ISSN 0280-5316
ISRN LUTFD2/TFRT--5859--SE

Vision Algorithms for ball on beam and plate

Magnus Espersson

Department of Automatic Control
Lund University
May 2010

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

Document name
MASTER THESIS

Date of issue
May 2010

Document Number
ISRN LUTFD2/TFRT--5859--SE

Author(s)
Magnus Espersson

Supervisor
Anders Robertsson, Vladimeros Vladimerou
Automatic Control, Lund
Rolf Johansson Automatic Control, Lund (Examiner)

Sponsoring organization

Title and subtitle
Vision Algorithms for ball on beam and plate (Algoritmer för datorseende för kula på bom och platta)

Abstract
The ball-and-beam process is one of the most famous control laboratory processes [1]. At the Department of Automatic Control at Lund Institute of Technology a light-weight balland-beam process has been made portable by implementing the controls on a laptop with a webcam. The main contribution of this master thesis was the testing and implementation of a variety of vision-based algorithms used to localize the beam, the ball on it and calculate the angle of the beam. A soft-real-time controller was also implemented which used the camera sampling period. Identification of gestures was used to set reference points on the beam. For further testing of the algorithms a ball-on-plate process was implemented using an IRB2400 6-axis robot, using hard-real-time

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
40

Recipient's notes

Security classification

Preface

This master thesis was written between September 2009 to May 2010 at the Department of Automatic Control at LTH, Lund University in Sweden.

I like to thank my advisors Vladimeros Vladimerou, who was my first line of support, and Anders Robertsson, who created this thesis, and my examiner Rolf Johansson. I also wish to express my gratitude to Rolf Braun, who build the ball-and-beam process for this thesis.

Contents

1. Introduction

Short descriptions about the different part of this thesis:

Chapter 2 presents the basics for the thesis.

Chapter 3 shows the implemented control theory.

Chapter 4 describes the different methods and compares them.

Chapter 5 discusses the results and how to improve the solutions.

Appendices are about important parts of the C code used in the software implementations and a manual of the programs for the ball-and-beam and the ball-and-plate.

1.1 Desired specifications

The goal is to get a beam-and-beam process working without the require of the electric ball position system and the encoder which measure the angle of the beam and replace them with an ordinary USB web-camera and vision algorithms. Without these systems the ball-and-beam process can be built smaller, cheaper and lighter, but this is not included in this thesis. This will enable the whole process to be portable when combined with a laptop to control the process. The communication between the computer and the process will be implemented to run with a USB interface or another standard port. See Fig. 1.1 to see the portable ball-and-beam process.

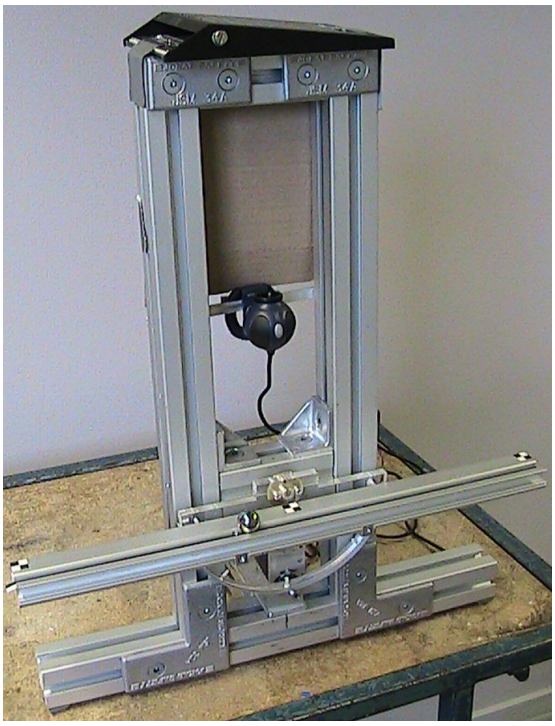

Figure 1.1: The portable ball-and-beam process.

The process should work with different beams and balls to make full use of the advantages of a ball-and-beam process with a camera. The variations of the beams can be different lengths, colors or even a curved beam as long as the beam is straight viewed from above. The balls can have different colors and sizes. The changes of the ball and the beam should not require any modifications in the software implementation.

To control a curved beam is not part of this thesis, but Matthias Busl has written *Modeling and control of a general ball and beam* [2] which contains a model for any curved beam. The model of the beam should be implemented between the controller for the position of the ball and the controller for the angle of the beam.

The software should be written in C code and should be as dynamic as possible to work with different frame resolutions, movement of the camera while running the process etc. The software should be easy to extend to work with similar processes. Also the software should have a basic graphical user interface (GUI), be able to run at an ordinary computer and control the process with a sample rate of 30 frames per second.

1.2 Advantages and drawbacks of using webcam vision for feedback

Some advantages with a ball-and-beam process with a camera:

- No problem arises if the ball loses connection to the beam in case of an uneven beam, an overaggressive controller or a bad connection between the beam and the ball.
- The ball-and-beam process can be simplified and therefore could be built cheaper, lighter and smaller. The process which is under development will be portable combined with a laptop, see Fig. 1.1.
- The power consumption for the portable ball-and-beam controlled by a laptop should only require a fraction compare to the standard ball-and-beam process combined with a stationary computer.
- Different kinds of balls can be used.
- The beam and the ball can be changed without modifying anything else.

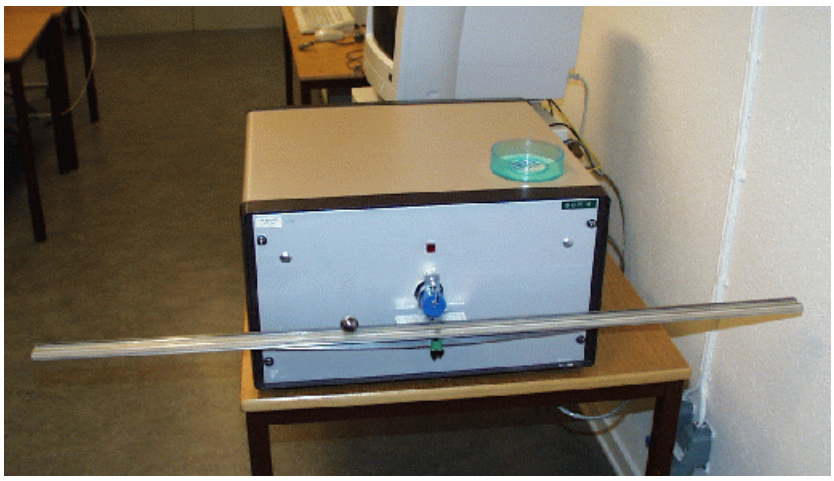

Figure 1.2: The original ball-and-beam process [11].

Drawbacks with a process with a camera:

- Depending on the specifications of the camera used the angle calculation has lower resolution.
- Processing the video stream is computationally more intense compared to getting feedback directly from a microcontroller.
- The light conditions must be fairly good and stable to maximize the contrast and minimize the shadow of the ball.
- The length of the beam is limited to roughly the distance between the beam and the camera. The new process which is under development uses a mirror to reduce the required height of the process, see the top of Fig. 1.1.
- A camera calibration is required for the camera for *some* algorithms.

1.3 Experimental setup

Hardware

The software can be adopted in various settings, but for the purposes of this thesis, the equipment currently used is as follows:

- A ball-and-beam process, see Fig. 4.13.
- A web-camera with V4L2 support.
- A computer running a Linux based operating system with the Comedi interface [3] installed.
- Markers and other visual queues.
- A checkerboard for the camera calibration.

Software

The software is implemented to run on a Linux operating system with the following:

- The ball-and-beam program or the ball-and-plate program.
- The OpenCV libraries for the GUI.
- The V4L2 library for the camera support.
- The ball-and-plate program also requires Matlab with Simulink to modify the model containing the controllers.

2. Methodology

2.1 The ball-and-beam process model

The ball-and-beam process consists of an electric motor which controls the angle of the beam. The task is to control the position of the ball which is placed on the beam. The motor is connected to the middle of the beam, therefore the gravity force acts at the beam can be neglected. The position of the ball is controlled by rotating the beam.

The gravity force component in Fig. 2.1 causes the ball to move when an angle is applied to the beam and is proportional to the angle of the beam.

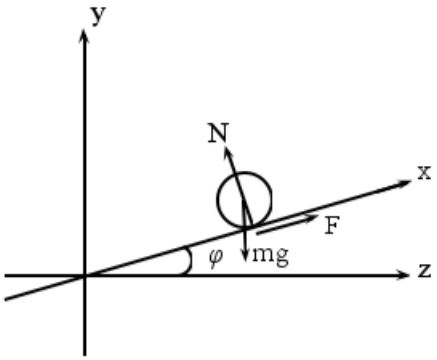

Figure 2.1: Ball-and-beam model with acting forces.

The angle of the beam is ϕ and x is the position of the ball. The Fig. 2.1 gives the following force equations [4]:

$$m \cdot \ddot{y} = -m \cdot g + N \cdot cos\phi + F \cdot sin\phi \tag{2.1}$$

$$m \cdot \ddot{z} = -N \cdot sin\phi + F \cdot cos\phi \tag{2.2}$$

By multiplying the first equation with $sin\phi$ and the second equation with $cos\phi$ and then add the two equations we get [4]:

$$m \cdot (\ddot{y} \cdot sin\phi + \ddot{z} \cdot cos\phi) = -m \cdot g \cdot sin\phi + F \tag{2.3}$$

Also the following relationship can be made from Fig. 2.1:

$$y = x \cdot sin\phi \tag{2.4}$$

$$z = x \cdot cos\phi \tag{2.5}$$

By differentiating the two equations above twice and multiplying the first equation with $sin\phi$ and the second one with $cos\phi$ the sum of the two new equations becames [4]:

$$\ddot{y} \cdot sin\phi + \ddot{z} \cdot cos\phi = \ddot{x} - x \cdot \dot{\phi}^2 \tag{2.6}$$

If the friction between the ball and the beam is neglected the following equations can be stated:

$$F \cdot r = J \cdot \dot{\omega} = -\frac{J \cdot \ddot{x}}{r} \tag{2.7}$$

$$J = \frac{2 \cdot m \cdot r^2}{5} \tag{2.8}$$

$$\Longrightarrow F = -\frac{2 \cdot m \cdot \ddot{x}}{5} \tag{2.9}$$

By combining Eq. 2.3, Eq. 2.6 and Eq. 2.9 we get:

$$m \cdot (\ddot{x} - x \cdot \dot{\phi}^2) = m \cdot g \cdot sin\phi - \frac{2 \cdot m \cdot \ddot{x}}{5} \tag{2.10}$$

Which can be rewritten as (small angle assumed):

$$\ddot{x} = \frac{5}{7} \cdot (-g \cdot sin\phi + x \cdot \dot{\phi}^2) = [\phi \approx 0, \dot{\phi} \approx 0] = -\frac{5}{7} \cdot g \cdot sin\phi \tag{2.11}$$

As result we get the transfer function for the ball position process:

$$G_x(s) = -\frac{5 \cdot g}{7 \cdot s^2} \tag{2.12}$$

The voltage to the electric motor is proportional to the velocity of the angle of the beam, if the motor is simplified. This gives [4]:

$$\dot{\phi}(t) = k_\phi \cdot u(t) \tag{2.13}$$

where k_ϕ has been found to be around 4.4 [4], $u(t)$ is the applied voltage to the motor. This gives the transfer function for the angle process:

$$G_\phi(s) = \frac{k_\phi}{s} \tag{2.14}$$

Eq. 2.12 and Eq. 2.14 gives the transfer function for the process:

$$G_x(s) \cdot G_\phi(s) = -\frac{5 \cdot g \cdot k_\phi}{7 \cdot s^3} \tag{2.15}$$

2.2 The Comedi library

The Linux Control and Measurement Device Interface (Comedi) [3] is an Open Source project which helps developers to connect computers to I/O devices. In this thesis an IOTech DAQBoard/2000 is used to connect the ball-and-beam process to the computer. The necessary configurations and the C code to handle the I/O communication is originally written by Anders Blomdell, at the Department of Automatic Control at LTH, Lund University.

2.3 Computer vision

This part is about the computer vision and the aids used to visually locate the beam, the ball and the reference point set by the user visually and calculate the angle of the beam. All the programming code is implemented in C code and the calculations are made in real-time with soft-deadlines.

