Robotic Chess Arm

Darion Kwasnitza

Table of Contents

1 Introduction……01

1.1 Literature Review
1.2 Goals of Project

2 Methodology……...02

2.1 Overview
2.2 Application
2.2.1 Hardware
2.2.2 Software

3 Results……10

3.1 Testing
3.2 Conclusion
3.3 Future Work

References……..18

Appendices……20

Appendix A - Code
Appendix B - Calculations

1 Introduction

Robotic arms are beginning to be popular in board games because of their ability to enable dynamic and interactive gameplay against human opponents. As an example, a group in Germany [5] developed a robotic arm capable of playing Settlers of Catan with enough accuracy and precision to not knock down any pieces on the board. In another example, a group in Italy[6] created a robotic arm that can play Italian checkers. Whether used for strategic training in a game such as chess or recreational play in games like snakes and ladders, robotic arms, combined with artificial intelligence, can enhance the overall board game experience. In addition to the robotic arms, controllers are important as they analyze moves, employ differing levels of strategy, and determine smooth movement of the robotic arm. As technology advances, robotic arms add a positive element to the world of board games.

1.1 Literature Review

Several robotic systems have been developed in recent years to explore automation in games, object detection, and human-computer interaction. Each takes a different approach depending on its goals, tools, and available budget.

One example is a robot developed by Ms. Kuan-Huang Yu to play Chinese Poker fully autonomously. It used a suction cupping method to handle the cards with an object detection camera to recognize and sort a hand of 13 cards [1]. The robotic arm used was the TM5-900 which has 6 joints for motion and costs over thirty thousand Canadian dollars. In contrast, a more cost-effective robotic arm was developed by Geethanjali College of Engineering and Technology using an Arduino and 4 servo motors [2]. Designed for applications such as automation, human-computer interaction, and education, this arm had three degrees of freedom and a simple clamping end effector.

At Al Akhawayn University, Ali Elouafiq focused on designing a robotic arm that could move chess pawns from one square to another [3]. His studies centered on the mathematics and kinematics of the arm behind the movement rather than the physical build. Another project involved creating an LED chessboard that used proximity reed sensors and the Stockfish chess engine to simulate a computer opponent [4]. After a human player made a move, the board lit up LEDs to show the simulated response move, giving the illusion of an active opponent. Bob Tech developed a Tic-Tac-Toe-playing robotic arm using an Arduino Mega 2560 and a MinMax algorithm [19]. Players input their moves using a membrane switch module, and the robot responds by physically carrying out the next best move.

In a project involving object detection, Josh Allen used a Haar Cascade classifier to detect electrical outlets within an image [18]. By training the classifier with positive and negative

samples; positive being images with electrical outlets, and negative being images without, it was able to learn and recognize outlet shapes.

James D. McCaffrey created a Python tool using the Stockfish engine to evaluate chess positions, both one at a time and in batches, using FEN notation [20]. Using Stockfish's subprocess module in Python, he was able to batch-process chess positions to look even further ahead of the depth he had originally searched [20].

Colour detection was explored by a research team at Qis College of Engineering and Technology, who tested OpenCV's ability to identify specific shade names based on RGB values [21]. By recognizing the exact RGB values of pixels in images and comparing those values to named colours, the research group achieved a 93% accuracy in recognizing and naming the exact colour [21]. Another team at Rajiv Gandhi College used HSV colour space to improve colour recognition in low-light conditions [22]. By converting the RGB value to the HSV (Hue, Saturation, Value) model, they were able to more accurately detect colour in low visibility settings than with the previous RGB model in low visibility.

While each project achieved something unique, they also had their challenges. The Chinese Poker robot worked well but relied on a very expensive industrial robotic arm, making it impractical for most [1]. The Arduino-based arm from Geethanjali was much cheaper but had limited precision due to basic servo motors [2]. Elouafiq's work focused on theory and didn't test the design in real-world conditions [3]. The LED chessboard provided visual feedback but didn't physically move pieces, so it wasn't fully automated [4]. Bob Tech's Tic-Tac-Toe robot was interactive but limited in complexity since it only handled a 3x3 Tic-Tac-Toe grid [19]. Josh Allen's outlet detection model worked best in ideal lighting but struggled with background noise [18]. McCaffrey's project was powerful for move calculation but didn't involve any hardware or visual detection [20]. The RGB-based colour detection system from Qis College [21] struggled in inconsistent lighting, and even with HSV, the Rajiv Gandhi team [22] had trouble distinguishing similar colours in dark or low-contrast settings. Despite their successes, these projects reveal the gap between ideal functionality and practical limits, as well as the difficulties of putting concepts into physical systems.

1.2 Goals of Project

The goal of this project is to create a robotic arm with a controller capable of fully autonomously playing chess against a human opponent.

2 Methodology

2.1 Overview

The development of a robotic chess arm requires both a hardware component and a software component as shown in Figure 1. By using a claw robotic arm, pieces will be moved, replaced, or removed from the board. The board state will be determined through camera object detection.

2.2 Application

2.2.1 Hardware

The main hardware required for the project is the robotic arm itself, the motors, and the controllers. Firstly, the robotic arm itself will be 3D printed using PLA. The robotic arm will be driven by motors with an appropriate torque rating. Motors will be connected to compatible motor drivers that are ultimately connected to an Arduino control board through a shield. The shield will need to manage not only the motors but also the fan, power supply, gripper connections, and anything else that might be connected to the robotic arm. The camera will be connected to a Raspberry Pi and will be positioned above the board, independent of other components of the robotic arm, as shown in Figure 1.

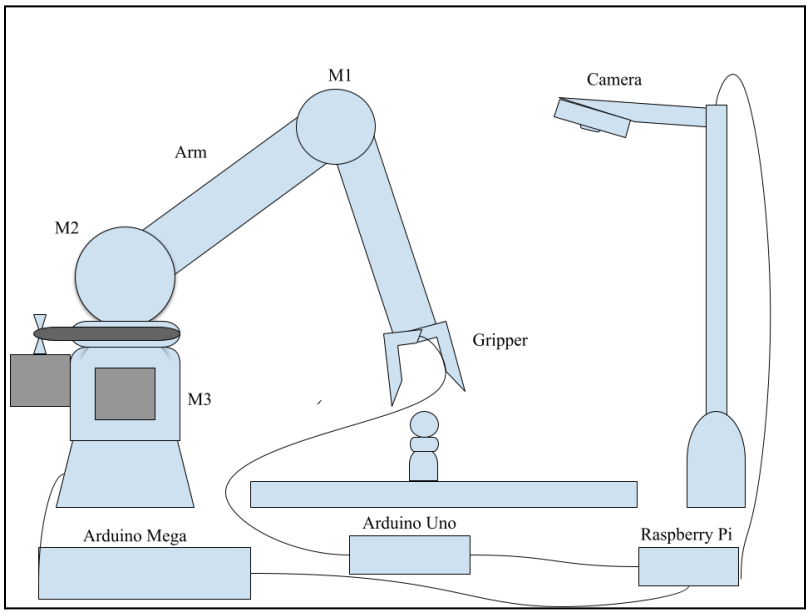


Figure 1. Simple Sketch of Robotic Arm, Camera, and Controller

First, different robotic arms exist, including the delta, XY plotter, and claw. For the current study, a robotic claw was decided upon (Figure 2); the specific robot claw design is from a creator named Ftobler [7]. Robotic claw arms are used in a variety of ways, from arcade machines to medical applications. This style of arm uses three motors. As opposed to other types of arms, the claw does not require a frame and can stand on its own. In addition, it was chosen because of its versatility and simplicity.

To pick up the chess pieces, a pincher gripper will be used (Figure 3); this gripper is designed by hkucs [15]. The reason this gripper will be used is because the two fingers close in parallel, meaning the chess piece has less of a chance of being knocked over. The design is also ideal because it has a long gripper that can reach the bottom of a chess piece without interfering with the top. This gripper will be powered by an SG90 mini servo motor. The end effector will be powered by a separate Arduino UNO (Table 2) because the SG90 has a 5V restriction (Figure 4).

