**Department of Mechanical and Mechatronics Engineering**
The University of Waterloo

# LEGO MINDSTORMS STOCKFISH CHESS ROBOT

**Prepared By:**
Jonathan Di Giorgio, Yulai Duan, Daniel Martire, Abbas Ali Asghar
200 University Ave W.
Waterloo, Ontario, N2L 3G1

April 6th, 2023

150 University Ave W.
Waterloo, Ontario, N2N 2N2

October 6, 2023

Prof Hulls, and Prof Teertstra
Department of Mechanical and Mechatronics Engineering
University of Waterloo
200 University Ave W.
Waterloo, Ontario, N2L 3G1

Dear Professor Hulls and Prof Teertstra,

This report, entitled Lego Mindstorms Stockfish Chess Robot, was written to complete our 1B academic term. This is our 1st design report for ME 101.

This robot was designed following all the design criteria outlined in the Lego Mindstorms project document for ME 101. This project was an amazing experience on the integration of mechanical and software design. It was an extremely valuable experience, and it was an honor to work on this project together.

This report was written entirely by us and has not received any previous academic credit at this or any other institution. This project took us 3 months to complete during the academic term.

Best Regards,

Jonathan DiGiorgio
ID: 21007800
1B Mechanical Engineering

Abbas Ali Asghar
ID: 21024129
1B Mechanical Engineering

Yulai Duan
ID: 21002475
1A Mechanical Engineering

Daniel Martire
ID: 21009182
1A Mechanical Engineering

# Table of Contents

List of Figures ..... iv
List of Tables ..... v
Summary ..... 1
1.0 Introduction ..... 2
1.1 Background ..... 2
1.2 Objective ..... 2
2.0 Preliminary Design ..... 3
2.1 Needs Analysis ..... 3
2.2 Conceptual Design ..... 5
3.0 Mechanical Design ..... 9
3.1 General Layout ..... 9
3.2 Mechanical Functions ..... 10
3.3 Verification of Design ..... 12
4.0 Software Design ..... 14
4.1 Description ..... 14
4.2 Functions and Data Storage ..... 16
4.3 Software Testing ..... 18
4.4 Significant & remaining challenges ..... 19
5.0 Project Management ..... 21
5.1 Work Breakdown Structure ..... 21
5.2 Project Scheduling ..... 22
6.0 Conclusions ..... 24
6.1 Conclusions ..... 24
6.2 Next Steps / Recommendations ..... 24
Appendix A – EV3 Source Code ..... 26
Appendix B – External Source Code ..... 36
References ..... 38

# List of Figures

Figure 1 Concept using magnets and sensors ........ 6
Figure 2 Concept using an arm, hook and manual control ........ 6
Figure 3 Concept using claw, overhead movement and mapped coordinates ........ 7
Figure 4 General mechanical layout of the robot ........ 9
Figure 5 Base robot structure ........ 10
Figure 6 Motor, rack and pinion housing mechanism, b) Belt-housing movement mechanism ........ 11
Figure 7 Wheel movement system ........ 12
Figure 8 Spending breakdown ........ 13
Figure 9 High-level software flowchart ........ 15
Figure 10 Data Structure Plan and Integer Value Meanings ........ 18
Figure 11 Gantt Chart for initial project schedule ........ 22
Figure 12 Gantt Chart for actual Project schedule ........ 23

# List of Tables

Table 1: Engineering Specification Table ........................................ 3
Table 2: Morphological Matrix........................................ 5
Table 3: Decision-Making Matrix ........................................ 7
Table 4: Function Descriptions........................................ 16
Table 5: Work Breakdown Structure........................................ 21

# Summary

The purpose of this project was to create a functional chess robot capable of making strategic moves and having full coverage of the board whilst limiting the amount human input required for the operation of the robot. Upon considering the different design concepts, a claw-gripper mechanism was chosen for the lift and release of each piece. The robot has a list of mechanical functions to support the lift-release process. A rack-pinion system was created and allows the gripper to fully extend and retract, operating the vertical axis. The horizontal axis is operated by a rotating belt system, which moves the rack-housing across the horizontal of the board. All these functions are assembled onto the base frame of the robot, which moves across the vertical of the board through motor attached wheels on the base.

The software design is what allows for these mechanical functions. A list of tasks the robot should be capable of completing was made upon considering the mechanical functions, the operating environment, and other reasonable factors such as the time restriction on the game. Most of the programming, however, was done outside of the EV3 brick, given that image recognition programs and chess engines such as Stockfish, were not easily accessible in RobotC. Regarding the RobotC code, written functions include such that would allow the robot to move across the x-axis, y-axis, and x-y axis simultaneously, and other functions regarding the game itself, such as performing an *enPassant* or a *castle* move. The software was heavily tested for the purpose of this robot being so dependent on accuracy and precision to serve its operating purpose. Upon finalizing the code, it was found that the software operated the robot with a high degree of precision.

Overall, the project itself is considered a major success, however many challenges remain that could better the operation of the robot. This includes finding sources to its declining accuracy with respect to time throughout the operating period of each game. As the pieces became less centered, the computer vision code (written separately in python) would blur, impacting the piece detection and resulting in the robot crashing. This could be improved upon given time for further measurement and testing of the robot.

# 1.0 Introduction

## 1.1 Background

Chess is one of the oldest and most popular board games of all time. It involves a set of white and black pieces, each of which is assigned a specific movement/attacking pattern. The goal of the game is to use the pieces and their abilities to hunt the enemy king such that he no longer has any way to escape an attack, better known as 'checkmating' him. It is the ultimate game of strategy and tactics.

Chess engines are coded algorithms which can determine the best move in a given position. They have become extremely popular in the chess space as they allow players to analyze games and see which moves they missed. The most popular chess engine in the space is Stockfish, a free to use open-source C++ algorithm with python support [1].

In recent years, the game has gained a lot of popularity and interest due to its simplicity and accessibility with the increasing promotion of online chess platforms. It was thought that creating a robot that could make strategic competitive moves would be a design challenge achievable with materials accessible to all members of the group. A further motive for this project was the consideration of individuals facing handicaps who might want a physical opponent to play against, or simply want to better themselves in the game of chess. Creating a robot that could perform tactical moves without human input would be a solution to these problems, and therefore became the basis for the creation of the robot.

## 1.2 Objective

Implementing the ideation and design concepts into the robot required a great deal of planning, research and troubleshooting. Considering the resources and experience levels of the group, making a perfect chess robot that fully satisfies all the original criterion for its design was improbable. Hence, factoring in the accessible resources, time constraints and other factors, the objective of this project was to fabricate a Lego chess robot that will provide an alternative means for playing the game of chess.

# 2.0 Preliminary Design

This section covers all the steps taken before official design begun. This includes the ideation, needs analysis and conceptual design.

## 2.1 Needs Analysis

A functional requirement of this robot would be for the robot to have full board coverage and be capable of communicating with the chess engine to make legal and reasonable chess moves. This derived a non-functional requirement which was to ensure the robot provides the opposing player with a near complete visual coverage of the board and its pieces throughout each turn. A constraint requirement that was faced upon designing the robot involved the cost, which was set to not exceed $25 for each group member and $100 total. A further constraint regarding the robot refers to performance of the robot, which outlines that the robot should take no longer than 1 minute in processing, calculating, and executing each move all together.

Considering the functional, non-functional, and constraint requirements pertaining to the design of the robot, the following need statement was created:

> A need exists for a robot to pick up pieces across the range of the board whilst making strategic moves in a chess match without the total design cost of the robot exceeding $25 per group member.

To provide a set of expectations and requirements for this project, the following design specifications table was created (Table 1), listed of functional, non-functional and constraint requirements.

*Table 1: Engineering Specification Table*

| No. | Characteristic | Relation | Value | Units | Verification Method | Comments |
|---|---|---|---|---|---|---|

| 1 | Cost | < | 20 | $ / person | Analysis | Research and consider total receipt of the materials required to build the robot |
|---|---|---|---|---|---|---|
| 2 | Visual Coverage | N/A | N/A | N/A | Demonstration | Show that the board can be fully seen from the perspective of the opponent |
| 3 | Time taken to make a move | < | 1 | min | Test | Recording the time it takes for the robot to perform its operations |
| 4 | Ability to pick up and move pieces | > | 90 | % | Test | Measure the success rate of the robot physically moving pieces (a failure would be moving to an incorrect square, dropping the piece, or knocking down other pieces) |
| 5 | Ease of use | > | 75 | % | Test | Have different individuals play the chess game and report convenience level of playing with the robot. Take the average. |
| 6 | Ability to communicate with a chess engine | NA | NA | NA | Demonstration | Measure the success rate of the robot inputting commands into the engine and getting reasonable outputs (valid moves that make sense given a position) |

| 7 | Safety | NA | NA | NA | Analysis | Ensure the robot does not damage individuals or the surrounding environment in a certain play setting. Consider sharp edges of the robot and pieces. |
|---|---|---|---|---|---|---|

It is worth noting however that these engineering specifications changed slightly throughout the course of the project. As the design process began, certain specifications seem less and less likely to work out. As such, updates were made to provide more realistic specifications for the robot. One such change was the change from 15 seconds to 1 minute for the maximum move time. In the early design process, it became clear that the robot would have to move slowly to be more precise. Taking this into account, the move time was switched and speed was sacrificed in favour of precision, which was a higher priority and more important for the functionality of the robot.

## 2.2 Conceptual Design

There were different design concepts that were considered feasible during the ideation phase. In order to organize these ideas, the group started with creating a morphological matrix (Table 2).

