Swinburne University of Technology

Hand-Eye Coordination Report

RME 40003

Robot System Design

Justin Sargent	9989900
Phillip Smith	7191731
Thomas Rappos	6336361

Contents

Chapter 1

Project Outline

This project explores the use of a robotic arm used for pick-and-place automation tasks. For dynamic operation and task-completion feed-back, computer vision is incorporated via a web-cam mounted in the work area.
More specifically, this project has been completed for an industry client, AME System Pty. Ltd., for automated cavity pin insertion. These pins are 3.5mm wide and are placed in holes between 5 and 9mm apart on a rubber socket, as shown in fig. 1.1(a) Due to this small size, a high degree of accuracy and repeatability is required.

To hold the socket in place, and determine the orientation, a holding box

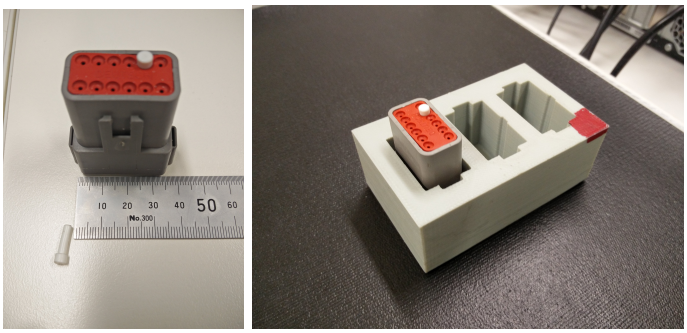


Figure 1.1: Image of pin and socket, and socket in holding box

has been produced which slots the sockets into a fixed position, shown in fig. 1.1(b). For this project, these boxes are not themselves placed in a fixed location within the work space. As such the main objective of this project is using the computer vision to determine the global position of the pin holes, via the fixed position of the socket in the box, and the global position and orientation of the box in the work space.

For this project, the confirmation of pins being ready in the supply is not being explored. This is due to the industry application being intended to use a vibrating bowl feed, and thus the assumption can be made pins will always be available. For this proof of concept, a rack of three pins is made, with the gripper positions being hard coded.

Chapter 2

Computer Vision

The computer vision of this project uses Open-CV for c++. This library contains many image-based functions for sourcing, manipulating, and presenting visual data. This library has been used for two roles, object detention, and robotic gripper required motion determination. Of these two task, the latter relies on the former.

Object detection

Environment preparation

To begin the computer vision process a background image is taken when no objects are present. A keyboard control was added, 'Y', which allows a new background to be set. This was seen to be useful, as a background image taken on start-up, would often be too bright. This was seen to be due to the automated settings of the camera not having finished calibrating. It was also added that this background was saved to file, so the system could be stopped, and restarted without needed to take a new background. However, it was advisable to take a new background every session, as lighting condition changes would cause issues with the object detection reporting false positives.
With the background set, a psudo-live stream of stills was captured from the camera. Each of these stills was then processed, as follows, before being discarded.

Object recognition

The first step of processing was to do a raw matrix subtraction of the background. This gave an image of only the objects not in the background image. For this subtraction to function correctly, the background image needed to be free of all free-moving objects, as this subtraction would recognise a lack of object as well. An example of this subtraction is shown in fig. 2.1.
This image was then converted to a binary matrix, with a threshold of each colour layer, and a a grey scale process, to combine the colour layers. This process is shown in fig. 2.2.

The resulting image was then eroded and dilated by one pixel layer. This erode function turns any white pixel touching to a black pixel into a black pixel, while the dilate function performs the opposite. This resulted in any small imperfections being removed from the image, while only slightly reducing the object recognition quality of the image. This is seen in fig. 2.3(a).

