LAB 2

The Tower of Hanoi with ROS

2.1 Important

Read the entire lab before starting and especially the "code explanation" section so you can be prepared for your lab session.

2.2 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:

- Record joint angles that position the robot arm at waypoints in the Tower of Hanoi solution

- 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.

2.3   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.

2.4   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.)
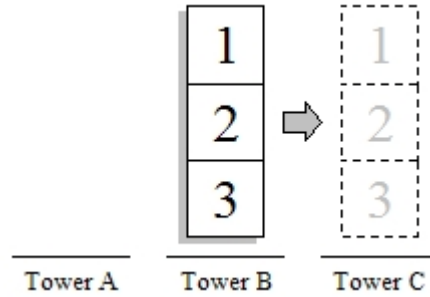
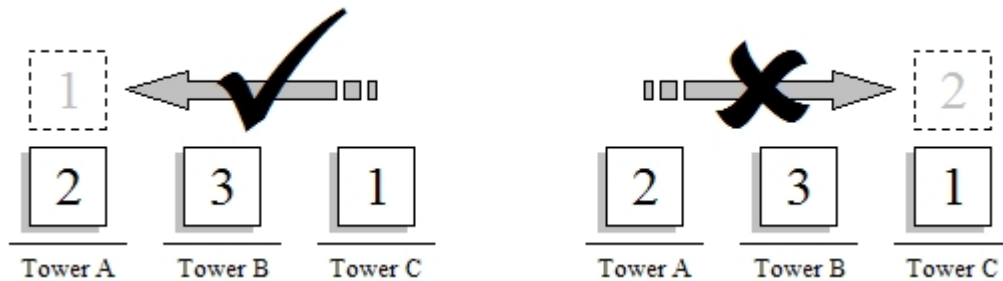Figure 2.1: Example start and finish tower locations.

Figure 2.2: Examples of a legal and an illegal move.

2.5 Task

The goal is to move a "tower" of three blocks from one of three locations on the table to another. An example is shown in Figure 2.1. The blocks are numbered with block 1 on the top and block 3 on the bottom. When moving the stack, two rules must be obeyed:

1. Blocks may touch the table in only three locations (the three "towers").

2. You may not place a block on top of a lower-numbered block, as illustrated in Figure 2.2.

For this lab, we will complicate the task slightly. 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.

2.6 Procedure

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

2. If you haven't already, download **lab2andDriverPy.tar.gz** from the course website and extract into your catkin workspace **/src** folder. Do this at a command prompt with the tar command, **tar -zxvf lab2andDriverPy.tar.gz**. You should see two folders **lab2pkg_py** and **drivers**. Compile your workspace with **catkin_make**. Inside this package you can find **lab2_exec.py** with comments to help you complete the lab.

   - **lab2_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 lab2_exec.py.
   - **package.xml** This file defines properties about the package including package dependencies.
   - To run lab2 code: In one terminal source it and run **roslaunch ur3_driver ur3_driver.launch**.
   - In second terminal source it and run **rosrun lab2pkg_py lab2_exec.py**

3. Use the provided tape to mark the three possible tower bases. You should initial your markers so you can distinguish your tower bases from the ones used by teams in other lab sections.

4. For each base, place the tower of blocks and use the teach pendant to find joint angles corresponding to the top, middle, and bottom block positions and orientation angles. Record these joint angles for use in your program.

5. Modify **lab2_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.

6. 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 and **Analog Input 0** are connected to this feedback. Use **Digital Input 0** or **Analog 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 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** or **Analog Input 0**. Normally once you figure out which message you will be using you need to import it in the **lab2_header** file that defines this message. The **lab2_header** file has already imported it in **lab2_header.py** for you. Use the explanation below and the given code in **lab2_exec.py** that creates the subscription to **ur3/position** and its callback function as a guide to subscribe to the rostopic that publishes the IO status.