The main contribution of this thesis is to implement the computer vision algorithms and integrate all the different parts of the software to one well working program. One of the goals is to be able to process 30 frames per second in real-time. The ball-and-beam process does not require this sampling rate, but the program should work with other processes that could require faster sampling rates. Also the faster update rate gives the advantage of being able to better cope erroneous measurements. Contemporary web-camera rates are usually 24 fps or more.

Open Source Computer Vision

Open Source Computer Vision (OpenCV) [5] is a collection of Open Source libraries which are specially designed for computer vision. By using OpenCV's libraries a lot of the basic computer vision functions are relative easy to implement and therefore a good and fast way to start writing computer vision programs. Also there are many helpful guides and forums on the internet.

More about OpenCV is found in *Learning OpenCV* by Bradski and Kaehler [6] and in the reference manuals available at OpenCV's homepage [5]. The reference manuals are available for the program languages C, C++ and Python.

Video for Linux 2

Video for Linux 2 (V4L2) [7] is an Application Programming Interface (API) which enables access to settings in various kinds of multimedia hardware from Linux based operative systems. V4L2 is embedded to most new standard Linux kernels and is an improvement of the first version, Video for Linux (V4L). To find out more about available settings and how to implement a basic frame capture program see the V4L2 specifications [8] or just to get a glimpse see Appendix A.5 for a short example. In this thesis V4L2 is required to be able to set a couple of parameters in the initialization of the web-camera.

The web-camera

The web-camera used in this thesis is a Logitech Quickcam Pro 9000. It is a pretty common USB web-camera and has support for V4L2. The V4L2 support is a requirement to be able to run the software in this thesis since the V4L2 is used in the implementation. The web-camera is capable of grabbing 30 frames per second, but since the auto exposure is active by default the performance is limited to around 15 to 20 frames per second in a room with normal light conditions.

How to get full performance, 30 frames per second, is presented in Section 4.1 and also a part of the implemented C code is presented in Appendix A.5.

Camera calibration

A camera calibration can be performed with a couple of visual tests with a large checkerboard. Some distortions are detected in the corners of the frames but the distortions are limited and do not interfere with the process.

The program used for the camera calibration is an example program included in the source code for the OpenCV 2.0.0. The program calculates the focal length and the principal point which are used later in the thesis, more about the calibration is found in Appendix B.2. Simplified the principal point is the x- and y-coordinate focus point in the frame. The calibration is required since the lens is not manufactured exactly in front of the center of the sensor [9]. Ideally the principal point should be in the middle of the frame.

Digital images

As described before one of the requirements was to be able to detect different kinds of balls. This means that no assumptions about the color of the ball can be made for instance. Therefore all frames from the web-camera are saved as black-and-white images.

If simplified, digital images are represented by a matrix for each dimension of the image. Since black-and-white images can be represented in only one dimension, this makes the image processing easier. The elements in the matrix represents the value of the pixel, one element for each pixel.

The values of the elements for a black-and-white image matrix are between 0 - 255, depending on the relation between black and white in the pixel. In the software the resolution of the captured frames are set to 640x480 pixels, but can of course be changed if required.

3. Controlling the ball on beam

To be able to have different modes depending on the situation when running the software a regulator is required, there are three modes for the ball-and-beam process:

- *OFF* mode: no output allowed.
- *BEAM* mode: control only the angle of the beam.
- *BALL* mode: control the position of the ball.

These modes are set by the user through the GUI.

The Fig. 3.1 shows how the two implemented controllers are connected and the available signals from the original ball-and-beam process.

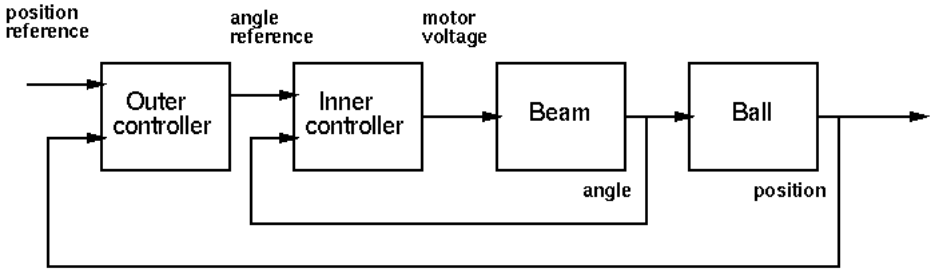

Figure 3.1: The control block of the original ball-and-beam process [11].

In this thesis the same block is used, the only difference is that all the input signals are determined by computer vision algorithms from the frames produced from the web-camera. The position of the ball and the reference set-point are scaled to -10 to 10, because this makes the process independent from the actual measured pixel values and the same scale is used in the original process. The calculated angle of the beam is converted to radians.

3.1 The inner controller

To control the angle of the beam a PI controller is used, see the inner loop in Fig. 3.1. A PI controller consists of two parts, one Proportional and one Integral part.

Equations for the PI controller [10]:

$$u(t) = K \cdot (e(t) + \frac{1}{T_i} \cdot \int^t e(\tau) \cdot d\tau) \tag{3.1}$$

$$U(s) = K \cdot (E(s) + \frac{1}{s \cdot T_i} \cdot E(s)) = P + I \tag{3.2}$$

where $e(t)$ is the difference between the set-point, y_{sp}, and the current value y and T_i is the *integral time*. The proportional part can be described as the current error times a factor. The integral part is often used to eliminate stationary errors.

The Proportional part

The proportional part is unsurprisingly the proportion of the error. A factor β is introduced to adjust the significance/weight of the set-point [10]:

$$P = K \cdot (\beta \cdot y_{sp} - y) \tag{3.3}$$

$$P(t_k) = K \cdot (\beta \cdot y_{sp}(t_k) - y(t_k)) \tag{3.4}$$

where t_k indicates discrete time since measurements are obtained by sampling.

The Integral part

The integral part is introduced to better cope with stationary errors. To correct the stationary error the integral part integrates the error between the set-point and the measured value and adds the error to the output signal.

Equations for the integral part [10]:

$$I(t) = \frac{K}{T_i} \cdot \int_0^t e(s) \cdot ds \tag{3.5}$$

$$\frac{dI}{dt} = \frac{K}{T_i} \cdot e \tag{3.6}$$

To prevent a change in the output signal when a parameter is changed and the process is in stationary mode a bumpless control implementation is required. The *bumpless parameter change* can be fixed by calculate the new integral part after the output signal been calculated or in other words precalculated to the next sample. By using the forward differences approximation this precalculation is possible.

With Eq. 3.6 combined with the forward differences approximation we get the following equation [10]:

$$\frac{I(t_{k+1}) - I(t_k)}{h} = \frac{K}{T_i} \cdot e(t_k) \tag{3.7}$$

$$I(t_{k+1}) = I(t_k) + \frac{K \cdot h}{T_i} \cdot e(t_k) \tag{3.8}$$

where t_{k+1} is the next step in discrete time and h is the sampling period.

Anti-windup Something that could be a significant problem if not taken care of when implementing the integral part is the so called *integral windup*. This occur when the output signal is limited for some reason, e.g. power limitations in a motor. When the set-point is outside the limits for the minimum or maximum value the integrator will start to windup to try correcting the output error. This will make the integral part grow without limits and when the set-point is changed into the possible limits again the integral part will still try to correct the output since the sum of the old error must be subtracted first. The windup makes the process unresponsive until the integral part is corrected by the current error [10].

So an *anti-windup* method must be implemented. The easiest solution is to set a limit to the growth integral part. Another solution is to use a *tracking* method. A tracking method limits the integral part when the output is saturated.

The following modification to Eq. 3.8 adds a tracking method:

$$I(t_{k+1}) = I(t_k) + \frac{K \cdot h}{T_i} \cdot e(t_k) + \frac{h}{T_r} \cdot (u - v) \tag{3.9}$$

where T_i is called the *tracking time constant* and decides how fast the integral part will rest if the output is saturated. The parameter u is the output from the actuator and v is the output from the controller.

3.2 The outer controller

To control the position of the ball a PID controller is used as an outer control loop, see Fig. 3.1. The PID controller is the most used controller in the industry by far [10]. A PID controller consists of three parts a Proportional, a Integral and a Derivative part. The proportional and integral part are the same as for the PI controller, because a PID controller with the derivative part set to zero works like a PI control, this is common setting in the industry.

Eq.s for the PID controller [10]:

$$u(t) = K \cdot (e(t) + \frac{1}{T_i} \cdot \int^t e(\tau) \cdot d\tau + T_d \cdot \frac{de(t)}{dt}) \tag{3.10}$$

$$U(s) = K \cdot (E(s) + \frac{1}{s \cdot T_i} \cdot E(s) + T_d \cdot E(s)) = P + I + D \tag{3.11}$$

The Derivative part

The equation for the PID controller is the same as the PI controller except with the derivative part added. The derivative part tries to calculate future control signals.

One advantage of divide the PID controller into three different parts is that it becomes relative easy for humans to understand how the controller works. Also the PID controller has high performance if it is well-tuned and is an old and well tested controller.

To limit the effects from noise the derivative part should not be implemented as a in Eq. 3.11. Instead the derivative part is modified to the following approximation [10]:

$$s \cdot T_d \approx \frac{s \cdot T_d}{1 + s \cdot T_d/N} \tag{3.12}$$

where N limit the gain at high frequencies. N is usually set to 10 - 20 and is called the *maximum derivative gain*.

In the real world it is a good idea to be able to modify the influence of the set point y_{sp}. This is made by adding a factor γ to the derivative part in Eq. 3.11 and combined with the approximation in Eq. 3.12 [10] we get:

$$D(s) = \frac{s \cdot T_d}{1 + s \cdot T_d/N}(\gamma \cdot Y_{sp}(s) - Y(s)) \tag{3.13}$$

Often is the factor γ set to zero because when the derivative part does not care about the set point.

Backward difference approximation The backward difference approximation is crucial for the implementing of the PID controller in this thesis. There are other approximations e.g. the forward difference approximation, but it is unstable for large values for the sample rate h and for small values for the deviate parameter T_d. The backward difference approximation does not have these drawbacks which is one of the reasons it is the most widely used approximation today [10].

Eq. 3.13 with γ set to zero:

$$\frac{T_d}{N} \cdot \frac{dD}{dt} + D = -K \cdot T_d \cdot \frac{dy}{dt} \tag{3.14}$$

Then apply the backward difference approximation:

$$\frac{T_d}{N} \cdot \frac{D(t_k) - D(t_{k-1})}{h} + D(t_k) = -K \cdot T_d \cdot \frac{y(t_k) - y(t_{k-1})}{h} \tag{3.15}$$

It is easy to see in Eq. 3.15 that the backward difference approximation uses the values from a sample back in time, therefore its name.

Eq. 3.15 can be rewritten to:

$$D(t_k) = \frac{T_d}{T_d + Nh} \cdot D(t_{k-1}) - \frac{K \cdot T_d \cdot N}{T_d + N \cdot h} \cdot (y(t_k) - y(t_{k-1})) \tag{3.16}$$

This equation is implemented to calculate the derivative part for the ball-and-beam process.

3.3 Kalman filter

A Kalman filter can predict a state of a system [12] and is implemented to predict the position of the ball-and-beam process. This makes the process less sensitive if the ball is not detected by the localization algorithm, because then the predicted position of the ball is used instead. The reason for the missed ball can be that the ball is covered by e.g. a hand or due to poor light conditions.

The Kalman filter is implemented by using the Kalman functions[1] from the OpenCV libraries. The equation for the Kalman prediction[2] [13]:

$$x'_k = A \cdot x_{k-1} + B \cdot u_k \tag{3.17}$$

$$P'_k = A \cdot P_{k-1} \cdot A^T + Q \tag{3.18}$$

where x'_k is the predicted state (here the position of the ball), A is the state transition model matrix, B is the control-input model matrix, x_{k-1} is the corrected state from the previous step, u_k is the external control parameter (here the angle of the beam), P'_k is the priori error covariance matrix, P_{k-1} is the posteriori error covariance matrix and Q is the covariance matrix.

The matrix for the state transition A is the relationship between two steps. The goal is to get a Kalman filter which is fast enough to follow a moving ball but not too fast. This because if one large erroneous localization of the position of the ball is made the affect to filter should be minimized. Therefore the state transition is determined experimental.

3.4 Tuning PI/PID controllers

The ball-and-beam process

The tuning of the controllers for the ball-and-beam process is done in a number of steps. First the inner controller is tuned and then the outer controller, see Fig. 3.1. The parameters for the PI and PID controller are same as in Eq. 3.1 and Eq. 3.10.

