ISSN 0280-5316
ISRN LUTFD2/TFRT--5686--SE

Virtual Environment for Development of Visual Servoing Control Algorithms

José Luis de Mena

Department of Automatic Control
Lund Institute of Technology
May 2002

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
May 2002

Document Number
ISRN LUTFD2/TFRT--5686--SE

Author(s)
José Luis de Mena

Supervisor
Rolf Johansson and Johan Bengtsson
Prof. Enrique Baeyens (Univ.de Valladolid, Spain).

Sponsoring organization

Title and subtitle
Virtual Environment for Development of Visual Servoing Control Algorithms.
(Virtuell omgivning för utveckling av datorsynbaserad robotreglering).

Abstract
Our problem considered was whether a virtual environment could be used for development of visual servoing control algorithms.

We have used a virtual environment for the comparison of several kinds of controllers. The virtual environment is done in Java, and it consists of two industrial robots, IRB 2000 and IRB 6, a camera stereo system with two cameras mounted on the end-effector of the IRB 6, and one rolling ball and one bar.

The experiment consists of tracking and grasping the ball using the different controllers. The robot IRB 2000 should grasp the rolling ball. The control of the robot is done in Matlab.

We have three controllers. These controllers are function of the difference between the ball and the gripper. First, we use P-controller with a proportional gain. Second, the image-based Jacobian control is used but this controller needs an improvement because the robot tracks the ball with a little delay, then we use this controller with feedforward. The robot grasps the ball when the error between the ball and the gripper is less than one tolerance. In these two controllers, the depth is calculated with the two cameras (stereovision), therefore cameras need to be calibrated. Third, the hybrid controller is used. It is a mix of image-based and position-based controller. We use X and Y in Image space and Z in Cartesian space. Now, the 3D reconstruction is done from motion. It means we do not need calibrated cameras and the depth is calculated with adaptive control techniques. This adaptive control is used for recovering on-line the velocity of the ball. When the estimation of the ball is stable, the robot starts tracking the ball.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
64

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through:
University Library 2, Box 3, SE-221 00 Lund, Sweden
Fax +46 46 222 44 22 E-mail ub2@ub2.se

1. Introduction

1.1 Background

Computer vision

Computer vision involves the capturing, understanding and processing of images. In the part of the image processing we can encounter many problems. For example it is hard to distinguish objects of different material or even of different geometrical shape because their image could be the same. Therefore it is difficult to interpret images using simple models.

A problem occurs when we try to determine the position of a three-dimensional object using a two-dimensional image. The coordinates that give us the actual depth of the object are difficult to be determined. Introducing several cameras allow us to have richer information on the observed scene. Most systems use only one camera giving a loss of depth information. This can complicate the design of the system if the depth has to be estimated on-line. A solution for improving depth information is to utilise several cameras in a configuration such that the desired target is in the field of view of all cameras, see [1].

Most multi-camera configurations use two cameras. This is called a stereo rig configuration. It allows, if the system is calibrated, to obtain a complete Euclidean reconstruction of the observed objects. The stereo system must solve two problems. The first, known as the correspondence problem, consists of determining which item in the left camera corresponds to which item in the right camera. Depth can be calculated from the disparity between corresponding feature points found in at least two images. The second problem is the reconstruction problem. It must be able to reconstruct the 3D world using left and right cameras images.

Visual servoing

Visual servoing is an approach to control robot manipulators that is based on the visual perception of robot and workpiece. Visual servoing involves the use of one or more cameras and a control system to control the position of the end-effector of the robot relative to the workpiece as required by the task. In visual servo systems, information obtained from the vision system is used to control the motion of the robot on-line, opposed to off-line systems that use vision to determine the initial state of the world, prior to task execution.

There are two different kinds of visual servo control systems; Position-Based Visual Servoing (PBVS) and Image-Based Visual Servoing (IBVS). In the first, features are extracted from the camera image, and used in conjunction with a geometric model of the target and a known camera model. The position of the target can be estimated with respect to the camera. The main advantage of this approach is that it controls the camera trajectory directly in Cartesian space. In the second, control values are computed on the basis of image features directly. The image-based approach may reduce computational delay, eliminate the necessity for image interpretation and eliminate errors due to sensor modeling and camera calibration. A disadvantage is that there is no direct control over the Cartesian velocities of the robot end effector. As a result, the robot can execute trajectories that are desirable in the image but which can be contorted in Cartesian space.

In our experiment, we are going to mix PBVS and IBVS to improve the results. This kind of visual control system is called Hybrid-Based Visual Servo (HBVS).

Computer graphics

Java 3D is a framework for writing programs to display and interact with 3D graphics. Java 3D is a standard extension to the Java 2 JDK. The API provides a collection of high-level constructs for creating, manipulating and rendering 3D geometry. Java 3D provides functions for creating imagery, visualizations, animations, and interactive 3D graphics applications, see [2].

The Java 3D API extension is designed as a high-level platform-independent 3D graphics programming API.

The Java 3D API is a hierarchy of Java classes which serve as interface to a scene graph-based rendering system. The programmer works with high-level constructs for creating and manipulating 3D geometric objects that reside in a virtual universe, which is then rendered.

The API is focused on modelling graphics. The details of rendering are handled automatically. By taking advantage of Java threads, the Java 3D renderer is capable of rendering in parallel. The renderer can also automatically optimize for improved rendering performance.

1.2 Previous work

Base for this work have been earlier works carried out in the Department of Automatic Control at the University of Lund. Michail Bourmpos, in his project, developed tracking and grasping using vision and control algorithms, [7]. The experimental setup consists of the two robots and the stereo rig as well as a metallic ball rolling on a metallic board. The goal was to track and grasp the ball that rolls on the metallic board. Images of this ball were captured by two cameras and then the robot was able to track and grasp the ball. But the cameras were not initially calibrated. The data received from the stereo rig was used as feedback. This project was done with a Image-Based Visual Servo System (IBVS).

Tomas Olsson, [8], involved the grasping of a marker through vision feedback using Image-Jacobian. From this project we obtained information about the position of the cameras with hand-eye calibration and with respect to the robotic manipulator IRB2000. This project has also been useful to obtain information about image Jacobian for moving objects.

The project done by Luis Manuel Conde and Duarte Miguel Horta, [9], involved visual servoing with cameras of a moving feature point (the center of a cross). The goal was to keep the feature point centred inside the images captured by two cameras. In this project the cameras were initially calibrated.

An article done by Peter I. Corke, [3], provides a tutorial introduction to visual servo control of robotics manipulators. Other article used is done by Francois Chaumette, [4]. This article develops the introduction of several cameras in a visual servoing loop, much like stereo vision. In another article, Peter Corke combines image-based and position-based control to guarantee better results, [5]. For the virtual system, we have used the previous work with the Java 3D Robot by Mathias Haage, [12], and we have also had to use [6], that can be found on the Internet at http://java.sun.com.

1.3 Problem formulation

This thesis focus on two aspects of visual servoing. Simulation of sensor-based robot control loops is an important prototyping tool during development and testing. Java 3D is investigated as an image generation tool for use in simulation of visual servoing loops. The other aspect is the comparison of a hybrid visual servoing control scheme with an image-

based approach where the Java 3D environment is used for off-line development and control performance comparisons.

Virtual simulation environment for fast prototyping experiment

The experiment is a virtual experiment. The environment should be created like the real environment. We can have several problem formulations in this section.

- *Connection between Matlab and Virtual Robot*

  Our control algorithms are done in Matlab environment and the virtual robot and image processing are done in Java. Therefore, Java and Matlab should be connected. Java should send to Matlab cameras images and Matlab should send to Java robot trajectories.

- *Connection between Camera Sensor and Virtual Robot*

  Images should be shown by the cameras. We have two cameras and it is the first step in the closed loop of the system. Cameras should be added to TCP of the Virtual IRB6 and have to see the bar and the ball trajectory. We can only send to Matlab the center point of the ball and of the gripper. The robot, walls, and the floor should remove from the image.

- *Seamless transition between real and virtual system*

  We really want to transfer our experiment in virtual environment to real system. We can have new problems like noise in the image. This experiment needs to be adapted to new system.

- *Calibration of the camera system*

  The system needs some information on where in the workspace the cameras are located, and the camera setup. We can get this information by calibrating the cameras and the stereo system. The calibration process should give us information on the intrinsic parameters of the cameras, and where the two cameras are located relative to the robot, see 2.2.

- *Simulated time*

Java is not capable of generating images at video rate. Matlab and Java do not use the same clock and computational and network delays in real system should be modeled in virtual system. The problem is how we can consider time in the simulated system.

- *Dynamics of virtual robot*

Our virtual robot Irb2000 should have the same dynamics as the dynamics of real robot. The both robots should have the same behaviour.

Control of the robot

The robot should track and grasp the ball. The robot has to be controlled in Matlab. We should use images and calculate the error between old and desired position. We have three kinds of controller but we do not know which it is the best. Position-based controller, image-based controller and hybrid controller are used in the experiment. Hybrid controller is a mix of position-based and image-based controller.

1.4 Report outline

Chapter 2 describes the theory used. We will describe image processing and stereo vision, the image-based Jacobian control, camera calibration, and the Java3D virtual robot with environment.

In Chapter 3 we give an overview of the system, its architecture and all software components we have used in the experiment. We describe the experiment in a virtual world. We will briefly mention the characteristics of the two virtual robotic manipulators and the cameras used. Then the experiment of tracking and grasping a moving object is described.

Chapter 4 explains how the virtual environment is done. Geometry, material, lights, universe are described in virtual environment. Transfer to real system is also explained. We present problems that exist when we want to do the same experiment in the real system.

In Chapter 5 we present our results for the tracking and grasping of a moving object experiment with several kind of controllers and different velocities. We also present our results for the calibration of the object.

Chapter 6 describes unexpected problems, we will comment the results obtained in the virtual experiment and visual servoing.

In the chapter 7 we give conclusions and the result of what have been done. We will also suggest some improvements for future work.

Chapter 9 describes my control software in Simulink and the methods created in Java.

2. Material and Methods

2.1 Image-based Jacobian control

In image-based visual servoing, control is performed in image space. The control error *e* is defined as

$$e = y_r - y \qquad (2.1)$$

where y_r and y are measured in image space coordinates. y_r is the desired position of the gripper and y is the measured current position. If we want to drive the error to zero, a control law is

$$u = \dot{y} = K_v e \qquad (2.2)$$

where K_v is a constant gain. The output u contains the desired velocities of the end-effector points in the image.

The control signal sent to the robot is defined in task space, as a trajectory calculated from a velocity screw. A velocity screw is defined as a m-vector of translational and angular velocities. If we let r be the m-dimensional coordinates of the end-effector in task space, and $y = f(r)$ be k-vector of image coordinates, this can be done using an image-Jacobian

$$J_v(r) = \left[\frac{\partial f}{\partial r}\right] = \begin{pmatrix} \frac{\partial f_1(r)}{\partial r_1} & \cdots & \frac{\partial f_1(r)}{\partial r_m} \\ \vdots & & \vdots \\ \frac{\partial f_k(r)}{\partial r_1} & \cdots & \frac{\partial f_k(r)}{\partial r_m} \end{pmatrix}$$

and then

$$\dot{y} = J_v(r)\dot{r} \qquad (2.3)$$

where $\dot{y}$ is the image-space velocity of a feature point and $\dot{r}$ is the velocity screw in task space.

The image Jacobian for two cameras fixed in workspace is calculated as follows. Using the intrinsic camera matrices K_1 and K_2 we can normalize the cameras with the equations (2.5) and (2.6), see section 2.2.

$$\begin{pmatrix} u_1 \\ v_1 \end{pmatrix} = K_1^{-1} x_1 \tag{2.4}$$

$$\begin{pmatrix} u_2 \\ v_2 \end{pmatrix} = K_2^{-1} x_2 \tag{2.5}$$

giving the projection equations for a normalized camera

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{Z} \begin{pmatrix} X \\ Y \end{pmatrix} \tag{2.6}$$

Since the end-effector is rigid object, its velocity screw contains a translational velocity and a angular velocity, both dependent on time.

$$\dot{r} = \begin{pmatrix} T_x & T_y & T_z & \omega_x & \omega_y & \omega_z \end{pmatrix}^T \tag{2.6}$$

The time derivatives of the coordinates of a point on the end-effector $(X\ Y\ Z)^T$, expressed in the camera frame, can be written as

$$\dot{X} = Z\omega_y - Y\omega_z + T_x \tag{2.7}$$

$$\dot{Y} = X\omega_z - Z\omega_x + T_y \tag{2.8}$$

$$\dot{Z} = Y\omega_x - X\omega_y + T_z \tag{2.9}$$

Differentiating the projection equations with respect to time, and using (2.7), (2.8) and (2.9), we get

$$\dot{u} = \frac{d}{dt}\left(\frac{X}{Z}\right) = \frac{\dot{X}Z - X\dot{Z}}{Z^2} = \frac{Z\omega_y - Y\omega_z + T_x}{Z} - X\frac{Y\omega_x - X\omega_y + T_z}{Z^2} =$$

$$= \omega_y - v\omega_z + \frac{T_x}{Z} - uv\omega_x + u^2\omega_y - u\frac{T_z}{Z} \qquad (2.10)$$

$$\dot{v} = \frac{d}{dt}\left(\frac{Y}{Z}\right) = \frac{\dot{Y}Z - Y\dot{Z}}{Z^2} = \frac{X\omega_z - Z\omega_x + T_y}{Z} - Y\frac{Y\omega_x - X\omega_y + T_z}{Z^2} =$$

$$= u\omega_z - \omega_x + \frac{T_y}{Z} - v^2\omega_x + uv\omega_y - v\frac{T_z}{Z} \qquad (2.11)$$

and written on matrix form

$$\begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} 1/Z & 0 & -u/Z & -uv & 1+u^2 & -v \\ 0 & 1/Z & -v/Z & -1-v^2 & uv & u \end{pmatrix} \begin{pmatrix} T_x \\ T_y \\ T_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} =$$

$$= J(u, v, Z) \cdot \dot{r} \qquad (2.12)$$

The velocity screws in the derivation above are expressed in the coordinate systems of the cameras, and are not the same for two different cameras. Therefore we need to express the velocity screws in the same coordinate system before stacking the equations for the two different cameras, see [8].

This gives us the expression for the Jacobian of two cameras, with n points in each camera

$$J_v(r) = \begin{pmatrix} J_v\left(u_1^{c_1}, v_1^{c_1}, Z_1^{c_1},\right) \\ \vdots \\ J_v\left(u_n^{c_1}, v_n^{c_1}, Z_n^{c_1},\right) \\ J_v\left(u_1^{c_2}, v_1^{c_2}, Z_1^{c_2},\right) \\ \vdots \\ J_v\left(u_n^{c_2}, v_n^{c_2}, Z_n^{c_2},\right) \end{pmatrix} \qquad (2.13)$$

The unknown depths Z_i^c are estimated from measurements of the robot end-effector pose and the calibration.

The four equations given by each point includes some redundancy, since the point only contains three degrees of freedom. For our camera setup, cameras are at the same height and close, the epipolar constraint can be approximated by $v_1=v_2$. Therefore we have to delete the row corresponding to the v-coordinate of the points in image 2.

2.2 Camera calibration

One important camera model is the pinhole or perspective camera, which is used in our experiment, see figure 2.1. The perspective camera model consists of a point O, called center of projection, and a plane π that is the image plane. The distance between O and π is the focal length f. The line perpendicular to π that goes through O is the optical axis.

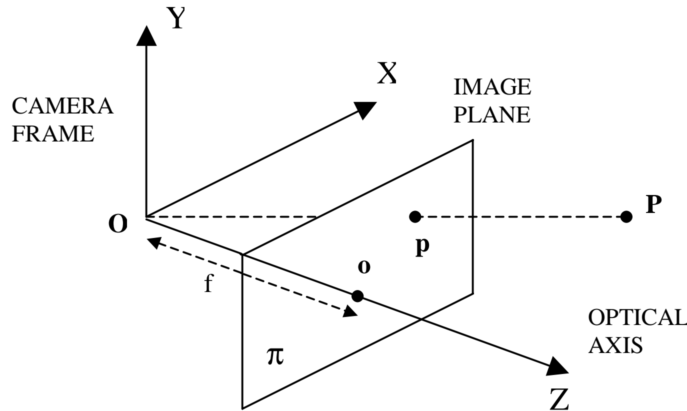


Figure 2.1 The perspective camera model

The projection equations for a point $(X\ Y\ Z)^T$ in Cartesian space for the perspective camera are given by

$$\lambda \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha & \gamma & u_0 & 0 \\ 0 & \beta & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.14}$$

where $\lambda = Z$ is the depth of the imaged point in the camera.

