Vietnamese – German University

Department of Electrical Engineering & Information Technology

DEVELOPING AN AUTOMATED SYSTEM FOR CHINESE CHESS

By

Nguyễn Huy Thong, Marticulation number: 1185034

Phan Đông Quang, Marticulation number: 1185018

Lưu Nguyễn Anh Vũ, Marticulation number: 1185092

First Supervisor: Dr. Nguyễn Minh Hiền
Second Supervisor: Biện Minh Trí

BACHELOR THESIS

Submitted in partial fulfillment of the requirements

for the degree of Bachelor Engineering in study program

Electrical Engineering & Information Technology,

Vietnamese – German University, 2020

Abstract

This work represent a new approach to challenge Chinese Chess - Xiangqi bot. Normally, to play with chess bot we require digital devices such as computers or mobile phones with the chess game installed. Therefore, we developed an automated system allowing users to enjoy that same sensation in real time.

Included in this work are the Chess board status captured by the camera with OpenCV to output changes made by the player for the logical algorithm called MiniMax algorithm with Alpha-Beta pruning to compute. The result output from the algorithm will then be sent to the SCARA arm to move the chess to desired position.

Keywords: Chinese Chess, Xiangqi, SCARA, Alpha-Beta, OpenCV

Disclaimer

We declare that this paper is a product of our own work, under the supervision of Dr. Nguyen Minh Hien, unless otherwise referenced. We also declare that all opinions, results and conclusions are our own.

Acknowledgements

This project could not be complete without the support and guidance provided by our professor supervisor and lab engineer, as well as the advices and help of our classmates.

Firstly, we would like to thanks our supervisor Dr. Nguyen Minh Hien for her accurate and on time guidance. Without her help, the recognition could never be that accurate, and the project won't ever be finished on time.

Secondly, we would like to thanks our dearest lab engineer, Mr.Bien Minh Tri. This is the first time we have work on such sophisticated system like the SCARA arm. Mr. Tri with his experience has been a great source of information for us to deal with all our problems.

And finally, We would like to show our appreciation for our friends and classmates, who have merrily share their experiences and advices, as well as provided us their mental and physically supports under the form of milk tea and coffee. They are the best bunch of friends one could ever hoped for.

Detailed Contributions Distribution

This thesis is the joint-work of three following students

- Nguyen Huy Thong. Marticulation number: 1185034
- Phan Dong Quang. Marticulation number: 1185018
- Nguyen Luu Anh Vu. Marticulation number: 1185092

The overall contribution distribution of the whole thesis is described as follows:

Nguyen Huy Thong:

- Propose the concept of the thesis
- Design the pieces' patterns
- Design the pieces' 3D models
- Research the recognition algorithms and OpenCV Library
- Propose the thesis's software basic model
- Build and configure the camera stand
- Decide on data structure and data transfer method
- Create pieces recognition program
- Test and debug the recognition program

- Cross-verify the efficiency and accuracy of data transfer
- contributing to writing on sections about the recognition part

Phan Dong Quang:

- Design and laser print the chess board
- Design and reinforces the SCARA arm
- Research on the manual calibration method
- Research on the kinematic model for SCARA arm
- Implement the SCARA arm firmware
- Conduct wiring on the electrical system of the machine
- Support in debugging the Minimax algorithm with Alpha-Beta pruning
- contributing to writing on sections about the SCARA arm

Luu Nguyen Anh Vu:

- Design the simulation program
- Create the methods to handle inputs from recognition algorithm
- Create the methods to transfer instructions to SCARA arm
- Propose a data structure to store information regarding the chess pieces
- Research the Alpha-Beta pruning method
- Propose the solution to problems arise from the algorithm
- Test and debug the chess engine and data handling methods
- Contributing to writing on sections about the Chess Engine.

Contents

List of Figures

Chapter 1

General Information

1.1 Introduction

1.1.1 Origin of the Thesis

This thesis was first developed as a seniour project inspired by our ex Lab engineer Vo Van Phung. At first, we wanted to create a system that is both entertaining and engaging for the tester. Therefore, a chess player appears to be an ideal proposal.

However, creating just a chess program seems to be a little bit too boring. Therefore, we decided to step our game up and create a whole physical system to support the chess playing program. We also want to add a touch of Eastern culture into our thesis, therefore the Chinese chess, which is also known as Xiangqi, was chosen instead of the normal international chess.

1.1.2 Chinese Chess

Chinese chess, or Xiangqi is a two-player board games that is widely played in China and Vietnam. The game depict the battle between two armies, with a river seperating them. It is believed that this game refers to the Chu - Han War which happened in 206BC since the river is often marked as either Chu's river or Han's border.

The pieces in the game are just short cylinder with their names carved on the top faces. There are seven types of piece: General, Advisors, Elephants, Chariots, Cannons, Horses and Soldiers, with the General as the most important one. The game is over when one's general is captured.

Figure 1.1: Chinese chess table

1.2 Overview

This is dedicated to explaining the general flow of the program and how it should be constructed.

1.2.1 Requirements for the product

The main purpose of the thesis is to build a fully automated Chinese chess playing machine. In order to fulfill that, this project has to satisfy below requirements:

- It has the ability to recognize the team, type and position of the pieces on the board;
- It has the ability to process the current board the able to give the next move;
- It has the ability to pick up and move the piece to the designated position.

1.2.2 General task flow

The task flow of the system is described in the 1.2.

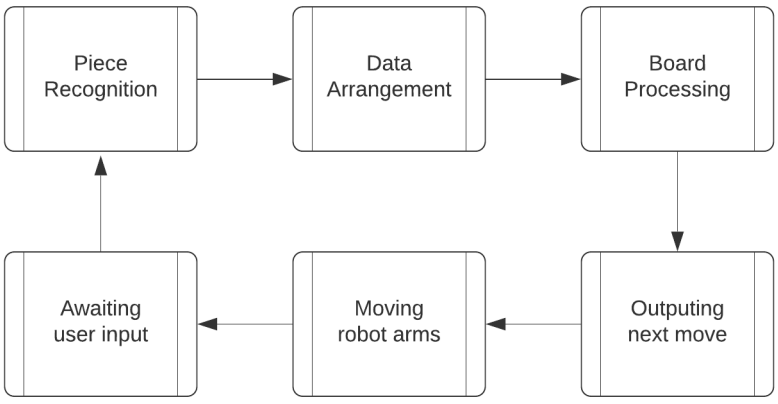


Figure 1.2: General Task Flow

When the program is opened, the camera captures the entire board and starts to recognize the names, teams, and position of the pieces on it. It is then transformed into a uniformed format and is sent to the chess state calculator. The calculator will then process the current board state and return the next move to the machine. The move is an output to the robot arm controller, and is translated into the instruction for the arm to move. As soon as the arm finished moving, the program is put into waiting mode until the player has completed their move and signals it to the program. Then, the loop continues.

1.2.3 Sectioning the system

For easier programming and debugging, the system is split into three different sub-systems that run independently from each other. The overall outline of the system is as follows:

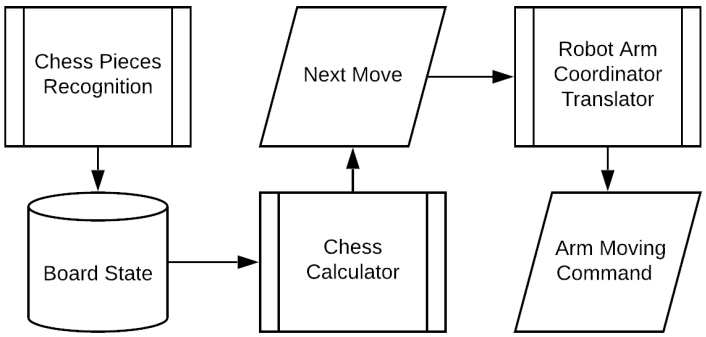


Figure 1.3: Program sections

The first sub-system is the piece recognition program. Its main purpose is to derive a complete table of names, teams and position of all the pieces on the board. This is expected to be heavily dependent on object recognition algorithm, which in this case is OpenCV.

The second sub-system is the chess calculator. By utilizing the data extracted in the previous program, this will calculate and export the optimal move for the machine.

The final sub-system is the translator. It will translate the move from the calculator into proper guidance for the robot arm so that the arm can properly pick the expected piece.

1.3 Recognition Algorithm

1.3.1 Requirements and Tools

To make the system stable and applicable when put to use, the sub-system should meet the following requirements:

- It should be able to recognize the correct position and teams;
- It should be able to recognize the correct types of the pieces;
- Fast processing.

The two first requirements ensure the accuracy of the result, while the latter is crucial to its application. In order to optimize the effort and result ratio, a pre-constructed library is utilized to ensure both the stability and performance of the system. The main library used in this program is OpenCV, which is an open sourced library for object recognition.

1.3.2 Sectioning the Part

To guarantee smooth operation, the sub-system is further divided into three main functions: (1) Position Recognition, (2) Team Recognition and (3) Pattern Recognition.

For each function, the following tools and algorithms can be used:

- Position Recognition: Hough Circle Transformation is used to detect the circles, which are the shape of the chess pieces for easy detection of the position.
- Team Recognition: This can be achieved by adding a red/green filter on the piece's image and then counting the non-zeroed pixel number.

Pattern Recognition: There are two components to this function: key-points detector/descriptor calculator and descriptor matchers. For the key-points and descriptor, either SURF or ORB can be used. For each of these algorithms there is a dedicated matcher, which are FLANN-based and Brute-Force matchers, respectively

1.4 Chess Engine

This section mainly focuses on the algorithm and data structure that form the chess engine. Two topics are included: (1) Requirements and (2) Program Sectioning.

1.4.1 Requirements

To make the program stable and applicable to real use, there are some requirements for this sub system to be met:

- Able to receive inputs from the recognition algorithm
- Able to generate the most optimum move based on the search depth of the program
- Able to send output to Arduino to process SCARA arm
- High precision and speed

1.4.2 Sub-System Sectioning

The Chess Engine consists of three main parts: Communication with Recognition algorithm, Minimax algorithm with Alpha-Beta prunning and communication with SCARA arm. Each part contains the following supportive functions which will be covered in later sections:

- Communication with Recognition algorithm: CSV files manipulation, Update chess piece list.
- Minimax algorithm with Alpha-Beta prunning: Move generator, Cloning Board state, Chess piece list manipulation, Minimax with Alpha-Beta pruning.
- Communication with SCARA arm: ReadButton, RunSCARA

1.5 Scara Robot Arm

This section mainly focuses on the design the SCARA arm robot for both hardware and control software.Two main topics are included: (1) Hardware design and (2) control software design

1.5.1 Requirements

The main purpose of the SCARA arm robot is used for pick and place the appropriate chess piece corresponding to the command sent by the Chess Engine.There are some requirements for the SCARA arm robot need to be met :

- The robot arm is able to communicate to Chess Engine program
- The robot arm is able to pick and place a chess piece from one position to the other

1.5.2 Sub-System Sectioning

The Chess Engine consists of three main parts: Communication with Recognition algorithm, Minimax algorithm with Alpha-Beta prunning and communication with SCARA arm. Each part contains the following supportive functions which will be covered in later sections:

The Scara system consists of two main parts : Command parser and Movement control.Each part is responsible for the following purpose and will be covered in the later sections ;

- Command parser : Read input data from computer,parsing the data to appropriate command, execute the commands.
- Movement control : processing the coordinate data of the Scara arm and control the arm move to appropriate position.

Chapter 2

Literature Review

2.1 Recognition Algorithm

2.1.1 HSV Color Space

Hue – Saturation – Value (HSV) is an alternative representation of the Red – Green – Blue (RGB) color model. This was first designed by computer graphic researchers in the 1970s to be the more accurate representative of how a human's eyes perceive a color.

In this color space, each hue has its colors arranged in a radical slice, a long a central axis. The more upward the axis, the closer the color is to that of black. Likewise, the color gets closer to that of white as the slice approaches the bottom.

The HSV representation models the way paints of different colors mix together, with the saturation dimension resembling various tints of brightly colored paint, and the value dimension resembling the mixture of those paints with varying amounts of black or white paint.

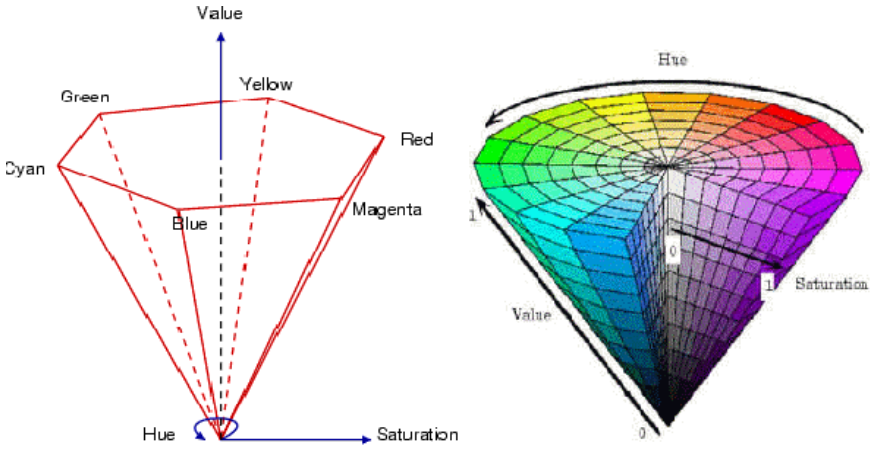


Figure 2.1: HSV Model

By default, the color space used in OpenCV are BGR, which stand for Blue – Green – Red. However, this color space proves to be unreliable on varying lighting condition. A green color range might become yellow under extreme light condition, which is unideal for distinguishing using BGR space.

Meanwhile, since there exist the Value and Saturation parameters, it is much easier for the HSV color space to eliminate or minimize the effect of

lighting on color recognition. **After multi testing**, the range for Red and Green color is defined as follows:

```
const int GreenLowH = 100;
const int GreenLowS = 0;
const int GreenLowV = 0;
const int GreenHighH = 165;
const int GreenHighS = 255;
const int GreenHighV = 255;

const int RedLowH = 160;
const int RedLowS = 0;
const int RedLowV = 0;
const int RedHighH = 255;
const int RedHighS = 255;
const int RedHighV = 255;
```

2.1.2 Hough Circle Transformation

2.1.2.1 Algorithm review

Due to the special shape of the chess pieces, which is in the form of a short cylinder, the pieces can be recognized by using a circle detection function. The most popular technique is the Hough Circle Transformation [21].

The Hough Circle Transformation is operated by first determining an approximate radius. The algorithm then proceeds to check every pixel and find a fitting circle [25]. The algorithm could be described by the flow chart in Figure 2.2.

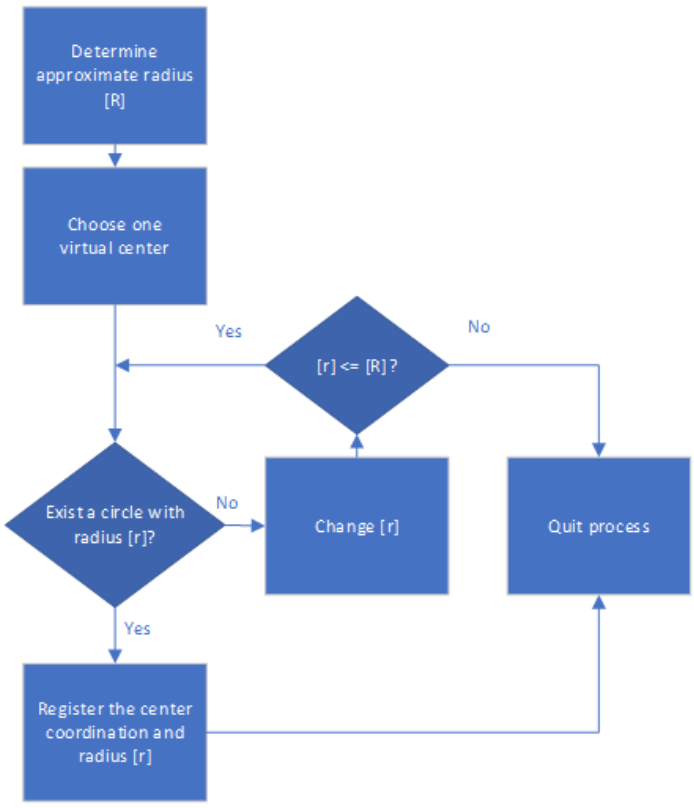


Figure 2.2: Hough Circle Algorithm Flow

2.1.2.2 Hough Circles Theory

In a two dimensional plane, a circle can be expressed as:

$$(x-a)^2 + (y-b)^2 = R^2 \tag{1}$$

with the point (a, b) being the center and R being the radius. Accordingly, in order to describe a circle, we need to find a set of three parameters a, b and R. Assuming that the radius R is fixed, the parameters' space will then be reduced to 2D [11].

With a constant R, what is left to be determined is to the 2D set (a, b). Assuming there exists a circle. For every point (x, y) on the original circle, we can define a circle centering at (x, y) with radius R. The intersection of all these circles would be the center of the original circle [11].

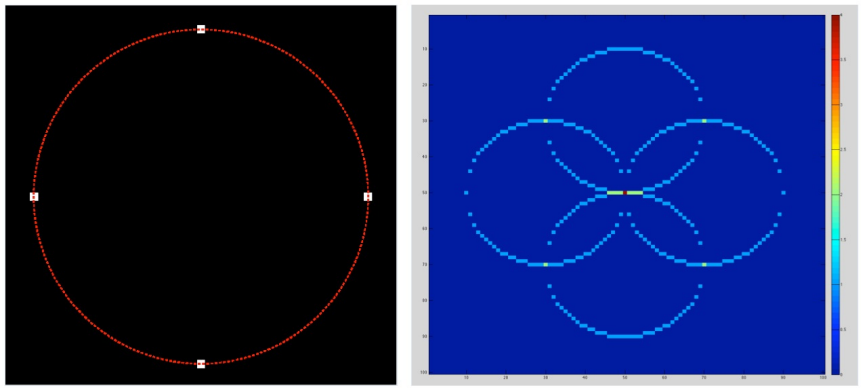


Figure 2.3: Hough transform on four points

In the Figure 2.3, we take 4 points on the original circle and draw additional circles with the same radius centering on each point. Those four circles intersect at one point, which will be the center of the original circle [1].

In practice, the image must go through the Canny Edge Detection, which includes Gaussian blur, intensity gradient calculation, non-maximum suppression and double threshold filter in order to be efficiently and effectively processed by Hough Circle Transform [4].

2.1.2.3 Gaussian Blur

Gaussian blur, or Gaussian smoothing, is a common method to reduce image noise and details in image processing. This is achieved by blurring an

image using a Gaussian function. In a two-dimensional setting, the Gaussian equation is expressed as follows:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

where (x, y) is the coordination of the point in the picture, and σ the standard deviation of the Gaussian distribution. When it is applied, the formula produces a surface whose contours are concentric circles with a Gaussian distribution from the center. Those values from the surface are used to build a convolution matrix, which is later applied to the original image. Each pixel's new value is set to a weighted average of the pixel's neighborhood.

$$B = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * A \tag{3}$$

Theoretically, the Gaussian function at every point will be non-zero. That means calculating a pixel will include the entire image. However, in discrete approximation of the Gaussian function, the pixels at a distance of more than 3σ do not have sufficient impact, and are usually considered zero [5].

Applying a Gaussian blur to an image is the same as convolving the image using a Gaussian function. Because of this characterstic, the Gaussian blur has the function of reducing the image's high-frequency components, which usually means sharper and more distinct edges. Therefore, it becomes a low-pass filter and is ideal as an effective pre-processing tools for many edge detection algorithms [18].

Figure 2.4: Gaussian Blur sample

2.1.2.4 Intensity Gradient Calculation

Gradient is a slope in light intensity of an image. More precisely, a gradient of an image represents the changes in the values of pixels. In a smooth region of an image, the pixels have similar values, therefore the derivative of that region will be close to zero. On the other hand, a great derivative value represents a drastic change in light intensity in the region [3].

In the Canny Edge Detection algorithm, the 3x3 Sobel X and Sobel Y filters are used to calculate the derivatives of the image.

$$G_x : \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \tag{4}$$

$$G_y : \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{5}$$

From this, the edge gradient and direction could be determined as follows:

$$G = \sqrt{{G_x}^2 + {G_y}^2} \tag{6}$$

$$\Theta = arctan2(G_x, G_y) \tag{7}$$

The edge gradient G represents the changes in light intensity of an image at a given pixel. Meanwhile, Θ represents the angle of the gradient, which is then converted into 4 bins: horizontal (0^o), vertical (90^o) and two diagonals (45^o and 135^o) [2].

2.1.2.5 Non-maximum Suppression

Non-maximum suppression is a technique used for thinning the edges of an image. Its main function is to find the "largest" edge available. This is achieved by running a 3x3 filter across every pixel on the gradient image. During the filtering process, only the centering pixels that are their corresponding local maxima are retained. Otherwise, the gradient value of the mentioned pixel is set to zero. It should be noted that the centering pixel is only compared to the 2 neighboring pixels with respect to its gradient orientation. For example, if the gradient orientation is 0^o, only the pixels directly to the left and those to the right of the centering pixel are used for comparison [2].

After finishing this step, a binary image of the original is produced. This image then continues to be processed using double threshold filters to achieve a final contour image.

2.1.2.6 Double threshold filter

Only the positive pixels on the generated binary mask are taken into consideration in this filter. Two values, i.e. maxVal and minVal, are predefined as two thresholds for the gradient value. If the gradient value of the pixel is greater than maxVal, that pixel is defined to be an edge. If the

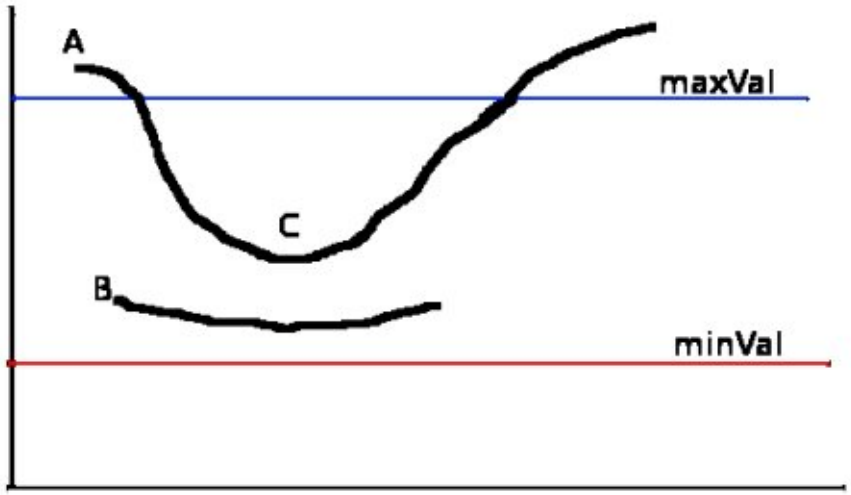


Figure 2.5: Double Thresholds Filter

value is smaller than minVal, the pixel is eliminated from the result image. Moreover, if the value is between the two thresholds, its neighbouring pixels are the criteria for defining the outcome: this pixel is an edge if one of its neighbouring pixels is also an edge.

Take three segments A, B and C as above. A is surely an edge, C is also considered an edge since it is between the two thresholds and neighbouring A. Meanwhile, B is not considered an edge and is eliminated while it is still between the thresholds since it is not the neighbour of any edge [3].

After this step, an image with only the edges is exported. If the situation requires, there might be after-processed noise filtering, such as removing any isolated/short edge pixels to make the edge detection much more accurate. The final result is sent to the Hough Circle Detection algorithm to extract the circles with the pre-defined radius.

2.1.2.7 Function setup

The Hough Circle Transform is a pre-built library belonging to the imgproc library of OpenCV. Its application is as follows:

```
std::vector<cv::Vec3f> HoughTrans(cv::Mat boardMat)
{
    std::vector<cv::Vec3f> circles;
```

```
cv::Mat grayMat;
cv::Mat copyMat = boardMat.clone();
cv::cvtColor(boardMat,grayMat,cv::COLOR_BGR2GRAY);

cv::HoughCircles(grayMat,circles,cv::HOUGH_GRADIENT,
(double) dp/100,
(double)boardMat.rows/minDist,
CannyEdgeDetect,CircleCenterDetect,
minRad,maxRad );

for( size_t i = 0; i < circles.size(); i++ )
{
    cv::Point center(
            cvRound(circles[i][0]),
            cvRound(circles[i][1]));
    int radius = cvRound(circles[i][2]);
    // circle center
    circle( copyMat, center, 3, cv::Scalar(0,255,0),
                                            -1, 8, 0 );
    // circle outline
    circle( copyMat, center, radius, cv::Scalar(0,0,255),
                                            3, 8, 0 );
}


return circles;
```

There are six parameters in declaring the `cv::HoughCirlces()` function, namely `dp`, `minDist`,`CannyEdgeDetect`, `CircleCenterDetect`, `minRad` and `maxRad` respectively.

`dp` is the inverse ratio of the accumulator resolution to the image resolution, which means the higher the `dp`, the more inaccurate the recognition is, but the less sensitive to the noise as well.

`minDist` is the minimum distance between each center. The `minRad` and `maxRad` also represent the minimum and maximum value of the circles that we want to recognize.

`CannyEdgeDetect` and `CircleCenterDetect` are the two thresholds for the Canny operator in Hough Circle Transform. The threshold used in the

Canny operator uses a method called "hysteresis". Most thresholds use a single threshold limit, which means if the edge values fluctuate above and below this value the line will appear broken (commonly referred to as "streaking"). Hysteresis counters streaking by setting an upper and lower edge value limit. Considering a line segment, if a value lies above the upper threshold limit it is immediately accepted. If the value lies below the lower threshold it is immediately rejected. Points which lie between the two limits are accepted if they are connected to pixels which exhibit strong response.

2.1.3 SURF

The Speeded Up Robust Features detection (SURF) is a local feature detector and descriptor, which is inspired by the Scale-invariant features transform (SIFT). This method was first presented by Herbert Bay at the 2006 European Conference on Computer Vision [15]. This algorithm can be applied for tasks such as object recognition, image registration, classification or 3D object construction.

The algorithm consists of three main steps: interest point detection, local neighbourhood descriptor, and matching discussed in the next subsections.

2.1.3.1 Detection Algorithm

The square-shaped filter is used as an approximation for Gaussian smoothing. Due to the simplicity of calculating with a square-shaped filter, the algorithm can be resolved much faster by using an integral image.

A blob detector based on the Hessian matrix is used to find the point of interest. Given a point **p** = (x, y) in the image **I**, the Hessian matrix **H**(p, s) at the point **p** and scale **s** is as followed:

$$H\left(p,s\right)=\begin{pmatrix}L_{\mathrm{xx}}(p,s) & L_{\mathrm{xy}}(p,s)\\ L_{\mathrm{yx}}(p,s) & L_{\mathrm{yy}}(p,s)\end{pmatrix} \tag{8}$$

where $L_{\mathrm{xx}}(p,s)$ is the convolution of the second order derivative of Gaussian with the image ***I****(x, y)* at the point ***p***. [14]

The determinant of the Hessian matrix is used as a measure of local change around the point, and points are chosen where this determinant is maximal. The mentioned determinant is also used for selecting the scale. The result of this steps is a list of points of interest, or key points.

2.1.3.2 Descriptor Calculation Algorithm

The descriptor describes the distribution of the intensity content within the neighbourhood of the key point of interest. It is similar to the gradient information extracted by SIFT [9]. However, this is built on the distribution of first order Haar wavelet responses in x and y directions, exploiting integral images for fast calculations, and only uses 64 dimensions, which reduces the time for feature computation drastically. There are two main steps

for descriptor calculation: (1) fixing a reproducible orientation and (2) constructing a square region aligned to the selected orientation and extract the SURF descriptor.