*Table 2: Morphological Matrix*

| SUBFUNCTION | CONCEPT #1 | CONCEPT 2 | CONCEPT 3 |
|---|---|---|---|
| **GRABBING PIECES** | Claw (Chosen) | Magnet | Hook |
| **MOVING GRABBER** | Overhead movement (Chosen) | Arm | Magnets underneath board |
| **DETECTING PIECES / MOVING TO PIECES** | Sensors on each square | Mapping board coordinates in code (Similar idea chosen) | Manual control |
| **DETERMINING WHERE THEY NEED TO BE MOVED** | Free source code chess engine (Chosen) | Design a chess engine in RobotC | Random moves |

After defining multiple concepts for each subfunction, 3 design concepts were chosen and sketched out.

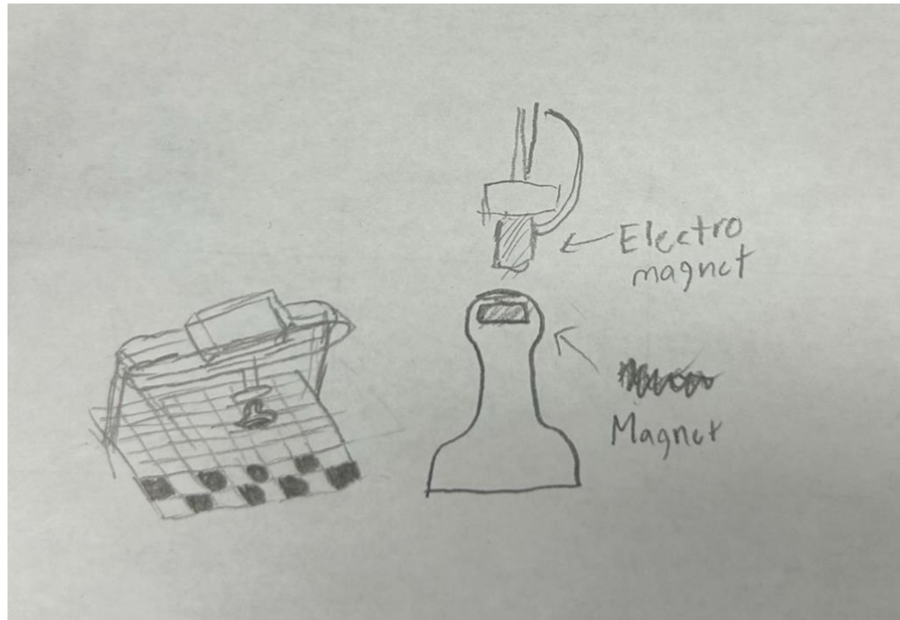


Figure 1 Concept using magnets and sensors

The first concept considered makes use of electromagnet to lift and drop individual pieces by controlling the presence of a current supply (Figure 1). The magnet would be moved around the board using the overhead movement system. The software aspect for this design was undecided.

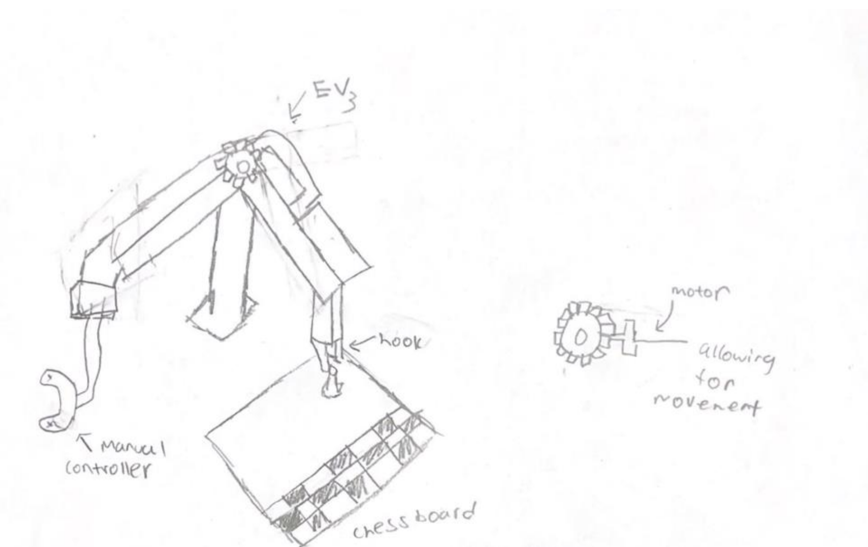


Figure 2 Concept using an arm, hook, and manual control

Concept two uses a claw gripper mechanism for picking and unloading pieces from the board shown in Figure 2. It would be operated using a manual controller that would allow the user to manipulate the x-y-z movements of the arm that the claw is attached to.

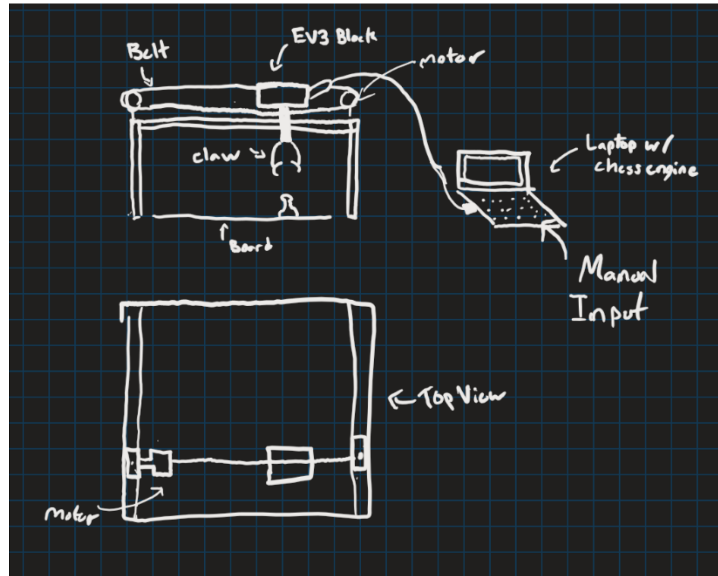


*Figure 3 Concept using claw, overhead movement and mapped coordinates*

The final design concept (the solution that was proceeded with) involves the same claw gripping mechanism that would operate a set of x-y-z motions by having the claw mount which operated vertical motion using a rack-pinion method, and the horizontal motion using a motor-belt attachment (Figure 3). The moves would be decided by a chess engine and controlled using code.

*Table 3: Decision-Making Matrix*

| **Criteria** | | | |
|---|---|---|---|
| Consistency | 0 | -1 | -1 |
| Cost | 0 | -1 | 0 |
| Ease of use | 0 | 0 | -1 |

| Ease of manufacturing | 0 | 1 | -1 |
|---|---|---|---|
| Total (+) | 0 | 1 | 0 |
| Total (-) | 0 | -2 | -3 |
| Total | 0 | -1 | -3 |

After the three designs were finalized, a decision-making matrix was created (Table 3) where concept 3 was chosen as the datum to which the other concepts were compared to. The manual controller method was unfavourable due to the risk of human error and potentially complicated and inaccessible controls. The magnetic attachment method would require the purchase of individual magnetic pieces and would have difficulty achieving precise piece lifting for each move, making it expensive and less precise. Therefore, considering the consistency, cost, ease of use and manufacturing, the claw gripper method was the design chosen.

# 3.0 Mechanical Design

This section covers the entire mechanical design process for the chess robot. It will cover each component which involved a design decision and show exactly how the components work together to create the final product.

## 3.1 General Layout

As mentioned, the design chosen makes use of a claw gripper mechanism to lift and release individual chess pieces. The gripper has full coverage of the board (x-y-z) directions. To operate the vertical a rack-pinion method is used, with the gripper movement cohesive to the rack-pinion movement. This system is attached to a belt, which slides along the horizontal of the board, and is responsible for motion in the x-axis. The y-axis (along the vertical of the board), is operated by a 3-wheel delivery system, moving the entire robot across the vertical of the board. The general layout of the robot can be seen in Figure 4.

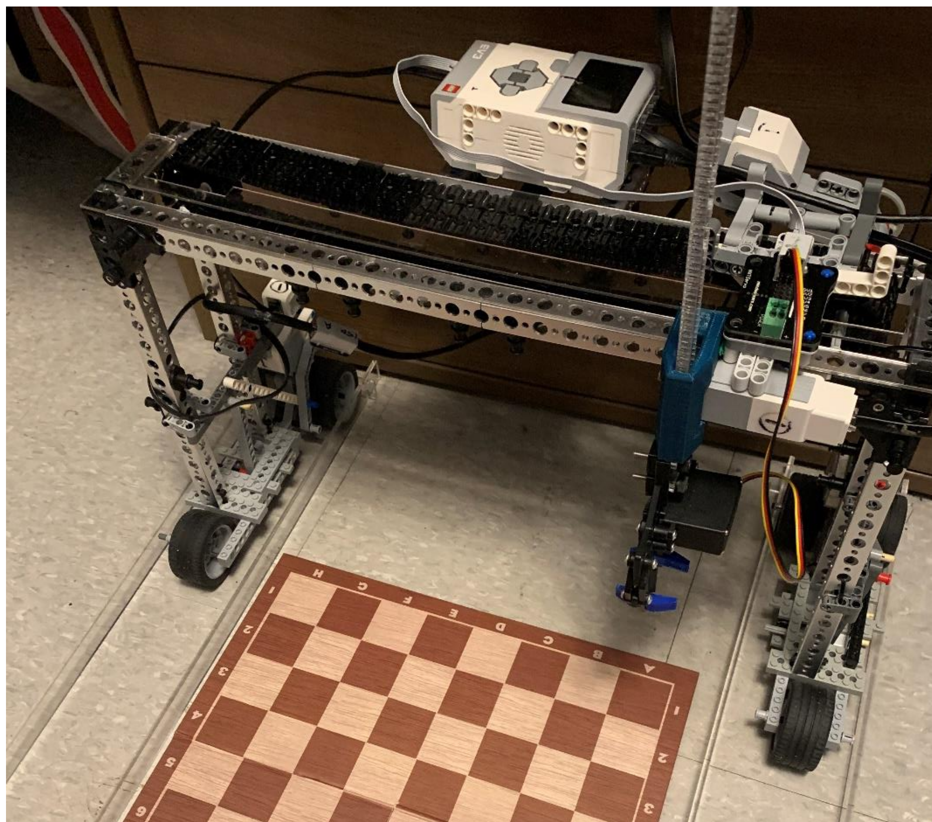

*Figure 4 General mechanical layout of the robot*

## 3.2 Mechanical Functions

All mechanical functions of the robot come to serve the purpose of successfully moving the chess pieces across and outside the board. However, this can be split into sub functional components. Firstly, a functional base structure had to be created to house the components for each of the other functions. The final design had to support the weight of all components as well as not tip over and is shown in Figure 5. The final design is a simple four-legged structure with an extension to house the EV3 brick. The EV3 brick can be placed into this extension and then zip-tied down for extra support. The change in centre of mass caused by this extension made the robot prone to tipping. However, the weight balanced out after placing the other components including the gripper and battery on the opposite side.

