Institute for Aerospace Studies
UNIVERSITY OF TORONTO

LABORATORY I

Meet Your TurtleBot 3 Waffle Pi

(Sensor & Actuator Programming)

ROB521 Autonomous Mobile
Robotics Winter 2021

By: Brandon Wagstaff – 2019

1 Introduction

This is the first laboratory exercise of ROB521—Autonomous Mobile Robotics. The 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. 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 the 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 equipment and software that will be used for all labs in ROB521. 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.

In particular, you are:

- *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 upload to the Waffle Pi*
  - *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 the functionality of their programs to the TA during in-person labs, as is usually done, you will be required to demonstrate functionality by either recording the specified rosbag (a tool in ROS that can store all information that passes through a specified topic) files or a screen capture of your program running in simulation.

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.

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 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 this year we will be using a simulated TurtleBot3 within Gazebo (a robot simulation package).

3.1 TurtleBot 3 Waffle Pi

The simulated TurtleBot 3 Waffle Pi model is equipped with a camera, a two-dimensional (range and bearing) 360° lidar unit, an inertial motion unit (IMU), a compass and a gyroscope (or gyro, for short).

The Waffle Pi software consists of firmware for the OpenCR board and 4 ROS packages. It uses the OpenCR board as a subcontroller to estimate position by using the driving motor encoder values. 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 further refined in addition to the estimates from wheel odometry. 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 this lab, we will rely on the `turtleBot3` and `turtlebot3_simulations` packages, which contain the remote control package and the Gazebo simulator, respectively.

3.2 Sensors and Actuators

When the robot is initialized, all sensor outputs are published to the corresponding “topics” (see Introduction to ROS in Quercus). 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 is 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; it has a graphical interface as shown in Figure 3. 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 3 for a visualization of the lidar data. The lidar output is published in the /scan topic.

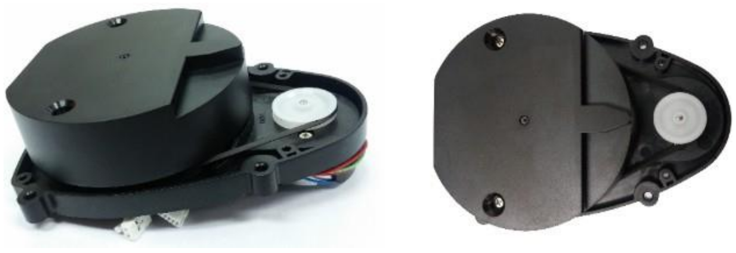
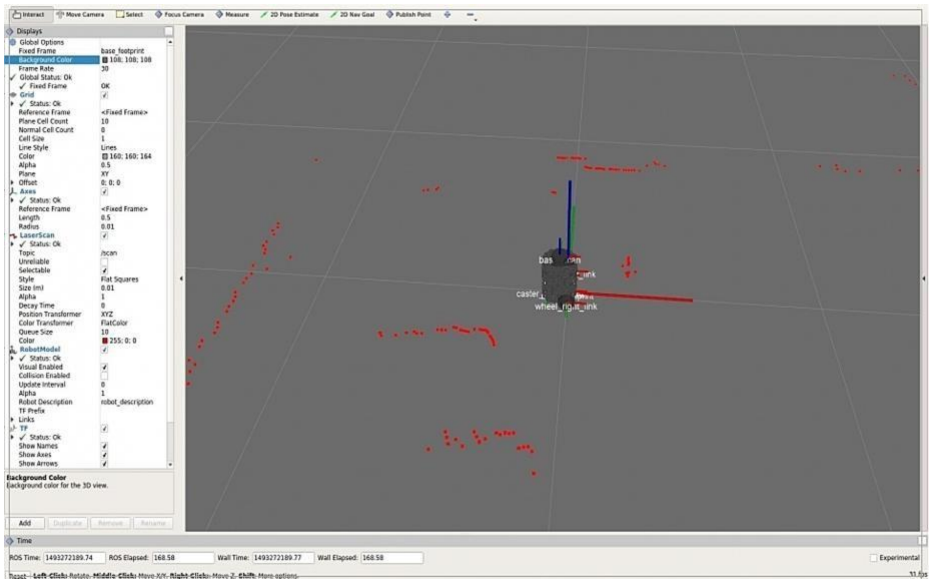