2.7   Lab2_exec.py Explained

First open up **lab2_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 **lab2_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 **lab2_exec.py** file as the correct version. **lab2_exec.py** is broken down into sections and described in more detail below.

```
import copy
import time
import rospy
from lab2_header import *

SPIN_RATE = 20
```

You can find **lab2_header.py** in the **lab2pkg_py** /scripts directory. It includes all needed files to allow **lab2_exec.py** to call ROS functionality. **SPIN_RATE** will be used as the publish rate to send commands to the ROS driver.

```
home = [120*PI/180.0, -90*PI/180.0, 90*PI/180.0, -90*PI/180.0, -90*PI/180.0, 0*PI/180.0]


# Hanoi tower location 1
Q11 = [120*PI/180.0, -56*PI/180.0, 124*PI/180.0, -158*PI/180.0, -90*PI/180.0, 0*PI/180.0]
Q12 = [120*PI/180.0, -64*PI/180.0, 123*PI/180.0, -148*PI/180.0, -90*PI/180.0, 0*PI/180.0]
Q13 = [120*PI/180.0, -72*PI/180.0, 120*PI/180.0, -137*PI/180.0, -90*PI/180.0, 0*PI/180.0]

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

```
digital_in_0 = 0;
analog_in_0 = 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)
```

This code is initializing three python "list" variables to be used as waypoints. These variables Q11, Q12 and Q13 will be used to command the robot to the six θ's assigned to the list. The Q11, Q12 and Q13 list elements are in radians and arranged in the order $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \}$. We also initialize the "current_position" using the "home" list.

```
Q = [ [Q11, Q12, Q13], \
      [Q11, Q12, Q13], \
      [Q11, Q12, Q13] ]
```

The lists, once created, are added to a multi-dimensional list for easy access. This structure will allow us to step through the array in a logical fashion. Currently all the columns are the same but you can change this as you see fit, perhaps letting row indicate the height in the stack and column tower location. Feel free to change the size and contents as appropriate for your algorithm.

```
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 **lab2node**'s callback function that is called when the **ur3_driver** publishes new position data. **ur3_driver** publishes new angle position data every 8ms, so this function **position_callback** is run every 8ms.

17

Next in **lab2_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

    # 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)
```

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 "**ur3**" node "**command**" 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 **lab2node** subscribes to "**ur3**" node "**position**" 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 "**ur3**" node and "**command**" subscriber and "**position**" publisher using the "**rostopic list**" command in your **catkin_ work directory**. Also use "**rostopic info**" to double check that "**command**" is "**ur3**" subscriber and "**position**" is a publisher. Also run "**rosmsg list**" to find the messages "**ur3_driver.msg.position**", "**ur3_driver.msg.command**".

```
    input_done = 0
    loop_count = 0

    while(not input_done):
        input_string = raw_input("Enter number of loops <Either 1 2 or 3> ")
        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
        else:
```

```
            print("Please just enter the character 1 2 or 3 \n\n")
```

This standard python code printing messages to the command prompt and recieving 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.

```
    while(loop_count > 0):

    #loop_count is the number of times to repeat the remaining code.
    #This value is entered by the user.

        move_arm(pub_command, loop_rate, home, 4.0, 4.0)

        rospy.loginfo("Sending goal 1 ...")
        move_arm(pub_command, loop_rate, Q[0][0], 4.0, 4.0)

        gripper(pub_command, loop_rate, suction_on)
        # Delay to make sure suction cup has grasped the block
        time.sleep(1.0)

        rospy.loginfo("Sending goal 2 ...")
        move_arm(pub_command, loop_rate, Q[0][1], 4.0, 4.0)

        rospy.loginfo("Sending goal 3 ...")
        move_arm(pub_command, loop_rate, Q[0][2], 4.0, 4.0)
```

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.

```
        loop_count = loop_count - 1
```

Repeat the moves **loop_count** number of times.

