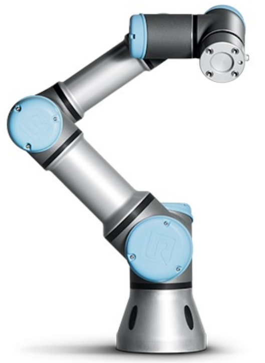
ECE 470
Introduction to Robotics
Lab Manual



Dan Block
Jonathan K. Holm
Jifei Xu
Yinai Fan

University of Illinois at Urbana-Champaign

Spring 2018

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. Spong, Hutchinson, and Vidyasagar's textbook *Robot Modeling and Control* (John Wiley and Sons: New York, 2006) 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 SH&V 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 lab sessions.

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
- 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: http://coecsl.ece.illinois.edu/ece470/UR3UserManual-en-US3-4-3.pdf

- UR3 Software Manual: http://coecsl.ece.illinois.edu/ece470/SoftwareManual-en-US3-4-3.pdf
- 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 lab sessions.

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) structure to program the UR3. For this lab you will continue to program the UR3 using its Teach Pendant but perform a similar task as 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: http://coecsl.ece.illinois.edu/ece470/UR3_User_Manual_en_US3-4-3.pdf
- UR3 Software Manual: http://coecsl.ece.illinois.edu/ece470/Software_Manual_en_US3-4-3.pdf
- 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.)

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.

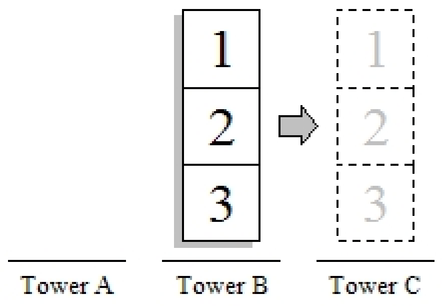


Figure 1.1: Example start and finish tower locations.

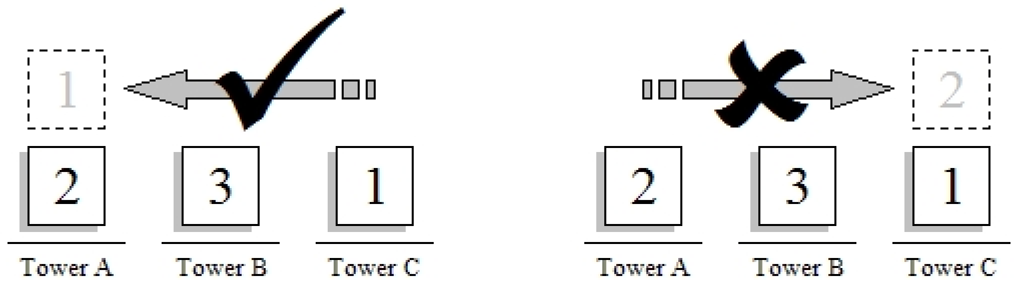


Figure 1.2: Examples of a legal and an illegal move.

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 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.

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 white stickers 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 turns in a report. In your own words, explain the teach pendant program your group wrote.

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 lab sessions.

2.2 Objectives

This lab is an introduction to controlling the UR3 robot using the Robot Operating System (ROS) and the C++ 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 cpp 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 B of this lab manual for details of ROS and C++ 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.)

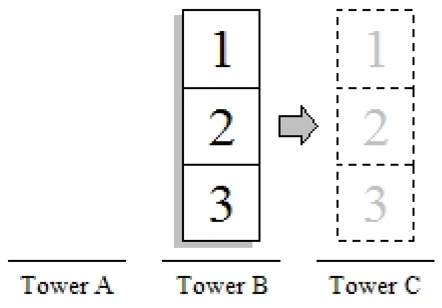


Figure 2.1: Example start and finish tower locations.

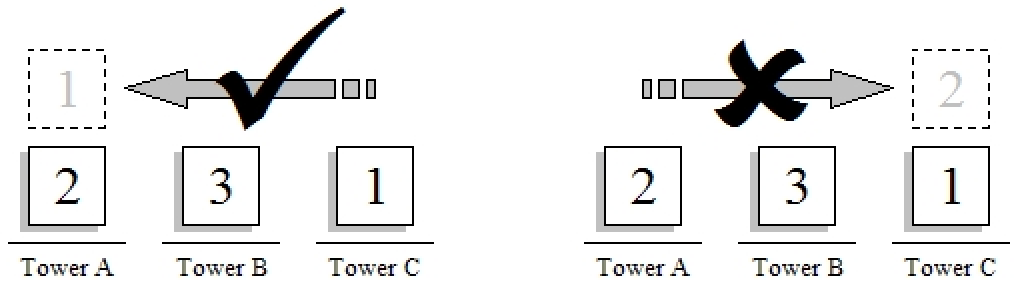


Figure 2.2: Examples of a legal and an illegal move.

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 cpp 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 B.

2. If you haven't already, download `lab2andDriver.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 lab2andDriver.tar.gz. You should see two folders lab2pkg and driver. Compile your workspace with catkin_make. Inside this package you can find lab2.cpp with comments to help you complete the lab.

    - `lab2.cpp` a file in src folder with skeleton code to get you started on this lab. See Appendix B for how to use basic ROS. 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.cpp.
    - `package.xml` This file defines properties about the package including package dependencies.
    - To run lab2 code: In one terminal source it and run `roslaunch ece470_ur3_driver ece470_ur3_driver.launch`. IMPORTANT: This .launch file needs to know your robot's IP address. See Appendix B for finding and setting your robot's IP. Then open a new termial, source it and run `rosrun lab2pkg lab2node`

3. Use the provided white stickers or 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.cpp 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 in the main function but provides function definitions for moving the arm and moving blocks (move_arm and move_block respectively). Once you

understand the starter code moving from one position to the next, clean up the code by completing the shell functions move_arm and move_block. Move_arm moves the tool to a specified location and move_block picks up a block from a tower and can place 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 isReady, create a global variable to communicate to your main() 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 #include the .h file that defines this message. The include file has already been #include in lab2.h for you. Use the explanation below and the given code in lab2.cpp that creates the subscription to ur3/position and its callback function as a guide to subscribe to the rostopic that publishes IO status.

2.7 Lab2.cpp Explained

First open up Lab2.cpp 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.cpp 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.cpp file as the correct version. Lab2.cpp is broken down into sections and described in more detail below.

```
#include "lab2pkg/lab2.h"
```

```
#define PI 3.14159265359
#define SPIN\_RATE 20  /* Hz */
```

You can find lab2.h in the lab2pkg/include directory. It includes all needed h files to allow lab2.cpp to call ROS functionality and standard C++ functions. SPIN_RATE will be used as the publish rate to send commands to the ECE470 ROS driver.

```
//arrays defining Waypoints, move arm to these points
double arr1[]={189.87*PI/180,-64.8*PI/180,126.91*PI/180,-150.81*PI/180,-90*PI/180,0*PI/180};
double arr2[]={212.23*PI/180,-45.45*PI/180,97.79*PI/180,-141.65*PI/180,-90*PI/180,0*PI/180};
double arr3[]={118.46*PI/180,-58.32*PI/180,80.58*PI/180,-200.64*PI/180,-90*PI/180,0*PI/180};

double arr4[]={0,0,0,0,0,0};

std::vector<double> Q11 (arr1,arr1+sizeof(arr1) / sizeof(arr1[0]));
std::vector<double> Q12 (arr2,arr2+sizeof(arr2) / sizeof(arr2[0]));
std::vector<double> Q13 (arr3,arr3+sizeof(arr3) / sizeof(arr3[0]));
std::vector<double> v (arr4,arr4+sizeof(arr4) / sizeof(arr4[0]));
```

This code is initializing four "vector" variables to be used as waypoints and velocity vectors. These variables Q11, Q12, Q13, and v will be used to command the robot to the six θ's assigned to the vector. The Q11, Q12 and Q13 vector elements are in radians and arranged in the order $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \}$. The v vector is the final velocity of each joint when the waypoint is reached. It makes sense for these velocities to be zero in our case where we want to move from point to point. But if the robot for example needs to move along a trajecory at constant speed it may be desirable to have a velocity vector other than zero. We will be keeping this v vector zero for this class. The assigning of this vector variable is a bit involved and confusing so it needs a bit of explaining. First a temporary double array is created containing the joint angles for a waypoint. Then that temporary array is use to assign the values of the vector. Q11 has type vector with double elements and it is initialized with a pointer to the beginning of the array arr1 and a pointer to the end of the array. The code sizeof(arr1)/sizeof(arr1[0]) is the total size of elements of the array arr1 divided by the size of an individual arr1 element. So in the case for arr1, it is a 6 element double array. Doubles are 8 bytes so sizeof(arr1) returns 48 and sizeof(arr1[0]) returns 8. So the initialization line could be rewritten `std::vector<double> Q11 (arr1,arr1+6);`.

```
// creating an array of these vectors allows us to iterate through them
// and programatically choose where to go.
std::vector<double> Q [3][3] = {  // Q[rows][columns]
    {Q11, Q12, Q13},
    {Q11, Q12, Q13},
    {Q11, Q12, Q13}
};
```

The vectors, once created, are added to a multi-dimensional array 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. The array is initialized to contain $std :: vector < double >$, and has 3 rows and 3 columns. Feel free to change the size and contents as appropriate for your algorithm.

```
bool isReady=1;
bool pending=0;

// Whenever ur3/position publishes info this callback function is run.
void position_callback(const ece470_ur3_driver::positions::ConstPtr& msg)
{
  isReady=msg->isReady;
  pending=msg->pending;

}
```

