ECE 470

Fall 2021

Experiment #2

The Tower of Hanoi with ROS

Code for Lab is in Appendix

Ralph Balita (rbalita2)
Mehmet Alp Kara(kara3)
Chun-Kai Yao(ckyao2)
TA: Tao, Chuyuan
Lab Section: Tuesday 9AM
Date Submitted: Oct 15, 2021

1. Introduction

Explain the background of the lab. Briefly explain the objective of the lab i.e. the goal and rules of ToH. Images would greatly aid in this explanation.

The Towers of Hanoi (ToH). is a centuries old puzzle usually played with disks and rods, but all that was present for this lab were distinctly colored bricks. The objective of the ToH is to move a stack of disks from one rod to another, or in this case, move a stack of bricks from one location to another. In the rules, the player may only move a smaller disk onto a larger disk, but never move a larger disk onto a smaller disk. In this case, the robot can only move the RED onto the YELLOW, and the YELLOW onto the GREEN. Another rule claims that the disks are only allowed to be stacked in three locations. With the restrictions of this lab, the bricks may only touch the table in three locations. Only one brick can be moved at a time.

Optimal Number of Moves. It is claimed that the amount of optimal moves, labeled *m*, needed to move a stack of *n* bricks from one location to another has been mathematically calculated to by $m = 2^n - 1$. In this lab, the robot is assigned to move a stack of 3 bricks. This means that the minimal number of movements, *m*, should ideally be 7.

Objective. The goal of this lab is to generate a program that will communicate back and forth with the UR3 to solve this particular puzzle. The user will be able to input the starting and final positions of the towers, and the robot will be expected to solve the puzzle with the help of callback functions.

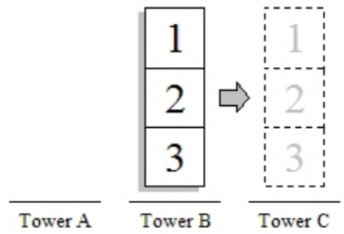
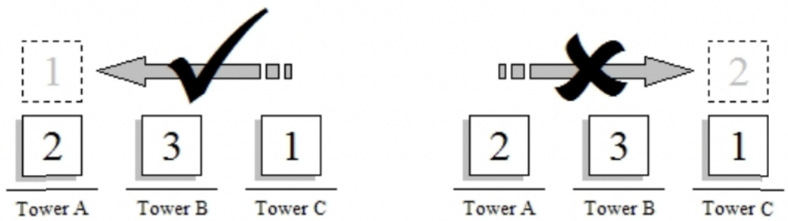


Figure 2.1: Example start and finish tower locations.



What was the focus of this lab? (Hint: ROS and implementing feedback)

Lab 1 introduced the concept of ToH, while Lab 2 introduced the implementation of the ROS and Python language to program the robot to complete the puzzle. The focus of this lab is to write a program using ROS and Python to mimic the maneuvers of Lab 1, and the following are the major components that are introduced:

ROS in Python Programming. This lab introduces several components of ROS, including robot-to-program communication, robot-to-user communication, Linux/ROS commands, creating a workspace, and running a node, which is explained in the following sections.

Implementing Feedback. An important part of programming a robot is implementing sensor feedback and user error feedback. In many ways, implementing feedback can provide the robot a bridge to communicate with the program and the program a bridge to communicate with the user. Whether it be by if-print statements or by rospy.loginfo(), implementing feedback provides an interface between what the robots’s state is and what the user expects the robot to do. This topic goes more in depth in the following sections.

2. Method

Explain how you did it

Lab 1 introduced the ToH without the use of Robotic Operating System(ROS) and Python Programming. Our group recorded the joint angles that would position the robotic arm into the 10 waypoints used for the ToH. Each of these positions can be labeled using a 3x3 array positions [x] [y], given that x = 0 and y = 0 would locate the position of the robotic arm to the bottom left brick waypoint, while x = 2 and y = 2 would locate the position to the top right waypoint. Another position, labeled as *central* and located above the middle tower, was used as a position for the robot to move between picking up a brick and placing it into a different position while avoiding collision.

For Lab 2, our group implemented the following algorithm in the main() function to allow the user to input any set of 2 locations (starting and final) as long as they are not the same location.

