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

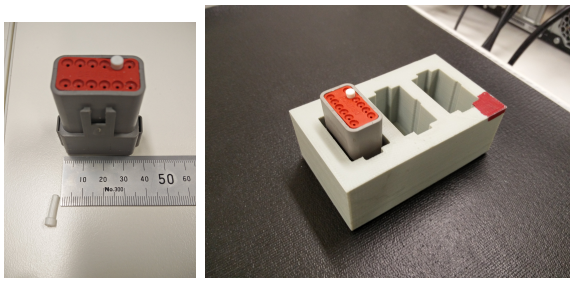


Figure 1.1: Image of pin and socket, and socket in holding box

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 first 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.
The secondary objective is having the robotic arm collect a pin, and place it in the pin hole by converting the pinhole location in the camera frame to the required position in the robotic arm frame. The process for placing the pin with the robotic arm will then be as follows:

- Lower pin halfway into hole
- Release pin
- Drive pin remaining distance into hole.

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.

2.1 Object detection

2.1.1 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. 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.

2.1.2 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).

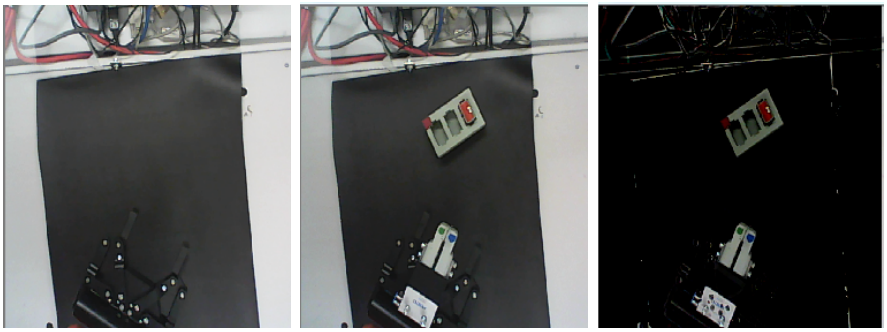


Figure 2.1: Background image, new capture with objects in place and resulting subtraction

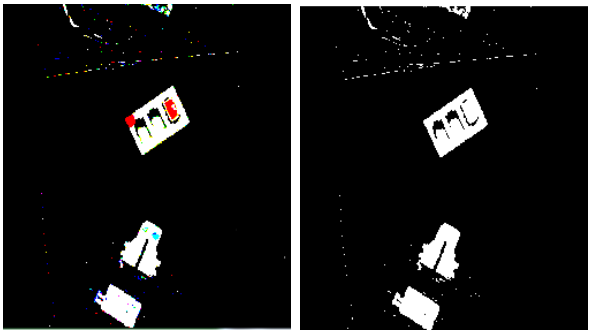


Figure 2.2: Threshold of each colour, and combining of colour layers to make single layer binary

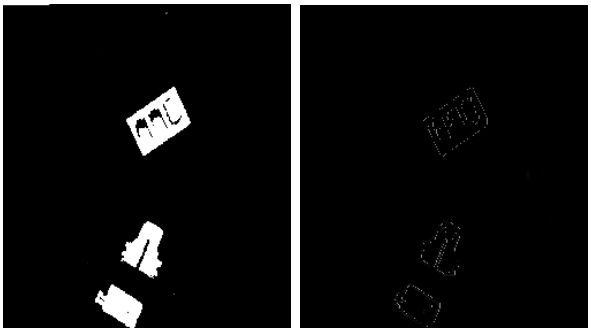


Figure 2.3: Erode and Dilate removes static from image, and Canny edge detection leaves only the outline of the objects

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

2.1.3 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.

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. (2.3) 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.

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

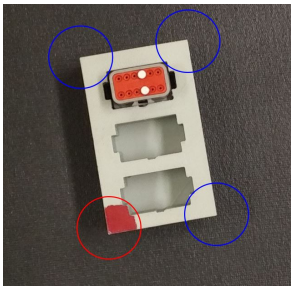


Figure 2.5: Red detection for reference corner

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°.