If we want to transform image-plane coordinates $(x\ y)^T$ to pixel-coordinates in the camera, we need to introduce a number of extra parameters that describe the CCD in the camera. These intrinsic parameters allow us to describe non-quadratic pixels and skew.

$$\lambda \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K_{3x3} R_{3x4} X \tag{2.15}$$

The matrix K is called the intrinsic camera matrix and R is the extrinsic camera matrix. With this last matrix, we could know the transformation that exists between the camera and the world reference frame. We would need a 3D translation vector T, which describes the relative positions of the origins of the two references frames, and an orthogonal matrix R which brings the corresponding axes of the two frames onto each other.

$$\lambda \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \cdot \begin{pmatrix} R & T \end{pmatrix} \cdot X^0 \tag{2.16}$$

where X^0 are object points in the object-centred coordinate system.

The images used in the calibration procedure should be taken from many different positions and orientations in order to get good estimate of the camera parameters.

The matrices obtained in the calibration are K_1 and K_2 that are the intrinsic camera parameters, $T_{c_2}^{c_1}$ that describes how the cameras are oriented with respect to each other, $T_b^{c_1}$ that is the transformation between camera 1 and the robot base, and T_g^t that is the transformation between the gripper frame and the frame attached to the calibration object. This matrix is not used in the experiment.

T_b^g that is the transformation from the robot base to the gripper frame can be calculated from the measurements of the joint positions and kinematics of the robot.

2.3 Communication

A virtual world can be used to test applications for the real world. In our case, a virtual robot has been used to test algorithms for control and image processing.

Using a virtual robot, it is possible to avoid collisions and bad trajectories before using the real robot and it is a fast and safe test bench for the system.

Our virtual robot prototype runs on Java platform and is communicated with other platforms using matcomm. Matcomm is a software package that was developed at Lund University, Department of Automatic Control. It is used for interprocess communication based on Berkeley sockets.

The virtual robot feedback is shown in the figure.

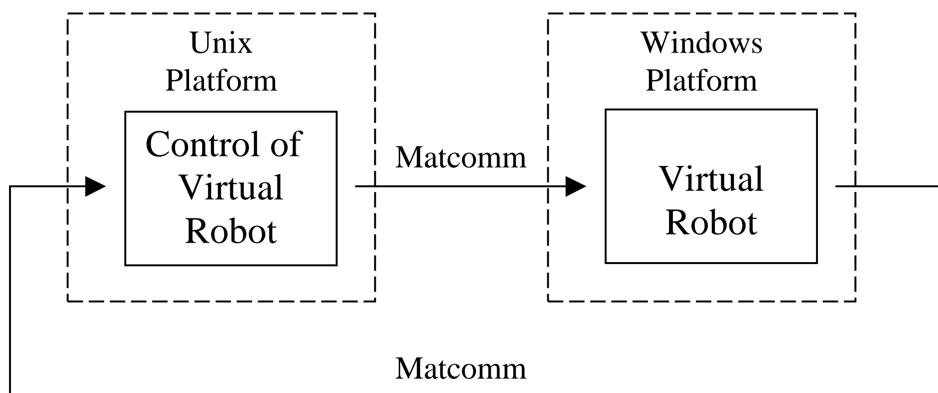


Figure 2.2 Communication and feedback control of virtual robot

The loop starts in the virtual robot and it sends images all time. The image is processed in Visual C++ (Windows environment) where it is received and processed. Later, we will explain in more detail all of this. This function returns the coordinates (X,Y) to give the path the robot should move for, but Cartesian coordinates will not be sent but joint angles coordinates. After that the joint values are sent the robot, the feedback loop is closed.

The virtual image is a view of the camera mounted on the end-effector, representing the visual sensor that performs the image acquisition to feedback the robot controller.

The image is generated on a PC running the Windows operating system by using socket inter-process communication. The image is then processed in Visual C++.

Data format

The object is a cube with the colours blue, red and green on each face. The image is received on each face as three matrices (R, G, B), each one having a size of 400x400 elements and being approximately 1Mbytes.

Image analysis

After the image acquisition, an image scan is done in order to find the cube centre. The scan starts on the first row and column and scans from the upper left corner to the right bottom corner. The image background is black and the cube colours are sharp. The colour mass center is calculated on regions with pixels of the same colour. Then, the colour mass centre is sent to the virtual robot.

2.4 Image processing and Stereo vision

Image processing

One of the simplest image analysis is image binarization of image thresholding, which can be thought of as an extreme form of gray-level quantization. Each pixel value in the image f is compared to a value T such that

$$g(n) = \begin{cases} 0 & \text{if} \quad f(n) \geq T \\ 1 & \quad\;\; f(n) < T \end{cases} \tag{2.17}$$

being n the number of pixel that is being analysed at this moment.

The threshold T used is of critical importance, since it controls the amount of information obtained. Different thresholds produces different binarized images. With one good threshold T, see lower, image thresholding can produce a result that is quite useful for further processing.

Thresholding is most commonly applied to images that can be characterized as having bimodal histograms. These histograms are often associated with images that contain objects and backgrounds having a significantly different average brightness. The goal is to separate the objects from the background, and to label them as object or as background. If the image

histogram contains well-separated modes associated with an object and with a background, then thresholding can be the means for achieving this separation.

A simple tool for identifying and labeling objects in a binary image is a process called region labeling. It is useful since once they are individually labeled, objects can be separately manipulated.

Region labeling seeks to identify connected groups of pixels in a binary image that all have the same binary value. The simplest such algorithm accomplishes this by scanning the entire image (left to right, top to bottom), searching for ocurrences of pixels of the same binary value and connected along the horizontal or vertical directions. A record of connected pixel groups is maintained in a separate label array having the same dimensions as f, as the image is scanned, see [10].

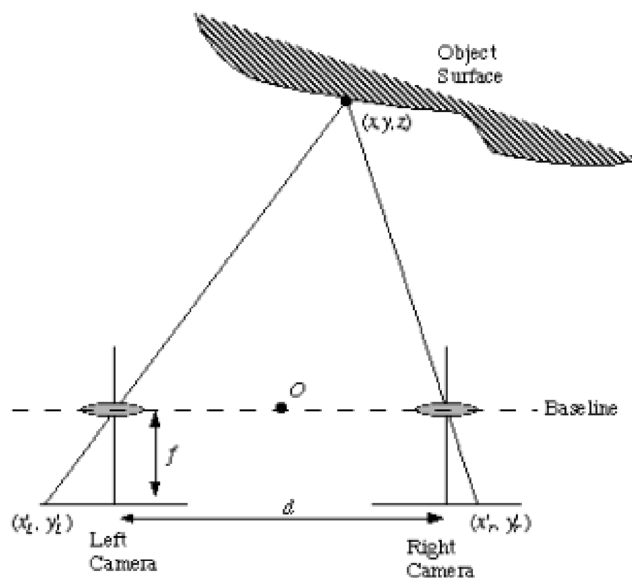


Figure 2.3 Stereo vision system

Stereo vision

Stereo vision refers to the ability to infer 3D information of the scene from two or more images taken from different viewpoints. It uses several cameras to solve for the loss of depth information.

Stereo vision has two problems. The correspondence problem has to solve which parts of the left and right images are projections of the same scene element, and the reconstruction

problem must reconstruct 3D-images given left and right images and the geometry of the stereo system.

To solve for the correspondence problem we have two ways; the correlation-based and the feature-based methods. The correlation-based methods apply to the totality of image points. Elements to match are image windows of fixed size, and the similarity criterion is a measure of the correlation between windows in the two images. The corresponding element is given by the window that maximizes the similarity criterion within a search region. This method is easier to implement and provide dense disparity maps. Feature-based method attempts to establish a correspondence between sparse sets of image features. This method restricts the search for correspondences to a sparse set of features. Instead of image windows, they use numerical and symbolic properties of features and instead of correlation like measures, they use a measure of the distance between feature descriptors. This method is suitable when a priori information is available about the scene, so that optimal features can be used. It is also insensitive to illumination changes and highlights.

2.5 Identification of robot dynamic

In our virtual world we do not have any object that is able to behave like our real robot. Then, we have to create an object with the same behaviour than our real robot. Therefore, we should identify the robot dynamic. To do it, we use an ARX model. The ARX model will be

$$y(t)+a_1 y(t-1)+\cdots+a_{n_a} y(t-n_a)=b_1 u(t-1)+\cdots+b_{n_b} u(t-n_b)+e(t) \qquad (2.18)$$

where e(t) is a white noise and

$$\begin{aligned} A(q) &= 1+a_1 q^{-1}+\cdots a_{n_a} q^{-n_a} \\ B(q) &= b_1 q^{-1}+\cdots b_{n_b} q^{-n_b} \end{aligned} \qquad (2.19)$$

Therefore, we have

$$y(t)=\frac{B(q)}{A(q)}\cdot u(t)+\frac{1}{A(q)}\cdot e(t) \qquad (2.20)$$

ARX model has two parts. One part called AR refers to the autoregressive part $A(q)y(t)$, and the other part X to the extra input $B(q)u(t)$ called the exogeneous variable in econometrics.

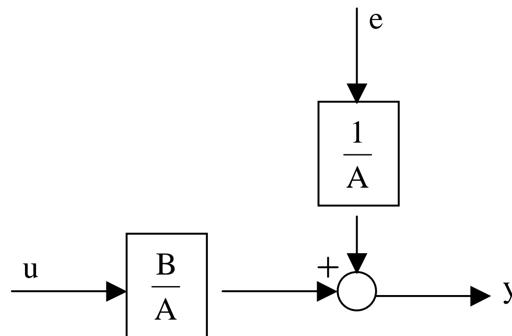


Figure 2.4 The ARX model structure

To be able to identify our system, we need the input-output data. We use the program *Exc_handler* to get the data record. This program creates a matlab-gui for sending excitation signals to and logging signals from the robotsystem. It is created at Department Of Automatic Control (Lund University). The input data will be a step input. We have to give it some parameters that are

Start level	0
Time for the step	1
Duration of step	500
End level	20
Duration of new step	500
End level	0

Therefore, we have a step input that goes up to 20 for 500 and goes down to 0 for 500.
Now, we have to choose the robot joint and the variable that we want to identify. We have to identify the six joints of the robot (one in each time) and the variable that we want is the position. We should record signals and later open the communication with the real robot. When it is done, we can start to record the signals.
This is done for a certain position of the robot and for each joint. If we want the identification is well done, we should have data in different positions of the robot for each joint. We have

used two positions for each joint, and these positions are the most significant in each joint. Therefore, we have 12 positions and 12 models.

As two positions are used for each joint, we should do a linear regression between them to obtain one only model in each joint, see [11]. Then, we should have 6 models at the end.

2.6 Simulated time

Java is not capable of generating images at video rate and computational and network delays in real system should be modeled in virtual system.

The solution is not to use the system clock, because Java and Matlab do not use the same clock. The solution is to use event-based time.

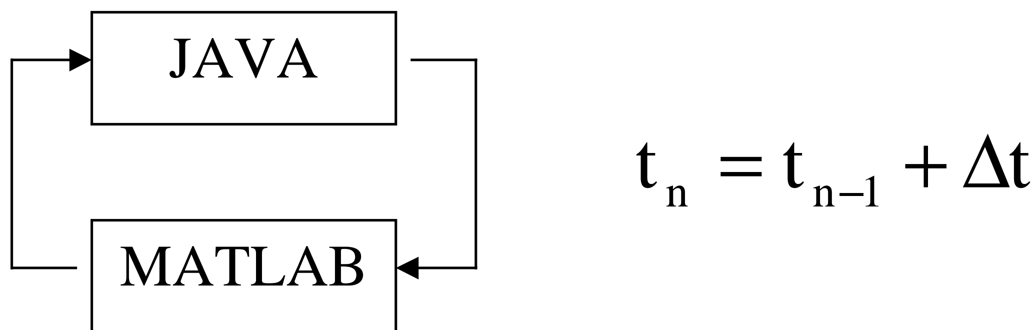


Figure 2.5 Simulate time

It means that when Java send the feature points to Matlab, the time is increased.

2.7 Java3D environment

A Java 3D program creates instances of Java 3D objects and places them into a scene graph data structure. The scene graph is an arrangement of 3D objects in a tree structure that completely specifies the content of a virtual universe, and how it is to be rendered. A scene graph is created using instances of Java 3D classes. The scene graph is assembled from objects to define the geometry, sound, lights, location, orientation, and appearance of visual

and audio objects. A Java 3D virtual universe is created from a scene graph[1] using the standard set of symbols. The nodes in the scene graph are the instances of Java 3D classes.

Java 3D programs can be written to run as stand alone applications, as applets in browsers which have been extend to support Java 3D, or both.

In Java 3D, there are three major ways to create new geometric content. One way uses the geometric utility classes for box, cone, cylinder and sphere. Another way is for the programmer to specify the vertex coordinates for points, line segments and polygonal surfaces. A third way is to use a geometry loader. If we want to create one geometry, we need to create a virtual universe and one coordinate system (Locale object). A visual object can be defined using just a Shape3D object and a Geometry node component.

Both interaction and animation are specified with Behavior objects. The Behavior class is an abstract class that provides the mechanism to include code to change the scene graph. With a Behavior object we can change the scene graph, or objects in the scene graph.

In Java3D, the portion of a scene where visual objects are illuminated by a particular light source is called that light object region of influence. The simplest region
of influence for a light source uses a Bound object. When a light source object influencing bounds intersects the bounds of a visual object, the light is used in shading the entire object. The influencing bounds of a light determine which objects to light, not which portions of objects to light.

2.8 Hybrid Control

Hybrid control is a mix of image and position control. In our experiment, we use the image space for the X and Y coordinates and the Cartesian space for Z coordinate. We use the image-Jacobian controller, see 2.1, for the X and Y and the PI-controller for the depth.

The algorithm that we use is based on the on-line estimation of the relative distance of the robot with respect to the camera. We do not know the depth of the ball, then adaptive control techniques are proposed. Therefore, this algorithm does not require accurate knowledge of the camera model and environment. The adaptive control schemes estimate depth-related parameters and not the dynamic parameters of the robot model. It can be used for the tracking of the ball when the depth information is not directly available. This adaptive

[1] A graph is a data composed of nodes and arcs. A node is a data element, and arc is relationship between data elements.

control is used for recovering on-line the velocity of the ball Tx(k), Ty(k) and Tz(k) in Cartesian coordinates. We use off-line calibration of the cameras.

The velocity of the ball in image space is

$$\begin{aligned} u^{(j)}(k) &= u_0^{(j)}(k) + \frac{S_x^{(j)}(k)}{Z_s^{(j)}(k)} \\ v^{(j)}(k) &= v_0^{(j)}(k) + \frac{S_y^{(j)}(k)}{Z_s^{(j)}(k)} \end{aligned} \qquad j = 1,\ldots,M \tag{2.21}$$

where M is the number of feature points of the ball that in our case, M = 3 and

$$\begin{aligned} S_x^{(j)}(k) &= -T_x(k) + x^{(j)}(k) \cdot T_z(k) \\ S_y^{(j)}(k) &= -T_y(k) + y^{(j)}(k) \cdot T_z(k) \end{aligned} \tag{2.22}$$

We know the position of the ball in image space and in world space in each time. ARMAX model is used in our experiment.

$$A_i\left(q^{-1}\right) \cdot y_i(k) = q^{-d} \cdot B_i\left(q^{-1}\right) \cdot u_{ci}(k) + \Gamma_i\left(q^{-1}\right) \cdot w_i(k) \qquad k>0,\ i = 1,2 \tag{2.23}$$

where

$$\begin{aligned} A_i(q^{-1}) &= 1 + \alpha_{i1} \cdot q^{-1} + \alpha_{i2} \cdot q^{-2} \\ B_i(q^{-1}) &= \beta_{i0} + \beta_{i1} \cdot q^{-1} \\ \Gamma_i(q^{-1}) &= 1 + \gamma_{i1} \cdot q^{-1} + \gamma_{i2} \cdot q^{-2} \end{aligned} \qquad i = 1,2 \tag{2.24}$$

The values of the coefficients depend on the values of the velocity T. The scalar input $u_{ci}(k)$ represents either Sx(k) or Sy(k), and the scalar $y_i(k)$ corresponds to the position of the ball. The introduced computational delays can be represented by the delay factor d. In our case, this factor will be one.

3. Experimental setup

The setup used consists of two industrial robots, Virtual IRB-2000 and Virtual IRB-6, a stereo vision system with two cameras, a bar and a rolling ball.

