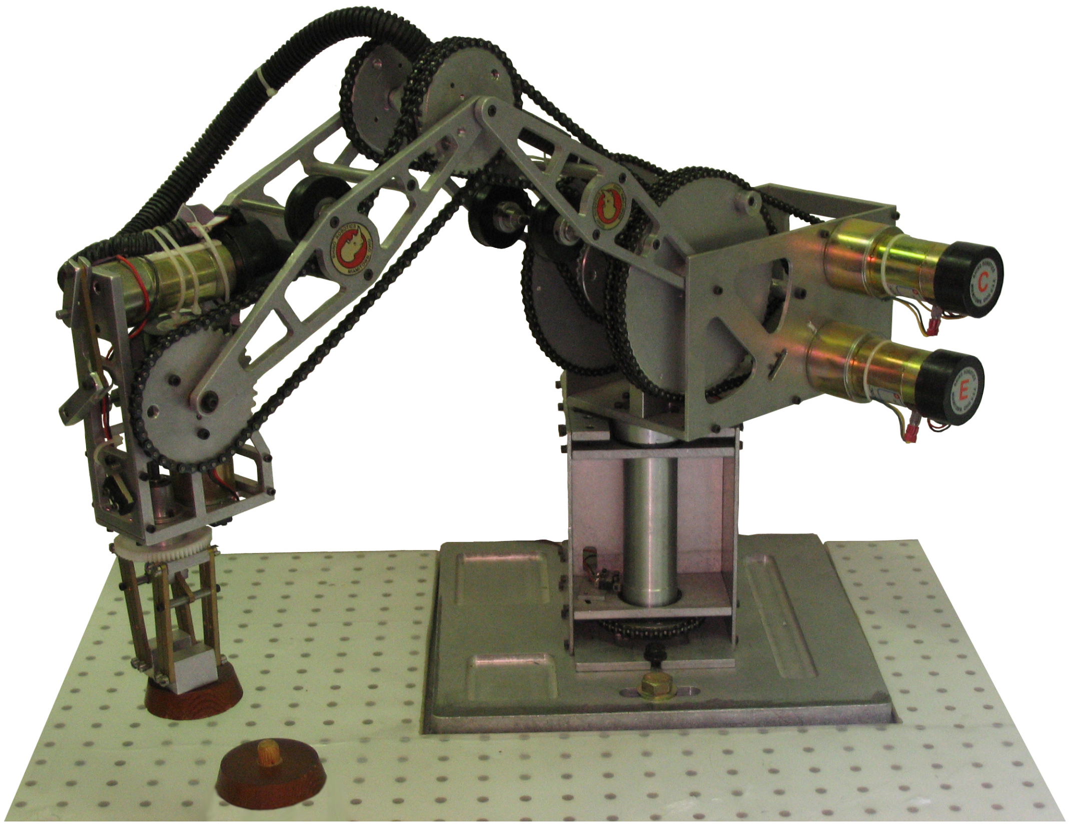
ECE 470
Introduction to Robotics
Lab Manual



Jonathan K. Holm
Jifei Xu
Yinai Fan

University of Illinois at Urbana-Champaign

Fall 2016

Acknowledgements

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 Robot Operating System (ROS) and serve as the foundation for more advanced courses on robot dynamics and control and computer vision. The course is cross-listed in four departments (Computer Science, Electrical & Computer Engineering, Industrial & Enterprise Systems 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 Rhino

1.1 Objectives

The purpose of this lab is to familiarize you with the Rhino robot arm, the hard home and soft home configurations, the use of the teach pendant, and the function of encoders. In this lab, you will:

- move the Rhino using the teach pendant
- send the Rhino to the hard home and soft home configurations
- store sequences of encoder counts as "programs"
- demonstrate at sequence of motions that, at minimum, places one block on top of another.

1.2 References

- Use of the teach pendant: Rhino Owner's Manual chapters 3 and 4.
- How to edit a motion program: Rhino Owner's Manual chapter 5.

1.3 Task

Using the teach pendant, each team will "program" the Rhino to pick and place blocks. The program may do whatever you want, but all programs must stack at least one block on top of another. Programs must begin and end in the hard home position.

1.4 Procedure

1. Turn on the Rhino controller (main power and motor power).
2. Put controller in teach pendant mode.
3. Experiment with the arm, using the teach pendant to move the motors that drive each axis of the robot.
    - Observe that the teach pendant will display a number for each motor you move. These numbers correspond to encoder measurements of the angle of each motor axis. By default, the teach pendant will move each motor 10 encoder steps at a time. You can refine these motions to 1 encoder step at a time by pressing `SHIFT + Slow` on the teach pendant.
4. `SHIFT + Go Hard Home`: moves the arm to a reference configurations based on the physical position of the motor axes and resets all encoders to zero.
5. `LEARN`: Enter learn mode on the teach pendant.
6. Program a sequence of motions: move a motor, `ENTER`, move another motor, `ENTER`, ...
    - Beware storing multi-axis motions as a single "step" in your program. The Rhino may not follow the same order of motor movements when reproducing your step. Break up dense multi-axis motions (especially when maneuvering near or around an obstacle) into smaller, less-dense steps.
    - Store gripper open/close motions as separate steps.
7. The final motion should be: `Go Soft Home, ENTER`. The soft home position simply seeks to return all encoders to zero counts.
    - If nothing has disturbed your Rhino, `Go Soft Home` should result in virtually the same configuration as `Go Hard Home`. *The Rhino will not allow you to use `Go Hard Home` as a "step" in your motion program.*

- If your robot has struck an obstacle, the encoder counts will no longer accurately reflect the arm's position with respect to the hard home position. If you use soft home to return the robot to zero encoder counts, it will be a different configuration than hard home. In such an instance, you will need to `Go Hard Home` to recalibrate the encoders.

8. `END/PLAY`: enter "play" mode.

9. `RUN`: executes the sequence of motions you have stored in the teach pendant's memory.

1.5 Report

None required. Finish pre-lab for lab 2.

1.6 Demo

Show your TA the sequence of motions your team has programmed. Remember, your program must successfully stack at least one block on another.

1.7 Grading

Grades will be pass/fail, based entirely on the demo.

LAB 2

The Tower of Hanoi

2.1 Objectives

This lab is an introduction to controlling the Rhino robots using the cpp programming language. In this lab, you will:

- record encoder counts for various configurations of the robot arm
- using prewritten cpp functions, move the robot to configurations based on encoder counts
- order a series of configurations that will solve the Tower of Hanoi problem.

2.2 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.

2.3 References

- Consult Appendix B of this lab manual for details of ROS and cpp functions used to control the Rhino.
- "*A Gentle Introduction to ROS*", Chapter 2 and 3.

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

- 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.)

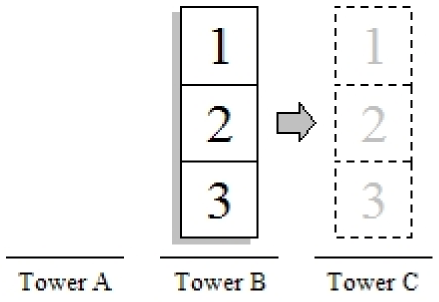


Figure 2.1: Example start and finish tower locations.