First the inner controller is tuned by just a pure proportional part and is increased until the performance is good. Then the integral part is tuned to eliminate stationary errors, but not used for the ball-and-beam since it is not required due to a powerful motor.

If the outer controller only consists of a PI controller the result is that the ball-and-beam does not try to stop the ball on the reference point. Instead at the reference point the proportional part is zero and the integral part is the sum of the error since the old reference point. So the ball continues past the reference point and then the integral part starts to decrease and the proportional part acts on the new error. The ball never stops and in best case oscillates around the reference point.

The derivative part is therefore added to make the ball stop at the reference point. The derivative part tries to calculate how to stop the ball at the reference point. With the ball-and-beam case the ball is stopped by rotating the beam so the speed of the ball is zero at the reference point, if the controller is well tuned.

The ball-and-plate process

The tuning of the ball-and-plate process is made similar to the ball-and-beam process, but the inner loop is different. The difference is that the ball-and-plate controllers can set the desired angles of the plate directly. The control of the angles of the plates is therefore controlled by adding a small *delta* to the angle in every sample period. If delta is set too large the security system for the robot notices a discontinuity and shuts down the robot.

3.5 Real-time systems

The implementation used for control the ball-and-beam process originates from a laboratory exercise from the course FRTN01 Real-time Systems[3] at LTH, Lund University. The laboratory exercise is implemented with Java code.

[1] `http://opencv.willowgarage.com/documentation/motion_analysis_and_object_tracking.html#cvkalman`

[2] `const CvMat* cvKalmanPredict(CvKalman* kalman, const CvMat* control=NULL)`

[3] `http://www.control.lth.se/course/FRTN01/`

In this thesis the vision algorithms are implemented in C code and the implementation of the controllers are available in both C code and Java code. The most significant difference between the C code and the Java code, except for the vision algorithms, is the extensive GUI which the Java implementation has. The GUI for the C program is limited.

All the implementations are designed for a computer running a Linux based operative system. Three different main implementations are available:

- A pure Java implementation, feedback from the Comedi interface.
- A pure C implementation, feedback from the vision algorithms.
- A C- and a Java-program runs simultaneously. The C-program runs the vision algorithms and provides the feedback to the Java-program through a socket. The Java-programs runs the controllers and allows the user to set the control parameters during run-time through its GUI.

The Java implementation

The pure Java program consists of three threads with different priorities:

- A regulator and the controllers, high priority.
- A GUI, medium priority.
- A reference generator for the set point, low priority.

To able to lock a thread *monitors* are used, in Java the monitors are available by *synchronized methods* and *synchronized blocks* [10]. Synchronized methods are used for lock a function and synchronized blocks are used for lock a block in a function. To lock a function or a block in a function the same keyword, `synchronized`, is used.

For synchronized methods in the implementation the function is locked for other threads and for the synchronized block all objects in the block are locked from the other threads. During the time a function or block is locked other threads which tries to get a lock will be stored in a queue. The order of the threads in the queue is not specified in the Java standard, but on most platforms the queue is sorted by priority [10].

The C implementation

In the pure C implementation all the calculations for the controllers and the localizations of the beam and the ball, the calculation of the angle of the beam and the detection of the visual reference are processed in real-time. But since the deadlines depends on the frame rate from the camera there is no hard-deadlines, because the program waits until a new frame is ready. This is called soft-deadlines [10].

The software creates two threads, see Fig. 3.2. The first thread, *Thread 1*, grabs the current frame from the web-camera and converts the image buffer to an OpenCV image. The other thread, *Thread 2*, makes all the localizations and calculations such as the Kalman filter for the prediction of the position of the ball. The thread for the frame grabbing, Thread 1, is created and joined again with the main thread, Thread 2, for every new frame, see Fig. 3.2. The purpose of the continuous creating and joining of the frame grabbing thread is to synchronize the latest available frame with the main thread which makes the localizations and calculations for the control output.

Delays If the frame rate is set to 30 frames per second the average time between two control signals are the sample period for the process which is around 33 milliseconds. But the frame which the calculations are based from may in worst case be 33 ms old. Therefore the delay from the start of grabbing the frame to a control signal can in worst case be around 66 ms. The average delay is assumed to be 40 - 50 ms.

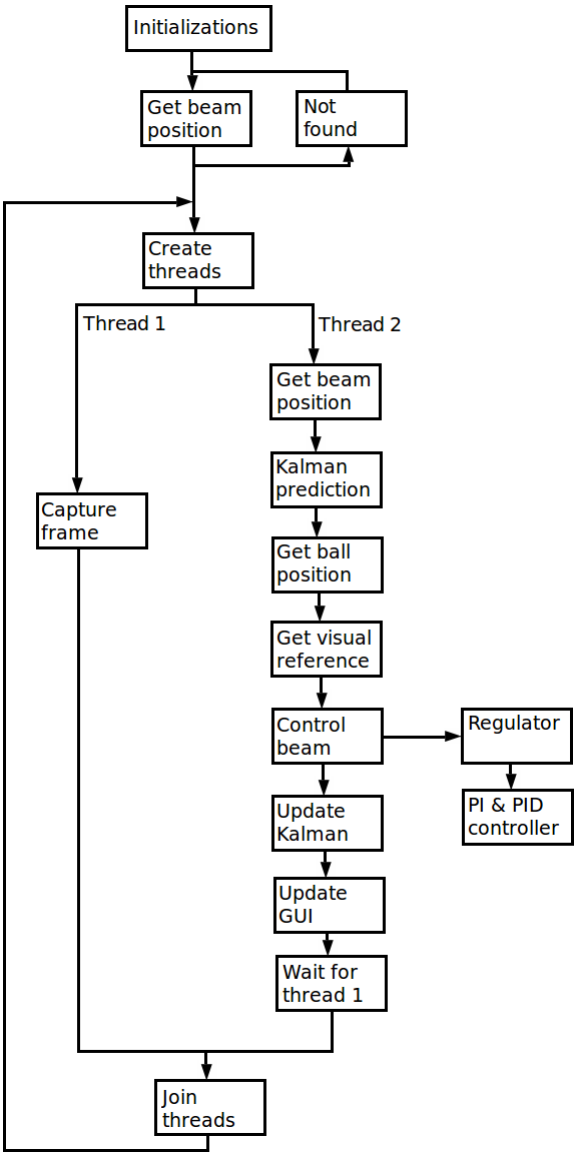

Figure 3.2: Flow graph of the pure C code software used for the ball-and-beam process.

The ball-and-beam process can be changed to a real-time program with hard-deadlines. This can be made by never joining the two threads, see Fig. 3.2. The web-camera could store the frames in a buffer instead and Thread 2 could fetch the frames from the buffer. The two threads would then be running independent of each other.

The ball-and-plate process

The ball-and-plate process runs with both soft- and hard-deadlines. Because the program is implemented the same way as in the software flow in Fig. 3.2, but with one important difference; the controllers are implemented in a Matlab Simulink model. The Simulink model runs on another computer and with hard-deadlines. The sample rate for the ball-and-plate controllers are 250 Hz.

4. Vision-based algorithms for feedback

4.1 Frame rate

The web-camera used for this thesis, see Section 2.3, automatically lowers the frame rate from the desired 30 frames per second if the light conditions is not really good. To be able to fulfill the 30 frames per second requirement the auto exposure feature in the used web-camera has to be turned off. By using the frame grabbing functions in OpenCV there is no way to turn off the auto exposure. To turn off the auto exposure feature the captured frames has to be fetched by the V4L2 API directly from the web-camera.

The V4L2 API specifications [8] provides most of C code required to grab a frame. By extending it with a couple of lines in the initialization of the web-camera the auto exposure is disabled and the frame rate set to 30 frames per second, see Appendix A.5 how to implement the methods above. The disabled auto exposure makes the frames slightly darker, since the light exposure is reduced, but the difference can be neglected especially considering the performance increases from 15 - 20 fps to 30 fps.

Since the initialization of the web-camera is made each time the program starts the user is not required to do any preconfigurations to get the 30 frames per second performance and the software should work with other cameras supporting the V4L2 API.

None of the modifications above would be required if another camera was used which did not override the set frame rate when the light conditions is not perfect. Then the standard functions from OpenCV's library would be enough to capture the frames, see Appendix A.2.

4.2 Locating the beam

The localization of the beam is calculated by a vision-based algorithm. The web-camera is placed above the process, straight above the beam.

Below are two methods described that were tested for the localization of the beam:

1. The Hough line transform

The Hough transform is designed to detect lines, circles and other distinct forms fast in an image. The Hough line transform is the original transform which later was generalized for other shapes. The general idea for the Hough line transform is that all pixels in an image can be a point on any number of lines. A (a,b) plan, there a is the slope and b is the intercepts of all lines that intersects the chosen pixel is created. This is done for all pixels in the image and the points in the (a,b) plane are added together, this summation which is the (a,b) plane can be called the *accumulator plane* [6].

The Hough line transform is used for the first solution to locate the beam with computer vision. The function used is the OpenCV function `cvHoughLines2`[1]. This function returns all lines that meets the function's input requirements. Example of input parameters are the minimum length of the line and the longest gap between points along a line.

In theory `cvHoughLines2` is a good solution because it works with any straight shape of the beam and the beam would not require the special markers the final solution is completely depended on for the localization of the beam.

The `cvHoughLines2` function makes it pretty easy to get the basic line detection, but in the real world it is very hard to specify the function's input parameters so only the wanted lines are found and not for instance a table nearby. Also it is very hard to detect lines that are exactly the same length as the beam due to the light conditions cannot be perfect all the time for all angles of the beam. The reason for this problem is that the edge detection is

[1] `CvSeq* cvHoughLines2(CvArr* image, void* line_storage, int method, double rho, double theta, int threshold, double param1=0, double param2=0)`

depending on hard-coded color thresholds. In addition light reflections from the beam make it even harder to get good results. A dynamic color threshold could have worked, but when a good color reference would be required and the solution would probability not be stable enough.

Most of the drawbacks could be eliminated by using an uniform background, great light conditions and a beam with a color that had high contrast in relation to the background, but then it would not be a good solution since a lot of the surrounding environment would had to be customized to fulfill these requirements.

2. Special markers

The next attempt to locate the beam was with a special marker, see Fig. 4.1, at each end of the beam to indicate the ends of the beam.

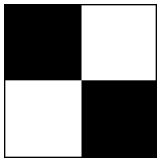

Figure 4.1: The marker.

The code below accesses the pixel values that the pointer `data` refers to, `data` points to the last frame from the web-camera.

The functions below are used for detecting the special marker, see Fig. 4.1.

```
/* Get the sum of colors represented by the pixels in a
 * diagonal '\'. */
int get_marker_color_1(int x, int y)
{
    //data: pointer to frame
    //WIDTH: width of the frame
    //x: the x-coordinate in the frame
    //y: the y-coordinate in the frame
    int v1 = 2;
    int v2 = 8;
    return  data[(y + 0)  * WIDTH + x + 0] + //one pixel
            data[(y + v1) * WIDTH + x + v1] +
            data[(y + v2 - v1) * WIDTH + x + v2 - v1] +
            data[(y + v2) * WIDTH + x + v2];
}


/* Get the sum of colors represented by the pixels in a
 * diagonal '/'. */
int get_marker_color_2(int x, int y)
{
    int v1 = 2;
    int v2 = 8;
    return  data[(y + v2) * WIDTH + x + 0] + //one pixel
            data[(y + v2 - v1) * WIDTH + x + v1] +
            data[(y + v1) * WIDTH + x + v2 - v1] +
            data[(y + 0) * WIDTH + x + v2];
}
```

Parameters By tuning the $v1$ and $v2$ parameters in the functions above the size of the markers the algorithm searches for changes. The functions reads the pixels in two diagonals.

The marker is detected by iterating through the whole frame and comparing the sum from the two functions above. The iteration is made pixel by pixel. The iteration over the whole frame is implemented with a double *while*-loop.

```
/* Use edge detection to localize a marker. */
void localize_marker()
{
    int x;
    int y = 0;
    int color_threshold = 150 * 4;
    while (y < HEIGHT - 8) {
        x = 0;
        while (x < WIDTH - 8) {
            int color1 = get_marker_color_1(x, y);
            int color2 = get_marker_color_2(x, y);
            int c12 = fabs(color1 - color2);

            //test if there is a black and a white area
            if (c12 > color_threshold) {
                marker_found = true;
                return;
            }
            x++;
        }
        y++;
    }
}
```

Parameters The parameter *color_threshold* can be tuned, a higher value decreases the false localizations, but requires the markers to have a higher contrast between the white and black areas, see Fig. 4.1. To make the algorithm more efficient the step for each iteration can be increased, but the increased step will lower the accuracy. In the algorithm above the step of the iteration is set to one, both vertical, y, and horizontal, x.

