EECS C106A Remote Lab 4 - Introduction to Mobile Robots[*]

Fall 2020

Goals

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

- Launch the TurtleBot in Gazebo and drive it around with your keyboard
- Run the `gmapping` example to perform SLAM, then plan through the mapped space
- Simulate robots with laser scanners in complex environments in STDR simulator.
- Control a unicycle model robot to autonomously navigate to a target location.

If you get stuck at any point in the lab you may submit a help request during your lab section at `https://tinyurl.com/106alabs20`. You can check you position on the queue at `https://tinyurl.com/106Fall20LabQueue`.

Note: For all labs this semester you may collaborate with a lab partner but we expect everyone to do every part of the labs themselves. You should work closely with your partner to overcome obstacles in the labs but each member of the team must do the lab themselves.

Relevant Tutorials and Documentation:

- `turtlebot_gazebo`: `http://wiki.ros.org/turtlebot_gazebo`
- `ROS tf package`: `http://wiki.ros.org/tf`
- `stdr_simulator`: `http://wiki.ros.org/stdr_simulator?distro=kinetic`
- `gmapping`: `http://wiki.ros.org/gmapping`
- `map_server`: `http://wiki.ros.org/map_server`

Contents

[*]Developed for Fall 2020 (the year of the plague) by Amay Saxena and Tiffany Cappellari.
A predecessor of this lab was developed by David Fridovich-Keil and Laura Hallock, Fall 2017. Further extended for Fall 2018 by Valmik Prabhu, Ravi Pandya, Nandita Iyer, and Philipp Wu, and for Fall 2019 by Amay Saxena and Aakarsh Gupta.

1 Mobile robots

Mobile robots are a large class of robots that are designed to be able to move around and explore environments through suites of sensors. One of the most popular kinds of mobile robots are tank drive or ”unicycle model” robots, which are equipped with a drive base consisting of two independently actuated wheels placed along the same axis, allowing them to rotate in place (Unlike a car, or ”bicycle model”). These robots are extremely agile, thanks to their ease of control and small footprint. Such robots are generally equipped with wheel encoders that allow us to estimate their position and velocity, along with some sort of visual sensor like an RGB camera, a Depth camera, or a laser scanner.

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. In the remote version of this class, we will use a simulated Turtlebot2 in the Gazebo simulation environment. The simulated turtlebot will also come with a simulated Kinect sensor, which will let us use images, depth maps, and pointclouds of the environment.

Additionally, we will also use a lighter 2D unicycle robot simulator called STDR sim, which is useful when we want to quickly spawn many robots in a minimal environment, where we may not need the full 3D simulation machinery of Gazebo.

2 Intro to the Lab

In this lab, we will help you get a feel the sorts of things you can do with a visual sensor-equipped mobile robot. 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 later on; 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.

3 Getting started with Git

Our starter code for this lab is a ROS package called `lab4_starter`. It is on Git for you to clone and so that you can easily access any updates we make to the starter code. First, create a new ROS workspace called `lab4`.

```
mkdir -p ~/ros_workspaces/lab4/src
cd ~/ros_workspaces/lab4/src
catkin_init_workspace

cd ~/ros_workspaces/lab4
catkin_make
```

Next, clone the starter code package into the `src` directory of your workspace, and build it.

```
cd ~/ros_workspaces/lab4/src
git clone https://github.com/ucb-ee106/lab4_starter.git
cd ~/ros_workspaces/lab4
catkin_make
source devel/setup.bash
```

You won't need to think about the starter code until section (6) We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

4 Turtlebot simulation in Gazebo

First, we need to install the packages we will need for the Turtlebot Gazebo simulation and visualization in RViz.

```
sudo apt-get install ros-kinetic-gmapping ros-kinetic-turtlebot-gazebo
sudo apt-get install ros-kinetic-turtlebot-simulator
sudo apt-get install ros-kinetic-turtlebot-teleop
sudo apt-get install ros-kinetic-turtlebot-rviz-launchers
```

We are now ready to start a Turtelobot simulation in Gazebo. Before we can launch the simulation environment, however, we need to allow the installed packages to affect our environment variables. So, either open up a new terminal window, or execute the command `source ~/.bashrc` before proceeding.

