EECS C106A: Remote Lab 1 - Introduction to ROS and Gazebo *

Fall 2020

Goals

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

- Set up a new ROS environment, including creating a new workspace and creating a package with the appropriate dependencies specified
- Use the `catkin` tool to build the packages contained in a ROS workspace
- Run nodes using `rosrun`
- Use ROS's built-in tools to examine the topics and services used by a given node
- Run a simple Gazebo simulation

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.

A quick note: Most of this lab is borrowed from the official ROS tutorials at http://www.ros.org/wiki/ROS/Tutorials. We've tried to pick out the material you'll find most useful later in the semester, but feel free to explore the other tutorials too if you're interested in learning more.

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.

Contents

*Developed by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).

1 What is ROS?

The ROS website says:

> ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.
>
> The ROS runtime "graph" is a peer-to-peer network of processes that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

This isn't terribly enlightening to a new user, so we'll simplify a little bit. For the purposes of this class, we'll be concerned with two big pieces of ROS's functionality: the *computation graph* and the *file system*.

1.1 Computation graph

A typical robotic system has numerous sensing, actuation, and computing components. Consider a two-joint manipulator arm for a pick-and-place task. This system might have:

- Two motors, each connected to a revolute joint
- A motorized gripper on the end of the arm
- A stationary camera that observes the robot's workspace
- An infrared distance sensor next to the gripper on the manipulator arm

To pick up an object on a table, the robot might first use the camera to measure the position of the object, then command the arm to move toward the object's position. As the arm nears the object, the robot could use the IR distance sensor to detect when the object is properly positioned in the gripper, at which point it will command the gripper to close around the object. Given this sequence of tasks, how should we structure the robot's control software?

A useful abstraction for many robotic systems is to divide the control software into various low-level, independent control loops that each manage a single task on the robot, then couple these low level loops together with higher-level logic. In our example system above, we might divide the control software into:

- Two control loops (one for each joint) that, given a position or velocity command, control the power applied to the joint motor based on position sensor measurements at the joint
- Another control loop that receives commands to open or close the gripper, then switches the gripper motor on and off while controlling the power applied to it to avoid crushing objects

- A sensing loop that reads individual images from the camera at 30 Hz
- A sensing loop that reads the output of the IR distance sensor at 100 Hz
- A single high-level module that performs the supervisory control of the whole system

Given this structure for the robot's software, the control flow for the pick-and-place task could be the following: The high-level supervisor queries the camera sensing loop for a single image. It uses a vision algorithm to compute the location of the object to grasp, then computes the joint angles necessary to move the manipulator arm to this location and sends position commands to each of the joint control loops telling them to move to this position. As the arm nears the target, the supervisor queries the IR sensor loop for the distance to the object at a rate of 5 Hz and issues several more fine motion commands to the joint control loops to position the arm correctly. Finally, the supervisor signals the gripper control loop to close the gripper.

An important feature of this design is that the supervisor need not know the implementation details of any of the low-level control loops. It interacts with each only through simple control messages. This encapsulation of functionality within each individual control loop makes the system modular, and makes it easier to reuse the same code across many robotic platforms.

The ROS computation graph lets us build this style of software easily. In ROS, each individual control loop is a *node* within the computation graph. A node is simply an executable file that performs some task. Nodes exchange control messages, sensor readings, and other data by publishing or subscribing to *topics* or by sending requests to *services* offered by other nodes (these concepts will be discussed in detail later in the lab).

Nodes can be written in a variety of languages, including Python and C++, and ROS transparently handles the details of converting between different datatypes, exchanging messages between nodes, etc.

1.2 File system

As you might imagine, large software systems written using this model can become quite complex (nodes written in different languages, nodes that depend on third-party libraries and drivers, nodes that depend on other nodes, etc.). To help with this situation, ROS provides a system for organizing your ROS code into logical units and managing dependencies between these units. Specifically, ROS code is contained in *packages*. ROS provides a collection of tools to manage packages that will be discussed in more detail in the following sections.

2 Setting Up and Using Your Virtual Machine

Your virtual machine should be set up and ready to use after completing Lab 0. It should be done *before* beginning this lab. If you have not completed Lab 0 yet, you must go back and finish it first.

Please note that if you delete your container your files saved in the VM will also be deleted so we **highly recommend you use GitHub** or some other form of version control just in case. This will also make it easier for collaboration between you and your lab partner. Just don't forget to make your repo **private**!

3 Gradescope Submission

In this class we use an autograder on Gradescope in order to automatically grade everyone's checkoffs. Please create a `.txt` file containing **only your SID** and nothing else (the file should only be 8 or 10 bytes in size) and upload it to the Lab Checkoffs assignment on Gradescope. The autograder will automatically run and you should see a score of 0 points. You **must** do this in order to get credit for the labs. You may name your file whatever you would like as long as it is a `.txt` file.

Throughout the semester we will automatically be rerunning the autograder at the end of each lab module (more about the modules here) to update your lab score. Please be aware that it is your responsibility to check your score periodically and make sure there isn't a mistake. If you believe there is a mistake please email Tiffany as soon as possible so that it can be corrected.

4 Initial configuration

The virtual machines you're using already have ROS installed from Lab 0. Open the `.bashrc` file, located in your root directory (denoted " ~ "), in a text editor (If you don't have a preferred editor, we recommend either Sublime Text or vim). Then add the following line to the end of the file if it is not already there:

```
source /opt/ros/kinetic/setup.bash
```

Save and close the file when you're done editing, then execute the command "`source ~/.bashrc`" to update your environment with the new settings. This line tells Ubuntu to run a ROS-specific configuration script every time you open a new terminal window. This script sets several environment variables that tell the system where the ROS installation is located.

5 Navigating the ROS file system

The basic unit of software organization in ROS is the *package*. A package can contain executables, source code, libraries, and other resources. A `package.xml` file is included in the root directory of each package. The `package.xml` contains metadata about the package contents, and most importantly, about which other packages this package depends on. Let's examine a package within the Baxter robot SDK as an example.

5.1 File system tools

ROS provides a collection of tools to create, edit, and manage packages. One of the most useful is `rospack`, which returns information about a specific package. For example, you can run the command

```
rospack find baxter_examples
```

in order to find the filepath for where `baxter_examples` is located.

Note: To get info on the options and functionality of many ROS command line utilities, run the utility plus "`help`" (e.g., just run "`rospack help`").

5.2 Anatomy of a package

In a terminal window, try running

```
roscd baxter_examples
```

What happened? what do you think the command `roscd` does? What do you think `rosls` does? Try it out!

The `baxter_examples` package contains several example nodes which demonstrate the motion control features of Baxter. The folder contains several items:

- `\src` - source code for nodes
- `package.xml` - the package's configuration and dependencies
- `\launch` - launch files that start ROS and relevant packages all at once
- `\scripts` - another folder to store nodes

Other packages might contain some additional items:

- `\lib` - extra libraries used in the package
- `\msg` and `\srv` - message and service definitions which define the protocols nodes use to exchange data

If you open the `package.xml` it should look something like this:

```
<?xml version="1.0"?>
<package>
  <name>baxter_examples</name>
  <version>1.2.0</version>
  <description>
    Example programs for Baxter SDK usage.
  </description>
```

```
  <maintainer email="rsdk.support@rethinkrobotics.com">
    Rethink Robotics Inc.
  </maintainer>
  <license>BSD</license>
  <url type="website">http://sdk.rethinkrobotics.com</url>
  <url type="repository">
    https://github.com/RethinkRobotics/baxter_examples
  </url>
  <url type="bugtracker">
    https://github.com/RethinkRobotics/baxter_examples/issues
  </url>
  <author>Rethink Robotics Inc.</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>rospy</build_depend>
  <build_depend>xacro</build_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>control_msgs</build_depend>
  <build_depend>trajectory_msgs</build_depend>
  <build_depend>cv_bridge</build_depend>
  <build_depend>dynamic_reconfigure</build_depend>
  <build_depend>baxter_core_msgs</build_depend>
  <build_depend>baxter_interface</build_depend>

  <run_depend>rospy</run_depend>
  <run_depend>xacro</run_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>sensor_msgs</run_depend>
  <run_depend>control_msgs</run_depend>
  <run_depend>trajectory_msgs</run_depend>
  <run_depend>cv_bridge</run_depend>
  <run_depend>dynamic_reconfigure</run_depend>
  <run_depend>baxter_core_msgs</run_depend>
  <run_depend>baxter_interface</run_depend>

</package>
```

Along with some metadata about the package, the `package.xml` specifies 11 packages on which `baxter_examples` depends. The packages with `<build_depend>` are the packages used during the build phase and the ones with `<run_depend>` are used during the run phase. The `rospy` dependency is important - `rospy` is the ROS library that Python nodes use to communicate with other nodes in the computation graph. The corresponding library for C++ nodes is `roscpp`. The `build_depend` tags indicate packages used during the build phase. The `run_depend` tags indicate packages used during runtime.

6 Creating ROS Workspaces and Packages

You're now ready to create your own ROS package. To do this, we also need to create a catkin workspace. Since all ROS code must be contained within a package in a workspace, this is something you'll do frequently. Don't forget to do all this *inside* of your VM.

6.1 Creating a workspace

A workspace is a collection of packages that are built together. ROS uses the `catkin` tool to build all code in a workspace, and do some bookkeeping to easily run code in packages. Each time you start a new project (i.e. lab or

final project) you will want to create and initialize a new catkin workspace.

For this lab, begin by creating a directory for all of your lab workspaces for the semester.

```
mkdir ros_workspaces
```

Then create a directory with

```
mkdir ros_workspaces/lab1
```

for Lab 1's workspace. The directory "`ros_workspaces`" will eventually conatin several lab-specific workspaces (named `lab1`, `lab2`, etc.)

Next, create a folder `src` in your new workspace directory (`lab1`). From inside the new `src` folder, run:

```
catkin_init_workspace
```

It should create a single file called `CMakeLists.txt`

After you fill `/src` with packages, you can build them by running "`catkin_make`" from the workspace directory (`lab1` in this case). Try running this command now, just to make sure the build system works. You should notice two new directories alongside `src`: `build` and `devel`. ROS uses these directories to store information related to building your packages (in `build`) as well as automatically generated files, like binary executables and header files (in `devel`).

Two other useful commands to know are `rmdir` to remove an empty directory and `rm -r` to remove a non-empty directory.

6.2 Creating a new package

Now you're now ready to create a package. From inside the `src` directory, run

```
catkin_create_pkg foo
```

Examine the contents of your newly created package, and open its `package.xml` file. By default, you will see that the only dependency created is for the catkin tool itself:

```
<buildtool_depend>catkin</buildtool_depend>
```

Next, we'll try the same command, but we'll specify a few dependencies for our new package. Return to the `src` directory and run the following command:

```
catkin_create_pkg bar rospy roscpp std_msgs geometry_msgs turtlesim gazebo_ros
```

Examine the `package.xml` file for the new package and verify that the dependencies have been added. You're now ready to add source code, message and service definitions, and other resources to your project.

6.3 Building a package

Now imagine you've added all your resources to the new package. The last step before you can use the package with ROS is to *build* it. This is accomplished with

```
catkin_make
```

You need to run it from the root of your workspace (`lab1` in this case).
`catkin_make` builds all the packages and their dependencies in the correct order. If everything worked, `catkin_make` should print a bunch of configuration and build information for your new packages "`foo`" and "`bar`", with no errors.

You should also notice that the `devel` directory contains a script called "`setup.bash`." "Sourcing" this script will prepare your ROS environment for using the packages contained in this workspace (among other functions, it adds "`~/ros_workspaces/lab1/src`" to the `$ROS_PACKAGE_PATH`). Run the commands

```
echo $ROS_PACKAGE_PATH
source devel/setup.bash
echo $ROS_PACKAGE_PATH
```

and note the difference between the output of the first and second `echo`.

Note: *Anytime that you want to use a non-built-in package, such as one that you have created, you will need to source the* `devel/setup.bash` *file for that package's workspace.*

To summarize what we've done, here's what your directory structure should look like:

```
ros_workspaces
  lab1
    build
    devel
      setup.bash
    src
      CMakeLists.txt
      foo
        CMakeLists.txt
        package.xml
      bar
        CMakeLists.txt
        package.xml
        include
        src
```

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20 for a TA to come and check your work. You should be able to:

- Show your TA that you have submitted a `.txt` file to the Lab Checkoffs assignment on Gradescope
- Explain the contents of your `~/ros_workspaces` directory
- Demonstrate the use of the `catkin_make` command
- Explain the contents of a `package.xml` file
- Use ROS's utility functions to get data about packages

---

7 Understanding ROS nodes

We're now ready to test out some actual software running on ROS. First, a quick review of some computation graph concepts:

- *Node*: an executable that uses ROS to communicate with other nodes
- *Message*: a ROS datatype used to exchange data between nodes
- *Topic*: nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages

Now let's test out some built-in examples of ROS nodes.

7.1 Running roscore

First, run the command

```
roscore
```

This starts a server that all other ROS nodes use to communicate. Leave `roscore` running and open a second terminal window (`Ctrl+Shift+T` or `Ctrl+Shift+N`).

As with packages, ROS provides a collection of tools we can use to get information about the nodes and topics that make up the current computation graph. Try running

```
rosnode list
```

This tells us that the only node currently running is `/rosout`, which listens for debugging and error messages published by other nodes and logs them to a file. We can get more information on the `/rosout` node by running

```
rosnode info /rosout
```

whose output shows that `/rosout` publishes the `/rosout_agg` topic, subscribes to the `/rosout` topic, and offers the `/set_logger_level` and `/get_loggers` services.

The `/rosout` node isn't very exciting. Let's look at some other built-in ROS nodes that have more interesting behavior.

7.2 Running turtlesim

To start additional nodes, we use the `rosrun` command. The syntax is

```
rosrun [package_name] [executable_name]
```

The ROS equivalent of a "hello world" program is turtlesim. To run turtlesim, we first want to start the `turtlesim_node` executable, which is located in the `turtlesim` package, so we open a new terminal window and run

```
rosrun turtlesim turtlesim_node
```

A turtlesim window should appear. Repeat the two `rosnode` commands from above and compare the results. You should see a new node called `/turtlesim` that publishes and subscribes to a number of additional topics.

8 Understanding ROS topics

Now we're ready to make our turtle do something. Leave the `roscore` and `turtlesim_node` windows open from the previous section. In a yet another new terminal window, use `rosrun` to start the `turtle_teleop_key` executable in the `turtlesim` package:

```
rosrun turtlesim turtle_teleop_key
```

You should now be able to drive your turtle around the screen with the arrow keys when in this terminal window.

8.1 Using rqt_graph

Let's take a closer look at what's going on here. We'll use a tool called `rqt_graph` to visulize the current computation graph. Open a new terminal window and run

```
rosrun rqt_graph rqt_graph
```

This should produce an illustration similar to Figure 1 (it might not look exactly the same but that's fine). In this example, the `teleop_turtle` node is capturing your keystrokes and publishing them as control messages on the `/turtle1/cmd_vel` topic. The `/turtlesim` node then subscibes to this same topic to receive the control messages.

Figure 1: Output of rqt_plot when running turtlesim.

8.2 Using rostopic

Let's take a closer look at the `/turtle1/cmd_vel` topic. We can use the `rostopic` tool. First, let's look at individual messages that `/teleop_turtle` is publishing to the topic. We will use "`rostopic echo`" to echo those messages. Open a new terminal window and run

```
rostopic echo /turtle1/cmd_vel
```

Now move the turtle with the arrow keys and observe the messages published on the topic. Return to your `rqt_graph` window, and click the refresh button (blue circle arrow icon in the top left corner). You should now see that a second node (the `rostopic` node) has subscribed to the `/turtle1/cmd_vel` topic, as shown in Figure 2.

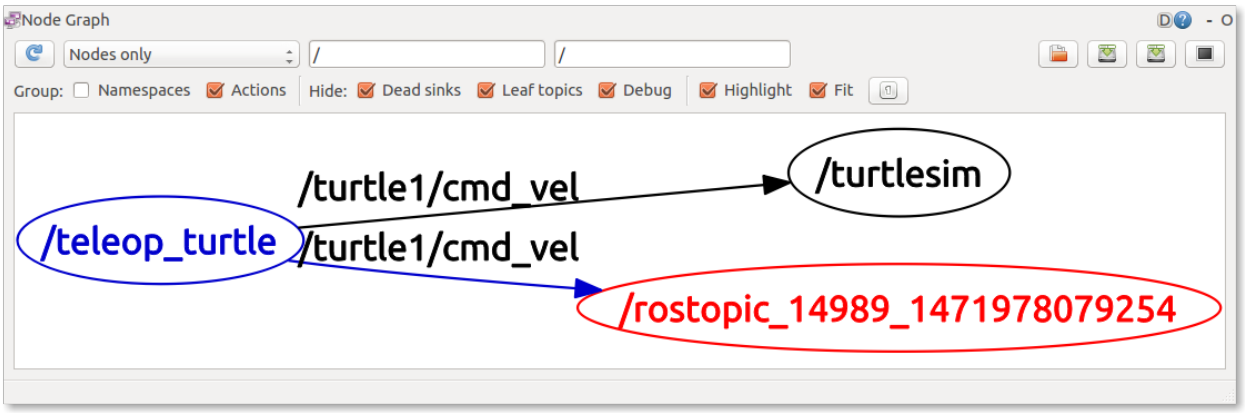

Figure 2: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

While `rqt_graph` only shows topics with at least one publisher and subscriber, we can view all the topics published or subscribed to by all nodes by running

```
rostopic list
```

For even more information, including the message type used for each topic, we can use the verbose option:

```
rostopic list -v
```

Keep the turtlesim running for use in the next section.

9 Understanding ROS services

Services are another method nodes can use to pass data between each other. While topics are typically used to exchange a continuous stream of data, a service allows one node to *request* data from another node, and receive a *response*. Requests and responses are to services as messages are to topics: that is, they are containers of relevant information for their associated service or topic.

