Swinburne University of Technology

Hand-Eye Coordination Report

RME 40003

Robot System Design

enter names and numbers

Justin Sargent	00000
Phillip Smith	7191731
Thomas Rappos	00000

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 plug insertion. These plugs are 1.5mm wide and are placed in cavities between 2 and 4mm apart. As such, a high degree of accuracy and repeatability is required.

double check size

include image of plug

Chapter 2

Gripper Design

With such small objects being handled, a custom gripper device was developed. The main gripper mechanism uses a Festo parallel, nematic gripper. While the gripping appendages have been custom made for this operation. Shown in fig. 2.1 are the open and closed CAD renderings of the gripper. Dark-grey being the Festo gripper, light grey being the main gripper appendage and green being the 3D printed gripper tips, modelled to specify grasp the cavity plugs.[1]

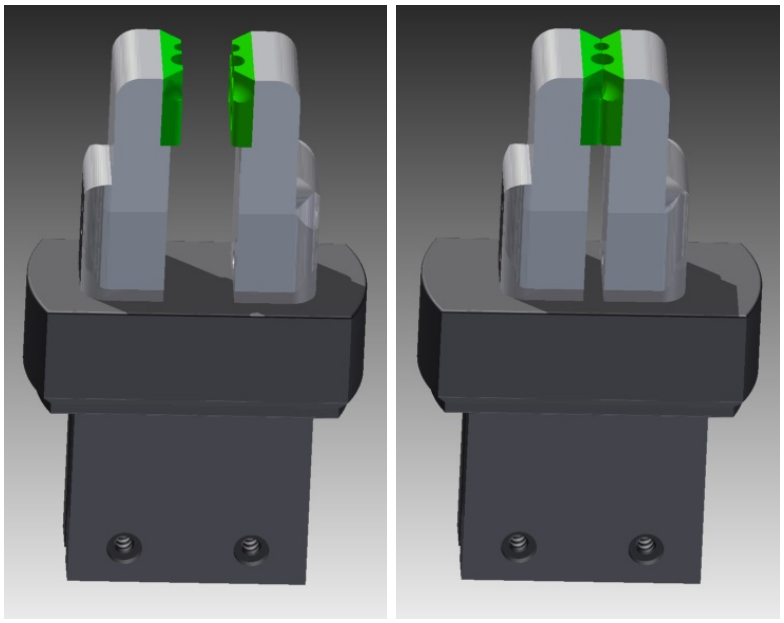


Figure 2.1: CAD rendering of plug gripper

[1] two cavity plug sizes were initially requested

Chapter 3

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.

3.1 Object detection

3.1.1 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.
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.

3.1.2 Object recognition

The first step of still 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. 3.1.

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. 3.2.

The resulting image was then eroded and dilated by one pixel layer.