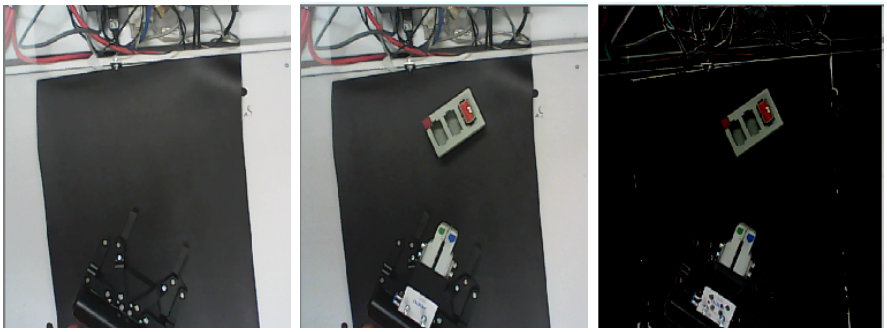


Figure 2.1: Background image, new capture with objects in place and resulting subtraction

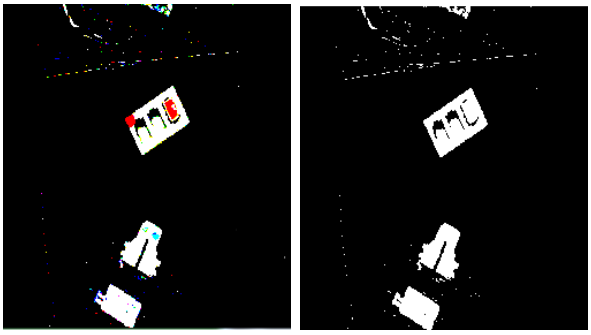


Figure 2.2: Threshold of each colour, and combining of colour layers to make single layer binary

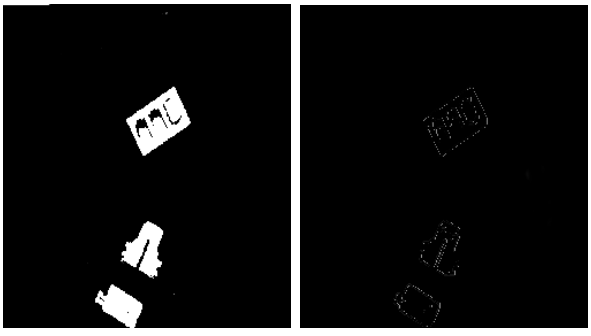


Figure 2.3: Erode and Dilate removes static from image, and Canny edge detection leaves only the outline of the objects

A Canny edge detection was then used, which finds object edges by leaving only white pixels that are touching black pixels behind. The resulting image is shown fig. 2.3(b).

Each group of remaining white pixels was now classified as a contour, via the *findContours* command. This process found any collection of white pixels as a polygon shape. It also listed the contour hierarchy, where contours are set as parents, and any contours inside its pixel range are defined as its children. The main holding box, and the gripper, are both objects not encased in larger objects. Therefore, only contours with no parents were further examined.

Each contour examined was fitted inside a rectangle with *minAreaRect*. This found the smallest rectangle, of any angular orientation, that could fit the whole contour. The size of this rectangle was then tested, anything outside the set threshold was ignored.
For each rectangle that was within the predicted size of the holding box or gripper, a classification and orientation was attempted as if the object were a holding box. If this process failed, the image was oriented as a gripper. If this failed, the object was deemed neither and discarded. From the example process, at the stage in fig. 2.3(b), three objects remain; the holding box, the gripper and another part of the gripper of no interest. Respectively, these objects would be assigned as a holding box; fail at being set as a holding box, and be assigned as a gripper, and fail at both assignings.
This process is shown in diagram in fig. 2.4.

Figure 2.4: Assigning process through try, catch, throw

Object orientation

Both the holding box and gripper are recognised in this process as *RotatedRect*s these are open-cv objects which consist of a size, angle and centre coordinate. However, the four corner points can be extracted from the object.