*Figure 5 Base robot structure*

Another main subfunction is picking and lifting the pieces up and down. It was realized that the gripper itself needed modifications to fit the shape and size of the chess pieces. Gripper attachment models were then designed and 3D printed to create a functional gripper. The vertical motion is operated by a pinion attached to a motor placed within a housing system which was also 3D modelled and printed and allows for the rack & pinion to function smoothly. The rack also required a specific design to fit the spacings along the pinion. Hence, a 1-foot rack was designed and laser cut using 4.5mm thick acrylic, allowing for the vertical function of the robot All these mechanical design decisions can be seen in Figure 6a).

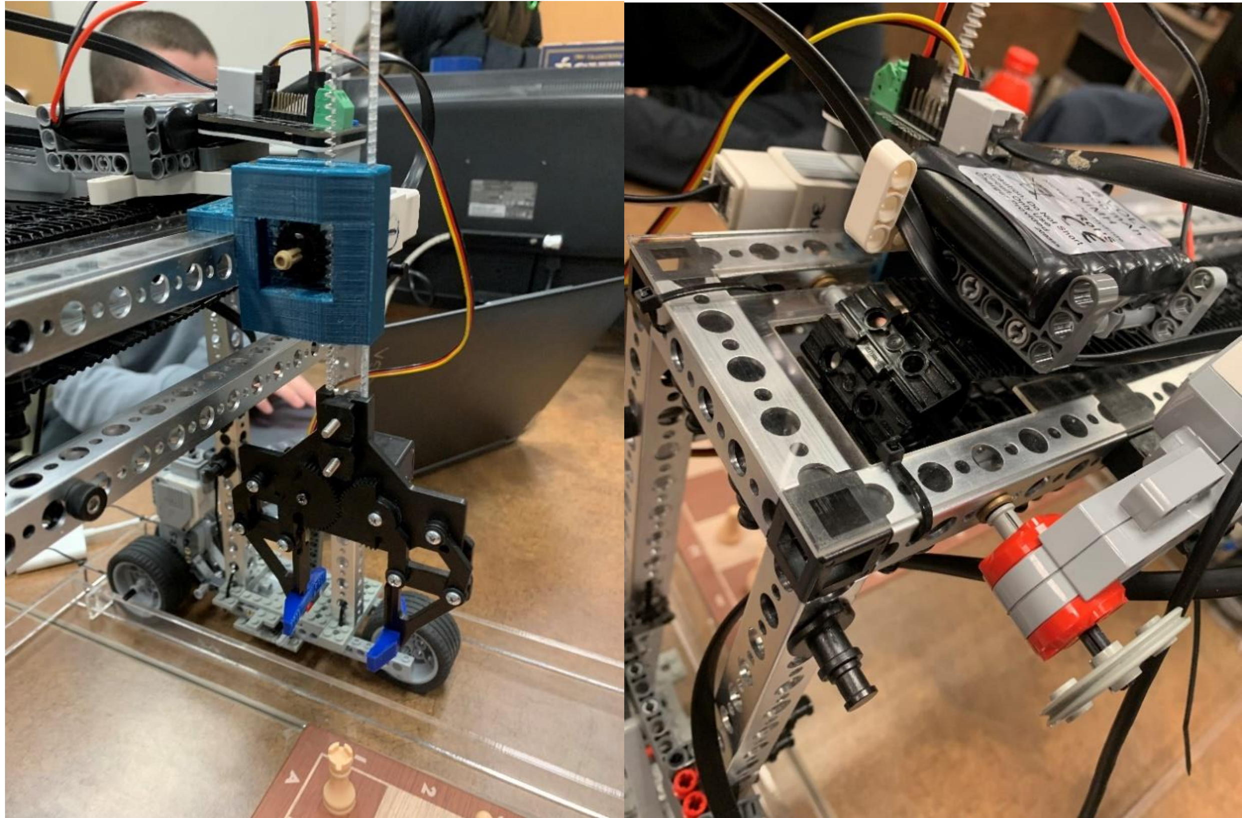

*Figure 6 Motor, rack and pinion housing mechanism, b) Belt-housing movement mechanism*

The second mechanical function operates the horizontal axis, moving the rack housing system using a belt attachment across an acrylic cover. The belt is placed on an axle through the Tetrix beams and powered by a large motor. Brass protectors were used to protect the plastic Lego axle from the harsh Tetrix metal as well as reduce frictional torque. Upon preliminary testing, it was found that the belt had difficulty in maneuvering the rack-housing system across the horizontal due to the sliding interface between the belt and the Tetrix frame being extremely uneven. To ameliorate this function, a sliding cover fit for the dimensions of the outer frame was CAD designed and laser cut in acrylic. This reduced the friction between the interfaces and increased the operation efficiency of the rotating belt function. Finally, due to the short nature of the Tetrix gripper and battery wires, a battery and mux structure was made using Lego which moves along with the belt and gripper. These mentioned design decisions can be seen in Figure 6b).

The third mechanical function serves the purpose of giving the robot board coverage across the vertical axis. To achieve this, the supporting rail attached to the entire frame of the operating system is placed on a 6-wheel movement system operated by motors. To increase structural

integrity of the frame-vehicle attachments, the system was heavily zip-tied and reinforced with interlocking pieces. Along with this, to ensure the vehicles do not slide out of their designated paths overtime, tracks were designed, and laser cut to ensure the wheels operate within a tolerance range. The wheel system is shown in Figure 7 (Note: the rails are not shown in Figure 7 but can be seen in Figure 4).

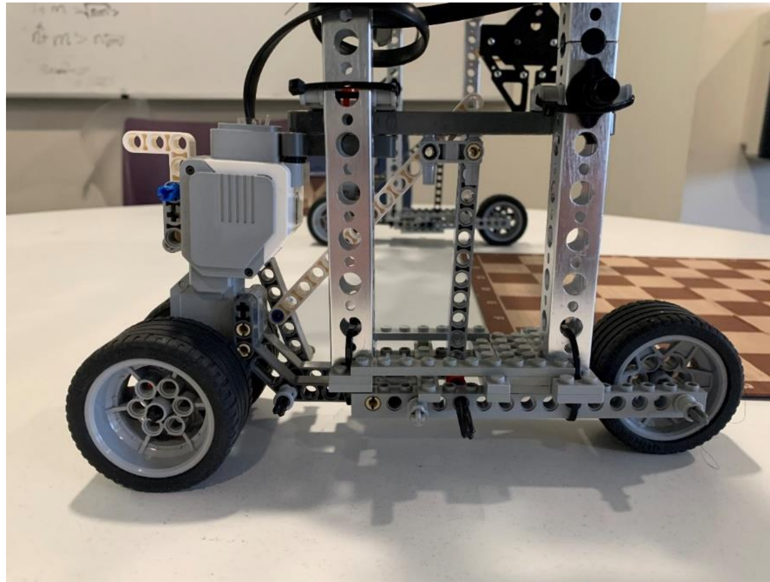

*Figure 7 Wheel movement system*

## 3.3 Verification of Design

In order to verify that the final design is suitable, a line-by-line verification of the engineering specifications (from Section 2.1) was done through a series of tests and analyses. A spreadsheet of group spending was kept throughout the entire process (Figure 8). After analyzing it appears that the group stayed within the budget of $25/person with about $10/person left in budget. Maintaining a low cost was a success, as the cost of the entire robot was well below the budget. This is mostly due to the fact that most parts were rented, and customised parts were only made when necessary.

| Jonathan | $9.02 | | Daniel | $0.00 |
| --- | --- | --- | --- | --- |
| Item | Cost | | Item | Cost |
| Gripper print | $1.02 | | | |
| Rack motor housing | $1.50 | | | |
| Racks | $0 | | Yulai | 24 |
| Rack motor housing 2 | $6.50 | | Item | Cost |
| | | | Acrylic sheet | $15.00 |
| Abbas | $24.03 | | Rack housing | $9.00 |
| Item | Cost | | | |
| Chess set | $15.00 | | Total | Budget |
| Zip ties | $9.03 | | $57.05 | $100 |

*Figure 8 Spending breakdown*

The visual coverage available to players during a game also met requirements, with the human player, any spectators, and the webcam used for the computer vision program all being able to clearly see the board during the game. During test games, the move times for various move types were recorded to see if they met requirements. For a normal passive move where only a single piece is moved, the robot typically took 20 seconds or less to move. Three of those seconds were allocated to thinking time for the chess engine and could be reduced if needed. Naturally, moves that involve capturing or castling took slightly longer but remained well under the time limit of one minute. The success rate in moving pieces was also measured during the test games. Any move where the robot failed to pick up a piece, moves to the wrong square, or knocked a piece down was a failure, and only the first 150 moves from the test games were used. In those 150 moves only 7 were failures, meaning the success rate was 95.33% and met requirements. It is worth noting that most of the failures occurred at later stages of games, where the robot lost accuracy and pieces were not centered in their squares. Communication with the chess engine worked flawlessly as long as a proper human move was obtained, meaning the computer vision was successful and the human made a legal move. The ease of use was measured by asking the players who tested the robot to rate it on a scale from 1-100% and taking the average rating. 10 players were asked, and the average rating was 95%. The players played chess with the robot in the same way they would play with any human, and the only extra task they had was to press a button on the laptop to take a picture and allow the robot to begin the process of making a move, making it very simple for the user. The final aspect tested was safety, which involved ensuring that the robot did not have any parts that could pinch or cut players, and it was clear that the robot met this requirement. Through the verification process, it was determined that all requirements were met, and the robot's performance surpassed expectations in many ways.