Figure 2: Lidar Sensor



```
Displays
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.

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

Figure 3: GUI of Lidar Sensor

Camera

The camera can be used for tasks such as feature mapping, bundle adjustment, and SLAM. 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

The Waffle Pi uses 2 Dynamixel actuators to drive the wheels. The motors can be operated by one of 6 operating modes, though the three most useful are

- Velocity control
- Torque control
- Position control

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

4 Getting Started

4.1 Remote Control of a Robot

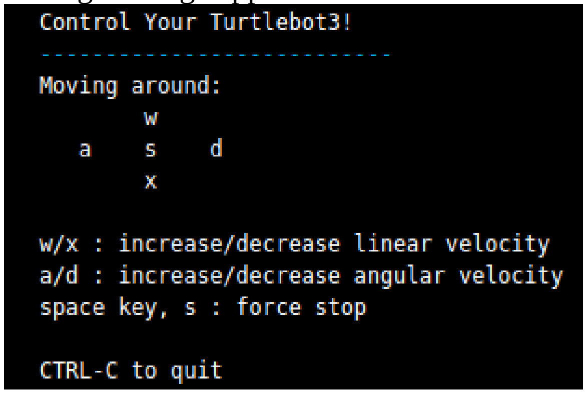
Follow the steps in the Connection Document to connect to your group's lab computer. Open a new terminal window on the lab computer and run:

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

This command should have started a roscore node and the Gazebo simulator nodes. Gazebo should have opened and your robot should appear on the screen. Open another terminal window and run:

```
$ 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', 'x' and control the translational and rotational velocity of the robot in units of meters per second and radians per second, respectively. Note the terminal with the message above will have to be your active terminal to control the robot.

Once you have confirmed that you can remotely control your robot using keyboard, press Ctrl-C to terminate the teleop node.

4.2 TurtleBot Topics

When TurtleBot Gazebo files have been executed, messages will be published from each node, such as sensors and actuators, to their corresponding *topics*.

Make sure the roscore and Gazebo nodes are running (which can be checked with `rosnode list`), 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 PC receives and processes the messages from the topics that are published by the user. Table 2 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 1: Subscribed Topics of TurtleBot PC

Topic Name	Message Type	Description
cmd_vel	geometry_msgs/Twist	Control the translational and rotational speed of the robot. Units in m/2, rad/s (actual robot control)

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) pf a model

Published Topics

Table 3 shows a list of the topics that are published by TurtleBot Gazebo. 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 the pose and twist information of objects in the Gazebo simulation. The state of a model is the state of its root *link*.

Table 3: Published Topics of TurtleBot PC

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 the

		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/rbg/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

To get more details on nodes and topics, run `$ rqt_graph` in terminal to check the publishing and subscribing activities of each node. A picture as in Figure 7 should pop-up in your window.

4.3 A Simple Publisher Node

Example

Let's examine the simple example in Figure 8 first. This example initializes a ROS node called 'talker' and then publishes messages to the topic 'chatter'.

