Experiment 2: ROS Basics and Communication Mechanisms

1. Experiment Objectives

- Understand the concept of ROS Nodes
- Master the Topic publish-subscribe mechanism
- Understand the role of the ROS Master

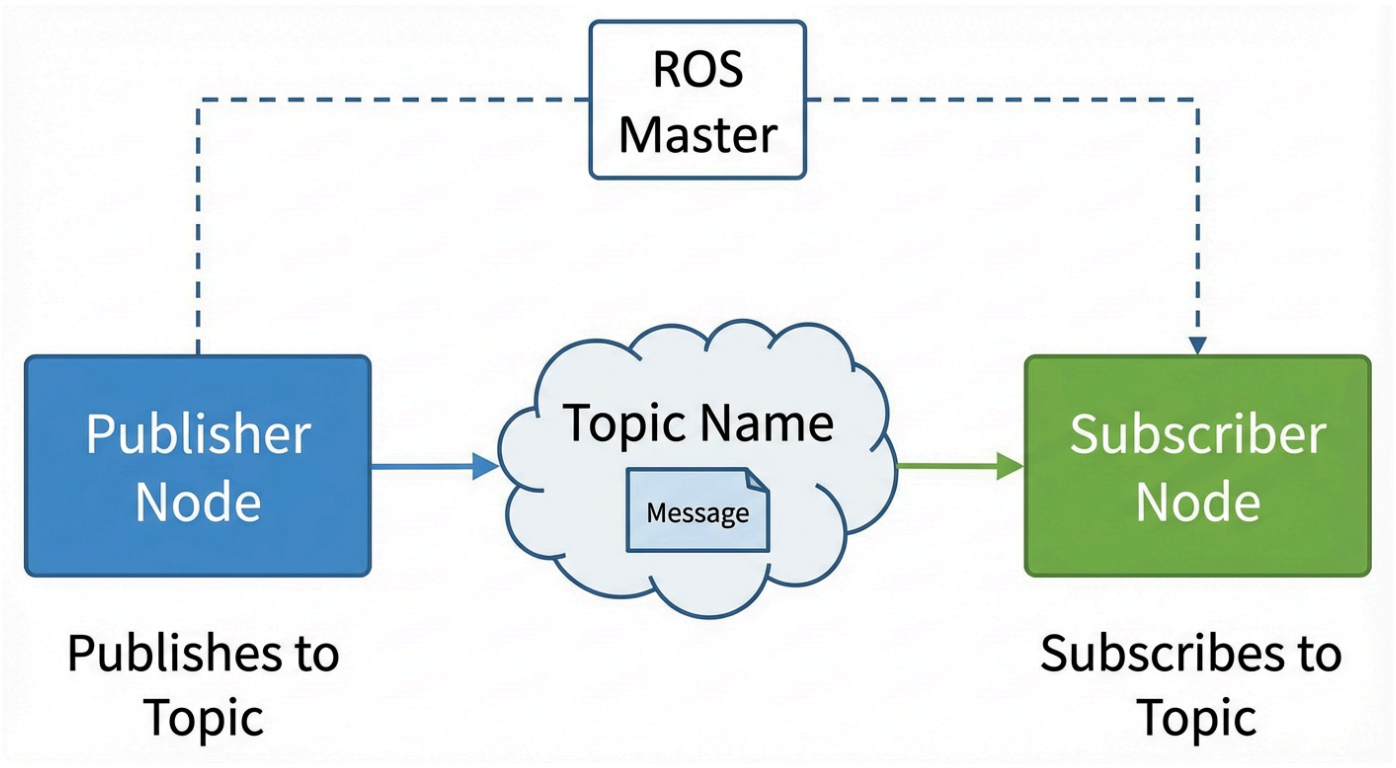
2. Core ROS Concepts

2.1 Basic Components

- **Node** : The basic operational unit of a ROS system; each node performs a specific task.
- **Topic** : A named channel for communication between nodes.
- **Master** : Provides coordination services for communication between nodes (requires `roscore`).

2.2 Node Communication

The most common way nodes communicate is based on topics. A publisher node names a topic and publishes messages to it. Another subscriber node subscribes to that topic.



3. Experiment Steps

First, download the course code repository from GitHub and rename it to catkin_ws.

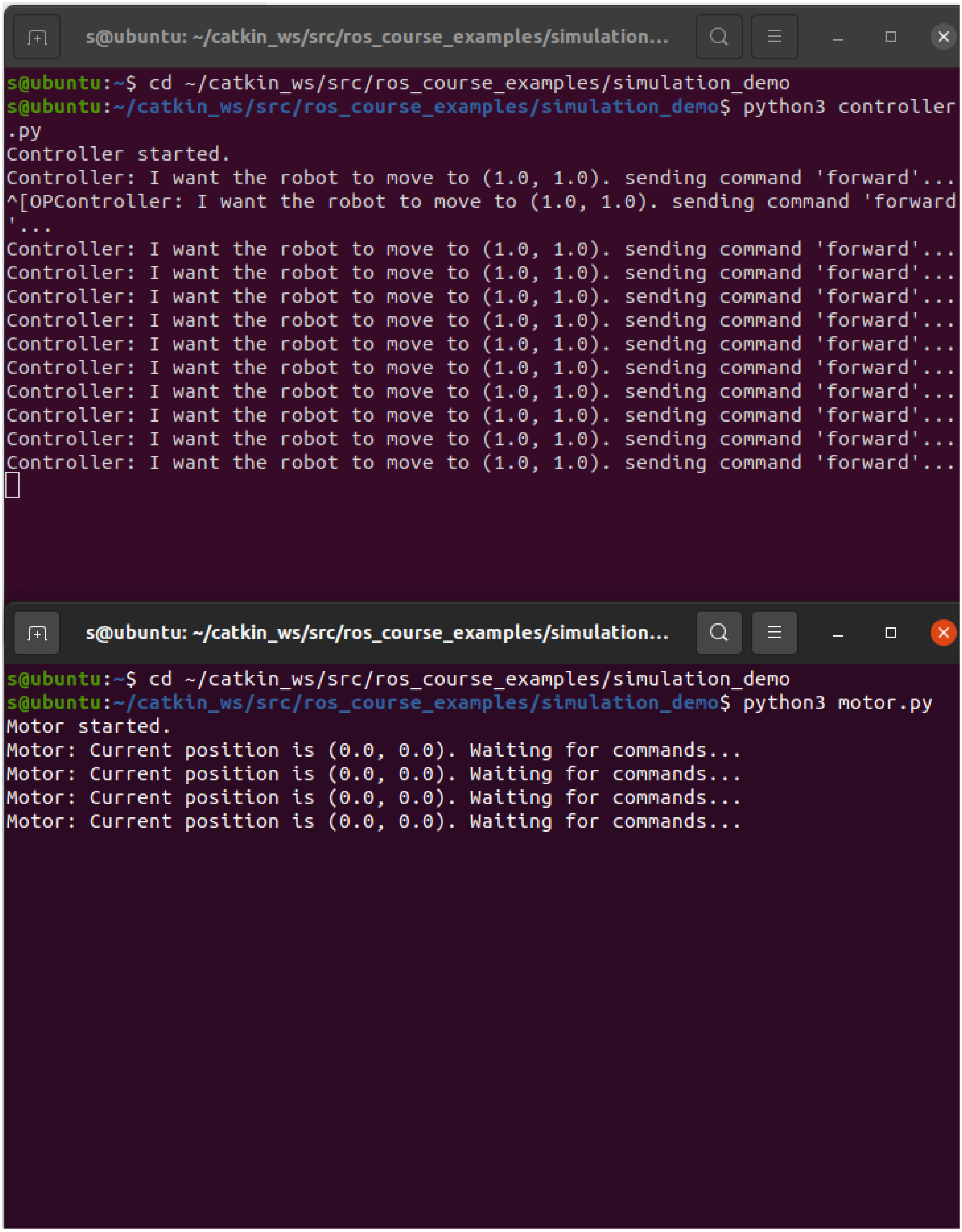
```
git clone https://github.com/AB-pixel-pixel/Embodied-AI-Exploration-Lab1.git
mv Embodied-AI-Exploration-Lab1 catkin_ws
cd catkin_ws
```

Experiment 2.1: Running two isolated programs

```
cd ~/catkin_ws/src/ros_course_examples/simulation_demo
python3 controller.py
```

```
# Terminal 2: Run Motor
cd ~/catkin_ws/src/ros_course_examples/simulation_demo
python3 motor.py
```

Execution result shown in the figure:



```
s@ubuntu: ~/catkin_ws/src/ros_course_examples/simulation...
s@ubuntu:~$ cd ~/catkin_ws/src/ros_course_examples/simulation_demo
s@ubuntu:~/catkin_ws/src/ros_course_examples/simulation_demo$ python3 controller
.py
Controller started.
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
^[OPController: I want the robot to move to (1.0, 1.0). sending command 'forward
'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
Controller: I want the robot to move to (1.0, 1.0). sending command 'forward'...
```

```
s@ubuntu: ~/catkin_ws/src/ros_course_examples/simulation...
s@ubuntu:~$ cd ~/catkin_ws/src/ros_course_examples/simulation_demo
s@ubuntu:~/catkin_ws/src/ros_course_examples/simulation_demo$ python3 motor.py
Motor started.
Motor: Current position is (0.0, 0.0). Waiting for commands...
Motor: Current position is (0.0, 0.0). Waiting for commands...
Motor: Current position is (0.0, 0.0). Waiting for commands...
Motor: Current position is (0.0, 0.0). Waiting for commands...
```

Observation : The two programs run independently and cannot communicate with each other.

Close these two programs now.

Experiment 2.2: Implementing Node Communication using ROS

We have already encapsulated the code into ROS nodes, the files are as follows: `src/ros_course_examples/nodes/motor_node.py` and `src/ros_course_examples/nodes/controller_node.py`.

As long as we start these two nodes, we can achieve node communication based on the ROS framework.

First, perform compilation.

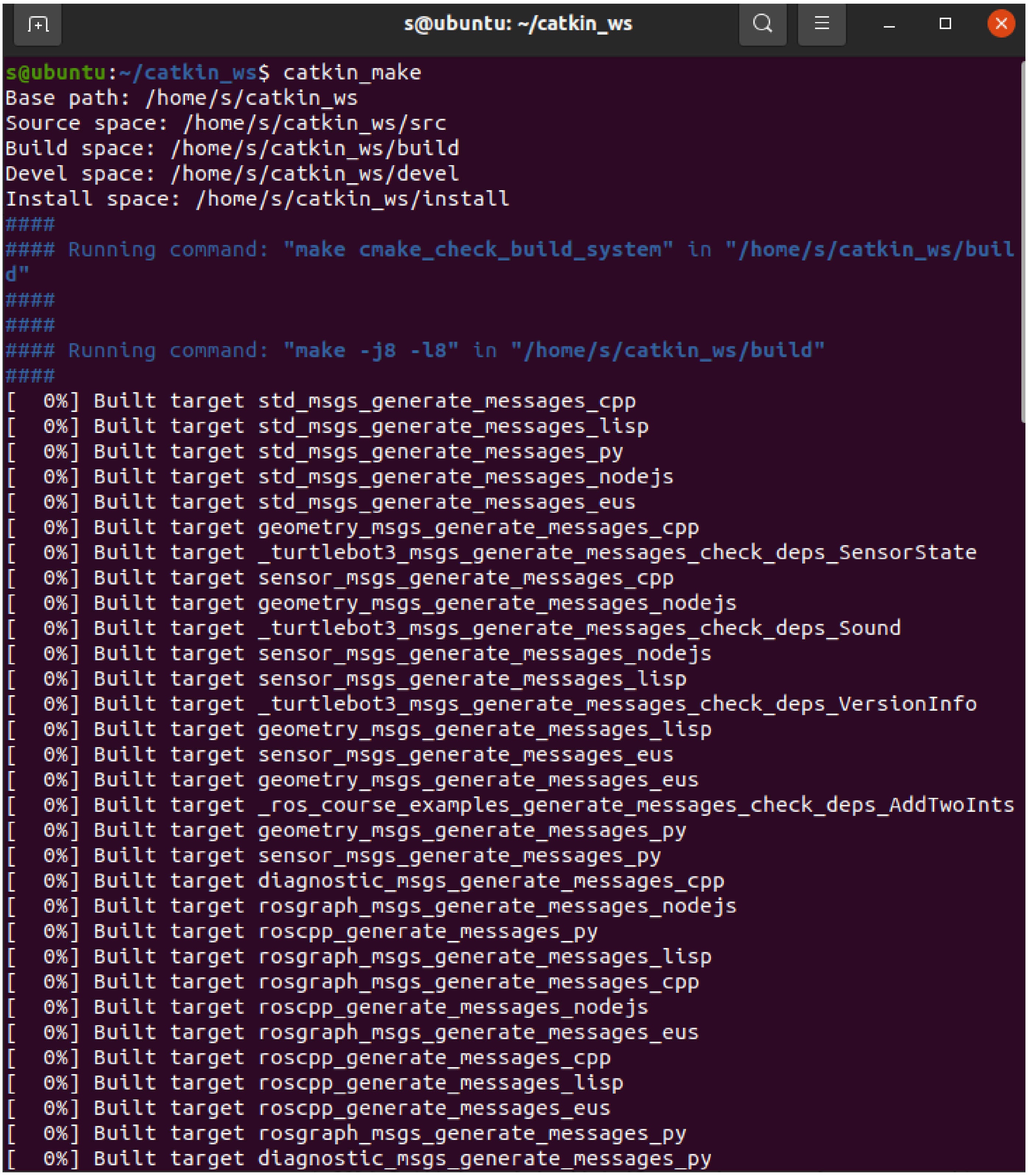
In Terminal 1: Compile the ROS workspace.

