Project 0: Review of Robot Operating System (ROS)*

Spring 2024

Recommended Completion Date: 1/31/24

Goals

This lab draws from various parts of Labs 1 - 4 and Lab 7 from 106A. For a deeper look at the material, please refer to these documents.

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, services, and messages used by a given node
- Write a new node that interfaces with existing ROS code
- Use the `ar_track_alvar` package to identify AR tags
- Connect to the lab robots and control them using the `MoveIt!` package

---

Note: Much of this lab is borrowed from the official ROS tutorials. We picked out the material you will find most useful in this class, but feel free to explore other resources if you are interested in learning more.

Contents

*Developed by Valmik Prabhu, Spring 2018. Extended by Valmik Prabhu and Chris Correa, Spring 2019. Compiled heavily from labs written by Aaron Bestick and Austin Buchan, Fall 2014, and Valmik Prabhu, Philipp Wu, Ravi Pandya, and Nandita Iyer, Fall 2018. Further developed by Tiffany Cappellari and Amay Saxena, Spring 2020. Additional edits and changes made by Jaeyun Stella Seo and Josephine Koe, Spring 2022. Revised for Spring 2023 by Han Nguyen. Revised for Spring 2024 by Kirthi Kumar.

1 Creating an Instructional Account

To create a lab account, visit `https://acropolis.cs.berkeley.edu/~account/webacct/` and click "Login using your Berkeley CalNet ID." On the next page, next to ee106a click "Create a new Account" and save this login information to use for your 106A labs! Make sure both partners do this so everyone has an account.

2 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 by demonstrating ROS's functionality through an example. Consider a two-joint manipulator arm for a pick-and place task.

A typical robotic system has numerous sensing, actuation, and computing components. Suppose our system has:

- Two motors, each connected to a revolute joint (orange and blue)
- A motorized gripper on the end of the arm (green)
- A stationary camera that observes the robot's workspace

To pick up an object, the robot might:

- Use the camera to measure the position of the object
- Command the arm to move toward the object's position
- Once properly positioned, 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 (and computer science in general) is to divide the control software into various low-level, independent control loops, each controlling a single task on the robot. In our example system above, we might divide the control software into:

- A control loop for each joint that, given a position or velocity command, controls 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

Given this structure for the robot's software, we then couple these low-level loops together via a single high-level module that performs supervisory control of the whole system:

- Query the camera sensing loop for a single image.
- Use a vision algorithm to compute the location of the object to grasp
- Compute the joint angles necessary to move the manipulator arm to this location
- Sends position commands to each of the joint control loops telling them to move to this position
- Signal the gripper control loop to close the gripper to grab the object

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 makes the system modular, making it easier to reuse code across robotic platforms.

In ROS, each individual control loop is known as a **node**, an individual software process that performs a specific 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.

We can then visualize the communication and interaction between different software components via a **computation graph**, where:

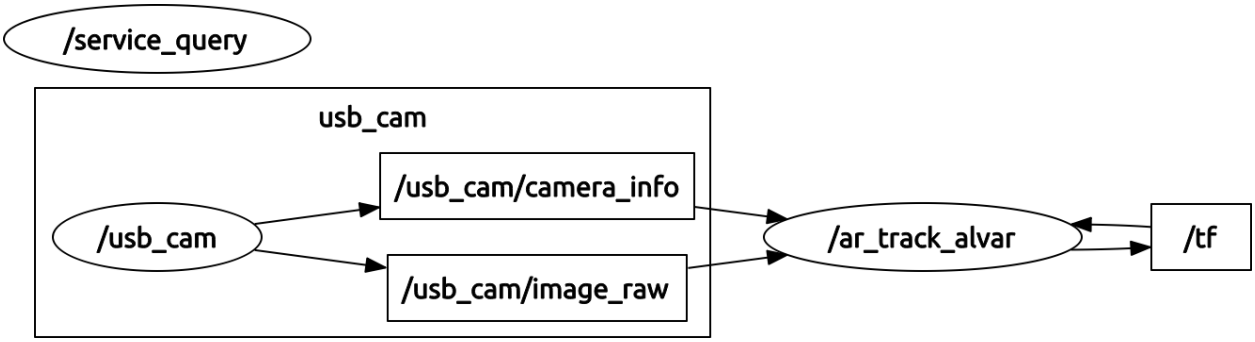

Figure 1: Example computation graph.