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
```

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 like `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
```

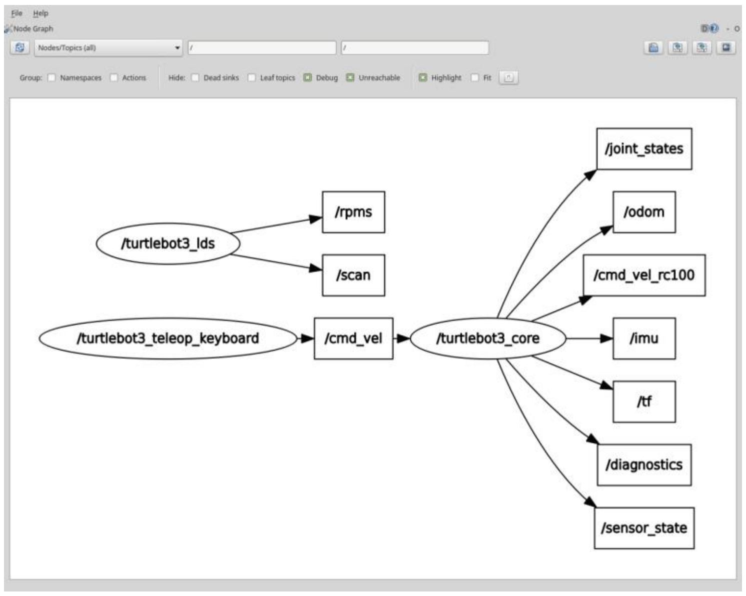


Figure 7: TurtleBot Nodes and Topics

```
#!/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 8: A Simple Example

In the `std msgs/String.msg` file, the message definition is string data. This means that there's only one field in `std msgs/String`, which is called the 'data' of string type. `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,

```
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; **note that 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. `rospy.loginfo(str)` does three things: prints to the terminal window, writes to the node's log file and writes to /rosout. `pub.publish(msg)` publishes a message type to the topic 'chatter'.