This solution is both fast to compute and very accurate, around two pixels accuracy with a resolution of 640x480. Since the position of the beam updates with a relative high frequency the process usually can handle if the camera or the beam is moved during run-time.

Optimization If all the beam markers are found in the last frame the localization function only searches for the new beam markers positions around the positions where the markers were found in last frame. This makes the localization faster and more robust. If one or more of the beam markers are not found in the last frame the localization function for the beam markers searches the whole frame again.

A significant advantage with this solution is that it can locate any number of markers with some additions to the function. The locating of the plate uses the same localization function as the localization of the beam uses.

4.3 Locating the plate

The purpose of localizing a plate is to be able to control the position of a ball placed on a plate. The plate gives the ball freedom to move in two dimensions. The ball-and-plate process can be seen in Fig. 4.2.

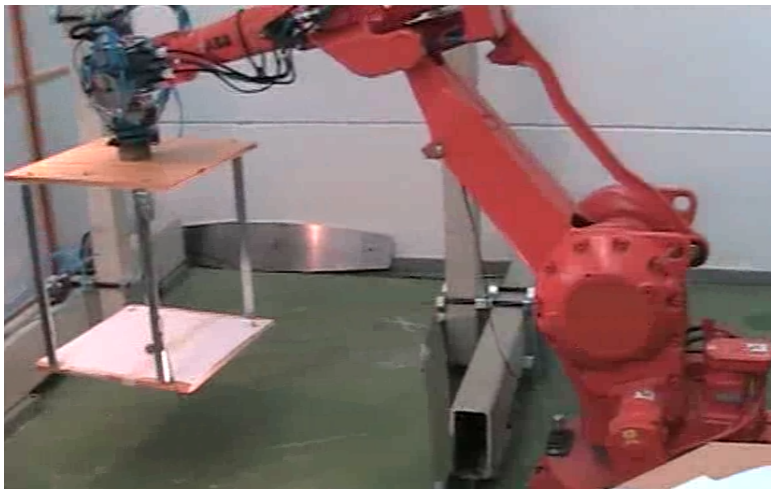

Figure 4.2: The ball-and-plate process.

As described earlier in the localization of the beam part, the locating of the plate uses the same localization function as the locating of the beam. The markers, see Fig. 4.1, used for the ball-and-beam process are also used for the ball-and-plate process. Three markers are placed at three of the four corners. By assuming the plate rectangular, the fourth corner is estimated from the positions of the three placed markers.

In order to identify the area which the ball is limited to, the software requires to know which marker represents which corner. We require that a marker is place at each of the two corners in the top of the frame. This because the two markers closest to the top of the frame are assumed to represent the upper limit of the plate. The third marker represents the down-left corner, if it is closer to the upper-left corner than the upper-left corner along the x-axis. Otherwise the third corner represents the down-right corner.

Robustness The only difference between the algorithm for the beam and one for the plate is that once the three markers are found the localization function searches only near the found markers. The program never searches the whole frame for markers again, unless the user does not activate the option through the GUI. This limitation is introduced to make the process more robust.

Optimization Like in the case for the beam the localization function for the plate used the last values of the found positions of the markers to make the search algorithm faster and minimize erroneous detections. The same marker cannot be found twice since a minimal distance between the found markers is required. Note that the markers not are exactly fixed, since through vibrations they appear to be moving even to a fixed camera.

4.4 Locating the ball on the beam

To simplify the locating of the ball on the beam the following assumption is made; the ball is somewhere along the beam. This assumption allows some important simplifications to be made, such as the area of the search for the ball becomes considerably smaller which makes the search much more computer efficient. Also since the search area is only along the beam there cannot be any erroneous detections of the ball outside the area of the located beam.

After the ball is detected the relative position of the ball is converted to a -10 to 10 scale, with -10 being the closest detectable location to the left marker, and 10 to the right one, and is sent to the controller of the ball position. The ball position is scaled depending on current calculated length of the beam, more about the calculated beam can be found in Section 4.6. The scaled position of the ball on the beam does not change if the height of the web-camera is changed or when the beam rotates.

Three different methods were tested for localization of the ball, as follows:

1. The Hough circle transform

The Hough circle transform is based from the Hough line transform, see Section 4.2. But instead of two dimensions, three dimensions are required, because the radius r and the x- and y-coordinates must be stored. Therefore the accumulator plane becomes an *accumulator volume*. The extra dimension requires much more memory and takes longer time to calculate [6].

The first attempt to locate something round the OpenCV function `cvHoughCicles`[2] is used. The algorithm is a more efficient Hough circle transform called Hough gradient method [6]. The function finds circles in an image and returns the center points and the radius of the found circles. It is easy to implement, but the circles must have sharp edges which makes it very hard to detect a stainless steel ball, since the steel ball reflects the surroundings and therefore the edge between the steel ball and the beam becomes too smooth.

2. Filter functions

The next method filters the image in order to make the ball become a black blob on the beam in the filtered image. To achieve this a couple of OpenCV filter functions are used after each other. The locating of the ball is made by detecting the black blob along of the beam. The iteration is made along the slop between the markers on the ends of the beam.

This solution works good enough to locate the ball in most of the frames, but the solution only detects dark balls and requires good light conditions. Another significant drawback is that all the filtration takes a lot of CPU time, the filtering cannot be made on the whole frame in real-time without decreasing frame rate.

3. Edge detection

The final implementation is the simplest and the fastest one. The algorithm iterates once per frame along the direction between the two markers on the ends of the beam, each step of the iteration is one pixel.

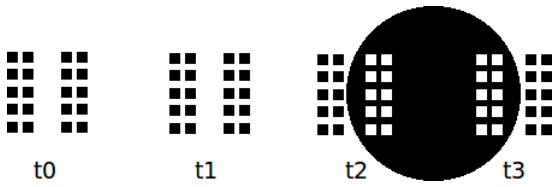

Figure 4.3: Edge detection with four iterations, $t0$, $t1$, $t2$ and $t3$. Each square represents a pixel, the columns of pixels are always vertical in relational to the frame. The black circle represents the ball in a frame.

In Fig. 4.3, the principle for the edge detection can be seen. The edge detection is made by comparing the sum of the pixel values of two short vertical lines of pixels and the sum of the pixel values of two identical lines which are moved forward a short distance. The two sharpest edges are considered as the start and end points of the ball. The solution is stable and works with different kinds of balls, which is a important requirement for this thesis to fulfill.

The function `get_color()`, see C code below, reads the values of the pixels in a column. The function is called by the edge detection algorithm. As seen in Fig. 4.3, the function is called four times in each iteration, once for each column.

```
/* Get the sum of the colors represented by five pixels in a column. */
static int get_color(int x, int y)
{
    int v = 4;
    return    data[(y - v)   * WIDTH + x] + //one pixel
              data[(y - v/2) * WIDTH + x] +
              data[(y + 0)   * WIDTH + x] +
```

[2] `CvSeq* cvHoughCircles(CvArr* image, void* circle_storage, int method, double dp, double min_dist, double param1, double param2, int min_radius, int max_radius)`

```
            data[(y + v/2) * WIDTH + x] +
            data[(y + v)   * WIDTH + x];
}
```

Parameters The parameter v determines the distance between the pixels in the column. By decreasing or increasing v the algorithm can be tuned to work better with smaller or larger balls.

The edge detection algorithm is implemented in function `find_ball()` and can be seen below. Some of the code is removed in order to make the main function of the algorithm easier to follow. In Appendix B.1 is a manual how to access the complete code.

```
/* Detecting the ball as the two largest color edges along the
 * beam. */
static void find_ball()
{
    /* Some initializations are removed in order to focus on the
     * main parts of the algorithm. x and y represents the x- and
     * y-coordinate in the frame. */
    //adjusts horizontal distance between columns of read pixels
    int delta = 8;
    //shortest diagonal of the ball allowed
    int dist_diag = 15;
    //same value as in get_color()
    int v = 4;
    //start x-coordinate for the first iteration
    // x1 and x2 are the end coordinates for the beam
    // CUT is to avoid iteration over the markers on the beam
    int x = x1 + CUT;
    int x_stop = x2 - CUT;
    int x1_tmp = WIDTH;
    int x2_tmp = x;
    //edge threshold
    first = second = 0;
    while (x < x_stop) {
        y = x * k1 + m1;

        color1 =     get_color(x - delta/2, y) +
                     get_color(x - delta, y);
        color2 =     get_color(x + delta/2, y) +
                     get_color(x + delta, y);
        color_diff = fabs(color1 - color2);

        //save the two largest color differences
        if (color_diff > first && x > x2_tmp + dist_diag) {
            first = color_diff;
            x1_tmp = x;
            y_tmp = y;
        } else if (color_diff > second && x > x1_tmp + dist_diag) {
            second = color_diff;
            x2_tmp = x;
        }
        x++;
    }
    /* Check diameter of found ball compare to the length of the
     * beam, easy to tune. */
    /* Consider the ball as missed if the new position of the ball is
     * far from the previous position. For each frame the ball is
```

```
     * missed the allowed "jump" is increased. */
}
```

Parameters The distance in pixels between the columns in Fig. 4.3 is depending on *delta*. The minimal diameter of the ball is set by the parameter *dist_diag*. The parameters *first* and *second* is the edge threshold between the beam and the ball, if set to zero no minimal threshold is used. The step of the iterations is set to one in order to get the highest possible resolution of the position of the ball.

Two drawbacks with the edge detection solution are found. The first drawback is that the iteration along the beam must not include the markers on the beam, because if they are included the markers are detected as the sharpest edges, which are their purpose when detecting the beam. Therefore the ball cannot be detected near the markers.

The second drawback is if the lighting condition is poor and a light source comes from the side, then the light produces a shadow which may be detected as the edge of the ball. The process still works, but the position of the ball is displaced from the center of the ball.

4.5 Locating the ball on the plate

The search for the ball is limited to the found plate area. The locating of the ball on the plate uses the same edge detection function as the beam mode uses, see Fig. 4.3, but with an additional condition.

In the ball-and-beam process the ball detection algorithm iterates only along the beam. The ball-and-plate process uses two algorithms, the first algorithm locates ball candidates along *direction1*, see Fig. 4.4, and then the second algorithm iterates along *direction2* at the localized candidate found by the first algorithm.

The second algorithm tests if the black-and-white ratio of the ball is the same in the top and bottom of the ball as in the middle of the ball, it also returns the y-coordinate for the ball. This algorithm reduces the risk of detecting the ball and the shadow as the ball.

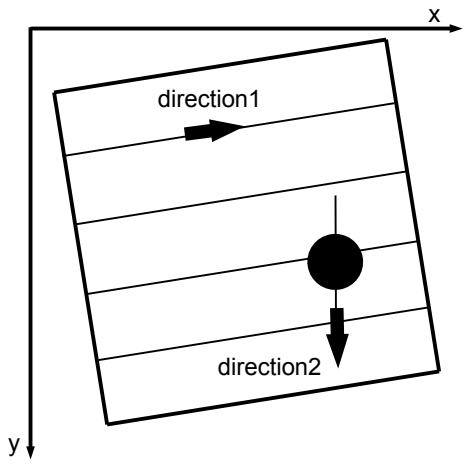

Figure 4.4: The iterations made to locating the ball on the plate. The point *(x,y) = (0,0)* is in the upper-left corner in an OpenCV image.

Parameters To make the algorithm which iterates along direction1 faster, the algorithm iterates only every fifth x-coordinate and every fourth y-coordinate, see Fig. 4.4. The tuning of these parameters is made by changing the parameters *x_step* and *y_step* in the source code, *ball.and.plate.c*, in the function *find_ball*().

To modify the two algorithms to be able to detect smaller or larger balls the width of the read pixels in each iteration can be changed. This is made by modifying the parameter *v* in the *get_color_v*() and *get_color_h*() functions.

4.6 Measuring the beam angle visually

In a step to make the ball-and-beam process portable the angle of the beam had to be measured visually by the web-camera. Then the angle encoder connected between the electric motor and the beam can be removed. Combined with the ball localization the Comedi interface for reading from the process is not required anymore.