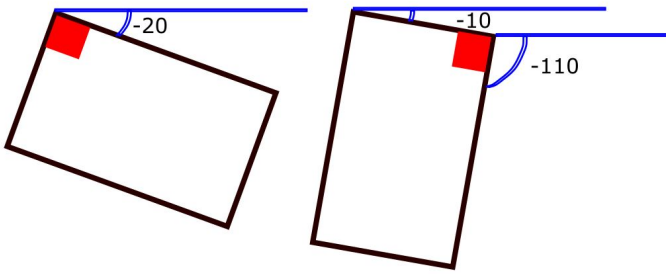


Figure 2.6

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

2.2 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.

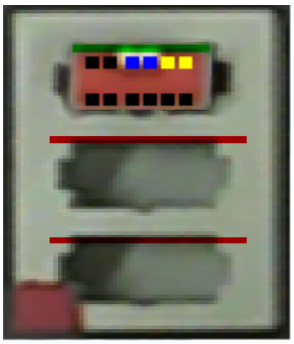


Figure 2.7: Holding box image, cropped down to only the box, and with all socket overlay shown

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.

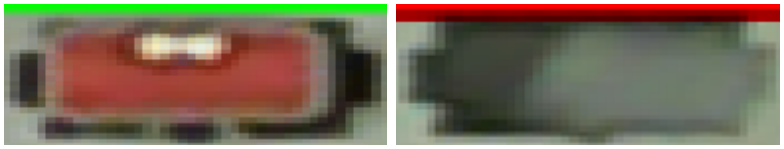


Figure 2.8: Active and empty socket place identified and presented with, respectively, green or red bar above

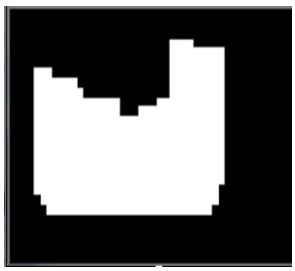
Sockets that have been identified as present are then further examined to find the exact area for pin hole mounting. Shown in fig. 2.9a is the original socket image. A threshold function of high red, low green, low blue is performed, fig. 2.9b. The image is eroded and dilated, fig. 2.9c, and a canny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

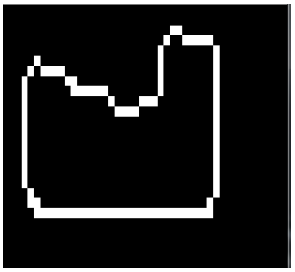
may be able to add stuff here if better res camera allows us to have an extra layer of positioning

(a) original sub-matrix

(b) Post-threshold

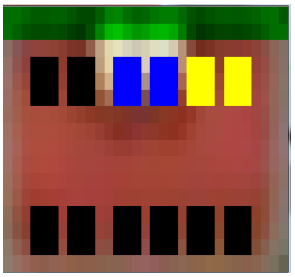


(c) Small specs removed



(d) Edge detection

(e) Matrix refit to edge of red area



(f) Pins identifed

Figure 2.9: Image processing of socket

2.3 Path creation

2.3.1 Pinhole and gripper location

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 unfilled pinhole (coloured yellow above) reports to its parent socket the centre pixel location in the pin, combined with the location of the pin within the socket. The socket then combines this location, and the location of the socket within the parent holding box, and reports this value to its parent.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.