The orientation is estimated by firstly calculating the Haar wavelet responses in x and y directions within a circular neighbourhood of a radius of $6\boldsymbol{s}$, with $\boldsymbol{s}$ is the scale of the detected key point. The sampling step is chosen to be $\boldsymbol{s}$, and the size of the wavelets are $4\boldsymbol{s}$. Integral images is applied for fast filtering. The Haar wavelet filters is shown in Figure 2.6, with the dark parts having the filter weights of -1 and the light parts having the filter weights of +1.

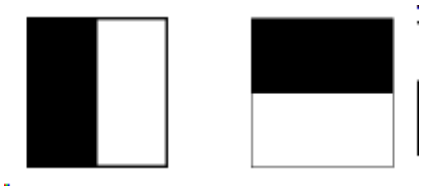
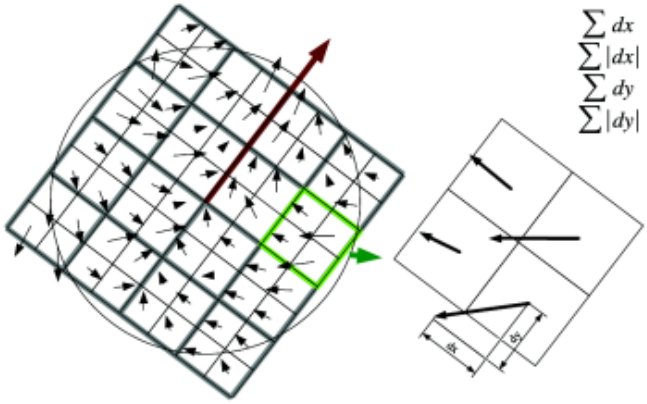


Figure 2.6: Haar Wavelet

After the wavelet responses are calculated and weighted with a Gaussian filter centred at the key point, the responses are presented as points in a 2D space. The dominant orientation is estimated by calculating the sum of all responses within a sliding window of $\frac{\pi}{3}$. The responses are summed as one presenting vector. The vector with the greatest length is chosen as the defining factor for the object's orientation.

Then, a square region centred around the interest point the oriented along the orientation is constructed with the size of $20\boldsymbol{s}$. This region is split up in smaller 4x4 square sub region for important spatial information preservation. Haar wavelet responses is computed for each sub-region at 5x5 regularly spaced sampled points. The way that descriptor is calculated as demonstrated in Figure 2.7[14].



To build the descriptor, an oriented quadratic grid with 4×4 square sub-regions is laid over the interest point (left). For each square, the wavelet responses are computed. The 2×2 sub-divisions of each square correspond to the actual fields of the descriptor. These are the sums dx, $|dx|$, dy, and $|dy|$, computed relatively to the orientation of the grid (right).

Figure 2.7: SURF Descriptor

There is also a matching algorithm of the SURF. However, due to the scope of this thesis, the mentioned algorithm is not discussed here. Instead, the FLANN-based matching method is used, and is discussed in the next Chapter.

2.1.4 FLANN

The Fast Library for Approximate Nearest Neighbours (FLANN) is a collection of algorithms for performing fast approximate nearest neighbour searches. It is able to choose the best algorithm and optimum parameters depending on the given dataset. The library is written in C++ and can be applied to either C, MATLAB or Python [16].

2.1.4.1 Nearest neighbour search

The Nearest neighbour search (NNS) is the optimization problem of finding the point in a given set that is most similar to a given point: the less similar the objects, the larger the function value. In 1973, Donald Knuth had defined the NNS problem as follows: given a set **S** of points in a space M and a query point $q \in M$, find the closest point in **S** to q [19].

Various solutions to this problem have been proposed and verified. However, the "curse of dimensionality" – an informal observation from the users – states that there is no exact solution for the NNS in high-dimensional Euclidean space using polynomial pre-processing and polylogarithmic search time [8].

There exist two main approaches to solve the problem: exact method and approximation method. While the exact method is complex to calculate and has a limited usage, it gives really accurate result as compensation. Meanwhile, the approximate nearest neighbour search is allowed to return points who is at most c times the distance from the query to its nearest points. In many cases, this approximate nearest neighbour is as good as the exact one, while the corresponding computational demand that must be done is significantly lower [13].

2.1.4.2 Fast Library for Approximate Nearest Neighbour

The mentioned library was first introduced in 2009 at the International Conference of Computer Vision Theory and Application by Marius Muja and David G. Lowe [16]. At that time, it was described as a system that would take any given dataset and desired accuracy, then use those to automatically determine the best algorithm and parameters value [16]. It was also released as an open source project on github at that time. The library was being updated regularly until its final stable build, which is version 1.8.4, was released in January 2013.

The optimal algorithm for fast approximate nearest neighbour search greatly depends on the structure of the dataset and the search precisions. The performance of the algorithm is also greatly influenced by its set of parameters, which include the number of randomized trees in the case of kd-trees or the branching factor and number of iterations in the case of hierarchical k-means tree. Therefore, the problem of choosing the optimal algorithm can be simplified to determining the parameters that give the best solution. Depending on the case, each of these three factors has a different importance. Therefore, a controller to the relative importance of the three factors is establish as followed:

$$cost = \frac{s + w_b b}{(s + w_b b)_{\text{opt}}} + w_m \tag{9}$$

where w_b is build time weight, w_m is memory weight, s represents the search time, b represents the tree build time, and $m = \frac{m_t}{m_d}$ represents the ratio of memory used for the tree to the memory used for the data. The lower the cost, the better the algorithm is.

The best algorithm and the optimum parameters are found by conducting a global exploration of the parameters, and later performing a local tuning of the best parameters. The first step is done by sampling the parameters space at multiple points and choose the values that give minimal costs. The second step is applying the Nelder-Mead downhill simplex method to locally fine tune the best parameters.

This optimization could be run on the full dataset to give the most accurate result. However, experiments have shown that just using 10 percent of the dataset is sufficient to find the values that perform close to the optimum parameters. The optimization only needs to be applied once for each type of dataset, and those parameters are recorded for applied for all future datasets of the same type.

2.1.5 ORB

Oriented FAST and Rotated BRIEF, which is abbreviated as ORB, is a new and open sourced algorithm developed in 2011. It is a fusion of Features from Accelerated Segment Test (FAST) keypoints detector and Binary Robust Independent Elementary Features (BRIEF) descriptor calculator. This has proven to be a more efficient and accurate when compared to SURF and FLANN in recognizing 2D pattern.

2.1.5.1 FAST and Oriented FAST

Features from Accelerated Segment Test, or FAST in short, is a corner detection method for extracting feature points. It was first published in 2006, and continues to be one of the most efficient keypoints detection algorithms on the market [12].

It speculates that, in order for a pixel **P** to be considered a corner, there have to be at least **N** contiguous pixels belonging to a circle (**P**,**R**) with the intensity difference **T** compared to the center pixel **P** [17].

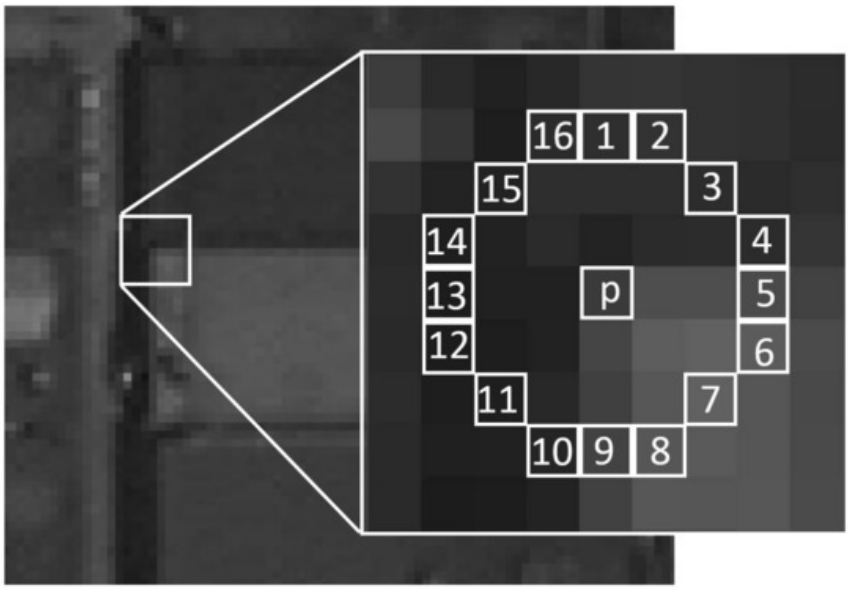


Figure 2.8: FAST Corner Detection

For example, on Figure 2.8, it could be observed that compare to the

pixel **p**, there are multiple pixels within a 3 pixel-radius with a great intensity difference compare to it such as the pixels numbered form 11 to 16, as well as those numbered from 6 to 8. Therefore, the point **p** is considered a corner, and is detected as one keypoint.

However, as the FAST algorithm only detects corners but not the orientation of the features, it can be quite tricky in recognizing rotated pattern. Therefore, the Oriented FAST has improved the algorithm by adding some modifications.

First, the weighted centroid of the patch with the located corner is computed. The direction of the vector from the corner point to the centroid is considered the orientation of the features. The moment of the patch is defined as:

$$m_{\mathrm{pq}} = \sum_{x,y} x^p y^q l(x,y) \tag{10}$$

With these moments, the centroid might be determined as:

$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}) \tag{11}$$

A vector from the corner's center O to the centroid C can be constructed. The resulted vector $\overrightarrow{\mathrm{OC}}$ is the orientation vector, and the orientation of the path is [20]:

$$\theta = arctan2(m_{01}, m_{10}) \tag{12}$$

2.1.5.2 BRIEF and Rotated BRIEF

BRIEF, which is an abbreviation for Binary Robust Independent Elementary Features, is an efficient and fast feature point descriptor. It was first published in 2010 as a way to speed up matching and reduce memory consumption compared to other algorithm such as SIFT or SURF.

It should be first noted that SIFT uses at least 512 bytes for every feature, and SURF takes around 256 bytes. Generating a vector for thousands of features are no small task for any system, especially for the embedded ones [9].

Therefore, instead of computing a long descriptor, BRIEF algorithm builds short descriptors directly by comparing the intensities of pairs of points in a patch.

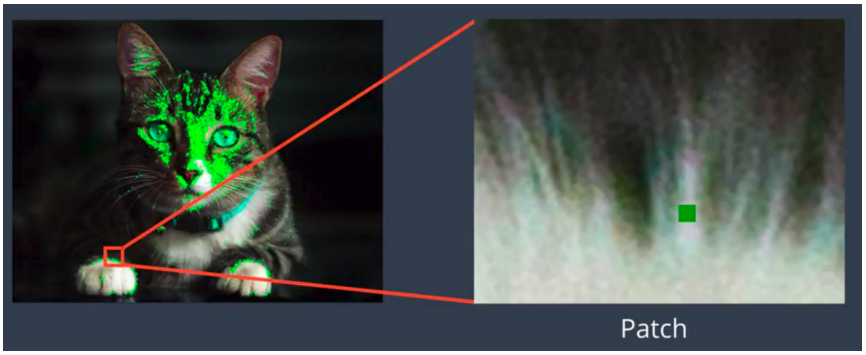


Figure 2.9: Image Patch Example

After smoothing the image patch $\mathbf{p}$ of size S x S, the test τ is defined as:

$$\tau\left(\mathbf{p};\mathbf{x},\mathbf{y}\right)=\begin{cases}1 & if\ \mathbf{p}\left(\mathbf{x}\right)<\mathbf{p}(\mathbf{y})\\ 0 & otherwise\end{cases} \tag{13}$$

where $\mathbf{p(x)}$ is the pixel intensity in a smoothed version of $\mathbf{p}$ at $\mathbf{x}=(u,v)^{T}$. Choosing a set of n_d ($\mathbf{x}$,$\mathbf{y}$) location pairs uniquely defines a set of binary test. Therefore, the BRIEF descriptor is the n_d – dimensional bit string

$$f\left(\mathbf{p};\mathbf{x},\mathbf{y}\right)=\sum_{1\leq i\leq n_d}2^{i-1}\ \ddot{\mathrm{I}}_{,,}\left(\mathbf{p};\mathbf{x},\mathbf{y}\right) \tag{14}$$

with n_d =128, 256 or 512 is the length of the binary feature vector. Therefore, it only takes $n_d/8$ bytes for storing a descriptor [20].

However, one of the problems for this algorithm is that it performs poorly with rotation. Therefore, by utilizing the θ computed from the Oriented FAST, ORB can "steer" the BRIEF descriptor according to the orientation of those keypoints. For any features set of n binary test at location ($\mathbf{x}$,$\mathbf{y}$), define the 2 x n matrix

$$S = \begin{pmatrix} x_1, \ldots, x_n \\ \hline y_n, \ldots, y_n \end{pmatrix} \tag{15}$$

A "steered" version of the matrix can be constructed using θ and the corresponding rotation matrix R_θ :

$$S_\theta \;=\; R_\theta \;\; S \tag{16}$$

Then, the steered BRIEF operator becomes:

$$gn(p,\ \theta) \;:= fn(p)|(xi, yi) \in S\theta \tag{17}$$

2.1.6 BruteForce Matcher

For a given keypoint K1 from the first set, the BruteForce algorithm takes every keypoint in the second set and calculates the distance. The keypoint K2 with the smallest distance will be considered its pair.

While principle-wise the most inefficient algorithm, when presented with a small number of descriptors, the result of this algorithm is normally the fastest since all it does is to perform procedural computation.

2.2 Chess Engine

2.2.1 MiniMax Algorithm

Minimax is a decision-making algorithm, typically used in a turn-based, two player games, the first player which is the computer and the second player which is the human. The theorem was proven by John von Neumann in 1928[7]. Minimax is superior to Brute-force algorithm due to the algorithm tries to find the best legal move for the computer even if the human plays the best move for him, when Brute-force only tries to find the best legal move for the computer and gives random results if there is no best legal move for it. This means that it maximizes the computer score when it chooses the computer move, while minimizing that score by choosing the best legal move for the human when it chooses the human move. For example, we have a search tree in Figure 2.10 with "RED" represents the highest score the computer can get whereas "BLUE" represents the highest score of the human player.

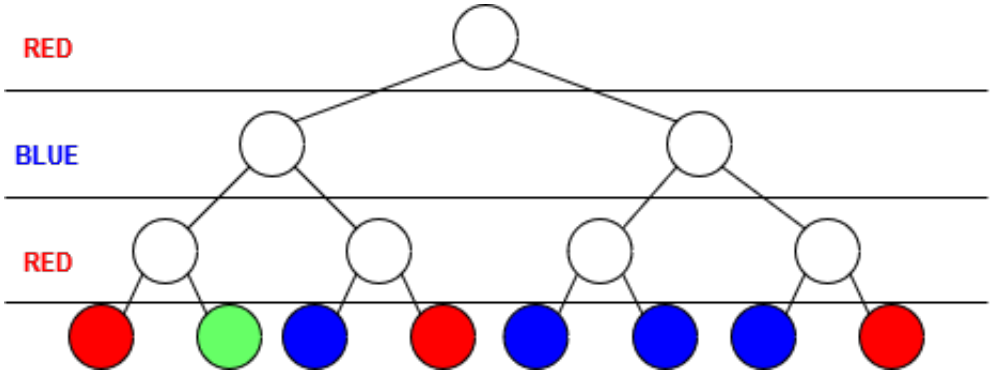


Figure 2.10: Minimax representation: Search tree

When the score is returned from the leaf node the "RED" node is reached, therefore only "RED" scores are chosen. After this step we can see that both child branches of the outer most left branch have "RED" score, which is shown in Figure 2.11.

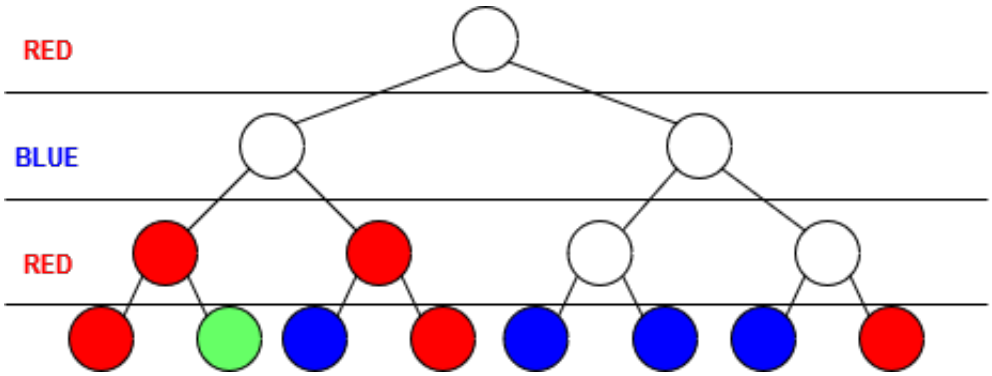


Figure 2.11: Minimax representation: Lowest node

However, when the score are returned to the "BLUE" node, there is no "BLUE" score to choose from the list, therefore one "RED" score is chosen as presented in Figure 2.12.

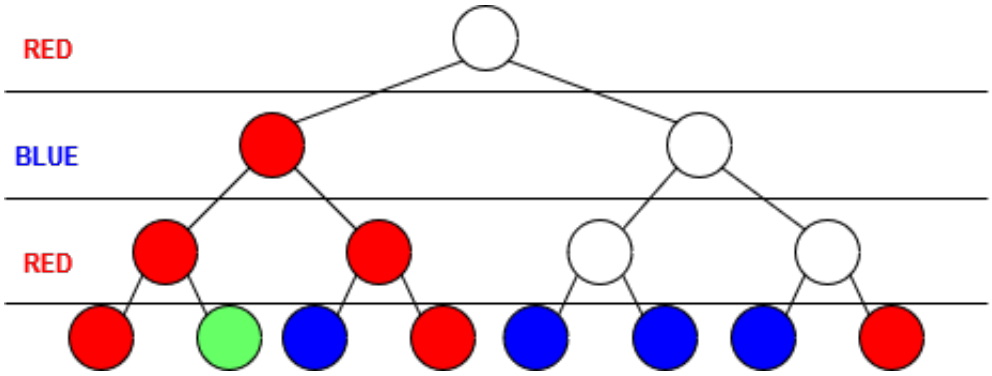


Figure 2.12: Minimax representation: Operation at "BLUE" node

Now the algorithm switch to the right handed branches with the same step as mentioned above. In Figure 2.13, after going through all the score, the highest node "RED" has one "RED" score at the left branch and one "BLUE" and obviously the left branch will be chosen. With this example, it can be observed that both sides are taken into consideration to make the final result, therefore Minimax algorithm provides high efficiency prior to the Brute-force algorithm.

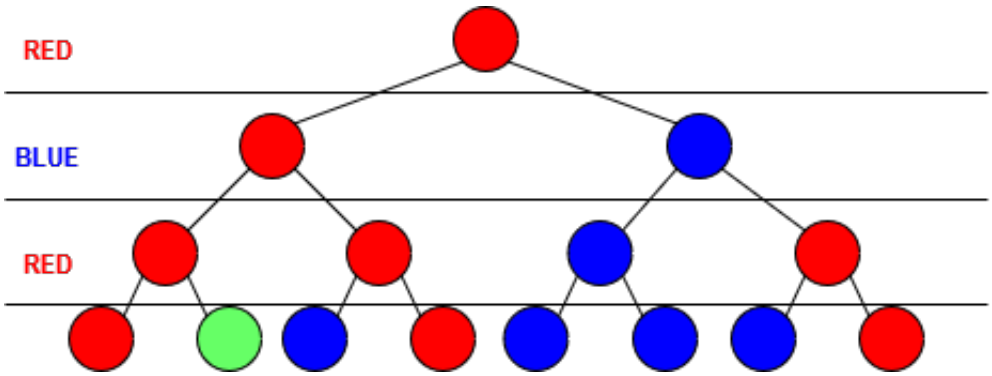


Figure 2.13: Minimax representation: Final result of the algorithm

Minimax is widely used in games such as Tic-Tac-Toe, Chess, Xiangqi,... because they share similar traits. In such games at least two players are indicated and every player has a specific number of possible strategies to choose from which result is a direct consequence of the combination of the chosen strategies of the players. Therefore, these game types are claimed to be perfect information games. According to Yngvi Bjornsson and Tony Marsland, the value of a such games is the outcome with perfect play by both sides and can be found by recursively expanding all possible continuations from each game state until game states with a known outcome are reached [6]. The minimax rule is then used to propagate the value of those outcomes back to the initial state. The MiniMax algorithm will create a game state tree from the current game state for anticipating possible game states that could happen in order to obtain the optimal next moves. During the tree creation process, players are switched in order.

The number of times the players being switched is called search depth, which value decreases at each switching turn. Depending on the depth value, the algorithm can create corresponding game state tree and increases the accuracy of the final result. At the leaf nodes or depth equals zero, value of board score is calculated and returned to the upper nodes without any further move calculations. In the first depth, the highest legal moves a player can make is 44 and for each legal move there might be up to 44 moves the opponent can make in the lower depth. Hence, the number of legal moves or "Nodes" can be calculated as:

$$Maximum\ Nodes = 44^k, k = maximum\ fixed\ search\ depth$$

2.2.2 Alpha Beta Pruning

When approaching to lower depth, the Minimax algorithm presents high performance time as the number of nodes is expanded by power scale. Figure 2.14 represents the number of nodes that needs to be handled from depth 1 to depth 5:

Depth	Nodes
1	44
2	1936
3	85184
4	3748096
5	164916224
k	44^k

Figure 2.14: Depth of search tree

Therefore, Alpha-Beta pruning method is used in combination with the Minimax algorithm to handle this problem. Alpha-Beta pruning is not actually a new algorithm but rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows low performance time and higher precision as deeper levels in the game tree can be reached. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta which handle the pruning process based on the node being Max or Min.

For further understanding of this method, we revise the example search tree used in the previous section but with numbers and letters for easier tracking of the process. In Figure 2.15, Alpha and Beta is transfer from the highest depth of the tree, with Alpha being the lowest score the computer can get, which is assumed -9999 and Beta with opposite configuration being 9999 and Alpha will only be changed at "Max" node and Beta being changed at "Min" node respectively.

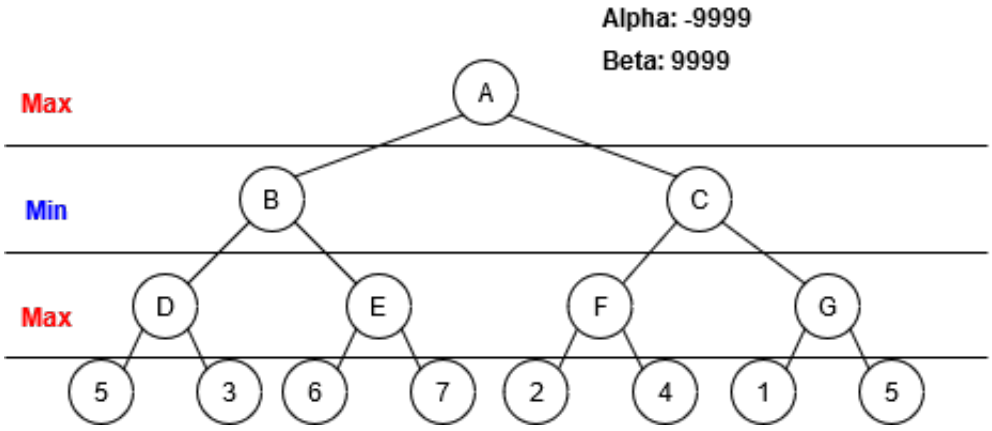


Figure 2.15: Alpha Beta representation: Alpha and Beta characteristic

- First, Node A will have to choose between B and C and B will be chosen first. For Node B, D will be chosen first and score 5 will first be returned to D.
- As D is a "Max" node, Alpha will be compared with and changed to 5 as it is having lower value. Then Alpha will be compared with Beta to check whether other branches of D should be checked. As Alpha is still lower than Beta, score 3 is returned to D and as 3 is lower than Alpha-now having value 5, the final value of D is 5.
- D now returns 5 to B, which is compared with Beta and changes Beta to 5. Therefore, the value at B is guaranteed 5 or lower. Node B now calls E.
- At E the value of Alpha and Beta is -9999 and 5 respectively because Beta is changed at B and passed down to E. First E look at its left branch which is 6. The Alpha at E is changed to 6 and at this moment the pruning condition when Alpha is larger than Beta is satisfied. Hence, the right branch of E is prunned and E will return 6 to B.
- At B, because the value of E is not lower than Beta-currently is 5, therefore the final value of B is 5, leading the search tree shown in Figure 2.16.

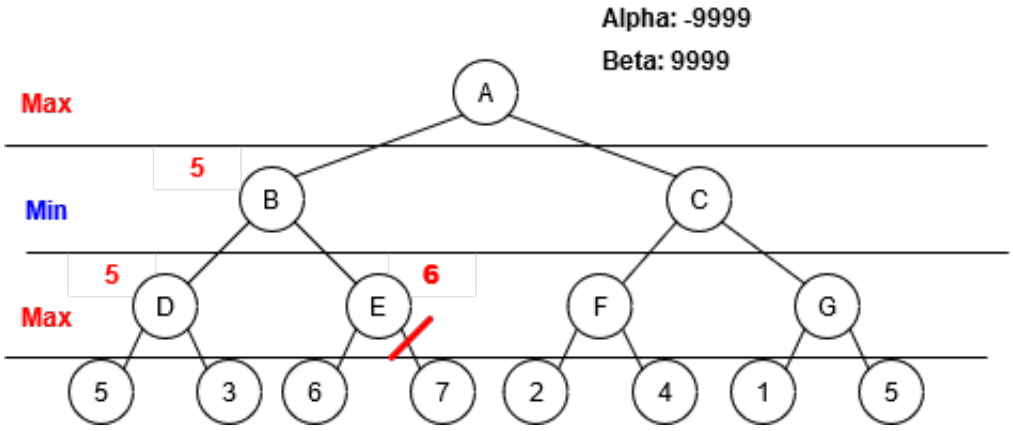


Figure 2.16: Alpha Beta representation: Score at left branch of A

- Now B returns 5 to A and Alpha at A is changed to 5. The score at A is guaranteed 5 or higher. A now calls C, C calls F with Alpha and Beta being 5 and 9999 respectively and 2 is first returned to F. With the same process, F holds the value of 4 with Alpha still being 5 as no child value of F is higher than it. F is returned to C.

- At C, Alpha is 5 and Beta is 9999 and with the new value returned by F Beta is changed to 4, satisfying the pruning condition by being lower than Alpha. Therefore, the entire right branch of C is prunned. The score of C is now returned to A. A-currently is 5, will have its score compared to 4 and as this is a "Max" node, the value of A will remain 5. Hence, the optimal result of the search tree is 5, which is shown in Figure 2.17.

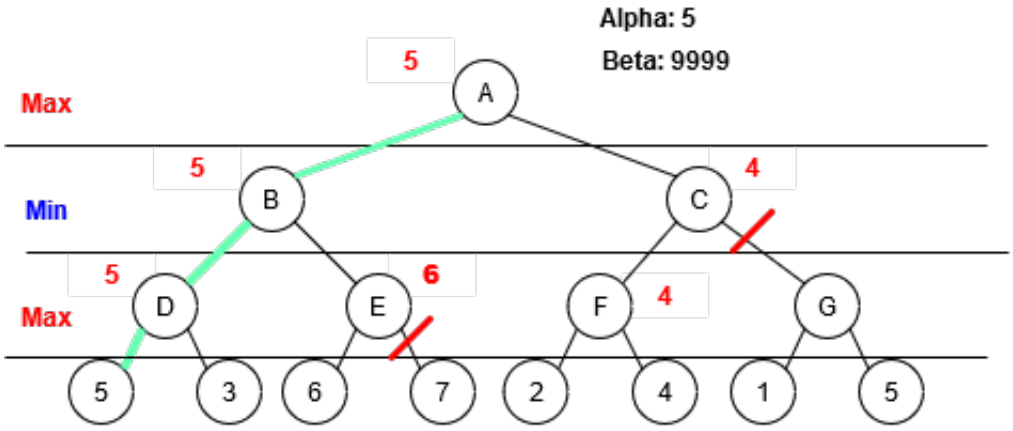


Figure 2.17: Alpha Beta representation: Final result

With the example above, it can be observed that the algorithm with the help from Alpha-Beta pruning method reduces the search tree size quite significantly and the efficiency is even more transparent when approaching to lower depths.

