EE106A: Lab 6 - Introduction to TurtleBot*

Fall 2018

Goals

By the end of this lab you should be able to:

- Launch the TurtleBot and drive it around with your keyboard
- Run the `gmapping` example to perform SLAM, then plan through the mapped space
- Put an AR tag on top of the TurtleBot and track it with `ar_track_alvar`
- Control the turtlebot to follow an AR tag on the ground.

Relevant Tutorials and Documentation:

- `ROS tf package`: `http://wiki.ros.org/tf`
- `ar_track_alvar`: `http://wiki.ros.org/ar_track_alvar`

Contents

1 Introduction to the TurtleBot

TurtleBot is one of the classic platforms for mobile robotics research and teaching. There are three versions, the most recent of which came out last year. We will be using the classic TurtleBot 2 in this class, since it is still the most common and best supported.

The TurtleBot 2 is an example of a tank drive or ”unicycle model” robot, which means that it can rotate in place (Unlike a car, or ”bicycle model”). The turtlebots we will use in class come equipped with a gyroscope and a Kinect. Kinects allow

*Developed by David Fridovich-Keil and Laura Hallock, Fall 2017. Extended by Valmik Prabhu, Ravi Pandya, Nandita Iyer, and Philipp Wu, Fall 2018 the Turtlebot to perceive both RGB color and depth information. In addition, there's an onboard Linux computer running ROS. The two main drive wheels have *encoders* that can measure how far each wheel has turned at any given time, and the bumper in the front can tell when the TurtleBot is in collision with an object.

We have five (or more) TurtleBots available for you to use, and each is color-coded according to the acrylic platforms holding the onboard computer and Kinect. The turtles have the following identifiers:

black	blue	green
red	yellow	orange
pink	*cyan*	*white*

The italicized turtlebots are forthcoming, and their names have not yet been set in stone.

1.1 Intro to the Lab

In this lab, we will help you get a feel the sorts of things you can do with a TurtleBot. We'll focus on two applications: *simultaneous localization and mapping* (SLAM), and control. SLAM is a method whereby the sensors are fused together to allow the robot to map (mapping) an environment while simultaneously locating itself within the map (localization). SLAM is so important to the robotics community that we will be learning how to implement our own version next lab; for now, the focus will be on getting a sense of what the sensors tell us and what it looks like when we combine information from multiple sources.

The other part of this lab is control. Control is one of the largest subdisciplines in robotics, and it's used everywhere from industrial robot arms to airplane autopilots to self-driving cars. For a particularly beautiful example of a control system in action, check out this video. While the controller we'll be implementing is far less complex, we hope it'll give you a teaser of the controls you'll be learning later in this class and (hopefully) in your future studies.

SLAM and mobile robot control are two of the biggest problems in the burgeoning field of autonomous driving, and we know that many students are interested in working in the field. We hope that the exposure you get in this lab primes you to learn more on your own, so you can work on tackling these problems in your final projects, your research, or your careers.

Portions of this lab draw heavily on the TurtleBot ROS tutorials, which may be found on the TurtleBot website. The online tutorials are a great resource for discovering what the TurtleBot can do and for debugging any issues you may encounter.

1.2 How do I turn this thing on?

You'll first need to reconfigure the network so that the TurtleBot is the ROS master rather than your computer. This can be done by opening your `.bashrc` file and adding the line

```
export ROS_MASTER_URI=http://[TurtleID].local:11311
```

Note that you should replace `[TurtleID]` with the network name of your TurtleBot (`blue`, `orange`, `black`, etc.).

Once you've made this change, you'll need to re-source your `.bashrc` file in any open terminal windows that you plan to use this lab. (Any terminal windows you open after making this change will automatically incorporate it, since they run `.bashrc` on launch.)

When you're done working with the turtlebots, you should remove this command from your `.bashrc` file. If you don't you won't be able to run a `roscore` on your home computer properly. Note that reverting this change will only take effect after logging out of your computer. To work on other things immediately, type

```
export ROS_MASTER_URI=http://localhost:11311
```

in all terminal windows you plan to use.

Sometimes, when trying to connect to your turtlebot, you'll get a hostname error. If you run into this error, see the Appendix at the end of the lab for a potential solution.

Before you can listen for messages or give commands to the TurtleBot, you'll have to turn it on. Begin by turning on the power and checking that you can `ping` the onboard computer. (Run `ping [TurtleId]` to make sure. Note that some of the TurtleBots take several minutes to wake up.) Now, before you can do anything interesting with the TurtleBot, you'll have to `ssh` into the onboard computer.

