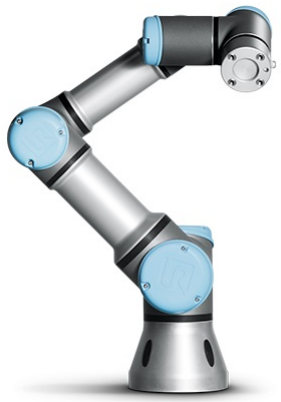
ECE 470
Introduction to Robotics
Lab Manual

ver 2.0



Dan Block
Jonathan K. Holm
Jifei Xu
Yinai Fan
Hang Cui
Yu Chen

University of Illinois at Urbana-Champaign

UR3 Python - ROS Interface

Contents

Preface

This is a set of laboratory assignments designed to complement the introductory robotics lecture taught in the College of Engineering at the University of Illinois at Urbana-Champaign. Together, the lecture and labs introduce students to robot manipulators and computer vision along with the Robot Operating System (ROS) and serve as the foundation for more advanced courses on robot dynamics, control and computer vision. The course is cross-listed in three departments (Electrical & Computer Engineering, Aerospace Engineering, and Mechanical Science & Engineering) and consequently includes students from a variety of academic backgrounds.

For success in the laboratory, each student should have completed a course in linear algebra and be comfortable with three-dimensional geometry. In addition, it is imperative that all students have completed a freshman-level course in computer programming. *MODERN ROBOTICS MECHANICS, PLANNING, AND CONTROL* (Kevin M. Lynch and Frank C. Park, 2017) is required for the lectures and will be used as a reference for many of the lab assignments. We will hereafter refer to the textbook as *MR* in this lab manual.

These laboratories are simultaneously challenging, stimulating, and enjoyable. It is the author's hope that you, the reader, share a similar experience.

Enjoy the course!

LAB 1

Introduction to the UR3

1.1 LAB 1 Week One

1.1.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

1.1.2 Objectives

The purpose of this lab is to familiarize you with the UR3 robot arm and its industrial programming interface called the teach pendant. In this lab, you will:

- Learn how to turn on and activate the UR3, and work with the teach pendant to create a simple program for the UR3
- Use the teach pendant to turn on and off the suction cup gripper and use the gripper in a program
- Demonstrate a sequence of motions that places one block on top of another.

1.1.3 References

- UR3 Owner's Manual:

  `https://www.universal-robots.com/download/?option=52870#section52851`
- UR3 Software Manual:

  `https://www.universal-robots.com/download/?option=53077#section53064`
- Universal Robots Academy

  `https://www.universal-robots.com/academy/`

1.1.4 Pre-Lab

Before you come to lab it is very important that you go through the training videos found at Universal Robots website `https://www.universal-robots.com/academy/`. These training sessions get into some areas that we will not be using in this class (for example you will not be changing safety settings), but go through all of the assignments as they will help you get familiar with the UR3 and its teach pendant. You also may want to reference these sessions when you are in lab.

1.1.5 Task

Using the teach pendant, each team will "program" the UR3 to pick and place blocks. The program may do whatever you want, but all programs must check three predefined locations for two blocks and stack one block on top of another at a fourth predefined position. You will use the gripper's suction feedback to determine if a block is located at one of the three starting block locations. The blocks must be aligned with each other in the stack of two.

1.1.6 Procedure

1. The Pre-Lab asked you to go through the basic UR3 training at Universal Robots website. This training should have shown you how to make simple programs to move the UR3. Initially your TA will demonstrate how to turn on and enable the UR3 as well as how to use the emergency stop button. Then use this lab time to familiarize yourself with the UR3 robot. First play around with simple programs that move the robot between a number of points.

2. To turn on the suction for the suction cup gripper, **Digital output 0** needs to be set high. Set low to turn off the suction. Also **Digital input 0** indicates if the suction cup is gripping something. It will return 1 if it is gripping an object and 0 if not. Modify your above program (or make a new one) to add activating on and off the suction cup gripper.

3. Create a program that defines four spots on the robot's table. Three of these spots are where it is possible a block will be initially located and with a certain orientation. There will only be two blocks. The user will place the blocks in two of the positions. The goal for the robot is to collect the two blocks and stack them on top of each other in the fourth define place on the robot's table. So you will need to use the suction cup gripper's feedback that indicates whether an object is being gripped or not. Then with some "**If**" instructions complete this task such that the user can put the two blocks in any of the three starting positions. When you are finished, you will demo your program to your TA showning that your program works when two blocks are placed and aligned in the three different configurations and also does not have a problem if only one block

or even no blocks are placed at their starting positions. Tips for creating this program:

- To turn on the suction cup, use the **Set** command and select **Digital Output 0** and turn it on or true. Set it to off or false to turn off the suction.
- **Digital Input 0** indicates if something has been gripped by the suction cup. Go to the **I/O** tab and turn on and off **Digital Output 0** and check which state of Digital Input 0 indicates gripped and upgripped.
- In the Structure tab under Advanced besides "**If ... else**", you may also want to use the Assignment to create a global worker variable that, for example, stores the number of blocks collected. In addition the **SubProg** item creates a subroutine that you may call when performing the same steps. The subroutine's scope allows it to see the variables you create with the **Assignment** item.
- You may want to name your **Waypoints**. This makes your program easier to read. In addition if the robot needs to go to the same point multiple times in your program you can command it to go to the same waypoint name.
- Under the Structure tab you can use the **Copy** and **Paste** buttons to copy a line of code and past it in a different subsection of your code. This cuts down on extra typing. Also note the **Move** up and down buttons along with the **Cut** and **Delete** buttons. Suppress is like commenting out a line of code.
- When you add an "**If**" statement and then click on the **Command** tab, tap in the long white box to pull up the keyboard for entering the if condition.

4. Demo this working program to your TA. Your TA may ask you to improve your positioning if the stack does not end up aligned well.

1.1.7 Report

None required. Look at Lab 1 Week Two and Start the longer reading assignment for Lab 2's pre-lab.

1.1.8 Demo

Show your TA the program you created.

1.1.9 Grading

- 10 points, completing the above tasks by the end of your two hour lab session.

- 90 points, successful demonstration.

1.2 LAB 1.5 Week Two, The Tower of Hanoi using the Teach Pendant

1.2.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

1.2.2 Objectives

This lab is numbered 1.5 because it continues the programming you learned in Lab 1 but also prepares you for Lab 2. In Lab 2 and forward you will be using the *Robot Operating System* (ROS) environment to program the UR3. For this lab you will continue to program the UR3 using its Teach Pendant but perform a similar task that will be required in Lab 2, solving a three block Tower of Hanoi puzzle. In this lab, you will:

- Move three stacked blocks from one position to another position using the rules specified for the Tower of Hanoi puzzle. Blocks should be aligned on top of each other.
- Use high level "**Move**" commands to move the UR3's Tool Center Point in linear and circular motions
- Time permitting play with other functionality of the teach pendant.

1.2.3 References

- UR3 Owner's Manual:

  `https://www.universal-robots.com/download/?option=52870#section52851`
- UR3 Software Manual:

  `https://www.universal-robots.com/download/?option=53077#section53064`
- Universal Robots Academy

  `https://www.universal-robots.com/academy/`
- 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.)

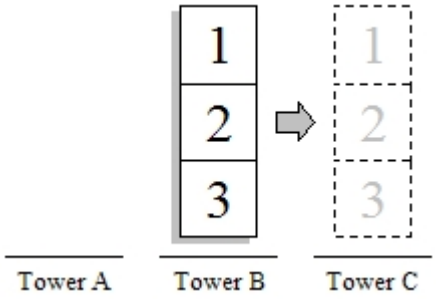


Figure 1.1: Example start and finish tower locations.

1.2.4 Pre-Lab

Read in more detail the UR3 Software Manual chapters 13 and 14. Additionally if for some reason you have not completed the training videos, go through the training videos found at Universal Robots website `https://www.universal-robots.com/academy/`. These training sessions get into some areas that we will not be using in this class (for example you will not be changing safety settings), but go through all of the assignments as they will help you get familiar with the UR3 and its teach pendant. You also may want to reference these sessions when you are in lab.

1.2.5 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 1.1. 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 1.2.

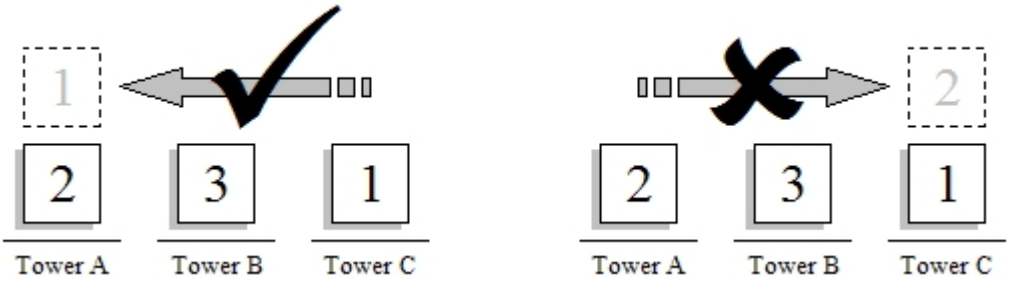


Figure 1.2: Examples of a legal and an illegal move.

1.2.6 Procedure

1. Choose the three spots on the robot's table where blocks can be placed when solving the Tower of Hanoi problem.
2. Use the provided colored 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.
3. Choose a starting position and ending position for the tower of three blocks. Future note: In Lab 2 the user will enter the start and stop positions.
4. Using the Teach Pendant create a program that solves the Tower of Hanoi problem. Instead of using **MoveJ** moves like in Lab 1, experiment with using **MoveL** and **MoveP** moves. **MoveL** moves the Tool Center Point (TCP) along a straight line, and **MoveP** is a process move that keeps the TCP moving at a constant speed and allows you to move along circular arcs. Reference these three "How To" articles from Universal Robots on creating circular arcs:
   - `https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/circle-using-movec-16270/`
   - `https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/circular-path-using-movepmovec-15668/`
   - `https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/circle-with-variable-radius-15367/`
5. Your program must have at least one obvious linear move and one obvious circular move that completely encircles one of the block positions.

1.2.7 Report

Each partner 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 are due one week after the final session of Lab 1.5 - before your lab session!
- Lab reports will be submitted online at GradeScope.

Your report should include the following:

- Briefly explain the rules of Towers of Hanoi (Introduction/Objective)
- Concisely explain your solution (Method)
- Discuss your circular movement and how you implemented it (Method)
- Note anything you learned about operating the robot (Conclusion)
  - How did you keep your block stacks neat?
  - Observations about **MoveJ**, **MoveL**, and **MoveP**?
- Make use of figures and tables as needed to aid in your explanation
- Read ECE 470: How to Write a Lab Report carefully so you know all the requirements

1.2.8 Grading

- 10 points, completed this section by the end of the two hour lab session.
- 70 points, successful demonstration.
- 20 points, report.

LAB 2

The Tower of Hanoi with ROS

2.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

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`
- A short tutorial for ROS by Hyongju Park. `https://sites.google.com/site/ashortrostutorial/`
- `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.)

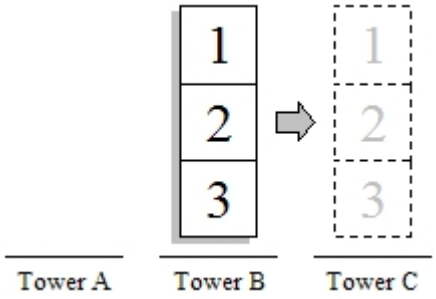


Figure 2.1: Example start and finish tower locations.

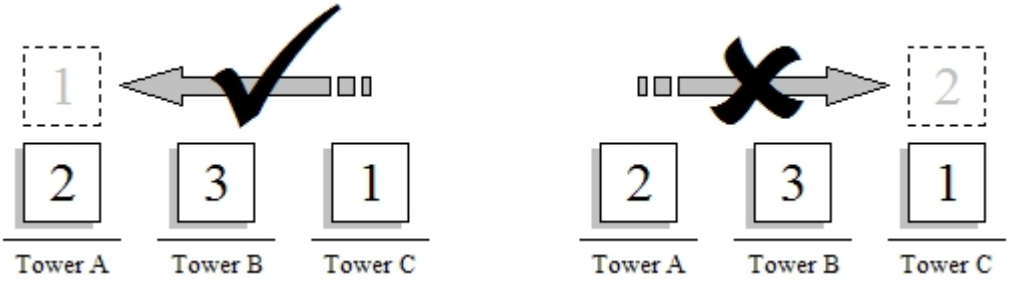


Figure 2.2: Examples of a legal and an illegal move.

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**.
   - Then run the lab2 ros node **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.