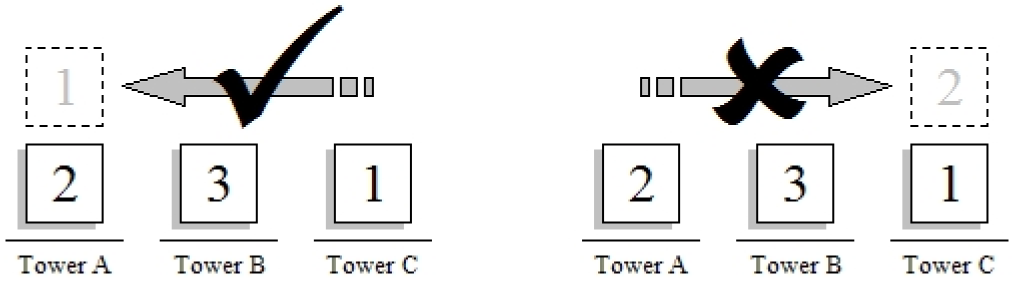


Figure 2.2: Examples of a legal and an illegal move.

2.4 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.

2.5 Procedure

1. Creat your own workspace as shown in Appendix B.
2. Download and extract `lab2.zip` from the course website and extract into you workspace /src folder. 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.
   - `CMakeLists.txt` a file that setup the necessary libraries and environment for compiling lab2.cpp.
   - `package.xml` This file defines properties about the package including package dependencies.
3. 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.
4. For each base, place the tower of blocks and use the teach pendant to find encoder values corresponding to the pegs of the top, middle, and bottom blocks. Record these encoder values for use in your program.
5. Write a cpp program that prompts the user for the start and destination tower locations (you may assume that the user will not choose the same location twice) and moves the blocks accordingly.

   Note: the "Mode" switch on the Rhino controller should be pointed to "Computer" before you run your ROS node `rosrun lab2 lab2`.

2.6 Report

No report is required. You must submit a hardcopy of your `lab2.cpp` file with a coversheet containing:

- your names
- "Lab 2"
- the weekday and time your lab section meets (for example, "Monday, 1pm").

2.7 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.

2.8 Grading

Grades are out of 2. Each successful demo will be graded pass/fail with a possible score of 1.

LAB 3

Forward Kinematics

3.1 Objectives

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

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

From now on, labwork is closely tied to each arm's differences.
Pick a robot and stick with it for the remaining labs.

3.2 References

- Chapter 3 of SH&V provides details of the DH convention and its use in parameterizing robots and computing the forward kinematic equations.
- A matlab version code for translating DH parameters to forward HT matrix is available online, ”Denavit Hartenberg Parameters” by Mahmoud KhoshGoftar.

(https://www.mathworks.com/matlabcentral/fileexchange/44585-denavit-hartenberg-parameters)

- The complete Robotica 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.

3.3 Tasks

3.3.1 Physical Implementation

The user will provide five joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$, all given in degrees. Angles $\theta_1, \theta_2, \theta_3$ will be given between $-180°$ and $180°$, angle θ_4 will be given between $-90°$ and $270°$, and angle θ_5 is unconstrained. The goal is to translate the desired joint angles into the corresponding encoder counts for each of the five joint motors. We need to write five mathematical expressions

$$\begin{aligned} enc_B(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) &= ? \\ enc_C(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) &= ? \\ enc_D(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) &= ? \\ enc_E(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) &= ? \\ enc_F(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) &= ? \end{aligned}$$

and translate them into cpp code (note that we do not need an expression for encoder A, the encoder for the gripper motor, the ros message command starts with encoder B). Once the encoder values have been found, we will command the Rhino to move to the corresponding configuration.

3.3.2 Theoretical Solution

Find the forward kinematic equations for the Rhino robot. In particular, we are interested only in the position d_5^0 of the gripper and will ignore the orientation R_5^0. We will use Robotica to find expressions for each of the three components of d_5^0.

3.3.3 Comparison

For any provided set of joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$, we want to compare the position of the gripper after your cpp function has run to the position of the gripper predicted by the forward kinematic equations.

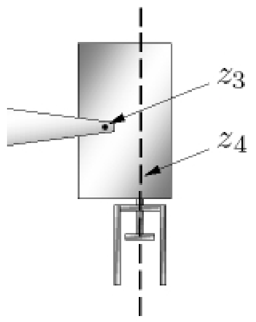


Figure 3.1: Wrist z axes do not intersect.

3.4 Procedure

3.4.1 Physical Implementation

1. Download and extract `lab3.zip` from the course website. Inside this package are a number of files to help complete the lab, very similar to the one provided for lab 2.

2. Before we proceed, we must define coordinate frames so each of the joint angles make sense. For the sake of the TA's sanity when helping students in the lab, DH frames have already been assigned in Figure 3.3 (on the last page of this lab assignment). On the figure of the Rhino, clearly label the joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$, being careful that the sense of rotation of each angle is correct.

   Notice that the z_3 and z_4 axes do not intersect at the wrist, as shown in Figure 3.1. The offset between z_3 and z_4 requires the DH frames at the wrist to be assigned in an unexpected way, as shown in Figure 3.3. Consequently, the zero configuration for the wrist is not what we would expect: *when the wrist angle $\theta_4 = 0°$, the wrist will form a right angle with the arm.* Please study Figure 3.3 carefully.

3. Use the rulers provided to measure all the link lengths of the Rhino. Try to make all measurements accurate to at least the nearest half centimeter. Label these lengths on Figure 3.3.

4. Now measure the ratio $\frac{encoder\ steps}{joint\ angle}$ for each joint. Use the teach pendant to sweep each joint through a 90° angle and record the starting and ending encoder values for the corresponding motor. *Be careful*

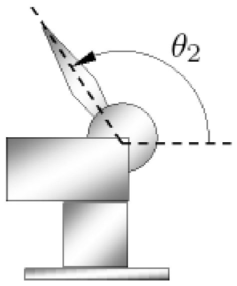


Figure 3.2: With the Rhino in the hard home position encoder D is zero while joint angle θ_2 is nonzero.

> *that the sign of each ratio corresponds to the sense of rotation of each joint angle.*

For example, in order to measure the shoulder ratio, consider following these steps:

- Adjust motor D until the upper arm link is vertical. Record the value of encoder D at this position.
- Adjust motor D until the upper arm link is horizontal. Record the value of encoder D at this position.
- Refer to the figure of the Rhino robot and determine whether angle θ_2 swept $+90^\circ$ or -90°.
- Compute the ratio $ratio_{D/2} = \frac{enc_D(1) - enc_D(0)}{\theta_2(1) - \theta_2(0)}$.

We are almost ready to write an expression for the motor D encoder, but one thing remains to be measured. Recall that all encoders are set to 0 when the Rhino is in the hard home configuration. However, *in the hard home position not all joint angles are zero*, as illustrated in Figure 3.2. It is tempting to write $enc_D(\theta_2) = ratio_{D/2}\theta_2$ but it is easy to see that this expression is incorrect. If we were to specify the joint angle $\theta_2 = 0$, the expression would return $enc_D = 0$. Unfortunately, setting encoder D to zero will put the upper arm in its hard home position. Look back at the figure of the Rhino with the DH frames attached. When $\theta_2 = 0$ we want the upper arm to be horizontal. We must account for the angular offset at hardhome.