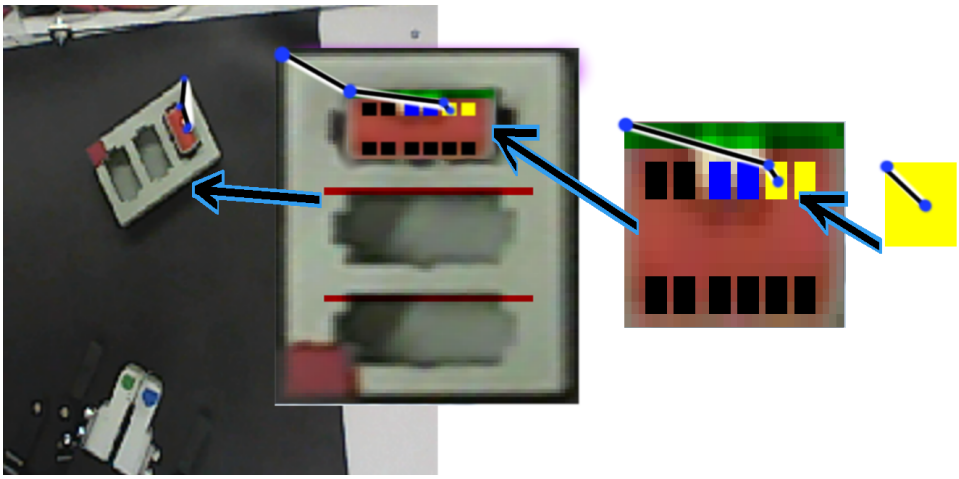


Figure 2.10: Local position, cascading to parent to find global pin position

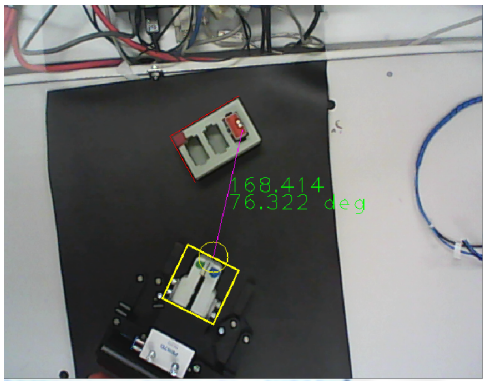


Figure 2.11: Gripper to pin hole vector display

To find a pixel singularity representation of the gripper, the half-way point between the blue and green corners, along the shared edge of the RotatedRect is reported to the main function.

2.3.2 Measurement correction

As this application is being deployed under fluorescent lighting, there is some oscillation in recordings 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 process is repeated 30 times, in quick succession, and a vector is made from gripper pixel to pin-hole centre pixel for each recording, there is a range of

values found.

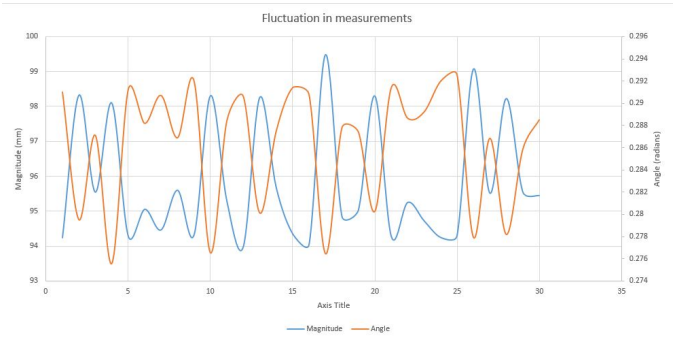


Figure 2.12: Vector magnitude and angle change over 30 records in close precession

To reduce this lighting issue, a collection of 30 location recordings are made. The maximum and minimum values of these records are then averaged to derive a value with some error tolerance.

2.3.3 Path to gripper arm co-ordinates

To move the gripper towards its target pin-hole destination for placement, the robot-frame position of the pin hole must be known. To find this, the vector from gripper to pin hole in the camera frame, found above, is broken down into Cartesian components and transformed into robot-space.

The following assumptions about the camera and workspace were made to simplify the design of the pixel to robot-frame conversion.

- No camera distortion is present
- The effects of field-of-view and perspective are ignored due to the heads-down and mainly depth-agnostic nature of the workspace
- The z-position of target pin-hole is constant and known

Before the conversion can be made, a configuration measurement must be preformed to create a vector Xc, aligned to the x-axis of the robot-frame, within the camera-frame. This is done by recording the camera-frame location of the gripper at the robot home position, and after moving the gripper 200mm along the x-axis of the robot-frame, using the measurement correction method of section 2.3.2. This allows for decomposition of the gripper-to-pin-hole vector Pc into terms of perpendicular vectors Xc and Yc (y axis). To find these Pc components the following equations are used.

$$Pcx * Xc = (Xc.Pc)/(|Xc||Xc|)Xc \tag{2.2}$$

$$Pcy * Yc = Pc - Pcx * Xc \tag{2.3}$$

These vectors are the pixel space components aligned to world-space axis. To obtain the world-space scaled vector of the pin-hole Pw, the constants Pcx and Pcy are multiplied by the pixel to world scaling factor (200mm / the magnitude of Xc (pixels)). The gripper can now be commanded to move the transformed world pin-hole vector, Pw, for placement of the pin.

