LAB 2

Get Started with ROS

2.1 Important

Before this lab, please Consult **Appendix** A of this lab manual for details of ROS Concepts (**Node**, **Topic**, **Message**).

2.2 Objectives

The purpose of this lab is to familiarize you with the ROS environment and its basic concepts of Node, Topic and Message through a simple turtle simulator. In this lab, you will:

- Learn how to show basic node, message and topic information.
- Learn how to publish and subscribe Messages to Topics.

2.3 References

- "*A Gentle Introduction to ROS*", Chapter 2 and 3. http://coecsl.ece.illinois.edu/ece470/agitr-letter.pdf
  - Chapter 2: 2.4 Packages, 2.5 The Master, 2.6 Nodes, 2.7.2 Messages and message types.
  - Chapter 3 Writing ROS programs.
- Turtlesim Documentation in ROS community. http://wiki.ros.org/turtlesim

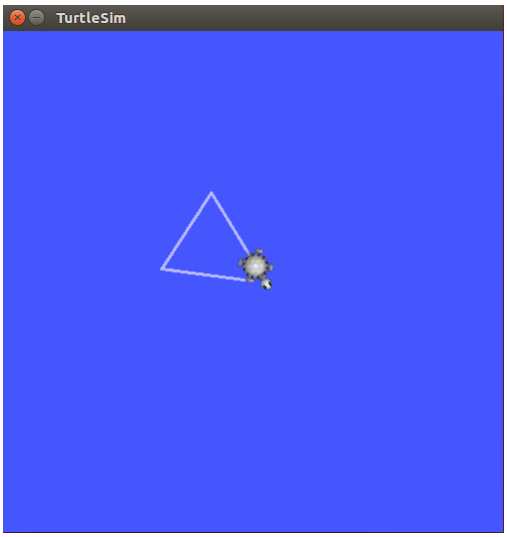

Figure 2.1: The turtlesim window

2.4 Procedure

1. **Starting** turtlesim: In three separate terminals, execute these three commands
   Hints: ctrl+alt+T(to start a new window), ctrl+shift+W(to close current window), ctrl+shift+E(split window vertically), ctrl+shift+O(split window horizontally)

   ```
   $ roscore
   $ rosrun turtlesim turtlesim_node
   $ rosrun turtlesim turtle_teleop_key
   ```

   If everything works correctly, you should see a graphical window similar to Figure 2.1. This window shows a simulated, turtle-shaped robot that lives in a square world. If you give your third terminal (the one executing the turtle_teleop_key command) the input focus and press the Up, Down, Left, or Right keys, the turtle will move in response to your commands, leaving a trail behind it.
   Making virtual turtles draw lines is not, in itself, particularly exciting. However, this example already has enough happening behind the scenes to illustrate many of the main ideas on which more interesting ROS systems are based.

2. **Node** is a running instance of a ROS program. The basic command to create a node (also known as "running a ROS program") is rosrun. ROS provides a few ways to get information about the nodes that are running at any particular time. To get a list of running nodes, try this command.

```
# rosrun package-name executable-name
$ rosnode list
```

If you do this in a new window, you will see a list of three nodes.

```
$ /rosout
$ /turtlesim
$ /teleop_turtle
```

The /rosout is a special node that is started automatically by roscore. Its purpose is somewhat similar to the standard output (i.e. std::cout).
The other two nodes should be fairly clear: They're the simulator (turtlesim) and the teleoperation program(teleop_turtle) we started ablove.
You can get some information about a particular node using this command:

```
$ rosnode info node-name
```

The output includes a list of **topics** for which that node is a publisher or subscriber, the services offered by that node, its Linux process identifier (PID), and a summary of the connections it has made to other nodes.

