Robotic Manipulation 2015

Project 1 (4752/4750: 100 points, 5752/5750: 130 points)
The Robot Operating System (ROS)

Due at the start of class, Friday, Oct. 2, 2015

Objective: To learn the fundamentals of ROS, to gain hands-on experience programming Baxter and the Baxter simulator, and to practice applying the concepts of rigid body transforms and inverse kinematics in ROS.

Instructions: This is a group exercise. Turn in to CMS one compressed file (zip or tarball) for each group of three students. include:

- your ROS package (be sure to do a "make clean" in your package before submission to minimize the size.
- a PDF with answers to the questions below, and
- a README file with instructions on how to run your code (both through the launch file and individual ROS commands).

Write the name of your group and the names of the members at the top of each of your files.

In addition, **you will demo your code to the instructors** at a time to be scheduled after the deadline.

Note on programming language: You may implement your code in any language supported by ROS. Python is recommended because it will save you development time and simplify your life. C++ and Matlab are also allowed. If you want to use something besides these, please discuss it with an instructor before you begin. Please refer to http://www.ros.org/ for language-specific naming conventions and compilation instructions.

Assignment: You are going to build a symbolic manipulation system in simulation and on Baxter. Your robot has one arm that is capable of picking up blocks. The blocks are numbered $1 \ldots n$, where n is a tunable parameter. The following block-stacking semantics apply. A block can be held by the robot gripper, it can be on the table, or it can be on top of exactly one other block. Likewise, a block can have at most one other block on top of it. Any number of blocks may rest directly on the table. A block can only be transported when the gripper is closed around it. The gripper must be opened in order to release the block onto the table or supporting block. It may be generally helpful to think of and refer to the table as a virtual "block 0" that cannot be grasped or moved.

I. (50 + 15 points) **Symbolic Simulator.**

(a) (5 points) Create a ROS package in which to organize your files.

```
$ cd ~/ros_ws/src
$ catkin_create_pkg <your_group_name>_proj1 std_msgs rospy roscpp
$ . ~/ros_ws/devel/setup.bash
$ roscd <package_name>
$ mkdir launch src msg srv
```

(b) (5 points) In this assignment, you will implement several ROS nodes. Build a launch file called proj1.launch to start all the nodes. Use XML syntax following the description at http://wiki.ros.org/roslaunch/XML to update the launch file as you complete the assignment.

Architecture: Your manipulation system will consist of two ROS nodes. The first node, `robot_interface`, will perform three functions:

a. maintain the current state of the robot and blocks,
b. accept commands to the robot, and
c. periodically report the state of the system.

Meanwhile, the second node, `controller`, will plan and execute sequences of symbolic actions to rearrange blocks.

(c) (5 points) Create message and service types (respectively, in the `msg/` and `srv/` directories inside your package). Then translate the following pseudocode into a service type.

```
boolean MoveRobot(enum Action, int Target)
```

You will create one message type, `State`, whose purpose is to provide the state of all symbolic objects and their relationships. Pick your own representation of the world state. Note that in order to compile the message and service, you will need to edit the file `CMakeLists.txt` and run `catkin_make` in the directory ∼`/ros_ws`.

(d) (15 points) Implement a server for performing symbolic manipulation actions. The server, named `robot_interface`, must perform two behaviors:

- Accept service requests on the service named `/move_robot`. If the action is valid, then change the world state appropriately and return success (`True`). Otherwise, return `False` without changing the world state. Valid actions are listed below.
- Periodically broadcast, at 1 Hz, the state of the world, using topic `/state`. In addition to the state Publisher, you may wish to also create a state Service type so that you don't have to wait up to a second. Doing so is optional, but it may be useful for Part II.

Valid actions for the robot are:

- open the gripper
- close the gripper
- move to a block, meaning the fingers are around a block.
- move over a block, such that if the gripper is holding a block, it is placed on top of the target block.

Consider the semantics of each of the operations depending on the current state of the world, and implement them appropriately (e.g. when the gripper is closed, the robot cannot "move to" a block).

When initializing the world state, the server must query two global ROS parameters:

- `/num_blocks` (integer): the number of blocks in the world
- `/configuration` (string): the initial configuration of the blocks. This must include at least `scattered`, `stacked_ascending`, and `stacked_descending`.

In the launch file, set the parameters and start a node running.

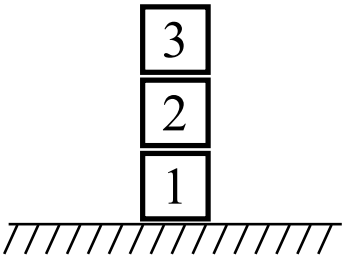

Figure 1: These blocks are stacked in an ascending sequence.

(e) (5 points) Test and debug your `robot_interface` server using the "rosservice", "rosnode", and "rostopic" commands. Give examples of how you used each command.