When orientating as a holding box, each of these corners has the surrounding pixels examined, with the corner that has the highest ratio of red being deemed the reference corner. This reference corner is shown in figure fig. 2.5. To derive the highest red ratio eq. (2.1) was used. By referencing the corner with the most red *ratio* rather than highest red mean, a white corner is not incorrectly identified. If a corner of a set red ratio value is not found, the classification as a holding box is aborted, and a gripper is tried, as discussed above.

$$mean(red)/(mean(blue) + mean(green) + mean(red)) \tag{2.1}$$

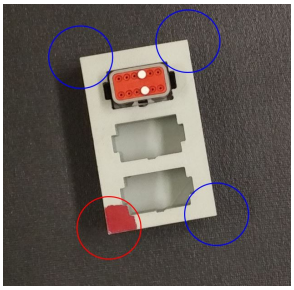


Figure 2.5: Red detection for reference corner

This reference corner was then used to determine the holding box true angle in relation to the camera. This is required as the angle of the RotatedRect is only in the range 0° to -90°. Shown in fig. 2.6, the RotatedRect is initially a short, wide rectangle at -20°. However, when rotated by -80° the RotatedRect becomes a tall, thin rectangle at -10°. To rectify this, the index of the reference corner from the list of corners returned from the RotatedRect function was examined, and the true angle was found with an addition of 0, 90°, 180° or 270°.

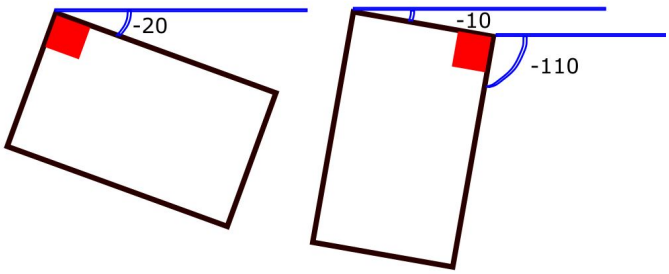


Figure 2.6

This orientation process is repeated for the gripper arm, but using the blue and green dots on the gripper fingers.

Pin detection and PinHole Allocation

With the holding box true angle determine, a copy of the image could be rotated and cropped to be only the holding box. This cropping can be seen in fig. 2.7.

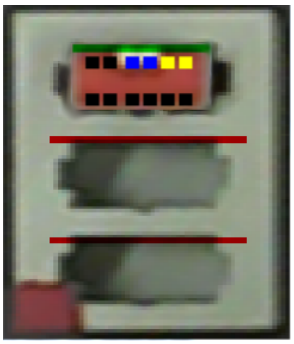


Figure 2.7: Holding box image, cropped down to only the box, and with all socket overlay shown

Data on this holding box could now be used to find the sockets in the holding box.[1] This information defines how large the box is in millimetres, so a pixel per millimetre ratio can be found. The box information also defines where the sockets should be (in millimetres), within the box.
By using the ratio, and the physical displacement of each socket, a *socket* object was made. Each socket object was passed the holding box image, its pixel starting point within this image, expected hight, and file name for CSV file with pin details. The socket started by cropping a reference matrix of the box image to only the useful area. Using reference copies, or *shallow copies*, has two main advantages: Firstly, the memory used and copy execution time is reduced, as the socket's stored *Mat* is only a pointer, not a whole new matrix. Secondly, the socket image can be edited for user feedback, and the original holding box image is effected. This allows only a holding box to be presented to the user with all socket information being shown, rather than loading each socket image separately. An example of this is shown in fig. 2.7.
Using its sub-matrix, the socket examined the mean red ratios, as used for the holding box corner, and determines if a socket was present, as shown in fig. 2.8.

[1] At this point, hard coding has been used, however if several holding boxes are used in future development, a bar code system will be introduced for identification and CSV extraction use.