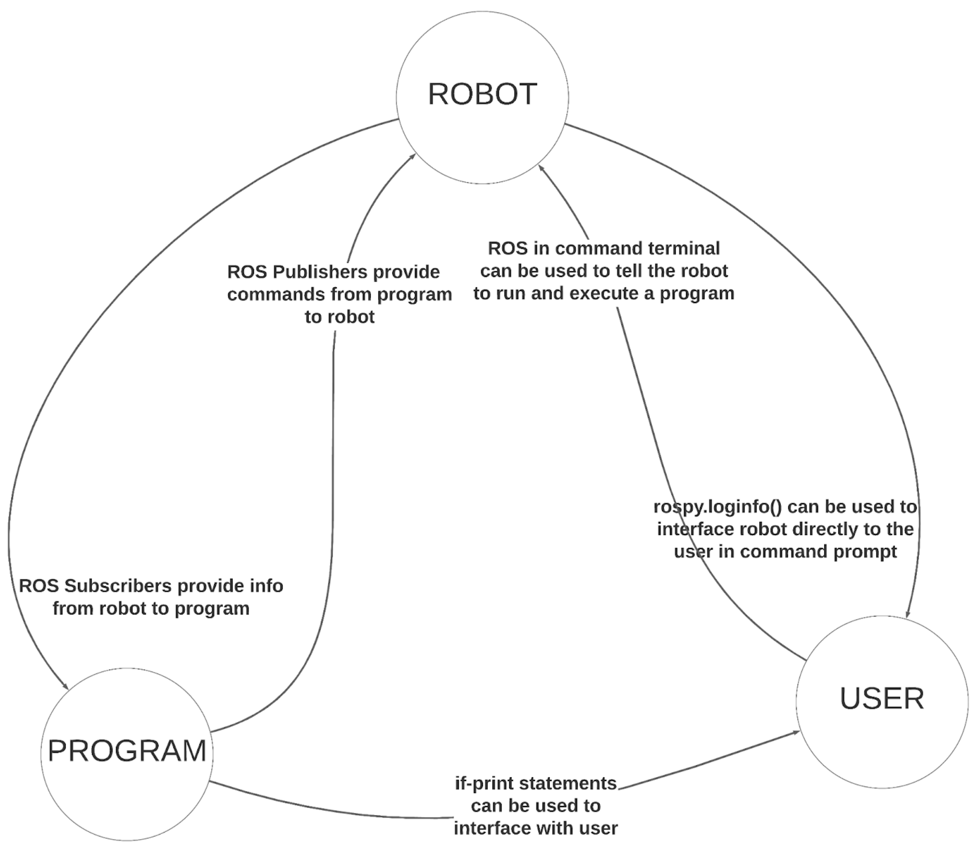
```
While (not input_done)
    Prompt user to enter start location
        If (start != 0) AND (start == 1 or 2 or 3), set start location
        Else terminate
    Prompt user to enter final location
        If (final != 0) AND (final != start), set final location
        Else terminate
    Program sets buffer, the location that is not start or final

Begin to move blocks
    First move_block, second move_block, …., seventh move_block
```

What is ROS and how does it work?

Robotic Operating Systems (ROS) is a framework that allows ROS developers to create programs for robots, without physically touching the robot. ROS particularly works on Linux Ubuntu and Linux Debian, not on Windows, which is why the VMWare used in this class is a necessary part in ROS program development.

For this lab, we learn how the Robot provides values to the program via Subscribers, the Program provides commands to the robot via Publishers, the Robot provides rospy.loginfo() to the user about its state, the User provides instructions to the robot to run, and the User and Program interface with each other by providing inputs and outputting states. This diagram below depicts the explanation above



How did you use the ROS commands (i.e. rostopic list, rostopic info, etc.) to complete your task?

ROS in Command Terminal. There are several steps that are necessary to understand which topics can be used by the program to subscribe to and which messages can be used by the program in a callback function.