After simulating the chess engines with alpha beta pruning at depth greater than three, problems start to arise. As the game advances closer to the end, EndGame moves occurs at a more frequent rate, which leads to the algorithm missing the true EndGame move as it belongs to later branches which are prunned because the most left branch has EndGame move value. This problem and solution are discussed in section 4.2.

2.3 Scara Robot Design

2.3.1 Scara Arm robot

2.3.1.1 Scara Arm model

SCARA robots arm is an option for small robotic assembly applications. SCARA is an abbreviate for Selective Compliance Articulated Robot Arm, meaning it is compliant in the X-Y axis, and rigid in the Z-axis. The SCARA configuration is suitable for pick and place an object from one place to another place without occupying the work space after processing.This would help to facilitate the image detection process.

Figure 2.18: SCARA robot model

The SCARA's structure consists of two arm joints,one at the base and the other at the intersection of two arms segment, an end effector for pick and place purpose at the end of the second arm segment.Each arm segments and the end effector are controlled independently by a motor.

2.3.1.2 Scara Arm model

Robot Kinematic model is a set of formulas which is used for computing the position and orientation of robot end effector relative to the original point of its coordination system.

There are two type of robot kinematics,forward kinematic and inverse kinematic.Forward kinematic uses joint angle as input and output the end effector position in Cartesian coordinate.Inverse kinematic converts the Cartesian coordinate of end effector to joint angles. [23]

The forward kinematic of SCARA arm is calculated by trigonometric method as follow :

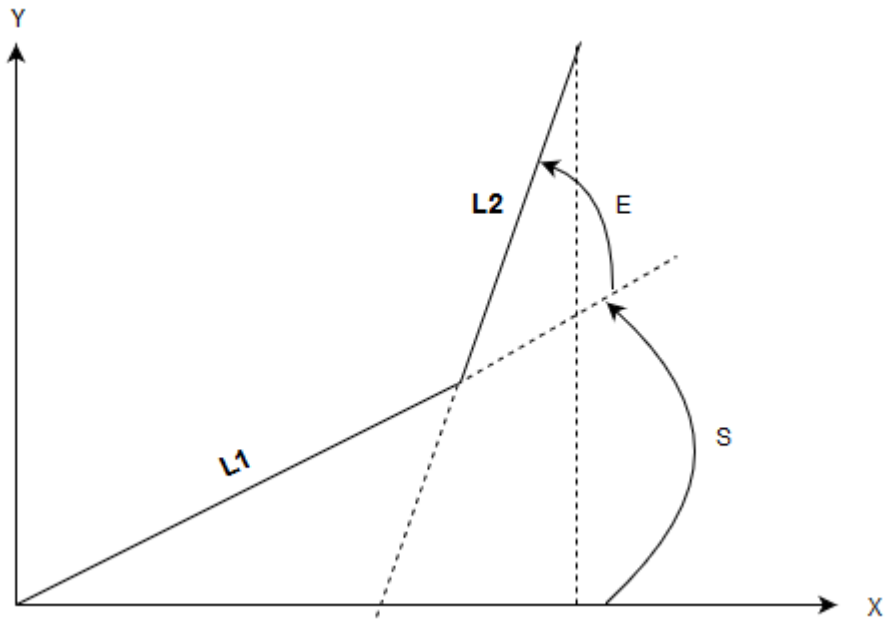


Figure 2.19: forward kinematic

$$\mathbf{X} = L1 * cos(S) + L2 * cos(S + E) \tag{18}$$

$$\mathbf{Y} = L1 * sin(S) + L2 * sin(S + E) \tag{19}$$

There are two configurations of the SCARA arm for calculating inverse kinematic,which is depicted in 2.20, elbow-down and elbow-up.These two configurations have similar formulas.Therefore,only the elbow-down inverse kinematic is discussed.

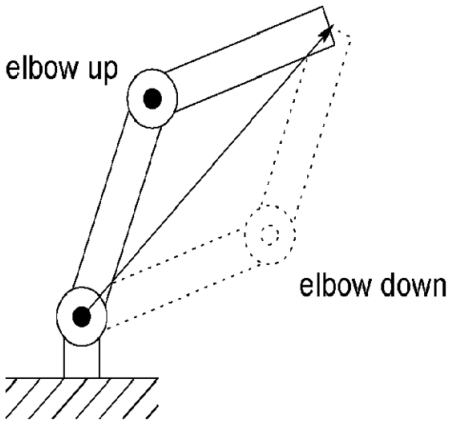


Figure 2.20: elbow-up and elbow-down configuration

The inverse kinematic is derived by trigonometric as follows:

$$E = arccos\left(\frac{X^2 + Y^2 - L_1^2 - L_2^2}{2L_1L_2}\right) \tag{20}$$

$$S + Q = arctan\,(Y, X) \tag{21}$$

$$Q = arccos\left(\frac{X^2 + Y^2 + L_1^2 + L_2^2}{2L_1\sqrt{X^2 + Y^2}}\right) \tag{22}$$

$$S = (S + Q) - Q \tag{23}$$

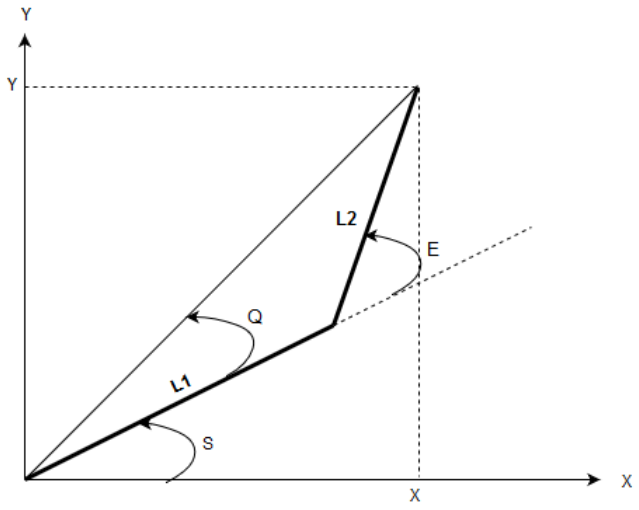


Figure 2.21: Inverse kinematic

The above formulas has a problem with **arctan function** when applying to control the robot arm position.The arctan function is not a one-to-one function,one given input results to multiple outputs and its output range is from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$.

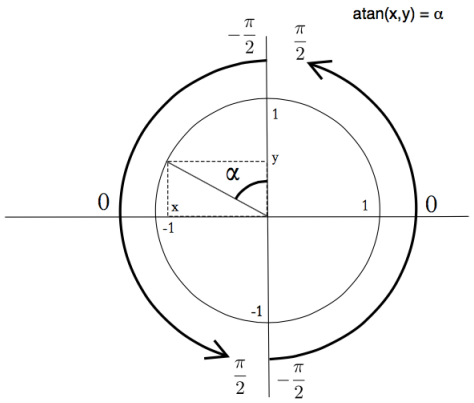


Figure 2.22: arctan output range

This lead to an ambiguous joint angle,the orientation and the appropriate value of the joint angle can not be determined.For example,given the input -1 for the function with $x = -5$ and $y = 5$,the arctan output is $-45°$ but the expected result is $135°$.Using the $-45°$ joint angle result, the robot arm can not move to the right position,$135°$, with the orientation is counter clock - wise for positive angle.

For the **arcos** function, its output range is $[0, \pi]$ and has counter-clockwise positive direction.This is coincide with the movement range of the second arm segment. Therefore, the output result can be used without any ambiguity in position and orientation.

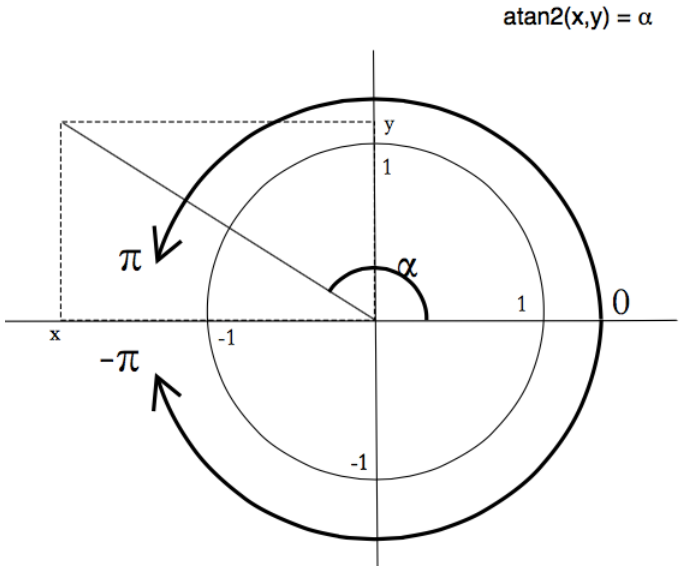


Figure 2.23: arctan2 output range 1

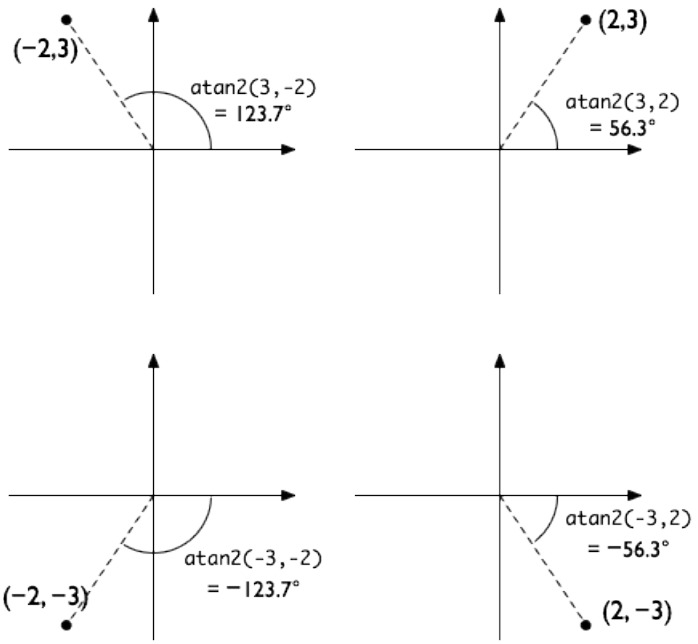


Figure 2.24: arctan2 output range 2

To overcome the problem of using arctan function, arctan2 is a solution. arctan2 function outputs range is $(-\pi, \pi]$.This help to determine appropriate angle value and correct orientation with respect to counter clock-wise direction.The example is illustrated in figure 2.24,with the output result for different angle,there is no ambiguity in the angle joint and its orientation.

2.3.2 Stepper Motor

Stepper motor is a brush-less DC motor that divides a full number of rotation into a number of equal discrete step.The motor position can be control to move and hold at one of these step by electrical pulse.It gets one step movement for a single pulse input.

The angle that the motor shaft rotates for a single pulse is called step angle,which is expressed in degree.For example, a motor with 200 steps per revolution has 1.8°.Since each pulse can cause motor rotate a precise angle, the motor to can be controlled without feedback.

Stepper motors work on the principle of electromagnetism. There is a soft iron or magnetic rotor shaft surrounded by the electromagnetic stators. When the stators are energized the rotor moves to align itself along with the stator (in case of a permanent magnet type stepper) or moves to have a minimum gap with the stator (in case of a variable reluctance stepper).

2.3.2.1 Stepper Motor Types

There are three types of stepper motor.They are different at how rotor and stator part are constructed for each types.

- Permanent Magnet stepper motor

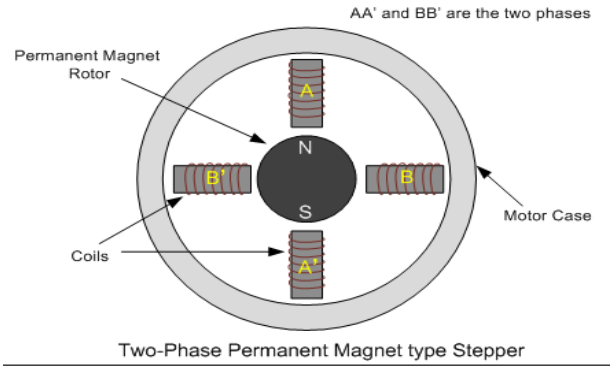


Figure 2.25: Permanent stepper motor

The rotor and stator poles of a permanent magnet stepper are not teethed. Instead the rotor have alternative north and south poles parallel to the axis of the rotor shaft.

When a stator is energized, it develops electromagnetic poles. The magnetic rotor aligns along the magnetic field of the stator. The other stator coils is then energized in the sequence so that the rotor moves and aligns itself to the new magnetic field. This way energizing the stators in a fixed sequence rotates the stepper motor by fixed angles.

- Variable Reluctance

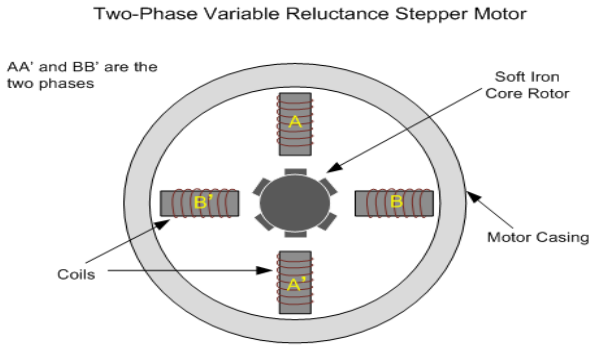


Figure 2.26: Reluctance Stepper Motor

The rotor of Variable Reluctance stepper motor construct from non-magnetic soft iron rotor with teeth as figure.Different from Permanent stepper motor, The teeth of rotor is attracted and aligned to stator pole based on the principle that minimum reluctance occurs with minimum gap [22]

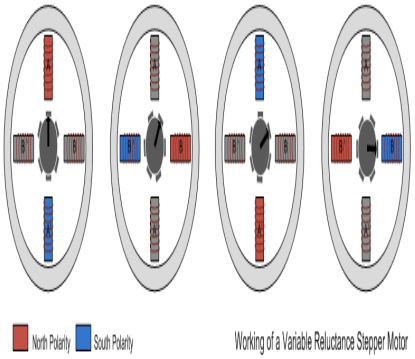


Figure 2.27: Reluctance Stepper Motor Mechanism

The teeth of the rotor are designed so that when they are aligned with one stator coil they get misaligned with the next stator coil. Now when the next stator coil is energized, the rotor moves to align its teeth with the next stator coil. [22] This way energizing stators in a fixed sequence completes the rotation of the step motor

- Hybrid Stepper motor

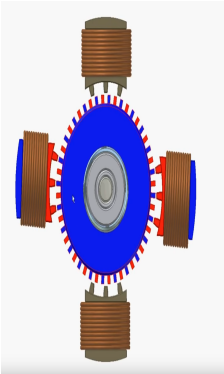


Figure 2.28: Hybrid stepper motor

Hybrid stepper motor structure and work principle is the combination of Variable Reluctance and Pemanent stepper motor.It has teeth on both rotor and stator.The rotor is made of permanent magnet and has two cup for north and south pole as picture below

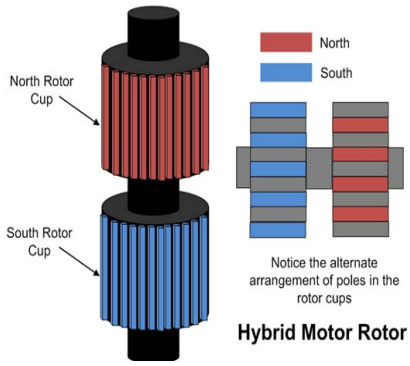


Figure 2.29: Hybrid Stepper Motor Stator

Rotation of a hybrid stepper motor is produced with the same control method as a permanent magnet stepping motor, by energizing individual windings in a positive or negative direction. When a winding is energized, a north and south pole are created.These generated poles attract the permanent poles of the rotor and the metal rotor teeth, which make the teeth of the stator aligned to the teeth of the rotor.[24]

This type of stepper has higher resolution due to arrangement of teeth on both stator and rotor part and holding torque in compare to the other two types[]

2.3.2.2 Stepper Motor Control Method

There are several different ways to drive the stepper motor.The first method is Wave Drive.In this method, one electromagnetic stator(coil) is energized at a time

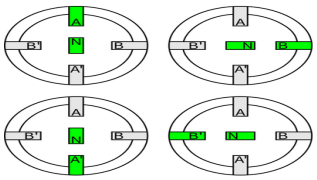


Figure 2.30: Wave Drive

When the coil A is energized (a south pole) as shown in green, it attracts the north pole of the rotor. Then when the coil B is energized and coil A is switched off, the rotor rotates 90°.This method provide a large steps angle and small torque because there is only one pole is active at a time.

The second methods is Two Phase, which provides higher torque output by energizing two adjacent poles at a time and work with the same principle of Wave Drive method.this does not improve the step angles or resolution of the motor.The motor will make a cycle with 4 steps.

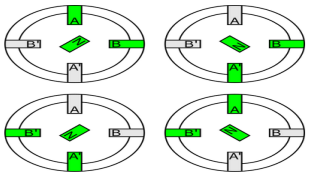


Figure 2.31: Two-Phase

The two above methods provide a low resolution for the motor, which is 4 steps per revolution.To increasing the resolution,either one or two pole is activated at any specific time.This third drive method is called One - Two phase method.

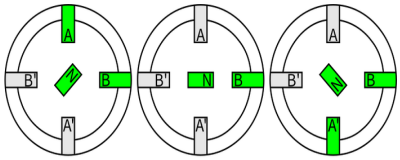


Figure 2.32: One Two Drive

At first, two coil A and B are activated, which attract the north pole of the rotor and rotate it 45°.After that, only the coil B is energized to rotate the rotor next 45°.The process is repeated by activating two coil and one coil alternately.This make the motor resolution increases from 4 steps per revolution to 8 steps per revolution.

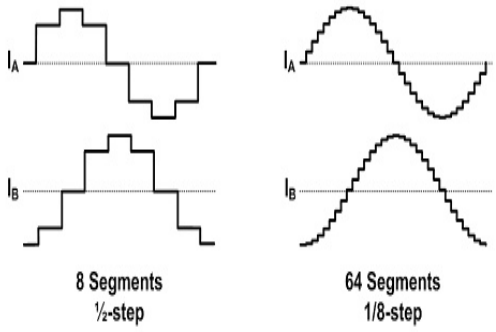


Figure 2.33: Micro step current

The final drive method is Micro step method,which increases the resolution of motor drastically.Instead of fully activating the coils as above methods,the current amplitude and direction through the coils is controlled so that the rotor can rotate with smaller steps angle.The current is adjust along the sine wave so that no coil is fully activated or inactivated.

Chapter 3

Implementation

3.0.1 Piece class Design

3.0.1.1 Requirements

The Piece class is designed with the purpose of storing every necessary piece of information about each piece, which include:

- The name of the piece in string format, i.e.: BlackGeneral;
- The type of the piece, which is an integer ranging from 0 to 6. The types are arranged in the descending order of values;
- The team of the piece: -1 stands for black team (the machine), and 1 stands for red team (the player);
- The real position (X-Y position) of the piece. This is the coordination for the center of the piece on the image;
- The board position (Column – Row position) of the piece. This is the position of the piece on the board. A position of (8,9) means that piece is on the 8^{th} column, 9^{th} row.

There was a debate of whether it is necessary to keep the image and the calculated descriptor of a piece in this object. However, it is deemed to be unnecessary since the program can quickly cuts the piece from the original image using its real position, calculates its descriptor and compares it to the database without the need of an intermediate allocator.

3.0.1.2 Piece initiation

In order to initiate a piece, there are three required types of data: the type of the piece, the team of the piece and its real position.

```
Piece::Piece(int _pieceType, int _pieceTeam, std::pair<int,int>
_realPos)
```

There is no other constructor except this one, which means this is the only way to create a new `Piece` object. The `pieceType` and `pieceTeam` are stored into the object's respective private members: `pieceType` and `pieceTeam`. Then, the name is generated from these two types of data:

```
if (this->pieceTeam == 1)
    {
        this->pieceName = "Red" + name[pieceType];
    }
    else
    {
        this->pieceName = "Black" + name[pieceType];
    }
```

where `name` is a vector containing the enumeration of piece's name:

```
std::vector<std::string> name = {"Cannon","Car","Horse",
                "Elephant","Advisor","General","Soldier"};
```

This vector is ordered in the descending order of the piece's value. The cannon is the most versatile piece, which has the highest score and on the first place of the list, while the soldier is the last. This is also the order in which the database is organised.

The real position is stored into the private member `realPos`, from which the board position `boardPos` is calculated as follows:

```
    this->boardPos = std::make_pair(
        (int)(_realPos.first/pieceWidth)+1,
        (int)(_realPos.second/pieceWidth)+1);
```

It should be noted that every position is a pair of two integers, which represent the horizontal and vertical position respectively. `pieceWidth` is a predefined value, which is the height of each chess box on the image. In this set up of the thesis, this constant is set to be 120px.

3.0.1.3 Other functions

All attributes of a piece is set up to be a private member of the `Piece` class to prevent unwanted tampering with the stored data. Therefore, it is required to have some form of input/output functions controlling the flow of data back and forth in the object. Input is handled through the `contructor` of the object already. Therefore, only four pieces of information are required to be extracted from this object:

```
std::pair<int,int> getRealPos();
std::pair<int,int> getBoardPos();
std::string getName();
int getType();
```

Those functions only contain one command: `return ...`. Its main function is to extract the data being hold by those members. That piece of data is used to create two files containing data for the ***green pieces vector*** and the ***red pieces vector***.

3.0.2 Hough Circle Implementation

3.0.2.1 Program Flow

TThe image of the board is formatted as a `cv::Mat` object named `boardMat`. The `cv::Mat` object is cloned and converted into a grayscale image called `grayMat`, which is then fed into the Hough Circle Transformation.

This image goes through the process of Canny Edge Detection, hysteresis edge detection and Hough circle recognition in order to export an array of circles within a designated radius range.

After the process, a three-dimensional vector of floating point numbers under the type of `cv::Vec3f` is generated. The first and second dimension of one element represent the x and y coordination of the circle, while the third dimension represents its radius.

3.0.2.2 Testing condition

In order to get the accurate parameters for the transformation, a program with 6 sliders is created for easier testing.

Due to the boundaries of the piece's pattern being relatively small, the value of `dp` should be larger than 1.2 in order to correctly recognize the outer circle. With at most 12 pieces can be arranged on a single row of the board, the `minDist` should be equal to the width divided by 12. With the resolution of 1920x1080px, each piece should be around 100x100px in size. Therefore, the range should be from 25 (`minRad`) to 75 (`maxRad`) for easier detection.

The other two parameters are estimated using track bars. Then those now defined parameters are used to conduct multiple tests for ensuring the stability and performance of the program.

3.0.2.3 Implementation

After various tweakings, we find the following parameters to be the most desirable in terms of performance:

```
//Variables and functions for Hough Circle Transformation
const int dp = 150;
const int minDist = 20;
const int CannyEdgeDetect = 45;
```

```
const int CircleCenterDetect = 100;
const int minRad = 25;
const int maxRad = 70;
```

Then, the function for HoughTrans is set up as follows:

```
std::vector<cv::Vec3f> HoughTrans(cv::Mat boardMat)
{
    std::vector<cv::Vec3f> circles;
    cv::Mat grayMat;
    cv::Mat copyMat = boardMat.clone();
    cv::cvtColor(boardMat,grayMat,cv::COLOR_BGR2GRAY);

    cv::HoughCircles(grayMat,circles,cv::HOUGH_GRADIENT,
    (double) dp/100,(double)boardMat.rows/minDist,
    CannyEdgeDetect,CircleCenterDetect,minRad,maxRad );
```

After this, the vector `circles` is created and filled with information about the detected circles. As a means of debugging and verifying the outcome, another function is called to draw the detected circles into the `copyMat` object.

```
for( size_t i = 0; i < circles.size(); i++ )
    {
        cv::Point center(cvRound(circles[i][0]),
        cvRound(circles[i][1]));
        int radius = cvRound(circles[i][2]);
        // circle center
        circle( copyMat, center, 3, cv::Scalar(0,255,0),
                        -1, 8, 0 );
        // circle outline
        circle( copyMat, center, radius, cv::Scalar(0,0,255),
                        3, 8, 0 );
    }
```

With the verification data set up, what remains to be done is to export the detected circle as a three-dimensional vector.

```
return circles;
```

```
}
```

This function takes in the `boardMat`, and return a `cv::Vec3f` object that contains all the detected circles in the image.

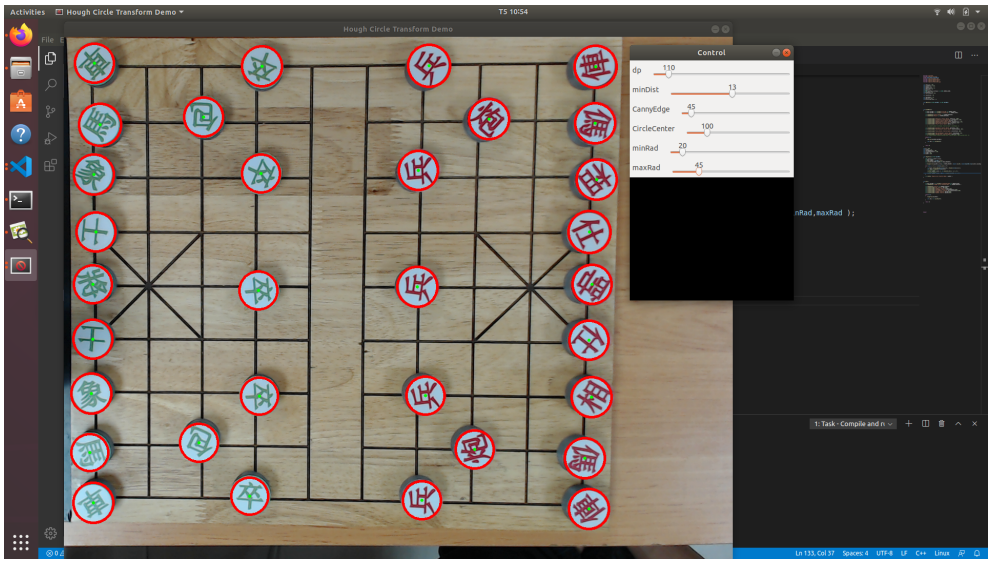


Figure 3.1: A Successful Circle Recognition

3.0.3 HSV Filter – Team Recognition

3.0.3.1 Program Flow

Using the `circles` vector from the Hough Circle Transformation, each pieces' images can be cropped from the board and stored into a vector of images.

These images are converted into HSV color space, and put into both the red filter and green filter, which have the output of `redMask` and `greenMask`. All the non-zero pixels are then counted for both masks, with that which has more pixels being the detected color.