Three main methods were used to measure the angle of the beam. The first method used only information on the apparent length of the beam, the second is based on absolute visual measurements using an encoder pattern placed on the process and the beam, and the third is based on a geometric approach using three markers.

1. Using the apparent length of the beam

As a first attempt to measure the angle of the beam the current visual length of the beam was used. The maximal length of the beam visually is found when the beam is parallel to the web-camera. To make this solution to work the beam had to be placed horizontal manually by the user or an alternative solution would be to control the angle of the beam around the horizontal position and save the maximal length of the beam.

After the maximal length of the beam is found the angle of the beam is calculated by comparing the current length of the beam and the maximal length of the beam combined with a basic trigonometric equation.

This solution has many problems. The most significant is that it has to be able to measure the length of the beam with high accuracy, especially since small changes of the angle are very hard to detect. Also the angle of the beam does not change relative much when the ball-and-beam process is run.

Another problem is if the camera is not positioned straight above the center of the beam, then the maximal length of the beam is when the beam is parallel to the lens of the web-camera not the floor. This can be solved by dynamic control of the ball to find the angle of the beam which the ball is stationary since then the beam would be horizontal.

To calculate the sign of the angle a third marker could have been placed at the center of the beam. By detect which direction the center marker moves when the angle changes the angle is easy to determine. This requires the center marker to not be on the axis of rotation.

2. Encoder-style detection

1+5 set strips This is an idea originally from Vladimeros Vladimerou and it requires two markers strips to detect the angle of the beam, see Fig. 4.5 and 4.6.

Figure 4.5: The angle marker strip for the reference angle.

Figure 4.6: The angle marker strip for the current angle.

The reference markers are placed on the process and are therefore stationary when the beam rotates. The markers for indicating the current angle are placed on a circular shaped surface at the side of the beam and are therefore following the rotation of the beam, see Fig. 4.7.

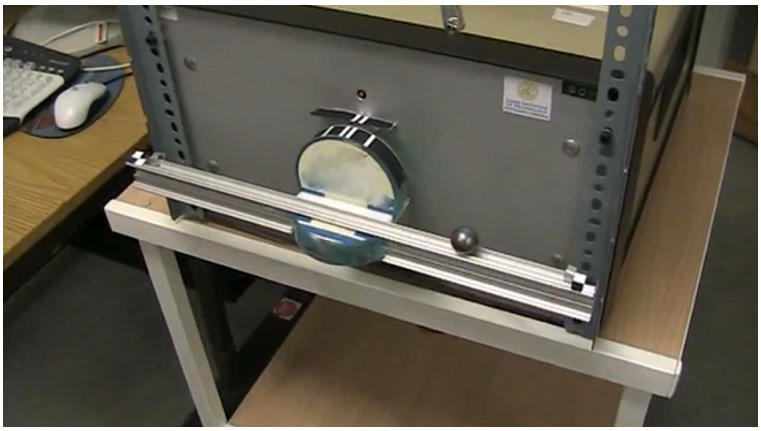

Figure 4.7: The ball-and-beam and the angle measurement strips.

Pseudo-code for the encoder-style detection algorithm.

```
1:  while y is close to the beam do
2:    x = xStart
3:    while x is close to the middle of the beam do
4:      y1 ← x
5:      color1 ← getColor(x − dx, y1)
6:      color2 ← getColor(x + dx, y1)
7:      if color2 − color1 > colorThreshold then
8:        markers[i++] ← x
9:      end if
10:   end while
11: end while
12: [nbrOfMarkersNearRef, indexOfFirstMarker] ← findMarkersNearRef(markers)
13: currentSet ← currentSetFcn(indexOfFirstMarker, nbrOfMarkers)
14: nbrOfPixels ← distanceFcn(currentSet, centerRef, centerCur)
15: return angle ← convert2Radians(nrbOfPixels)
```

Parameters and functions x and y are the current x- and y-coordinates in the frame, *xStart* is the x-coordinate where the iteration begins.

$color1$ and $color2$ are the results from the $getColor()$ function which returns the pixels values.

The minimal color edge threshold is set by $colorThreshold$ and can be tuned to e.g. work in different light conditions.

The array *markers* contains the positions of the i number of found markers.

The function $findMarkersNearRef()$ returns the number of angle markers that are close to the angle reference markers, $nbrOfMarkers$, limited by a threshold which is calculated from the distance between the two reference markers. The function also returns the lowest index of the markers near the reference markers, $indexOfFirstMarker$.

The $currentSetFcn()$ function determines the current set of the five possible, see Fig. 4.6, which is the set closest to the reference markers, $currentSet$.

The parameter $nbrOfPixels$ is the result of the function $distanceFcn()$ which calculates the number of pixels between the *center points*, see the thin lines in Fig. 4.8, of the reference angle markers, $centerRef$, and the current angle markers, $centerCur$, and then adds distance between the *center points* of the middle set of the current angle markers and the current set of angle markers.

The last function, $convert2Radians()$, converts the distance $nbrOfPixels$ to radians.

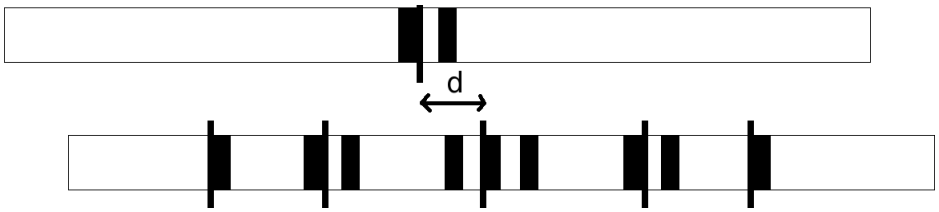

Figure 4.8: The angle marker strips with a non-zero angle. The thin lines represents the *center points* of the groups of markers. *d* is the distance between the reference angle and the current angle measured in pixels.

The detection of each angle marker is done with edge detection from white and black, so the result is slightly shifted to the left, see Fig. 4.8, but is the same for all markers so it does not interfere with the calculations. Each thin line represents an angle which is calibrated manually and put into the source code, see Fig. 4.9 how to do the calibration.

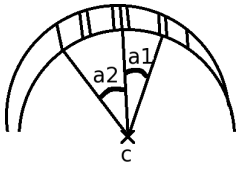

Figure 4.9: The measured angle references. *c* is the center point of the circle that the current angle strip is place upon. The angles to measure are $a1$ and $a2$.

To get a better accuracy of the angle measurements there are five groups of markers as seen in Fig. 4.6. The difference between the reference markers and the current angle markers is calculated from the closest group of markers, see Fig. 4.10. This minimized the distance between the two center points, which is good if the web-camera is not placed directly over the middle of the beam.

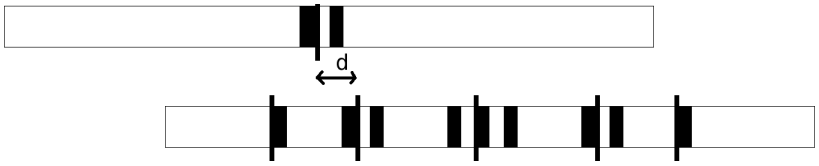

Figure 4.10: The angle marker strips with a non-zero angle.

By measuring the number of pixels between the same sets of markers that the angle $a1$ represents in Fig. 4.9, a conversion between pixels and radians is possible, because that angle is known. The result is used for calculating the current angle by measuring the difference between the reference markers and the current angle markers in pixels and convert the number of pixels to radians.

1+3 set strips Some testing with different colors, numbers of sets of markers and thicknesses of the markers were made. The best results is with the colors of the markers inverted, see Fig. 4.11 and Fig. 4.12, since the white strips are easier to find with computer vision compare to the black strips. Also the sets of markers are downsized from five to only three sets because the beam does not rotate that much. The angle measurement accuracy is still about the same and the software becomes a bit more stable since less can go wrong with the detection of the angle sets.

The algorithm, which calculates the angle, can be describe with the same pseudo-code as the algorithm in the *1+5 set strips* subsection uses. The only difference is that some of the called functions are simplified, due the reduction from five sets of markers to three sets.

Figure 4.11: The angle marker strips for the reference angle.

Figure 4.12: The angle marker strips for the current angle.

The sets of markers are placed at a circular surface. The angle between the sets of markers is manually measured, see Fig. 4.9, but only angle $a1$ is required. The manual measurement is only required once if no changes are made to the distances between the markers sets, the measured angle is modified in the *ball.and.beam.c* source code in the main-function. The software measures the difference between the sets of markers in pixels and combines the results so the number of radians per pixels can be calculated. The measured angle of the beam is presented in radians.

Robustness to camera motion If the web-camera is moved from its stationary place straight above the angle measurement strips the visual effects are minimized by placing the angle reference set and the current angle set very close together both vertical and horizontal. Also they are placed in the rotation center of the beam so the height of the markers of the current angle set do not changed when the beam rotates.

The process still works if height of the camera is changed, the required modifications in the implementation are made dynamically.

3. Geometric three-marker method

The position of the ball is scaled to -10 to 10 as described previously. This is done by comparing the positions of the two markers at the ends of the beam and the position of the ball. A change of the position of the ball is assumed to be linear for the previous solutions, but the linearity is only true when the ends of the beam are equally long from the camera.

The non-linearity is a visual effect caused by that the ends of the beam moves away respective towards the camera when the beam rotates. An object that moves closer to the camera covers more pixels and therefore has grown in size in the frame. A changed angle makes a stationary point on the beam to change its relative distances to the markers of the end of the beam. Therefore seems the stationary position on the beam move, if seen from the camera. The linear approximation is good enough to control the position of the ball roughly, but a more accurate approximation is wanted.

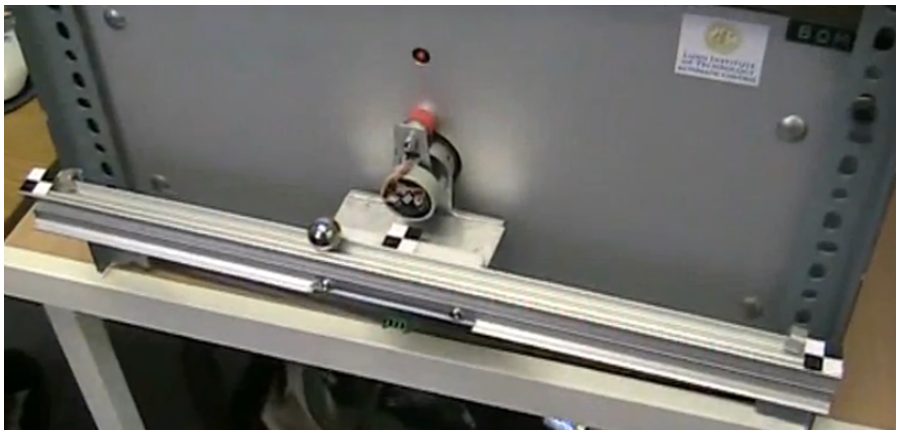

Figure 4.13: The ball-and-beam with the three markers.

The geometric three-marker method requires a center marker placed on the beam, see Fig. 4.13. For the ball-and-beam process used in this thesis the center marker could not be placed on the center point between the markers on ends of the beam, since then the ball would roll over the center marker. The displacement is dynamically adjusted for in the software.

The algorithm used for the calculation of the angle of the beam assumes that the center marker is on the axis of rotation, but this is not the case, which can be seen in Fig. 4.13. If a new process is built the center marker should be place on the axis of rotation.

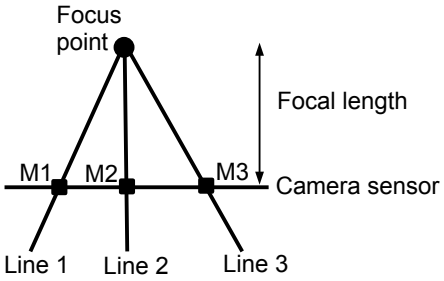

Figure 4.14: Three markers, $M1$, $M2$, $M3$, as the camera sees them. Projection lines drawn from the focus point through the markers.

The algorithm for the geometric three-marker method requires the following: the relative distances between the two end markers and the center marker in the frame, the focal length and the principal point, see Appendix B.2. The algorithm works only if the distances between the markers in reality are equal. The focus point should be placed straight above the center marker to get the most accurate calculated angle. Also the plan of the beam rotation should be parallel with the sensor plane of the camera.

