Eye-In-Hand Visual Servoing
with a 4-Joint Robot Arm

An Introductory Robotics Workshop
at MIT CSAIL

Marsette Vona (vona@mit.edu)
Kevin Quigley (kevin.quigley@gmail.com)
Daniela Rus (rus@mit.edu)

February 2010

Contents

1 Introduction and Safety Procedures 2
  1.1 Safety, Park, Home, and Stop . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2 Forward and Inverse Kinematics 4
  2.1 Keeping the Wrist Horizontal . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.2 Forward Kinematics & the Jacobian . . . . . . . . . . . . . . . . . . . . . . 6
  2.3 Inverse Kinematics: the Inverse Jacobian Method . . . . . . . . . . . . . . 9

3 Image Processing: Detecting the Ball 10
  3.1 Attaching the Camera . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  3.2 Color Filtering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  3.3 Detecting the Ball Using Blobs . . . . . . . . . . . . . . . . . . . . . . . . 12

4 Visual Servoing to Follow the Ball 13
  4.1 Tracking the Ball . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
1 Introduction and Safety Procedures

Time to spend in this section: 20 minutes.

In this 2.5 hour exercise you will learn to control a robot arm, using a small video camera, so its movements track a ball. The general idea of using a camera to continuously drive the motion of a robot to satisfy some goal is called visual servoing. In our case, the camera is mounted on the arm itself, near the gripper, in an eye-in-hand configuration. First in section 2 you will learn about the kinematics of the arm by performing a few preliminary experiments. Then in section 3 you will learn about the image processing steps used to detect the location of the ball relative to the camera. Finally, in section 4 you will put these two capabilities together to make the arm follow the ball as you move it around.

Figure 1 shows all the components of the experimental setup you’ll be using. The main elements are

- A robot arm with 4 revolute (i.e. rotating) joints actuated by model-airplane servo motors, plus a 5th motor that opens and closes a gripper.

- A small USB camera that you will mount on the arm near the gripper at the beginning of section 3. For now, keep the camera and its hardware in the plastic bag.

- A laptop computer that will control the arm and process the images from the camera. We have developed software that you will use throughout the exercise. It should already
be running on the laptop. We’ll refer to this software from now on as “the GUI.” It
has three tabs at the top which correspond to the three sections of the exercise—for
now you should have the “Kinematics” tab selected.

• A vertical white board with graph paper, and a pen that can be gripped by the arm.
You will use these in section 2. Please do not remove the cap from the pen until
you’re ready—the ink dries out quickly. We’ve also provided pencils for you to
use for calculations and notes, please only use the pen as instructed later in section 2.

1.1 Safety, Park, Home, and Stop

Though the motors in this arm are not strong enough to cause
serious damage, please use caution and do not put your fingers
near the moving parts of the arm when it’s turned on. Also
be careful not to let hair or dangling jewelry get tangled in
the arm joints or springs (tie hair back).

In an emergency, for example, if the arm is making an an unusual
noise, or you have pinched a finger, you can quickly disable the
arm by pulling the power plug from the arm. Locate it now by
referring to figure 1, but don’t pull it unless absolutely necessary.

The arm has a number of wires. Watch carefully that these
do not get caught as the arm moves.

Besides your safety, we also need to avoid damage to the arm itself. In some cases, the
arm may reset itself. This is generally caused by the arm drawing too much power for
a brief period, somewhat like a circuit breaker in your home. If the arm does reset itself it
will lose power. If this happens, please ensure that nothing is obstructing the arm,
and that no wires are caught, and then click “home” to re-energize it and return it to
the home position. Note that in this instance the arm will move very rapidly.

You will notice that the arm will make a humming noise when it’s turned on. This means
the motors are active and trying to hold the position of the arm against gravity. If the arm
collides with something and gets stuck, the motors may not be strong enough to free it. This
will usually cause an unusually loud noise and will also make the motors heat up; they could
eventually be damaged. Do not let the arm stay too long in such a stalled position. If you
won’t be using the arm for a while, park it.

You can put the arm in a controlled stop by clicking the “stop” button. This
leaves the arm powered and in its current position.

Initially the arm is in its “park” position, shown in figure 2. Clicking the “home” button
in the GUI will move the arm to the home position, also shown in the figure.
2 Forward and Inverse Kinematics

Time to spend in this section: 70 minutes.