This is lab2node's callback function that is called when the ece470_ur3_driver publishes new position data. In LAB 2 we are only interested in whether the robot has reached the previous waypoint. The ece470_ur3_driver::positions::ConstPtr type additionally has current joint positions and velocities but we are not using them in this callback. The variables isReady and pending have opposite means. In other words when isReady is true, pending is false. pending was used in an earlier version of this code but currently is not being used. When isReady is false, the robot has not reached the commanded waypoint. When isReady is true, the robot has reached its previous commanded position and is ready for another command. isReady is a global variable so that the main() code below can use this variable to know when it is appropriate to send another waypoint to the robot.

```
int main(int argc, char **argv)
{
  int inputdone = 0;
  int Loopcnt = 0;
  ros::init(argc, argv, "lab2node");
  ros::NodeHandle nh;

  ros::Publisher pub_command=nh.advertise<ece470_ur3_driver::command>("ur3/command",10);

  ros::Subscriber sub_position=nh.subscribe("ur3/position",1,position_callback);

  ros::ServiceClient srv_SetIO = nh.serviceClient<ur_msgs::SetIO>("ur_driver/set_io");

  ur_msgs::SetIO srv;
  ece470_ur3_driver::command driver_msg;
```

To start as a ROS node the ros:init() 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 ece470_ur3_driver::command message which consists of the desired robot joint angles and the duration it takes to get to these desired angles. Next lab2node subscribes to "ur3" node "position" publisher. Whenever new joint angles and status variables are ready to be sent, the callback function "position_callback" is called and passed the message ece470_ur3_driver::positions which contains the six joint angles and status info. Finally lab2node uses the "ur_driver" node "set_io" service to turn on and off digital output 0 to turn on and off the suction cup gripper. ur_msgs::SetIO is the service message containing the desired state of the IO pin. Variable "srv" will be used to issue the suction cup command and driver_msg will be used to command the joint angle waypoints.
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 "ece470_ur3_driver::positions", "ece470_ur3_driver::command" and "ur_msgs::IOStates". Finally for the service run "rosservice list" to find the "ur_driver" node and "set_io" service, "rosservice info" to check the values needed when calling the "ur_driver/set_io" service and "rossrv list" to find the "ur_msgs::SetIO" service message.

```
std::string inputString;
while (!inputdone) {
  std::cout << "Enter Number of Loops <Either 1 2 or 3>";
  std::getline(std::cin, inputString);
  std::cout << "You entered " << inputString << "\n";
  if (inputString == "1") {
    inputdone = 1;
    Loopcnt = 1;
  } else if (inputString == "2") {
    inputdone = 1;
    Loopcnt = 2;
  } else if (inputString == "3") {
    inputdone = 1;
    Loopcnt = 3;
  } else {
    std:cout << "Please just enter the character 1 2 or 3\n\n";
  }
}
```

This standard C++ code printing messages to the command prompt and recieving text input from the command prompt. It loops until the correct data is input.

```
while(!ros::ok()){};

ROS_INFO("sending Goals");

ros::Rate loop_rate(SPIN_RATE);
int spincount = 0;
```

Here the code waits for ROScore to be executed and ready. ROS_INFO prints a message to the command prompt. ros::Rate loop_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(Loopcnt > 0) {
```

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

```
  driver_msg.destination=Q[0][2];
  pub_command.publish(driver_msg);
  spincount = 0;
  while (isReady) {
    ros::spinOnce();
    loop_rate.sleep();
    if (spincount > SPIN_RATE) {
      pub_command.publish(driver_msg);
      ROS_INFO("Just Published again driver_msg");
      spincount = 0;
    }
    spincount++;
  }
  ROS_INFO("waiting for rdy");
  while(!isReady)
  {
    ros::spinOnce();
    loop_rate.sleep();
  }
```

This block of code publishes a new waypoint, in this case Q3, to the ece470_ur3_driver driver. But if for some reason the ece470_ur3_driver does not receive the published command within one second it will publish to the ece470_ur3_driver again. After the waypoint is published wait until the driver is at the waypoint and ready for the next command. Let's detail each line:

- Set Publisher variable driver_msg to new waypoint Q3.
- Publish new waypoint command to ece470_ur3_driver command.
- Loop until isReady is false. isReady turning to false indicates that ece470_ur3_driver has recieved the current command and is moving the robot to the new waypoint.
- ros::spinOnce is called to allow ROScore to run other ROS processes as needed.
- Put our ROS node to sleep until the next 50ms period is complete.
- If spincount increments to SPIN_RATE, 20 in our case, that means one second has passed and ece470_ur3_driver has not recived the published command. republish the same waypoint command.
- Once isReady is false exit the first while loop and enter the second while loop. Sit here now waiting for isReady to be true so that the code can send another waypoint to the robot if needed.

```
ROS_INFO("sending Goals 1");
driver_msg.destination=Q[0][0];
pub_command.publish(driver_msg);
spincount = 0;
while (isReady) {
  ros::spinOnce();
  loop_rate.sleep();
  if (spincount > SPIN_RATE) {
    pub_command.publish(driver_msg);
    ROS_INFO("Just Published again driver_msg");
    spincount = 0;
  }
  spincount++;
}
ROS_INFO("waiting for rdy");
while(!isReady)
{
  ros::spinOnce();
  loop_rate.sleep();
}
```

Repeat the same steps to send the waypoint Q[0][0] to the robot arm.

```
    srv.request.fun = 1;
    srv.request.pin = 0;
    srv.request.state = 1.0;
    if (srv_SetIO.call(srv)) {
      ROS_INFO("True: Switched Suction ON");
    } else {
      ROS_INFO("False");
    }
    ROS_INFO("Value = %.2f",SuctionValue);
```

Here the service urmsgs::SetIO is used to turn on the suction cup gripper by turning on digital output 0. srv.request.fun = 1 ? Set the pin to command to digital out 0. Then set the state of the I/O pin to on by setting state to 1.0. Call the service.

```
    ROS_INFO("sending Goals 2");
    driver_msg.destination=Q[0][1];
    driver_msg.duration=2.0;
    pub_command.publish(driver_msg);
    spincount = 0;
    while (isReady) {
      ros::spinOnce();
      loop_rate.sleep();
      if (spincount > SPIN_RATE) {
        pub_command.publish(driver_msg);
        ROS_INFO("Just Published again driver_msg");
        spincount = 0;
      }
      spincount++;
    }
    ROS_INFO("waiting for rdy");
    while(!isReady)
    {
      ros::spinOnce();
      loop_rate.sleep();
    }
```

Repeat the same steps to send the waypoint Q[0][1] to the robot arm.

```
    srv.request.fun = 1;
    srv.request.pin = 0;
    srv.request.state = 0.0;
    if (srv_SetIO.call(srv)) {
      ROS_INFO("True: Switched Suction OFF");
    } else {
      ROS_INFO("False");
    }
```

Here the service urmsgs::SetIO is used to turn off the suction cup gripper by turning off digital output 0. srv.request.fun = 1 ? Set the pin to command to digital out 0. Then set the state of the I/O pin to off by setting state to 0.0. Call the service.

```
    ROS_INFO("sending Goals 3");
    driver_msg.destination=Q[0][0];
    driver_msg.duration=1.0;
    pub_command.publish(driver_msg);
    spincount = 0;
    while (isReady) {
      ros::spinOnce();
      loop_rate.sleep();
      if (spincount > SPIN_RATE) {
        pub_command.publish(driver_msg);
        ROS_INFO("Just Published again driver_msg");
        spincount = 0;
      }
      spincount++;
    }
    ROS_INFO("waiting for rdy");
    while(!isReady)
    {
      ros::spinOnce();
      loop_rate.sleep();
    }
```

Repeat the same steps to send the waypoint Q[0][0] to the robot arm.

```
    Loopcnt--;
  }
  return 0;
}


int move_arm(ros::Publisher pub_command , ros::Rate loop_rate, std::vector<double> dest, float duration
{
    int error = 0;
    return error;
}

int move_block(ros::Publisher pub_command ,
                ros::Rate loop_rate,
                ros::ServiceClient srv_SetIO,
                ur_msgs::SetIO srv,
                int start_loc,
                int start_height,
                int end_loc,
                int end_height)
```

