LAB 3

The Tower of Hanoi with ROS

3.1 Important

Read the entire lab before starting and especially the “Grading” section so you are aware of all due dates and requirements associated with the lab. Hopefully you are reading this well before your lab section meets as given the compressed schedule, it is very important that you arrive at lab well prepared. This semester, the more you do prior to your lab session, the more you will get out of the short time you have with the TA.

3.2 Objectives

This lab is an introduction to controlling the UR3 robot using the Robot Operating System (ROS) and the Python programming language. In this lab, you will:

- Modify the given starter python file to move the robot to waypoints and enable and disable the suction cup gripper such that the blocks are moved in the correct pattern.
- If the robot suction senses that a block is not in the gripper when it should be, the program should halt with an error.
- Program the robot to solve the Tower of Hanoi problem allowing the user to select any of three starting positions and ending positions.

3.3 Pre-Lab

Read "*A Gentle Introduction to ROS*", available online, Specifically:

- Chapter 2: 2.4 Packages, 2.5 The Master, 2.6 Nodes, 2.7.2 Messages and message types.
- Chapter 3 Writing ROS programs.

3.4 References

- Consult Appendix A of this lab manual for details of ROS and Python functions used to control the UR3.
- "*A Gentle Introduction to ROS*", Chapter 2 and 3. `http://coecsl.ece.illinois.edu/ece470/agitr-letter.pdf`
- `http://wiki.ros.org/`
- Since this is a robotics lab and not a course in computer science or discrete math, feel free to Google for solutions to the Tower of Hanoi problem.[1] You are not required to implement a recursive solution.

[1] `http://www.cut-the-knot.org/recurrence/hanoi.shtml` (an active site, as of this writing.)

3.5 Task

3.5.1 Standard Tower Of Hanoi

As you hopefully know by now, the normal way that the Tower of Hanoi puzzle is set up is as a stack of discs or blocks as seen in Figure 3.1. This makes many of the rules obvious such as that you cannot move a lower block while a higher block still rests on it. This arrangement is hard to simulate in Gazebo (The simulator we will be using). Stacked blocks do not behave in a stable manner and thus it is difficult to create neat stacks without disturbing/toppling them. As an alternative, we have modified the puzzle to work with the simulator.

3.5.2 Simulated Tower Of Hanoi

In our version of Tower of Hanoi, we have laid the blocks flat on the table as seen in Figure 3.2. Now, instead of rising vertically from the table, the blocks "rise" as they move farther from the viewer. Thus the "towers" form the columns of a grid, while the "height" depends on the rows of the grid. We will make use of this matrix like structure to organize our waypoints in the code.
Color has been used to help keep track of the blocks in the simulator:

1. Red - Top Block
2. Yellow - Middle Block
3. Green - Bottom Block

This "2D" version doesn't have gravity to enforce building from "bottom" to "top", but you should still code it as if it has gravity (i.e. no floating blocks). You may not place a block on top of a lower-numbered block, as illustrated in Figure 3.3. (For example, no green block on top of red or yellow blocks.) An example of a legal move can be seen in Figure 3.4.
In addition, unlike an actual vacuum gripper, the vacuum gripper in the simulator has to go to the center of an object in order to actually grip it. Due to this difference, you will be given two files that record the locations of towers corresponding to the actual lab environment and the simulator environment.
For this lab, we will complicate the task slightly. Instead of a set start and end position, your python program should use the robot to move a tower from *any* of the three locations to *any* of the other two locations. Therefore, you should prompt the user to specify the start and destination locations for the tower.
Additionally, you will make use of suction feedback to verify that you have grasped the desired block successfully. If a block is missing, the robot should stop the puzzle, shut off the gripper and return to its home position.

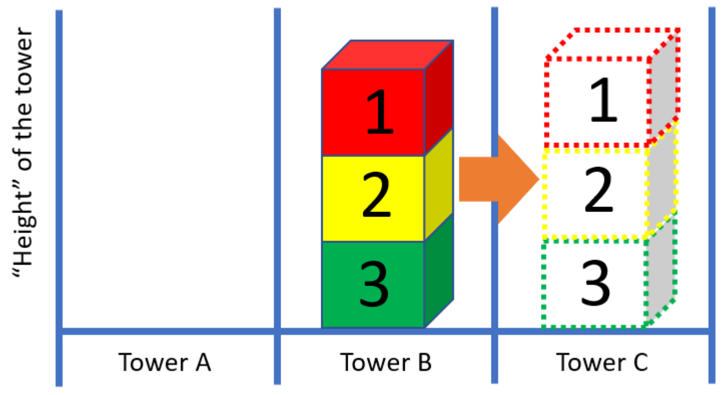

Figure 3.1: Standard Tower of Hanoi configuration with blocks stacked on top of each other.

