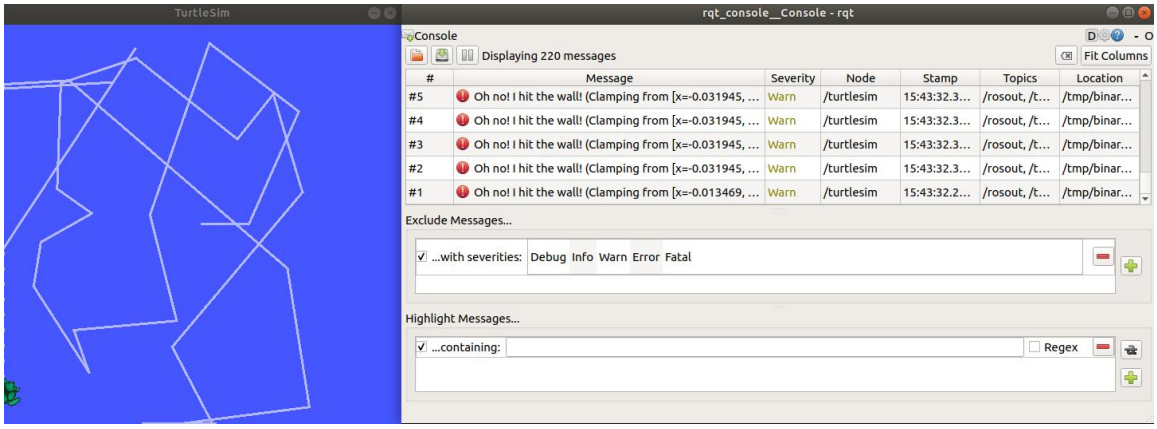
- **[rqt_topic] Data drawing tool**

```
rosrun rqt_plot rqt_plot
```