ROS commands in command terminal	
`rostopic list`	Used to find what topics you would like to subscribe to
`rostopic info`	Used to describe topic you would like to subscribe to
`rosmsg list`	Used to find message to be outputted in callback function
`rosmsg info`	Used to describe message to be outputted in callback function
`roslaunch ur3_driver ur3_driver.launch` `rosrun lab2pkg_py lab2_exec.py`	Used in series in order to run the executable file, in this case, the lab3_exec.py. This step is more in depth in the next section

ROS in Programs. There are several steps that are necessary to initializing Publishers (commands from program to robot) and Subscribers (values from robot to the program)

ROS commands in programs	
`# Initialize ROS node` `rospy.init_node('lab2node')`	In main(), we want to start the code with an initialization of the node
`# Initialize publisher for ur3/command` `pub_command = rospy.Publisher('ur3/command', command, queue_size=10)`	In main(), we want to initialize a PUBLISHER, which will be used to command the robot
`# Initialize subscriber to ur3/position` `sub_position = rospy.Subscriber('ur3/position',position,position_callback)`	In main(), we want to initialize a SUBSCRIBER, which will be used to read POSITION values from the robot
`# Initialize subscriber to ur3/gripper` `sub_gripped = rospy.Subscriber( 'ur3/gripper_input',gripper_input,gripped_callback)`	In main(), we want to initialize a SUBSCRIBER, which will be used to read GRIPPER values from the robot
`rospy.loginfo`	Prints a message to the command prompt.

ROS in Command Terminal. Running a node, lab2_exec.py. There are several steps that are necessary to remember when running the node that we built. Here is a set of commands that we used and guided comments

Creating a workspace in Commands Terminal

```
mkdir -p catkin_rbalita2/src   // creates src folder for workspace
cd catkin_rbalita2/src         // directs to folder created
catkin_init_workspace          // makes this folder the src folder, and creates
                                  the build and devel folders in your workspace
```

Building the workspace. Always do in Command Terminal

```
cd ~/catkin_rbalita2/
Catkin_make                    // build workspace
```

After compilation is complete, I can now launch ROS commands. Open Gazebo

```
cd catkin_rbalita2/src/ur3_driver         // directs to ur3_driver folder
Roslaunch ur3_driver ur3_gazebo.launch    // this should open gazebo

CTRL + SHIFT + N                          // opens new tab
source devel/setup.bash                   // do this every time you open a new tab to use tabs
```

Make your execute file executable

```
Cd ~                                            // cd to main
Cd catkin_rbalita2/src/lab2andDrivers/scripts   // direct to scripts of lab
chmod +x lab2_exec.py                           // makes the lab2_exec file executable
```

See your robot move

```
Cd ~                                      // cd to main
Cd catkin_rbalita2/src/lab2andDrivers     // direct to lab folder

Rosrun lab2pkg_py lab2_exec.py                         // For Gazebo
Rosrun lab2pkg_py lab2_exec.py --simulator True        // For simulation
Rosrun lab2pkg_py lab2_exec.py --simulator False       // Run it on Hardware
```

How did you implement feedback?

Feedback comes in several forms. Error feedback(software) and Sensor feedback (robot)

Sensor Feedback came in several forms for this lab. (`ROBOT -> PROGRAM`)

1. Suction Feedback. This ROS topic callback function allows us to know if the gripper is "gripping". It provides a state in which we can later use to tell the robot to either continue with the instructions or HALT because there is no suction to a brick being applied

ROS Gripper Callback Function. allows us to get the state of the suction cup (when published)

```
def gripped_callback(msg):
   global digital_in_0         #defines digital input (is gripper on)
   global analog_in_0          #defines analog input (is gripper on)
   digital_in_0 = msg.DIGIN    #sets state of suction to be the state of DIGIN from robot
   analog_in_0 = msg.AIN0      #sets state of suction to be the state of ANIN from robot
```

ROS Gripper Callback Function in main(), initialized as a subscriber

```
sub_gripped = rospy.Subscriber('ur3/gripper_input', gripper_input, gripped_callback)
```

2. Position Feedback. This position callback allows us to set the program to have access to the position of the robot. By communicating with the robot about its position, we will be able to set the current and next positions of the robot. In particular, we set the theta values and the current position values both from the msg being given from the robot.