9.1 Using rosservice

The `rosservice` tool is analogous to `rostopic`, but for services rather than topics. We can call

```
rosservice list
```

to show all the services offered by currently running nodes.

We can also see what type of data is included in a request/response for a service. Check the service type for the `/clear` service by running

```
rosservice type /clear
```

This tells us that the service is of type `std_srvs/Empty`, which means that the service does not require any data as part of its request, and does not return any data in its response.

9.2 Calling services

Let's try calling the the `/clear` service. While this would usually be done programmatically from inside a node, we can do it manually using the `rosservice call` command. The syntax is

```
rosservice call [service] [arguments]
```

Because the `/clear` service does not take any input data, we can call it without arguments

```
rosservice call /clear
```

If we look back at the `turtlesim` window, we see that our call has cleared the background.

We can also call services that require arguments. Use `rosservice type` to find the datatype for the `/spawn` service. The query should return `turtlesim/Spawn`, which tells us that the service is of type `Spawn`, and that this service type is defined in the `turtlesim` package. Use `rospack find turtlesim` to get the location of the `turtlesim` package (hint: you could also use "`roscd`" to navigate to the `turtlesim` package), then open the `Spawn.srv` service definition, located in the package's `/srv` subfolder. The file should look like

```
float32 x
float32 y
float32 theta
string name
---
string name
```

This definition tells us that the `/spawn` service takes four arguments in its request: three decimal numbers giving the position and orientation of the new turtle, and a single string specifying the new turtle's name. The second portion of the definition tells us that the service returns one data item: a string with the new name we specified in the request.

Now let's call the `/spawn` service to create a new turtle, specifying the values for each of the four arguments, in order:

```
rosservice call /spawn 2.0 2.0 1.2 "newturtle"
```

The service call returns the name of the newly created turtle, and you should see the second turtle appear in the `turtlesim` window.

10 A Quick Introduction to Gazebo

In many of the upcoming labs, we will be using the Gazebo simulation environment. To start Gazebo, make sure `roscore` is running and run

```
rosrun gazebo_ros gazebo
```

The Gazebo GUI should then pop up. To verify it is properly publishing to ROS topics, open another terminal window and run

```
rostopic list
```

You should see a list like this:

```
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
```

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. You should be able to:

- Explain what a *node*, *topic*, and *message* are
- Drive your turtle around the screen using arrow keys
- Use ROS's utility functions to view data on topics and messages
- Show your Gazebo GUI and explain what you think the topics being published are for.

EECS C106A: Remote Lab 2 - Writing Publisher/Subscriber Nodes in ROS*

Fall 2020

Goals

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

- Write ROS nodes in Python that both publish and subscribe to topics
- Define custom ROS message types to exchange data between nodes
- Create and build a new package with dependencies, source code, and message definitions
- Write a new node that interfaces with existing ROS code
- Use the powerful functionality of `tf2` in your own ROS node.

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 the queue at https://tinyurl.com/106Fall20LabQueue.

A quick note: Most of Labs 1 and 2 is borrowed from the official ROS tutorials at http://www.ros.org/wiki/ROS/Tutorials. We've tried to pick out the material you'll find most useful later in the semester, but feel free to explore the other tutorials too if you're interested in learning more.

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.

Contents

*Developed by Aaron Bestick and Austin Buchan, Fall 2014. Converted to remote by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).

1 Introduction

In Lab 1, you were introduced to the concept of the ROS computation graph. The graph is populated with *nodes*, which are executables that perform some internal data processing and communicate with other nodes by *publishing* and *subscribing* to *topics*, or by calling *services* offered by other nodes.

In this lab, you will explore how to write nodes that publish and subscribe to topics. ROS provides library code that takes care of most of the details of transmitting data via topics and services, which makes writing your own nodes quick and easy. You'll also learn a bit more about `tf2`, a useful package for computing transforms.

2 Examine a publisher/subscriber pair

Often the quickest way to learn new programming concepts is to look at working example code, so let's take a look at a publisher/subscriber pair that's already been written for you.

Our starter code is on Git for you to clone and so that you can easily access any updates we make to the starter code. It can be found at https://github.com/ucb-ee106/lab2_starter.git. You can clone it by running

```
git clone https://github.com/ucb-ee106/lab2_starter.git
```

and then you are free to move the directory `lab2` into your `ros_workspaces` directory. We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

You will notice that you now have an unbuilt workspace named "`lab2`". After moving it to your `ros_workspaces` directory, build this workspace using "`catkin_make`". Recall that any packages you wish to run must be under one of the directories on the `ROS_PACKAGE_PATH`. Source the appropriate "`setup.bash`" file ("`source devel/setup.bash`") so that ROS will be able to locate the packages in the `lab2` workspace. Verify that ROS can find the newly unzipped package using "`rospack find chatter`".

Examine the files in the `/src` directory of the `chatter` package, `example_pub.py` and `example_sub.py`. Both are Python programs that run as nodes in the ROS graph. The `example_pub.py` program generates simple text messages and publishes them on the `/chatter_talk` topic, while the `example_sub.py` program subscribes to this same topic and prints the received messages to the terminal. In a new terminal window start the ROS master with the "`roscore`" command, then, in the original terminal, try executing "`rosrun chatter example_pub.py`", which should produce an error message. In order to run a Python script as an executable, the script needs to have the executable permission. To fix this, run the following command from the directory containing the example scripts:

```
chmod +x *.py
```

Now, try running the example publisher and subscriber in different terminal windows and examine their behavior.

Study each of the files to understand how they function. Both are heavily commented. What happens if you start multiple instances of the publisher or subscriber in different terminal windows?

3 Write a publisher/subscriber pair

3.1 What you'll be creating

Now you're ready to write your own publisher/subscriber pair using the example code as a template. Your new publisher and subscriber should do the following:

Publisher

1. Prompt the user to enter a line of text (you might find the Python function `raw_input()` helpful)

   ```
   Please enter a line of text and press <Enter>:
   ```

2. Generate a message containing the user's text and a timestamp of when the message was entered (you might find the function `rospy.get_time()` useful)

3. Publish the message on the `/user_messages` topic
4. Prompt the user for input repeatedly until the node is killed with `Ctrl+C`

Subscriber

1. Subscribe to the `/user_messages` topic and wait to receive messages
2. When a message is received, print it to the command line using the format

   ```
   Message: <message>, Sent at: <timestamp>, Received at: <timestamp>
   ```

   (Note that the final `<timestamp>` is NOT part of the sent message; your new message type should contain only a single message and timestamp. Where does it come from?)
3. Wait for more messages until the node is killed with `Ctrl+C`

3.2 Steps to follow

To do this, you'll need to complete the following steps:

1. Create a new package (let's call it `my_chatter`) with the appropriate dependencies. If you have difficulty, refer to Lab 1.
2. Define a new message type that can hold both the user input (a string), and the timestamp (a number), and save this in the `msg` folder of the new package (discussed below)
3. Place the Python code for your two new nodes in the `src` directory of the package (if you create the Python file from scratch, you will need to make the file executable by running "`chmod +x your_file.py`")
4. Build the new package
5. Run and test both nodes

It might be interesting to see if you can detect any discrepancy between when the messages are created in the publisher and when they are received by the subscriber; this is why we ask you to print both!

3.2.1 Defining a new message

The sample publisher/subscriber from the previous section uses the primitive message type `string`, found in the `std_msgs` package. However, we need a message type that can hold both a string and a numeric (`float`) value, so we'll have to define our own.

A ROS message definition is a simple text file of the form

```
<< data_type1 >>  << name_1 >>
<< data_type2 >>  << name_2 >>
<< data_type3 >>  << name_3 >>
...
```

(Don't include the << and >> in the message file.)

Each `data_type` is one of

- `int8, int16, int32, int64`
- `float32, float64`
- `string`
- other msg types specified as `package/MessageName`
- variable-length `array[]` and fixed-length `array[N]`

and each name identifies each of the data fields contained in the message.

Create a new message description file called `TimestampString.msg`, and add the data types and names for our new message type. Recall that you can create a new file by using `nano` or `subl` and providing the file name in the terminal. Save this file in the `/msg` subfolder of the `my_chatter` package. (You may need to create this directory.)

Now we need to tell `catkin_make` that we have a new message type that needs to be built. Do this by uncommenting (remove the `<!-- -->`) the following two lines in `my_chatter/package.xml`:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Next, update the following functions in `my_chatter/CMakeLists.txt` so they read exactly as follows, uncommenting and adding information as necessary:

```
find_package(catkin REQUIRED COMPONENTS rospy std_msgs message_generation)

add_message_files(FILES TimestampString.msg)

generate_messages(DEPENDENCIES std_msgs)

catkin_package(CATKIN_DEPENDS rospy std_msgs message_runtime)
```

At this point, you can build the new message type using `catkin_make`. You can verify that this worked by confirming the existence of the file `~/ros_workspaces/lab2/devel/lib/python2.7/dist-packages/my_chatter/msg/_TimestampString.py`. (This is the how ROS implements your high-level message descriptions as Python classes.) Inspect this file if you are curious, but *do not modify it*. You can confirm the contents of your new message type by running "`rosmsg show TimestampString`". Keep in mind that `TimestampString` is a class, and the input message must be instantiated as an object of this class.

To use the new message, you need to add a corresponding `import` statement to any Python programs that use it:

```
from my_chatter.msg import TimestampString
```

Do this for both the publisher and subscriber that you are writing.

Checkpoint 1

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

- Explain all the contents of your `lab2` workspace
- Discuss the new message type you created for the `user_messages` topic
- Demonstrate that your package builds successfully
- Demonstrate the functionality of your new nodes using `TimestampString`

---

4 Write a controller for turtlesim

Now let's write a new controller for the turtlesim node you used in the lab last week. This node will replace `turtle_teleop_key`. Since the `turtlesim` node is the subscriber in this example, you'll only need to write a single publisher node. Create a new package `lab2_turtlesim` (that depends on `turtlesim` and other appropriate packages) to hold your new `turtle_controller` node (you will need to create a new file for this node).

Your node should do the following:

- Accept a command line argument specifying the name of the turtle it should control (e.g., running

  ```
  rosrun lab2_turtlesim turtle_controller.py turtle1
  ```

  will start a controller node that controls turtle1). The Python package `sys` will help you get command line arguments.

- Publish velocity control messages on the appropriate topic (`rostopic list` could be useful) whenever the user presses certain keys on the keyboard, as in the original `turtle_teleop_key`. (It turns out that capturing individual keystrokes from the terminal is slightly complicated — it's a great bonus if you can figure it out, but feel free to use `raw_input()` instead). We recommend using WASD keys for simplicity but feel free to use the arrow keys instead.

  Refer back to Lab 1 if you need a refresher on how to launch turtlesim and how to use `turtle_teleop_key`.

When you think you have your node working, open a turtlesim window and spawn multiple turtles in it. Then see if you can open multiple instances of your new turtle controller node, each linked to a different turtle. What happens if you start multiple instances of the node all controlling the same turtle?

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20 for a TA to come and check off your work. You should be able to:

- Explain all the contents of your `lab2_turtlesim` package
- Show that your new package builds successfully
- Demonstrate the functionality of your new turtle controller node by controlling two turtles one at a time

---

5 tf_echo and tf Listeners

5.1 Using tf_echo

You can find more information about `tf` here: http://wiki.ros.org/tf2/Tutorials

`tf` is a powerful tool that can help us transform between frames. Let's try using it with a simulation of a Baxter robot.

To set up your environment, make a shortcut (symbolic link) to the Baxter environment script `~/rethink_ws/baxter.sh` using the command

```
ln -s ~/rethink_ws/baxter.sh ~/ros_workspaces/lab2/baxter.sh
```

from the root of your workspace. Now let's ssh into our robot

```
./baxter.sh sim
```

then run `source devel/setup.bash` so your new workspace is on the `$ROS_PACKAGE_PATH`.

Now we are going to run a launch file to begin a simulated environment. Launch files can be used to simultaneously start multiple ROS nodes as well as a roscore with a single run command so that you don't need to run roscore and your various nodes from several different terminal windows.

Now let's run our launch file.

```
roslaunch baxter_gazebo baxter_world.launch
```

`tf_echo` reports the transform between any two frames broadcast over ROS. You can use it from your terminal with the following syntax:

```
rosrun tf tf_echo [reference_frame] [target_frame]
```

Now let's try using `tf` with our new Baxter simulation. In a new terminal window try running

```
rosrun tf tf_echo base left_hand
```

You will then see the transformations from the robot's base (the reference frame) to its left hand (the target frame) displayed as the `tf_echo` listener receives the frames broadcast over ROS.

Now let's try running an example script and observe how the output of `tf_echo` changes.

```
rosrun baxter_examples joint_velocity_wobbler.py
```

You can also try moving the robot yourself by running

```
rosrun baxter_examples joint_position_keyboard.py
```

5.2 Writing a tf Listener

`tf` is more than just a command line utility. It's a powerful set of libraries that you can use to find transforms between different frames on your robot. You'll be writing a listener node using `tf2`, which is the newer, supported version of `tf`. The `tf2` package is ROS independent, so you need to import `tf2_ros`, which contain ROS bindings of the various `tf2` functionalities. You can import it in your code with the following line:

```
import tf2_ros
```

A `Buffer` is the core of `tf2` and stores a buffer of previous transforms. To create an instance of a `Buffer` use the following line:

```
tfBuffer = tf2_ros.Buffer()
```

A `TransformListener` subscribes to the `tf` topic and maintains the tf graph inside the `Buffer`. To create an instance of `TransformListener` use the following:

```
tfListener = tf2_ros.TransformListener(tfBuffer)
```

The function `tfBuffer.lookup_transform(...)` looks up the transform of the target frame in the source frame. The output is of type `geometry_msgs/TransformStamped` (documentation for this type can be found here).

```
trans = tfBuffer.lookup_transform(target_frame, source_frame, rospy.Time())
```

Here are some `tf` exceptions you might want to catch:

```
tf2_ros.LookupException
tf2_ros.ConnectivityException
tf2_ros.ExtrapolationException
```

To catch an exception in Python you can create a try/except block (you might know this format as a try/catch block in most other programming languages). You should consider making a try/except block when using functions such as `lookup_transform` since exceptions can occur often and will crash your program when encountered. With a try/except block, your node will be able to handle exceptions and will not shut down if one occurs. You can write one with the following format:

```
try:
        <code to execute>
except (<exception>, <exception>, . . .):
        <code to execute if an exception occurs>
```

Task 3: Write a tf listener node `tf_echo.py` that duplicates the functionality of the `tf_echo` command line utility. Like the `tf_echo` command, your node should also take in a target frame and a source frame as command line arguments (the Python library `sys` might be helpful to look at). Please also note that you shouldn't need to create a subscriber for your node (Why do you think this is?). Display your node's output in another window alongside the `tf` data discussed above and ensure that the outputs are the same. Note: you do not have to format your output the same way, but the position and orientation should be the same.

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/106alabs20 for a TA to come and check off your work. You should be able to:

- Demonstrate usage of `tf_echo` with Baxter
- Demonstrate that your `tf_echo` node and the built-in `tf_echo` produce the same output

---

EECS C106A: Remote Lab 3 - Forward Kinematics/Coordinate Transformations*

Fall 2020

Goals

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

- Compute the forward kinematics map for a robotic manipulator
- Compare your own forward kinematics implementation to the functionality provided by ROS
- Make Baxter/Sawyer move to simple joint position goals
- View the sensor and state data published by Baxter using RViz

---

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 your 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:

- `Baxter SDK`: https://github.com/RethinkRobotics/sdk-docs/wiki/API-Reference
- `Sawyer SDK`: http://sdk.rethinkrobotics.com/intera/API_Reference
- `Baxter Joint Position Control Examples` : https://github.com/RethinkRobotics/sdk-docs/wiki/Joint-Position-Example
- `Sawyer Joint Position Control Examples` : http://sdk.rethinkrobotics.com/intera/Joint_Position_Example

Contents

*Developed by Aaron Bestick, Austin Buchan, Fall 2014. Modified by Victor Shia and Jaime Fisac, Fall 2015; Dexter Scobee and Oladapo Afolabi, Fall 2016; David Fridovich-Keil and Laura Hallock, Fall 2017. Ravi Pandya, Nandita Iyer, Phillip Wu, and Valmik Prabhu, Fall 2018. Converted to remote by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).

Introduction

Coordinate transformations are one of the fundamental mathematical tools of robotics. One of the most common applications of coordinate transformations is the forward kinematics problem. Given a robotic manipulator, forward kinematics answers the following question: Given a specified angle for each joint in the manipulator, can we compute the orientation of a selected link of the manipulator relative to a fixed world coordinate frame or a frame attached to another point on the robot?

This lab will explore this question in two parts, which need not be done in order. In Part 1, you'll use the code you wrote as part of the prelab to write the forward kinematics map for one of Baxter's arms, then you'll compare your results against some of ROS's built-in tools. In Part 2, you'll explore Baxter/Sawyer's basic joint position control functions, and take a quick look at how ROS helps you manage the coordinate transformations associated with all of Baxter/Sawyer's moving parts.

1 Getting started with Git

Our starter code is on Git for you to clone and so that you can easily access any updates we make to the starter code. It can be found at https://github.com/ucb-ee106/lab3_starter.git. First, create a new ROS workspace called `lab3`.

```
mkdir -p ~/ros_workspaces/lab3/src
cd ~/ros_workspaces/lab3/src
catkin_init_workspace

cd ~/ros_workspaces/lab3
catkin_make
```

Next clone our starter code by running

```
git clone https://github.com/ucb-ee106/lab3_starter.git
```

wherever you would like and then you are free to move the files into your lab's workspace into the appropriate subdirectories. We highly recommend you make a **private** GitHub repository for each of your labs just in case.

2 Forward kinematics

As discussed in lecture, the forward kinematics problem involves finding the configuration of a specified link in a robotic manipulator relative to some other reference frame, given the angles of each of the joints in the manipulator. In this exercise, you'll write your own code to compute the forward kinematics map for one of the Baxter robot's arms.