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.

```
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.

2.7.1 Report

Each partner 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 are due one week after the final session of Lab 2 - before your lab session!
- Lab reports will be submitted online at GradeScope.

Your report should include the following:

- You have already explained Towers of Hanoi and your solution, so we dont need to repeat that
- 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?
  - 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
- Re-Read ECE 470: How to Write a Lab Report carefully so you know all the requirements
- Include your **lab2_exec.py** code as an Appendix to your report

2.8 Demo

Your TA will require you to run your program twice; on each run, the TA will specify a different set of start and destination locations for the tower.

2.9 Grading

- 20 points, by the end of the two hour lab session, show your TA that your ROS program moving the UR3 to the three Tower of Hanoi stack locations, and it is able to subscribe to the ROS node publishing the input status of the suction cup gripper.
- 60 points, successful demonstration.
- 20 points, report.

LAB 3

Camera Sensing and Particle Filter

3.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

3.2 Motivation

In order to design an intelligent robot to effortlessly carry out tasks, the robot must first be able to estimate a "state" from sensor data. For example, if a robot needs to navigate a path out of a room, it first must figure out where it is relative to the exit. If an autonomous car wants to avoid a pedestrian, it first must gather detection data and figure out the relative position of the person. If we had perfect knowledge of the world, determining what to do is relatively straightforward. Unfortunately, there is a great deal of uncertainty (noise) in our sensors (and most robot environments), and most variables are not directly measurable.

State estimation is meant to address the problem of estimating quantities that are not perfectly or directly observable from sensor data. By gathering sensor data, we can design a state estimation scheme that can recursively update it's belief of the true state variables from the data. In this lab, you will implement a standard technique for localization, which will help the UR3 find itself in a maze.

So with this lab we are taking a leap ahead in what you know about moving a robot arm from one position (state) to another position. In fact, we will be

giving you code libraries that you you will be developing for yourself in Lab 4 and Lab 5. We are doing things in this order to give you motivation for all the linear algebra and geometry you will need to solve to develop labs 4 and 5. So you will not yet know how to move the robot from one position to the next but we will be giving you code to step the robot from one grid position to the next position.

Our first step in this lab is to measure the location (or state) of the end effector of the robot arm. You will be placing an orange circle at the end of the robot arm and you should think of this as a point that needs to move around a grid course in the camera's field of view. For this exercise we are not worried about whether the arm links position over the grid walls, we are only interested in the orange circle's position.

When trying to understand the pose or state of a robotic system, it is always important to keep in mind the error in the measurements and model of the system. In the homework particle filter simulation, noise is added to simulate this error. For our actual implementation we are going to use the camera installed above your robot's bench as our measurement of where the robot (orange disk) is located. This of course will introduce error in the measurement so the actual run will not have to add noise to the measurement.

Besides working with the particle filter in this lab, another important aspect of this lab is introducing you to the **OpenCV** computer vision library. For this lab we will be asking you to just use the **simpleBlobDetector** algorithm to find centroids of orange blobs. In lab 6 you will again be asked to use openCV to find centroids and that task can probably be done with **simpleBlobDetector** again. It is the hope of this lab though to show you that simple web searches of openCV topics can show you other algorithms/techniques to perform similar tasks. In lab 6 you will have the freedom to experiment more with openCV and even change the lab assignment to your liking to solve a different but similar problem than the one assigned. This will allow you to experiment with other algorithms/functions of openCV.

3.3 Objectives

- Become more familiar with ROS.
- Introduce the computer vision development library, **OpenCV**.
- Use the OpenCV class **simpleBlobdetector** to find the centroid of orange "blobs.
- Find the transformation from camera pixel coordinates to robot world coordinates and then in the final exercise transform to maze coordinates.

- Implement the same particle filter exercise assigned in homework but now instead of simulating the measurement of the orange blob/turtle robot's state use the camera to measure the coordinates (state) of the robot's position.

3.4 References

HSV Color Space:

- `https://en.wikipedia.org/wiki/HSL_and_HSV`
- https://stackoverflow.com/questions/10948589/

Simple Blob detector:

- `www.learnopencv.com/blob-detection-using-opencv-python-c/`
- https://stackoverflow.com/questions/8076889/how-to-use-opencv-simpleblobdetector
- `https://www.programcreek.com/python/example/89350/cv2.SimpleBlobDetector`
- `https://www.programcreek.com/python/example/71388/cv2.SimpleBlobDetector_Params`

3.5 Tasks

- Use the USB camera and the **OpenCV** library's **simpleBlobDetector** class to find blobs of orange color and locate their center coordinates in pixels.
- Determine the perspective transform of the camera. – Find the transformation matrix that transforms the centers found by the camera in pixels coordinates (row,column) to World robot coordinates (x_w, y_w) in meters. Normally perspective transform involves finding the intrinsic, extrinsic, and distortion parameters that describe the mapping between 3D world points to 2D image points. For our purposes we will make several assumptions that will vastly simplify the transformation:
  - The image plane is always parallel to the table (z_w and z_c are parallel).
  - The yaw angle difference between the world coordinates and camera coordinates is considerably small (x_w and x_c are almost parallel).
  - Fish eye distortion of the camera has been corrected.

Several parameters must be specified in order to implement the equations. Specifically, we are interested in θ the rotation between the world frame and the camera frame – which is expected to be fairly small since we aligned the camera before each lab section. β, the scaling constant between distances in the world frame and distances in the image. We will compute these parameters by measuring object coordinates in the world frame and relating them to their corresponding coordinates in the image.

- Use the measurement of the center of the orange disk attached to the robot's end effector as the measurement of the object traversing the maze of your particle filter homework. Now instead of knowing exactly where the vehicle (orange blob) is in the maze you will be measuring the state of the vehicle using the above camera.

3.6 Procedure

3.6.1 Threshold camera image to distinguish only bright orange pixels

1. Starter code for Lab 3 is given at the lab website `coecsl.ece.illinois.edu/ece470` under Lab 3. Download this **tar.gz** file to your catkin directorys **src** folder and then extract.

   Explore into the **lab3pkg_py/scripts** folder and double check the properties of **lab3_image_tf_exec.py**, **lab3_image_exec.py** and **lab3_move_exec.py** are set to executable.

2. Open a terminal window, change to your catkin directory and run **catkin_make**.

3. Now you can run the starter code.

   ```
   $ source devel/setup.bash
   $ roslaunch ur3_driver vision_driver.launch
   ```

   Open a second terminal window

   ```
   $ source devel/setup.bash
   $ rosrun lab3pkgpy lab3_image_tf_exec.py
   ```

   You should see two Windows open. One is the cameras video with an added horizontal black line and small red circle. Later in this lab you will use the horizontal line to straighten the mounting of the camera if it got moved or bumped. For the red dot towards the top left corner of the image look at the text printing in the terminal. A three number (H,S,V) value is printing. You can use this pixel value to help you find the range of H,S,V (Hue, Saturation, Value) values for an orange color.

   Side Note: What is the H,S,V color space? Please see the reference link to the Wikipedia page discussing color spaces. You are probably most

familiar with the R,G,B (red,green,blue) color space. Here each pixel of an image has a R,G,B value. With H,S,V colors are placed on a color wheel and you indicate which color you are interested in by selecting an angle range. In **OpenCV** the range of the angle is from 0 degrees to 180 degrees. (You will find other implementations that use 0-360). For example a blueish color would be in the angle range of 110 degrees to 130 degrees. The S value is a number between 0 and 255 with 0 being very white and washed out and 255 the full, clear color. The V value is also a number between 0 and 255 with 0 being very dark and shaded and 255 the full clear color. The main reason we use the H,S,V color space in this lab is that it makes finding a color with different shading and lighting conditions much easier.

4. At this point take a look at the given python files in the **lab3pkg_py/scripts** folder. This section will only be modifying the files **lab3_func.py** and **lab3_image_tf_exec.py**. Take a look at these two files. Notice in **lab3_func.py**, in the function **blob_search**, the code is converting the RGB image to HSV. Then using a range for blueish pixels the **inRange** function is used to threshold/mask the image to a binary image of only blueish pixels. (Binary image means all blueish pixels are ones and all other pixels are zeros). Next the masked image is cropped to only use the portion of the image that views the maze. (You will adjust the size of this cropped image in a few steps). Then at the bottom of the function both the color image and the cropped binary image are displayed in separate windows. (Notice in those windows there is a button that allows you to save the current image to a file.)

5. So your first goal is to find the range of Hue, Saturation and Value that allows your program to find the orange circle in all spots of the cameras view. To find these HSV ranges you will need to do some trial and error. You are supplied with two tools to help you find the correct range. One is the red dot printed for you on the cameras video frames. Notice that when you put a colored block or the orange circle under that red dot the terminal window prints out the H,S,V value for that pixel. Experiment with a few colors and see the H,S,V values change. The code for putting this red dot on the image and displaying the H,S,V values to the terminal is in the **lab3_func.py** file towards the end of the **blobsearch** function. The **cv2.circle** function is used to add the red dot and the print statement prints out the H,S,V values. One issue with this method is that it only shows one pixel and therefore only one lighting condition. You need to make sure your range of H, S, V values work in all areas of the cropped image. So going on from here, you need to do some trial and error. Pick a range for H and S and V, change your code with that range and see how well the cropped image shows only the orange disk in all of its area. We have also supplied you with some off line code, **HSV.py**, that takes a single picture file and prints out a H,S,V range for the pixels selected.

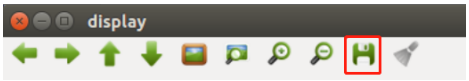


Figure 3.1: How to snapshot the video feed.

6. Once you have found a good H,S,V range for the orange disk and your benches lighting conditions, you need to make sure the cropped image views all the area of the maze. First step is to look at the full color image and make sure the camera is turned so the black horizontal line is also horizontal with respect to the robots base. Have your TA help you with this. (Of course try your best not to bump the camera so that this process does not have to happen too often.). Then return to looking at the cropped binary image and move the orange disk to all corners of the maze. Make sure that the orange disk is seen in the cropped image at all the corners. If the cropped image does not see all corners of the maze you will need to modify the values **crop_top_row**, **crop_bottom_row**, **crop_top_col** and **crop_bottom_col** to move where the top and bottom corners of the cropped image are located. Note that the full image size is 480 rows by 640 columns.

3.6.2 Use the OpenCV simpleBlobdetector library function to find the centroid in pixels of one or multiple circular orange blobs

1. At this point you need to do a bit of reading at the **OpenCV** websites listed in the Reference section and perform your own web search to figure out how use **OpenCV**s **simpleBlobdetector** library. Part of the goal of this assignment is learn **OpenCV** and use web searches to find examples of different library functions.

   For this lab you should only us the **simpleBlobdetector** algorithm but in Lab 6, if you wish, you will be allowed to use additional **OpenCV** features to recognize objects seen by the camera. You are given starter code to get you started with **simpleBlobdetector**. See in **lab3_func.py** the function **blob_search_init**. This is where you need to set the parameters for **simpleBlobDetector**. By default all the filterBy parameters are set to False. You need to decide which of these filters will work best to find the orange disks in the cropped image. There are also other parameters you will need to set in this function which are not given to you. Once you think you have set all the correct parameters, switch to the **blob_search** function and find the comment that shows you where to call the detect method of **simpleBlobDetector**.

   The detect method returns **keypoints**, which is a variable that has the centroid and other information of each blob found. If there is only one orange blob then **keypoints** only has one element. If there are more blobs

then **keypoints** also has the centroids for those blobs. The web pages listed in the References section show example code of getting information out of the **keypoints** variable. Use the **print()** function to print out the number of blobs found and the coordinates of the first blob. Also use the **cv2.drawKeypoints()** function to draw a circle around each blob in the color video frames. **simpleBlobDetector** is being passed the cropped image. The centroid found is in pixel units. (0, 0) is the top left corner of the cropped image. Therefore you will need to add **crop_top_row** and **crop_top_col** to the **Keypoint**'s centroid values before calling the **cv2.drawKeypoints()** function with the full color image. **Keypoints** is a list of the **Keypoint** class. One of the elements in the **keypoint** class is pt, which is a tuple. To modify an element in a tuple, first change the tuple to a list with the **list()** function. Modify the element of the newly created list and finally set the **Keypoint.pt** tuple element to the modified list by converting the list to a tuple with the **tuple()** function. Figuring out the correct parameters for **simpleBlobDetector** will take some trial and error.

2. Using two orange disks, move them to many different locations in the maze and print out the centroids found. Make sure that these centroids make sense in pixels where the (0, 0) camera origin is the top left corner.

3.6.3 Perspective Transform

1. Read appendix C.3 before proceeding further.

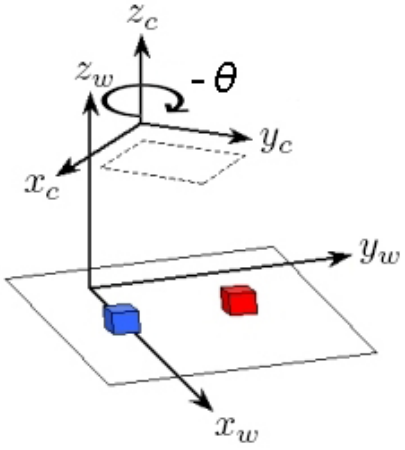


Figure 3.2: Arrangement of the world and camera frames.

2. Begin by writing the equations we must solve in order to relate image and world frame coordinates. You will need to combine the intrinsic and extrinsic equations for the camera; these are given in appendix C.3. Write

equations for the world frame coordinates in terms of the image coordinates.

$$\begin{aligned} x_w(r,c) &= \\ y_w(r,c) &= \end{aligned}$$

3. There are six unknown values you must determine: $O_r, O_c, \beta, \theta, T_x, T_y$. The principal point (O_r, O_c) is given by the row and column coordinates of the center of the image. We can easily find these values by dividing the `width` and `height` variables by 2.

$$\begin{aligned} O_r &= \frac{1}{2} height = \\ O_c &= \frac{1}{2} width = \end{aligned}$$

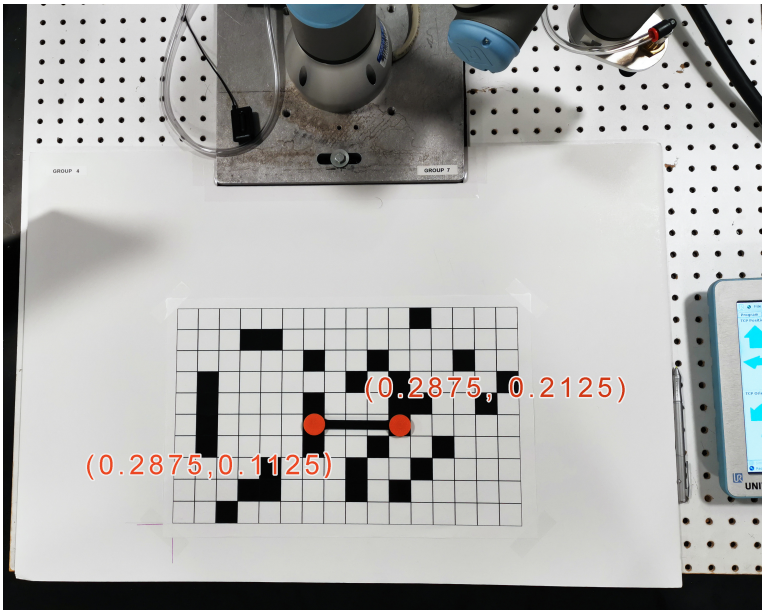
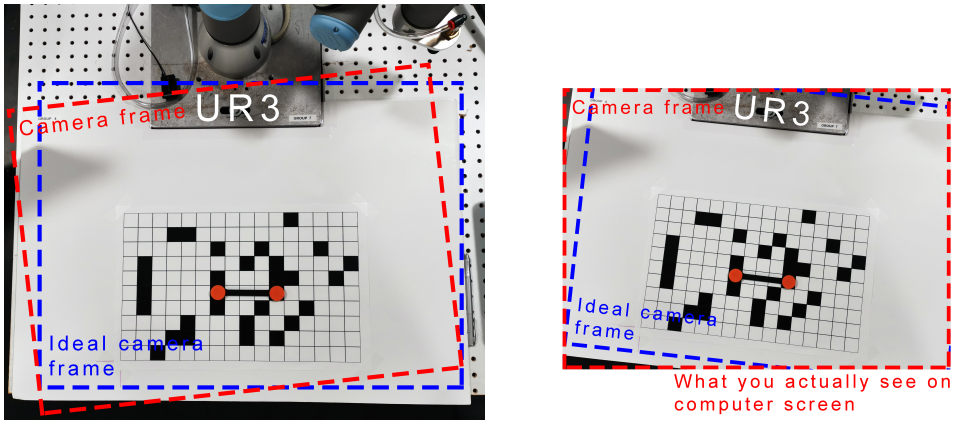


Figure 3.3: Where to place the calibration stick.

You will calculate the parameters β, θ, T_x, T_y in the python script:

lab3_image_tf_exec.py

First, place the calibration stick in the position ($x_w = 0.2875$, $y_w = 0.1125$ and $x_w = 0.2875$, $y_w = 0.2125$) shown in the Figure 6.3. Please keep anything orange (including your hands) away from the camera's field of view. Open a terminal window:



(a) (b)

Figure 3.4: (a) Overhead view of the table; the rectangle delineated by blue dashed lines represents the desired camera's field of view, while the red dashed lines represents the actual camera's field of view (b) The actual image seen by the camera. Notice that x_c increases with increasing row values, and y_c increases with increasing column values.