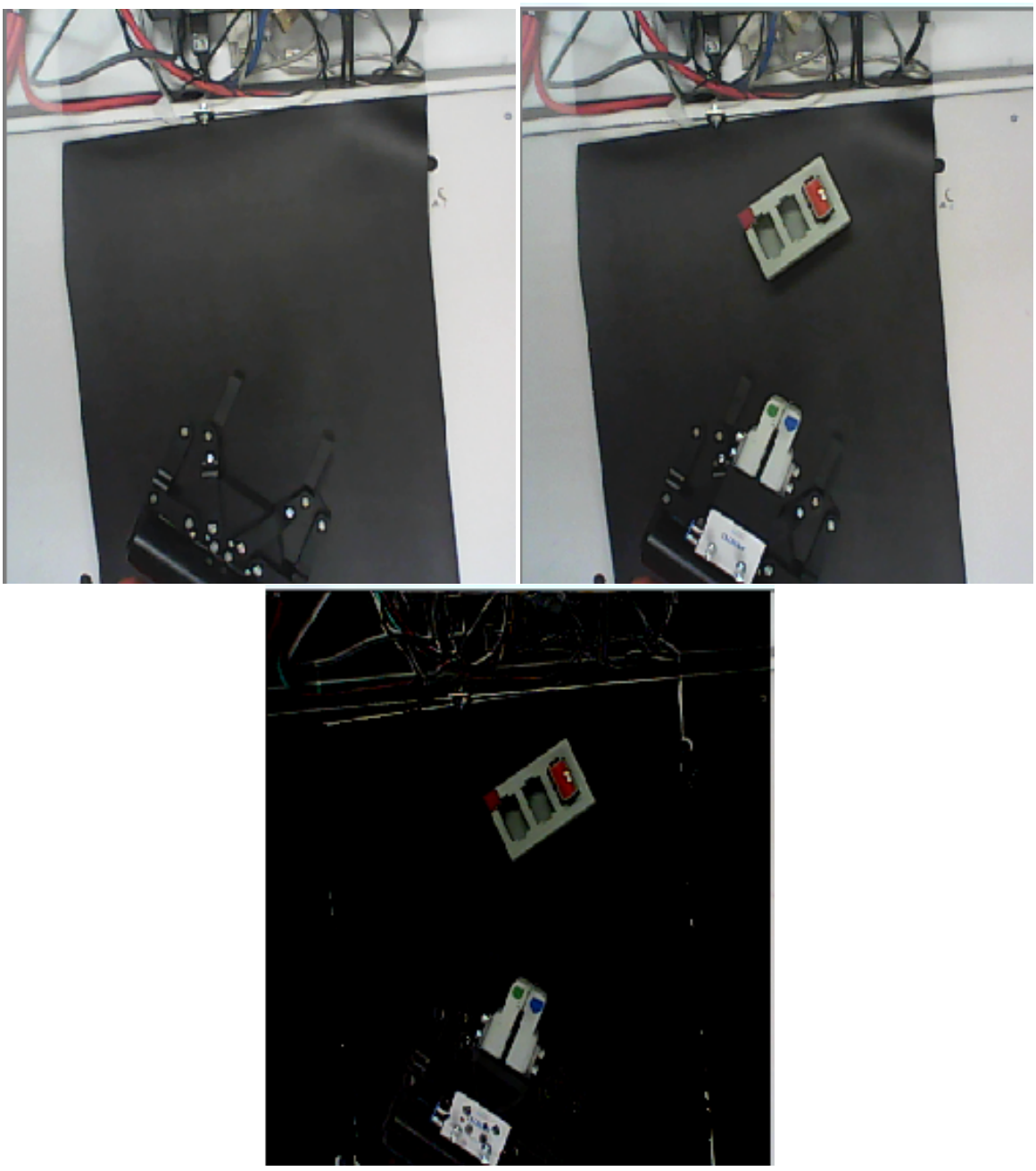


Figure 3.1: Background image, new capture with objects in place and resulting subtraction