```
{
    int error = 0;
    return error;
}
```

Two function definitions are provided and should be used to complete the assignment. Functions are useful when the same procedure is used many times. In the main function, there were several places where the arm was moved and this requires about 20 lines of code for each move. The move_arm function takes as its input the ros::Publisher, the ross:Rate, the target destination and duration. Its output is an integer that indicates different types of errors. It is up to the student to decide what the errors may be and how to use them. By replacing each of the blocks of code in the main function with a call to move_arm, the code is cleaner. In a similar fashion, 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 the gripper, you will compartmentalize the calls in the move_block function. In addition to taking ros node information, it also has the service information for interfacing with the gripper. Use this function to compartmentalize moving a block from one tower to another. The start and end locations are integers given to tower positions and the heights are integers for blocks in the stack.

2.8 Report

Each partner must submit a hardcopy of your `lab2.cpp` file with a coversheet containing:

- your name in a larger font and your partner's names in a smaller font.
- "Lab 2"
- the weekday and time your lab section meets (for example, "Monday, 1pm").
- a number of paragraphs explaining in your own words the work you performed to complete this lab.

2.9 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.

2.10 Grading

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

LAB 3

Forward Kinematics

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 lab sessions.

3.2 Objectives

The purpose of this lab is to compare the theoretical solution to the forward kinematics problem with a physical implementation on the UR3 robot. In this lab you will:

- parameterize the UR3 following the Denavit-Hartenberg (DH) convention
- use Robotica to compute the forward kinematic equations for the UR3
- write a C++ function that moves the UR3 to a configuration specified by the user.

3.3 References

- Chapter 3 of SH&V provides details of the DH convention and its use in parameterizing robots and computing the forward kinematic equations.

- The complete Robotica function discription manual is available in pdf form on the course website. Additionally, a "crash course" on the use of Robotica and Mathematica is provided in Appendix A of this lab manual.
- If you prefer Matlab Mahmoud KhoshGoftar has shared a set of M-files that translate DH parameters to tranformation matrixes, "Denavit Hartenberg Parameters" by Mahmoud KhoshGoftar. (https://www.mathworks.com/matlabcentral/fileexchange/44585-denavit-hartenberg-parameters)

3.4 Tasks

3.4.1 Theoretical Solution

Find the forward kinematic equations for the UR3 robot using the Danavit Hartenburg method. Solve for T_6^0 using Robotica and note the equations for the translation vector from the end effector frame to the base frame, d_6^0.

3.4.2 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:

$$\begin{aligned} 60 &< \theta_1 < 315 \\ -185 &< \theta_2 < 5 \\ -10 &< \theta_3 < 150 \\ -190 &< \theta_4 < -80 \\ -120 &< \theta_5 < 80 \\ -180 &< \theta_6 < 180 \end{aligned}$$

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

3.4.3 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 C++ forward kinematics function.

3.5 Procedure

3.5.1 Theoretical Solution

1. Construct a table of DH parameters for the UR3 robot. So it is easier to debug lab issues, use the DH frames already established in Figure 3.1. The dimensions of the UR3 can be found in figure Figure 3.2, except for the suction cup tool offsets. Use a ruler to find the approximate suction cup offsets and then ask your TA to see if you measured correctly.

2. Use Mathematica with the Robotica package to find the translation matrix using the UR3's DH parameters. Modify the given example Mathematica file so that it finds and properly displays the forward kinematic equation for the UR3. Consult Appendix A of this lab manual for assistance with Mathematica.

3. **OR** Use matlab "Denavit Hartenberg Parameters" to calculate the forward kinematics. Use simplify() in matlab as you see fit.

3.5.2 Implementation on UR3

1. Download and extract lab3.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 lab3 starter code correctly. Also so that ROS registers this new package "lab3pkg", run "rospack list" and you should see lab3pkg as one of the many packages.

2. One item to note that hopefully does not cause too much confusion. For this LAB 3 and the remaining labs we will be defining θ_1 to be $180°$ different from what the teach pendant displays. So when you command the robot to move to certain joint angles, you will see that π has been added the θ_1's value. This change in θ_1 makes the DH coordinate systems more standard.

3. You will notice that in lab3pkg/src that there are now two .cpp files. lab3.cpp is the main() code and lab3func.cpp defines the important forward kinematic functions. We divide them up in this fashion so that the forward kinematic functions can easily be used in the remaining labs. A library will be compiled that you can include in your next labs to call the forward kinematic calculations. In this lab you will

be mainly changing lab3func.cpp. You can of course change lab3.cpp but most of its functionality has already been given to you. Study the code and comments in lab3.cpp and lab3func.cpp 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 that correctly populates the six DH homogeneous translation matrixes (DH2HT). The C++ code uses the package "eigen" to create and multiply matrixes. The given code simply multiples six identity matrixes. Your job is to change those identity matrixes to the six DH matrixes for the UR3.

4. Once your code is finished and compiled, run it using "rosrun lab3pkg lab3node theta1 theta2 theta3 theta4 theta5 theta6" with all angles in degrees. Remember that in another command prompt you should have first ran roscore and other drivers using "roslaunch ece470_ur3_driver ece470_ur3_driver.launch"

3.5.3 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. 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 along with the value calculated by Robotica. *The value displayed by your ROS node and Robotica should be identical.* Call these calculated vectors d_1 and d_2. Note that Mathematica expects angles to be in radians, but you can easily convert from radians to degrees by adding the word `Degree` after a value in degrees. For example, `90 Degree` is equivalent to $\frac{\pi}{2}$.

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 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 $error_2$.

3.6 Report

Each lab partner assemble the following items in the order shown.

1. Coversheet containing your name and then your partner's names, "Lab 3", and the weekday and time your lab section meets (for example, "Tuesday, 3pm").

2. A figure of the UR3 robot with DH frames assigned, all joint variables and link lengths shown, and a complete table of DH parameters.

3. The forward kinematic equation as a function of $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \}$ for *the tool position only* of the UR3 robot. Robotica and the matlab "DH parameters" will generate the entire homogenous transformation between the base and tool frames

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

   but you only need to write out the equations for the position of the tool frame with respect to the base frame

   $$d_6^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = [\textit{vector expression}].$$

4. For each of the two sets of joint variables you tested, provide the following:

   - the set of values, $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$
   - the measured position of the tool frame, r_i
   - the predicted position of the tool frame, d_i
   - the (scalar) error between the measured and expected positions, $error_i$.

5. A brief paragraph (2-4 sentences) explaining the sources of error and how one might go about reducing the error.

6. Paragraphs (in your own words) and sketches explaining how you found each of the DH parameters even if that parameter happens to be zero.

3.7 Demonstration

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

3.8 Grading

- 10 points, by the end of the first two hour lab session, show your completed DH parameters 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.

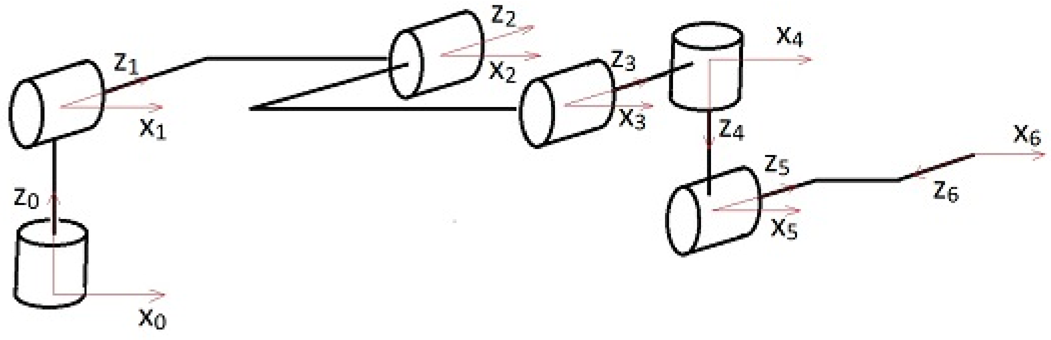


Figure 3.1: UR3 robot with DH frames assigned. Suction Cup Gripper Attached.

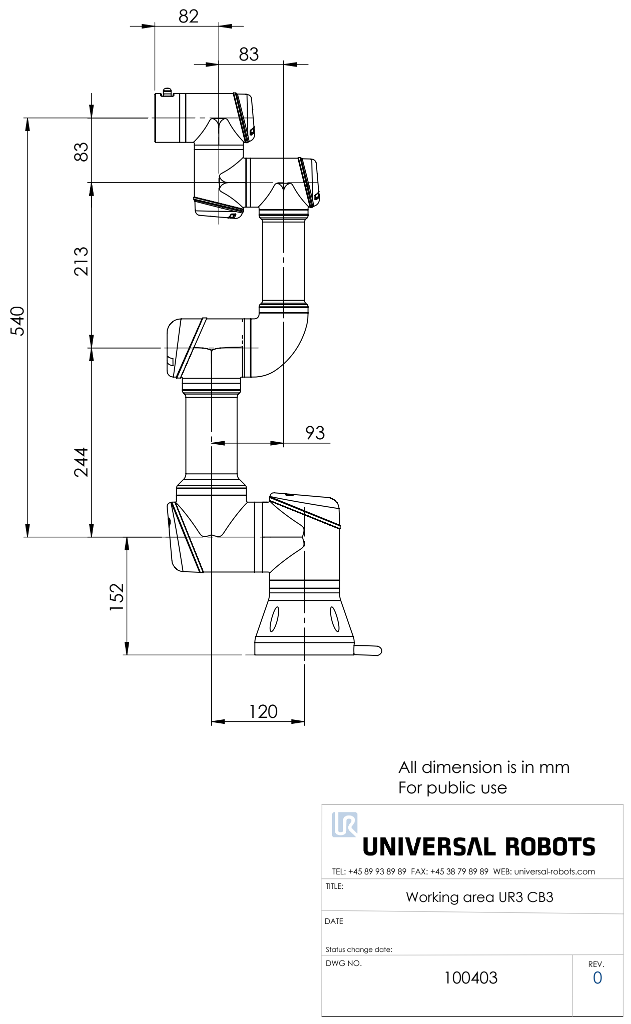


Figure 3.2: Approximate Dimensions of the UR3 in mm.

LAB 4

Inverse 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 check-points by the end of lab sessions.

4.2 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 C++ function that moves the UR3 to a point in space specified by the user.

4.3 Reference

Chapter 3 of SH&V provides multiple examples of inverse kinematics solutions.

4.4 Tasks

4.4.1 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.

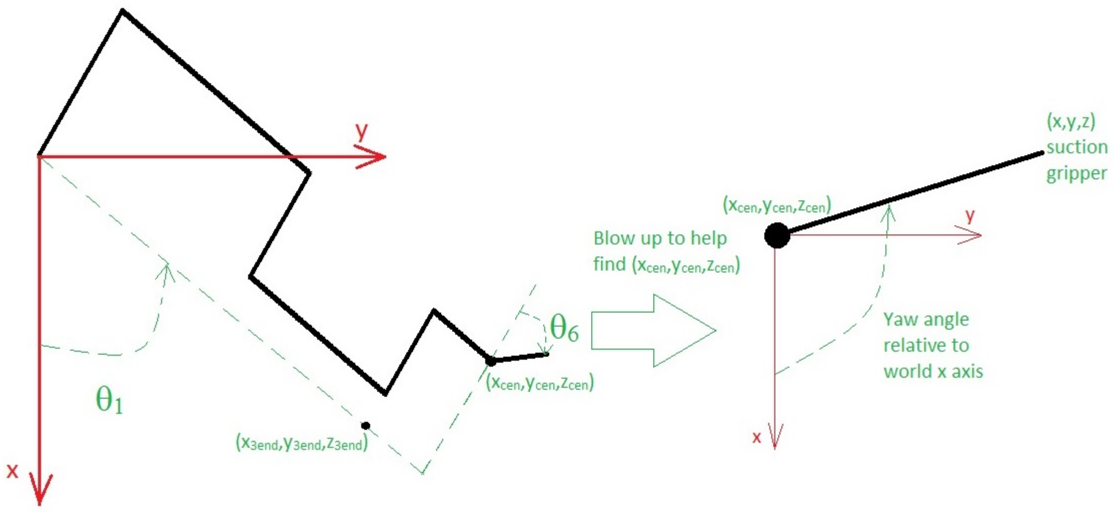


Figure 4.1: Top View Stick Pictorial of UR3. Note that the coordinate frames are in the same direction as the World Frame but not at the World frame's origin. One origin is along the center of joint 1 and the second is along the center of joint 6.

Given a desired suction cup end effector point 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 5 (DH paramater d5) is 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

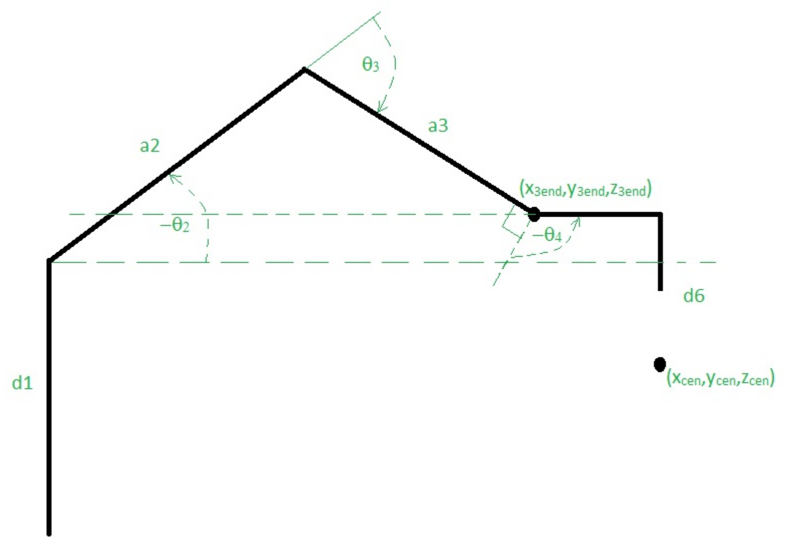

θ3
a3
a2
$(x_{3end}, y_{3end}, z_{3end})$
$-\theta_2$
$-\theta_4$
d6
d1
$(x_{cen}, y_{cen}, z_{cen})$

Figure 4.2: Side View Stick Pictorial of UR3 without Suction Gripper.

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 4.1.

2. θ_1, by drawing a top down picture of the UR3, Figure 4.1, and using x_{cen}, y_{cen}, z_{cen} that you just calculated. Remember that link 5 is always parallel the to table.