```
$ source devel/setup.bash
$ roslaunch ur3_driver vision_driver.launch
```

Open a second terminal window:

```
$ source devel/setup.bash
$ rosrun lab3pkgpy lab3_image_tf_exec.py
```

You will modify this code (which will print these values to the terminal by default as zeros) to calculate β, θ, T_x, T_y values so that you can use them in **lab3_image_exec.py**.

Note that you may have to rerun this procedure to obtain these values when you return to the lab, as other groups will be using the same camera and may re-position the camera when you are not present. To keep these values relatively consistent, **PLEASE TRY NOT TO BUMP OR MOVE THE CAMERA.**

β is a constant value that scales distances in space to distances in the image. That is, if the distance (in unit length) between two points in space is d, then the distance (in pixels) between the corresponding points in the image is βd. Note that the calibration stick is exactly 0.1 m long.

$$\beta =$$

θ is the angle of rotation between the world frame and the camera frame. Figure 6.4 gives an overhead view of the blocks with a hypothetical cutout

representing the image captured by the camera. Because the camera's x and y axes are not quite parallel to the world x and y axes, the blocks appear rotated in the image. You may expect this parameter to be fairly small.

$$\theta =$$

T_x, T_y (unit in meters), the origin of the world frame expressed in the camera frame.

Substitute these values into the two equations you derived in step 2 above and solve for the unknown values.

$$\begin{aligned} T_x &= \\ T_y &= \end{aligned}$$

3.6.4 Implement particle filter HW problem with real world vision measurements

This section of the lab is an extension of the particle filter homework problem assigned in lecture. You will need to have the homework problem completed before attempting this section. In the homework problem, the location of the turtle was the center of a grid with some additional noise added to the position to make the simulation a bit more realistic. Using the USB camera viewing the robots work environment, we will introduce a measurement of the x, y position of a point at the end of the robots gripper. The below steps will walk you through moving your python code from your homework code into the **lab3_move_exec.py** file. These are the TODO sections of the homework.

1. The **Maze** class does not need any changes or additions
2. The **default_2D_Model** class also needs no changes but do note that there is a change from the homework with the **readingSensor** function. **readingSensor** is now passed the camera measured centroid of the detected orange blob.
3. In the **particleFilter** class **__init__** function add your code to create the initial set of particles and weights.
4. In the **Sample_Motion_Model** method, add your homework code that samples the motion model to propagate the particles.
5. In the **Measurement_Model** method, add your homework code that reads the **sensorReading** and calculates weights for each particle given the measurement. Make sure the weights are normalized.
6. In the **calcPosition** method, find the estimated position of the robot given the particles and their weights. **calcPosition** returns the estimation.

Once all these sections of code are filled in with your homework solution, you are ready to try out the program.

- You will need to first run **lab3_image_exec.py** which publishes the centroid of the orange blob, **rosrun lab3pkg_py image_exec.py**.
- Then in another terminal run **lab3_move_exec.py**, **rosrun lab3pkg_py move_exec.py**.
- After a few seconds the robot arm will move the orange dot on its gripper to the 0,0 grid position. The world location of that grid in meters is approximately (x=0.3875, y=-0.0375). Do a search in **lab3_move_exec.py** and find the variables **gridzero_inRobotWorld_x** and **gridzero_inRobotWorld_y**. You will see that x is set to 0.3875 but y is -0.0405, slightly different from -0.0375. On one of the benches we found that this coordinate brought the orange circle directly over the 0,0 grid. If you robots orange circle does not position itself directly over grid 0,0 adjust (**gridzero_inRobotWorld_x**, **gridzero_inRobotWorld_y**) appropriately.
- Now you are ready to try out your code. Use the arrows to move the orange circle through the course. How well does your estimate stay with the actual position of the robot? Does it take a number of moves before your particle filter converges close to the correct position? Do you notice any differences between your homework simulation and this run using actual coordinates measured by the above camera? Try moving the robot through the same path as the homework takes and then also experiment with taking other paths.

3.7 Report

Each partner will submit a lab report using the guidelines given in the ECE470: How to Write a Lab Report document. Note:

- Lab reports are due one week after the final session of Lab 3 - before your lab session.
- Lab reports will be submitted online at GradeScope

Your report should include the following:

- What were the three major topics this lab focused on?
  - Steps you took, challenges you had to find the HSV range for the bright Orange color.
  - Overview of the **OpenCV simpleBlobdector** class, and the parameters of the class you used. Also any parameter of the **simpleBlobdector** that you tried but did not help or you could not figure out.

  - Steps you took to find the transformation equations from the cameras pixel coordinates to the UR3s world coordinates. Were there any difficulties you ran into?
  - Steps you took to move your particle filter HW solution into the **lab3_move_exec.py** file.
  - How well did your particle filter work given coordinates found by the USB camera?
  - Screen shots of your filter at work are recommended to help explain your observations.

- Make use of code snippets as needed to aid in your explanation.
- Include your entire **lab3_func.py, lab3_image_exec.py, lab3_image_tf_exec.py** files in an appendix. Additionally include the parts of **lab3_move_exec.py** that you changed in this same appendix.

3.8 Demonstration

1. Demonstrate your **lab3_image_exec_tf.py** code converting the colored image to a binary image and then calling the **simpleBlobDector** class to find centroids of all oranges blobs of color. In addition demonstrate the calculation of the **beta, tx** and **ty** parameters.
2. Demonstrate your **lab3_image_exec.py** code displaying the world coordinates of a single orange object in the grid course. Make sure that it is able to find the orange object in all spots of the course.
3. Demonstrate your particle filter working.

3.9 Grading

- 30 points, by the end of each of the three weeks allotted for this lab, your TA will give you 10 points for attending lab and working diligently on the lab assignment.
- 50 points, successful demonstration
- 20 points, report.

LAB 4

Forward Kinematics

4.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

4.2 Objectives

The purpose of this lab is to implement the exponential forward kinematics on the UR3 robot to estimate the end-effector pose given a set of joint angles. In this lab you will:

- Solve the exponential forward kinematic equations for the UR3.
- Write a Python function that moves the UR3 to a configuration specified by the user.
- Compare your exponential forward kinematic estimation with the actual robot movement.

4.3 References

- Chapter 4 of *Modern Robotics* provides details of how to construct an exponential forward kinematic for an open-chain robot.

4.4 Tasks

4.4.1 Theoretical Solution

Find the forward kinematic equations for the UR3 robot using the exponential forward kinematics method. Solve for T_6^0 and record the 3-by-1 translation

vector from the end-effector frame to the base frame, d_6^0.

4.4.2 Physical Implementation

The user will provide six joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, all given in degrees. The angle ranges are as follows:

$$\begin{aligned} -45^\circ < \theta_1 < 45^\circ \\ -180^\circ < \theta_2 < 0^\circ \\ -20^\circ < \theta_3 < 140^\circ \\ -170^\circ < \theta_4 < -5^\circ \\ -140^\circ < \theta_5 < 70^\circ \\ 180^\circ < \theta_6 < 0^\circ \end{aligned}$$

Then using a ROS Python program move the joints to these desired angles. Your program should additionally calculate and display the translation matrix T_6^0 that rotates and translates the end effector's coordinate frame to the base frame.

4.4.3 Comparison

For any provided set of joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, roughly compare with a ruler and square the position of the suction cup gripper to the value calculated by your Python forward kinematics function.

4.5 Procedure

4.5.1 Theoretical Solution

1. In exponential forward kinematics (especially for an open chain kinematics), there are three components that are essential for the calculation.

   - A fixed base frame f_0
   - Robots zero configuration (or some might call it initial position)
   - A well-defined end-effector frame f_6

   Virtually, You can define the base frame, the end-effector frame, and zero configuration arbitrarily. As long as you remain consistent with your definitions, the end results will be identical for the same robot. However, doing so will post difficulties when your TA tries to verify your answer or debug your process. Therefore we encourage you to use the same frame placement and zero configuration for both convenience and safety purposes. See Figure 4.1.

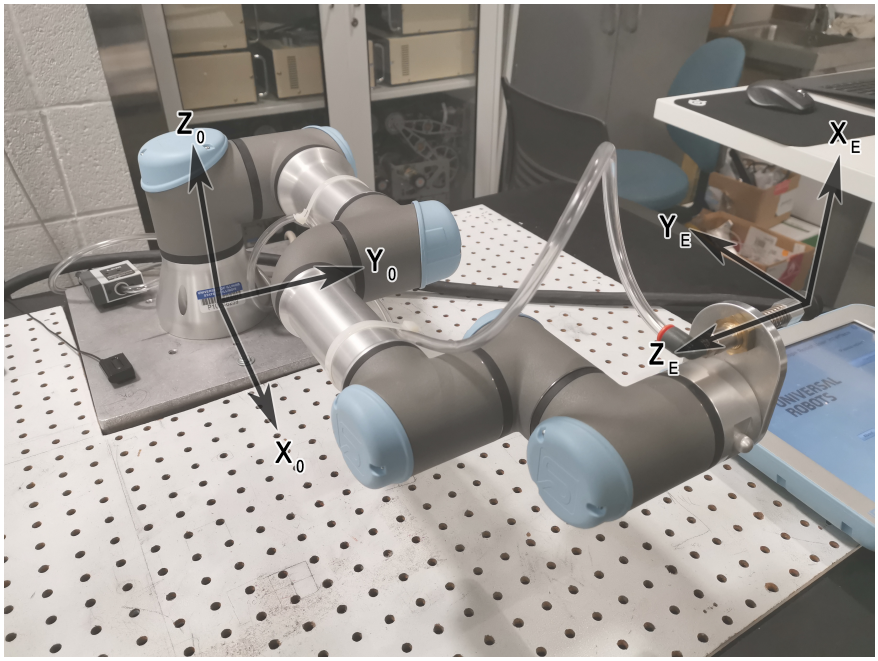


Figure 4.1: zero configuration and frame placement for UR3.

Moreover, please note that in this lab, all the UR3 robots have a predefined zero configuration. The pose shown in Figure 4.1 is **NOT the TRUE zero configuration** according to the teach pendant. In this pose (Figure 4.1), you would read the following joint angles from **the teach pendant**:

Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
180.0°	0.0°	0.0°	-90.0°	0.0°	0.0°

In the **teach pendant's** zero configuration, (due to how the robot is originally mounted) the robot arm would face the far edge of the table and the wrist joint 2 would point into the table, which is an awkward pose to make measurements and do experiments. Hence, we added some angle offsets in the function script such that:

```
return_value[0] = theta1 + PI
return_value[1] = theta2
return_value[2] = theta3
return_value[3] = theta4 - (0.5*PI)
return_value[4] = theta5
return_value[5] = theta6
```

Any joint angle measurements from the robot that are passed to your ROS program will be corrected to the zero configuration shown in Figure 4.1. In other words, now if you set all the joint of UR3 to zero (zero configuration) in **ROS**:

Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
0.0°	0.0°	0.0°	0.0°	0.0°	0.0°

You would see the UR3 poses as Figure 4.1 showed.

2. The sole purpose of the forward kinematics algorithm is to estimate the end-effector's pose (orientation and position) with respect to the base frame given a set of joint angles. This information can be packed and calculated in a homogeneous transformation matrix taught by Lynch Park in *Modern Robotics*:

$$\mathrm{T}(\theta) = e^{[S_1]\theta_1} e^{[S_2]\theta_2} ..... e^{[S_{n-1}]\theta_{n-1}} e^{[S_n]\theta_n} M$$

While our objective is to solve for the homogeneous transformation matrix $T(\theta)$ of the motion caused by certain joint angle configurations, there are seven matrices we need to specify:

- The screw axes S_1 to S_6 (we have six here because UR3 has six joints) expressed in the base frame, corresponding to the joint motions when the robot is at its zero configuration (aka Figure 4.1)
- The end-effector pose M (a 4x4 homogeneous transformation matrix) with respect to the base frame, corresponding to the zero configuration

A screw axis S can be represented by $S = (\omega, v)$, where the ω is the joint axis defined by the **right hand rule** (Figure 4.2). v can be computed by $v = -\omega \times q$, where q is the displacement (position) of the joint center with respect to the base frame.

4.5.2 Implementation on UR3

1. Download and extract **lab4Py.tar.gz** into the "**src**" directory of your catkin directory. Don't forget "**source devel/setup.bash**". Then from your base catkin directory run "**catkin_make**" and if you receive no errors you copied your lab4 starter code correctly. Also so that ROS registers this new package "**lab4pkg_py**", run "**rospack list**" and you should see **lab4pkg_py** as one of the many packages.

2. You will notice that in **lab4pkg_py/scripts** there are now three *.py files. **lab4_exec.py** is the main() code, **lab4_func.py** defines the important forward kinematic functions. **lab4_header.py** defines the header files. We divide them up in this fashion so that the forward kinematic functions

Figure 4.2: The "right hand rule"

can easily be used in the remaining labs. This way you will be creating, in a sense, a library that you can include in your next labs to call the forward kinematic calculations. In this lab you will be mainly editing **lab4_func.py**. You can of course change **lab4_exec.py** but most of its functionality has already been given to you. Study the code and comments in **lab4_exec.py** and **lab4_func.py** to see what the starter code is doing and what parts you are going to need to change. Your job is to add the code in function **Get_MS()** that correctly populates the six screw axes S_1 to S_6 as well as the transformation matrix M. The Python code uses the "numpy" module to create and multiply matrixes; meanwhile, matrix exponential "expm()" is achieved by including "Scipy" module. Then, with the S and M values you populated, write the code in function **lab_fk()** that generates and prints the homogeneous transformation matrix $T(\theta)$.

3. Once your code is finished and compiled, run it using " **rosrun lab4pkg_py lab4_exec.py [theta1] [theta2] [theta3] [theta4] [theta5] [theta6]**" with all angles in degrees. Remember that in another command prompt you should have first ran roscore and drivers using " **roslaunch ur3_driver ur3_driver.launch**"

4.5.3 Comparison

1. Your TA will select two sets of joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ when you demonstrate your working ROS program.

2. Run your ROS node and move the UR3 to each of these configurations.

With a ruler and square, measure the x,y,z position vector of the center of the gripper for each set of angles. (Note: The aluminum base that UR3 sits on is a 300mm x 300mm square. You can use it as your reference if you find measuring from the base of UR3 being difficult.) Call these vectors r_1 and r_2 for the first and second sets of joint angles, respectively.

3. Compare these hand measurements to the translation vector calculated by your ROS node. Call these calculated vectors d_1 and d_2.

4. For each set of joint angles, Calculate the error between the measured and calculated kinematic solutions. We will consider the error to be the magnitude of the distance between the measured center of the gripper and the location predicted by the forward kinematic equation:

$$error_1 = \|r_1 - d_1\| = \sqrt{(r_{1x} - d_{1x})^2 + (r_{1y} - d_{1y})^2 + (r_{1z} - d_{1z})^2}.$$

A similar expression holds for $error_2$.

4.6 Report

Each partner 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 are due one week after the final session of Lab 4 - before your lab session!
- Lab reports will be submitted online at GradeScope.

Your lab report should include the following:

- Coversheet containing your name and then your partner's names, "Lab 4", and the weekday and time your lab section meets (for example, "Tuesday, 3pm").
- Explain how you determined $\mathcal{S}_1$ thru $\mathcal{S}_6$ and $\mathbf{M}$ - include figures as needed.
- Include a figure that shows all frames and axes used.
- Include a table of all ω and q used and the v derived.
- Use Matlab's symbolic toolbox to generate the final homogeneous transform T_6^0 as a function of $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$ and include the calculated d_6^0 in your report.

$$T_6^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = \begin{bmatrix} R_6^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) & d_6^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) \\ 0\ 0\ 0 & 1 \end{bmatrix}$$

- Substitute in your given joint angles to verify your solution.

- For each test point, include:
  - The given $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$
  - The calculated d and measured r vectors
  - The scalar error
- Include a brief discussion of sources of error.

As appendices to your report, include the following:

- Your **lab4_func.py** code and **lab4_exec.py** if it was edited.
- Your Matlab code used to calculate T_6^0.

4.7 Demonstration

Your TA will require you to run your program twice, each time with a different set of joint variables.

4.8 Grading

- 10 points, by the end of the first two hour lab session, show your completed **Get_MS()** function to your TA.
- 10 points, by the end of the second two hour lab session, show your TA an attempt of your code trying to move the robot to a desired position.
- 60 points, successful demonstration.
- 20 points, individual report.

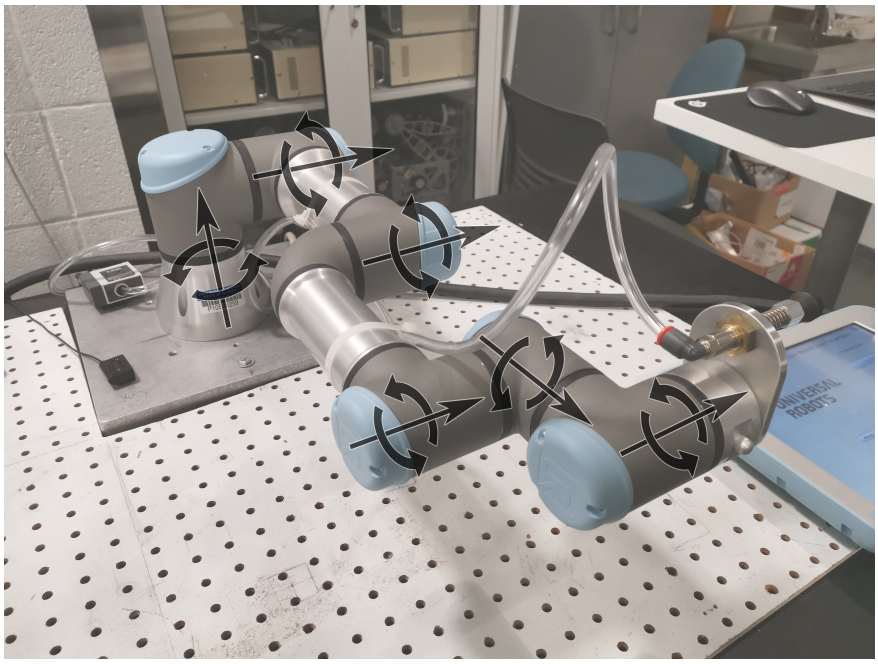


Figure 4.3: Screw axes on UR3.

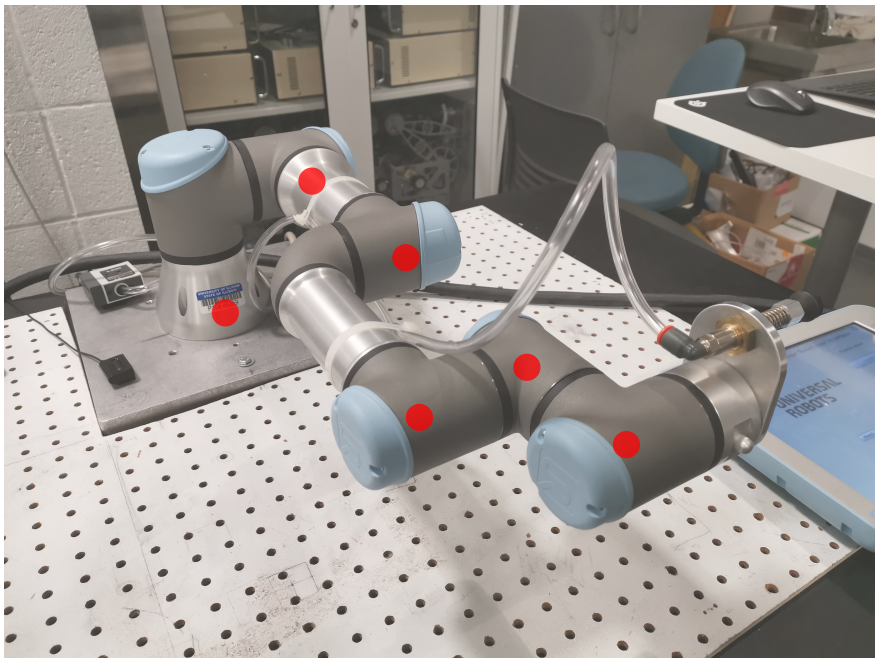


Figure 4.4: Joint center locations on UR3.

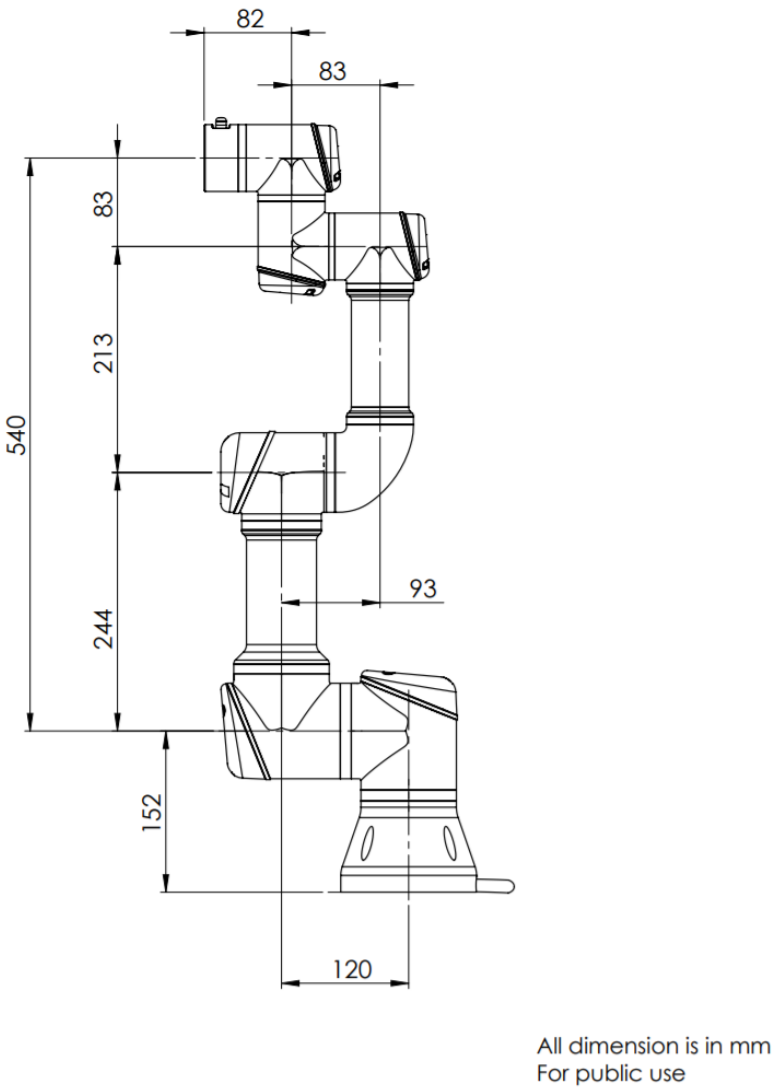


Figure 4.5: Approximate Dimensions of the UR3 in mm. *EACH UR3 IS SLIGHTLY DIFFERENT DUE TO ITS MANUFACTURE TOLERANCE.

LAB 5

Inverse Kinematics

5.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

5.2 Objectives

The purpose of this lab is to derive and implement a solution to the inverse kinematics problem for the UR3 robot. In this lab we will:

- Derive elbow-up inverse kinematic equations for the UR3
- Write a Python function that moves the UR3 to a point in space specified by the user.

5.3 Reference

Chapter 6 of *Modern Robotics* provides multiple examples of inverse kinematics solutions.

5.4 Tasks

5.4.1 Solution Derivation

Make sure to read through this entire lab before you start into the Solution Derivation section. There are some needed details not covered in this section.
Given a desired end-effector position in space $(x_{grip}, y_{grip}, z_{grip})$ and orientation $\{\theta_{yaw}, \theta_{pitch}(fixed), \theta_{roll}(fixed)\}$, write six mathematical expressions that yield

values for each of the joint angles. For the UR3 robot, there are many solutions to the inverse kinematics problem. We will implement only one of the *elbow-up* solutions.

- In the inverse kinematics problems you have examined in class (for 6 DOF arms with spherical wrists), usually the first step is to solve for the coordinates of the wrist center. The UR3 does not technically have a spherical wrist center but we will define the wrist center as z_{cen} which equals the same desired z value of the suction cup and x_{cen}, y_{cen} are the coordinates of θ_6's z axis. In addition, to make the derivation manageable, add that θ_5 will always be $-90°$ and θ_4 is set such that link 7 and link 9 are always parallel to the world x,y plane.

- Solve the inverse kinematics problem in the following order:

  1. x_{cen}, y_{cen}, z_{cen}, given yaw desired in the world frame and the desired x,y,z of the suction cup. The suction cup aluminum plate (link 9) has a length of 0.0535 meters from the center line of the suction cup to the center line of joint 6. Remember that this aluminum plate should always be parallel to the world's x,y plane. See Figure 5.2.
  2. θ_1, by drawing a top down picture of the UR3, Figure 5.1, and using x_{cen}, y_{cen}, z_{cen} that you just calculated.
  3. θ_6, which is a function of θ_1 and yaw desired. Remember that when θ_6 is equal to zero the suction cup aluminum plate is parallel to link 4 and link 6.
  4. x_{3end}, y_{3end}, z_{3end} is a point off of the UR3 but lies along the link 6 axis, Figure 5.1. For example if $\theta_1 = 0°$ then $y_{3end} = 0$. If $\theta_1 = 90°$ then $x_{3end} = 0$. First use the top down view of the UR3 to find x_{3end}, y_{3end}. One way is to choose an appropriate coordinate frame at x_{cen}, y_{cen} and find the translation matrix that rotates and translates that coordinate frame to the base frame. Then find the vector in the coordinate frame you chose at x_{cen}, y_{cen} that points from x_{cen}, y_{cen} to x_{3end}, y_{3end}. Simply multiply this vector by your translation matrix to find the world coordinates at x_{3end}, y_{3end}. For z_{3end} create a view of the UR3, Figure 5.2, that is a projection of the robot onto a plane perpendicular to the x,y world frame and rotated by θ_1 about the base frame. Call this the side view. Looking at this side view you will see that z_{3end} is z_{cen} offset by a constant.
  5. θ_2, θ_3 and θ_4, by using the same side view drawing just drawn above to find z_{3end}, Figure 5.2. Now that x_{3end}, y_{3end}, z_{3end} have been found use sine, cosine and the cosine rule to solve for partial angles that make up θ_2, θ_3 and θ_4. Hint: In this side view, a parallel to the base construction line through joint 2 and a parallel to the base construction line through joint 4 are helpful in finding the needed partial angles.

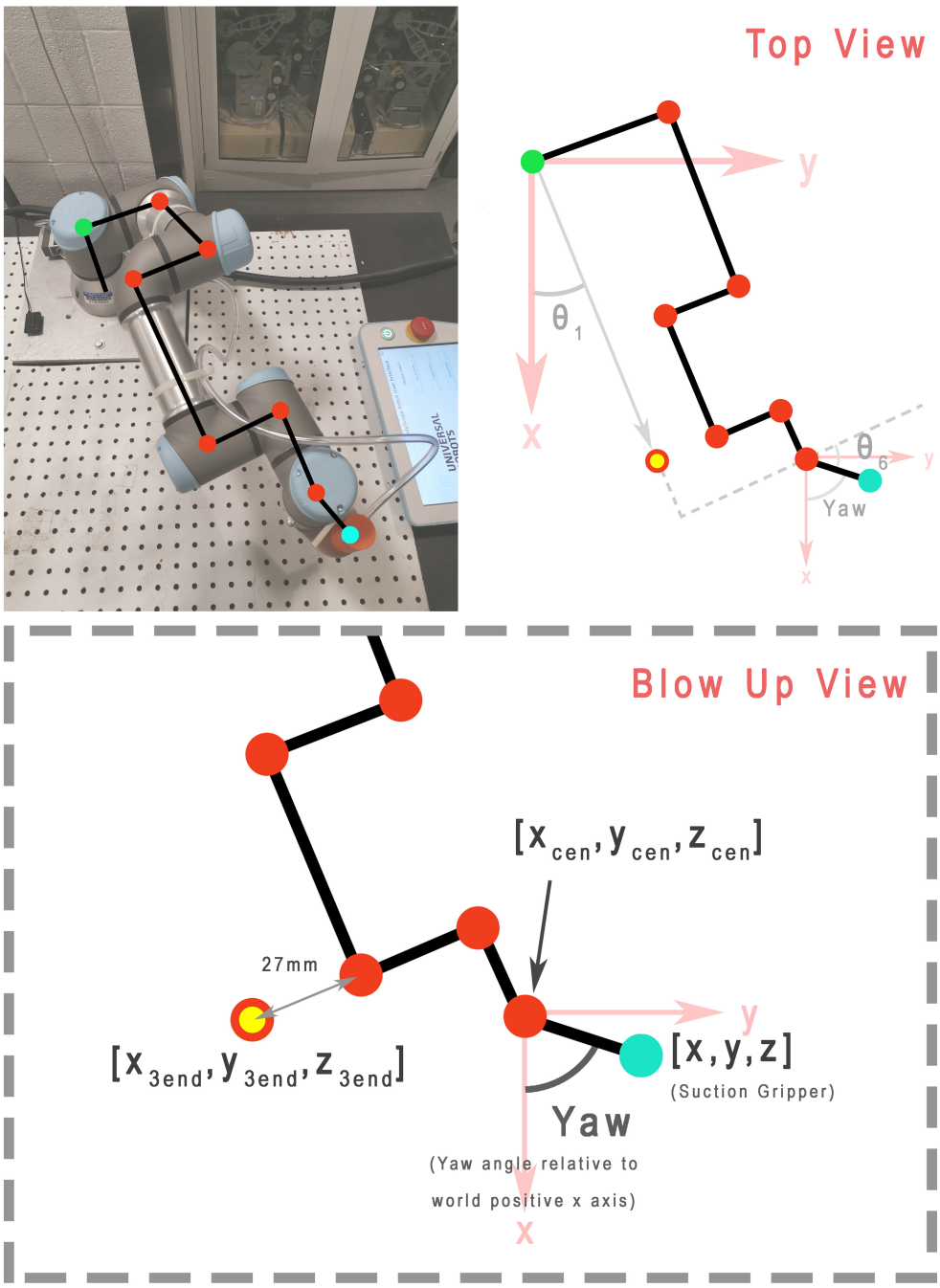


Figure 5.1: Top View Stick Pictorial of UR3. Note that the coordinate frames are in the same direction as the World Frame but not at the World frame's origin. One origin is along the center of joint 1 and the second is along the center of joint 6.

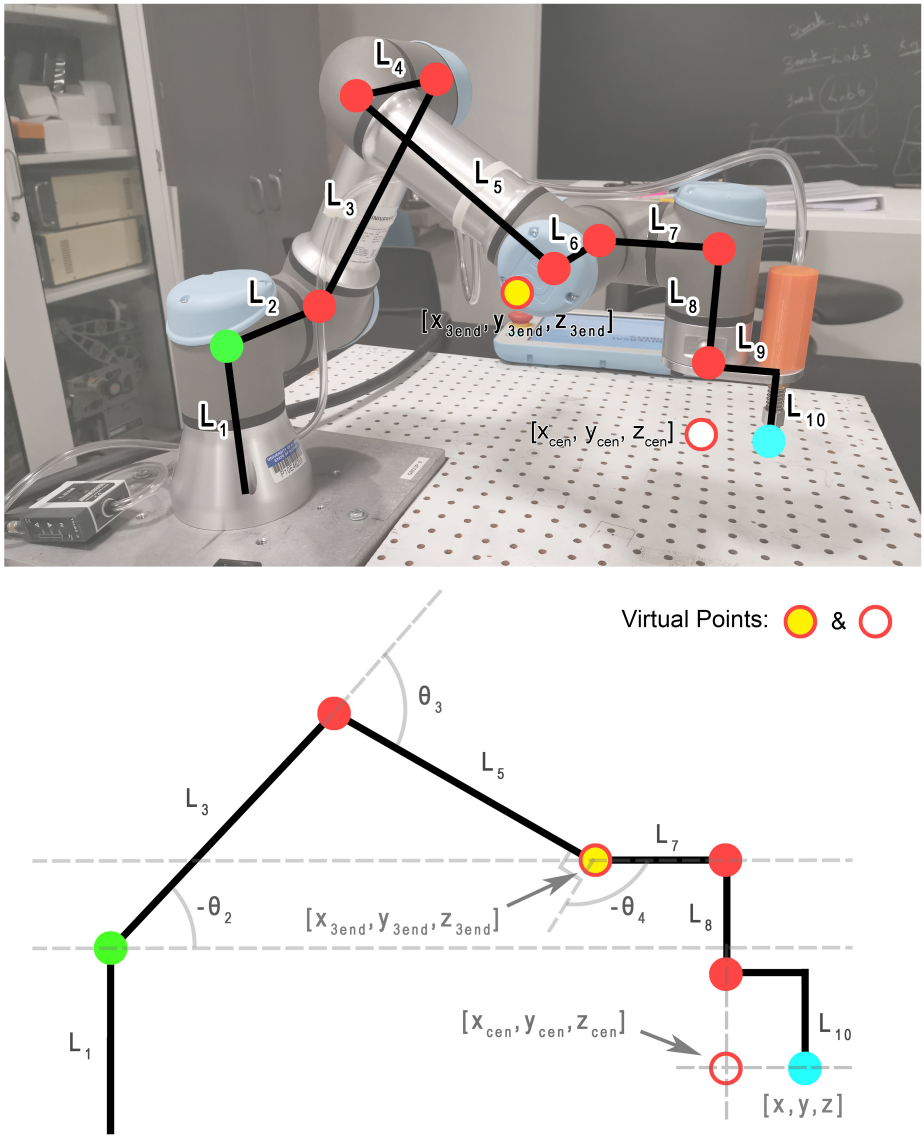


Figure 5.2: Side View Stick Pictorial of UR3.

5.4.2 Implementation

Implement the inverse kinematics solution by writing a Python function to receive world frame coordinates $(x_{Wgrip}, y_{Wgrip}, z_{Wgrip}, yaw_{Wgrip})$, compute the desired joint variables $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, and command the UR3 to move to that pose using functions written in Lab4.

5.5 Procedure

- Download **lab5Py.tar.gz** from the course website and extract it in your "**src**" directory. You will notice that there are three .py files. **lab5_exec.py**, **lab5_func.py** and **lab5_header.py**. The **lab5_func.py** file again will be compiled into a library so that future labs can easily call the inverse kinematic function. Like Lab 4, most of the needed code is given to you in **lab5_exec.py**. Your main job will be to add all the inverse kinematic equations to **lab5_func.py**. Please refer to the intermediate steps below to perform the inverse kinematic calculations. If you look at **lab5_header.py** it includes **lab4_header.py**. This allows you to call the functions you created in **lab4_func.py**.

- In your code (This is repeating the derivation steps above):

  1. Establish the world coordinate frame (frame w) centered at the corner of the UR3's base shown in Figure 5.3. The x_w and y_w plane should correspond to the surface of the table, with the x_w axis parallel to the sides of the table and the y_w axis parallel to the front and back edges of the table. Axis z_w should be normal to the table surface, with up being the positive z_w direction and the surface of the table corresponding to $z_w = 0$.
     We will solve the inverse kinematics problem in the base frame (frame 0), so we will immediately convert the coordinates entered by the user to base frame coordinates. Write three equations relating coordinates $(x_{Wgrip}, y_{Wgrip}, z_{Wgrip})$ in the world frame to coordinates $(x_{grip}, y_{grip}, z_{grip})$ in the base frame of the UR3.

$$x_{grip}(x_{Wgrip}, y_{Wgrip}, z_{Wgrip}) =$$
$$y_{grip}(x_{Wgrip}, y_{Wgrip}, z_{Wgrip}) =$$
$$z_{grip}(x_{Wgrip}, y_{Wgrip}, z_{Wgrip}) =$$

  2. Given the desired position of the gripper $(x_{grip}, y_{grip}, z_{grip})$ (in the base frame) and the yaw angle, find wrist's center point $(x_{cen}, y_{cen}, z_{cen})$.

$$x_{cen}(x_{grip}, y_{grip}, z_{grip}, yaw) =$$
$$y_{cen}(x_{grip}, y_{grip}, z_{grip}, yaw) =$$
$$z_{cen}(x_{grip}, y_{grip}, z_{grip}, yaw) =$$

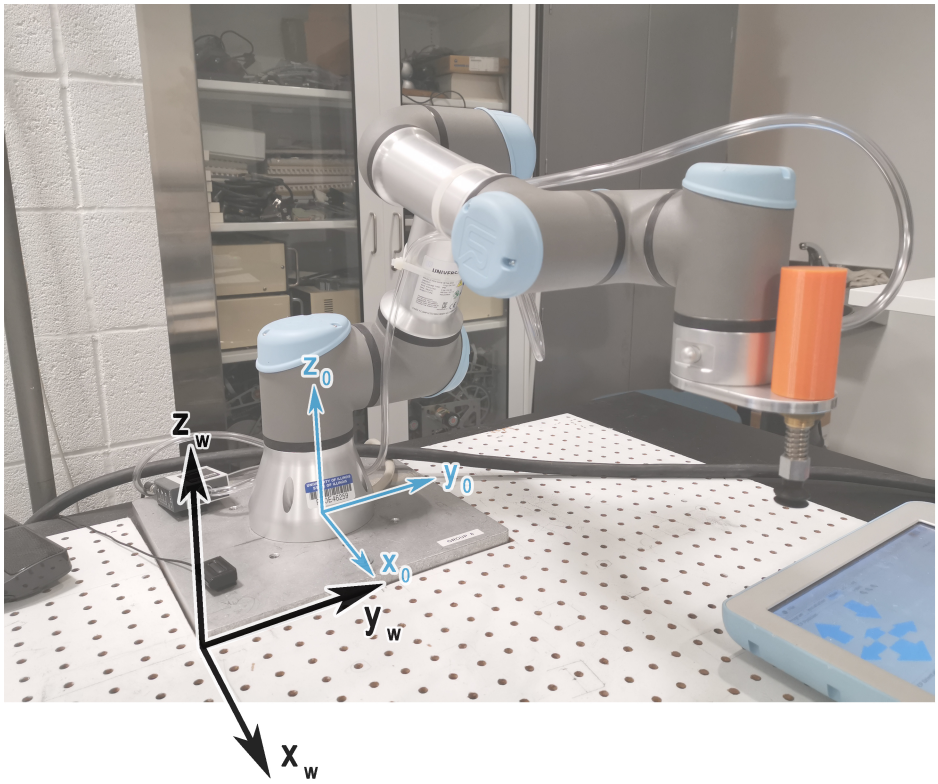


Figure 5.3: Correct location and orientation of the world frame.

3. Given the wrist's center point $(x_{cen}, y_{cen}, z_{cen})$, write an expression for the waist angle θ_1. Make sure to use the **atan2()** function instead of **atan()** because **atan2()** takes care of the four quadrants the x,y coordinates could be in.

$$\theta_1(x_{cen}, y_{cen}, z_{cen}) = \tag{5.1}$$

4. Solve for the value of θ_6, given yaw and θ_1.

$$\theta_6(\theta_1, yaw) = \tag{5.2}$$

5. Find the projected end point $(x_{3end}, y_{3end}, z_{3end})$ using $(x_{cen}, y_{cen}, z_{cen})$ and θ_1.

$$x_{3end}(x_{cen}, y_{cen}, z_{cen}, \theta_1) =$$
$$y_{3end}(x_{cen}, y_{cen}, z_{cen}, \theta_1) =$$
$$z_{3end}(x_{cen}, y_{cen}, z_{cen}, \theta_1) =$$

6. Write expressions for θ_2, θ_3 and θ_4 in terms of the end point. You probably will want to define some intermediate variables to help you with these calculations.

$$\theta_2(x_{3end}, y_{3end}, z_{3end}) =$$
$$\theta_3(x_{3end}, y_{3end}, z_{3end}) =$$
$$\theta_4(x_{3end}, y_{3end}, z_{3end}) =$$

7. Now that your code solves for all the joint variables (remember that θ_5 is always $-90°$) send these six values to the Lab 4 function **lab_fk()**. Do this simply to check that your inverse kinematic calculations are correct. Do keep in mind that your forward kinematic equations calculate the x,y,z position relative to the base frame not Lab 5's new world frame. Therefore, please change your forward kinematic function such that it displays the coordinates of the tip of the end effector in terms of the world frame instead of the UR3's base frame. Double check that the x,y,z point that you asked the robot to go to is the same value displayed by the forward kinematic equations.