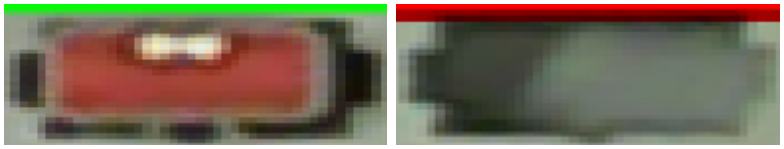
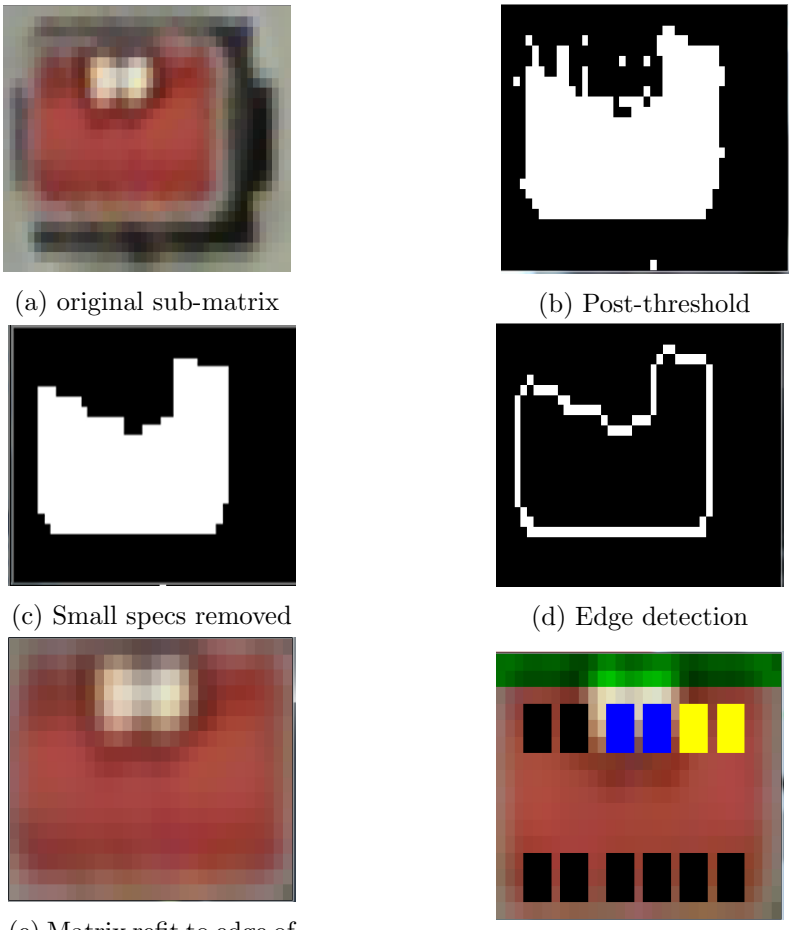


Figure 2.8: Active and empty socket place identified and presented with, respectively, green or red bar above

Sockets that have been identified as present are then further examined to find the exact area for pin hole mounting. Shown in fig. 2.9a is the original socket image. A threshold function of high red, low green, low blue is performed, fig. 2.9b. The image is eroded and dilated, fig. 2.9c, and a canny edge detection is conducted, fig. 2.9d. The image is then cropped to fit the resulting contour, fig. 2.9e.

Using a CSV dataset, the locations of the pins is then assigned. The newly allocated pin hole is examined to find if a pin is present. To do this, the mean value of all three colour channels is compared with the 3 mean channels of the socket, if it is 25% greater, a (white) pin is determined to be present.
A second CSV file then provides information on which pin holes should be filled. The resulting pin layout is shown in fig. 2.9f, black pin holes are not filled, and should not be, blue have already been filled, and yellow still need to be filled.

may be able to add stuff here if better res camera allows us to have an extra layer of positioning



(a) original sub-matrix

(b) Post-threshold

(c) Small specs removed

(d) Edge detection

(e) Matrix refit to edge of red area

(f) Pins identifed

Figure 2.9: Image processing of socket

Path creation

Gripper to pin path creation

With the pin holes assigned, the next available pinhole is requested from the main function. To do this, each layer between the main image and the pin holes are re-framed within their parent.
To begin, the first pinhole that should be filled, but not currently filled (coloured yellow above) reports to its parent socket the location within the socket. The socket then combines this location, and the location of the socket within the holding box, and reports this value to its parent holding box.
The holding box then re-orientates this coordinate to the global frame, and combines it with the holding box's location in the global frame. This final,

global position of the pin is then reported to the main function. A diagram of this process is shown in figure fig. 2.10.