We will use the launch file `turtlebot_world.launch` from the `turtlebot_gazebo` package. If we launch this file without any additional arguments, a Turtlebot will be launched in a default environment. However, we would like to use our own custom environment. In Gazebo, environments are specified using `.world` files. Each world is composed of one or more `models`. For this lab, we will use the provided `room.world` file in `lab4_starter/worlds`.

We will specify this world file by means of the argument `world_file`, as shown below. Usually, we would have to specify the complete absolute path to the world file, but we can use the `rospack` utility to make our lives easier. Run the following command to launch the environment. Note that the below command is all ONE LINE.

```
roslaunch turtlebot_gazebo turtlebot_world.launch
    world_file:=$(rospack find lab4_starter)/worlds/room.world
```

You should see a Gazebo window pop up with a single Turtlebot surrounded by various objects. Next, we will learn how to move the robot around.

4.1 Controlling the TurtleBot

TurtleBot commands are sent over the topic `/mobile_base/commands/velocity`. 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 run the following:

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

Try driving the TurtleBot around. Use `rostopic list` to see what topics are being published. Use

```
rostopic type /mobile_base/commands/velocity
```

to find out what message type the Turtlebot uses to accept inputs. You should find that the type of message is a `Twist`.

The topic `odom` (for "odometry") publishes estimates of the robot's position computed by reading the wheel encoders. Use `rostopic echo` to examine what happens to the odometry readings as you move the robot around.

4.2 Visualizing the sensors

The Turtelbot has on-board a Kinect camera, which is an incredibly powerful sensor, that gives you an RGB image, a depth map, and a pointcloud, and can hence be used for a large number of perception applications. In this lab we will only be using it for laser-scan based SLAM, but we encourage you to explore the functionality of all the topics published by the sensor, as you may choose to use them for your final project. For now, let us visualize some of these topics. Open up a new terminal, and run the following commands to open up RViz with a visualization of the robot.

```
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 Display Type > Image >. Then in the newly added Image display, change the "Image Topic" to `/camera/depth/image_raw` to add a visualization of the Kinect depth image. Examine the RViz display. What happens when you move the TurtleBot around? Try changing the topic of the image display to `/camera/rgb/image_raw` to see the RGB image that the robot sees instead.

5 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 environment. 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 processes except Gazebo** and run