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 DH parameter d2.

4. x_{3end}, y_{3end}, z_{3end} is a point off of the UR3 but lies along the z_3 DH axis, Figure 4.1. 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 4.2, 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 4.2. 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. Note that in the side view the length link 1 is d1, link2 is a2 and link3 is a3. 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.

4.4.2 Implementation

Implement the inverse kinematics solution by writing a C++ 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 LAB3.

4.5 Procedure

- Download lab4.tar.gz from the course website and extract it in your "src" directory. You will notice again that there are two .cpp files. lab4.cpp and lab4func.cpp. The lab4func.cpp file again will be compiled into a library so that future labs can easily call the inverse kinematic function. As in LAB 3, most of the needed code is given to you in lab4.cpp. Your main job will be to add all the inverse kinematic equations to lab4funcs.cpp. Note that it will be very helpful to add worker variables in your C code to perform the inverse kinematic calculations. If you look at lab4.h it includes lab3.h. This allows you to call the functions your created in lab3funcs.cpp.

- 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 4.3. 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

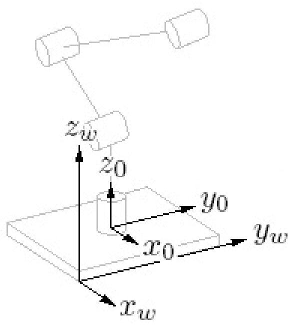


Figure 4.3: Correct location and orientation of the world frame.

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.

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

Be careful to reference the location of frame 0 as your team defined it in LAB 3.

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 $(x_{cen}, y_{cen}, z_{cen})$.

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

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.

$$\theta_1(x_{cen}, y_{cen}, z_{cen}) = \tag{4.1}$$

4. Solve for the value of θ_6, given yaw and θ_1.
$$\theta_6(\theta_1, yaw) = \tag{4.2}$$

5. Find the projected link 3 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 link 3 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 LAB3 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 4's new world 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.

4.6 Report

Each partner assemble the following items in the order shown.

1. Coversheet containing your name and your partner's names, "Lab 4", and the weekday and time your lab section meets (for example, "Tuesday, 3pm").

2. A cleanly 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.* Please be kind to your TA and invest the effort to make your diagrams clean and easily readable.

3. For each of the two sets of positions and orientations you demonstrated, provide the following:
    - the set of values $\{(x_{Wgrip}, y_{Wgrip}, z_{Wgrip}), yaw\}$
    - the measured location of the tool
    - the (scalar) error between the measured and expected positions.
4. A brief paragraph (2-4 sentences) explaining the sources of error and how one might go about reducing the error.
5. Print out of your code.

4.7 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.

4.8 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 C code.
- 50 points, successful demonstration.
- 20 points, report.

LAB 5

Image Processing

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 lab sessions.

5.2 Objectives

This is the first of two labs whose purpose is to integrate computer vision and control of the UR3 robot. In this lab we will:

- separate the objects in a grayscaled image from the background by selecting a threshold greyscale value
- identify each object with a unique color
- eliminate misidentified objects and noise from image
- determine the number of significant objects in an image.

5.3 References

- Chapter 11 of SH&V provides detailed explanation of the threshold selection algorithm and summarizes an algorithm for associating the objects in an image. Please read all of sections 11.3 and 11.4 before beginning the lab.

- Appendix C of this lab manual explains how to work with image data in your code. Please read all of sections C.1 through C.3 before beginning the lab.

5.4 Tasks

5.4.1 Separating Objects from Background

The images provided by the camera are colored, and then converted to gray grayscaled. That is, each pixel in the image has an associated grayscale value 0-255, where 0 is black and 255 is white. We will assume that the image can be separated into background (light regions) and objects (dark regions). We begin by surveying the image and selecting the grayscale value z_t that best distinguishes between objects and the background; all pixels with values $z > z_t$ (lighter than the threshold) will be considered to be in the background and all pixels with values $z \leq z_t$ (darker than the threshold) will be considered to be in an object.

We will implement an algorithm that minimizes the within-group variance between the background and object probability density functions (pdfs). Once the threshold value has been selected, we will replace each pixel in the background with a white pixel ($z = 255$) and each pixel in an object with a black pixel ($z = 0$).

5.4.2 Associating Objects in the Image

Once objects in the image have been separated from the background, we want to indentify the separate objects in the image. We will distinguish among the objects by assigning each object a unique color. The pegboards on the workspace will introduce many small "dots" to the image that will be misinterpreted as objects; we will discard these false objects along with any other noise in the image. Finally, we will report to the user the number of significant objects identified in the image.

5.5 Procedure

5.5.1 Separating Objects from Background

1. Read section 11.3 in SH&V and Appendix C in this lab manual before proceeding further.

2. Download the following files from the course website, extract to the src folder of your catkin workspace:

   ```
   lab56.tar.gz
   ```

   The cv_camera package is the driver of the camera, the lab56 package contains the codes for lab 5 and 6. You should go over Appendix C for details of the code.

3. Compile the workspace, run the UR3 and camera driver (described in Appendix C).

   ```
   $ roslaunch ece470_ur3_driver ece470_lab56_driver.launch
   ```

   Then open another shell (source setup.bash as always) and run lab5's node:

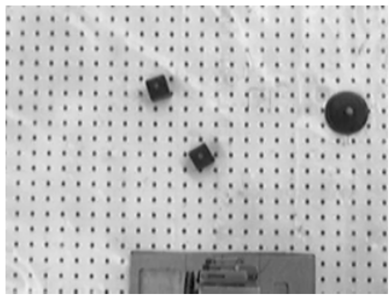
   ```
   $ rosrun lab56pkg lab56node
   ```

   You will see 4 video windows (some could be hidden behind others), and if you left or right click on one of the windows, the position of the point you clicked is shown on the shell. You will modify the code "lab56.cpp" so that the 4 windows will show an original colored image (already there), a gray scaled image (already there), a thresholded image and an image showing different objects with a different color.

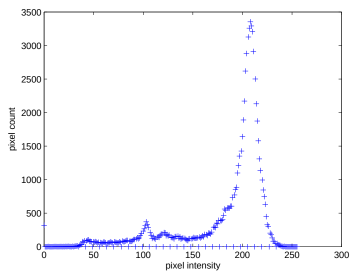
4. We will first need to build a histogram for the grayscaled image. Define an array `H` with 256 entries, one for each possible grayscale value. Now examine each pixel in the image and tally the number of pixels with each possible grayscale value. An example grayscaled image and its histogram are shown in Figure 5.1.

5. The text provides an efficient algorithm for finding the threshold grayscale value that minimizes the within-group variance between background and objects. Implement this algorithm and determine threshold z_t. Some comments:

   - Do not use integer variables for your probabilities.
   - When computing probabilites (such as $\frac{H[z]}{N \times N}$) be sure the number in the numerator is a floating point value. For example, `(float)H[z]/NN` will ensure that c++ performs floating point division.



(a)



(b)

Figure 5.1: A sample image of the tabletop (a) the grayscaled image (b) the histogram for the image.

- Be aware of cases when the conditional probability $q_0(z)$ takes on values of 0 or 1. Several expressions in the iterative algorithm divide by $q_0(z)$ or $(1 - q_0(z))$. If you implement the algorithm blindly, you will probably divide by zero at some point and throw off the computation.

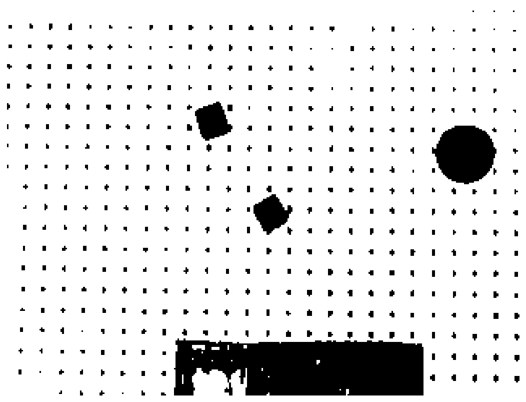


Figure 5.2: Sample image after thresholding.

6. Again, consider each pixel in the image and color it white if $z > z_t$, black if $z \leq z_t$. Figure 5.2 shows the same image from Figure 5.1 (a) after thresholding.

5.5.2 Associating Objects in the Image

1. Read section 11.4 in SH&V and Appendix C in this lab manual before proceeding further.

2. Implement an algorithm that checks 4-connectivity for each pixel and relates pixels in the same object. The difficult part is noting the equivalence between pixels with different labels in the same object. There are many possible ways to accomplish this task; we outline two possible solutions here, although you are encouraged to divise your own clever algorithm.

    - A simple but time consuming solution involves performing a raster scan for each equivalence. Begin raster scanning the image until an equivalence is encountered (for example, between pixels with label 2 and pixels with label 3). Immediately terminate the raster scan and start over again; every time a pixel with label 3 is found, relabel the pixel with 2. Continue beyond the point of the first equivalence until another equivalence is encountered. Again, terminate the raster scan and begin again. Repeat this process until a raster scan passes through the entire image without noting any equivalencs.

- An alternative approach can associate objects with only two raster scans. This approach requires the creation of two arrays: one an array of integer labels, the other an array of pointers for noting equivalences. It should be noted that this algorithm is memory expensive because it requires two array entries for each label assigned to the image. Consider the following pseudocode.