3.1 Virtual IRB-2000

The IRB-2000 is built-up by two larges arms and a wrist, see figure 3.1. The robot has 6 degrees of freedom, which means the end-effector can be moved to any desired position and orientation within task space, and a repeatable precision of 0.1mm. Joints 1,4 and 6 are cylindrical joints, while the other three joints, 2,3 and 5, are revolute joints.

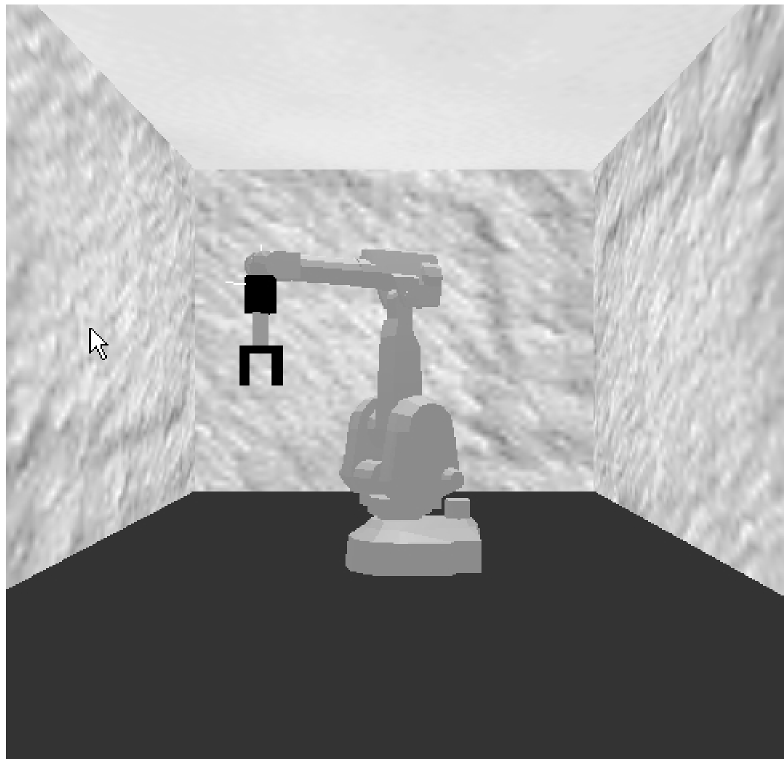


Figure 3.1 Virtual IRB-2000 robot

The robot is controlled from Matlab/Simulink by sending trajectories of position to robot through the network.

3.2 Virtual IRB-6 and Camera System

The IRB-6 is a robot with 5 degrees of freedom and a repeatable precision of 0.2mm. Joints are connected with six links. Joints 1 and 5 are cylindrical joints, while joints 2,3 and 4 are revolute joints.

The camera system consists of two cameras attached to a rig, which is mounted on the end-effector of the IRB-6 robot, see figure 3.2. The picture format is 400 x 400 pixels.

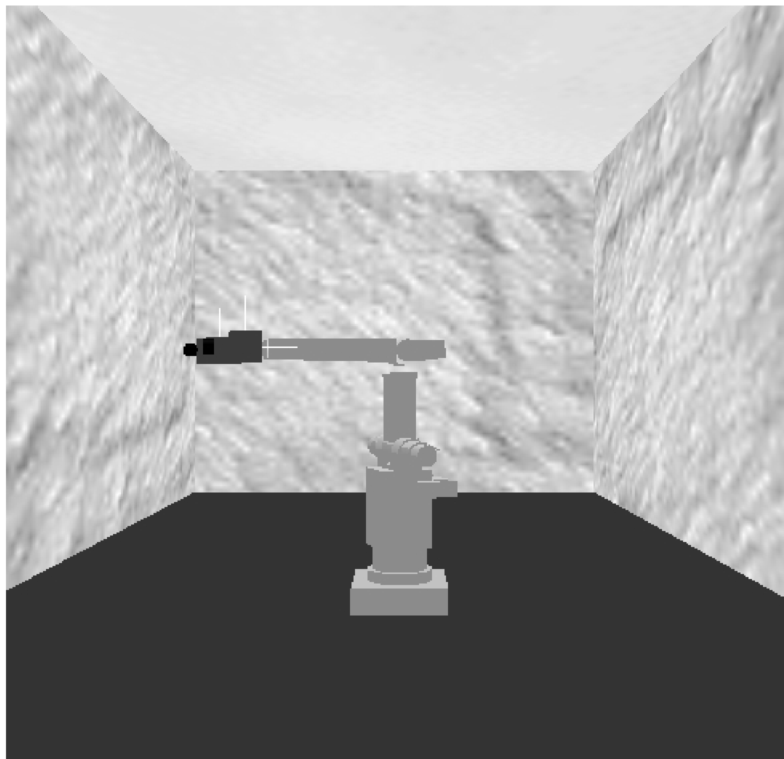


Figure 3.2 Virtual IRB-6 robot and camera system

Calibration of the camera system

In our experiment, we need to calibrate the cameras. The software of this calibration is provided by Tomas Olsson. Therefore we are going to use a method that is described next. We mount a calibration object (20cm x 20cm paper with a printed chessboard pattern) to the robot gripper itself, and use the known kinematics of the robot to get the rotation and translation of the calibration object between the images. The extrinsic parameters are not

completely unknown because we can get information about how the calibration object has moved. Then, we will need to estimate much fewer parameters.

We calibrate the system using image information and cartesian robot gripper information. The position and orientation of the gripper is obtained from measurement of the joint angles and using the kinematics of the robot. Image information is obtained by taking a picture of the planar calibration object with each of the two cameras for n different robot gripper positions, in our case we use 3 different robot gripper positions. One gripper position of them is shown in the figure 3.3. The image on the left is from camera 1 and the image on the right is from camera 2.

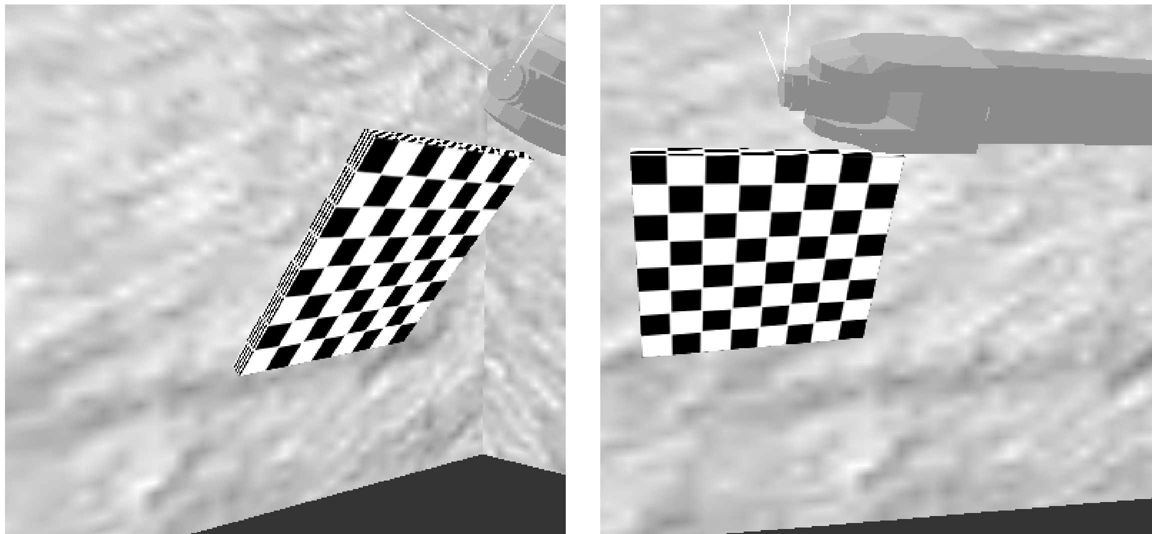


Figure 3.3 One gripper position in the calibration of the cameras

3.3 Bar and ball

The bar is rectangular and its colour is blue. It is leaning, see Appendix B, because the ball has to roll above it. Our ball is a red sphere. Ball dimensions are the same than the bar width.

The objects colour is very important due to image processing. Colours are sensed as near-linear combinations of long, medium, and short wavelenghts, which correspond to the three primary colours that are used in video camera systems: Red (R), Green (G), and Blue (B). Therefore, we have three matrices (R, G, B) in each image. If we want to send to matlab ball and bar images by separate, bar and ball colours should be different.

We do not have troubles with the colour because the red one is in a matrix and the blue one is in other matrix.

3.4 Software

SUN workstations are used for software development, as well as for robot operator interaction. Robots are controlled from VME-based embedded computers.

The data are transmitted between a SUN workstation running on a UNIX operating system and a Windows NT station.

For these data transmissions we have used a software developed at the University of Lund, Department of Automatic Control, called Matcomm. This software works with the TCP/IP protocol to transmit data between computers in a network where different system might exist. The reason to use matcomm is that it is an easy way to connect with another computer or module by sockets. Matcomm can be used in Unix and Windows environment. Data are sent, through matcomm, in a matrices format using sockets.

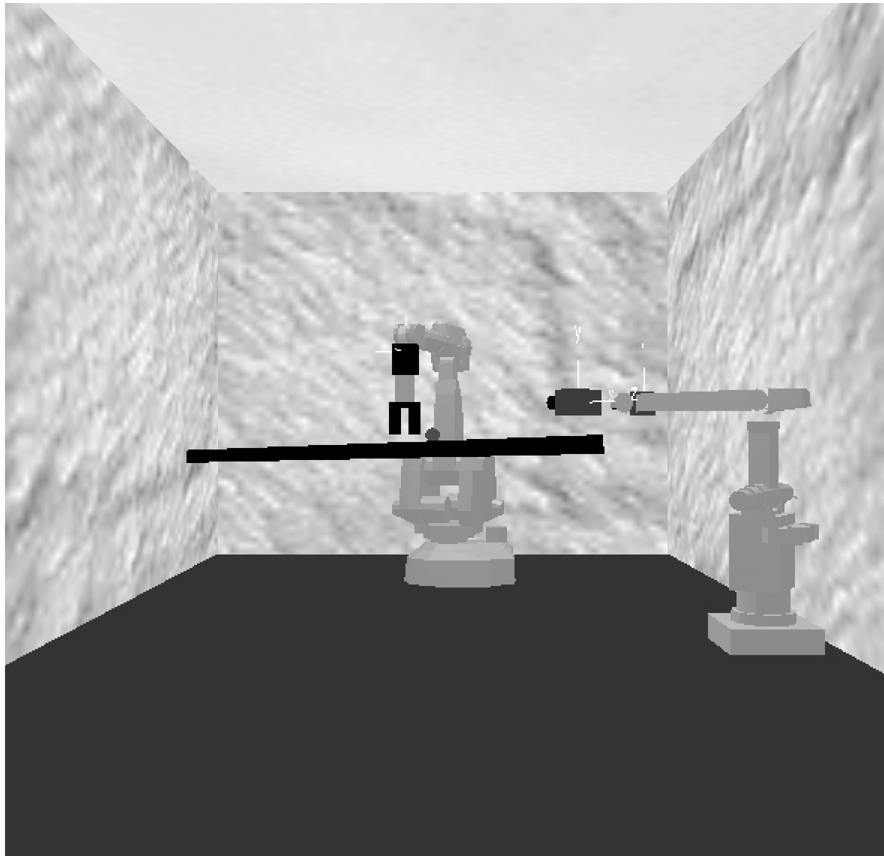


Figure 3.4 System setup

System setup starts in the cameras. Cameras mounted on the IRB-6 robot provide us with images. These images are processed in the PC running Windows platform. Then data are sent to the SUN workstation which runs the control algorithm for the IRB-2000 robot, see figure 3.4.

4. Virtual prototype

4.1 Java robot implementation

A Java3D virtual universe is created from a scene graph. The scene graph is assembled from objects to define the geometry, lights, location, orientation, and appearance of visual objects.

The Java3D scene graph is constructed of Node objects, see 1.1, in parent-child relationships forming a tree structure where a node is the root. There is only one path from the root of a scene graph to each leaf node. The scene graph is formed from the trees rooted at the Locale objects. Each scene graph has a single VirtualUniverse. The VirtualUniverse object has a list of Locale objects. A Locale object provides a reference point in the virtual universe. Each Locale object may serve as the root of multiple subgraphs of the scene graph. A BranchGroup object is the root of the subgraph, or branch graph. There are two different categories of scene subgraph but we only use one that is content branch group. This one specifies the contents of the virtual universe (geometry, appearance, behavior, location and lights).

```
vu = new VirtualUniverse();
javax.media.j3d.Locale locale = newjavax.media.j3d.Locale(vu);
```

Therefore, each Locale object in the virtual universe establishes a virtual world Cartesian coordinate system and it serves as the reference point for visual objects in a virtual universe.

After creating this part, we have to create the geometric content, that is, the shape of visual objects. A Shape3D scene graph node defines a visual object. The Shape3D object does not contain information about the shape or colour of a visual object. The Shape3D object can refer to one Geometry node component and one Appearance node component, and a visual object can be defined using just a Shape3D and a Geometry node component. Then, there are three ways to create geometric content (geometry utility classes, vertex coordinates for points and polygonal surfaces, and the geometry loader). We only use the first one for box, cylinder and sphere geometric primitives. A primitive object provides more flexibility by specifying shape without specifying colour. In a geometric primitive utility class, we cannot change the

geometry, but can change the appearance. These classes give us the flexibility to have multiple instances of the same geometric primitive where each can have a different appearance.

```
Sphere sph = new Sphere(0.04f, Sphere.GENERATE_NORMALS, 80,
app);
Box bx = new Box(2.91f,0.045f,0.045f,app);
Cylinder cyl = new Cylinder(0.08f,0.18f,app);
```

Data in a Geometric object are often insufficient to fully describe how an object looks. We need an Appearance object. An Appearance object can refer to several different appearance attribute objects such as material, texture or colouring attributes. ColoringAttributes controls how any primitive is coloured. Color3f objects specify a colour as a combination of red, green and blue (RGB) values. For single-precision floating point data, colour values range between 0.0 and 1.0, inclusive.

```
Color3f red    = new Color3f(1f, 0f, 0f);
Color3f green  = new Color3f(0f, 1f, 0f);
Color3f blue   = new Color3f(0f, 0f, 1f);
Color3f white   = new Color3f(1f, 1f, 1f);
Color3f black   = new Color3f(0f, 0f, 0f);
Appearance app = new Appearance();
app.setMaterial(new Material(objColor, black, objColor,
white,100f));
```

We should also create lights. Java3D shades visual objects based on the combination of their material properties and the lights in the virtual universe. Shading results from applying a lighting model to a visual object in the presence of light sources. In the real world, the colours we perceive are a combination of the physical properties of the object, the characteristics of the light sources, the objects' relative positions to light sources, and the angle from which the object is viewed. Java3D uses a lighting model to approximate the physics of the real world. The lighting model equation depends on three vectors: the surface normal, the light direction and the direction to the viewer eye in addition to the material properties of the object and the light characteristics.

The lighting model incorporates three kinds of real world lighting reflections: ambient, diffuse, and specular. Ambient reflection results from ambient light, constant low level light, in a scene. Diffuse reflection is the normal reflection of a light source from a visual object,

and specular reflections are the highlight of a light source from an object, which occur in certain situations. The lighting model is applied for each of the RGB colour components. Colour is a combination of many wavelenghts of light, not just three.

The portion of a scene where visual objects are illuminated by a particular light source is called that light object's region of influence. The simplest region of influence for a light source uses a Bound object and the setInfluencingBounds() method of the light.

```
AmbientLight bulb1 = new AmbientLight(new Color3f(1f,0f,0f));
PointLight   bulb2 = new PointLight(new
Color3f(1.0f,0.38f,0f),new Point3f(0.0f,0.0f,4.0f),new
Point3f(0.0f,0.1f,0.0f));
bulb1.setInfluencingBounds(bounds);
bulb2.setInfluencingBounds(bounds);
```

When a light source object's influencing bounds intersects the bounds of the visual object, the light is used in shading the entire object. The influencing bounds of a light determine which objects to light, not which portions of objects to light.

Ambient light objects provide light of the same intensity in all locations and in all directions. The lighting model calculates the ambient reflection of light as the product of AmbientLight intensity and the ambient material properties of visual object.

The PointLight is an omni-directional light source whose intensity attenuates with distance and has a location. PointLight objects approximate bare light bulbs, candles, or other light sources without reflectors or lenses. PointLight only participates in diffuse and specular reflections of the lighting model, so if we vary the location of a PointLight object we will change the shading of visual objects in a scene. The second parameter of the PointLight object is the light position and the third one set this light's current attenuation values and places it in the parameter specified.