Chapter 3

Robotic Arm Control

3.1 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 destinations from the computer vision, to the robotic arm, therefore 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, and send commands. One such command can move the gripper to any point in the arm's reachable workspace, 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, while holding the z axis value. An example of the command stack is shown in fig. 3.1.

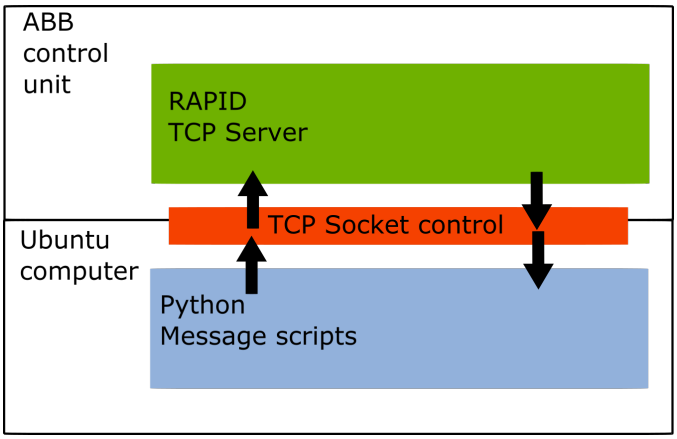


Figure 3.1: Communication layers of OpenABB

3.2 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

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

commands via $PyRun_SimpleString$.
The command stack was thus extended as shown in fig. 3.2

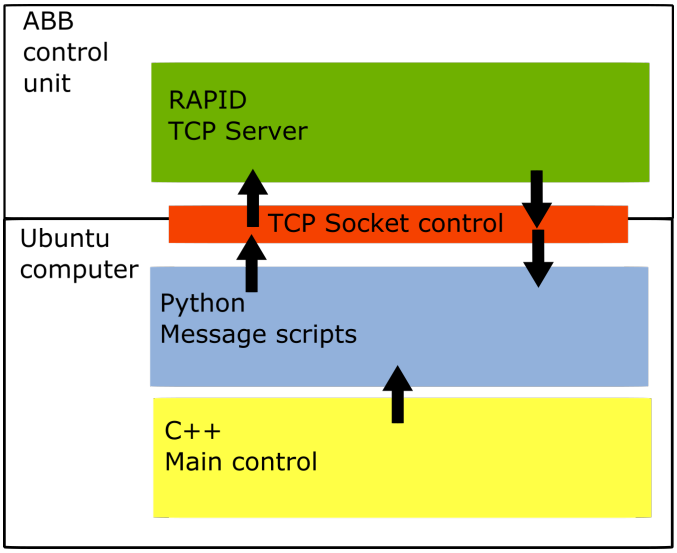


Figure 3.2: Communication stack with added c++

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 difference between robotic arm frame, and the camera frame.

