Institute for Aerospace Studies
UNIVERSITY OF TORONTO

LABORATORY I

Meet Your TurtleBot 3 Waffle Pi

(Sensor & Actuator Programming)

ROB521: Mobile Robotics and Perception
Winter 2023

1 Introduction

Welcome to ROB521—Mobile Robotics and Perception! This course will encompass a total of four labs and a design project, all of which are to be completed within a week of the scheduled practicum periods. Each of the four labs will grow in complexity and are intended to demonstrate important robotic concepts presented during the lectures. It is imperative that you properly understand the concepts and methods studied in the lab as they will provide the basis of all future labs.

We will specify tasks you need to do in the Assignment section of the lab handout. For all labs, assignments will have two components, simulation and experiments. Simulation tasks can be finished entirely in simulator (without accessing a real robot). These tasks are set up so that you could finish most, if not all, of the implementation before your in-person lab session. The experiment tasks are to be finished during lab sessions, verifying your simulation results on a real robot. It is quite difficult to complete all of the work during the lab times, even assuming good programming experience in the team. Therefore, we strongly recommend that you complete the simulation tasks before your lab time slot.

2 Objective

The objective of this laboratory exercise would be to familiarize you with the equipment and software that will be used for all labs in ROB521. In particular, you are:

- *To set up ROS environment for running simulation*
- *To learn about the robot's hardware and its suite of sensors*
- *To learn how to write ROS programs to command the robot in both simulation and on a real robot, i.e.,*
  - *How to write a simple Python ROS node*
  - *How to acquire data from the sensors*
  - *How to drive the actuators*

Possessing these capabilities will permit you to tackle the future labs.

2.1 Lab Deliverables

There are no deliverables for this lab. However, for future labs, look for a summary of the deliverables and their mark distribution at the end of the document. If you have to submit a lab report, include a short description of what your code is doing, and what the robot does as a result of running the code. Provide context that will allow the TA's to understand what your robot is doing in order for them to properly gauge your success. Importantly, state whether you were able to complete the deliverable.

If not, explain why you weren't able to complete the task. Avoid writing more than a paragraph for each deliverable.

Finally, include a copy of your code for the TA's to look over. The code itself will not be marked, but it must be presented to demonstrate that each group has independently written their solution. If the code does not look complete, TA's will have access to all of the computers, and will be able to run the code themselves to confirm that the deliverables have been completed.

3 Equipment & Software

The TurtleBot 3 Waffle Pi is a ROS-based, fully programmable, mobile robot. It is the most popular open-source robot with strong sensor lineups and modular actuators. The TurtleBot platforms have been developed by and are available from Robotis Inc; ROS is managed and maintained by Open Robotics. There are three official TurtleBot 3 models: the Burger, the Waffle, and the Waffle Pi. (Obviously, the designers worked overtime going without lunch or dinner.) For this course, we will be using TurtleBot 3 Waffle Pi and we will also be using a simulated TurtleBot3 within Gazebo (a robot simulation package).

3.1 TurtleBot 3 Waffle Pi

The TurtleBot 3 Waffle Pi model uses the Raspberry Pi as the single board computer. It is equipped with an openCR board, a Raspberry Pi Camera and a 2 dimensional (range and bearing) 360^o Lidar unit in addition to an inertial motion unit (IMU), a compass and a gyroscope (or gyro, for short).

The Waffle Pi software consists of firmware of OpenCR board and 4 ROS packages. It uses OpenCR as a subcontroller to estimate position by calculating the driving motor encoder value. Acceleration and angular velocity are obtained from the IMU and gyro that are mounted on the OpenCR board, from which position and orientation (i.e., pose) can be estimated. The velocity of the driving motors can be controlled by publishing the command in the upper-level software.

The 4 ROS packages are:

- `turtlebot3`
- `turtlebot3_msgs`
- `turtlebot3_simulations`
- `turtlebot3_applications`

For the in-person lab experiments, we will mainly rely on the `turtleBot3` package, which contains remote control package and bring-up package.