5. Use the provided protractors to measure the joint angles when the Rhino is in the hard home position. We will call these the joint offsets and identify them as θ_{i0}. Now we are prepared to write an expression for the motor D encoder in the following form:

$$enc_D(\theta_2) = ratio_{D/2}(\theta_2 - \theta_{20}) = ratio_{D/2}\Delta\theta_2.$$

   Now, if we were to specify $\theta_2 = 0$, encoder D will be set to a value that will move the upper arm to the horizontal position, which agrees with our choice of DH frames.

6. Derive expressions for the remaining encoders. You will quickly notice that this is complicated by the fact that the elbow and wrist motors are located on the shoulder link. Due to transmission across the shoulder joint, the joint angle at the elbow is affected by the elbow motor *and the shoulder motor*. Similarly, the joint angle at the wrist is affected by the wrist, elbow, and shoulder motors. Consequently, the expressions for the elbow and wrist encoders will be functions of more than one joint angle.

   It is helpful to notice the trasmission gears mounted on the shoulder and elbow joints. These gears transmit the motion of the elbow and wrist motors to their respective joints. Notice that the diameter of the transmission gears is the same. This implies that the change in the shoulder joint angle causes a change in the elbow and wrist joint angles of equal magnitude. Mathematically, this means that θ_3 is equal to the change in the shoulder angle added or subtracted from the elbow angle. That is,

$$\Delta\theta_3 = ((\theta_3 - \theta_{30}) \pm (\theta_2 - \theta_{20})).$$

   A similar expression holds for $\Delta\theta_4$. It is up to you to determine the $\pm$ sign in these expressions.

3.4.2 Theoretical Solution

1. Construct a table of DH parameters for the Rhino robot. Use the DH frames already established in Figure 3.3.

2. Write a Robotica source file containing the Rhino DH parameters. Consult Appendix A of this lab manual for assistance with Robotica.

3. Write a Mathematica file that finds and properly displays the forward kinematic equation for the Rhino. Consult Appendix A of this lab manual for assistance with Mathematica.

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

3.4.3 Comparison

1. The TA will supply two sets of joint angles $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$ for the demonstration. Record these values for later analysis.

2. Run the executable file and move the Rhino to each of these configurations. 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. In Mathematica OR Matlab, specify values for the joint variables that correspond to both sets of angles used for the Rhino. Note the vector $d_5^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5)$ for each set of angles. Call these 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 two forward 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 $error_2$. Because the forward kinematics code will be used to complete labs 4 and 6, we want the error to be as small as possible. Tweak your code until the error is minimized.

3.5 Report

Assemble the following items in the order shown.

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

2. A figure of the Rhino robot with DH frames assigned, all joint variables and link lengths shown, and a complete table of DH parameters.
3. A clean derivation of the expressions for each encoder. Please sketch figures where helpful and draw boxes around the final expressions.
4. The forward kinematic equation for *the tool position only* of the Rhino robot. Robotica and the matlab "DH parameters" will generate the entire homogenous transformation between the base and tool frames

   $$T_5^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = \begin{bmatrix} R_5^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) & d_5^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) \\ 0\ 0\ 0 & 1 \end{bmatrix}$$

   but you only need to show the position of the tool frame with respect to the base frame

   $$d_5^0(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = [\textit{vector expression}].$$

5. For each of the two sets of joint variables you tested, provide the following:
   - the set of values, $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$
   - 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, $error_i$.
6. A brief paragraph (2-4 sentences) explaining the sources of error and how one might go about reducing the error.

3.6 Demo

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

3.7 Grading

Grades are out of 3. Each successful demo will be graded pass/fail with a possible score of 1. The remaining point will come from the lab report.

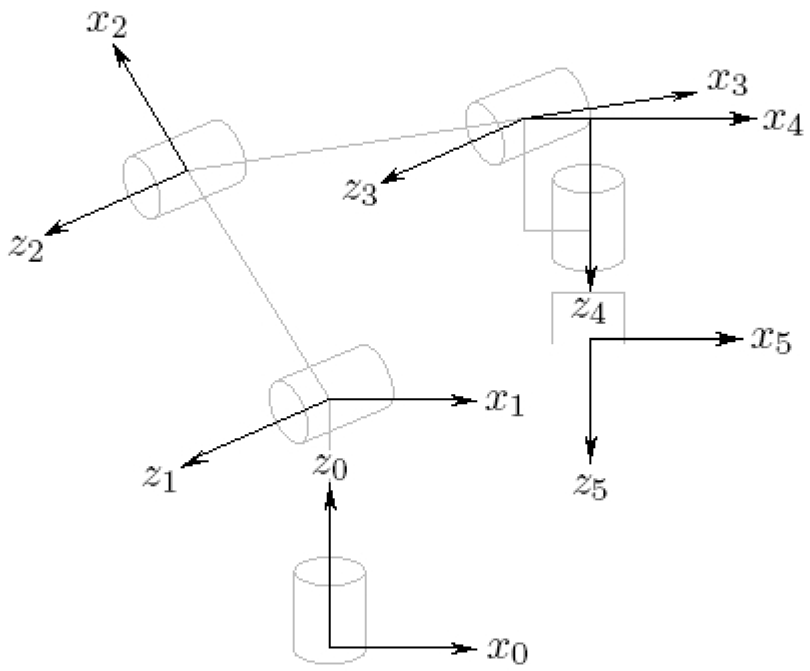


Figure 3.3: Rhino robot with DH frames assigned.

LAB 4

Inverse Kinematics

4.1 Objectives

The purpose of this lab is to derive and implement a solution to the inverse kinematics problem for the Rhino robot, a five degree of freedom (DOF) arm without a spherical wrist. In this lab we will:

- derive the elbow-up inverse kinematic equations for the Rhino
- write a cpp function that moves the Rhino to a point in space specified by the user.

4.2 Reference

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

4.3 Tasks

4.3.1 Solution Derivation

Given a desired point in space (x, y, z) and orientation $\{\theta_{pitch}, \theta_{roll}\}$, write five mathematical expressions that yield values for each of the joint variables. For the Rhino robot, there are (in general) four solutions to the inverse kinematics problem. We will implement only one of the *elbow-up* solutions.

- In the inverse kinematics problems we 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. Next we would solve the inverse position problem for the first three joint variables.

  Unfortunately, the Rhino robots in our lab have only 5 DOF and no spherical wrists. Since all possible positions and orientations require a manipulator with 6 DOF, our robots cannot achieve all possible orientations in their workspaces. To make matters more complicated, the axes of the Rhino's wrist do not intersect, as do the axes in the spherical wrist. So we do not have the tidy spherical wrist inverse kinematics as we have studied in class. We will solve for the joint variables in an order that may not be immediately obvious, but is a consequence of the degrees of freedom and wrist construction of the Rhino.

- We will solve the inverse kinematics problem in the following order:

  1. θ_5, which is dependent on the desired orientation only
  2. θ_1, which is dependent on the desired position only
  3. the wrist center point, which is dependent on the desired position and orientation and the waist angle θ_1
  4. θ_2 and θ_3, which are dependent on the wrist center point
  5. θ_4, which is dependent on the desired orientation and arm angles θ_2 and θ_3.

