Lab 4 - Introduction to TurtleBot(3)*

EECS/ME/BIOE C106A/206A Fall 2022

Goals

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

- Launch the TurtleBot and drive it around with your keyboard
- Run the necessary commands to perform SLAM and LiDAR mapping, 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
- `starter code`: https://github.com/ucb-ee106/106a-fa22-labs-starter/tree/main/Lab4

Contents

*Extended and adapted for Turtlebot3's by Emma Stephan and Marius Wiggert, Fall 2022. Developed by David Fridovich-Keil and Laura Hallock, Fall 2017. Extended by Valmik Prabhu, Amay Saxena, Aakarsh Gupta, Ravi Pandya, Nandita Iyer, and Philipp Wu, Fall 2019

1 Introduction to the TurtleBot

TurtleBot is one of the classic platforms for mobile robotics research and teaching. There are three versions, we just upgraded to the most recent one, TurtleBot 3 this year.

The TurtleBot 3 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 360 degree LiDAR for depth information and an RGB camera for color perception. In addition, there's an onboard Raspberry Pi computer running ROS. The two main drive wheels have *encoders* that can measure how far each wheel has turned at any given time.

We have six TurtleBots available for you to use, each with a unique name: kiwi, lemon, apple, mango, cherry, banana.

1.1 Intro to the Lab

In this lab, we will help you get a feel for 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.

1.2 How do I turn this thing on?

If you recall in Lab 1, you first must ensure that your `.bashrc` file has the correct lines uncommented for working with TurtleBots (and the lines needed for Sawyer packages are commented out). Please refer to the Lab 1 doc linked above if you have forgotten how to check this.

Next you will be changing the `.bashrc` file of the TurtleBot. First, please run the command you tested in lab 1 that allows you to view available IP addresses of all robots and computers (cat /etc/hosts), and look at what the corresponding IP address of your Turtlebot is. Now, unplug your Turtlebot3 from the charging station, and turn on the switch on the side opposite the blue battery pack. Before you ssh into the TurtleBot, we suggest pinging your TurtleBot to ensure it is turned on. This can be done with the following command, where "fruitname" refers to the specific fruit name on the wood plate on top of your TurtleBot.

```
ping fruitname
```

You can ssh into the TurtleBot using the login info below:

```
ssh fruitname@fruitname
Password: fruitname2022
```

Once you have ssh'd into your TurtleBot, you need to configure the network so that your computer is the ROS master, but the TurtleBot is the host. This can be done by opening the `.bashrc` file on your TurtleBot and ensuring that the following lines exist in the file: *NOTE: THESE LINES SHOULD ALREADY EXIST TOWARDS THE BOTTOM OF YOUR FILE. YOU SHOULD NOT HAVE TO EDIT THIS FILE, WE JUST WANT YOU TO LOOK AT IT.*

```
export ROS_MASTER_URI= your workstation ip
export ROS_HOSTNAME= your turtlebot ip
```

Once you've assured these lines are correct in both your local and Turtlebot `.bashrc` files, you'll need to re-source your `.bashrc` file in any existing open terminal windows that you plan to use this lab, meaning that you source your local computer's file in any terminal windows that are not ssh'd into the Turtlebot, and you source the Turtlebot file in any terminals that are ssh'd into the Turtlebots. Remember, both cases require the exact same command for sourcing the `.bashrc` file that we used in lab 1. (Any terminal windows you open after making this change will automatically incorporate it, since they run `.bashrc` on launch.) You should recognize that this was the case for lab 1 and lab 2, and will be the case for all labs.

By now you should know that when you plan to use ROS, you must ALWAYS run roscore. Do this before moving to the next step.

Now run the TurtleBot *bringup* sequence:

```
roslaunch turtlebot3_bringup turtlebot3_robot.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 Reading the IMU

Let's start by reading the IMU. An IMU is a combination of accelerometers, gyroscopes, and magnetometers which are used in conjunction to orient the robot. A 6-axis IMU is the most common type of IMU, and contains 3 accelerometers and 3 gyroscopes. Accelerometers measure acceleration along one axis (so three measure along x, y, and z) while gyros measure rotational velocity about one axis (so three measure about x, y, and z).

See if you can figure out on which topic the turtlebot's IMU is being published. (*Hint*: the online documentation for `rostopic list` may be useful.) Once you've found it, call `rostopic echo` on the appropriate topic and see if you can figure out to what all the numbers correspond. It might help to have one person pick up the TurtleBot and hold it steady, then move it in each direction. Which values can you measure?

1.4 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 turtlebot3_teleop turtlebot3_teleop_key.launch
```

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

2 An example application: SLAM

Now we'll learn how to visualize the turtles sensors and fuse the information to construct a map. 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.