```
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() # 50ms

    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
```

The **move_arm()** function takes as parameters **pub_cmd**, which is the publisher to **ur3_driver** commanding a new position for the robot to move to. rate is the sleep rate this function will sleep in between checking if the robot has reached the commanded position. This is necessary so that other ROS processes are given processor time during **move_arm**'s wait for the robot to get to the commanded position. **dest** is the six joint angle destinations, in radians, that the robot will be commanded to move to. **vel** is the velocity that each joint will move going to the destination. **accel** is the acceleration that each joint will move goint to the destination. The code create a variable **driver_msg** which is the command message to be sent **ur3_driver**. **driver_msg** is assigned the destination, **dest**, acceleration, **accel**, and velocity, **vel**. In addition the state of the suction cup gripper, **current_io_0**, is assigned

to **driver_msg**. This is the last state of the gripper, On or Off, commanded by the function **gripper()**. Next the **driver_msg** is published to **ur3_driver** with the **pub_cmd.publish(driver_msg)** instruction. The **while(at_goal == 0)** loop, loops until the robot arm has reached the commanded position or at least with in 0.0005 radians. If for some reason the first publish does not send correctly, after five seconds the command is published again. This will repeat until the robot arm reaches the commanded position.

```python
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
```

The **gripper()** function is very similar to the **move_arm()** function above. The same **pub_cmd** and rate are passed to **gripper()** but only the On/Off desired state of the suction cup gripper is the remaining parameter. Looking at the **move_arm()** function **gripper()** looks very similar but the robot joint destination is the current state of the arm so the robot is already at that position

so it does not move. All that the command changes is whether the suction gripper is On or Off by setting **driver_msg.io_0** equal to the passed parameter **bool io_0**. See the **move_arm()** description for more details on this code.

```
def move_block(pub_cmd, loop_rate, start_loc, \
               start_height, end_loc, end_height, \
               vel, accel):
    error = 0
    return error
```

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.

2.8 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 GradeScope.

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 **lab2_exec.py** code as an Appendix to your report as described in lab report guidelines.

2.9 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.

2.10 Grading

- 75 points, successful demonstration.

- 20 points, report.

- 5 points, attendance.

Appendix A

ROS Programming with Python

A.1   Overview

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

- The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

- For more details about ROS: `http://wiki.ros.org/`

- How to install on your own Ubuntu: `http://wiki.ros.org/ROS/Insta llation`

- For detailed tutorials: `http://wiki.ros.org/ROS/Tutorials`

A.2   ROS Concepts

The basic concepts of ROS are nodes, Master, messages, topics, Parameter Server, services, and bags. However, in this course, we will only be encountering

the first four.

- **Nodes** "programs" or "processes" in ROS that perform computation. For example, one node controls a laser range-finder, one node controls he wheel motors, one node performs localization ...

- **Master** Enable nodes to locate one another, provides parameter server, tracks publishers and subscribers to topics, services. In order to start ROS, open a terminal and type:

  ```
  $ roscore
  ```

  roscore can also be started automatically when using roslaunch in terminal, for example:

  ```
  $ roslaunch <package name> <launch file name>.launch
  # the launch file for all our labs:
  $ roslaunch ur3_driver ur3_driver.launch
  ```

- **Messages** Nodes communicate with each other via messages. A message is simply a data structure, comprising typed fields.

- **Topics** Each node publish/subscribe message topics via send/receive messages. A node sends out a message by publishing it to a given topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.In general, publishers and subscribers are not aware of each others' existence.
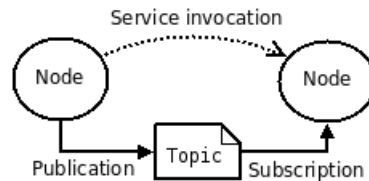

Figure A.1:   source: `http://wiki.ros.org/ROS/Concepts`

A.3   Before we start..

Here are some useful Linux/ROS commands

- The command "ls" stands for (List Directory Contents), List the contents of the folder, be it file or folder, from which it runs.

