dECE 470: Lab2 - Tower of Hanoi

Ashank Behara

abehara2

TA: Weihang (Eric) Liang

Section: Thursday 3:00 PM

Submitted: October 18, 2020

Objective

The objective of this lab was to get a 6-axis UR3 Robot to successfully complete the Tower of Hanoi puzzle. The objective of this puzzle is to move a stack of blocks, decreasing in size, from its starting position to one of two other positions on the board. No other positions except for the 3 available cannot be used and blocks must be stacked such that every block is bigger than the one under it in a stack. In the case of our experiment, green is the “biggest” block, yellow is the “middle” block, and red is the “smallest”. *Figure 1* shows a Tower of Hanoi problem where the aim is to move the blocks from the peg on the right to the peg on the left. The second and third pictures are valid intermediate steps of the solution. Notice how only the three available pegs are being used and the stacks always go from bottom to top, large to small.

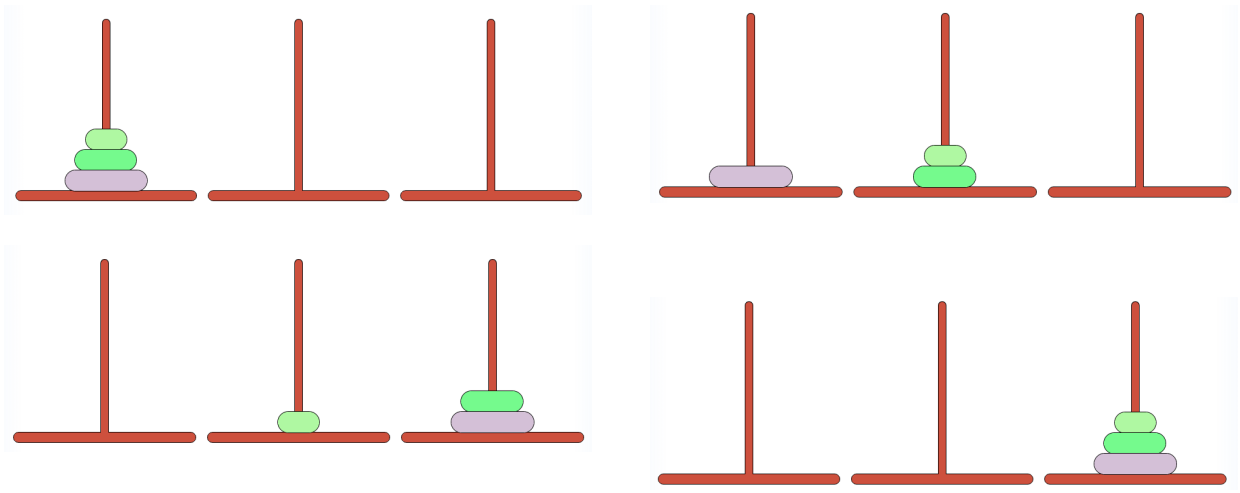

Figure 1

Given three rings, our blocks in the scope of this lab, and three positions, only seven moves are needed to solve this problem for all situations. In this image, blocks are stacked vertically, but in the following demo, the vertical direction is along the red axis. Here is a visualization of a correct solution to the Tower of Hanoi.

Demo: https://github.com/abehara2/Robotics/tree/master/UR3_advanced_control

Method

Apart from solving the Tower of Hanoi problem, the goal was to build a foundation of ROS and using it to interface with other sensors and controller inputs.

ROS is a middleware that enables the communication between hardware and software components of an integrated system and is more or less a distributed system. ROS nodes are deployed in different areas across the system and operate independently by default. These nodes do not stay independent and often communicate with their respective hardware components and are then allowed to send this information amongst any other node. A master node facilitates and initializes this communication process. Nodes can also stream and pass information between other nodes as well. ROS utilizes topics for this communication. Think of a topic as a water well where people and animals can take and put water into it. Nodes can "publish" or send information to this topic or "subscribe" and receive information. An important thing to note is that nodes can be both publishers and subscribers and can publish and subscribe multiple topics at a time. ROS topics have a message attached to them also that can be viewed to see what information the ROS topic holds. To see all the topics we use `rostopic list` and `rostopic info` can be used to 'print specific information about the selected ROS topic. A good way to represent the overall structure of ROS is as a directed graph.

One of the focuses of this lab was to implement feedback for a gripper end effector. To see the information the topic was sending, I first used `rostopic list` to find the topic that the gripper was publishing to and used `rostopic echo /ur3/gripper_input` to see the message being returned. The message returned an object with three values: DIGIN, AIN0, and AIN1. The value I used to implement the feedback was DIGIN as the values were easier to interpret. DIGIN = 1 meant a block was picked up and DIGIN = 0 meant that a block was not picked up.