- Nodes are represented by ovals (ie `/usb_cam` or `/ar_track_alvar`).

- Topics are represented by rectangles (ie `/usb_cam/camera_info` and `/usb_cam/image_raw`).
- The flow of information to and from topics and represented by arrows. In the above example, `/usb_cam` publishes to the topics `/usb_cam/camera_info` and `/usb_cam/image_raw`, which are subscribed to by `/ar_track_alvar`.
- While not shown here, services would be represented by dotted arrows.

3 Initial configuration

The lab machines you're using already have ROS and the Sawyer robot SDK installed, but you'll need to perform a few user-specific configuration tasks the first time you log in with your class account.

3.1 Workspace setup

Clone the starter code repository by running the following command from your home folder (~):

```
cd ~
git clone https://github.com/ucb-ee106/106a-fa23-labs-starter.git ros_workspaces
```

This clones the directory as "`ros_workspaces`" (we'll explain what this name means later), which should contain all of the starter code from last semester's labs.

Run the following command to rename the remote from "`origin`" to "`starter`".

```
cd ~/ros_workspaces
git remote rename origin starter
```

Now, if you ever want to pull updated starter code, you'd execute the following command:

```
git pull starter main
```

3.2 Shell Environment Setup

The `.bashrc` file is a script that is executed whenever a new interactive non-login shell is started; it provides a convenient way to configure and customize your Bash shell environment.

Open the `.bashrc` file (located in your home directory, `~/.bashrc`) in a text editor. If you don't have a preferred editor, we recommend Sublime Text, which is preinstalled on lab computers and can be accessed using `subl <filename>`. Your `~/.bashrc` should look like

```
# Filename: .bashrc
# Description: This is the standard .bashrc for new named accounts.
#
# Please (DO NOT) edit this file unless you are sure of what you are doing.
# This file and other dotfiles have been written to work with each other.
# Any change that you are not sure off can break things in an unpredicatable
# ways.

# Set the Class MASTER  variables and source the class master version of .cshrc

[[ -z ${MASTER} ]] && export MASTER=${LOGNAME%-*}
[[ -z ${MASTERDIR} ]] && export MASTERDIR=$(eval echo ~${MASTER})

# Set up class wide settings
for file in ${MASTERDIR}/adm/bashrc.d/* ; do [[ -x ${file} ]] && . "${file}"; done

# Set up local settings
for file in ${HOME}/bashrc.d/* ; do [[ -x ${file} ]] && . "${file}"; done
```

There is no need to fully understand the code above, but note that ~/.bashrc runs a series of set-up scripts, including all files in the /home/ff/ee106a/adm/bashrc.d/ folder. One file of interest is 95−select_robot.sh, which looks like

```
case "$HOSTNAME" in
   c105-1|c105-2|c105-3|c105-4|c105-5|c105-11 )
       source /home/ff/ee106a/turtlebot_setup.bash
       echo "Setup Turtlebot!" >&2
       ;;
   c105-6|c105-7|c105-8|c105-9|c105-10 )
       source /home/ff/ee106a/sawyer_setup.bash
       echo "Setup Sawyer!" >&2
       ;;
   * )
       echo "This is not a Sawyer or Turtlebot workstation!" >&2
       ;;
esac
```

This runs a different custom configuration script for ROS based on whether we are at a Turtlebot (1-5) or Sawyer lab station (6-10).