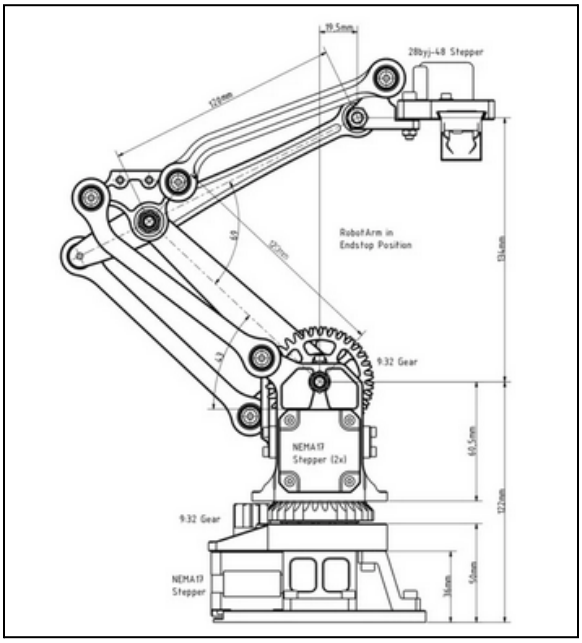


Figure 2. Ftobler's Design of Arm

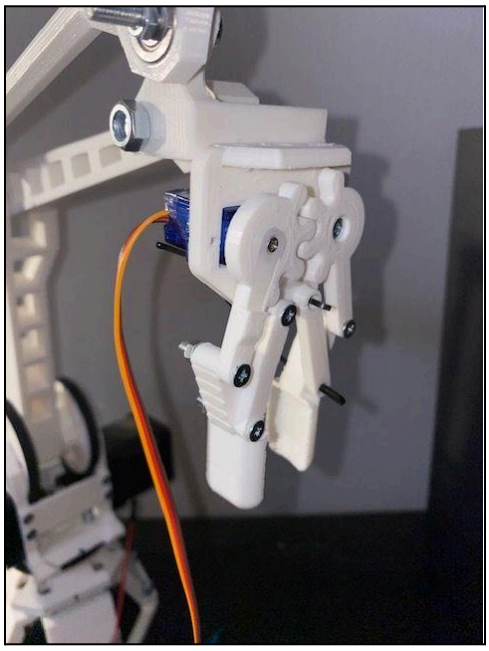


Figure 3. Image of End Effector

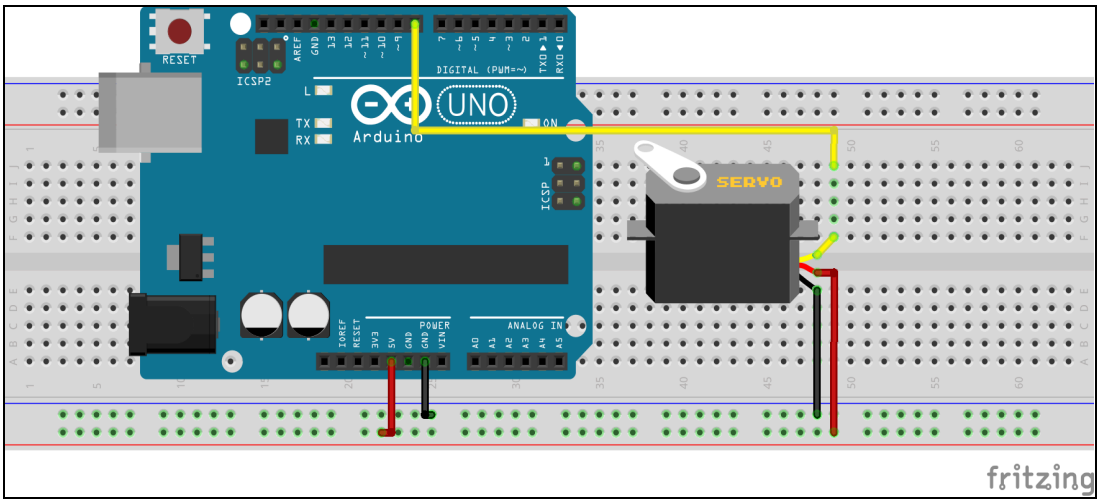


Figure 4. SG90 Servo Motor Connection to Arduino UNO

Next, to control the motors, the Arduino MEGA 2560 will be used. The MEGA has 54 digital IO pins, 256kb of flash memory, a 16MHz clock speed, and 8kb of SRAM (Table 1). These specs are enough to control all 3 motors. The 16MHz clock speed provides accurate timing for stepper and servo motors. In addition to this, having many IO pins allows the MEGA to control multiple components like the A4988 motor drivers, the SG90 gripper servo motor, fans, and limit switches. The 256kb flash memory is enough to handle the code needed for the movement of the arm.

	Arduino Uno	Arduino Mega 2560
Processor	Atmega328P	Atmega2560P
Clock Speed(MHz)	16	16
Flash Memory (kB)	32	256
SRAM(bB)	2	8
Voltage (V)	5	5
Digital IO Pins	14	54
Shield Compatibility	Yes	Yes

Table 1. Arduino Information Table

The motors chosen for the project are the NEMA 17 double-stack stepper motors. These motors have a holding torque of 89 oz-in with a 12V 4A power supply [11]. The NEMA 17 can move up to 3000 steps per second with a maximum torque of ~65 oz-in [11]. The necessary

torque for the arm is ~45 oz-in. Calculations to determine the torque that is required are provided in Appendix B.

In conjunction with the Arduino MEGA and NEMA 17 stepper motors, the Ramps 1.4 shield will allow enough room for the motor board, end stops, gripper wiring, and fan (Figure 7). The Ramps 1.4 shield requires 12V DC. The compatible motor drivers for the NEMA 17 double-stack stepper motors are the A4988 motor drivers. These motor drivers come with micro-stepping jumpers to allow for smoother movement. The D8 - D10 connections allow for hardware components such as fans to be connected to ensure the motor drivers do not overheat (Figure 7). A 12V 5010 fan will be connected to help prevent motor drivers from overheating. The 12V 4A power supply will be connected to the Ramps 1.4 board through a 2.1mm Female DC Jack.

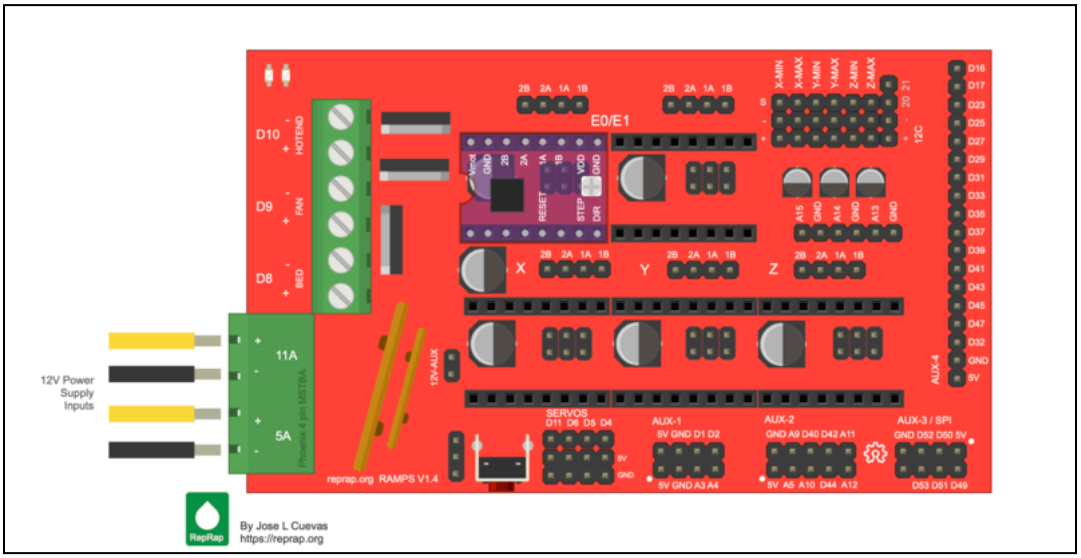


Figure 7. Ramps 1.4. Board Schematic [13]

Next, to power a camera and run Stockfish (a chess-solving engine in Python), the Raspberry Pi communicates with the Arduino MEGA through USB. The Raspberry Pi Model 3 B was chosen for this study because it's operating system runs Python natively, has enough processing power for object detection and chess engine computations, and offers enough built in USB ports to connect an Arduino Uno, Arudino MEGA, and a camera. A full circuit diagram of the entirety of the hardware is shown in Figure 8.

