EECS C106B: Lab 0B - Review of Robot Operating System (ROS) *

Spring 2021

Goals

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

- Use the `catkin` tool to build the packages contained in a ROS workspace with the appropriate dependencies specified
- 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
- Simulate the lab robots and control them using the `MoveIt!` package
- Approximate the pose of AR tags in the field of view of a Baxter camera

---

Contents

1 Creating ROS Workspaces and Packages **2**
- 1.1 Creating a workspace . . . 2
- 1.2 Creating a New Package . . . 2
- 1.3 Building a package . . . 2
- 1.4 File System Tools . . . 3
- 1.5 Anatomy of a package . . . 3

2 Understanding ROS nodes **4**
- 2.1 Running roscore . . . 4
- 2.2 Running turtlesim . . . 5

3 Understanding ROS topics **5**
- 3.1 Using rqt_graph . . . 5
- 3.2 Using rostopic . . . 6
- 3.3 Examining ROS messages . . . 6

4 Understanding ROS services **7**
- 4.1 Using rosservice . . . 8
- 4.2 Calling services . . . 8

5 Understanding ROS Publishers and Subscribers **8**

6 Writing a controller for turtlesim **9**

*Converted to remote by Josephine Koe, Spring 2021 (the academic year of the plague). 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.

7 Connecting to the Robot **10**

8 Using MoveIt **10**
8.1 Using the MoveIt GUI 10

9 AR Tags **11**
9.1 AR Tag Tracking Setup 11

10 Typo Reporting **12**

1 Creating ROS Workspaces and Packages

Begin by creating a directory for all of your project workspaces for the semester.

```
mkdir ~/ros_workspaces/
```

1.1 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 and initialize a new catkin workspace.

For this lab, begin by creating a directory for the workspace. Create the directory `lab0_ws` in your `ros_workspaces` folder. The directory "`ros_workspaces`" will eventually contain several project-specific workspaces (named `proj1_ws`, `proj2_ws`, etc.) Next, create a folder `src` in your new workspace directory (`lab0_ws`). **From 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** (`lab0_ws` 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`).

1.2 Creating a New Package

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

```
catkin_create_pkg lab0_turtlesim rospy roscpp std_msgs geometry_msgs turtlesim
```

Our package is called `lab0_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++, `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 2.

1.3 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 `lab0_ws` 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 `lab0_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/lab0_ws/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.*

1.4 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`. In a terminal window, try running

```
roscd baxter_examples
```

This should move you to the directory at which the `baxter_examples` package is located. What do you think the other commands do? Note that these commands only work on packages in the `$ROS_PACKAGE_PATH`, so make sure to source the relevant workspace before using these commands.

1.5 Anatomy of a package

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 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`.

2 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.

2.1 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.

2.2 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.

3 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.

3.1 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 1. 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 subscribes to this same topic to receive the control messages.

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

Figure 1: Output of rqt_graph when running turtlesim.

3.2 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.

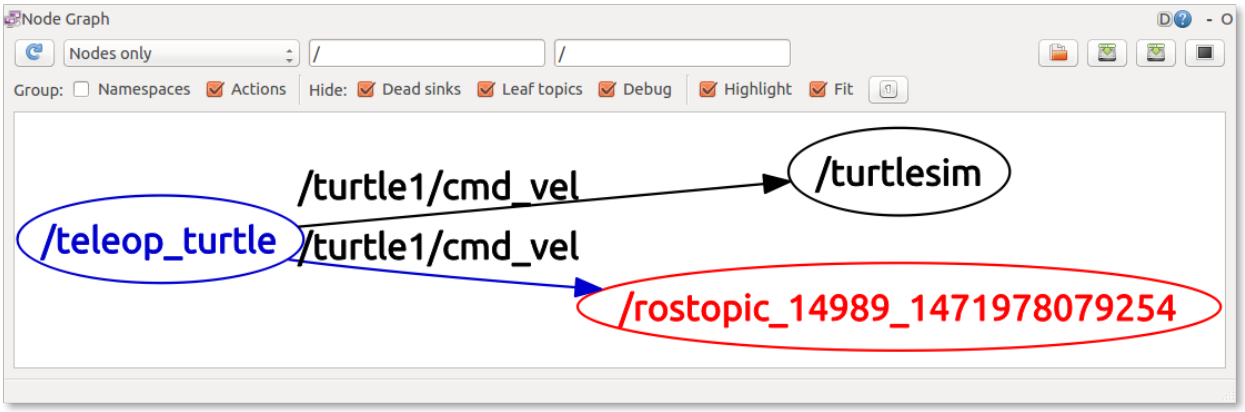

Figure 2: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

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
```

3.3 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
/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:

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
...
```

Each `fieldtype` is one of

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

and each `fieldname` identifies each of the data fields contained in the message.