In the Turtlebot turtlebot_setup.bash script, we

- Run source /opt/ros/noetic/setup.bash, which sets up the default ROS environment in the terminal.
- Set the $ROS_MASTER_URI environment variable, which specifies where the ROS Master can be reached, to your local computer. The ROS Master will be detailed later in Section **??**.

On the other hand, in the Sawyer sawyer_setup.bash script, we

- Run source /opt/ros/eecsbot_ws/devel/setup.bash, which sets up the ROS environment in the terminal just like source /opt/ros/noetic/setup.bash does, but also includes additional packages to interact with the Sawyer robot.
- Sets $ROS_MASTER_URI to be the Sawyer robot instead of the local computer.

4 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.

4.1 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 you will want to create a new catkin workspace.

For this lab, begin by creating a directory for the workspace. Create the directory `proj0` in your home `ros_workspaces` folder. The directory "`ros_workspaces`" will eventually contain several project-specific workspaces (named `proj1`, `proj2`, etc.) Next, create a folder `src` in your new workspace directory (`proj0`).

After you fill `src` with packages, you can build them by running "`catkin_make`" **from the workspace directory (`proj0` 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`).

4.2 Creating a New Package

Let's create a new package. From the `src` directory, run

```
catkin_create_pkg proj0_turtlesim rospy roscpp std_msgs geometry_msgs turtlesim
```

Our package is called `proj0_turtlesim`, and we add `rospy`, `roscpp`, `std_msgs`, `geometry_msgs`, and `turtlesim` as dependencies. `rospy` and `roscpp` allow ROS to interface with code in Python and C++, and `std_msgs` and `geometry_msgs` are both message libraries. Messages are data structures that allow ROS nodes to communicate. You'll learn more about them in Section 5.

4.3 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`. Run the command again from the `proj0` directory.

```
catkin_make
```

`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 package "`proj0_turtlesim`" 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/proj0/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: *Any time 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.*

4.4 File System Tools

When working with ROS, you will invariably be working with many packages stored in many places. ROS provides a collection of tools to navigate. Some of the most useful are `rospack`, `rosls`, and `roscd`. Type

```
rospack find intera_examples
```

This should return the directory at which the `intera_examples` package is located. What do you think the others do? Note that these commands only work on packages in the `$ROS_PACKAGE_PATH`, so make sure to source the relevant workspace with your `.bashrc` file before using these commands. If this command does not work for you, check your bashrc file. Which block do you think needs to be uncommented to find the packages related to intera (Sawyer)?

4.5 Anatomy of a package

`cd` into `/opt/ros/eecsbot_ws/src/intera_examples`. The `intera_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

Open the `package.xml` file with the command `subl package.xml`. It should look something like this:

```
<?xml version="1.0"?>
<package format="3">
  <name>intera_examples</name>
  <version>5.3.0</version>
  <description>
    Example programs for Intera SDK usage.
  </description>

  <maintainer email="rsdk.support@rethinkrobotics.com">
    Rethink Robotics Inc.
  </maintainer>
  <license>Apache 2.0</license>
  <url type="website">http://sdk.rethinkrobotics.com/intera/</url>
  <url type="repository">
    https://github.com/RethinkRobotics/intera_sdk
  </url>
  <url type="bugtracker">
    https://github.com/RethinkRobotics/intera_sdk/issues
  </url>
  <author>Rethink Robotics Inc.</author>

  <buildtool_depend>catkin</buildtool_depend>

  <buildtool_depend condition="$ROS_PYTHON_VERSION == 2">python-setuptools</buildtool_depend>
  <buildtool_depend condition="$ROS_PYTHON_VERSION == 3">python3-setuptools</buildtool_depend>
  <build_depend>rospy</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>intera_core_msgs</build_depend>
  <build_depend>intera_motion_msgs</build_depend>
  <build_depend>intera_interface</build_depend>

  <exec_depend>rospy</exec_depend>
  <exec_depend>actionlib</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>control_msgs</exec_depend>
  <exec_depend>trajectory_msgs</exec_depend>
  <exec_depend>cv_bridge</exec_depend>
  <exec_depend>dynamic_reconfigure</exec_depend>
  <exec_depend>intera_core_msgs</exec_depend>
  <exec_depend>intera_motion_msgs</exec_depend>
  <exec_depend>intera_interface</exec_depend>
  <exec_depend>joystick_drivers</exec_depend>

</package>
```

Along with some metadata about the package, the `package.xml` specifies 11 packages on which `intera_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`.