3.2 Hardware

3.2.1 Raspberry Pi 3B

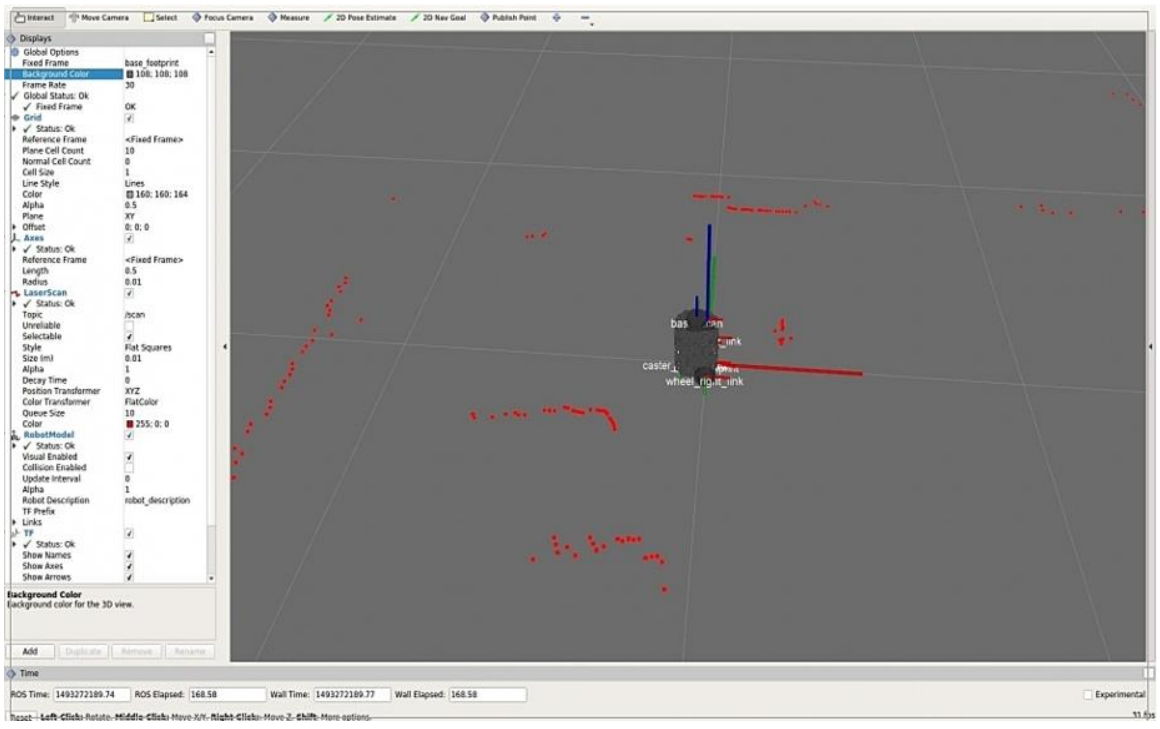
A Raspberry Pi 3B (Figure 1) is used as the TurtleBot PC. It is mainly responsible for collecting data from sensors such as the lidar, IMU, camera, and gyro. All the sensor output would be published to the corresponding "topics" (see Introduction to ROS). A more detailed description is presented in the next section. The Raspberry Pi is also responsible for receiving the commands from a remote PC and commanding the corresponding actuators or sensors.

Figure 1: Raspberry Pi 3B+

3.2.2 Lidar Sensor

The Waffle Pi is equipped with a 2D 360° lidar (light detection and ranging) sensor, but perhaps better referred to as a 2D laser scanner (Figure 2). It is capable of sensing the obstacles (including surfaces) around the robot in the plane of the sensor. The scan rate is 300±10 rpm and the angular resolution is 1°. The output of this sensor is an array of length 360. (A visualization of lidar data is shown in Figure 3.)

Figure 2: Lidar sensor



```
Global Options
  Fixed Frame            base_footprint
  Background Color       108; 108; 108
  Frame Rate             30
✓ Global Status: Ok
  ✓ Fixed Frame          OK
Grid                     ✓
  ✓ Status: Ok
  Reference Frame        <Fixed Frame>
  Plane Cell Count       10
  Normal Cell Count      0
  Cell Size              1
  Line Style             Lines
  Color                  160; 160; 164
  Alpha                  0.5
  Plane                  XY
  Offset                 0; 0; 0
Axes                     ✓
  ✓ Status: Ok
  Reference Frame        <Fixed Frame>
  Length                 0.5
  Radius                 0.01
LaserScan                ✓
  ✓ Status: Ok
  Topic                  /scan
  Unreliable             ☐
  Selectable             ✓
  Style                  Flat Squares
  Size (m)               0.01
  Alpha                  1
  Decay Time             0
  Position Transformer   XYZ
  Color Transformer      FlatColor
  Queue Size             10
  Color                  255; 0; 0
RobotModel               ✓
  ✓ Status: Ok
  Visual Enabled         ✓
  Collision Enabled      ☐
  Update Interval        0
  Alpha                  1
  Robot Description      robot_description
  TF Prefix
  Links
TF                       ✓
  ✓ Status: Ok
  Show Names             ✓
  Show Axes              ✓
  Show Arrows            ✓

Background Color
Background color for the 3D view.

ROS Time: 1493272189.74   ROS Elapsed: 168.58   Wall Time: 1493272189.77   Wall Elapsed: 168.58
```

Figure 3: the GUI of Lidar sensor

3.2.3 Dynamixel XM430

The Waffle Pi uses 2 Dynamixel actuators (Figure 4) to drive the wheels. The motors can be operated by one of 6 operating modes, including

- Velocity control
- Torque control
- Position control

Figure 4: Dynamixel motors

3.2.4 OpenCR 1.0

The OpenCR control board (Figure 5) acts as a sub-controller for Waffle Pi, which can command the robot's Dynamixel motors and various sensors. It has an IMU and a gyro that can be used in various applications.