```
roslaunch turtlebot_gazebo gmapping_demo.launch
```

Now 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. In order to get a good floorplan, you will need to slowly and methodically explore the environment. Try to drive slowly toward solid features like walls and solid furniture. Initially, you will also see that the robot keeps jumping around in RViz. This is because, as the name suggests, the SLAM algorithm is trying to simultaneously map the environment while also trying to localize itself within the environment. So every-time the algorithm updates its estimate of the robot's position, the robot jumps to that new position in RViz. If we do a good job of exploring the environment, then eventually our estimates of both the map and the robot's position will stabilize.

Try to drive around and build a decent map of the environment. What are some exploration strategies that you found fruitful?

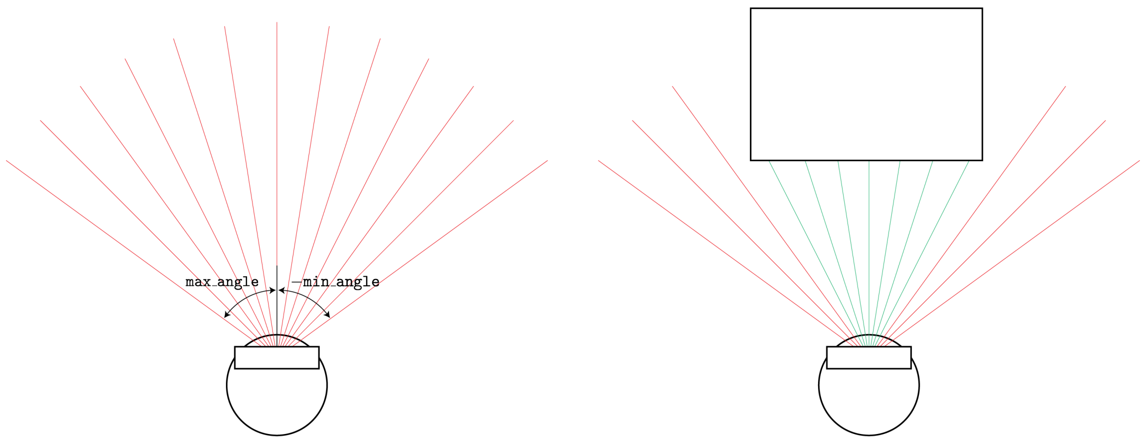

Figure 1: Top view of a mobile robot equipped with a laser scanner at the front of the robot. Rays are emitted from the sensor distributed along uniform angles to the heading direction, from `min_angle` to `max_angle`. Counter-clockwise is considered the positive direction. Here, only 13 rays are illustrated, but a real sensor would emit hundreds. When an obstacle is in range of the sensor, we measure the distance to the nearest intersection along each ray. Any rays that are not hitting an obstacle within the range `min_range` to `max_range`, then the output distance for that ray is simply NaN.

Checkpoint 1

Submit a checkoff request at `https://tinyurl.com/106alabs20` for a TA to check off your work. You should be able to address the following questions:

- How good is the floorplan generated by the gmapping algorithm?
- Do you find that sometimes the mapping algorithm gets irreparably confused? When does this happen?
- What were some good exploration strategies for building a map?

You should also be able to plan and execute paths through this map. How well does this work?

---

6 A More Minimal Simulator: STDR Sim

Note that mobile robots are generally confined to the ground, and as such can only really move in 2D space. They can be fully characterized with an (x, y) position and an orientation angle θ with respect to the x-axis. Often, then, the tasks we want mobile robots to perform, such as creating floorplans or exploring spaces, are best described as 2D problems. It is therefore often sufficient to simulate a mobile robot in a 2D environment, without having to worry about the complex computations a 3D simulation engine like Gazebo would perform. This can prove to be both efficient and highly effective when dealing with 2D tasks. Moreover, having such a lightweight simulation environment also makes it easy to test applications involving multi-robot collaboration, a popular area of study in mobile robotics, since many robots can be spawned cheaply.

In such a simulation, instead of dropping the robot into a 3D environment, we instead drop it into a 2D floorplan. The sensor output is assumed to be a *laser-scan*. A laser scanner is a sensor that records the distance to the nearest obstacle along rays emanating out from the sensor along a uniform horizontal plane, as shown in figure (1).

In this lab, we will use a simulator called STDR sim, which is a 2D multi-robot unicycle model simulator. STDR allows us to easily set up custom environments and robots with various sensors, as we shall see. In this lab, we will only explore STDR sim at a surface level. If you are interested in using STDR in your final project, you are encouraged to read the official documentation on your own to learn more about creating robots, configuring sensors, and creating custom environments.

First, clone the packages needed to run STDR simulator into your workspace and build them. This can take a few minutes.

```
cd ~/ros_workspaces/lab4/src
git clone https://github.com/stdr-simulator-ros-pkg/stdr_simulator.git
cd ~/ros_workspaces/lab4
catkin_make
```

We can now start a basic simulation. But first, remember that we need to make sure to set up our environment variables correctly so that ROS can find the new packages. Do this by running, as always,

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

6.1 Starting a new simulation

So far, we have been using `roslaunch` statements a fair bit, so it is worth talking about ROS ”Launch” files. Launch files are a way of configuring and bringing up multiple ROS nodes with a single command, in a modular fashion. As we shall see, launch files are a great way of reusing code from different packages to suit our own needs by simply changing the way the nodes are configured inside the launch file.

We have provided you with two launch files that will start-up STDR simulator with different configurations. Let's start up a maze like environment with a single robot equipped with a laser-scanner.

```
roslaunch lab4_starter stdr_maze_env.launch
```

This will bring up the STDR GUI with our configured environment. We can also bring up RViz to visualize this set up.