In this section you will start moving the arm and learn about its kinematics. Kinematics is the study of motion without considering its causes—physical forces and torques. For this arm we will always assume that the motors provide whatever torques are necessary to move to a commanded position. This is a common simplifying approach in robotics, though you should also know that the control of many more advanced robots goes beyond the pure kinematic approach.

First you will do a short exercise to derive an equation which moves the wrist joint, $j_3$ (see figure 3), based on the current angles of joints $j_1$ and $j_2$, so that the gripper always remains flat and parallel to the ground. Then, you will put a pen in the gripper and quantitatively measure the effects of moving $j_1$ and $j_2$—this is called forward kinematics. In most cases what we really want is the opposite, called inverse kinematics: given a desired movement of the gripper, find a corresponding movement of the joints. In the final part of this section you will learn one method for computing inverse kinematics.

2.1 Keeping the Wrist Horizontal

We can put the arm’s joints into two categories: $j_0$ and $\{j_1, j_2, j_3\}$. The axis of rotation of $j_0$ is vertical, and thus its motion has no effect on the up-and-down motion (i.e. along the world-frame $y$ axis) of the gripper, but it can change the direction the gripper faces, $\theta$, by rotating it in an arc about the $y$ axis. The axes of $j_1$, $j_2$, and $j_3$ are all horizontal, and we can think of their combined effect on the gripper pose in three components: (1) the height $h$ of the gripper along the world-frame $y$ axis, (2) the radial distance $r$ from the gripper to the $y$ axis, and (3) the pitch angle $\rho$ of the gripper about the grip-frame $x$ axis.

Try it. Make sure the “Kinematics” tab is visible in the GUI (the tabs are at the top of the window). Turn the arm on by hitting the “home” button in the GUI (or if you already
turned it on, hit “home” again to return it to the home position. Now type in a value in the \( j_0 \) box (in degrees, and respecting the limits) and hit “move”. Then return the joint to its home position by restoring the original angle (the home angles are also listed in figure 2). Do the same for each of the other joints, but only move one at a time (and then return it to the home angle) so that you can see the effect of each joint separately.

**As a simplifying design choice, we are going to control the arm in such a way that \( \rho \) is always zero, keeping the wrist horizontal and the gripper flat and parallel to the ground.** One way to do this is to continuously have the computer calculate a value for \( j_3 \), given the current values of \( j_1 \) and \( j_2 \), that keeps \( \rho \) zero. \textbf{What is the formula that the computer should calculate?} Look carefully at the definition of the joint angles in figure 3 and write the formula in the box below. Your formula can and should include the variables \( j_1 \) and \( j_2 \). Use a pencil in case you want to change your answer.

\[
\hat{j}_3 = \text{[formula]} \]

Now you’ll use the GUI to have the computer do this calculation automatically. Home the arm, and then type your formula into the box for \( j_3 \). You can type the formula directly—the GUI will understand symbols like +, −, *, and /, as well as variable references like \( j_1 \) and \( j_2 \). Now type in some new angles for \( j_1 \) and \( j_2 \) and hit “move”. Does the wrist remain (essentially) horizontal? If not, check your formula.

\textbf{Only the field } \( j_3 \text{ can accept other joint references}.\)
2.2  Forward Kinematics & the Jacobian

With \( \rho \) fixed at zero, and \( j_3 \) slaved to \( j_1 \) and \( j_2 \), we can think of the arm as a mathematical function, or *mapping*, that takes three inputs—the joint angles \( j_0, j_1, \) and \( j_2 \)—and produces three outputs: the gripper height \( h \), radial distance \( r \), and angle \( \theta \). This is called the *forward kinematic mapping*, or just the *forward kinematics*, of the arm.