```
int label[100];
int *equiv[100];
int pixellabel[height][width];

initialize arrays so that:
equiv[i] = &label[i]
pixellabel[height][width] = -1 if image pixel is white
pixellabel[height][width] = 0 if image pixel is black

labelnum = 1;
FIRST raster scan
{
  Pixel = pixellabel[row][col]
  Left  = pixellabel[row][col-1]
  Above = pixellabel[row-1][col]

  you will need to condition the
  assignments of left and above
  to handle row 0 and column 0 when
  there are no pixels above or left

  if Pixel not in background (Pixel is
                     part of an object)
  {

    if (Left is background) and
          (Above is background)
    {
      pixellabel[row][col] = labelnum
      label[labelnum] = labelnum
      labelnum ++
    }
    if (Left is object) and
                     (Above is background)
      pixellabel[row][col] = Left

    if (Left is background) and
                     (Above is object)
      pixellabel[row][col] = Above

    EQUIVALENCE CASE:
    if (Left is object) and
            (Above is object)
    {
      smallerbaselabel = min{*equiv[Left],
       *equiv[Above]}

      min = Left if smallerbaselabel==
                          *equiv[Left]
                 else min = Above

      max = the other of {Left, Above}

      pixellabel[row][col] =
                     smallerbaselabel

      *equiv[max] = *equiv[min]
      equiv[max] = equiv[min]
    }
  }
}

Now assign same label to all pixels in
  the same object
SECOND raster scan
     Pixel = pixellabel[row][col]
     if Pixel not in background (Pixel is
                        part of an object)
          pixellabel = *equiv[Pixel]
```

For an illustration of how the labels in an image change after the first and second raster scans, see Figure 5.3. Figure 5.4 shows how equivalence relations affect the two arrays and change labels during the first raster scan.

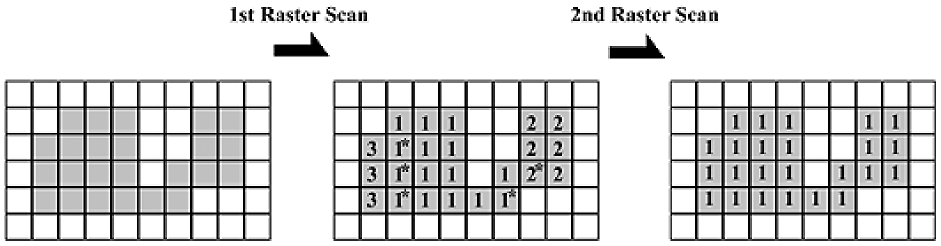


Figure 5.3: Pixels in the image after thresholding, after the first raster scan, and after the second raster scan. In the first image, there are only black and white pixels; no labels have been assigned. After the first raster scan, we can see the labels on the object pixels; an equivalence is noted by small asterisks beside a label. After the second raster scan, all the pixels in the object bear the same label.

3. Once all objects have been identified with unique labels for each pixel, we next perform "noise elimination" and discard the small objects corresponding to holes in the tabletop or artifacts from segmentation. To do this, compute the number of pixels in each object. We could again implement a within-group variance algorithm to automatically determine the threshold number of pixels that distinguishes legitimate objects from noise objects, but you may simply choose a threshold pixel count yourself. For the objects whose pixel count is below the threshold, change the object color to white, thereby forcing the object into the background. Figure 5.5 provides an example of an image after complete object association and noise elimination.
4. Report to the user the number of legitimate objects in the image.

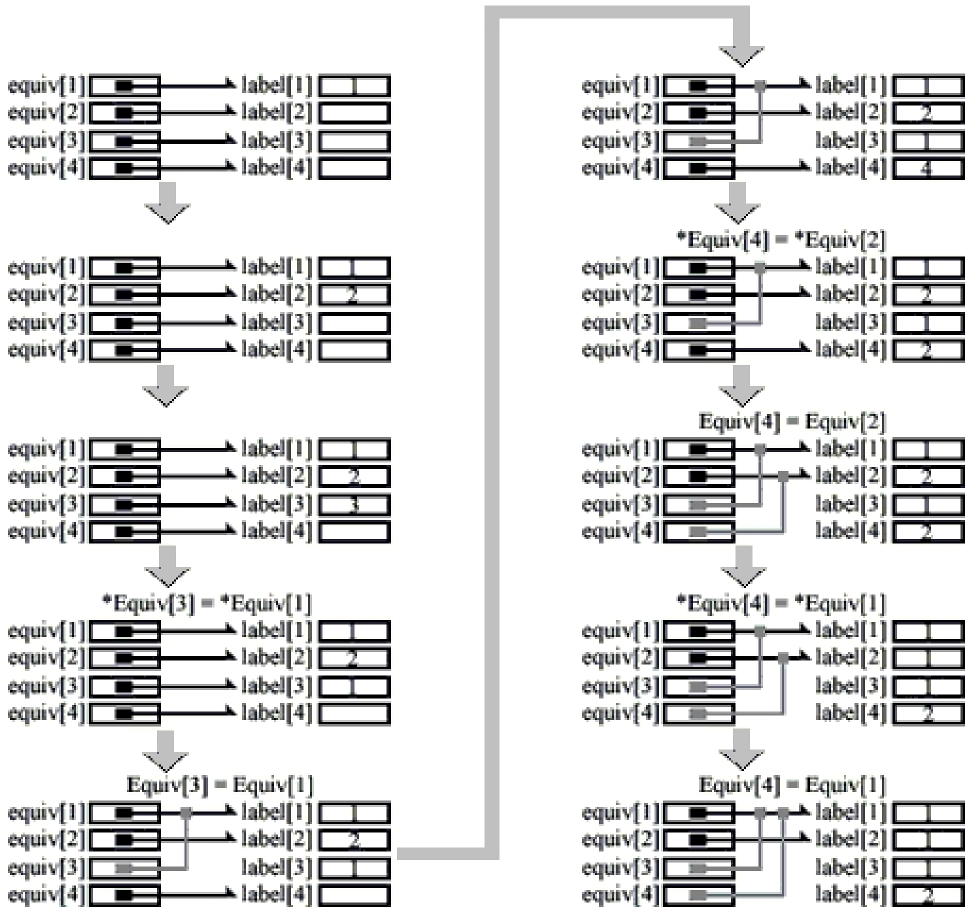


Figure 5.4: Evolution of the pixel labels as equivalences are encountered.

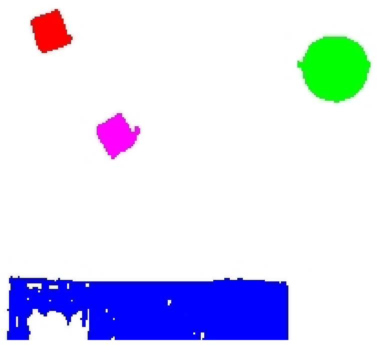


Figure 5.5: Sample image after object association.

5.6 Report

As a group, submit your lab56.cpp file by emailing it as an attachment to your TA. First, rename the file with the last names of your group members. For example, if Barney Rubble and Fred Flintstone are in your group, you will submit `RubbleFlintstone5.cpp`. Make the subject of your email "Lab 5 Code." Individually, write a number of paragraphs explaining your associating objects algorithm from section 5.5.2. It may help to intermix your source code with this explanation.

5.7 Demo

You will demonstrate your working solution to your TA with various combinations of blocks and other objects.

5.8 Grading

- 10 points, by the end of the first two hour lab session, demonstrate to your TA a reasonable attempt to finish the work of section 5.5.1.
- 10 points, by the end of the second two hour lab session, demonstrate to your TA that you have finished the work in section 5.5.1. Additionally demonstrate to your TA a reasonable attempt to finish the work of section 5.5.2.

- 60 point, successful demonstratoin.
- 20 points, report.

LAB 6

Camera Calibration

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 lab sessions.

6.2 Objectives

This is the capstone lab of the semester and will integrate your work done in labs 3-5 with forward and inverse kinematics and computer vision. In this lab you will:

- find the image centroid of each object and draw crosshairs over the centroids
- develop 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 object in the image
- Using the prewritten point-and-click functions, command the robot to retrieve a block placed in veiw of the camera and move it to a desired location.

6.3 References

- Chapter 11 of SH&V explains the general problem of camera calibration and provides the necessary equations for finding object centroids. Please read all of sections 11.1, 11.2, and 11.5 before beginning the lab.
- Appendix C of this lab manual explains how to simplify the intrinsic and extrinsic equations for the camera. Please read before beginning the lab.

6.4 Tasks

6.4.1 Object Centroids

In lab 5, we separated the background of the image from the significant objects in the image. Once each object in the image has been distinguished from the others with a unique label, it is a straightforward task to indentify the pixel corresponding to the centroid of each object.

6.4.2 Camera Calibration

The problem of camera calibration is that of relating (row,column) coordinates in an image to the corresponding coordinates in the world frame (x_w, y_w, z_w). Chapter 11 in SH&V presents the general equations for accomplishing this task. For our purposes we may make several assumptions that will vastly simplify camera calibration. Please consult section C.3 in this lab manual and follow along with the simplification of the general equations presented in the textbook.

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

The final task of this lab integrates the code you have written for labs 3-5. Your LAB 5 code provides the processed image from which you have now generated the world coordinates of each object's centroid. We can relate the unique color of each object (which you assigned in LAB 5) with the coordinates of the object's centroid. Using the prewritten point-and-click functions, you may click on an object in an image and feed the centroid coordinates to your LAB 4 inverse kinematics function. Recall that the LAB4 function lab_invk() given the desired x,y,z world position returns the needed joint angles. Publish these joint angles to the ur3/command topic.

You will bring together your LAB 3-5 code and the prewritten point-and-click functions in the following way:

- the user will left click on a block in the image console,
- your code will command the UR3 to move to a pose above that block,
- command the UR3 to grip the block'
- return to the home position with the block in its grip,
- the user will click on an unoccupied portion of the image,
- the UR3 will move to the corresponding region of the table and release the block.

6.5 Procedure

6.5.1 Object Centroids

1. Read section 11.5 in SH&V before proceeding further.
2. Edit the `associateObjects` function you wrote for lab 5. Implement the centroid computation equations from SH&V by adding code that will identify the centroid of each significant object in the image.
3. Display the row and column of each object's centroid to the user.
4. Draw crosshairs in the image over each centroid.

6.5.2 Camera Calibration

1. Read sections 11.1 and 11.2 in SH&V and section C.3 in this lab manual before proceeding further. Notice that, due to the way row and column are defined in our image, the camera frame is oriented in

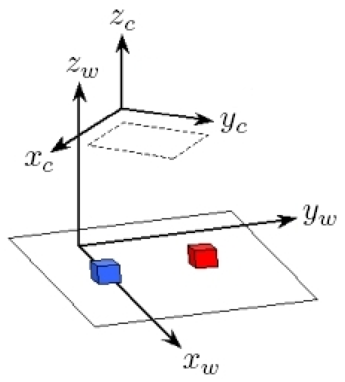
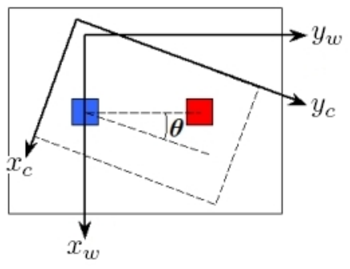


Figure 6.1: Arrangement of the world and camera frames.

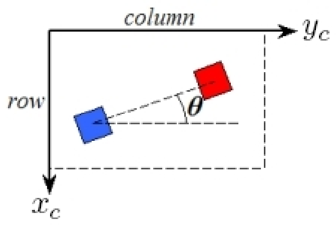
a different way than given in the textbook. Instead, our setup looks like Figure 6.1 in this lab manual.

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 the textbook and simplified in section C.3 of this lab manual. Write equations for the world frame coordinates in terms of the image coordinates.

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



(a)



(b)

Figure 6.2: (a) Overhead view of the table; the rectangle delineated by dashed lines represents the camera's field of vision. (b) The image seen by the camera, showing the diagonal line formed by the blocks. Notice that x_c increases with increasing row values, and y_c increases with increasing column values.

3. There are six unknown values we 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.

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