# 4.0 Software Design

This section outlines the software design process for the EV3 Chess Robot and covers the RobotC portion of the project. For more information regarding the Python design process, refer to Appendix B.

## 4.1 Description

The first step taken towards the software design process was creating a list of tasks the robot must perform along with a list of assumptions regarding the robot's operating environment. The final task list decided on by the group was as follows:

- Move x and y simultaneously,
- Move z,
- Pick up piece,
- Drop piece,
- Ability to resign,
- Read instructions from files sent by the computer,
- Move according to instructions,
- Allow manual calibration with buttons,
- Display final game result and end,
- Display total elapsed time,
- Gets out of the way of opponent during their move,

Along with this task list came the assumptions necessary for programming the robot. Assumptions are important as they simplify code, reducing the time spent both coding and computing. They also explain to others the way in which the code is expected to work in real life. The assumptions for this project are as follows:

- The robot's chess engine, 'Stockfish', will never lose to a human [2],
- The user follows all screen prompts properly,
- The robot accumulates no rotational displacement,
- All pieces are placed in the center of their respective squares,
- The opposing player plays only legal moves,
- After the robot has moved, all previous moves are final,
- One game will not last longer than 59 minutes 59 seconds,

- All basic rules of chess must be followed

From these two lists, a high-level flowchart outlining the entire program structure was created for main (Figure 9). The flowchart was extremely helpful in communicating the software design process to the entire group, and it set up a standard for which all the functions would be written.

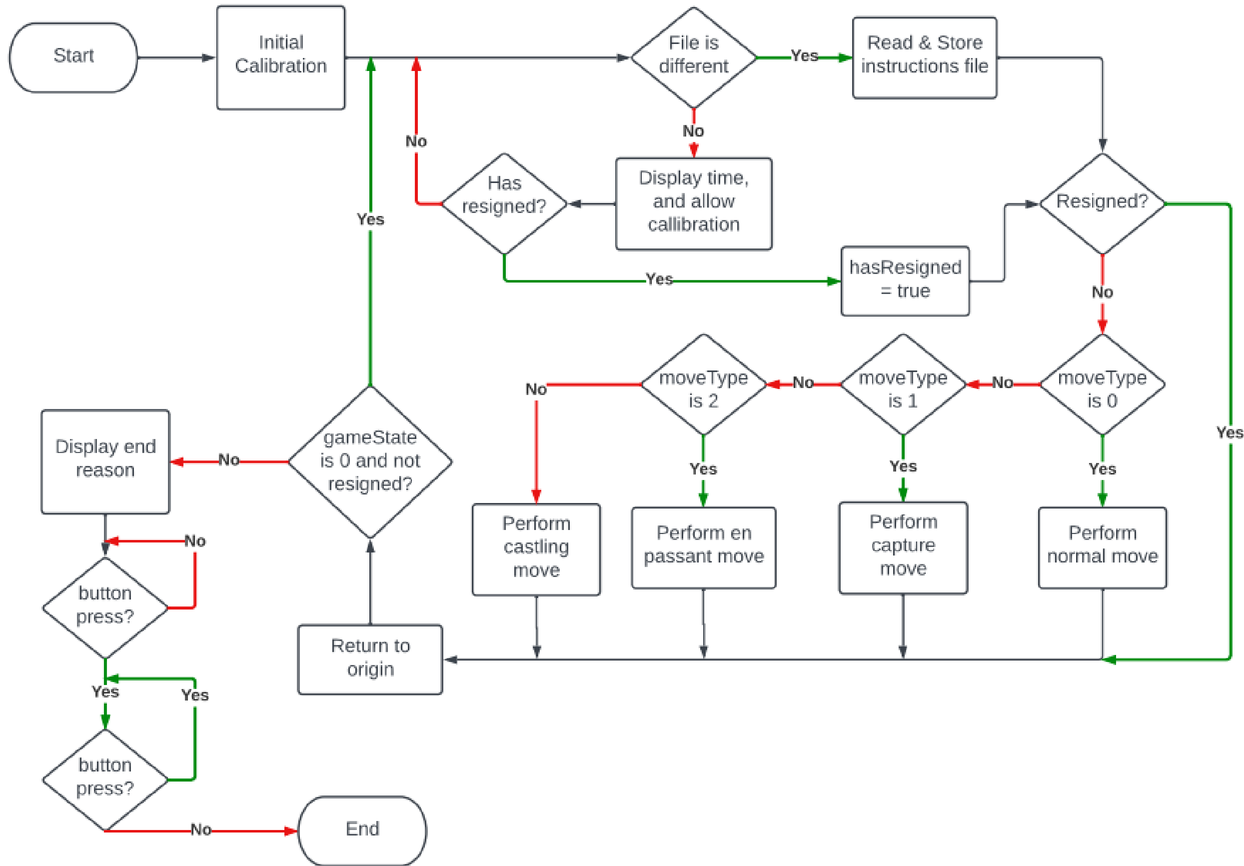


*Figure 9 High-level software flowchart*

However, these program descriptions only cover a small portion of the programming work needed to complete this project. Most of the programming work went into the code outside of the EV3 brick. Since it was too hard to find or create a chess engine and image recognition program that could work in RobotC, the group had to settle for code written in Python to supplement the RobotC program through file communication. The general process of this external code is to detect the opponents move using OpenCV and NumPy, then use an open-source chess engine known as Stockfish to output the next best move for the robot, then use mouse and keyboard manipulation from PyAutoGUI to automatically upload an instructions file to the EV3 robot [1, 3, 4]. The instructions file contains the move given by Stockfish, an integer representing the move type,

optional additional move coordinates for en passant and castling moves, and another integer representing whether the game is over or continuing. This report will not go much further in depth on this code since this is out of the scope of both this course and this project. However, there is more information on this external code found in Appendix B.

## 4.2 Functions and Data Storage

Before proceeding with the programming, the group broke the program down into a list of function prototypes with descriptions, as shown in Table 4. This made it easy to divide functions between one another and allow options to choose from. These functions ended up becoming the functions present in the final source code (Appendix A).

*Table 4: Function Descriptions*

| Function Prototype | Description | Written By: |
|---|---|---|
| void moveXY (int collumn, int row) | Uses the encoders of motors A, B and C to facilitate the movement of the gripper arm in the plane parallel to the ground. A and B encoders were used separately in the column direction to reduce the amount of rotation error. Encoder C was used for row direction movement. All three motors decreased in power following a linear equation as the encoder reached the desired position. | Jonathan DiGiorgio |
| void moveZ (int dir) | Takes a direction (1 for down, -1 for up) and then moves the rack and pinion using motor D up to the specified constant vertical distance. | Jonathan DiGiorgio |
| void liftPiece() | Uses the tetrix gripper servo functions along with the moveZ function to perform the act of lifting a piece. | Abbas Asghar |
| void dropPiece() | Uses the tetrix gripper servo functions along with the moveZ function to perform the act of dropping a piece. | Abbas Asghar |
| void movePiece (int* moveCoord) | Uses the move coordinate and the movement functions above to perform a normal chess move. | Daniel Martire |
| void moveCapture (int* moveCoord, int & captureCount) | Uses the move coordinate and the movement functions above to perform a capture chess move. This function keeps count of the number of captures so it does not place captured pieces on the same square off the board. | Daniel Martire |
| void moveEnPassant (int* moveCoord, int* enpasCoord, int & captureCount) | Uses the move coordinate, special move coordinate and the movement functions above to perform an en-passant chess move. This function keeps count of the number of captures so it does not place captured pieces on the same square off the board. | Jonathan DiGiorgio |

| void moveCastle (int* moveCoord, int* casCoord) | Uses the move coordinate, castle coordinate and the movement functions above to perform a castling move. | Yulai Duan |
|---|---|---|
| void displayTime() | When called, it will take the total elapsed time and format it into standard 00:00 style, then display it on the EV3 screen in big font. | Yulai Duan |
| void allowCalibration() | Must be used within a while loop. Detects button presses on the EV3. Each button corresponds to positive or negative X or Y motion for manual calibration when needed. | Jonathan DiGiorgio |
| Void readCoord (TFileHandle & fin, int* coord) | Uses the file IO library to read a coordinate within the instructions file. Firstly it reads in 4 chars, then converts them to the standardized coordinate system for this program. This is done by subtracting the int value of chars 'a' or '1' depending on if a letter or number is read. | Jonathan DiGiorgio |
| bool readCoordIsDifferent (int* coord) | Uses the readCoord function to read coordinates from the file and detects if a new coordinate (and therefore file) has been uploaded. Returns true if it is different and false otherwise. | Jonathan DiGiorgio |
| void readInstructions (int* moveCoord, int &moveType, int* specialMoveCoord, int &gameState, bool &hasResigned) | Starts by waiting until a new file is uploaded using the readCoordIsDifferent function. While the robot waits for a new file it is possible to calibrate the robot or click the resign button to resign. When a new file is uploaded it stores the 2 coordinates, move type and game state. | Jonathan DiGiorgio |
| void displayGameEnd (int gameState) | Takes in the game state and displays the game ending message based on which type of game state it is. | Yulai Duan |
| task main() | Integrates all the functions. Allow for calibration, waits for and reads a newly uploaded file, performs move based on the move type, loops if the game has not ended and player has not resigned, and displays ending message once it has ended. | Jonathan DiGiorgio |

Data storage within the program was decided as a group so that each function could be created with the same underlying data structures. It was decided that all the needed information to play a game of chess could be stored within two 1D arrays one bool and three integer variables, excluding the occasional temporary variables needed in certain functions. The two arrays store coordinates for the move. Two of the integers store pre-determined values for move types and game states. This data storage model is depicted in Figure 10, along with how the instructions from the file will be stored.