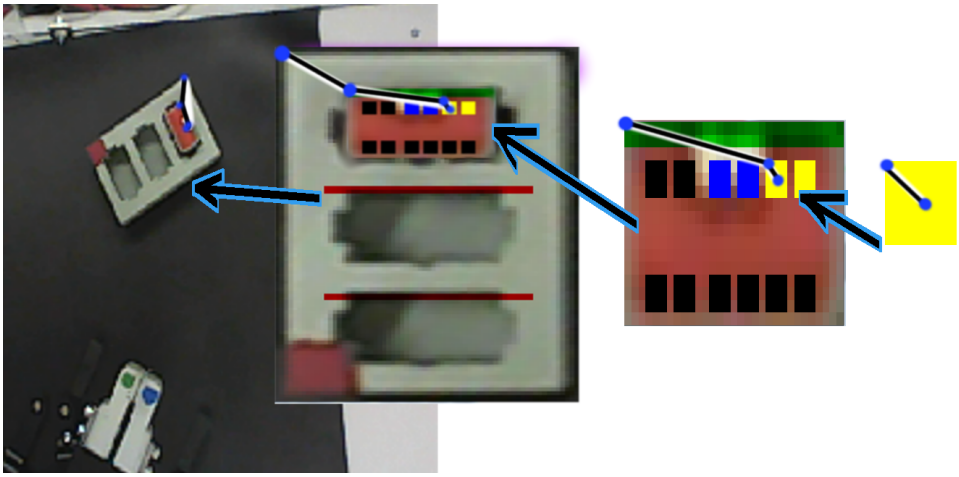


Figure 2.10: Local position, cascading to parent to find global pin position

The gripper single point coordinate is then determined by taking the half-way point between the green and blue corners of the gripper.

With the two singularity points for the gripper and pin identified, a vector is derived between them. This vector is given a magnitude and angle, and displayed to the user, as shown in fig. 2.11.

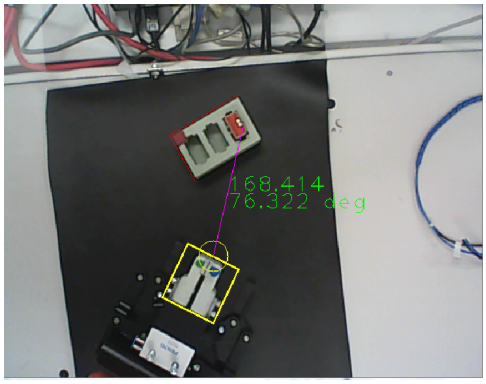


Figure 2.11: Gripper to pin hole vector display

Measurement correction

As this application is being deployed under fluorescent lighting, there is some oscillation in lighting of the holing box and gripper. From this, the measurements recorded above cannot be assumed to be completely accurate.

The graph shown in fig. 2.12 shows that if the above *object detection* and *path measurement* process is repeated 30 times, in quick succession, there is a range of values recorded.

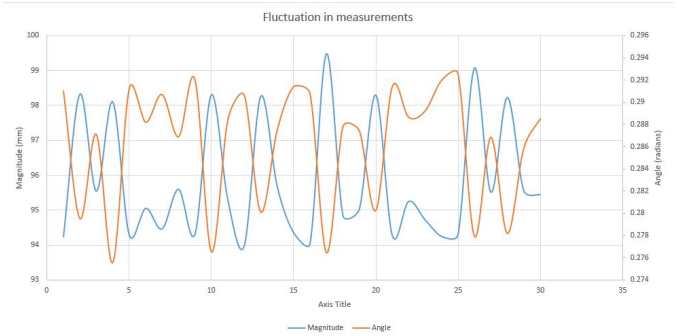


Figure 2.12: Vector magnitude and angle change over 30 records in close precession

To reduce this lighting issue, the maximum and minimum values observed in this graph are taken, and averaged to derive an error reduced measurement.

Path to gripper arm corrdinates

Vector converted to gripper coordinates.

Chapter 3

Robotic Arm Control

The Denavit-Hartenberg algorithm above gives a close approximation to the movement and joint rotation behaviour of the ABB robot arm. Due to the nature of elbow and wrist rotations, a 'd' offset is given for links 4 and 6, allowing a rotation about the length of a previous link.