●**[rqt_console] Log output tool**

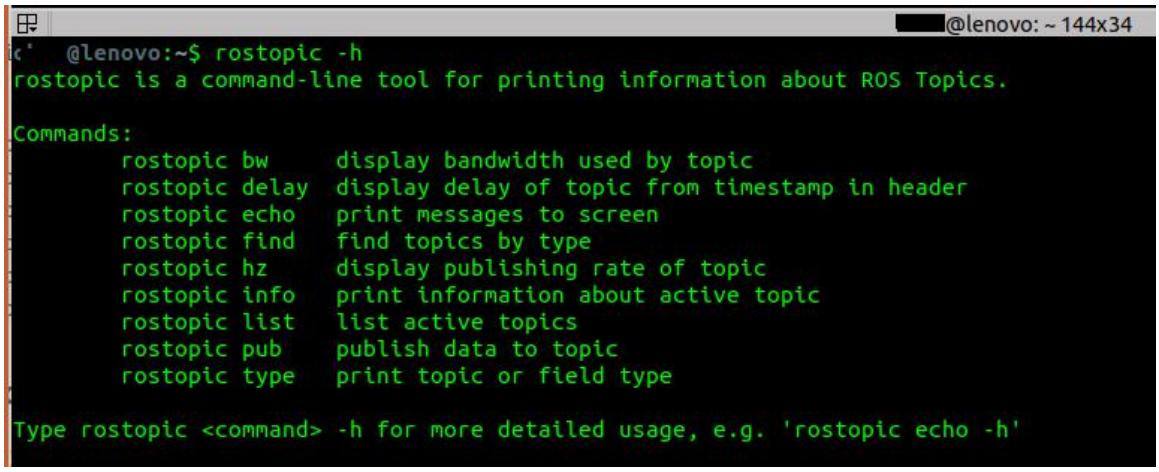
```
rosrun rqt_console rqt_console
```



2.4 [rostopic] Introduction

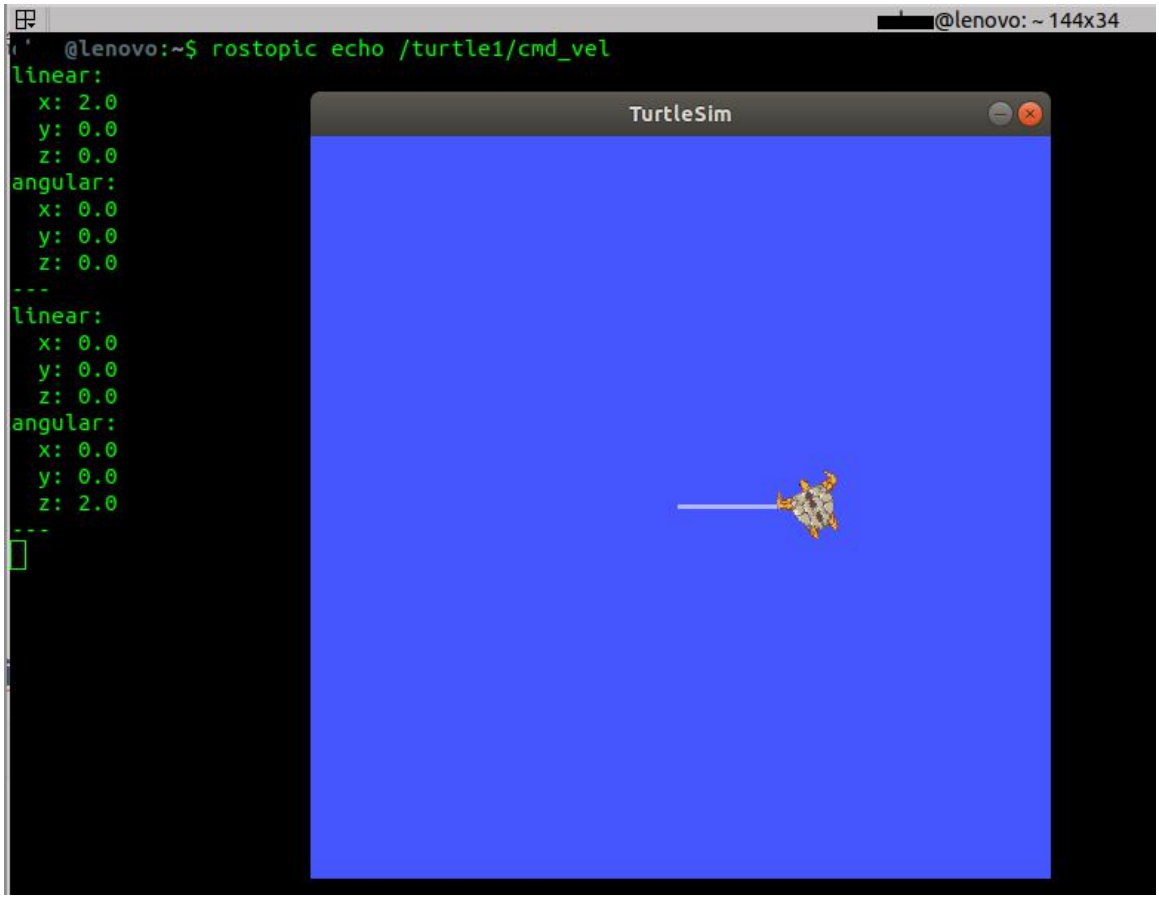
●**[rostopic]** get information about ROS topics.

```
rostopic -h
```



●**[rostopic echo]** show data posted on a topic

```
rostopic echo /turtle1/cmd_vel
```



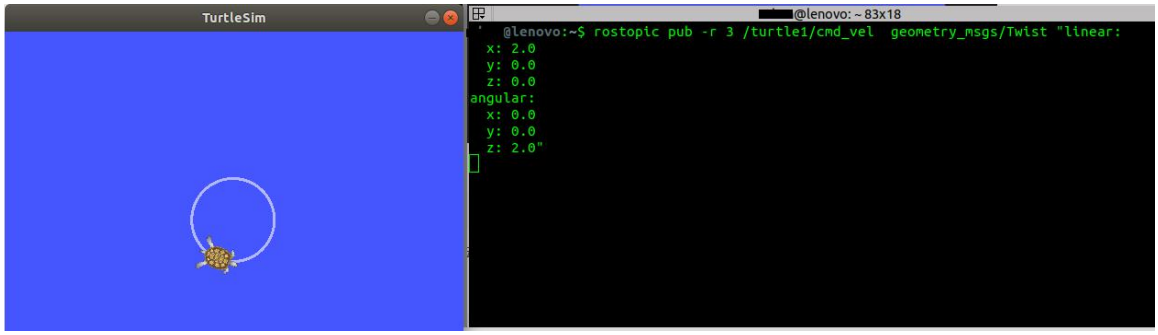
●**[rostopic echo]** view the type of a topic

```
rostopic type /turtle1/cmd_vel
```