5.6 Report

Each partner 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 are due one week after the final session of Lab 5 - before your lab session!
- Lab reports will be submitted online at GradeScope.

Your lab report should include the following:

- Coversheet containing your name and your partner's names, "Lab 5", and the weekday and time your lab section meets (for example, "Tuesday, 3pm").
- A clearly written derivation of the inverse kinematics solution for each joint variable $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$. The equations and the diagrams can be the same as your partners but the explanation should be in your own words. You **must** include figures in your derivation. Diagrams should be clear and easily read.
- For each test point include:
  - The given $\{(x_{w_{grip}}, y_{w_{grip}}, z_{w_{grip}}), \theta_{yaw}\}$
  - The measured position
  - The scalar error

- Include a brief discussion of sources of error.

As appendices to your report, include the following:

- Your **lab5_func.py** code and **lab5_exec.py** if it was edited.

5.7 Demo

Your TA will require you to run your program twice, each time with a different set of desired position and orientation. The first demo will require the UR3 to reach a point in its workspace off the table. The second demo will require the UR3 to reach a configuration above a block on the table with sufficient accuracy to pick up the block.

5.8 Grading

- 10 points, by the end of the first two hour lab session, show your TA your equations for finding x_{cen}, y_{cen}, z_{cen}.
- 10 points, by the end of the second two hour lab session, show your TA your equations for $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$.
- 10 points, by the end of the third two hour lab session, show your TA your inverse kinematic equations for $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ implemented in Python code.
- 50 points, successful demonstration.
- 20 points, report.

LAB 6

Camera Sensing and Integration into the World Frame for a Pick and Place Task

6.1 Important

Read the entire lab before starting and especially the "Grading" section as there are points you will receive by completing certain sections or checkpoints by the end of the lab session(s).

6.2 Objectives

This is the capstone lab of the semester and will integrate your work done in previous labs with python, ROS, forward and inverse kinematics and OpenCV. In this lab you will:

- Use OpenCV functions to find the centroid of each colored block and draw crosshair at that centroid along with a circle around the object.
- Develop transformation equations that relate pixels in the image to coordinates in the world frame
- Report the world frame coordinates (x_w, y_w) of the centroid of each block in the cameras view.
- Sort the two colors of blocks to predefined positions in the robot's work area.

6.3 References

- Review the vision setup and processing parts of Lab 3 as many items will be repeated here in Lab 6.
- Re-visit Appendix C which explains how to simplify the intrinsic and extrinsic equations for the camera.
- HSV Color Space:
  - `https://en.wikipedia.org/wiki/HSL_and_HSV`
  - https://stackoverflow.com/questions/10948589/
- Simple Blob detector:
  - `www.learnopencv.com/blob-detection-using-opencv-python-c/`
  - https://stackoverflow.com/questions/8076889/how-to-use-opencv-simpleblobdetector
  - `https://www.programcreek.com/python/example/89350/cv2.SimpleBlobDetector`
  - `https://www.programcreek.com/python/example/71388/cv2.SimpleBlobDetector_Params`

6.4 Tasks

6.4.1 Color Thresholding and Object Centroids

In the same way you used the HSV color space in Lab3 to find the orange color, find HSV ranges to identify both bright pink and bright green blocks. Then use **OpenCV** library functions to locate the centroid of the four blocks placed in the camera's view. In Lab 3, you were required to use the **SimpleBlobDectector** class to find the orange circles centroid. Here in Lab 6, you can again use the **SimpleBlobDectector** class to find all the blocks centroids, but if you would like to experiment with other OpenCV function, you are encouraged to do so.

6.4.2 Camera Setup

The problem of camera setup is that of relating (row,column) coordinates in an image to the corresponding coordinates in the world frame (x_w, y_w, z_w). For our purposes we may make several assumptions that will vastly simplify camera calibration. Please consult Appendix C and follow along with the simplification of the general equations.
Several parameters must be specified in order to implement the equations. Specifically, we are interested in θ the rotation between the world frame and the camera frame and β the scaling constant between distances in the world frame and distances in the image. We will compute these parameters by measuring object coordinates in the world frame and relating them to their corresponding coordinates in the image.

6.4.3 Pick and Place

Your program should:

- Move the Robot out of the way so that the camera can view Lab 3's entire grid area in order to find the blocks randomly placed in that area.
- Use **OpenCV** functions to find the centroid (x_w, y_w) of each block.
- Display a center point and a circle around each discovered block in the video display window.
- Command the robot to pick up each block one at a time and place it in the green predefined positions or the pink predefined positions.
- When finished with all blocks exit from the program.

6.5 Procedure

6.5.1 Threshold camera image to distinguish the colors of choice

Note the below procedure in section 6.5.1, 6.5.2 and 6.5.3 is very similar to lab3 and uses lab3 files.

1. Return to your lab3 directory and run:

   ```
   $ source devel/setup.bash
   $ roslaunch ur3_driver vision_driver.launch
   ```

   Open a second terminal window

   ```
   $ source devel/setup.bash
   $ rosrun lab3pkgpy lab3_image_tf_exec.py
   ```

   You should see two Windows open. One is the cameras video with an added horizontal black line and small red circle. Later in this lab you will use the horizontal line to straighten the mounting of the camera if it got moved or bumped. For the red dot towards the top left corner of the image look at the text printing in the terminal. A three number (H,S,V) value is printing. You can use this pixel value to help you find the range of H,S,V (Hue, Saturation, Value) values for three colors. First the same orange color you found in lab3 so you can use the calibration stick to help setup your camera. Then in addition, you should find the H,S,V range for the bright pink and green blocks.

   Side Note: What is the H,S,V color space? Please see the reference link to the Wikipedia page discussing color spaces. You are probably most familiar with the R,G,B (red,green,blue) color space. Here each pixel of an image has a R,G,B value. With H,S,V, colors are placed on a color

wheel and you indicate which color you are interested in by selecting an angle range. In **OpenCV** the range of the angle is from 0 degrees to 180 degrees. (You will find other implementations that use 0-360). For example a blueish color would be in the angle range of 110 degrees to 130 degrees. The S value is a number between 0 and 255 with 0 being very white and washed out and 255 the full, clear color. The V value is also a number between 0 and 255 with 0 being very dark and shaded and 255 the full clear color. The main reason we use the H,S,V color space in this lab is that it makes finding a color with different shading and lighting conditions much easier.

2. At this point take a look at the given python files in the **lab3pkg_py/scripts** folder. This section will only be modifying the files **lab3_func.py** and **lab3_image_tf_exec.py**. Take a look at these two files. Notice in **lab3_func.py**, in the function **blob_search**, the code is converting the RGB image to HSV. Then, leaving off where you ended in lab3, using a range for orange pixels the **inRange** function is used to threshold/mask the image to a binary image of only orange pixels. (Binary image means all orangeish pixels are ones and all other pixels are zeros). Next the masked image is cropped to only use the portion of the image that views the grid. (You will adjust the size of this cropped image in a few steps). Then at the bottom of the function both the color image and the cropped binary image are displayed in separate windows. (Notice in those windows there is a button that allows you to save the current image to a file.)

3. So your first goal is to find the 3 ranges of Hue, Saturation and Value that allows your program to find the orange calibration stick, and the pink and green blocks in the robot's work area from the camera's view. To find these HSV ranges you will need to do some trial and error. You are supplied with two tools to help you find the correct range. One is the red dot printed for you on the cameras video frames. Notice that when you put a colored block or the orange circle under that red dot the terminal window prints out the H,S,V value for that pixel. Experiment with a few colors and see the H,S,V values change. The code for putting this red dot on the image and displaying the H,S,V values to the terminal is in the **lab3_func.py** file towards the end of the **blobsearch** function. The **cv2.circle** function is used to add the red dot and the print statement prints out the H,S,V values. One issue with this method is that it only shows one pixel and therefore only one lighting condition. You need to make sure your range of H, S, V values work in all areas of the cropped image. So going on from here, you need to do some trial and error. Pick a range for H and S and V, change your code with that range and see how well the cropped image shows only the orange disk, and the pink or green blocks in all of its area. We have also supplied you with some off line code, **HSV.py**, that takes a single picture file and prints out a H,S,V range for the pixels selected.

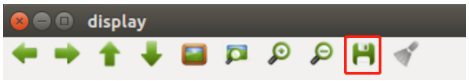


Figure 6.1: How to snapshot the video feed.

4. Once you have found good H,S,V ranges for orange calibration stick, and the pink and green blocks and your benches lighting conditions, you need to make sure the cropped image views all the area of the grid. First step is to look at the full color image and make sure the camera is turned so the black horizontal line is also horizontal with respect to the robots base. Have your TA help you with this. (Of course try your best not to bump the camera so that this process does not have to happen too often.). Then return to looking at the cropped binary image and move the blocks to all corners of the grid. Make sure that the blocks are seen in the cropped image at all the corners. If the cropped image does not see all corners of the grid you will need to modify the values **crop_top_row**, **crop_bottom_row**, **crop_top_col** and **crop_bottom_col** to move where the top and bottom corners of the cropped image are located. Note that the full image size is 480 rows by 640 columns. (If you have to change **crop_top_row**, **crop_bottom_row**, **crop_top_col** and **crop_bottom_col**, make a note to change these also in your lab6 code later in section 6.5.4.)

6.5.2 Use the OpenCV simpleBlobDetector library function to find the centroids of 2 Pink and 2 Green Blocks

1. The steps below show how to use **simpleBlobDetector** to find the centroids. First you will find the centroid of the orange color to prepare for setting up the camera. Then in section 6.5.4 you will change this code to look for pink, then green blocks.

   You are given starter code to get you started with **simpleBlobDetector**. See in **lab3_func.py** the function **blob_search_init**. This is where you need to set the parameters for **simpleBlobDetector** (hopefully already done in lab3). By default all the filterBy parameters are set to False. You need to decide which of these filters will work best to find the orange disks in the cropped image. There are also other parameters you will need to set in this function which are not given to you. Once you think you have set all the correct parameters, switch to the **blob_search** function and find the comment that shows you where to call the detect method of **simpleBlobDetector**.

   The detect method returns **keypoints**, which is a variable that has the centroid and other information of each blob found. If there is only one block with certain color then **keypoints** only has one element. If there are more blobs then **keypoints** also has the centroids for those blobs.

The web pages listed in the References section show example code of getting information out of the **keypoints** variable. Use the **print()** function to print out the number of blobs found and the coordinates of the first blob. Also use the **cv2.drawKeypoints()** function to draw a circle around each blob in the color video frames. **simpleBlobDetector** is being passed the cropped image. The centroid found is in pixel units. (0, 0) is the top left corner of the cropped image. Therefore you will need to add **crop_top_row** and **crop_top_col** to the **Keypoint**'s centroid values before calling the **cv2.drawKeypoints()** function with the full color image. **Keypoints** is a list of the **Keypoint** class. One of the elements in the **keypoint** class is pt, which is a tuple. To modify an element in a tuple, first change the tuple to a list with the **list()** function. Modify the element of the newly created list and finally set the **Keypoint.pt** tuple element to the modified list by converting the list to a tuple with the **tuple()** function. Figuring out the correct parameters for **simpleBlobDetector** will take some trial and error.

2. Using the orange calibration stick, move them to many different locations in the maze and print out the centroids found. Make sure that these centroids make sense in pixels where the (0, 0) camera origin is the top left corner.

6.5.3 Camera Setup

Again, a repeat of lab3 procedure:

1. Read Appendix C.3 before proceeding further.

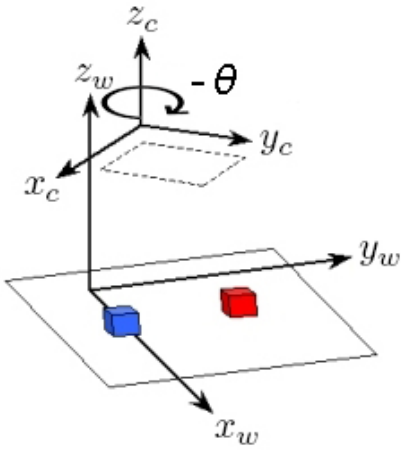


Figure 6.2: Arrangement of the world and camera frames.

2. Begin by writing the equations we must solve in order to relate image and world frame coordinates. You will need to combine the intrinsic and extrinsic equations for the camera; these are given in Appendix C.3. Write equations for the world frame coordinates in terms of the image coordinates.

$$\begin{aligned} x_w(r,c) &= \\ y_w(r,c) &= \end{aligned}$$

3. There are six unknown values you must determine: $O_r, O_c, \beta, \theta, T_x, T_y$. The principal point (O_r, O_c) is given by the row and column coordinates of the center of the image. We can easily find these values by dividing the `width` and `height` variables by 2.

$$\begin{aligned} O_r &= \frac{1}{2} height = \\ O_c &= \frac{1}{2} width = \end{aligned}$$

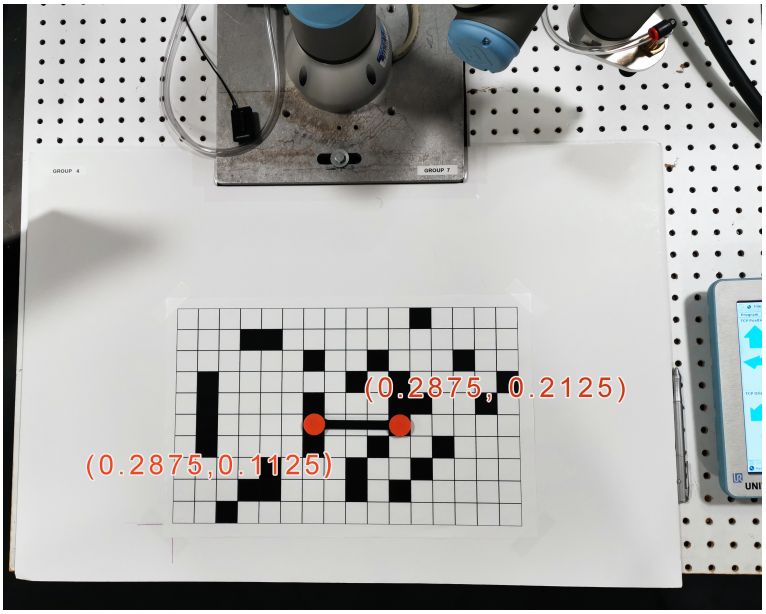


Figure 6.3: Where to place the calibration stick.

You will calculate the parameters β, θ, T_x, T_y in the python script:

lab3_image_tf_exec.py

First, place the calibration stick in the position ($x_w = 0.2875$, $y_w = 0.1125$ and $x_w = 0.2875$, $y_w = 0.2125$) shown in the Figure 6.3. Please keep

anything orange (including your hands) away from the camera's field of view. Open a terminal window:

```
$ source devel/setup.bash
$ roslaunch ur3_driver vision_driver.launch
```

Open a second terminal window:

```
$ source devel/setup.bash
$ rosrun lab3pkgpy lab3_image_tf_exec.py
```

You will modify this code (which will print these values to the terminal by default as zeros) to calculate β, θ, T_x, T_y values so that you can use them in **lab6_image_exec.py**.

Note that you may have to rerun this procedure to obtain these values when you return to the lab, as other groups will be using the same camera and may re-position the camera when you are not present. To keep these values relatively consistent, **PLEASE TRY NOT TO BUMP OR MOVE THE CAMERA.**

β is a constant value that scales distances in space to distances in the image. That is, if the distance (in unit length) between two points in space is d, then the distance (in pixels) between the corresponding points in the image is βd. Note that the calibration stick is exactly 0.1 m long.

$$\beta =$$

θ is the angle of rotation between the world frame and the camera frame. Figure 6.4 gives an overhead view of the blocks with a hypothetical cutout representing the image captured by the camera. Because the camera's x and y axes are not quite parallel to the world x and y axes, the blocks appear rotated in the image. You may expect this parameter to be fairly small.

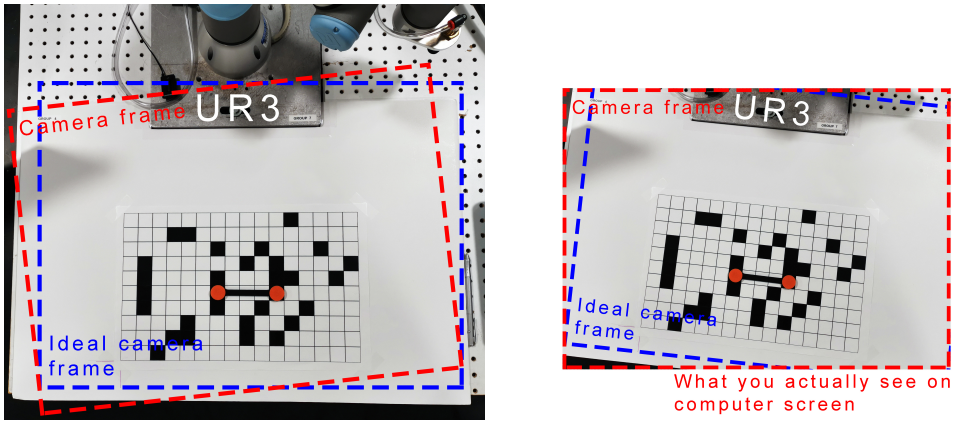
$$\theta =$$

T_x, T_y (unit in meters), the origin of the world frame expressed in the camera frame.

Substitute these values into the two equations you derived in step 2 above and solve for the unknown values.

$$\begin{aligned} T_x &= \\ T_y &= \end{aligned}$$

These values will be copied into your lab6 code in the next section.



(a) (b)

Figure 6.4: (a) Overhead view of the table; the rectangle delineated by blue dashed lines represents the desired camera's field of view, while the red dashed lines represents the actual camera's field of view (b) The actual image seen by the camera. Notice that x_c increases with increasing row values, and y_c increases with increasing column values.

6.5.4 Pick and Place

Now that you have the camera setup and the values for β, θ, T_x and T_y you are ready to start modifying the starter code for Lab 6.

- Download the starter code from the website and add it to your catkin directory. Take a look at the four given Python files: **lab6_header.py**, **blob_search.py**, **lab6_func.py** and **lab6_exec.py**.
- **Lab6_header.py**: This file does not need to be changed. It just includes items needed for this application. If you would like to add other Python features you can of course modify this file.
- **Lab6_func.py**: You will need to copy into this file work you have done in Labs 4 and 5. **Get_MS()**, **lab_fk()** and **lab_invk()** are all functions that you created in those previous labs. You will use some or all of these functions to move the UR3 from one X, Y, Z point to another point.
- **Blob_search.py**: You will be modifying/adding code in this file. The comments in this file will help you understand what you need to change. You should notice that this file is very similar to the starter code given in Lab 3 to call the **SimpleBlobDetector** class. You will modify these functions to find the centroid of either bright green blocks or bright pink blocks depending on the value passed in the parameter color. The **blob_search()** function should return the world coordinates of the center of the colored

blocks. Make sure to change β, θ, T_x and T_y to the values for your bench found in 6.5.3.

- **Lab6_exec.py**: You will need to add to this file. The comments in this file are again helpful. **move_block()** function will be similar to your **move_block()** function in lab 2, but here you will be using inverse kinematics to figure out what thetas will bring the arm to the desired points. **Image_callback()** is completed for you but make sure to understand what it is doing. It is called each image of the video feed from the USB camera and finding the world coordinates of pink and green blocks and storing these coordinates to global variables so **main()** can access these variables. **main()** function has comments to show you where to add your code. The initial code in **main()** starts the Publisher and Subscribers and then moves the robot arm out of the view of the camera. Once the arm is out of the way it starts the video processing of the USB camera and sleeps for one second. During that one second your vision processing code is run on each image and is finding the world coordinates of the pink and green blocks and storing the values to the global variables **xw_yw_R** and **xw_yw_G**. The code you need to develop then takes the **xw_yw_R** variable and commands the robot to pick up the two pink blocks, one at a time, and place them in the red area. Then using **xw_yw_G** place the two green blocks in the green area.

6.6 Report

Each partner 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:

- Due to the end of semester, your TA will determine the due date for all Lab 6 requirements. Note that late assignments are unlikely to be accepted.
- Lab reports will be submitted online at GradeScope.

Your lab report should include the following:

- Explain how you performed blob detection and include any parameters that you set and how you set them.
- Give a brief overview of how your code achieves the goals of Lab 6.
- Explain how you you used the lessons from Labs 1 to 5 to achieve the goals of Lab 6.
  - Explain what you learned and how it was used.
  - Remember to be specific.
  - A complete report should be able to include lessons from each lab.

As appendices to your report, include the following:

- Your **lab6_func.py** code
- Your **lab6_exec.py** code
- Your **Blob_search.py** code
- Any other code that your wrote or edited

6.7 Demo

You will demonstrate to your TA your code which draws crosshairs over the centroid of each object in an image and reports the centroid coordinates in (row,column) and $(x, y)_w$ coordinates. The robot should place the pink blocks in the "pink block" locations and the green blocks in the "green block" locations.

6.8 Grading

- 10 points, by the end of the first two hour lab session, show your TA that you have finished section 6.5.1 and have started work on section 6.5.3.
- 10 points, by the end of the second two hour lab session, show your TA that you have finished section 6.5.3 and have made good progress on section 6.5.4.
- 60 points, successful demonstration.
- 20 points, report.

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/Installation`
- 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.

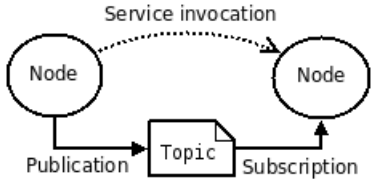


Figure A.1: source: `http://wiki.ros.org/ROS/Concepts`

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.

- 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 -zxvf lab2andDriver.tar.gz".

  "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_driver.launch
```