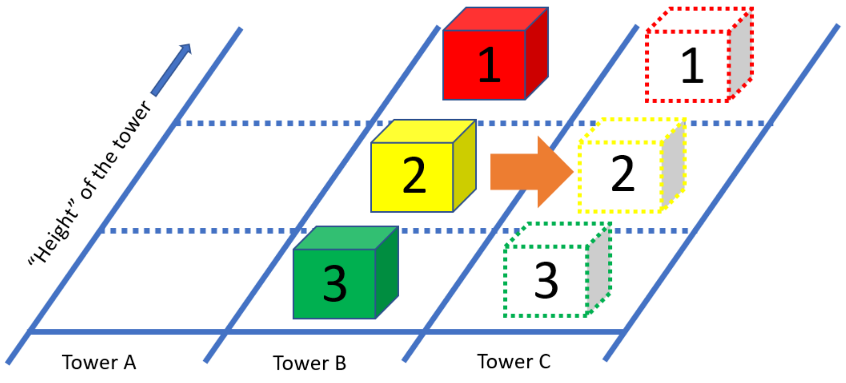

Figure 3.2: Example start and finish tower locations in the simulated version of the puzzle.

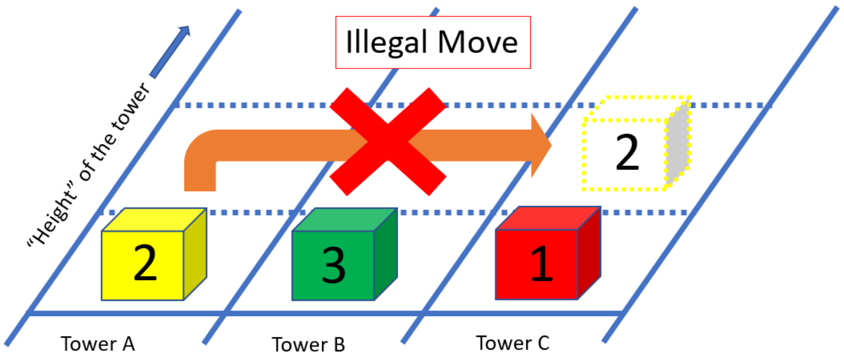

Figure 3.3: Example of an illegal move.

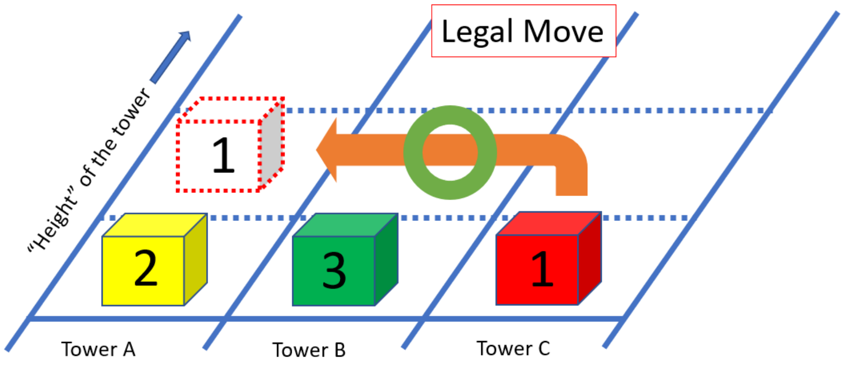

Figure 3.4: Example of a legal move.

3.6 Procedure

1. Create your own workspace as shown in Appendix A.

2. Copy package **lab3pkg_hanoi** into your workspace from **code_student**. All your coding work will be done in the script **lab3_exec.py** with comments. Refer to **README.md** on how to run your code and robot!