Matlab Simulations

The Matlab Robotic, Vision and Control library (RVC) is useful for mathematically representing configured links on a robot arm using Denavit-Hartenberg notation. When using the algorithm, links can be created by a user and added to a defined Robot object using the SerialLink() function. From here, a single position may be mapped using an array of elements equal to the amount of links. Each element represents either joint translation for prismatic joints or rotation for rotary joints. The order in which the joints are represented in the array need to be the same order in which the links were created in the Denavit-Hartenberg algorithm.

Open ABB

For this project, a basic parallel port was available for communication between the computer vision and the robotic arm. The parallel port allowed for single bits to be used as flags which triggered the robotic arm to perform action sets. For this project we needed to send specific travel distinations from the computer vision, to the robtic arm, therfore a more advance communication was required. To achieve this, OpenABB [1] was used. This system has the robotic controller running a basic TCP server via RAPID code. A python script is used on the computer to connect to this server,

[1] $https://github.com/robotics/open_abb$

Table 3.1: Denavit-Hartenberg Algorithm

Links	θ	d	a	α
1	θ	0	0	0
2	$\theta 2$	0	0.07	$\pi/2$
3	$\theta 3$	0	0.36	0
4	$\theta 4$	0.38	0	$\pi/2$
5	$\theta 5$	0	0	$-\pi/2$
6	$\theta 6$	0.065	0	$\pi/2$

and send commands. One such command can move the gripper to any point in the arms range/ field of operations, something like that by sending the three axis coordinates.
This command list was extended by the team to include gripper control and moving about the x and y axis, relative to the current position. An example of the command stack is shown in fig. 3.1.

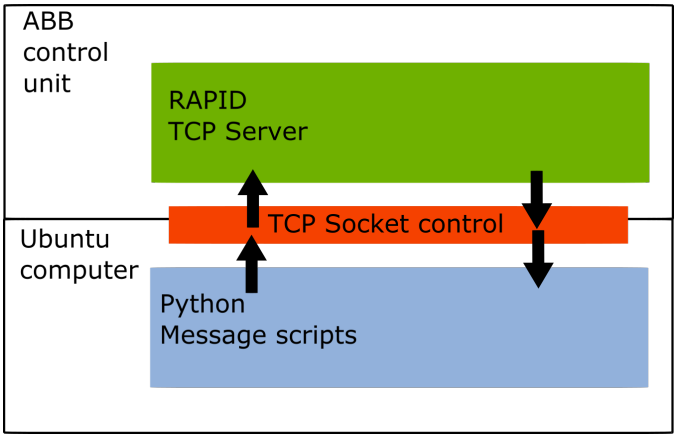


Figure 3.1: Communication layers of OpenABB

Coordination

As the computer vision was made in c++, and the openABB c++ code requires a full ROS installation, it was instead chosen to implement a python calling system in c++. This interface called a python run-time, and sent commands via $PyRun_SimpleString$.
The command stack was thus extended as shown in fig. 3.2

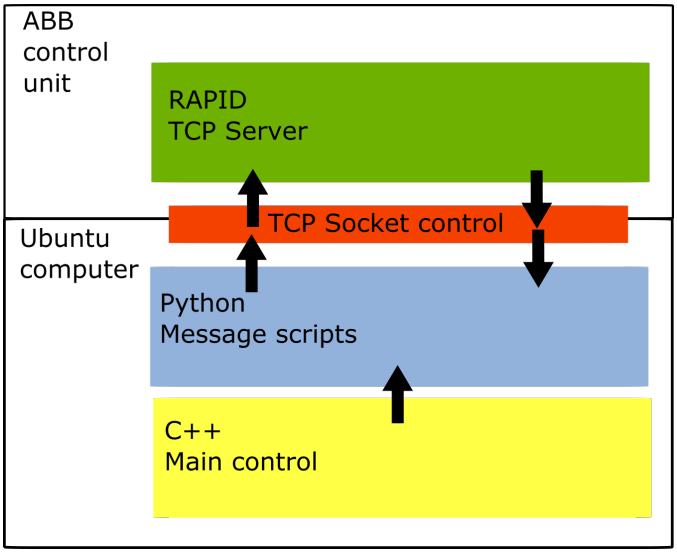


Figure 3.2: Communication stack with added c++

Chapter 4

Operation Flow

With the system's ability to know where to move to, and the ability to coordinate these movements, the operation flow charts can now be presented. Figure 4.1 shows the standard operation flow, with purple action points being computer vision processes, and pink being robotic arm actions. Figure 4.2 shows the operation for calibrating the robotic arm frame, and the camera view frame.