The username for all turtlebots is `turtlebot`, and the password is `EE106A18`.

```
ssh [user]@[TurtleID].local
```

Now run the TurtleBot *bringup* sequence:

```
roslaunch turtlebot_bringup minimal.launch --screen
```

The TurtleBot is now launched, along with all of its sensors, and it is ready to receive motion commands. When you're done using a turtlebot, you can close the ssh connection by typing `exit` in the command line.

1.3 Controlling the TurtleBot

TurtleBot commands are sent over the topic `cmd_vel`. Later, we'll ask you to build your own autonomous controller, but for now, just use the built-in keyboard teleoperation node. Open a new terminal window and `ssh` into the TurtleBot (keep the minimal.launch running). Then run the following:

```
roslaunch turtlebot_teleop keyboard_teleop.launch --screen
```

Try driving the TurtleBot around. What happens to the IMU messages when you're driving, especially when you start and stop?

1.4 Visualizing the Kinect

Kinect is an incredibly powerful sensor, so even though we will not be making extensive use of it in the labs for this course, we expect you to get some familiarity with it, and we hope that some of you will use one in your final project. There are actually several different versions of the Kinect, and they each work a little bit differently. If you're interested, we encourage you to figure out how the depth sensing actually works in each one; you might find it useful later on if you use one for a final project.

For now, let's just examine the ROS interface. On the TurtleBot, run

```
roslaunch turtlebot_bringup 3dsensor.launch
```

Then, on your local computer, run

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Change Global Options > Fixed Frame to `base_link`, then modify the view type to `ThirdPersonFollower (rviz)` (upper right corner). Next, use Add > By Topic > `/camera/depth_registered/image` to add a visualization of the Kinect data. Examine the RViz display. What happens when you move the TurtleBot around?

2 An example application: SLAM

Now that we've seen how some of the TurtleBot's sensors work, let's see what happens if we fuse information coming from multiple sensors together. Specifically, we will run a built-in demo that performs simultaneous localization and mapping, or SLAM, to create a 2D floor plan of the lab. We'll explain a bit more about SLAM and how it works in Lab 8. For now, just try to get a rough sense of what's going on under the hood.

First, close down all existing RViz displays and all programs running on the TurtleBot except for the *bringup* sequence. Log into the TurtleBot's onboard machine and run

```
roslaunch turtlebot_navigation gmapping_demo.launch
```

Now, on your own machine, run

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

If you run the keyboard teleoperation node now, you should be able to drive the TurtleBot around and generate a floor plan. See if you can figure out what the different colors on the map correspond to. How do you think the map is being generated?

3 Planning with MoveIt!

The `gmapping` demo automatically integrates with the MoveIt! motion planning library. (See Lab 5 for more uses of this library.) Try playing around with this planning functionality by clicking the "2D Nav Goal" button at the top of the RViz window, then clicking the desired goal point on the map and dragging in the direction you want the TurtleBot to be facing. Your TurtleBot should then navigate to this point. A couple of notes:

- The keyboard teleoperation node will override the motion planner's control. If your TurtleBot isn't moving to your indicated location, close this node and try again.
- The RViz display can sometimes be a bit buggy when running on your local machine. If the planner isn't working, `ssh` into the TurtleBot using

  ```
  ssh [user]@[TurtleID].local -XC
  ```

  and try running the RViz display on the TurtleBot itself. (It will likely run a bit slower. The `-XC` above allows compressed X11 forwarding, which means you can run graphical applications remotely.)

Checkpoint 1

Get a TA to check your work. At this point, you should be able to show us a floor plan of the lab and interpret what you see. Can you explain any flaws in the map you generated? Why does the TurtleBot have difficulty near some objects like chairs and table legs? Why *doesn't* it have as much difficulty with moving obstacles like other TurtleBots?

You should also be able to plan and execute paths through this map. How well does this work? Based on what you see, can you predict when Skynet will destroy humanity?

---

4 Localization with AR tags

In Lab 8, we will be creating our own version of the `gmapping` SLAM demo you just ran. One thing we'll need is a way to determine where the TurtleBot is — that is, we'll need to *localize* the TurtleBot. To do so, we will use our own version of GPS: AR tags!

Ask the GSI for an AR tag, and attach it to to top of the TurtleBot (if there is not already one on the TurtleBot). Get an additional AR tag (with a different identifier than the one on your TurtleBot).

You will be using the webcam attached to your computer to compute the relative transformation between the TurtleBot and the AR tag on the ground, which will serve as the origin.