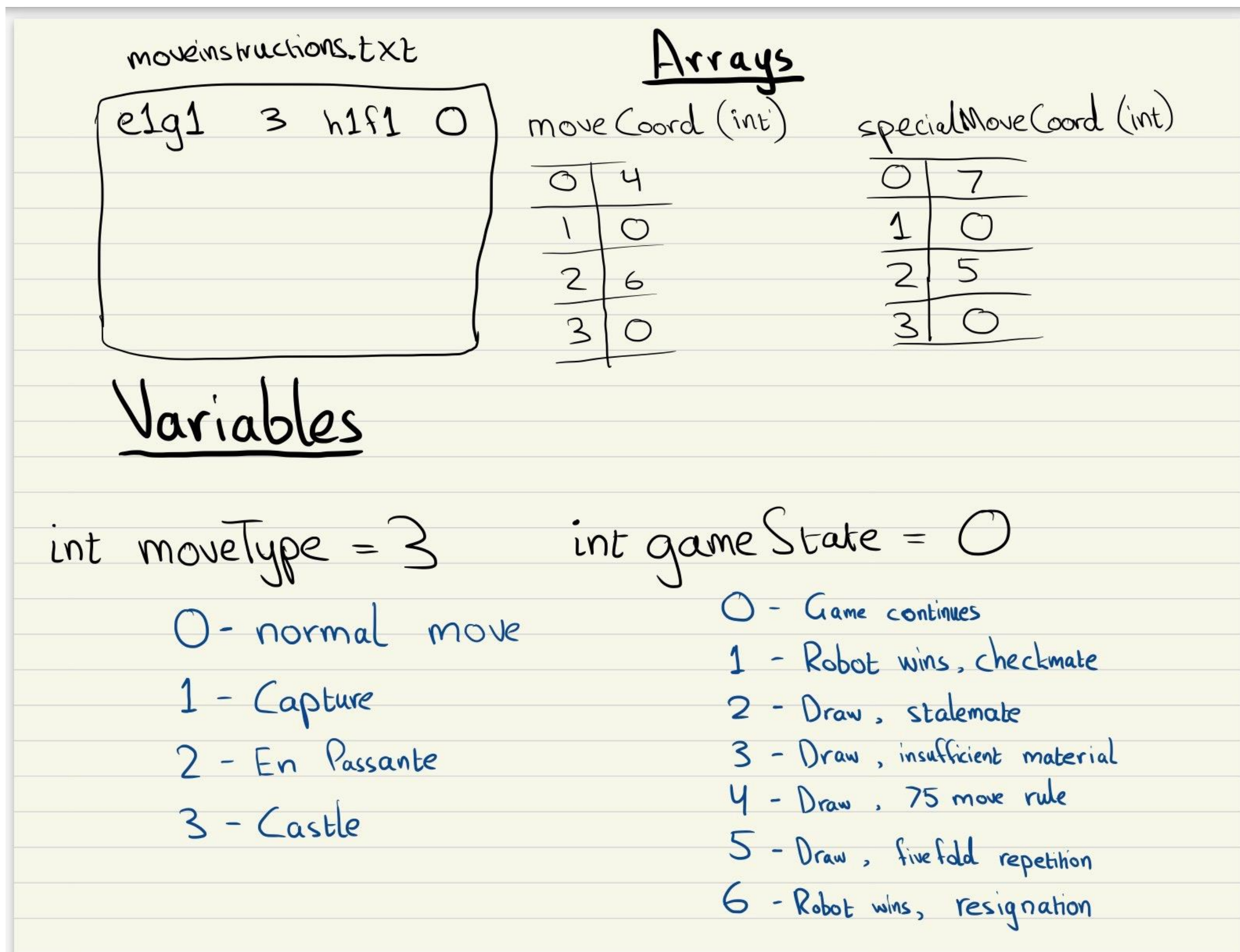


*Figure 10 Data Structure Plan and Integer Value Meanings*

The other integer being the capture count, which simply ticks up after any type of capturing move is performed by the robot. The Boolean is used to store whether the opponent has resigned.

## 4.3 Software Testing

Software testing was an extremely important part of the design process, as small errors could result in detrimental results in a very precise game like chess. Thankfully a lot of the functions were simple void movement functions which were easily tested. For example, the first 8 functions in Table 4 were tested by simply calling them with desired testing parameters/coordinates and then observing the resulting movements. Capture moves were checked to make sure captured pieces were not placed on top of each other. The functions which simply display something were easily tested by calling it with test parameters and checking to make sure the correct thing is

displayed. For the time function it was as simple as watching it count up to a minute and making sure it was well formatted the entire time. The calibration function was tested simply by pressing the four possible buttons and making sure the motors moved respectively.

The functions involved with file input required heavier testing than the above functions. This involved uploading many test instruction files and analyzing what the EV3 read from them. This was done by printing debug strings on the screen which display what the function has read as well as adding button presses waits to break code into sections and look for errors. Many different variations of instruction files were tested to ensure that the file input was always correct.

It must also be noted that once a semi-functional system was developed, each of the functions were tested by simply playing games against the robot and noting any recurring issues or bugs. Using all these test methods resulted in a program which is reliable and consistent.

## 4.4 Significant & remaining challenges

One of the largest challenges in writing the RobotC code was handling file inputs. This is due to the fact that file inputs are handled through a library which was not taught and is not a standard C library. It took some trial and error to understand how the library and its functions work. Some bugs with file input were often encountered when reading files in a loop too fast. On some occasions the program would read in incorrect data from a file with no clear explanation as to why after many hours of debugging. Many solutions had to be implemented to handle these file input bugs and create a reliable program

Although the robot was functional and met all the desired goals, there are still certain areas in which it is slightly inconsistent and requires improvement. The most noticeable issue that it had was the decline in accuracy over the duration of the game. As more moves were made, the robot became less accurate in moving the claw to the centre of a square. It always went to the correct square, but the claw would not be perfectly centred at later stages of the game. This led to issues with picking up the pieces on rare occasions where the decrease in accuracy was more significant. More importantly, the lack of centred pieces would eventually affect the computer vision Python code, causing it to detect the wrong move which would then crash the code. To help alleviate this issue, 3D-printed rails were added to help guide the wheels and ensure they remained straight. This was effective, but the issue was still present to a lesser degree, warranting, additional

improvements in all components of the code. In the Python code, a feature could be added to ask for manual move input if there was an error to avoid crashing. In the RobotC program, more precise tuning of constants and a more accurate way of recalibrating the robot's position mid-game would be warranted. The constants the represent the distance of one square relative to rotations made by the motors could be more precise to reduce the accumulated error in the robot's movement. This could be done through more rigorous measurements and testing. For the recalibration, an additional function could be made to recalibrate the robot after a certain amount of turns or after a certain amount of error has been detected.

Another issue with the robot was its inability to handle promotions. If a pawn was promoted, it would be assumed that it is promoted to a queen, but the robot would continue moving the pawn as a queen instead of replacing it with the proper queen piece. Adding an option for the player to select what type of piece to promote to would be relatively simple to implement. To handle swapping the pawn with the correct piece type, a row of extra pieces at predefined locations could be added to the sides of the board. This would be similar to how the robot handles removing captured pieces and it would allow the robot to easily remove the pawn off the board and replace it with the correct piece.

# 5.0 Project Management

To assist the completion of the robot before the deadline, the main tasks were split up into subtasks and designated to different team members. As the project moved forward, some new tasks were introduced and changes from the initial project schedule had to be made to facilitate the completion of the new tasks.

## 5.1 Work Breakdown Structure

The work breakdown structure in Table 5 displays how tasks and subtasks were divided and how time was designated to each task. The frame building was estimated to take 7 hours to complete. The entire team was assigned to work together on building the Tetrix frame, which included the belt housing for x-motion and the wheel base for y-motion. Yulai was instructed to design the rail for the wheels and Abbas was instructed to design the cover for the belt. The EV3 brick connection was assembled by the entire team, which was just a Tetrix extension to the belt housing.