A Simple Subscriber Node

Example

The following example initializes a ROS node 'listener' and then subscribe from 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",
```

Notes on Running the Example Publisher and Subscriber

If you have coded these files and wish to run them, then place the files in your `~/catkin_ws/src/rob521_labs/lab1/nodes` folder. You will need to alter the permissions of these files using the terminal command `chmod +x {filename}.py`. Next, run another terminal command `rospack profile`. Now, you should be able to use the ros command `rosrun rob521_lab1 {filename}.py` to start both files. Each file will require its own `rosrun` command and terminal window. Additionally, you must have a `roscore` running in another terminal – for a total of three terminals. Further help on running these nodes can easily be found online on the ROS website.

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, then 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 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. **Be careful to either use a latch or sleep for 2 seconds after creating your publisher object (cmd_pub), especially if you're only sending a single message, or else your message may not actually be published. See** https://answers.ros.org/question/107521/when-is-it-safe-to-publish-after-declaring-a-rospypublisher/ **for more info.**

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 `$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch` file is up and running. You can run `rosnode list` and/or `rostopic list` to check.

You can use `$ 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. Note that once you've typed `rostopic pub -1 /cmd_vel` typed, you can use TAB to fill in the other fields (msg type, msg) with default values.

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 9.

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:

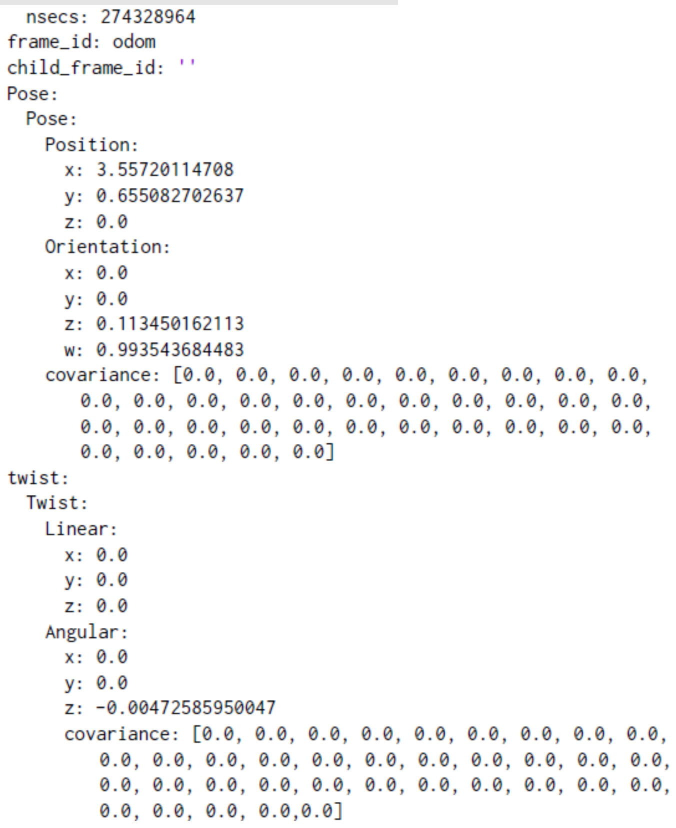
```
rospy.init_node('odometry')
odom_subscriber=rospy.Subscriber('odom',Odometry,callback,queue_size=1)
```

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, so you need to transfer that to a yaw 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
    rospy.loginfo(cur_pose)
```



```
  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: ‘odom’ Output

Once you have finished, run `$ rosrun rob521_lab1 odometry.py` 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.

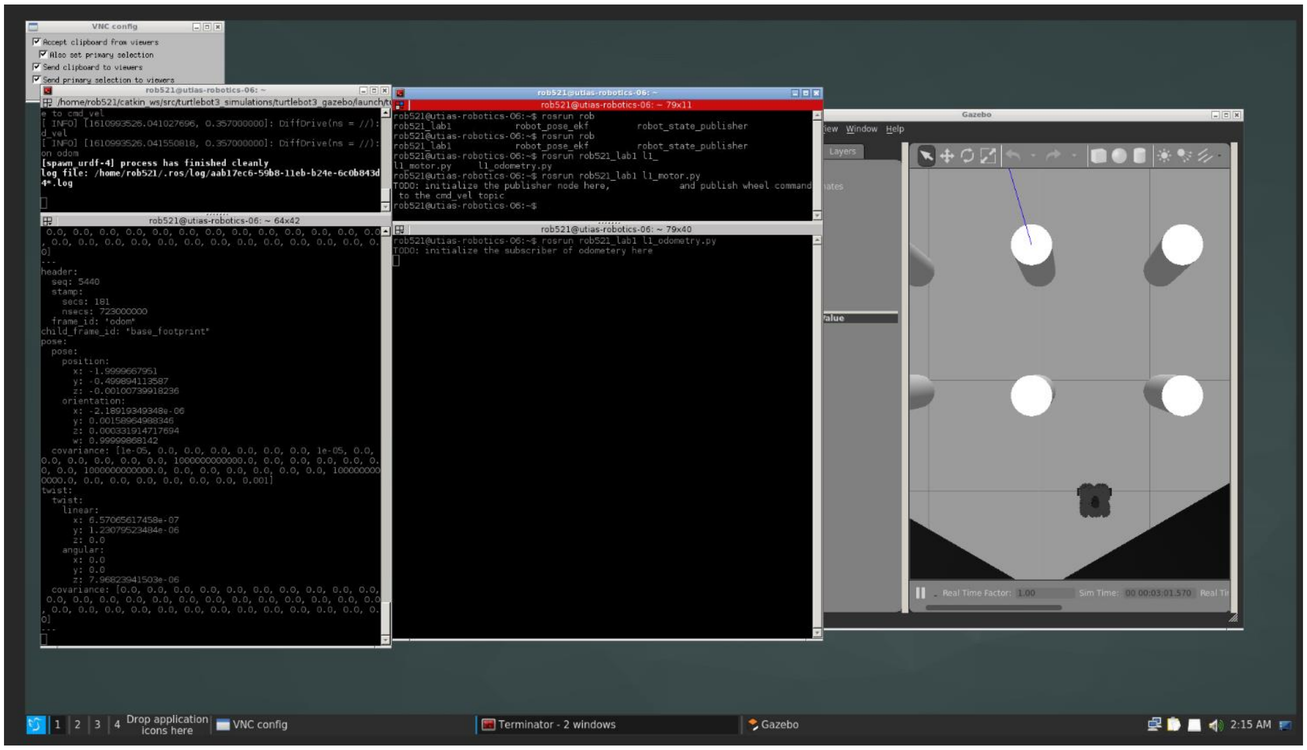
Launch the subscriber node `l1_odometry.py` and then launch the publisher node `l1_motor.py` . Examine the output of the odometry: does the output reflect the current position of the robot? How accurate is it?

Example Lab Deliverable: When you have successfully completed this task, you are required to demonstrate the functionality by recording the successful run (i.e the robot moves forward 1m, rotates 360°, and stops).
To demonstrate this please record a mp4 video that includes three windows:

- A window of the robot performing this maneuver in gazebo.
- A window where you print out the (x, y, θ) calculated by l1_odometry.
- A window of the `$ rostopic echo /odom` command

A screenshot from an example submission can be seen in Figure 10 – on the next page. The left window is echoing /odom, the center window prints out the results from l1_odometry, and the right window shows the gazebo simulation. You do not need to use the exact simulation environment shown in Figure 10.

This lab deliverable is an example of what can be expected from future labs and **does not need to be submitted** to the TAs. A lab report is not required for this week.


VNC config
Accept clipboard from viewers
Also set primary selection
Send clipboard to viewers
Send primary selection to viewers
Gazebo
View Window Help
Layers
Value
Real Time Factor: 1.00
Sim Time: 00 00:03:01.570
1 2 3 4 Drop application icons here
VNC config
Terminator - 2 windows
Gazebo
2:15 AM