We can get the mobile data type of the little turtle through the rostopic command as **[geometry_msgs/Twist]**

●**[rostopic pub]** publish data in a topic

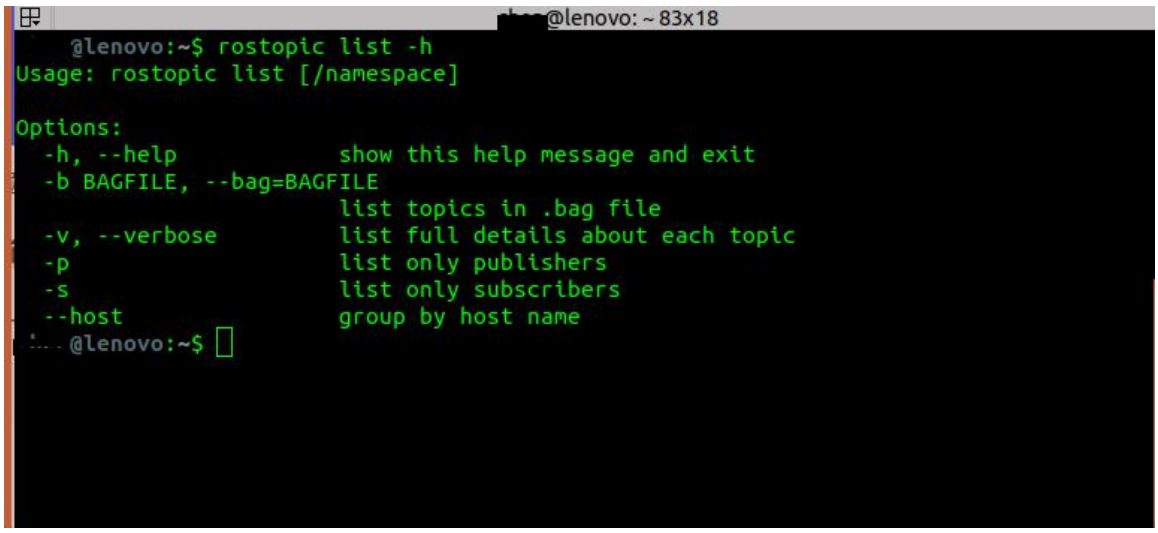
```
rostopic pub -r 3 /turtle1/cmd_vel  geometry_msgs/Twist "linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0"
```



r is for rate, and the following number stands for the number of times the topic is posted in one second.

2.5 [rostopic list] lists all currently subscribed and published topics.

rostopic list -h



Use the -v option in **[rostopic list]** to display detailed information about the published and subscribed topics and their types.

rostopic list -v

```
[redacted]@lenovo: ~ 148x40
   @lenovo:~$ rostopic list -v

Published topics:
 * /place/cancel [actionlib_msgs/GoalID] 3 publishers
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher
 * /pickup/goal [moveit_msgs/PickupActionGoal] 3 publishers
 * /move_group/cancel [actionlib_msgs/GoalID] 3 publishers
 * /execute_trajectory/cancel [actionlib_msgs/GoalID] 3 publishers
 * /move_group/goal [moveit_msgs/MoveGroupActionGoal] 3 publishers
 * /execute_trajectory/goal [moveit_msgs/ExecuteTrajectoryActionGoal] 3 publishers
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
 * /attached_collision_object [moveit_msgs/AttachedCollisionObject] 3 publishers
 * /joint_states [sensor_msgs/JointState] 1 publisher
 * /rosout [rosgraph_msgs/Log] 7 publishers
 * /trajectory_execution_event [std_msgs/String] 3 publishers
 * /pickup/cancel [actionlib_msgs/GoalID] 3 publishers
 * /turtle1/color_sensor [turtlesim/Color] 1 publisher
 * /planning_scene [moveit_msgs/PlanningScene] 1 publisher
 * /place/goal [moveit_msgs/PlaceActionGoal] 3 publishers
 * /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
 * /pickup/result [moveit_msgs/PickupActionResult] 3 subscribers
 * /tf [tf2_msgs/TFMessage] 3 subscribers
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
 * /execute_trajectory/result [moveit_msgs/ExecuteTrajectoryActionResult] 3 subscribers
 * /tf_static [tf2_msgs/TFMessage] 3 subscribers
 * /place/feedback [moveit_msgs/PlaceActionFeedback] 3 subscribers
 * /pickup/feedback [moveit_msgs/PickupActionFeedback] 3 subscribers
 * /move_group/fake_controller_joint_states [sensor_msgs/JointState] 1 subscriber
 * /pickup/status [actionlib_msgs/GoalStatusArray] 3 subscribers
 * /place/status [actionlib_msgs/GoalStatusArray] 3 subscribers
 * /move_group/feedback [moveit_msgs/MoveGroupActionFeedback] 3 subscribers
 * /rosout [rosgraph_msgs/Log] 1 subscriber
 * /place/result [moveit_msgs/PlaceActionResult] 3 subscribers
 * /execute_trajectory/feedback [moveit_msgs/ExecuteTrajectoryActionFeedback] 3 subscribers
 * /move_group/result [moveit_msgs/MoveGroupActionResult] 3 subscribers
 * /execute_trajectory/status [actionlib_msgs/GoalStatusArray] 3 subscribers
 * /move_group/status [actionlib_msgs/GoalStatusArray] 3 subscribers
```