- The orientation problem is simplified in the following way: instead of supplying an arbitrary rotation matrix defining the desired orientation, the user will specify θ_{pitch} and θ_{roll}, the desired wrist pitch and roll angles. Angle θ_{pitch} will be measured with respect to the z_w, the axis normal to the surface of the table, as shown in Figure 4.1. The pitch angle will obey the following rules:

  1. $-90^\circ < \theta_{pitch} < 270^\circ$
  2. $\theta_{pitch} = 0^\circ$ corresponds to the gripper pointed down
  3. $\theta_{pitch} = 180^\circ$ corresponds to the gripper pointed up
  4. $0^\circ < \theta_{pitch} < 180^\circ$ corresponds to the gripper pointed away from the base
  5. $\theta_{pitch} < 0^\circ$ and $\theta_{pitch} > 180^\circ$ corresponds to the gripper pointed toward the base.

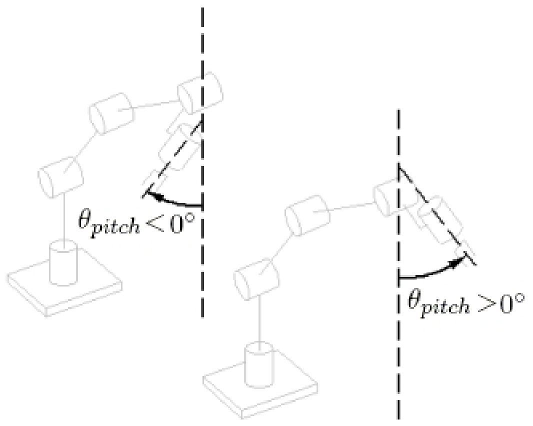


Figure 4.1: θ_{pitch} is the angle of the gripper measured from the axis normal to the table.

4.3.2 Implementation

Implement the inverse kinematics solution by writing a cpp function to receive world frame coordinates (x_w, y_w, z_w), compute the desired joint variables $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$, and command the Rhino to move to that configuration using the `lab3.cpp` function written for the previous lab.

4.4 Procedure

1. Download and extract `lab4.zip` from the course website. Inside this package are a number of files to help complete the lab. You will also need to copy functions implemented in `lab3.cpp` file which you wrote for the previous lab to current lab4.cpp.

2. Establish the world coordinate frame (frame w) centered at the corner of the Rhino's base shown in Figure 4.2. The x_w and y_w plane should correspond to the surface of the table, with the x_w axis parallel to the sides of the table and the y_w axis parallel to the front and back edges of the table. Axis z_w should be normal to the table surface, with up being the positive z_w direction and the surface of the table corresponding to $z_w = 0$.

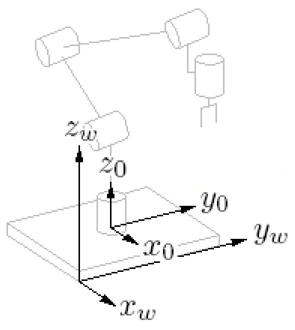


Figure 4.2: Correct location and orientation of the world frame.

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_w, y_w, z_w) in the world frame to coordinates (x_0, y_0, z_0) in the base frame of the Rhino.

$$x_0(x_w, y_w, z_w) =$$
$$y_0(x_w, y_w, z_w) =$$
$$z_0(x_w, y_w, z_w) =$$

Be careful to reference the location of frame 0 as your team defined it in lab 3, as we will be using the functions copied from `lab3.cpp` file you created.

3. The wrist roll angle θ_{roll} has no bearing on our inverse kinematics solution, therefore we can immediately write our first equation:

$$\theta_5 = \theta_{roll}. \tag{4.1}$$

4. Given the desired position of the gripper (x_0, y_0, z_0) (in the base frame), write an expression for the waist angle of the robot. It will be helpful to project the arm onto the $x_0 - y_0$ plane.

$$\theta_1(x_0, y_0, z_0) = \tag{4.2}$$

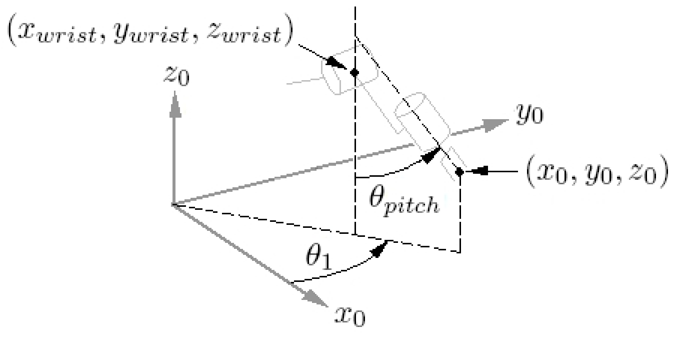


Figure 4.3: Diagram of the wrist showing the relationship between the wrist center point $(x_{wrist}, y_{wrist}, z_{wrist})$ and the desired tool position (x_0, y_0, z_0) (in the base frame). Notice that the wrist center and the tip of the tool are not both on the line defined by θ_{pitch}.

5. Given the desired position of the gripper $(x, y, z)^0$ (in the base frame), desired pitch of the wrist θ_{pitch}, and the waist angle θ_1, solve for the coordinates of the wrist center. Figure 4.3 illustrates the geometry at the wrist that involves all five of these parameters.

$$x_{wrist}(x_0, y_0, z_0, \theta_{pitch}, \theta_1) =$$
$$y_{wrist}(x_0, y_0, z_0, \theta_{pitch}, \theta_1) =$$
$$z_{wrist}(x_0, y_0, z_0, \theta_{pitch}, \theta_1) =$$

   Remember to account for the offset between wrist axes z_3 and z_4, as shown in Figure 4.3.

6. Write expressions for θ_2 and θ_3 in terms of the wrist center position

$$\theta_2(x_{wrist}, y_{wrist}, z_{wrist}) = \tag{4.3}$$
$$\theta_3(x_{wrist}, y_{wrist}, z_{wrist}) = \tag{4.4}$$

   as we have done in class and numerous times in homework.

7. Only one joint variable remains to be defined, θ_4. *Note:* $\theta_4 \neq \theta_{pitch}$ (see if you can convince yourself why this is true). Indeed, θ_4 is a function of θ_2, θ_3 and θ_{pitch}. Again, we must take into consideration the offset between the wrist axes, as shown in Figure 4.3.

$$\theta_4(\theta_2, \theta_3, \theta_{pitch}) = \tag{4.5}$$

8. Now that we have expressions for all the joint variables, enter these formulas into your `lab4.cpp` file. The last line of your file should call the `lab_angles` function you wrote for lab 3, directing the Rhino to move to the joint variable values you just found.

4.5 Report

Assemble the following items in the order shown.

1. Coversheet containing your 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 $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$. *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 $\{(x_w, y_w, z_w), \theta_{pitch}, \theta_{roll}\}$
   - 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.

4.6 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 Rhino to reach a point in its workspace off the table. The second demo will require the Rhino to reach a configuration above a block on the table with sufficient accuracy to pick up the block.

4.7 Grading

Grades are out of 3. Each successful demo will be graded pass/fail with a possible score of 1. The remaining point will come from the lab report.

LAB 5

Image Processing

5.1 Objectives

This is the first of two labs whose purpose is to integrate computer vision and control of the Rhino 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.

5.2 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.7 before beginning the lab.

5.3 Tasks

5.3.1 Separating Objects from Background

The images provided by the camera are 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 ($z = 0$).