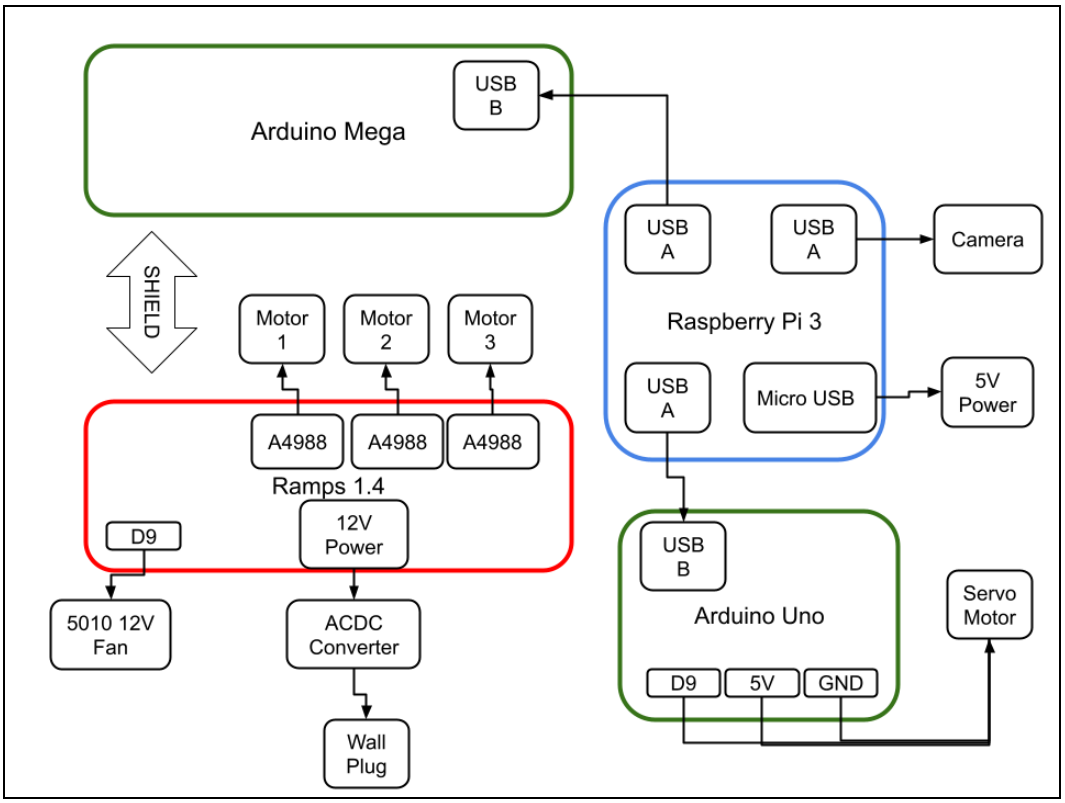


Figure 8. Circuit Diagram of Hardware System

2.2.2 Software

For the project, the software controls the movement of the robotic arm and the end effector. It also detects the chessboard state and solves for new moves. The process starts when a player makes a move, which triggers object detection to update the board state. There are three modes of detecting the chessboard state: automatic mode, manual mode, and manual FEN mode. Automatic mode detects the chessboard in the frame and all pieces on the board and then waits for user input to send the move to Stockfish. Manual mode works similarly to automatic mode, but the user is prompted to click the four corners of the chess board in the displayed frame for manual calibration before sending the move to Stockfish. Manual mode will only work when a display of the camera frame is available for the user. Manual FEN mode requires the user to input a FEN string, which is sent directly to Stockfish for processing. The code for the different modes is provided in Appendix A. The new board state is then sent to the chess solver to find the best move. The solver then sends the move to the Arduino MEGA, which controls the robotic arm to carry out the move. After the robot completes its move, the cycle repeats. A complete flowchart is shown below in Figure 9.

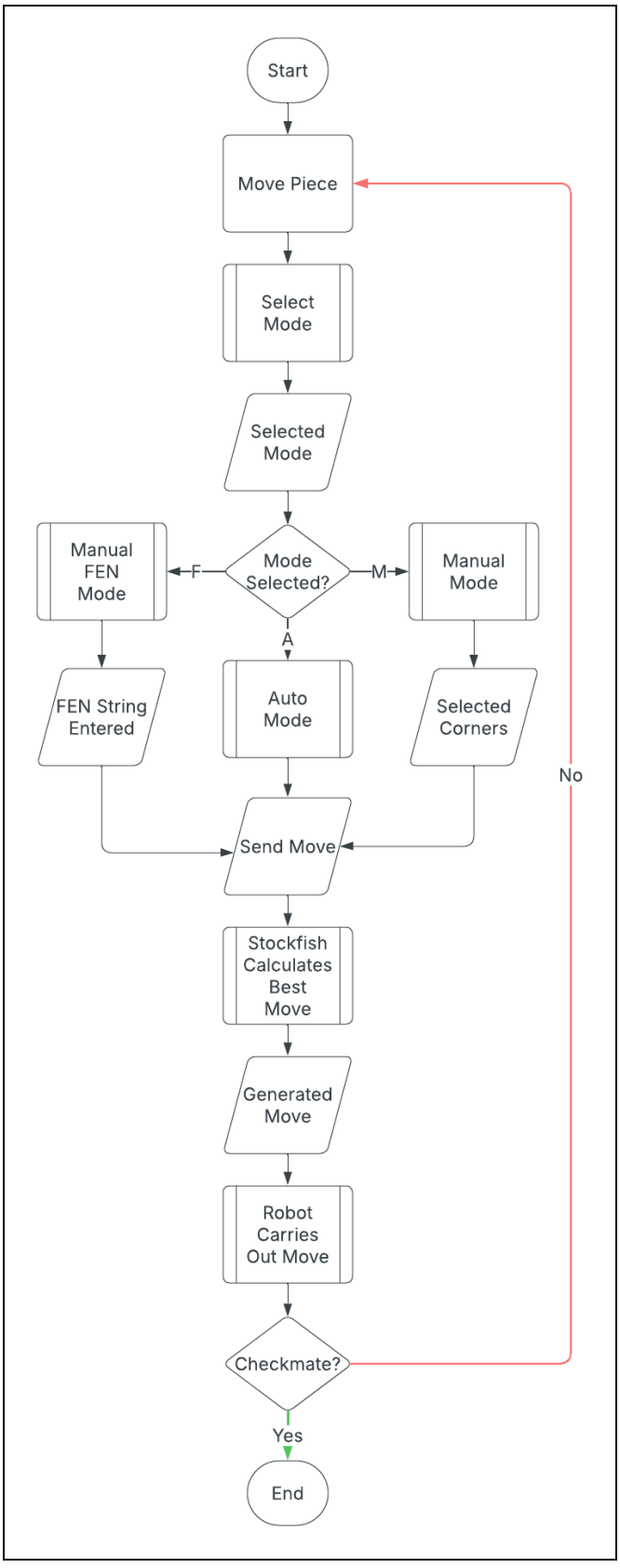


Figure 9. Flowchart

The program uses OpenCV in Python to process the video feed and perform object detection, including identifying the chessboard and locating pieces on it. To detect the chessboard, the system uses a live video feed from the Raspberry Pi's connected camera. In automatic mode, the image is first converted to grayscale, and then the image is blurred to reduce noise. Once this is done, edge detection is applied to find the outlines of shapes. The code looks for the largest contour with four corners and a roughly square shape, assuming it's the chessboard. Ensuring the chess board has a bold and opposite background is extremely important for automatic mode to work correctly. Once found, the four corners are used to warp the board into a top-down view. In manual mode, the user clicks the four corners of the board directly, and the same warping process is applied based on the selected points. This makes sure the board is always aligned properly for the next steps, no matter the lighting or camera angle, but it also means that if the board is moved within the frame, the original alignment will be off.