2.1 Kinematic functions

You should have already completed the relevant kinematic functions in `kin_func_skeleton.py` as part of your pre-lab. Take a look at it to refresh your memory as you will need to use this file for this lab.

2.2 Writing the forward kinematics map

Writing the forward kinematics map for a serial chain manipulator involves the following steps:

1. Define a reference "zero" configuration for the manipulator at which we'll say $\theta = 0$, where $\theta = [\theta_1, \ldots, \theta_n]$ is the vector of joint angles for an n-degree-of-freedom manipulator
2. Choose where on the robot to attach the fixed base frame and the moving tool frame

3. Write the coordinate transformation from the base to the tool frame when the manipulator is in the zero configuration ($g_{st}(0)$)

4. Find the axis of rotation (ω_i) for each joint as well as a single point q_i on each axis of rotation (all in the base frame)

5. Write the twist ξ_i for each joint in the manipulator

6. Write the product of exponentials map for the complete manipulator

7. Multiply the map by the original base-to-tool coordinate transformation to get the new transformation between the base and tool frames ($g_{st}(\theta)$, now as a function of the joint angles)

Task 1: Using the code from the pre-lab and referring to the textbook if necessary, write a Python function that computes the coordinate transformation between the base and tool frames for the Baxter arm pictured below (steps 3-7 above). Your function should take an array of 7 joint angles as its only argument and return the 4x4 homogeneous transformation matrix $g_{st}(\theta)$. Refer to Figure 1 for the parameters of the Baxter arm. The only other parameter you should need is the rotation matrix

$$R = \begin{bmatrix} 0.0076 & -0.7040 & 0.7102 \\ 0.0001 & 0.7102 & 0.7040 \\ -1.0000 & -0.0053 & 0.0055 \end{bmatrix}$$

where

$$g_{st}(0) = \begin{bmatrix} R & q \\ 0 & 1 \end{bmatrix}$$

for the appropriate value of q.

Note: Copying the information into Python from the diagram below can take a while, so we have done it for you in `lab3_skeleton.py`.

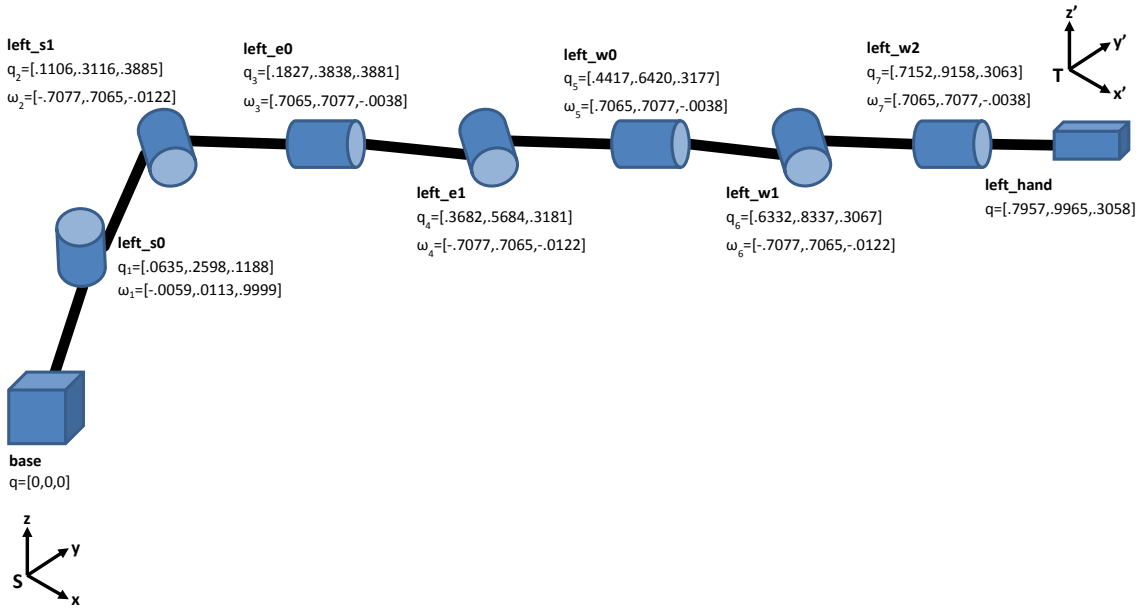

Figure 1: Baxter arm parameters.

2.3 Compare with built-in ROS functionality

Once you think you have your forward kinematics map finished, you'll compare with with some built-in functions offered by ROS.

To do this, we'll use a new tool called `rosbag`, which allows you to record and play back all the messages published on a set of topics, in order to test pieces of your software. I recorded a set of data from Baxter while I moved its left arm around. Find the `baxter.bag` file (from `lab3_starter`), start `roscore`, and play the file with

```
rosbag play baxter.bag
```

Notice how you can pause playback with the space bar and view the published messages with the usual tools like `rostopic list` and `rostopic echo`.

Try `rostopic echo`-ing the `robot/joint_states` topic, which gives the current joint angles of all joints in Baxter's left and right arms, as well as those of the head and torso. Using knowledge from `rostopic echo`, you can figure out what joint angles correspond to Baxter's left arm. (Hint: names starting with 'left_' correspond to the left arm.)

Next, try running the command

```
rosrun tf tf_echo base left_hand
```

while the bag file is playing. Any ideas about the data that's displayed?

Task 2: Write a subscriber node `forward_kinematics.py` that receives the messages from the `robot/joint_states` topic, plugs the appropriate joint angles from each message into your forward kinematics map from the last task, and displays the resulting transformation matrix on the terminal. Display this in another window alongside the `tf` data discussed above. Do you notice any differences? What do you think the "RPY" portion of the tf message is?

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain how you constructed your functions in your pre-lab
- Explain the functionality of your `forward_kinematics` node and demonstrate how it works
- Demonstrate that your `forward_kinematics` node and `tf` produce similar outputs (What's different? Are the values actually different? Could you translate one to the other?)

---

3 Make Baxter/Sawyer move

In this section, you'll explore some of Baxter/Sawyer's basic position control functionality. Close all running ROS nodes and terminals from the previous part, including the one running `roscore`, before you begin.

To set up your environment, make a shortcut (symbolic link) to the Baxter environment script `~/rethink_ws/baxter.sh` using the command

```
ln -s ~/rethink_ws/baxter.sh ~/ros_workspaces/lab3/baxter.sh
```

from the root of your workspace. Use one of the following lines to ssh into either the Baxter or Sawyer (intera) robot environment respectively:

```
./baxter.sh sim
./intera.sh sim
```

then run `source devel/setup.bash` so your new workspace is on the `$ROS_PACKAGE_PATH`.

Baxter and Sawyer have different interface packages (`baxter_interface` and `intera_interface`, respectively), but they are virtually identical. Don't forget to check that you have the correct package imported! The main difference is the obvious one: Sawyer only has one arm! This means that whenever you try to move an arm on Sawyer, it must be the **right** one. On Baxter, you may use either arm.

Before beginning to run anything on the robot, ssh into the environment and run

```
roslaunch baxter_gazebo baxter_world.launch
```

if you are using the Baxter model or

```
roslaunch sawyer_gazebo sawyer_world.launch
```

if you are using the Sawyer model. This will start a Gazebo window in which you should be able to see a model of your robot.

Then enable the robot by running

```
rosrun baxter_tools enable_robot.py -e
```

or

```
rosrun intera_interface enable_robot.py -e
```

in a new terminal window.

Now open up a new terminal and then open the `baxter_examples` package inside the `baxter_ws` workspace and examine the `scripts/joint_position_ keyboard.py` file, which allows you to move the Baxter's limbs using the keyboard. If you are using a Baxter, run the program and test its commands. If you are on a Sawyer, run the corresponding example in the `intera_sdk/intera_examples` package. Note that you don't need to start `roscore` — launch files already start a roscore for you!

Instead of publishing directly to a topic to control Baxter/Sawyer's arms (as with turtlesim), the respective SDKs provide a library of functions that take care of the publishing and subscribing for you.

Task 3: Create and open a new package called `joint_ctrl` in your `lab3` workspace. What dependencies will be needed? (Hint: Include `baxter_examples/intera_examples` as a dependency.) Make a copy of the `joint_position_keyboard.py` file (from the appropriate package, depending on which type of robot you are using) inside the `joint_ctrl` package, giving it a new name. Edit your copy so that instead of capturing keypresses, it prompts the user for a list of seven joint angles, then moves to the specified position. (Hint: You might have to call `limb.set_joint_positions()` repeatedly at some interval, say, 10ms, while the robot is in the process of moving to the new position.) The `set_joint_positions()` function takes a single argument, which should be a Python dictionary object mapping the names of each joint to the desired joint angles (e.g., `{'left_s0': 0.0, 'left_s1': 0.53, ..., 'left_w2':1.20}`). Dictionaries are used as follows:

```
# Create an empty dictionary
test_dict = {}

# Add values to the dictionary
test_dict['key1'] = 'value1'
test_dict['a_number'] = 1.024

# Read values from the dictionary
print(test_dict['key1'])
print(test_dict['a_number'])

# Output:
# value1
# 1.024

# You can also create a dictionary with a literal expression
test_dict2 = {'key1': 'value1', 'a_number': 1.024}
```

Test your code with several different combinations of joint angles and observe the results. Once you get your code to work, run the command

```
rosrun tf tf_echo base left_hand
```

or

```
rosrun tf tf_echo base right_hand
```

as appropriate and observe the output as you move the robot around. Any ideas what the data represents?

Finally, run

```
rosrun rviz rviz
```

Once RViz loads, ensure that `Displays > Global Options > Fixed Frame` is set to `world`. Next, click the Add button and add a RobotModel object to the window so you can see the robot move. Any thoughts as to where RViz gets the data on the robot's position?

Next, add two copies of the Axes object to the display. In the Displays pane of the left side of the screen, set the Reference Frame of one Axes object to `/base` and the other to `/right_hand`. You should see both sets of axes displayed on Baxter. What do you think the axes represent?

Finally, remove both Axes objects and add a single TF object to the display. What happens?

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Demonstrate the code you wrote to set Baxter/Sawyer's joint positions
- Use RViz to display the different state and sensor data topics published by Baxter/Sawyer
- Explain what the Axes and TF displays in RViz represent

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?

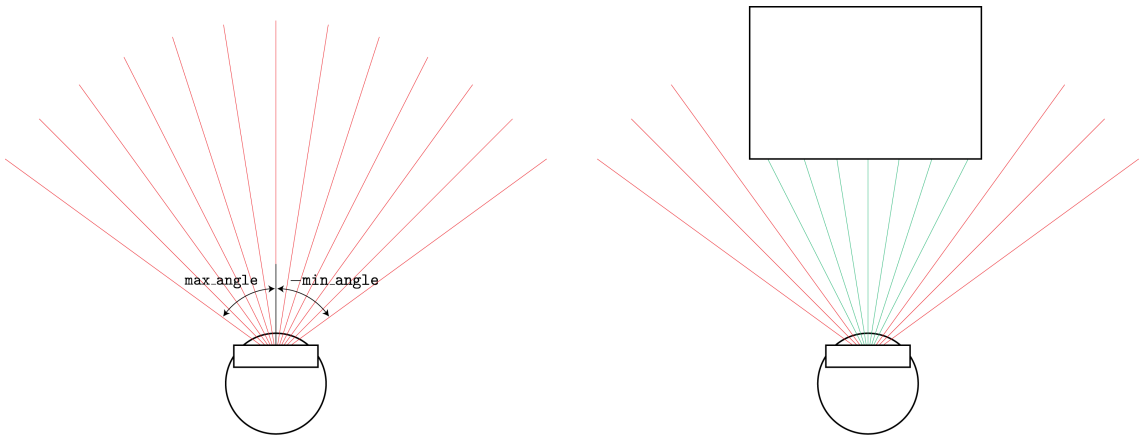

Figure 1: Top view of a mobile robot equipped with a laser scanner at the front of the robot. Rays are emitted from the sensor distributed along uniform angles to the heading direction, from `min_angle` to `max_angle`. Counter-clockwise is considered the positive direction. Here, only 13 rays are illustrated, 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).

EECS C106A: Remote Lab 5 - Inverse Kinematics and Path Planning *

Fall 2020

Goals

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

- Use MoveIt to compute inverse kinematics solutions and plan paths with the ROS action server for Baxter
- Create a visualization of Baxter's kinematic structure, as defined in the URDF file
- Use MoveIt to move Baxter's gripper(s) to a specified pose in the world frame and perform a rudimentary pick and place task.
- Plan and execute paths with obstacles and orientation constraints on Baxter.
- Understand the difference between open-loop and closed-loop control.
- Implement a trajectory-tracking controller on Baxter.

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.

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.

Contents

*Developed by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).
Based on previous labs developed by Aaron Bestick and Austin Buchan, Fall 2014. Modified by Laura Hallock and David Fridovich-Keil, Fall 2017. Modified by Valmik Prabhu, Nandita Iyer, Ravi Pandya, and Phillip Wu, Fall 2018.

Introduction

In Lab 3, you investigated the *forward kinematics* problem, in which the joint angles of a manipulator are specified and the coordinate transformations between frames attached to different links of the manipulator are computed. Often, we're interested in the *inverse* of this problem: Find the combination of joint angles that will position a link in the manipulator at a desired location in $SE(3)$.

It's easy to see situations in which the solution to this problem would be useful. Consider a pick-and-place task in which we'd like to pick up an object. We know the position of the object in the stationary world frame, but we need the joint angles that will move the gripper at the end of the manipulator arm to this position. The *inverse kinematics* problem answers this question.

The MoveIt package also includes a variety of powerful path planning functionality — in fact, to move between your specified end effector positions, MoveIt was using this functionality behind the scenes. In this lab, you'll get acquainted with path planning on a Baxter robot.

This lab has three parts. In Part 1, you'll learn how to use ROS's built in functionality to compute inverse kinematics solutions for a robot as well as what a URDF is. In Part 2, you'll use inverse kinematics to program Baxter to perform a simple manipulation task with constraints. Finally, in Part 3 you will learn about open loop and closed loop controllers.

1 Getting started with Git

Our starter code is on Git for you to clone and so that you can easily access any updates we make to the starter code. It can be found at https://github.com/ucb-ee106/lab5_starter.git. First, create a new ROS workspace called `lab5`.

```
mkdir -p ~/ros_workspaces/lab5/src
cd ~/ros_workspaces/lab5/src
catkin_init_workspace

cd ~/ros_workspaces/lab5
catkin_make
```

Next clone our starter code by running

```
git clone https://github.com/ucb-ee106/lab5_starter.git
```

wherever you would like and then you are free to move the files into your lab's workspace into the appropriate subdirectories. We highly recommend you make a **private** GitHub repository for each of your labs just in case.

2 Installing Baxter MoveIt Configuration

In this lab, we will use a package called MoveIt, which provides an inverse kinematics solver and other motion planning functionality. You should have already installed the bulk of MoveIt in Lab 0, however, we now also need to install another package called `baxter_moveit_config` that will allow us to use MoveIt to solve IK problems as well. First open up a terminal and run

```
git clone https://github.com/ros-planning/moveit_robots.git
```

Now, we only care about the Baxter configuration files so we can go ahead and delete the other robot packages.

```
rm -rf moveit_robots/atlas_moveit_config
rm -rf moveit_robots/atlas_v3_moveit_config
rm -rf moveit_robots/iri_wam_moveit_config
rm -rf moveit_robots/r2_moveit_generated
```

Note: You can actually do the above commands all in one line by just separating the different directories you want to remove with spaces. We split it up here for you since it would be too long to render in a single line in the PDF and, depending on your PDF viewer, make copying and pasting weird.

Finally, move the remaining files that we want into your `rethink_ws` and build and source your workspace

```
mv moveit_robots ~/rethink_ws/src
cd ~/rethink_ws
catkin_make
source devel/setup.bash
```

3 Inverse Kinematics

An inverse kinematics solver for a given manipulator takes the desired end effector configuration as input and returns a set of joint angles that will place the arm at this position. In this section, you'll learn how to use ROS's built-in inverse kinematics functionality.

3.1 Specify a robot with a URDF

While the `tf2` package is the de facto standard for computing coordinate transforms for forward kinematics computations in ROS, we have several options to choose from for inverse kinematics.

Both MoveIt and `tf2` are generic software packages that can work with almost any robot. This means we need some method by which to specify a kinematic model of a given robot. The standard ROS file type for kinematic descriptions of robots is the Universal Robot Description Format (URDF). The URDF file for Baxter is contained in the `baxter_description` package. (Note: The entire Baxter SDK, including the `baxter_description` package, should be in your `~/rethink_ws`. To find the URDF, run `roscd baxter_description`). Also, note that `baxter.urdf` is actually deprecated: modern versions of ROS use the xacro package to describe the same information more cleanly. We will not examine these xacro files in this lab, but we encourage you to look through them if you plan to write your own URDF files for your final project. Because `baxter.urdf` is deprecated, it no longer exists in the Baxter SDK packages you installed so we have provided it in `lab5_starter`.

Task 1: It's hard to visualize the actual robot by staring at an XML file, so ROS provides a tool that creates a more informative kinematic diagram. Navigate to the folder containing your copy of the Baxter URDF (that we provided with the starter code) and run

```
check_urdf baxter.urdf
urdf_to_graphiz baxter.urdf
```

Open the PDF file that this command creates. Some of the nodes in the diagram should look familiar from Lab 3, but some are new. What do you think the blue and black nodes represent? In addition to Baxter's limbs, what are some of the other nodes included in the URDF, and how might this information be useful if you want to use Baxter's sensing capabilities?

3.2 Compute inverse kinematics solutions

To use MoveIt, you must first start the `move_group` node, which offers a service that computes IK solutions. The file `move_group.launch`, in the provided starter code, loads the URDF description of Baxter onto the ROS parameter server, then starts the `move_group` node.