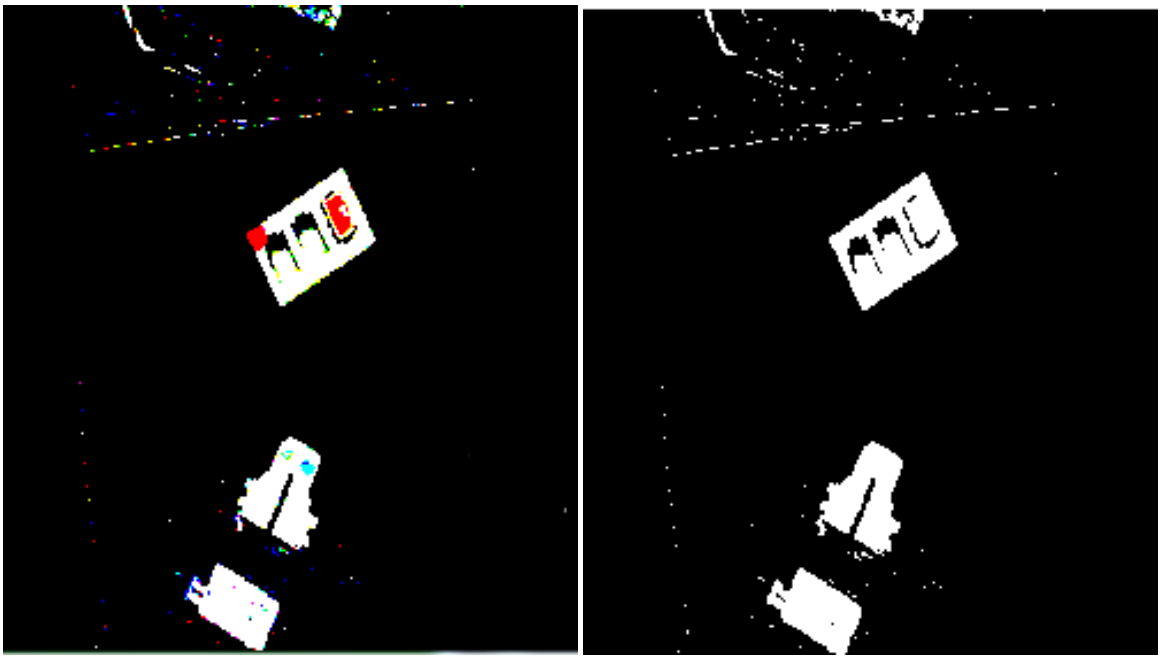


Figure 3.2: Threshold of each colour, and combining of colour layers to make single layer binary

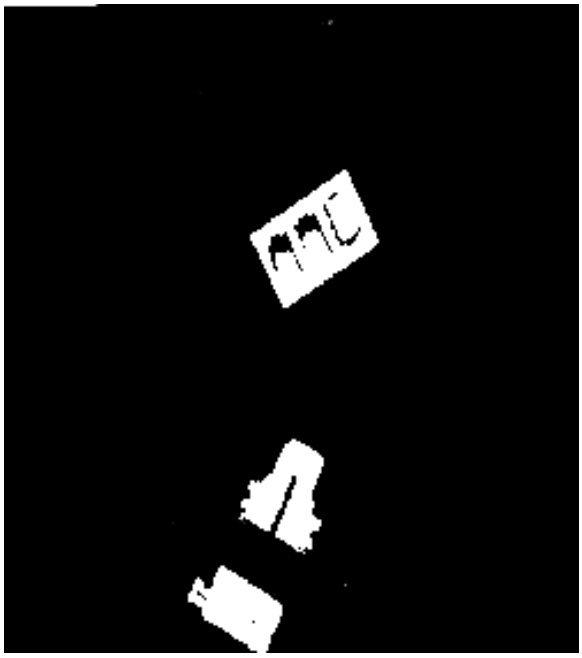


Figure 3.3: Erode and Dilate removes static from image

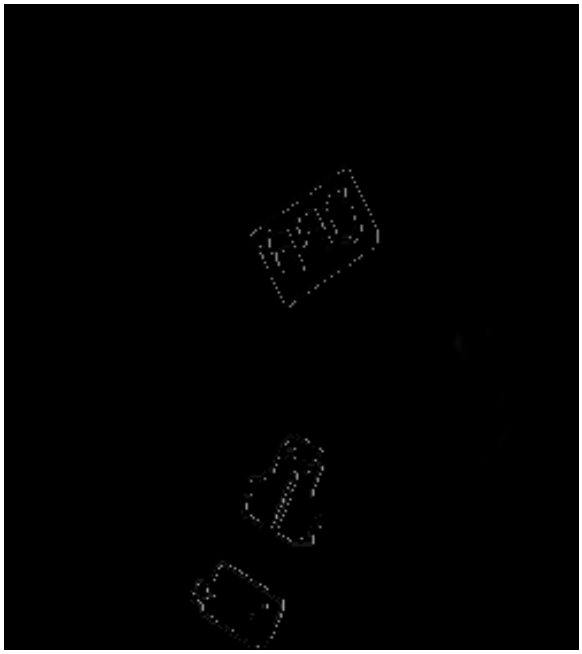


Figure 3.4: Canny edge detection leaves only the outline of the objects

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. 3.3.