After the board is warped, the program checks each square to see if a chess piece is present there. It converts each square from RGB to HSV, which helps make colour detection more reliable in different lighting conditions. Each piece type and colour combination has a different coloured sticker on the top and, therefore, has its own HSV colour range. The program applies a mask of all possible colours one at a time, and if more than 20 pixels of a specific HSV range is detected within a square, then the program marks that square with the detected piece. Morphological filtering is applied to clean up the image and reduce noise, which helps prevent incorrect detection. The Python code for chessboard detection and object detection can be found in Appendix A.

To get the arm to move to a certain position, a valid move must be made and sent from the Raspberry Pi via USB serial communication to the Arduino MEGA, where the move will be executed. To generate and validate the best moves, the Stockfish engine in Python will be used. The Stockfish engine [10] searches all possible moves within a set depth given a FEN string [9]. A FEN string is a representation of the current board state [9]. An example of a FEN string is:

"rnbqkbnr/pppp1ppp/8/4p3/6p1/5p2/PPPPP2P/RNBQKBNR b KQkq - 0 2".

Given the FEN string, Stockfish can then generate the next best move based on a set difficulty level (0 -20) and depth of search. The generated output move from Stockfish is communicated to the Arduino MEGA and will get the robot to move to certain squares. The Python code for Stockfish can be found in Appendix A.

The robotic arm moves using three NEMA 17 stepper motors that control the base, shoulder, and elbow joints. For each square, there are two positions: one slightly above the board and one lower down where the gripper can pick up or drop a piece. When a move command is sent to the Arduino MEGA, the arm moves from its home position to the start square, lowers

down to grab the piece, grips it, lifts it, moves to the destination square, and then releases the piece before returning home. The gripper itself is controlled by a separate Arduino Uno using serial communication.

To make sure the movements are smooth, the program calculates how much each motor needs to rotate based on the difference between the current and target angles. These angle differences are converted into motor steps using a set value for steps per degree. Calculations for the steps per degree can be viewed in Appendix B. The AccelStepper library was used to control each motor. When going to a position, the base motor first moves by itself to rotate the arm from 0 to 180 degrees, and then the shoulder and elbow motors move together to position the arm accurately. The program waits until each part of the movement is done before moving on to the next step of the movement so there is no overlapping movement. The code for movement sequencing logic and motor control can be found in Appendix A.

To calculate the exact joint angles needed to reach a specific position, the program uses Newton's Method. This is a numerical way to solve the arm's inverse kinematics equations by starting with an initial guess and improving it step by step. At each step, the program calculates how close the current guess is to the target position and makes adjustments using something called a Jacobian matrix. The process repeats until the arm's calculated position is close enough to the target. The angles are also kept within safe limits to make sure the motors don't try to move beyond what the arm can physically handle, and these limits are predefined. The inverse kinematic equations can be found in Appendix B, and the Newton's Method code was generated using a program called Wolfram Alpha [23].

The robot knows which square to go to based on a move string sent from the Raspberry Pi, such as "e2e4". This string represents the source and destination squares. The Raspberry Pi runs Stockfish, which calculates the best move, then sends it to the Arduino Mega over serial. Once the Mega receives the move, it looks up the predefined angles and positions for each square and executes the movement. During the move, the Arduino Mega also communicates with an Arduino Uno that controls the gripper. The Arduino Uno listens for these messages and uses a servo motor to open or close the gripper accordingly based on the message it receives. The code for the gripper and the move breakdown can be found in Appendix A.

3 Results

3.1 Testing

Test #	Description	Pass /Fail
1	Camera detects the chessboard	Pass
2	Camera warps chessboard and applies squares	Pass

3	Camera detects colours	Pass
4	Camera detects different pieces based on colour	Pass
5	Camera detects pieces on chessboard	Pass
6	Camera generates FEN string from detected board state	Fail
7	The robotic arm moves 90 degrees	Pass
8	The robotic arm resets to home after moving	Pass
9	Stockfish generates the next best move based on an inputted FEN string	Pass
10	The robotic arm moves to the appropriate square if in reach	Pass
11	The robotic arm picks up and drops off a piece	Pass

Table 2. Table showing sample test descriptions and results

The success of test 1 and test 2 are shown in Figure 10. On the left side of the image, it can be seen that the largest contour, the chessboard, has been located by the camera. The chessboard needed to have a thick outline that was contrasted from the background so that it would be detected as having contours. On the right side of Figure 10, there is a warped version of the chessboard, and there are squares drawn on the chessboard that correlate the the physical squares of the chessboard.

The success of test 3 is shown in Figure 11. The left side of the screen is the original board with no mask, and the right side of the screen is the original board with an HSV mask applied based on the HSV calibration settings. The HSV calibration screen was created to get specific HSV ranges for each colour. In Figure 11, an HSV mask of min values [68, 75, 188] and max values [104, 166, 255] were applied to filter out everything besides the light blue, which can be seen as a white dot on the right side of the screen. It can also be seen that there is some noise on the right-hand side of the mask from the dark blue king, which is why morphological filtering was used to filter out this small noise and why for a piece to be detected, it must have more than 20 pixels detected in the square.