To set up your environment, make a shortcut (symbolic link) to the Baxter environment script `~/rethink_ws/baxter.sh` using the command

```
ln -s ~/rethink_ws/baxter.sh ~/ros_workspaces/lab5/baxter.sh
```

from the root of your Lab 5 workspace.

Connect to the Baxter simulation by running `./baxter.sh sim` as you did in Lab 3. and start up the world simulation by running

```
roslaunch baxter_gazebo baxter_world.launch
```

and then enable the robot by running

```
rosrun baxter_tools enable_robot.py -e
```

Next, echo the `tf` transform between the robot's base and end effector frames by running

```
rosrun tf tf_echo base [gripper]
```

where `[gripper]` is `left_gripper` or `right_gripper` since we are working with Baxter.

Now create a new package called `ik` with dependencies on `rospy`, `moveit_msgs`, and `geometry_msgs`. Move `move_group.launch` into the new package's `/launch` subdirectory (you may have to create this subdirectory) and save `baxter.urdf` in the package's `src` subdirectory. Then run the launch file with `roslaunch` to start the MoveIt node. MoveIt is now ready to compute IK solutions.

Task 2: Edit `service_query.py` to be a node that prompts the user to input (x, y, z) coordinates for an end effector configuration, then constructs a `GetPositionIK` request, submits the request to the `compute_ik` service offered by the `move_group` node, and prints the vector of joint angles returned to the console. A couple of tips:

1. The `GetPositionIK` service takes as input a message of type `PositionIKRequest`, which is complicated. The most important part is the `pose_stamped` field, which specifies the desired configuration of the end effector (see our `service_query.py` for an example on how to use it).
2. The orientation of the end effector is specified as a quaternion. Since we're not worried about rotation, you can set the four orientation parameters to any values such that their norm is equal to 1. For reference, a value of $(0.0, \pm 1.0, 0.0, 0.0)$ will have Baxter's grippers pointing straight down.
3. Baxter robots have both a left and right arm, and the gripper frame is called either `left_gripper` or `right_gripper` depending on which arm you use.

Task 3: Now lets test your IK solutions from MoveIt using your FK node from Lab 3. Copy over the necessary files you need to run your `forward_kinematics.py` node from that lab (including the node's file itself) and edit the node so that it now takes in your own inputs rather than subscribing to the robot. Run it with the input being the solution of your IK node for some chosen end effector position and verify that the numbers match.

Does the IK solver always give the same output when you specify the same end effector position? If not, why not? Any ideas why the solver sometimes fails to find a solution? Again, keep in mind which arm you are using on the Baxter. Your Lab 3 code was made specifically for the left arm.

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain what a URDF is
- Validate the output of the MoveIt IK service by solving IK for different poses, plugging these returned joint angles back into your Lab 3 forward kinematics function, and verifying that you get the original pose back
- Explain whether IK solutions are unique or not and why

---

4 Planning with Baxter

In this section, you'll use inverse kinematics to program Baxter to perform a simple manipulation task of picking up a small object and placing it elsewhere.

MoveIt's path planning functions are accessible via ROS topics and messages, and a convenient RViz GUI is provided as well. In this section, you'll learn how use MoveIt's planning features via both of these interfaces.

4.1 Using the MoveIt GUI

In your workspace create a package named `planning` inside the `src` directory of your `lab5` workspace. It should depend on `rospy`, `roscpp`, `std_msgs`, `moveit_msgs`, `geometry_msgs`, `tf2_ros`, `baxter_tools`, and `intera_interface`.

Create a folder in the new package called `launch` and copy the `baxter_moveit_gui_noexec.launch` file from `lab5_starter` into the new folder. Now move the file `spawn_table_and_cube.py.py` into `planning/src`.

Also move the `models` folder into your `planning` directory as well (just leave it in the root of the package). Don't forget to build and source your workspace and make sure python files are executables!

4.1.1 Basic planning

Now ssh into the Baxter robot and use `roslaunch` to run `baxter_moveit_gui_noexec.launch`. The MoveIt GUI should appear with a model of the Baxter robot. In the Displays menu, look under "MotionPlanning" $\Longrightarrow$ "Planning Request." Under "Planning Request" can check the "Query Start State" and "Query Goal State" boxes to show the specified start and end states. You can now set the start and goal states for the robot's motion by dragging the handles attached to each end effector. When you've specified the desired states, switch to the "Planning" tab, and click "Plan." The planner will compute a motion plan, then display the plan as an animation in the window on the right. In the Displays menu, under "Motion Planning" $\Longrightarrow$ "Planned Path." If you select the "Show Trail" option, the complete path of the arm will be displayed. The "Loop Animation" option might also be useful for visualizing the robot's motion. Note that execution will not work (for now), because execution has been disabled in the launch file. Later in the lab, when we're working on the robot, you'll be able to execute paths through the GUI as well.

4.2 Using the action server interface

The MoveIt GUI provides a nice visualization of the solutions computed by the planner, but in a real world system, the start and end states would likely be generated by another ROS node, rather than by dragging the robot's arms in a graphical environment. The GUI is just a front end for MoveIt's actual planning functionality, which is accessible via ROS topics and messages. MoveIt's ROS interface allows you to define environments, plan trajectories, and execute those trajectories on a real robot, all by using the same `move_group` node you used in the previous section.

A complicated task like planning and executing a trajectory would be difficult to coordinate using simple topics and services. Therefore, the interface to the `move_group` node's planning functionality uses a third type of communication within ROS known as an *action server*. The ROS website provides the following description:

> In any large ROS-based system, there are cases when someone would like to send a request to a node to perform some task and also receive a reply to the request. This can currently be achieved via ROS services.
>
> In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The `actionlib` package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

As the description states, the `move_group` action server interface allows us to plan and execute trajectories, as well as monitor the progress of a trajectory's execution and even stop the trajectory before it completes.

An action server and client exchange three types of messages as a request progresses:

- **Goal:** Specifies the goal of the action. In our case, the goal includes the start and end states for a motion plan, as well as any constraints on the plan (like obstacles to avoid).
- **Result:** The final outcome of the action. For a motion plan, this is the trajectory returned by the path planner, as well as the actual trajectory measured from the robot's motion.

- **Feedback:** Data on the progress of the action so far. For us, this is the current position and velocity of the arm as it moves between and start and end states.

These messages are exchanged over several topics, as shown in Figure 1. You can use `rostopic echo` to view the topics as you would with normal ROS topics.

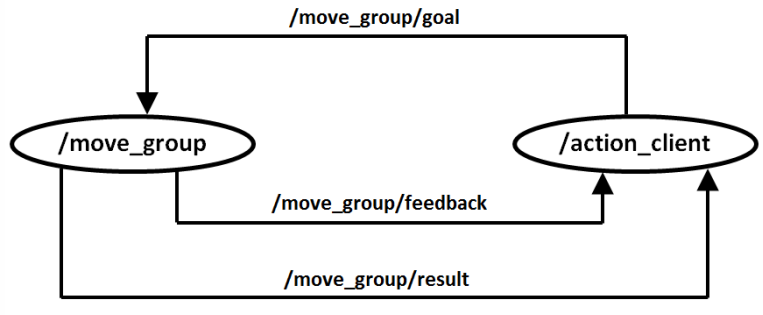

Figure 1: Action server topics

The data contained in these three message types is specified by a `.action` file. The file for the `move_group` action server is located in the `/action` subdirectory of the `moveit_msgs` package. Open this file and examine it.

4.2.1 Test a simple action client

From `lab5_starter`, move `action_client.py` into the appropriate location in your `planning` package. Run `baxter_moveit_gui_noexec.launch` using `roslaunch` to start the `move_group` node, followed by `action_client.py`. The second file should request a motion plan from the action server, then print the result to the terminal. You should also be able to see an animation of the plan in RViz.

Task 4: Now let's try this using the action server. Modify `action_client.py` so that you can input the start and goal states for the manipulator at the terminal, rather than having them hard coded into the program. Since there are a lot of angles, just make it so you can input **the first four joint angles of both the start and goal states**. Test this with several combinations of joint angles and comment on the results. Does the planner always return the same result given identical requests? If not, do you have any ideas why?

4.3 Path Planning

Complete the following steps to run MoveIt on Baxter. (Remember to ssh into the robot in each terminal window that will be interacting with the robot.)

1. ssh into the robot by running

```
./baxter.sh sim
```

2. Begin the Gazebo simulation by running

```
roslaunch baxter_gazebo baxter_world.launch
```

3. Enable the robot by running

```
rosrun baxter_tools enable_robot.py -e
```

4. Start the trajectory controller by running

```
rosrun baxter_interface joint_trajectory_action_server.py
```

5. Start MoveIt by running

```
roslaunch baxter_moveit_config demo_baxter.launch right_electric_gripper:=true
        left_electric_gripper:=true
```

Note: Because of how your PDF viewer may render this file, if you copy and paste the above line you may be missing a space between

`right_electric_gripper:=true`

and

`left_electric_gripper:=true`

in which case MoveIt will still launch but it will not work properly so be sure to check that your command is correct.

MoveIt is now ready to compute and execute trajectories on the robot. Please note that for the rest of this section you will need the `joint_trajectory_action_server` and the `demo_baxter.launch` files running.

Note: If things look like they are getting "stuck" in Gazebo, try closing everything and rerunning all the commands again.

4.3.1 Planning with obstacles and orientation constrains

Copy `path_test.py` and `path_planner.py` from `lab5_starter` to the `src` folder inside your `planning` package, and examine the two files. Rather than putting all the base MoveIt code into the `path_test.py` script, we have encapsulated it into the `PathPlanner` class inside `path_planner.py`. Make sure to understand both pieces of code before continuing.

Make sure that `path_test.py` is an executable. Run `path_test` using:

```
rosrun planning path_test.py
```

The robot should loop through three poses in series. As the arm moves, pay attention to the orientation of the right gripper. Does it remain at the same orientation throughout the motion?

While end effector orientation during a manipulator's motion is sometimes unimportant, in other cases it can be critical. Examples include moving liquid-filled containers and performing "peg-in-hole" insertion tasks. MoveIt allows you to plan paths with orientation constraints using the same interface as before.

As mentioned in the introduction, path planning algorithms can also solve the problem of planning with obstacles present in the environment around the robot. The `PathPlanner` class contains the `add_box_obstacle` function, which adds a box-shaped obstacle to the planning scene. While it's possible to add more complex shapes, including shapes from STL or OBJ files, it's rarely worth doing so, as more complex geometries will simply increase computation time.

Task 5: Edit `path_test.py` to add a box representing the table to your scene. For example, try making an obstacle at pose

```
X = 0.5, Y = 0.00, Z = 0.00
X = 0.00, Y = 0.00, Z = 0.00, W = 1.00
```

and for the size you might want to try

```
X = 0.40, Y = 1.20, Z = 0.10
```

. You should see a green box appear in the RViz window, at the position you create the object. Now try creating an artificial "wall" in the air near Baxter's arm.

4.4 Grippers

It's easy to operate Baxter's grippers programatically as part of your motion sequence. Move the file `gripper_test.py` into `planning/src` and (un)comment the appropriate `import` statement so that you are only importing the gripper class from either the `intera_interface` package or the `baxter_interface` package (here let's just use Baxter for now). Make sure the file is in the `src` directory of your `planning` package and try this by running

```
rosrun planning gripper_test.py
```

with the `baxter_world` simulation running which calibrates and then opens and closes Baxter's right gripper in the simulation.

4.5 Pick and Place

Now let's try and do a simple pick and place ourselves. Run this command with the `baxter_world` simulation up

```
rosrun planning spawn_table_and_cube.py
```

Note: This may take about a minute or so to load after the node finishes running. Once it completes, you should see a table and a cube in your Gazebo environment.

Now let's try seeing where the objects are in relation to the world frame. All the objects' transformations are published to the topic `gazebo/model_states`. Try viewing them by running

```
rostopic echo gazebo/model_states
```

Note: You may encounter an error that says "`WARNING: no messages received and simulated time is active. Is /clock being published?`". If this happens, try running "`rosparam set use_sim_time false`" and then try again.

This topic contains 3 arrays: a name array, a pose array, and a twist array. You can use the name array to find what index your desired object is located in and then use that index to find the object's pose and twist (you don't necessarily need to use this in your code for this lab but it is good information to have for a final project).

Now, based on the location of the block and the robot's arm and end effector, what positions do you think Baxter's arm gripper needs to go to in order to pick up the block, move it over, and then put the block back down on the table? Write down the poses. Please note though that the positions of the block and table are given with respect to the Gazebo coordinates which are different from coordinates with respect to the fixed frame base that you see in RViz. From some trial and error we believe the base frame in RViz's z-axis is shifted about 92 cm above the Gazebo world frame. The x and y-axes appear to be about the same.
Note: We recommend first running the command

```
rosrun baxter_tools tuck_arms.py -u
```

to move the robot's arms to a more neutral position above the table to make the planning a little easier. The easiest way may be to move along just one or two axes at a time. You can view the end effector's position in the Rviz window that pops up when you launch MoveIt.

Task 6: Make a copy of the `path_test.py` file in the `planning/src/` directory. Modify your file so that it moves the arm through the series of poses that you recorded earlier and attempt to perform a pick and place. Try adding a table obstacle constraint as well (for reference the dimensions of the table are 0.913, 0.913, 0.04). Do you need an orientation constraint? You should use code from `gripper_test.py` to open and close the grippers at an appropriate time. Can you make the arm return to the same position, repeatedly, with good enough accuracy to pick up an item?

We don't expect your pick and place task to work perfectly (especially this semester with Gazebo lag playing a part) so please just be able to show us that you can move the Baxter's arm to some series of poses whether or not it can actually pick and place the object.

After completing this task, you might have noticed that open loop manipulation is, in general, difficult. In particular, it's hard to estimate the position of the object accurately enough that the arm can position the gripper

around it reliably. Do you have any ideas about how we might use data from the additional sensors on Baxter to perform manipulation tasks more reliably?

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain what an action server is and why it is useful
- Move Baxter using an action server
- Be able to plan a path with orientation and obstacle constraints
- Perform a pick and place task. This doesn't have to work perfectly (or at all), but at least make an attempt. If pick and place doesn't work, what could you do to improve it?

---

5 Controlling Baxter

Now you'll be replacing MoveIt's default execution with a controller of your own. Copy `controller.py` from `lab5_starter` to the `src` folder inside your `planning` package, and examine the file. `controller.py` implements an open-loop, or feed-forward, velocity controller to follow a provided path (a `moveit_msgs/RobotTrajectory` message, which is returned by MoveIt's plan method). You'll be modifying this file to implement a PD (proportional derivative) and a PID (proportional integral derivative) controller on the robot.

The robot trajectory contains not only the desired position at each point, but also the desired velocity (as the number of path waypoints goes to infinity, this becomes the derivative of the desired position). An open loop velocity controller simply sets the manipulator's velocity as the desired velocity at the current point in the path.

If the distance between each waypoint in the path is small, and robot executes each command perfectly, we would expect to see the robot's actual trajectory exactly equal the desired trajectory. However, the world is rarely perfect, and the robot must deal with environmental disturbances, time delays, inaccurate sensors, and imperfect actuators. This is why engineers generally implement closed-loop, or feedback, control. Let's first analyze the performance of the feed-forward controller.

5.1 Feed-Forward (Open Loop) Control

Task 7: Edit the original `path_test.py` to use the controller instead of MoveIt's default execution, then test out the controller. Be sure to check the path in RViz before hitting Enter. You may want to disable your orientation constraints and remove the tables as well.

After each execution, the controller displays a plot of each joint value over time, with the target in green and the measured value in blue. How does the performance look in the plot? Use `tf` (either `tf_echo` or do it programmatically) to check the final end effector pose against the goal. How does the open-loop controller compare to the built-in controller?

5.2 Feedback (Closed Loop) Control

Now you'll be implementing closed loop control on these robots. PID (proportional integral derivative) control is ubiquitous in industry because it's both intuitive and broadly applicable. You'll be learning about PID control in class these next few lectures. The control law is:

$$u = u_{ff} + K_p e + K_i \int_0^t e\,dt + K_d \dot{e} \tag{1}$$

where e is the state error $q_d - q$ (where q_d is the desired position/joint angles and q the current position/joint angles) and u_{ff} is the feedforward term (desired velocity). The controller consists of a proportional term which pulls the error towards zero, a derivative term which dampens the controller and reduces oscillation, and an integral term which compensates for constant error sources (like gravity).

Task 8: First, edit `controller.py` to implement a *PD* controller (ignore the integral term) and execute it on the robot. You should be using the gains specified in `path_test.py` and should not need to change them. How do the plots change from when you used the open-loop controller? What could still be improved?

Finally, add the integral term to implement a PID controller and execute it on the robot. Once again, you should not need to change any of the gains. How do the plots change?

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point, you should be able to:

- Explain the whole of `controller.py`.
- Execute paths using the PID controller.
- Discuss the performance of the open loop, PD, and PID controllers against the default controller.
- What do you think K_w is for? Why is it needed?

EE106A: Lab 6 - Computer Vision *

Fall 2020

Goals

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

- Explain the concept behind pointclouds and what they represent
- Implement some basic techniques in image segmentation and explain your results
- Explain the concept of pinhole cameras and describe how they work
- Combine information from different sensors to isolate an object of interest. (Build a point cloud segmentation pipeline)

Relevant Tutorials and Documentation:

- OpenCV Python Tutorial

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.

Contents

*Converted to remote by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague). Developed by Amay Saxena and Grant Wang, Fall 2019.

1 Introduction

This lab will introduce you to various techniques used in processing data from advanced sensors like the Intel RealSense. In particular, you will learn about image segmentation, point cloud processing, and how we can combine various data modalities to leverage the strengths and weaknesses of each data form.

You will write your own *point-cloud segmentation pipeline*. This means that by the end of this lab, you will be able to isolate an object of interest in a point-cloud, by combining information you get from RGB image sensors.