5.3.2 Associating Objects in the Image

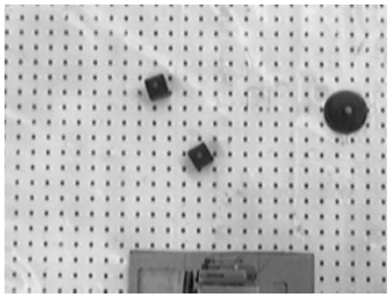
Once objects in the image have been separated from the background, we want to indentify the separate objects in the image. We will distinguish among the objects by assigning each object a unique color. The pegboards on the workspace will introduce many small "dots" to the image that will be misinterpreted 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.

5.4 Procedure

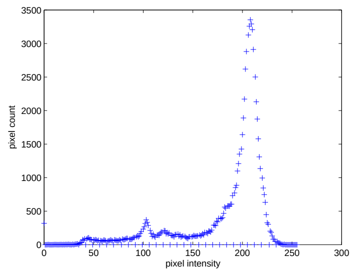
5.4.1 Separating Objects from Background

1. Read section 11.3 in SH&V and sections C.1 through C.7 in this lab manual before proceeding further.

2. Download and extract `visionlabs.zip` from the course website. Inside this package are the files that handle the interface with the camera and the rhino and generate the image console. You will be editing the file `vision_labs.cpp` for both labs 5 and 6.



(a)



(b)

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

3. 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.

4. 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.
   - 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.

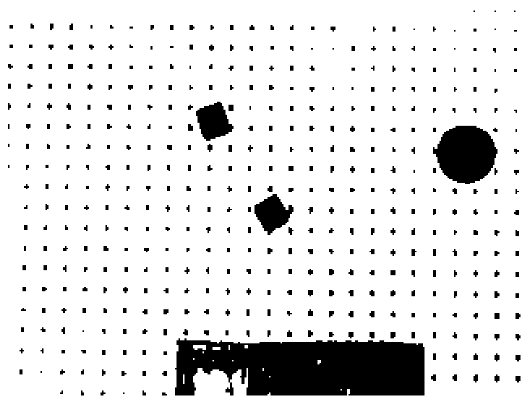


Figure 5.2: Sample image after thresholding.

5. 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.

5.4.2 Associating Objects in the Image

1. Read section 11.4 in SH&V and sections C.1 through C.7 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.

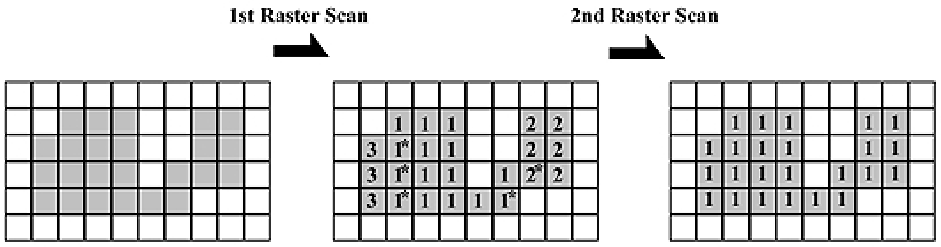


Figure 5.3: Pixels in the image after thresholding, after the first raster scan, and after the second raster scan. In the first image, there are only black and white pixels; no labels have been assigned. After the first raster scan, we can see the labels on the object pixels; an equivalence is noted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

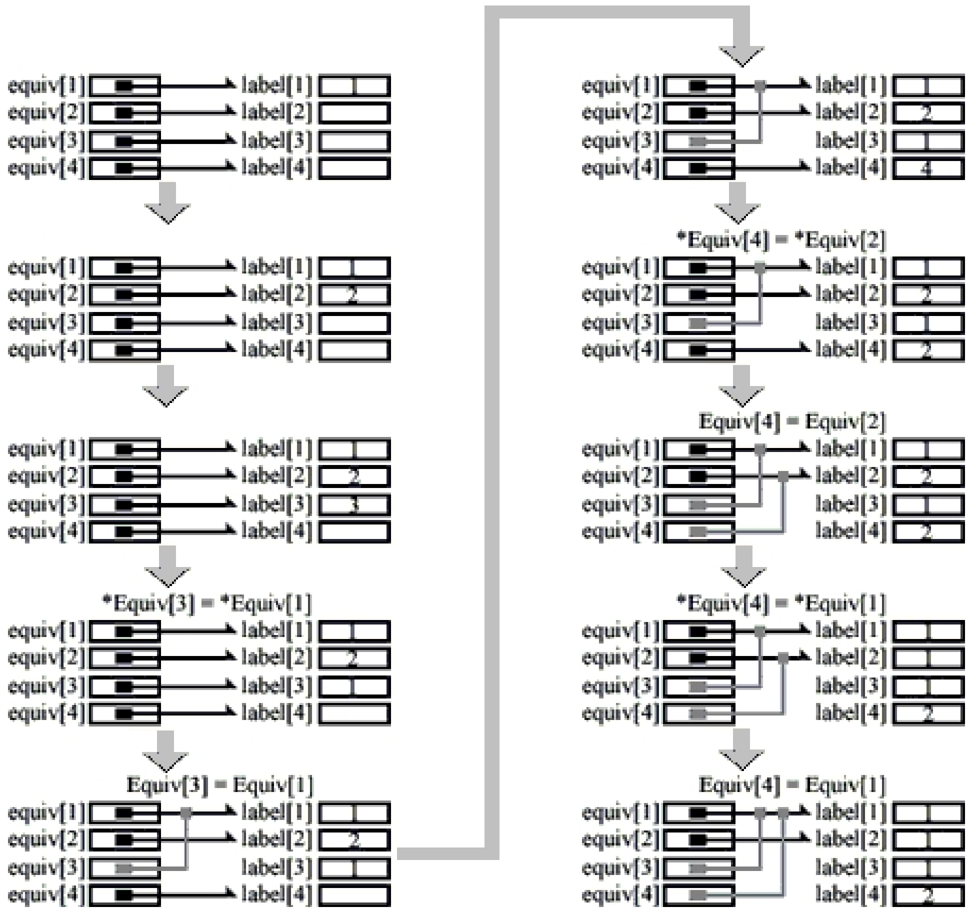


Figure 5.4: Evolution of the pixel labels as equivalences are encountered.

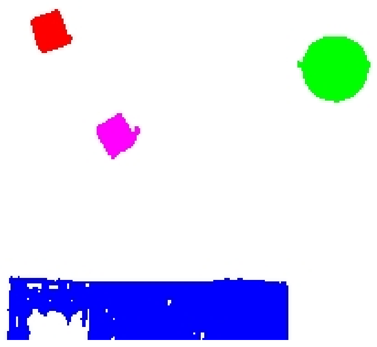


Figure 5.5: Sample image after object association.

5.5 Report

No report is required for this lab. You must submit your `vision_labs.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."

5.6 Demo

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

5.7 Grading

Grades are out of 3, based on the TA's evaluation of your demo.

LAB 6

Camera Calibration

6.1 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
- *Bonus*: 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.

6.2 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 all of section C.8 before beginning the lab.

6.3 Tasks

6.3.1 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.

6.3.2 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.8 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.

6.3.3 *Bonus*: 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 code. Your lab 4 code computes the necessary joint angles and calls your lab 3 code to move the Rhino to that configuration.

We will be working with blocks that have wooden pegs passing through their centers. From the camera's perspective, the centroid of a block object corresponds to the coordinates of the peg. You will bring together your lab

3-5 code and the prewritten point-and-click functions in the following way: the user will click on a block in the image console, your code will command the Rhino to move to a configuration above the peg for that block, you will command the Rhino to grip the block and then return to the home position with the block in its grip, the user will click on an unoccupied portion of the image, and the Rhino will move to the corresponding region of the table and release the block.