A important aspect in Java3D is the material properties of a visual object. They are specified in the Material object of an appearance bundle. The Material object specifies ambient, diffuse, specular, and emissive colours and a shininess value. Each of the first three colours is used in the lighting model to calculate the corresponding reflection. The emissive colour allows visual object to glow. The shininess value is only used in calculating specular reflections.

```
appearance.setMaterial(new Material(Color3f ambientColor,
Color3f emissiveColor, Color3f diffuseColor, Color3f
specularColor, float shininess));
```

4.2 Dynamics of robot in the virtual world

Our virtual robot must move the same than the real robot. The virtual robot has to be created with the same kinematics than the real one.

We have done the identification of the real robot to put it in the virtual robot.
The virtual robot is done in several steps. Each joint is created by separated, and later all joints are joined. Then, we have one model for each joint. We should put this model in the virtual robot.

Therefore, dynamics of the virtual robot will have the same behaviour than the real robot.

4.3 Transfer to real system

When we want to transfer our virtual experiment to real system, we have some problems.

The image processing algorithms used was sensitive to noise caused by lighting conditions. The noise introduced in the tracking of the ball in the images can be quite large. The high amplitude of the noise introduced can create many problems in the correct controlling of the robotic manipulator.

Another important problem is the occlusion. The robot follows the rolling ball and is always closer to the cameras. Then, it is possible the ball is occluded by the IRB 2000 for a few samples. If this occurs, we lose track of the ball and our control of the robot would not be correct. Occlusion is a standard problem when dealing with systems that use feedback. The simplest solution could be to assume that for these few samples a normal linear interpolation would be enough, but this idea was not used.

The solution for these problems could be a predictor for the trajectory of the ball. We can use a Kalman filter, but before doing this, we have to identify the ball model.

Identification of the ball

To obtain the model equation we used the Subspace Model Identification (SMI) toolbox for Matlab, created by Professor Verhagen and Dr. Haverkamp, [14]. We have to identify the ball using this program. Our system will not inputs and we use information

obtained by the identified initial conditions of the state to predict the ball movement. The initial condition of the ball when entering the image, would always be different, since the initial velocity and position of the motion differs from experiment to experiment. This proves that we should not apply just a static state-space model from prediction but use the Kalman gain to adapt to each different movement.

Using the SMI Toolbox, the first we have to do is to use the command *dordpo*. With this command, we deliver information about the order of the linear time invariant (LTI) state space model we are about to identify. It also acts as a pre-processor for the command *dmopo* used next. The latter actually estimates the system matrices A and C, as well as the Kalman gain. We could have estimated the matrices B and D as well, but our model is assumed to have no input there is no need for such calculation. When all of this is done, we have to estimate the initial conditions of the states of our model. To do it, we use the command *dinit*. It is calculated with the estimated system matrices and with the set of input/output data.

Kalman filter

Let us consider a general estimation problem where the state space model of our system is:

$$x_{k+1} = A \cdot x_k + B \cdot u_k + v_k \qquad (4.1)$$
$$y_k = C \cdot x_k + D \cdot u_k + e_k$$

with $E(e_k) = 0$, $E(v_k) = 0$ and $E\{v \cdot v^T\} = R_1$, $E\{e \cdot e^T\} = R_2$

Therefore, the Kalman filter for prediction of x based on the data at time k is:

$$\hat{x}_{k+1|k} = A \cdot \hat{x}_{k|k-1} + B \cdot u_k + K_k \cdot \left(y_k - C \cdot \hat{x}_{k|k-1}\right) \qquad (4.2)$$
$$\hat{y}_k = C \cdot \hat{x}_{k|k-1} + D \cdot u_k$$

where

$$K_k = A \cdot P_k \cdot C^T \cdot \left(R_2 + C \cdot P_k \cdot C^T\right)^{-1} \qquad (4.3)$$

and P_k is obtained as the positive semidefined solution of the stationary Riccati equation, see [11]:

$$P_{k+1} = A \cdot P_k \cdot A^T + R_1 - A \cdot P_k \cdot C^T \cdot \left(R_2 + C \cdot P_k \cdot C^T\right)^{-1} \cdot C \cdot P_k \cdot A^T$$
$$P_0 = R_0 = E\left\{x_0 \cdot x_0^T\right\} \quad (4.4)$$

We can see that the recursive equations for the calculation at each step of the Kalman gain K_k are of no use, since the variance, correlation and autocorrelation properties of the noise are unknown. Then we should extract simpler equations for our system. The first step would be to identify a good model for our system. When we identify our system, we do not have input so this is put to zero. Therefore our system does not have neither B nor D.
Our error of prediction then is $y_k - C \cdot \hat{x}_{k|k-1}$ and equations are

$$\hat{x}_{k|k-1} = A \cdot \hat{x}_{k-1|k-2} + K_k \cdot \left(y_{k-1} - C \cdot \hat{x}_{k-1|k-2}\right) \quad (4.5)$$
$$\hat{y}_k = C \cdot \hat{x}_{k|k-1}$$

In our experiment, if we do the identification how we have said previously, the state-space models are of second order and the matrices estimated are:

$$A_1 = \begin{pmatrix} 1.003 & 0.0253 \\ 0.0008 & 0.9656 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0.9997 & 0.0010 \\ -0.0019 & 1.0007 \end{pmatrix}$$
$$C_1 = \begin{pmatrix} -0.2256 & 0.3282 \\ -0.2718 & -0.2267 \end{pmatrix} \qquad C_2 = \begin{pmatrix} -0.2386 & -0.2613 \\ -0.2614 & 0.2369 \end{pmatrix} \quad (4.6)$$
$$K_1 = \begin{pmatrix} -1.4780 & -2.7717 \\ 1.7895 & -0.2512 \end{pmatrix} \qquad K_2 = \begin{pmatrix} -1.5931 & -2.1297 \\ -1.4366 & 0.8528 \end{pmatrix}$$

and the initial values are

$$x_0^1 = \begin{pmatrix} -480.0909 \\ -159.6556 \end{pmatrix} \qquad x_0^2 = \begin{pmatrix} -711.6168 \\ -282.5228 \end{pmatrix} \quad (4.7)$$

5. Results

5.1 Calibration of the cameras

The size of the calibration object is approximately 20 x 20 cm and the object used is a checkered pattern with 49 corner features. The resulting output from the calibration of the stereo system are the intrinsic camera parameter matrices K_1 and K_2, transformation matrix between camera 1 and the robot base $T_b^{c_1}$, relative pose matrix of one camera respect to other one $T_{c_2}^{c_1}$, and we also get the relative pose of the calibration object relative to the gripper frame T_g^t.

The results from the calibration used in our experiment are

$$K_1 = \begin{pmatrix} 483.5674 & -0.0519 & 199.8091 \\ 0 & 483.6177 & 200.6856 \\ 0 & 0 & 1 \end{pmatrix}$$

$$K_2 = \begin{pmatrix} 482.3813 & -0.0650 & 200.4702 \\ 0 & 482.4237 & 200.4365 \\ 0 & 0 & 1 \end{pmatrix}$$

$$T_{c_2}^{c_1} = \begin{pmatrix} 0.9052 & -0.0001 & -0.4249 & 0.7938 \\ 0.0000 & 1.0000 & -0.0003 & -0.0003 \\ 0.4249 & 0.0002 & 0.9052 & 0.1022 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_b^{c_1} = \begin{pmatrix} -0.6365 & -0.7711 & 0.0144 & 1.2369 \\ -0.0078 & -0.0123 & -0.9999 & 1.1648 \\ 0.7712 & -0.6366 & 0.0018 & 1.8304 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_g^t = \begin{pmatrix} 0.0001 & 0.0000 & 1.0000 & 0.1471 \\ 1.0000 & -0.0001 & -0.0001 & -0.1487 \\ 0.0001 & 1.0000 & -0.0000 & -0.0100 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

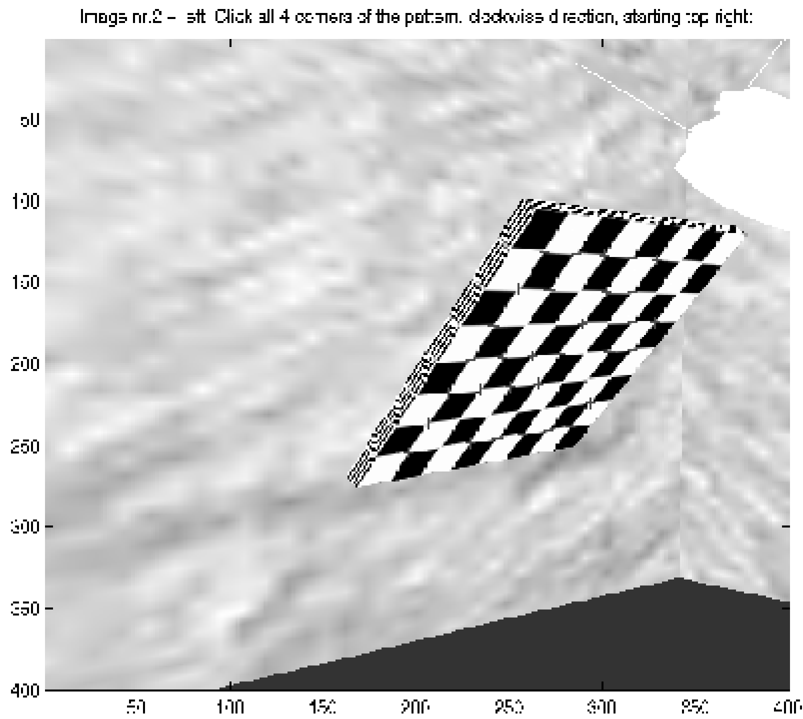


Figure 5.1 Position for calibrating cameras

When we run the calibration program, this program give us the error that can exist between our matrices obtained and the 'real' matrices and it is

$$e = 0.043 \text{pixels}$$

As we can see, this value is very small and therefore, we can say the results obtained are good. Other way to know if our results are really good is to look the matrices and test it. If we look at $T_{c_2}^{c_1}$ for example, we see that the translation matrix is

$$t_{c_2}^{c_1} = \begin{pmatrix} 0.7938 & -0.0003 & 0.1022 \end{pmatrix}$$

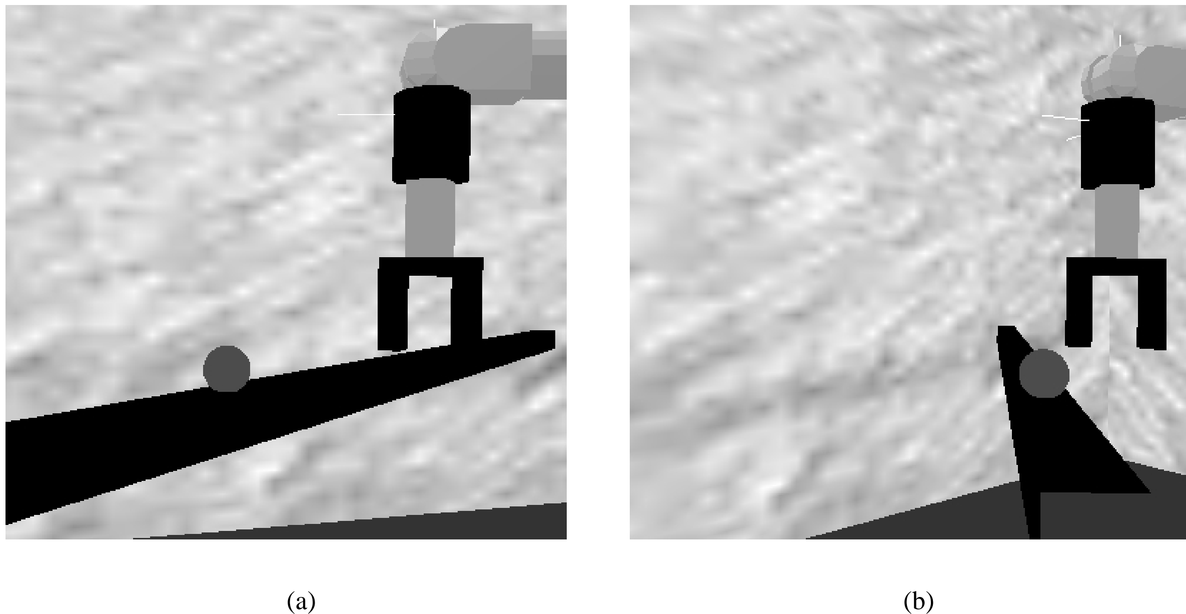
and this value is the distance approximated that exists between two cameras.

In the figure 5.1 we can see an image from camera 1 used to calibrate the camera. This image is binary and clicking all 4 corners of the pattern, the program calculates all corners that exist in the image.

Then, we can affirm the results of the calibration of the cameras are correct.

5.2 Virtual experiment

In virtual software, we should exactly reproduce our real system. Then, we have created irb2000 and irb6 robots, ball, bar where the ball rolls, cameras to watch images, and walls, the ceiling and the floor of the room. The experiment has to be watched on the screen, then we have constructed four frames. The first is to watch the room. The both robots and their environment (ball, bar, cameras, and walls, ceiling and floor) are seen in the fix camera, see figure 3.4.



(a) (b)

Figure 5.2 Irb6 camera 1 (a) and Irb6 camera 2 (b)

Other two frames are created for two cameras in irb6, see appendix B. These cameras are done to watch all the bar, and, of course, the trajectory of the ball and of the gripper. As we have two cameras, we should separate them to be able to see the trajectory of the ball very well. Therefore, the ball and the gripper need to be every time on these screens, see figure 5.2.

Last frame is created for the camera in irb2000. This camera should be in the TCP of the irb2000 to see how the robot tracks and grasps the ball. But it was not possible because when the robot (and of course the camera) is very close to the ball, the camera is not able to see anything. Cameras in Java only see things that are separated a certain distance. Therefore, the camera is mounted above the TCP, see figure 5.3. We can watch the bar and the robot. In theory, if the robot tracks the ball perfectly, we would not have to watch the ball because the robot covers the ball. This is a good way to know if our experiment is all right.

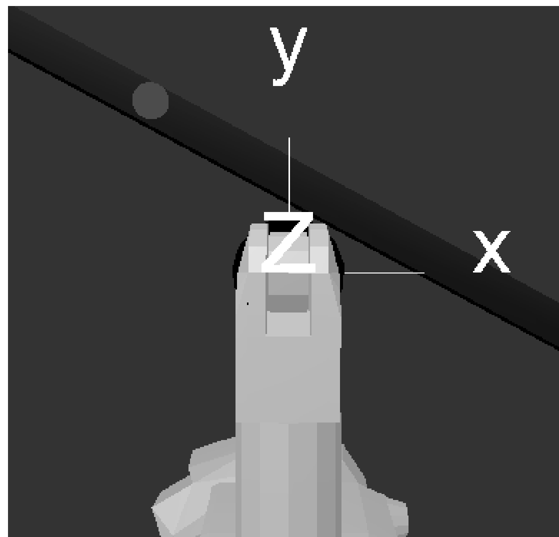


Figure 5.3 Irb2000 camera

5.3 Tracking and grasping the ball

The goal of our experiment is tracking and grasping the ball. We have an initial position of the ball and of the gripper. The first step we should do is tracking the ball.

Tracking the ball

The gripper and the ball will have a certain positions. At the beginning they have an initial position. This initial position was selected in such as a way we are able to see them all the time through cameras. The initial position is shown in the figure 3.4 (fix camera) and 5.2

(cameras in irb6). In the gripper we only mind that the green material is seen because we are going to calculate the tracking with this part.

The position of the gripper will be always like the initial position. The gripper will be always looking at floor, in this direction. In this way, the gripper looks at the ball and the bar in all trajectory, see figure 5.4. Therefore, our goal is that the gripper is above the ball all time. The center point of the gripper should be above the center point of the ball a certain distance. We are talking about this distance lower.