2 Starter Code

Create a new workspace in your `~/ros_workspaces` directory called `lab6` and clone the starter code into the `src` subdirectory. After creating your workspace and copying over the contents of `lab6_starter` into `lab6/src`, use `catkin_make` to build your workspace.

```
git clone https://github.com/ucb-ee106/lab6_starter.git
```

After creating your workspace and copying over the contents of `lab6_starter` into `lab6/src`, use `catkin_make` to build your workspace. Then, go into `segmentation/src` and make all python files executable using

```
chmod +x *.py
```

The starter code includes the following packages:

1. `ros_numpy`: A very useful ROS package that can be used to convert between ROS message datatypes and numpy datatypes. Check out the documentation at http://wiki.ros.org/ros_numpy.
2. `segmentation`: This is the package that contains all the skeleton code. All files that you will need to edit are in this package.
3. `ddynamic_reconfigure`: Allows us to update parameters without restarting nodes. You will not need to interact with this package in this lab, but know that it is there.

3 Intel RealSense

In this lab, you will be using data collected from an Intel RealSense camera (see Figure 1), which is a powerful sensor equipped with an RGB camera, a depth camera, an Inertial Measurement Unit, and an IR sensor which is capable of producing a rich colored point-cloud. We will concern ourselves with the RGB camera and the point-cloud. You can get similar data from other RGBD sensors like the Kinect. You can also simulate many such sensors in Gazebo or other simulators, and we encourage you to look into this for your final projects so you can process data collected in real time. For this lab, we wanted you to be able to use real world data, so we will be providing you with data that we collected offline in the form of a ROS "bag" file.

Figure 1: Intel RealSense Camera

3.1 Bag files

Bag files are the canonical "ROS" way of recording data (recall using a bag file back in Lab 3 to "simulate" a Baxter robot). The `rosbag` utility allows us to record messages and TF data being published and then play it back later in the form of a "bag" file. In this lab, we have provided you with a bag file which you can download from here. Unfortunately, bag files are often quite large and we were unable to store it in the GitHub with the rest of the starter code. You should place this file in the `bagfiles` subdirectory of `lab6_starter`. This bag file was recorded while a RealSense camera was pointed at a green MegaBlocks block being moved around on a table. We will be using this bag file as a drop-in replacement for the sensor. Recall that when you implement algorithms using ROS, your nodes only care about what topics and TF frames are being published, so your code can be completely agnostic to if the publishing is being done by a specific sensor or by a datastream like `rosbag`.

Use the following command to play the bagfile in an infinite loop. First, you will have to start-up a master node using `roscore`. This node will be our stand-in for a real sensor.

```
rosbag play -l bagfiles/realsense.bag
```

Use `rostopic list` to look at the various topics being published.

4 Point-Clouds

A point-cloud is a kind of data type used to represent 3D scenes. A point-cloud is simply an unordered set of points. Point-clouds are a very popular 3D data modality. It is the kind of data returned by LiDAR sensors, which are staple as the primary kind of sensor used by self-driving cars.

Typically, these points are given by just their (x, y, z) co-ordinates in the camera's reference frame, but they may include additional dimensions for additional data that the sensor captures for each point, like color. Indeed, the pointclouds published by the RealSense will include 7 dimensions for each point, (x, y, z, r, g, b, a), where (x, y, z) are the co-ordinates of the point, (r, g, b) is the color of the point in RGB format, and a is the intensity registered by the sensor.

We can visualize the point-cloud being published in Rviz. First, open up Rviz by running

```
rosrun rviz rviz
```

Change the fixed frame to be `camera_depth_optical_frame`. Next, add a new Display of type `PointCloud2`. Set the topic for this display to `/camera/depth/color/points`. You may have to wait a bit for Rviz to begin registering pointcloud messages (as you can imagine, point-cloud messages tend to be pretty heavy). To get a better look at the pointcloud, you may have to check the box labelled `Invert Z Axis` in the right hand panel.

You should begin to see the point-cloud displayed in RViz in real time as a big cloud of points. A typical scan from the RealSense will include $\sim 100,000$ points. Note that the refresh rate on point-cloud topics tends to be much lower than that of Image topics (which can be visualized with minimal-to-no lag).

Now, also add two `Image` displays. Set the topic for one to `/camera/color/image_raw`. Set the topic for the other one to `/camera/depth/image_rect_raw`. These two displays will display the RGB and the Depth images respectively.

Our goal for this lab will be to take a pointcloud from the RealSense, and then isolate an object of interest in the scene. We will then publish a new pointcloud to a new topic, that will only contain points that correspond to the object of interest (a process called "segmentation").

Unfortunately, due to the unordered nature of point-clouds, it is usually very difficult to extract useful inferences from point-clouds alone, without first processing them into a different kind of data-structure (like a proximity graph, mesh, or voxel grid). This is because a regular point-cloud has no real structural arrangement to go off of - each point stands alone, and no order is guaranteed.

On the other hand, it is comparatively much easier to extract such features from an RGB image. The inherent structure of a pixel grid means that we have geometric cues to go off of when trying to locate an object of interest. Pixels corresponding to the same object are also close together in the pixel grid. This is also reflected in the state of the art of machine learning methods for object detection. Deep learning methods on images are leagues ahead of the state of the art in object detection directly on point-clouds, owing in large part to the unstructured nature of point-clouds.

4.1 An approach to point-cloud segmentation

Let's say we want to filter out all points from a pointcloud that do not correspond to some object of interest. We'll use a green MegaBloks block. We already claimed that it will be difficult to locate the block in the pointcloud directly. But as it turns out, we will be able to detect it easily the RGB image. So, how can we use this to our advantage to detect the block in the pointcloud?

Well, we can take advantage of the fact that both the RGB image and the pointcloud are scanning the same scene. So our strategy will be this: First, we will locate the block in the RGB image. Then, we will figure out how points in the point-cloud correspond to pixels in the image. After all, each point is just a 3D co-ordinate for some point in the scene that we took the RGB image of, so we should be able **to map each point to some pixel in the image.** Next, we simply keep those points which map to a pixel that was detected as belonging to the green block.

So then, the first step will be to detect the block in the RGB image. Our aim will be to assign to each pixel in the image either a 1 or a 0. A pixel gets a 1 if it belongs to the block, and a 0 otherwise. Such a grid is called a "segmentation" or "segmentation map" of the original image, and the process of producing assigning such a class label to each pixel in an image is called "image segmentation".

Next, we will map each point in the point-cloud to some pixel of the image. Then we will keep any point that lands on a pixel with a 1 in the segmentation map, and discard any point that lands on a 0. Finally, we simply create a new point-cloud that comprises of only the points that we kept, and publish it to some new topic for visualization.

5 Image Segmentation

Image segmentation will allow us to "segment" or partition out our specific object of interest in the image, which in this case will be our MegaBloks block. Our first step will be to create some functions that we can use for image segmentation and then apply it to our block to detect it. There are a number of different ways that image segmentation can be performed, but the ones that we will particularly look at are segmentation via thresholding, edge-detection, and clustering. These are all methods widely used in computer vision and robotics. Note though, since this is not a computer vision course, we will only touch on these methods at a surface-level and the results will be far from perfect. They will, however, be enough to accomplish our task and get you introduced into the rich field of computer vision.

5.1 Color and Grayscale Images

Before we start implementing our segmentation, we need to get you acquainted with some of the common models used to represent images. The two particular models we will focus on are RGB and grayscale. Let's begin by talking about RGB images. The idea behind RGB images is that with the 3 base colors red, blue, and green (hence RGB), we can mix them to create pretty much any color that most humans can distinctly recognize. Thus, we can represent an image as a 3 color channel system where each pixel contains 3-channels of red, green, and blue intensity values combined, and we can adjust these three values to change the particular color at a pixel. This representation allows

us to more formally define a color image as a 3-dimensional matrix, where each dimension is of size height x width of the original image and each element constitutes a pixel that can take on any intensity value between 0-255 (each color channel is 8 bits). What color do you think we get when we combine equal intensities of red, green, and blue?

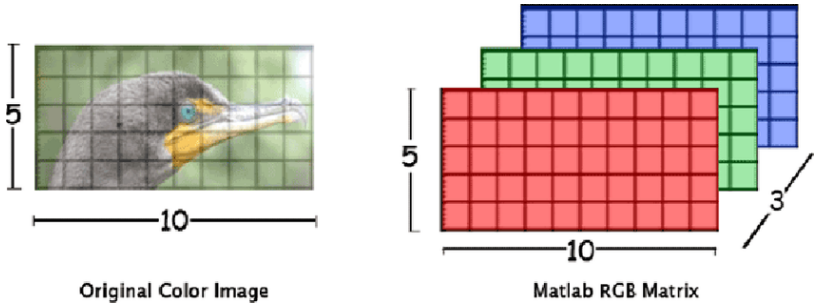

Figure 2: Matrix RGB representation of Color Image.

Sometimes, though, we may be only interested in a more compact representation - each pixel is just the particular amount of light. This is where grayscale images come in to play. Think of a grayscale image as a simpler version of an RGB image. Each pixel of a grayscale image represents just intensity (amount of light) and is composed entirely of shades of gray (mixtures of black and white). Black is of weakest intensity and white is of strongest. Using the grayscale model, any image is now in the form of a single channel 2-D matrix, where each element corresponds to a pixel and takes on an intensity value between 0 (pure black) and 255 (pure white). See Figure 3 for a visualization.

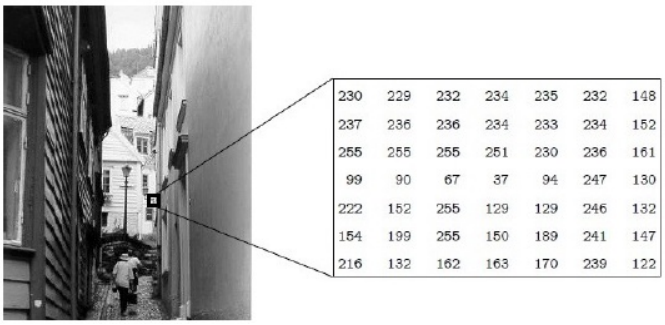

Figure 3: Matrix grayscale image.

A good amount of image processing and computer vision deals with extrapolating information from and manipulating the values in these image matrices. We will be exploring the properties of and manipulating grayscale and RGB images in image segmentation.

5.2 Thresholding

The simplest method of image segmentation is through the thresholding method. We will particularly explore grayscale thresholding, although this method can be generalized to color-based thresholding as well. The thresholding method makes use of the fact that if we have an object of interest and a background of a different grayscale intensity, we can "threshold" or clip the image pixel values by setting values above the threshold to be white and those below the threshold to be black (or you can do the other way around as well). More specifically in this lab, we will keep things simpler by working with objects on our table (which has a mostly uniform background) and we will define a threshold range that represents the intensity of the table and attempt to subtract it out. We will set a pixel to be low if in this range (is part of the background), else high (is foreground e.g. our object).

The result of this threholded image is a binary image (not grayscale) with 1's (white) representing our foreground or object of interest, and 0's (black) representing everything else.

Let's implement our thresholding strategy in code. Open up `image_segmentation.py` and implement the function `threshold_segment_naive`.

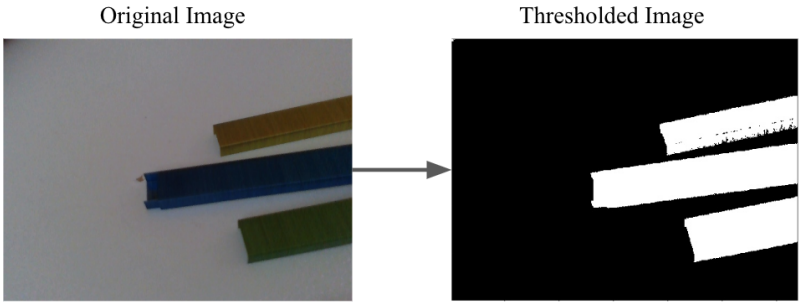

Figure 4: Thresholding an image.

We've included a couple sample images (`lego.jpg`, `legos.jpg`, and `staples.jpg`) for you to test your implementation. Go down to the `main` function and uncomment the line `test_thresh_naive` if it is not already. Pass in the image of interest into `read_img`.

Then run the script:

```
rosrun segmentation image_segmentation.py
```

You will need to tune the `lower_thresh` and `upper_thresh` values depending on the image.

5.3 Edge Detection

Thresholding is a good choice when we know that the background of the image and the objects of interest in the image have quite different grayscale intensities, allowing us to threshold easily and partition the two. However, such scenarios are not always the case, and if the background and object have quite similar grayscale properties, all bets are off. Luckily though, objects have a boundary that separate them from the background: the edges. If we can detect the edges of different objects, then we have knowledge of where the objects are in the scene.

An edge can be defined as any region in the image where there is a sharp change in intensity. A way of expressing this change in intensity is using derivatives or gradients (the multi-variable equivalent of the derivative). Imagine if we were to flatten out our grayscale image matrix into a 1-D signal of pixel intensities and a part of the image looked something like this when the pixel values are plotted:

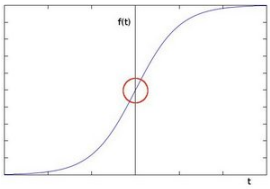

Figure 5: The point highlighted by the red circle represents a sharp change in intensity.

A sharp change in the first derivative reflects a sharp change in intensity of the image, or where there is an edge. So if we can compute the first derivatives of the pixels of our image in both the horizontal and vertical directions, we can combine them to get the gradient. And pixels where the gradient is high are where edges occur.

A way to compute the gradients at each pixel is by using Sobel Filters, which are widely used in edge detection. To use Sobel filters to detect edges, we perform what is known as a convolution between our filters represented by the matrices in Equation 2 and the image of interest I to output G_x and G_y, which represent the horizontal and vertical first derivatives of our image. We define K_x and K_y to be:

$$K_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \tag{1}$$

We define G_x and G_y to be:

$$G_x = K_x * I \qquad\qquad G_y = K_y * I \tag{2}$$

The result is a new image matrix G_x which represents the first derivative in the horizontal direction at each pixel of the original image I and G_y the vertical direction. The details of convolution are beyond the scope of this course, and all you have to know is that the 3x3 Sobel filter slides across each 3x3 region of interest in the image, a dot product is being computed between this filter (matrix) and the region of interest, and the outputted value represents the derivative at that pixel. For a good visualization of how convolution in imaging works, we refer you to this, and if you want to learn more about the math this is a start or take EE 120/123. (the link shows convolution of images for RGB images, since we're only working with grayscale, we only care about performing convolutions on a single dimension). One nuance is that before we perform convolution, we need to pad our original image with 0's on all sides, why do you think that's the case?

Finally, we approximate the gradient G at each pixel by combing G_x and G_y using the following equation:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3}$$

G will give us all the detected edges (locations where the gradient is high) of the original image. Let's now implement this in code. We will take advantage of some existing functions in OpenCV (a popular computer vision package). One subtle thing we will do in our implementation is to blur the image first using Gaussian blurring (see starter code for more details). This is done commonly to remove unwanted noise in the image. Fill in the function `edge_detect_naive` in `image_segmentation.py`. To test your implementation, uncomment the line `test_edge_naive` in `main` and try it on some sample images provided.

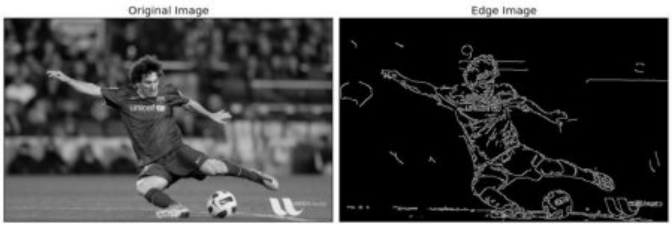

Figure 6: Edge detection using the Canny Edge Detector.

What we've just implemented is actually the first two steps of the famous Canny Edge Detector. To test how your edge detector does in comparison, uncomment out `test_edge_canny` and compare your images. What differences do you see?

5.4 Clustering

We've now looked at two methods of image segmentation that take an analytic approach. The last method of image segmentation we will explore takes a more empirical approach by treating an RGB image pixels as a set of data points with features that we are interested in using for classification. Note that we will be using RGB images instead in this part, rather than grayscale in the previous two methods. The features in this case are the Red, Green, and Blue intensity values at each pixel. Clustering is the task of partitioning the set of data points into different groups, or "clusters", such that data points in the same cluster are assigned the same label because they have more similar feature values to other data points in its cluster than those outside of the cluster. (i.e. pixel values with similar RGB intensities will be clustered with one another).

In the context of Image Segmentation, we can generate clusters for pixels of the image that share similar RGB intensities – they are part of the same thing in the image. Each cluster is a segmentation of the image that could represent the background, an object, etc. Figure 7 gives a high-level idea of how we are performing segmentation via clustering. Each data point, or pixel is plotted. Pay attention to the axes, which represent the features: RGB intensity values of each data point. Notice how pixels part of the same object in the image share the same cluster.

The algorithm we will use to generate these clusters is the K-Means algorithm, an unsupervised classification algorithm. We will not be going into the details of K-means. The steps of the algorithm, however, are actually pretty simple and we suggest you read through this step by step guide to get a feel of how the algorithm works. We will leverage the built in `kmeans` function in opencv. All you have to know is that we will feed as input into the kmeans the pixels of our image as the data points, and K-means will output a clustered image that has the same label for the

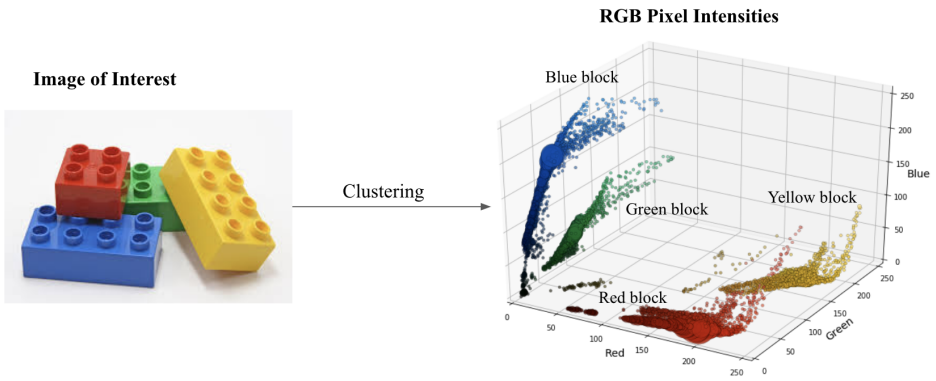

Figure 7: Clustering the pixels of our image via Red, Green, and Blue intensities as our features.

set of pixels in the same cluster but different from other clusters (this is our segmented image). We have provided the function `do_kmeans` that uses `opencv` to perform kmeans clustering for you.