   The remaining paramaters β, θ, T_x, T_y will change every time the camera is tilted or the zoom is changed. Therefore you must recalibrate these parameters each time you come to the lab, as other groups will be using the same camera and may reposition the camera when you are not present.

4. 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 coorespoinding points in the image is βd. Place two blocks on the table in view of the camera. Measure the distance between the centers of the blocks using a rule. In the image of the blocks, use centroids to compute the pixels between the centers of the blocks. Calculate the scaling constant.

$$\beta =$$

5. θ is the angle of rotation between the world frame and the camera frame. Please refer to section C.3 of this lab manual for an explanation of why we need only one angle to define this rotation instead of two as described in the textbook. Calculating this angle isn't difficult, but it is sometimes difficult to visualize. Place two blocks on the table in view of the camera and arranged in a line parallel to the world y axis as shown in Figure 6.1. Figure 6.2 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 in a diagonal line in the image. Using the centroids of the two blocks and simple trigonometric fuctions, compute the angle of this diagonal line and use it to find θ, the angle of rotation between the world and camera frames.

$$\theta =$$

6. The final values remaining for calibration are T_x, T_y, the coordinates of the origin of the world frame expressed in the camera frame. To find these values, measure the world frame coordinates of the centers of two blocks; also record the centroid locations produced by your code. Substitute these values into the two equations you derived in step 2 above and solve for the unknown values.

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

7. Finally, insert the completed equations in your `lab56.cpp` file in the onMouse() function. Your code should report to the user the centroid location of each object in the image in (row,column) and world coordinates (x_w, y_w).

6.5.3 Pick and Place

When the user clicks inside an image, the row and column position of the mouse is passed into the "onMouse" function in lab56.cpp.

1. The OnMouse function should enable you to left click onto one of the blocks on the screen and then tell the robot to grip the block and return to an intermediate position. Inside the OnMouse callback function first convert the pixel coordinate to world coordinates. Then pass this coordinate to LAB4's lab_invk() function. lab_invk() returns the need joint angles. Publish these joint angles to the ur3/command topic. Finally move the robot to an intermediate point waiting for a right click. After you get this working well you may want to add a few other points in robot's movement so that it doesn't hit other possible blocks in the robot's work space.

2. On a right click, move the block to the right clicked position and release the block. Return to a start position that you choose.

6.6 Report

As a group, submit your `lab56.cpp` file by emailing it as an attachment to your TA. First, rename the file with the last names of your group members. For example, if Barney Rubble and Fred Flintstone are in your group, you will submit `RubbleFlintstone6.cpp`. Make the subject of your email “Lab 6 Code.” Individually wirte a number of paragraphs explaining the algorithm you implemented to calibrate the camera coordinate frame into the UR3’s world frame in section 6.5.2.

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. For the pick and place task, you will also demonstrate that the robot will retrieve a block after you left click on it in the image. The robot must also place the block in another location when you right click on a separate position in the image.

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 6.5.2.
- 10 points, by the end of the second two hour lab session, show your TA that you have finished section 6.5.2 and have made good progress on section 6.5.3.
- 60 points, successful demonstration.
- 20 points, report.

LAB 7

Extra Credit

7.1 Objectives

In Lab 6, you combined the work of all the previous labs to perform a simple computer vision task. The demonstration in lab 6 is not particularly useful on its own but can be thought of as a proof of concept or motion primitive for a more complex application.
For extra credit you may choose to complete any of the following objectives.
Do not let your work on these projects interfere with the completion of your regularly scheduled lab work either in time management or corrupting code. Furthermore, the maximum grade you can receive from doing extra credit in the lab class is 100%.

1. **ROS:** ROS is a powerful set of software libraries and is easy to use once it has been set up. For this task you are required to create a single package which combines all the functionality of the previous lab assignments.

2. **Precision Manipulation** In lab 6 you created a function that converted pixel coordinates to world coordinates. However this only worked in a single plane. Extend this function to work for blocks placed at any height and anywhere in the image. You may prompt the user for the height of the block.

3. **Computer Vision Algorithms:** Measure and label each stack of blocks in the image with its height using nothing but the image on the screen.

4. **Automatic Tower of Hanoi:** Combine parts 2 and 3 to recursively play the Tower of Hanoi game with the tallest stack of blocks in the image.

7.2 References

1. **ROS:**
   - Look at the package files "CMakeLists.txt" and "package.xml" in lab 3 for an example.
   - Packages: `http://wiki.ros.org/ROS/Tutorials/CreatingPackage`
   - CMakeLists: `http://wiki.ros.org/catkin/CMakeLists.txt`
2. **Precision Manipulation**
   - Chapter 11 of SH&V explains the general problem of camera calibration and one method of finding the camera parameters.
   - Both OpenCV and Mathworks have methods for generating the camera parameter and rotation matrices you need to describe the position of the camera in terms of world coordinates. Feel free to use either.
     - OpenCV documentation: Camera Calibration and 3D Reconstruction.
     - Mathworks `https://www.mathworks.com/help/vision/ug/single-camera-calibrator-app.html`
   - A calibration checkerboard can be found at: `http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration?action=AttachFile&do=view&target=check-108.pdf`
3. **Computer Vision Algorithms:**
4. **Automatic Tower of Hanoi:**
   - The recursive Tower of Hanoi solution can be found at `http://www.cut-the-knot.org/recurrence/hanoi.shtml` .

7.3 Tasks

7.3.1 ROS

For this assignment you will be creating your own ROS package using the roscreate-pkg command. In the previous labs, each ROS package was a associated with a single node, e.g. lab3pkg lab3node. For this assignment you will be creating a new package that depends only on the drivers package given to you and not on any of the lab assignment packages. Expected Hierarchy:

- ece470pkg
  - gohome - move arm into home position.
  - goaway - move arm out of image frame.
  - gostandby - move arm into predefined standby position away from table.
  - movej - move to joint angles with angles given in degrees as arguments.
  - movel - move tool tip to rectilinear location x y z yaw in world coordinates.
  - locate - returns current joint angles, tool location in both robot base and world frame.
  - camera_view - displays current camera image.
  - camera_process - displays all the processed image windows required for lab 5.
  - camera_move - move the arm to position of click in image window.
  - hanoi - prompt user for input and executes Tower of Hanoi program.

Because the functionality used in these nodes is shared, you will be required to create at least one library file like lab3func.cpp which contains all the functions your nodes will need (you may have more library files if you prefer).

7.3.2 Precision Manipulation

Much like lab 6 you are being asked to click on a point in the camera image and have the arm go to that location. However this is now a three dimensional problem and accuracy is required throughout the image. You will be given a stack of blocks placed randomly in the image and when the user clicks on the stack, the program should prompt the user for the height of the blocks. With this information the suction cup should move directly over the top of the stack.

7.3.3 Computer Vision Algorithms

In lab 5 you were asked to implement an image processing algorithm which classified objects and determined their centroid. For this assignment you will need to design and implement an algorithm to determine the height of a stack of blocks from the image.
To determine the height, you will need to identify some feature of the stack which is known and varies with height. For example, we know the width of the block and as it gets closer to the camera it will look larger. From this information we can solve for the height.

7.3.4 Automatic Tower of Hanoi

The goal of this assignment is to move a stack of blocks, with arbitrary height and location, to a preset final location without violating the rules of the Tower of Hanoi game. Full credit is awarded for fully automatic completion of the game, while user input for location and height of the stack will result in partial credit.