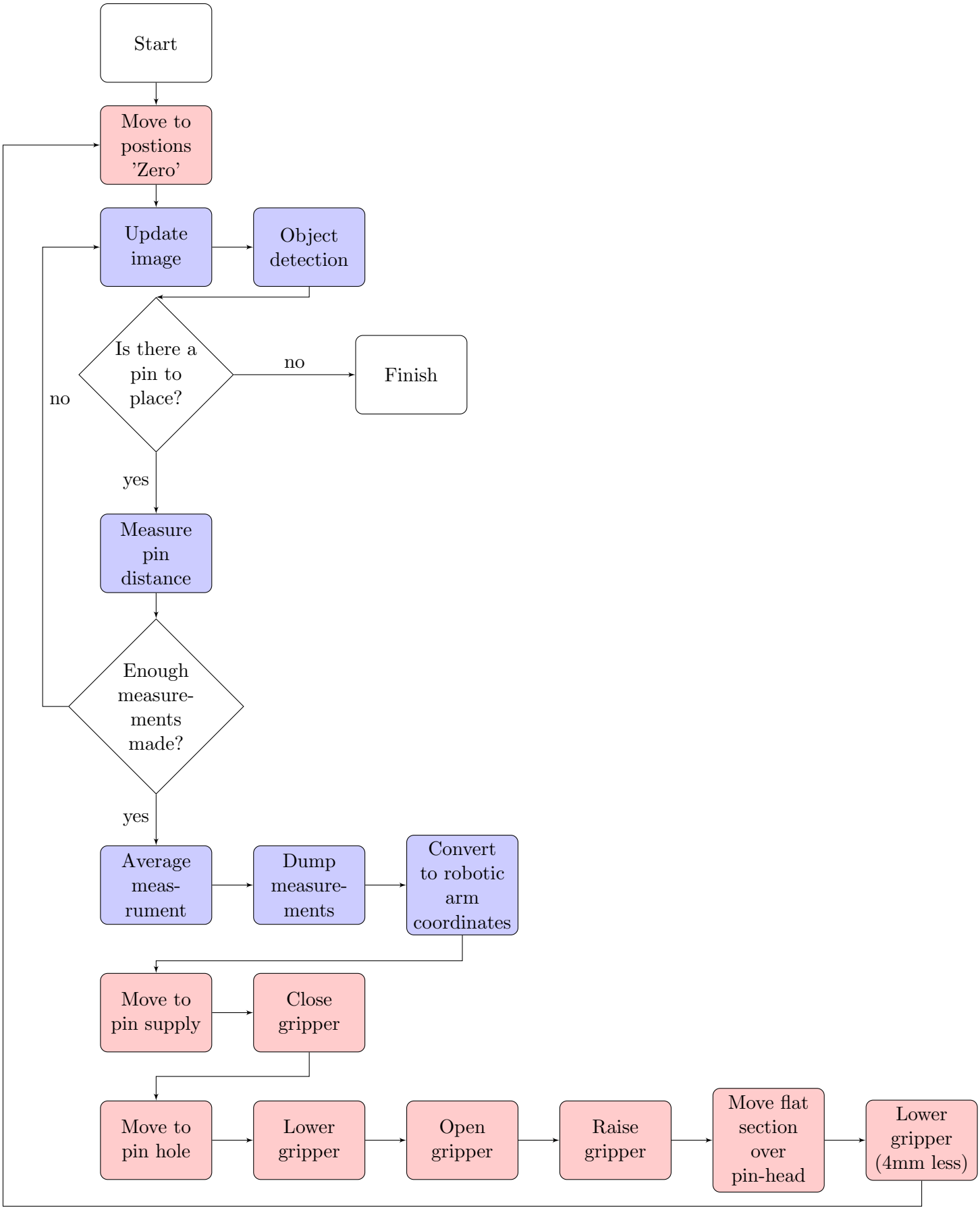


Figure 4.1: Main Process

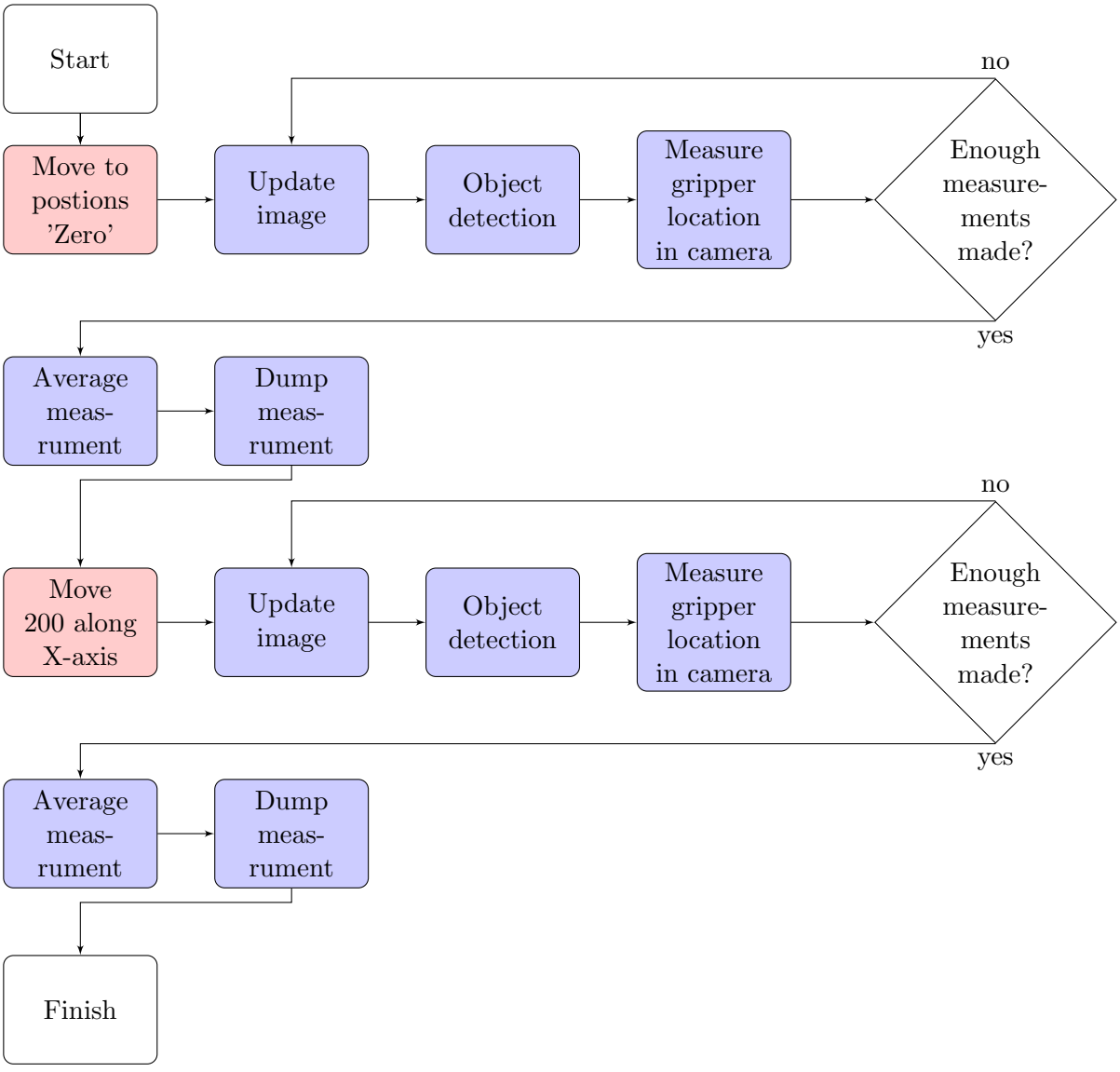


Figure 4.2: Calibration

Chapter 5

Matlab Simulation

5.1 Denavit-Hartenberg Algorithm

Links	θ	d	a	α	Physical
1	θ	0	0	0	Base
2	$\theta 2$	0	0.07	$\pi/2$	Shoulder
3	$\theta 3$	0	0.36	0	Elbow
4	$\theta 4$	0.38	0	$\pi/2$	Twist-1
5	$\theta 5$	0	0	$-\pi/2$	Wrist
6	$\theta 6$	0.108	0	$\pi/2$	Twist-2

Table 5.1: Denavit-Hartenberg Algorithm

The Denavit-Hartenberg matrix 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 joints 4 and 6, allowing a rotation about the length of a previous link.
Assumptions are made with the position of joints 4 and 6, where joint 4 is located roughly in the middle of joints 3 and 5 on the physical robot, however since it's rotation is about the link axis it is irrelevant where this joint is located, so long as it is within the linkage length. That is to say, if this link is rotated by a joint at any point along its length, the rotational behaviour of further joints will be the same. Due to this behaviour, joint 4 can be simulated at one end of the robot's forearm, instead of at its physical location on the robot's forearm.
A problem occurs when attempting to set a joint which rotates about the link axis. When the Matlab link configuration has an 'a' or 'd' offset after any joint which rotates about a link axis, Matlab cannot construct the robot arm model correctly in the simulation. This is due to the 'a' and 'd' offset lengths both being perpendicular to the link axis. In order to fix this problem and simplify the Denavit-Hartenberg algorithm, it is easier to set the distance between joints 4 and 5 to zero then rotate joint 5 into the correct orientation. It was also assumed that joint 6 can be located at the end-effector in order to benefit the simulation so that movements and translations will include that distance in calculations. This means that the link between joints 5 and 6 will simulate the end-effector.
Regardless of alterations made in order to simplify the algorithm notation or fix errors produced in a Matlab simulation, the motion characteristics are equivalent to the physical movements of the ABB robot.