ROS Position Callback Function. allows us to get the position of the robot (when published)

```
def position_callback(msg):
Set thetas[] array to the msg.position[]          # gathers info from robot msg
Set current_position[]array to the thetas[]       # sets prog variables to values attained
Set current_position_set = True                   # assuming all functions go well,
```

ROS Gripper Callback Function in main(), initialized as a subscriber

```
sub_position = rospy.Subscriber('ur3/position', position, position_callback)
```

Error Feedback came in several forms for this lab. (`PROGRAM -> USER`)

1. If the user's inputs for the starting and ending location of the towers were the same, then the program will display an error message saying "Overlap of start and end"

Same start and end location error. allows us to halt the system if no brick is present

```
# . . . . . code for main()
while(not input_done):
start_loc = raw_input("Enter the Start Location <Either 1 2 3 or 0 to quit> ")
if(int(start_loc) == 1 or int(start_loc) == 2 or int(start_loc) == 3):
   input_done = 1
   start_loc = int(start_loc)
elif (int(start_loc) == 0):
   print("Quitting... ")
   sys.exit()

# . . . . . code for main()

while(not input_done):
end_loc = raw_input("Enter the End Location <Either 1 2 3 or 0 to quit>")
if(int(end_loc) == start_loc):
   print("Overlap with the Start Location; Please choose another End Location \n\n")
elif((int(end_loc) == 1 or int(end_loc) == 2 or int(end_loc) == 3) and int(end_loc) !=start_loc):
   input_done = 1
   end_loc = int(end_loc)
elif (int(end_loc) == 0):
   print("Quitting... ")
   sys.exit()

# . . . . . code for main()
```

2. If the robot were to move to a particular position and sense no brick is present, then the robot will halt, go back to the central waypoint, and the program will display an error message saying that "unexpected missing brick". This is done because the algorithm used to solve the 3 brick ToH works with starting and ending tower positions, so the only ways that an error with a missing brick would occur is if the brick has fallen down or the user took the brick out of the expected position.

No brick error example. allows us to halt the system if no brick is present

```
if digital_in_0 == False:
       rospy.loginfo("Block Missing; Moving back to home ...")
       move_arm(pub_cmd,loop_rate,home,4.0,4.0)
       rospy.loginfo("Ungripping")
       gripper(pub_cmd,loop_rate,suction_off)
       sys.exit()
```

3. Data and Results

Analyze the data you collected

There was no particular data that was collected for this lab experiment. The purpose of this lab was to introduce the ROS and Python programming languages through the Towers of Hanoi puzzle. The measures that were taken focused solely on getting the program to work, launching the program onto hardware, and seeing if the outcome of programming in ROS and Python created similar results to that of programming the robot the UR3 teaching pendant. By taking the input joint angles for all waypoints from Lab 1, it is expected that we can use this information when running an ROS/Python program and have the same outcome.

Running the program through simulation:

- **Test Passed.** Terminal messages indicated that the robot moved to central position, moved to moveblock_1_start, turned on suction, received suction feedback, moved to central position, moved to moveblock_1_end, turned off suction and repeated this cycle until the ToH was complete

Running the program through hardware:

- **Test Passed.** Terminal messages indicated that the robot moved to central position, moved to moveblock_1_start, turned on suction, received suction feedback, moved to central position, moved to moveblock_1_end, and turned off suction repeated this cycle until the ToH was complete
- **Test Passed.** Robot moved in real time, somewhat faster than the time when the robot was programmed on the UR3 teaching pendant. Speed was at 4.0 in program

Testing for Feedback

- **Test Passed.** User inputted the same starting and ending location of the towers. The program displayed an error message saying that “The Tower is already in final position”
- **Test Passed.** The user took the brick out of the expected position.

An error analysis and discussion of sources of error

Although all the test cases and demo procedures have been passed, there may have been small robotic arm positioning errors. This may have been a result from the recording of the joint angles in Lab 1. The errors did not result in significant deviations from our expected waypoint positions. Considering that the joint angles were recorded to the tenths of a degree, setting the joint angles in the program by rounding the values to the nearest degree may have caused the positioning of the robotic arm to deviate slightly. The errors were limited. It can be said that these small errors were truly insignificant to executing the objective of the lab.

4. Conclusion

Summarize what you did and the results of your data. Discuss what you learned from the lab

The first objective of the lab was to be introduced to the ROS and Python programming language and familiarize ourselves with how these tools can be used in both simulation and in real life hardware. The underlying and motivating objective of this Lab 2 was to recreate the Towers of Hanoi puzzle from Lab 1, but while programming the robot in ROS and Python. In particular, we learned how to use callback functions, set subscribers, set publishers, and how to utilize each of them to communicate back and forth with a robot. Overall, the things that we learned from this lab will be useful for the final project, where we will need to use similar callback functions and set similar subscribers and publishers, and to run ROS on the command terminal to run the simulation.