7.4 Procedure

Since this is an extra credit assignment, no procedure is given. There are some hints about how you may wish to proceed but you are not bound to follow them if you can complete the requirements in some other fashion.

7.4.1 ROS

Don't try and do all of this at once. Lab 3 provide a good example of how to handle libraries and node creation. Start with a single node (maybe the locate node) and then add the other nodes one at a time.

7.4.2 Precision Manipulation

Use Matlab's cameracalibrator command to find the camera calibration and rotation matrices to go from image to world frame and subtract an offset to based on where your world origin is located. Then use the knowledge of the camera height and the height of the block stack to determine where the arm should go based on the distance of the stack from the image center.

7.4.3 Computer Vision

The top plane of this block is the only feature that you can see when the tower is directly under the camera so it may be the feature you try and identify. You may also be able to use some knowledge of how the centroid is located in a tower to estimate the height of the blocks.
For the demonstration, you are allowed to choose the color of your top block. Use this to help you consistently identify block features.

7.4.4 Automatic Tower of Hanoi

This assignment relies on several of the other parts and even if the full project is out of reach the partial completion will still get you some points.

7.5 Report

No report is necessary but you must submit your code by emailing it as an attachment to your TA using the same naming convention as the previous labs.

7.6 Demo

ROS:

- Show successful compilation of the ece470pkg.
- Show node definitions, dependencies and include files for all function libraries and nodes.
- Show that motion nodes move to the correct locations.
- Show that the camera nodes produce the correct images and can be used to move the robot.

Precision Manipulation:

- Block stacks of heights between 1 and 6 blocks will be placed in all four quadrants (relative to world origin) of the camera image. Your program should be able to place the tool tip above the top block when you click on the top of the stack and input the height of the stack.

Computer Vision:

- Block stacks with heights between 1 and 6 blocks will be placed in the camera image. When you click on the stacks in the image, the height in blocks should be displayed.
- Since computer vision can be tricky, students may use whatever color block they wish for the top of the stack.

Automatic Tower of Hanoi:

- A source tower of 3-6 blocks will be placed somewhere in the camera image. You must have it move the stack to a predefined destination location without violating the game rules. Full credit is awarded if no user input is required. Partial credit is awarded if the stack location and height need to be provided.

7.7 Grading

Each of the assignments is worth 1 bonus point and a total of 4 points can be earned for the completion of all extra assignments. Please note that the maximum grade you can receive for the entire lab course 100%. Extra credit above 100% does not apply.

1. **ROS:**
    - 12.5 points out of 100 - successful demonstration of all motion nodes.
    - 12.5 points out of 100 - successful demonstration of all camera based nodes.
2. **Precision Manipulation:**
    - 12.5 points out of 100 - suction cup above block for 3/4 of image space, some errors allowed.
    - 12.5 points out of 100 - suction cup in center of block for entire work area.

3. **Computer Vision:**

   - 12.5 points out of 100 - Successfully identify known feature that varies with height. Show the geometry and explain that you have all the parameters necessary to solve for the height of the stack.
   - 12.5 points out of 100 - Successfully estimate block height from image.

4. **Automatic Tower of Hanoi:**

   - 12.5 points out of 100 - Recursively solve arbitrary Tower of Hanoi with user input of source stack height and source location via mouse click.
   - 12.5 points out of 100 - Recursively solve Tower of Hanoi based only on image data.

Appendix A

Mathematica and Robotica

A.1 Mathematica Basics

- To execute a cell press <`Shift`>+<`Enter`> on the keyboard or <`Enter`> on the numberic keypad. Pressing the keyboard <`Enter`> will simply move you to a new line in the same cell. In addition you can select the ”Evaluation–Evaluate Notebook” menu item and all your cells will be evaluated.

- Define a matrix using curly braces around each row, commas between each entry, commas between each row, and curly braces around the entire matrix. For example:

$$M = \{\{1,7\},\{13,5\}\}$$

- To display a matrix use `MatrixForm` which organizes the elements in rows and columns. If you constructed matrix `M` as in the previous example, entering `MatrixForm[M]` would generate the following output:

$$\begin{pmatrix} 1 & 7 \\ 13 & 5 \end{pmatrix}$$

  If your matrix is too big to be shown on one screen, Mathematica and Robotica have commands that can help (see the final section of this document).

- *To multiply matrices do not use the asterisk.* Mathematica uses the decimal for matrix multiplication. For example, `T=A1.A2` multiplies matrices A1 and A2 together and stores them as matrix T.

- Notice that Mathematica commands use square brackets and are case-sensitive. Typically, the first letter of each word in a command is capitalized, as in `MatrixForm[M]`.

- Trigonometric functions in Mathematica operate in radians. It is helpful to know that π is represented by the constant `Pi` (Capital 'P', lowercase 'i'). You can convert easily from a value in degrees to a value in radians by using the command `Degree`. For example, writing `90 Degree` is the same as writing `Pi/2`.

A.2 Robotica

- Robotica was written here at the University of Illinois in 1993 by John Nethery and M.W.Spong. In 2016 Mohammad Sultan and Aaron T. Becker from the University of Houston completely refreshed and enhanced Robotica and have given it the version 4.0. The easiest way to use Robotica is just by example. The below steps are going to have you check out Robotica and run the given example. Then you simply need to modify the example with the DH parameters for the UR3. NOTE: Robotica displays a stick figure of the manipulator associated with your DH parameters. To make the stick figure look reasonable for the UR3 use units of decimeters instead of meters or millimeters.

- First open a CMD prompt in Windows and change directory to where you would like to store your Robotica files. At the CMD prompt type:
  ”git clone https://github.com/RoboticSwarmControl/robotica”.
  This will check out Robotica and an example notebook robotica_V4_example.nb.

- Start Mathematica and load robotica_V4_example.nb. Once loaded you will see that this example is using a robot configuration that has a prismatic joint and two revolute joints. If the stick figure robot picture is not displayed select the Evaluation–Evaluate Notebook menu item to run the notebook.

- Modify the ”ex” matrix, which is a matrix of all the DH parameters, for the UR3. The matrix is made up of five rows. Row 1 is specifying if the movable joint is prismatic ’p’ or revolute ’r’. Row 2 DH paramter ’a’ (called ’r’ here in Robotica V4.0). Row 3 DH parameter

'α'. Row 4 DH parameter 'd'. Row 5 DH parameter 'θ'. Note that the variables are called q1,q2,.... REMEMBER to use units of decimeters for the UR3.

- Change the dimensions of the A and T matrixes being printed to 6th order instead of 3rd order.
- Once you have the UR3 DH parameters entered run the menu command Evaluation–Evaluate Notebook and after a few seconds the UR3 stick figure should be displayed.
- Now you can use the given controls to move around the animation of the UR3. Note the equations for the T_6^0. Also note in the first picture of the robot the H matrix (T_6^0) is calculated for the thetas that were set by the given sliders.

A.3 What Must Be Submitted with Robotica Assignments

For homework and lab assignments requiring Robotica, you must submit each of the following:

1. figure of the robot clearly showing DH frames and appropriate DH parameters
2. table of DH parameters
3. matrices relevant to the assignment or application, simplified as much as possible and displayed in `MatrixForm`.

Do not submit the entire hardcopy of your Mathematica file. Rather, cut the relevant matrices from your print out and paste them onto your assignment.

Appendix B

C Programming in ROS

B.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

B.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 ece470_ur3_driver ece470_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.

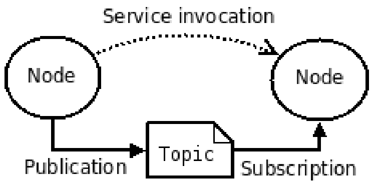


Figure B.1: source: http://wiki.ros.org/ROS/Concepts

B.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, you can also use "ctrl+z". You have to be carefull with "ctrl+z" though as it can leave some 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' package, you can type in your console