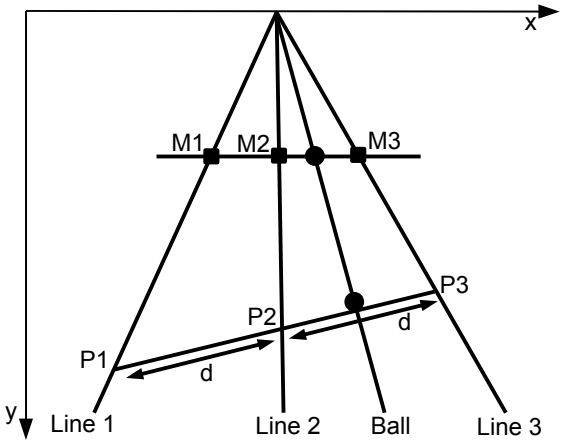

Figure 4.15: Same as Fig. 4.14 but with the calculated beam and a projection line for the ball, *Ball*. The distance d is between the points $P1$ to $P2$ and $P2$ to $P3$. x and y represents the coordinate system.

The algorithm for calculating the position of the beam in the lower part of Fig. 4.15 is implemented in the ball-and-beam program. Since the distances between the three markers on the beam are in reality equal, the calculated beam should have the same property. Therefore the distances between the points, $P1$, $P2$ and $P3$, see Fig. 4.15, should be equal and a single line should intersect the three points.

Parameters and functions x and y are the x- and y-coordinates, see Fig. 4.15.

$P1/P2$ are the points on $Line1/Line2$, see Fig. 4.15. $P1$ is predefined, $P2$ is iterated along Line 2 and $P3$ is a candidate for the solution. If $P3$ is a point on Line 3 a solution is found.

$M1/M2/M3$ refers to the markers in Fig. 4.15.

The first if-statement limits the maximal number of iterations to the half. This can be made because the distances between the markers, $M1/M2/M3$, gives the sign of the angle of the beam. The parameters *MIN* and *MAX* limits the number of iterations.

$P(x)$ refers to the x-coordinate of the point P.

The iterations are made along Line 2, see Fig. 4.15. The iterations are made by increasing the y parameter which is the y-coordinate.

The angle of the beam, *angle*, is calculated by a function, $trigFcn()$, which uses arcsine and the coordinates from $P1$ and $P3$.

Pseudo-code for the calculated beam algorithm.

```
1:  if M3(x) − M2(x) < M2(x) − M1(x) then
2:      MIN = 0
3:      MAX = P1(y)
4:  else
5:      MIN = P1(y)
6:      MAX = 2 ∗ P1(y)
7:  end if
8:  for y = MIN to MAX do
9:      P2(x) ← y
10:     P3 ← P2 + (P2 − P1)
11:     if Line3(x, y) == P3 then
12:         Break
13:     end if
14: end for
15: angle ← trigFcn(P1, P3)
16: ballPos ← intersectFnc(ballLine, calcBeam)
```

The beam is represented by a line, *calcBeam*. The parameter *ballLine* is the projection line of the ball, *Ball*, see Fig. 4.15. The intersection of *ballLine* and *calcBeam* gives the position of the ball, *ballPos*.

It is important to realize that the algorithm only relay on the relative distances, the results from the calculations do not have a unit. The algorithm starts with a predetermined y-coordinate which gives the point $P1$ on Line 1, see the pseudo-code above and the Fig. 4.15. If the start point, $P1$, is changed the result is still be the same, see lemma 4.1.

If the distance between $M1$, see Fig. 4.15, and $M2$ is longer than the distance between $M2$ and $M3$ on the camera sensor, then the angle of the beam will be negative, otherwise the angle will be positive.

Depending on the sign of the angle the iteration along Line 2 starts either at the top of the frame and stops at the same y-coordinate as the chosen point $P1$, if no matching third point, $P3$, is found. Else $P2$ has the same y-coordinate as the point $P1$ is the first iteration and the iteration stops when no solution can be found. This limit makes the algorithm more efficient.

At this stage of the algorithm a new second point, $P2$, is found in every iteration. The third point, $P3$, is calculated by assuming a line though all three points and that all points have the same distance to its closest point. But since the third point is only a candidate for the solution the algorithm tests if the third point, $P3$, is a point on Line 3. If this is the case then a solution is found, otherwise the iteration continues to the next point along Line 2.

When the algorithm finds the calculated beam two very important results can be calculated. The first result is a position of the ball with high accuracy. Because the position of the ball is the intersection between the calculated beam and the projection line for the ball, *Ball*, see Fig. 4.15. This eliminates the problem with the non-linear position of the ball when an angle is applied to the beam.

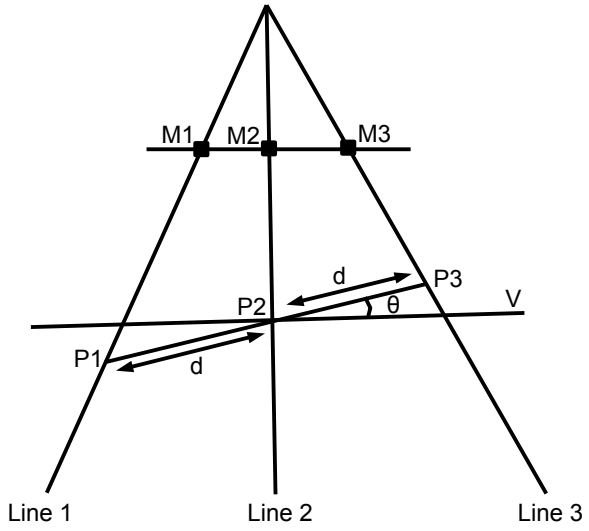

Figure 4.16: Same as Fig. 4.15 but with the angle of the beam, θ. The line V is parallel to the camera sensor.

The second result from the algorithm is the angle of the beam, θ, see Fig. 4.16. The resolution of the angle is higher then in the previous methods in Section 4.6. The angle is calculated by assuming a parallel line, V, to the camera sensor which intersects the projection Line 2 and the calculated beam in the same point, $P2$, see Fig. 4.16. Then, it is trivial to calculate the angle θ with a basic trigonometric equation.

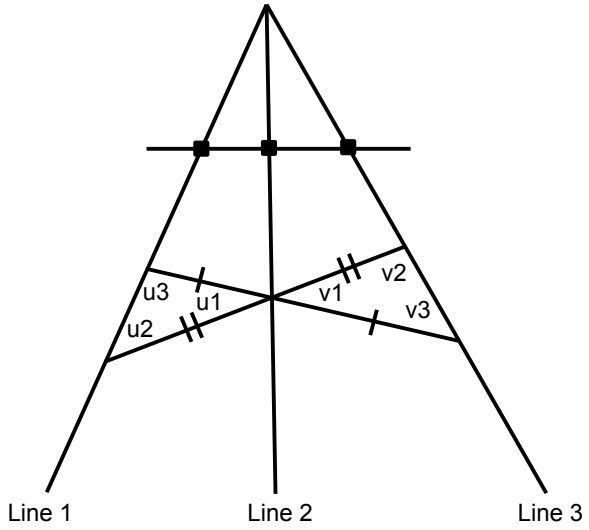

Figure 4.17: If two different solutions was found. $u1/2/3$ and $v1/2/3$ are the angles.

LEMMA 4.1
Given the projections of the beam markers, $M1$, $M2$ and $M3$, see Fig. 4.16, on the image plane, there is a unique angle θ formed by *any* collinear segments of equal length, d, with the center point, $P2$, on line V independent of the location of the centroid and the length of the segments. □

Proof. See Fig. 4.17, we begin by notice that the two angles $u1$ and $v1$ are equal. The laws of cosine gives that the angles $u3$ and $v3$ are also equal, because the length of the line segments, marked with short lines in Fig. 4.17, are equally long and the angles $u1$ and $v1$ are also equal. This gives that the angles $u2$ and $v2$ must be equal as well.

The result is that if the solution is not unique then Line 1 and Line 3 must be parallel. If Line 1 and Line 3 is parallel then the distance between the camera and the beam must go towards infinity. So the angle of the beam from the calculated beam algorithm is unique. ■

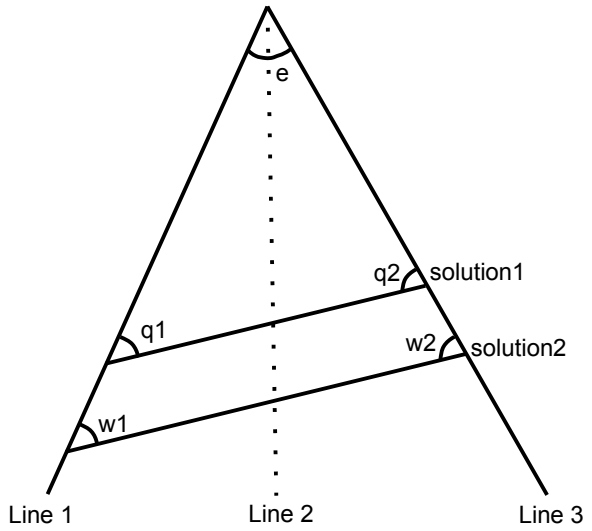

Figure 4.18: Two solutions drawn. e, $q1$, $q2$, $w1$ and $w2$ are the angles.

THEOREM 4.1
The calculated beam algorithm, see the pseudo-code in the geometric three-marker section, terminates and determines the correct angle of the beam. □

Proof. By lemma 4.1 there is a unique angle, θ, see Fig. 4.16, formed by line segments of equal length, d, with the endpoint projections $M1$, $M3$ and the centroid projection, $M2$, in the image frame. The angle θ is independent of the location of the center point, $P2$, in line V. Therefore it is enough to find one such point and length of the segments. The calculated beam algorithm does so and based on that determines the angle, which is unique independently of which length of the segments or which centroid was found. ■

The results from this section makes it possible to get accurate position of the ball and that the angle of the beam is found with higher accuracy than with previously methods. To get the most accurate results the camera should be placed as close to the beam as possible to minimize the quantized effect of the pixels.

A drawback with the solution is that a camera calibration is required to get the focal length and the principal point, see Appendix B.2. A problem with the solution is the sensitivity to movements of the camera sideways, since then the distances between the markers changes in the frame and therefore both the position of the ball and the angle of the beam are affected.

5. Conclusions

The main contribution of this master thesis is to be able to control a ball-and-beam process with the feedback from computer vision algorithms. The feedback is the angle of the beam and the position of the ball. These contributions makes it possible to control a portable version of the ball-and-beam process which lacks of feedback from a microcontroller. Since then the process can be build lighter, smaller and cheaper. A portable version of the ball-and-beam process is under development (May 2010), see Fig. 1.1, at the Department of Automatic Control at LTH, Lund University.

The computer vision algorithm for the localization of the ball is implemented to work with a number of different balls. Since the algorithm is designed to work also with balls with colors that are similar to the color of the beam some erroneous detections of balls are hard to prevent, especially if there is no ball on the beam. Also the algorithm cannot always detect a ball with relative high speed, since the contours of the ball becomes too smooth in the frames. To minimize this problem a Kalman filter was implemented to predictive the position of the ball.

To improve the localization of the ball on the beam the algorithms used for the ball-and-plate process could have be modified to minimize the erroneous detections of the ball and the shadow. Also the algorithm used in the ball-and-plate to localize the ball does not detect the markers as the ball.

The beam is localized by a computer vision algorithm which detects the three markers which are placed on the beam. One marker at each end of the beam and one marker at the middle of the beam. The two markers at the ends of the beam are enough to locate the beam, the third marker in the middle is used for the calculation of the angle of the beam.

The only limit for the detection of the markers is that the markers cannot be rotated to any angle. An algorithm that detects all angles was implemented, but it increased erroneous detections drastically. The algorithm can of course be improved, but it is not required for the ball-and-beam or the ball-and-beam process.

The calculation of the angle of the beam does not have enough accuracy to be able to control the position of the ball without some smaller oscillations. To improve the accuracy the frame resolution can be set to a higher value, the resolution is a trade-off between accuracy and computer computation requirements.

The algorithm which calculates the angle of the beam uses a linear iteration to find the solution. A faster solution would have been to implement the iteration with a binary search instead.

As an additional part for the thesis the algorithm for the localization of the ball is extended to be able to locate a ball on a plate. This modification makes it possible to control the position of a ball on a plate. The extended localization algorithm for the ball combined with the algorithm for locating the markers makes it possible to control the position of the ball on a plate and is implemented and tried.