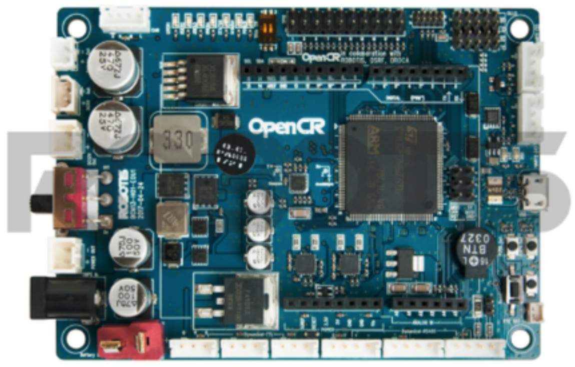


Figure 5: OpenCR 1.0 embedded board

3.3 Simulated TurtleBot 3 Waffle Pi in Gazebo

When developing autonomy software for robots, we often start with testing in a simulator as opposed to a real robot. In this course, we will use Gazebo to simulate our TurtleBot 3 Waffle Pi. Gazebo simulates robot dynamics (i.e. how robot move given a certain control input) based on a prior robot model. It also simulates the onboard sensors mentioned in Section 3.2 based on sensor models. Essentially, the Gazebo provides a relatively good approximation of what a real robot would move and perceive. In terms of software structure, Gazebo completely replaces the real robot. We should be able to run the same code with both the simulator and the real robot.

4 Development Environment

In this section, we will introduce the development environments for running simulation and working with a real robot.

For in-person lab experiments, you will use one of the desktop PCs in MY570 as a Remote PC, which interfaces with the Raspberry Pi, the TurtleBot PC (Raspberry Pi). Remote PC usually performs resource consuming tasks such as data processing and high-level planning for localization and navigation. In contrast, the onboard

TurtleBot PC controls the robot components and collects sensor readings that require communication. Both PCs have similar development environments, since they both run on the base operating system Linux(Ubuntu 20.04) and ROS (Noetic Ninjemys); however, they use different ROS packages for their jobs.

For simulations, you only need one PC for all the aforementioned tasks. In particular, Gazebo will be responsible for the TurtleBot PC's jobs, controlling robot components and collecting sensor readings. To enable smooth transition from simulation to experiments, we will use ROS Noetic Ninjemys in Ubuntu 20.04.

In the following sections, we will help you set up development environments for both simulation and experiments. Note that you only need to set up the simulation environments before lab.

4.1 Setting up Simulation Environment

As mentioned before, we will run simulations on Ubuntu 20.04 with ROS Noetic Ninjemys. In this course, we provide two options for accessing Ubunutu 20.04, UofT computers via virtual access or installing Ubuntu on your own laptop. We will share more information soon on Quercus. Please proceed once you have access to a Ubuntu 20.04 PC.

If you choose to use UofT computers, you only need to install the TurtleBot3 Gazebo Simulation Package under your catkin workspace by cloning it from github (Step 3 in Section 4.1.2).

4.1.1 Installing ROS Noetic

Setup your computer to accept software from packages.ros.org.

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

Set up your keys

```
$ sudo apt install curl # if you haven't already installed curl\\
$ curl -s \
https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc |
sudo apt-key add -
```

Make sure your Debian package index is up-to-date:

```
$ sudo apt update
```

Install Noetic Desktop Full (we need the Gazebo packages that come with this version)

```
  $ sudo apt install ros-noetic-desktop-full
```

You must source this script in every bash terminal you use ROS in.

```
  $ source /opt/ros/noetic/setup.bash
```

You could also add this line to the `~/.bashrc` file.

```
  $ echo 'source /opt/ros/noetic/setup.bash' >> ~/.bashrc
```

4.1.2 Install TurtleBot3 ROS Packages

Install Dependent ROS Packages for TurtleBot

```
  $ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
  ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
  ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
  ros-noetic-rosserial-python ros-noetic-rosserial-client \
  ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
  ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
  ros-noetic-compressed-image-transport ros-noetic-rqt* \
  ros-noetic-rviz ros-noetic-gmapping \
  ros-noetic-navigation ros-noetic-interactive-markers
```

Install TurtleBot3 Packages

```
  $ sudo apt install ros-noetic-dynamixel-sdk
  $ sudo apt install ros-noetic-turtlebot3-msgs
  $ sudo apt install ros-noetic-turtlebot3
```

Install TurtleBot3 Gazebo Simulation Package under your catkin workspace

```
$ cd ~/catkin_ws/src/
$ git clone -b noetic-devel \
https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
```

Don't forget to source the setup bash from your catkin workspace

```
$ source devel/setup.bash
```

You also need to setup the environment variable to identify the TurtleBot 3 model we are using

```
$ echo 'export TURTLEBOT3_MODEL=waffle_pi' >> ~/.bashrc
```

4.2 Setting up Experiment Environment

ROS environment and packages have been set up for you on lab stations and Raspberry Pi. To start your in-person lab experiments, you only need configure the network environment on both Remote PC and TurtleBot PC so that they can communicate with each other.

