CS 5742: Project 1
Answers to Project Questions

Taylor Clawson tsc83
Joseph Corbett-Davies jwc292
Spencer Steel shs257

October 2 2015

1 Symbolic Simulator

- Part E
  - **rosservice**
    We used rosservice to make sure that our move_robot was properly registered with the ROS master and that it could be called.
    Examples:
    - `rosservice list` - This checked that ROS knew about our service
    - `rosservice /call move_robot block_num target_block` - This command enabled us to test the service by passing it a block number to move to target block (this syntax has since changed to enable bimanual mode, but this was the original use)
  - **rosnode**
    We used rosnode to see that each node we ran was known by ROS master and that it was running when we wanted it running.
    Examples:
    - `rosnode list` - Listed all rosnodes that ROS master would know about.
    - `rosnode ping /robot_interface` - This command let us see if our node was actually up and running with roscore.
  - **rostopic**
    Rostopic was used to listen in on specific topics to make sure the correct things were happening in addition to issuing commands to baxter.
    Examples:
    - `rostopic echo /state` or `/rosout` etc. - We use this command often to follow state and make sure the correct things are happening both symbolically and when we used the baxter simulator and baxter.
    - `rostopic pub /command std_msgs/String "<command>"` - This is a very important use of the command. After initializing our robot interface's state we use this command to change state.

- Part G
  In order to create a bimanual version we realized we needed to change parts of the state message to include the location of the two grippers (which blocks each one would work with) and whether or not each gripper was open and closed. After making changes to state our strategy was to change the `MoveRobot` service to encode both left and right arm actions in a single message (where ideally the `robot_interface` node could command both arms and allow for faster execution of many action combinations) however our current solution is to move one arm after the other—the non-active arm is given a null action.

  To try and inherently avoid conflict, the right arm is used to manipulate odd numbered blocks (which are preferentially located to the right of the workspace, according to our symbolic conventions and control policies) and vice-versa for the left arm. As we found, encoding a single arm's motion per service call (as we essentially did in our final `controller` node) allows for much simpler generation of control policies as they do not have try to include simultaneous actions in both arms.

2 Real Robot

- Part A
  To compute the pose of Baxter's arms we decided to use the Python API for Baxter. After creating a limb object (from the API) we could call the function endpoint_pose() which returns a point and an orientation of Baxter's gripper. From this information we had the position and orientation of some point fixed in the gripper frame. We based all other block locations and movement commands off of this point and the measured block height.

- Part E
  As discussed in part 1 section g, we did implement a bimanual mode, however we do not control the arms in parallel. One arm moves and once it is finished the other arm may move. This does not allow Baxter to finish the task any faster than single armed version. In fact it may slow him down a bit depending on the configuration.
  Obviously if we got the arms to work in parallel we would potentially be able to increase the speed at which Baxter performs these tasks, but at the moment bimanual movement is not faster.