```
roslaunch stdr_launchers rviz.launch
```

6.2 Environment Configuration

STDR simulator uses YAML files to configure ”maps” and ”robots”. Let's examine the launch file we ran above to see how this works. Open up `stdr_maze_env.launch` in the text editor of your choice. You should see the following contents:

```
<launch>

    <include file="$(find stdr_robot)/launch/robot_manager.launch" />

    <node type="stdr_server_node"
        pkg="stdr_server"
        name="stdr_server"
        output="screen"
        args="$(find stdr_resources)/maps/sparse_obstacles.yaml"/>

    <node pkg="tf" type="static_transform_publisher"
        name="world2map"
        args="0 0 0 0 0 0  world map 100" />

    <include file="$(find stdr_gui)/launch/stdr_gui.launch"/>

    <node pkg="stdr_robot"
        type="robot_handler"
        name="$(anon robot_spawn)"
        args="add $(find lab4_starter)/robots/simple_robot.yaml 1 2 0" />

</launch>
```

This launch file spins up several nodes that together constitute the simulation environment.

1. The first `include` statement makes sure that the file `robot_manager.launch` gets launched whenever this launch file does. `robot_manager` is an in-built node in STDR simulator that handles spinning up and controlling robots.

2. The next `node` statement starts up a ROS node called `stdr_server` from the `stdr_server` package, which constitutes the simulation backend. This node also takes an argument specifying the map that should be loaded. Here, we are loading a map specified in the file `sparse_obstacles.yaml` located in the package `stdr_resources`. More on this later.

3. The next `node` is a static transform publisher. This is a quick way to spin up a node that constantly publishes a TF transform between two frames. This allows us to quickly add a new frame attached rigidly to some frame that TF already knows about, simply by letting one of the frames be the name of the new frame and the other frame being an existing frame. Another use is to connect two independent TF trees. Here, we are using this node to connect the frames `world` and `map` together using an identity transform. The six numbers `0 0 0 0 0 0` specify the desired static transform in `[x, y, z, yaw, pitch, roll]` notation.

4. Next, we `include` a launch file that starts up the STDR GUI. It is possible to run the simulation without starting a GUI, and this is often desirable in some advanced cases.

5. Finally, we create a node that spawns a robot in our map. Like maps, robot configurations are also specified in STDR sim through a `.yaml` file. In this case, we are using the file `simple_robot.yaml` that we included for you in the Lab 4 starter package. The arguments also specify the location at which the robot should be spawned, in `[x, y, theta]` notation, with units meters, meters, and radians respectively. Here, we use `[1, 2, 0]`.

Let's take a closer look at the map and robot specification.

6.2.1 Map configuration

As we stated above, STDR uses YAML files to specify map configurations. Let's take a look at the file `sparse_obstacles.yaml`. First, `roscd` into the right directory

```
roscd stdr_resources/maps
```

and look at the files in that directory. Of special interest to us, are the files `sparse_obstacles.yaml` and `sparse_obstacles.png`. Examine the contents of `sparse_obstacles.yaml`. You should see the following

```
image: sparse_obstacles.png
resolution: 0.02
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.6
free_thresh: 0.3
negate: 0
```

A map YAML file in STDR references a png image to get the actual desired layout of the map. Take a look at `sparse_obstacles.png`. You should see a black and white image that looks like the map we saw pop up in the simulator. Indeed, this is the file from which STDR reads the map data. This is a standard format specified by the package map_server, which is used by STDR to load maps. Much of the following description is drawn from the `map_server` documentation. The YAML file has the following fields:

1. image : Path to the image file containing the occupancy data; can be absolute, or relative to the location of the YAML file

2. resolution : Resolution of the map, meters / pixel

3. origin : The 2-D pose of the lower-left pixel in the map, as (x, y, yaw), with yaw as counterclockwise rotation (yaw=0 means no rotation). Many parts of the system currently ignore yaw.

4. occupied_thresh : Pixels with occupancy probability greater than this threshold are considered completely occupied.

5. free_thresh : Pixels with occupancy probability less than this threshold are considered completely free.
6. negate : Whether the white/black free/occupied semantics should be reversed (interpretation of thresholds is unaffected)

The PNG file specifying the map can be any arbitrary RGB image. Each pixel is interpreted as specifying its occupancy in its grayscale value (from 0 to 255). If the image is RGB, then the R, G, and B values are averaged to get the grayscale value. This grayscale value is then interpreted as follows

1. First we convert the integer grayscale value `x` to a floating point number `p` depending on the interpretation of the negate flag from the yaml.
   (a) If negate is false, p = (255 - x) / 255.0. This means that black (0) now has the highest value (1.0) and white (255) has the lowest (0.0).
   (b) If negate is true, p = x / 255.0. This is the non-standard interpretation of images, which is why it is called negate, even though the math indicates that x is not negated. Nomenclature is hard.
2. Compare `p` to `occupied_thresh` and `free_thresh` to decide on the cell's occupancy.
   (a) If `p > occupied_thresh`, output the value 100 to indicate the cell is occupied.
   (b) If `p < free_thresh`, output the value 0 to indicate the cell is free.

This information is then packaged into a ROS message of type `OccupancyGrid` and is loaded into the simulation.

6.2.2 Robot configuration

Next, let's examine the file `lab4_starter/robots/simple_robot.yaml`.