5.2 Matlab implementation

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.
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. For the ABB robot, only rotational joints are present.
To begin with, a starting position is defined for robot arm joint rotation, in this case all angles are set to zero degrees, which represents the unmodified model of the Denavit-Hartenberg algorithm. In this pose it is easy to see whether or not the joints are configured correctly and that the model does in fact mirror the basic shape of the physical model being simulated.
The next step is to define the x,y,z position and rotation for each point of motion which will be simulated. This can be done by simply assigning variables to represent that information as in appendix A, lines 45-105. In order to run an effective simulation, the position and rotations must be compiled into a transformation matrix. This matrix defines all translational and rotational movement required between the reference frame and the desired destination frame. In order to convert the raw variables representing the position and rotation of the end-effector into a transform matrix, two functions are used. One is transl(x,y,z), which represents translation and takes in the x, y and z position of a point and the other is rpy2tr(rx,ry,rz), which represents the roll, pitch and yaw angle which rotate about x, y and z respectively. The result of these two functions are multiplied together and saved as the transformation matrix for the specified point, in the order as seen in appendix A, lines 108-114. A transformation matrix for the initial position is also required. This is achieved easily by using the forward-kinematic function fkine(q), where q is a 1xn array representing the joint rotations where n is the number of joints on the robot arm.
In appendix A, lines 121-126, two arrays are initiated. The first one is a mask representing the degrees of freedom available in the joint path calculations. The second array represents a joint angle position array which acts as a reference point. This point is useful by forcing the movement of the arm to conform to the specified joint angles. This helps to simulate movement that stays within the bounds of the work area of the robot. As a solution for each joint converges with the corresponding reference angle, the angles will only change when a joint has the least convergence.
Although we have a transformation matrix for each position we wish to simulate, what we really need are the joint angles values in radians, which are

the exact rotations of every joint which together correspond to the position of the end-effector. In order to get a joint angle array from a transformation matrix, we can use the inverse-kinematic function ikine(T,qr,M), where T is a transformation matrix for the point, qr is the reference array used to restrict motion and M is the mask for the degrees of freedom available. For the ABB robot, the joint angles at the simulated points are calculated as in appendix A, lines 121-135. When finding the inverse kinematic of the transformation matrix, the function ikine() takes into account the reference array and the degrees of freedom in order to find a single solution for the joint angle configuration which makes up the desired position and rotation. Users must be careful, however, as having an inappropriate reference array may cause the calculations to diverge, resulting in sub-optimal solutions which may even result in simulation movement outside of the immediate work area.

Appendix A

Matlab Code

```
% Simple example of using the Matlab Robot toolbox v9.9 for RME40003 Robot
% Systems Design, 2014.
%
% The example is for a three-link RPR manipulator that was covered in the
% lectures and the tutorials.
%
% Example developed by M. Dunn, 2014, Swinburne University of Technology,
% Melbourne, Australia
%
% References::
% - Robotics, Vision & Control, Chap 7
%   P. Corke, Springer 2011.

clear L

%% Define the link parameters for the manipulator
% Set the fifth parameter as 0 for a rotary joint, 1 for a prismatic joint
% Set the "modified" flag to true (we are using the modified
% Denavit-Hartenberg convention in RME40003)

sF = 1.0;
%              theta    d       a         alpha    R(0)/P(1)
L(1) = Link([ 0        0       0         0        0], 'modified');
L(2) = Link([ 0        0       0.07      -pi/2    0], 'modified');
L(3) = Link([ 0        0       0.36      0        0], 'modified');
L(4) = Link([ 0        0.38    0         -pi/2    0], 'modified');
L(5) = Link([ 0        0       0         pi/2     0], 'modified');
L(6) = Link([ 0        0.108   0         -pi/2    0], 'modified');

%% Set up the manipulator
robot = SerialLink(L, 'name', 'RME40003 Project');

%% Define some poses
%start pose
q0 = [      0, -pi/4, -pi/4,      0,      0,      0];