6.4 Procedure

6.4.1 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.

6.4.2 Camera Calibration

1. Read sections 11.1 and 11.2 in SH&V and section C.8 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 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.8 of this lab manual. Write equations for the world frame coordinates in terms of the image coordinates.

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

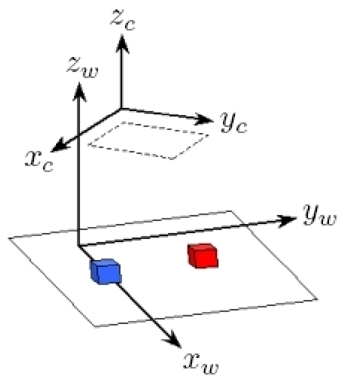


Figure 6.1: Arrangement of the world and camera frames.

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.

$$\beta =$$

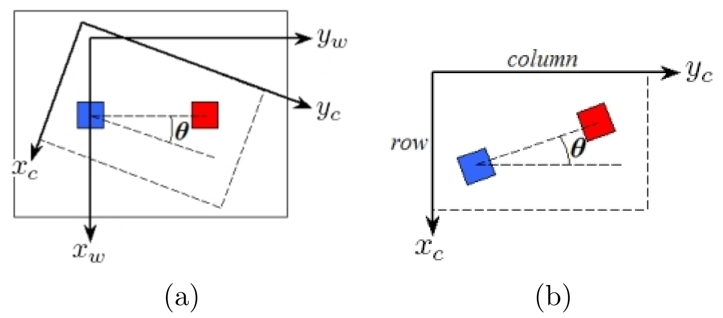


Figure 6.2: (a) Overhead view of the table; the rectangle delineated by dashed lines represents the camera's field of vision. (b) The image seen by the camera, showing the diagonal line formed by the blocks. Notice that x_c increases with increasing row values, and y_c increases with increasing column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

5. θ is the angle of rotation between the world frame and the camera frame. Please refer to section C.8 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 into your `vision_labs.cpp` file. 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).

6.4.3 *Bonus*: Pick and Place

Near the bottom of `vision_labs.cpp` are two functions: lab_pick and lab_place. Typing "pick" in the console command line will prompt the user to click on a block using the mouse. When the user clicks inside an image, the row and column position of the mouse is passed into the lab_pick function. The *color* of the pixel that the mouse is pointing to is passed to the function. Similarly, typing "place" in the command line will pass row, column, and color information to the lab_place function.

1. Build a data structure that holds the centroid location (x_w, y_w) and unique color for each object in the image.

2. Inside lab_pick, reference the data structure to identify the centroid of the object the user has selected based on the color information provided. Call the function you wrote for lab 4 to move the robot into gripping position over the object. You may assume that the only objects the user will select will be the blocks we have used in the lab. Use a z_w value that will allow the Rhino to grasp the peg of a block.

3. Grip the block and return to the softhome position.

4. Inside lab_place, compute the (x_w, y_w) location of the row and column selected by the user. Again, call lab_movex to move the Rhino to this position.

5. Release the block and return to softhome.

6.5 Report

No report is required for this lab. You must submit your `vision_labs.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."

6.6 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. If you are pursuing the bonus task, you will also demonstrate that the robot will retrieve a block after you click on it in the image. The robot must also place the block in another location when you click on a separate position in the image.

6.7 Grading

Grades are out of 3, based on the TA's evaluation of your demo, divided as follows.

- 1 point for crosshairs drawn over the centroids of each object in the image
- 2 points for correctly reporting the world frame coordinates of the centroid of each object
- 1 bonus point for successfully picking up and placing a block.

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.

- 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 Writing a Robotica Source File

- Robotica takes as input the Denavit-Hartenberg parameters of a robot. Put DH frames on a figure of your robot and compute the table of DH paramters. Open a text editor (notepad in Windows will be fine) and enter the DH table. You will need to use the following form:

```
DOF=2
  (Robotica requires a line between DOF and joint1)
joint1 = revolute
a1     = 3.5
alpha1 = Pi/2
d1     = 4
theta1 = q1
joint2 = prismatic
a2     = 0
alpha2 = 90 Degree
d2     = q2
theta2 = Pi
```

- You may save your file with any extension, but you may find '.txt' to be useful, since it will be easier for your text editor to recognize the file in the future.
- Change the degrees of freedom line ('`DOF=`') to reflect the number of joints in your robot.
- Using joint variables with a single letter followed by a number (like 'q1' and 'q2') will work well with the command that simplifies trigonometric notation, which we'll see momentarily.

A.3 Robotica Basics

- Download 'robotica.m' from the website. Find the root directory for Mathematica on your computer and save the file in the `/AddOns/ExtraPackages` directory.
- Open a new notebook in Mathematica and load the Robotica package by entering

```
<< robotica.m
```

- Load your robot source file. This can be done in two ways. You can enter the full path of your source file as an argument in the `DataFile` command:

```
DataFile["C:\\directory1\directory2\robot1.txt"]
```

  Notice the double-backslash after `C:` and single-backslashes elsewhere in the path. You can also enter `DataFile[]` with no parameter. In this case, Mathematica will produce a window that asks you to supply the full path of your source file. You do not need double-backslashes when you enter the path in the window. *You will likely see a message warning you that no dynamics data was found.* Don't worry about this message; dynamics is beyond the scope of this course.

- Compute the forward kinematics for your robot by entering `FKin[]`, which generates the A matrices for each joint, all possible T matrices, and the Jacobian.
- To view one of the matrices generated by forward kinematics, simply use the `MatrixForm` command mentioned in the first section. For example, `MatrixForm[A[1]]` will display the homogeneous tranformation `A[1]`.

A.4 Simplifying and Displaying Large, Complicated Matrices

- Mathematica has a powerful function that can apply trigonometric identities to complex expressions of sines and cosines. Use the `Simplify` function to reduce a complicated matrix to a simpler one.

Be aware that `Simplify` *may take a long time to run.* It is not unusual to wait 30 seconds for the function to finish. For example, `T=Simplify[T]` will try a host of simplification algorithms and redefine `T` in a simpler equivalent form. You may be able to view all of the newly simplified matrix with `MatrixForm`.

- Typically, matrices generated by forward kinematics will be littered with sines and cosines. Entering the command `SimplifyTrigNotatation[]` will replace `Cos[q1]` with $\mathtt{c}_1$, `Sin[q1+q2]` with $\mathtt{s}_{12}$, etc. when your matrices are displayed. Executing `SimplifyTrigNotation` will not change previous output. However, all following displays will have more compact notation.

- If your matrix is still too large to view on one screen when you use `MatrixForm`, the `EPrint` command will display each entry in the matrix one at a time. The EPrint command needs two parameters. The first is the name of the matrix to be displayed, the second is the label used to display alongside each entry. For example, entering `EPrint[T,"T"]` will display all sixteen entries of the homogeneous transformation matrix `T` individually.

A.5 Example

Consider the three-link revolute manipulator shown in Figure A.1. The figure shows the DH frames with the joint variables $\theta_1, \theta_2, \theta_3$ and parameters a_1, a_2, a_3 clearly labeled. The corresponding table of DH parameters is given in Table A.1.

