Lab Session 2

Release date: Friday, 11 February 2022
Due date and time: 10 am on Friday, 18 February 2022

This lab session will introduce you to the TurtleBot in simulation, as well as writing a publisher to give commands to the robot to get it to move in a specified direction.

Making sure your git clone is up-to-date

Before you do anything else, make sure your git clone of lab2 is up-to-date:

cd $HOME/catkin_ws/src

git pull

This should get the new lab2 files under src/lab2.

Set up the singularity environment

Using steps described in lab 1, create a singularity container and then a few xterm terminals to use for later.

Gazebo simulation environment

In a terminal, run the following command:

roslaunch turtlebot_gazebo turtlebot_world.launch

> ### If you are working over a remote connection:
> Use "vglrun" at the beginning of the command above:
>
> vglrun roslaunch turtlebot_gazebo turtlebot_world.launch
>
> If you are working in an on-campus computer lab, ignore these blue boxes.

(The first time you run this command, it may take a few minutes to load. Please be patient and wait until a new window with a virtual environment opens up.)

This brings up the Gazebo simulator where you can see the Turtlebot and objects in a simulated environment. Use your mouse's scroll wheel to zoom in and out. Press and hold your mouse's middle button, while moving the mouse. This will rotate your view in the world. Similarly, experiment with your mouse's left and right buttons. Explore the three-dimensional virtual environment and the robot.

At the top of your screen, you will see a menu:

The leftmost pointer mode is currently chosen. In this mode, your mouse will simply move your viewer in the world. Now choose the second from the left (). In this mode, go and hold on an object in the virtual world, and drag it around. In this mode you can move the objects in the world. You can move the robot as well just like any other object.

Turtlebot ROS nodes and topics

You now have running the nodes that provide basic functionalities of the TurtleBot. For example, this will bring up the controller for the TurtleBot's mobile base, drivers for TurtleBot's sensors, etc. If you want to see a list of all the ROS nodes that are running, in a new terminal do this:

rosnode list

and inspect the result. Many of the nodes are named to explain their functionality, so try to guess what each one does. Similarly, to see all the ROS topics that are created, do this:

rostopic list

These are all the topics required for the complete functioning of the TurtleBot. Can you see the one named "mobile_base/commands/velocity"? As the name suggests, this topic is used to send velocity commands to the mobile base of the robot. We will use it today.

Understanding the mobile_base/commands/velocity topic

To see the publishers and subscribers to the topic, run:

rostopic info /mobile_base/commands/velocity

The subscriber of this topic controls the mobile base. To move the robot, we will write a new node that will publish to this topic.

The rostopic info command also shows us the type of the topic. It is geometry_msgs/Twist. If we want to publish to the topic, we need to use this type. What is in a geometry_msgs/Twist type? The command "rosmsg show" is useful for that:

rosmsg show geometry_msgs/Twist

The result shows us that Twist is actually a type that contains two other types in it, specifically the Vector3 type. Each Vector3 type includes three float values. Notice that one of the Vector3 types are named 'linear' and the other 'angular'. It is becoming more clear now: This is a message type to specify linear and angular velocities in three dimensions.

Handling dependencies

In our package we will write code that uses message types from geometry_msgs, which is simply another ROS catkin package. When one package uses types defined in another package, this is called a dependency: Our package needs to depend on geometry_msgs. We indicate dependencies through the package.xml file in a package. Go to the lab2 directory, open the package.xml file in it, and add this somewhere between the <package> tags:

<build_depend>geometry_msgs</build_depend>

Go back to your catkin workspace and catkin_make.

Simple forward/backward motion

You will find the file firstwalk.py in lab2 directory. Open and inspect it. Particularly notice that since we need access to the geometry_msgs/Twist message type, the script imports it like this:

from geometry_msgs.msg import Twist

You always need to import a message type if you are going to use it in a script.

Also notice how we set the x component of the linear velocity to a desired value.

Run this script and see what happens (you might need to make the script executable before you run it) :

rosrun lab2 firstwalk.py

From inspecting the script you should notice that the reason that the robot only moves backwards and forwards is because we only assigned values to the linear x component of one of the vectors. You will have noticed that we did not assign any angular velocity at all. This is why the robot only moves backwards and forwards without turning. Experiment with publishing instructions while altering the values of the velocity. Note that each Vector3 has 3 components (x, y and z). See what the results of differing combinations are. Try to understand why.

Notice that the firstwalk.py script uses the velocity 0.2 to command the wheels. The units are in m/s.

Circle and square

To get the robot turning we must also supply a value for the angular velocity. More specifically the z component of the angular velocity.

Exercise 1. Create a script (name it exercise1.py) that will continuously drive the robot in a circle. You will need to use a loop to constantly publish the message, however only one Twist message will actually need to be created. It is important to remember that you can deliver angular and linear velocities at the same time. A well traced fluid circle comes from one motion of both linear and angular velocities combined. **Important: Please read the section "Writing infinite loops in ROS" in the Appendix of this document, before starting to work on this exercise.**

Exercise 2. Create a script (name it exercise2.py) that will trace a square by driving the robot. The unit of the angular velocity is in radians/sec, positive representing anti-clockwise movement. Take that into consideration when trying to program a turn for a nice square.

