Institute for Aerospace Studies
UNIVERSITY OF TORONTO

LABORATORY I

Meet Your TurtleBot 3 Waffle Pi

(Sensor & Actuator Programming)

ROB301 Introduction to Robotics
Fall 2020

1 Introduction

This is the first laboratory exercise of ROB301—Introduction to Robotics. The course will encompass a total of four labs and a design project. Each of the four labs will grow in complexity and is intended to demonstrate important robotic concepts presented during the lectures. Our robot of choice: *Turtlebot 3 Waffle Pi* running the operating system *ROS*.

It is imperative that you properly understand the concepts and methods studied in this lab as they will provide the basis of all future labs.

2 Objective

The objective of this laboratory exercise would normally be to familiarize you with the hardware and software that will be used for all labs and the project in ROB301. However, this year, you'll be learning about the robot's hardware through simulation only. In this and future labs, reference to "hardware" must be interpreted in the virtual sense.

This first lab is designed for you

- *To learn about the robot's hardware and its suite of sensors*
- *To learn how to write ROS programs to command the 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 as well as the design project at the end of the course.

2.1 Lab Deliverables

Since students cannot demonstrate functionality to the TA during in-person labs, as is usually done, you will be required to demonstrate functionality by recording the specified rosbag files (rosbag is a tool in ROS that can store all information that passes through a specified topic, such as the velocity inputs to the robot in the topic `/cmd_vel`).

In this and future labs, look for deliverables in **bold text** throughout the documentation and follow the instructions for each. For this lab, there is a single deliverable in Section 5.1 that will be graded.

In addition to completing the deliverables, you will be required to submit a short report for each lab. In your lab report, for each deliverable, include a short

description of what your code is doing, and what the robot does as a result of running the code, i.e., what happens within your recorded rosbag file. Provide context that will allow the TAs to understand what your robot is doing during the rosbag recording in order for them to gauge your success properly. Most important, state whether you were able to complete the deliverable. If not, explain why you weren't. Do not write more than a paragraph for each deliverable.

Finally, include a copy of your code for the TAs to look over. The code itself will not be marked, but it must be presented to demonstrate that each team has independently written their solution. Be aware that, if the code does not look complete, TAs 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 one of 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 breakfast or lunch.) For this course, we will be using the TurtleBot 3 Waffle Pi and this year we will be using a simulated TurtleBot 3 within Gazebo.

3.1 TurtleBot 3 Waffle Pi

The simulated TurtleBot 3 Waffle Pi model is equipped with several sensors: a camera, a two-dimensional (range and bearing) 360° lidar unit, and an inertial motion unit (IMU).

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.

3.2 Sensors and Actuators

When the robot is initialized, all sensor outputs are published to the corresponding "topics" (see Introduction to ROS). A more detailed description is presented in the following sections.

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 1). It is capable of sensing the obstacles (including surfaces) around the robot in the plane of the sensor; it has a graphical interface as shown in Figure 2. The scan rate is 300±10 rpm and the angular resolution is 1°. The output of this sensor is an array of length 360. See Figure 2 for a visualization of the lidar data. The lidar output is published in the `/scan` topic.

Figure 1: Lidar Sensor

Camera

The camera is a downward-facing camera that will be used in future labs for line detection (for a line-following task). There are many camera topics, with an example being `/camera/rgb/image_raw` for the raw image feed.

IMU

The `/imu` topic publishes the IMU data. This consists of the 3-axis linear acceleration, and 3-axis angular velocity readings, expressed within the reference frame of the IMU.

Dynamixel XM430 Actuators

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

- Velocity control,
- Torque control,
- Position control.

We will be using velocity control, which can be published to the `/cmd_vel` topic.