5 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.

5.1 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.

5.2 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.

6 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.

6.1 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 like Figure 2. 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 2: Output of rqt_plot when running turtlesim.

Note: *The* `rqt_graph` *package has been known to behave erratically. If you don't see the exact same graph, everything's probably fine.*

6.2 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 3.

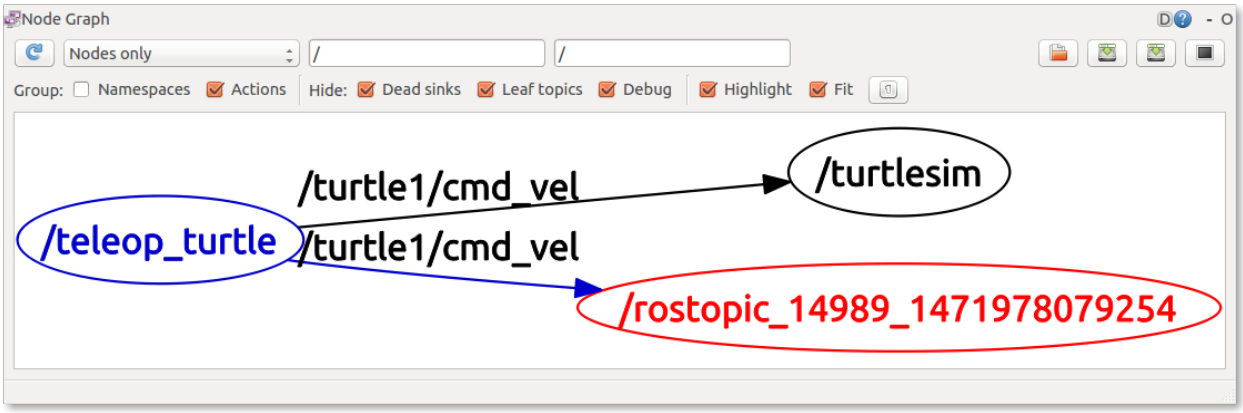

Figure 3: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

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
```

6.3 Examining ROS messages

Inter-node communication is done via messages, so understanding how to examine already-existing messages is an essential skill. Let's take a deep dive into the turtlesim command messages. Your `rostopic list` should produce the following output:

```
/rosout
/rosout_agg
/statistics
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

We'll be looking at the `/turtle1/cmd_vel` topic. Type

```
rostopic info /turtle1/cmd_vel
```

As you can see, the message "`Type`" is `geometry_msgs/Twist`. Here `Twist` is the name of the message, and it's stored in the package `geometry_msgs`. ROS also has utility methods for messages, in addition to those for packages and topics. Let's use them to learn more about the `Twist` message. Type

```
rosmsg show geometry_msgs/Twist
```

Your output should be

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

What do you think this means? Remember that a ROS message definition takes the following 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.

Keep the turtlesim running for use in the next section.

7 Understanding ROS services

Services are another way for nodes 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.

7.1 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.

7.2 Calling services

Let's try calling 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.

8 Understanding ROS Publishers and Subscribers

Our starter code for this project can be found here. You can clone it by running

```
git clone https://github.com/ucb-ee106/106b-sp23-project-starter
```

In fact, all of our project starter code will be in this repository for the semester. You can find the starter code for each project in their respective folders labeled "`proj0`", "`proj1a`", etc. At the moment, there is no simple way to clone sub-directories of the repository, so we recommend pull this one and drag out the contents you need.

