ECE 470: Lab 3
The Tower of Hanoi with ROS

Shenghua Ye, Xuan Tang, Jie Wang
Student ID: 3200112420, 3200115277 3200112404
Instructor: Yang Liangjing

March 21, 2024

1 Introduction

1.1 Background

The Tower of Hanoi is a classic puzzle that involves moving a stack of disks from one peg to another, with the constraint that a larger disk cannot be placed on top of a smaller one. This lab aimed at using UR3e robot arm to solve this puzzle in a simulated environment using Robot Operating System(ROS) and Python.

This lab extends our understanding of the Tower of Hanoi problem, as we did using Teach Pendant in Lab1. Code provide a more efficient and flexible control over the robot's movements and task management, enhancing the complexity and scalability of the solutions in a more scientific appraoch.

1.1.1 Robot Operating System (ROS)

ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. ROS provides standardized protocols for communication between different components of a robot system, such as sensors, actuators, and processing units.

1.1.2 Python Programming

Python was chosen as the programming language for this lab due to its readability and the extensive support it offers for ROS through various packages and libraries.

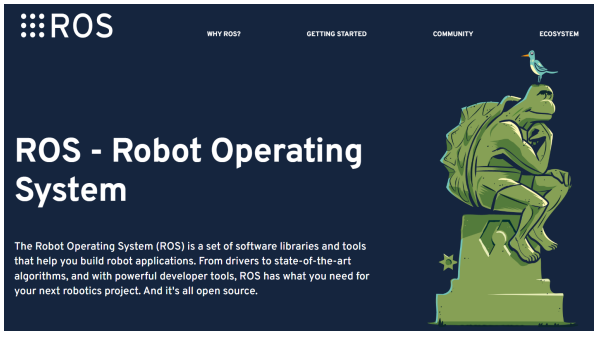

Figure 1: ROS is a special OS for the robot

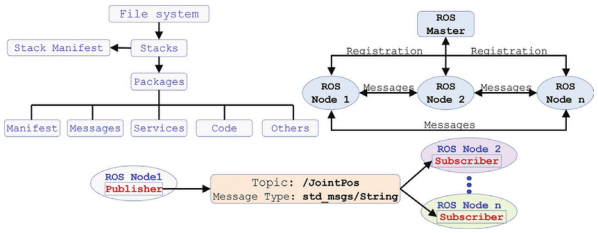

Figure 2: ROS file architecture and nodes communicating system in Python

1.2 Objective

The primary objective of this lab was to gain hands-on experience with ROS and Python for controlling a robotic arm. Specific goals included:

- Modifying *'lab3_ exec.py'*to move the robot to specified waypoints.
- Enabling and disabling a suction cup gripper to move blocks to the sepcific waypoints.
- Solving the Tower of Hanoi problem with TA-selected start and end positions.
- Implementing suction feedback to verify successful block grasping.

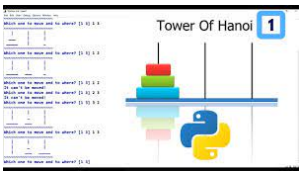

Figure 3: Hanoi Tower Problem can be efficiently solved with Recursive Function

2 Method

2.1 Environment Setup

The lab environment was set up following the instructions in the lab manual and *readme.md*. We created a workspace and copying the necessary package and environment for compiling the python file into it. Then, we need to enter our workspace folder and source our workspace every time we open a new terminal.

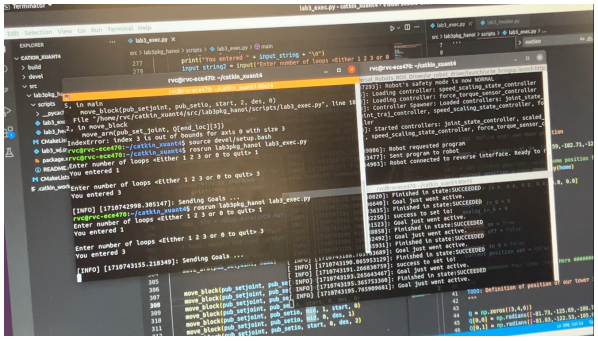

Figure 4: We forgot to source all three terminals at first, spending a lot of effort fighting against the air

Notice: Always source the terminal before you run your code, as this links the specific Python code to your terminal.

```
> root@your_pc: ~/catkin_{your workspace}$ source devel/setup.bash
```