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. 3.4.

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. For the detection of the main holding box, and the gripper, which are at the highest level contour were wanted. Therefore, only contours with no parents were 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 contour was then tested, anything under or over a set threshold was removed.
For each rectangle that was within the size domain, a classification and orientation was attempted as 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 fig. 3.4 3 objects remain, the holding block, the gripper and another part of the gripper of no interest. These first two will identify as expected, with the third being discarded after failing both.

3.1.3 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. 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. 3.5. To derive the highest red ratio eq. (3.1) 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.

$$mean(red)/(mean(blue) + mean(green) + mean(red)) \tag{3.1}$$

This reference corner was then used to determine the holding box true angle

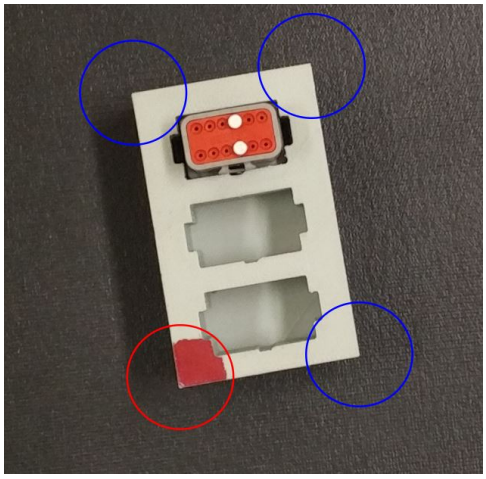


Figure 3.5: Red detection for reference corner

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. 3.6, 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°.

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

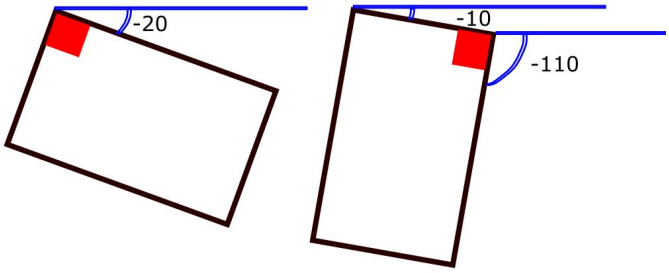


Figure 3.6

3.2 Pin detection and PinHole Allocation

With the holding block true angle determine, a copy of the image can be rotated and cropped to be only the holding block. This cropping can be seen in fig. 3.7.

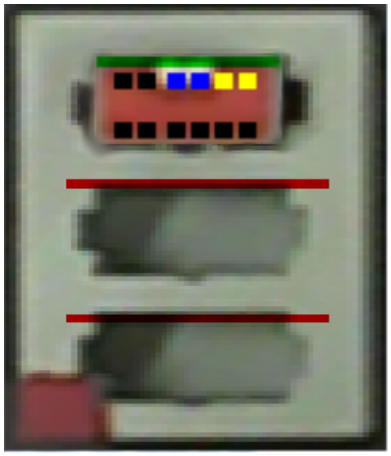


Figure 3.7: Holding block image, cropped down to only the block, and with all socket overlay shown

Data on this holding block can now be used for finding the sockets in the holding block. [1] This information defines how large the block is in millimetres, so a pixel per millimetre ratio can be found. The block information also defines were the sockets should be (in millimetres), within the block. By using the ratio, and the physical displacement of each socket, a socket object can now be made. Each socket object is passed the holding block image, its pixel starting point within this image, expected hight, and file

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

name for CSV file with pin details. The socket starts by cropping a reference matrix of the block image. 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 block 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. 3.7.

Using its sub-matrix, the socket examines the mean red ratios, as seen for the holding block corner, and determines if a socket is present, or if the socket place is empty, as shown in fig. 3.8.