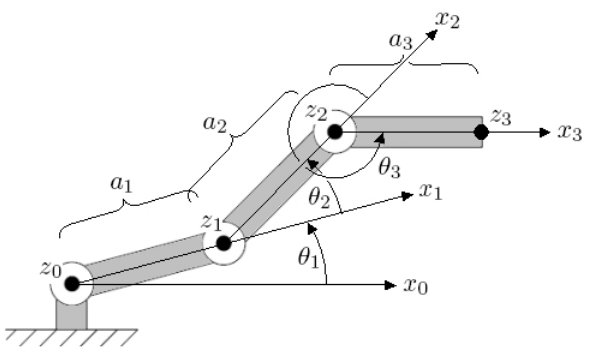


Figure A.1: Three-link revolute manipulator with DH frames shown and parameters labeled. The z axes of the DH frames are pointing out of the page.

joint	a	α	d	θ
1	a_1	0	0	θ_1
2	a_2	0	0	θ_2
3	a_3	0	0	θ_3

Table A.1: Table of DH parameters corresponding to the frames assigned in Figure A.1.

We open a text editor and create a text file with the following contents.

```
DOF=3

The Denavit-Hartenberg table:
joint1 = revolute
a1     = a1
alpha1 = 0
d1     = 0
theta1 = q1
joint2 = revolute
a2     = a2
alpha2 = 0
d2     = 0
theta2 = q2
joint3 = revolute
a3     = a3
alpha3 = 0
d3     = 0
theta3 = q3
```

We save the file as `c:`
`example.txt` . After loading Mathematica, we enter the following commands.

```
<< robotica.m
DataFile["C:\\example.txt"]
```

If all goes well, the Robotica package will be loaded successfully, and the example datafile will be opened and its contents displayed in a table.

```
No dynamics data found.
Kinematics Input Data
---------------------
Joint   Type        a    alpha   d   theta
1       revolute    a1   0       0   q1
2       revolute    a2   0       0   q2
3       revolute    a3   0       0   q3
```

Now generate the forward kinematic matrices by entering the following command.

```
FKin[]
```

Robotica will generate several status lines as it generates each A and T matrix and the Jacobian. Now we will view one of these matrices. We decide to view the T_0^3 matrix. Entering

```
MatrixForm[T[0,3]]
```

generates the output

$$\begin{pmatrix} \text{Cos[q1 + q2 + q3]} & \text{-Sin[q1 + q2 + q3]} & 0 & \text{a1 Cos[q1] + ...} \\ \text{Sin[q1 + q2 + q3]} & \text{Cos[q1 + q2 + q3]} & 0 & \text{a1 Sin[q1] + ...} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

which is too wide to fit on this page. We wish to simplify the notation, so we enter

```
SimplifyTrigNotation[]
```

before displaying the matrix again. Now, entering the command

```
MatrixForm[T[0,3]]
```

generates the much more compact form

$$\begin{pmatrix} c_{123} & -s_{123} & 0 & a1c_1 + a2c_{12} + a3c_{123} \\ s_{123} & c_{123} & 0 & a1s_1 + a2s_{12} + a3s_{123} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

A.6 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. Also, remember to simplify the matrices as much as possible using the techniques we have presented in section A.4 of this Appendix.

- Mathematically simplify the matrix using `Simplify`.
- Simplify the notation using `SimplifyTrigNotation`. Remember that all mathematical simplification must be completed *before* running `SimplifyTrigNotation`.
- If the matrix is still too large to view on the screen when you use `MatrixForm`, use the `EPrint` command to display the matrix one entry at a time.

Appendix B

C Programming in ROS

List: how to make directory, where to download, how to unzip, how to source, how to compile, how to run launch files, how to run functions, What is the launch file doing , what is the run function doing.

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 the 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 rhino_ros rhino_start.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.

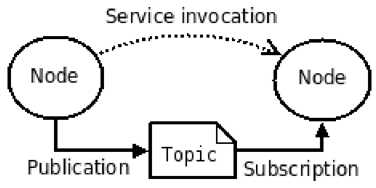


Figure B.1: source: http://wiki.ros.org/ROS/Concepts

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 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 'lab2' package, you can type in your console

  ```
  $ rospack find lab2
  ```

- To move directly to the directory of a ROS package, use roscd. For example, go to lab2 package directory

```
$ roscd lab2
```

- 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

It is recommended that you have your workspace created in your own usb drive to work with, so no other groups sharing the same workstation will be able to tamper with your codes. And have a backup copy in your laptop or cloud drive.

- First format your usb drive to be compatible with Linux(Ext4). (Ask TA or Google)
- Then go to your usb root directory following procedures below.

```
$ cd ~
$ cd /media/youbot/ #double press Tab to show available choices
<media_name_1> usb_dir_name #example output for double Tab
$ cd /media/youbot/usb_dir_name # press enter
```

- Let's create a catkin workspace. You can use a different name than catkin_ws if you'd like but it is recommended that you leave the word catkin in the directory name.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/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_ws/
$ 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:

  ```
  $ source devel/setup.bash
  ```

B.5 Running A Node

- Now, you can copy all the packages you want to the src directory. For example, for Lab 2, download lab2.tar.gz, serial.tar.gz, rhino_ros.tar.gz into your src directory. Then right click on each tar.gz file and extract them to the current directory. Finally, go to a terminal you have opened and make sure you are in the workspace root directory and run catkin_make to Cmake all the cpp files.

- After compilation is complete, we can start running our own nodes. For example our lab2 node. However, before running any nodes, we must have roscore running. This is taken care of by running a launch file.

  ```
  $ roslaunch rhino_ros rhino_start.launch
  ```

  This command runs both roscore and a ros-to-rhino serial communication process that acts as a subscriber waiting for a command message that controls the Rhino's motors via the serial interface.

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

- Run your node with the command rosrun in the new command prompt. Example of running lab2 node in lab2 package:

  ```
  $ rosrun lab2 lab2
  ```

B.6 Simple Publisher and Subscriber Tutorial

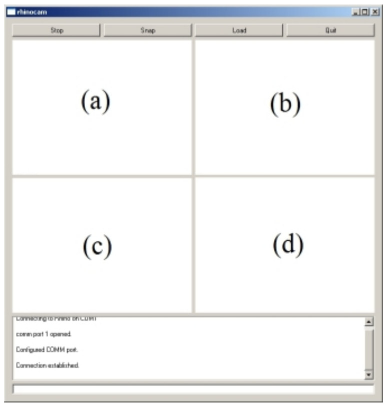
Please refer to the webpage:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c%2B%2B)

Appendix C

Notes on Computer Vision

C.1 Image Console

For the first four labs, we have used the text-only remote environment to interface with the Rhino. 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 a Qt-based graphical console that displays images and interfaces with the camera and the Rhino simultaneously. The layout of the console window is shown in Figure C.1.



```
Connecting to Rhino on COM1
comm port 1 opened.
Configured COMM port.
Connection established.
```

Figure C.1: Image processing console containing four image frames: (a) the active color image (b) the active grayscaled image (c) the thresholded image (d) the image with false objects discarded and remaining objects uniquely colored.

Across the top of the console window are four buttons:

- *Start/Stop* is used to begin or end realtime use of the camera, in which the four frames are constantly updated.
- *Snap* can be used while the camera is not in realtime mode to process the current image seen by the camera.
- *Load* is used to open an image that has been previously saved using Logitech software.
- *Quit* closes the console window.