Open up `image_segmentation.py` again and fill in the `cluster_segment` function. Comprehensive line by line notes have been provided in this part to get you started. Note, we first downsample the image before passing it into the K-Means algorithm to speed up clustering (why do you think that's the case?), and then when we want to return our segmented image we upsample the clustered image back to our original size. This has already been implemented for you, just make sure you're passing in the right variables to your function calls.

To test your implementation, go down to the `main` function and uncomment the line `test_cluster`. Play around with the parameter `n_clusters`. What should you set this value to be depending on the image?

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Demonstrate that your thresholding, edge-detection, and clustering implementations can segment all the images provided: `rgb_test.jpg`, `legos.jpeg`, and `lego.jpg`.
- Explain how your implementations worked for each method of image segmentation.
- Explain the performance differences and shortcomings of different segmentation methods on different images.

---

6 Projective Geometry and The Camera Matrix

Now that we have a segmentation map for the image where we can isolate which pixels belong to the green block, our next objective is to compute a correspondence between points in the point-cloud and pixels in the image. Essentially, we would like to figure out how a point in 3D space gets transformed by the camera lens and projected onto the image plane. Or more formally, we would like to compute the transform between 3D co-ordinates in the camera reference frame (X, Y, Z) to image plane co-ordinates (u, v).

6.1 Homogenous Co-ordinates

It is sometimes convenient to define points in terms of homogenous co-ordinates, wherein we append a 1 to the co-ordinate representation of points. This often allows us to get away with having only linear transformations (of the type $f(x) = Ax$) in a situation where we would otherwise need an affine transformation (of the type $f(x) = Ax + b$).

A point (X, Y) in $\mathbb{R}^2$ can be represented in homogenous coordinates by adding one additional dimension. The homogenous representation of this point would be (XT, YT, T) where T is an additional dummy variable. For any value of T, this will be a valid homogenous representation of the point (X, Y). In this way, we associate a point (X, Y) in $\mathbb{R}^2$ with a line in $\mathbb{R}^3$.

A point (X', Y', T) given in homogenous coordinates can be converted to the point (X, Y) that it represents by simply dividing through by the dummy coordinate:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} X'/T \\ Y'/T \end{bmatrix}$$

Feel free to think of this coordinate system as just a construction designed to make some of the math more convenient. We will comment on why this correspondence between points on a plane and lines in 3D arises in the problem at hand in a later section.

6.2 The Pinhole Camera

We will model our camera as a *Pinhole Camera*. The pinhole camera model defines the geometric relationship between a 3D point and its 2D corresponding projection onto the image plane. When using a pinhole camera model, this geometric mapping from 3D to 2D is called a *perspective projection*.

Let's denote the center of the perspective projection (the point in which all the rays intersect) as the optical center or camera center and the line perpendicular to the image plane passing through the optical center as the optical axis (see Figure 8). Additionally, the intersection point of the image plane with the optical axis is called the principal point. The pinhole camera that models a perspective projection of 3D points onto the image plane can be described as follows.

6.3 The Intrinsic Camera Matrix

We will model our camera as a standard pinhole model camera. Consider Figure 8.

Here, we take the co-ordinates of the point p in the camera reference frame to be (X, Y, Z), and the image plane co-ordinates of the projected point p' to be (u, v). Our final objective is find a pair of natural numbers (U, V) that are the index of the pixel onto which the point p is projected.

The standard way to define the axes of the camera reference frame is to have the z-axis point in the direction that the camera is looking in, and have the x and y axes aligned with the image plane. We can show that the equation relating the image plane coordinates to 3D space coordinates is given by:

$$\begin{bmatrix} u' \\ v' \\ w \end{bmatrix} = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4}$$

Where (u', v', w) is a *homogenous coordinate representation* for the image plane coordinates (u, v) of the point. To recover (u, v) we simply need to divide through by the dummy coordinate:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{w} \begin{bmatrix} u' \\ v' \end{bmatrix} \tag{5}$$

In the above two equation, we have the following:

- (X, Y, Z) is the 3D space coordinates of the point p in the camera's reference frame.
- (x_0, y_0) are the coordinates of the center of the image frame in a coordinate frame affixed to the top left of the image. (x_0, y_0) is given in pixels.
- f_x and f_y are *focal lengths* of the lens, in pixels. Since our camera produces a rectangular image (which wouldn't happen with a pure pinhole camera), we model it as having two focal lengths, one corresponding to the height of the image, and one corresponding to its width.
- s is *axis skew*. This parameter will be nonzero if the given lens produces a *shear* distortion in the image; in other words, it encodes situations where the pixels are parallelograms rather than rectangles or squares. This is not the case for our camera, so for us s will be zero.

The 3x3 matrix in Equation 4 is called the *Intrinsic Camera Matrix*. *Camera Calibration* is the process of estimating this matrix, along with a host of other parameters.

Finally, we can get the indices of the pixel onto which the point is projected as the floor of the image plane coordinates:

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} \lfloor u \rfloor \\ \lfloor v \rfloor \end{bmatrix} \tag{6}$$

The RealSense has already been calibrated for you, and its calibration data is continuously published to the topic `camera/color/camera_info`. The typical message type used by such topics is the `sensor_msgs/CameraInfo` message type.

Task 1: Look up documentation on the `sensor_msgs/CameraInfo` message type, and then implement the function `get_camera_matrix` in the file `main.py`. This function should accept a ROS Message of the above type, and should return the 3x3 camera intrinsic matrix as a numpy array. You will need to figure out what field in this message type gives you this matrix (*Hint 1:* Typically, this matrix is represented by the letter K) (*Hint 2:* `numpy.reshape` may be useful here).

6.4 Projecting the Point-cloud

Now, we are ready to project our pointcloud onto the frame of the RGB image. For a given point $\hat{p}$ in some world reference frame, we can compute the indices (U, V) such that $\hat{p}$ gets projected onto the pixel `image_array[V][U]` using the following steps:

1. Represent $\hat{p}$ in the reference frame of the camera. If the transform taking $\hat{p}$'s reference frame to that of the camera is (R, t), then find $p = R\hat{p} + t$. We will need to do this, since the pointcloud is generated in the reference frame of the depth camera, which has a slight offset from the RGB camera.
2. Find q' - the homogenous representation of the image plane coordinates of p - using eq (4):
$$q' = Kp$$
3. Convert the homogenous representation $q' = (u', v', w)$ into an ordinary point $p' = (u, v)$ on the plane using eq (5).
4. Finally, convert the coordinates (u, v) into pixel indices (U, V) by taking the floor, as in eq (6).

Task 2: Fill in the function `project_points` in file `pointcloud_segmentation.py`. This function takes as input a pointcloud of (x, y, z) points given as a 3xN numpy array, the camera intrinsic matrix, the (R, t) transform that converts points in the pointcloud to the reference frame of the RGB camera, and the dimensions of the image produced by the RGB camera. This function should return a new numpy array of integers, of size 2xN, where the ith pair is the pixel coordinates (U, V) of the ith point in the pointcloud.

Fill in the blanks in the code to implement steps 1-4 above.
Once you have an implementation, run the sanity test with

```
rosrun segmentation test_projection.py
```

You will see a reference image, and the result of projecting a corresponding reference pointcloud using your projection code. Make sure that the two frames match appropriately.

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Using the sanity test, show your TA that your code correctly projects the reference pointcloud.
- Explain what each entry in the camera intrinsic matrix represents.

---

7 Putting it all together

Now you have working implementations of both image segmentation and point-cloud projection. Next, we will start up a node that subscribes to topics for the RGB image, point-cloud, and camera info from the RealSense sensor, and will then publish a segmented point-cloud to a new topic. This node has been written for you, and you can look at its implementation in the file `main.py`. This node performs the following tasks, in order:

1. Get a tuple (I_t, P_t, K_t). I_t is the most recent RGB image at time t, P_t is the most recent point-cloud at time t, and K_t is the most recent camera intrinsic matrix at time t. We can get this by subscribing to three topics, one for each of those datapoints.

2. Use your image segmentation code to create a segmented image I'_t from I_t. This segmented image should have a 1 for any pixels that belong to your object of interest, and 0 elsewhere.

3. Use your pointcloud projection code to project every point of P_t onto pixels of the segmented image I'_t.

4. Create a new pointcloud P'_t by keeping any points from P_t that landed on a nonzero pixel, and discarding any point that landed on a zero pixel. P'_t should now only contain points belonging to the object of interest.

5. Publish P'_t to a new topic `segmented_points`.

The following are a few salient features of this node.

7.1 Time Synchronization

Notice that in step 1 above, we need to acquire three data points (I_t, P_t, K_t) from three different datastreams (topics). However, we also want to ensure that (I_t, P_t, K_t) come from approximately the same time. For instance, it may be the case that the topic publishing pointclouds has much greater lag than the other topics, or that one of the three topics is stalled for some reason. In this case, if we just naively use the last received message from each of the topics, then we will be projecting an old pointcloud onto a new image, which will give us an incorrect result. So we want some way to ensure that the pointcloud and image we use were collected close to each other.

The built-in ROS package `message_filters` has functionality that will give us exactly what we want. Instead of defining three naive subscribers, each with its own callback, we can instead define a single *message filter* that listens to three topics, and only lets through triplets of messages that satisfy the "approximate time" condition. The `ApproximateTimeSynchronizer` implements this functionality, and it allows us to define one single callback that takes three message arguments. The time synchronizer filter will make sure that only triplets of data with close timestamps get sent to this callback.

7.2 Queue Processing

One way to implement steps 1-5 is to perform all computations inside the subscriber callback, and then publish from within the callback itself. While this is certainly possible, it is not advisable. Instead, the callback simply pushes the triple (I_t, P_t, K_t) onto a queue, from where a separate routine pops, processes, and publishes.

Task 3: We will now put everything together and process both images and pointclouds from the sensor. Make sure the bag file from earlier is running.

In `image_segmentation.py`, fill in the function `segment_image` with an image segmentation algorithm of your choice. By default, it uses your thresholding implementation (you will need to put in the right thresholds). You can also experiment with using the clustering implementation instead.

Open up RViz. Set the fixed frame to `camera_depth_optical_frame`. In the right hand side window, check the box labelled `Invert z-axis`. Create an `Image` display and set its topic to `camera/color/image_raw`. Now create a `Pointcloud2` display, and set its topic to `segmented_points` (you may not able to do this until after you have started up `main.py`).

Now start the main node with the following command. You should see a point-cloud appear in RViz with only your MegaBloks block visible in it. How accurate this is will be a function of your image segmentation implementation, so feel free to play around with the hyperparameters until you get something that looks good. Also note that if you are using clustering, there may be noticeable lag in this pointcloud. This is to be expected, and happens because your image segmentation implementation is slow. You may be able to speed it up by, for instance, downsampling the image by a greater factor before segmentation and then upsampling the result.

```
rosrun segmentation main.py
```

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Show your segmented pointcloud in RViz to your TA.
- Explain the functioning of `main.py`.

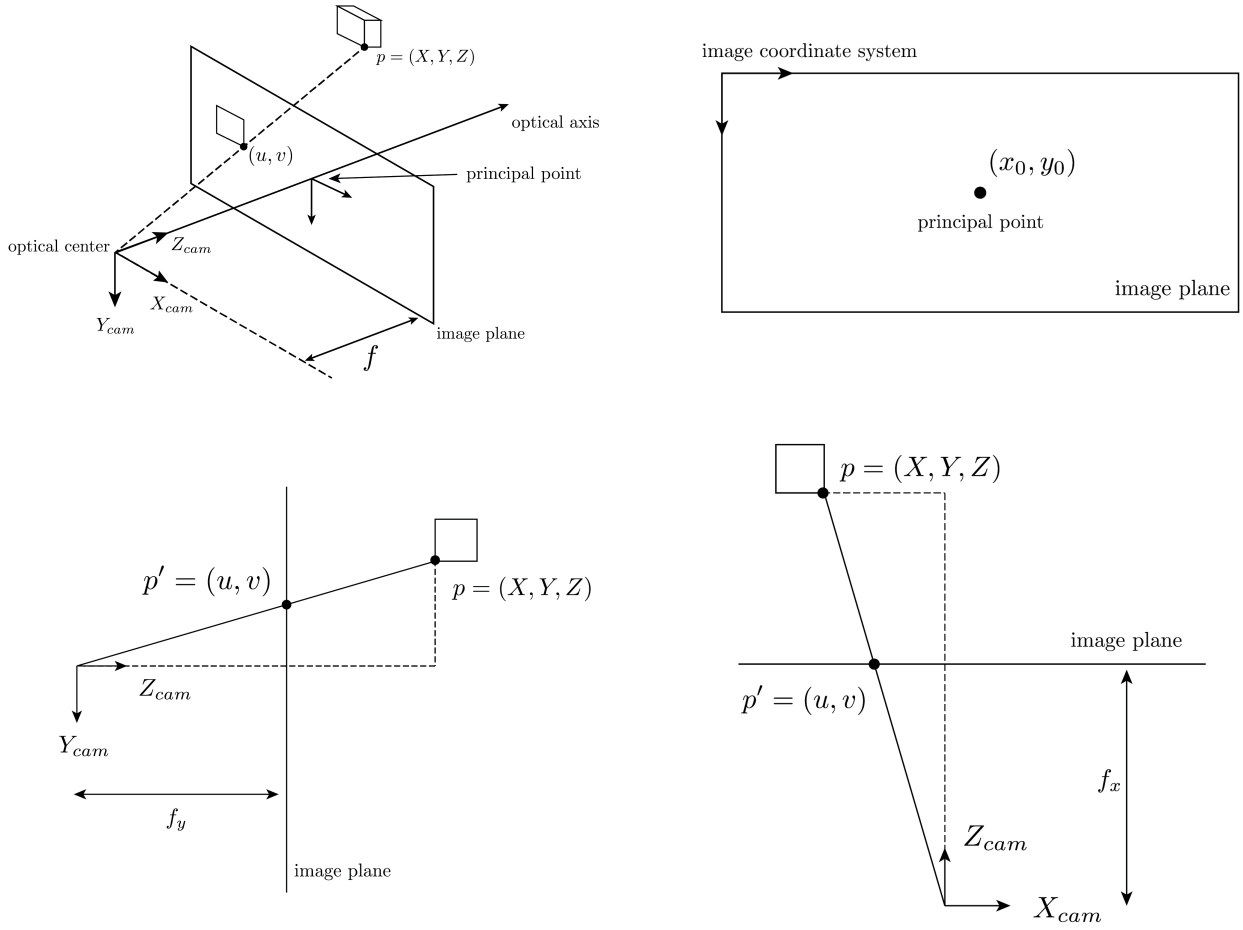

Figure 8: Geometry behind a Pinhole Camera

EE106A: Lab 7 - Multiview Geometry and Feature Tracking *

Fall 2020

Goals

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

- Use OpenCV to undistort imges from calibrated cameras and extract keypoint features from images.
- Match features between two images of the same scene to find corresponding points.
- Reject outlier matches using the epipolar constraint.
- Triangulate feature matches to find their location in 3D space.

Relevant Tutorials and Documentation:

- OpenCV Python Tutorial
- Feature Matching Python Tutorial

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.

Contents

*Developed for Fall 2020 by Ritika Shrivastava, Jay Monga, and Amay Saxena

1 Introduction

In this lab you will work with data collected from stereo RGB camera pair mounted on a drone. This data comes from the EuRoC MAV dataset. You will write a node that subscribes to two image streams coming from the left and right cameras of the stereo pair respectively. Your node will then extract relevant visual features from the images and triangulate them to find the 3D coordinates of the feature points. Your node will then publish the resulting pointcloud of visual features, which you will visualize in RViz.

2 Starter Code

Create a new workspace in your `~/ros_workspaces` directory called `lab7` and clone the starter code into the `src` subdirectory. After creating your workspace and copying over the contents of `lab7_starter` into `lab7/src`, use `catkin_make` to build your workspace.

```
git clone https://github.com/ucb-ee106/lab7_starter.git
```

After creating and building your workspace, navigate to the `ros_numpy` directory and run

```
python setup.py install --user
```

The starter code includes the following packages:

1. `ros_numpy`: A very useful ROS package that can be used to convert between ROS message datatypes and numpy datatypes. Check out the documentation at `http://wiki.ros.org/ros_numpy`.
2. `stereo_pointcloud`: This is the package that contains all the skeleton code. All files that you will need to edit are in this package.

3 Overview

In this lab you will build a ROS node that subscribes to two image topics, one from the left camera and one from the right camera of a stereo camera pair mounted on a drone. Then, with each pair of `(left, right)` images, the node will:

1. Extract visual features from the left and right image.
2. Match visual features between the two images in a nearest-neighbours fashion to come up with a preliminary set of corresponding points in the two images.
3. Reject spurious matches by enforcing the epipolar constraint between feature matches in the two images.
4. Triangulate to find the 3D coordinates of each feature point in the robot's reference frame.

5. Publish a pointcloud consisting of the locations of all interesting features as seen from the robot's reference frame.

In order to do this, you will write code to perform feature extraction and matching, to perform outlier rejection using the known epipolar geometry between the two cameras, and to triangulate a feature match to compute its spatial location given its image coordinates as seen from the left and right cameras.