In fact, there are mathematical formulas which compute \( h, r, \) and \( \theta \) given \( j_0, j_1, \) and \( j_2 \). We can think of the arm as a physical calculator that computes these formulas—when you set a particular set of input joint angles, the gripper moves and you could physically measure the resulting \( h, r, \) and \( \theta \) (soon you will actually do this!). For this arm it turns out that the formulas are relatively simple. They involve the link lengths \( l_0, l_1, l_2, \) and \( l_3 \) (see figure 3) and trigonometric functions (sine and cosine) of the joint angles. Can you come up with the formulas? Don’t spend more than about 10 minutes on this; you can work on it more later at home if you’re interested, but the answer won’t be critical for anything else we do in the exercise. You’ll need some knowledge of trigonometry.

\[
\begin{align*}
  h &= \\
  r &= \\
  \theta &= 
\end{align*}
\]

The forward kinematic mapping is interesting, but think about the opposite: what if we had formulas that computed \( j_0, j_1, \) and \( j_2 \) starting from given \( h, r, \) and \( \theta \)? That would be obviously useful, because it would allow us to move the gripper to any (reachable) target location in space. Such a reverse mapping is called the *inverse kinematics* of the arm. In many cases it’s actually possible to derive and write out formulas for the inverse case, just like you wrote above for the forward kinematics. But these are usually more complicated. There is another way.

If you start the arm in an initial pose, and then make a small “tweaks” to each joint angle in turn, you can measure the *local* effect of each joint on each of the three outputs \( h, r, \) and \( \theta \). That is, a delta joint angle \( dj_i \) will result in delta gripper motions \( dh, dr, \) and \( d\theta \). If you divide the latter by the former, you are effectively taking a derivative\(^1\). Six derivatives, to be precise; for example, \( dh/dj_1 \) is the derivative of the gripper height with respect to motion of joint 1, and \( d\theta/dj_0 \) is the derivative of the gripper angle with respect to motion of joint 0. Note: the numeric value of these derivatives generally depends on the initial pose of the arm, but they exist and can be computed for any pose.

**Let’s try it.** Given the forward kinematic mapping, the computer can (and we have already programmed it to) automatically calculate the necessary derivatives. But for you to get a feeling for what the computer is doing, you’re going to measure a few of them manually.

---

\(^1\)If you have never learned about derivatives, which are taught in calculus, don’t worry. In this handout you can substitute the word “sensitivity” for “derivative”, as in “\( dh/dj_1 \) is the sensitivity (or derivative) of the gripper height with respect to change in angle of \( j_1 \): \( dh/dj_1 \) is the ratio of change in \( h \) to change in \( j_1 \).”
To simplify things, you’re going to skip \( j_0 \) and \( \theta \). You can probably see quickly that \( \theta = j_0 \), and that

\[
\frac{d\theta}{dj_0} = 1, \quad \frac{d\theta}{dj_1} = 0, \quad \frac{d\theta}{dj_2} = 0.
\]

So you are going to work only with \( j_1, j_2, h, \) and \( r \). That gives us four derivatives to measure, and you’re going to arrange them in a 2x2 matrix:\(^2\)

\[
J = \begin{bmatrix}
\frac{dr}{dj_1} & \frac{dr}{dj_2} \\
\frac{dh}{dj_1} & \frac{dh}{dj_2}
\end{bmatrix}.
\]

This is called a *Jacobian matrix* after the nineteenth-century mathematician Carl Gustav Jacob Jacobi (no kidding).

Get the pen (don’t take the cap off yet) and put it in the gripper as shown in figure 4. Home the arm and then hit the “grab pen” button in the GUI to close the gripper appropriately.

Once the pen is correctly gripped, take off the cap. Now, **being careful not to let the pen touch the paper yet**, slowly move the graph paper board so that the plastic base is flush with the left side of the arm base, as shown in figure 5.

You’re going to rotate \( j_0 \) so that the pen is almost, but not quite, touching the paper. Enter the value -42 into the \( j_0 \) box in the GUI (keep the other joint angles at their current settings) and hit “move”. The pen will approach the paper. If it hits the paper briefly, that’s OK. If it hits the paper and stays there, move the paper back a little so that there is a small gap. **Don’t let the pen stay on the paper too long or it will make a blotch.** If the

---

\(^2\)Never studied matrices? Don’t panic. All you really need to know is that a matrix is a table of numbers. All the other mathematics will be given to you, and it only involves basic addition, subtraction, multiplication, and division.
pen has not yet hit the paper, **give the paper a slight and gentle push from the back** so that it briefly comes in contact with the pen. The longer you hold it, the larger the mark the pen will make. Just make a small mark, because later you’ll be measuring it’s position and you want to be able to easily find the center. You should have one mark on the paper at this point. Use your pencil to label it “o” for “origin”.