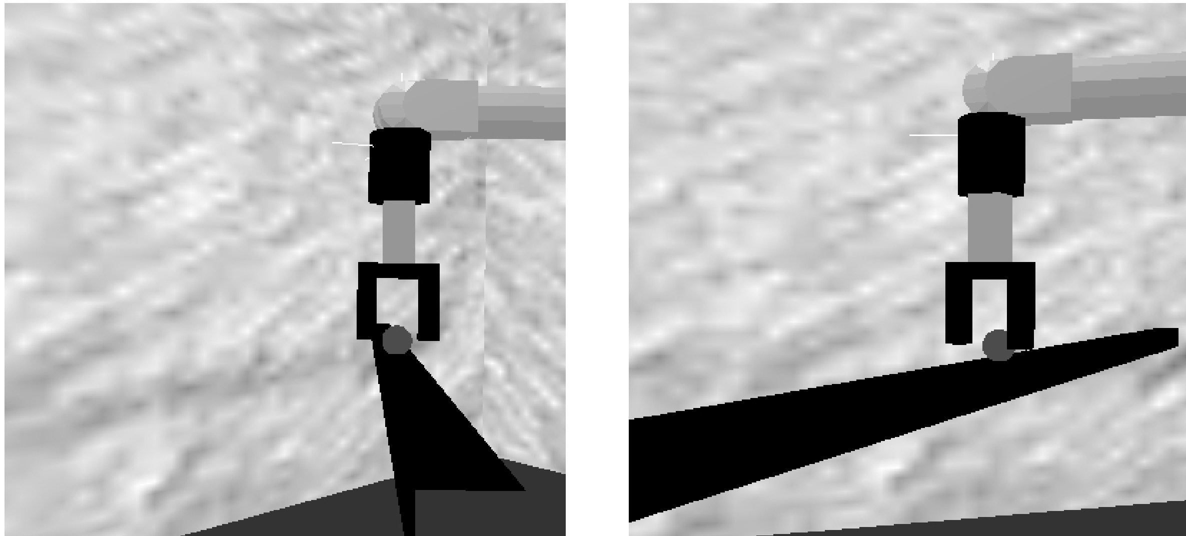


Figure 5.4 Tracking of the ball

Other important result is that the distance between the gripper and the ball in the Y axis is changing with the time, it is a dynamic distance. This is done because the gripper should go over to the ball. The gripper will grasp the ball in a certain time, so the distance that exists between the gripper and the ball should be less at the end than at the beginning. The initial distance between both objects is 108.7 pixels and we decrease this distance in 1.3 pixels each time. In this way, we get that the ball and the gripper are close to each other when the ball is at the end of the bar.

Moreover, if we want to have a static distance the gripper crashes with the bar and that is impossible as in the real system as in the virtual system. The gripper would only crash at the beginning of the bar, and the experiment would not be possible.

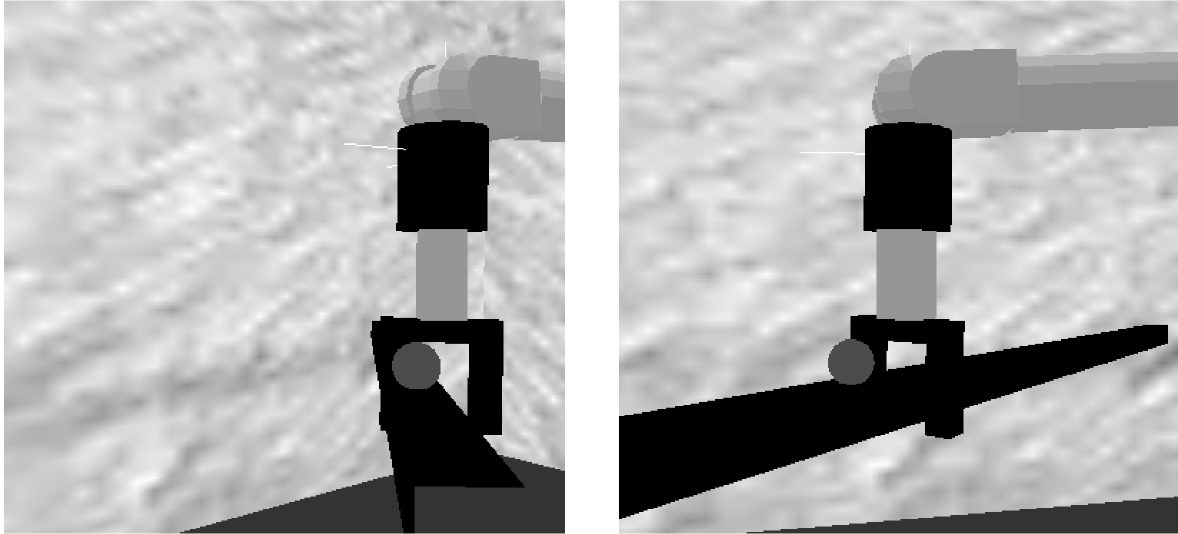


Figure 5.4 Robot crashes with a static distance

To track the ball, we need a controller that can do this. We have used two kinds of controller, P-controller and Image Jacobian controller, and we have compared the results. We can see in chapter 5.3.

Grasping the ball

So far, tracking of the ball is only done. Now, we need to grasp the ball.

Really, we do not know when grasping should be done. We could have two possibilities. A possibility is to do the grasping near the end of our motion and assuming that we are in position to grasp. And the another possibility is grasping the ball when the error between the position of the ball and of the gripper is less than a certain tolerance. The first one can give us problems if the tracking is bad done but we always try to grasp the ball at the same place. The second one should be better because the error has to be as small as we want. The problem is that we can not know where the gripper grasps the ball exactly. We can know the zone more or less, but the exact place not.

We have used the second option because we think it is the best option. We get the exact coordinates of the objects and we calculate the error between them. This error is

compared with a tolerance that we choose. If it is less or equal than our tolerance, we should grasp the ball. If it is not like that, we continue tracking the ball.

As we do not know the exact place where we grasp the ball, we can play with our tolerance until we find the perfect place. That place should be close to the end of the bar. Therefore, we are going to use the trial and error method.

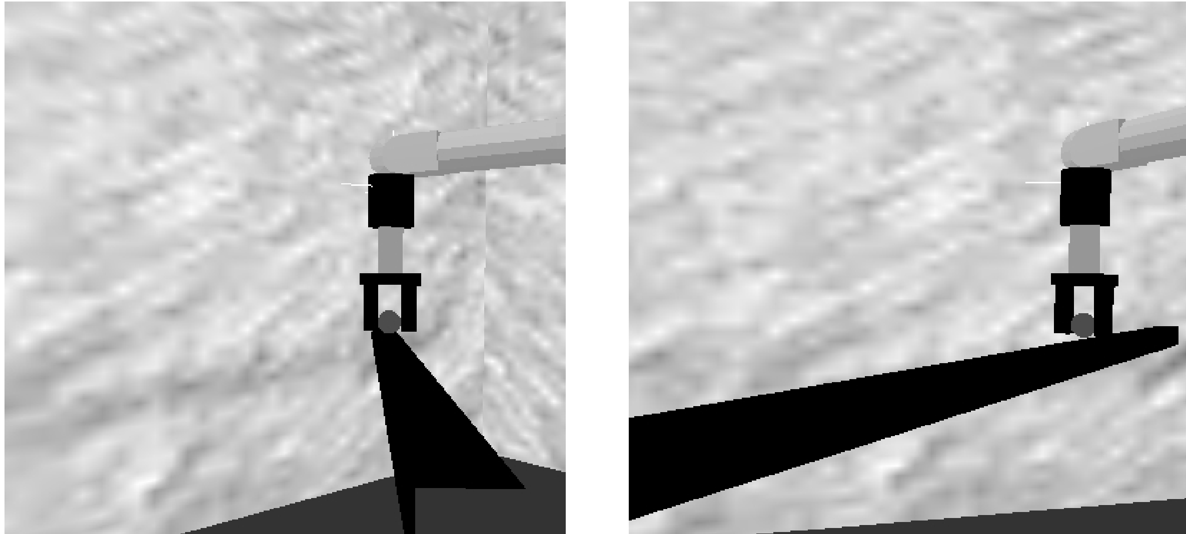


Figure 5.5 Robot grasps the ball

In the figure 5.5 the moment of grasping the ball is shown. In that moment, the robot closes the gripper and the ball is caught. In that instant, we have a little problem. The ball gives a 'jump' to attach to the robot. The problem is in the Java code. We have to remove the ball from the image and later add it to the robot, see figure 5.8 and appendix B.

When the robot grasps the ball, the robot should go back to initial position.

5.4 P-controller and Image Jacobian controller

The goal of the control algorithms is to do the movement of the robot as smooth, stable and fast as possible. The trajectories are calculated with respect to the initial and desired position. The desired position is calculated from feedback from vision.

P-controller

A proportional controller in continuous time is described by:

$$u(t_k) = k \cdot e(t_k) \tag{5.1}$$

where *k* is the proportional gain of the controller. We are going to need three gains because we have three different controllers, one for each Cartesian coordinate. The properties of the controllers differ from coordinate to coordinate.

The proportional controller is a function of the error, which in our case is the difference between the current position of the ball and the current position of the Virtual IRB2000 TCP.

Therefore, as we are dealing with a discrete time controller, we have

$$\begin{aligned} u_x(k \cdot h) &= K_x \cdot (x_{ball}(k \cdot h) - x_{TCP}(k \cdot h)) \\ u_y(k \cdot h) &= K_y \cdot (y_{ball}(k \cdot h) - y_{TCP}(k \cdot h)) \\ u_z(k \cdot h) &= K_z \cdot (z_{ball}(k \cdot h) - z_{TCP}(k \cdot h)) \end{aligned} \tag{5.2}$$

Image Jacobian controller

The image jacobian method is shown in 2.1. Anyway, let us see in what it consists.

In an image-based visual servo, the control error is defined as

$$e = y_r - y \tag{5.3}$$

where y_r is the image of the ball and y is the image of the gripper. We should drive this error to zero. In the heading 2.2, we can see how the image jacobian is. Then, we know the control law will be

$$e = J_v(r) \cdot \dot{r} \tag{5.4}$$

where $\dot{r}$ is the corresponding velocity screw in the task space. Therefore, we know the error e and can know the jacobian with the equation (2.12), so we can calculate the velocity screw as

$$\dot{r} = J_v^{-1}(r) \cdot e \tag{5.5}$$

Comparation between P and Image Jacobian controllers

The proportional controller was tuned using the trial and error method. The values selected from this procedure for the gains were:

$$\begin{aligned} K_x &= -0.6 \cdot 2 \\ K_y &= -0.6 \cdot 3 \\ K_z &= 1.5 \cdot 2 \end{aligned} \qquad (5.6)$$

We are going to compare the error between the both controllers. The actual errors for the P and Jacobian controller are shown in figure 5.6 and 5.7 respectively.

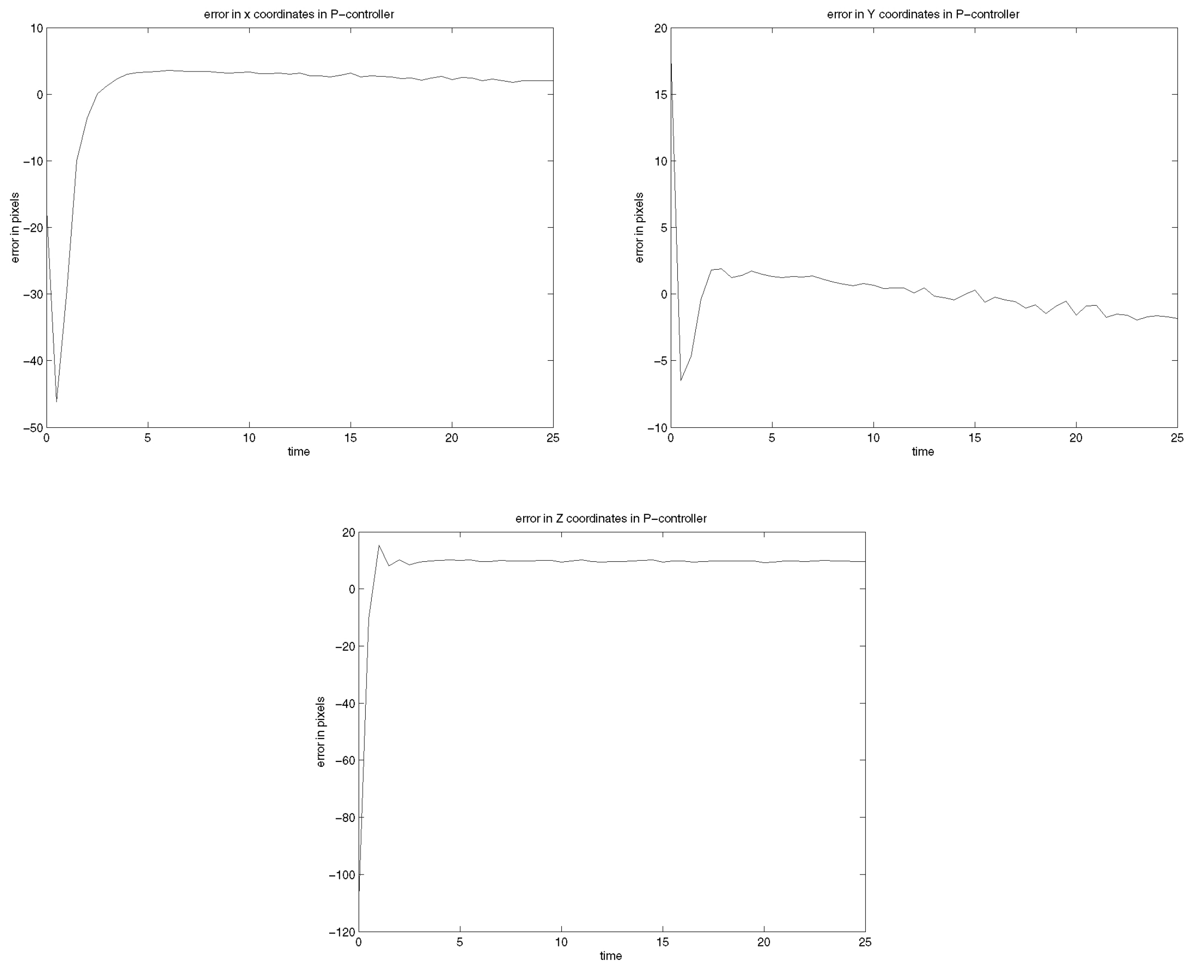


Figure 5.6 Errors of the P-controller

If we look at these figures, P-controller errors are bigger than Jacobian controller. The error in X direction is better in Jacobian controller because it is driven to zero while in P-controller it is not zero.

In Y direction, we must say that we are changing the reference value in 1.3 pixels because the gripper should grasp the ball, so the distance between the gripper and the ball should be less. In this direction, we see the error in IJ controller is much better although the error is not very good.

In Z direction, we can see that in P-controller we have a systematic error. The error in Z direction is calculated as the difference between the error in X direction in camera one and the error in X direction in camera 2. This systematic error is because of the angles of cameras are different, then the distance between the ball and the gripper in X direction is not the same. In IJ-controller we see two peaks in the trajectory. This is because of the occlusion of the ball.