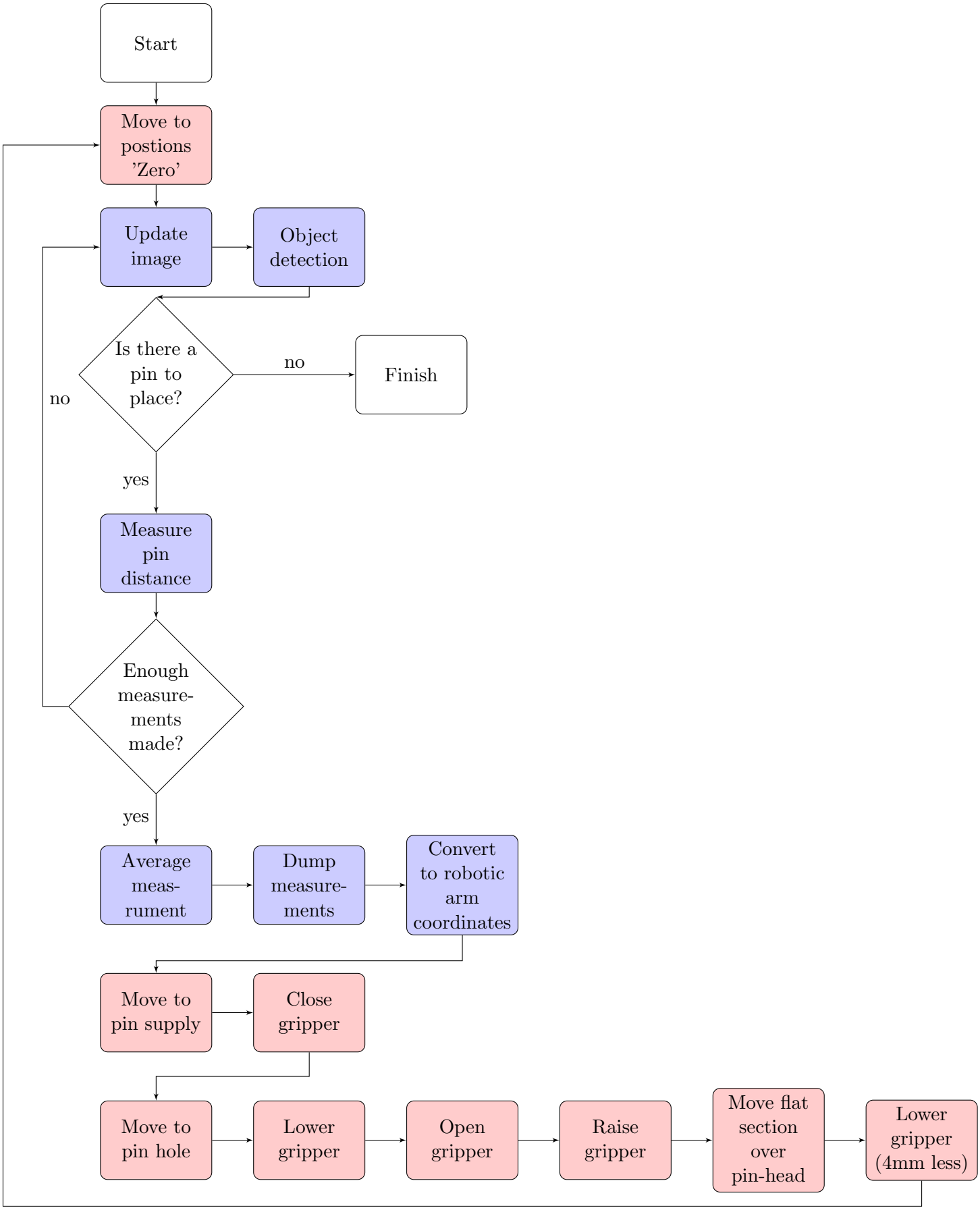


Figure 4.1: Main Process