You’ll now move $j_1$ and $j_2$ small amounts, relative to their current (home) positions, and make two additional marks.

1. First move $j_1$: leaving the other joints at their current settings, type in a new value for $j_1$ that moves it by a small angle, say +10 degrees, relative to its current position. Since you’ll need to return it in a moment, use the GUI to do the math. The current value for $j_1$ should be -45. So type in the box “-45+10”.

2. Hit “move” and then gently push the paper again to make a mark.

3. Label the new mark “1” with your pencil.

4. Now return $j_1$ to its original setting of -45, and hit “move” again.

5. Finally, do the same procedure for $j_2$, labeling the third mark “2”.

Carefully move the paper away from the pen without making any new marks. **Put the cap back on the pen**, and lay the board flat on the table with the paper face up (don’t pull it off the board, and don’t remove the base from the board; just let the base hang off the side of the table). Use the ruler to measure (**in centimeters**) the horizontal and vertical coordinates of marks labeled “1” and “2” relative to the mark labeled “o”. That is, consider “o” to be the origin of a Cartesian coordinate system and label the coordinates of
the other two points. Divide these coordinates by the delta joint angle (10 degrees) to fill in the Jacobian matrix below. The vertical coordinates are changes in the gripper height $h$ relative to the start pose, and the horizontal coordinates are changes in the gripper radial position $r$ relative to the start pose. So if your points have coordinates $(r_1, h_1)$ and $(r_2, h_2)$ relative to the origin you will fill in the matrix with the values of four divisions (remember it’s easy to divide by 10, just move the decimal place one position to the left):

$$J = \begin{bmatrix} \frac{dr}{dj_1} & \frac{dr}{dj_2} \\ \frac{dh}{dj_1} & \frac{dh}{dj_2} \end{bmatrix} = \begin{bmatrix} \frac{r_1}{10} & \frac{r_2}{10} \\ \frac{h_1}{10} & \frac{h_2}{10} \end{bmatrix} = \begin{bmatrix} \frac{r_1}{10} & \frac{r_2}{10} \\ \frac{h_1}{10} & \frac{h_2}{10} \end{bmatrix}.$$ 

The Jacobian can be used to predict the expected gripper motion that would occur due to a small motion of the joint angles relative to the current pose. The inputs in this case are two numbers $e_{j_1}$ and $e_{j_2}$ giving the relative joint motion, and the outputs are two numbers $e_r$ and $e_h$ giving the relative gripper motion. The computation works like this:

$$\begin{bmatrix} e_r \\ e_h \end{bmatrix} = J \begin{bmatrix} e_{j_1} \\ e_{j_2} \end{bmatrix} = \begin{bmatrix} \frac{dr}{dj_1} & \frac{dr}{dj_2} \\ \frac{dh}{dj_1} & \frac{dh}{dj_2} \end{bmatrix} \begin{bmatrix} e_{j_1} \\ e_{j_2} \end{bmatrix} = \begin{bmatrix} (\frac{dr}{dj_1})e_{j_1} + (\frac{dr}{dj_2})e_{j_2} \\ (\frac{dh}{dj_1})e_{j_1} + (\frac{dh}{dj_2})e_{j_2} \end{bmatrix}.$$ 

You won’t use this formula directly today, but it leads us to a more interesting formula which computes $e_{j_1}$ and $e_{j_2}$ given $e_r$ and $e_h$.

### 2.3 Inverse Kinematics: the Inverse Jacobian Method

As you may have already learned, it is generally possible to \textit{invert} a matrix\(^3\). For a 2x2 matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

the formula for computing the inverse matrix $A^{-1}$ (also 2x2) is

$$A^{-1} = \begin{bmatrix} \frac{d}{(ad - bc)} & \frac{-b}{(ad - bc)} \\ \frac{-c}{(ad - bc)} & \frac{a}{(ad - bc)} \end{bmatrix}.$$ 