  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 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 lab2dannode node in lab2danpkg package:

  ```
  $ rosrun lab2pkg_py lab2_exec.py
  ```

A.6 More Publisher and Subscriber Tutorial

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

Appendix B

V-rep Simulation

The robotic control lab (3071 ECEB) receives a large volume of students with the expansion of ECE470 and ECE489. Lab space and work benches can be in high demand toward the end of the semester. Therefore, we highly encourage students to build and test your robot algorithms in simulation environments before deploying them on actual robots to relieve lab usage. Although any simulation software and environment that serve this purpose are welcomed, we recommend *V-rep* for those of you who don't have any preference. *V-rep* is a robot simulator with integrated development environment that is avaliable in Windows, Mac os, and Linux. Within *V-rep*'s simulation environment, each object/model can be individually controlled via an embedded script, a ROS node, or a remote API client. And the script that is used to control robots in simulation can be written in C/C++, Python, Java, Lua, Matlab or Octave. These features make *V-rep* adapt well to this course well and give students lots of freedom when conducting their own robot experiments without a physical robot.

B.1 Download and Install V-rep on your PC

- Download the latest version of V-REP PRO EDU that suits your PC's operation system from: `http://www.coppeliarobotics.com/downloads.html`

- For Windows users, launch and install the setup.exe file on your computer. It seems *V-rep* prefers to be installed on C drive.

- For Mac users, unzip the V-REP_PRO_EDU_Vx_x_x_Mac.zip folder and rename it if you prefer. Open a terminal inside the unzipped folder directory and type this command:

  ```
  $ ./vrep.app/Contents/MacOS/vrep
  ```

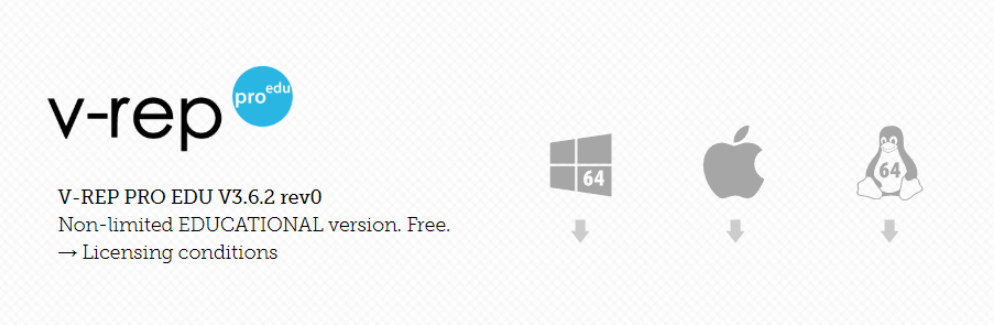


Figure B.1: Download the latest version of V-REP PRO EDU

- For Linux users, unzip the V-REP_PRO_EDU_Vx_x_x.tar folder and rename it if you prefer. Open a terminal inside the unzipped folder directory and type this command:

  ```
  $ ./vrep.sh
  ```

- To learn more about using *V-rep* via command lines, visit: `http://www.coppeliarobotics.com/helpFiles/en/commandLine.htm`

B.2 Setup remote API to interface with V-rep

- Follow instructions on `http://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm` to enable the remote API with the programming language you prefer. (We recommend Python for ECE470)

- You will be saving your remote API code in a different directory (any directory you want but preferably not the directory that contains *V-rep*'s main program). For example, if you are using Python, create a new directory with a name you like, this new directory must contain the following files (besides the .tttt scene file and your Python remote API script) for the remote API function to work (See Figure B.2):

  vrep.py

  vrepConst.py

  remoteApi.dll, remoteApi.dylib or remoteApi.so (depends on your operation system)

- If you are a Python 3 user, we do have a remote API starter code for you to play with: `http://coecsl.ece.illinois.edu/ece470/ece470_vrep_Linux.zip` `http://coecsl.ece.illinois.edu/ece470/ece470_vrep_Win64.zip`

ece470_sim.ttt	7/31/2019 10:43 AM	TTT File	526 KB
example.py	7/30/2019 7:50 PM	PY File	8 KB
remoteApi.so	6/27/2019 2:37 PM	SO File	111 KB
vrep.py	6/27/2019 2:37 PM	PY File	68 KB
vrepConst.py	6/27/2019 2:37 PM	PY File	42 KB

Figure B.2: An example of a working directory

B.3 Add a robot in your scene

- Drag any robot of interest from the menu on the left hand side to the scene.

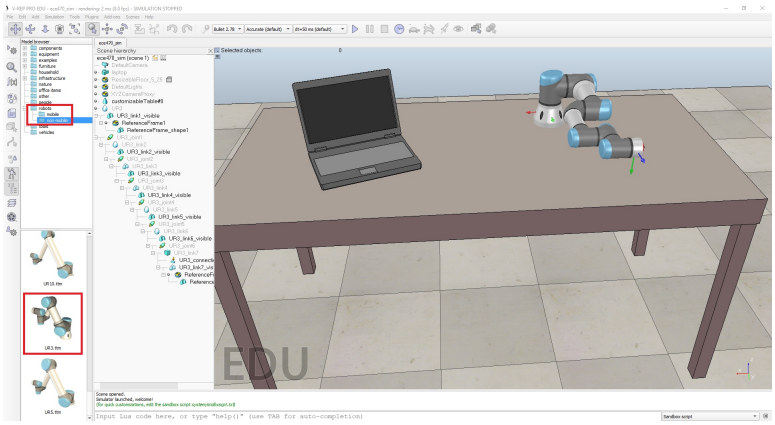


Figure B.3: Adding robot to your scene

- Re-position the robot using the move and rotate icons locate on the top menu bar.
- Delete the default child script (Figure B.4) from your robot before you try your remote API program, otherwise you will have two scripts that control the same robot.
- Save the scene you created as .tttt file in the directory that contains your remote API script.

B.4 Important notes!!

- The dimension of the UR3 in *V-rep* is somewhat different than the actual UR3 we use in lab. Therefore, when building your forward kinematics

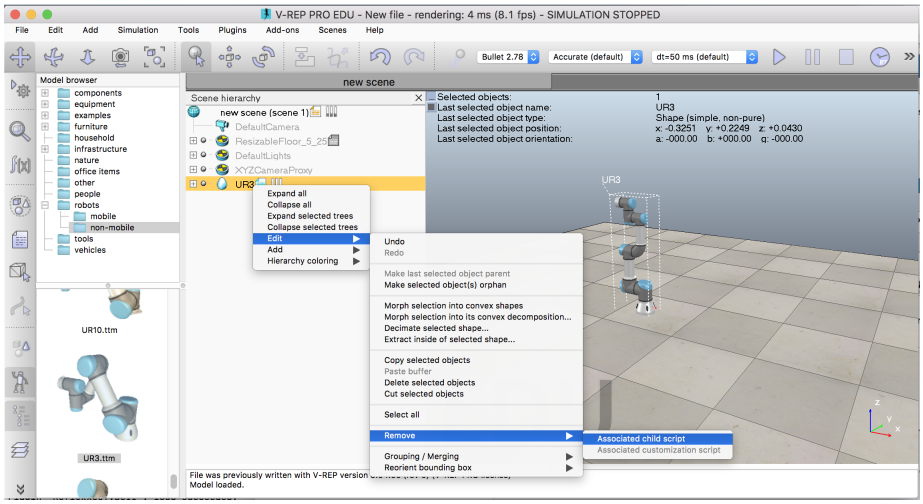


Figure B.4: Delete the default child script from ur3

function in *V-rep*, **DO NOT** use the dimensions from UR3's engineering drawing. Instead, you can use the following API function:

```
vrep.simxGetObjectPosition()
```

to obtain the distance between two joint centers.

- The drawback of using numerical solvers is the inconsistency between each simulation's result, especially in less significant digits (e.g. ans = 1.2345644 vs. ans = 1.234598). You may add a rounding step after some parameters of interest to reduce the chance of being inconsistent (e.g. round(ans) = 1.2346).

- For detailed information on *V-rep* API functions:

  Python

  `http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm`

  Matlab

  `http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsMatlab.htm`

  C++

  `http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm`

Appendix C

Notes on Computer Vision

C.1 OpenCV

C.1.1 Camera Driver

A camera driver in ROS is needed to read the image from the USB camera. Here we are using a driver called "cv_camera". This package is provided in folder:

```
src/drivers/cv_camera
```

To load **both** this camera driver and the ur3_driver use a new launch file:

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

Once the driver is loaded, the camera will start working and the image data is published to topic

```
/cv_camera_node/image_raw
```

which is a ROS image message.
If you see error "HIGHGUI ERROR: V4L: device /dev/video0: Unable to query" when launching the driver file, then probably the camera's connection is poor, unplug and plug in the camera cable again, then try to launch the driver file again. It's fine to have warning "Invalid argument".

There is also a separated camera driver called "camera_calibration", which is used during camera fish eye correction as a dedicated driver. Your Lab 3 and Lab 6 program will not use it.

C.1.2 Accessing Image Data

ROS and OpenCV are using different image data types, we have to convert ROS images to OpenCV images in order to use OpenCV environment for image processing. The library "cv bridge" will help you convert between ROS images and OpenCV images, it's also already included in ROS Indigo. In the provided

code for Lab 3, the conversion is done in the class "ImageConverter", this class subscribes to the image published by the camera, converts it to an OpenCV image, then converts it back to a ROS image and finally publishes it to a new topic:

/image_converter/output_video

It's necessary to briefly explain the structure of the provided code for Lab 3. As well as the image conversion, the image processing is also done in this code. You will also be asked to publish to ur3/command to command the robot to move to a desired spot in the image's field of view. After compilation, the code will generate a node. The logic of code is:

```
#include libraries

class ImageConverter:

    def __init__(self, SPIN_RATE):

        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/cv_camera_node/image_raw",
        Image, self.image_callback)
        self.coord_pub = rospy.Publisher("/coord_center", String, queue_size=10)
        self.loop_rate = rospy.Rate(SPIN_RATE)
        self.detector = blob_search_init()

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


    def image_callback(self, data):

        global theta
        global beta
        global tx
        global ty

        try:
            # Convert ROS image to OpenCV image
            raw_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
        except CvBridgeError as e:
            print(e)

        # Flip the image 180 degrees
        cv_image = cv2.flip(raw_image, -1)
```

```
        # cv_image is normal color image
        blob_image_center = blob_search(cv_image, self.detector)

        if(len(blob_image_center) == 0):
            print("No blob found!")
            self.coord_pub.publish("")
        else:
            x = int(blob_image_center[0].split()[0])
            y = int(blob_image_center[0].split()[1])
            # print("Blob found! ({0}, {1})".format(x, y))

            # Given theta, beta, tx, ty, calculate the world coordinate of x,y namely xw, yw

            xw = ??
            yw = ??

            xy_w = str(xw) + str(' ') + str(yw)
            self.coord_pub.publish(xy_w)


def main():

    SPIN_RATE = 20 # 20Hz

    rospy.init_node('lab3ImageNode', anonymous=True)

    ic = ImageConverter(SPIN_RATE)

    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down!")

    cv2.destroyAllWindows()


if __name__ == '__main__':
    main()
```

All the work is done inside of class "ImageConverter". Here, if you never used a class, think of it as a self defined structure, with variables and functions as its member. A class consists of its constructor, destructor(not always required), public members and private members.