Inside `proj0`, you will find a package called `chatter`. Move this to the `src` directory in your `proj0` workspace, build it using `catkin_make`, and source the workspace.

Now, examine the files in the `src` directory inside `chatter`. `example_pub.py` and `example_sub.py` are both 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.

Close your turtlesim nodes from the previous section, but leave `roscore` running. In a new terminal, type

```
rosrun chatter example_pub.py
```

This 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.

9 Writing a controller for turtlesim

Let's replace `turtle_teleop_key` with a new controller, and learn how to interact with previously existing ROS code. Go back to the starter code and put `controller.py` into the `src` folder inside the `proj0_turtlesim` package you created earlier.

We need `controller.py` to have the following functionality:

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

  ```
  rosrun proj0_turtlesim controller.py turtle1
  ```

  will start a controller node that controls turtle1).

- Publish velocity control messages on the appropriate topic 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 `input()` and the WASD keys instead.)

Your first step is to figure out what topic to which to publish and which message type to use. Once you've figured that out, edit lines 16 and 34 accordingly. Then edit the "Your Code" section starting at line 44 to query the user for a command, process it, and set it as a variable of the correct message type.

Concept Checkpoint:

Since this project is optional and being released alongside project 1, there is no check off requirement. However, you can ask a TA during your lab section or office hours to check if your understanding of ROS is sound. By now, you should be able to answer the following questions:

1. What is the difference between a topic and service?
2. What does roscore do?
3. Demonstrate your turtlesim teleop controller.

---

10 AR Tags

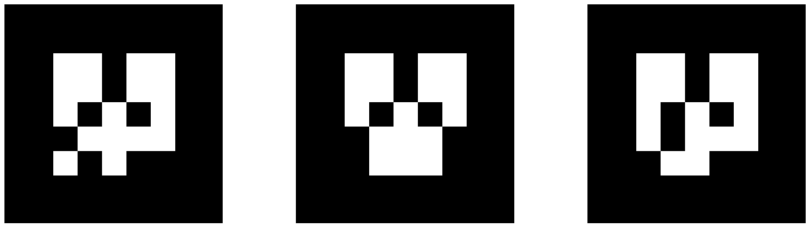

Figure 4: Example AR Tags

AR (Augmented Reality) Tags have been used to support augmented reality applications to track the 3D position of markers using camera images. An AR Tag is usually a square pattern printed on a flat surface, such as the patterns in Figure 4. The corners of these tags are easy to identify from a single camera perspective, so that the homography to the tag surface can be computed automatically. The center of the tag also contains a unique pattern to identify multiple tags in an image. When the camera is calibrated and the size of the markers is known, the pose of the tag can be computed in real-world distance units. There are several ROS packages that can produce pose information from AR tags in an image; we will be using the `ar_track_alvar` [1] tutorial.

In EECS106A we used the webcams in the lab to track AR tags, and we will do so again in this lab. In other applications, we may use Sawyers's hand cameras since we can get the pose of the AR tag relative to the robot.

10.1 Webcam Tracking Setup

1. Clone the `ar_track_alvar` package to the `src` directory of your `proj0` workspace.

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

   Next go back to the `src` directory of your workspace and install `usb_cam`

   ```
   git clone https://github.com/ros-drivers/usb_cam.git
   ```

2. In the starter, you will find a package called `ar_tracking_pkg`. Move it to the `src` directory of your `proj0` workspace.

3. Next, we need to make sure we have access to the camera calibration parameters for our Logitech webcams. Move the `camera_info` folder present in the resources folder into your `~/.ros` directory:

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

4. If any other parameters have changed, such as the name of the webcam, make sure they are consistent in the launch file (i.e., ensure that you are properly using either the Microsoft or the Logitech parameters).