The inverse of the Jacobian is exactly what we need in order to solve for $e_{j_1}$ and $e_{j_2}$ given a desired $e_r$ and $e_h$:

$$\begin{bmatrix} e_{j_1} \\ e_{j_2} \end{bmatrix} = J^{-1} \begin{bmatrix} e_r \\ e_h \end{bmatrix} = \begin{bmatrix} \frac{dj_1}{dr} & \frac{dj_2}{dh} \\ \frac{dj_1}{dr} & \frac{dj_2}{dh} \end{bmatrix} \begin{bmatrix} e_r \\ e_h \end{bmatrix} = \begin{bmatrix} (\frac{dj_1}{dr})e_r + (\frac{dj_2}{dh})e_h \\ (\frac{dj_1}{dr})e_r + (\frac{dj_2}{dh})e_h \end{bmatrix}.$$ 

Above, you computed the four entries of the Jacobian for this arm at its home position. Now use the 2x2 matrix inversion formula to compute the entries of the corresponding inverse Jacobian (you can use the “general calculation” box in the GUI like a calculator):

\(^3\)Actually, not all matrices can be inverted directly. To be invertible, the matrix must be square (same number of rows as columns) and must also satisfy a second more subtle property that depends on its numeric values. It is possible for a Jacobian matrix to fail one or both of these conditions, in which case the robot is said to be at a \textit{singular configuration}. This is interesting, but we don’t have time to go into the details here.
\[ J^{-1} = \begin{bmatrix} \frac{dj_1}{dr} & \frac{dj_2}{dh} \\ \frac{dj_1}{dr} & \frac{dj_2}{dh} \end{bmatrix} = \begin{bmatrix} \phantom{\text{Matrix}} \\ \phantom{\text{Matrix}} \end{bmatrix}. \]

Let’s pick a small relative gripper motion, say \( er = 2\text{cm} \) and \( eh = -1\text{cm} \), compute the corresponding relative joint motion, and use the result to actually move the arm.

\[
\begin{bmatrix} e_j_1 \\ e_j_2 \end{bmatrix} = J^{-1} \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} \phantom{\text{Matrix}} \\ \phantom{\text{Matrix}} \end{bmatrix}. 
\]

Take the cap back off the pen and return the paper back so that the pen is aligned over the origin point you marked previously. Type in expressions that move \( j_1 \) and \( j_2 \) relative to their current position by adding your computed \( e_j_1 \) and \( e_j_2 \) (for example, if the current value of \( j_1 \) in the GUI is -45 and you computed 10.2 for \( e_j_1 \), you could type in “-45+10.2”), hit the “move” button, and make a final (fourth total) mark. Carefully move the paper away again, put the cap back on the pen, take the pen out of the gripper, and measure the coordinates of your new mark. Remember, the goal was to move the gripper 2cm out radially and 1cm down from the origin. How did you do?

Though it’s laborious to manually invert Jacobians, the computer can do it easily. Later on, the computer will be constantly computing Jacobians and inverting them to find small joint motions that move the gripper to track the ball.

### 3 Image Processing: Detecting the Ball

*Time to spend in this section: 30 minutes.*

In this section you’ll learn about how we detect the ball using video images from a camera. In general it can be very challenging to rapidly process a raw video stream and extract features of interest. Our approach will be in two stages. First we filter the raw image to select some pixels based on the color of the ball, and reject the others. This produces a binary image where a pixel is white if it matches the ball color and black otherwise. Second, we search for a color blob within an appropriate size range. If no blob is found we have not correctly detected the ball. Otherwise, we assume that the blob is enclosed in a circle which corresponds to the ball. The output of all this image processing will be three features (in this case, just numbers) \( u, v, d \): \( (u, v) \) are the pixel coordinates of the center of the ball in the image and \( d \) is the apparent diameter of the ball, again in pixels. In the final part of the exercise these three numbers will be used to derive the necessary motion of the gripper to track the ball, and then the Jacobian inverse kinematics will be used to map that desired gripper motion into joint commands that move the arm.
3.1 Attaching the Camera

To give you some experience in dealing with the practicalities of building robots, we leave
the final assembly of the camera to you. In the camera bag you will find

• the camera with two black mounting screws
• a small screwdriver
• some twist-ties
• an orange ping-pong ball

Turn the arm off by clicking “park”, and carefully attach the camera to the
gripper using the provided screws. So they don’t get lost, the screws are initially given
to you already screwed part of the way in. Remove them (being careful not to drop them),
and line the camera up in the gripper as shown in figure 6. Be sure to catch the notch at
the rear of the camera base with the metal bar on the gripper—this will keep the camera
from swinging around. Carefully re-thread the screws and tighten them. Make sure they go
in straight and do not over-tighten or the threads in the plastic may strip.