3. The primary mechanism that ROS nodes use to communicate is to send **Messages**. Messages in ROS are organized into named **Topics**. The idea is that a node that wants to share information will publish messages on the appropriate topic or topics; a node that wants to receive information will subscribe to the topic or topics that it's interested in. The ROS master takes care of ensuring that publishers and subscribers can find each other. This idea is probably easiest to see graphically, and the easiest way to visualize the publish-subscribe relationships between ROS nodes is to use this command:

   ```
   $ rqt_graph
   ```

   In this case, you will see something like Figure 2.2. In this graph, the ovals represent nodes, and the directed edges represent publisher-subscriber relationships. The graph tells us that the node named /teleop_turtle publishes messages on a topic called /turtle1/cmd_vel, and that the node named /turtlesim subscribes to those messages.
   Now we can understand at least part of how the turtlesim teleoperation system works. When you press a key, the /teleop_turtle node publishes messages with those movement commands on a topic called /turtle1/cmd_vel. Because it subscribes to that topic, the turtlesim_node receives those messages, and simulates the turtle moving with the requested velocity. The important points here are:

   - The simulator doesn't care (or even know) which program publishes those cmd_vel messages. Any program that publishes on that topic can control the turtle.

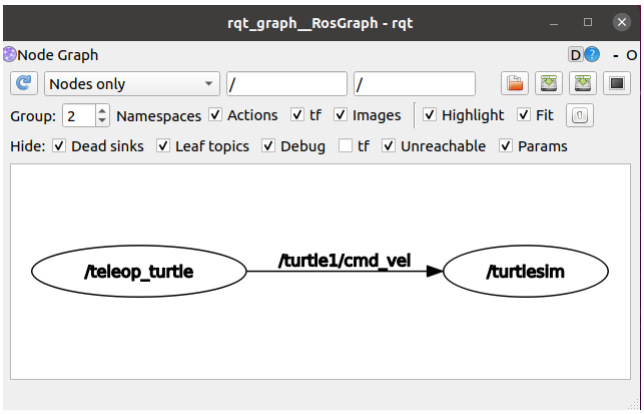

Figure 2.2: The rqt_graph interface, showing the graph for the turtlesim example

- The teleoperation program doesn't care (or even know) which program subscribes to the cmd_vel messages it publishes. Any program that subscribes to that topic is free to respond to those commands.

4. Next let's take a closer look at the topics and messages themselves. To get a list of active topics, use this command:

```
$ rostopic list
```

In our example, this shows a list of five topics:

```
$ /rosout
$ /rosout_agg
$ /turtle1/cmd_vel
$ /turtle1/color_sensor
$ /turtle1/pose
```

You can see the autual messages that are being published on a single topic using the rostopic command, take /turtle1/pose as an exmaple:

```
$ rostopic echo /turtle1/pose
```

your terminal will update pose data in real-time if you control the turtle by keyboard.

Also you can learn more about a topic using the rostopic info command:

```
$ rostopic info /turtle1/pose
```

You should see output similar to this:

```
Type: turtlesim/Pose
```

```
Publishers:
  * /turtlesim (http://rvc-ece470:43601/)

Subscribers: None
```

The most important part of this output is the very first line, which shows the **message type** of that topic. In the case of **/turtle1/pose**, the message type is **turtlesim/Pose**. The message type of a topic tells you what information is included in each message on that topic, and how that information is organized.

To see details about a message type, use a command like this:

```
$ rosmsg show message-type-name
```

Let's try using it on the message type for /turtle1/pose that we found above:

```
$ rosmsg show turtlesim/Pose
```

The output is:

```
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

The format is a list of fields, one per line. Each field is defined by a built-in data type (like int8, bool, or string) and a field name. The output above tells us that a turtlesim/Pose is a thing that contains five float32 items.

Another example, one we'll revisit several times, is geometry_msgs/Twist. This is the message type for the /turtle1/cmd_vel topic, and it is slightly more complicated:

```
geometry_msgs/Vector3 linear
        float64 x
        float64 y
        float64 z
geometry_msgs/Vector3 angular
        float64 x
        float64 y
        float64 z
```

In this case, both linear and angular are **composite fields** whose date type is geometry_msgs/Vector3. The indentation shows that fields named x, y, and z aremembers within those two top-level fields. That is, a message with type geometry_msgs/Twist contains exactly six numbers, organized into two vectors called linear and angular. Each of these numbers has the built-in type float64.

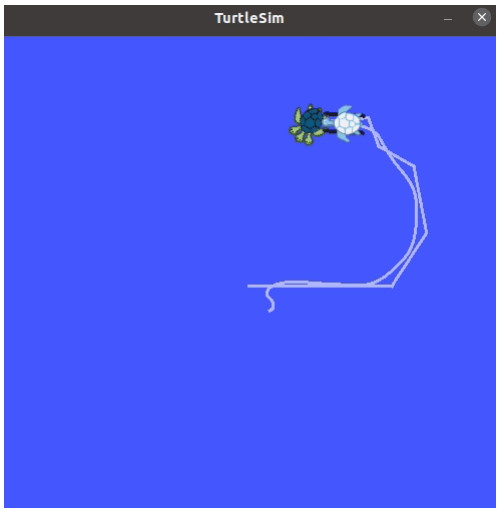

Figure 2.3: turtle chase and catch game

2.5 Task

In order to master above concepts, here is one simple practice for you. We will have two turtles, as is shown in Figure 2.3. One is leader and the other is follower. The leader turtle is controlled by keyboard input and the follower turtle should be programed to follow the trajectory. What you should do is to receive pose data of both turtles and publish command velocity to the follower.

1. Establish your own workspace as shown in Appendix A and all your programming work will be done here.
2. Copy package **lab2pkg_turtle** into your workspace. All your source code is in folder **ECE470_LAB** on your desktop. Do Not Modify it!
3. Complete your code and see how to run it in **README.md**.

2.6 Report

Each partner will submit a lab report using the guidelines given in the "ECE 470: How to Write a Lab Report" document. Please be aware of the following:

- Lab reports are due one week after the final session!
- Lab reports will be submitted online at BlackBoard.

2.7 Demo

Show your TA the program you created.

2.8 Grading

- 10 points, completed this section by the end of the two hour lab session.
- 70 points, successful demonstration.
- 20 points, report.

Appendix A

ROS Programming with Python

A.1 Overview

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.

- The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

- For more details about ROS: http://wiki.ros.org/
- How to install on your own Ubuntu: http://wiki.ros.org/ROS/Installation
- For detailed tutorials: http://wiki.ros.org/ROS/Tutorials
- "*A Gentle Introduction to ROS*", Chapter 2 and 3. http://coecsl.ece.illinois.edu/ece470/agitr-letter.pdf

A.2 ROS Concepts

The basic concepts of ROS are nodes, Master, messages, topics, Parameter Server, services, and bags. However, in this course, we will only be encountering the first four.

- `Nodes` "programs" or "processes" in ROS that perform computation. For example, one node controls a laser range-finder, one node controls he wheel motors, one node performs localization ...
- `Master` Enable nodes to locate one another, provides parameter server, tracks publishers and subscribers to topics, services. In order to start ROS, open a terminal and type:

  ```
  $ roscore
  ```

  roscore can also be started automatically when using roslaunch in terminal, for example:

  ```
  $ roslaunch <package name> <launch file name>.launch
  # the launch file for all our labs:
  $ roslaunch ur_robot_driver ur3e_bringup.launch
  ```

- `Messages` Nodes communicate with each other via messages. A message is simply a data structure, comprising typed fields.
- `Topics` Each node publish/subscribe message topics via send/receive messages. A node sends out a message by publishing it to a given topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.In general, publishers and subscribers are not aware of each others' existence.

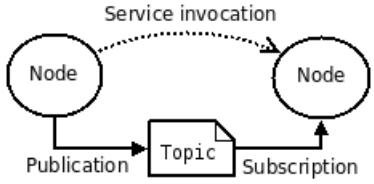

Figure A.1: source: http://wiki.ros.org/ROS/Concepts

A.3 Before we start..

Here are some useful Linux/ROS commands

- The command "ls" stands for (List Directory Contents), List the contents of the folder, be it file or folder, from which it runs.

```
$ ls
```

- The "mkdir" (Make directory) command create a new directory with name path. However is the directory already exists, it will return an error message "cannot create folder, folder already exists".