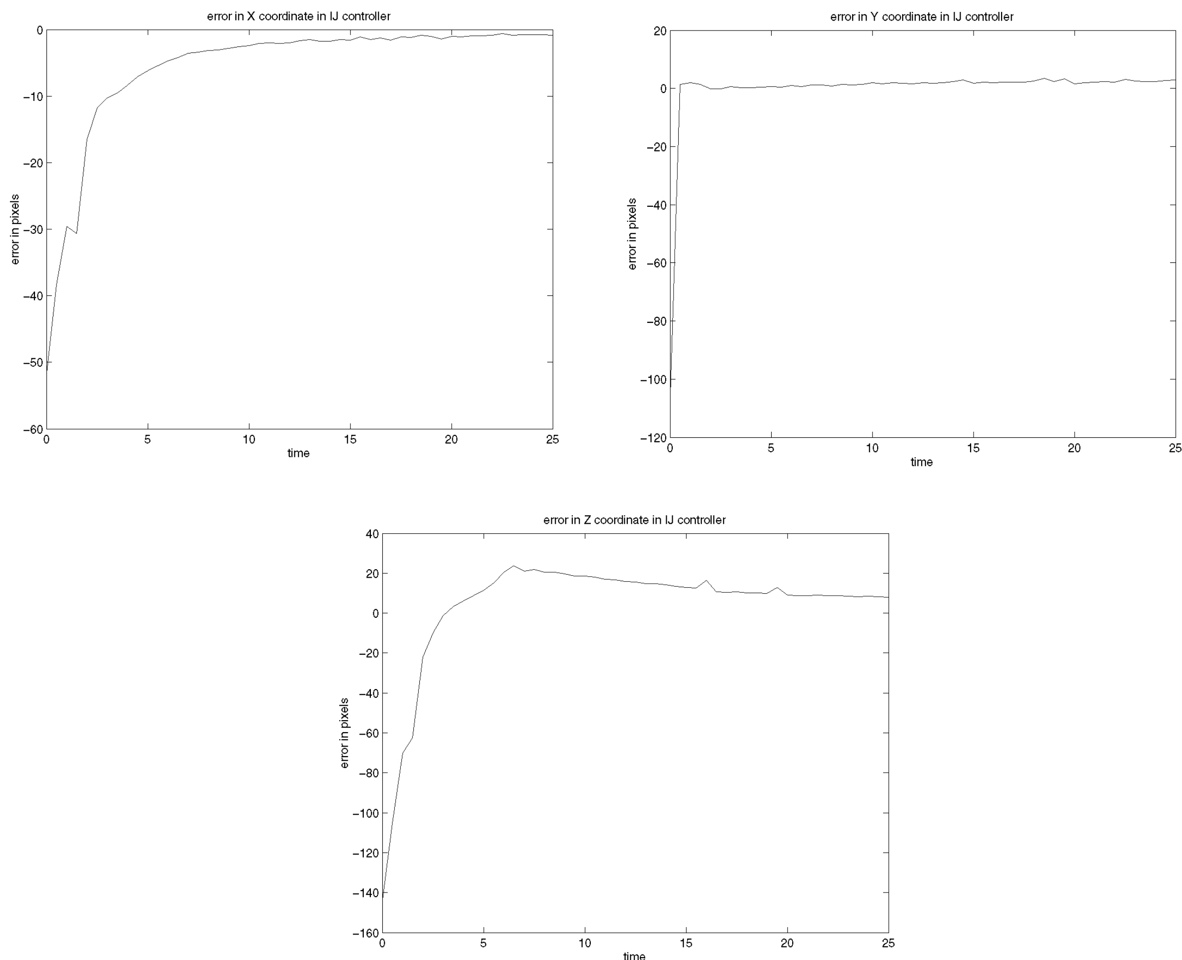


Figure 5.7 Errors of the Image Jacobian controller

When the robot tracks the ball, we see the robot has a little delay. It means the robot can not grasp the ball. We should improve this controller. In 5.4 we will see that we are going to use a velocity of reference as feedforward.

With these results, we can say the Jacobian controller is better than P-controller.

5.5 Image Jacobian control with feedforward

We want to eliminate this error, so we have to add a new term. We want the gripper tracks the ball faster than before. So, the velocity of the ball and of the gripper should be the same.

We are going to add one term to our control law of Image Jacobian controller.

Now, we have other more feedback but in this case it is with the velocity of the ball. We add the velocity of reference to the error. The scheme of this is shown in the figure 5.8.

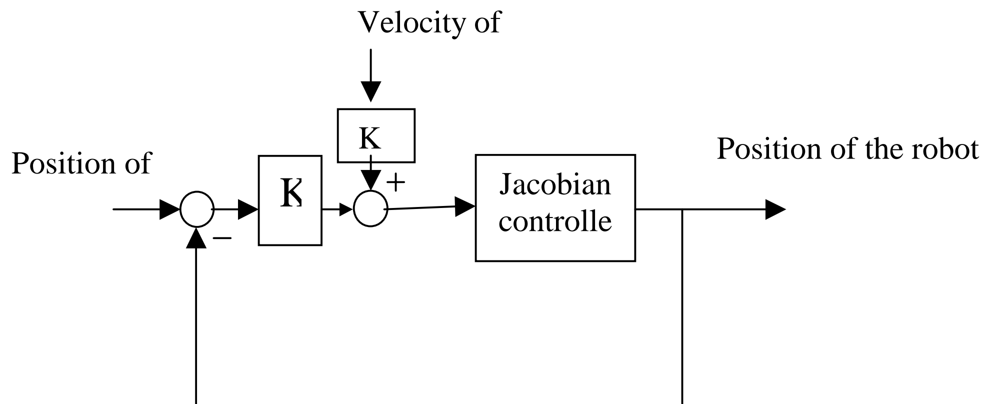


Figure 5.8 Feedback control with the reference velocity

The control law will be

$$K \cdot e + K_v \cdot v_{ref} = J_v(r) \cdot \dot{r} \tag{5.7}$$

Therefore, we know the error e and the reference velocity, then we can calculate the velocity screw.

$$\dot{r} = J_v^{-1}(r) \cdot \left(K \cdot e + K_v \cdot v_{ref}\right) \tag{5.8}$$

As we can see, we now use two gains to improve the system. In our experiment we have worked with different gains but the best results are in figures 5.9.
This controller was tuned using the trial and error method. The values selected from this procedure for the gains were:

$$K = 2 \qquad K_v = 0.02 \tag{5.9}$$

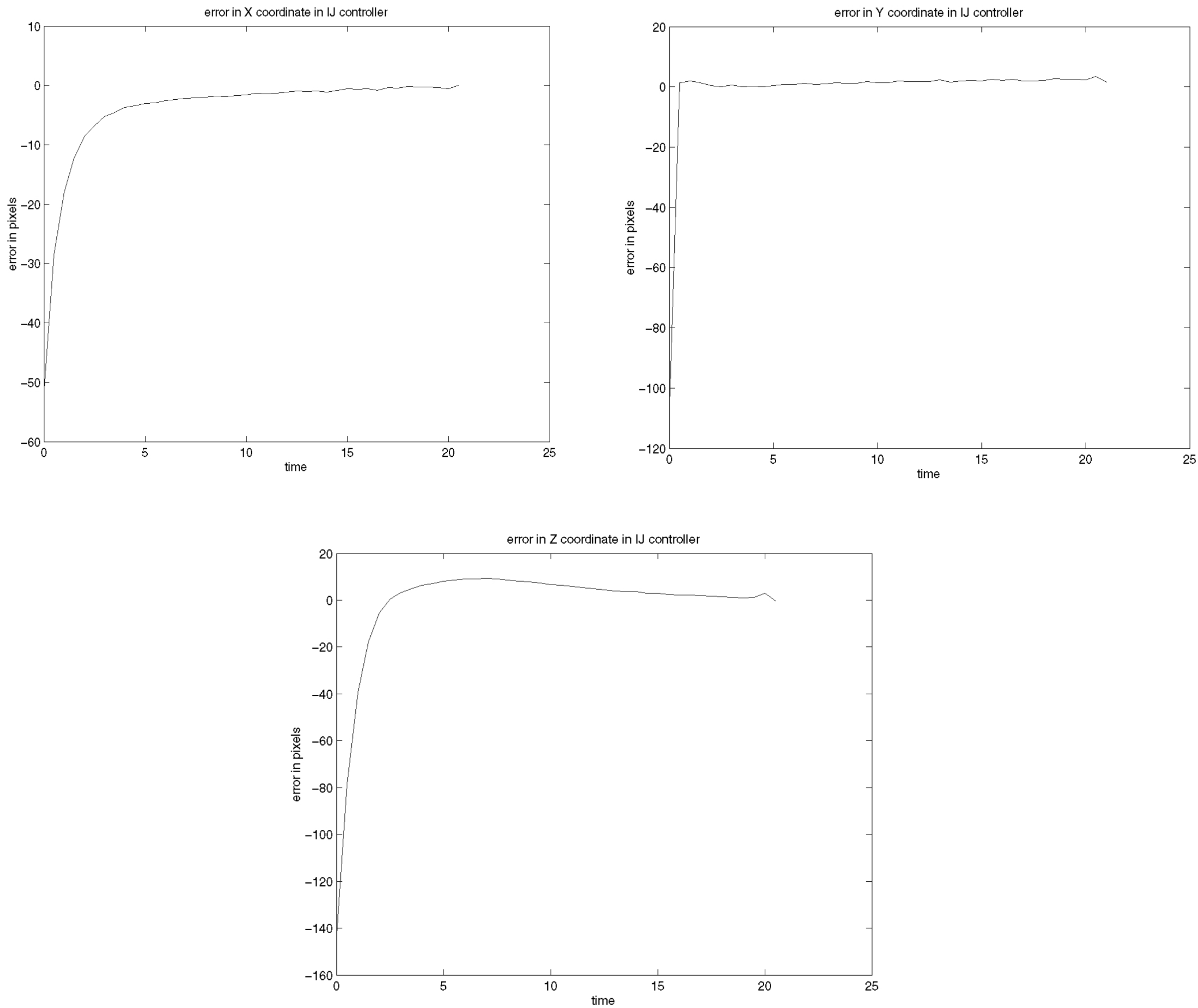


Figure 5.9 Errors in the Jacobian controller with reference velocity and K = 2 and $K_v = 0.02$

We can see in figures that the error in X direction is quite same. In the Z coordinate, we see that in the figure 5.7 we had two peaks, but now we do not have these peaks, we do not have occlusions, so in this direction we have improved our controller. If we look in the Y direction, it is also better because the error is driven to zero.

The reference value is also changed like in the last experiment. We can say our results are very good in all axis.

When we are in time 20th approximately, we are going to grasp the ball. The ball is grasped when the error between the ball and the gripper is less than one tolerance we put.

5.6 Different velocities of the ball

This controller works all right for a determined velocity of the ball, but we can see what occur if we change the velocity of the ball. We are going to work with the controller seen in 5.4.

The previous experiments worked with a velocity increment of the ball $\Delta u = 0.32\, m/s$.

We are going to do several experiments increasing the velocity of the ball. The robot will now have to move faster to track the ball. The gains of the controller should be changed if we want to obtain good results.

The following figure shows us the value of the gains for each velocity increment.

Δ**u**	**k**	**Kv**
0.64	3	0.036
0.96	4.5	0.044

Figure 5.10 Gains and ball velocities of the image jacobian controller with reference velocity

Other important thing is that the sampling rate used for this system is 10Hz. The sampling rate of the PC computer is 15Hz but we have used 10Hz for simplification.

In the next figures, we see the results obtained with these gains and velocities shown in the figures 5.11, and 5.12.

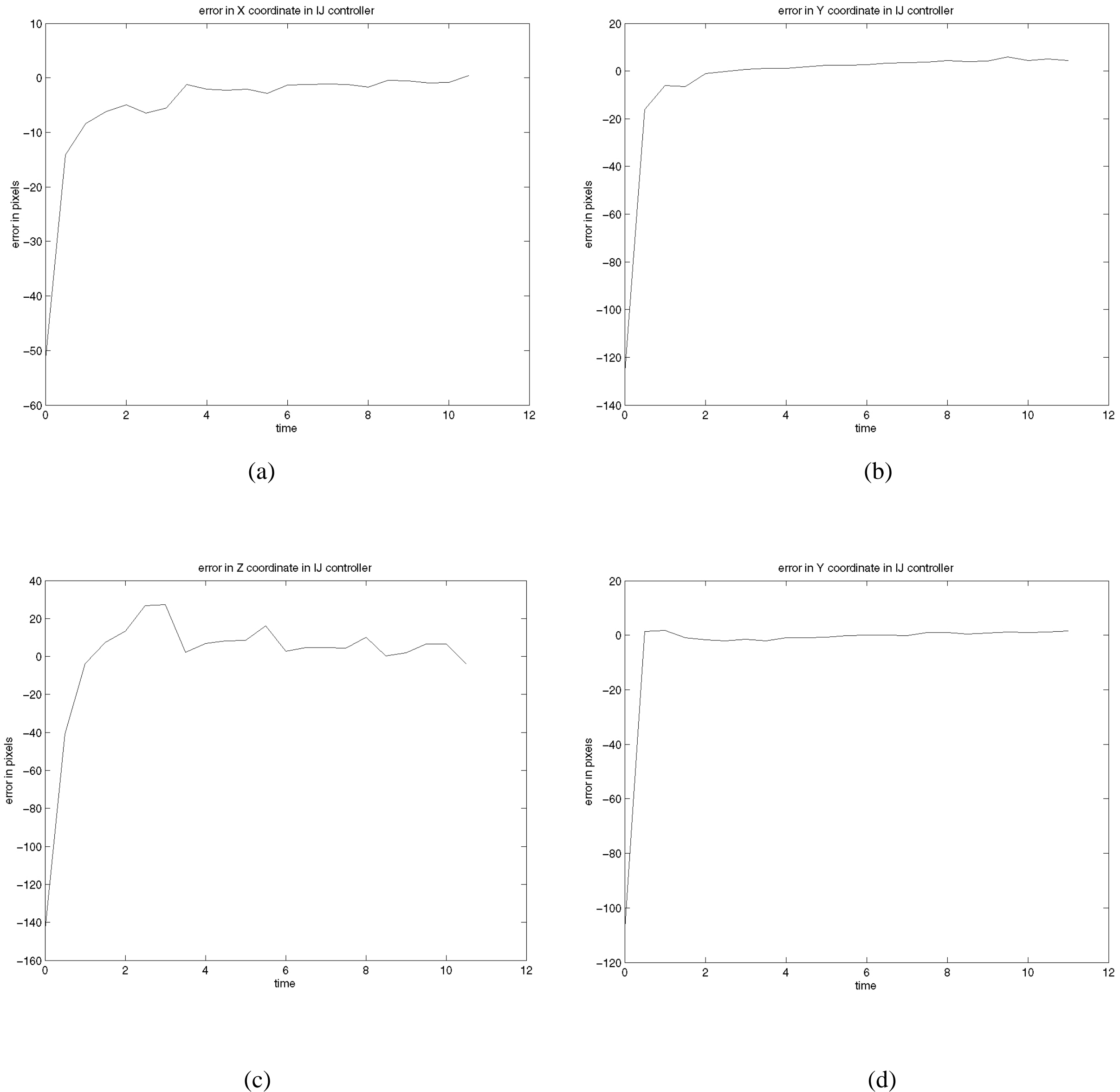


Figure 5.11 IJ-controller with reference velocity with k = 3 and Kv = 0.036

(a), (b), (c) for $\Delta ref = 2.6$ and (d) for $\Delta ref = 1.3$

We can see that the results are worse with high velocities than small velocities. If we increase the velocity, results are worse. In the figure 5.11, we see that in the direction X, the error is almost zero in the time 9th. Before, we get to have error zero in second time. In the Z direction, the result is quite bad because the error is too big and we have three peaks, so it means we have 3 occlusions. And in Y direction, our reference value is 2.6 and we see the error is not zero, so the controller is not very good. But if we have a reference value as 1.3, we see in the figure that the error is almost zero and then, the results are very good. The problem

is that with 1.3 we can not grasp the ball because the distance between the ball and the gripper is very long.

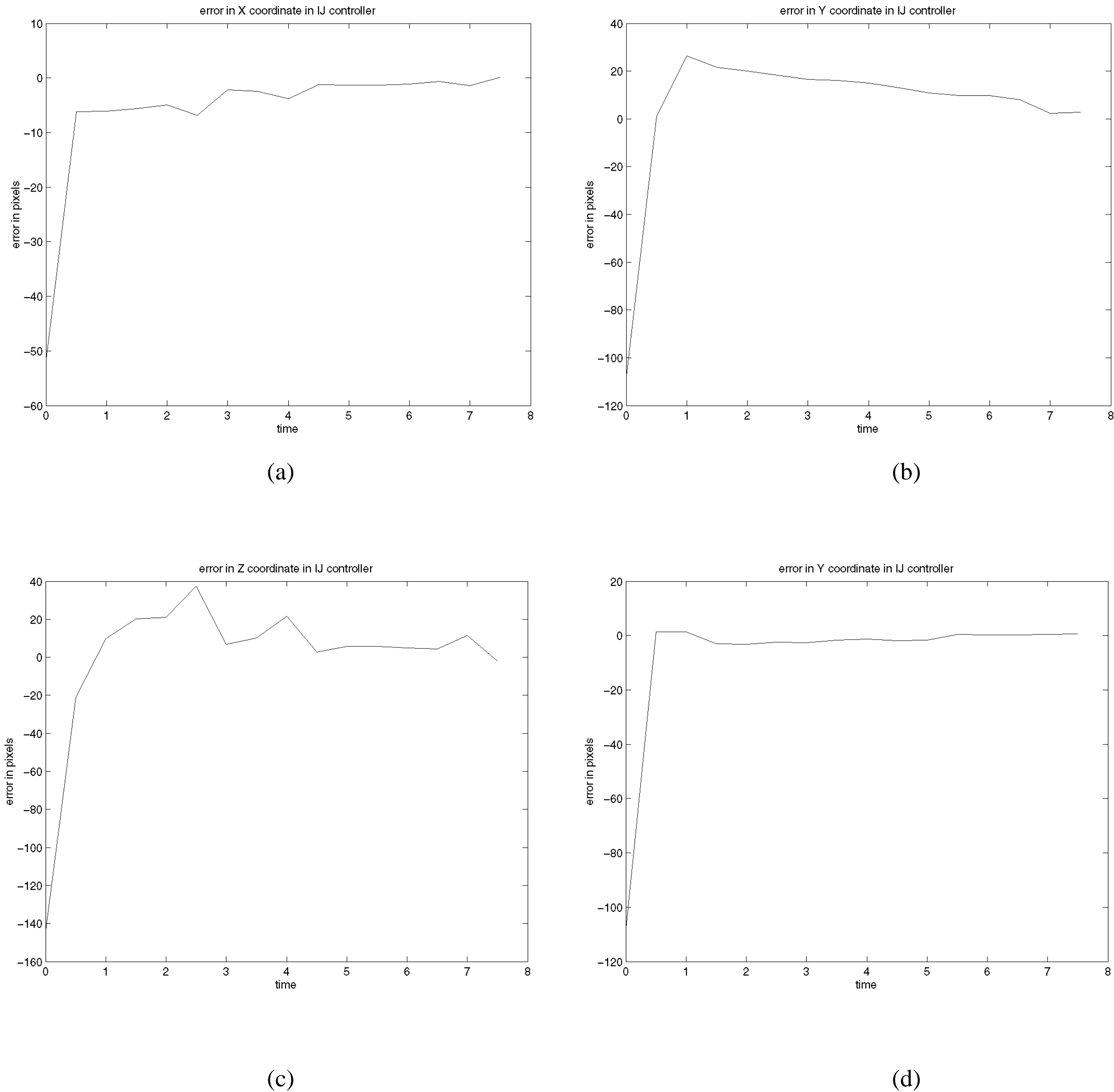


Figure 5.12 IJ-controller with reference velocity with k = 4.5 and Kv = 0.044

(a), (b), (c) for Δref = 3.2 and (d) for Δref = 1.3

In the figure 5.12, we see that in the X coordinate the result is worse than last experiment. In the Z coordinate, the occlusions are also bigger but the error is zero at the end. In the Y axis, we have one big peak in the first time and the error is zero at the 7th time. It is slow. This result is with reference value of 3.2 but if we put 1.3, we see that the error is almost zero and the results are good. But it is impossible to grasp the ball with this value.