2.1 Visualizing the LiDAR Sensor

LiDAR is an incredibly powerful sensor that provides depth estimates by sending laser pulses and measuring how long they take to return. Even though we will not be making extensive use of the raw measurements in the labs for this course, we expect you to get some familiarity with it.

First open the RViz viewer on our workstation. For that, run

```
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

You should now see a map with your Turtlebot at the center. For now turn off the Map by unchecking this box. you should now see a lot of green dots in RViz these are the LiDAR depth measurements of the robot. Try this: take the Turtlebot in your hand and move it around and see what happens to the depth measurements? What happens if your hold your hand in front of the rotating LiDAR, can you see that in RViz?

2.2 Mapping the room with Lidar

Now shut down the slam module node on the terminal and restart it with the robot standing still on the ground. This time we leave the Map checked because this is the map the robot creates using SLAM.

Now run the keyboard teleoperation node and drive the TurtleBot around to generate a floor plan of a part of the lab room. See if you can figure out what the different colors on the map correspond to. How do you think the map is being generated?

2.3 Saving the Map

Now we want to save our map so we can then localize ourselves in it afterwards and then plan with the map information.

For that create a lab4 folder in your ros_workspaces folder, then create a saved_maps folder in it. Now spin up a node on your workstation that will save the file to this folder.

```
rosrun map_server map_saver -f ~/ros_workspaces/lab4/saved_maps/slam_map
```

You can examine the map file that got created in this folder.

3 Planning on your SLAM map with MoveIt!

Now we will load up the map, localize the robot in it and then move around.

3.1 Loading the map in the navigation module

Lets start the navigation module in RViz, it requires the map you just saved and then uses the MoveIt! motion planning library under the hood to plan on this map. (See Lab 5 for more uses of this library.)First make sure your slam node is shut down, then from your workstation run (this is a single line not multiple lines):

```
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
 ↪  map_file:=$HOME/ros_workspaces/lab4/saved_maps/slam_map.yaml
```

3.2 Localizing the robot on the map

Now we must perform Initial Pose Estimation before running the Navigation as this process initializes where our robot is on the map (the navigation module does not know that a priori). For that click the 2D Pose Estimate button in the RViz menu. Now click on the map where the actual robot is located and drag the large green arrow toward the direction where the robot is facing. Now you see a bunch of tiny green arrows around the robot, these are the estimated locations the robot thinks it is at in the map.

Move the robot manually back and forth a bit (using the teleop node) to collect the surrounding environment information and narrow down the estimated location of the TurtleBot3 on the map.

3.3 Move it around with Motion Planning

Now that our robot is fairly certain where it is on our map, lets move it around. You can select a navigation goal for the robot bu 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.

Checkpoint 1

Submit a checkoff request for a staff member to come and check off your work. At this point you should be able to:

- Drive the TurtleBot around with your keyboard.
- Show 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 detecting other TurtleBots, even if they are moving?
- Plan and execute paths to a certain position and orientation through this map. How well does this work?

---

4 Roslaunch

Create a new workspace called `lab4` in your `ros_workspaces` directory. Refer to lab 1 if you need a refresher on how to do this. Our starter code for this lab is on GitHub for you to clone so that you can easily access any updates we make to the starter code (Link). You can clone it by running

```
git clone https://github.com/ucb-ee106/106a-fa22-labs-starter/tree/main/Lab4
```

Move the files from the repo into the src folder inside your lab4 workspace. Inside the src folder, run the following command:

```
git clone https://github.com/machinekoder/ar_track_alvar.git -b noetic-devel
```

We also highly recommend you make a private GitHub repository for each of your labs just in case. You should build your workspace after including these packages using `catkin_make`.

These packages will help us retrieve images from a webcam and locate AR Tags in those images. In the last part of this lab, you will edit the provided python file `turtlebot_control.py`.

For the next parts of this lab, we will make use of ROS "Launch" files as a way of configuring and bringing up multiple nodes with a single command. In particular, you will use two launch files provided to you in order to publish images from a USB webcam to a ROS Topic (in this section), and in order to locate objects using AR Tags (in the next section).

Create a folder called `camera_info` and put the file `head_camera.yaml` into this newly created folder. Now make sure you're in the directory that contains `camera_info` and run the following command to move this folder into a ROS subdirectory so that the camera can be set up correctly.