5. References

List of references

Buddies, Science. "The Tower of Hanoi." *Scientific American*, Scientific American, 26 Oct. 2017,https://www.scientificamerican.com/article/the-tower-of-hanoi/#:~:text=The%20tower%20of%20Hanoi%20.

"Ros for Beginners: What Is Ros?" *The Construct*, 10 July 2020, https://www.theconstructsim.com/what-is-ros/.

6. Appendix

Additional information if needed

Important Variables for Program

```
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
```

Implemented functions for program

Function	Description: TODO's
`def gripped_callback(msg):`	#define a topic callback function that will get the state of the suction cup
`def move_block(pub_cmd, loop_rate, start_loc, start_height, end_loc, end_height):`	#define a function that will move a block in the following steps - Move to central location (central) - Move arm to first location (start_loc, start_height) - Turn on gripper - If digital_input (gripped) is false, produce error & HALT - If digital_input (gripped) is true, move block to central - Move arm to second location (end_loc, end_height) - Turn off gripper - Move to central location (central)
`def main():`	#complete the function so that the program will take an input from the user (first and last location of the tower) and the robot will complete the task. // one may implement this by either hard coding the values for the locations OR they may implement this by realizing that locations can be relabeled as "first", "secondary", and "final" location

Code for lab2_exec.py

```
#!/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
import copy
import time
import rospy
import rospkg
import numpy as np
import yaml
import sys
from math import pi
from lab2_header import *

# 20Hz
SPIN_RATE = 20

# UR3 home location
home = np.radians([159.45, -67.17, 90.18, -113.04, -90.17, 39.38])

# Hanoi tower location 1
Q11 = [145.76*pi/180.0, -54.86*pi/180.0, 91.51*pi/180.0, -126.67*pi/180.0, -90.23*pi/180.0, 25.62*pi/180.0]
Q12 = [145.76*pi/180.0, -49.64*pi/180.0, 93.6*pi/180.0, -133.98*pi/180.0, -90.25*pi/180.0, 25.59*pi/180.0]
Q13 = [146*pi/180.0, -44.43*pi/180.0, 94.59*pi/180.0, -140.18*pi/180.0, -90.27*pi/180.0, 25.80*pi/180.0]
Q21 = [159.46*pi/180.0, -59.52*pi/180.0, 99.74*pi/180.0, -130.25*pi/180.0, -90.21*pi/180.0, 39.36*pi/180.0]
Q22 = [159.48*pi/180.0, -53.41*pi/180.0, 100.40*pi/180.0, -137.03*pi/180.0, -90.23*pi/180.0, 39.33*pi/180.0]
Q23 = [159.49*pi/180.0, -48.12*pi/180.0, 102.67*pi/180.0, -144.58*pi/180.0, -90.25*pi/180.0, 39.32*pi/180.0]
Q31 = [171.41*pi/180.0, -57.51*pi/180.0, 96.60*pi/180.0, -129.13*pi/180.0, -90.23*pi/180.0, 51.68*pi/180.0]
Q32 = [171.75*pi/180.0, -52.35*pi/180.0, 98.58*pi/180.0, -136.27*pi/180.0, -90.25*pi/180.0, 51.61*pi/180.0]
Q33 = [171.77*pi/180.0, -46.29*pi/180.0, 99.43*pi/180.0, -143.18*pi/180.0, -90.28*pi/180.0, 51.58*pi/180.0]

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

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

############## Your Code Start Here ##############
"""
TODO: Initialize Q matrix
"""

Q = [ [Q11, Q12, Q13], \
     [Q21, Q22, Q23], \
     [Q31, Q32, Q33] ]
############### Your Code End Here ###############

############## Your Code Start Here ##############

"""
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 gripped_callback(msg):
```

```python
   global digital_in_0
   global analog_in_0

   digital_in_0 = msg.DIGIN
   analog_in_0 = msg.AIN0




############### Your Code End Here ###############


"""
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
```

```
        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_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


############## Your Code Start Here ##############

def move_block(pub_cmd, loop_rate, start_loc, start_height, \
               end_loc, end_height):
    global Q
    global home
    global digital_in_0

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

    error = 0
    rospy.loginfo("Moving to start location: " + str(start_loc) + " " + str(start_height))
```

```
    move_arm(pub_cmd,loop_rate,Q[start_loc-1][start_height-1],4.0,4.0)
    rospy.loginfo("Gripping")
    gripper(pub_cmd,loop_rate,suction_on)
    time.sleep(1.0)
    if digital_in_0 == False:
        rospy.loginfo("Block Missing; Moving back to home ...")
        move_arm(pub_cmd,loop_rate,home,4.0,4.0)
        rospy.loginfo("Ungripping")
        gripper(pub_cmd,loop_rate,suction_off)
        sys.exit()
    rospy.loginfo("Moving to home location")
    move_arm(pub_cmd,loop_rate,home,4.0,4.0)
    rospy.loginfo("Moving to end location: " + str(end_loc) + " " + str(end_height))
    move_arm(pub_cmd,loop_rate,Q[end_loc-1][end_height-1],4.0,4.0)
    rospy.loginfo("Ungripping")
    gripper(pub_cmd,loop_rate,suction_off)
    rospy.loginfo("Last: Moving to home location")
    move_arm(pub_cmd,loop_rate,home,4.0,4.0)



    return error


############### Your Code End Here ###############


def main():

    global home
    global Q
    global SPIN_RATE

    # 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 callback function
    sub_gripped = rospy.Subscriber('ur3/gripper_input', gripper_input, gripped_callback)

    ############### Your Code End Here ###############


    ############## Your Code Start Here ##############
    # TODO: modify the code below so that program can get user input

    input_done = 0
    loop_count = 0

    while(not input_done):
        start_loc = raw_input("Enter the Start Location <Either 1 2 3 or 0 to quit> ")
        print("Entered Start Location: " + start_loc + "\n")

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