Keep the turtlesim running for use in the next section.

4 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.

4.1 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.

4.2 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.

5 Understanding ROS Publishers and Subscribers

Clone our starter by running

```
git clone https://github.com/ucb-ee106/lab0_starter.git
```

wherever you would like. Inside the repository, you will find a package called `chatter`. Move this to the `src` directory in your `lab0_ws` 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.

6 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 folder and put `controller.py` into the `src` folder inside the `lab0_turtlesim` package you created earlier.

Edit `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 lab0_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 which message type to use and what topic to publish to. Then query the user for a command, process it, and set the published message as a variable of the correct message type. Remember to make your `controller.py` script executable before running it.

Sanity Checkpoint

You do not need to get checked off at this sanity checkpoint, but be prepared to answer the following questions during your checkoff after completing the entire lab.

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

---

7 Connecting to the Robot

Most of the work in this class will be done on a Baxter or Sawyer robot, so we will now teach you how to simulate them. (We will be using the Baxter in this lab, but the Sawyer is operated almost identically.) Close all running ROS nodes and terminals from the previous part, including the one running `roscore`, before you begin.

In this section, we will be using a launch file to start up the Baxter simulation. Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

Note: *With a real robot, you would first have to* `ssh` *into the robot with the* `baxter.sh` *or* `intera.sh` *scripts in* `~/106b_packages_ws`*. These scripts allow you to run commands on the physical robot from your computer. You can still enter into a simulation environment by running* `./baxter.sh sim` *or* `./intera.sh sim`*, and this might be useful if you are working on a computer that interacts with the robot in both simulation and hardware because the script resets certain environment variables for each scenario. Since we will only be working in simulation, running these scripts is not strictly necessary in this case.*

Start up the Baxter Gazebo simulation by running

```
roslaunch baxter_gazebo baxter_world.launch
```

To test that the arms work, we'll run Baxter's `tuck_arms` script. This script is a good test to run, and places the arms in a reasonably-useful configuration. First, enable the robot with

```
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 you are working with Baxter. (When working with Sawyer, use `right_gripper_tip`.)

While you are echoing the `tf` transform, run tuck arms

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

You should see the transformation that `tf` returns changing as the arm moves. The `tf` package is an incredibly useful utility, which you will likely use quite often.

8 Using MoveIt

MoveIt is a useful path planning package that abstracts the interaction 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.

8.1 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. Begin the Gazebo simulation and enable the robot like you did in the last section, then run Baxter's joint trajectory controller with the command