```
mv camera_info ~/.ros
```

Don't forget to `source devel/setup.bash` and `catkin_make` in the root of your workspace after doing this. RUN THESE COMMANDS FREQUENTLY! If you get any errors during this lab that you are unsure how to debug, sourcing your bashrc file and then running those two commands should always be your first step.

Examine the file `run_cam.launch` in the `launch` directory of the package `lab4_cam`. This is an XML file that specifies several nodes for ROS to launch, with various parameters and topic renaming directions. This file can be used to create a node that reads data from a connected webcam and publishes it to a new topic called `/usb_cam/image_raw`. Ensure that your webcam is plugged into the monitor or computer, and run the launch file using the command:

```
roslaunch lab4_cam run_cam.launch
```

Verify that this node is publishing image information with `rostopic`. You can also inspect the messages being published using the `rostopic echo` command.

Pro Tip: when inspecting messages that contain large arrays (like images) in the terminal, it is helpful to know some additional arguments to modify the behavior of `rostopic echo`. In particular, the `-c` argument tells rostopic to flush the previous message from the buffer before printing a new one, and the `--noarr` argument tells rostopic to not print out arrays to the terminal, choosing instead to just display the data type and size of the array.

Inspect the `/usb_cam/image_raw` topic using the following command:

```
rostopic echo -c --noarr /usb_cam/image_raw
```

Inspect the `/usb_cam/camera_info` topic using the following command:

```
rostopic echo /usb_cam/camera_info
```

Each of these parameters is specific to the camera that you're using. They will differ slightly from the parameters being used at different work stations despite the fact that they're all the same model. To learn more about calibrating cameras, click on this link.

Next run an instance of the `image_view` node with the following command:

```
rosrun image_view image_view image:=/usb_cam/image_raw
```

You should now see a window with the video stream from the webcam on top of the monitor. This command shows an example of renaming the topic "`image`" (which is what `image_view` subscribes to by default) to "`/usb_cam/image_raw`" (which is what `usb_cam` publishes to). Use `rqt_graph` to verify that these nodes are connected via a topic.

5 TF

Before we can start talking about localizing objects in space, we need to briefly mention how we typically deal with various reference frames in ROS. tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

Tf maintains these frames in a data structure called the "TF Tree", which allows us to efficiently query the transforms between any pair of frames in our system that are being published. Lab 3 deals with how you can write your own TF Publisher that publishes information about frames to this TF Tree. In this lab, we will simply make use of the TF library to look up the transform between two frames of interest.

In particular, we will make use of tf to retrieve the transform between an AR Tag on the Turtlebot and an AR Tag on the ground. This will allow us to get the location of the robot in the reference frame of some origin on the ground, which will eventually allow us to write a controller to bring the robot to a desired location.

6 Localization with AR tags

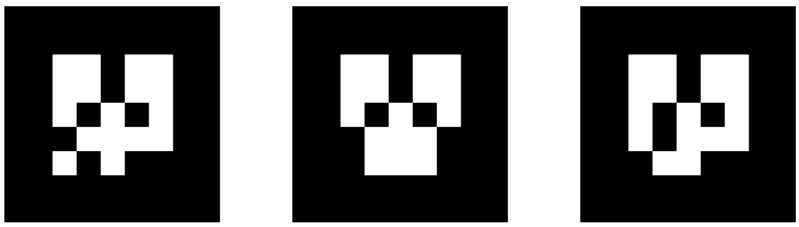

Figure 1: Example AR Tags

AR (Augmented Reality) Tags have been used to support augmented reality applications to track the 3D position of markers using camera images. An AR Tag is usually a square pattern printed on a flat surface, such as the patterns in Figure 1. The corners of these tags are easy to identify from a single camera perspective, so that the location of the tag can be inferred from the expected size of the tag and the skew detected in the image. The center of the tag also contains a unique pattern to identify multiple tags in an image.

When the camera is calibrated and the size of the markers is known, the pose of the tag can be computed in real-world distance units. The package then automatically publishes this frame to the TF Tree, from where we can query transforms

related to it just like any other frame. In this lab, we will place AR Tags on top of the turtlebots and on the ground to localize the turtlebot with respect to the floor.

There are several ROS packages that can produce pose information from AR tags in an image; we will be using the `ar_track_alvar` [1] package.

6.1 Webcam Tracking Setup

Your lab directory should already contain the `ar_track_alvar` package. To use this package to begin tracking AR Tags with the webcam, we need the following things:

1. A topic that will provide the camera images in which the tags should be detected. This will be the `/usb_cam/image_raw` topic that is published by the node we created in the last section.
2. Camera calibration information. Camera calibration is the process of computing parameters that specify how points in 3D space get projected onto the image frame. These parameters are crucial in computing the pose of the AR Tag. You will learn more about how camera information can be used to project points from 3D space onto the image frame in Lab 6. For now, your webcams have been calibrated for you, and these parameters are being published to a topic called `/usb_cam/cam_info` by the same node as above, in the form of a `CameraInfo` message.
3. The size of the AR Tag. (The tags used in this lab are $16.5cm \times 16.5cm$)

To begin tracking, we need to use a launch file that starts up a node to read image data and begin publishing poses for AR Tags visible in the image with respect to the reference frame of the camera. In this file, we specify parameters used by the AR Tracking node to compute poses for the tags. In particular, we specify parameters such as the image topic, the camera calibration information topic, and the size of the AR Markers. Inspect the provided launch file `ar_track.launch`. Can you tell how each parameter is being used?

6.2 Visualizing results

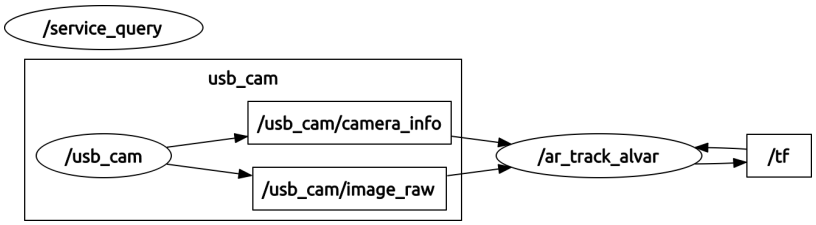

Figure 2: RQT Graph using AR Tags

Once the tracking package is installed, you can run tracking by launching `ar_track.launch`.