We use the `catkin_make` command to compile ROS packages. Its functions are:

- Compile source code: Compile C++ source files into executables, mark Python scripts as executable.
- Handle dependencies: Automatically resolve and link dependencies between packages.
- Generate configuration files: Create environment configuration scripts such as devel/setup.bash.
- Build message types: Compile custom msg, srv, and action files.

```
cd ~/catkin_ws
catkin_make
```

Rough schematic:



```
s@ubuntu:~/catkin_ws$ catkin_make
Base path: /home/s/catkin_ws
Source space: /home/s/catkin_ws/src
Build space: /home/s/catkin_ws/build
Devel space: /home/s/catkin_ws/devel
Install space: /home/s/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/s/catkin_ws/buil
d"
####
####
#### Running command: "make -j8 -l8" in "/home/s/catkin_ws/build"
####
[  0%] Built target std_msgs_generate_messages_cpp
[  0%] Built target std_msgs_generate_messages_lisp
[  0%] Built target std_msgs_generate_messages_py
[  0%] Built target std_msgs_generate_messages_nodejs
[  0%] Built target std_msgs_generate_messages_eus
[  0%] Built target geometry_msgs_generate_messages_cpp
[  0%] Built target _turtlebot3_msgs_generate_messages_check_deps_SensorState
[  0%] Built target sensor_msgs_generate_messages_cpp
[  0%] Built target geometry_msgs_generate_messages_nodejs
[  0%] Built target _turtlebot3_msgs_generate_messages_check_deps_Sound
[  0%] Built target sensor_msgs_generate_messages_nodejs
[  0%] Built target sensor_msgs_generate_messages_lisp
[  0%] Built target _turtlebot3_msgs_generate_messages_check_deps_VersionInfo
[  0%] Built target geometry_msgs_generate_messages_lisp
[  0%] Built target sensor_msgs_generate_messages_eus
[  0%] Built target geometry_msgs_generate_messages_eus
[  0%] Built target _ros_course_examples_generate_messages_check_deps_AddTwoInts
[  0%] Built target geometry_msgs_generate_messages_py
[  0%] Built target sensor_msgs_generate_messages_py
[  0%] Built target diagnostic_msgs_generate_messages_cpp
[  0%] Built target rosgraph_msgs_generate_messages_nodejs
[  0%] Built target roscpp_generate_messages_py
[  0%] Built target rosgraph_msgs_generate_messages_lisp
[  0%] Built target rosgraph_msgs_generate_messages_cpp
[  0%] Built target roscpp_generate_messages_nodejs
[  0%] Built target rosgraph_msgs_generate_messages_eus
[  0%] Built target roscpp_generate_messages_cpp
[  0%] Built target roscpp_generate_messages_lisp
[  0%] Built target roscpp_generate_messages_eus
[  0%] Built target rosgraph_msgs_generate_messages_py
[  0%] Built target diagnostic_msgs_generate_messages_py
```

After execution it will:

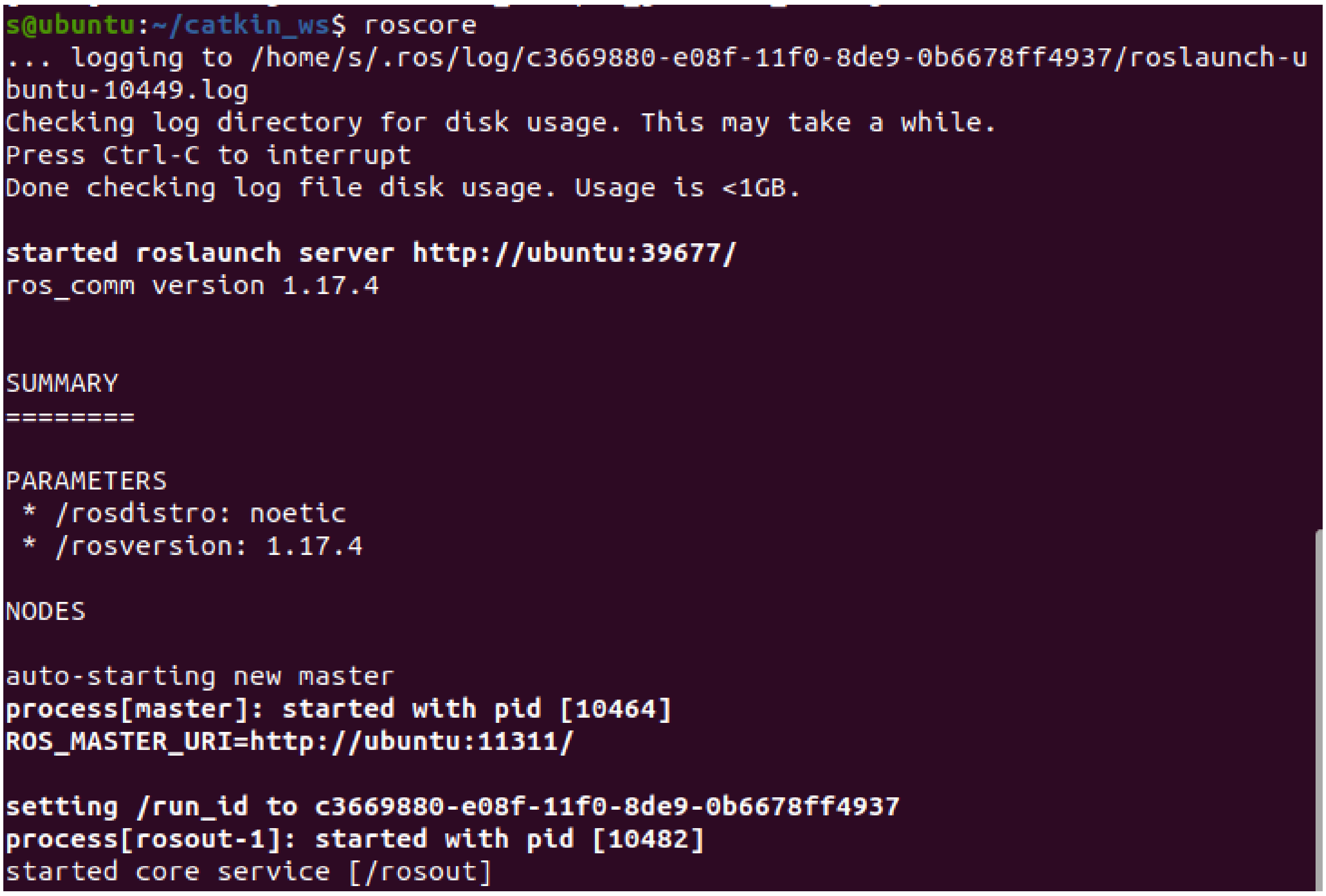
- Compile code in the build/ directory.
- Output results to the devel/ directory.
- Generate environment variables usable by ROS.

Terminal 1: Start the ROS Master node.

The `roscore` command can start the Master node:

- Function: Manages registration and discovery of all nodes.
- Role: Allows Publishers and Subscribers to find each other.
- Analogy: Like a central server, connecting nodes.
- **Note: During the experiment, we do not close the terminal running roscore.**

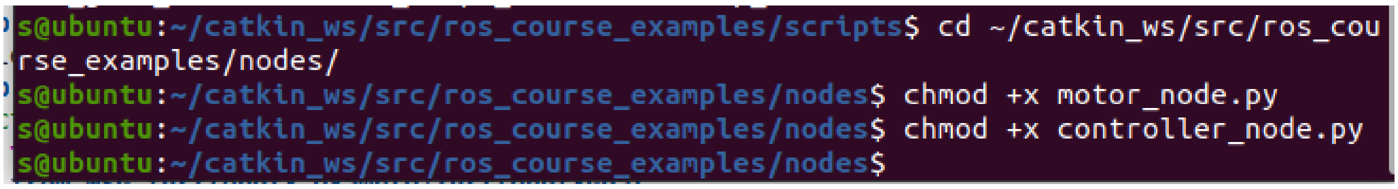
```
roscore
```



```
s@ubuntu:~/catkin_ws$ roscore
... logging to /home/s/.ros/log/c3669880-e08f-11f0-8de9-0b6678ff4937/roslaunch-u
buntu-10449.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:39677/
ros_comm version 1.17.4


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.17.4

NODES

auto-starting new master
process[master]: started with pid [10464]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to c3669880-e08f-11f0-8de9-0b6678ff4937
process[rosout-1]: started with pid [10482]
started core service [/rosout]
```

Grant execution permissions to the node code:

```
cd ~/catkin_ws/src/ros_course_examples/nodes/
chmod +x motor_node.py
chmod +x controller_node.py
```



```
s@ubuntu:~/catkin_ws/src/ros_course_examples/scripts$ cd ~/catkin_ws/src/ros_cou
rse_examples/nodes/
s@ubuntu:~/catkin_ws/src/ros_course_examples/nodes$ chmod +x motor_node.py
s@ubuntu:~/catkin_ws/src/ros_course_examples/nodes$ chmod +x controller_node.py
s@ubuntu:~/catkin_ws/src/ros_course_examples/nodes$
```

Start Communication Demo

Since we need to use the ROS framework, we need to use ROS commands to start the code instead of simply using python commands.

This is the `rosrun` command:

- Function: Run a node within a ROS package.
- Features: Simple, direct, suitable for testing or debugging.
- Example usage:

```
# rosrun <ros package name> <executable file name>
rosrun turtlesim turtlesim_node
```

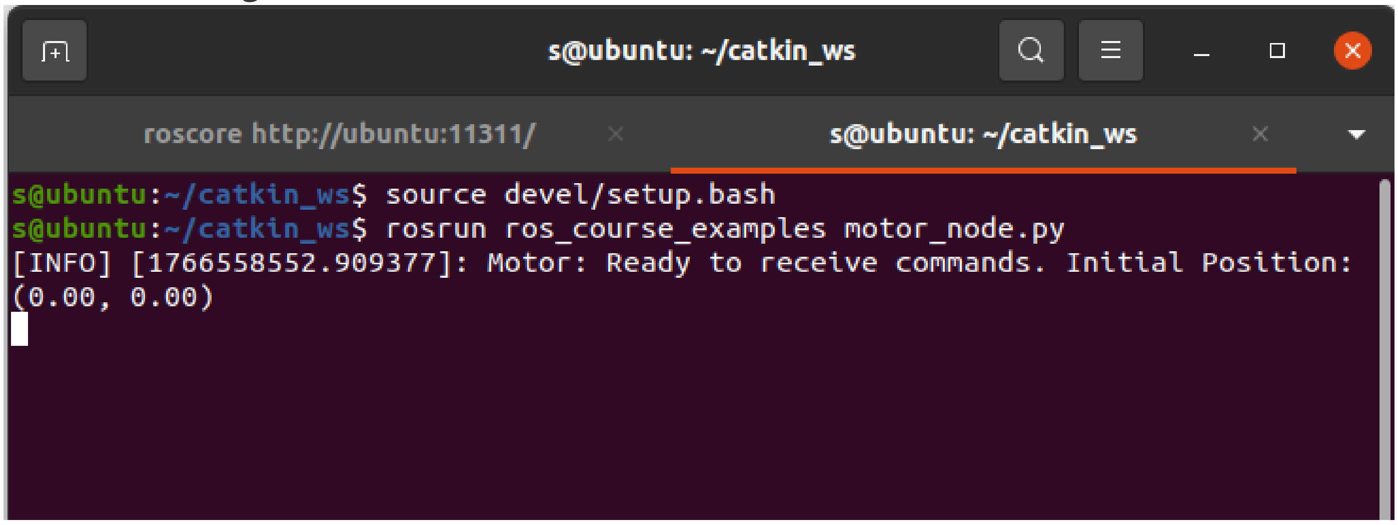
- If the executable file requires arguments, they can also follow directly: rosrun pkg exe arg1 arg2 ...

Terminal 2: Start the motor node.

Note: `source devel/setup.bash` helps ros commands find corresponding executable files.

```
cd ~/catkin_ws
source devel/setup.bash
rosrun ros_course_examples motor_node.py
```

You will see that this node (program) is waiting for information, and its position does not change.