  ```
  $ ls
  ```

- The "mkdir" (Make directory) command create a new directory with name path. However is the directory already exists, it will return an error message "cannot create folder, folder already exists".

  $ mkdir <new_directory_name>

- The command "pwd" (print working directory), prints the current working directory with full path name from terminal

  $ **pwd**

- The frequently used "cd" command stands for change directory.

  $ **cd** /home/user/Desktop

  return to previous directory

  $ **cd** ..

  Change to home directory

  $ **cd** ˜

- The hot key "ctrl+c" in command line **terminates** current running executable. If "ctrl+c" does not work, closing your terminal as that will also end the running Python program. **DO NOT USE "ctrl+z" as it can leave some unknown applications running in the background**.

- If you want to know the location of any specific ROS package/executable from in your system, you can use "rospack" find "package name" command. For example, if you would like to find 'lab2pkg_py' package, you can type in your console

  $ rospack find lab2pkg_py

- To move directly to the directory of a ROS package, use roscd. For example, go to lab2pkg_py package directory

  $ roscd lab2pkg_py

- Display Message data structure definitions with rosmsg

  $ rosmsg show <message_type>   #Display the fields in the msg

- rostopic, A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages.

  ```
  $ rostopic echo /topic_name      #Print messages to screen
  $ rostopic list            #List all the topics available
  $ rostopic pub <topic-name> <topic-type> [data...]
  #Publish data to topic
  ```

A.4   Create your own workspace

Since other groups will be working on your same computer, you should backup your code to a USB drive or cloud drive everytime you come to lab. This way if your code is tampered with (probably by accident) you will have a backup.

- Log on to the computer as 'ur3' with the password 'ur3'. If you log on as 'guest', you will not be able to use ROS.

- First create a folder in the home directory, mkdir catkin_(yourNETID). It is not required to have "catkin" in the folder name but it is recommended.

  ```
  $ mkdir −p catkin_(yourNETID)/src
  $ cd catkin_(yourNETID)/src
  $ catkin_init_workspace
  ```

- Even though the workspace is empty (there are no packages in the 'src' folder, just a single CMakeLists.txt link) you can still "build" the workspace. Just for practice, build the workspace.

  ```
  $ cd ~/catkin_(yourNETID)/
  $ catkin_make
  ```

- **VERY IMPORTANT**: Remember to **ALWAYS** source when you open a new command prompt, so you can utilize the full convenience of Tab completion in ROS. Under workspace root directory:

  ```
  $ cd catkin_(yourNETID)
  $ source devel/setup.bash
  ```

A.5   Running a Node

- Once you have your catkin folder initialized, add the UR3 driver and lab starter files. The compressed file lab2andDanDriver.tar.gz, found at the class website contains the driver code you will need for all the ECE 470 labs along with the starter code for LAB 2. Future lab compressed files will only contain the new starter code for that lab. Copy lab2andDriverPy.tar.gz to your catkin directories "src" directory. Change directory to your "src" folder and uncompress by typing "tar -xvf lab2andDriver.tar.gz". You can also do this via the GUI by double clicking on the compressed file and dragging the folders into the new location.

  "cd .." back to your catkin_(yourNETID) folder and build the code with "catkin_make"

- After compilation is complete, we can start running our own nodes. For example our lab2node node. However, before running any nodes, we must have roscore running. This is taken care of by running a launch file.

```
$ roslaunch ur3_driver ur3_gazebo.launch
```

is used for the simulator.

```
$ roslaunch ur3_driver ur3_driver.launch
```

is used on the real robot.
This command runs both roscore and the UR3 driver that acts as a subscriber waiting for a command message that controls the UR3's motors.

- Open a new command prompt with "ctrl+shift+N", cd to your root workspace directory, and source it "source devel/setup.bash".

- We may also need to make lab2_exec.py executable.

```
$ chmod +x lab2_exec.py
```

- Run your node with the command rosrun in the new command prompt. Example of running lab2node node in lab2pkg package:

```
$ rosrun lab2pkg_py lab2_exec.py —simulator True
```

Note that the "--simulator True" tells it you are running on the simulator, while "--simulator False" tells the program you are running on real hardware. This flag is currently only applicable to Lab 2.

A.6   More Publisher and Subscriber Tutorial

Please refer to the webpage: `http://wiki.ros.org/ROS/Tutorials/Writing PublisherSubscriber(c%2B%2B)`