```
robot:
  robot_specifications:
    - footprint:
        footprint_specifications:
          radius: 0.2000
          points:
            []
    - initial_pose:
        x: 0
        y: 0
        theta: 0
    - laser:
        laser_specifications:
          max_angle: 1.0
          min_angle: -1.0
          max_range: 4.0
          min_range: 0.05
          num_rays: 667
          frequency: 10
          frame_id: laser_0
          pose:
            x: 0
            y: 0
            theta: 0
          noise:
            noise_specifications:
              noise_mean: 0.5
              noise_std: 0.05
```

This file configures a robot with the following properties. All distances are in meters and all angles in radians, unless specified otherwise.

1. `radius` 0.2, in meters.
2. `initial_pose`, the identity.
3. A `laser` scan sensor with the following specifications.
   (a) `max_angle` and `min_angle` specify the angular range of the emanated lasers. See figure (1).
   (b) `max_range` and `min_range` specify the furthest and nearest distances at which obstacles will be correctly detected.
   (c) `num_rays` is the number of rays that will be emanated from the sensor. The higher this number, the more granular the laser scan is.
   (d) `frequency` is the number of times per second that a laser scan sensor reading should be published.
   (e) `pose` is the pose of the sensor relative to the robot. Here, we just want a sensor looking straight ahead in the front of the robot, so this is just the identity.
   (f) `noise` simulated sensor noise.

To create your own robots and maps, you simply need to use YAML files like the ones above with your own specifications. Look at the package `stdr_resources` for more example maps and robots that you can adapt for your own applications.

6.3 Published topics

Now with this simulation up and running, use `rostopic list` to see what topics are being published by the simulation. Identify the topics on which:

1. The robot accepts input commands. Also verify that the message type of this topic is `Twist`.
2. The robot publishes laser scanner data.
3. The robot publishes odometry data.

Hint: `cmd_vel` is a popular name for topics on which unicycle model robots accept inputs.

6.4 Keyboard Teleop

Now, let's make our robot move. Unfortunately, STDR does not come with a built-in keyboard teleop executable. Does this mean we will have to write one from scratch ourselves? Thanks to the modularity of ROS, the answer is no. In fact, we can reuse the same code that we used to control the Turtlebot. Recall that the Turtlebot accepts commands through a ROS topic of message type `Twist`. The `keyboard_teleop.launch` launch file that we used to control the Turtlebot works by launching a node that reads keyboard strokes, converts them into an appropriate `Twist` message, and then publishes this message to the right topic to send inputs to the robots. Since our simulated robot also accepts commands the same way, all we need to do is redirect the output of the tele-op node to the right topic. We will do this by simply modifying the launch file. First, create a new package called `stdr_teleop` with basic dependencies where we will store our code.

```
cd ~/ros_workspaces/lab4/src
catkin_create_pkg stdr_teleop std_msgs rospy roscpp
```

Next, create a copy of the `keyboard_teleop.launch` file in a new `launch` subdirectory of our package. We will call the new file `stdr_keyboard.launch`.