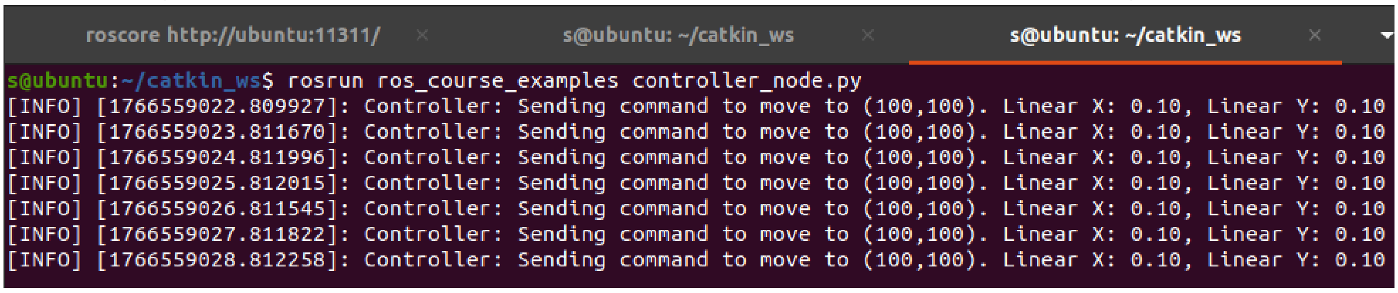


```
s@ubuntu:~/catkin_ws$ source devel/setup.bash
s@ubuntu:~/catkin_ws$ rosrun ros_course_examples motor_node.py
[INFO] [1766558552.909377]: Motor: Ready to receive commands. Initial Position:
(0.00, 0.00)
```

Terminal 3: Start the controller node. Next, let's control it.

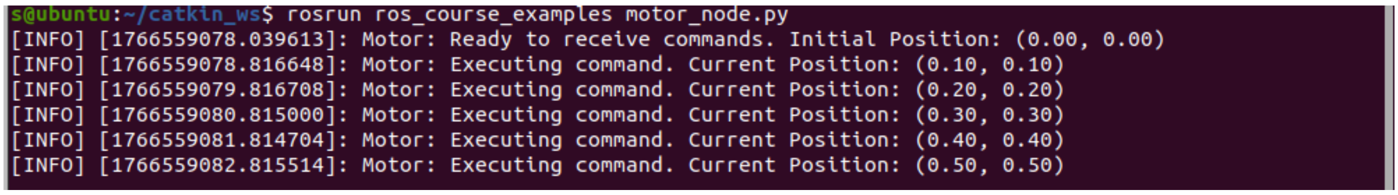
```
cd ~/catkin_ws
source devel/setup.bash
rosrun ros_course_examples controller_node.py
```

Observing the terminal, we can discover that it is constantly sending commands:



```
s@ubuntu:~/catkin_ws$ rosrun ros_course_examples controller_node.py
[INFO] [1766559022.809927]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559023.811670]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559024.811996]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559025.812015]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559026.811545]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559027.811822]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559028.812258]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
```

Return to view Terminal 2 (Motor node).



```
s@ubuntu:~/catkin_ws$ rosrun ros_course_examples motor_node.py
[INFO] [1766559078.039613]: Motor: Ready to receive commands. Initial Position: (0.00, 0.00)
[INFO] [1766559078.816648]: Motor: Executing command. Current Position: (0.10, 0.10)
[INFO] [1766559079.816708]: Motor: Executing command. Current Position: (0.20, 0.20)
[INFO] [1766559080.815000]: Motor: Executing command. Current Position: (0.30, 0.30)
[INFO] [1766559081.814704]: Motor: Executing command. Current Position: (0.40, 0.40)
[INFO] [1766559082.815514]: Motor: Executing command. Current Position: (0.50, 0.50)
```

Observed:The Controller sends speed commands, and the Motor receives them and updates the position.

However, wouldn't it be too complicated to enter a command line every time a program is started?

ROS provides a unified startup configuration mechanism that can start multiple nodes with one click; this is the `roslaunch` command.

`roslaunch`:

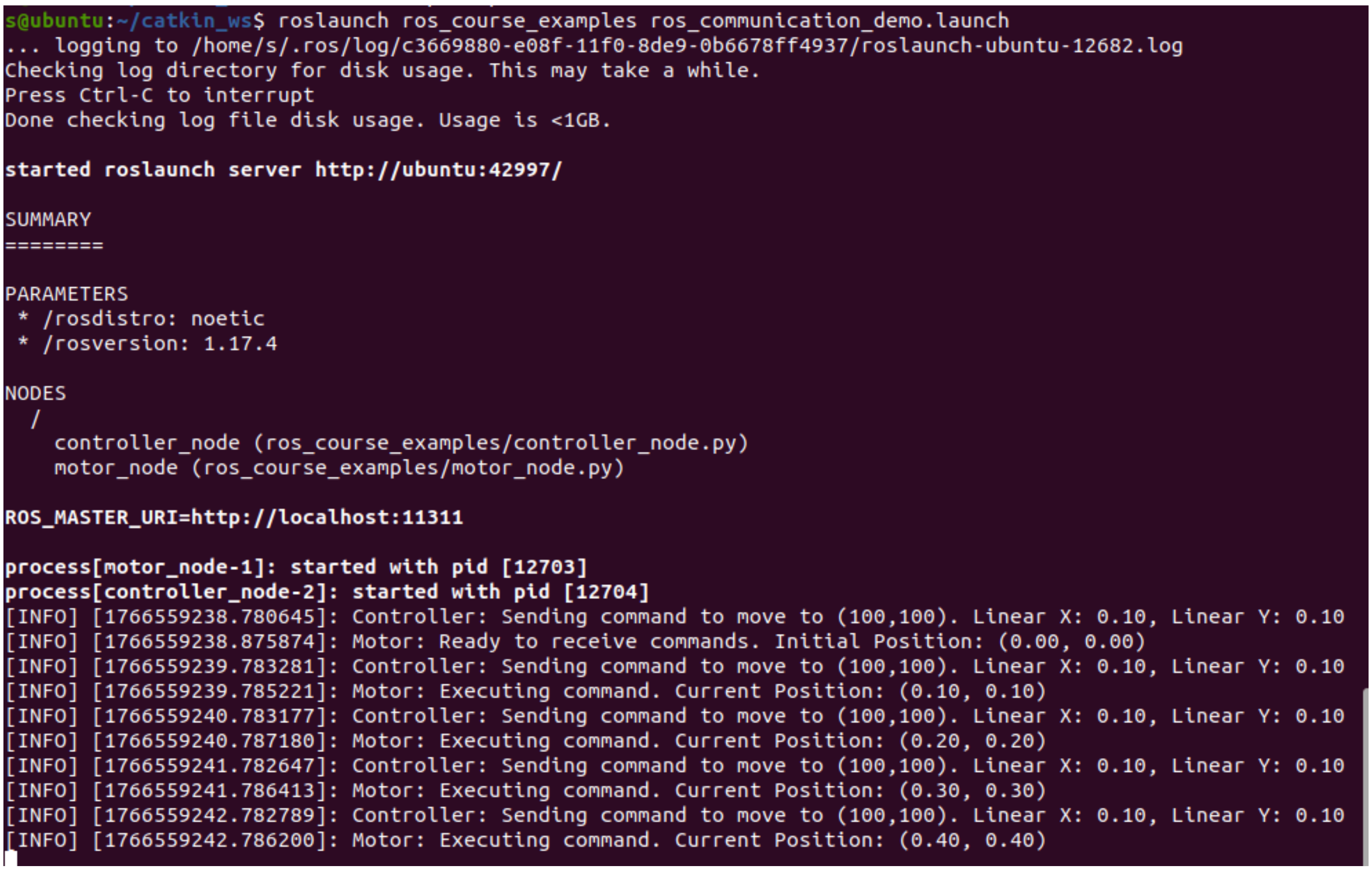
- Function: Start multiple nodes at once through .launch files.
- Set parameters for nodes, for example, setting parameters for relevant planning algorithms according to the size of the robot.
- Remap topics, suitable for changing communication relationships without changing code.
- Set namespaces, suitable for multiple robots using the same code.
- Launch files are usually located in the launch/ directory of the package (not mandatory, but convention).
- Example:

```
# roslaunch <ros package name> <launch file name within package>
roslaunch turtlesim turtlesim_demo.launch
```

Close Terminal 2 and Terminal 3, execute in Terminal 3:

```
cd ~/catkin_ws
source devel/setup.bash
roslaunch ros_course_examples ros_communication_demo.launch
```

You can observe that both nodes have started.



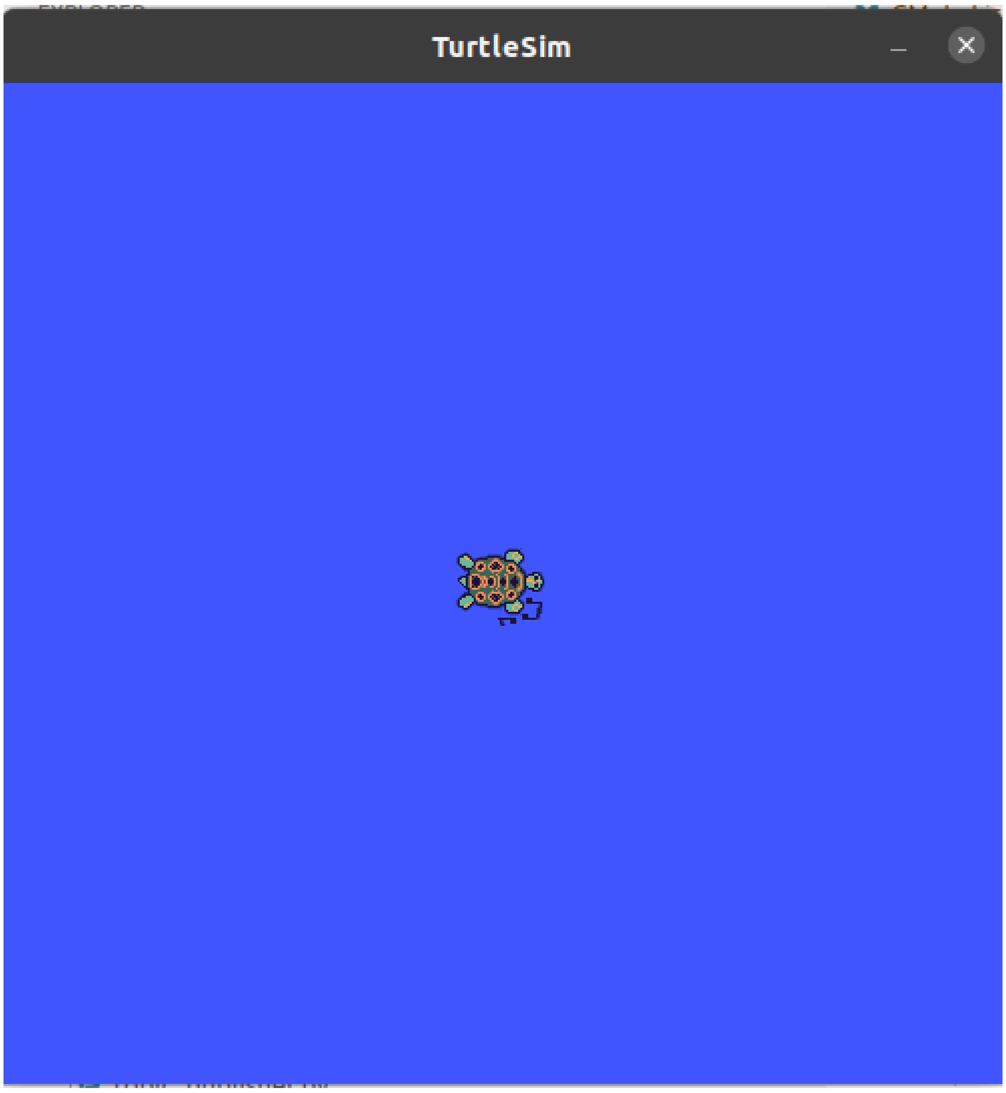
```
s@ubuntu:~/catkin_ws$ roslaunch ros_course_examples ros_communication_demo.launch
... logging to /home/s/.ros/log/c3669880-e08f-11f0-8de9-0b6678ff4937/roslaunch-ubuntu-12682.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:42997/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.17.4

NODES
  /
    controller_node (ros_course_examples/controller_node.py)
    motor_node (ros_course_examples/motor_node.py)

ROS_MASTER_URI=http://localhost:11311

process[motor_node-1]: started with pid [12703]
process[controller_node-2]: started with pid [12704]
[INFO] [1766559238.780645]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559238.875874]: Motor: Ready to receive commands. Initial Position: (0.00, 0.00)
[INFO] [1766559239.783281]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559239.785221]: Motor: Executing command. Current Position: (0.10, 0.10)
[INFO] [1766559240.783177]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559240.787180]: Motor: Executing command. Current Position: (0.20, 0.20)
[INFO] [1766559241.782647]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559241.786413]: Motor: Executing command. Current Position: (0.30, 0.30)
[INFO] [1766559242.782789]: Controller: Sending command to move to (100,100). Linear X: 0.10, Linear Y: 0.10
[INFO] [1766559242.786200]: Motor: Executing command. Current Position: (0.40, 0.40)
```

Before proceeding to the following experiments, you can close this program first (**Note: do not close the terminal running roscore**).