  ```
  $ mkdir <new_directory_name>
  ```

- The command "pwd" (print working directory), prints the current working directory with full path name from terminal

  ```
  $ pwd
  ```

- The frequently used "cd" command stands for change directory.

  ```
  $ cd /home/user/Desktop
  ```

  return to previous directory

  ```
  $ cd ..
  ```

  Change to home directory

  ```
  $ cd ~
  ```

- The hot key "ctrl+c" in command line **terminates** current running executable. If "ctrl+c" does not work, closing your terminal as that will also end the running Python program. **DO NOT USE "ctrl+z" as it can leave some unknown applications running in the background**.

- If you want to know the location of any specific ROS package/executable from in your system, you can use "rospack" find "package name" command. For example, if you would like to find 'lab2pkg_py' package, you can type in your console

  ```
  $ rospack find lab2pkg_py
  ```

- To move directly to the directory of a ROS package, use roscd. For example, go to lab2pkg_py package directory

  ```
  $ roscd lab2pkg_py
  ```

- Display Message data structure definitions with rosmsg

  ```
  # Display the fields in the msg
  $ rosmsg show <message_type>
  ```

- rostopic, A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages.

```
# Print messages to screen
$ rostopic echo /topic_name
# List all the topics available
$ rostopic list
# Publish data to topic
$ rostopic pub <topic-name> <topic-type> [data...]
```

A.4 Create your own workspace

Since other groups will be working on your same computer, you should backup your code to a USB drive or cloud drive everytime you come to lab. This way if your code is tampered with (probably by accident) you will have a backup.

- Log on to the computer as 'rvc' with the password '123456'. If you log on as 'guest', you will not be able to use ROS.
- First create a folder in the home directory, mkdir catkin_(yourNETID). It is not required to have "catkin" in the folder name but it is recommended.

  ```
  $ mkdir -p catkin_(yourID)/src
  $ cd catkin_(yourID)/src
  $ catkin_init_workspace
  ```

- Even though the workspace is empty (there are no packages in the 'src' folder, just a single CMakeLists.txt link) you can still "build" the workspace. Just for practice, build the workspace.

  ```
  $ cd ~/catkin_(yourID)/
  $ catkin_make
  ```

- **VERY IMPORTANT**: Remember to **ALWAYS** source when you open a new command prompt, so you can utilize the full convenience of Tab completion in ROS. Under workspace root directory:

  ```
  $ cd catkin_(yourID)
  $ source devel/setup.bash
  ```

A.5 Running a Node

- Once you have your catkin folder initialized, add the UR3 driver and lab starter files. The compressed file lab2andDanDriver.tar.gz, found at the class website contains the driver code you will need for all the ECE 470 labs along with the starter code for LAB 2. Future lab compressed files will only contain the new starter code for that lab. Copy lab2andDriverPy.tar.gz to your catkin directories "src" directory. Change directory to your "src" folder and uncompress by typing "tar -xvf lab2andDriver.tar.gz". You

can also do this via the GUI by double clicking on the compressed file and dragging the folders into the new location.

"cd .." back to your catkin_(yourID) folder and build the code with "catkin_make"

- After compilation is complete, we can start running our own nodes. For example our lab2node node. However, before running any nodes, we must have roscore running. This is taken care of by running a launch file.

  ```
  $ roslaunch ur_robot_driver ur3e_bringup.launch
  ```

  This command runs both roscore and the UR3 driver that acts as a subscriber waiting for a command message that controls the UR3's motors.

- Open a new command prompt with "ctrl+shift+N", cd to your root workspace directory, and source it "source devel/setup.bash".

- We may also need to make lab2_exec.py executable.

  ```
  $ chmod +x lab2_exec.py
  ```

- Run your node with the command rosrun in the new command prompt. Example of running lab2node node in lab2pkg package:

  ```
  $ rosrun ur3e_driver_ece470 ur3e_driver_ece470
  # Another terminal
  $ rosrun lab2pkg_py lab2_exec.py
  ```

A.6 More Publisher and Subscriber Tutorial

Please refer to the webpage: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c%2B%2B)