```

```
% 1:Base-rot
% 2:shoulder-rot
% 3:elbow-rot
% 4:elbow-twist
% 5:wrist-pivot
% 6:wrist-twist


%% Raw position and rotation data for each point
%point1

x1 = 0.273/sF;
y1 = -0.740/sF;
z1 = 0.02/sF;

%radians
rotx1 = pi;
roty1 = 0;
rotz1 = 0;

%point2
x2 = 0.273/sF;
y2 = -0.740/sF;
z2 = -0.1/sF;

%radians
rotx2 = pi;
roty2 = 0;
rotz2 = 0;

%point3
x3 = 0.273/sF;
y3 = -0.740/sF;
z3 = 0.02/sF;

%radians
rotx3 = pi;
roty3 = 0;
rotz3 = 0;

%point4
x4 = 0.4/sF;
y4 = -0.4/sF;
z4 = 0.02/sF;
```

```
81
82 %radians
83 rotx4 = pi;
84 roty4 = 0;
85 rotz4 = 0;
86
87 %point5
88 x5 = 0.4/sF;
89 y5 = -0.4/sF;
90 z5 = -0.1/sF;
91
92 %radians
93 rotx5 = pi;
94 roty5 = 0;
95 rotz5 = 0;
96
97 %point6
98 x6 = 0.4/sF;
99 y6 = -0.4/sF;
100 z6 = 0.02/sF;
101
102 %radians
103 rotx6 = pi;
104 roty6 = 0;
105 rotz6 = 0;
106
107
108 %% Point translation homogeneous matrixes
109 T1 = transl(x1,y1,z1)*rpy2tr(rotx1,roty1,rotz1);
110 T2 = transl(x2,y2,z2)*rpy2tr(rotx2,roty2,rotz2);
111 T3 = transl(x3,y3,z3)*rpy2tr(rotx3,roty3,rotz3);
112 T4 = transl(x4,y4,z4)*rpy2tr(rotx4,roty4,rotz4);
113 T5 = transl(x5,y5,z5)*rpy2tr(rotx5,roty5,rotz5);
114 T6 = transl(x6,y6,z6)*rpy2tr(rotx6,roty6,rotz6);
115
116
117 %% Find the forward kinematics of the pose
118 fk_q0 = robot.fkine(q0);
119
120
121 %% Find the inverse kinematics for fk_qs and fk_qe
122 % Set a mask for the degrees of freedom
123 M = [1, 1, 1, 1, 1, 1];
124
```

```
% Point set for forced convergence
qi = [0,-pi/4,-pi/4,0,pi/2,-pi/2];

% Calculate the inverse kinematics of each point
ik_q0 = robot.ikine(fk_q0,qi,M); %fk of initial angles, initial position
    estimation, DOF settings
ik_q1 = robot.ikine(T1,qi,M);
ik_q2 = robot.ikine(T2,qi,M);
ik_q3 = robot.ikine(T3,qi,M);
ik_q4 = robot.ikine(T4,qi,M);
ik_q5 = robot.ikine(T5,qi,M);
ik_q6 = robot.ikine(T6,qi,M);

%% Find the joint trajectory for ten steps between the two poses
% Set the number of points
num_points = 10;

% Moving from point to point
jt0_1 = jtraj(ik_q0,ik_q1,num_points);
jt1_2 = jtraj(ik_q1,ik_q2,num_points);
jt2_3 = jtraj(ik_q2,ik_q3,num_points);
jt3_4 = jtraj(ik_q3,ik_q4,num_points);
jt4_5 = jtraj(ik_q4,ik_q5,num_points);
jt5_6 = jtraj(ik_q5,ik_q6,num_points);
jt6_0 = jtraj(ik_q6,ik_q0,num_points);

% Set the workspace of the robot
robot.plotopt = {'workspace',[-1, 1, -1, 1, -1, 1]};

% Plot the animations
while 1
        robot.plot(jt0_1);
        pause(1);
        robot.plot(jt1_2);
        pause(1);
        robot.plot(jt2_3);
        pause(1);
        robot.plot(jt3_4);
        pause(1);
        robot.plot(jt4_5);
        pause(1);
        robot.plot(jt5_6);
        pause(1);
        robot.plot(jt6_0);
```

```
168             pause(1);
169 end
```