2.6 [rosmsg] introductions of news

rosmsg -h

```
[redacted]@lenovo: ~ 148x40
   @lenovo:~$ rosmsg -h
rosmsg is a command-line tool for displaying information about ROS Message types.

Commands:
        rosmsg show     Show message description
        rosmsg info     Alias for rosmsg show
        rosmsg list     List all messages
        rosmsg md5      Display message md5sum
        rosmsg package  List messages in a package
        rosmsg packages List packages that contain messages

Type rosmsg <command> -h for more detailed usage
```

View currently open messages

rosmsg list

```
@lenovo: ~ 148x40
@lenovo:~$ rosmsg list
actionlib/TestAction
actionlib/TestActionFeedback
actionlib/TestActionGoal
actionlib/TestActionResult
actionlib/TestFeedback
actionlib/TestGoal
actionlib/TestRequestAction
actionlib/TestRequestActionFeedback
actionlib/TestRequestActionGoal
actionlib/TestRequestActionResult
actionlib/TestRequestFeedback
actionlib/TestRequestGoal
actionlib/TestRequestResult
actionlib/TestResult
actionlib/TwoIntsAction
actionlib/TwoIntsActionFeedback
actionlib/TwoIntsActionGoal
actionlib/TwoIntsActionResult
actionlib/TwoIntsFeedback
actionlib/TwoIntsGoal
actionlib/TwoIntsResult
actionlib_msgs/GoalID
actionlib_msgs/GoalStatus
actionlib_msgs/GoalStatusArray
actionlib_tutorials/AveragingAction
actionlib_tutorials/AveragingActionFeedback
actionlib_tutorials/AveragingActionGoal
actionlib_tutorials/AveragingActionResult
actionlib_tutorials/AveragingFeedback
actionlib_tutorials/AveragingGoal
actionlib_tutorials/AveragingResult
actionlib_tutorials/FibonacciAction
actionlib_tutorials/FibonacciActionFeedback
```

View the information contained in the message

```
rosmsg info turtlesim/Pose
```

```
@lenovo:~$ rosmsg info turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

The data type of the custom message is somewhat different from python and C++.

bool	uint8_t	bool
int8	int8_t	int
int16	int16_t	int
int32	int32_t	int
int64	int64_t	int , long
uint8	uint8_t	int
uint16	uint16_t	int
uint32	uint32_t	int
uint64	uint64_t	int , long
float32	float	float
float64	float	float
string	std:string	str , bytes
time	ros:Time	rospy.Time
duration	ros::Duration	rospy.Duration

3. Create project engineering

3.1 Create a workspace

Work space (a warehouse), which is loaded with various ROS projects, which is convenient for system organization, management and call. It is a folder in the visual graphical interface. The ROS code we write is usually placed in the workspace.

We create a dofbot_ws workspace as a warehouse for our robotic arm project.

```
mkdir -p ~/dofbot_ws/src     # Create
cd dofbot_ws/          # Enter the workspace
catkin_make       # Compile
```

After completing the above steps, you will see three folders devel/;build/;src/ under the dofbot_ws/ directory.

•src/: ROS catkin package (source code package)

•build/: catkin (CMake) cache information and intermediate files
•devel/: generated object files (including header files, dynamic link libraries, static link libraries, executable files, etc.), environment variables.

3.2 Create package functon package

You need to use the [catkin_create_pkg] command under [dofbot_ws/src] to create a package.