Affix your webcam so that it has a good view of both the TurtleBot's AR tag and the static tag on the ground. Use RViz and `ar_track_alvar` to visualize the location of the TurtleBot relative to the frame defined by the AR tag on the ground. You'll need to set the fixed frame in `RViz` to correspond to the AR tag we've placed on the ground, and you'll also need to make sure that the AR tag on top of the TurtleBot is linked to a frame on the TurtleBot (e.g., `base_link`). You should look up the `static_transform_publisher` within the `tf` package to do this ($Hint$: the transform $[x\ y\ z\ r_x\ r_y\ r_z]$ from the top plate of the Turtlebot to the base link of the Turtlebot is $[0.014\ \ 0\ \ -0.397\ \ 0\ \ 0\ \ 0]$). What visualizations, topics, and transform echoes are useful in debugging your system?

Create a new ros workspace called `lab6` and refer to Lab 4 for how to acquire and use the `ar_track_alvar` package. In the `ar_track_alvar` package you will need to edit `webcam_track.launch` by changine the `marker_size` argument to be 17.7. This is to account for the use of larger AR tags.

Checkpoint 2

Get a TA to check your work. At this point, you should be able to drive the TurtleBot around with your keyboard and visualize in RViz where your AR tag localization node thinks the TurtleBot is with respect to the tag on the ground.

---

5 Proportional Control with Turtlebots

You've been using the MoveIt GUI to command the robot to move to a location. Instead, we'll write a rudimentary controller to do so for us. We'll be commanding the turtlebot to drive until it touches the AR tag on the ground.

This section will synthesize tools you've learned in Labs 2, 3, and 4 in a working system; if you need a refresher, you're encouraged to refer to the earlier lab documentation, particularly concerning the `turtlesim` keyboard controller, your implementation of `tf_echo`, and use of the `ar_track_alvar` package.

A feedback controller works by taking the error between the current state and the desired state, and using it to generate a control input. We'll be implementing a *proportional* controller, or P controller, so the control input will be proportional to the error. For this problem, let's take the turtlebot's XY position in space as our state: $q = [x, y]^T$. Incorporating angle would make this problem significanlty harder (why do you think this is the case?) so we ignore it in this exercise. A proportional control law could look like this:

$$\dot{q} = K(q_d - q) \tag{1}$$

where $\dot{q}$, or the velocity, is our control input and q_d is our desired state. Expanded, it could look like this:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} K_{xx} & K_{yx} \\ K_{xy} & K_{yy} \end{bmatrix} \begin{bmatrix} x_d - x \\ y_d - y \end{bmatrix} \tag{2}$$

Of course, turtlebots are a bit more complicated, because they cannot drive sideways. This is called a *nonholonomic* constraint. If you choose to take EECS C106B/206B or an advanced dynamics class in the mechanical engineering department, you'll learn a lot more about them. We cannot control $\dot{y}$, only $\dot{x}$ and $\dot{\theta}$. Therefore, we'll modify the control law to look like this:

$$\begin{bmatrix} \dot{x} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} K_1 & 0 \\ 0 & K_2 \end{bmatrix} \begin{bmatrix} x_d - x \\ y_d - y \end{bmatrix} \tag{3}$$

Here we get rid of the two non-diagonal terms and determine $\dot{x}$ solely using $x_d - x$, and $\dot{\theta}$ using $y_d - y$. Note that x and y, as well as $\dot{x}$ should be determined in the *body frame* of the turtlebot, rather than the spatial velocity. Given this information, what sign should K_1 be? What about K_2?

Create a new ROS package called `turtlebot_control` with `rospy, roscpp, std_msgs,` and `geometry_msgs` as dependencies. Download the `lab6_resources.zip` file from Bcourses in your new package You will be editing `turtlebot_control.py` to include a proportional controller, which will command the turtlebot to drive to the target AR tag. You'll be getting position information using the `tf2_ros`, which you learned about in Lab 3, and you'll be sending velocity messages using a publisher, which you did with `turtlesim` in Lab 2. Approximate magnitudes of K_1 and K_2 should be 0.3 and 1 respectively. You can run this file by running

```
rosrun turtlebot_control turtlebot_control.py frame1 frame2
```

where `frame1` is the TF frame of your turtlebot, and `frame2` is the TF frame of the target AR tag.

Checkpoint 3

Get a TA to check your work. At this point, you should be able to:

- Command the Turtlebot to drive to the target AR tag.
- Explain if/why the Turtlebot doesn't reach the AR tag exactly.
- Move the target tag and have the turtlebot follow.
- Describe how you would improve the performance of your controller (you don't need to implement these improvements).

---

6 Appendix

ssh hostname error

If you are running into a hostname error try the following changes.

```
export ROS_HOSTNAME=$(hostname --short).local
```

file in your `.bashrc` file to

```
export ROS_HOSTNAME=192.168.1.[COMPUTER_NUMBER]
```

where `[COMPUTER_NUMBER]` is your computer number (1 through 10). Note that you should change it back after you get the turtlebot working, otherwise you'll have to edit the `.bashrc` file everytime you work with a new computer.