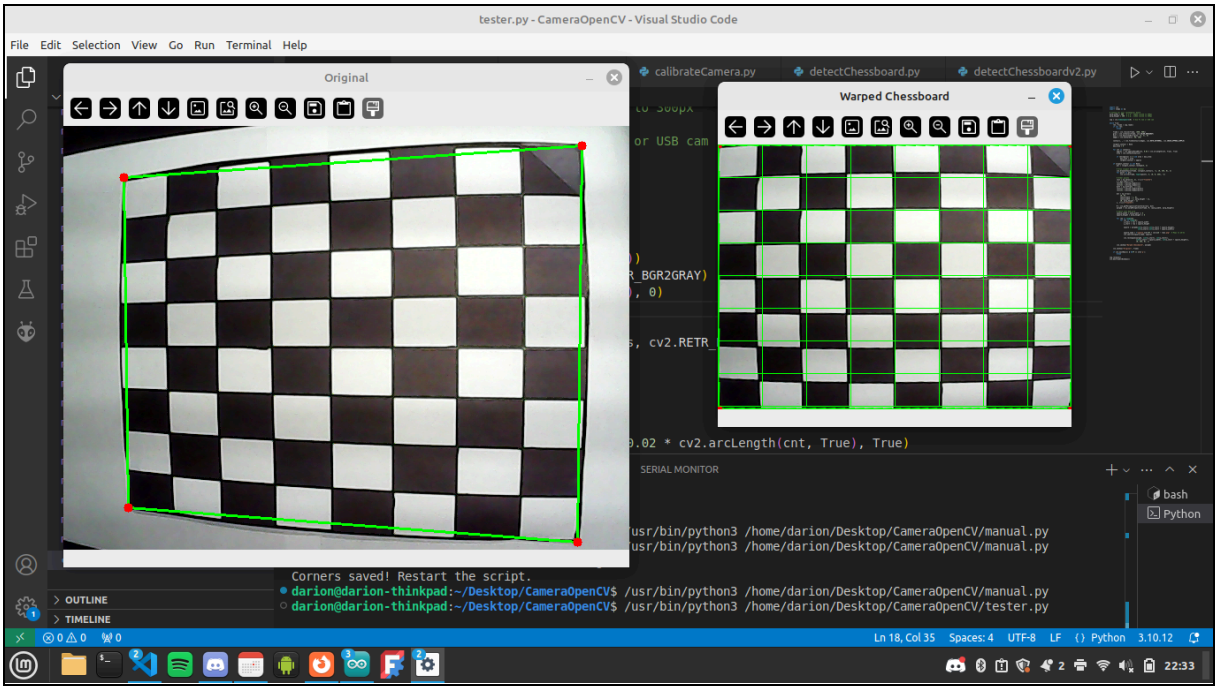


Figure 10. Chessboard and Square Detection

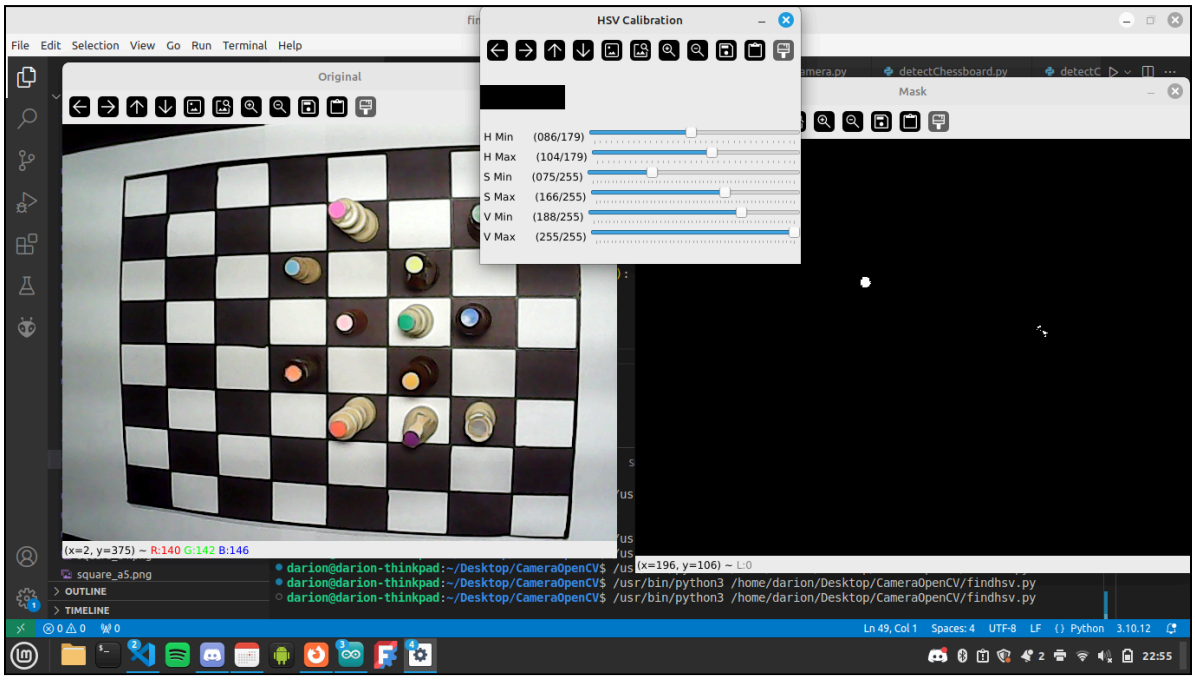


Figure 11. Colour Detection Using HSV Mask

The success of tests 4 and 5 are shown below in Figure 12. Figure 12 shows multiple black pawns scattered around the board, each with a pink sticker. The camera has placed ‘p’ in the square where it has detected a black pawn, and all of them are being detected correctly. However, Figure 13 shows the same board state where 1 of the pawns is not being recognized.

This was a common issue that was being run into during automatic and manual mode testing, likely due to slight changes in lighting.

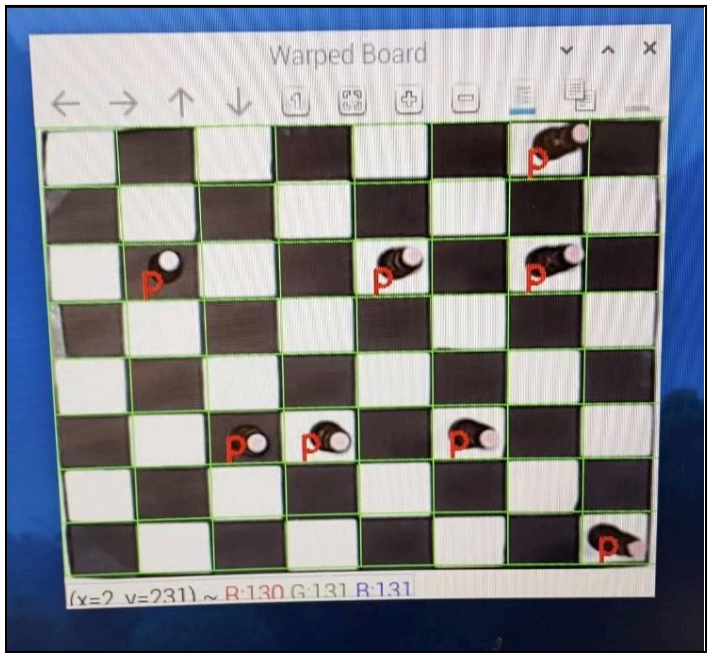


Figure 12. Colour Detection Success

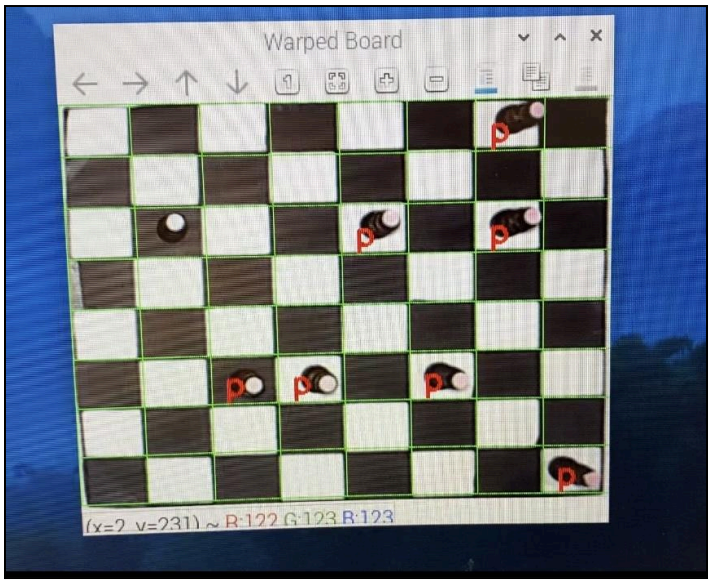


Figure 13. Colour Detection Partial Failure

The faulty detection shown in Figure 13 was a large issue when trying to generate a FEN string in automatic mode, as every piece of the board needed to be perfectly detected at once along with the chessboard itself. Because of this, test 6 was a failure in automatic mode. However, in manual mode, where the chessboard was detected frequently, there were instances

of success, as shown in Figure 14. Figure 14 is the same board state as shown in Figures 12 and 13 and as the FEN string generated exactly matches the board state, meaning there was partial success in manual mode.

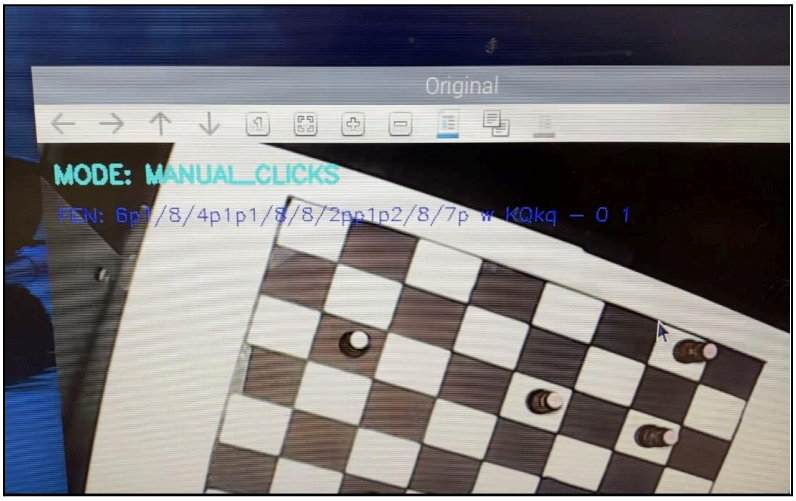


Figure 14. FEN String Generated from Board State in Manual Mode

The success of tests 7 and 8 is shown in Figures 15 and 16. Figure 15 shows the robotic arm at the home position. Figure 16 shows the robotic arm after it has moved 90 degrees. These tests show that the calculations for theta not and for steps per degree are both working correctly.