Therefore, this last controller is not very good. We can affirm that as bigger velocity, we have worse controller.

5.7 Hybrid controller

Our adaptive controller is based on the cost function

$$J_i(k+d) = E\left\{\left[y_i(k+d) - y_i^*(k+d)\right]^2 + \rho_i \Delta u_{ci}^2(k) \middle| F_k^i\right\} \qquad (5.11)$$

So, using a modified one-step-ahead predictor, see [16], the adaptive control law is

$$\Delta u_{ci}(k) = -f_i^{'T}(k) \cdot q_i^{'}(k) \qquad (5.12)$$

where

$$f_i^{'T}(k) = \left(y_i(k), y_i(k-1), \Delta u_{ci}(k-1), \Delta u_{ci}(k-2), y_i^*(k+1), y_i^*(k), y_i^*(k-1)\right)$$

and $q_i^{'}(k)$ is the vector that contains the coefficients of the polynomials

$$\frac{\beta_{i0}}{\beta_{i0}+\rho_i}\left\{A_i^{opt}\left(q^{-1}\right), \frac{\rho_i}{\beta_{i0}}\left[\Gamma_i\left(q^{-1}\right)-1\right]\cdot q, -\Gamma_i\left(q^{-1}\right)\right\} \qquad (5.13)$$

The next step in our algoritm is the computation of Tx(k), Ty(k), and Tz(k). From adaptive controller, the $u_{ci}(k)$ is either the Sx(k) signal or the Sy(k) signal.

$$H(k)T(k) = S(k) \qquad (5.14)$$

where the matrix H(k) is

$$H(k) = \begin{bmatrix} -1 & 0 & x^{(1)}(k) \\ 0 & -1 & y^{(1)}(k) \\ -1 & 0 & x^{(2)}(k) \\ 0 & -1 & y^{(2)}(k) \\ -1 & 0 & x^{(3)}(k) \\ 0 & -1 & y^{(3)}(k) \end{bmatrix} \tag{5.15}$$

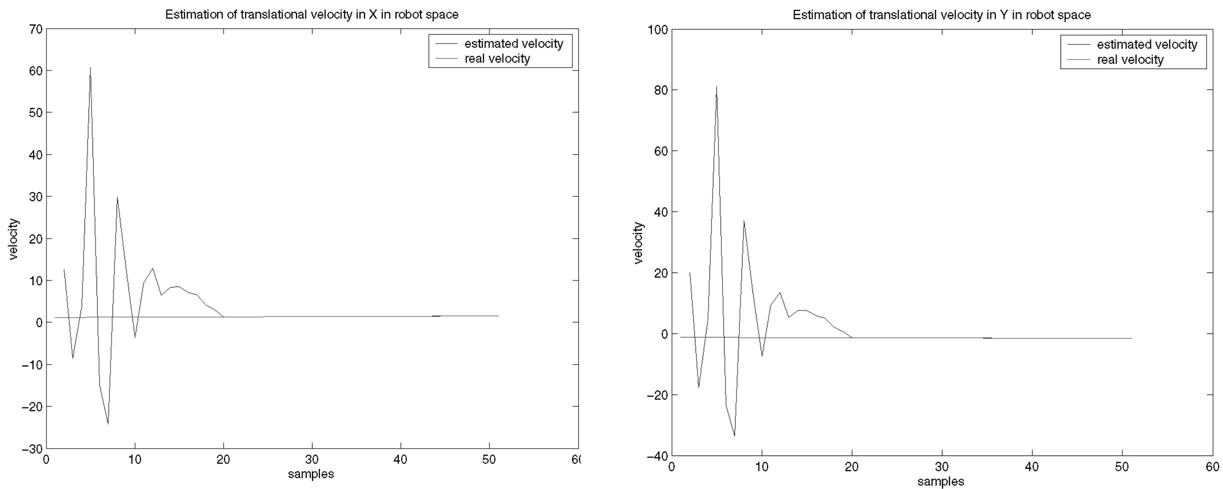
Then, the translational velocity vector T(k) will be

$$T(k) = \left(H^T(k) \cdot H(k)\right)^{-1} H^T(k) \cdot S(k) \tag{5.16}$$

As we can see, we have three feature points which are the center point of the ball, and the two points of the diameter of the ball.

In the real system, the ball has acceleration, so we should use it in our experiment. The velocity in the end of the bar is different than the one in the beginning of the bar. Now, the velocity is not constant, so the adaptive controller should give us the velocity estimated at each time. The robot tracks the ball when the velocity is estimated very well.

The results of the estimation are shown in the following figures.



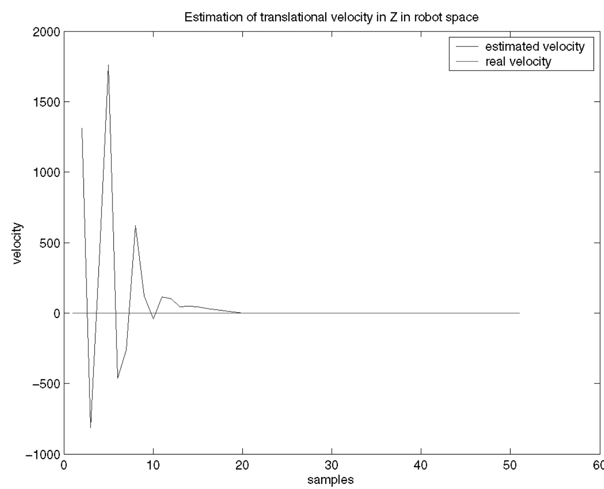


Figure 5.13 Estimation of translational velocity in robot space

We can see these figures better if we do a zoom of them.

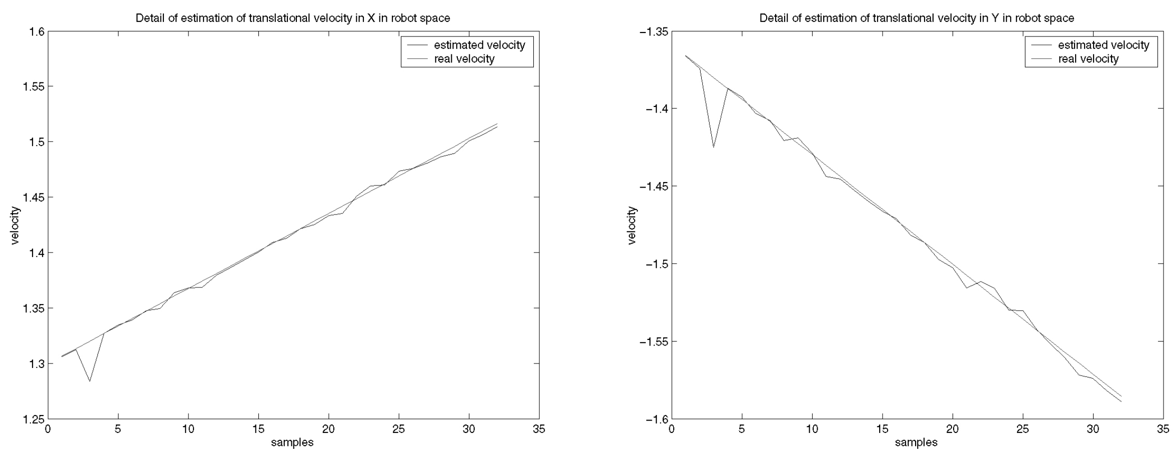


Figure 5.14 Zoom of the figure 5.13 from sample 20th to the end

We can see the estimation is very good from the sample 24th to the end. Therefore, while the velocity estimated is not stable, we can not track the ball.

In the Z direction (in Cartesian space), the velocity is very small because the bar is little leaning (the angle of the bar with the axis X is 0.05 radians). Therefore, the velocity estimated is the same as the real velocity.

If we see these velocities in image space, the movement is now in the Z direction and we can see in the figures that the velocity is not constant. In the other directions, the velocity is very small, see figure 5.15.

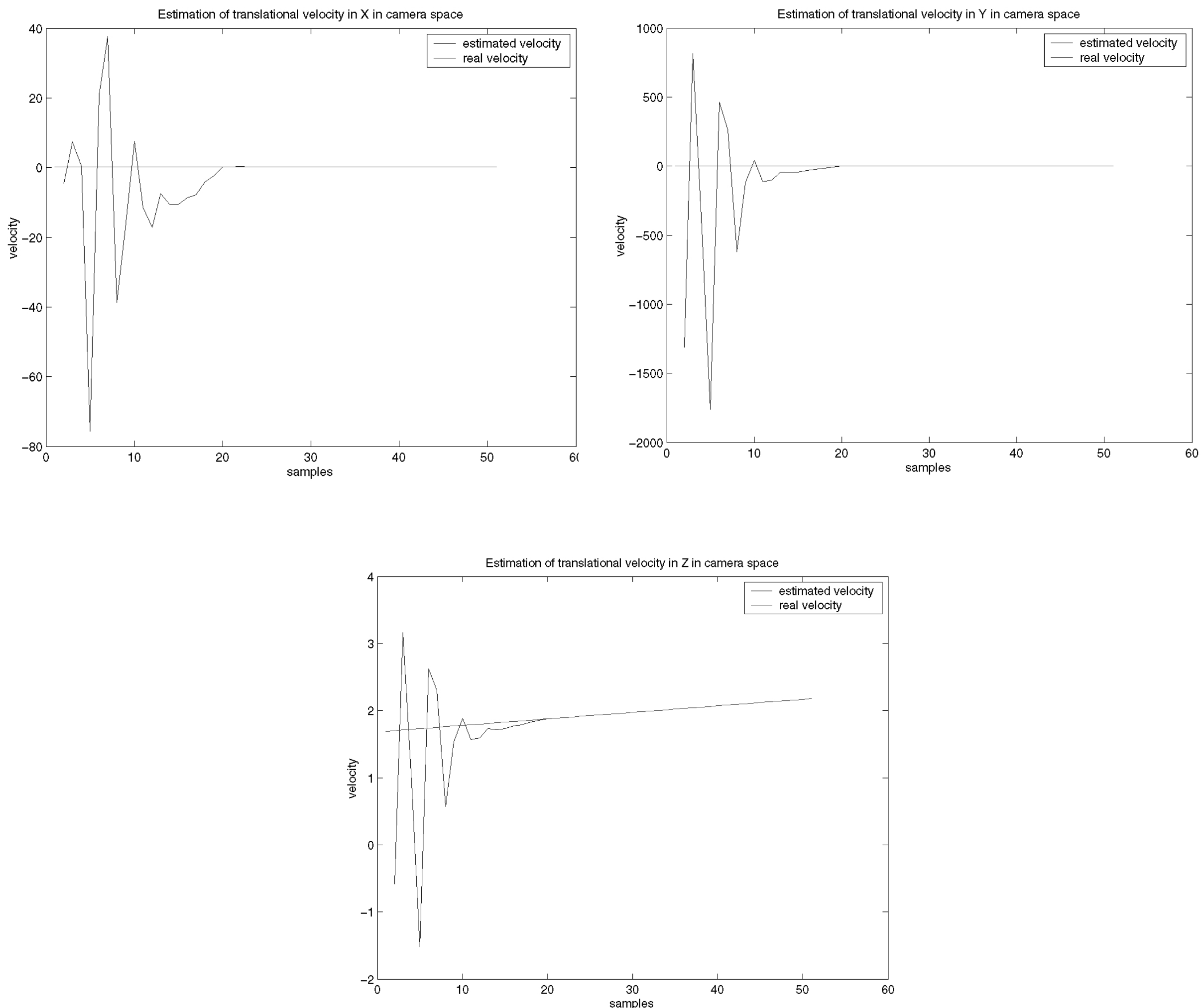


Figure 5.15 Estimation of translational velocity in camera space

We can see that the estimation is perfect from the sample 24th more or less. In this frame, the velocities in X and Y direction are almost zero because the rolling ball is on the bar and the latter one is in the Z direction. The most important estimation, in this frame, is the Z value.

In the figure 5.16, the adaptive controller law is shown. It is stable since the 24th sample.

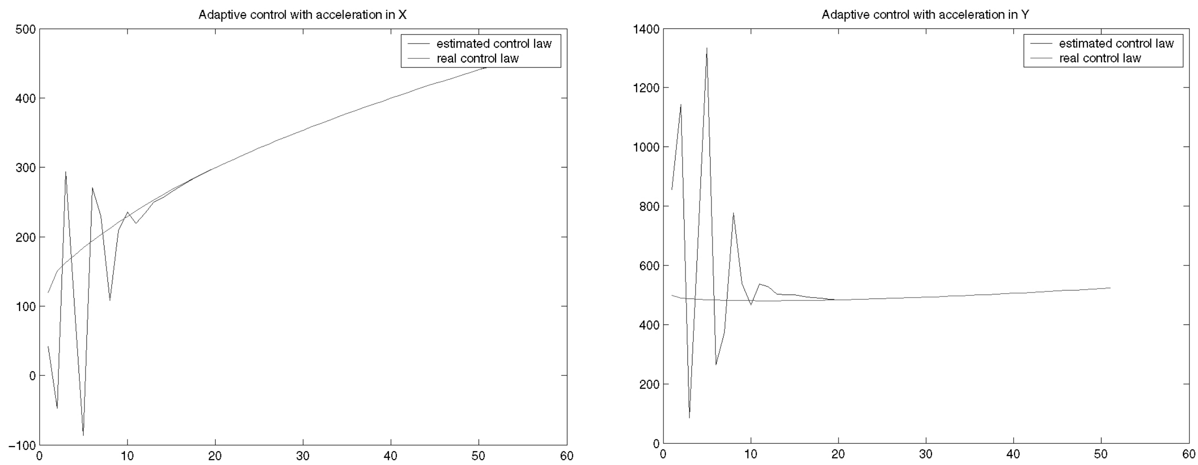
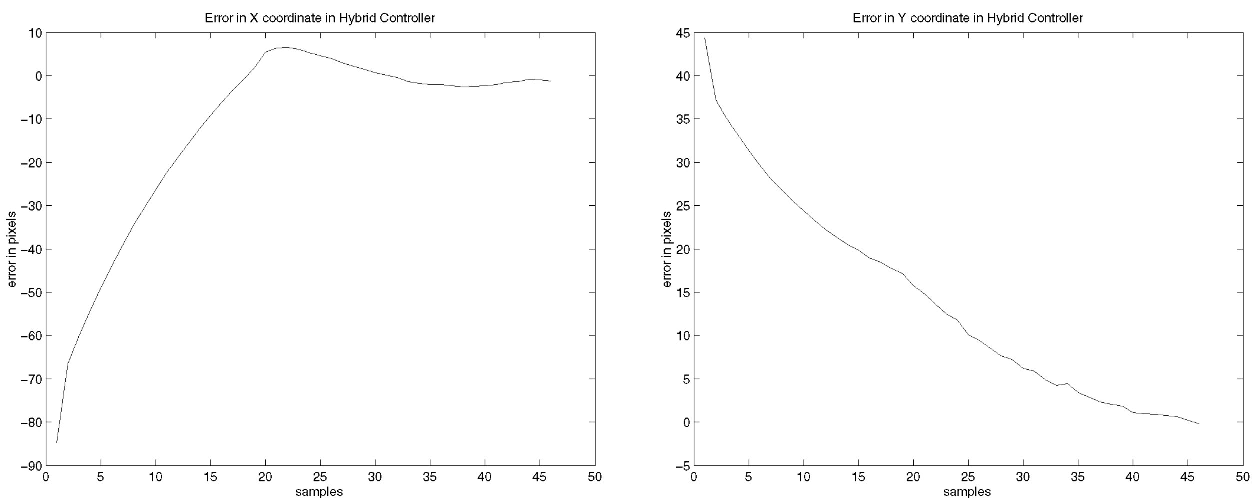


Figure 5.16 Adaptive controller with accelerated ball

Hybrid controller

We have said that the robot tracks the ball when the estimation of ball velocity is stable. The robot is put in one position such that when the ball is below the robot, this one starts tracking the ball. This position is when the velocity is stable.