(f) (15 points) Implement a second node, `controller`, to control the robot. The controller's job, beginning from any arbitrary initial state, is to direct the robot to modify the state of the world in a particular way. Implement three modes: *scatter*, *stack_ascending*, and *stack_descending*. Use a ROS topic `/command`, of type `std_msgs/String`, to command the controller to execute a mode. Hint: you do not need to provide the optimal solution.

(g) (15 points – optional for 4000-level students) Write a bimanual (i.e. two-armed) version of the above two nodes. Use a ROS parameter to set the number of arms. Create a fourth mode for the `controller`, called *odd_even*, which simultaneously creates two stacks of blocks according to their parity (order can be ignored). Think carefully about the semantics of parallel manipulation with two arms before you begin. What is your strategy to control the two arms in parallel? Do you drive one arm per message or encode both arms' behaviors in a single command? What are the trade-offs, and how do they effect the performance of the robot?

II. (50 + 15 points) **Real Robot (Baxter).**

Baxter Simulator. Not to be confused with the Symbolic simulator in Part I. In order to test and debug your code before running on the real robot, you will need to run the simulator. **Test all robot code on the simulator first!** This assignment will introduce you to running the real robot as well as the simulator. The procedure is nearly identical. From the ∼`/ros_ws` directory, enter the simulator environment with

```
$ . baxter.sh sim
$ roslaunch baxter_gazebo baxter_world.launch
```

When your code works in simulation, exit the simulator environment, and enter the real robot's Baxter environment with

```
$ . baxter.sh
```

Baxter operation. Turn on the real robot by pushing the power button located at waist level in the back. The robot is fully booted when the colored ring at the top of its head glows green. At this point, it is ready to accept commands.

To get acquainted with programming the robot, there are many resources online:

- http://sdk.rethinkrobotics.com/wiki/Hello_Baxter
- http://sdk.rethinkrobotics.com/wiki/Example_Programs
- http://sdk.rethinkrobotics.com/wiki/Learning

Note that there are two ways to interface with the real robot and Gazebo simulator:

(a) Using ROS topics

(b) Using the Python-only Baxter API at http://api.rethinkrobotics.com/

Figure 2: Emergency stop button

Safety. When operating Baxter, safety is paramount. Any time the robot is in operation, one person should have the emergency stop in hand and ready to activate. The e-stop should be pressed any time Baxter is in danger of harming a person, an object, or the robot itself. When activated, the e-stop will cause Baxter's arms to loose power and they will slowly fall to rest on the table or at Baxter's sides.

Complete the following exercises. For details, see the Baxter tutorials.

(a) (10 points) Use a command-line switch or ROS parameter to control whether `robot_interface` commands the real robot or only runs a symbolic simulation.

To compute the pose of the fingers in Cartesian coordinates, you need to do forward kinematics. Rather than implementing it yourself, you may wish to use ROS's TF (transform) library: http://wiki.ros.org/tf2 or use the Python Baxter API.

Which technique did you pick? Does it directly give the position of the center of the fingertips? If not, think about how you can compute it. If using TF, you can visualize the TF tree using rviz ("rosrun rviz rviz") or view_frames ("rosrun tf view_frames").

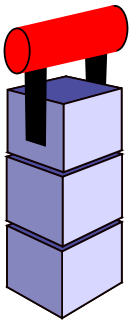

Figure 3: Initial grasp pose before running your code.

(b) (15 points) Implement a pick-and-place routine within `robot_interface` that accepts source and destination coordinates (Cartesian pose). You will need to call Baxter's inverse kinematics routine twice per invocation. Think about what "global" coordinate frame you are operating in.

(c) (10 points) Augment `robot_interface`'s world state representation with the Cartesian pose of each block. Initialize the block poses by querying the robot's initial configuration at runtime. The initial hand pose should be positioned as shown in Figure 3.

Note that you will need to stack the blocks manually (in any pose you wish) and place Baxter's hand around the blocks yourself so that it has the correct initial state. You may find the Gripper Cuff Control example useful for this task:
`http://sdk.rethinkrobotics.com/wiki/Gripper_Cuff_Control_Example`

Note that from the initial hand pose and the initial ROS parameters, "/num_blocks" and "/configuration", it is possible to infer the pose of every block. You do not need to implement an initial "scattered" state – only "stacked_ascending" and "stacked_descending".

(d) (15 points) Implement the control policy for the three commands:
stack_ascending, *stack_descending* and *scatter*.
The stacked ascending configuration is shown in Figure 1. A scattered configuration is one in which all blocks rest directly on the table.

In some circumstances, you will need to arbitrarily select destination poses for blocks. Any pose on the table is fine as long as the robot is later able to retrieve the block again when needed. Implement all the special cases for this.

(e) (15 points – optional for 4000-level students) Write a bimanual (two-armed) version of this assignment. You have the freedom to place the stacks where you like so that they are in the joint workspace of the two arms. Can you make the two-armed version run faster than the one-armed implementation?

III. (15 points – extra credit) Implement a solution to the Towers of Hanoi Problem for an arbitrary number of blocks. Points awarded at the discretion of the teaching staff.