   - **lab3_exec.py** a file in scripts folder with skeleton code to get you started on this lab. See Appendix A for how to use basic ROS. Students are encouraged to make their own "cheat sheet" for some commonly used ROS and Linux commands. Also read carefully through the below section that takes you line by line through the starter code.
   - **CMakeLists.txt** a file that sets up the necessary libraries and environment for compiling lab3_exec.py.
   - **package.xml** This file defines properties about the package including package dependencies.
   - **README** To run lab3 code on real robot, please read it **FIRST**!
   - Every time you open a new terminal, make sure to source your workspace by running **source devel/setup.bash** in your workspace folder.

3. Modify **lab3_exec.py** to prompt the user for the start and destination tower locations (you may assume that the user will not choose the same location twice) and move the blocks accordingly using the suction cup to grip the blocks. The starter file performs basic motions but provides a function definition for moving blocks (**move_block**). Once you understand the starter code moving from one position to the next, clean up the code by completing the shell function **move_block**. **move_block** picks up a block from a tower and places it on another tower. You may also create other functions for prompting user input and solving the Tower of Hanoi problem given starting and ending locations but these are not required.

4. Add one more feature to your program. As you saw in Lab 1.5, the Coval device that is creating the vacuum for the gripper also senses the level of suction being produced indicating if an item is in the gripper. Recall that **Digital Input 0** are connected to this feedback. Use **Digital Input 0** to determine if a block is held by the gripper. If no block is found where a block should be, have your program exit, turn off the gripper, return home and print an error to the console.
   To figure out how to do this with ROS you are going to have to do a bit of "ROS" investigation. Use "**rostopic list**", "**rostopic info**" and "**rosmsg list**" to discover what topic to subscribe and what message will be recieved in your subscribe callback function. Once you find the topic and message run "**rosmsg info**" to figure out what variable you will need

to read from the message sent to your callback function. Just like the global variables thetas that save the positions of the robot joints inside the call back **position_callback()**, create global variables to communicate to your code the state of **Digital Input 0**. Normally once you figure out which message you will be using you need to import it in the **lab3_header** file that defines this message. The **lab3_header** file has already imported it in **lab3_header.py** for you. Use the explanation below and the given code in **lab3_exec.py** that creates the subscription to **/joint_states** and its callback function as a guide to subscribe to the rostopic that publishes the IO status.

3.7 Lab3_exec.py Explained

First open up **lab3_exec.py** and read through the code and its comments as this is the latest version of Lab 2's starter code. Below is the same **lab3_exec.py** file listing with code comments removed and possible small differences due to changes in the lab. If you find a difference go with the actual **lab3_exec.py** file as the correct version. **lab3_exec.py** is broken down into sections and described in more detail below.

```
import copy
import time
import rospy
import numpy as np
from lab3_header import *

# 20Hz
SPIN_RATE = 20

# UR3 home location
home = np.radians([-87.42, -103.45, -72.77, -79.02, 89.95, 129.93])

# UR3 current position, using home position for initialization
current_position = copy.deepcopy(home)

thetas = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

digital_in_0 = 0
analog_in_0 = 0

suction_on = True
suction_off = False

current_io_0 = False
current_position_set = False

Q = None
```

You can find **lab3_header.py** in the **lab3pkg_py/scripts** directory. It includes all needed files to allow **lab3_exec.py** to call ROS functionality. **SPIN_RATE** will be used as the publish rate to send commands to the ROS driver. This block

also initializes positions such as the home position and a global variable to store the current position. There are also some other global variables for storing the input/feedback states and also some constants. You should use these global variables and constants in different functions to help you finish the task.
Q is a list that stores all the necessary waypoints for the robot to pick and place the blocks in order to solve the Tower of Hanoi. In each entry of Q[tower index][block height][above block/on block], there are six angles in radians that correspond to the arm's six joint angles. Q is defined as illustrated below:

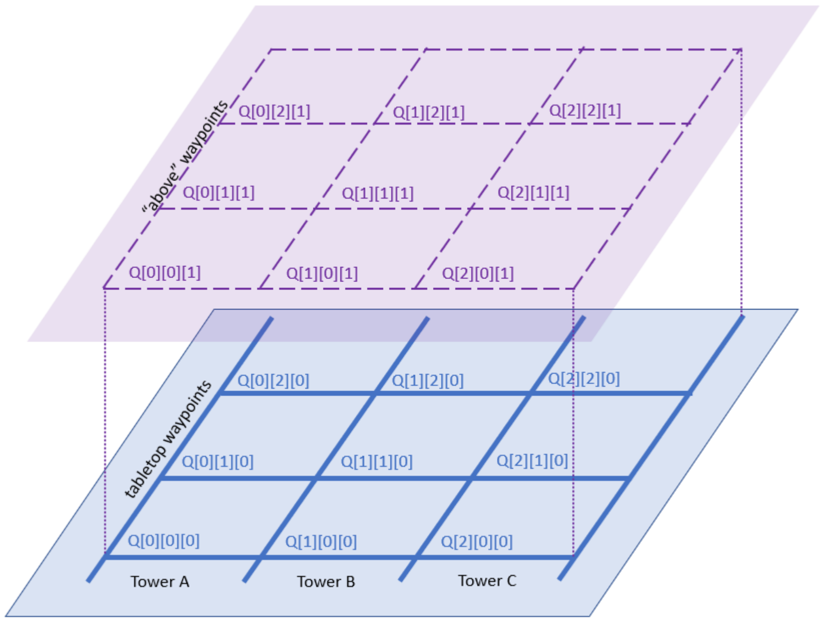

Figure 3.5: Waypoints Matrix Q

```
def position_callback(msg):

    global thetas
    global current_position
    global current_position_set

    thetas[0] = msg.position[0]
    thetas[1] = msg.position[1]
    thetas[2] = msg.position[2]
    thetas[3] = msg.position[3]
    thetas[4] = msg.position[4]
    thetas[5] = msg.position[5]

    current_position[0] = thetas[0]
    current_position[1] = thetas[1]
    current_position[2] = thetas[2]
    current_position[3] = thetas[3]
    current_position[4] = thetas[4]
    current_position[5] = thetas[5]

    current_position_set = True
```

This is **lab3_node**'s callback function that is called when the **ur_hardware_interface** publishes new position data. **ur_hardware_interface** publishes new angle position data every 8ms, so this function **position_callback** is run every 8ms. Next in **lab3_exec.py** are the function **gripper()** and **move_arm()**. These functions are passed variables that are initialized at the beginning of the file. The program runs from the main function, so it will be explained first and then we will come back to **gripper()** and **move_arm()**.

```
def main():
    global home
    global Q
    global SPIN_RATE

    # Initialize ROS node
    rospy.init_node('lab3_node')

    # Initialize publisher for ur3e_driver_ece470/setjoint with buffer size of 10
    pub_setjoint = rospy.Publisher('ur3e_driver_ece470/setjoint',JointTrajectory,queue_size=10)

    # Initialize subscriber to /joint_states and callback fuction
    sub_position = rospy.Subscriber('/joint_states', JointState, position_callback)
```

To start as a ROS node the **rospy.init_node()** function needs to be called. Then the node needs to setup which other nodes it receives data from and which nodes it sends data to. This code first specifies that it will be publishing a message to the "**ur3e_driver_ece470**" node "**setjoint**" subscriber. The message it will be sending is the **command** message which consists of the desired robot joint angles, the velocity of each joint and the acceleration of each joints. Next **lab3_node** subscribes to "**ur_hardware_interface**" node "**joint_states**" publisher. Whenever new joint angles are ready to be sent, the callback function "**position_callback**" is called and passed the message **position** which contains the six joint angles. As an exercise in lab, see if you can list the "**ur3e_driver_ece470**", "**ur_hardware_interface**" node and "**ur3e_driver_ece470/setjoint**" subscriber and "**joint_states**" publisher using the "**rostopic list**" command in your **catkin_ work directory**. Also use "**rostopic info**" to double check that "**ur3e_driver_ece470/setjoint**" is a subscriber and "**joint_states**" is a publisher. Also run "**rosmsg list**" to find the messages.

```
    input_done = 0
    loop_count = 0

    while(not input_done):
        input_string = raw_input("Enter number of loops <Either 1 2 3 or 0 to quit> ")
        print("You entered " + input_string + "\n")
        if(int(input_string) == 1):
            input_done = 1
            loop_count = 1
        elif (int(input_string) == 2):
            input_done = 1
            loop_count = 2
        elif (int(input_string) == 3):
            input_done = 1
            loop_count = 3
        elif (int(input_string) == 0):
            print("Quitting... ")
            sys.exit()
        else:
            print("Please just enter the character 1 2 3 or 0 to quit \n\n")
```