```
rob521@utias-robotics-06: ~
/home/rob521/catkin_ws/src/turtlebot3_simulations/turtlebot3_gazebo/launch/t
e to cmd_vel
[ INFO] [1610993526.041027696, 0.357000000]: DiffDrive(ns = //):
d_vel
[ INFO] [1610993526.041550818, 0.357000000]: DiffDrive(ns = //):
on odom
[spawn_urdf-4] process has finished cleanly
log file: /home/rob521/.ros/log/aab17ec6-59b8-11eb-b24e-6c0b843d
4*.log
```

```
rob521@utias-robotics-06: ~ 64x42
 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]
---
header:
  seq: 5440
  stamp:
    secs: 181
    nsecs: 723000000
  frame_id: "odom"
child_frame_id: "base_footprint"
pose:
  pose:
    position:
      x: -1.9999667951
      y: -0.499894113587
      z: -0.00100739918236
    orientation:
      x: -2.18919349348e-06
      y: 0.00158964988346
      z: 0.000331914717694
      w: 0.99999868142
  covariance: [1e-05, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1e-05, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 1000000000000.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 1000000000000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 100000000
0000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.001]
twist:
  twist:
    linear:
      x: 6.57065617458e-07
      y: 1.23079523484e-06
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 7.96823941503e-06
  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]
---
```

```
rob521@utias-robotics-06: ~
rob521@utias-robotics-06: ~ 79x11
rob521@utias-robotics-06:~$ rosrun rob
rob521_lab1           robot_pose_ekf        robot_state_publisher
rob521@utias-robotics-06:~$ rosrun rob
rob521_lab1           robot_pose_ekf        robot_state_publisher
rob521@utias-robotics-06:~$ rosrun rob521_lab1 l1_
l1_motor.py     l1_odometry.py
rob521@utias-robotics-06:~$ rosrun rob521_lab1 l1_motor.py
TODO: initialize the publisher node here,            and publish wheel command
 to the cmd_vel topic
rob521@utias-robotics-06:~$
```

```
rob521@utias-robotics-06: ~ 79x40
rob521@utias-robotics-06:~$ rosrun rob521_lab1 l1_odometry.py
TODO: initialize the subscriber of odometery here
```

Figure 10: A single frame from an example submission

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.

7 Additional Resources

1. Introduction to ROS, ROB521 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.