![two mounting screws](image)

![fit notch in plastic over metal bar](image)

Figure 6: Details of the camera mount.

Robots often have many wires and cables, and properly handling these is one of the
challenges of building robots in practice. **Find a good way to fasten the USB cable
from the camera to the arm using the twist ties.** Remember you need to provide
enough slack so all the joints of the arm can move without pulling too much on the cable.
Also try to mount the cable so it won’t get caught on the moving parts of the arm.

Plug the cable in to the laptop, and switch from the kinematics GUI to the Image
Processing GUI by clicking on the middle tab at the top. Verify that you see the moving
camera image in the GUI.

3.2 Color Filtering

The first stage of image processing will filter the raw image to select only pixels whose color
is close to the ball color. There are two parts to this: setting the “nominal” color, and then
setting tolerance values relative to this, because the actual color of each pixel of the ball image will vary a little depending on lighting.

To set the nominal color, click on the center of the ball in the center image. You can re-do this at any time to set a better nominal color. The selected color will show up in the color patch below the center image. The reset button can also be used to remove any existing selection.

If the filtered images are noisy (heavily speckled), the “smooth image” slider can be adjusted. Smoothing eliminates noise by including the values of surrounding pixels. Note that this will also lose detail as edges become less sharp.

Now you’ll try two methods for setting the tolerance values, based on two different color parametrization or color spaces. You may already be familiar with the first, which is the RGB space. Here a color is identified by three numbers which give the intensities of the red, green, and blue components. Think about what will happen when lighting conditions change, e.g. if one of your partners walks between the ball and a light source. In general, this will cause all three of the RGB components to change, possibly all by a significant amount. Try setting thresholds based on the RGB parametrization and see if you can get a reliable binary image which selects all and only pixels in the ball. Use the white board to shade/unshade the ball and see if your thresholds work in both cases.

There are alternatives to RGB. In particular, we will look at the HLS parametrization, which stands for hue, lightness, and saturation. Hue gives the base color. Lightness gives the intensity (brightness) of the color. Saturation determines how “strong” the color is (think of a light watercolor paint vs. a heavy house paint—both could have the same hue but the house paint would likely have a higher saturation). Imagine how the HLS components will vary as lighting changes: hue should stay relatively unchanged, while saturation and especially lightness could change significantly. Thus, thresholding based on HLS, and hue in particular, can give better and more reliable color detection than RGB. Adjust the HLS thresholds so that you get a good binary image. Again use the white board to shade/unshade the ball and try to get a single set of thresholds to work in either case.

3.3 Detecting the Ball Using Blobs

As humans we can visually look at the binary image and we “see” the circle representing the perimeter of the ball. If we made a printout we could draw this circle over the image and we could estimate its center location \((u,v)\) and its diameter \(d\) (again, these are measured in pixels at this stage). But it would be very laborious if we had to manually do this for every image—there are around ten new images per second in the video stream! Clearly we want the computer to take care of this task. How can we implement this particular vision task—detecting blobs?

There are several ways to do it. Here, we use a technique which looks to see if pixels are connected to pixels previously assigned to a blob:

1. Scan through the pixels one by one, from left to right and top to bottom. This ensures that most pixels above and to the left of the current pixel will be labeled appropriately. For those in the top and left rows, consider it an “unlabeled pixel” in the steps below.

2. If the pixel above the current one is labeled, accept that label.
3. If the pixel to the left of the current one is labeled, accept that label.

4. If both are labeled, choose one, and make note that both refer to the same blob.

5. If neither are labeled, leave the current pixel unlabeled and move on.

At the end, labels marked as referring to the same blob will need to be combined. The largest blob should be that of the ball, although it will not always look like the ball!

We have implemented most of this procedure already, and the computer will do it (very quickly!) for each video frame, at a rate of about 10 frames per second. You need to tune the min and max diameter thresholds. Tune the thresholds until you get the ball reliably detected. If you are having trouble with false detections, try using the white board as a background in the image.