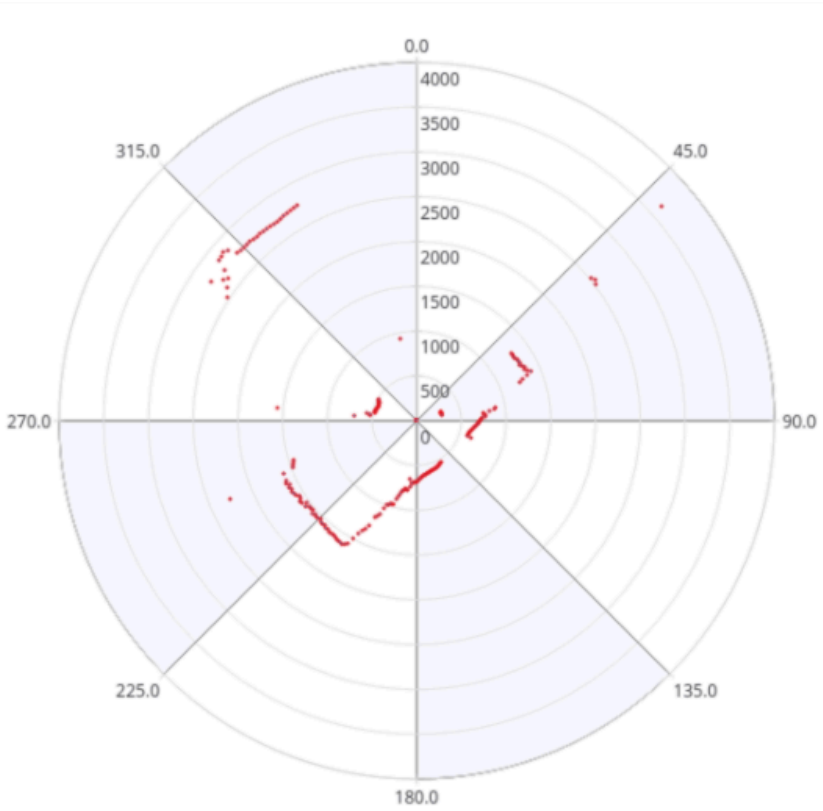

Figure 2: GUI of Lidar Sensor

4 Getting Started

4.1 Remote Control of Robot

On your PC, open a new terminal window (or your favourite SSH client such as PuTTY) connect to the remote Linux PC that was allocated to you:

```
$ ssh X@100.92.1.Y
```

where X is your account name, either `rob301` for PRA0101 or `rob301_2` for PRA0102, and Y is between 21 and 34. After entering the password, you will be connected to the remote Linux PC. Then, use the following command to launch the Lab 1 simulation environment:

```
$ roslaunch rob301_simulation lab_01.launch
```

This will start an empty world in the Gazebo simulation. Now you can leave that running on the terminal, and then SSH once again into the remote PC to start the

Figure 3: Dynamixel Motors

graphical interface Gzweb. Run the following commands in the new terminal window:

```
$ cd ~/gzweb && npm start -p $GZWEB_PORT
```

To visualize the simulation, open up your browser on your local PC, and enter the IP address and the port number in the address bar. Remember to replace the IP address and the port number in the example below—the port is 8080 for the rob301 account, and 8081 for rob301_2, e.g.,

```
http://100.92.1.29:8080
```

In a new SSH terminal, run the launch file `turtlebot3_teleop_key.launch` by issuing the following command:

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

You should see the following message appear in the terminal:

```
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
```

This node will retrieve the key inputs of 'w', 'a', 'd', 's', 'x' and control the translational and rotational velocity of the robot in units of metres per second and radians per second, respectively.

You should be able to see your robot moving in the Gzweb simulation according to your keyboard control. Once you have confirmed that you can remotely control your robot using keyboard, press Ctrl-C in the terminal to terminate the teleop node.