The localization of the ball on the plate is able to detect the ball and not the shadow in most cases. The algorithms are implemented to localize balls with relative normal sizes as good as possible. The algorithms does not detect large nor small balls to minimize erroneous localizations. This limitation can be tuned.

References

[1] W. Yu (2009). Nonlinear PD regulation for ball and beam system. *International Journal of Electrical Engineering Education*, 46(1), 59-74.

[2] M. Busl, *Modelling and simulation of a general ball and beam process with interaction*, Projects in Automatic control FRT090, Department of Automatic Control, Lund University, Lund 2010.

[3] Comedi:
`http://www.comedi.org/`

[4] The ball-and-beam model:
`http://www.control.lth.se/~kurstr/Exercise3_05/ballandbeammodel.pdf`

[5] OpenCV:
`http://opencv.willowgarage.com/wiki/`

[6] G. Bradski and A. Kaehler, *Learning OpenCV*, p.153-161, O'Reilly Media, Inc., 2008.

[7] V4L2:
`http://linux.bytesex.org/v4l2/`

[8] V4L2 API specifications:
`http://v4l2spec.bytesex.org/`

[9] T. A. Clarke; X. Wang; J. G. Fryer (1998). The Principal Point and CCD Cameras. *The Photogrammetric Record*, 16(92), 293-312.

[10] K. E. Årzén, *Real-Time Control Systems*, p. 13-15, 50-68, 186-203, Department of Automatic Control, Lund University, Lund 2008.

[11] Source for images:
`http://www.control.lth.se/user/FRTN01/Exercise3/Exercise3.html`

[12] T. Glad, L. Ljung, *Reglerteori Flervariabla och olinjära metoder*, p. 144-161, Studentlitteratur, Lund 2003.

[13] R. Johansson, *System Modeling and Identification*, p. 462-464, Prentice Hall, Lund 2009.

A. Software manual

The purpose of this manual is to show how to implement a software to capture frames from a web-camera, get access to the pixels in the frame and display the results as a simple GUI. The manual also gives some hints about how to implement the V4L2 API.

All the OpenCV functions, options, descriptions of the function in- and out-parameters and much more can be found at OpenCV's reference pages[1].

A.1 Create a GUI with OpenCV functions

The C code below shows how a window is initialized using functions from the OpenCV library.

```
/* Initializations of the images. */
//size of image
CvSize size = cvSize(WIDTH, HEIGHT);
//create images
IplImage *image      = cvCreateImage(size, 8, 1);
IplImage *image_copy = cvCreateImage(size, 8, 1);
//pointer to the pixels in 'image_copy'
uchar *data = (uchar *) image_copy->imageData;
//title of window/gui
cvNamedWindow("Title of window", 1);
```

This code creates two images with the resolution WIDTH x HEIGHT, 8 bits of color values and 1 channel. Black-and-white images only requires one channel, RGB (Red-Green-Blue) images requires three channels. The reason two images are created in this thesis is because the web-camera stores the current frame in the first image-storage and the second image-storage contains the last frame. The variable `data` gives access to the pixels in the image it is pointing to. The $(x,y) = (0,0)$ point in an OpenCV image is in the upper-left corner.

A.2 Capture a frame

Capture and display a frame with OpenCV functions

It is easy to capture and display a frame with the functions from the OpenCV library, but the OpenCV library does not give full access to all the settings what the V4L2 API can access if the camera supports V4L2.

```
//index of camera (-1 = do not care which camera)
CvCapture* capture = cvCaptureFromCAM(0);
//grab a raw frame
cvGrabFrame(capture);
//convert the raw frame to an OpenCV image
image = cvRetrieveFrame(capture);
//copy the image
cvCopy(image, image_copy, NULL);
```

[1] `http://opencv.willowgarage.com/documentation/index.html`

Capture a frame with the V4L2 API

If the web-camera supports V4L2 a lot of settings in the web-camera can be modified, see the V4L2 API specifications[2] which also contains the source code to an example how to initialize the camera, fetch a frame and how to stop the camera.

Bellow is a function which is basically the only code added to the implementation in the V4L2 API example to be able to use the example to handle everything from the initialization of the camera, grabbing the frames and then stopping the camera from streaming. The pixels are stored in a vector.

```
/* Convert frame from a buffer to an OpenCV image. */
static void process_image(const void *p)
{
    size_t x, y;
    uchar *data = (uchar *) frame->imageData;
    unsigned char *pixel = (unsigned char *) p;
    for (y = 0; y < HEIGHT; y++) {
        for (x = 0; x < WIDTH; x++) {
            data[y * WIDTH + x] = pixel[(y * WIDTH + x) * 2];
        }
    }
}
```

The reason to convert the image buffer to an OpenCV images is to be able to benefit from all the functions from the OpenCV library. This function stores a black-and-white image.

A.3 Draw with OpenCV functions

To draw or modify a pixel in a frame can be done manually like this:

```
int black = 0;
data[y * WIDTH + x] = black;
```

The OpenCV library contains functions which can draw and print text:

```
//draw a black line from (x1,y1) to (x2,y2)
cvLine(image, cvPoint(x1, y1), cvPoint(x2, y2), CV_RGB(0, 0, 0),
       3, CV_AA, 0);
//draw a white circle with center point (x,y) and radius 'r'
cvCircle(image, cvPoint(x, y), r, CV_RGB(255, 255, 255), 3, 8, 0);
//write text on image
CvFont font;
double hScale = 0.5;
double vScale = 0.5;
int lineWidth = 1;
cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX, hScale, vScale, 0,
           lineWidth, CV_AA);
cvPutText(image, "To quit press: 'q'", cvPoint(x,y), &font,
          cvScalar(255,255,0,0));
```

More drawing functions and information about the in-parameters can be found on OpenCV's reference pages[3]. A reminder; the functions above will modify the image, so all drawing and similar should be done as a last step before displaying the image.

[2] `http://v4l2spec.bytesex.org/`

[3] `http://opencv.willowgarage.com/documentation/drawing_functions.html`

A.4 The Graphical User Interface

The Graphical User Interface (GUI) is created with OpenCV's standard window functions[4] [5]. The GUI presents the streaming frames from the web-camera as the background and some text and graphical markings are added above the frame. The added texts are e.g. current frame rate per second and available modes to change between e.g. the *OFF* mode or the *BEAM* mode. The GUI also displays error messages that can be helpful for the user. The graphical markings are e.g. the position of the ball and the beam to make it easier for the user to detect that the locating algorithms are working.

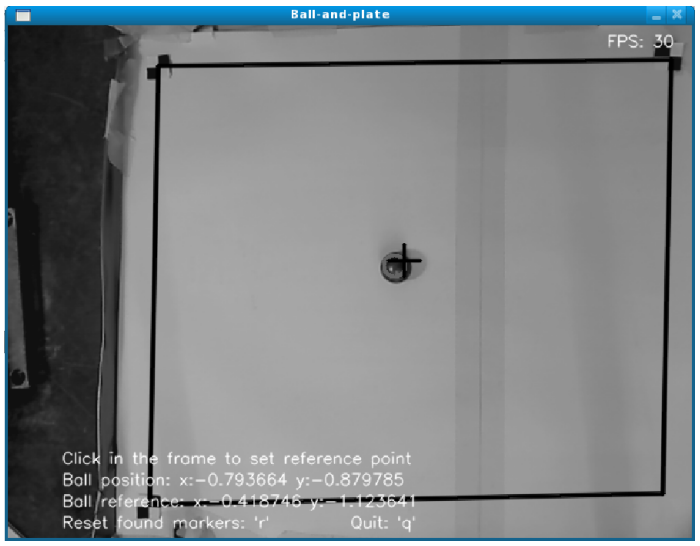

Figure A.1: The GUI for the ball-and-plate process.

The GUI also records any keystrokes made by the user[6] and makes it possible to let the user press a key to control some of the functions in the software. Since the GUI only records one keystroke at the time the available controls through the GUI are somewhat limited, but fulfills the requirements for a simple GUI.

As seen in the flow diagram for the software, see Fig. 3.2, the GUI is updated last in the second thread because then the latest results can be shown and the graphical markings and the text are drawn directly in the same frame copy as e.g. the localization the ball function used.

The following code displays the results from the previous code examples:

```
//display frame
cvShowImage("Title of window", frame_copy);
//wait, required to show the image and save any pressed key
char c = cvWaitKey(10);
if (c == 'q') { //quit
    //close all windows
    cvDestroyAllWindows();
    exit(0);
}
```

If the wait-function is set to under 10 milliseconds the window will probably not update. The `cvWaitKey(int delay=0)`-function also stores any pressed key.

[4] `int cvNamedWindow(const char* name, int flags)`
[5] `void cvShowImage(const char* name, const CvArr* image)`
[6] `int cvWaitKey(int delay=0)`

A.5 Control the web-camera with V4L2 API code

The code below is the most important modifications made in the V4L2 API example. The purpose of this C code is to make it possible to get the maximum 30 frames per second from the web-camera Logitech Quickcam Pro 9000, without any conditions for the light.

The code is divided in to three parts, in the first part the frame rate is set. The `ioctrl` function is a low-level communication function to communicate with the web-camera. The light exposure mode is set to make it possible to set the priority of the light exposure in the last part.

These initializations are made before the web-camera starts to stream video frames.

```
//set frame rate
streamparm.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
streamparm.parm.capture.timeperframe.numerator   = 1;
streamparm.parm.capture.timeperframe.denominator = FPS;
if (-1 == ioctl (fd, VIDIOC_S_PARM, &streamparm)) {
    switch (errno) {
        case EINVAL:
        perror("Parameter setting not supported");
        break;

        default:
        errno_exit ("VIDIOC_S_PARM");
        break;
    }
}

//set exposure mode
control.id = V4L2_CID_EXPOSURE_AUTO;
control.value = 3;
if (-1 == ioctl (fd, VIDIOC_S_CTRL, &control)) {
    perror("EA Parameter setting not supported");
}

//disable auto exposure
control.id = V4L2_CID_EXPOSURE_AUTO_PRIORITY;
control.value = false;
if (-1 == ioctl (fd, VIDIOC_S_CTRL, &control)) {
    perror("EAP Parameter setting not supported");
}
```

B. The source code

B.1 Subversion

As a results of this thesis five different implementations are available. The source code is available through a subversion[1] server at the Department of Automatic Control[2], LTH, Lund University.

A manual for the ball-and-beam software can be found in Appendix B.2. The manual is based on the solution with the geometric three-marker method, see Section 4.6. A manual for each version of the program is not necessary since the programs are very similar. The most significant difference is that the geometric three-marker methods requires a camera calibration and the encoder-style detection version require a manual angle calibration.

To check out the source code type:

```
svn co https://www2.control.lth.se/svn/L004-RBBotFleet/
```

In the new folder *L004-RBBotFleet/trunk/ball.and.beam/* can the different versions of the implementations be found.

To compile a Java or a C code implementation type:

```
make
```

and to run the Java code:

```
java Main
```

and for the C code type (if the current folder is the *src* folder):

```
../bin/./ball.and.beam
```

Below is the different versions of the implementations described shortly, the titles are the names of the folders:

ball.and.beam_java

This folder contains a pure Java solution with all the feedback received through the Comedi interface. The source code is originally from the course FRTN01 Real-time Systems at LTH, Lund University. The real-time aspects for both the Java and the C implementations can be found in Section 3.5.

The GUI allows the user to set the control parameters during run-time and presents graphs of the position of the ball and the output signal.

ball.and.beam_java_c

The solution consists of two programs, one implemented in Java code and one implemented in C code. The C program runs the vision algorithms and sends the position of the ball to the Java program. The GUI for all the C implementations, even the ball.and.plate, are basically the same as in Fig. A.1.

The Java program is the same as in the ball.and.beam_java solution, except that the feedback of the position of the ball is received from the C program instead of the Comedi interface. The communication between the Java and the C program is send through a simple socket.

ball.and.beam_c_angle_from_comedi

The third solution is a pure C program. The C code contains vision algorithms which locates the position of the ball. The feedback of the angle of the beam is received from the Comedi interface.

ball.and.beam_c_encoder_style_detection

The folder contains a pure C code implementation with all the feedback from the vision algorithms. The angle of the beam is calculated by the encoder-style detection algorithm with

[1] `http://subversion.tigris.org/`
[2] `http://www.control.lth.se/`

1+3 set strips, see Section 4.6. The angle *a*, see Fig. B.1, must be calibrated and adjusted in the source code.