3.1 The Data

Unfortunately, we are not able to acquire a drone for every student given the current situation. Luckily, the EuRoC MAV Dataset gives us a nice collection of data from a drone flying around an indoor room that we can use. On the linked page, you can see that there is data taken from several scenes. We will only be focusing on the "Machine Hall 01" dataset, although you are free to play around with the other data on your own. There are links to download bag files and other data, however we have combined all our relevant data into one bag file with a shortened trajectory to reduce file size.

Since bag files store raw information from a wide variety of topics, including those carrying Image messages, they can be quite large in size. Because of this, we are distributing the bag file via Goolge Drive instead of through the git repo. It should be about 1.4GB in size, so make sure you have space for it before downloading the file. You can get the bag file by navigating to the `/bagfiles` folder of the `lab7_starter` package.

```
cd ~/ros_workspaces/lab7/src/stereo_pointcloud/bagfiles
```

and running

```
chmod +x download_bag.sh
./download_bag.sh
```

This will take a few minutes.

You should see a newly created `drone_data.bag` file. Recall that bag files let us record all messages published over select ROS topics to be played back later. We can see what information our bag file has recorded by running

```
rosbag info drone_data.bag
```

You should be able to see a lot of useful information about the bag file here, including its size, its duration, the topics recorded and their message types, etc. We can play back the recorded information by running

```
rosbag play drone_data.bag
```

and you should be able to see similar information about the published topics through a simple `rostopic list` (don't forget to start a ROS master node first!).

Since our bagged data is now being published, we can view it in Rviz to get a feel for what information we have to work with. Open up Rviz with

```
rosrun rviz rviz
```

and then start playing the bag file. You will want to set Global Options > Fixed Frame to be `world`, and then you can use Add > Display Type > TF to see the different frames of the drone moving over time as the bag file plays out. To see the actual view of the drone's cameras, you need to do Add > Display Type > Image twice and the topic of one Image display to `/left/image_raw` and the the other Image display to `/right/image_raw`. You should be able to see a stream of image data from each camera. You may notice that the images appear to have some curvature near their edges, due to distortion from the camera. We will explain how to deal with this later.

Armed with knowledge of the information we can publish from our bag file, we can now setup our subscribers for the main node of the project, located in the `__init__` function of `stereo_pointcloud.py`.

3.2 Task 0

By now, you should be able to visualize messages stored in the bag file through Rviz and properly instantiat subscribers in `stereo_pointcloud.py`.

4 Camera Calibration Review: Pinhole Model

Like in Lab 6, we will be using calibrated cameras in this lab, which means that we know their camera intrinsic matrices K. Recall from lecture and Lab 6 that if $p = (X, Y, Z)$ is a point written in the camera's reference frame, then the homogeneous image coordinates $x = (u, v, 1)$ of the image of p are given by

$$x = \frac{1}{Z} K p$$

further recall that we rearrange the above equation for convenience, and instead state that there exists a scalar "depth" λ such that

$$\lambda x = K p \tag{1}$$

4.1 Normalized image coordinates

For convenience, we will "normalize" our image coordinates. In particular, if x is the pixel coordinates of a point p, then we call $\bar{x} = K^{-1}x$ the *normalized image coordinates* of p. These normalized coordinates have the property that

$$\lambda \bar{x} = p \tag{2}$$

i.e. we can ignore the multiplication by K on the right hand side. When the camera matrix K is known (i.e. the camera is calibrated), we can always freely switch between the normalized and regular coordinates simply by multiplying by K^{-1}. In most of this lab, we will work with normalized image coordinates for simplicity, so your implementations should always start by normalizing the coordinates of an image point before using them.

4.2 Image Distortion

One of the ways in which our camera differs from a true pinhole camera is that it adds nonlinear distortion to the image in addition to the standard pinhole projection. We model this in the form of a "radial-tangential" distortion. When you visualized the raw images in RViz in the previous section, you may have noticed that straight lines do not always show up as straight in the image. Instead, they show up curved, with the curvature being stronger towards the periphery of the image. Luckily, we have a good model for the distortion of this image. As part of the camera calibration parameters, we also have the *distortion coefficients* of the camera, which parameterize this distortion. We can use these coefficients to undistort the image using the `opencv` function `cv2.undistort`. This has already been done for you.

Make sure to have the bagfile playing, and then run the main node using

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

The undistorted images are now being published to the topics `/drone/left/undistorted_image` and `/drone/right/undistorted_image`. Visualize one of these topics in RViz and compare the image to the undistorted version.

5 Stereo Vision

A *calibrated stereo pair* is a pair of cameras attached rigidly to the robot such that (1) both cameras are calibrated, so the camera matrices K_1, K_2 are known and (2) the static transform $g_{21} = (R, T)$ is known. We then generally speak of the "left" and "right" cameras of a given stereo pair when referring to them. In this lab, we will be working with data collected from a drone that is equipped with such a calibrated stereo pair of cameras.

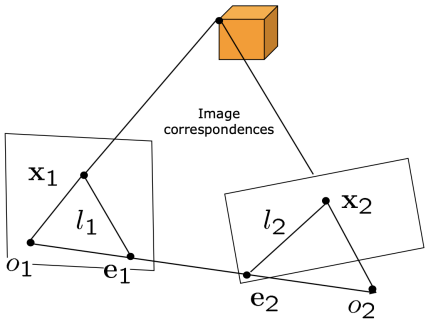

Figure 1: A pair of cameras looking at the same scene. o_1 and o_2 are the optical centers of the cameras. x_1 and x_2 are the image coordinates of the corner of the cube in image 1 and image 2 respectively. Since x_1 and x_2 are the coordinates of the same point in the two image frames, they are called *image correspondences* or *corresponding points* between the images. l_1 and l_2 are the epilines (see section 7.3).

5.1 Corresponding points

A pair of image coordinates x_1 in the frame of the first camera and x_2 in the frame of the second camera are called *corresponding points* if they are both the image of the same point in 3D space. A set of such pairs are then called a set of *point correspondences*. In order to infer the structure of the scene from our stereo pair of cameras, we will need to first find a set of point correspondences between the two images. Recall that given only the image coordinates of a point in one image, we cannot extract its location in 3D space, since all points lying along some ray starting at the optical center of the camera get projected to the same point in the image plane. However, given the coordinates of the same point in two *distinct* images (i.e. when there is a nonzero shift between the two cameras), we can in fact solve for the 3D coordinates of the point they represent. We will see in more detail how to do this in section 8.

In this lab, our goal is to extract visually interesting features from our image stream and track their location in 3D space. In order to do this, we will need to find the corresponding images of the feature point in both cameras of our stereo pair. Then, we will be able to infer the location of that point in the robot's reference frame.

6 Feature Extraction

In dealing with image data we often wish to abstract away some information from the image that allows us to compare the scenes represented in different images in some quantitative way. This is often done by extracting visual features from the image, and then computing a description vector for each feature. There is no concrete definition for what constitutes a "visual feature", and different applications require different choices of features and descriptors. For real-time computer vision applications, a popular choice is to use visual primitives like corners and edges. The most popular feature extraction and description algorithms used in real-time applications such as structure-from-motion and visual SLAM usually extract corners or "interest points". i.e. they extract point-like features from the image. Such features are known as *keypoints*. Point-like features are convenient because the location of such a feature in an image can be described by a single set of 2D image coordinates, and the feature corresponds to objects in the real world in a straightforward way: it is the image of some *point* in 3D space.

Once we have extracted a keypoint, we also need to extract a *feature description* which is computed by featurizing a patch of pixels centered at the keypoint. Popular feature description algorithms include SURF, SIFT, BRIEF, ORB, BRISK etc, which you are encouraged to look up on your own. These algorithms largely differ from each other in how they choose to describe the neighbourhood of the keypoint.

6.1 Feature description

We wish to use the visual features we extract to compare different images of the same scene to detect when two keypoints correspond to the same object in the scene. As such, we would like the feature description to have some desirable properties. An ideal feature descriptor would be one that always gave us a unique description vector for each point in 3D space, so that we would get perfect matches whenever we compared feature descriptions of the same point in two different images. This is, of course, impossible, so we seek feature descriptions with some desirable set of approximate guarantees. Different feature extraction algorithms claim different invariance properties, but in general

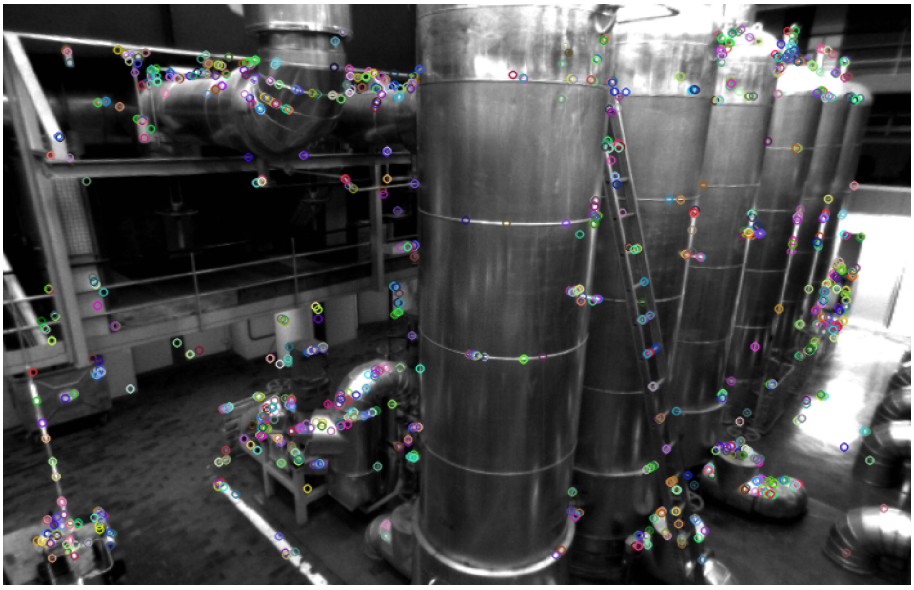

Figure 2: BRISK Features extracted from one frame of the trajectory. This is an image from the left camera of the stereo pair. Notice that only visually distinctive points in the image get extracted, such as corners or points of sudden intensity variation.

we would like our features to be robust to translations and rotations. i.e. a feature corresponding to a point in the scene should get approximately the same descriptor even after the camera rotates or translates by some amount. Additionally, algorithms vary in terms of the kinds of feature description vectors they extract. For instance, feature descriptions extracted by the SURF and SIFT algorithms are vectors of 128 floating point numbers, while those extracted by BRIEF, ORB or BRISK are 512-bit long binary vectors.

6.2 BRISK - Binary Robust Invariant Scalable Keypoints

In this lab, we will be using BRISK, which stands for *Binary Robust Invariant Scalable Keypoints*. The BRISK extractor is optimized for computational speed, while maintaining the same repeatability and invariance guarantees in practice as other state of the art algorithms. It is a keypoint descriptor, and uses the AGAST corner detection algorithm as its underlying feature extractor. It produces a 512-bit binary description vector for each keypoint. In the next section, we will discuss how using binary descriptors makes feature matching very computationally efficient.

7 Feature Matching

Recall that our goal is to come up with corresponding points between the left and right image of the stereo camera pair. As described above, we will begin by extracting BRISK features from both images. We will then compare the description vectors of the features in both images to find the closest matches. We will do this in a nearest neighbours fashion.

7.1 Nearest Neighbours Search

We have extracted feature descriptors $\{f_i\}_{i=1}^n \subset \mathbb{R}^d$ from image 1 and $\{g_i\}_{i=1}^m \subset \mathbb{R}^d$ from image 2. For each feature f_i, we compute the distance of this feature to every feature in image 2 (in the $d-$dimensional feature space). We then do the same for each feature in image 2, finding its distance to each feature in image 1. A pair (f_i, g_j) is taken to be a *match* if g_j is the closest neighbour to f_i in image 2, *and* f_i is the closest feature to g_j in image 1.

Note that in order to perform this matching, we need to define the *distance* between two feature descriptors. When the feature descriptors f_i and g_j are just floating point vectors, we can use the standard euclidean norm $d(f_i, g_j) = \|f_i - g_j\|$. Recall, however, that BRISK, the feature extractor we are using, outputs descriptors of *binary* vectors. So to compare two such features, we will be using a different distance metric called the *Hamming distance*.

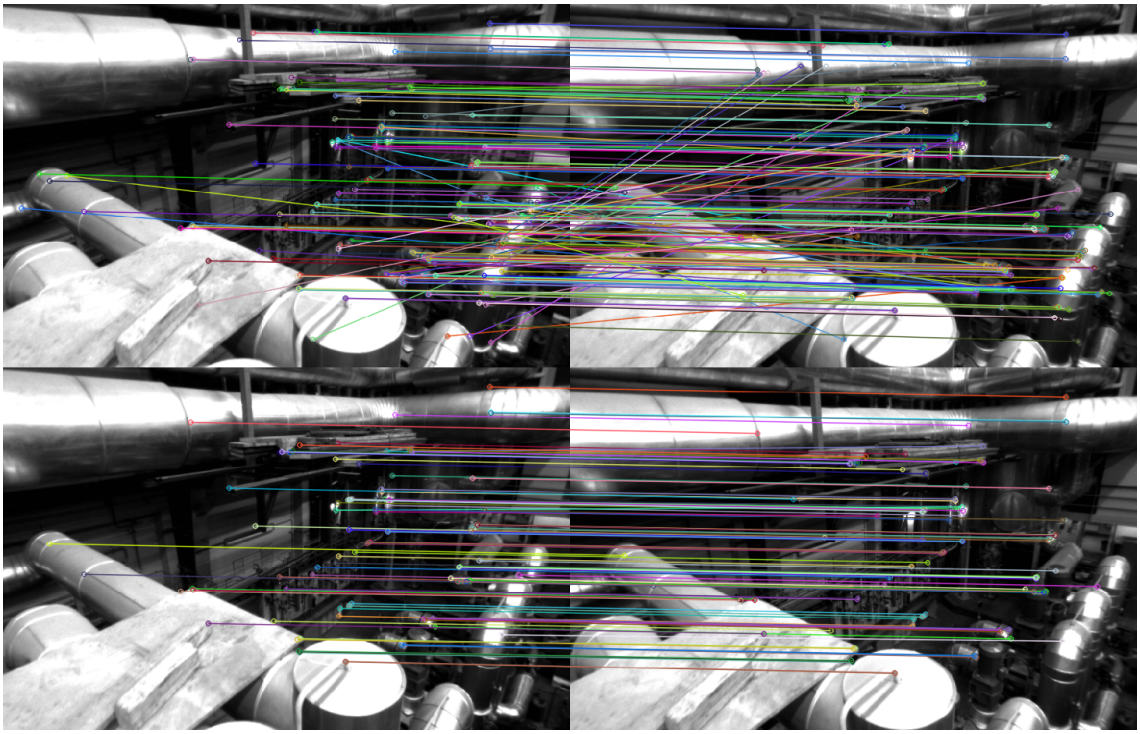

Figure 3: Shows the result of feature matching between a pair of images from the left and right cameras respectively. Top: set of feature matches after simple nearest-neighbours matching. Bottom: filtered matches after rejecting all matches that violate the epipolar constraint.

7.2 Hamming Distance

The *Hamming distance* is a distance metric defined on the space of binary sequences. Given two binary sequences $a = (a_1, \cdots, a_d)$ and $b = (b_1, \cdots, b_d)$ with $a_i, b_i \in \{0, 1\}$, the Hamming distance $d(a, b)$ between them is the number of bits in which a and b *differ*. In other words, it is the minimum number of entries that would need to be changed to make a and b match. Note that unlike euclidean distance between floating point vectors, the Hamming distance between two binary vectors is always an integer.

Using binary description vectors allows us to compare them using the Hamming distance, which is generally faster to compute than the Euclidean distance between comparably-sized floating point descriptors. This is because the hamming distance can be compared using an XOR operation, which is much more efficient than floating point multiplications.

$$d(a, b) = \sum_{i=1}^{d} a_i \boxplus b_i$$

where $\boxplus$ is XOR. We can see that the hamming distance between two bitstrings is exactly equal to the number of active bits in their bitwise XOR.

7.2.1 Task 1

First, uncomment the call to `self.process_images` in the function `camera_callback` in `stereo_pointcloud.py`. Then complete the `extract_and_match_features` function in `stereo_pointcloud.py` by creating a `cv2.BFMatcher` object and then completing the call to `matcher.match`.

- Hint: You will need to pass in an argument to specify the distance function OpenCV should use.
- Hint: This link will be helpful.

To test your implementation, make sure the bag file is playing and first run the main node using

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

and then visualize your feature matches by running

```
rosrun image_view image_view image:=/drone/matches
```

Note: You will get a print statement from stereo_pointcloud.py saying that the number of keypoints in the pointcloud is 0. This is expected, as we have not implemented triangulation yet and so are not publishing any pointclouds.

This will show you an array of four images. The top two images are the left and right images with the feature matches marked on them. The bottom two are the same left and right images. Currently, both the top and bottom will show the same feature matches. In the next section, we will implement functionality to reject bad matches. Once we do that, we will be able to run this visualization again and we will see the filtered matches on the bottom.

7.3 Outlier Rejection: Enforcing the Epipolar Constraint

Despite our best efforts at using nearest neighbours to match features, we will inevitably end up with false matches. This is simply due to the fact that our features are a very local description of the keypoints and hence spurious matches are possible between different points that locally look similar. We want some way to detect and eliminate such spurious matches. We will do this by enforcing the *Epipolar constraint*.

Let (x_1, x_2) be a feature match with *normalized* image coordinates x_1 in image 1 and x_2 in image 2. We know the transform (R, T) between the two cameras, and hence we know the essential matrix $E = \hat{T}R$. Recall that if these two keypoints do in fact correspond to the same point in 3D space, then they must satisfy the Epipolar constraint, i.e. it must be the case that

$$x_2^T E x_1 = 0$$

Of course, since we are dealing with real world noisy data, this quantity will never be exactly zero even for correct matches. So, we will define some geometrically meaningful error for when this constraint is violated.