```
catkin_create_pkg package depends
```

package is the package name, depends is the name of the dependent package, which can depend on multiple packages.

```
cd ~/dofbot_ws/src/
catkin_create_pkg dofbot_moveit roscpp rospy rosmsg
cd ~/dofbot_ws/
```

Create a new dofbot_moveit function package under the current path, including:

```
├── CMakeLists.txt
├── include
|   └── dofbot_moveit
├── package.xml
└── src
```

[Catkin_create_pkg] has completed the initialization of the software package, filled in **[CMakeLists.txt]** and **[package.xml]**, and filled the dependencies into these two files.

●Generally function package architecture.

```
CMakeLists.txt: The compilation script of the current package. It is usually necessary to add compile-time dependencies to C++ code, perform operations.
package.xml: package related information. Usually add some ros library support
include folder: store c++ header files
config folder: store parameter configuration files, the format is yaml
Launch folder: Store the .launch file.
meshes folder: model storage folder
src folder: c++ source code
scripts folder: python source code
srv folder: store the defined service
msg folder: store custom message protocol
action folder: store custom actions
```

●Start the ros node from the command line

```
1. Enter the directory where the python file is located, and run the python file name.py directly
```

Some files need python3, run python3 filename.py
E.g:
cd ~/dofbot_ws/src/dofbot_moveit/scripts/
python 02_motion_plan.py
2. Enter the workspace, compile first, then update the environment, and finally rosrun starts
cd ~/dofbot_ws/
catkin_make
source devel/setup.bash
Python file
rosrun dofbot_moveit 02_motion_plan.py
For C++ files, 02_motion_plan here is not the file name, but the target file after compilation
rosrun dofbot_moveit 02_motion_plan

3.3 [CMakeLists.txt] Introduction

●Overview

This file specifies the rules from the source code to the target file. When the catkin compilation system is working, it will first find the **[CMakeLists.txt]** under each package, and then compile and build according to the rules.

●Format

[CMakeLists.txt] The basic syntax is still in accordance with CMake, and Catkin added a small number of macros, the overall structure is as follows:

```
cmake_minimum_required() #CMake version
project()            #Project name
find_package()       #Find others needed for compilation CMake/Catkin package
catkin_python_setup()
add_message_files()
add_service_files()
add_action_files()
generate_message()     # catkin adds new macros to generate msg/srv/action interfaces in different language versions
catkin_package()       #catkin adds a new macro to generate the cmake configuration of the current package for other packages that depend on this package to call
add_library()
add_executable()       # Generate executable binary file
add_dependencies()
target_link_libraries()
catkin_add_gtest()
install()            # Install it on local
```

3.4 **[package.xml]** Introduction

●Overview

[Package.xml] is a necessary file for catkin package, which contains the package name, version number, content description, maintainer, software license, compile and build tools, compile dependencies, run dependencies and other information.

●Format

```
<pacakge>  root tag file
<name>  package name
<version>  version number
<description>  content description
<maintainer>  maintainer
<license>  Software license
<buildtool_depend>  Compile the build tool, usually catkin
<depend>  Specify dependencies as dependencies required for compilation, export, and operation, most commonly used
<build_depend>  compile dependencies
<build_export_depend>  export dependencies
<exec_depend>  run dependencies
<test_depend>  Test case dependencies
<doc_depend>  Document dependencies
```

4.dofbot custom message

4.1 Create custom message

●Create feature package

```
cd ~/dofbot_ws/src/
catkin_create_pkg dofbot_info roscpp rosmsg rospy
```

●Create custom news

```
mkdir srv
cd srv/
touch kinemarics.srv
```

[Kinemarics.srv] is a custom message file that needs to describe the format of the message. We can edit the code as follows.

```
# request
float64 tar_x
float64 tar_y
float64 tar_z
float64 Roll
float64 Pitch
float64 Yaw
```

```
float64 cur_joint1
float64 cur_joint2
float64 cur_joint3
float64 cur_joint4
float64 cur_joint5
float64 cur_joint6
string  kin_name
---
# response
float64 joint1
float64 joint2
float64 joint3
float64 joint4
float64 joint5
float64 joint6
float64 x
float64 y
float64 z
float64 Roll
float64 Pitch
float64 Yaw
```

4.2 Environment setup

●Configurate [Package.xml] file.

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

●Configurate [CMakeLists.txt]

Add [message_generation] in [find_package], the results are as follows.