Conclusion

This lab was a great learning experience. Prior to this lab, I felt that I was comfortable with the topics introduced in this lab, but did not know how to implement feedback. After seeing how the the other callback function was written and figuring out how to display the message in terminal, this turned out to not be an issue. Another interesting thing about my experience was that I had never heard of the Tower of Hanoi. Playing the game itself, which is linked in the references section, helped immensely when coming up with a viable algorithm to solve the Tower of Hanoi for any given configuration.

References

Tower of Hanoi. (n.d.). Retrieved October 18, 2020, from https://www.mathsisfun.com/games/towerofhanoi.html

Appendix

```
#!/usr/bin/env python

'''
We get inspirations of Tower of Hanoi algorithm from the website below.
This is also on the lab manual.
Source: https://www.cut-the-knot.org/recurrence/hanoi.shtml
'''

import os
import argparse
```

```python
import copy
import time
import rospy
import rospkg
import numpy as np
import yaml
import sys
from lab2_header import *

# 20Hz
SPIN_RATE = 20

# UR3 home location
home = np.radians([120, -90, 90, -90, -90, 0])

# 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

"""
TODO: define a ROS topic callback funtion for getting the state of suction cup
Whenever ur3/gripper_input publishes info this callback function is called.
"""

def gripper_callback(msg):

    global digital_in_0
    global analog_in_0

    digital_in_0 = msg.DIGIN
    analog_in_0 = msg.AIN0

"""
Whenever ur3/position publishes info, this callback function is called.
"""
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


def gripper(pub_cmd, loop_rate, io_0):

    global SPIN_RATE
    global thetas
    global current_io_0
    global current_position

    error = 0
    spin_count = 0
    at_goal = 0

    current_io_0 = io_0

    driver_msg = command()
    driver_msg.destination = current_position
    driver_msg.v = 1.0
    driver_msg.a = 1.0
    driver_msg.io_0 = io_0
    pub_cmd.publish(driver_msg)

    while(at_goal == 0):
        if( abs(thetas[0]-driver_msg.destination[0]) < 0.0005 and \
            abs(thetas[1]-driver_msg.destination[1]) < 0.0005 and \
            abs(thetas[2]-driver_msg.destination[2]) < 0.0005 and \
            abs(thetas[3]-driver_msg.destination[3]) < 0.0005 and \
            abs(thetas[4]-driver_msg.destination[4]) < 0.0005 and \
            abs(thetas[5]-driver_msg.destination[5]) < 0.0005 ):

            at_goal = 1
        if (spin_count > SPIN_RATE*5):
            pub_cmd.publish(driver_msg)
            rospy.loginfo("Just published again")
            spin_count = 0

        spin_count = spin_count + 1

    return error


def move_arm(pub_cmd, loop_rate, dest, vel, accel):
```

```
    global thetas
    global SPIN_RATE

    error = 0
    spin_count = 0
    at_goal = 0

    driver_msg = command()
    driver_msg.destination = dest
    driver_msg.v = vel
    driver_msg.a = accel
    driver_msg.io_0 = current_io_0
    pub_cmd.publish(driver_msg)

    loop_rate.sleep()

    while(at_goal == 0):

        if( abs(thetas[0]-driver_msg.destination[0]) < 0.0005 and \
            abs(thetas[1]-driver_msg.destination[1]) < 0.0005 and \
            abs(thetas[2]-driver_msg.destination[2]) < 0.0005 and \
            abs(thetas[3]-driver_msg.destination[3]) < 0.0005 and \
            abs(thetas[4]-driver_msg.destination[4]) < 0.0005 and \
            abs(thetas[5]-driver_msg.destination[5]) < 0.0005 ):

            at_goal = 1
            #rospy.loginfo("Goal is reached!")

        loop_rate.sleep()

        if(spin_count >  SPIN_RATE*5):

            pub_cmd.publish(driver_msg)
            rospy.loginfo("Just published again driver_msg")
            spin_count = 0

        spin_count = spin_count + 1

    return error

def move_block(pub_cmd, loop_rate, start_loc, start_height, \
               end_loc, end_height):
    global Q

    ### Hint: Use the Q array to map out your towers by location and "height".

    move_arm(pub_cmd, loop_rate, Q[start_loc][start_height][1], 4.0,4.0) #move to sta
    move_arm(pub_cmd, loop_rate, Q[start_loc][start_height][0], 4.0,4.0) #move down to
    gripper(pub_cmd, loop_rate, suction_on) #enable gripper
    time.sleep(0.5)
    if (digital_in_0 == 0):
        print("Invalid Tower of Hanoi... Missing block :(")
        move_arm(pub_cmd, loop_rate, home, 4.0,4.0) # move up from end
        sys.exit()
```

```
    move_arm(pub_cmd, loop_rate, Q[start_loc][start_height][1], 4.0,4.0) #move up fro