As a first step, we need to log into the TurtleBot PC. We will be doing it remotely using ssh. Open a terminal window on Remote PC. We recommend that you use *terminator*, which is installed on the computers. Within *terminator*, right-click and choose "split horizontally" or "split vertically" to open additional terminals within the window. A static IP address has been set up for each TurtleBot PC and will be given to you at the beginning of the lab session. Make sure your TurtleBot PC is powered on, and in one of the open terminals, run the following command to remote-access its terminal via ssh:

```
$ ssh ubuntu@\{The static IP address of Pi\}
$ ssh ubuntu@192.168.0.101 # for example
```

Now enter the password for your TurtleBot PC, which is 'turtlebot'. You will now see your usual command line replaced with `ubuntu@raspberrypi: ~$`. It means that you are now logged in and working on a terminal from your Raspberry Pi. As long as this terminal window stays opened, the ssh connection will continue to exist.

You also need to know the IP address of the Remote PC. Open a terminal window on the Remote PC and use 'ifconfig' command to get the IP address of the remote PC (e.g. 192.168.7.100):

```
$ ifconfig
```

Now, we are ready to set the network environment variables on both the Remote PC and TurtleBot PC. On both PCs, open the `/.bashrc` script on the terminal window.

```
$ nano ~/.bashrc
```

Scroll to the bottom of the file to set up the ROS network variables. Note that on the Remote PC, you will find the `ROS_MASTER_URI` have been configured for you with a unique IP address and `ROS_PORT`.

```
# Remote PC ~/.bashrc

export ROS_HOSTNAME={remote_PC IP}
export ROS_MASTER_URI={remote_PC IP}:{remote_PC ROS_PORT}
```

Please **DO NOT** change the `ROS_HOSTNAME` and `ROS_PORT`. They are set up so that multiple remote access groups can run simulations at the same time.
Configure the Turtlebot PC `~/.bashrc` file to share the same `ROS_MASTER_URI` as the Remote PC and declare its `ROS_HOSTNAME`

```
# TurtleBot PC ~/.bashrc

export ROS_HOSTNAME={TurtleBot IP}
export ROS_MASTER_URI={remote_PC IP}:{remote_PC ROS_PORT}
```

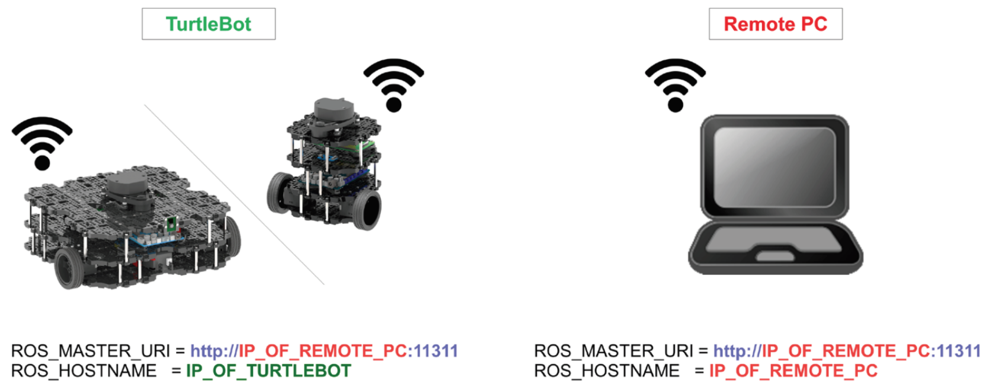

TurtleBot
Remote PC
ROS_MASTER_URI = http://IP_OF_REMOTE_PC:11311
ROS_HOSTNAME = IP_OF_TURTLEBOT
ROS_MASTER_URI = http://IP_OF_REMOTE_PC:11311
ROS_HOSTNAME = IP_OF_REMOTE_PC

Figure 6: network configuration for Remote PC and TurtleBot PC

The above commands are summarized in Figure 6. After modifying the ROS network variables , source the /.bashrc file for the changes to take effect:

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

Note that the computer and robot may already be correctly configured prior to your lab session. Nevertheless, it's good to check and make sure that the computer's IP hasn't changed.

5 Getting Started

In this section, you will first validate the development setup in the previous section by teleoperating the TurtleBot 3. Then, you will check the TurtleBot 3 ROS topics that of our interests. You will be finishing this section with a simple ROS node example to get you started on the assignment of this lab.

5.1 Testing Simulation Setup: Remote Control of Robot

Open a new terminal window and use the following command to run the roscore program at port `ROS_PORT` which is defined in the `~/.bashrc` file

```
$ roscore --port $ROS_PORT
```

Please make sure to include the port number; otherwise, other groups use the same station with you at the same time can not run simulations.