```
cd ~/ros_workspaces/lab4/src/stdr_teleop
mkdir launch
cd launch
cp /opt/ros/kinetic/share/turtlebot_teleop/launch/keyboard_teleop.launch stdr_keyboard.launch
```

Now open the file `stdr_keyboard.launch`. This file launches a single node called `turtlebot_teleop_keyboard` from the package `turtlebot_teleop`. Of particular interest to us is the following line:

```
<remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel_mux/input/teleop"/>
```

This line redirects messages being published to the topic `turtlebot_teleop_keyboard/cmd_vel` (to which the node `turtlebot_teleop_keyboard` publishes by default) to the topic `cmd_vel_mux/input/teleop` instead (which is one of the topics from which the Turtlebot accepts inputs. We simply need to remap it to the correct topic.

Replace "`cmd_vel_mux/input/teleop`" in the above line with the topic from which your robot in STDR sim accepts inputs (this was one of the topics you identified in the previous section).

And that's it! You can now use this node to control your robot in STDR sim. Simply `catkin_make` and `source devel/setup.bash` from your workspace directory and then run

```
roslaunch stdr_teleop stdr_keyboard.launch
```

Drive the robot around. Use `rostopic echo` to see messages being published by your keyboard telelop node to the input topic. Also examine the odometry topic as you drive your robot around.

6.5 GMapping SLAM with STDR

As we saw above, a consequence of the modularity of ROS is that we can write algorithms in a robot agnostic way, and simply remap outputs and inputs to the correct ROS topics to have it work with our particular set-up. In section 5, we used the `gmapping` SLAM algorithm with the Turtlebot. Now, we will use it in STDR sim by remapping topics appropriately. Let's examine the following command.

```
rosrun gmapping slam_gmapping scan:=<scan_topic> _base_frame:="/robot0"  map:=/gmapping/map
```

Where you should replace `<scan_topic>` with the name of the topic you identified in section (6.3), without the angle brackets.

Here, we are launching a node called `slam_gmapping` from the `gmapping` package. Through command line arguments we have specified which topic we expect the laser-scan sensor messages to be published to. Additionally, we have specified that the base frame of the robot is called `/robot0`. Finally, we are specifying the topic to which we would like `gmapping` to publish the generated map - `/gmapping/map`.

In a new terminal, run the above command. Open up RViz and change the topic of the "Map" display to `/gmapping/map`. RViz will now start displaying the map that `gmapping` is generating in real time based on the sensor readings. Use the keyboard teleop node to move the robot around to cover different parts of the environment and examine how the map gets built.

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20 for a TA to check off your work. At this point, you should be able to:

- Correctly identify the ROS topics in section (6.3).
- Explain how maps and robots are specified in STDR simulator.
- Drive the robot around in STDR sim and show a map being built by the SLAM algorithm in RViz.
- Explain the contents of the `stdr_keyboard.launch` file.

---

7 Proportional Control

In this section we will write our own controller to command the robot to autonomously move to a specified location in its environment in STDR simulator. You will synthesize all 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.

7.1 Launch test environment

Use the provided `control_task_env.launch` file to set up the environment in which we will complete this task. This file will bring up a simple robot with no sensors in a 10m by 10m map. The target location, marked with an "X" on the map, will be in the center of the map, at location `[5, 5]`.

The file takes as arguments the starting position `[x, y, theta]` where the robot should be spawned, as shown.

```
roslaunch lab4_starter control_task_env.launch x:=1 y:=1 theta:=0
```

Once running, the pose of the robot will correspond to the TF frame `robot0` and the target position will constantly be published as a TF frame `target`. Your task will be to write a controller to autonomously drive the robot to this target location.

7.2 Writing a feedback 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 robot'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, unicycle model robots 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_d and y_d, as well as $\dot{x}$ should be determined in the *body frame* of the robot, rather than the spatial frame. Given this information, what sign should K_1 be? What about K_2?

The provided `lab4_starter` package comes with a script called `unicycle_control.py`. You will be editing the `controller` function in this script to include a proportional controller, which will command the robot to drive to the target location. You will also need to make this script an executable by running

```
chmod +x unicycle_control.py
```

in the directory where it is located.

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 lab4_starter unicycle_control.py robot_frame target_frame
```

where you should replace `robot_frame` with the name of the TF frame of your robot, and `target_frame` with the TF frame of the target location.

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/106alabs20 for a TA to check off your work. At this point, you should be able to:

- Command the robot to drive to the target location from 3 different starting locations around the map.
- Comment on the performance characteristics of this controller. What would happen to the generated input if the robot started very very far away from the target? What if it started very close to the target? Would this be desirable behavior in a real system?
- Describe how you would improve the performance of your controller (you don't need to implement these improvements).