```
# CMake/Catkin package
find_package(catkin REQUIRED COMPONENTS
      roscpp
      rosmsg
      rospy
      moveit_core
      moveit_msgs
      message_generation
      )
```

●Add [dd_service_files], as shown below.

```
#catkin Add custom Message/Service/Action file.
add_service_files(
      FILES
```

```
      kinemarics.srv
)
```

[Kinemarics.srv] must be the same as the name of the srv file you created, and it must be in the srv directory, otherwise compilation problems will occur

●Add **[generation_msg]**, as shown below.

```
#catkin enerate msg/srv/action interfaces in different language versions
 generate_messages(
   DEPENDENCIES
   std_msgs  # Or other packages containing msgs
 )
```

●Modify catkin_package, the results are as follows.

```
#catkin
catkin_package(
      #  INCLUDE_DIRS include
      #  LIBRARIES dofbot_info
        CATKIN_DEPENDS roscpp rosmsg rospy message_runtime
      #  DEPENDS system_lib
)
```

4.3 Compile

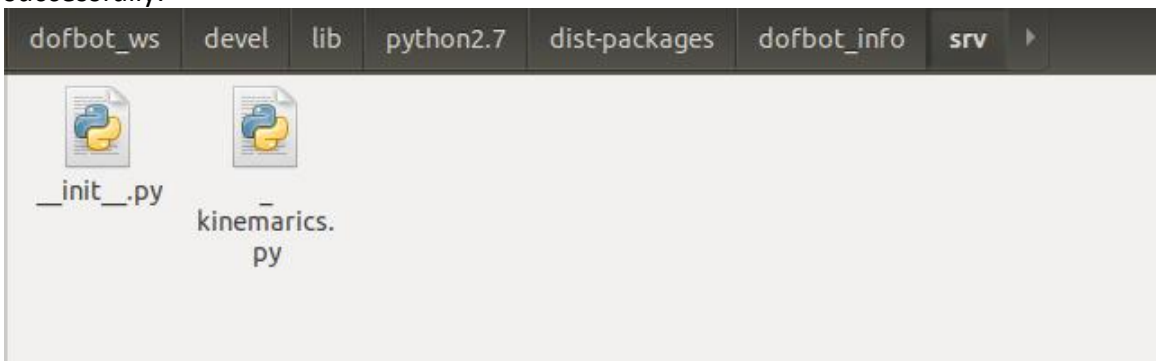
Enter the workspace directory, run and compile.