*Table 5: Work Breakdown Structure*

| Task | | | | | |
|---|---|---|---|---|---|
| **Project** | | | 96h | | |
| | 1.0 Frame Building | | | 7h | |
| | | 1.1 Tetrix frame | | | 4h |
| | | 1.2 Design and Laser cut rails | | | 2h |
| | | 1.3 EV3 brick placement | | | 1h |
| | 2.0 Communicating with Chess Engine and OpenCV | | | 45h | |
| | | 2.1 Data between Stockifsh and EV3 | | | 10h |
| | | 2.2 Modifying stockfish I/O | | | 10h |
| | | 2.3 Formating engine output into EV3 Commands | | | 5h |
| | | 2.4 OpenCV Move Recognition | | | 20h |
| | 3.0 Gripper | | | 14h | |
| | | 3.1 Assemble and measure gripper | | | 1h |
| | | 3.2 Design and laser cut racks | | | 5h |
| | | 3.3 Design and 3D print modifications and housing | | | 8h |
| | 4.0 Gripper Movement | | | 10h | |
| | | 4.1 X-Movement (Wheels) | | | 3h |
| | | 4.2 Y-Movement (Belt) | | | 2h |
| | | 4.3 Movement Functions | | | 5h |
| | 5.0 Final Program | | | 20h | |
| | | 5.1 Instructions file handling | | | 5h |
| | | 5.2 Miscellaneous and main functions | | | 10h |
| | | 5.3 RobotC-Python Integration | | | 5h |

Getting the robot to communicate with a chess engine and integrating OpenCV for move detection was estimated to take 45 hours. Jonathan was assigned the task of getting the robot to communicate with Stockfish. It was estimated to take him 25 hours to get Stockfish to generate moves and output them to the robot. Daniel was designated to get move detection to work using OpenCV. Getting the move detection to work took Daniel about 20 hours.

Designing the gripper was estimated to take 14 hours. Jonathan was assigned to make 3D attachments to help the gripper pick the pieces. Jonathan also designed the 3D housing for gripper motor. Coding the gripper movement on RobotC was estimated to take 10 hours. The functions were split up between all team members. After the movement functions were done, the final program had to be written using movements functions and file handling from Stockfish output. The final program was estimated to take 20 hours to complete, with everyone contributing. Jonathan worked on the RobotC-Python integration since he had control over the Python code.

## 5.2 Project Scheduling

Chess Robot

SIMPLE GANTT CHART by Vertex42.com
https://www.vertex42.com/ExcelTemplates/simple-gantt-chart.html

Project Start: Sun, 4-2-2023
Display Week: -6

| TASK | ASSIGNED TO | PROGRESS | START | END |
|---|---|---|---|---|
| **Frame Building** | | | | |
| Tetrix frame | | | 2-17-23 | 2-17-23 |
| Design and Laser cut rails | | | 2-17-23 | 2-19-23 |
| EV3 Brick Connection | | | 2-17-23 | 2-17-23 |
| **Communicating with Chess Engine** | | | | |
| Data between Stockifsh and EV3 | | 30% | 2-11-23 | 2-28-23 |
| Modifying Stockfish I/O | | 40% | 2-19-23 | 2-28-23 |
| Formating engine output into EV3 Commands | | | 3-1-23 | 3-20-23 |
| **Gripper** | | | | |
| Assemble and measure Gripper | | | 2-17-23 | 2-17-23 |
| Design and 3D print modifications | | | 2-18-23 | 2-22-23 |
| Design and code vertical motion | | | 3-6-23 | 3-24-23 |
| **Gripper movement** | | | | |
| X-movement (wheels) | | | 2-27-23 | 3-13-23 |
| Y-movement (Belt) | | | 3-6-23 | 3-24-23 |
| Movement & Board mapping code | | | 2-27-23 | 3-24-23 |

*Insert new rows ABOVE this one*

Feb 13, 2023 | Feb 20, 2023 | Feb 27, 2023 | Mar 6, 2023 | Mar 13, 2023 | Mar 20, 2023

*Figure 11 Gantt Chart for initial project schedule*

The initial project schedule in Figure 10 underestimated the time it would take to complete all the tasks and subtasks. At this point, it had not been decided to use computer vision to detect opponent's move, and the human move was going to be manually inputted instead. However, a more automatic approach to the design with minimal manual input was desired, so the computer vision input was favored. Thus, Daniel started working on computer vision. Since this was out of scope for this project, the Python code was designed to allow for manual move input in case computer vision code detection did not work.

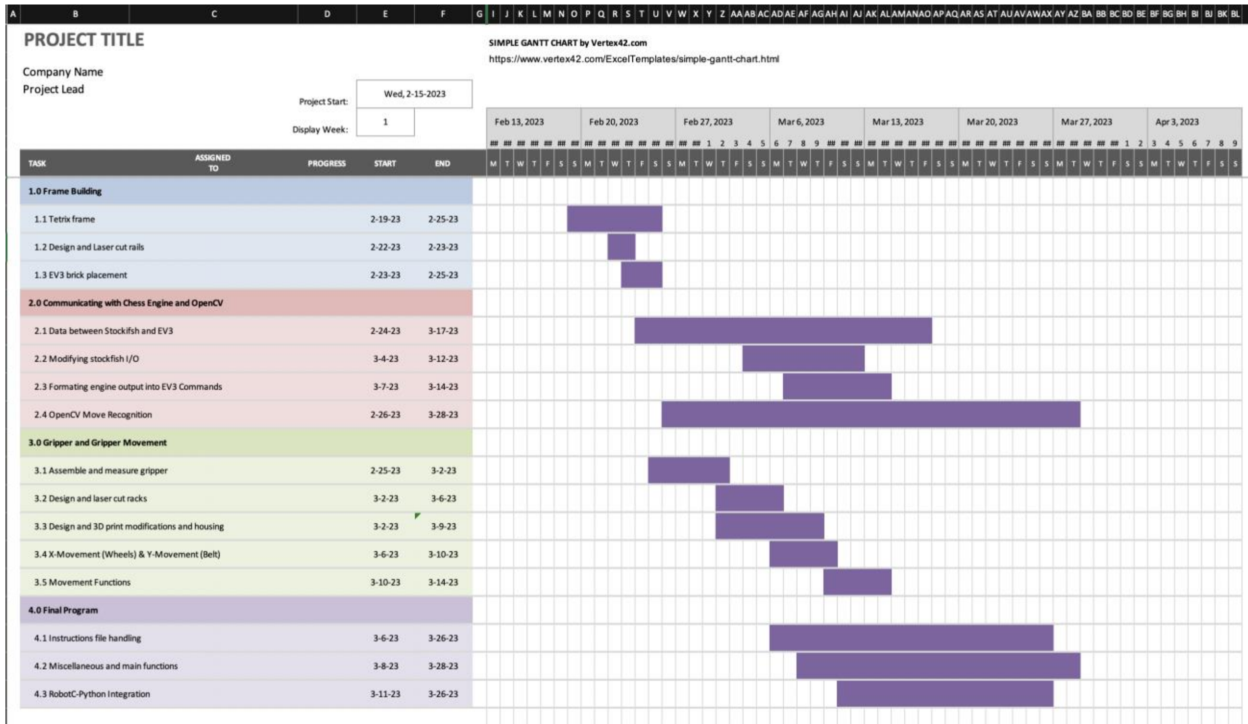


*Figure 12 Gantt Chart for actual Project schedule*

The initial project schedule displayed in Figure 11 shows how long the subtasks took to complete. Communicating with the chess engine and created the computer vision code took most of the project time. Since the group worked together on most of the mechanical aspects of the robot, those tasks were completed faster than the coding parts. A lot of time towards the end of the project was spent debugging the main RobotC program and having it communicate with the Python code.

# 6.0 Conclusions

## 6.1 Conclusions

Overall, the design for the robot was successful. It was easy to use and accessible, made moves in a short period of time, and effectively communicated with a chess engine so that it could play a game of chess against any opponent. After many conceptual designs and prototypes, the final mechanical design involves triple access movement where each direction used its own set of motors and a motor encoder. The x movement used wheels, the y movement used a belt and rail system, and the z movement used a rack and gears. The software component operated in two parts, with the first operating on a laptop and the second operating on the Lego EV3 brick. The laptop code involved detecting the human move through computer vision, getting a response move from stockfish, and sending the move on a file to the robot. The EV3 code involved evaluating that move to determine what squares the claw had to move to, and how much the motors had to turn to move that number of squares. Everything functioned as intended and all requirements were met. The only deficiencies of the robot were the decrease in accuracy over time due to a buildup of slight errors in physical movement, and the failure of the computer vision code that resulted from that inaccuracy.

## 6.2 Next Steps / Recommendations

The robot is far from perfect, and there are several things that the group recommends to anybody attempting to expand on this design. The wheels should be replaced with a movement system which is more precise in its movement, for example a gear or thin wheel which can slot into a laser cut groove that only allows straight motion. A large issue with normal wheels is the tendency to rotate over time due to very small imperfections in alignment. Another solution to this problem could be to have the robot stationary in the y direction and have the board itself move in the y direction.

Another recommendation is to design gripper attachment or even a new gripper such that it does not topple over other pieces. Occasionally the robot will knock over adjacent pieces due to the wide nature of the Tetrix gripper. Fixing this problem would lead to a much more enjoyable playing experience.

A final recommendation is to research ways to increase precision and reliability of both the robot and code. Occasionally the python image recognition code crashes due to inconsistencies in lighting and piece placements, and the robot tends to make imprecise moves every now and then. Fixing these problems through research and testing would be optimal for a more enjoyable playing experience.