3.0.3.2 Implementation

One image is processed through a 6 track bars program to define a rough set of parameters. These parameters are later tested under various lighting conditions to ensure the accuracy the algorithm.

```
//Variables and functions for HSV Filter
const int GreenLowH = 100;
const int GreenLowS = 0;
const int GreenLowV = 0;
const int GreenHighH = 165;
const int GreenHighS = 255;
const int GreenHighV = 255;

const int RedLowH = 160;
const int RedLowS = 0;
const int RedLowV = 0;
const int RedHighH = 255;
const int RedHighS = 255;
const int RedHighV = 255;
```

Using the above parameters, the boardMat is processed with the function, and turns out a pair of integer that represents the counts for green pixels and red pixels.

```
std::pair<int,int> HSVFilter(cv::Mat boardMat)
```

```
{
...
    return std::make_pair(greenCount,redCount);
}
```

At first, the 12 parameters are integrated into four different `cv::Scalar` objects named `greenLow`, `greenHigh`, `redLow`, `redHigh` as four thresholds for color filtering:

```
std::pair<int,int> HSVFilter(cv::Mat boardMat)
{
    cv::Scalar greenLow = cv::Scalar(GreenLowH,GreenLowS,GreenLowV);
    cv::Scalar greenHigh = cv::Scalar(GreenHighH,GreenHighS,GreenHighV);

    cv::Scalar redLow = cv::Scalar(RedLowH,RedLowS,RedLowV);
    cv::Scalar redHigh = cv::Scalar(RedHighH,RedHighS,RedHighV);
```

Then, the initial board image is converted into HSV colour space for the filter process. Two `cv::Mat` objects named `greenBoard` and `redBoard` are also created to store the resulted masks for green and red filters respectively.

```
    cv::Mat HSVMat;
    cv::cvtColor(boardMat,HSVMat,cv::COLOR_BGR2HSV_FULL);
    cv::Mat greenBoard, redBoard;
```

Figure 3.2: Sample of two masks

After everything has been initialized, the filter is started using the `cv::inRange` function. `greenBoard` and `redBoard` are now storing the filtered image in the color of green and red, respectively.

```
cv::inRange(HSVMat,greenLow,greenHigh,greenBoard);
cv::inRange(HSVMat,redLow,redHigh,redBoard);
```

The final step of this process is to count the number of non-zero pixels on both the green and red masks using the `cv::countNonZero()`, and then to export those as the results of the HSV filter function.

```
int greenCount = cv::countNonZero(greenBoard);
int redCount = cv::countNonZero(redBoard);
```

3.0.4 ORB and BruteForce

3.0.4.1 ORB versus SURF

After various tests using a `MassTest` program, the SURF algorithm has proven to be inefficient and ineffective in detecting the Chinese Chess pattern. This is due to the blob-detector behavior of the SURF algorithm. Since the chess pattern mainly consists of sharp edges and strokes, the blobs that SURF detected are easily mistaken for one another, which ultimately leads to the inaccuracy of the algorithm.

Various tests have been performed, with multiple parameters for the SURF object initialization. However, the accuracy only hits 80% at its highest. Meanwhile, the ORB gives much better results since it mainly tracks the edges of the patterns, which is vastly different from one type to another.

3.0.4.2 Program Flow

First, the database's descriptor is calculated. Then, for each member in the `redPiece` or `blackPiece` vectors, the descriptor of the cropped image is calculated, and then matched with each of the corresponding database by `BruteForce` matcher. The matches are then filtered using Lowe's distance test to arrive at the total number of good matches. The one with the greatest number of good matches is the type of the tested piece

3.0.4.3 Parameters Defining

The ORB object was first established using the following parameters:

- `nFeatures`: the maximum number of features to retain;
- `scaleFactor`: the pyramid decimation ratio. The higher the parameter, the smaller the pyramid size. Increasing this ratio will improve the processing time, but will reduce the covered area;
- `nLevels`: the number of pyramid levels;
- `edgeThreshold`: the size of the border where features are not detected;
- `firstLevel`: the level of pyramid to put the source image on;
- `patchSize`: size of the patch used by the oriented BRIEF descriptor;

- `fastThreshold`: the threshold for detecting the edge using FAST algorithm.

The `nFeatures` is set at 1500 since our pieces are only around 100x100px in size, so the value of 1500 should be sufficient. Other parameters are checked through a seven-trackbar program to get a rough set, then doing various tweakings to ensure the accuracy and stability of the outputs.

```
//Variables and functions for ORB Calculate and BFMatching
const int nFeatures = 1500;
const int scaleFactors = 126;
const int nLevels = 16;
const int edgeThreshold = 20;
const int firstLevel = 3;
const int WTA_K = 2;
const cv::ORB::ScoreType scoreType = cv::ORB::HARRIS_SCORE;
const int patchSize = 37;
const int fastThreshhold = 22;
const int ratioThresh = 70;
```

3.0.4.4 ORB Calculation

Suppose an image of a piece is import as a `cv::Mat` object, the ORB Calculation descriptor is calculated and is exported back as the result.

```
cv::Mat ORB_Calculate(cv::Mat baseData)
{
....
    return descriptor;
}
```

At first, the keypoints are stored as a `std::vector` of `cv::KeyPoint` objects. The blank descriptor is also generated as an empty `cv::Mat` object. The piece's image is imported from the constructor and then converted into grayscale color space for the detection and computing algorithm.

```
    std::vector<cv::KeyPoint> keypoint;
    cv::Mat descriptor;
    cv::cvtColor(baseData,baseData,cv::COLOR_BGR2GRAY);
```

Then, the ORB detector is created as a pointer to a `cv::ORB` object. This is declared as `cv::Ptr<cv::ORB>`. This declaration allows the object to be deleted after each loop, which makes handling and controlling the flow of program much easier since the constant re-declaration of such a complex object can have a negative effect on the stability of the function. The `detector` object is initialized with all the tested parameters to ensure that the function is going to work as expected.

```
cv::Ptr<cv::ORB> detector = cv::ORB::create(nFeatures,(float)
    scaleFactors/100, nLevels,edgeThreshold,firstLevel,WTA_K,
    scoreType,patchSize,fastThreshhold);
```

Then, the keypoints are detected by calling the `detect` function from the `detector` object. After that, those keypoints are computed into a single descriptor, and are stored in a `cv::Mat` object with the same names.

```
    detector->detect(baseData,keypoint);
    detector->compute(baseData,keypoint,descriptor);
```

3.0.4.5 BruteForce matcher

The BruteFroce matcher function `BF_Match()` takes in two descriptors as two `cv::Mat` objects, compares them and returns the number of good matches between the two.

```
int BF_Match(cv::Mat descriptor1, cv::Mat descriptor2)
{
...
    return good_matches.size();
}
```

The matcher is declared as a pointer to a `cv::DescriptorMatcher` object. Then, it is initialized by using two parameters:
`normType  = cv::NORM_HAMMING` and
`crossCheck = false`.
Since BF Matcher is being used with ORB descriptor, which is a binary string descriptor, a Hamming distance norm is required to compute and

compare the two descriptors. The `crossCheck` is disabled since our desciptors are not sufficiently large, so Lowe's ratio test is being used as an alternative checking condition in this algorithm.

```
cv::Ptr<cv::DescriptorMatcher> matcher = cv::BFMatcher::create
                                          (cv::NORM_HAMMING, false);
```

Since a knn-match is utilized here, the matches should be stored as a 2D vector of `cv::DMatch` objects. Then, two descriptors and the `matches` objects are fed into the `matcher->knnMatch()` function with the dimension of `2` to create a list of matches.

```
std::vector<std::vector<cv::DMatch>> matches;

matcher->knnMatch(descriptor1,descriptor2,matches,2);
```

After the matches are generated, they have to pass Lowe's ratio test to reduce the number of matches, leaving only the good matches. When performing a `knnMatch` with the dimension of 2, the matches are saved as a pair of best and second best point. The factor that defines the "goodness" of the match is the distance between the two keypoints. When the best point is not much different from the second best point, the match is considered not good and is eliminated. In the following piece of code, we can see that if the best point is smaller than the `ratio_thresh` multiplied with the second best point, it is considered a good match and is saved into the `good_matches` vector. The average `distance` of all the good matches is also calculated as a second means to compare multiple matches.

```
float ratio_thresh = (float) ratioThresh/100;
float distance = 0;
std::vector<cv::DMatch> good_matches;
for (size_t i = 1; i < matches.size(); i++)
{
    if (matches[i].size() != 0)
    {
        float bestPoint = matches[i][0].distance;
        float secondPoint = matches[i][1].distance * ratio_thresh;
        if (bestPoint < secondPoint)
```

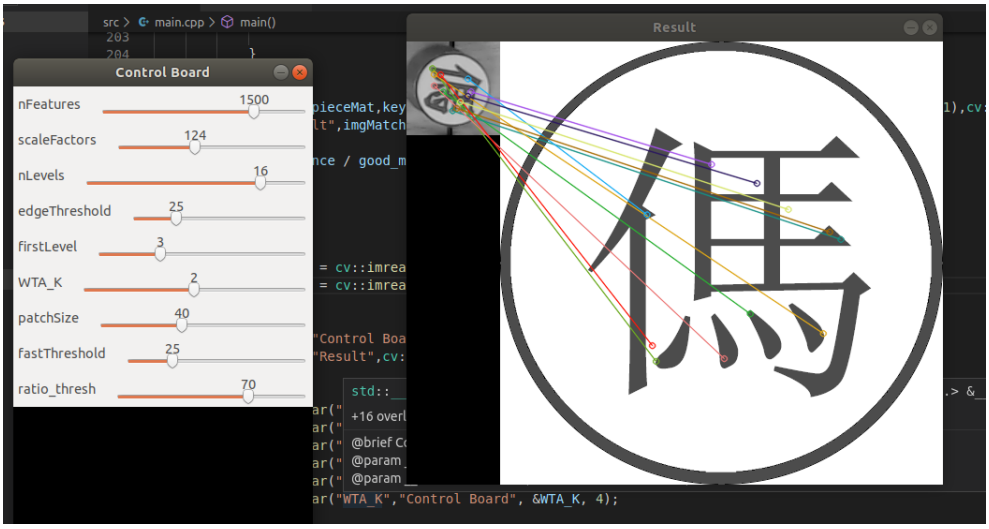


Figure 3.3: Matching the Descriptors

```
            {
                good_matches.push_back(matches[i][0]);
                distance = distance + matches[i][0].distance;
            }
        }
    }

    distance = distance / good_matches.size();
}
```

However, multiple tests have shown that the size of the `good_matches` vector is a sufficiently reliable means of comparison, so the average distance is not included in the output any longer.

3.0.5 Main Program Implementation

3.0.5.1 Camera capture and Image cropping

This function is created to grant access to the external webcam, saving the captured image as a `cv::Mat` object with sufficient clarity for accurate recognition.

```
cv::Mat importPic()
{
    return returnImg;
}
```

First, the camera is initialized by declaring a `cv::VideoCapture` object. The internal camera of the laptop is camera 0, and the external is camera 1. However, using only `camera(1)` will lead to initializing the camera with the wrong driver, which results in quite poor image quality with resolution of only 640x480px. Therefore, the camera that should be initialized is `camera(1 + cv::CAP_DSHOW)`, with `cv::CAP_DSHOW` being a constant to guarantee that we are using the DirectShow driver for our camera. Then, there should be a brief pause to ensure that the camera has been turned on properly.

```
cv::VideoCapture camera(1 + cv::CAP_DSHOW);
cv::waitKey(50);
```

Then, the properties of the image should be declared by using `camera.set()`. The Logitech HD Camera has the ability to capture image with Full HD resolution, so the parameters of the image are set to 1920x1080px.

```
camera.set(cv::CAP_PROP_FRAME_HEIGHT, 1080);
camera.set(cv::CAP_PROP_FRAME_WIDTH, 1920);
```

A `cv::Mat` object is declared to store the image captured from the camera. However, since the camera is set up at a distance from the board for stability purpose, there exists a requirement for cropping the picture so that the position calculation can be correct. The cropping is conducted by utilizing two constants called `paddLeft` and `paddTop`, which stand for left padding and top padding respectively. The function cuts from the point of (`paddTop`,`padLeft`) to the end of the image, and saves that section to another

`cv::Mat` object called `returnImg`. That object is set as the output of the whole camera capture function.

```
cv::Mat image;
camera >> image;

cv::Mat returnImg;
returnImg = image(cv::Rect(paddLeft,paddTop,
        image.cols - paddLeft - 300, image.rows - paddTop));

cv::imwrite("source.png",returnImg);
```

3.0.5.2 Main Process

The program starts by first capturing the image from the camera:

```
cv::Mat boardMat = importPic();
```

Then, four vectors are initialized. Two vectors for storing the descriptors of the database as `cv::Mat` object, and the other two for storing pieces' data as `Piece` object.

```
std::vector<cv::Mat> redDesc, greenDesc;
std::vector<Piece> redArray, greenArray;
```

After that, the database's descriptors are calculated by running a loop through every member of the database, and by pushing them back into their respective `xxDesc` vector.

```
tempMat = cv::imread(redPath,cv::IMREAD_COLOR);
redDesc.push_back(ORB_Calculate(tempMat));
tempMat = cv::imread(greenPath,cv::IMREAD_COLOR);
greenDesc.push_back(ORB_Calculate(tempMat));
```

The board image `boardMat` is processed through Hough Cirlces Transformation to create a vector of `cv::Vec3f`, which is named circles. This vector stores data about every detected circle in the `boardMat`, and is later utilized in detecting chess pieces.

```
std::vector<cv::Vec3f> circles = HoughTrans(boardMat);
```

A `for()` loop is run through every member of the `circles` vector. With the information about the coordination of the circle extracted from it, the image of a piece is cropped from the `boardMat`. Its descriptor is then calculated and is stored into the `descriptor` variable. The image is also converted into HSV color space and is filtered by both green and red filters to arrive at the numbers of green and red pixels available in the region.

```
cv::Mat piece = boardMat(cv::Rect(circles[i][0]-50,
            circles[i][1]-50,100,100));
cv::Mat descriptor = ORB_Calculate(piece);
std::pair<int,int>  value = HSVFilter(piece);
```

The value pair of two integers is the switch for determining whether the piece is a red piece or a black one. Then, the piece is compared to every descriptor of the respective database.

```
for (int count = 0; count < 7; count++)
{
    int result = BF_Match(descriptor, greenDesc[count]);
    if (bestMatch < result)
    {
        bestMatch = result;
        bestMatchNo = count;
    }
}
```

Then, the `bestMatch` is the type of the detected piece. With all these information, a piece is created and pushed into its respective vector.

```
Piece tempPiece(bestMatchNo,-1,position);
 xxArray.push_back(tempPiece);
```

As another safeguard, the name and position of the piece is written into the `boardMat` object.

```
std::string text = tempPiece.getName() +
```

```
            std::to_string(tempPiece.getBoardPos().first) + " " +
            std::to_string(tempPiece.getBoardPos().second);
    cv::putText(boardMat,text, center,cv::FONT_HERSHEY_COMPLEX_SMALL,
            1,cv::Scalar(255,255,255),2);
```

The `boardMat` object is later saved into the hard drive as a `png` image for further checking/debugging.

```
    cv::imwrite("result.png",boardMat);
```

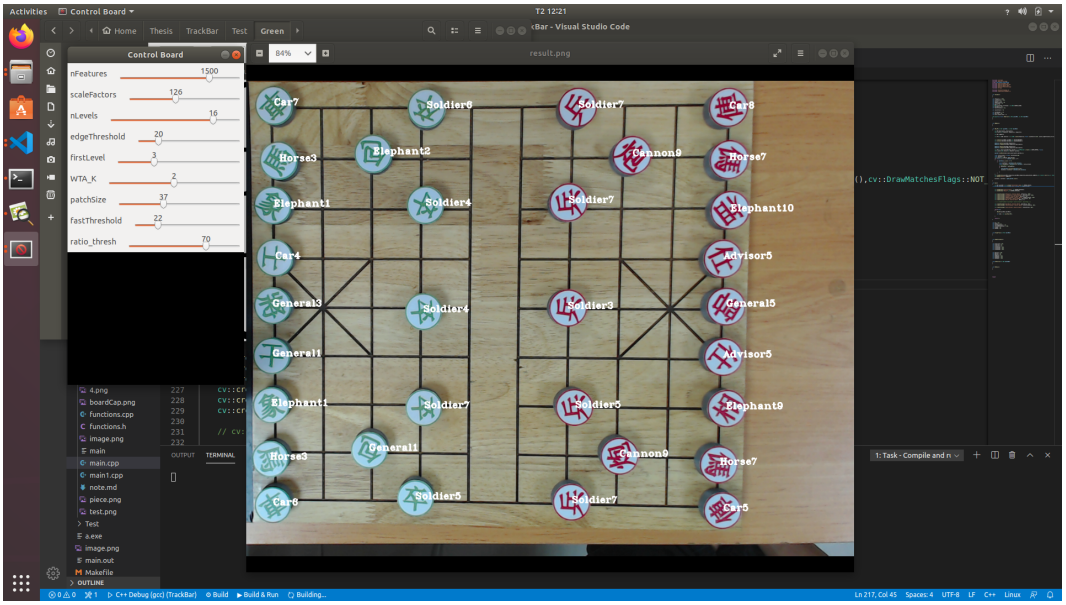


Figure 3.4: A Sample Result Image

3.0.5.3 Result formatting

For easy and reliable data transfer between many parts of the system, it is recommended that the CSV (comma-separated value) file structure be used. A standard CSV file for storing the green pieces should look like the following table:

Name	Position	Type
BlackSoldier	(7,9)	7

Name	Position	Type
BlackHorse	(8,5)	3
BlackSoldier	(8,3)	7
BlackGeneral	(9,5)	6

This is conducted by utilizing `the std::ofstream` from the standard library of C++. First, an `std::ofstream` object is created and is opened for output.

```
std::ofstream output;
output.open(fileName, std::ofstream::out | std::ofstream::trunc);
```

Then, for every piece in the array, an output stream is created by adding a comma (,) between its name, position and type.

```
for (auto piece: pieceArray)
    {
        output  << piece.getName() << ',' << (char)34 << "("
                << piece.getBoardPos().first << ','
                << piece.getBoardPos().second << ")"
                << (char)34 << ','
                << piece.getType() + 1 << '\n';
    }
```

After that, the output stream is closed, and the writing is finished.

```
output.close();
```

3.1 Chess Engine

3.1.1 Communication

3.1.1.1 Input Handling

The chess engine receives data from the recognition algorithm through data extraction from CSV files. In earlier version of the system, data used to be transfer via TCP/IP protocol. This method presents high risk such as data loss during the transfer due to unstable connection. In this version, as data written to CSV files are later received and extracted to update the chess piece list of the chess engine, this method provide solid connection between the recognition algorithm and the chess engine because data are guaranteed to be fully transferred without loss, unless the recognition algorithm fails to write to the CSV file properly.

Name	Pos
RedSoldier	(7, 1)
RedCar	(10, 9)
RedAdvisor	(10, 4)
RedCannon	(8, 2)
RedSoldier	(7, 9)
RedCar	(10, 1)
RedCannon	(8, 8)
RedAdvisor	(10, 6)
RedGeneral	(10, 5)
RedElephant	(10, 7)
RedElephant	(10, 3)
RedSoldier	(7, 5)
RedSoldier	(7, 7)
RedHorse	(10, 2)
RedSoldier	(7, 3)
RedHorse	(10, 8)

Figure 3.5: Input of Red pieces from Recognition Algorithm

As data in the received CSV file are raw and in chaotic order as compared to the chess piece list, mapping chess piece index with index from the chess piece list is necessary. Figure 3.6 represents the steps to map the data together and Figure 3.7 shows the two lists after being mapped:

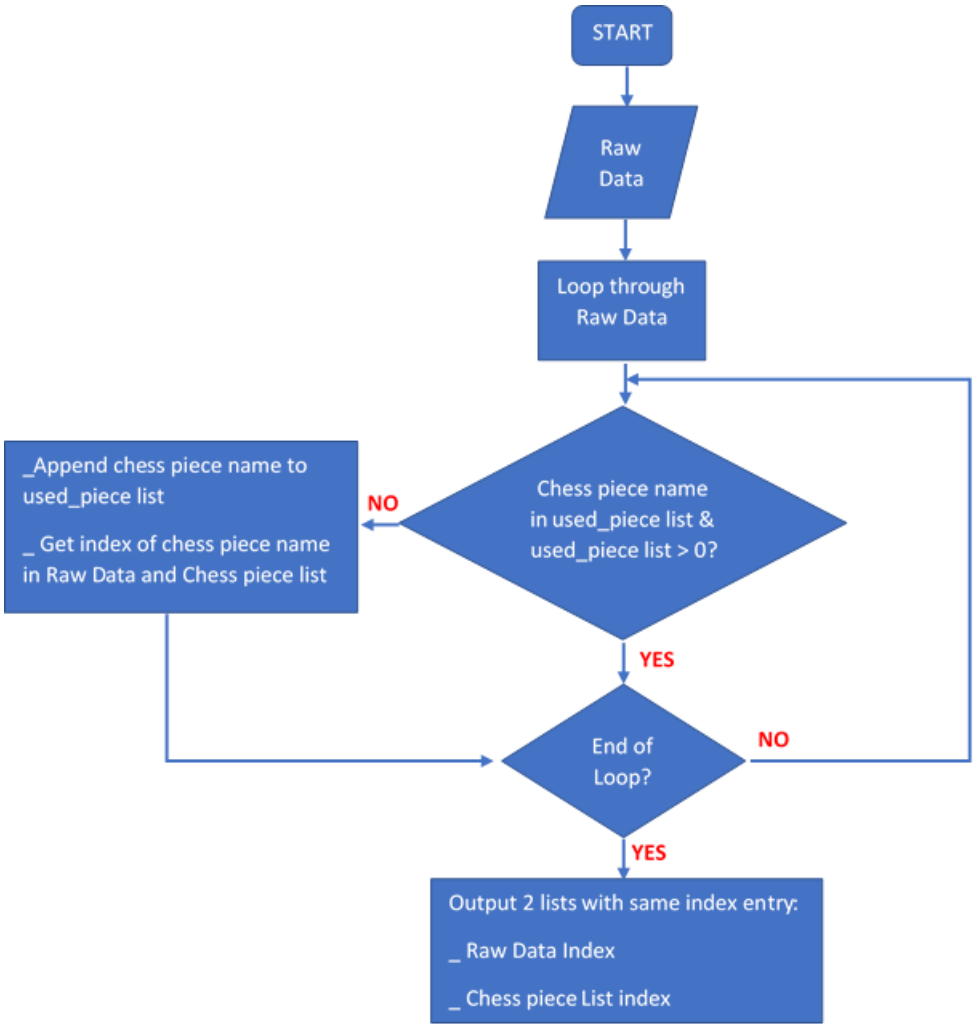


Figure 3.6: Flow chart of raw data and chess piece list index mapping

Name	Index New Data	Index Chess piece list
BlackCar	[0, 1]	[0, 1]
BlackCannon	[2, 3]	[2, 3]
BlackHorse	[4, 5]	[4, 5]
BlackSoldier	[6, 7, 8, 9, 10]	[6, 7, 8, 9, 10]
BlackElephant	[11, 12]	[11, 12]
BlackAdvisor	[13, 14]	[13, 14]
BlackGeneral	[15]	[15]
RedSoldier	[16, 20, 27, 28, 30]	[22, 23, 24, 25, 26]
RedCar	[17, 21]	[16, 17]
RedAdvisor	[18, 23]	[29, 30]
RedCannon	[19, 22]	[18, 19]
RedGeneral	[24]	[31]
RedElephant	[25, 26]	[27, 28]
RedHorse	[29, 31]	[20, 21]

Figure 3.7: Raw data and chess piece list after index mapping

After having the mapped index lists, one list is chosen to be looped through. For each iteration of the list, the raw data child index list is looped through. Each iteration of the raw data child index list has its respective position in raw data compared to that of each of the child index element of chess piece list. If the same position appears on both sides, then that index of child index list will be removed to prevent repetitive encounters in later iteration. However, when difference occurs between both sides, then the index of the child index will be checked to make sure it is the last index in the child index list. If the condition is satisfied, then the current index of child index list of raw data will be appended to an unused list, otherwise the program will ignore and continue with the looping flow. At the end of each iteration of the list, the index child list of the chess piece list will be checked. If the length of the list is greater than zero, then the position of the chess piece in the chess piece list having that index will be replaced by the position having its index stored in the unused list. The following pseudo code will further explain this complex process:

```
for i in raw_data_index:
    for j in raw_data_index[i]:
        for k in raw_data_index[i]:
            if Raw_data_list['Pos'][j] = chess_piece_list['Pos'][k]:
                remove k from chess_piece_list_index[i]
                break
            else:
                if k is the last index in chess_piece_list_index[i]:
                    unused_list.append(j)
                else:
                    Ignore and move on
    if len(chess_piece_list_index[i]) > 0:
        Replace: chess_piece_list['Pos'][chess_piece_list_index[i][0]]
        with:    Raw_data_list['Pos'][unused_list]
        Reset unused_list
```

3.1.1.2 Chess Engine-SCARA Arm Communication

The chess engine sends its calculated result to the SCARA arm via Serial communication. After outputting a result, a tuple which consist of current position, position calculate from the chess engine and the chess piece capture condition will be sent to the Arm Control class, which handles inputs from the chess engine and then pass the corresponding G codes to the SCARA arm. There are two main methods in Arm Control: ReadButton() and RunSCARA(). Furthermore, there are five main operations: MoveChess(), MoveServo(), CheckDoneStatus(), ReturnHome() and Magnet().

- MoveChess() function accepts position as parameter and pass them into X and Y arguments of G code G0 X< $Xvalue$ >Y< $Yvalue$ >. This code will move the SCARA arm to the exact location based on X and Y.
- MoveServo() function handles the tool on SCARA arm and decide whether it goes up or down using code "G5".
- CheckDoneStatus() function creates a while loop and listens to "Done" string sent from the SCARA arm. When the precise string is received, it will end the function, hence this function works as a delay anchor for SCARA operation to finish before receiving any new instructions from the chess engine.
- ReturnHome() function moves the SCARA arm to its original position using code "G28". This function is always called after each time SCARA finishes its instruction from the chess engine.
- Magnet() function handles the active status of the solenoid on the tool using code "G6". If the status is ON, the solenoid will emit magnetic force to pull the chess piece to it and release it vice versa.

ReadButton() function shares the same method implementation as CheckDoneStatus() but it is only called after the SCARA arm finishes the instructions sent from the chess engine. After the player finishes his turn, a button will be pressed and ReadButton() delayed the program waiting for the "Done playing" signal sent from the SCARA arm. On the other hand, RunSCARA accepts capture status as parameter and gives out a sequence of operations based on that parameter. Figure 3.8 represents the operation sequence of RunData():

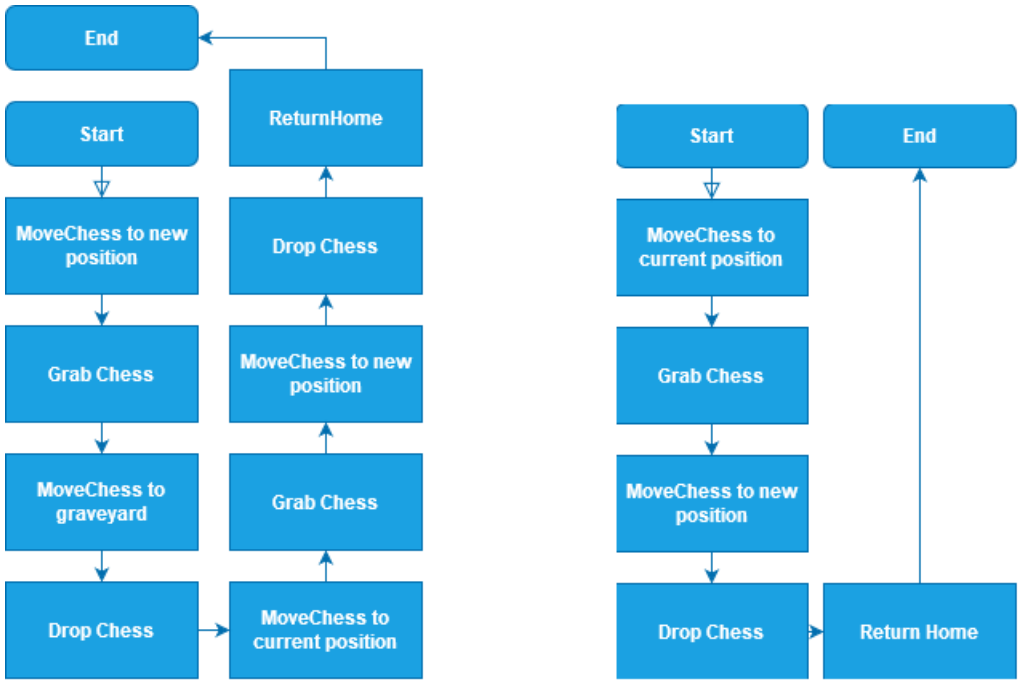


Figure 3.8: RunData() sequences (Left: capture-True, Right: capture-False)

3.1.2 Alpha-Beta Pruning Implementation

3.1.2.1 Recursive Function

Before any deeper research into the algorithm, the definition of recursive function needs to be investigated. A recursive function is a function that calls itself during execution. Therefore, in operation a recursive function is called several times, outputting result at the end of each iteration. With recursive function, a program can be written efficiently with minimal amount of code. However, debugging recursive functions can be a challenge as they can cause infinite loops and other unexpected results if not written properly.