Open a new terminal window and start a Gazebo simulation with

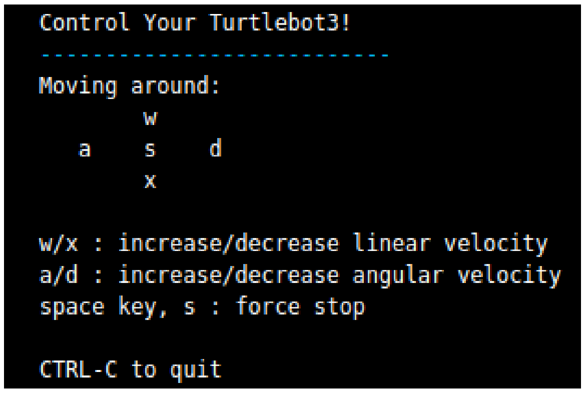
```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

which runs a launch file the turtlebot3_gazebo ROS package.
In another terminal window, run the launch file turtlebot3_teleop_key.launch by issuing the following command:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Note that you could tab to auto-complete ROS packages names and file names. Make sure you correctly source the setup bash files before running this test.

After you launched the teleoperation file, You should see the following message appears in the terminal (Figure 7).



```
Control Your Turtlebot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, s : force stop

CTRL-C to quit
```

Figure 7: `turtlebot3_teleop` package message.

This node will retrieve the key inputs of ‘w’, ‘a’, ‘d’, ‘x’ and control the linear and angular velocity of the robot in m/sec and rad/sec respectively.

Once you have confirmed that you can control your simulated TurtleBot using keyboard, press Ctrl-C to terminate the teleop node. **Leave roscore running and the `turtlebot3_empty_world.launch` launch file running by keeping their terminal windows open**. They are the essential programs for us to control and communicate with the simulated robot in Gazebo.

5.2 Testing Experiment Setup: Remote Control of Robot

Recall that for experiments, we have two PCs, the Remote PC (lab stations) running high-level planning and data processing and the TurtleBot PC on TurtleBot 3 Waffle Pi.
On the remote PC, open a new terminal window and use the following command to run the roscore program:

```
$ roscore --port $ROS_PORT
```

Again, please make sure to include the port number.

On TurtleBot PC, run the launch file `turtlebot3_robot.launch` from the `turtlebot3_bringup` package in a new terminal window.

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch --screen
```

This will start the communication between TurtleBot PC and the sensors and actuators. In a new terminal window on the Remote PC, run the launch file `turtlebot3_teleop_key.launch` by issuing the following command:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

The same print out will show up on the Remote PC (Figure. 7). Once you have verified that you can remote control your TurtleBot using keyboard, press Ctrl-C to terminate the teleop node. **Leave roscore running and the `turtlebot3_robot.launch` launch file running by keeping their terminal windows open**.

Note the differences between running simulations and experiments. The core control node (the teleop node in this case) is the same. We use different packages for the robot.

5.3 TurtleBot 3 Topics

This section is intended to be done during the lab session on a real robot. However, you could also check the topics in simulation. You will find a few topics that are published and subscribed by nodes on the TurtleBot PC are now done by the Gazebo simulator.

After we launched the `turtlebot3_robot.launch` file from the `turtlebot3_bringup` package, messages will be published from each node, such as sensors and actuators, to their corresponding *topics*.

Make sure that the roscore and `turtlebot3_bringup` package are running, verify the various topics that are being published or subscribed using the command `$ rostopic list`. After typing this command in terminal, you should see the following topics being listed:

```
/cmd_vel
/cmd_vel_rc100
/diagnostics
/imu
/joint_states
/odom
/rosout
/rosout_agg
/rpms
/scan
/sensor_state
/tf
```

5.3.1 Subscribed Topics

Table 1 is a list of subscribed topics of the robot. The TurtleBot PC receives and processes the messages from the topics that are published by the user.

Topic Name	Message Type	Description
cmd_vel	geometry_msgs/Twist	Control the translational and rotational speed of the robot. unit in m/s, rad/s (actual robot control)
motor_power	std_msgs/Bool	Dynamixel Torque On/Off
reset	std_msgs/Empty	Reset Odometry and IMU Data

Table 1: Subscribed topics of TurtleBot

5.3.2 Published Topics

Table 2 is a list of the topics that are published by the TurtleBot PC. You do not need to know all the published topics, but it would be important to know some of them such as 'odom', 'imu', and 'scan'. User can subscribe to those topics to retrieve the information published by the robot.

Topic Name	Message Type	Description
sensor_state	turtlebot_node /TurtlebotSensorState	Topic that contains the value
joint_states	sensor_msgs/JointState	Checks the position (m), velocity (m/s) and effort (N·m) when the wheels are considered as joints.
battery_state	sensor_msgs/BatteryState	Contains battery voltage and status
scan	sensor_msgs/LaserScan	Topic that confirms the scan values of the LiDAR mounted on the TurtleBot3
imu	sensor_msgs/Imu	Topic that includes the attitude of the robot based on the acceleration and gyro sensor.
odom	nav_msgs/Odometry	Contains the TurtleBot3's odometry information based on the encoder and IMU.