5. Run `catkin_make` from the workspace (this may take a while).

6. Find or print some AR Tags. There should be a class set in Cory 105. If you are having difficulty finding them, please ask your Lab TA or a Lab Assistant. Please only use these for testing and leave them unmodified so others can use them. The `ar_track_alvar` documentation has instructions for printing more tags that you can use in your projects.

10.2 Visualizing results

Once the tracking package is installed, you can run tracking by launching the `webcam_track.launch` file. Launch files allow you to run multiple nodes at once and pass parameters to the parameter server (like the `camera_info_url` parameter you just edited). Do this by typing:

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

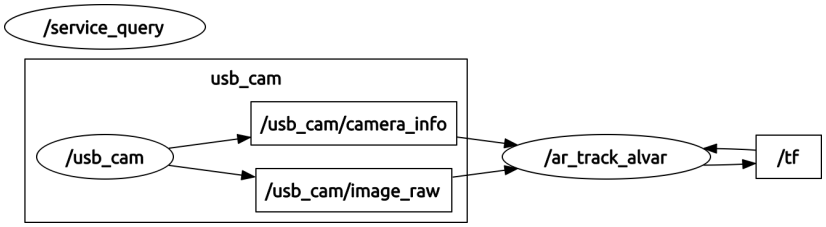

Figure 5: RQT Graph using AR Tags

```
roslaunch ar_tracking_pkg webcam_track.launch
```

Using `rostopic list`, you should see topics `/visualization_marker` and `/ar_pose_marker` being published. They are only updated when a marker is visible, so you will need to have a marker in the field of view of the camera to get messages. Running `rqt_graph` at this point should produce something similar to Figure 5. As this graph shows, the tracking node also updates the `/tf` topic to have the positions of observed markers published in the TF Tree.

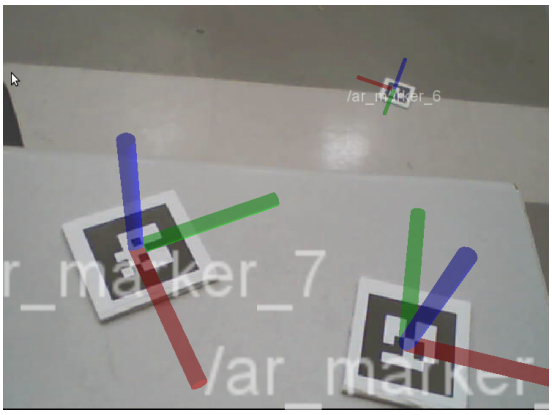

Figure 6: Tracking AR Tags with webcam

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

```
rosrun rviz rviz
```

From the Displays panel in RViz, add a "Camera" display. Set the Image Topic of the Camera Display to the appropriate topic (`/usb_cam/image_raw` for the starter project), and set the Global Options Fixed Frame to `usb_cam`. (Note: you may need to place an AR tag in the field of view of the camera to cause the `usb_cam` frame to appear and for the Camera display to show an image.) You should now see a separate docked window with the live feed of the webcam. Finally, add a TF display to RViz. At this point, you should be able to hold up an AR Tag to the camera and see coordinate axes superimposed on the image of the tag in the camera display. Figure 6 shows several of these axes on tags using the lab webcams. Making the marker scale smaller and disabling the Show Arrows option can make the display more readable. This information is also displayed in the 3D view of RViz, which will help you debug spatial relationships of markers for your project. Alternatively, you can display the AR Tag positions in RViz by adding a Marker Display to RViz. This will draw colored boxes representing the AR Tags.