3.1.2.2 Algorithm Flow

The algorithm, which from the current moment to the end of the research will be addressed as Alpha-Beta algorithm and Alpha-Beta function, is a recursive function that accepts input consisting of four main parts: Board State object, Search depth, Boolean variable isMax and the algorithm pruning variables: Alpha and Beta. At the first time calling Alpha-Beta function, the current Board State was chosen and the initial search depth will be used. As in section 3.2.2.7, when the Board Score is negative, it means that Black Player is having an advantage, so if the chess engine is chosen as a Black Player, the final result should be as low as possible, therefore variable isMax is initially chosen as False. Finally, as it is the first time Alpha-Beta function was called, Alpha will be -9999 and Beta will be 9999, the smallest score and largest score respectively.

At the beginning of the algorithm, the Board State object will be used to detect End Game condition. If the condition is satisfied, Board Score will be a multiplication of the current Player variable with infinity. To simplify the numbers, an extremely large number of 99999999 was chosen to represent infinity. The Board Score is then returned to higher depth due to the recursive characteristic.

If the first condition was not fulfilled, search depth is put into consideration because when search depth reaches depth zero, Board Score will be collected as mentioned in section 3.2.2.7, then returned to higher depth. On the other hand, if search depth is higher than 0, then a list of legal chess moves will be generated, which will be further discussed in section 3.2.2.4. For each element of the move list, a new Board State will be created. Then the new Board State along with depth - 1, the opposite of the isMax Boolean, the current Alpha and Beta will be inputs and the Alpha-Beta function will continue to be recalled until search depth reach depth 0 or End Game condition is fulfilled.

Whenever the Alpha-Beta function returns to higher depth, isMax Boolean is brought into consideration. If it is currently True, then the new Alpha will be the maximum value between the current Alpha and the returned Board Score or the new Beta will be the minimum value between the current Beta and the returned Board Score if current isMax is False. The updated Alpha or Beta will then be compared to each other. If Beta is equal or less than Alpha and search depth is at its highest depth, then the result consisting of the Piece and its new position will be returned and end the algorithm, whereas the Board Score will be returned to higher depth if otherwise. If Beta is still larger than Alpha, the Board Score will be appended to a list of Board Score. At this step isMax Boolean is considered once more. If isMax is True, then from the list of Board Score the index of the maximum value is returned with the same method as when comparing Beta with Alpha or the index of minimum value is returned if isMax is False.

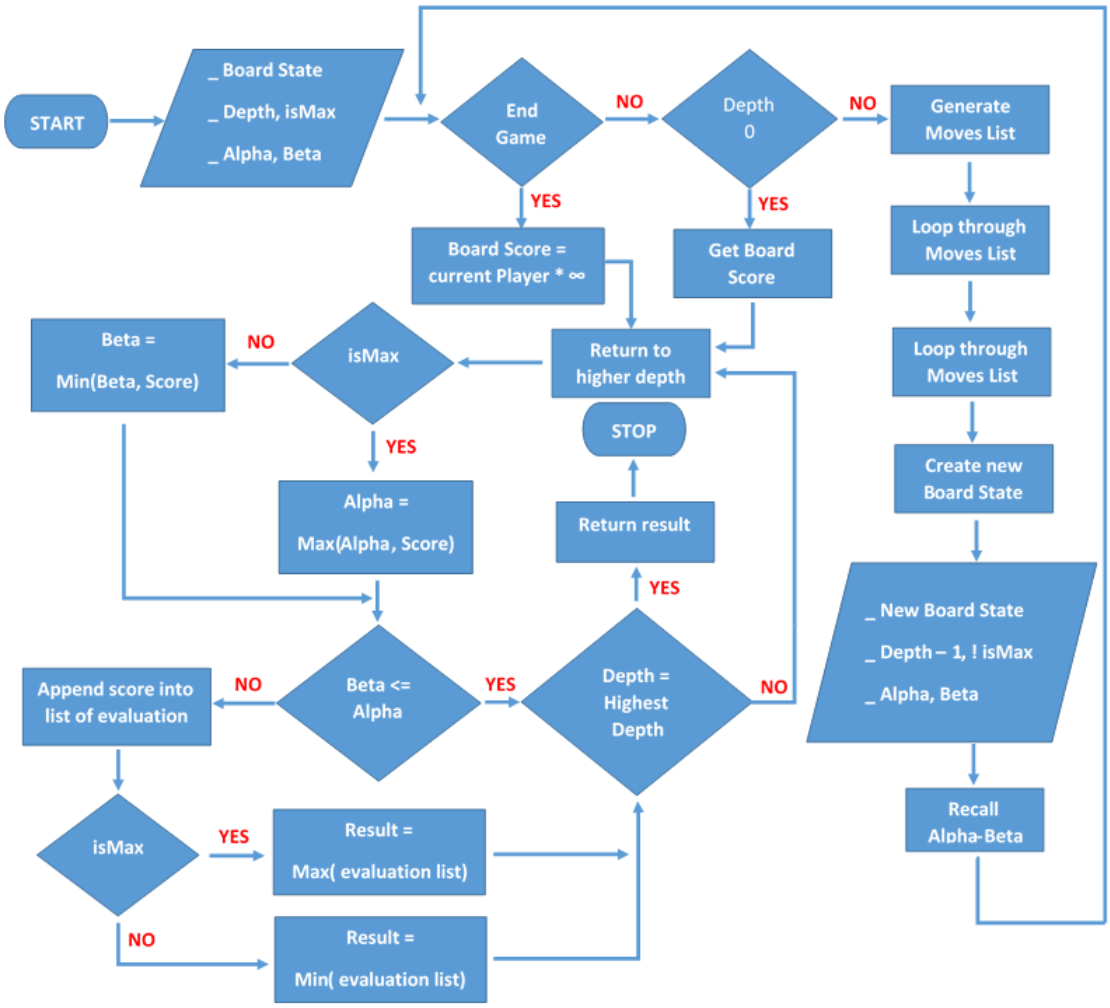


Figure 3.9: Alpha Beta Flow Chart

3.1.2.3 Data Structure

In this program, the data structure is built around dictionaries and lists. A Python dictionary is an unordered, changeable and indexed collection of keys and values written between curly brackets. Values inside a dictionary can be accessed with the respective key using statement: $yourdict[' < key > ']$. In this program, the chess piece list is a list of dictionaries, each dictionary represent a chess piece with its keys being the characteristics of the piece such as: Name, Symbol, Team, Piece Value and Current Position.

A CSV is a comma-separated values file, which allows data to be saved in a tabular format. CSV files store data in rows and the values in each row is separated with a separator, also known as a delimiter. Since CSV files are plain text, they're easier to import into a spreadsheet or another storage database, regardless of the used specific software. The CSV files are used to store the original chess piece list, to convert updates from the recognition algorithm to the chess engine and the performance time of the Alpha-Beta algorithm. Figure 3.10 represent the creation of the chess piece list.

```
def resetBoardState(self):
    # Load information into
    data_excel = pd.read_csv("src\\BoardData.csv")
    # print(data_excel)
    self.pieceList = []
    for i in range(len(data_excel)):
        # print(ast.literal_eval(data_excel["Pos"][i]))
        self.pieceList.append({
            'Name': data_excel["Name"][i],
            'Pos': ast.literal_eval(data_excel["Pos"][i]),
            'Symbol': data_excel["Symbol"][i],
            'Team': data_excel["Team"][i],
            'Score': data_excel["Score"][i]
        })
    return self.pieceList
```

Figure 3.10: Chess piece list implementation

3.1.2.4 Move Generator

To continue the algorithm flow after End Game and Depth conditions being not fulfilled, all the possible moves for the current player chess pieces must be collected. Therefore, output for this function will be a list called possibleMoves containing tuples of chess piece item and its legal position on the chess board. For each chess piece there are specific rules which are stored in class Rules and the possible moves were collected by calling "possibleMoveFor$< PieceType >$(current chess piece item, chess piece list)" function.

Since the first part of the name of these functions are the same, by looping through the current player chess piece list, all possible move functions can be called respectively as arguments of $eval()$ function base on the type of the chess piece which is split from its name using regular expression $(? : Black|Red)(.*)$. The " ?: " expression standing before "$Black|Red$" means that the string containing Black or Red will be grouped together but not captured. On the other hand, expression $(.*)$ means that "." will match any character, " $*$ " will match from 0 or more occurrences and all of it will be captured. Therefore, if the complete name of the chess piece is "BlackCannon", then after using the regular expression we will receive string "Cannon" and function $possibleMoveForCannon(BlackCannon, pieceList)$ will be called.

After the possible moves were collected, they will be added to possibleMoves list along with its chess piece item as a tuple. Figure 3.11 represents the full implementation of the function:

```
for i in range(len(pieceList)):
    movelist = eval('self.gamerule.possibleMoveFor{}(pieceList[i],self.pieceList)'.
                    format(re.findall('(?:Black|Red)(.*)', pieceList[i]['Name'])[0]))
    # movelist = self.gamerule.possibleMoveForGeneral(pieceList[i], self.pieceList)
    for item in movelist:
        possibleMoves.append((pieceList[i], item))
return possibleMoves
```

Figure 3.11: Get all possible moves function

3.1.2.5 Chess Piece And Position Value

Alpha-Beta function cannot operate normally without a scoring mechanism. The evaluation of the result is based on the sum of chess pieces value and their positions value. Chess piece value is an integer that determine the importance of the chess pieces. For example the General piece is the most important piece in the game, hence it is given a value number of 160000. This value is stored in the csv file along with other characteristics of the chess piece and can be modified with ease. Below is the list of chess pieces value of Black player and the same value will be applied to the Red player respectively:

Name	Symbol	Pos	Team	Score
BlackGeneral	BG1	(1, 5)	-1	160000
BlackCar	BC1	(1, 1)	-1	600
BlackCar	BC2	(1, 9)	-1	600
BlackCannon	BO1	(3, 2)	-1	285
BlackCannon	BO2	(3, 8)	-1	285
BlackHorse	BH1	(1, 2)	-1	270
BlackHorse	BH2	(1, 8)	-1	270
BlackSoldier	BS1	(4, 1)	-1	30
BlackSoldier	BS2	(4, 3)	-1	30
BlackSoldier	BS3	(4, 5)	-1	30
BlackSoldier	BS4	(4, 7)	-1	30
BlackSoldier	BS5	(4, 9)	-1	30
BlackElephant	BE1	(1, 3)	-1	120
BlackElephant	BE2	(1, 7)	-1	120
BlackAdvisor	BA1	(1, 4)	-1	120
BlackAdvisor	BA2	(1, 6)	-1	120

Figure 3.12: Chess Piece Value of Black Player

On the other hand, the positions value of the chess piece represents the importance of that piece being proportional to the distance of the opponent, hence the more important the other player chess piece is, the higher the position value of this chess piece has. Negative position values are also added to the list to represent bad move of the chess piece. This value is determined by heuristic rules and referenced in other popular Chinese chess AI program[10]. For example, the Black Car will have position value higher when that position is closer to the Red General and reaches its peak of 26 when standing in front of it. However it has the lowest score of -2 when it stays at its initial position. Since these position value is scaled with the chance of capturing the opponent chess pieces, the list of position value is built only for aggressive chess pieces such as: Cars, Cannons, Horses and Soldiers. For two player the position value list will have similar traits but in opposite direction due to its chess pieces are on opposite site. Below is the position value list of the Black Car and Red Car piece:

```
self.BlackCar = [
    [-2, 10, 6, 14, 12, 14, 6, 10, -2],
    [8, 4, 8, 16, 8, 16, 8, 4, 8],
    [4, 8, 6, 14, 12, 14, 6, 8, 4],
    [6, 10, 8, 14, 14, 14, 8, 10, 6],
    [12, 16, 14, 20, 20, 20, 14, 16, 12],
    [12, 14, 12, 18, 18, 18, 12, 14, 12],
    [12, 18, 16, 22, 22, 22, 16, 18, 12],
    [12, 12, 12, 18, 18, 18, 12, 12, 12],
    [16, 20, 18, 24, 26, 24, 18, 20, 16],
    [14, 14, 12, 18, 16, 18, 12, 14, 14]]
```

```
self.RedCar = [
    [14, 14, 12, 18, 16, 18, 12, 14, 14],
    [16, 20, 18, 24, 26, 24, 18, 20, 16],
    [12, 12, 12, 18, 18, 18, 12, 12, 12],
    [12, 18, 16, 22, 22, 22, 16, 18, 12],
    [12, 14, 12, 18, 18, 18, 12, 14, 12],
    [12, 16, 14, 20, 20, 20, 14, 16, 12],
    [6, 10, 8, 14, 14, 14, 8, 10, 6],
    [4, 8, 6, 14, 12, 14, 6, 8, 4],
    [8, 4, 8, 16, 8, 16, 8, 4, 8],
    [-2, 10, 6, 14, 12, 14, 6, 10, -2]]
```

Figure 3.13: Chess Piece Value of Black Car and Red Car

3.1.2.6 EndGame Move

"EndGame" move is a terminology for a move that captures the opponent's General which will end the game. When encountering an EndGame move, the Board Score is not taken normally but it becomes a product of 99999999 which represents infinity and the current player integer which is -1 for Black player and 1 for Red player. This creates the highest impact on the game as if Alpha or Beta is replaced by this value, all the branches beyond will be pruned as condition $Alpha \geq Beta$ will be satisfied, which will greatly reduced performance time.

```
if self.EndGame() == True:
    result = {
        "Score": '',
        "Piece": '',
        "Pos": ''
    }
    pass

    result['Score'] = (99999999) * self.player.getCurrentPlayer()

    return result
```

Figure 3.14: EndGame Condition Check

The EndGame move function will accept Boolean input of isGeneralExist() function returns output of True or False to alert the chess engine to stop the game. isGeneralExist() accepts the opponent chess piece list as input and the last element or first element will be checked whether there exist a General piece as mentioned in Figure 3.15. The General piece will be put at the end or at the top of the chess piece list to avoid unnecessary search time by using for loop over the list.

```
def isGeneralExist(self):

    opp_pieceList = self.getOpPlayerPieceList()
    if (opp_pieceList[0]['Name'] == "RedGeneral") or (opp_pieceList[0]['Name'] == "BlackGeneral"):
        # print("Still exist")
        return True
    else:
        # print("No general")
        return False
```

Figure 3.15: isGeneralExist() Function implementation

3.1.2.7 Board State and Board Score

Board State represents the object of class State, which can also be stated as a bridge object connecting all other classes that form this chess engine. The Board State controls all operation regarding changing position or removing chess pieces in the Player class piece list, detecting if the game has ended or not, transferring the piece list information to user interface class to draw the board and finally it contains the Minimax algorithm with Alpha Beta pruning.

Board score is created based on the definition of chess piece value and chess piece position value. When the board score is created, a "for loop" will be conducted to run through all the element in the piece list. Whenever a black piece element is encountered, its piece value is combined with its position value and then the total is added to BlackScore respectively for all black pieces. For red pieces, the process is similar, leading to the Board score being the difference between RedScore and BlackScore as presented in Figure 3.16. In the algorithm Board Score can only be collected when search depth is zero, hence most of the Board Score comes from the lowest depth, except for EndGame move.

```
            BlackScore += (pieceList[j]['Score'] + self.eval.posValue(pieceList[j]['Name'], pieceList[j]['Pos']))

            RedScore += (pieceList[j]['Score'] + self.eval.posValue(pieceList[j]['Name'], pieceList[j]['Pos']))

    return RedScore - BlackScore
```

Figure 3.16: Board Score Implementation

The Board score has high impact on the flow of the game as it determines whether the move is cost worthy or not. When combining with Alpha and Beta, board score can also decide whether the branches beyond current branch should be pruned or not, hence reduced performance time for the algorithm. In evaluating operation, when the current node is Max, then the maximum value between Alpha and the board score will be the new Alpha and vice versa for Beta and node Min as stated in fig 3.17.

```
# If the Node is Max => compare score with Alpha
if isMax:
    alpha = max(alpha, score)
# If Node is Min => compare score with Beta
else:
    beta = min(beta, score)
# Check if alpha is larger than beta. If so, prunes the later branches
if beta <= alpha:
    print("Prun")
    return eval_result[-1]
```

Figure 3.17: Board Score evaluation with Alpha and Beta

If in the current node the board score is not better than Alpha or Beta, then it will be appended into a list. If searching is done for the current node and no branch is pruned, then the maximum or minimum score will be chosen from the list based on the status of the node being Max or Min. If more than one best value are found, then the comparing algorithm will choose the first board score that it encounters as presented in fig 3.18.

```
# Append board score into a list of score
for j in range(len(eval_result)):
    score_list.append(eval_result[j]['Score'])
# If the Node is Max => Get index of max value from list
if isMax:
    index = score_list.index(max(score_list))
# If the Node is Min => Get index of min value from list
else:
    index = score_list.index(min(score_list))

return eval_result[index]
```

Figure 3.18: Getting Maximum or Minimum board score from score list

3.1.2.8 Cloning Board State

After obtaining the legal move list of all the chess pieces, each move will represent an exclusive board state with respective board score. However, with only the main chess piece list, creating each board state with the exclusive move representing it will cause major error to the piece list as it alter the main position of each chess piece in the list, hence the cloning the Board State method is introduced. To clone an object, two approach can be taken into consideration: Shallow Copy and Deep Copy.

To shallow copy an object is to simply create a new one with the same construction and populate it with references to the child objects in the original.It means that a shallow copy of a collection is a copy of the collection structure, not the elements and the two collections will share the individual elements. Hence whenever a change happen to the copied object, the main object is also affected, which does not solve the main problem of not changing the chess piece list.

On the other hand, deep copy is a recursive copying process. When an object is deep copied, a new collection object will firstly be constructed, then copies of the child objects in the original are recursively being populated. In case of deep copy, a copy of object is copied in other object. It means that a deep copy of a collection is two collections with all of the elements in the original collection duplicated. Hence, any changes made to a copy of object do not reflect in the original object, which satisfies the cloning board state condition. In Figure 3.19, after the board state was copied, the chess piece position in chess piece list is changed using movePiece() function to find the score in lower depth without affecting the original chess piece list. After the result was calculated, all the cloned board state will automatically be deleted.

```
def nextState(self, piece, pos):
    # Copy current pieceList to new state obj
    nextState = copy.deepcopy(self)

    nextState.movePiece(nextState.player.getCurrentPieceList(), piece, pos)

    return nextState
```

Figure 3.19: Cloning board state implementation

3.1.2.9 Simulation Procedure

To experiment the efficiency of the chess engine a simulation program was conducted on two different CPU with two players being two chess engines:

- CPU1: $Intel(R)Core(TM)i7-7700CPU@3.60GHz, 4Cores, 8LogicalProcessor$
- CPU2: $Intel(R)Core(TM)i3-4030UCPU@1.90GHz, 2Cores, 4LogicalProcessor$

The simulation program was written in Python as a simple user interface for debugging purposes. The chess board was built with the following rules:

- Chess pieces of Red player will have symbol starting with letter "R" and "B" vice versa for Black player.
- "C", "O", "H", "E", "A", "G", "S" are types symbols for Cars, Cannons, Horses, Elephants, Advisors, Generals and Soldiers respectively.
- A number representing the order for each chess piece of same type is used. In example: "BO1" means the first Black Cannon.
- Symbol "$\sim\sim$" will represent the river in the chess board.

To draw the board two "for" loops will be called, the outer "for" goes through ten rows and the inner goes through nine columns. Whenever the row and column match the position of a chess piece predetermined in the chess piece list, the specific symbol of the chess piece will be put in that position, otherwise the symbol "*" will be used to indicate blank position. Figure 3.20 is the chess board representation used for the algorithm:

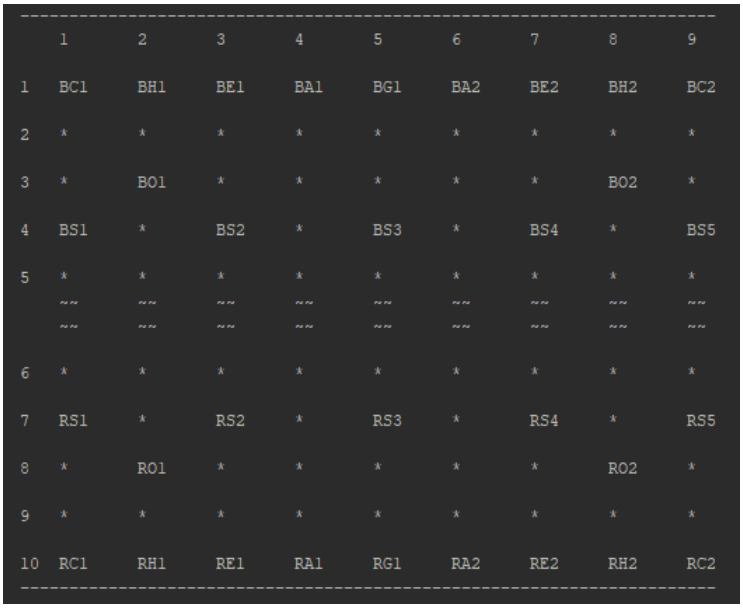


Figure 3.20: Simulation graphic user interface

Choosing the appropriate search depth and the order of the elements in the possibleMoves list for the engine is crucial as it will affect not only the precision but also the performance time of the algorithm. For this simulation, chess engine for Red player remains depth two when chess engine for Black player has search depth from range two to four respectively. On the other hand, due to the nature of the data structure, sorting chess piece and alternating their orders can be done with ease, hence during this simulation, two chess pieces orders were used:

- Order 1: Car - Cannon - Horse - Soldier - Elephant - Advisor - General
- Order 2: General - Car - Cannon - Horse - Soldier - Elephant - Advisor

Each time Alpha-Beta function is called, to capture the performance time, function time() of library time is used at the start and end of the Alpha-Beta function and the performance time will be the subtraction between the start and end variables. Then it will be stored along with a counting variable as a tuple into a list which in the end when the game stopped, the list will be converted to csv file for ease of data collecting.

3.2 Scara Arm Design and Control

3.2.1 Hardware Design

This section discusses the Scara arm design, electrical components which are used to control the arm and design of chess board, chess piece to work with both the pick and place tool and vision part.

3.2.1.1 Overall View

For pick and place the chess pieces purpose, 2 degrees of freedom (DOF) Scara arm model is printed in plastic and built. The model has two advantageous. It provides a large work space which cover all the position on the chess board and does not occupy space on the chess board after functioning, which avoids disturbing the chess pieces recognition process

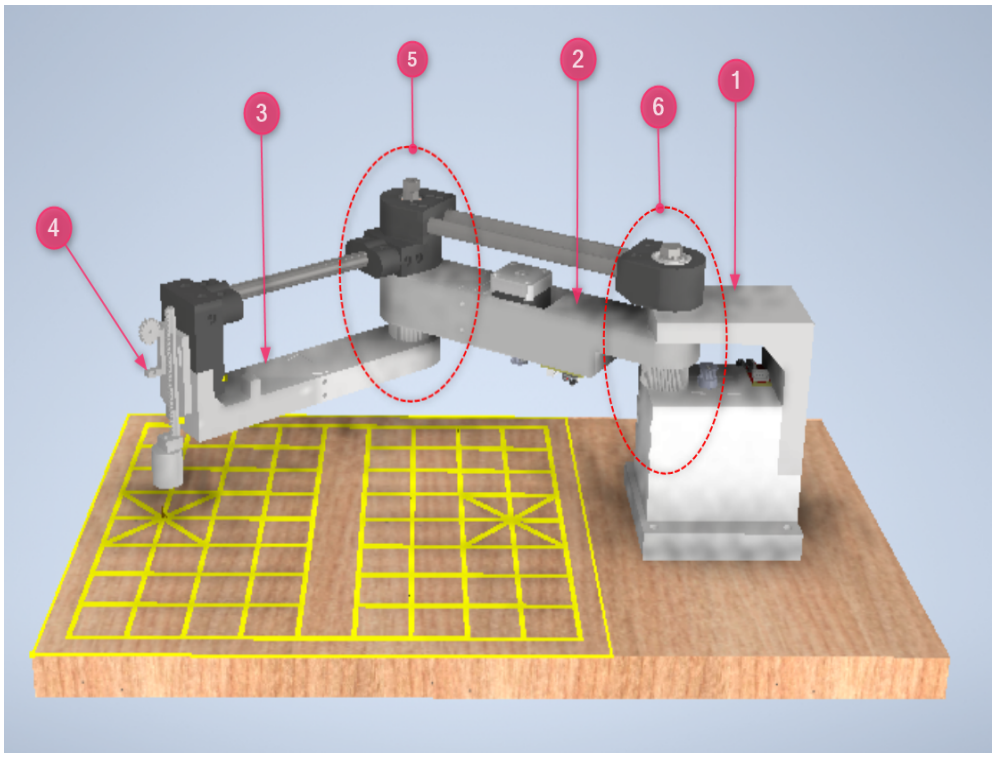


Figure 3.21: Scara Arm Design

The arm model is separated into the following main parts and numbered in figure 3.21:

- Base (1)
- First arm (2)
- Second arm (3)
- Pick and place tool (4)
- Rotation joints (5) (6)

3.2.1.2 Rotation joint

The rotation joint is designed to connect arm segments to each other or to the robot base so that each parts at the joint can rotate freely without affecting position of the other parts. The rotation joint consists of three main parts, which is denoted in figure 3.23 as follows:

- Ball bearing (A)
- Support ring (C)
- Arm support part (R)
- arm segment part (D , E)

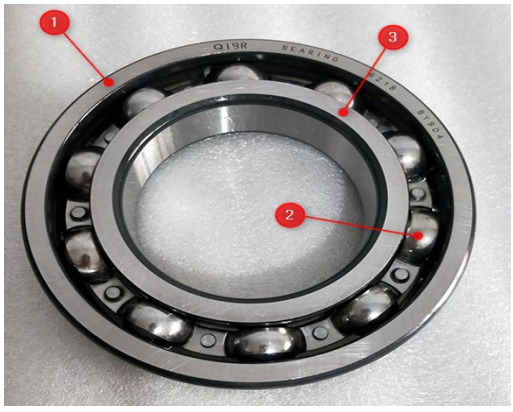


Figure 3.22: Ball bearing

Ball bearing is a type of rolling-element bearing that uses balls to maintain the separation between the bearing races.The function of a ball bearing is to connect two machine part that rotate relative to one another in such a manner that the frictional resistance to motion is reduced.