Experiment 2.3: TurtleSim Communication Experiment

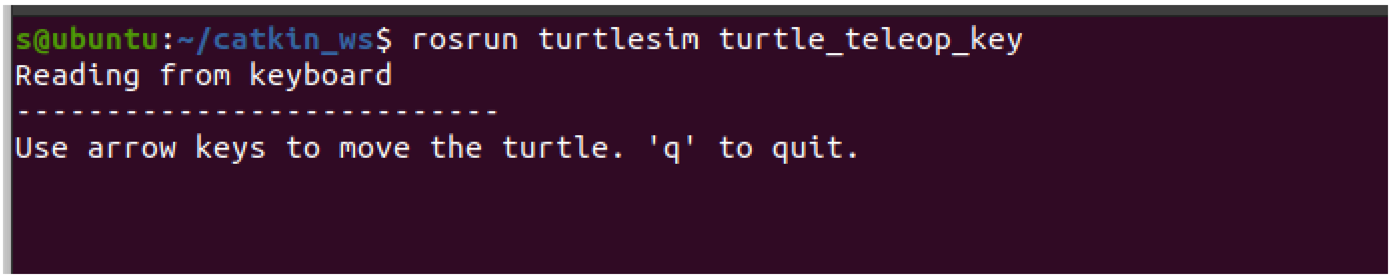
Terminal 2: Start Turtle Simulation

```
rosrun turtlesim turtlesim_node
```



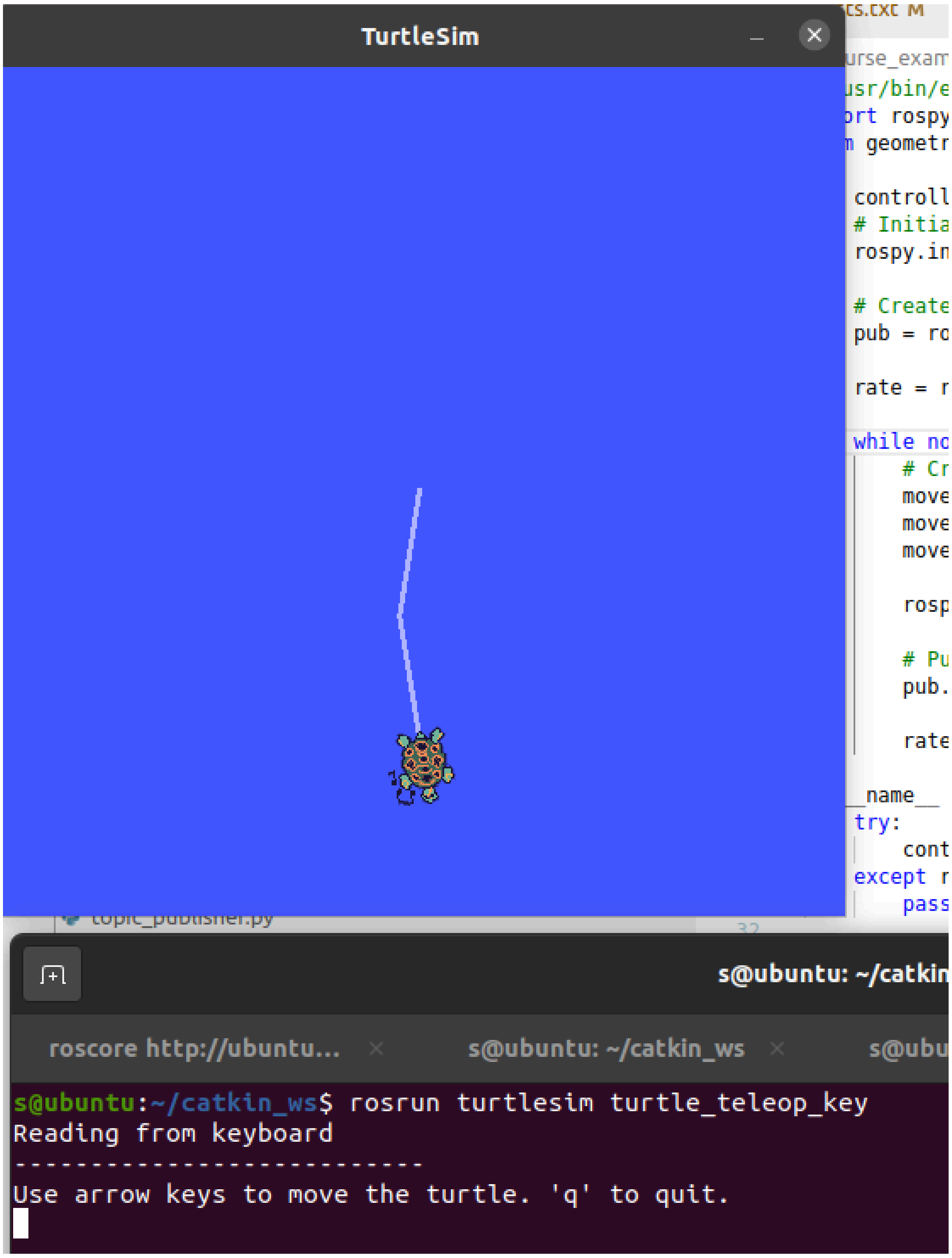
Terminal 3: Start Keyboard Control

```
rosrun turtlesim turtle_teleop_key
```



```
s@ubuntu:~/catkin_ws$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle. 'q' to quit.
```

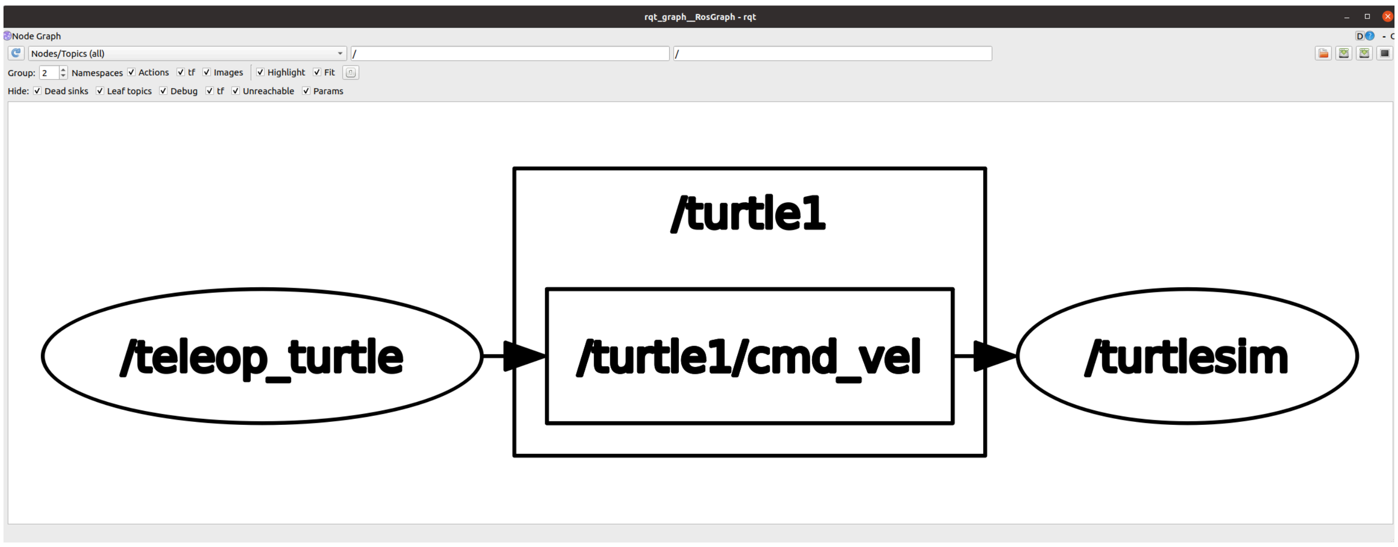
Pressing arrow keys in Terminal 3 can control the turtle movement.



```
s@ubuntu:~/catkin_ws$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle. 'q' to quit.
```

Terminal 4: Visualize Communication Graph

```
rqt_graph
```



Communication Process Analysis :

1. `turtle_teleop_key` node listens for keyboard input.
2. Publishes speed commands to the `/turtle1/cmd_vel` topic.
3. The information published to the topic is linear speed and angular speed; the message type is `geometry_msgs/Twist`.
4. `turtlesim_node` subscribes to `/turtle1/cmd_vel`.
5. Receives speed commands and executes movement.

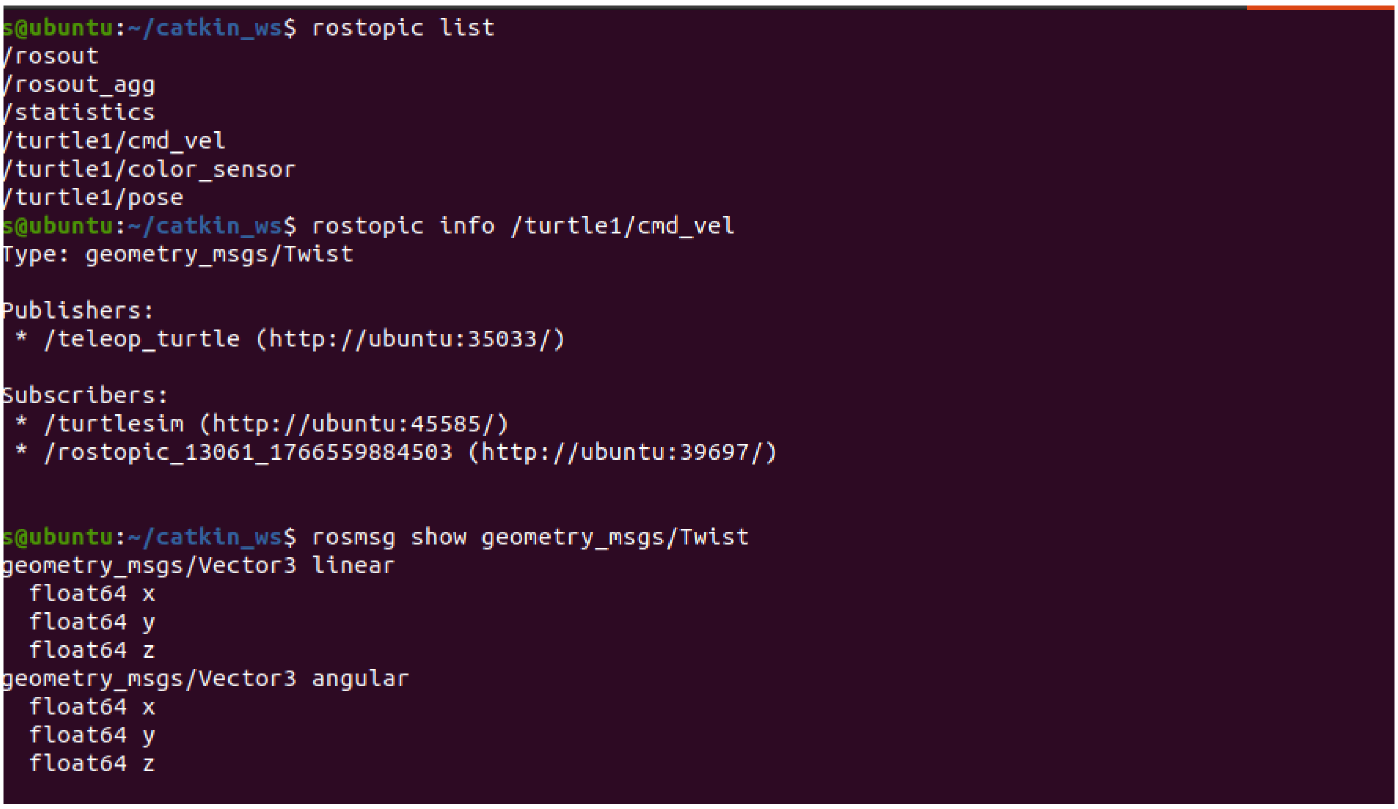
Experiment 2.4: Viewing Topic Information

All the above information can be observed through ROS commands.

```
# List all topics
rostopic list
```