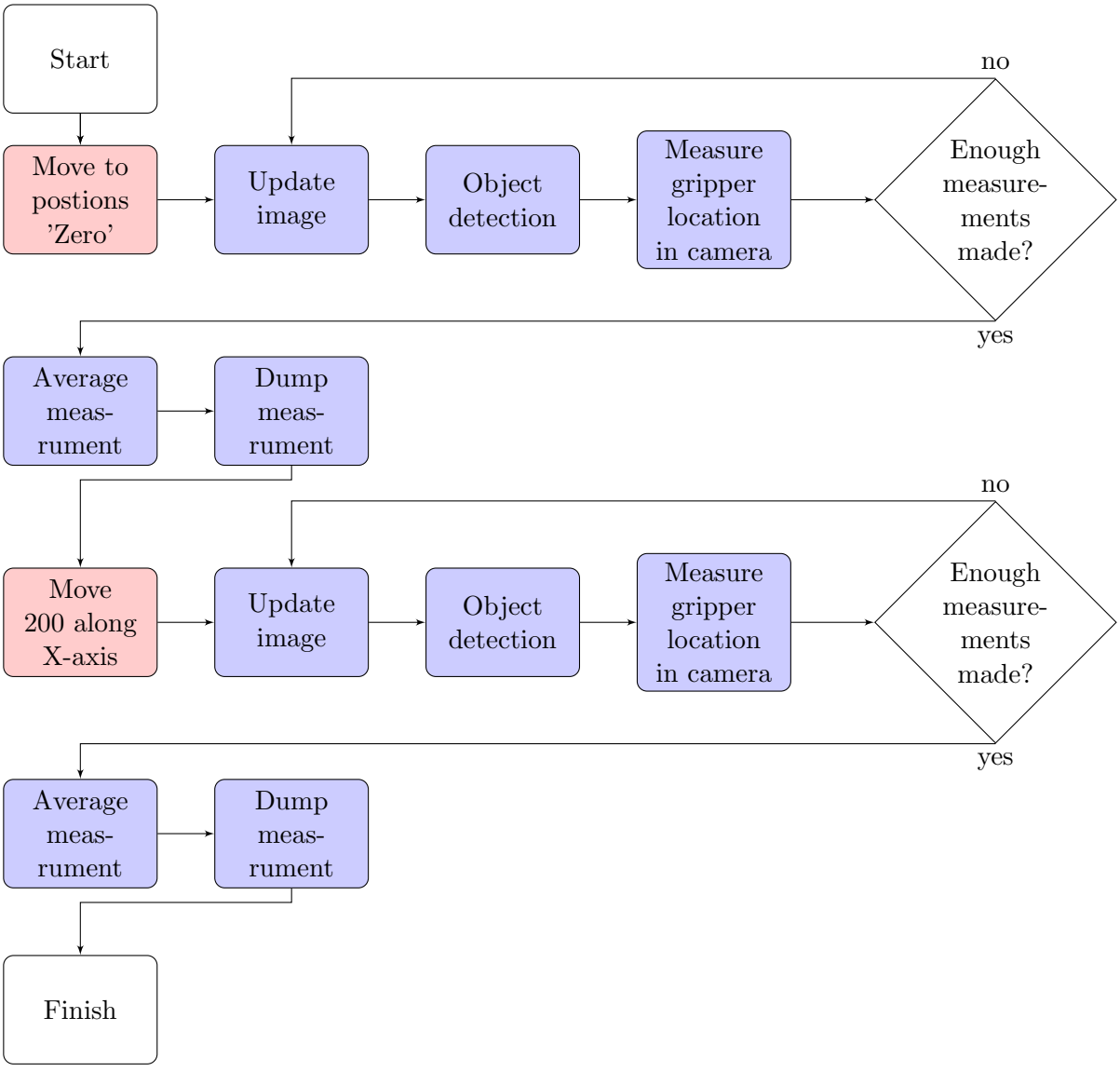


Figure 4.2: Calibration