Ball bearing contains three main parts: the outer race (1),the inner race (2) and the balls (2) between the two races as numbered in figure 3.22

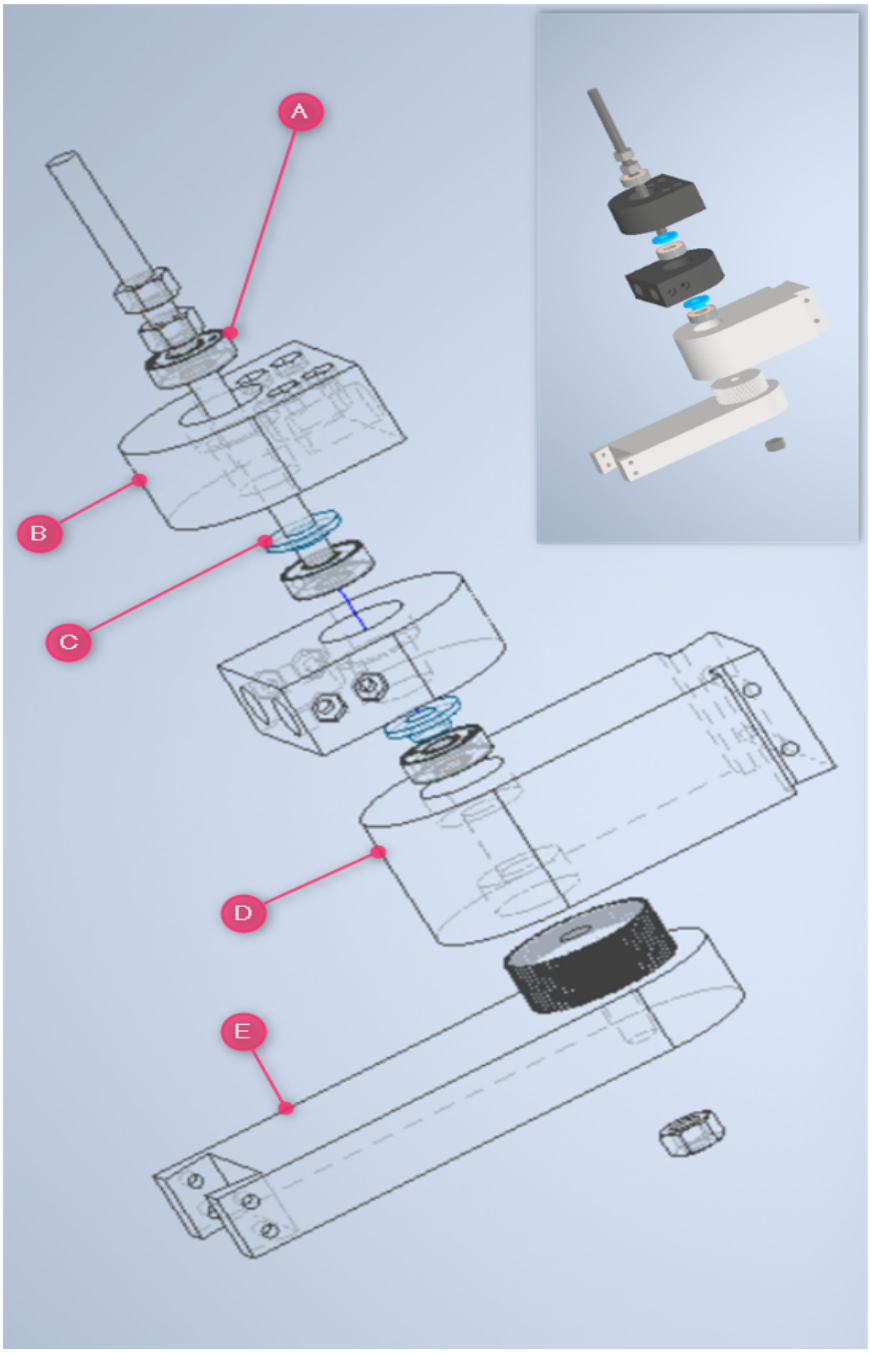


Figure 3.23: Rotation joint Arm-1 and Arm-2

All the components are connected by nuts,bolts and a threaded rod from top to bottom.Arm support parts is used for balancing the arm due to its long range from the robot base, 25cm for each arm segment.

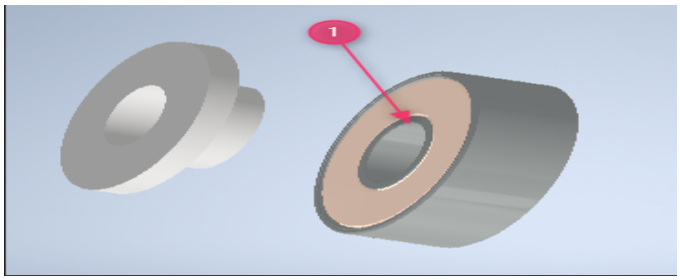


Figure 3.24: Ring and Bearing

The support rings are placed between surface of each parts and the bearings. One of the support ring face contacts to the inner race of the ball bearing, which is numbered as 1 in 3.24 and the other face contact to the support parts surface. This structure help the parts rotate freely without asserting large friction forces between their surfaces.

3.2.1.3 Arm Segment and Support part

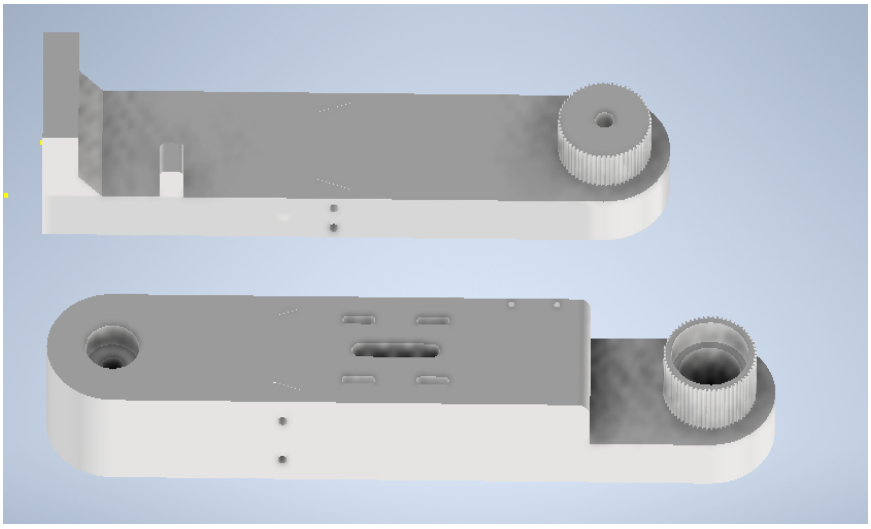


Figure 3.25: Rotation joint Arm1 and Base

The two robot arm parts are 250mm length each,due to the limitation of 3d printer in printing size, the arms are separated into two segments and connected by connector and screws

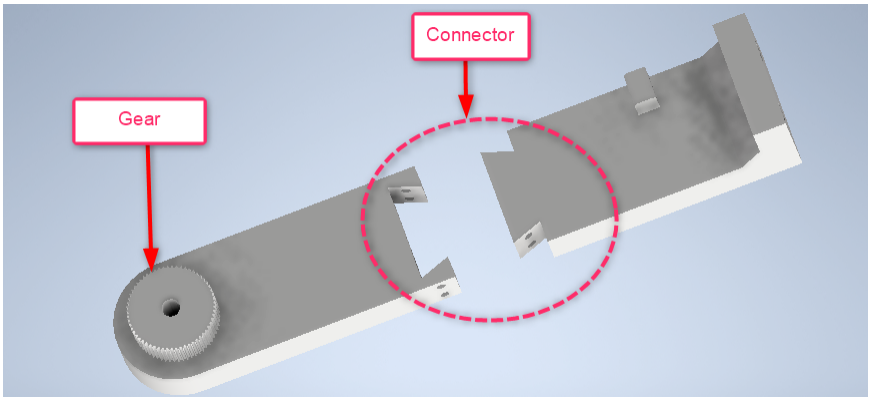


Figure 3.26: Rotation joint Arm1 and Base

Figure 3.27: Pulley Motor

Each robot arm has a 2mm pitch - 62 teeth gear. The gear is used for connecting the arm to the motor by GT2 closed loop timing belts and the GT2 pulley- 16 teeth on the motors as in figure 3.29.The GT2 pulley is fixed to the motor shaft as depicted in figure 3.27.The Timing belt is a rubber belt which is used for synchronizing the rotation motion of the rotor and the

rotation joint.The belt,the timing belt and the pulley must have the same circular pitch which is 2mm in this design in order to transfer the rotation movement.Circular pitch is the distance from one point on the gear tooth to the corresponding point on an adjacent gear tooth.

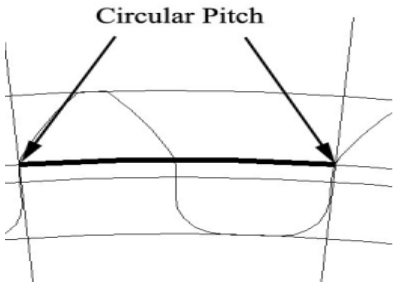


Figure 3.28: Circular Pitch

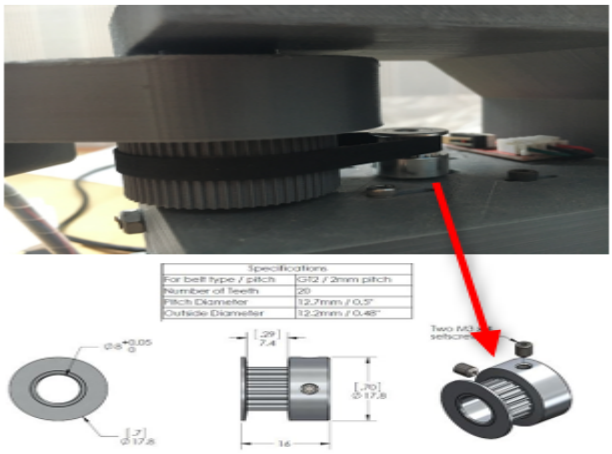


Figure 3.29: Motor Arm Connection

The length of the timing belt is calculated with the below formula[26] :

$$(DL + DS) * \frac{\pi}{2} + 2 * L + \frac{(DL - DS)^2}{4 * L} \tag{24}$$

DL : diameter of the gear
DS : diameter of GT2 pulley

L : distance between the center of gear and pulley

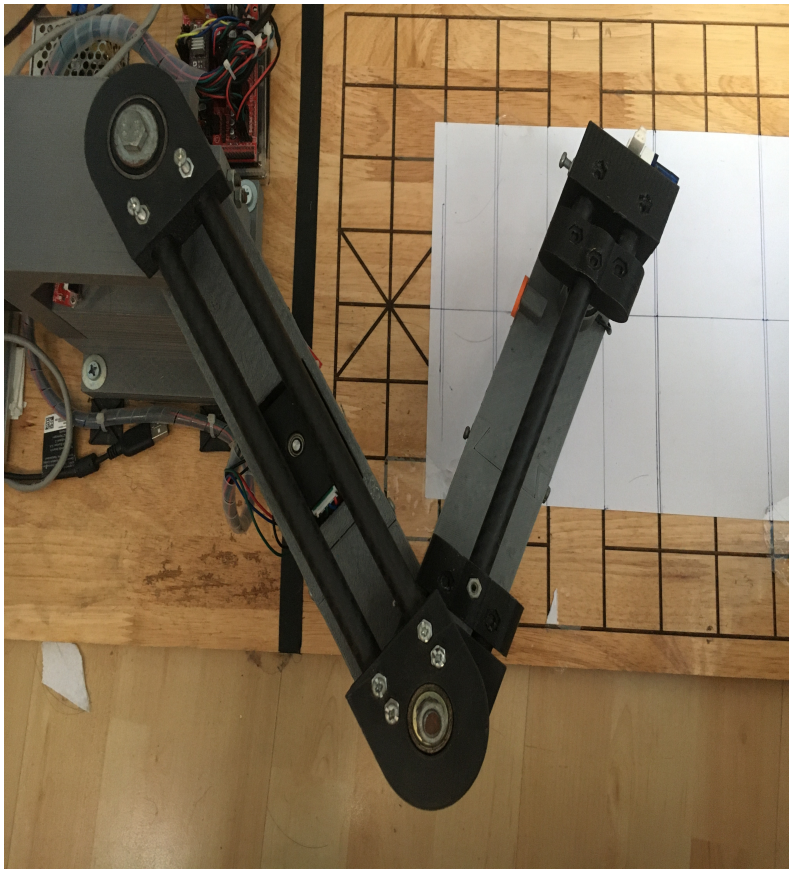


Figure 3.30: Support Part

Due to the length of each arm segment, supported parts is added to each rotation joint and at the end of the second arm segment.They are connected by carbon tubs in order to level up the both two arm segments.Carbon tub is chosen because of its lightweight and hard characteristic.These are fixed to the support part by screws as show in figure 3.30

3.2.1.4 Motor and End-stop switch placement

The SCARA robot uses two motors to control two arm segments independently.One motor is placed at the robot base and the other,which uses for

control the second arm segment is placed on the first arm segment. Beside using motor to control robot movement,end-stop switches are also used to detect the homing position of the robot.When the arm segment touch the switch,its movement will be stopped.Similar to the placement of the motor,end - stop for the second arm segment is placed at the bottom of the first arm segment and the other one is at the base.

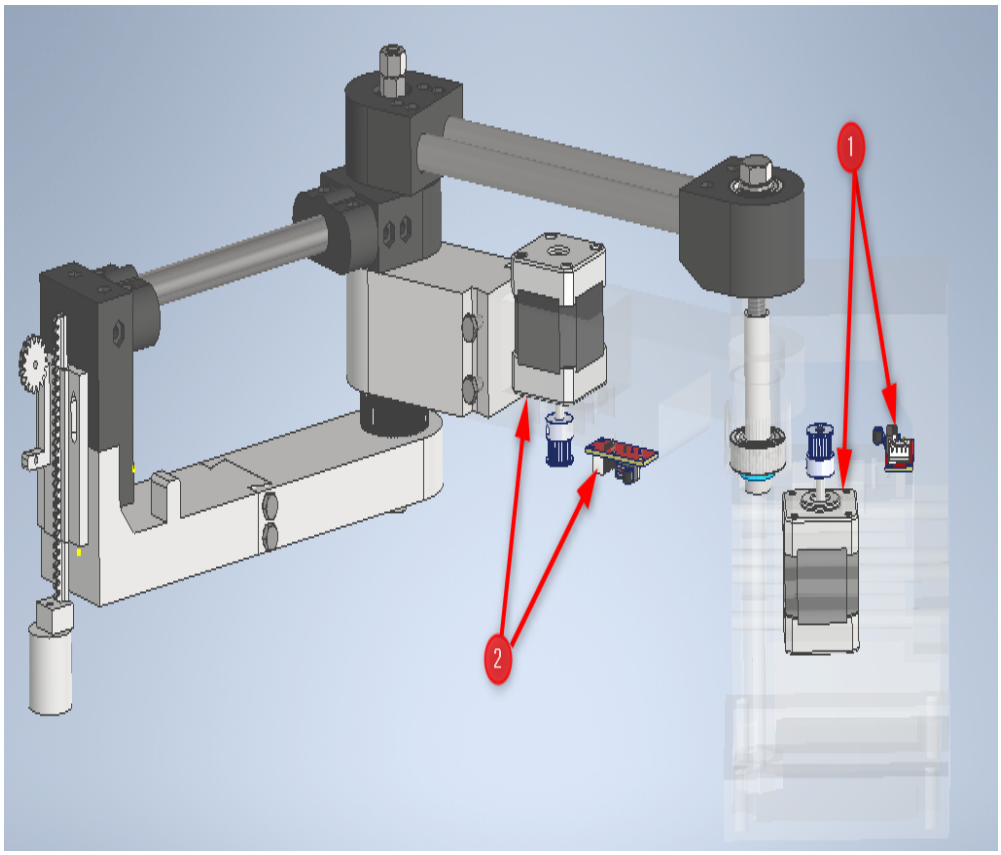


Figure 3.31: Motor-End stop Placement

Figure 3.31 shows the position of the motor and end stop switch for each arm segment with number 1 is for first arm and the other for the second arm.

Because the motors and the arm segments is connected through the rubber timing belt,which is discussed above,hence to make sure the belt is tighted a small gap is placed at the motor position.The motor gap at the bottom of the first arm segment is illustrated in figure 3.33,and there is a

similar gap on the robot base

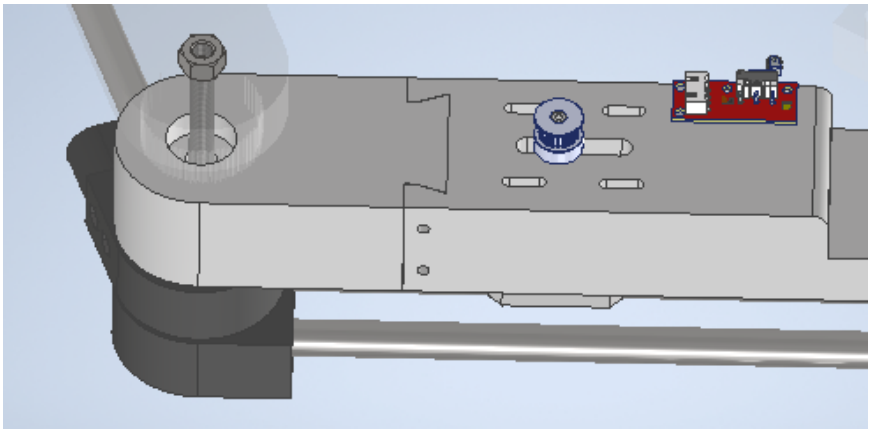


Figure 3.32: Pick and Place tool

3.2.1.5 Pick and Place tool

A pick and place tool is attached at the end of the second arm segment.The tool consists of a servo motor, a electrical magnetic, a gear and gear slider.

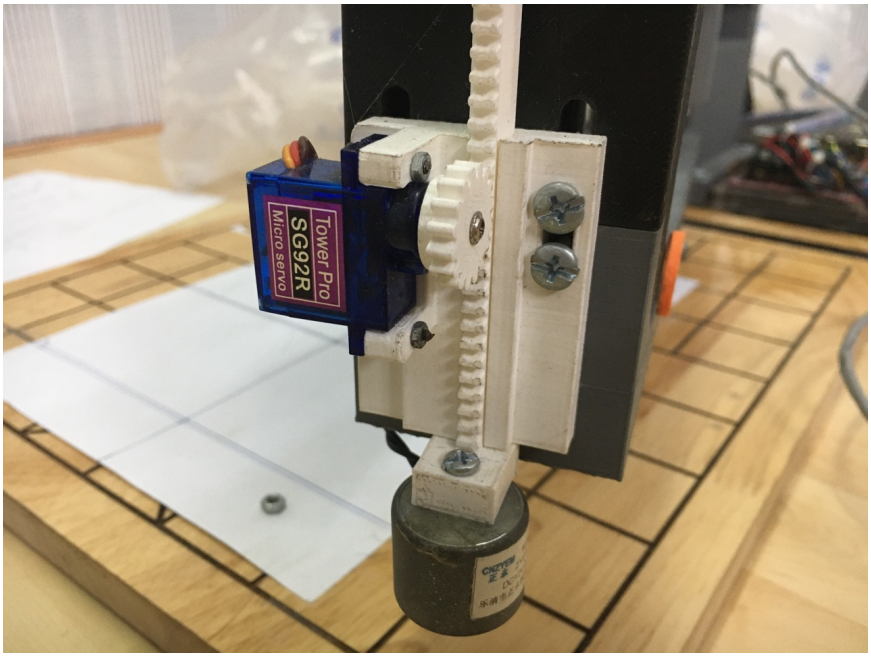


Figure 3.33: Motor placement gap

The chess piece is attached an iron ring beneath the icon layer for pick and place purpose.The electrical magnetic will be activated to pick the chess piece and turn off to release it.The servo motor will move the electrical magnetic up and down in the pick and place process for avoiding collision to other chess pieces

The SCARA arm is placed between the chess board for facilitating the pick and place process .The Cartersian coordinate of the chess board is coincident to the Cartersian coordinate of the SCARA arm,the original point(0,0) is under the center of the first arm joint.

3.2.2 Electrical Component

This section discusses about the selection of electrical component for the SCARA robot.

The Arduino Atmega 2560 and Ramps 1.4 shield are chosen for controlling the SCARA arm robot. The Arduino board is responsible for processing the input data from the computer and the Ramps 1.4 shield is responsible for providing power to the stepper motors and other electrical components.

Figure 3.34: Arduino Board

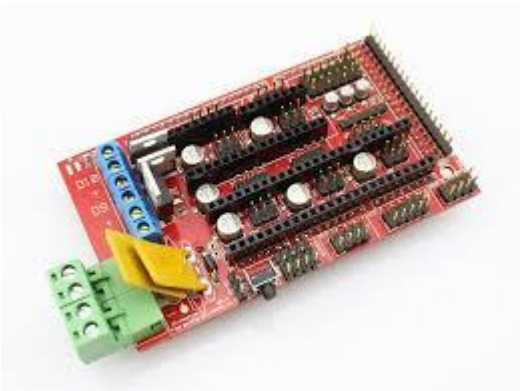


Figure 3.35: Ramps 1.4

The Ramps 1.4 is used due to its provided connection interface for required components of the SCARA robot. These components are stepper motor, stepper motor driver, end stop switch, servo and electrical magnetic.

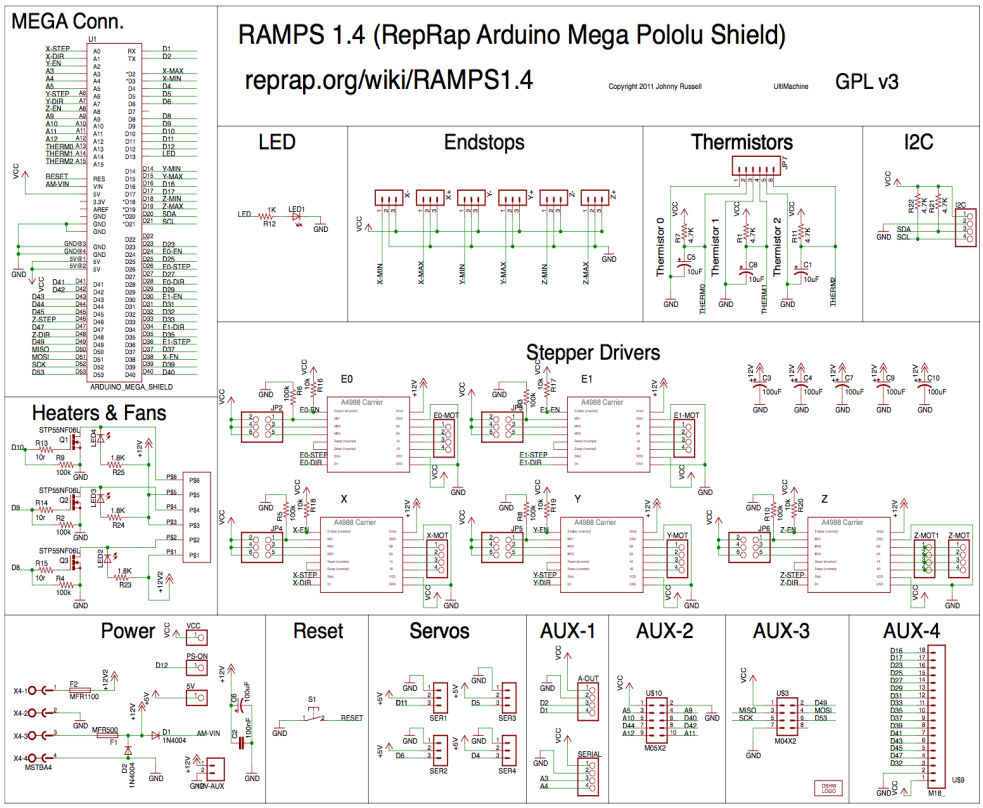


Figure 3.36: Ramps 1.4 connect interface

The Nema 17 stepper motor is used for control the SCARA robot.It provides 200 step resolution and 50 N·cm holding torque, which is suitable for pick and place the light weight designed chess piece.The given holding torque also help in pick and place process because the arm segment need to be hold standstill in that process at certain point and the .Furthermore, with 200 step resolution, when working with a stepper driver, the resolution can be increase, which provide a smooth movement.

Figure 3.37: Nema17 Motor

In this design stepper driver DRV8825 is used, it provide up to $\frac{1}{32}$ micro step control mode. By using the stepper driver, the trolling process will be facilitated by just input only the number of steps that the motor need to be executed. The driver will handle the motor input current amplitude to make it move correctly.

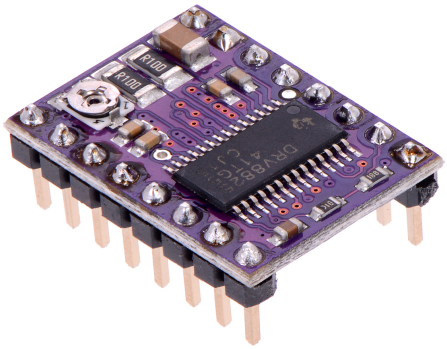


Figure 3.38: Stepper Driver DRV882

To control the pick and place tool, the micro servo is selected instead of a stepper motor because of its light weight characteristic, which will not impact on the arm balance level. Hence, the chess piece is small, so that the servo can provide enough torque to lift it up and down.

Figure 3.39: MicroServo

For detecting the home position, the end stop switch module V1.2 is used. The end stop provides a built in de-bounce circuit which help to prevent boucing problem when the arm segment touch the switch.

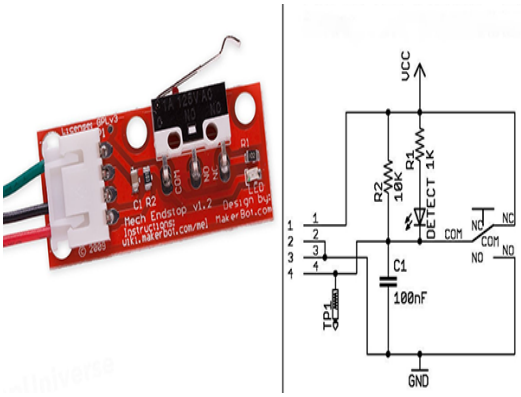


Figure 3.40: Endstop

3.2.3 Control Software

This section discusses about the control software for the SCARA arm robot, the calibration process for each rotation joint and the whole SCARA arm robot to work with the designed chess board

3.2.3.1 Configuration

The configuration file is used for storing all required data to control the SCARA arm. In the current program, X_AXIS represent for the second arm segment (the outer arm), and Y_AXIS represent for first arm segment (the inner arm)

The code for the configuration file is shown below.

```
    Configuration.h   files
#ifndef CONFIGURATION_H
#define CONFIGURATION_H

//RAMPS_V_1_4

#define X_STEP_PIN         54
#define X_DIR_PIN          55
#define X_ENABLE_PIN       38
#define X_MIN_PIN           3

#define Y_STEP_PIN         60
#define Y_DIR_PIN          61
#define Y_ENABLE_PIN       56
#define Y_MIN_PIN          18
#define PlayerButton       19

// This determines the communication speed of the printer
#define BAUDRATE 115200
// Servo Variables
#define SERVO_PIN          11
#define MAGNET_PIN         10
const float Pi = 3.14159;

//SCARA Robot Variables
```

```
float x_steps_per_degree = 55.1;
float y_steps_per_degree = 55.1;

const float PRIM_ARM_LENGTH = 250:
const float SEC_ARM_LENGTH = 250;
const float HOME_POS_OFFSET_X = 0;
const float HOME_POS_OFFSET_Y = 0;

const float Y_AXIS_HOME_ANGLE = 318.73;
const float X_AXIS_HOME_ANGLE = 162.2;

const float X_MAX = 180;                    //The primary arm can
go straight out only
const float X_MIN = X_AXIS_HOME_ANGLE;      //The primary arm can
bend in the same angle as the end stop
const float Y_MAX = 180;                    //The secondary arm
can only go out until the end stop angle
const float Y_MIN = Y_AXIS_HOME_ANGLE;      //The minimuam angle
is symmetrical to the end stop
```

The step per unit of each motor is calculated using the follow formula:

$$\frac{Stepper\_Motor\_Revolution}{360} \times micro\_step\_mode \times gear\_ration \qquad (25)$$

as discussed in the Electrical component section, Nema 17 motor resolution is 200, the micro step mode used in this SCARA arm design is $\frac{1}{32}$.The gear ratio is the ratio between the gear at arm segment and the GT2 pulley, which is 3.1.Hence the step per unit for both two motors is 55.111, however this result will be changed after the calibration process

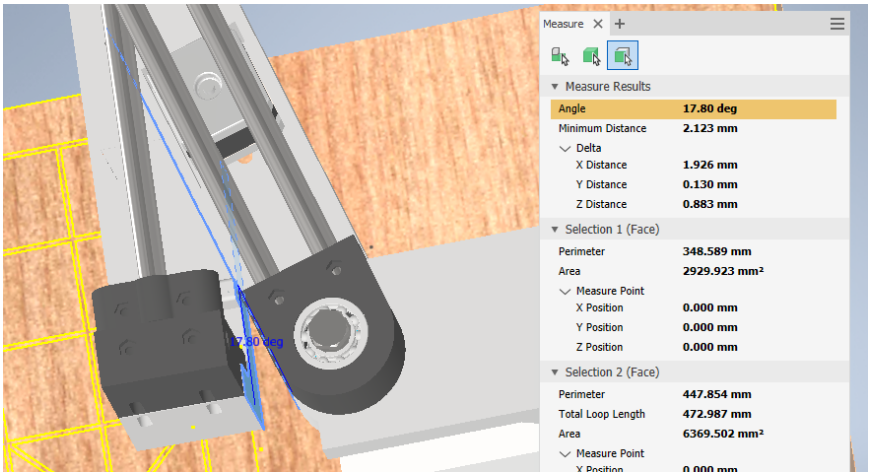


Figure 3.41: Outer arm segment homing angle

The home angle is get from the 3D model, the end stop switches are placed so that the home angle for each arm segment is approximately close to the 3D model. The calibration process is then compensate the error between the model and the real position.