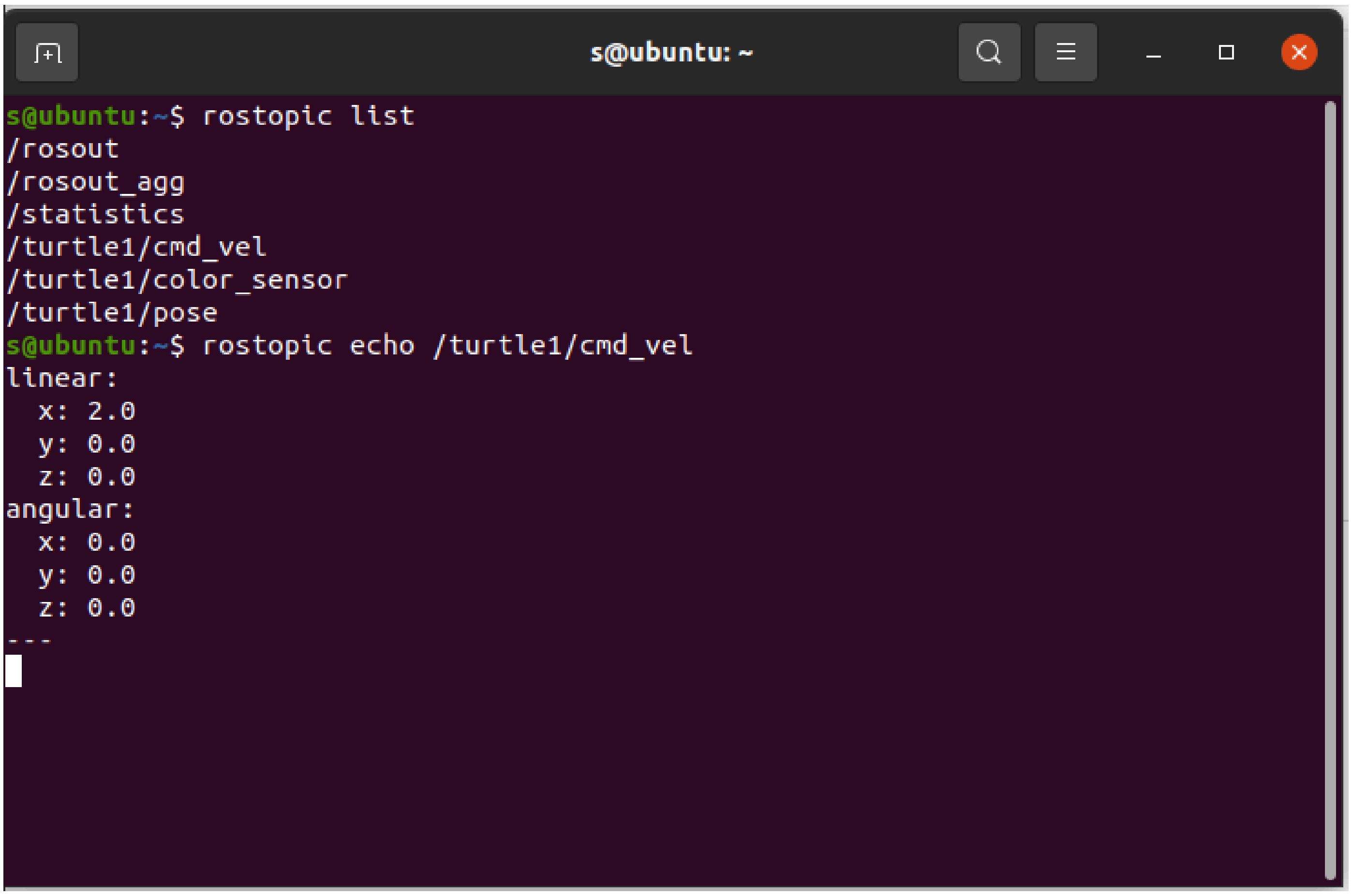
```
# View topic information
rostopic info /turtle1/cmd_vel
```

```
# View message type definition
rosmsg show geometry_msgs/Twist
```



```
s@ubuntu:~/catkin_ws$ rostopic list
/rosout
/rosout_agg
/statistics
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
s@ubuntu:~/catkin_ws$ rostopic info /turtle1/cmd_vel
Type: geometry_msgs/Twist

Publishers:
 * /teleop_turtle (http://ubuntu:35033/)

Subscribers:
 * /turtlesim (http://ubuntu:45585/)
 * /rostopic_13061_1766559884503 (http://ubuntu:39697/)


s@ubuntu:~/catkin_ws$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

```
# View topic data
rostopic echo /turtle1/cmd_vel
```



```
s@ubuntu:~$ rostopic list
/rosout
/rosout_agg
/statistics
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
s@ubuntu:~$ rostopic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Experiment 3: Gazebo Simulation Environment

1. Experiment Objectives

- Master the use of the Gazebo simulator
- Learn to load and save simulation worlds
- Understand the World file structure
- Master the loading of robot models
- Message Type

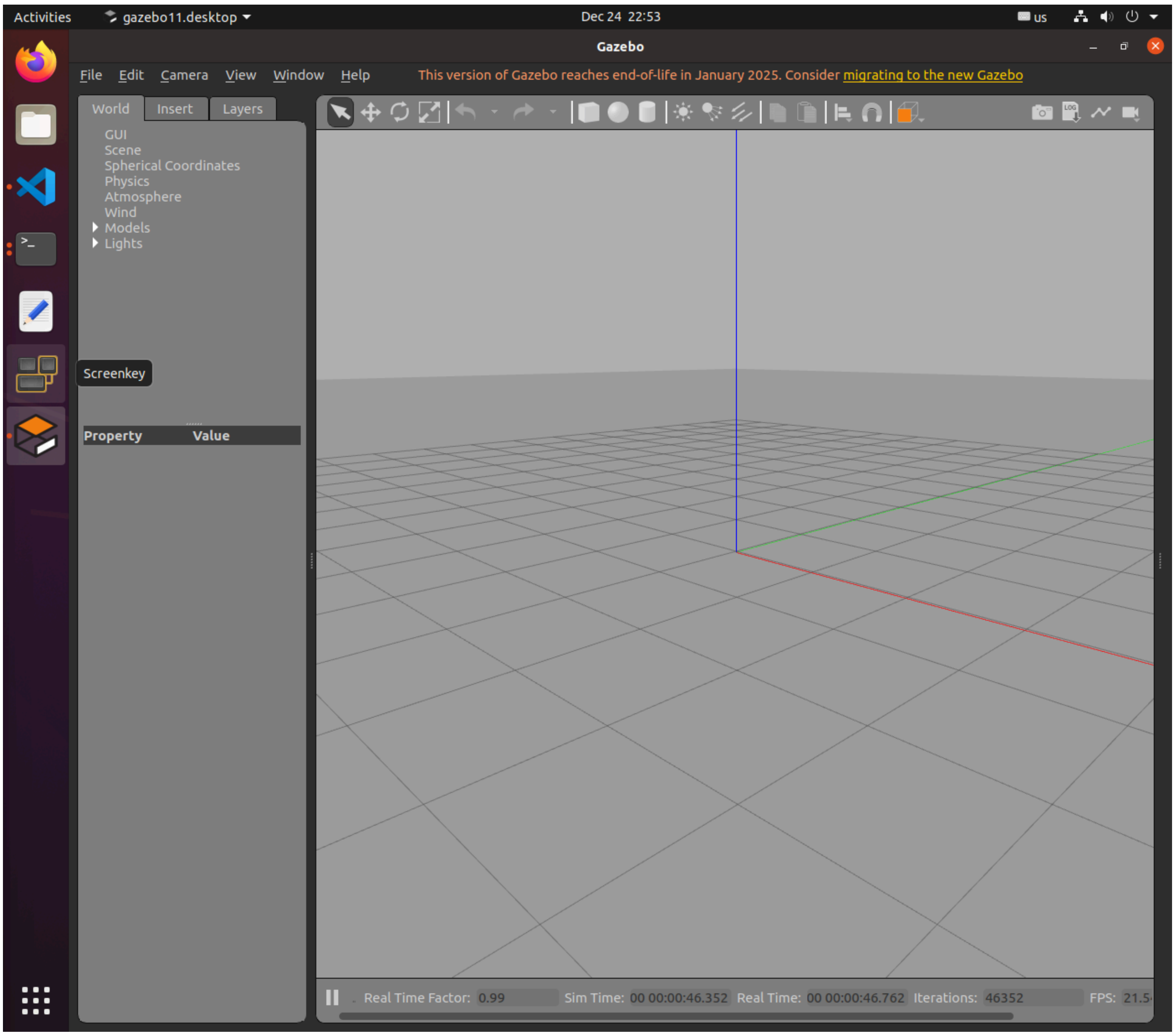
2. Gazebo Core Functions

- **Physics Engine** : Simulates real physical laws (gravity, collisions, friction).
- **Sensor Simulation** : LiDAR, cameras, IMU, etc.
- **ROS Integration** : Seamless communication with ROS.
- **Visualization** : 3D scene rendering.