```
    input_done = 0
    while(not input_done):
        end_loc = raw_input("Enter the End Location <Either 1 2 3 or 0 to quit>")
        print("Entered End Location: " + end_loc + "\n")

        if(int(end_loc) == start_loc):
            print("Overlap with the Start Location; Please choose another End Location \n\n")
        elif((int(end_loc) == 1 or int(end_loc) == 2 or int(end_loc) == 3) and int(end_loc) != start_loc):
            input_done = 1
            end_loc = int(end_loc)
        elif (int(end_loc) == 0):
            print("Quitting... ")
            sys.exit()
        else:
            print("Please just enter the character 1 2 3 for the End Location or 0 to quit \n\n")

    buffer_loc = 1
    if (start_loc == 1 and end_loc == 2) or (start_loc == 2 and end_loc == 1):
        buffer_loc = 3
    elif (start_loc == 1 and end_loc == 3) or (start_loc == 3 and end_loc == 1):
        buffer_loc = 2



    ############### Your Code End Here ###############

    # 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)

    ############## Your Code Start Here ##############
    # TODO: modify the code so that UR3 can move tower accordingly from user input

    not_gripped = False

    rospy.loginfo("Home Movement ...")
    move_arm(pub_command,loop_rate,home,4.0,4.0)

    rospy.loginfo("First Movement ...")
    move_block(pub_command, loop_rate, start_loc, 1, end_loc, 3)

    rospy.loginfo("Second Movement ...")
    move_block(pub_command, loop_rate, start_loc, 2, buffer_loc, 3)

    rospy.loginfo("Third Movement ...")
    move_block(pub_command, loop_rate, end_loc, 3, buffer_loc, 2)

    rospy.loginfo("Fourth Movement ...")
    move_block(pub_command, loop_rate, start_loc, 3, end_loc, 3)

    rospy.loginfo("Fifth Movement ...")
    move_block(pub_command, loop_rate, buffer_loc, 2, start_loc, 3)

    rospy.loginfo("Sixth Movement ...")
    move_block(pub_command, loop_rate, buffer_loc, 3, end_loc, 2)

    rospy.loginfo("Seventh Movement ...")
    move_block(pub_command, loop_rate, start_loc, 3, end_loc, 1)

    rospy.loginfo("Finished")


    ############### Your Code End Here ###############
```

```python
if __name__ == '__main__':

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