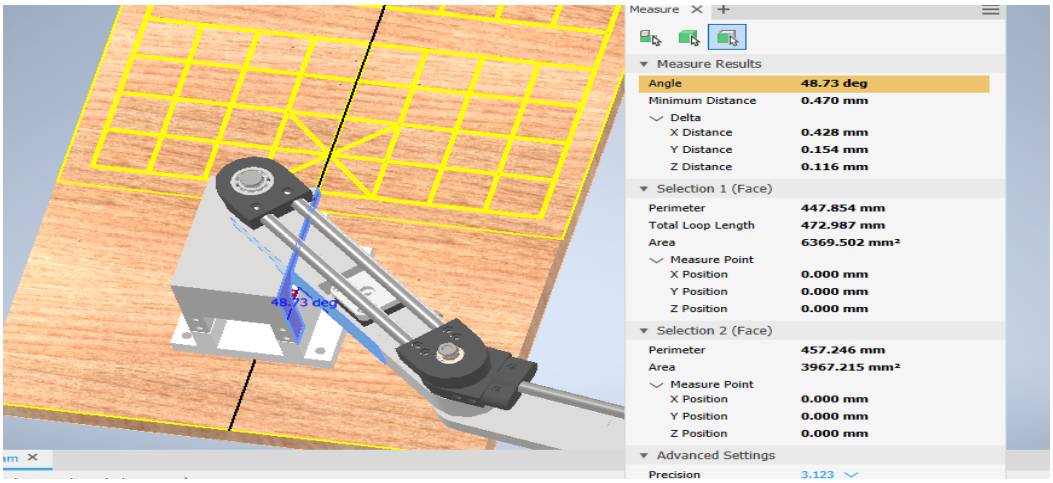


Figure 3.42: Inner arm segment homing angle

3.2.3.2 Command Reader

The command parser is responsible for reading command from computer and controlling the arm to do the appropriate action.

The computer transfer the commands to the Arduino board by serial communication protocol. The data received by the Arduino is stored as string and then parsed to appropriate command.

The command format is similar to GCODE format, which is used widely in 3D printer. For example, G28 command is used for homing both two arm segment,G0 xXyY will move the arm to position (x,y). The G character represent a group of movement commands and the number after the G character corresponds to a commands.

```
// comm variables
#define MAX_CMD_SIZE 96
#define BUFSIZE 8
char cmdbuffer[BUFSIZE][MAX_CMD_SIZE];
//bool fromsd[BUFSIZE];
int bufindw = 0;
int buflen = 0;
```

Figure 3.43: Com variable

The received data from computer is stored in cmdbuffer variable as fig.BUFSIZE variable is used for determine how many commands would be saved in the Arduino at a time.if the data sent to the Arduino is exceed the BUFSIZE, the board will stop receiving data until the old commands are excited.

Two main functions get_command() and process_command() are placed in the Arduino loop, there for the board will looping through these functions and execute them whenever it activated.

```
void loop()
{
  if (buflen < 3)
    get_command();

  if (buflen)
  {

    process_commands();

    buflen = (buflen - 1);
    bufindr = (bufindr + 1) // BUFSIZE;
  }
}

inline void get_command()
{
  while (Serial.available() > 0 && buflen < BUFSIZE)
  {
    serial_char = Serial.read();
    if (serial_char == '\n' || serial_char == '\r' || serial_count
>= (MAX_CMD_SIZE - 1))
    {
      if (!serial_count)
        return; //if empty line

      buflen += 1;
      comment_mode = false; //for new command
      serial_count = 0;     //clear buffer
    }
    else
    {
        cmdbuffer[bufindw][serial_count++] = serial_char;
    }
  }
}
```

The string is then checked whether it contains the command a not by

two function code_seen() and code_value().For example, with G28 command, the code_seen() is used for check the G character in the received string and the code_value will check the number 28.

Two function code_seen() and code_value() is showed in the below code snippet

```
inline bool code_seen(char code)
{
  strchr_pointer = strchr(cmdbuffer[bufindr], code);
  return (strchr_pointer != NULL); //Return True if a character was
found
}

inline float code_value()
{
  return (strtod(&cmdbuffer[bufindr][strchr_pointer - cmdbuffer[bufindr]
+ 1], NULL));
}
```

The code segment below shows some implemented commands for the SCARA robot

```
inline void process_commands()
{
  char *starpos = NULL;

  if (code_seen('G'))
  {
    switch ((int)code_value())
    {
    case 0: // G0 -> G1
    case 1: // G1 Linear move
    case 5: //Servo control
    case 6: //Magnetic Manipulation
    case 7:
    case 10: //rotate arm segments
    case 13: //Reset arm segment soft position
```

```
    case 14: //check endstop status
    case 15: //Homing only inner arm
    case 16: //Homing only inner arm
    case 28: //G28 Home all Axis one at a time
  }
  else
  {
    Serial.println("Unknown command:");
    Serial.println(cmdbuffer[bufindr]);
  }

  ClearToSend();
}
```

3.2.3.3 Movement Control

For controlling the stepper motor, the Accel Stepper library is used. The library provides functions to control the motors with stepper driver in different micro step mode. It also allow to synchronize the movement of two stepper motor so that they are able to move and stop nearly at the same time.

In this SCARA robot arm, two motor is controlled at constant speed with micro step mode is 32 as discussed in the configuration file section and move synchronized.

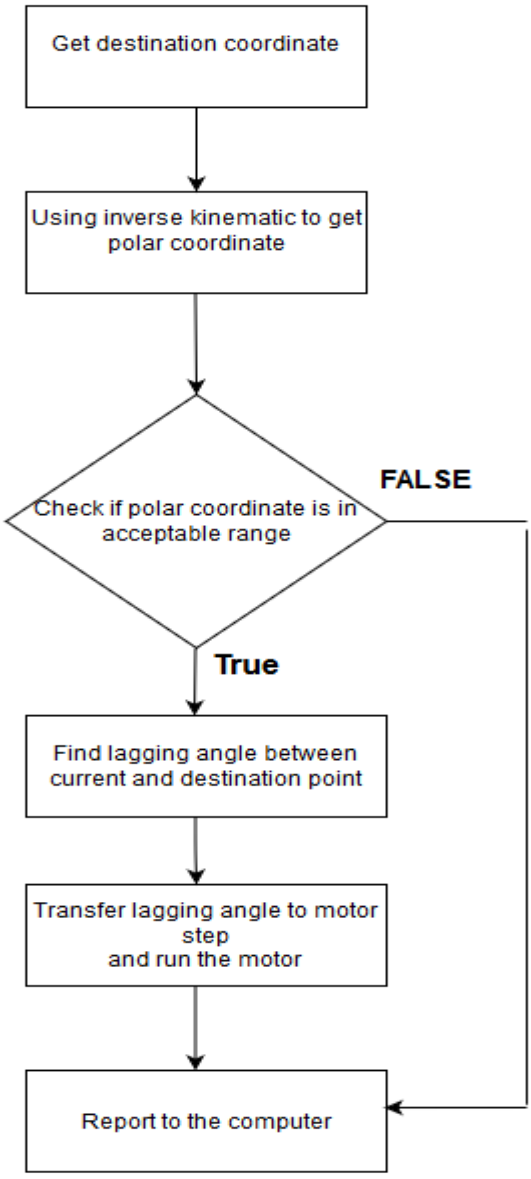


Figure 3.44: Movement Flow Chart

The workflow of the movement control is shows in figure3.44.The kine-

matic implementation of the SCARA model is showed in the code below.

```
//Forward kinematic
void Scara_to_Cartersian(float m_arm_degree, float m_forearm_degree)
{
  start_degree[0] = arm_degree;
  start_degree[1] = forearm_degree;

  arm_degree = m_arm_degree * 0.0174533;
  forearm_degree = m_forearm_degree * 0.0174533;

  start_cart[0] = PRIM_ARM_LENGTH * cos(arm_degree)
                    + SEC_ARM_LENGTH * cos(arm_degree +forearm_degree);
  start_cart[1] = PRIM_ARM_LENGTH * sin(arm_degree)
                    + SEC_ARM_LENGTH* sin(arm_degree + forearm_degree);

}
//Forward kinematic
void Cartesian_to_Scara(float x_cartesian, float y_cartesian)
{
  float tophalf;
  float bottomhalf;

  DistB = sqrt(pow(x_cartesian, 2) + pow(y_cartesian, 2));

  Theta = (atan2(y_cartesian, x_cartesian)) * 180 / Pi;

  Phi = (acos((pow(DistB, 2) + pow(PRIM_ARM_LENGTH, 2)
         - pow(SEC_ARM_LENGTH, 2))
         / (2 * DistB *PRIM_ARM_LENGTH))) * 180/Pi;

  tophalf = -pow(SEC_ARM_LENGTH, 2)
            - pow(PRIM_ARM_LENGTH, 2)
            + pow(DistB, 2);

  bottomhalf = 2 * SEC_ARM_LENGTH * PRIM_ARM_LENGTH;

  forearm_degree = (((acos(tophalf / bottomhalf)) * 180 / Pi));
```

```
  arm_degree = ((Theta - Phi));

  if ( arm_degree <= -180 ){
     arm_degree = 360 - arm_degree
  }
}
```

The motor is controlled with the positive direction is counter clock-wise. Therefore the arm_degree variable which represent for the angle of inner arm must be positive for detecting the appropriate rotation direction between current and destination coordinate. However, the acrtan2 function output result in quadrant III and IV is negative, that why the 360 degree need to be minus to the angle to convert it to positive value as show in the forward kinematic code snippet.

The process of moving the SCARA arm to a new coordinate, which is illustrated by the flow chart in figure 3.44 is showed at the below code snippet.

```
void Scara_to_Cartersian(float m_arm_degree, float m_forearm_degree)
{
  start_degree[0] = arm_degree;
  start_degree[1] = forearm_degree;

  arm_degree = m_arm_degree * 0.0174533;
void scara_move()
{

    //Offset the cartesian coordinates as the zero of the arm is
actually under the first pivot
    start_cart[0] = (start_cart[0] + HOME_POS_OFFSET_X);
    start_cart[1] = (start_cart[1] + HOME_POS_OFFSET_Y);

    destination_cart[0] = (destination_cart[0] + HOME_POS_OFFSET_X);
    destination_cart[1] = (destination_cart[1] + HOME_POS_OFFSET_Y);

    // Report_Info();
```

```
    Cartesian_to_Scara(start_cart[0], start_cart[1]);

    start_degree[0] = forearm_degree;
    start_degree[1] = arm_degree;

    Cartesian_to_Scara(destination_cart[0], destination_cart[1]);

    destination_degree[0] = forearm_degree;
    destination_degree[1] = arm_degree;

    if(isCoordinateValid(destination_degree))
    {
        degree_to_steps();
        run_motor();

        start_cart[0] = destination_cart[0] - HOME_POS_OFFSET_X;
        start_cart[1] = destination_cart[1] - HOME_POS_OFFSET_X;

        Report_Info()
    }
    else
    {
         Report_Info()
    }
}
```

3.2.3.4 Bresenham and line trajectory

Bresenharm is a line drawing algorithm.It is used for generating a set of points in order to form a close approximation to a straight line between two points. The algorithm is usually used for drawing line in bitmap image, however, due to its fast computation advantage, it is tested to use in control the SCARA robot process.

The idea of Bresenham's algorithm is to avoid floating point multiplication and addition to compute points on a straight line by using y = mx +c formula. Instead of using the previous method, the algorithm will approximate points close to the line only by adding and subtracting integer

operation.

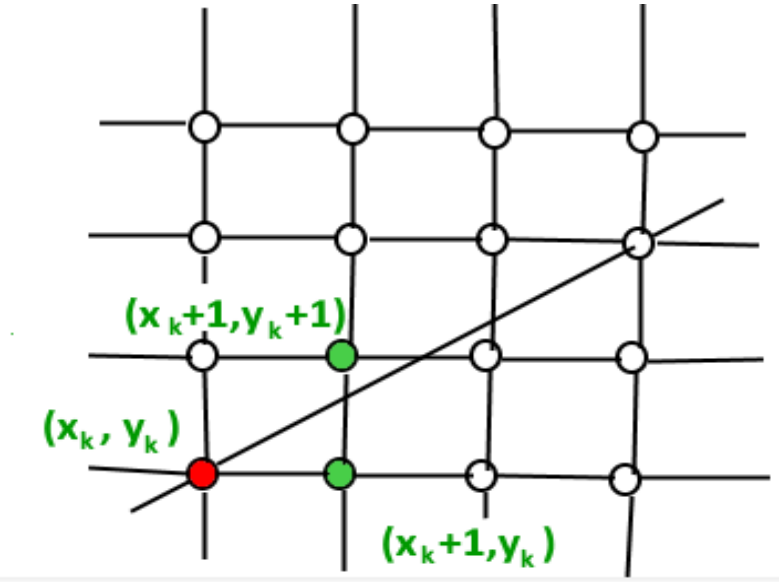


Figure 3.45: Bresenham illustration

To approximate the points, the algorithm will increase x by 1 unit and decide whether to choose point y or y+1.The point is selected based on the distance of the line to each y value, which one is closest to the line will be selected.

Using a decision formula, which calculated the difference these to distance, the next value of y can be determined.

$$D = 2\Delta X - \Delta Y \tag{26}$$

If $D > 0$ the point (X+1, Y+1) is selected, else the next point is (X +1, Y).

Using the above formula, a set of points between two points can be generated and use for control the SCARA arm to move in a straight line.

```
Line(x0,y0, x1,y1)
    dx = x1 - x0
    dy = y1 - y0
    D = 2*dy - dx
    y = y0
```

```
for x from x0 to x1
    Scara_move(x,y)
    if D > 0
           y = y + 1
           D = D - 2*dx
    end if
    D = D + 2*dy
```

The above pseudo code has been applied to the SCARA robot for testing. Due to the long arm segments, which is 250mm each, and the constant speed control approach currently used for the robot, the delay time between each movement from point to point causing an unstable movement, shaking the second arm segment. Hence, the algorithm is not applied to the current arm design.

3.2.4 Calibration

The calibration procedure includes two processes. The first process is calibrating each of arm segments to make sure it really rotates an expected angle and reducing the amount of error for second process of calibration. The second part is calibrating the whole SCARA robot system to make it move to a desire coordinate with an acceptable error.

Both two calibrating processes are conducted manually by replacing the electrical magnetic at pick and place tool to a pen. The pen is used for marking the points on paper to check the rotation angle of each arm segment and the real coordinate of the arm end effector (or the electrical magnetic).For increasing the accuracy in measurement, a caliper with accuracy of 0.1 mm is used.

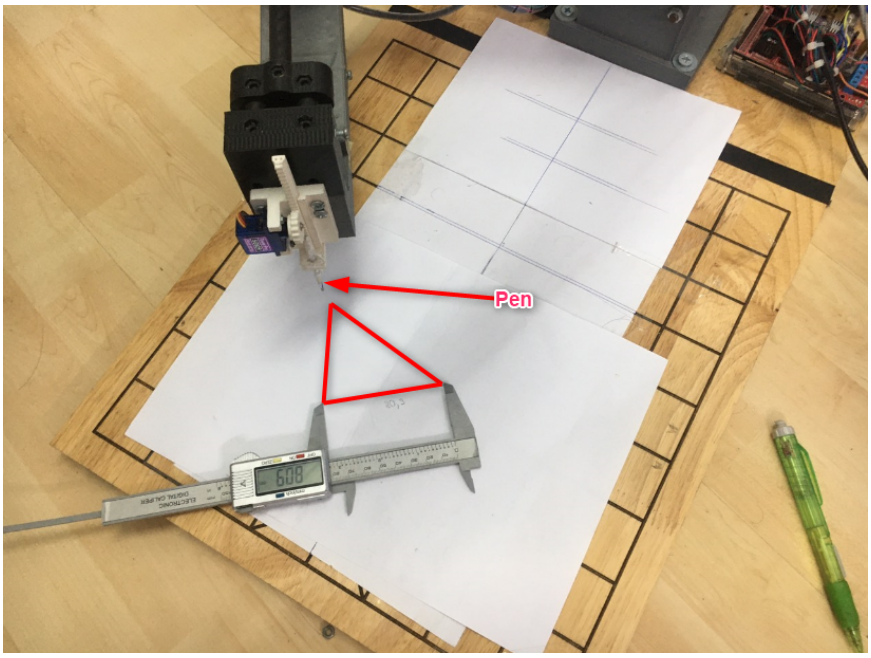


Figure 3.46: Calibration process 1

For calculating the rotation angle of the outer arm segment, two point is marked on paper. Given the length of the arm segment and distance between two point, using the law of Cosines, the real rotated angle of the arm segment can be calculated.

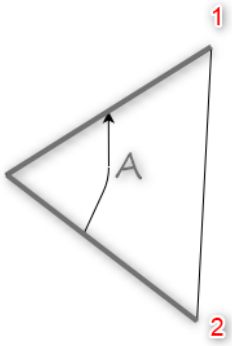


Figure 3.47: Outer arm calibration

The two gray line is the outer arm segment, point 1 and 2 is the marked point by the pencil. The below formula result the rotated angle A of the arm segment.

$$\text{ArcCos}\,(A) = \frac{-(\text{BC}^2 - 2\ ARM\_LENGTH^2)}{2 * ARM\_LENGTH} \tag{27}$$

The process of measure rotate angle of the inner arm segment is depicted as figure 3.48 , in this process, only the inner arm segment is rotated, the outer arm segment is hold standstill.

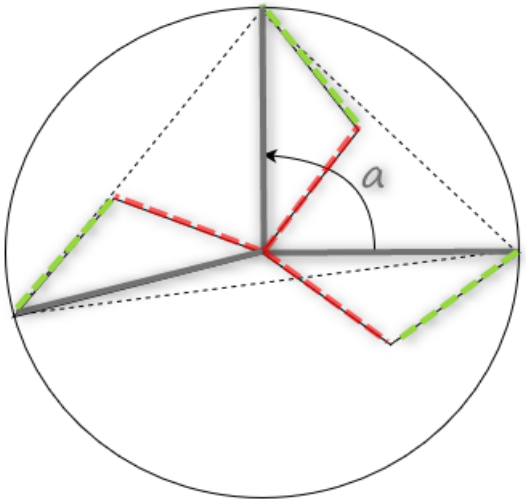


Figure 3.48: Inner arm calibration

Three point B,C,D is marked on paper when rotating the inner arm segment, which is represent in red in figure 3.48. The radius of BCD triangle circumscribed circle is calculated to get distance from the original point (0,0) to the end effector of the robot. Using the circle radius and law of Cosines, the angle **a** is derived. This angle is equal to the rotated angle of the inner arm segment because the outer arm segment is hold standstill in the whole process.

The circumscribed circle radius is calculated as follows:

1. Calculating the half triangle perimeter

$$P = \frac{BC + BC + CD}{2} \tag{28}$$

2. Calculating the triangle area with Heron formula

$$S = \sqrt{P * (P - BC) * (P - BD) * (P - DC)} \tag{29}$$

‘1

3. Calculating the radius of the circle

$$R = \frac{BC * CD * BD}{4 * S} \tag{30}$$

For calibrating the whole SCARA robot, the arm is controlled to move to a number of points on the x = 0 axis, then these point is marked on paper and measured the distance from the correct position points to derived their coordinatation.

Figure 3.49: Calibration process 2

The x = 0 is chosen as the reference axis to facilitate the process of derived coordinate of marked points. showed in figure is a paper which is attached above the chess board with line coincide to the chess board line, these line help to coordinate the marked points. Beside that, using the x = 0 or other parallel axis will show the error of placing the SCARA robot base when trying to make it coincide to the chess board coordinate.

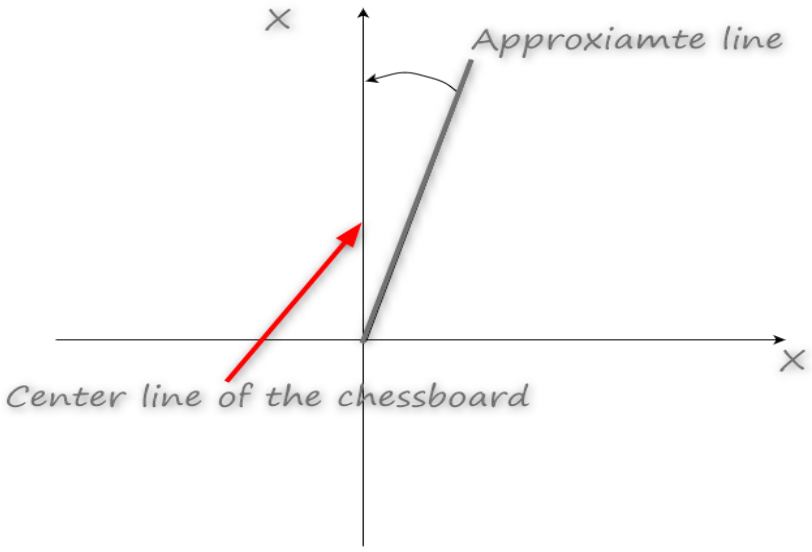


Figure 3.50: Lagging angle between two coordinates

Using the marked point and least square fit method, an approximate line of these point is derived. The least square fit is deployed to reduce the error when measuring the coordinate of these points manually. The angle between the approximate line and the reference axis is calculated and used for adjust the coordinate of the input points to the SCARA robot. The input points are then rotated as the derived angle before calculating for the arm to move.

Chapter 4

Results And Conclusion

4.1 Recognition Program Result

4.1.1 Circle detection algorithm

The Hough circle transformation algorithm is tested with multiple images from the webcam. In order to verify the accuracy of the detection, each detected circles are drawn with a thick red outline. Moreover, a debugging session is also used to check whether the data is stored in the correct structure.

At first, there were many issues with the detection such as:

- The algorithm misrecognizes the electric magnet, which also is a small cylinder, with a piece
- Some pieces are not recognizable even though the almost identical pieces next to them are detected just fine
- Sometimes, the algorithm is too sensitive it recognizes the glow of the ceiling light as a piece.
- The taken (defeated) pieces, which are situated in the graveyard, are also detected by the algorithm.

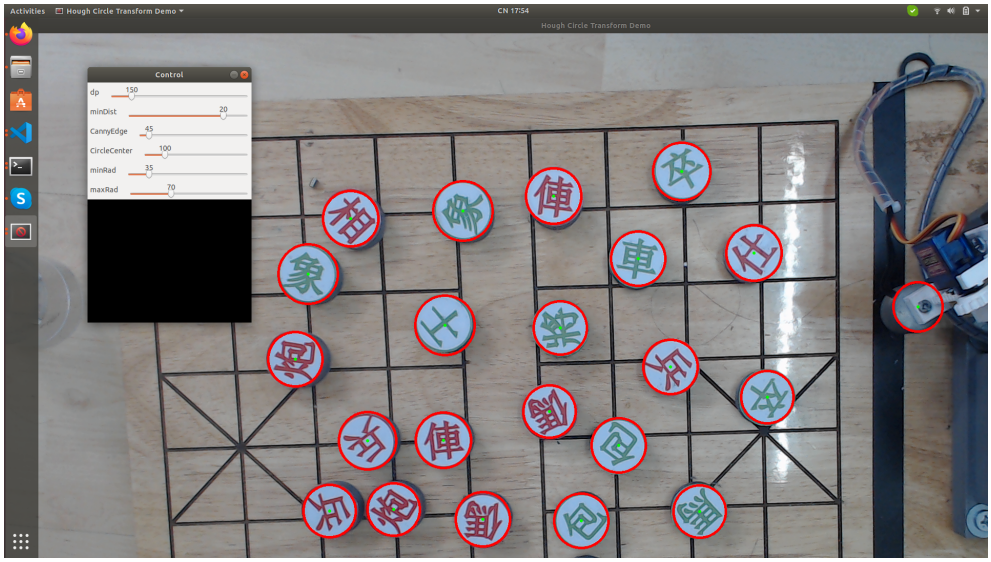


Figure 4.1: A False Recognition Scenario

The issues are solved one by one. For the first and the last issue, it can be fix by cropping the taken image to be just the board only. This solution requires some manually configuration within the code of the program. While that solution is working well, it could be improved by marking the board with a thick border, and detect a rectangle instead. The position of the graveyard is also changed for easier access and less effect on the piece detection.

The second issue, through multiple testing, has its root cause found: the edge is too thin, that multiple filter, the Canny cannot detect that as an edge. This can be solved by creating a new piece set with thicker border, as well as increase the `dp` parameter in Hough Circle Transform function.

By doing that, the third issue is also solved. Higher `dp` and increasing the level in pyramid layers help with eliminating the noise from light.

After working on those solution, the algorithm is working really fine now, with the accuracy hitting 95% through many trials. The 5% misdetection is due to extreme lighting conditions, which shouldn't happen in real scenarios.

4.1.2 Color filter algorithm

Color filter is a really tough thing to test. Since the image is converted into HSV color space before filtering, the lighting condition should have a really big effect on the image's data.

The main issue is under some circumstance, the green pattern is changed into yellow, which is again closer to red than to green according to the HSV color space. However, the inverse thing didn't seem to happen, which means it really rare that a green piece is recognize as a red piece.

Therefore, the issue is solved by carefully altering the two threshold for both the green filter and the red filter. Many lighting condition is tested, and the final result is a stable working algorithm under natural light and white artificial light.

4.1.3 Pattern recognition algorithm

The pattern recognition algorithm is pretty straight forward in conducting and testing the result. After using the track bars program to find the optimal parameters for pattern recognition, the function works very well with it.