# Appendix A – EV3 Source Code

LINK TO SOURCE CODE: [https://github.com/jddigior/EV3-Stockfish-Chess-Robot/blob/main/FinalDraft.c](https://github.com/jddigior/EV3-Stockfish-Chess-Robot/blob/main/FinalDraft.c)

```
#include "PC_FileIO.c"
#include "EV3Servo-lib-UW.c"

//All in mm
const float Z_DIST = 80;    //Distance gripper has to move downwards
const float SQUARE_DIST = 38;
const float GEAR_RAD = 7.0;
const float BELT_RAD = 19.1;
const float WHEEL_RAD = 27.7;

//Miscellaneous constants
const int COORD_SIZE = 4;
const int CAPTURE_ROW = 9;

//Motor Powers
const int X_POWER = 20;
const int Y_POWER = 20;
const int Z_POWER = 20;
const int APPROACH_POWER = 8;

//Moves the wheels and belt simultaneously
void moveXY(int collumn, int row)
{
    //Calculate the desired angles of rotation
    float initEncRow = nMotorEncoder[motorA];
    float X_ANGLE = row * SQUARE_DIST * 180/(WHEEL_RAD*PI) - initEncRow;
    float initEncCol = nMotorEncoder[motorC];
    float Y_ANGLE = collumn * SQUARE_DIST * 180/(BELT_RAD*PI) - abs(initEncCol);

    //Control each motor power with its respective encoder
    //We acknowledge that A and B could be done with just one encoder,
    //  however this resulted in a noticable decrease in driving straightness
    while (abs(nMotorEncoder[motorA] - initEncRow) < abs(X_ANGLE)
            || abs(nMotorEncoder[motorB] - initEncRow) < abs(X_ANGLE)
            || abs(nMotorEncoder[motorC] - initEncCol) < abs(Y_ANGLE))
    {
        if (abs(nMotorEncoder[motorA] - initEncRow) < abs(X_ANGLE))
        {

            //motors are updated with decreasing motor powers
```

```
            //  based on how close they are to the target
            int powA = X_POWER - (X_POWER - APPROACH_POWER)
                     * abs((nMotorEncoder[motorA] - initEncRow)/(X_ANGLE));
            if (X_ANGLE > 0)
                motor[motorA] = powA;
            else
                motor[motorA] = -powA;
        }
        else
            motor[motorA] = 0;

        if (abs(nMotorEncoder[motorB] - initEncRow) < abs(X_ANGLE))
        {
            int powB = X_POWER - (X_POWER - APPROACH_POWER)
                     * abs((nMotorEncoder[motorA] - initEncRow)/(X_ANGLE));
            if (X_ANGLE > 0)
                motor[motorB] = powB;
            else
                motor[motorB] = -powB;
        }
        else
            motor[motorB] = 0;

        if (abs(nMotorEncoder[motorC] - initEncCol) < abs(Y_ANGLE))
        {
            int powY = Y_POWER - (Y_POWER - APPROACH_POWER)
                     * abs((nMotorEncoder[motorC] - initEncCol)/(Y_ANGLE));
            if (Y_ANGLE > 0)
                motor[motorC] = -powY;
            else
                motor[motorC] = powY;
        }
        else
            motor[motorC] = 0;

    }
    motor[motorA] = motor[motorB] = motor[motorC] = 0;
}

// moves therack and pinion up or down
// 1 for down, -1 for up
void moveZ(int dir)
{
    float initEnc = nMotorEncoder[motorD];
```

```
    const float Z_ANGLE = Z_DIST * 180/(GEAR_RAD*PI);

    while (abs(nMotorEncoder[motorD] - initEnc) < Z_ANGLE)
    {
        int pow = Z_POWER - (Z_POWER - APPROACH_POWER)
                * abs((nMotorEncoder[motorD] - initEnc)/(Z_ANGLE));
        if (dir == 1)
        {
            motor[motorD] = pow;
        }
        else
        {
            motor[motorD] = -2*pow; //account for gravity working against rack
        }

    }

    motor[motorD] = 0;
}

//Lifts the piece directly under the gripper
void liftPiece()
{

    setGripperPosition(S1, 1, 40);
    moveZ(1);
    wait1Msec(100);

    setGripperPosition(S1, 1, 0);
    wait1Msec(500);
    moveZ(-1);
}

//Drops the piece to the square directly under the gripper
void dropPiece()
{
    moveZ(1);
    wait1Msec(100);

    setGripperPosition(S1, 1, 40);
    wait1Msec(100);
    moveZ(-1);
}

//Moves piece from one square to another
```

```
void movePiece(int* moveCoord)
{
    moveXY(moveCoord[0], moveCoord[1]);
    liftPiece();
    moveXY(moveCoord[2], moveCoord[3]);
    dropPiece();
}

//Captures piece then moves
void moveCapture(int* moveCoord, int & captureCount)
{
    // Uses int division and modulus to ensure captured pieces arent placed in
the same location twice
    int capCoord[COORD_SIZE] = {moveCoord[2], moveCoord[3],
                captureCount % 8, CAPTURE_ROW + captureCount / 8};

    movePiece(capCoord);
    movePiece(moveCoord);
    captureCount++;
}

//Performs en passant capture
void moveEnPassant(int* moveCoord, int* enpasCoord, int & captureCount)
{
    int capCoord[COORD_SIZE] = {enpasCoord[0], enpasCoord[1],
                captureCount % 8, CAPTURE_ROW + captureCount / 8};

    movePiece(capCoord);
    movePiece(moveCoord);
    captureCount++;
}

//Performs castling move
void moveCastle(int* moveCoord, int* casCoord)
{
    movePiece(casCoord);
    movePiece(moveCoord);
}

//displays current time elapsed since start of game
void displayTime()
{
    displayBigTextLine(13, "Time %02i:%02i", time1[T1]/1000/60,
                            (time1[T1]/1000) % 60);
}
```

```
/*
    Padding format taken from:
    https://www.includehelp.com/c-programs
    /input-an-integer-value-and-print-with-padding-by-zeros.aspx
*/


//Must be used in a fast loop
//Allows the player to manually callibrate the gripper arm to A1 square
void allowCalibration()
{
    if(getButtonPress(buttonUp))
    {
        motor[motorC] = APPROACH_POWER;
        nMotorEncoder[motorC] = 0;
    }
    else if(getButtonPress(buttonDown))
    {
        motor[motorC] = -APPROACH_POWER;
        nMotorEncoder[motorC] = 0;
    }
    else if(getButtonPress(buttonLeft))
    {
        motor[motorA] = motor[motorB] = -APPROACH_POWER;
        nMotorEncoder[motorA] = nMotorEncoder[motorB] = 0;
    }
    else if(getButtonPress(buttonRight))
    {
        motor[motorA] = motor[motorB] = APPROACH_POWER;
        nMotorEncoder[motorA] = nMotorEncoder[motorB] = 0;
    }
    else
        motor[motorA] = motor[motorB] = motor[motorC] = 0;
}

//parses instructions file into number arrays to be read later
void readCoord(TFileHandle & fin, int* coord)
{
    char temp = ' ';
    for(int index = 0; index < COORD_SIZE; index++)
    {
        readCharPC(fin, temp);

        //file follows letter-number coordinate convention (i.e. a1a3)
        //This ensures the correct char is subtracted
```

```
        if(index % 2 == 0)
            coord[index] = (int)temp - (int)'a';
        else
            coord[index] = (int)temp - (int)'1';

        /*
        Used to fix an extremely weird bug where after a new file was
        inserted the read characters would be converted into two or three digit
        numbers that were random except for the first digit
        (which was the desired output)
        */
        if (coord[index] > 100)
            coord[index] = coord[index] / 100;
        else if (coord[index] > 10)
            coord[index] = coord[index] / 10;
    }
}
// char to int convertion: https://sentry.io/answers/char-to-int-in-c-and-cpp/


//Checks to see if the current file uploaded is different from the previous
bool readCoordIsDifferent(int* coord)
{
    bool isDifferent = false;
    int lastCoord[4];

    for(int index = 0; index < COORD_SIZE; index++)
    {
        lastCoord[index] = coord[index];
        //Populates the previous coordinates before reading again
    }

    TFileHandle fin;
    bool fileOkay = openReadPC(fin, "MoveInstructions.txt");
    if(!fileOkay)
        displayBigTextLine(5, "File not Opened", 10);

    wait1Msec(50);
    readCoord(fin, coord); //read in the new coord

    closeFilePC(fin);

    //in chess you can never have two consecutive moves starting on the same
    // square therefore only the first two coords need be checked
    if (coord[0] == lastCoord[0] && coord[1] == lastCoord[1])
```

```
        isDifferent = false;
    else
    {
        isDifferent = true;
        /* for debugging
        displayString(1, "%i %i %i %i", coord[0], coord[1],
                                lastCoord[0], lastCoord[1]);
        */

    }

    return isDifferent;
}

//Read the entire instructions file into memory
void readInstructions(int* moveCoord, int &moveType, int* specialMoveCoord,
                      int &gameState, bool &hasResigned)
{

    while(!readCoordIsDifferent(moveCoord))
    {
        displayTime();
        allowCalibration();
        if(SensorValue[S3] == 1)
        {
            while(SensorValue[S3] == 1){}
            hasResigned = true;
            return;
            //Leave the function immediately if a resign is sent
        }
    }
    wait1Msec(50);

    TFileHandle fin;
    bool fileOkay = openReadPC(fin, "MoveInstructions.txt");
    if(!fileOkay)
        displayBigTextLine(5, "File not Opened", 10);

    readCoord(fin, moveCoord);

    readIntPC(fin, moveType);

    //2 and 3 correspond to enpassant and castling moves
    if (moveType == 2 || moveType == 3)
        readCoord(fin, specialMoveCoord);
```

```
    else
    {
        for(int index = 0; index < COORD_SIZE; index++)
        {
            specialMoveCoord[index] = 0;
            //redundant but zeroed out for debugging
        }
    }

    readIntPC(fin, gameState);

    closeFilePC(fin);
}

//Diplays respective message if game has ended
//No win condition has been coded since it is impossible to beat a chess engine
void displayGameEnd(int gameState)
{
        if (gameState == 1)
    {
        displayBigTextLine(1, "Game over -Loss! ");
        displayBigTextLine(3, "Checkmate! ");
    }
    else if (gameState == 2)
    {
        displayBigTextLine(1, "Game over -Draw! ");
        displayBigTextLine(3, "Stalemate! ");
    }
    else if (gameState == 3)
    {
        displayBigTextLine(1, "Game over -Draw! ");
        displayBigTextLine(3, "Insuf Material! ");
    }
    else if (gameState == 4)
    {
        displayBigTextLine(1, "Game over -Draw! ");
        displayBigTextLine(3, "75 Move Rule! ");
    }
    else if (gameState == 5)
    {
        displayBigTextLine(1, "Game over -Draw! ");
        displayBigTextLine(3, "Fivefold Rep! ");
    }
    else
    {
```

```
        displayBigTextLine(1, "Game over -Loss! ");
        displayBigTextLine(3, "Resignation! ");
    }
    displayTime();
}

task main()
{
    int moveCoord[COORD_SIZE] = {0,0,0,0};
    int specialMoveCoord[COORD_SIZE] = {0,0,0,0};

    int captureCount = 0;
    int moveType = 0;
    int gameState = 0;
    bool hasResigned = false;

    //Initial calibration steps
    //Moves gripper down to align with A1 square
    moveZ(1);
    displayTextLine(5, "Place A 1 under arm ");
    displayTextLine(6, "Press enter to continue... ");
    while(!getButtonPress(buttonEnter)){
        allowCalibration();
    }
    while(getButtonPress(buttonEnter)){}

    //lift gripper and waits for pieces to be placed
    moveZ(-1);
    displayTextLine(5, "Place down pieces");
    displayTextLine(6, "Press enter to continue... ");

    while(!getButtonPress(buttonEnter)){}
    while(getButtonPress(buttonEnter)){}

    displayTextLine(5, "Play!");
    displayTextLine(6, "  "); // Clear line 6

    time1[T1] = 0;

    do
    {
        displayTime();
        readInstructions(moveCoord, moveType, specialMoveCoord,
                         gameState, hasResigned);
```

```
        //for debugging
        /*
        displayTextLine(9, "%i%i%i%i", moveCoord[0],moveCoord[1],
                        moveCoord[2],moveCoord[3]);
        displayTextLine(10, "%i", moveType);
        displayTextLine(11, "%i%i%i%i", specialMoveCoord[0],specialMoveCoord[1],
                        specialMoveCoord[2],specialMoveCoord[3]);
        displayTextLine(12, "%i", gameState);
        */

        if(!hasResigned)
        {
            if (moveType == 0)
                movePiece(moveCoord);
            else if (moveType == 1)
                moveCapture(moveCoord, captureCount);
            else if (moveType == 2)
                moveEnPassant(moveCoord, specialMoveCoord, captureCount);
            else
                moveCastle(moveCoord, specialMoveCoord);
        }
        moveXY(0,0); //Reset position back to origin

    } while (gameState == 0 && !hasResigned);

    displayGameEnd(gameState);

    displayBigTextLine(5, "Press any button");
    displayBigTextLine(7, "to shut down");
    while(!getButtonPress(buttonAny) && SensorValue[S3] != 1)
    {
        displayTime();
    }
    while(getButtonPress(buttonAny) || SensorValue[S3] == 1){}
}
```