Definition. The *epiline* corresponding to point x_2 is a line in the image plane of frame 1 which is the set of all points in image 1 that satisfy the epipolar constraint with x_2. If we define $l_1 = E^T x_2$, then the epiline is the set of all points x_1 in normalized homoegeneous image 1 coordinates that satisfy $l_1^T x_1 = 0$.

Likewise, the *epiline* corresponding to point x_1 is a line in the image plane of frame 2 which is the set of all points in image 2 that satisfy the epipolar constraint with x_1. If we define $l_2 = E x_1$, then the epiline is the set of all points x_2 in normalized homoegeneous image 2 coordinates that satisfy $l_2^T x_2 = 0$.

Let $l_1 = E^T x_2 = (a_1, b_1, c_1)$. Then the first epiline is the set of all points $x_1 = (u_1, v_1, 1)$ that satisfy the equation $a_1 u_1 + b_1 v_1 + c_1 = 0$. Likewise, if $l_2 = E x_1 = (a_2, b_2, c_2)$ then the second epiline is the set of all points $x_2 = (u_2, v_2, 1)$ that satisfy the equation $a_2 u_2 + b_2 v_2 + c_2 = 0$.

If x_1 does not lie on the epiline corresponding to x_2 then we should reject the match. In turn, if x_2 does not lie on the epiline corresponding to x_1, then we should reject the match. So we will define an error function that will measure the sum of distance between x_1 and the first epiline and the distance between x_2 and the second epiline. We can use well-known formulas for the distance between a point and a line to do this computation. Our error function can then be written as

$$e(x_1, x_2) = \frac{|a_1 u_1 + b_1 v_1 + c_1|}{\sqrt{a_1^2 + b_1^2}} + \frac{|a_2 u_2 + b_2 v_2 + c_2|}{\sqrt{a_2^2 + b_2^2}}$$

where $(a_1, b_1, c_1)^T = E^T x_2$, $(a_2, b_2, c_2)^T = E x_1$ and $x_i = (u_i, v_i, 1)$. Finally, to reject outliers, we will loop through the set of matches $\{(x_1^{(i)}, x_2^{(i)})\}_{i=1}^n$, and keep a match $(x_1^{(j)}, x_2^{(j)})$ if $e(x_1^{(j)}, x_2^{(j)}) < \delta$, for some threshold δ that we pick. Note that the units of δ are pixels, and it is the sum of the distances between each point to its respective epiline.

7.3.1 Task 2

Implement the functions `FilterByEpipolarConstraints` and `epipolar_error` in `epipolar.py`. To do this you will need to:

1. Enforce epipolar constrains on the image by computing the essential matrix, normalizing the image coordinates, and computing the epilines for the matches. (*Hint*: Go back to Lab 6 for a reminder on what the Camera Intrinsic Matrix contains.)

2. Calculate the distances between epilines and their corresponding points to retrieve the epipolar error.

3. Tune the threshold `epipolar_threshold` being passed into the constructor for `Stereo_Pointcloud`. This is the threshold being used by your epipolar constraint filtering function. Recall that this threshold is a distance in units of pixels. This will be `0.07` by default. It gets passed in in the `__main__` section of `stereo_pointcloud.py` (at the bottom of the file). You should tune it until you get about 20-250 total matches per frame.

Once you have completed this, run the visualization again. This time, the result should be a stream of images with visualizations of all matched features as the top pair of images and only matches that satisfy epipolar constraints marked in the bottom pair. Take note of the number of filtered matches you are getting on average per image. This information is printed by the node `stereo_pointcloud` as "Total matches". You should tune the threshold for epipolar filtering until you get a decent number of matches (around 20-250).

Note that you may get a large variability in number of good feature matches per image along the trajectory (not all frames will be feature rich), so you should sample a good chunk of the trajectory before deciding on a threshold.

Run the main node using:

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

and visualize the feature matches using:

```
rosrun image_view image_view image:=/drone/matches
```

Tip: Make sure the bag file is running.

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain the different topics being published by the bag file.
- Show the moving body frame of the drone relative to the world frame in Rviz.
- Display a stream of distorted images against their un-distorted images in Rviz.
- Display stream of matched features before and after enforcing epipolar constraints using `image_view`. Was your node successful at rejecting bad matches?

---

8 Triangulation

After matching features and rejecting outlier matches, we are left with a set of corresponding points between the left and right image. For each such pair (x_1, x_2), we wish to find the coordinates of the point p they represent in 3D space in the robot's body frame. We have the transform (R, T) between the two camera frames. Let X_1 and X_2 be the 3D coordinates of the point p in the reference frames of cameras 1 and 2 respectively. We will once again assume that the image coordinates (x_1, x_2) are given in normalized form, so that there exist positive depth scalars λ_1, λ_2 such that $\lambda_1 x_1 = X_1$ and $\lambda_2 x_2 = X_2$. Now we can write

$$\lambda_2 x_2 = \lambda_1 R x_1 + T \tag{3}$$

where the only unknowns are λ_1 and λ_2. We can then recast this into a matrix equation

$$\begin{bmatrix} -Rx_1 & x_2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = T \tag{4}$$

Now if we define $A = [-Rx_1 \ x_1] \in \mathbb{R}^{3\times 2}$, $\lambda = (\lambda_1, \lambda_2)^T \in \mathbb{R}^2$, then the above can be expressed as solving for λ in the equation $A\lambda = T$.

However, there are two additional considerations when we deal with real data. Firstly, our measurements for x_1 and x_2 are not perfect, so in general equation (4) may have no exact solutions. Hence, we should find the least squares solution. i.e. we should solve

$$\min_{\lambda \in \mathbb{R}^2} \|A\lambda - T\|_2^2 \tag{5}$$

which has the closed form solution $\lambda^* = (A^\top A)^{-1} A^\top T$. Secondly, it may occasionally be the case that our image measurements are noisy enough that the solution to equation (5) gives us negative values for λ_1 and λ_2. In this case, we should simply throw away that point, as we do not have enough information to locate it.

After solving the above, we will end up with two depths λ_1 and λ_2, which will give us two possible values for the coordinates of p in camera frame 2. These are

$$p_2' = \lambda_2 x_2$$
$$p_2'' = \lambda_1 R x_1 + T$$

We should pick the average of these solutions:

$$p_2 = (p_2' + p_2'')/2 \tag{6}$$

This will give us our estimate of p in the second camera frame. Finally, we can find p by transforming p_2 to be in the robot's body frame instead.

8.0.1 Task 3

Complete the function `least_squares_triangulate` in `stereo_pointcloud.py` so that we publish a pointcloud constructed from matched points in the two images. This function should return your estimate of the input point's 3D location in the reference frame of the right camera. This will involve

1. Finding the least squares solution to 4 for each set of matched image points. You will need to discard solutions where λ_1 and λ_2 is negative.
2. Using the least squares solution to find a 3D point for each image, and taking the average of the point, as shown in equation (6).

Once you are done implementing this function, you can test the whole pipeline by running the main node.

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

You will now be able to visualize the 3D points you found in Rviz. You can do this by using Add > By Display Type > PointCloud2 and then changing its topic to `/drone/pointcloud`. Make sure that you have Global Options > Fixed Frame set to `world`. Finally, set the Decay Time parameter of the PointCloud2 display to some big number greater than 100. By default, Rviz will only display the last published message to a specified topic, whereas we want our published points to persist in order to create a visualization of the environment as the drone flies. We can fix this by changing the "Decay Time" parameter of the display. It is 0 by default, so the pointlcoud messages get cleared as soon as a new message is published. Setting this value to some positive x will cause points to persist for x seconds before being erased. The bag file plays for about 100 seconds so some value $x \geq 100$ will suffice.

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Triangulate corresponding world points for a set of matched image points.
- Display point cloud in Rviz generated from corresponding image points.
- Comment on the quality of the pointcloud. Is it noisy? Are there a lot of stray points?
- Answer the following question: Even though the floor is visible in most of our images, it is not included in the pointcloud at all. Why is this the case?

EECS C106A: Remote Lab 8 - Building Occupancy Grids with STDR Simulator*

Fall 2020

Goals

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

- Use the ROS parameter server to set parameter values that can be shared across multiple nodes
- Understand and explain how an occupancy grid works and when to use one
- Map out the lab space using your own custom occupancy grid
- Point out any important deficiencies in your implementation

---

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.

Contents

Introduction

In this lab, we will build and test one of the most useful data structures in mobile robotics: the occupancy grid.[1] The key idea behind the occupancy grid is to represent space as — you guessed it — a grid, in which every cell, or *voxel*, is either occupied or free. Since nothing is ever really certain in life (i.e., measurements are noisy), occupancy grids actually keep track of the *probability* that each cell is occupied. When the robot receives a measurement of the environment, typically from a laser scanner, it updates these probabilities to incorporate the new information.

This lab is broken up into three phases:

*Developed by David Fridovich-Keil and Laura Hallock, Fall 2017. Updated by Valmik Prabhu, Nandita Iyer, Ravi Pandya, and Philipp Wu, Fall 2018. Converted to remote by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).

[1] A Google search for "occupancy grid" turns up lots of great references that go into more detail.

1. Learn how to use the ROS parameter server.
2. Write the key steps in an occupancy grid update.
3. Test your implementation and identify any shortcomings.

1 Getting started with Git

Our starter code for this lab is a ROS package called `lab8_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 `lab8`.

```
mkdir -p ~/ros_workspaces/lab8/src
cd ~/ros_workspaces/lab8/src
catkin_init_workspace

cd ~/ros_workspaces/lab8
catkin_make
```

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

```
cd ~/ros_workspaces/lab8/src
git clone https://github.com/ucb-ee106/lab8_starter.git
cd ~/ros_workspaces/lab8
catkin_make
source devel/setup.bash
```

We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

1.1 Setting up STDR Simulator

Recall that we introduced you to STDR Simulator in Lab 4. We will again be using it today in Lab 8. First, clone the appropriate packages:

```
cd ~/ros_workspaces/lab8/src
git clone https://github.com/stdr-simulator-ros-pkg/stdr_simulator.git
```

In Lab 4 we had you create a new package and file in order to use the turtlebot teleop keyeboard launch file to control our STDR Sim robot model instead. You can go back to Lab 4 and repeat these steps if you need to or you can find and copy over the package.

```
cp -r ~/ros_workspaces/lab4/src/stdr_teleop ~/ros_workspaces/lab8/src
```

If you choose to copy over the files, be sure to change any references to `lab4_starter` to `lab8_starter` in order for it to work.

Now build and source your workspace (this may take a few minutes)

```
cd ~/ros_workspaces/lab8
catkin_make
source devel/setup.bash
```

2 The ROS parameter server

We haven't really exposed you to the ROS parameter server before, but since it is one of the more useful features of ROS, we want you to get some practice using it.[2] ROS parameters are key-value pairs that you can specify when launching nodes (e.g., in a `launch` file) that may be queried by those nodes at run-time. This can be an extremely useful tool for writing flexible code and for enforcing that multiple nodes hold the same value for some particular variable.

Inside your `lab8_starter` package you will find two source files and two launch files along with the usual `CMakeLists.txt` and `package.xml` files. Open each of these files and make sure that they all make sense to you (one of them should look very familiar!).

Inside the file `demo.launch`, you'll see that a number of command-line arguments are declared (along with default values). These arguments are then mapped to specific parameters in a node called `mapper`. These parameters will need to be read in by that node at run-time.

Task 1: Open the file `occupancy_grid_2d.py` and locate the `LoadParameters` function. We've loaded one parameter for you, but you'll need to finish this function by loading the rest. Note that two of the variables in `occupancy_grid_2d.py`, `x_res` and `y_res`, are not on the parameter server. How do you think you should generate these variables (You should not be editing the launch file)?

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Run `demo.launch` with no errors.
- Explain each parameter you have loaded in `Load Parameters`.

---

[2] See http://wiki.ros.org/rospy_tutorials/Tutorials/Parameters for a more detailed description of the server's purpose and usage.

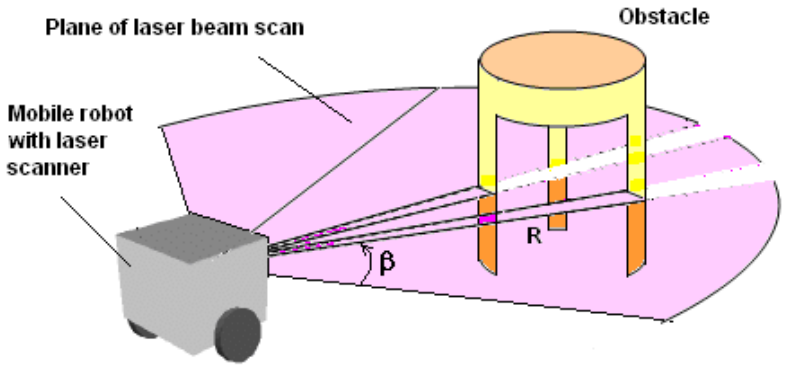

Figure 1: Diagram of a mobile robot with a laser scanner.

3 Generating & updating the occupancy grid

Now for the fun part! In the file `occupancy_grid_2d.py` file, locate the function `SensorCallback` and fill in the details. The main idea here is that each grid cell contains the *log-odds ratio of occupancy*. That is, if p_{ij} is the probability of occupancy at cell (i, j), then the cell actually stores log-odds $\ell_{ij} \triangleq \log\left(\frac{p_{ij}}{1-p_{ij}}\right)$. This may seem like an unnecessary mathematical complication, but it's actually very useful: if we stored probabilities directly, we'd run into trouble trying to keep all of our probabilities positive when performing updates.

When a scan ray terminates at a particular cell, that cell's log-odds ratio is incremented by some small amount — i.e., $\ell \longleftarrow \ell + \Delta_{occ}$ — and then thresholded for numerical stability. Likewise, when the ray passes through a cell (and does not terminate there), that cell's log-odds ratio is decremented by some other amount — i.e., $\ell \longleftarrow \ell + \Delta_{free}$, where by convention Δ_{free} is negative — and similarly thresholded. In particular, these increments are computed as the log-odds ratios corresponding to *the probability that a cell is occupied given that a ray terminates there* and *the probability that a cell is occupied given that a ray passes through it.* Note that if our sensors were perfect, these values would correspond to 1 and 0, respectively; if that were the case, what would the log-odds update values be?

Before starting any edits, read through the inline comments and try to understand what the function is doing at each step. This callback function receives a `sensor_msgs/LaserScan` message, which represents a single line depth scan around the robot (as would be generated by a LIDAR). The scan begins at some angle, gathers range information at a certain angular increment, and ends at some second angle. Use `rosmsg show` or the online ROS documentation to see the contents of this message.

The callback function iterates through each ray of the scan using the `enumerate` function (look up the documentation for this function if you don't understand what it's doing). The first thing you'll be implementing is finding the angle of the ray in the *fixed frame*. A quick look at `demo.launch` shows that the fixed frame is called `robot0`, while the sensor frame is `robot0_laser_0`.

Note: if you move the turtlebot manually (say by picking it up), the odometry won't be able to detect it and the `odom` frame will be wrong. If you do this, restart the bringup sequence on your turtlebot to reset the odom frame.

The next thing you'll be doing is "walking" backwards along the ray from the scan point to the sensor, updating the log-odds in each voxel the ray passes through. The `numpy.arange` function can be helpful in defining your loop. The function `PointToVoxel`, defined below `SensorCallback`, may be useful as well. If a voxel is occupied, you should increase the log odds at that voxel by your occupied update value, thresholding it at your occupied threshold value. If a voxel is free, you should increase the log odds at that voxel by your free update value, thresholding it at your free threshold value. Remember that you should only be updating each voxel once per ray.

Task 2: Complete the `SensorCallback` function. When you're done, try running the launch file again, and make sure you don't get any error messages.

3.1 Testing your occupancy grid

Look back at `demo.launch` again. You'll notice that the node's main source file is `mapping_node.py`, not `occupancy_grid_2d.py`. (Although this project is small by most standards, it is generally good practice to separate the actual executable node

file from other files implementing different classes that your node uses.) Examine how the `mapping_node.py` file creates an occupancy grid, initializes it, and on success just idles. If you trace that initialization call into the `OccupancyGrid2d` class, you'll see that initialization loads all parameters, registers publishers and subscribers, and sets up any other class variables. If any of that fails, it returns `False`, which causes the whole node to crash. This is a very safe way to build your system because it minimizes the chance that your code crashes mid-operation. We strongly encourage you to use this sort of architecture in your projects.

Now let's try testing out our occupancy grid! First we need to start up our simulation.

```
roslaunch lab8_starter stdr_maze_env.launch
```

Next run `demo.launch` (there should be no errors)

```
roslaunch lab8_starter demo.launch
```

We then need to start up RViz and our keyboard teleop node

```
roslaunch stdr_launchers rviz.launch
roslaunch stdr_teleop stdr_keyboard.launch
```

Task 3: In RViz, find and add the appropriate topic where your occupancy grid is being published to. You should be able to see a mostly red and blue map being generated in RViz as you drive your robot around. Do you notice any systematic errors? Where are they coming from, and how would you address them?

Next, experiment with changing some of the parameters defined in your parameter server. While you can simply change the values in your launch file, it's cleanest (and most convenient) to set them via command line so you can experiment with many different values without changing the defaults. (Hint: You've actually done this before using Baxter/Sawyer — `electric_gripper` is a parameter value!)

Experiment with changing the downsampling rate parameter. What is the downsampling rate's function, and why is it important? (The comments in the `occupancy_grid_2d.py` file might be helpful here.)

Lastly, experiment with changing the resolution of the map. (Note that the length of each cell isn't explicitly defined in the parameter server but can be calculated from the values there; which parameters do you need to modify to make the cells larger and smaller?) How does your map behave differently? Do you notice any change in error patterns?

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Demonstrate the odometry-based localization and the associated map.
- Compare your map with the one you generated in Lab 4 using the `gmapping` demo.
- Explain any bottlenecks in the code — what's the slowest part of the computation?
- Change a parameter of the launch file from the command line.