3. Experiment Steps

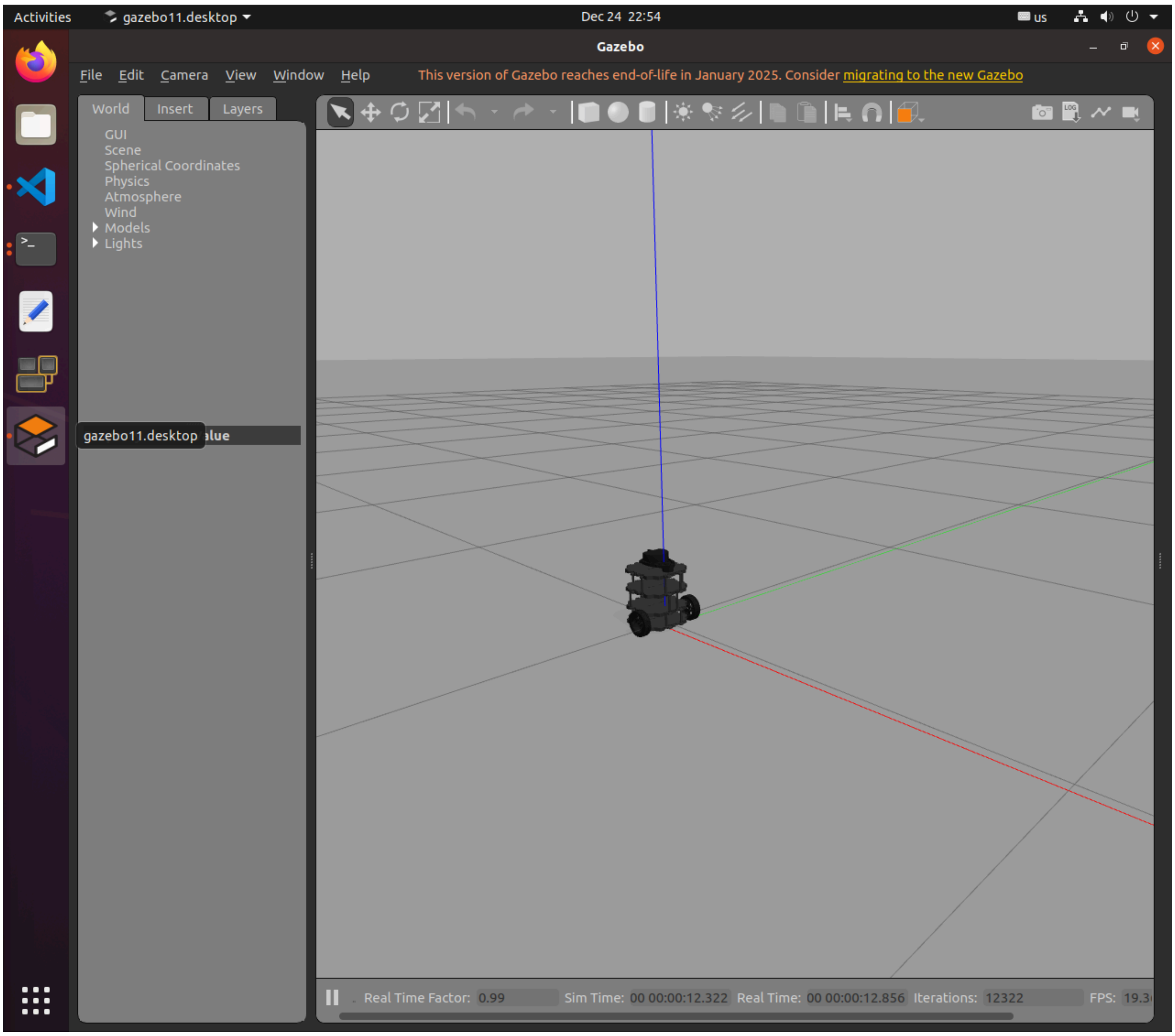
3.1 Start Empty World

```
gazebo
```



Or start using ROS

```
source ~/catkin_ws/devel/setup.bash
export TURTLEBOT3_MODEL=waffle
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

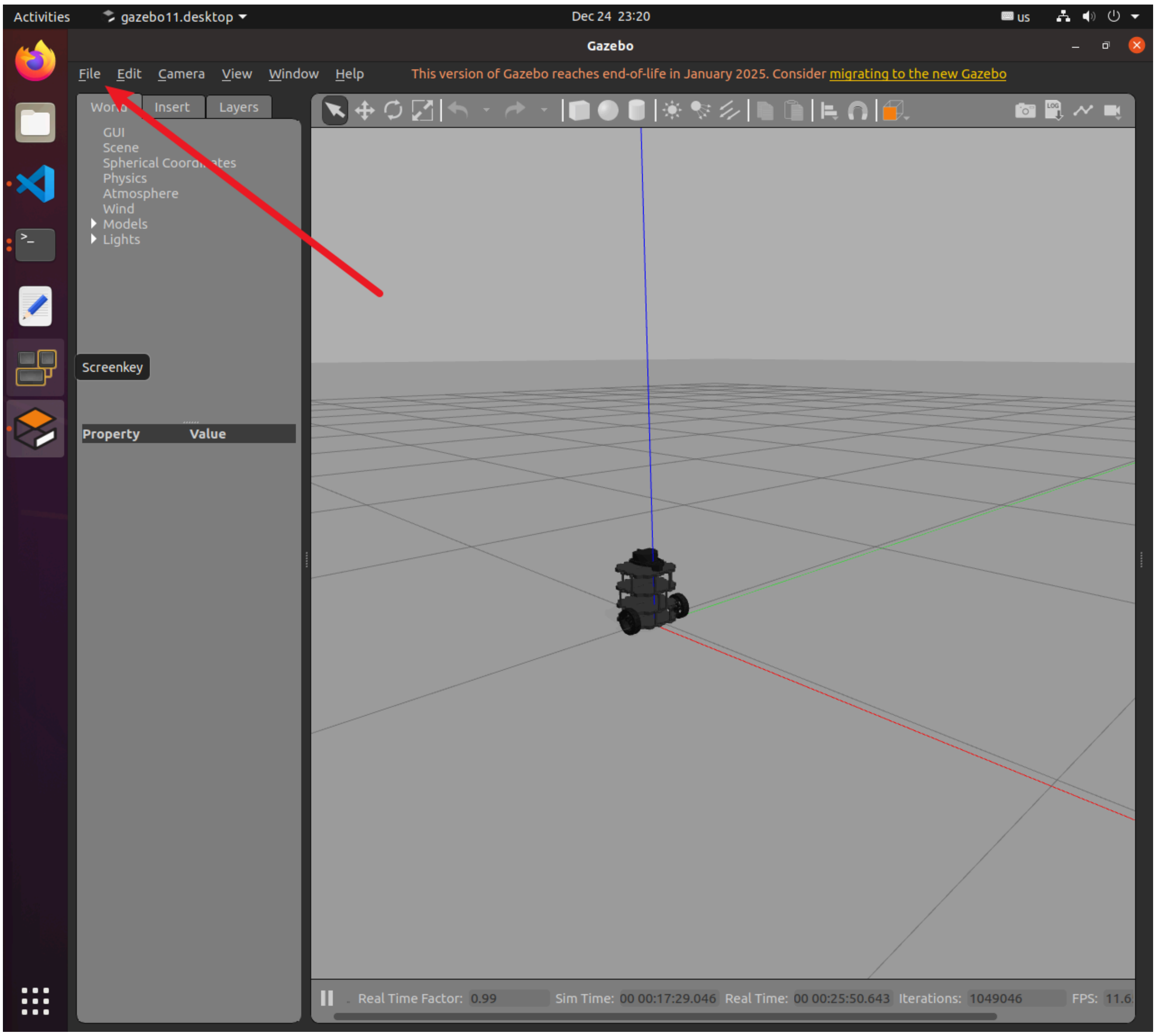


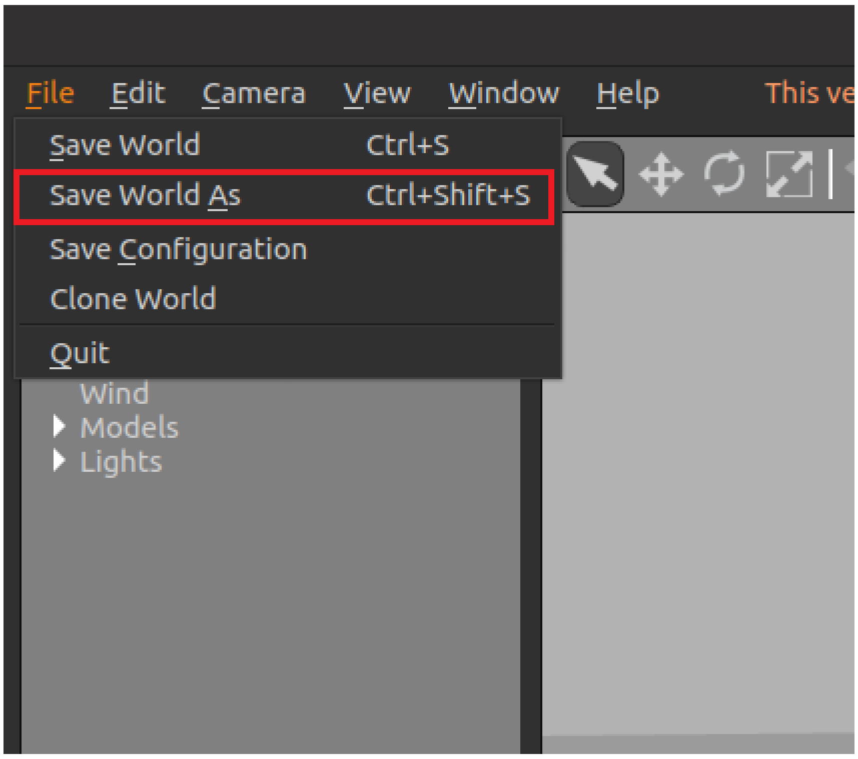
3.2 Build Custom Scene

1. **Insert Models** : Drag objects from the left panel into the scene.
2. **Adjust Parameters** :

- Position (x, y, z): Position coordinates
- Orientation (roll, pitch, yaw): Attitude angles
- Scale: Zoom size

1. **Save World** : `File -> Save World As -> my_world.world`



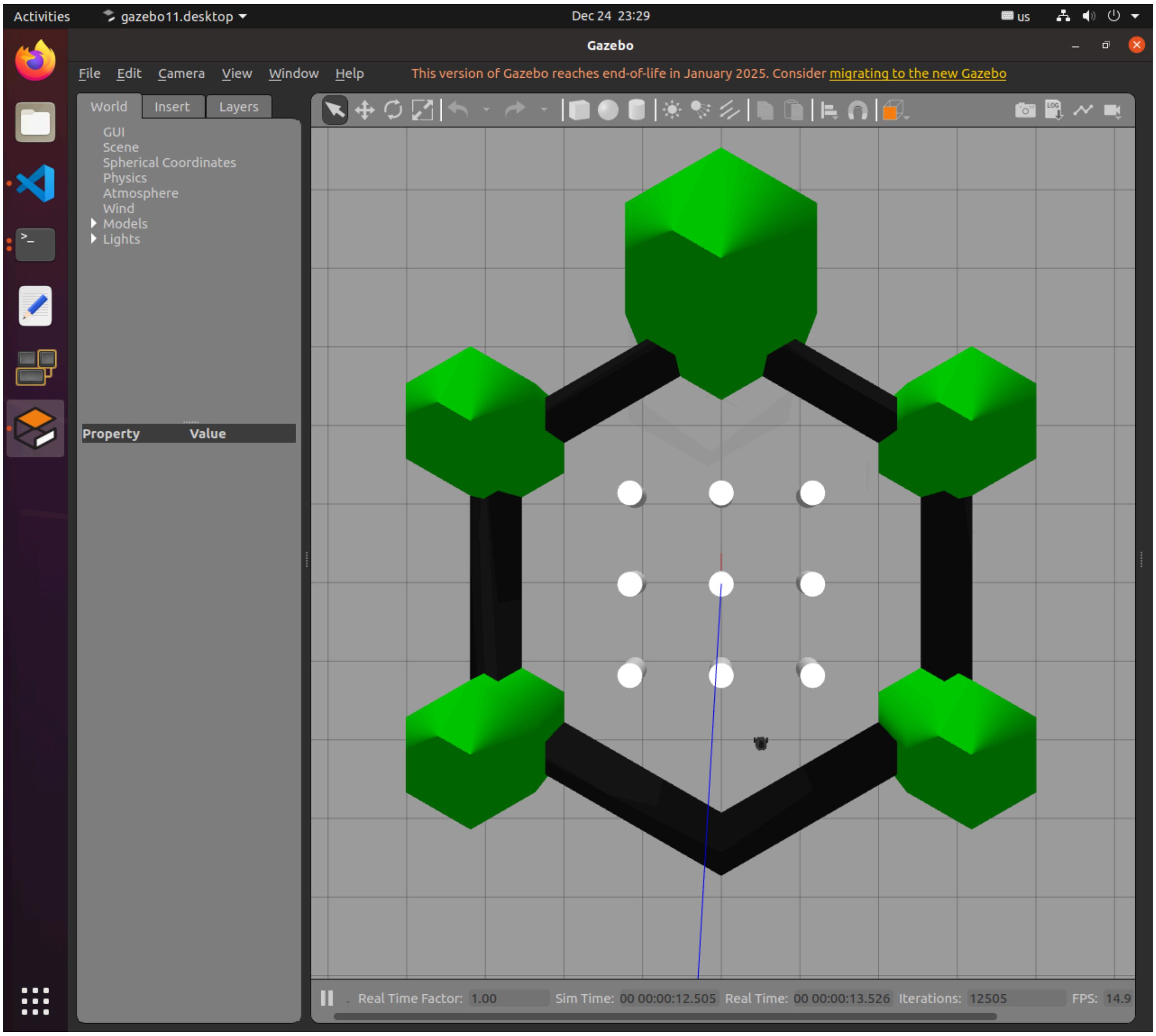


3.3 Load Custom World

```
gazebo my_world.world
```

Method 2: ROS launch file

```
source ~/catkin_ws/devel/setup.bash
export TURTLEBOT3_MODEL=waffle
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```



This version of Gazebo reaches end-of-life in January 2025. Consider migrating to the new Gazebo

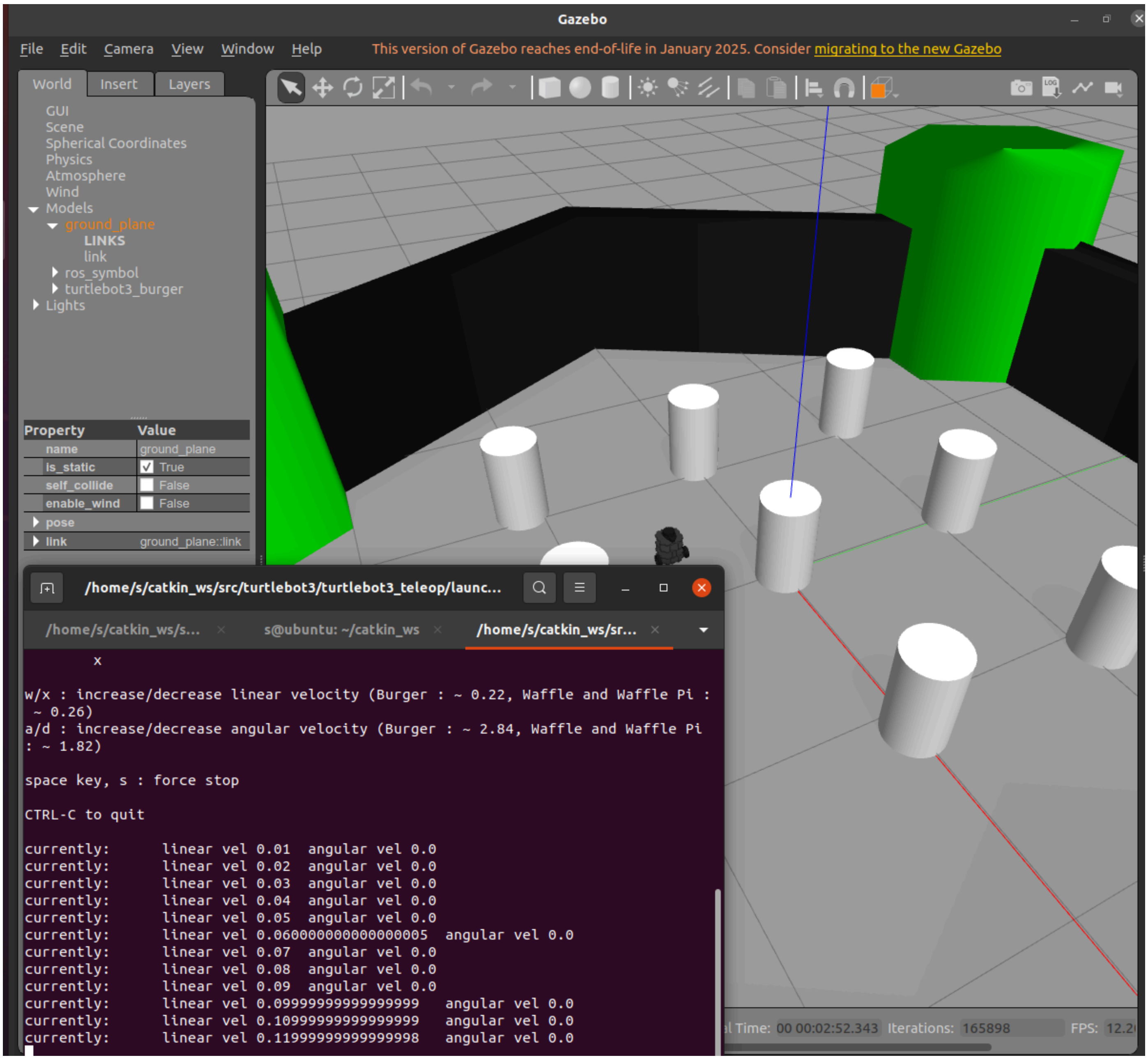
```
Real Time Factor: 1.00   Sim Time: 00 00:00:12.505   Real Time: 00 00:00:13.526   Iterations: 12505   FPS: 14.9
```