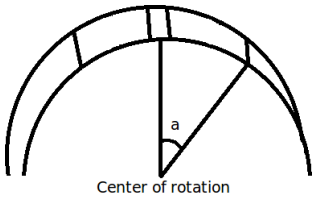

Figure B.1: The angle, *a*, between the sets of markers.

ball.and.beam_c_geometric_three_marker_method

This is the final version of the ball-and-beam software implementation and is a pure C program. All the feedback is calculated by the vision algorithms. To calculate the angle of the beam the geometric three-marker method, see Section 4.6, is implemented. Much more about this program can be found in Appendix B.2.

camera_calibration

This folder contains the camera calibration program implemented in C++ code. The code is copied from the source code of OpenCV 2.0.0. The camera calibration is required for the geometric three-marker method. More about the camera calibration can be found in Appendix B.2.

ball.and.plate

To control the ball on the plate two programs are required. The first is the C program which contains the algorithms for locating the plate and the ball. The second part is a Matlab Simulink model, the model contains the controllers. See Appendix B.3 for more details.

B.2 Manual for the ball-and-beam process

This manual is mainly for the *ball.and.beam_c_geometric_three_marker_method* program, see Appendix B.1, but can be used for the other C implementations with some modifications. The feedback is from the vision algorithms and the angle of the beam is calculated by the geometric three-markers method, see Section 4.6.

Warning

Be always prepared for that the beam might move uncontrollably at any point during *BALL* or *BEAM* mode. There are a couple of safety functions implemented, but they are not fail proof. Reasons for the possible failures can be: erroneously detection of a marker or the ball, due to e.g. changes of the light conditions. Also an automatic update of the operative system or other heavy processes can make the program to fail.

Introduction

To be able to run the *ball.and.beam* program the OpenCV library must be installed and the camera is required to have support for the V4L2 API. Also a camera calibration is required, see Appendix B.2. The results from the camera calibration are manually written in the source code, *ball.and.beam.c*.

The camera should be placed directly above the beam. Most camera sensors have higher resolution (more pixels) on the width then on the height, so the camera should be placed so the beam is represented by as many pixels as possible to minimize the quantization created by the pixels. Also place the camera so the frames are not upside-down on the monitor.

These are the main functions of the software:

- Capture video images from the web-camera.

- Locate the beam, the ball, the visual set-point and calculate the angle of the beam from the frames.
- Control the angle of the beam in order to control the position of the ball.
- Convert the output signal to communicate with the ball-and-beam process.
- Provide a simple GUI.

If the camera not already is calibrated see Appendix B.2. A general flow graph of the software can be seen in Fig. 3.2.

Compile the program with:

```
make
```

and to run it type:

```
../bin./.ball.and.beam /dev/videoX
```

The path to the camera `/dev/videoX` is only required if the path to the camera is not `/dev/video0`, which is the standard path to the first mounted camera.

Before the program enters the main infinite loop the three markers on the beam must be found once. If the three markers are not found the program will run a small infinite loop until three markers are found. The GUI will print *Searching for beam* until the three markers are found.

Camera calibration

To be able to run the *ball.and.beam_c_geometric_three_marker_method* software the focal length and the principal point must be found. These parameters are required for the final ball.and.beam program, because the geometric three-marker method, see Section 4.6, depends on these parameters. The camera calibration can be made with the *Camera Calibration Toolbox for Matlab*[3] or any other similar tutorial.

An alternative solution, which does not require Matlab, is to use a camera calibration available from the source code of OpenCV 2.0.0[4], the same program is copied to the subversion server, see Appendix B.1. All functions in the program and much more about other functions are presented on OpenCV's camera calibration homepage[5]. For the camera calibration a checkerboard is required.

OpenCV's camera calibration is compiled with:

```
./build_all.sh
```

and can be run with the following parameters:

```
./calibration -w 6 -h 8 -n 10 -o camera.yml -op -oe
```

The parameters `-w` and `-h` specifies the number of inner corners of the checkerboard vertical and horizontal, `-n` number of frames, `-o` the output file, `-op` detected feature points and `-oe` extrinsic parameters detection.

The first step to calibrate the camera is to put the checkerboard in front of the camera and start the calibration program. When all the inner corners have been found by the software the current frame is stored and the search for all the inner corners restart. To get good results the checkerboard should be moved around and have different angles for all stored frames.

As result of the camera calibration the file *camera.yml* is created. The file contains the focal length, the principal point and much more. The file can be opened as an ordinary text-file and then a vector *data* under *camera_matrix* can be found. The vector can be represented as a matrix which contains the following parameters:

$$data: \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

The parameter f_x and f_y are the focal lengths and the c_x and c_y are the coordinates for the principal point. If the beam is more horizontal than vertical in the frame, use the x-values. The unit of the results are pixels.

[3] `http://www.vision.caltech.edu/bouguetj/calib_doc/`

[4] `http://sourceforge.net/projects/opencvlibrary/files/` folder path: samples/c/calibration.cpp

[5] `http://opencv.willowgarage.com/documentation/camera_calibration_and_3d_reconstruction.html`

Manual calibration of the camera

When the Graphical User Interface (GUI) starts two manually calibrations are required. First make sure that the beam markers are inside the GUI even when the beam rotates. Turn off the power to the ball-and-beam process to make it much easier to rotate the beam by hand.

The second calibration is to make sure the camera sensor is parallel to the beam when the beam is horizontal. First find the angle of the beam there it is horizontal by putting a ball on the beam and find the angle where the ball is stationary. Now the calculated value of the angle should be zero, the program prints the calculated angle of the beam in the terminal. If the angle is not zero tilt the camera sideways until the requirement is fulfilled.

The graphical user interface

The GUI consists of a window with the video stream from the camera. If the three markers are found in the current frame lines are drawn between the markers. The position of the ball is marked with a black circle and the predicted position by the Kalman filter is drawn as a grey circle. In the upper-right corner the number of frames per seconds are printed and are updated once every tenth seconds. The GUI for the ball-and-plate process is very similar and can be seen in Fig. A.1.

In the bottom half of the window are some instructions how to control the program, e.g. what key to press to change from OFF mode to *BEAM* mode. There are three different modes. The OFF mode turns off the output, in *BEAM* mode the angle of the beam is set to zero. In *BEAM* mode the regulator controls the position of the ball.

If the ball is not found for a third of a second and the regulator is in *BALL* mode the program will change the regulator to OFF mode to avoid the beam to rotate unpredictably.

Demo mode The reference point for the position of the ball is set to the middle of the beam if the *demo* mode is not activated. If *demo* mode is activated the reference point for the position of the ball is set visually. In the GUI two lines are drawn near the beam, between these lines the software is searching for a finger or similar. When a finger is found the position of the finger is the new reference point for the ball position.

The algorithm for detecting the visual reference point is basically the same as the edge detection algorithm for the localization of the ball. A detected finger is drawn as a black circle and an old reference is drawn as a grey circle.

B.3 Manual for the ball-and-plate process

Warning

The robot which controls the ball-and-plate process can move very fast and unpredictably. Never be in range of the robot when it is activated and be always ready to activate the emergency stop.

Introduction

The ball-and-plate process consists of two plates, see Fig. 4.2. The plate in the bottom is where the ball is placed. The plate straight above is connected with the lower plate by metal bars on the side. Under the upper plate the camera is placed facing down to the plate below. On the top of the two plates a special tool is screwed on to the upper plate. The special tool makes it possible to fasten the two plates to the robot. The robot used in this thesis is an IRB2400 and has six joints. Two joints are required to tilt the plate to control the position of the ball in two dimensions.

The ball-and-beam program consists of two separate programs. The first program is a program written i C code which locates the plate and the position of the ball. The other program is a Matlab Simulink[6] model which consists of the controllers required for controlling

[6] `http://www.mathworks.com/products/simulink/`

the position of the ball. The communication between the C program and the Simulink model is handled by LabComm[7].

It does not matter if the Simulink model or the visual program starts first. The visual program tries to connect to the Simulink model once every second. The Simulink model receives only zeros for the ball position and the ball reference, which is in the middle of the located plate, until the LabComm connection is established.

The Simulink model

How to run the Simulink model and how to e.g. log signals can be found on an internal wiki-page[8] available for the Department of Automatic Control at LTH, Lund University. The Simulink model is presented by a simple GUI and called *OpCom*. OpCom gives the ability to change parameters for the process, most of the parameters for the controllers are available through here, see Fig. B.2.

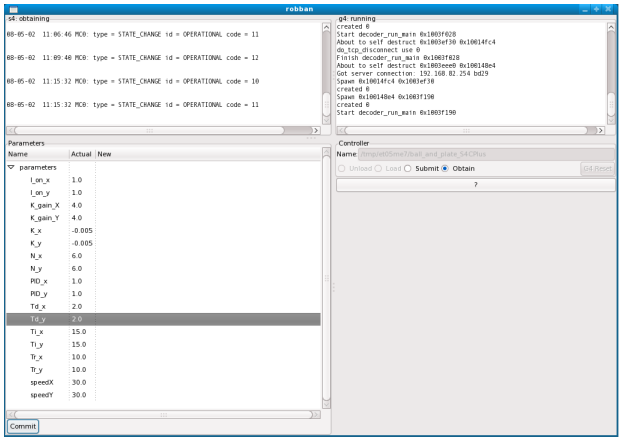

Figure B.2: The Opcom GUI.

The angles of all the joints of the robot is available in the Simulink model and can be used for the control, Fig. B.3 shows a screenshot of the model. When the model is started the program moves three of the joints on the robot to their predefined positions. When the predefined positions are archived by the controllers the plate is flat so a ball can be placed stationary on the lower plate. This is controlled by two controllers, the third controller rotates the plate so that the other two joints only controls the plate in one dimension each.

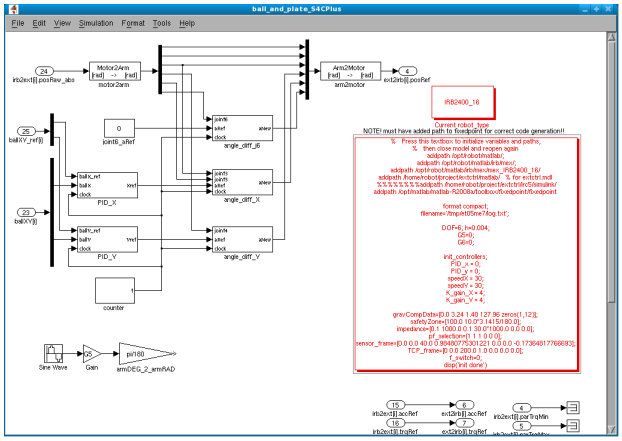

Figure B.3: A screenshot of the Simulink model.

[7] `http://torvalds.cs.lth.se/moin/LabComm`
[8] `https://www2.control.lth.se/internal/ML_robotlab`

The vision part

The program which takes care of the localizations of the plate and the ball is originally the same as the ball-and-beam program. The most significant changes are that the algorithms for localing the ball iterates for the ball in two dimensions instead of just along the beam and the controllers and the algorithm for calculating the angle of the beam are removed.

Compile the program with:

```
make
```

Start the vision program with:

```
./ball.and.plate /dev/videoX
```

The path to the camera `/dev/videoX` is only required if the path to the camera is not the standard path, `/dev/video0`.

Before the program enters the main infinite loop the three markers on the plate must be found at least once. If the three markers are not found the program will run a small infinite loop until three markers are found. The GUI will print *Searching for beam* until the three plate markers are found.

To better handle different light conditions the threshold for the edges of the markers are set dynamically. If a marker is missed the threshold is lowered and is then raised slowly again to minimize erroneous detections of the markers. There are hard-coded minimal and maximal values for the threshold.

The host address and the port number to the LabComm connection is hard-coded in the source file *babXY_labcomm_driver.c*.

The GUI The GUI for the ball-and-beam can be seen in Fig. A.1. The GUI works similar to the GUI used to the ball-and-beam process, see Appendix B.2, but with two significant changes. By left-click with the mouse in the GUI the reference point is set, if the mouse bottom is hold down the reference point is updated when the mouse is moving. The reference point is set to the middle of the located plate from the beginning and a new reference point is marked with a cross. The other change is that the GUI does not have the ability to set the mode of the controllers since the controllers are implemented in the Simulink model and are modified by through OpCom.