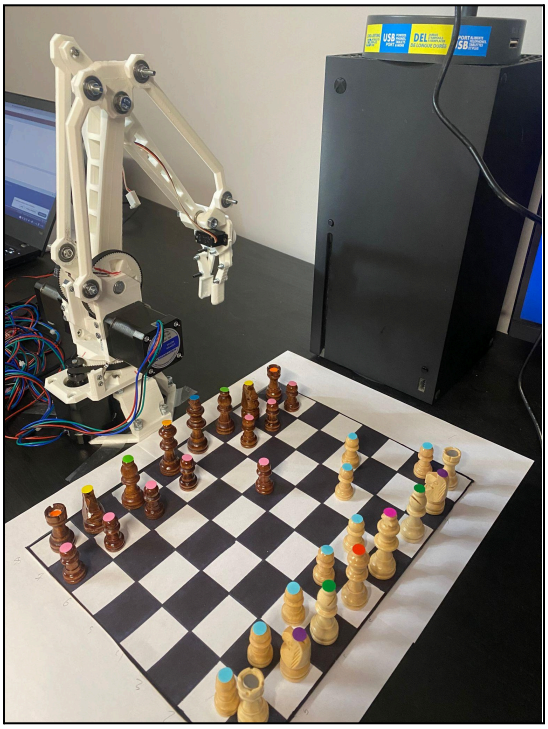


Figure 15. Robotic Arm at Home Position

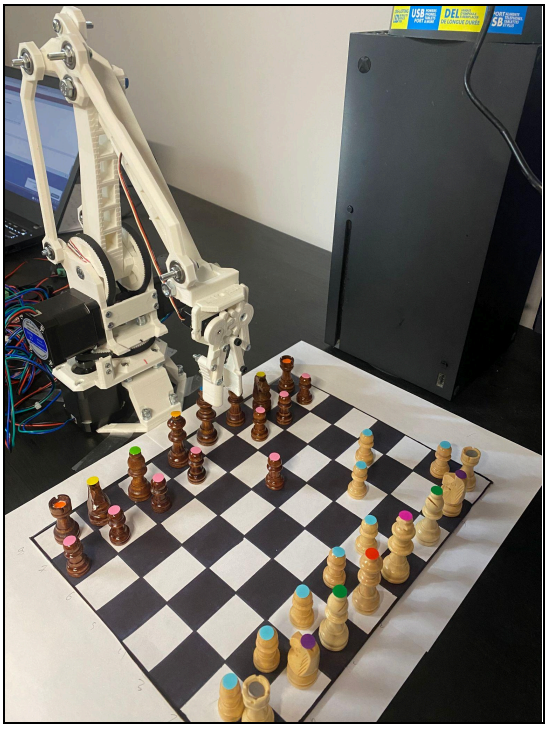


Figure 16. Robotic Arm at 90 Degrees

The success of test 8 is shown in Figure 17. A board state with a checkmate is given to Stockfish, and it returns the correct move “d8h4” that would be necessary to checkmate the opponent.

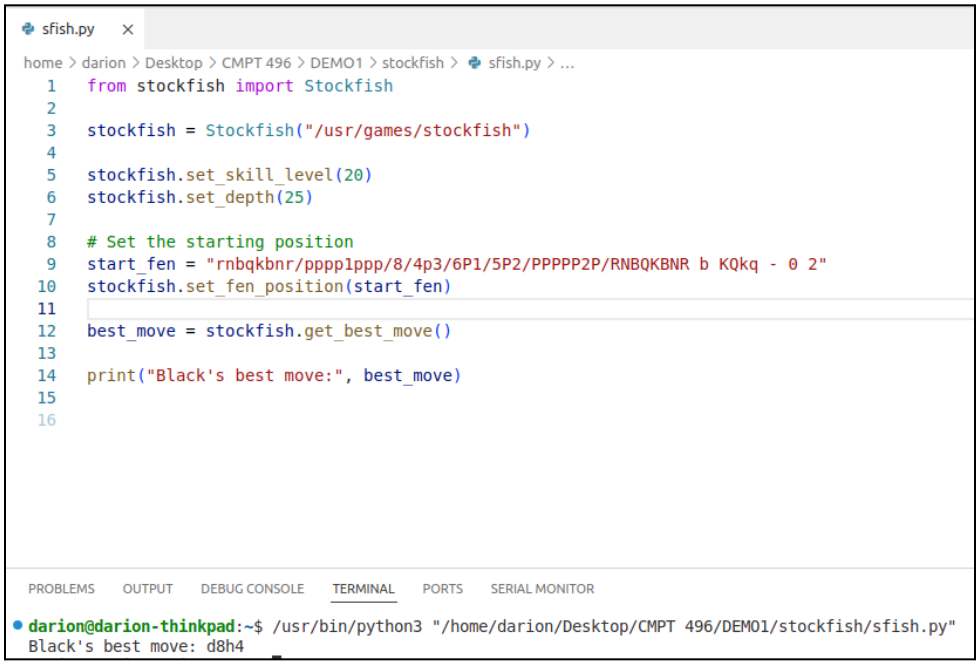


Figure 17. Stockfish Generating Next Best Move

The success of tests 10 and 11 is shown in Figure 18. Figure 18 is a collage of photos showing the robotic arm receiving an “e7e5” command and its corresponding movement. The robotic arm starts from home, goes to hover over square e7 and then moves straight down and picks up the pawn. The robotic arm then raises and moves to square e5, where it lowers and sets the pawn down before rising again and returning home. Figure 18 shows a successful move from start to finish. This test also shows that the calculation of theta one and theta two are correct as the arm goes to the correct squares based on their respective angles. It is important to note that the robotic arm fails to reach the far corners of the board and, therefore, cannot play all possible moves from Stockfish. The physical limitations of the robot have been taken into account, and if a square returned from Stockfish returns one of the squares that is unreachable, a new move is generated. This is a current limitation of this project.

Figure 18. Robotic Arm Executing a Move

3.2 Conclusions

The goal of this project was to create a robotic arm with a controller capable of fully autonomously playing chess against a human component. Overall, the results conclude that the goal of this project was not fully met. Although the robotic arm can move and pick up pieces

based on an entered FEN string, it fails to do so in automatic mode. In manual mode, it has instances of success, but manual mode is not fully autonomous and therefore, the goal of this project was not met. Using manual FEN mode to bypass the object detection allows for the robotic arm to consistently make moves, but again, it is not fully autonomous. The results show that there is not enough consistency using the methods chosen to allow the robotic arm to play chess fully autonomously against a human opponent.

3.3 Future Work

Many things could be improved in this project. It would be beneficial to change the gripper and make it smaller so it does not interfere with pieces when it picks something up. The movement method is quite good but would benefit from having some sort of consistent initial homing method and an attached board so that even if the robotic arm moves, so does the board. The object detection portion of the project would benefit from memory, starting from an initial inputted board state and then just detecting changes in that board state instead of the entire board. The robot also cannot reach all of the squares on the chess board currently, so it would be beneficial to reprint the arms longer or use a smaller chess board. The biggest benefit to this project would be to work on the object detection as the methods of the movement are easily applicable to other boards if the gripper and arms get reprinted correctly.