4.2 TurtleBot Topics

When the Lab 1 launch file has been executed, messages will be published from each node, such as sensors and actuators, to their corresponding *topics*. While the `lab01.launch` file is still 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:

```
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/camera/rgb/image_raw/compressedDepth
/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/camera/rgb/image_raw/compressedDepth/parameter_updates
/camera/rgb/image_raw/theora
/camera/rgb/image_raw/theora/parameter_descriptions
/camera/rgb/image_raw/theora/parameter_updates
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/imu
/joint_states
/odom
/rosout
/rosout_agg
/scan
/tf
```

Subscribed Topics

Table 1 displays a list of subscribed topics for the robot. The TurtleBot receives and processes the messages from the topics that are published by the user. Table 2

Table 1: Subscribed Topics of TurtleBot

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)

displays a list of topics subscribed by Gazebo. In the following context, the pose (position + orientation) and twist (velocity + angular velocity) of a rigid-body object is referred to as its *state*. In Gazebo, a *link* refers to a rigid body with given inertial, visual, and collision property, and a *model* is a conglomeration of bodies connected by *joints*.

Table 2: Subscribed Topics of Gazebo

Topic Name	Message Type	Description
gazebo/set_link_states	gazebo_msgs/LinkStates	Sets the state (pose/twist) of a link
gazebo/set_model_states	gazebo_msgs/ModelStates	Sets the state (pose/twist) of a model

4.2.1 Published Topics

Table 3 shows a list of the topics that are published by the TurtleBot. You do not need to know all the published topics but it is important to know some of them such as 'odom', 'imu' and 'scan'. Users can subscribe to these topics to retrieve the information published by the robot.

Table 4 shows a list of topics published by Gazebo, containing pose and twist information of objects in Gazebo simulation. The state of a *model* is the state of its root *link*.

4.3 A Simple Publisher Node

Let's examine the simple example in Figure 4 first. This example initializes a ROS node called 'talker' and then publish messages to the topic 'chatter'.

Table 3: Published Topics of TurtleBot

Topic Name	Message Type	Description
sensor_state	turtlebot_node/ TurtlebotSensorState	Topic that contains the value
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 robot's odometry information based on the encoder and IMU
camera/rgb/image_raw	sensor_msgs/Image	The raw images from the color camera.
camera/rgb/image_raw/ compressed	sensor_msgs/CompressedImage	The compressed raw image buffer from the color camera.

Table 4: Published Topics of Gazebo

Topic Name	Message Type	Description
clock	rosgraph_msgs/Clock	Publish simulation time
gazebo/link_states	gazebo_msgs/LinkStates	Publishes states of all the links in simulation
gazebo/model_states	gazebo_msgs/ModelStates	Publishes states of all the models in simulation

Adding path to Python interpreter. To make sure that your script is executed as a Python script, you need the line

```
#!/usr/bin/env python
```

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
```

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

    def talker():
        pub = rospy.Publisher('chatter', String, queue_size=10)
        rospy.init_node('talker', anonymous=True)
        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
```

Figure 4: A Simple Example

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

Initializing the message type. The command `std_msgs.msg.String` is a relatively simple message type; you could directly publish it as `pub.publish(hello_str)` . However, when you need to publish more complicated message types, you will have to initialize the message as follows:

```
msg = String()
msg.data = hello_str
```

Here `msg = String()` initializes a new empty message of type String and then the field 'data' is updated to `hello_str` in line 2.

Declaring publisher and initializing the node. Next we find the publisher initialization:

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

The first line declares that your node is publishing to the 'chatter' topic using the message type String, which is actually from the class imported at the top of the script `std_msgs.msg`. The second line initializes the 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's can run at the same time. Next, `rate=rospy.Rate(10)` initializes a Rate object. *Important Note:* Without calling `rate.sleep()` in the loop, messages will be published at a maximum frequency and may cause issues with the motor control.

Writing the main loop. The standard `rospy` main loop is

```
while not rospy.is_shutdown():
    hello_str = "hello world"
    msg = String()
    msg.data = hello_str

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

You should check the `rospy.is_shutdown()` flag to execute your program in the main loop. Now `rospy.loginfo(str)` does three things: Prints to the terminal window, writes to the node's log file and writes to `/rosout`. Finally, `pub.publish(msg)` publishes a message type to the topic 'chatter'.

4.4 A Simple Subscriber Node

The following example initializes a ROS node 'listener' and then subscribes to the topic 'chatter':

```
#!/usr/bin/env python
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)
    #spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

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

Let's break it down. The code for subscriber node is similar to the publisher node. However, subscriber uses a new 'callback'-based mechanism for subscribing to topics.