We use the image-Jacobian controller to control the robot in X and Y direction (in robot frame) and the PI-controller in Z direction (in robot frame).



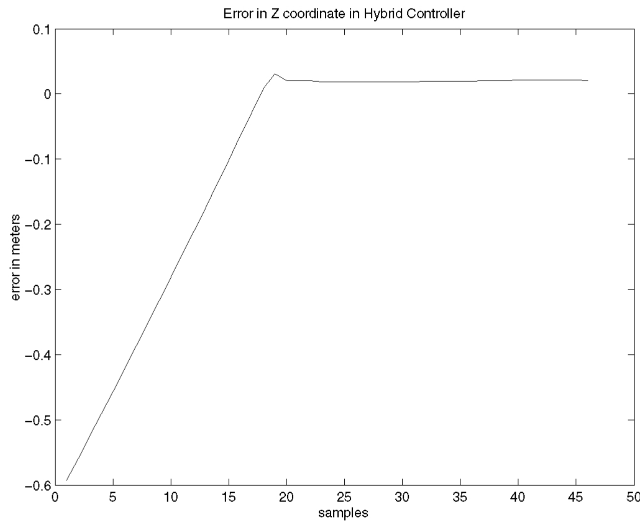


Figure 5.17 Error in the tracking of the ball

We see that in the X coordinate, the error is driven to zero, but there is a small error. But this error is almost zero, so the robot tracks the ball well. In the Y direction, the error goes to zero as well, but if we compare these results with the one in Jacobian controller, see figure 5.9, we can say that these results are worse than the others. In the Z direction, the error is also worse in this controller. It is due to we only use one camera (monovision) and in the other experiment we use two cameras (stereovision). If we use two cameras with this controller, the result would be much better.

6. Discussion

6.1 Virtual experiment

Slowness of the virtual experiment

When we run the program in Java, we can see it is very slow. We would need to know why this slowness exists. The program that we have done has a lot of functions and we are going to measure the time that these functions spend. With these results, we can know where the error is. Functions time is given in milliseconds in the figure 6.1 and we use the method System.currentTimeMillis() to know this. In this figure, the most representative functions in time are only shown.

	Time
Get images from camera1	931
- Snapshot	701
- Get data	190
Thresholding	50
Calculate center point of objects	80

Figure 6.1 Java Functions time in ms

As we can see in the figure 6.1, the biggest time is when we get images from the camera. But we have two cameras, so that time must be multiplied by two. We also see the problem is when we get the snapshot to the image. That time is very big, so we can look for the problem in the function snapshot. It is shown in the figure 6.2. We can also see that the tresholding and the center point of the ball and the gripper are calculated in a few milliseconds.

We see the method waitForOffScreenRendering() spends all the time of snapshot. But that method is an intern function of Java and we can not do anything. If we remove this method, the snapshot is not taken and the image would not be recorded.

	Time
SetOffScreenBuffer()	10
RenderOffScreenBuffer()	0
WaitForOffScreenRendering()	681

Figure 6.2 Snapshot functions time in ms

Jumps of the ball

The ball gives a ‘jump’ when the virtual robot grasps the ball, as we have said sometimes. This is because if we want that the ball is moved with the robot, the ball must be removed from the room and later, the ball is added to the robot. This is explained later in more detail.

6.2 Visual servoing

We have done several experiments with different velocities of the ball. We have increased the velocity in two and three times. We can say that the maximum velocity for grasping the ball with the current sampling rate is almost 1m/s. This velocity is very high and it is difficult to grasp the ball, but we got it. If we increase the velocity, the error is too big and it is impossible to do the experiment.

7. Conclusions and future work

7.1 Conclusions

The subject of this thesis has been the combination of image-based and pose-based to visual control servoing. We have used the image Jacobian control to do it. We have also compared this controller with P-controller and we saw Jacobian controller is better than P-controller. The hybrid controller should be better but we only use one camera, so we have seen this controller is worse.

When the robot tracks the ball, the J-controller is not enough because the robot is slower than the ball and the error in the depth z is bigger with the time. We have to add a reference velocity to error to eliminate this last one as far as possible. We have seen this new situation is very best. The robot is capable of tracking and, later, grasping the ball testing the error between the position of the ball and the position of the gripper.

Java3D is investigated as an image generation tool for use in simulation of visual servoing loops. But it produces some limitations, for example the execution time or the jump of the ball.

The image processing is done in Java and is simplified using a ball red and a green gripper because we have to send this information to Matlab. This work is done in virtual system but if we want to transfer to real system we should solve some problems like for example the noise in the image of cameras. We should add a Kalman filter and predict the trajectory of the ball.

The calibrated cameras are used for depth estimation in the image Jacobian experiments, but later in hybrid control we use uncalibrated cameras with adaptive control techniques.

The camera calibration method uses a number of images of a planar calibration object to estimate both the intrinsic camera parameters and the location of the camera frames relative to the robot. The calibration object is attached to the robot end-effector, and therefore the extrinsic camera parameters are not completely unknown. This extra information is used in the algorithm, in order to increase the robustness.

7.2 Future work

The first future work is to implement these experiments in real system.

In our experiment, the system in Java is very slow. So, future work should be done in Java implementation. Future work could do that the system is faster in execution and real time. It could try to not use the method waiForScreenRendering() and search other method that is able to do the same in less time.

Future work should also be done in control algorithms. To throw the ball could be tried. Now, the ball is rolling. If the ball is thrown, the trajectory is not the same and the velocity is bigger. The trajectory would not be a straight line, but a parabola. It would be much more difficult and other technique should be used.

In the current experiment, the gripper is vertical with respect to the ground. However, the human motion for a similar movement would be having our hand in a certain angle with respect to the ground. This should give thoughts in how the actual grasping of the ball should be performed by the industrial robot.

8. References

[1] Emanuele Trucco and Alessandro Verri. 'Introductory techniques for 3-D computer vision', Prentice Hall, 1998.

[2] Peter I. Corke. 'Visual Control of Robots: high-performance visual servoing', Research Studies Press Ltd, 1996.

[3] Gregory D. Hager and Peter I. Corke. 'A Tutorial on Visual Servo Control', IEEE Transactions on Robotics and Automation, 1996.

[4] Enzio Malis, Francois Chaumette, Sylvie Boudet. 'Multi-cameras visual servoing', IEEE International Conference on Robotics and Automation, 2000.

[5] Peter I. Corke and Seth A. Hutchinson. 'A New Hybrid Image-Based Visual Servo Control Scheme', IEEE Conference on Decision and Control, 2000.

[6] Dennis J. Bouvier. 'Java 3D API Tutorial'.

[7] Michail Bourmpos. 'Vision based robotic grasping tracking of a moving object', Master Thesis, Department of Automatic Control, University of Lund, 2001.

[8] Tomas Olsson. 'Vision-supported Force-controlled Robotic Grasping', Master Thesis, Department of Automatic Control, University of Lund, 2000.

[9] Luis Manuel Conde and Duarte Miguel Horta. 'Computer Vision and Kinematic Sensing in Robotics', Master Thesis, Department of Automatic Control, University of Lund, 2001.

[10] Al Bovik. 'Handbook of Image & Video Processing', Academic Press

[11] Lennart Ljung. 'System identification. Theory for the user', Prentice Hall 1987.

[12] Mathias Haage and Klas Nilsson. 'On the scalability of visualization in manufacturing', IEEE Conference, 1999

[13] Johan Bengtsson and Anders Ahlstrand. 'A Robot Playing Scrabble Using Visual Feedback', Master Thesis, Department of Automatic Control, University of Lund, 1999.

[14] B. Haverkamp and M. Verhagen. 'SMI Toolbox, State Space Model Identification Software for Multivariable Dynamical Systems', Version 1, Delft University of Technology, 1997

[15] Rolf Johansson. ‘System Modeling and Identification’, Department of Automatic Control, University of Lund, 1998.

[16] Graham C. Goodwin and Kwai Sang Sin. ‘Adaptive filtering prediction and control’, Prentice-Hall Inc., 1984.

[17] Karl J. Åstrom and Björn Wittenmark. ‘Adaptive control’, Lund Institute of Technology, 1995.

9. Appendix

A Control Software

The experiment has several parts. We must initialise it, run it and close it.

If we want to initialise the experiment, we have to run the Matlab function virtual_setup.m. This function opens two sockets between SUN computer (it is called 'Euler') and PC computer (it is called 'Lagrange'), each in one direction. Server called 'Banana' is to read images and 'Paprika' is to send the trajectories of the robot. File of calibrated cameras data is also opened. All variables are cleared their initial values. The robot goes to a prespecified initial position. In this position, we are able to see all the time through the cameras the green image of the TCP.

Our Simulink model is shown in the figure A.1. It is composed of several steps.

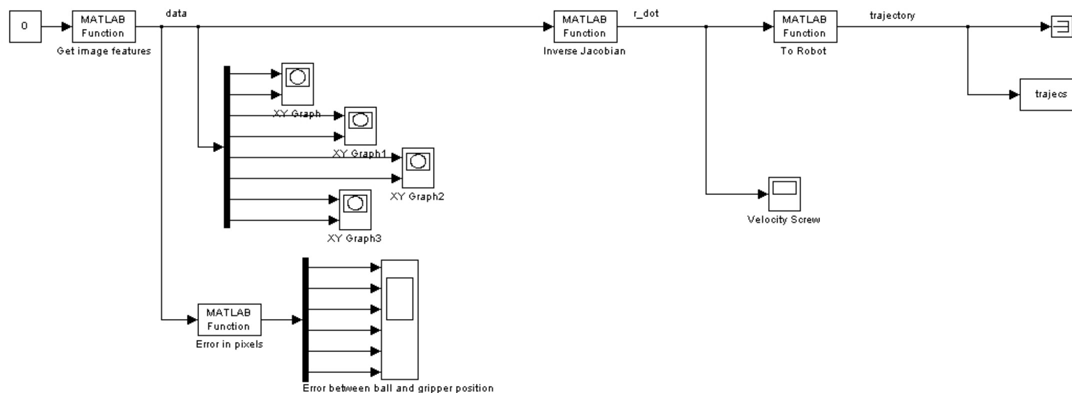


Figure A.1 Simulink model of virtual experiment

In the first step, we get image features. We get the ball and gripper position. We also calculate the error between ball position and gripper position and we plot them.

In the second step, we calculate the velocity screw from image data. We calculate the depth z of the robot with its position and $T_b^{c_1}$ that is the transformation between camera 1 and the robot base. Therefore, the depth z is the distance between cameras and the gripper (green

material). The image jacobian is also calculated. We use the depth z, and the position of the gripper in the X and Y axis.

We also calculate a reference velocity that is useful to control the robot better. This velocity is useful the error calculated can be less.

In this step, we compare our error with a tolerance to grasp the ball. If the error is less than the tolerance, the robot should grasp the ball. When we grasp the ball, we only calculate the trajectory that the robot has to follow until the initial position.

In the last step, the new position of the robot is calculated from velocity screw. We do an interpolation between the old and new position and we send to robot.

When we are going to grasp the ball, the first step we do is to send to Java one value and so the robot knows it should grasp the ball. Later, we send the trajectory the robot has to follow.

B Virtual Software

In virtual software, we should exactly reproduce our real system. Then, we must create irb2000 and irb6 robots, ball, bar where the ball runs, cameras to see images, and walls, the ceiling and the floor of the room. If we want to see everything on the screen, we should create frames. We construct four frames, one for the fix camera, two for cameras in irb6, and last one for the camera in irb2000 tcp.

Cameras in both robots must be attached to their tcps because cameras are there in real experiment. The two cameras in irb6 have to be separated.

B.1 Leaning bar

The bar is leaning a little bit in real experiment for rolling the ball. If we want to do this in Java3D, we have to create a Coordinate System attached to the bar. If we know two points of the bar P_1 and P_2:

$$v_x = P_2 - P_1 \qquad \text{(B.1)}$$

and v_z will be the projection of z axis orthogonal to v_x.

$$v_z = a_z - (v_x \bullet a_z) \cdot v_x \tag{B.2}$$

where a_z is the z axis and v_y will then be cross product between v_z and v_x, see figure B.1.

$$v_y = v_z \times v_x \tag{B.3}$$

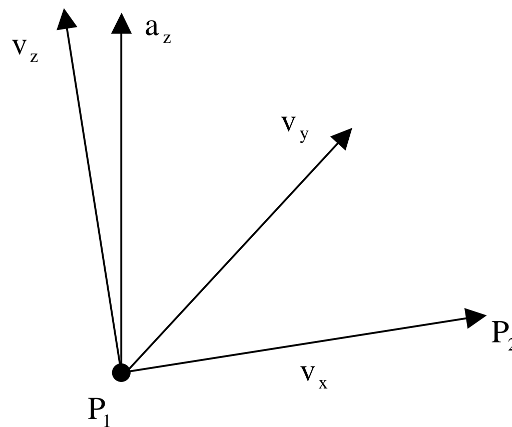


Figure B.1 Transform3D for the bar

The center of the Coordinate System is the point P_1.

Therefore my transformation matrix is a 4x4 matrix

$$M = \left(\begin{array}{ccc|c} v_x & v_y & v_z & P_0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right) \tag{B.4}$$

B.2 Ball trajectory

To calculate the ball trajectory we must interpolate between two known points of the trajectory. As the ball is rolling on the bar and we know points P_1 and P_2, see

B.1, we can know two points of the trajectory lifting up these two points. Therefore the trajectory would be

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = (1-u)\cdot\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} + u\cdot\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \quad \text{(B.5)}$$

It depends on the ball velocity but we do not know it. It can be approximated giving a constant value C and adding to u.

$$u = u + C \quad \text{(B.6)}$$

B.3 Image processing

Images are read from cameras in Java. Images are composed by 3 colours (red, green and blue). Then we have 3 matrices for each image. Thresholding is now done for each matrix. We should have 3 different thresholdings, one for each matrix. Our thresholdings will be such as the ball and the gripper are only seen.

After that, we have to send to matlab the ball and gripper coordinates. But we are only going to send the center point. So, we calculate the center point of the ball and the gripper.

B.4 Matlab – Java Connection

We have two sockets to connect Matlab and Java. In one socket, Java sends to Matlab images and in another socket, Matlab sends to Java the trajectory of the robot. The communication between these two softwares is slow, so ball and robot movement are very slow.

B.5 Calculation of the images center point

When we have cameras images with the thresholding done, we should calculate the center point of these images. We have two images, one for the ball and the other one for the gripper. We cross image from left to right and top to bottom. When we have one pixel with value 255f, we add one to my variable. When we have crossed all figure, we divide by the number of pixels that are 255f.

B.6 Grasping the ball

When the error is less than one tolerance, we must grasp the ball. To do this operation in Java is complicated. The ball while is rolling belongs to Virtual Universe and to its Coordinate System (Locale) but if we want the robot grasp the ball, this one should belong to the robot. Therefore, we have to remove the ball from the main Coordinate System and add it to IRB2000 TCP. We should calculate the transform between main Coordinate System and Robot Coordinate System and set the correct position of the ball. This leads to that the ball gives a jump. Now, the ball moves attached the robot. The gripper should close to grasp the ball.
When we have done this, the robot moves to the initial position.