    move_arm(pub_cmd, loop_rate, Q[end_loc][end_height][1], 4.0,4.0) #move across fro
    move_arm(pub_cmd, loop_rate, Q[end_loc][end_height][0], 4.0,4.0) #move down from e
    gripper(pub_cmd, loop_rate, suction_off) #disable gripper
    move_arm(pub_cmd, loop_rate, Q[end_loc][end_height][1], 4.0,4.0) # move up from en

    error = 0
    return error

def main():

    global home
    global Q
    global SPIN_RATE

    # Parser
    parser = argparse.ArgumentParser(description='Please specify if using simulator or
    parser.add_argument('--simulator', type=str, default='True')
    args = parser.parse_args()

    # Initialize rospack
    rospack = rospkg.RosPack()
    # Get path to yaml
    lab2_path = rospack.get_path('lab2pkg_py')
    yamlpath = os.path.join(lab2_path, 'scripts', 'lab2_data.yaml')

    with open(yamlpath, 'r') as f:
        try:
            # Load the data as a dict
            data = yaml.load(f)
            if args.simulator.lower() == 'true':
                Q = data['sim_pos']
            elif args.simulator.lower() == 'false':
                Q = data['real_pos']
            else:
                print("Invalid simulator argument, enter True or False")
                sys.exit()

        except:
            print("YAML not found")
            sys.exit()

    # Initialize ROS node
    rospy.init_node('lab2node')

    # Initialize publisher for ur3/command with buffer size of 10
    pub_command = rospy.Publisher('ur3/command', command, queue_size=10)

    # Initialize subscriber to ur3/position and callback fuction
    # each time data is published
    sub_position = rospy.Subscriber('ur3/position', position, position_callback)

    ############## Your Code Start Here ##############
    # TODO: define a ROS subscriber for ur3/gripper_input message and corresponding ca
```

```
    sub_gripper = rospy.Subscriber('ur3/gripper_input', gripper_input, gripper_callbac

    input_done = 0
    loop_count = 0
    start = 0
    destination = 0

    while(not input_done):
        input_start = raw_input("Enter spawn column <Either 1 2 3 or 0 to quit> ")
        print("You entered " + input_start + "\n")

        if(int(input_start) == 1):
            input_done = 1
            start = 1
        elif (int(input_start) == 2):
            input_done = 1
            start = 2
        elif (int(input_start) == 3):
            input_done = 1
            start = 3
        elif (int(input_start) == 0):
            print("Quitting... ")
            sys.exit()
        else:
            print("Please just enter the character 1 2 3 or 0 to quit \n\n")

        input_destination = raw_input("Enter destination column <Either 1 2 3 or 0 to
        print("You entered " + input_destination + "\n")

        if(int(input_destination) == 1):
            input_done = 1
            destination = 1
        elif (int(input_destination) == 2):
            input_done = 1
            destination = 2
        elif (int(input_destination) == 3):
            input_done = 1
            destination = 3
        elif (int(input_destination) == 0):
            print("Quitting... ")
            sys.exit()
        else:
            print("Please just enter the character 1 2 3 or 0 to quit \n\n")

    while(rospy.is_shutdown()):
        print("ROS is shutdown!")
    rospy.loginfo("Sending Goals ...")
    loop_rate = rospy.Rate(SPIN_RATE)
    goal = destination
    initial = start
    auxillary = 0

    if (start == destination):
        print("Tower of Hanoi Solved!")
        sys.exit()
```

```
    if ((goal == 1 and initial == 2) or (goal == 2 and initial == 1)):
        auxillary = 3

    if ((goal == 1 and initial == 3) or (goal == 3 and initial == 1)):
        auxillary = 2

    if ((goal == 3 and initial == 2) or (goal == 2 and initial == 3)):
        auxillary = 2

    move_block(pub_command, loop_rate, initial - 1, 2, destination - 1, 0)
    move_block(pub_command, loop_rate, initial - 1, 1, auxillary - 1, 0)
    move_block(pub_command, loop_rate, destination - 1, 0, auxillary - 1, 1)
    move_block(pub_command, loop_rate, start - 1, 0, destination - 1, 0)
    move_block(pub_command, loop_rate, auxillary - 1, 1, start - 1, 0)
    move_block(pub_command, loop_rate, auxillary - 1, 0, destination - 1, 1)
    move_block(pub_command, loop_rate, start - 1, 0, destination - 1, 2)
    move_arm(pub_cmd, loop_rate, home, 4.0,4.0) # move up from end

    print("Tower of Hanoi Solved!")

if __name__ == '__main__':

    try:
        main()
    # When Ctrl+C is executed, it catches the exception
    except rospy.ROSInterruptException:
        pass
```