Table 2: Published topics of TurtleBot

To get more details on nodes and topics, run ‘ `$ rqt_graph` in terminal to check the publishing and subscribing activities of each node on TutrleBot PC. A picture as in Figure 8 should pop-up in your window:

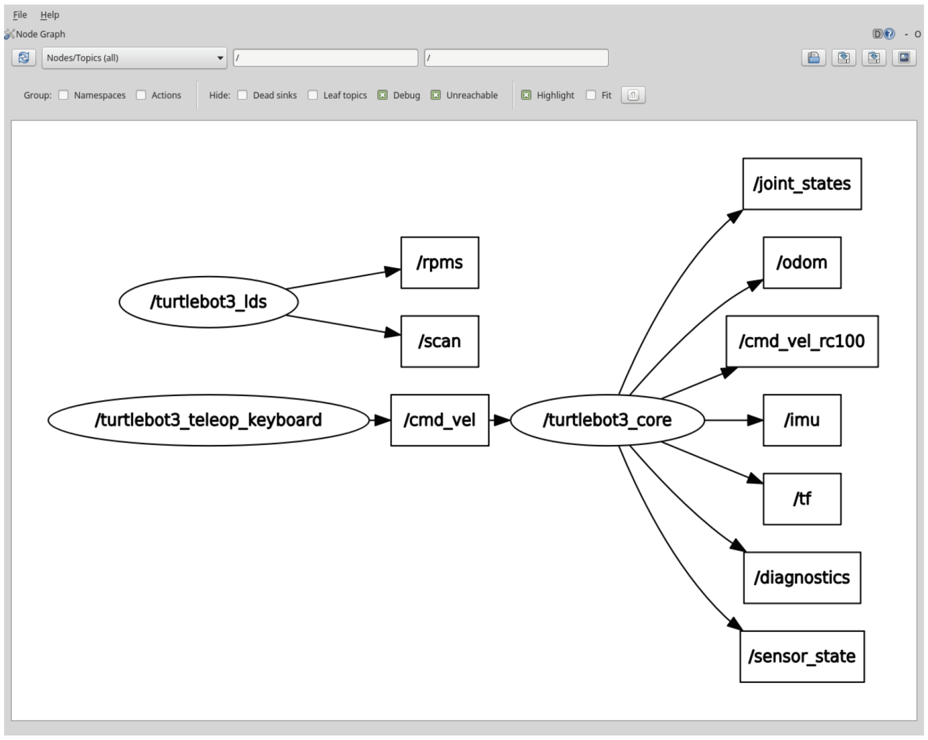


Figure 8: TurtleBot nodes and topics

5.4 A Simple Publisher Node

Example

Let's first examine a simple example shown below. This example initialize a ROS node called 'talker' and then publish messages to the topic 'chatter'.

```
#!/usr/bin/env python3
    import rospy
    from std_msgs.msg import String

    def talker():
        rospy.init_node('talker', anonymous=True)
        pub = rospy.Publisher('chatter', String, queue_size=10)
        rate = rospy.Rate(10) #10Hz
        while not rospy.is_shutdown():
            hello_str = "hello world"
            msg = String()
            msg.data = hello_str

            rospy.loginfo(msg.data)
            pub.publish(msg)
            rate.sleep()

    if __name__ == '__main__':
        try:
            talker()
        except rospy.ROSInterruptException:
            pass
```

Adding Path to Python Interpreter: To make sure that your script is executed as a Python 3 script, you need the line

```
#!/usr/bin/env python3
```

to be declared at the top for every python ROS node.

Importing dependent message types and libraries You need to import rospy to write a ROS node. Hence

```
import rospy
from std_msgs.msg import String
```

The `std_msgs.msg` import is for us to use the `std_msgs/String` message type for publishing.

Initializing the ROS node: In the main talker function, we find
which declares a ROS node under the name 'talker'. In ROS, nodes are uniquely named. The `anonymous=True` flag allows rospy to choose a unique name for the "talker" node such that multiple "talker" can run at the same time.

```
rospy.init_node('talker', anonymous=True)
```

Declaring a publisher: Next, we declare a topic publisher

```
pub = rospy.Publisher('chatter', String, queue_size=10)
```

which means our 'talker' node will publish to the 'chatter' topic using the message type String, which is we have imported at the top of the script from `std_msgs.msg`.

Laying out the main loop: A typical rospy main loop looks like follows

```
rate = rospy.Rate(10) #10Hz
while not rospy.is_shutdown():
    # Do Some Work
    rate.sleep()
```

The first line initializes a Rate object set at 10Hz. It is called inside loop so that the while loop is running at 10Hz. It is also a good practice to check for `rospy.is_shutdown()` in the main loop so that the node could be properly shut down when you quit the program.

Initializing and publishing messages: Inside the main loop, we have at the beginning

```
hello_str = "hello world"
msg = String()
msg.data = hello_str
```

where we initialize an empty message (second line) and populate it with the information we want to publish (third line). In this case, the message type is String. In the `std_msgs/String.msg` file, you will find its definition which has only one field, "data" of string type.
After the message is created, we can finally publish it with

```
pub.publish(msg)
```

Note that there's another line in the example `rospy.loginfo(msg.data)`. It will do three things: print to the terminal window, write to the Node's log file and write to /rosout.