  ```
  $ rospack find lab2pkg
  ```

- To move directly to the directory of a ROS package, use roscd. For example, go to lab2pkg package directory

  ```
  $ roscd lab2pkg
  ```

- 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
#Publish data to topic.
$ rostopic pub <topic-name> <topic-type> [data...]
```

B.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
```

B.5 Running A Node

- Once you have your catkin folder initialized, add the UR3 driver and lab starter files. The compressed file lab2andDriver.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 lab2andDriver.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 ece470_ur3_driver ece470_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. IMPORTANT: This launch file needs to know the IP address of your bench's UR3 robot arm. If you just ran the roslaunch command you may have recieved an error. Enter Ctrl-C to exit the roslaunch command. This error is either due to a wrong IP address in the launch file or your PC is not currently connected to the correct network card. Your Linux PC has two ethernet cards installed and their connections are call "Internet" and "Robot Connection." In the top right corner of the Linux Desktop, select the network icon and check that both connections are connected. To find your robot's IP address you will need to use the UR3's teach pendant. Go to the "Robot Setup" item in the main screen and then select "Network". Note your robot's IP address. Then edit catkin_(yourNETID)/src/drivers/ece470_ur3_driver/launch/ece470_ur3_driver.launch and change the IP address to your robot's IP address.

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

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

  ```
  $ rosrun lab2pkg lab2node
  ```

B.6 Simple Publisher and Subscriber Tutorial

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

Appendix C

Notes on Computer Vision

C.1 OpenCV in ROS

For the first four labs, we have used the text-only remote environment to interface with the UR3. In labs 5 and 6, we will work with computer vision and consequently will need to view images as we process them. We will use OpenCV library in ROS to interface with the camera and the UR3 simultaneously. You don't have to install this library since it's already included in ROS Indigo or any newer version of ROS.

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 ece470_ur3_driver ece470_lab56_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".

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 5, the conversion is done in the class ”ImageConverter”, this class subscribes to the image published by the camera, converted to an OpenCV image, then converted back to a ROS image and finaly published to a new topic:

```
/image_converter/output_video
```

It’s necessary to briefly explain the structure of the provided code for LAB 5. 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
{
public:

  ImageConverter()
  {
   the subscriber goes here, it will convert received ROS image to
   OpenCV image then call the function imageCb() which is public for
   this class,when a new image is read.
  }
  ~ImageConverter()
  {
  }
  void imageCb()
  {
    This function will be called whenever a new image is received, then we
    call the private functions thresholdImage() and associateObjects() to
    threshold the image and associate objects in image. Then show them in
    some windows.
  }
private:
  Mat thresholdImage()
  {
This private function takes a gray scale image as input, gives a black
and white binary image as output.
  }
  Mat associateObjects()
  {
This private function takes a black and white image as input and then associates
the objects in the image.
  }
};

void onMouse()
{
This is a callback function outside of the class, it's called whenever a mouse
click is detected.
}

int main(int argc, char** argv)
{
  In main, we just need to declare the class:

  ImageConverter ic;

  return 0;
}
```

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. ImageConverter() is the destructor.

- imageCb(): A public function of this class. It means we can call this function outside of the class by refering to ImageConverter::imageCb(). We'll call some private functions here.

- thresholdImage() and associateObjects(): Private functions, which can only be used inside the class. We will modify the code for these two functions.
- onMouse(): A callback function outside of the class. This function will be run whenever there's a mouse click detected.
- 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.

The OpenCV image is of class cv::Mat. You can find documentation for this class at:

http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html#*mat*

We will only be using a few of this classes functionality. One such item is the clone() member function. This will make a copy of the captured image.

```
cv::Mat image = some_existing_image.clone();
```

We can read the number of rows and columns:

```
int height = image.rows;
int width = image.cols;
```

Most importantly, the pixel values are stored in it's vector member "data", we can read and write the value of each pixel:

```
uchar pixel_value; // pixel value if type of uchar
pixel_value = gray_img.data[10]; //read the 10th pixel value
bw_img.data[11]= 255; // change the 11th pixel to white
// Note: that gray_img.data[] is a one dimensional array
// but stores all rows and columns of the entire image. Its
// index is calculated by index = row*COLUMN_LENGTH+column
```

C.1.3 Some Useful OpenCV Functions

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

- cvtColor(): Converts an image from one color space to another. Example:

```
cvtColor( old_image, new_image, CV_BGR2GRAY );
//convert old_image to gray scale image and save to new_image.
```

- threshold(): Applies a fixed-level threshold to each array element..
  Example:

  ```
  threshold( gray_image, bw_image, 100.0, 255.0, cv::THRESH_BINARY);
  // threshold the gray scale image gray_image and save the resulting
  // binary image to bw_image
  ```

- findContours(): Finds contours in a binary image.
- moments():Calculates all of the moments up to the third order of a polygon or rasterized shape.
- circle():Draws a circle

There are many functions that will help you edit the images in OpenCV, you can find the documentation of above functions and more at:

```
http://docs.opencv.org/2.4/
```

Or just google the task you want to do with the image, there are always related functions.

C.2 Introduction to Pointers

Instead of loading the pixels of our images into large variables, we will instead use pointers to reference the images. A pointer is a special variable that stores the *memory address* of another variable rather than the its value.

- & before a variable name references the address of a variable.
- * before a variable name in a declaration defines the variable as a pointer.
- * before a pointer name in a statement references the value at the address stored in the pointer.

Consider the following segment of code.

```
1  void main()
2  {
3     float r = 5.0;
4     float pi = 3.14159;
5     float *ptr;
6     // define a pointer to a floating-point number
7
8     ptr = &pi;
9     console_printf("The address in memory of
10                    variable pi is %i", ptr);
11    console_printf("The circumference of the
12                    circle is %f", 2*(*ptr)*rad);
13 }
```

The output from this segment of code would be as follows.

```
The address in memory of variable pi is 2287388
The circumference of the circle is 31.415901
```

The * before the variable name in Line 5 declares `ptr` to be a pointer. Line 7 uses the & operator to assign `ptr` the address in memory of the variable `pi`. Notice that `pi` is a floating-point number and `ptr` was declared as a pointer to a floating-point number. From the output of Line 8 we see that `ptr` contains only the address of the variable `pi`, not the value of the variable. To make use of the value of `pi`, we employ the * operator, as shown in the output of Line 9.

C.3 Simplifying Camera Calibration

In lab 6, we need to calibrate the camera. That is, we want to derive formulas to compute an object's coordinates in the world frame $(x, y, z)^w$ based on row and column coordinates in the image $(x, y)^c$. The necessary math is found in chapter 11 of SH&V. For this lab, however, we may make a few assumptions that will greatly simplify the calculations. We begin with the following intrinsic and extrinsic equations:

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

$$\text{extrinsic: } p^c = R^c_w p^w + O^c_w$$

with the variables defined as in Table C.2.

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).

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)	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.

We also note that the row value is associated with the x^c coordinate, and the column value is associated with the y^c coodinate. 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. *This is different than the description and figure in the text!* (In order to match the text, we would need row or column to increment in a reverse direction) As a consequence, we define $R_w^c = R_{z,\theta}$ instead of $R_{z,\theta} R_{x,180^\circ}$.

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

$$\begin{aligned} R_w^c &= R_{z,\theta} \\ &= \begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned}$$

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

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

The intrinsic and extrinsic equations are now

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

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

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.

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