3.4 Control Robot Movement

```
source ~/catkin_ws/devel/setup.bash
export TURTLEBOT3_MODEL=waffle
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Operating Instructions :

- `W/A/S/D` or `Arrow Keys`: Control movement
- `X`: Stop
- `Q/Z`: Increase/decrease speed



Property	Value
name	ground_plane
is_static	✓ True
self_collide	False
enable_wind	False
pose	
link	ground_plane::link

```
        x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi :
 ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi
: ~ 1.82)

space key, s : force stop

CTRL-C to quit

currently:      linear vel 0.01  angular vel 0.0
currently:      linear vel 0.02  angular vel 0.0
currently:      linear vel 0.03  angular vel 0.0
currently:      linear vel 0.04  angular vel 0.0
currently:      linear vel 0.05  angular vel 0.0
currently:      linear vel 0.060000000000000005  angular vel 0.0
currently:      linear vel 0.07  angular vel 0.0
currently:      linear vel 0.08  angular vel 0.0
currently:      linear vel 0.09  angular vel 0.0
currently:      linear vel 0.09999999999999999  angular vel 0.0
currently:      linear vel 0.10999999999999999  angular vel 0.0
currently:      linear vel 0.11999999999999998  angular vel 0.0
```

```
l Time: 00 00:02:52.343  Iterations: 165898  FPS: 12.2
```

Experiment 4: RViz Visualization

1. Experiment Objectives

- Master the use of the RViz visualization tool

- Learn to add and configure display items
- Understand visual representation of sensor data
- Master interface interaction and viewpoint control

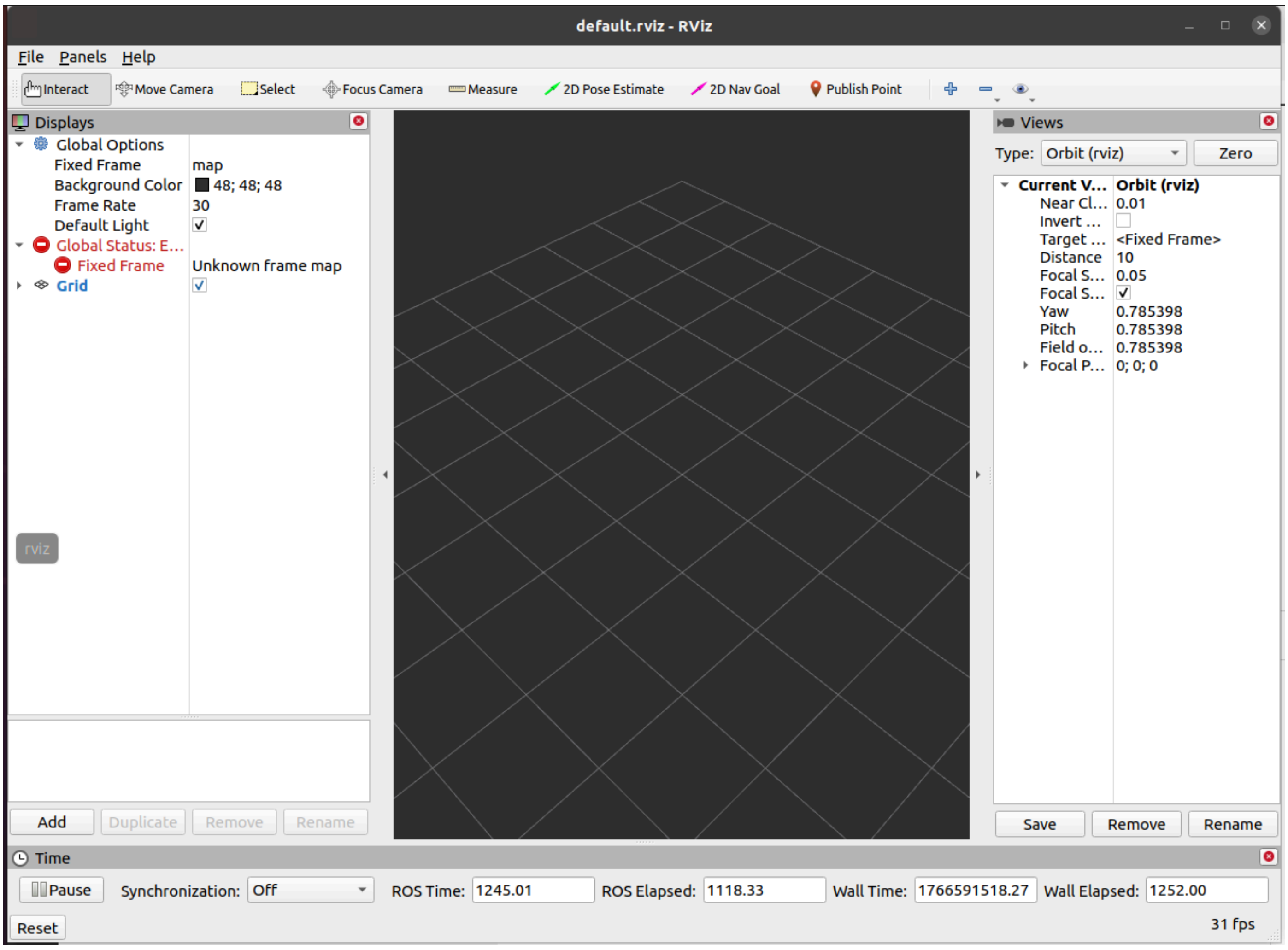
2. Data Types Displayable by RViz

- **Robot Model** : 3D robot model
- **LaserScan** : LiDAR scan data
- **PointCloud2** : 3D point cloud data
- **TF** : Coordinate system transformation relationships
- **Image** : Camera images
- **Odometry** : Odometry trajectory
- **Path** : Planned path
- **Map** : Occupancy grid map

3. Introduction to RViz Startup Methods

Start directly

```
rviz
```


default.rviz - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Displays
Views
rviz
Add Duplicate Remove Rename
Save Remove Rename
Zero
Time
Pause
Reset

Global Options
Fixed Frame map
Background Color ■ 48; 48; 48
Frame Rate 30
Default Light ✓
Global Status: E...
Fixed Frame Unknown frame map
Grid ✓

Type: Orbit (rviz)
Current V... Orbit (rviz)
Near Cl... 0.01
Invert ...
Target ... <Fixed Frame>
Distance 10
Focal S... 0.05
Focal S... ✓
Yaw 0.785398
Pitch 0.785398
Field o... 0.785398
Focal P... 0; 0; 0

Synchronization: Off ROS Time: 1245.01 ROS Elapsed: 1118.33 Wall Time: 1766591518.27 Wall Elapsed: 1252.00

31 fps

3.1 Basic RViz Interface Operations

Viewpoint Control

- **Left mouse drag** : Rotate view
- **Mouse wheel** : Zoom view
- **Shift + Left drag** : Pan view
- **Shift + Wheel** : Pan up/down
- **Middle mouse drag** : Pan (some systems)

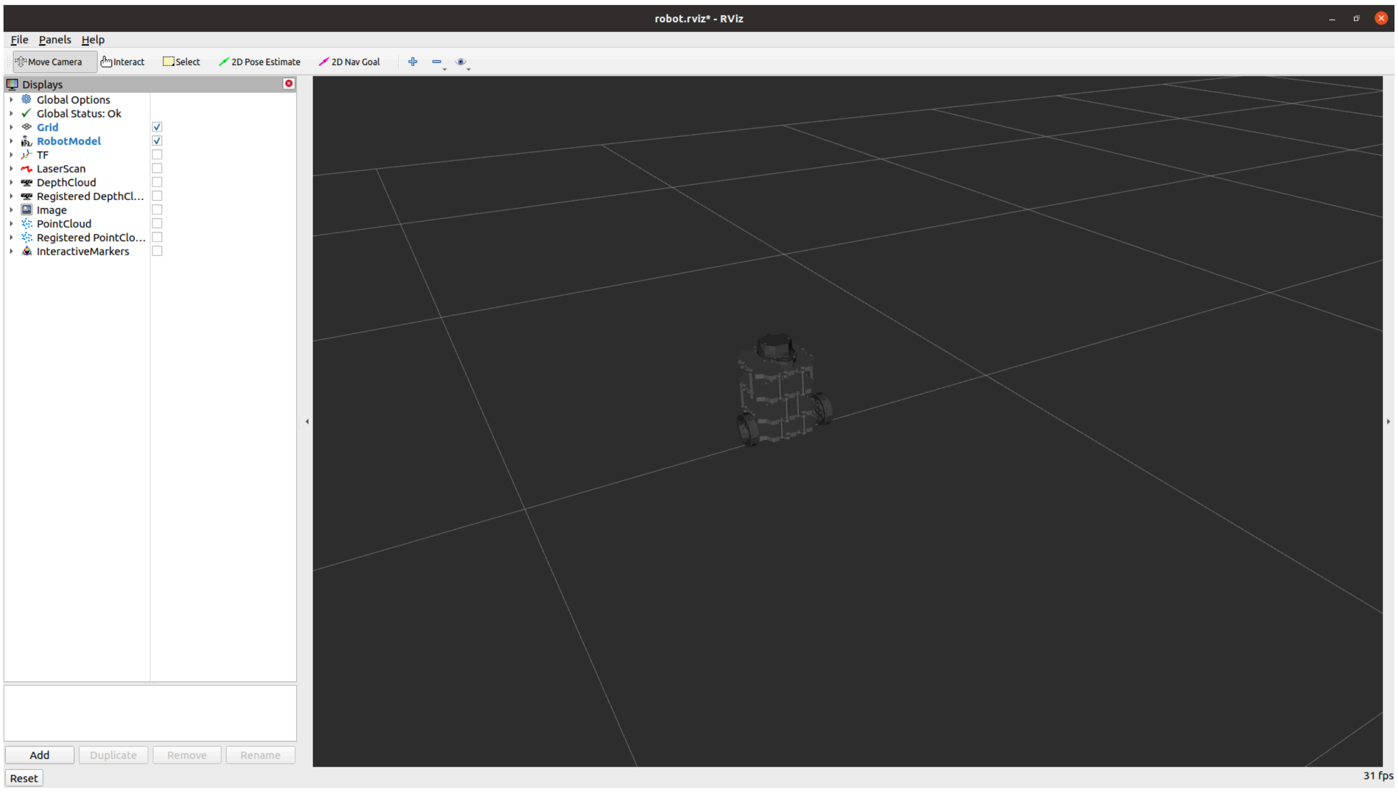
4. Experiment Steps

4.1 Start Simulation and Visualization

```
# Terminal 1: Start gazebo
source ~/catkin_ws/devel/setup.bash
```

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