References

[1] Yu, K. (2023, November). “Integration of Robotics, Computer Vision, and Algorithm Design: A Chinese Poker Self-Playing Robot”. *ARXIV* [Online]. Available: https://arxiv.org/pdf/2312.09455

[2] Chenchireddy, K. (2024, September). “Development of robotic arm using Arduino controller”. *IAES International Journal of Robotics and Automation* [Online]. Available: https://ijra.iaescore.com/index.php/IJRA/article/viewFile/20676/13086

[3] Ali, E. (2012, April). “Design and Engineering of a Robotic Arm”. *ARXIV* [Online]. Available: https://arxiv.org/pdf/1204.1649

[4] MaxChess. (2016, January). “Wooden Chess Board with Piece Recognition”. *Hackster.io* [Online]. Available: https://www.hackster.io/Maxchess/wooden-chess-board-with-piece-recognition-872ffb

[5] Liszewski, A. (2016, April). “Ultra-Precise Catan-Playing Robots Will Never Mess Up the Game Board”. *Gizmodo* [Online]. Available: https://gizmodo.com/ultra-precise-catan-playing-robots-will-never-mess-up-t-1768872371

[6] Fabris, G. et al. (2023, December). “Playing Checkers with an Intelligent and Collaborative Robotic System”. *MDPI* [Online]. Available: https://www.mdpi.com/2218-6581/13/1/4

[7] ftobler. (2016, August). “RobotArm”. *Thingiverse* [Online]. Available: https://www.thingiverse.com/thing:1718984

[8] Gudino, M. (2021, February). “Arduino Uno vs. Mega vs. Micro”. *Arrow* [Online]. Available: https://www.arrow.com/en/research-and-events/articles/arduino-uno-vs-mega-vs-micro

[9] CHESS.com. “Forsyth-Edwards Notation (FEN)”. *Chess.com* [Online]. Available: https://www.chess.com/terms/fen-chess

[10] Zhelyabuzhsky, I. (2022, July). “Stockfish 3.28.0”. *Python Software Foundation* [Online]. Available: https://pypi.org/project/stockfish/

[11] PBCLinear. “Stepper Motor Nema 17”. *PBCLinear* [Online]. Available: https://pages.pbclinear.com/rs/909-BFY-775/images/Data-Sheet-Stepper-Motor-Support.pdf

[12] ZeroToHeroEngineering. (2020, July). “How to set VRef for A4988 and DRV8825 stepper motor drivers”. *Youtube* [Online]. Available: https://www.youtube.com/watch?v=BV-ouxhZamI

[13] RepRap. (June 2024). “Ramps 1.4”. *RepRap* [Online]. Available: https://reprap.org/wiki/RAMPS_1.4

[14] Pololu. (2001). “Minimal wiring diagram”. *Pololu Robotics and Electronics* [Online]. Available: https://www.pololu.com/picture/view/0J3360

[15] hkucs-makerlab. (2020, May). “RobotArm - Mini Servo Gripper”. *Thingiverse* [Online]. Available: https://www.thingiverse.com/thing:4394894

[16] electronics-lab. (2016, January). “Using the SG90 Servo Motor With An Arduino”. *Electronics-lab open source hardware projects* [Online]. Available: https://www.electronics-lab.com/project/using-sg90-servo-motor-arduino/

[17] Winters, R. (2002). “Raspberry PI 3 Pinout Diagram | Circuit Notes”. *Jameco* [Online]. Available: https://www.jameco.com/Jameco/workshop/CircuitNotes/raspberry-pi-circuit-note.html -

[18] Allen, J. (2015). “Object Detection Using OpenCV”. *GithubPages* [Online]. Available: https://johnallen.github.io/opencv-object-detection-tutorial/

[19] bobn2tech. (2018, April). “Tic-Tac-Toe Board Game with Robotic Arm”. *Project Hub* [Online]. Available: https://projecthub.arduino.cc/bobn2tech/tic-tac-toe-board-game-with-robotic-arm-6f9f78

[20] McCaffrey, J. (2024, May). “Programmatically Analyzing Chess Games Using Stockfish With Python”. *WordPress* [Online]. Available: https://jamesmccaffrey.wordpress.com/2024/05/02/programmatically-analyzing-chess-games-using-stockfish-with-python/

[21] Raguraman, P et al. (2021, January). “colour Detection of RGB Images Using Python and OpenCV”. *IJSCSEIT* [Online]. Available: https://www.researchgate.net/publication/349355136_colour_Detection_of_RGB_Images_Using_Python_and_OpenCv

[22] Dabekar, S et al. (2023, June). “Colour Detection Using Python and OpenCV”. *IJARSCT* [Online]. Available: https://ijarsct.co.in/Paper11343.pdf

[23] Wolfram Alpha. “Inverse Kinematics Using Newton’s Method”. Wolfram Alpha [Online]. Available: https://www.wolframalpha.com

Appendices

Appendix A - Code

Auto Mode:

```
while True:
    ret, frame = cap.read()
    if not ret:
        break
    # resize frame, easier for pi
    frame = cv.resize(frame, (640, 480))
    display_frame = frame.copy()
    # gray, blur and find edges and contours
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    blur = cv.GaussianBlur(gray, (5, 5), 0)
    edges = cv.Canny(blur, 80, 200)
    contours, _ = cv.findContours(edges, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
    # init largest
    largest_contour = None
    max_area = 0
    # if auto, look for largest contour
    if mode == "auto":
        for cnt in contours:
            #https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga0012a5fdaea70b8a9970165d98722b4c
            approx = cv.approxPolyDP(cnt, 0.02 * cv.arcLength(cnt, True), True)
            area = cv.contourArea(cnt)
            # if all four contours and bigger then reset max area and largest contour
            if len(approx) == 4 and area > max_area:
                x, y, w, h = cv.boundingRect(approx)
                aspect_ratio = w / float(h)
                if 0.8 < aspect_ratio < 1.4:
                    max_area = area
                    largest_contour = approx

    pts = None
    if largest_contour is not None and mode == "auto":
        pts = largest_contour.reshape(4, 2)
```

Manual Mode:

```
    elif mode == "manual_clicks" and len(clicked_points) == 4:
        pts = np.array(clicked_points, dtype="float32")

    if pts is not None:
        rect = np.zeros((4, 2), dtype="float32")
        s = pts.sum(axis=1)
        rect[0] = pts[np.argmin(s)]
        rect[2] = pts[np.argmax(s)]
        diff = np.diff(pts, axis=1)
        rect[1] = pts[np.argmin(diff)]
        rect[3] = pts[np.argmax(diff)]

        dst = np.array([
            [0, 0],
            [warp_width - 1, 0],
            [warp_width - 1, warp_height - 1],
            [0, warp_height - 1]
        ], dtype="float32")
        # warp perspective
        M = cv.getPerspectiveTransform(rect, dst)
        warped = cv.warpPerspective(frame, M, (warp_width, warp_height))

        fen, warped = draw_board_labels(warped)

        # show warped board
        cv.imshow("Warped Board", warped)
```

Manual FEN Mode and Switching Modes:

```
273     elif key == ord('f'):
274         mode = "manual_fen"
275         manual_fen_override = input("Enter custom FEN string here: ")
276         print(f"Manual FEN loaded: {manual_fen_override}")
277         # set fen position and manually overide for fools mate
278         if validate_fen(manual_fen_override):
279             stockfish.set_fen_position(manual_fen_override)
280             # fools mate for demo
281             if manual_fen_override == "rnbqkbnr/pppppppp/8/8/6P1/8/PPPPPP1P/RNBQKBNR b KQkq - 0 1":
282                 best_move = "e7e5"
283                 print("Detected special FEN - forcing move to:", best_move)
284             # if past opening of fools mate
285             else:
286                 best_move = stockfish.get_best_move()
287             print("Stockfish best move:", best_move)
288             # send to robot to do
289             send_to_robot(best_move)
290         else:
291             print("Invalid FEN entered.")
292         # return to auto mode
293         mode = "auto"
294         print("Returning to AUTO mode")
295     # switch to manual mode and reset clicked points
296     elif key == ord('m'):
297         mode = "manual_clicks"
298         clicked_points = []
299         print("Switched to MANUAL CLICK MODE")
300     #switch to auto mode
301     elif key == ord('a'):
302         mode = "auto"
303         clicked_points = []
304         print("Switched to AUTO MODE")
305     # reset clicked points when in manual mode
306     elif key == ord('r') and mode == "manual_clicks":
307         clicked_points = []
308         print("Manual clicks reset.")
```