# Appendix B – External Source Code

LINK TO SOURCE CODE: https://github.com/jddigior/EV3-Stockfish-Chess-Robot

The Python code consists of a series of components involving human move detection, getting a response move from a chess engine, and sending that information to the robot. Viewing this code as a black box, the input would be a picture of the board after every move and the output would be a file including the chess engine move and some extra information about that move to assist in instructing the robot's movement.

To determine the move that the human player makes, Python code using the OpenCV and NumPy libraries are used [5] [6]. This process involves using OpenCV to detect the board at the start of the game, using it again after every move to find the board state, and then using NumPy array methods to deduce what the move is.

Before the start of the game, an image of the board is taken. The image has the median blur, bilateral filter, Canny edge detection, and probabilistic Hough lines transformation functions from OpenCV applied to it [7] [8] [9] [10]. This detects all significant lines in the image. These lines will then undergo a series of filters. They are first classified as horizontal or vertical, approximated into a single x or y value, and filtered to remove lines outside of the board. Then the remaining lines are clustered together into 9 horizontal and 9 vertical lines using the OpenCV k-means clustering function [11]. The intersections of all the horizontal and vertical lines are then stored in a sorted 9x9 array.

The piece detection component avoided the typical object detection neural network in favour of a faster and less computationally intensive method. Only the colour and locations of the pieces needed to be detected, as Stockfish was already keeping track of the types of pieces on each square. As such, a neural network to detect the type of piece would have been a waste of time and effort. Instead, two 8x8 arrays describing the board before and after every human move are used for the move detection. Each entry corresponds to a square on the board, and its value would be a 1, -1, or 0 to represent a white piece, a black piece, or no piece. To obtain the array, an image is taken after the human move where one copy has a gaussian blur applied and another has a gaussian blur followed by the Canny edge detection function [8] [9]. The code then goes through each square by slicing the image based on the coordinates gained from the board detection. If there are a certain amount of non-zero values in the square from the edge detected

image, then a piece is present. If there is a piece present, the average pixel value of the corresponding square in the non-edge detected image will be analyzed to determine if the piece is white or black.

To determine the move based on the two board state arrays, they are subtracted from each other to obtain a third array. In this array, the only non-zero values will be the ones that change over the course of the human's move. Based on the values and coordinates of the non-zero entries, the move can be deduced. As an example, if white plays the move e2e4, the e2 square will go from a 1 to a 0 and end up with a value of –1 in the third array, while the e4 square will go from a 0 to a 1, giving it a value of 1. Using a series of if checks, the move can then be deduced and sent to the chess engine. Once a response move is obtained, the board state will be updated using a similar process so that it can act as the initial board state for the next human move.

For the chess engine integration, the Pychess and Stockfish libraries for Python were used [12] [13]. Both libraries keep track of the chess game but serve different functions. After a human move is obtained, the Stockfish chess engine from the Stockfish library will be used to get a move for the robot to make in response. Then the Pychess library will be used to check whether the move type and the game status. The move type is described as a single integer where 0 is a normal move, 1 is a direct capture, 2 is an en passant, and 3 is castling. The game status is another single integer where 0 means that the game can continue, and the non-zero numbers mean that the game has ended and represent the different reasons for the end. Additionally, any extra coordinates necessary for the en passant and castling move types will be determined. The Stockfish move, move type, additional coordinates, and game status are then written onto a file and sent to the robot.

In order to upload files to the EV3 robot a python library known as PyAutoGUI was used [4]. PyAutoGUI allows for easy control of mouse and keyboard through python code. A simple file upload script was created by using the mouse and keyboard to click the download button in the RobotC IDE and then typing in the file path to the instruction text file. This had to be done as there is no way to directly upload it without mouse/keyboard control.

The script will first take and store the coordinates of the "download" button, by hovering over it in the RobotC IDE then calling a coordinate finder function. With the location now stored, it can click on the download button, type in the path and hit enter whenever needed.

# References


[1] e. a. Tord Romstad, "Github," 4 December 2022. [Online]. Available: https://github.com/official-stockfish/Stockfish. [Accessed 3 April 2023].

[2] "Square Off," 7 September 2022. [Online]. Available: https://squareoffnow.com/blog/stockfish-chess-engine/#:~:text=It%20is%20near%20impossible%20for,been%20able%20to%20beat%20Stockfish.. [Accessed 3 April 2023].

[3] e. a. Gary Bradsky, "Github," Intel, 28 December 2022. [Online]. Available: https://github.com/opencv/opencv. [Accessed 3 April 2023].

[4] A. Sweigart, "Github," [Online]. Available: https://github.com/asweigart/pyautogui. [Accessed 30 January 2023].

[5] OpenCV. [Online]. Available: https://opencv.org/releases/. [Accessed 4 February 2023].

[6] NumPy, "NumPy Documentation," [Online]. Available: https://numpy.org/doc/stable/. [Accessed 14 January 2023].

[7] J. Ding, "ChessVision: Chess Board and Piece Recognition.," Stanford University, [Online]. Available: https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf. [Accessed 17 February 2023].

[8] "Image Filtering Using Convolution in OpenCV," LearnOpenCV, [Online]. Available: https://learnopencv.com/image-filtering-using-convolution-in-opencv/. [Accessed 10 February 2023].

[9] "Edge Detection Using OpenCV.," LearnOpenCV, [Online]. Available: https://learnopencv.com/edge-detection-using-opencv/. [Accessed 11 February 2023].

[10] K. Bapat, "Hough Transform using OpenCV (C++/Python)," LearnOpenCV, [Online]. Available: https://learnopencv.com/hough-transform-with-opencv-c-python/. [Accessed 17 February 2023].

[11] S. Shivakumar, "OpenCV kmeans.," Educaba, [Online]. Available: https://www.educba.com/opencv-kmeans/. [Accessed 20 February 2023].

[12] "Stockfish 3.28.0.," Python Package Index, [Online]. Available: https://pypi.org/project/stockfish/. [Accessed 4 February 2023].

[13] "python-chess: a chess library for Python.," python-chess, [Online]. Available: https://python-chess.readthedocs.io/en/latest/. [Accessed 11 March 2023].