2.2 Q Table for Space Parameter

Here we use the same table as Lab1, which contains $3 \times 4 = 12$ waypoints. Each waypoint is described using a 6-dimension radian vector, as the URe4 has 6 degree of freedom. We set the robot arm mannually and read the waypoints' location into the Computer one by one. Although our Q table implementation differed from the one described in the Lab 3 manual(*18 waypoints*, it still operated correctly while saving more time and effort.

```
import numpy as np
Q = np.zeros((3,4,6))
Q[0,0] = np.radians([-81.73,-125.69,-106.78,-37.44,90.29,356.47])
Q[0,1] = np.radians([-81.83,-122.53,-105.69,-41.70,90.31,356.35])
Q[0,2] = np.radians([-81.64,-119.07,-105.25,-45.59,90.32,356.53])
Q[0,3] = np.radians([-81.63,-110.23,-96.98,-62.70,90.36,356.43])
Q[1,0] = np.radians([-68.50,-124.51,-109.76,-35.64,90.28,9.71])
Q[1,1] = np.radians([-68.61,-121.41,-108.49,-40.02,90.30,9.58])
Q[1,2] = np.radians([-68.25,-117.77,-107.53,-44.61,90.32,9.92])
Q[1,3] = np.radians([-68.88,-109.64,-101.87,-58.41,90.36,9.21])
Q[2,0] = np.radians([-56.42,-126.69,-105.64,-37.58,90.28,21.78])
Q[2,1] = np.radians([-56.15,-122.37,-105.23,-42.32,90.31,22.03])
Q[2,2] = np.radians([-56.15,-119.15,-103.80,-46.96,90.33,22.00])
Q[2,3] = np.radians([-56.15,-109.84,-96.94,-63.13,90.36,21.90])
```

2.3 Coding

The starter Python file provided was modified to include functions for moving the robot arm, gripping and releasing blocks, and solving the Tower of Hanoi puzzle.

First, we defined global variables for the UR3e robotic arm's home position and target position (Q matrix) for the disks at different heights and locations on the Tower of Hanoi setup. And we initialized the a ROS publisher for /ur3e_driver_ece470/setio message and a ROS subscriber for /ur_hardware_interface/io_states message and corresponding callback function in the main function.

Then, we implement the gripper_input_callback function to update the state of the gripper based on the information published on certain ROS topics (/ur_hardware_interface/io_states), like the position_callback function defined for us already. And we implemented the gripper control function (gripper) to control the state of the robotic arm's suction gripper by publishing to the /ur3e_driver_ece470/setio topic.

After that, we completed the move_block function that encapsulates the steps to move a disk from one rod to another. It uses the move_arm function to navigate the arm above the start disk, lower to grip the disk, use the gripper function to turn the gripper on, move it to the target location, and release it.

At last, we implemented the input functionality that allows the user to choose the start and end position of the Haoni Tower, and we also implemented the main logic to solve the Tower of Hanoi puzzle by moving disks from the start rod to the destination rod, using an intermediate rod as needed.

All the codes are listed in the Appendix part.

2.4 Testing

The program was tested in the simulated environment provided, ensuring that the robot could move blocks between towers and that the suction feedback mechanism was functioning correctly.

3 Data and Results

The UR3e robot was successfully programmed to solve the Tower of Hanoi puzzle. The robot was able to move blocks from any starting position to any ending position as specified by the user. The suction feedback feature was tested and confirmed to halt the program with an error when a block was not successfully grasped.