Colour Detection:

```
 99  def detect_piece(square_img):
100      hsv = cv.cvtColor(square_img, cv.COLOR_BGR2HSV)
101      # for each piece and lower, upper bound in piece_colours loop
102      for piece, (lower, upper) in piece_colors.items():
103          # apply mask
104          mask = cv.inRange(hsv, np.array(lower), np.array(upper))
105          # set kernel and reduce noise
106          # https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html
107          kernel = np.ones((5, 5), np.uint8)
108          mask = cv.morphologyEx(mask, cv.MORPH_OPEN, kernel)
109          mask = cv.morphologyEx(mask, cv.MORPH_CLOSE, kernel)
110          # check for at least 20 pixels
111          if cv.countNonZero(mask) > 20:
112              return piece
113      return ""
114
```

Gripper Communication:

```
71  # sends OPEN to uno through serial
72  def grip_open():
73      gripper_serial.write(b'OPEN\n')
74      gripper_serial.flush()
75
76  # sends CLOSE to uno through serial
77  def grip_close():
78      gripper_serial.write(b'CLOSE\n')
79      gripper_serial.flush()
80
81
82  def send_to_robot(move):
83      # debug
84      print(f"Sending move to robot: {move}")
85      mega_serial.write((move + '\n').encode())
86      mega_serial.flush()
87
88      # close gripper is ready
89      if wait_for_mega("READY_TO_GRIP"):
90          grip_close()
91          time.sleep(0.7)
92
93      # release gripper if ready
94      if wait_for_mega("READY_TO_RELEASE"):
95          grip_open()
96          time.sleep(0.7)
97
```

Validate Move With Stockfish:

```
115 def validate_fen(fen):
116     ranks = fen.split(" ")[0].split("/")
117     if len(ranks) != 8:
118         return False
119     for rank in ranks:
120         squares = 0
121         for char in rank:
122             if char.isdigit():
123                 # A-H
124                 squares += int(char)
125             elif char.isalpha():
126                 #1-8
127                 squares += 1
128         if squares != 8:
129             return False
130     return True
131
```

Gripper Movement:

```
#include <Servo.h>
Servo gripper;
const int servoPin = 9;
const int openAngle = 40;
const int closeAngle = 0;

void setup() {
  Serial.begin(115200);
  gripper.attach(servoPin);
  gripper.write(openAngle);
}

void loop() {
  if (Serial.available()) {
    String command = Serial.readStringUntil('\n');
    command.trim();

    if (command == "OPEN") {
      gripper.write(openAngle);
      Serial.println("Gripper opened");
    }
    else if (command == "CLOSE") {
      gripper.write(closeAngle);
      Serial.println("Gripper closed");
    }
  }
}
```

Move From and To:

```
void moveFT(Angles from, Angles to){
  double deltaThetaNot = to.thetaNot - from.thetaNot;
  double deltaTheta1 = to.theta1 - from.theta1;
  double deltaTheta2 = to.theta2 - from.theta2 + deltaTheta1;

  int stepsThetaNot = deltaThetaNot * 40;
  int stepsTheta1 = deltaTheta1 * 40;
  int stepsTheta2 = deltaTheta2 * 40;

  baseMotor.move(stepsThetaNot);
  elbowMotor.move(stepsTheta2);
  shoulderMotor.move(stepsTheta1);

  while (baseMotor.distanceToGo() != 0){
    baseMotor.run();
  }
  while (elbowMotor.distanceToGo() != 0 || shoulderMotor.distanceToGo() != 0){
    elbowMotor.run();
    shoulderMotor.run();
  }
}
```

Handle Any Move:

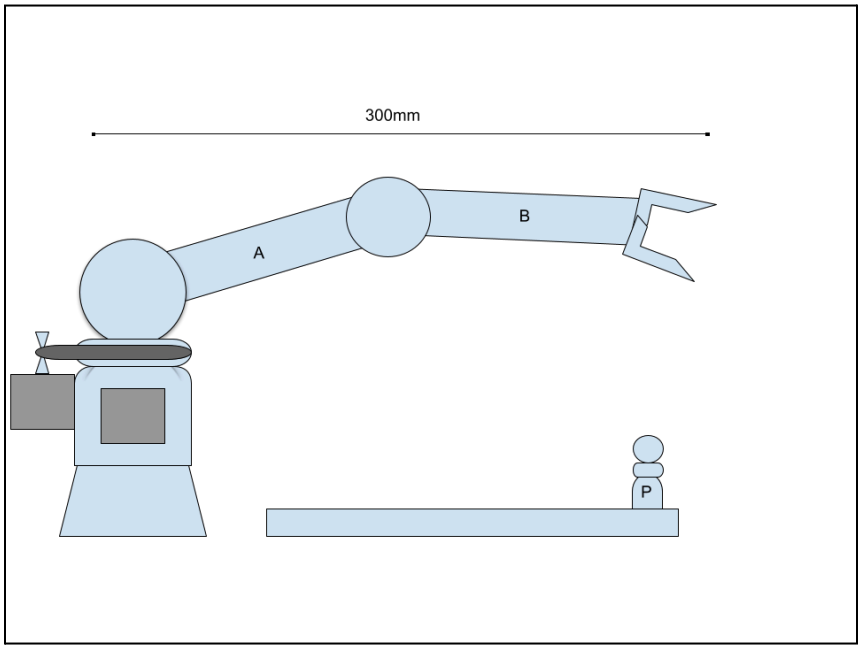
```
void handleMove(const String& input) {
  if (input.length() != 4) {
    Serial.println("Invalid move format.");
    return;
  }

  String fromStr = input.substring(0, 2);
  String toStr = input.substring(2, 4);
  Square fromSq, toSq;

  if (!getSquareByName(fromStr, fromSq)) {
    Serial.print("Invalid from-square: "); Serial.println(fromStr);
    return;
  }
  if (!getSquareByName(toStr, toSq)) {
    Serial.print("Invalid to-square: "); Serial.println(toStr);
    return;
  }

  moveFT(homeA, fromSq.upA);       delay(200);
  moveFT(fromSq.upA, fromSq.downA); delay(200);
  Serial.println("READY_TO_GRIP"); delay(1000);
  moveFT(fromSq.downA, fromSq.upA); delay(200);
  moveFT(fromSq.upA, toSq.upA);     delay(200);
  moveFT(toSq.upA, toSq.downA);     delay(200);
  Serial.println("READY_TO_RELEASE"); delay(1000);
  moveFT(toSq.downA, toSq.upA);     delay(200);
  moveFT(toSq.upA, homeA);          delay(200);
}
```

Handling Moves and Motor Control:

```
void setup() {
  pinMode(38, OUTPUT); digitalWrite(38, LOW);
  pinMode(56, OUTPUT); digitalWrite(56, LOW);
  pinMode(62, OUTPUT); digitalWrite(62, LOW);
  pinMode(9, OUTPUT);  digitalWrite(9, HIGH);

  Serial.begin(115200);
  while (!Serial) {}

  baseMotor.setMaxSpeed(500); baseMotor.setAcceleration(200);
  shoulderMotor.setMaxSpeed(500); shoulderMotor.setAcceleration(200);
  elbowMotor.setMaxSpeed(500); elbowMotor.setAcceleration(200);

  baseMotor.setPinsInverted(true, false, false);
  elbowMotor.setPinsInverted(true, false, false);
}

void loop() {
  if (Serial.available()) {
    String command = Serial.readStringUntil('\n');
    command.trim();
    Serial.print("Received: ");
    Serial.println(command);
    handleMove(command);
  }
}
```

Appendix B - Mathematics

Torque Calculations:
T = F x D
D = L/2 = (0.3m / 2) = 0.15m
F = Fp + (Fa + Fb) = (Mp x g) + (Mab x g) = (0.014kg x 9.81m/s2) + (0.2kg x 9.81m/s2) = ~2.1
T = (2.1N x 0.15m) = 0.315N-m x 141.611 oz-in = 44.6 oz-in



Steps Per Degree Calculations:

1. 360 degrees / 1.8 degrees = 200 steps/rev x 16 (microstepping) = 3200 steps/rev
2. Gear ratio: 90/20 = 4.5
3. 4.5 x 3200 = 14400 steps/rev
4. 14400 / 360 = 40 step/degree

Inverse Kinematic Equations:
R = L1Sin(θ) + L2Sin(θ2)
Z = L1Cos(θ) + L2Cos(θ2)
L1 = length of first arm
L2 = length of second arm