The only issue to have ever happened during the test for patter recognition is the misrecognition between those pair: red car and red elephant,

red general and red advisor, green cannon and green horse. The cause for this mistake is if you spin 90 degrees of one piece, the detected edges are in the same place as the other one, and the algorithm mistake them altogether. This is fixed by simply increase the number of data pieces. Instead of one image per piece, it would be now four images that are 90 degrees rotated form each other. This ensure that the piece is recognized correctly no matter the rotation its take

However, there is another potential issue: The advisors on both side has too little edges. While the other pieces should have around 20+ good matches with its data, the advisor only has around 3 to 7 of them. While this is not an issue right now, it could be the cause for some misrecognition in the future and should be checked carefully.

4.1.4 Output and information transfer

Overall, the program is working well almost all the time. Once it is called, the program will take a picture, process it and output tow CSV files as greenPiece.csv and redPiece.csv. Under multiple scenarios, the output gets to around 92% accuracy, with the 8% failure due to extreme lighting conditions and sudden power cutoff.

Therefore, it could be safely concluded that the program is working as intended, and fulfills its requirements.

4.2 Chess Engine Simulation Results

4.2.1 Algorithm Setback And Solution

During the simulation process, from observation of log files, a problem occurred which is shown Figure 4.2:

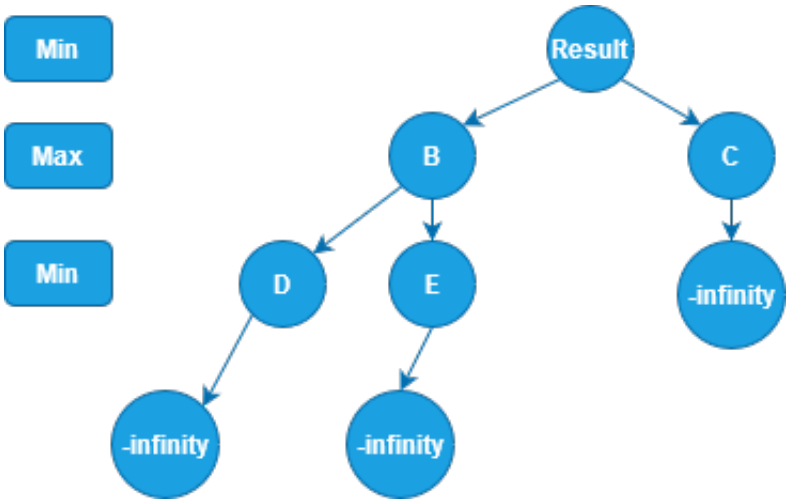


Figure 4.2: Alpha-Beta algorithm set back

- EndGame move is found in the most left branch of the tree. Since at depth 1 node is Min, therefore at D the value is -Infinity.
- However, when the value of D is returned to B, it is discarded and appended into a list because at B the node is Max. Since the game is shifting to its EndGame state, all child nodes below B has the value of -Infinity, hence when a Max value must be chosen from B, only -Infinity is chosen, leading to B having value of -Infinity.
- When B is returned to final result, since it is a Min node, result will have the value -Infinity and Beta will be set to -Infinity, satisfying the condition Beta $\leq$ Alpha. Therefore, any node after B node will be automatically pruned, leading to node C which has the True EndGame move being discarded.

- Because of this, node B is chosen again and again until it has True EndGame move. This situation extends the steps taken and increase the rate of the game being in infinite loop.

To solve this problem, Alpha-Beta algorithm pruning process must be postponed until the true EndGame move is found. To achieve this goal, whenever depth zero is reached or EndGame move is found, a tuple of both board score and the current depth + 1 must be collected instead of just the score. Furthermore, one more fixed parameter storing the highest depth must be added to the algorithm. Hence, whenever an EndGame move is found, if the depth stored along with it is less than the highest depth, then the pruning process will not be executed. With this new approached, infinite loop case is removed and less steps are taken to end the game. Figure 4.3 represents the improvement in this new approach:

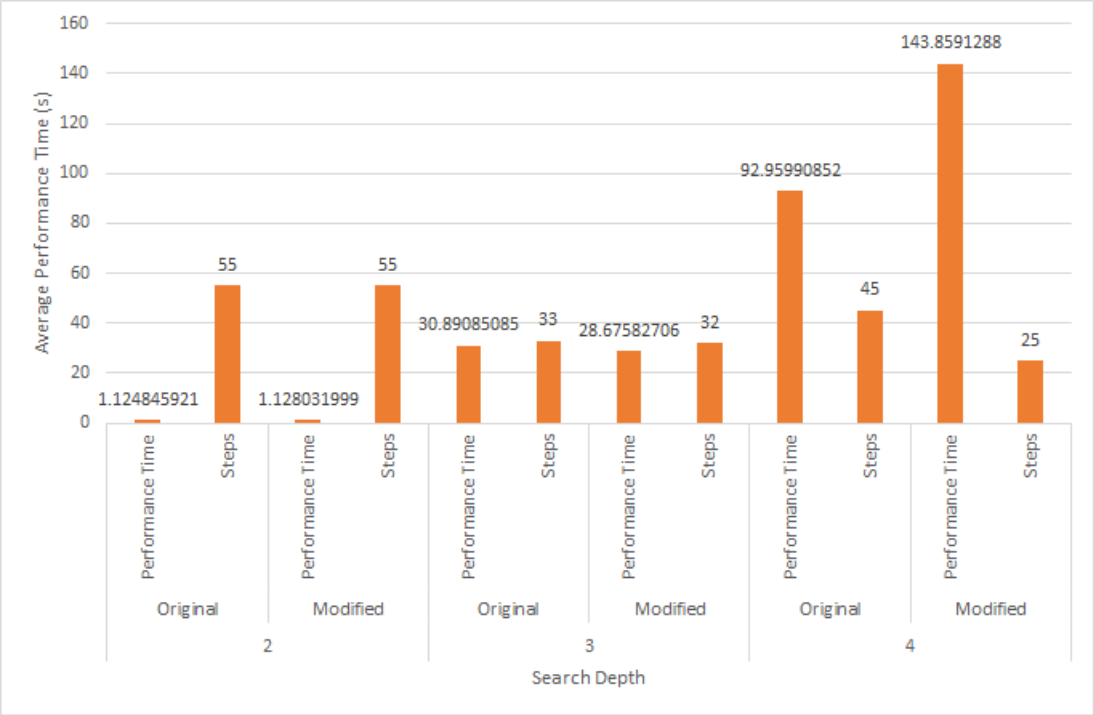


Figure 4.3: Performance time, efficiency between original and modified Alpha-Beta algorithm

4.2.2 Efficiency, Precision And Draw Back From Lower Depths

As mentioned in section 3.2.2.9, two CPU is used for the simulation of the chess engine: CPU1 is a PC and CPU2 is a Laptop. Three run times are conducted and for the entire simulation process, the chess engine operated on CPU2 has the performance time of about 3.155065409 times higher than that of CPU1. Hence, the stronger hardware configuration the chess engine has the better it operates. Figure 4.4 represents the performance time of CPU1 and CPU2:

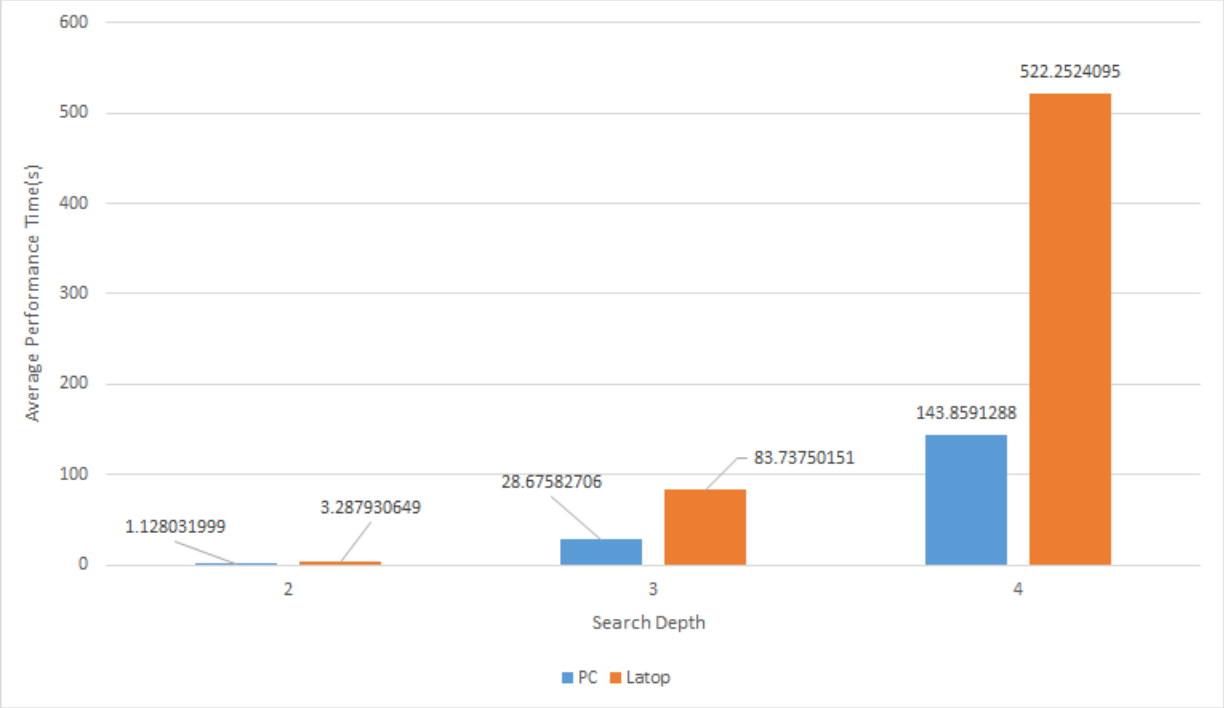


Figure 4.4: Performance time between Laptop (CPU2) and PC (CPU1)

From test runs of CPU1 ranged from depth 2 to depth 4, the average performance time and the steps to end the game were recorded. From depth 2 to depth 4, the performance time rise up to about 127 times its original performance time from 1.128 second to 143.86 second. On the other hand, steps taken to win the game drops almost 50% from 55 steps to only 25 steps. In conclussion, the lower the depth gets, the higher the precision of the chess engine will be and the longer time it will takes to generate the result. Figure 4.5 illustrates the relation between performance time and efficiency of the chess engine:

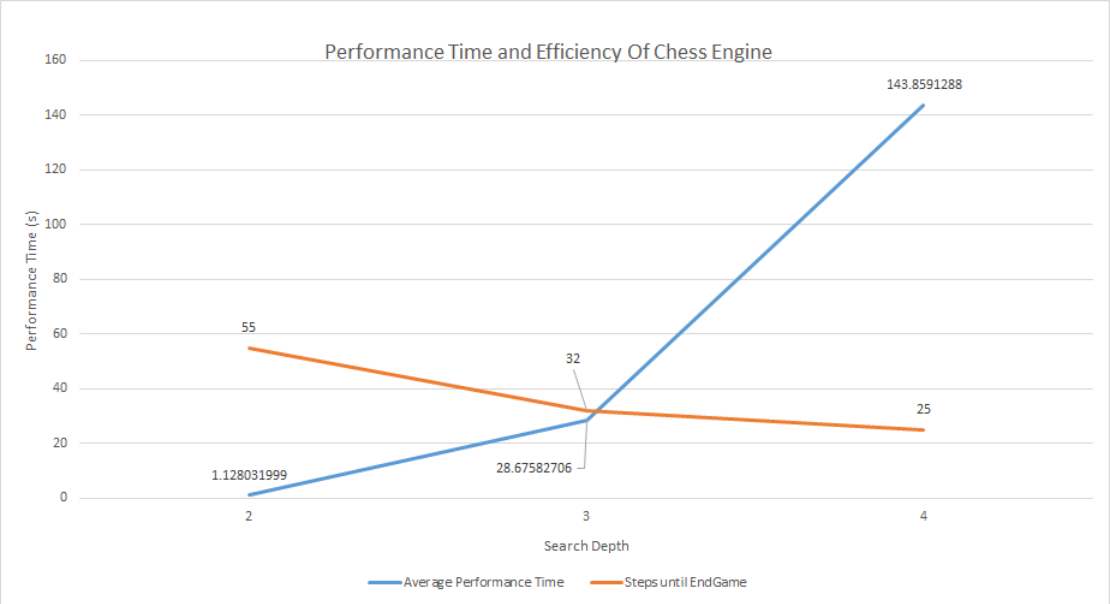


Figure 4.5: Performance time and Steps to take until EndGame

For the last simulation test, two orders of the chess piece list is presented and the simulation is conducted on CPU1. For the second order, as placing the Generals in the first position of the list instead of last in the first order, performance time is greatly reduced and the difference will be clearer in lower depths operations. Therefore, ordering of the chess pieces leaves great impact on the overall efficiency of the chess engine and the most important pieces are advised to be placed at the beginning of the list as most good moves are originate from them.

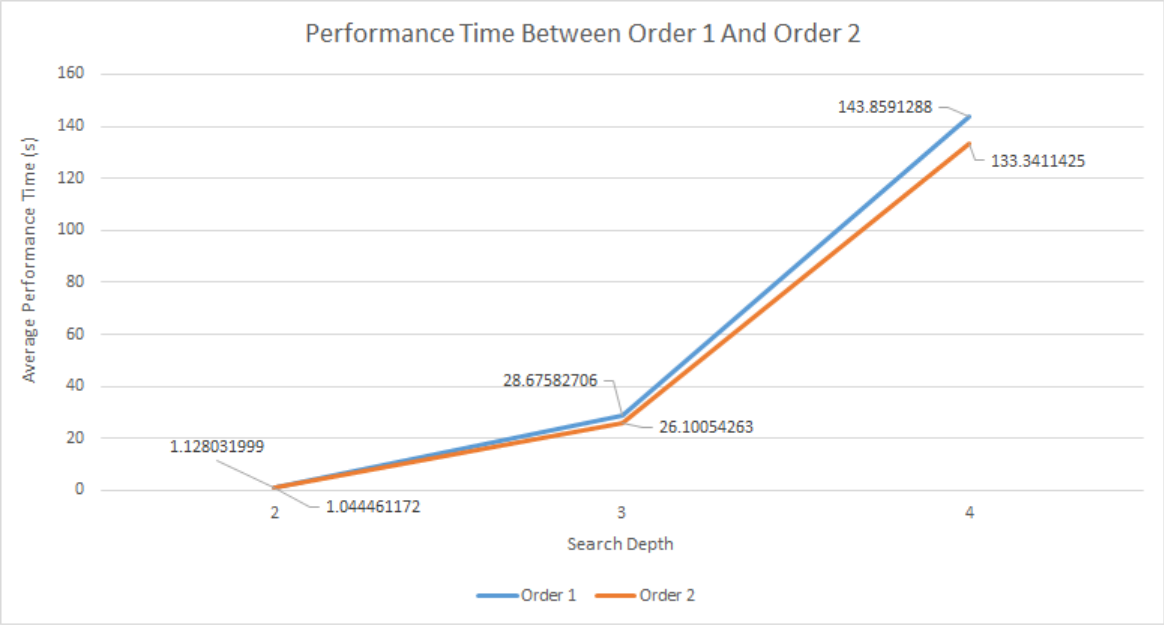


Figure 4.6: Performance time between two chess piece lists orders

4.3 Scara Calibration Result

4.3.1 Arm Segment Calibration

Number of time	Outer-Arm Lengt	Distance	Rotated Angle	Expected Angle	Error	Step Per Unit	Average Error
4	250	85.9	19.78498275	20	98.92491375	55.11	98.92269499
3	250	128.4	29.76053965	30	99.20179883		
9	250	126.4	29.28652767	30	97.6217589		
1	250	128.2	29.71311514	30	99.04371714		
11	250	134.7	31.25713297	32	97.67854053		
7	250	142	32.99816611	33	99.99444276		
8	250	141	32.75921458	33	99.27034721		
10	250	137.5	31.92402833	33	96.73947977		
6	250	152	35.39602327	35	101.131495		
2	250	149.8	34.86715958	35	99.62045593		
5	250	171.5	40.1195121	40	100.2987803		
1	250	129.2	29.95028993	30	99.83429976	56.562	100.047254
2	250	143.3	33.30902408	33	100.9364366		
3	250	150.9	35.13149484	35	100.3756995		
4	250	138	32.04323298	32	100.1351031		
5	250	86.7	19.97112286	20	99.85561429		
6	250	94.6	21.81220161	22	99.14637094		

Figure 4.7: Outer arm calibration result

Current Step Per Unit	BC	CD	DB	Rotated Angle	Expected Angle	Error	Avarage Error	Difference	Step Adjusted
55.11	51.4	51.4	99.7	28.2130065	30	94.04335	99.16476851	0.835231	55.5703
	56.8	56.8	108.5	34.46687193	33	104.4451			
	60.1	59.6	114.4	34.3751054	35	98.21459			
	63.3	63.3	120	37.16445622	37	100.4445			
	68.1	68.1	128.2	39.47054319	40	98.67636			
55.57029607	57	57	110.1	30.06031059	30	100.201	100.124967	-0.12497	55.50085155
	67	67	128.5	32.94529021	33	99.83421			
	68.2	68.2	130.1	34.96364009	35	99.89611			
	70.2	70.2	133.1	37.11465581	37	100.3099			
	75.7	75.7	142.2	40.15343671	40	100.3836			

Figure 4.8: Inner arm calibration result

In the processes of calibrating the rotation angle for arm segments,multiple rotation angles is tested with different arm positions.Because the process is

conducted manually, hence the result cannot have a high accuracy.However, for the pick and place purpose, the error about 1% is acceptable.The reason for the error come from two sources.One is from the gear timing belt system and the other is from the measurement.

The result in white section from both figure 4.7 and 4.8 is the result of calibration and is used as parameter for step per unit variable.

4.3.2 Whole Arm Calibration

Index	X (Origin)	Y (Origin)	X_1	Y_1	X_2	Y_2	Error_X	Error_Y
2	1	124	-2	127	1.2	126.1	0.2	2.1
3	1	126	-2	129	1.3	129.1	0.3	3.1
4	1	166	-3	167	1.1	166.1	0.1	0.1
5	1	168	-3	170	1.2	169.1	0.2	1.1
6	1	208	-4	209	1.2	209.9	0.2	1.9
7	1	210	-4	211	1	212	0	2
8	1	250	-5	250	1	252	0	2
9	1	294	-7	294	1.2	295	0.2	1
11	1	334	-8	334	3.4	334.17	2.4	0.17
12	1	336	-8	336	3.5	336.2	2.5	0.2
13	1	376	-10	376	1.3	376.2	0.3	0.2
14	1	378	-10	378	1.5	378	0.5	0
15	1	420	-12	420	0.3	420.2	-0.7	0.2
16	1	460	-13	460	0.7	460.24	-0.3	0.24
Approximate line				$\hat{y}$ = -29.27548X + 85.06651			0.421429	1.02214
Angle				1.956 Degree				

Figure 4.9: Whole arm calibration

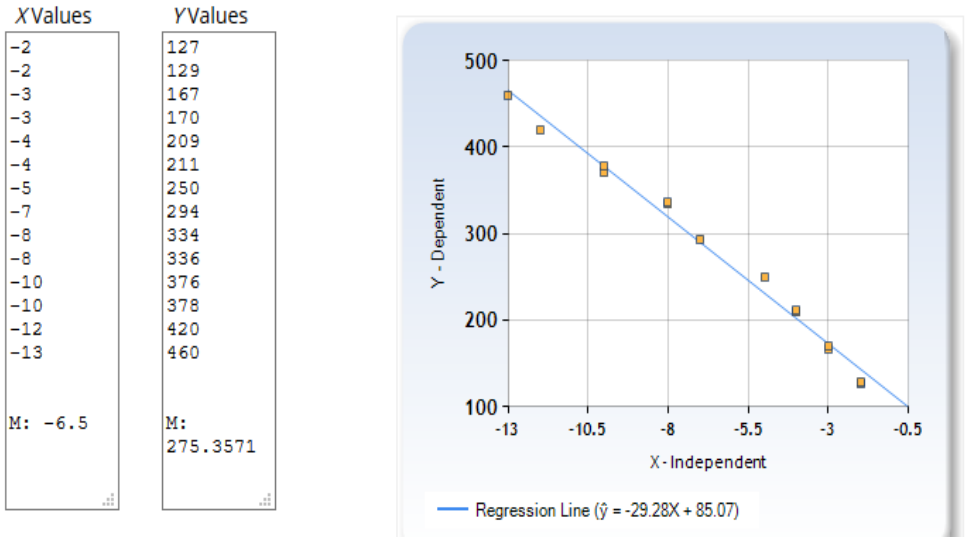


Figure 4.10: Linear regression

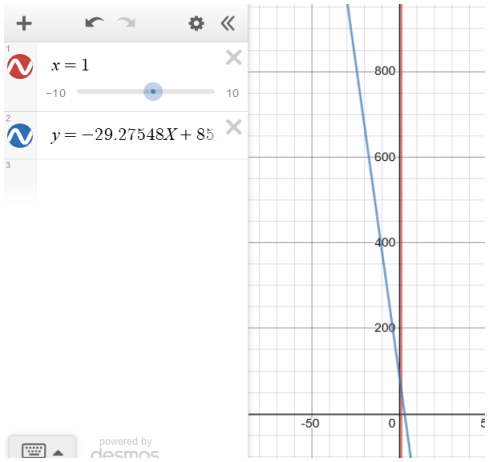


Figure 4.11: Whole arm calibration

The columns (Y_1;X_1) is the set of points measured for the first time to get the approximate line and the next two columns is the set of point measured after applying the rotation point process. The calibration for the whole arm shows that the arm coordinate is lagged behind the coordinate of the chess board about 1.956° as show in figure 3.50 .Hence to adjust the SCARA arm,the input point will be rotated 1.956° degrees before go to the

movement process.With this rotation angle the average error for X coordinate is $\pm$ 0.42 mm and for Y coordinate is $\pm$ 1 mm.This is an acceptable error the SCARA arm because the pick and place object is chess pieces has diameter of 3 cm,hence the error in movement with not make the pick and place tool miss picking.

Bibliography

[1] Carolyn Kimme, Dana Ballard, and Jack Sklansky. "Finding circles by an array of accumulators". In: *Communications of the ACM* 18.2 (Feb. 1975), pp. 120–122. ISSN: 0001-0782. DOI: 10.1145/360666.360677. URL: https://doi.org/10.1145/360666.360677 (visited on 05/17/2020).

[2] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (Nov. 1986). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 679–698. ISSN: 1939-3539. DOI: 10.1109/TPAMI.1986.4767851.

[3] Rachid Deriche. "Using Canny's criteria to derive a recursively implemented optimal edge detector". en. In: *International Journal of Computer Vision* 1.2 (June 1987), pp. 167–187. ISSN: 1573-1405. DOI: 10.1007/BF00123164. URL: https://doi.org/10.1007/BF00123164 (visited on 05/17/2020).

[4] J. Illingworth and J. Kittler. "The Adaptive Hough Transform". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-9.5 (Sept. 1987), pp. 690–698. DOI: 10.1109/TPAMI.1987.4767964.

[5] R.A. Haddad and A.N. Akansu. "A class of fast Gaussian binomial filters for speech and image processing". In: *IEEE Transactions on Signal Processing* 39.3 (Mar. 1991). Conference Name: IEEE Transactions on Signal Processing, pp. 723–727. ISSN: 1941-0476. DOI: 10.1109/78.80892.

[6] Yngvi Björnsson and T. Anthony Marsland. "Multi-cut alpha-beta-pruning in game-tree search". In: *Theoretical Computer Science* 252 (Feb. 2001), pp. 177–196. DOI: 10.1016/S0304-3975(00)00081-5.

[7] Tinne Hoff Kjeldsen. "Arch. Hist. Exact Sci. 56 (2001) 39–68c©Springer-Verlag 2001John von Neumann's Conception of the MinimaxTheorem: A Journey Through DifferentMathematical Contexts". In: 56 (2001), pp. 39–68.

[8] Richard Bellman. *Dynamic Programming*. English. Reprint edition. Mineola, N.Y: Dover Publications, Mar. 2003. ISBN: 978-0-486-42809-3.

[9] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *Int. J. Comput. Vision* 60.2 (Nov. 2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94 (visited on 10/07/2019).

[10] Shi-Jim Yen et al. "Computer chinese chess". In: *ICGA Journal* 27.1 (2004), pp. 3–18.

[11] Mohamed Rizon et al. "Object Detection using Circular Hough Transform". In: *American Journal of Applied Sciences* 2 (Dec. 2005). DOI: 10.3844/ajassp.2005.1606.1609.

[12] Edward Rosten and Tom Drummond. "Machine Learning for High-Speed Corner Detection". en. In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 430–443. ISBN: 978-3-540-33833-8. DOI: 10.1007/11744023_34.

[13] Alexandr Andoni and Piotr Indyk. "Near-optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 117–122. ISSN: 0001-0782. DOI: 10.1145/1327452.1327494. URL: http://doi.acm.org/10.1145/1327452.1327494 (visited on 10/07/2019).

[14] Herbert Bay et al. "Speeded-Up Robust Features (SURF)". In: *Computer Vision and Image Understanding*. Similarity Matching in Computer Vision and Multimedia 110.3 (June 2008), pp. 346–359. ISSN: 1077-3142. DOI: 10.1016/j.cviu.2007.09.014. URL: http://www.sciencedirect.com/science/article/pii/S1077314207001555 (visited on 10/07/2019).

[15] Ryuji Funayama et al. "Robust interest point detector and descriptor". en. US20090238460A1. Sept. 2009. URL: https://patents.google.com/patent/US20090238460A1/en?q=FUNAYAMA+RYUJI%2c&q=TUYTELAARS+TINNE&q=B.+HERBERT&inventor=YANAGIHARA+HIROMICHI%2c&assignee=VAN+GOOL+LUC%2c&oq=FUNAYAMA+RYUJI%2c+YANAGIHARA+HIROMICHI%2c+VAN+GOOL+LUC%2c+TUYTELAARS+TINNE+and+B.+HERBERT (visited on 10/07/2019).

[16] Marius Muja and David Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." In: vol. 1. Jan. 2009, pp. 331–340.

[17] Edward Rosten, Reid Porter, and Tom Drummond. "Faster and Better: A Machine Learning Approach to Corner Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.1 (Jan. 2010). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 105–119. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2008.275.

[18] Estevão S. Gedraite and Murielle Hadad. "Investigation on the effect of a Gaussian Blur in image filtering and segmentation". In: *Proceedings ELMAR-2011*. ISSN: 1334-2630. Sept. 2011, pp. 393–396.

[19] Donald E. Knuth. *The Art of Computer Programming, Volumes 1-4A Boxed Set*. English. 1 edition. Amsterdam: Addison-Wesley Professional, Mar. 2011. ISBN: 978-0-321-75104-1.

[20] Ethan Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. ISSN: 2380-7504. Nov. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[21] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. English. 4 edition. New York, NY: Pearson, Mar. 2017. ISBN: 978-0-13-335672-4.

[22] *Stepper Motor : Basics, Types and Working*. Engineers Garage. Library Catalog: www.engineersgarage.com. July 5, 2019. URL: https://www.engineersgarage.com/article_page/stepper-motor-basics-types-and-working/ (visited on 05/21/2020).

[23] *Kinematics: Why Robots Move Like They Do*. URL: https://blog.robotiq.com/kinematics-why-robots-move-like-they-do (visited on 05/21/2020).

[24] *PSoC 3 Stepper Motor Control with Precision Microstepping — DigiKey*. URL: https://www.digikey.co.il/en/articles/psoc-3-stepper-motor-control-with-precision-microstepping (visited on 05/21/2020).

[25] Harvey Rhody. "Lecture 10: Hough Circle Transform". en. In: (), p. 22.

[26] *Timing Belts and Pulleys Sizing & Measurement — MISUMI Blog*. URL: https://blog.misumiusa.com/timing-belts-pulleys-sizing-measurement/ (visited on 05/21/2020).