```
roslaunch lab4_cam ar_track.launch
```

Leave this node running. You should see topics `/visualization_marker` and `/ar_pose_marker` being published. They are only updated when a marker is visible, so you will need to have a marker in the field of view of the camera to get messages.

Running `rqt_graph` at this point should produce something similar to Figure 2. As this graph shows, the tracking node also updates the `/tf` topic to have the positions of observed markers published in the TF Tree.

To get a sense of how this is all working, you can use RViz to overlay the tracked positions of markers with camera imagery. With the camera and tracking node running, start RViz with:

```
rosrun rviz rviz
```

[1] http://wiki.ros.org/ar_track_alvar

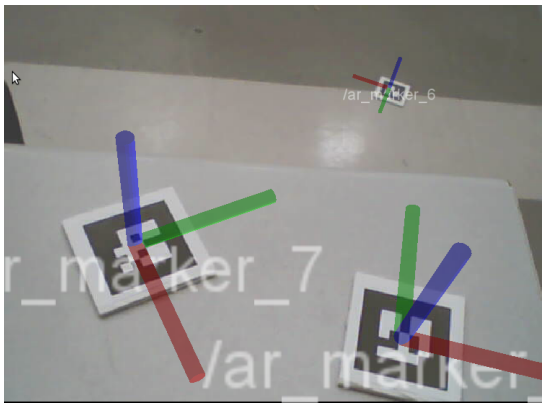

Figure 3: Tracking AR Tags with webcam

From the Displays panel in RViz, add an "Image" display. Set the Image Topic of the Image Display to the appropriate topic (`/usb_cam/image_raw` for the starter project), and set the Global Options Fixed Frame to `usb_cam`. (**Note:** you may need to place an AR tag in the field of view of the camera to cause the `usb_cam` frame to appear.) You should now see a docked window with the live feed of the webcam.

From the Displays panel in RViz, add a "Camera" display. Set the Image Topic of the Image Display to the appropriate topic (`/usb_cam/image_raw` for the starter project). Finally, add a TF display to RViz. At this point, you should be able to hold up an AR Tag to the camera and see coordinate axes superimposed on the image of the tag in the camera display. Figure 3 shows several of these axes on tags using the lab webcams. Making the marker scale smaller and disabling the Show Arrows option can make the display more readable. This information is also displayed in the 3D view of RViz, which will help you debug spatial relationships of markers for your project.

Alternatively, you can display the AR Tag positions in RViz by adding a Marker Display to RViz. This will draw colored boxes representing the AR Tags.

6.3 Localizing the Turtlebot

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 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.

Checkpoint 2

Submit a checkoff request for a staff member to come and check off your work. At this point you should be able to:

- Explain the contents of the `run_cam.launch` file.
- Drive the TurtleBot around with your keyboard.
- Visualize (in RViz) where your AR tag localization node thinks the TurtleBot is with respect to the tag on the ground.

---

7 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 the tools you've developed in the last few labs 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. 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 significantly 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? You will be editing `turtlebot_control.py` script to include a proportional controller, which will command the Turtlebot to drive to the target AR tag. 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

```
python3 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. If python3 does not work, try python instead.

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).

---

8 Appendix

If you are running into errors try the following: ONCE AGAIN, DOUBLE CHECK THAT YOU HAVE RUN THE FOLLOWING:

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

```
catkin_make
```

Change the /.bashrc file in your `.bashrc` file to

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

where `[COMPUTER_NUMBER]` is your computer number (1 through 10).