Concept Checkpoint:

By now you should be able to:

1. Show RViz with AR tracking.

2. Walk through and explain various aspects of `webcam_track.launch`.

---

11 Connecting to the Robot

Most of the work in this class will be done on a Sawyer robot, so we will now teach you how to connect them. Close all running ROS nodes and terminals from the previous part, including the one running `roscore`, before you begin. **Additionally, ensure that you have been trained by the course instructors in the proper safety procedures (including use of the e-stop button) and etiquette for running Sawyer.**

Navigate to the root folder of your workspace (`~/ros_workspaces/proj0`), and make a symbolic link to the Baxter environment script
`/opt/ros/eecsbot_ws/intera.sh` using the command

```
ln -s /opt/ros/eecsbot_ws/intera.sh ~/ros_workspaces/proj0/
```

Note: *Make sure your* `.bashrc` *has the correct block uncommented.*

Now run `./intera.sh [name-of-robot].local` (where `[name-of-robot]` is either `azula`, `alice`, `amir`, `ada`, or `alan`) in your folder to set up your environment for interacting with Sawyer. This clears the `$ROS_PACKAGE_PATH`, so you'll need to run `source devel/setup.bash` again.

To test that the Sawyer arm works, we'll run Sawyer's `sawyer_tuck` script. This script is a good test to run, and places the arms in a reasonably-useful configuration. First, enable the robot with:

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

(Sawyer may already be enabled, in which case this command will do nothing.)

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 `right_gripper_tip`. If the robot does not have gripper attached use `right_hand` instead.

With the joints enabled, grasp the sides of Sawyer's wrist, placing the arm in a gravity-compensation mode, where it can be moved easily by hand. (*Note*: If you have never used gravity compensation mode and are having trouble manipulating the robot, **ask an instructor for assistance**. You shouldn't have to use much force to move the robot around!)

You should see the transformation that `tf` returns changing as you move the arm. The `tf` package is an incredibly useful utility, which you will likely use quite often. Now we'll run tuck arms:

```
roslaunch intera_examples sawyer_tuck.launch
```

12 Using MoveIt

MoveIt is a useful path planning package that abstracts the intreraction between third-party planners, controllers, and your code. Its path planning functions are accessible via ROS topics and messages, and a convenient RViz GUI is provided as well. In this section, we'll just look at the GUI. For more practice, look at labs 5 and 7 from EECS 106A.

12.1 Using the MoveIt GUI

In this section, you'll use MoveIt's GUI to get a basic idea of what types of tasks path planning can accomplish. Make sure you have run the `intera.sh` script in each terminal window you use. First run Sawyer's joint trajectory controller with the command

```
rosrun intera_interface joint_trajectory_action_server.py
```

Next, in a new window, start MoveIt with

```
roslaunch sawyer_moveit_config sawyer_moveit.launch electric_gripper:=true
```

for Sawyer, omitting the last argument(s) if your robot does not have a gripper.

The MoveIt GUI should appear with a model of the Sawyer robot. In the Displays menu, look under "MotionPlanning" $\implies$ "Planning Request." Under "Planning Request" check the "Query Goal State" box to show the specified end states (while you can query the start state as well, we're currently connected to the robot, and the robot's current state is the default start state). You can now set the goal state 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" $\implies$ "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.

When you're satisfied with the motion you see in the simulation, click "Execute." MoveIt will send the plan to a controller, which will execute it. Remember that when running the robot, the EStop should **always** be within reach.

Note: *Never use the "Plan and Execute" option in the MoveIt GUI. Always examine the path visually before hitting execute to ensure safety.*

Concept Checkpoint

By now you should be able to:

1. Move the Sawyer arm using the MoveIt! GUI.
2. Show your Lab TA that you got all questions correct on the Robot Usage Quiz.
3. Show that you have submitted a .txt file with your SID to the Lab Paper Questions assignment on Gradescope.

---

13 What's Next?

See if you can try out Lab 7 or 8 from Fall 2023's EECS 106A if you haven't tried them already. Particularly, Lab 7 will be extremely useful in Project 1.

14 Typo Reporting

This is a lot of text, and our tex editor doesn't even do spell check. If you notice any typos or things in this document or the starter code which you think we should change, please let us know by telling a Lab TA. It's really easy to miss things, so please help us out and make the course better for the next generation of students.