4 Visual Servoing to Follow the Ball

Time to spend in this section: 30 minutes.

You are now going to put together the work you did in the two prior sections and make the arm move to track the ball. This is an example of visual servoing, where we use measurements from a video image to continuously drive a robot to satisfy some goal. In our case, the goal is to keep moving the arm so that the ball is centered between the fingers of the gripper. Our configuration is called eye-in-hand because we’ve actually mounted the camera so that it is carried by the gripper. An alternate choice would have been eye-to-hand, where the camera would instead be mounted at a fixed location in the environment.

The overall method works like this:

1. For each video frame (again, we get about 10 per second) try to detect the center location \((u, v)\) and apparent diameter \(d\) of the ball in pixels. It turns out that we can do some relatively simple math to convert these into more meaningful units, namely the Cartesian coordinates \(C = (X_c, Y_c, Z_c)\) of the center of the ball in a camera frame, see figure 7. We won’t go into the details, but basically \((u, v)\) determine a ray from the camera through the center of the ball and \(d\) can be used to estimate the distance of the ball from the camera along that ray since we know the size of the ball.

2. The current actual position of the ball is now estimated as \(C = (X_c, Y_c, Z_c)\), relative to the camera. We want to move the gripper, and hence also the camera, so that the ball appears at a fixed goal location \(G = (X_g, Y_g, Z_g)\), also in camera frame. So we calculate the current errors \(E = C - G = (X_c - X_g, Y_c - Y_g, Z_c - Z_g)\), and we pass these to an inverse kinematics routine similar to (but slightly more advanced) than the one you developed in section 2 to get the change in joint angles \(\Delta j = (\Delta j_0, \Delta j_1, \Delta j_2)\) that will move the gripper to reduce the error. We do this in three stages:

   (a) First, we clamp the error. You can think of this geometrically as follows: the full error \(E\) tells how to travel all the way from the goal location \(G\) to the current location of the ball \(C\). The distance could be arbitrarily far. If it’s greater than
some maximum, we clamp it to that maximum. Remember that the Jacobian changes depending on the initial pose of the arm. Clamping the error like this helps ensure that we take only a small step, because once we do so, we’ll have to compute a new Jacobian.

(b) Second, we map the clamped error through an inverse Jacobian to get the change in each joint angle $\Delta j = (\Delta j_0, \Delta j_1, \Delta j_2) = J^{-1}(-E_{\text{clamped}})$.

(c) We could potentially just move the arm at this point, but if we always moved by the full delta amount, the arm’s motion could be jerky and unstable. We scale the deltas by multiplying them by a gain factor that is between 0 and 1 (thus they are always reduced). This results in more stable (but slower) motion.

4.1 Tracking the Ball

We have already implemented most of the above procedure, but we leave it to you to tune the error clamping and gain. **Switch to the final “Visual Servoing” tab in the GUI.** Leave your image processing settings intact—these are going to be actively used in the visual servoing portion. You can always switch back to the image processing tab to adjust them.

The Visual Servoing tab shows the original camera image as well as the HLS-thresholded binary image. Whenever the ball is correctly detected you will see a red circle drawn over the image. Make sure that you are seeing one red circle; if not, check your image processing settings. The goal location is marked by a blue cross—when the ball is at-goal the cross should approximately span the circle.
Arm motion is initially disabled—**enable it now by clicking “move”**. You’ll **tune the gain value first**, it’s initially set at its minimum. Hold the ball in view of the camera and keep it relatively fixed (don’t move it around very much). Now slowly increase the gain value and move the ball very slowly. Move slow enough so that the arm can keep up with you. Keep increasing the gain—the arm should move more and more quickly. Periodically stop moving the ball and watch the arm as it “settles.” Is it oscillating? If so, the gain has become too high. Reduce it a little to avoid oscillation. This is the best gain setting.

Now move on to **adjust the error clamping**. Initially the clamp value is at its maximum. Move the ball a little more quickly, **but keep it within the visibility of the camera**. The arm will move but it may over-compensate. Reduce the clamping in increments to avoid this over-compensation.

Finally, see if you can get the arm to grab the ball!