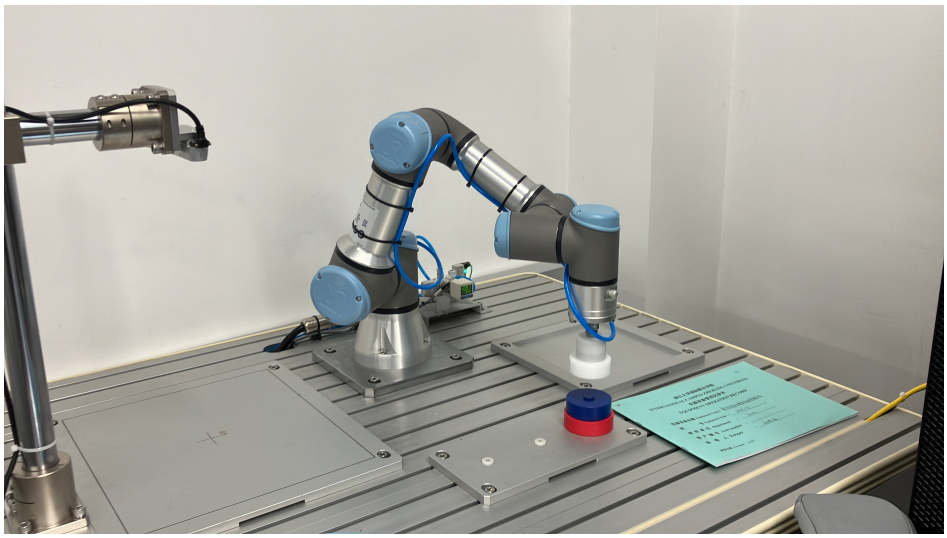

Figure 5: Figure 1. last step

4 Conclusion

The lab provided valuable experience in using ROS and Python for robotic control. Basically, a program is designed to control a UR3e robotic arm to solve a Tower of Haoni puzzle using ROS. Through this lab, we gain a valuable experience of developing ROS architecture, like how to publish commands or subscribe to sensor feedback. We also gain a first glimpse the use of specific message types for controlling the robot and reading its sensors introduces you to custom ROS messages and their applications.

5 References

1. M. Quigley et al., "ROS: An Open-source Framework for Robotics," in The International Journal of Robotics Research, vol. 31, no. 3, pp. 235-246, 2012.
2. "A Gentle Introduction to ROS," University of Illinois at Urbana-Champaign, ECE 470, http://coecsl.ece.illinois.edu/ece470/agitr-letter.pdf.
3. "UR3e User Manual," Universal Robots, https://www.universal-robots.com/products/ur3e/.

6 Appendix

6.1 Our Code

```
# The key functions that we implemented listed here
# a ROS topic callback function for getting the state of suction cup
def gripper_input_callback(msg):
  global current_io_0
  current_io_0 = msg.digital_in_states[0].states

 # a function for ROS Publisher to publish your message to the corresponding Topic
 def gripper(pub_setio, io_0):
  msg = Digital()
  msg.pin = 0
  msg.state = io_0
  pub_setio.publish(msg)
  time.sleep(2)

 # a function to move block from start to end
 def move_block(pub_set_joint, pub_set_io, start_loc, start_height, end_loc,
    end_height):
    global Q
    move_arm(pub_set_joint, Q[start_loc][3])
    move_arm(pub_set_joint, Q[start_loc][start_height])
    gripper(pub_set_io, True)
    move_arm(pub_set_joint, Q[start_loc][3])
    move_arm(pub_set_joint, Q[end_loc][3])
    move_arm(pub_set_joint, Q[end_loc][end_height])
    gripper(pub_set_io, False)
    move_arm(pub_set_joint, Q[end_loc][3])


def main():

  global home
  global Q
  global SPIN_RATE
  rospy.init_node('lab3_node')
  pub_setjoint = rospy.Publisher('ur3e_driver_ece470/setjoint',JointTrajectory,
    queue_size=10)

  # TODO: define a ROS publisher for /ur3e_driver_ece470/setio message
  pub_setio = rospy.Publisher('/ur3e_driver_ece470/setio', Digital, queue_size=10)

  sub_position = rospy.Subscriber('/joint_states', JointState, position_callback)
  # TODO: define a ROS subscriber for /ur_hardware_interface/io_states message and
    corresponding callback function
  sub_io = rospy.Subscriber('/ur_hardware_interface/io_states', IOStates,
    gripper_input_callback)

  # TODO: modify the code below so that program can get user input
  input_done = 0
  loop_count = 0
  start = 0
  mid = 1
  des = 2

  while(not input_done):
    input_string = input("Enter number of loops <Either 1 2 3 or 0 to quit> ")
    print("You entered " + input_string + "\n")
    input_string2 = input("Enter number of loops <Either 1 2 3 or 0 to quit> ")
    print("You entered " + input_string2 + "\n")

    start = int(input_string)
    des = int(input_string2)
    another = 3 - start - des

    input_done = 1
```

```
61
62  # Check if ROS is ready for operation
63  while(rospy.is_shutdown()):
64    print("ROS is shutdown!")
65
66  rospy.loginfo("Sending Goals ...")
67
68  loop_rate = rospy.Rate(SPIN_RATE)
69        # TODO: Operate the Hanoi Tower Game
70  # set to home before start
71  move_arm(pub_setjoint, home)
72  move_block(pub_setjoint, pub_setio, start, 2, des, 0)
73  move_block(pub_setjoint, pub_setio, start, 1, mid, 0)
74  move_block(pub_setjoint, pub_setio, des, 0, mid, 1)
75  move_block(pub_setjoint, pub_setio, start, 0, des, 0)
76  move_block(pub_setjoint, pub_setio, mid, 1, start, 0)
77  move_block(pub_setjoint, pub_setio, mid, 0, des, 1)
78  move_block(pub_setjoint, pub_setio, start, 0, des, 2)
79  move_arm(pub_setjoint, home)
```