Below the four image frames is a text window and a command line. Remote enivornment commands can be entered at the command line just as in previous labs. Ouput from your program will be displayed in the text window above the command line.

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 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 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 The QRgb/QRgba Data Type

We will associate four 8-bit integers (type `uchar`) with each pixel of an image, namely the red, green, and blue intensities (0-255) and the transparency value alpha. We lump these four values into a single 32-bit integer (type `uint`) called a QRgba, as illustrated in Figure C.2.

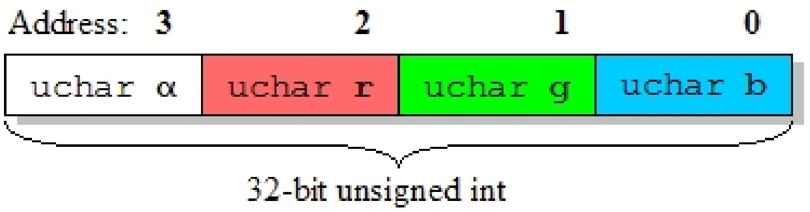


Figure C.2: Illustration of the QRgba data type. Data type uchar is read "unsigned character," and can be thought of as an 8-bit nonnegative integer.

For our purposes, we will ignore the transparency value and concentrate solely on the red, green, and blue intensities. Due to this restriction, we will use the QRgb function, which hides the transparency value from the user (no pun intended). Our images will still have alpha values associated with each pixel, but the transparency information of a pixel assigned using the QRgb function defaults to zero.

Suppose we have a 32-bit QRgba integer `pixel` and a 8-bit uchar singlecolor. We can extract the individual color intensities using the following functions.

- `singlecolor = qRed(pixel)`
- `singlecolor = qGreen(pixel)`
- `singlecolor = qBlue(pixel)`

Remember that if QRgba is a pixel in a *grayscaled* image, then all the color intensities are the same. That is, `qRed(pixel)`=`qGreen(pixel)`=`qBlue(pixel)`=the gray intensity of the pixel.

C.4 The QImage Object

The images from the camera will be supplied to our code in the form of a pointer to a QImage object, from Trolltech's open source edition of Qt. In c++, an object is basically a collection of variables and functions. In particular, we list here some of the functions contained inside the QImage object that give us useful information about the image. For instance, imagine we have received `photo` as a pointer to a QImage object. Then we can gain information about the image to which `photo` points by calling the functions shown in Table C.1.

function	returns	data type
`photo->bits()`	mem addr of first pixel's data	uchar *
`photo->numbytes()`	total number of bytes in image	int
`photo->height()`	height in pixels of image	int
`photo->width()`	width in pixels of image	int

Table C.1: Some functions of the QImage pointer `photo`. The asterisk in the first row indicates that `photo->bits()` returns a pointer to a uchar.

- More details on the information contained in the QImage object can be found in the Qt Assistant, located in the Qt folder of the Start menu.

We will be working with our images on a pixel-by-pixel basis, so it is useful to know how the image is stored in memory. Consider Figure C.3. The `bits()` function gives us the memory address of the first byte of data, which is the blue intensity of the first pixel. Each pixel is represented by a Rgba quadruplet, with each value occupying a subsequent location in memory, the blue intensity for the second pixel following the alpha value for the first pixel, etc.

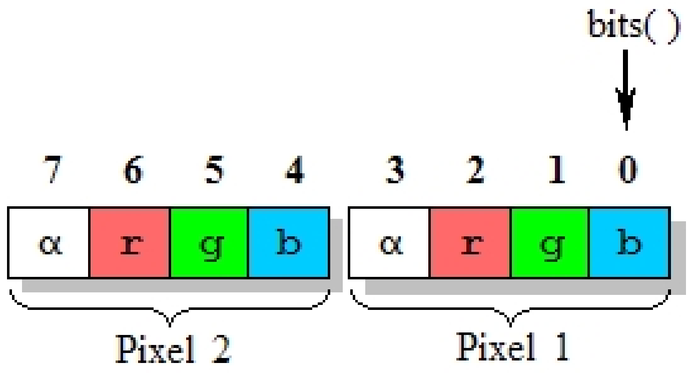


Figure C.3: The first two pixels of QImage as they appear in memory. Pixel 1 is located at row 1 and column 1 of the image; Pixel 2 is at row 1, column 2.

C.5 Referencing the Contents of a QImage

We can access the image data in several ways; we illustrate two methods here. Again, assume that we have received `photo`, a pointer to a QImage.

1. **Linear Reference** We can move sequentially through memory from one pixel to the next. This is useful for tasks that do not require knowledge of adjacent pixels in an image, like selecting the grayscale threshhold. Consider the following code, which builds a histogram for the grayscale values in the image.

```
int totalpixels;
uchar graylevel;
uchar *pfirstbyte;
int H[256];
     // histogram

totalpixels = photo->numBytes()/4;
pfirstbyte  = photo->bits();

for(int i=0; i<256; i++) H[i]=0;
     // initialize histogram bins to zero

for(int i=0; i<totalpixels; i++)
{
```

```
15     graylevel = *(pfirstbyte + i*4);
16      // reads the blue intensity of each pixel
17     H[graylevel]=H[graylevel]+1;
18      // increment appropriate histogram bin
19 }
```

Each pixel has 4 bytes of data associated with it, so Line 6 divides the number of bytes in the image by 4 to compute the number of pixels in the image. Line 9 initializes the bins in the histogram to zero. The second loop (Lines 11-15) moves sequentially through memory, reading the gray intensity of each pixel and incrementing the corresponding bin in the histogram. For our grayscaled image, red=green=blue. Therefore it suffices to check only the blue value for the gray intensity.

2. **Geometric Reference** We can work with each pixel based on its row and column location in the image. This is critical for tasks that involve knowledge of adjacent pixels, such as associating all the pixels in each object. Consider the following code that colors each pixel with red intensity corresponding to its row in the image and with blue intensity corresponding to its column in the image.

```
1  int height, width;
2  int red, green, blue;
3  QRgb *pfirstphotorgb, *rgbaddress;
4
5  height = photo->height();
6       // image dimension
7  width  = photo->width();
8       // image dimension
9  pfirstphotorgb = (QRgb*)photo->bits();
10      // address of rgb triple for pixel 1
11
12 for(int row=0; row<height; row++)
13 {
14      for(int col=0; col<width; col++)
15      {
16           red  = row % 255;
17           green = 0;
18           blue  = col % 255;
19            // address of the rgb triple
20 // for pixel at row, col
21           rgbaddress = pfirstphotorgb +
22                        row*width + col;
23           *(rgbaddress) = qRgb(red,green,blue);
24       }
25 }
```

Lines 5 and 6 determine the dimensions of the image. Since `photo->bits()` returns the address of a pointer to uchar data, Line 7 type casts the address to be that of QRgb data. Line 17 calculates the address corrosponding to the pixel at the specified row and column. Line 18 assigns to the pixel the desired red, green, and blue values to the pixel.

C.6 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:

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

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

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^c_w	$= [T_x T_y T_z]^T$ origin of world frame expressed in camera frame
R^c_w	rotation expressing camera frame w.r.t. world frame.

Table C.2: 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 $R^c_w = R_{z,\theta}$ 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

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

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.

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