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

 def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin() # simply keeps python from exiting until this node is stopped

if __name__ == '__main__':
    listener()
```

5.5 A Simple Subscriber Node

Example The following example initializes a ROS node 'listener' and then subscribe from the topic 'chatter'.

The code for subscriber node is similar to the publisher node. However, subscriber uses a new callback mechanism for subscribing to topics.

Declaring a subscriber: In the main function, after initializing the ros node, we have

```
rospy.Subscriber("chatter", String, callback)
```

which means the node has subscribed to the 'chatter' topic of the type `std_msgs.msgs.String`. When a new message is received, the given callback function, `callback()` in our case, is triggered with the message as the default first argument.

Defining the callback function:

The callback function is where you process the incoming messages

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

6 Assignment

6.1 Simulation

6.1.1 Task 1: publish to ‘cmd_vel’

Your task is to write a simple publisher node to control the wheels. The goal is to command the wheel to go forward 1 m, rotate 360°, and then stop.

First, navigate to the directory `~/catkin\_ws/src/rob521\_labs/lab1/nodes` and open the file `l1_motor.py`. The code skeleton is provided to you, which is similar to the example above, but you need to complete the functions and then test the code on the robot.

The topic you need to publish to is `cmd_vel`, and the message type is `geometry_msgs.msg.twist`.
The message definition is as follows:

```
Vector3 linear
    float64 x
    float64 y
    float64 z

Vector3 angular
    float64 x
    float64 y
    float64 z
```

First thing you need to do is to import rospy and relevant message types.

```
import rospy
from geometry_msgs.msg import Twist
```

Then, initialize the publisher:

```
cmd_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
```

Initialize the message and define the linear and angular velocity as the following.

```
twist=Twist()
twist.linear.x=0.1
twist.angular.z=0.1
cmd_pub.publish(twist)
```

Now you can publish the message to the topic ‘cmd_vel’. The motors will subscribe to the topic ‘cmd_vel’ and move according to the commands you send.

Write a simple program to command the robot to go forward for 1 meter, then rotate clockwise for 360 degrees, and then stop. You can use rospy.loginfo() to print out any useful debugging messages in realtime.

Let's run the file and examine it. Make sure that a `roscore` and the Gazebo `turtlebot3_empty_world` package is up and running on your PC by typing the following command in terminal:

```
$ roscore --port $ROS_PORT
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

You can use rosrun `$ rosrun rob521_lab1 l1_motor.py` to run the python script you just wrote. Note that `rob521_lab1` is the package name, and `l1_motor.py` is the file you just modified. If you wish to terminate the program while the motor is still running, press ctrl+c to exit the node, and then run

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist -- '[0, 0, 0]' '[0, 0,
0]'
```

which will stop the motor.

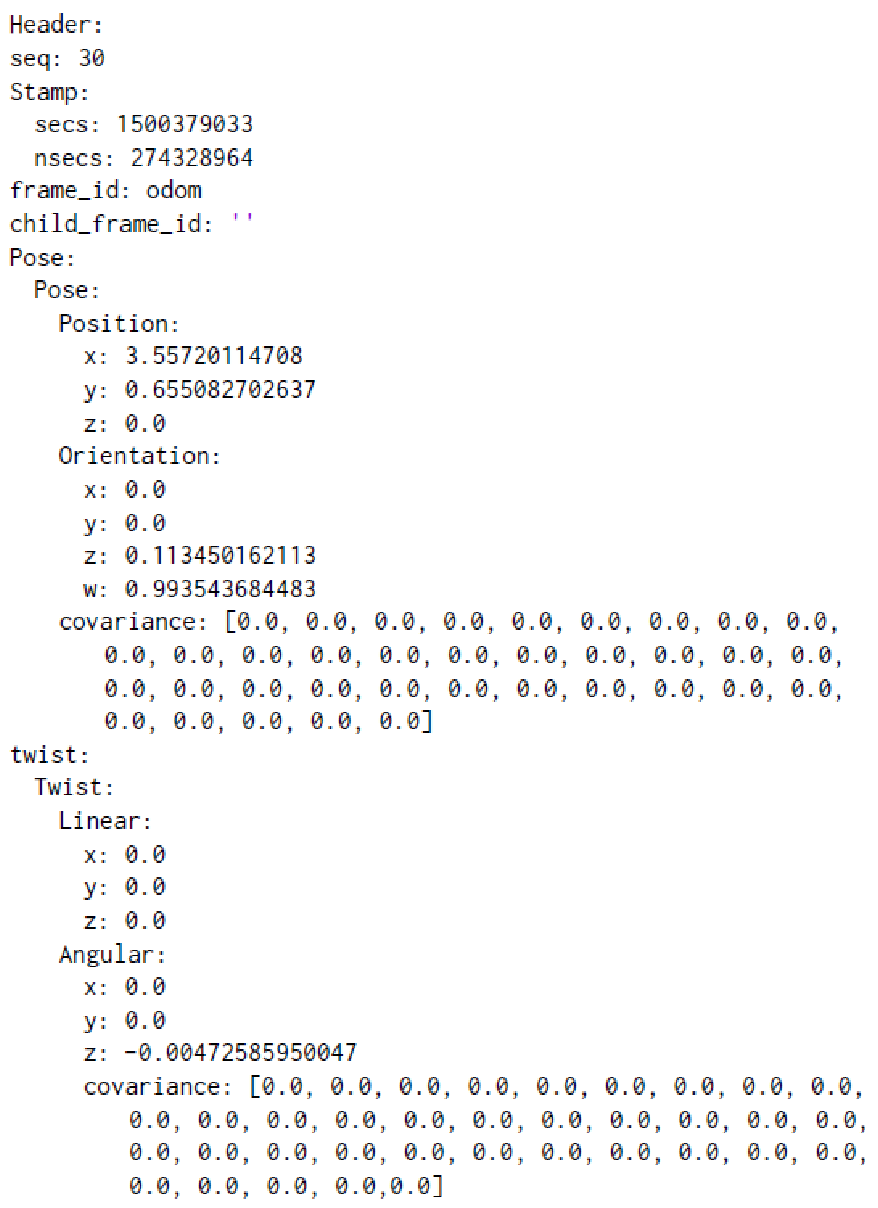
6.1.2 Task 2: subscribe from 'odom'

Now, let's write a simple node that subscribes to the odometry topic called 'odom'. The goal is to retrieve the current pose of the robot. The pose of the robot is defined as $[x, y, \theta]$.

The message type of 'odom' is `nav_msgs.msg.Odometry`. It is defined as below in the .msg file.

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
    Pose pose
        Point position
            float64 x
            float64 y
            float64 z
        Quaternion orientation
            float64 x
            float64 y
            float64 z
            float64 w
        float64[36] covariance
geometry_msgs/TwistWithCovariance twist
```