```
cd ~/dofbot_ws/
catkin_make
source devel/setup.bash
```

4.4 View verification

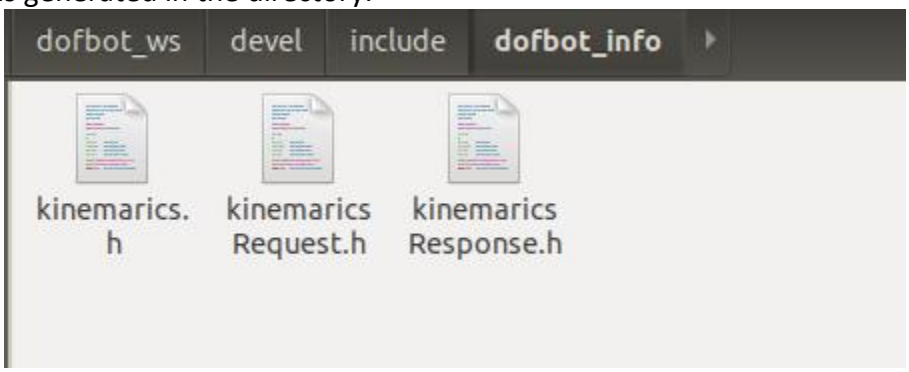
●C++ header files

Enter the include directory of devel, if the header file description is generated, the custom message is created
successfully.



●python py file

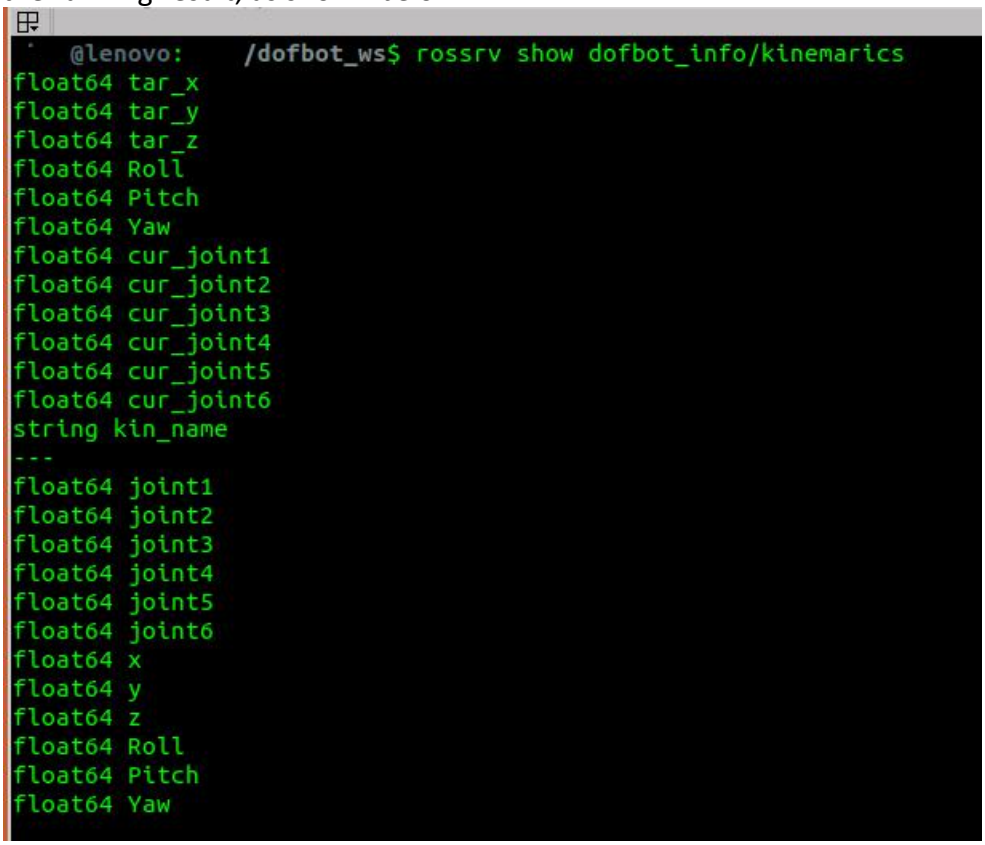
Enter the [lib/python2.7/dist-package] directory of devel and check whether a directory with the same name as the package is generated, and whether the corresponding py file is generated in the directory.



●Verification by rossrv tool

```
rossrv show dofbot_info/kinemarics
```

Check the running result, as shown below.



5.Server communication

●C++ code to create server side

Main function

```
/*
* This is the ROS server for the pros and cons of the robotic arm
 * Note: The end mentioned refers to the 5th steering gear rotation center,
 */
int main(int argc, char **argv) {
   // ROS node initialization
   ros::init(argc, argv, "dofbot_server");
   // Create node handle
   ros::NodeHandle n;
   cout << " Waiting to receive ******" << endl;
   // Create server
   ros::ServiceServer server = n.advertiseService("dofbot_kinemarics", srvicecallback);
   // block
   ros::spin();
   return 0;
}
```

Callback function:

```
bool srvicecallback(dofbot_info::kinemaricsRequest &request,
dofbot_info::kinemaricsResponse &response) {
   if (request.kin_name == "fk") {
      double joints[]{request.cur_joint1, request.cur_joint2, request.cur_joint3,
request.cur_joint4,
                  request.cur_joint5};
      ...
   }
   if (request.kin_name == "ik") {
                  ...
   }
   return true;
}
```

●Python code to implement the client

Create node and client (color sorting as an example)

```
# ROS node initialization
self.n = rospy.init_node('dofbot_identify', anonymous=True)
# Create a client to get the inverse solution result
self.client = rospy.ServiceProxy("dofbot_kinemarics", kinemarics)
```

Send request message:

```
def server_joint(self, posxy):
   '''
```

```
    Post position request, get joint rotation angle
    :param posxy: Location point x,y coordinates
    :return: Rotation angle of each servo
    '''
    # Wait for the server to start
    self.client.wait_for_service()
    # Create message package
    request = kinemaricsRequest()
    request.tar_x = posxy[0]
    request.tar_y = posxy[1]
    request.kin_name = "ik"
    try:
            response = self.client.call(request)
       if isinstance(response, kinemaricsResponse):
       # Obtain the inverse solution response result
       pass
       return joints
    except Exception:
       rospy.loginfo("arg error")
```