This standard python code printing messages to the command prompt and receiving text input from the command prompt. It loops until the correct data is input.

```
    # Check if ROS is ready for operation
    while(rospy.is_shutdown()):
        print("ROS is shutdown!")

    rospy.loginfo("Sending Goals ...")

    loop_rate = rospy.Rate(SPIN_RATE)
```

Here the code waits for **roscore** to be executed and ready. **rospy.loginfo** prints a message to the command prompt. **rospy.Rate(SPIN_RATE)** sets up a class **loop_rate** that can be used to sleep the calling process. The amount of time that the process will sleep is determined by the **SPIN_RATE** parameter. In our case this is set to 20Hz or 50ms. **loop_rate** does not wake up 50 ms after it has been called, instead it wakes up the process every 50ms. **loop_rate** keeps track of the last time it was called to determine how long to sleep the process to keep a consistent rate.

```
    move_block(pub_setjoint, pub_setio, start, 0, des, 2)
```

This moves the arm to a number of positions to give you a start at how to program the robot to move to different positions. See the move_arm and gripper function definitions below.

```
    def move_arm(pub_setjoint, dest):
    msg = JointTrajectory()
    msg.joint_names = ["elbow_joint", "shoulder_lift_joint",
       "shoulder_pan_joint","wrist_1_joint", "wrist_2_joint", "wrist_3_joint"]
    point = JointTrajectoryPoint()
    point.positions = dest
    point.time_from_start = rospy.Duration(2)
    msg.points.append(point)
    pub_setjoint.publish(msg)
    time.sleep(2.5)
```

The **move_arm()** function takes as parameters **pub_setjoint**, which is the publisher to **ur3e_driver_ece470** commanding a new position for the robot to move to. **dest** is the six joint angle destinations, in radians, that the robot will be commanded to move to. The code creates a variable **msg** which is the command message to be sent **ur3e_driver_ece470**. **msg** is assigned the **joint_names**, **positions** and **time_from_start**. Next the **msg** is published to **ur3e_driver_ece470** with the **pub_setjoint.publish(msg)** instruction. **time.sleep(2.5)** will sleep 2.5s waiting for the robot getting to the commanded position.

```
def move_block(pub_setjoint, pub_setio, start_loc, start_height, end_loc, end_height):
    global Q
```

The **move_block()** function definition is provided and should be used to complete the assignment. Functions are useful when the same procedure is used many times. To move a block, multiple arm movements are necessary along with gripper actuation. Instead of cluttering the main with many calls to **move_arm** and **gripper()**, you will compartmentalize the calls in the **move_block** function. Use this function to compartmentalize moving a block from one tower to another. The start and end locations are integers given to tower positions and the heights are integers for blocks in the stack.

3.8 Report

Each student will submit a lab report using the guidelines given in the "ECE 470: How to Write a Lab Report" document. Please be aware of the following:

- Lab reports will be submitted online at BlackBoard.
- The report will be due **Two** weeks after your lab session for Lab 2. Exact times and dates can be seen on BlackBoard.

Your report should include the following:

- Briefly explain the objective of the lab i.e. the goal and rules of Tower of Hanoi. Images would greatly aid in this explanation.
- What was the focus of this lab? (Hint: ROS and implementing feedback)
- With that in mind you should cover the following (in detail):
  - What is ROS and how does it work? (What kind of figure would help explain this?)
  - How did you use the ROS commands (i.e. **rostopic list**, **rostopic info**, etc.) to complete your task?
  - How did you implement feedback?
- Make use of code snippets as needed to aid in your explanation

- **Read "ECE 470: How to Write a Lab Report" carefully so you know all the requirements.**
- Unless your TA gives other guidance, include your **lab3_exec.py** code as an Appendix to your report as described in lab report guidelines.

3.9 Demo

Your TA will require you to run your program (at least) twice; on each run, the TA will specify a different set of start and destination locations for the tower. They will also test that suction feedback has been implemented correctly.

3.10 Grading

- 80 points, successful demonstration.
- 20 points, report.