Initializing the node. Consider the lines

```
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
rospy.spin()
```

The first line initializes the node under the name 'listener' with anonymous set to True, such that multiple 'listener' nodes can run simultaneously.

The second line declares that the node is subscribing to the 'chatter' topic of the type `std_msgs.msgs.String`. When a new message is received, the 'callback' function is triggered with the message as the default first argument.

Finally, the line `rospy.spin()` keeps the node from exiting until the node has been terminated.

Defining the callback function. The 'callback' function is triggered every time a new message arrives…

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

5 Assignment

5.1 Task 1: Publish to 'cmd_vel'

Your task is now 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/ROB301_simulation/nodes/lab1` and open the file `lab1_motor.py`. The skeleton code 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
```

The first thing you need to do is to import `rospy` and relevant message types:

```
from geometry_msgs.msg import twist
```

Then initialize the publisher node,

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

Initialize the message type and define the linear and angular velocity:

```
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 m, then rotate clockwise for 360° and 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 the `lab01.launch` file and Gzweb is up and running on your remote PC by typing the following commands in terminals:

```
$ roslaunch rob301_simulation lab_01.launch
```

```
$ cd ~/gzweb && npm start -p $GZWEB_PORT
```

You can enter `$ rosrun rob301_simulation lab01_motor.py` on a new terminal to run the Python script you just wrote. Note that `rob301_simulation` is the package name, and `lab01_motor.py` is the file you just modified. You can refresh your browser to visualize the simulation on Gzweb. 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. (Alternatively, you can restart the teleop node to stop your model by pressing the space key, but make sure to never run teleop node and `lab01_motor.py` at the same time.)

Lab Deliverable 1: When you have successfully completed this task, you are required to demonstrate functionality by recording the successful run (i.e., the robot moves forward 1 m, and rotates 360° and stops). To demonstrate this, a rosbag file must be recorded by running:

```
$ rosbag record /cmd_vel /odom
```

prior to running `lab1_motor.py`. Remember to include in your report a summary of what the robot does within your rosbag file. State if you completed the task or explain why you were not able to.

5.2 Task 2: Subscribe to "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, θ), the Cartesian coordinates and the rotational coordinate relative to some specified laboratory frame.

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 5.

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 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_quaternion(quart)
    cur_pose = (point.x, point.y, theta)
    rospy.loginfo(cur_pose)
```

Note that the orientation of the robot is expressed in a quaternion (more about those in AER301), so you need to transfer that to angle using the following formulas:

```
def get_yaw_from_quaternion(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
```

```
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 5: 'odom' Output

Once you have finished, run `$ rosrun rob301_simulation lab01_odometry.py` on your remote PC to verify the output. You can terminate the node by typing Ctrl-C.

5.3 Task 3: Run Subscriber Node and Publisher Node Simultaneously

Now that 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 `lab01_odometry.py` and then launch the publisher node `lab01_motor.py`. Examine the output of the odometry: Does the output reflect the current position of the robot? How accurate is it?

6 Concluding Remarks

You should by now have a reasonable familiarity with the TurtleBot 3 Waffle Pi robot and a pretty good grasp on ROS. And you should be ready for Lab II…

7 Additional Resources

1. Introduction to ROS, ROB301 Handout, 2019.
2. Robotis e-Manual, `http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`.
3. "SSH: Remote control your Raspberry Pi," The MagPi Magazine, `https://www.raspberrypi.org/magpi/ssh-remote-control-raspberry-pi/`.
4. Official ROS Website, `https://www.ros.org/`.
5. ROS Wiki, `http://wiki.ros.org/ROS/Introduction`.
6. Useful tutorials to run through from ROS Wiki, `http://wiki.ros.org/ROS/Tutorials`.
7. ROS Robot Programming Textbook, by the TurtleBot3 developers, `http://www.pishrobot.com/wp-content/uploads/2018/02/ROS-robot-programming-book-by-turtlebo3-developers-EN.pdf`.