```
rosrun baxter_interface joint_trajectory_action_server.py
```

Next, in a new window, start MoveIt with

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

The MoveIt GUI should appear with a model of the Baxter robot. In the Displays menu, check the "Query Goal State" box under "MotionPlanning" $\Longrightarrow$ "Planning Request" 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 in RViz.

When you've specified the desired states, switch to the "Planning" tab. Make sure that in the "Options" section that "Velocity Scaling" and "Acceleration Scaling" are both set to 1, then click "Plan". The planner will compute a motion plan, then display the plan as an animation in the window on the right. The Displays menu has some useful visualizing options that you can select under "Motion Planning" $\Longrightarrow$ "Planned Path", such as the "Show Trail" option which displays the complete path of the arm and the "Loop Animation" option which loops the visualization of 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 in Gazebo.

Note: *With a real robot, never use the "Plan and Execute" option in the MoveIt GUI. Always examine the path visually before hitting execute to ensure safety.*

Keep the Gazebo and the RViz windows open for the next section.

9 AR Tags

Figure 3: 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 3. 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 `tuw_marker_detection` [1] package.

9.1 AR Tag Tracking Setup

1. With the Gazebo and the RViz windows open from the last section, start up the AR tag tracker with

   ```
   roslaunch ar_tags ar_track.launch
   ```

   A new window should pop up that shows what the Baxter's right hand camera is seeing.

2. In the Gazebo window, navigate to the "Insert" tab. You should see a list of models that you can insert into the simulation. Pick one of the models named "aruco_visual_marker_{#}" (without "_pad") and place it in the view of the right hand camera so you can see the entire AR tag in the camera window from the previous step. You can use Gazebo's Translation Mode (the 4-arrow move symbol in the top left of the screen) to drag the AR tag in Gazebo and adjust its position.

3. In real life, AR tags can be finicky. If the AR tag is too close or too far from the camera, it may be difficult to calculate its pose. Adjust the AR tag's position until the tag in the camera window has a 3D box outline on it with a set of coordinate axes as shown in Figure 4. This means that the pose of the tag can be calculated.

4. In RViz, add a TF display. Disable all frames except "t{#}" (the number should be the same as the marker number you inserted into Gazebo). Verify visually as shown in Figure 5 that the transform axes capture the approximate position and orientation of the AR tag.

---

[1] https://github.com/tuw-robotics/tuw_marker_detection.git

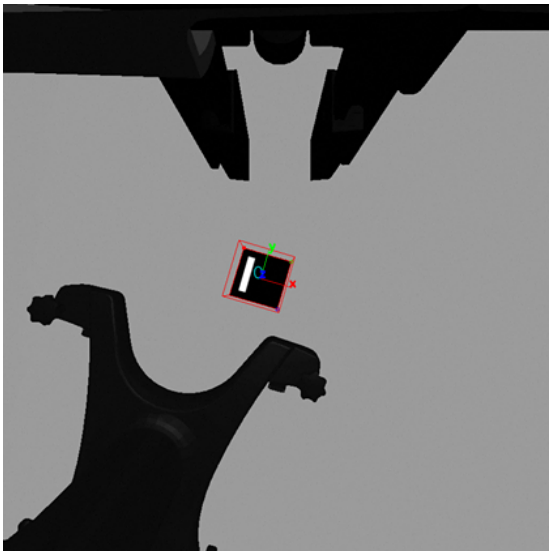

Figure 4: Example of successful detection of an AR tag

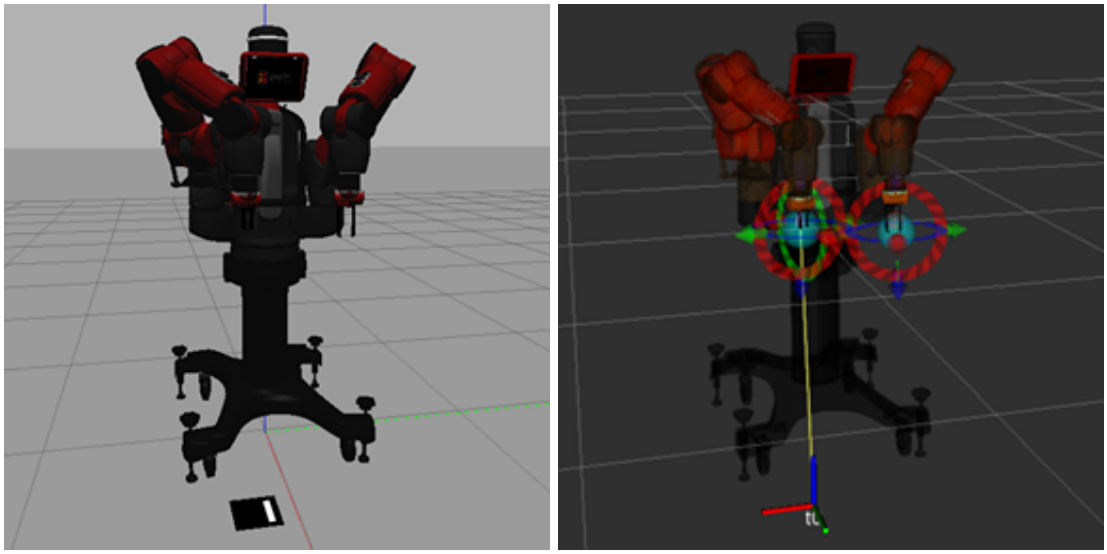

Figure 5: Visualization in Gazebo and RViz of AR tag relative to Baxter's right hand camera

Checkoff

Ask a TA for a checkoff for this lab. Be prepared to:

1. Demonstrate moving the Baxter arm using the MoveIt! GUI.
2. Show RViz with the transform for an AR tag.

---

10 Typo Reporting

If you notice any typos or things in this document or the starter code which you think we should change, please let us know. It's really easy to miss things, so please help us out and make the course better for the next generation of students.