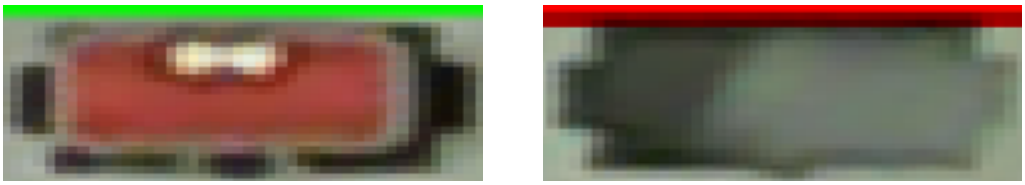


Figure 3.8: Active and empty socket place identified and presented with, respectively, green or red bar above

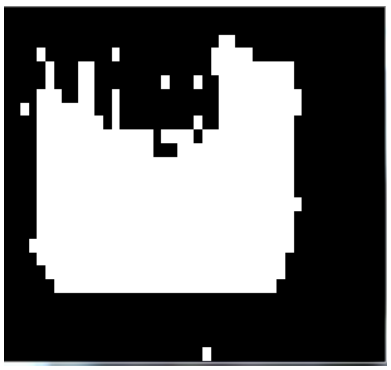
Sockets that have been identified as present are then further examined to find the exact area for pin hole mounting. Shown in fig. 3.9a is the original socket image. A threshold function of high red, low green, low blue is then performed, fig. 3.9b. The image is eroded and dilated, fig. 3.9c, and a canny edge detection is conducted, fig. 3.9d. The image is then cropped to fit the resulting contour, fig. 3.9e.

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.

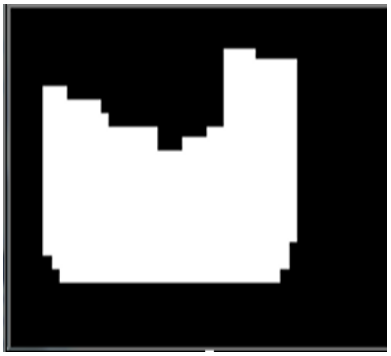
may be able to add stuff here if better res camera allows us to have an extra layer of positioning

A second CSV file then provides information on which pin holes should be filled. The resulting pin layout is shown in fig. 3.9f, black pin holes are not filled, and should not be, blue have already been filled, and yellow still need to be filled.

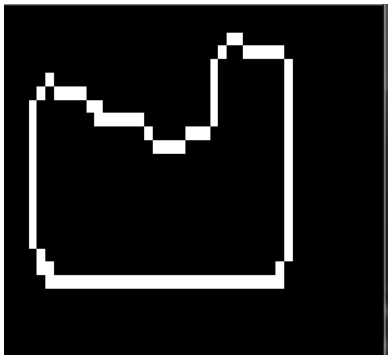
(a) original sub-matrix



(b) Post-threshold

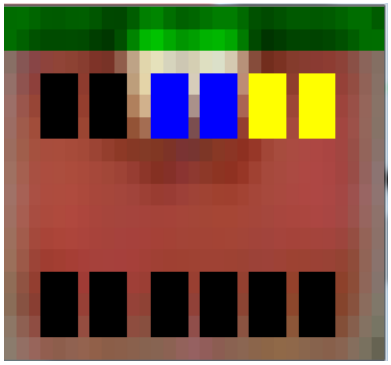


(c) Small specs removed



(d) Edge detection

(e) Matrix refit to edge of red area



(f) Pins identifed

Figure 3.9: Image processing of socket

3.3 Path creation

workds words words.

Chapter 4

Robotic Arm Control

4.1 Denavit-Hartenberg Algorithm

Links	θ	d	a	α
1	$\theta 1$	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	$\pi/2$

4.2 Matlab Simulations

4.3 Open ABB

For this project, a basic parallel port could not be used. As more complex data transfer was required.
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 distances. To achieve this, OpenABB $https://github.com/robotics/open_abb$ 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, 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, while leaving the z value constant. An example of the command stack is shown in **??**.

Chapter 5

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 was sent commands via $PyRun_SimpleString$.
The command stack was thus extended as shown in **??**

Appendix A

Initial project request

get pdf of project request paper