```
# Terminal 2: Start RViz
source ~/catkin_ws/devel/setup.bash
roslaunch turtlebot_rviz_launchers view_robot.launch
```



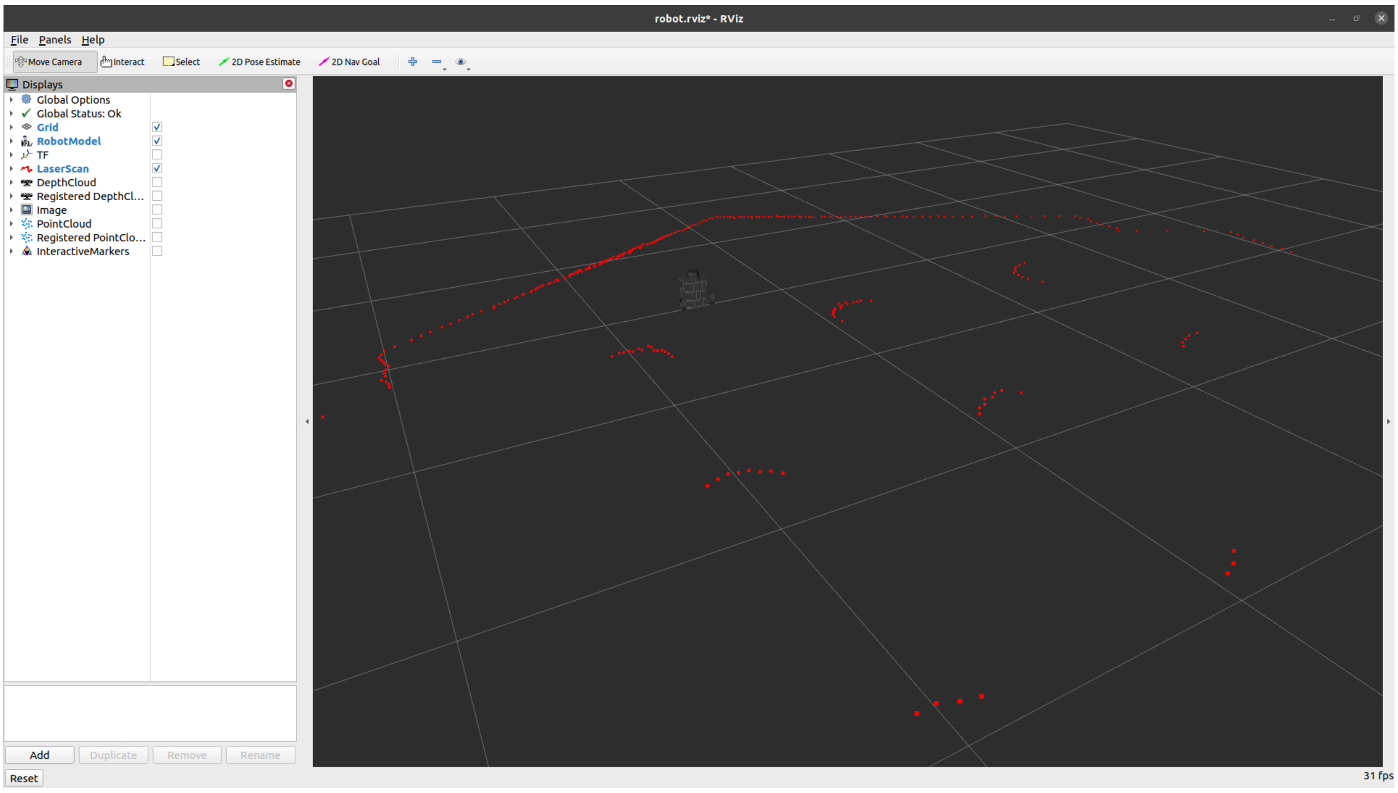
4.2 Detailed Steps to Add Display Items (Display)

Example 1: Adding LiDAR Data (Or just check the LaserScan box, like if the rviz launch already configured rviz)

1. Click the **"Add"** button in the bottom left corner.
2. Select the **"By display type"** tab in the pop-up window.
3. Find and double-click **"LaserScan"** .
4. Expand **"LaserScan"** in the left Displays panel.
5. Configure parameters:

```
   Topic: /scan           # Click dropdown to select /scan
   Size (m): 0.05         # Adjust point size
   Style: Points          # Display style
   Color Transformer: Intensity  # Color map
```

6. Observe red scan points displaying obstacle positions.

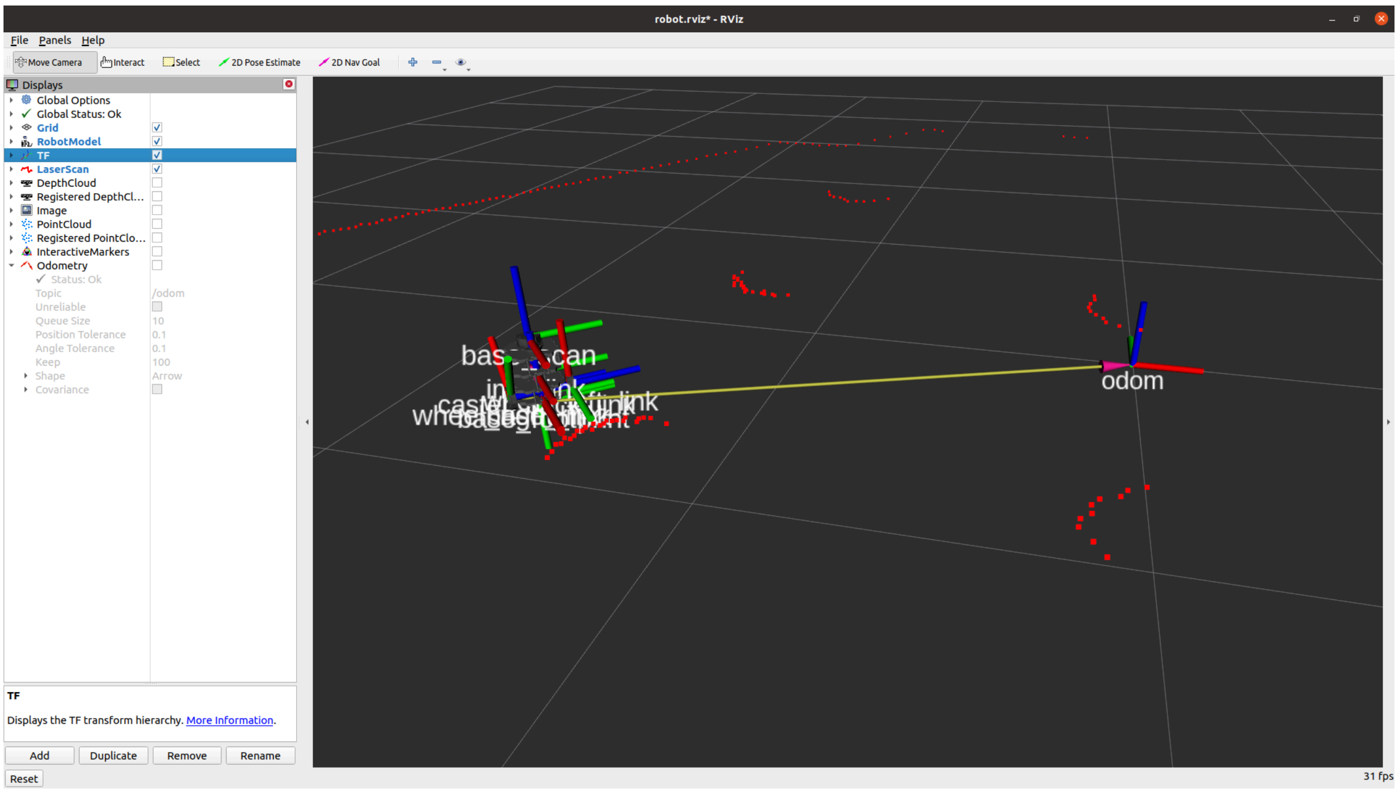


Example 2: Adding Coordinate Systems (TF)

1. Click **"Add"** -> Select **"TF"** .
2. Configure parameters:

```
     Show Axes            # Show axes
     Show Arrows          # Show arrows
   Marker Scale: 0.5      # Adjust axes size
   Update Interval: 0     # Update interval (0=fastest)
```

3. Observe red, green, and blue arrows (representing X/Y/Z axes).



```
Displays
  Global Options
  Global Status: Ok
  Grid                      ✓
  RobotModel                ✓
  TF                        ✓
  LaserScan                 ✓
  DepthCloud
  Registered DepthCl...
  Image
  PointCloud
  Registered PointClo...
  InteractiveMarkers
  Odometry
    Status: Ok
    Topic                   /odom
    Unreliable
    Queue Size              10
    Position Tolerance      0.1
    Angle Tolerance         0.1
    Keep                    100
    Shape                   Arrow
    Covariance
```

TF

Displays the TF transform hierarchy. More Information.

4.3 Configuration Table for Common Display Items

Display Type	Recommended Topic	Function	Key Parameters
RobotModel	(Default)	Displays 3D robot model	`Robot Description: robot_description`
LaserScan	`/scan`	LiDAR scan data	`Size: 0.05`,`Style: Points`
Odometry	`/odom`	Odometry trajectory	`Keep: 100`,`Shape: Arro`
TF	(No setting needed)	Coordinate system relationships	`Show Names: `,`Marker Scale: 0.5`
Map	`/map`	Occupancy grid map	`Color Scheme: map`

4.4 Adjust Fixed Frame (Reference Coordinate System)

What is Fixed Frame?

- The display of all data in RViz requires a reference coordinate system.
- Different scenarios require selecting different Fixed Frames.

Selection Suggestions

Scenario	Fixed Frame	Effect
Observing robot movement	`odom`	Viewpoint follows robot
Debugging sensors	`base_link`	Viewpoint locked onto the robot

Setting Method

1. Expand **"Global Options"** at the top.
2. Click the **"Fixed Frame"** dropdown menu.
3. Select `odom`.

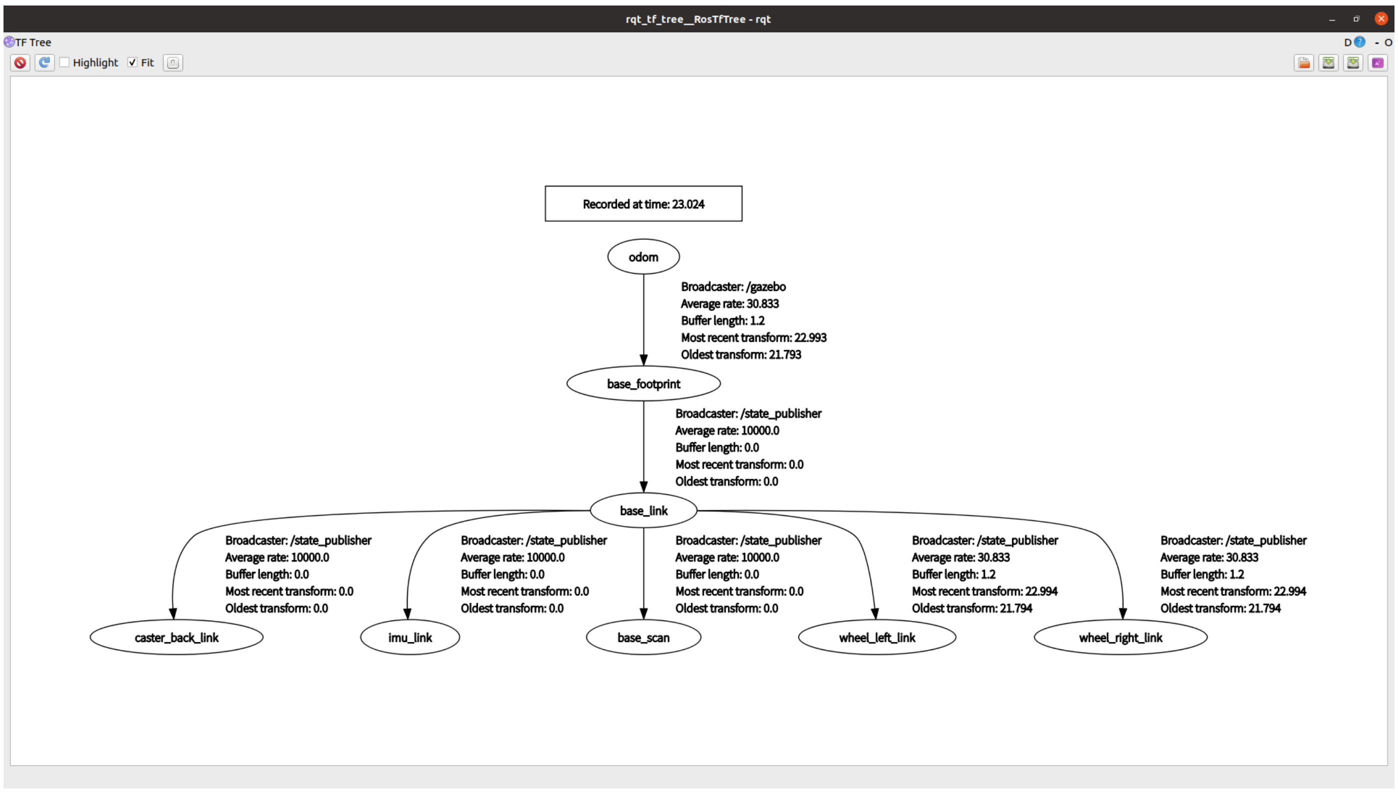
Note : If Fixed Frame is set incorrectly, all display items will turn gray or not show.

4.5 View TF Tree

Command to view TF tree diagram

```
rosrun rqt_tf_tree rqt_tf_tree
```

View TF Transforms



6. Common Troubleshooting

Issue	Cause	Solution
Display items turn red/gray	Topic not published	`rostopic list`Check if topic exists
Cannot see robot model	Fixed Frame error	Change to `odom` or `base_link`
Laser data does not show	Wrong Topic selected	Confirm it is `/scan`
TF shows "No transform"	TF tree incomplete	Check `rosrun tf view_frames`

Suggestions for After-class Practice

- Practice Linux command-line operations more.
- Try modifying sample code parameters and observe the effects.
- Use `rqt_graph` and `rostopic` tools to analyze the system.
- Read the official ROS Wiki documentation.

Recommended Resources

- ROS Wiki
- Gazebo Tutorials
- TF Tutorials