Great! You now know how to move the TurtleBot.

Bumper input

When you start up the TurtleBot software it does not only start the controller for the mobile base, but it also starts drivers for sensors. If you look at the output of 'rostopic list' again, you can see different sensor topics. Here we will mention only one, but you can experiment with others as well. The topic "/mobile_base/events/bumper" is a topic that outputs values based on the bumpers on the TurtleBot base. If you have the real Turtlebot in front of you, please locate the two bumpers on the left and right of the robot. Use rostopic info to inspect this topic.

rostopic info /mobile_base/events/bumper

You will see that the messages are of type kobuki_msgs/BumperEvent. You should use rosmsg show to inspect the details of the BumperEvent message type.

One quick way to make sure this topic works as expected is to listen to it on command line using "rostopic echo":

rostopic echo /mobile_base/events/bumper

(Ignore any warnings you might get about “/clock”)

Now, if you were following this using the real robot, you could press on the bumpers with your hand, while you keep your eye on this terminal window. You should see that the state variable is set to 1, when bumper is pressed.

In the simulator, you will not be able to press the bumper with your hands. Instead, move the robot and/or the objects in the virtual world, such that there is an object in close proximity in front of the robot. Then run firstwalk.py to make the robot move forward. When the robot hits the obstacle, you should again see the bumper state being set to 1.

Exercise 3. Copy your script that moves the robot on a square into a new file (name it exercise3.py), and change it such that the robot stops when a bumper is pressed. You will need to add a subscriber to the correct topic.

Submission instructions: When you think you have completed this worksheet, add, commit, and push your new files (exercise1.py, exercise2.py, exercise3.py, and any other related files you might have created) and changes to the git repository with the correct git message “Lab2 completed.”. **You must do this before the deadline** and your commit with this specific message is what informs us that you have submitted your files and we can then check your files. Specifically, you can use the commands below to commit and push your code to git with the correct message:

cd $HOME/catkin_ws/src/lab2/src/

git add <list-of-files-you-want-to-push-to-git>

git commit -m “Lab2 completed.”

git push

Please also read the other useful information in the following pages.

Other useful information

The bumper is a simple sensor. There are many other sensors on the TurtleBot, including cameras.

Cameras

There are two cameras on the TurtleBot. One camera is the 3D sensor's camera. The 3D sensor provides both RGB and depth values which are published as separate ROS topics. The other camera is the laptop's camera, which you are also welcome to use. The laptop camera exists only on the real Turtlebot, not in simulation.

Now we will explain how you can use the image streams from the 3D sensor.

3D sensor

Now look at the list of topics again ("rostopic list"). There should be plenty of /camera/ topics including /rgb/ and /depth/. For example, /camera/rgb/image_raw is the topic for the raw RGB image from the camera. You can try "rostopic echo" on it, but it will dump a large binary stream; i.e. "rostopic echo" is not that useful to inspect binary messages. But ROS developers provide a tool to listen to and display images. The tool is called image_view. Run it like this:

rosrun image_view image_view image:=/camera/rgb/image_raw

You should see the RGB image from the robot's 3D sensor in a new window. This image is the "live" image: if you move your robot, for example by executing the first_walk.py, this image should reflect what your robot sees.

Quit image_view after you are done. In the next lab session, you will learn how to write a node that listens to an image topic and make the robot react to what it sees.

The 3D sensor also outputs depth (distance) values. The topic "/camera/depth/points" include the depth data. Execute "rostopic echo" on it and you will see lots of values being streamed. It is not binary like the RGB image, but still difficult to interpret. We will use yet another tool to inspect this data. This tool is called Rviz (for "ros visualizer"). Rviz is a powerful and useful tool that you should get familiar with. Here is the rviz user guide: http://wiki.ros.org/rviz/UserGuide . Run it:

rosrun rviz rviz

If you are working over a remote connection:

Use "vglrun" at the beginning of the command above:

vglrun rosrun rviz rviz

If you are working in an on-campus computer lab, ignore these blue boxes.

As described in Section 4 of the rviz user guide, add a new display for the PointCloud2 message type. On the left pane, enter the topic name for this display as /camera/depth/points. Change the value of the "Fixed Frame" field under "Global Options", and select "odom". You should now see the 3D point cloud from the 3D sensor. It is also "live": if you move the robot, the point cloud should reflect what the robot sees. You can use your mouse buttons to move around in the 3D display in Rviz. After you are done, quit rviz.

Appendix.

Writing infinite loops in ROS

If you want to write infinite loops in your ROS code, one way to do this in Python is:

```
while True:
        <other commands>
```

Please do **not** use the loop style above when using ROS. Instead, here is the correct way to do it:

```
while not rospy.is_shutdown():
        <other commands>
```

The code above will loop until rospy is shutdown, e.g. when you ctrl-c your code.