- ImageConverter(): The constructor of this class. It will be executed when

a variable of this class is declared. We will put the subscriber in the constructor.

- image_callback(): A member function of this class. It contains all the image-processing related operations such as performing Blob search and publishing coordinates.
- main(): In main(), we just need to declare a variable which is ImageConverter class. By issuing this declaration, all the code of the class becomes active.

C.1.3 Some Useful OpenCV Functions

Here are some useful OpenCV Functions (You may not use all of these):

- cv2.flip()
- cv2.inRange()
- cv2.cvtColor()
- cv2.SimpleBlobDetector_Params()
- cv2.SimpleBlobDetector_create()
- cv2.drawKeypoints()
- cv2.circle()

C.2 Camera Calibration

C.2.1 Camera Placement and Fish Eye Correction

PLEASE CONSULT WITH YOUR TA IF YOU FEEL THE NEED TO PERFORM THE FOLLOWING CALIBRATION. DO NOT DO THIS WITHOUT YOUR TA'S SUPERVISION.

In order to simplify the camera calibration procedure, some preparatory work is required so that the assumptions we make in Lab 3 is valid.

- First, launch the camera driver so that we can access the camera:

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

- In an new terminal, run the test Python script:

  ```
  $ rosrun lab3pkg_py lab3_image_tf_exec.py
  ```

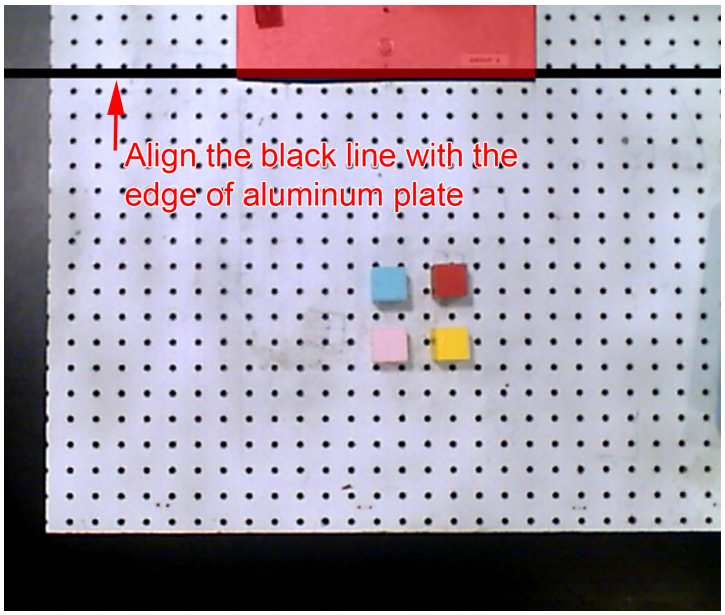


Figure C.1: Check if your camera is in place.

- Then you will see a window with live footage from the camera (Figure C.1). Please make sure your camera is pointing straight down (so that the camera frame is parallel to the table); meanwhile, verify that the black line showing in the footage is aligned with the front edge of the aluminum plate.

- Your camera is now correctly placed and ready for further calibration steps.

Now we can move on to the Fish Eye Correction. Please note that the Fish Eye Correction has already been done before each lab section to preserve time. The following steps are here for TA's reference.

- First, launch the camera calibration driver:

  ```
  $ roslaunch usb_cam usb_cam.launch
  ```

- In an new terminal, run the fish eye calibration script:

  ```
  $ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.025 image:=/usb_cam/image_raw camera:=/usb_cam --no-service-check
  ```

- A calibration window will appear and be ready to "sample" the calibration board (the checker board).
- Make sure your samples are as diverse (linearly independent) as possible.
  – Try place the checker board near the camera frame's center, edges and corners; with different orientations and tilt angles; at both far and close distances(Figure C.2).
- Repeat the last step until the "CALIBRATE" button becomes active(Figure C.3).
- Click "CALIBRATE" and wait until the "SAVE" and "COMMIT" buttons become active. Then click "SAVE" and "COMMIT".
- Then the program will generate a ".yaml" file which stores the necessary camera parameters to perform fish eye correction. This file will be used next time when you launch the camera driver.

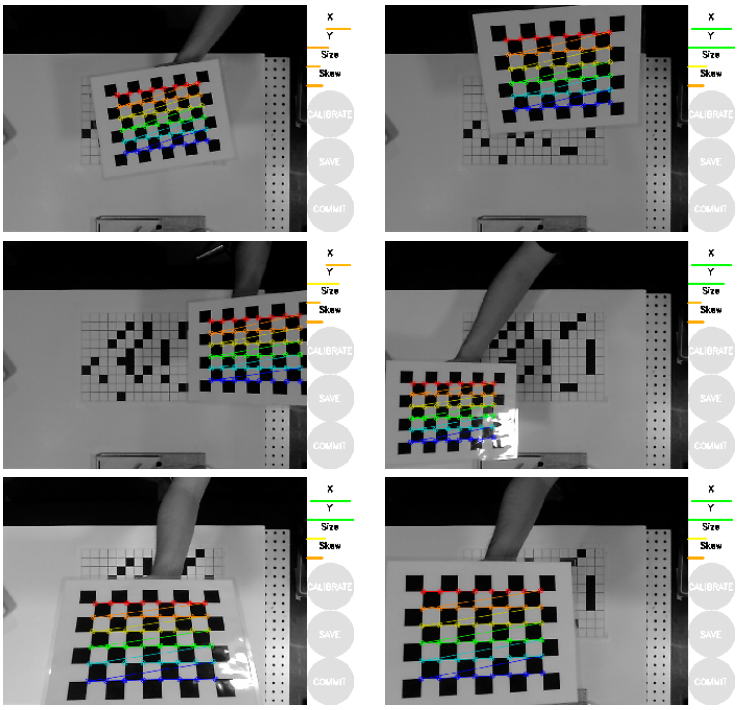


Figure C.2: Ensuring the diversity of samples

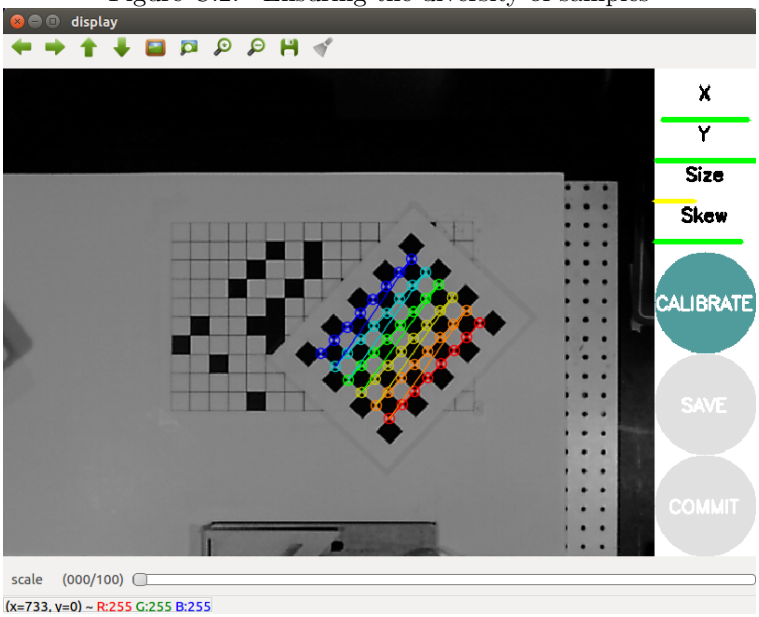


Figure C.3: Click "CALIBRATE" when it is active.

C.3 Simplified Perspective Transform

To extract spacial information from camera images, we need to derive formulas to compute an object's coordinates in the world frame $(x, y, z)^w$ based on row and column pixel coordinates in the image $(x, y)^c$. For this lab, however, we may make a few assumptions that will simplify the calculations. We begin with the following intrinsic and extrinsic equations: (variables defined as in Table C.1)

$$\text{intrinsic:} \quad \begin{aligned} r &= \tfrac{f_x}{z^c} x^c + O_r \\ c &= \tfrac{f_y}{z^c} y^c + O_c \end{aligned}$$

$$\text{extrinsic:} \quad p^c = R_w^c p^w + O_w^c$$

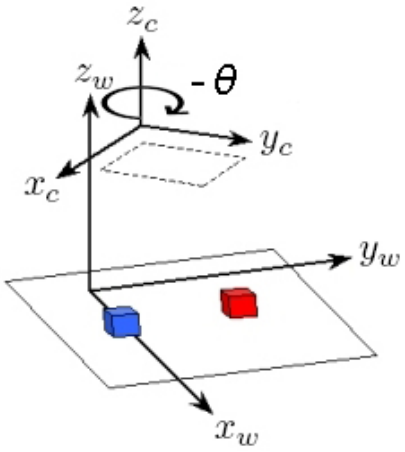


Figure C.4: Arrangement of the world and camera frames.

The intrinsic equation contains camera's internal parameters like focal length and principal point, while the extrinsic equation describes the placement of the camera (how the camera is mounted relative to the world coordinate).

The image we have is "looking straight down" the z^c axis. Since the center of the image corresponds to the center of the camera frame, (r, c) does not correspond to $(x^c, y^c) = (0, 0)$. Therefore, you must adjust the row and column values using (O_r, O_c).

We also note that the row value is associated with the x^c coordinate, and the column value is associated with the y^c coordinate. By this definition, and the way row and column increase in our image, we conclude that the z^c axis points up from the table top and into the camera lens. As a consequence, we define $R_w^c = R_{z,\theta}$ instead of $R_{z,\theta} R_{x,180^\circ}$.

Name	Description
(r, c)	(row,column) coordinates of image pixel
f_x, f_y	ratio of focal length to physical width and length of a pixel
(O_r, O_c)	(row,column) coordinates of *principal point* of image (principal point of our image is center pixel)
(x^c, y^c)	(meters) coordinates of point in camera frame
p^c	$= [x^c y^c z^c]^T$ coordinates of point in camera frame
p^w	$= [x^w y^w z^w]^T$ coordinates of point in world frame
O_w^c	$= [T_x T_y T_z]^T$ origin of world frame expressed in camera frame
R_w^c	rotation expressing camera frame w.r.t. world frame.

Table C.1: Definitions of variables necessary for camera calibration.

Therefore, we make the following assumptions:

1. The z axis of the camera frame points straight up from the table top, so z^c is parallel to z^w. Modifying the equation from the text, we have

$$\begin{aligned} R_w^c &= R_{z,\theta} \\ &= \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned}$$

2. All objects in our image will be the blocks we use in the lab, therefore all objects have the same height. This means that z^c in the intrinsic equations is the same for every object pixel. Since we can measure z^c, we can ignore the computation of z^c in the extrinsic equation.

3. The physical width and height of a pixel are equal ($s_x = s_y$). Together with assumption 2, we can define

$$\beta = \frac{f_x}{z^c} = \frac{f_y}{z^c}.$$

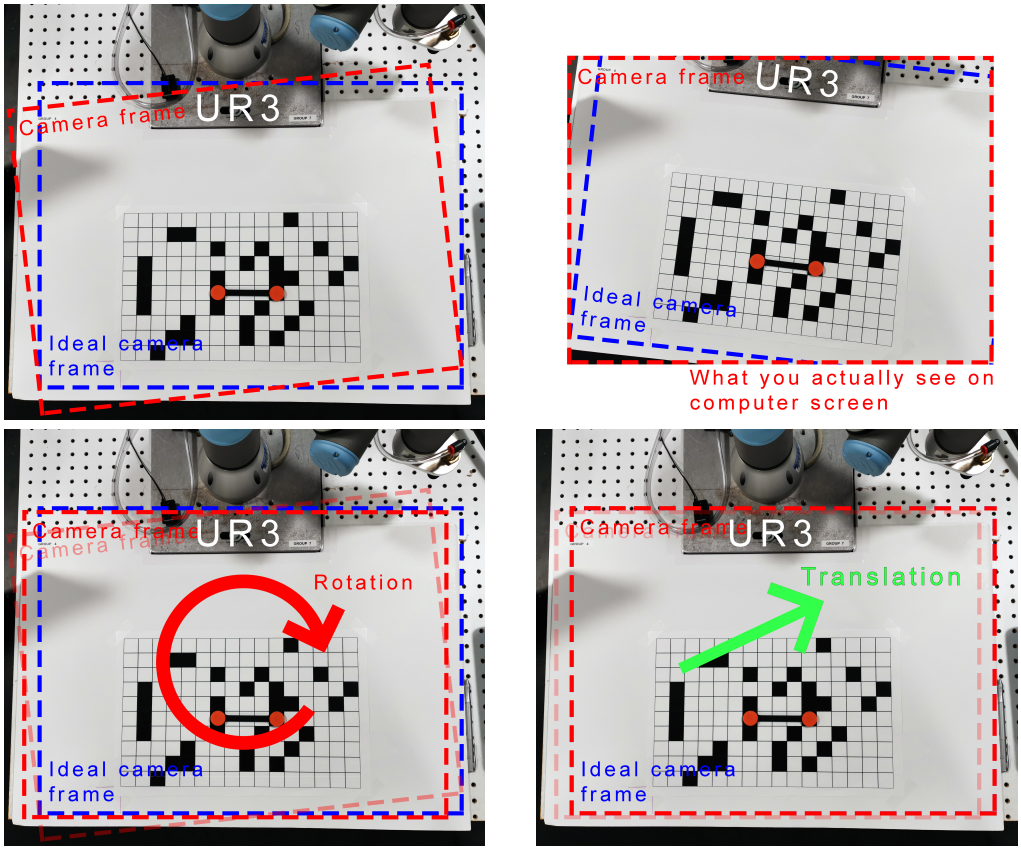


Figure C.5: (a) The rotation matrix R_w^c will compensate for the camera's angular misalignment (caused by θ). (b) The translation matrix T_w^c, on the other hand, corrects the linear misalignment in x and y axes.

The intrinsic and extrinsic equations are now

$$\text{intrinsic:} \quad \begin{array}{l} r = \beta x^c + O_r \\ c = \beta y^c + O_c \end{array}$$

$$\text{extrinsic:} \quad \begin{bmatrix} x^c \\ y^c \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x^w \\ y^w \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}.$$

For our purposes, we are interested in knowing a point's coordinates in the world frame $(x, y)^w$ based on its coordinates in the image (r, c). Therefore, you must solve the four equations for the world coordinates as functions of the image coordinates.

$$\begin{array}{rl} x^w(r, c) & = \\ y^w(r, c) & = \end{array}$$