You can verify the structure by typing an echo command `$ rostopic echo /odom` in the terminal to print out the odometry information. The output should look like that in Figure 9.



```
Header:
seq: 30
Stamp:
  secs: 1500379033
  nsecs: 274328964
frame_id: odom
child_frame_id: ''
Pose:
  Pose:
    Position:
      x: 3.55720114708
      y: 0.655082702637
      z: 0.0
    Orientation:
      x: 0.0
      y: 0.0
      z: 0.113450162113
      w: 0.993543684483
    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]
twist:
  Twist:
    Linear:
      x: 0.0
      y: 0.0
      z: 0.0
    Angular:
      x: 0.0
      y: 0.0
      z: -0.00472585950047
      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]
```

Figure 9: Example odom output.

The command `from nav_msgs.msg import Odometry` imports the message type for odometry output. Then the following commands initialize the node and the subscriber:

```
rospy.init_node('odometry')
odom_subscriber=rospy.Subscriber('odom',Odometry,callback,queue_size=1)
```

We can define a callback function and inside the function, retrieve the position and orientation from the message.

```
def callback(odom_data):
    point=odom_data.pose.pose.position
    quart=odom_data.pose.pose.orientation
    theta=get_yaw_from_quarternion(quart)
    cur_pose = (point.x, point.y, theta)
    rospy.loginfo(cur_pose)
```

Note that the orientation of the robot is expressed in quaternion, so you need to transfer that to angle using the following formulas:

```
def get_yaw_from_quarternion(q):
    siny_cosp = 2* (q.w*q.z + q.x*q.y)
    cosy_cosp = 1 - 2*(q.y*q.y + q.z*q.z)
    yaw = math.atan(siny_cosp/cosy_cosp)
    return yaw
    rospy.loginfo(cur_pose)
```

Once you have finished the file, run `$ rosrun rob521_lab1 l1_odometry.py` on your PC to verify the output. You can terminate the node by typing Ctrl+C.

6.1.3 Task 3: Running subscriber node and publisher node simultaneously

Now you have written a simple subscriber node of odometry, and a simple publishing node of motors, let's try running them together.

Launch the subscriber node `l1_odometry.py` first, and then launch the publisher node `l1_motor.py`. Examine the output of the odometry, does the output reflects the current position of the robot? How accurate is it?

6.2 Lab Experiments

6.2.1 Task 4: Verify simulation results

Repeat Task 1 to 3 on a real TurtleBot 3. Instead of running gazebo, run the launch file from the `bring_up` package to start communication,

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch --screen
```

Remember to run your publisher and subcription node on the Remote PC.

Examine the output of the odometry, does the output reflects the current position of the robot? How accurate is it compared to simulation?

7 Deliverable Summary

You do not need to submit anything for this lab.

8 Additional Resources

1. ROBOTIS e-Manual: http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
2. "SSH: Remote control your Raspberry Pi," The MagPi Magazine https://www.raspberrypi.org/ma
   remote-control-raspberry-pi/
3. Official ROS website - https://www.ros.org/
4. ROS Wiki - http://wiki.ros.org/ROS/Introduction
5. Useful tutorials to run through from ROS Wiki - http://wiki.ros.org/ROS/Tutorials
6. ROS Robot Programming Textbook, written by the TurtleBot3 developers - http://www.pishrobot.com/wp-content/uploads/2018/02/ROS-robot-programming-book-by-turtlebo3-developers-EN.pdf