Autonomous Robots

Education and project:

BSc in Robot Systems Engineering

2. Semester Project

Submission date:

20. June 2024

Supervisor:

Jan Petersen

Group members:

Aksel Munksgaard-Ottosen, Emil De Maeyer, Jonathan Dalsgaard, Mads-Christian André Søgaard Hyrup, Pascal Nestås Bjørstorp, Thomas Faber Jensen

Faculty of Engineering

University of Southern Denmark

Abstract

This project aimed to develop a robotic system using a Universal Robot 5 CB3 (UR5 CB3) to play checkers, focusing on surface detection and iterative performance improvement. Key components included a custom-designed gripper, computer vision for board detection, and AI for game strategy.

The gripper, constructed with 3D-printed parts and reinforced with steel hardware, was designed to withstand forces up to 50 N. It demonstrated durability and effectiveness in handling checkers pieces, with features such as embedded nuts. Despite some flexibility issues, the gripper's design ensured reliability and ease of repairability.

Computer vision techniques enabled consistent detection of the checkerboard, facilitating reliable gameplay with minimal human intervention. However, precision improvements in camera calibration and alignment were identified as necessary.

The AI component utilized MinMax algorithms with alpha-beta pruning to optimize gameplay strategies. Despite computational limitations, the AI showed significant improvements over time, iterating through multiple generations to enhance performance. To remedy the computational limitations of the MinMax algorithm, an AI using a database was implemented.

Overall, the project successfully integrated robotic manipulation, computer vision, and AI to create an autonomous checkers-playing robot. This project demonstrates some of the potential of robotics and artificial intelligence in recreational activities.

Table of Contents

1 Introduction **2**

2 Software **3**
2.1 Agile Software Development . . . 3
2.2 Checkers: The game . . . 3
2.2.1 AI . . . 5
2.3 Communication with the robot . . . 11
2.4 ATmega . . . 12
2.4.1 Communication . . . 12
2.4.2 ATmega644PA program . . . 12
2.4.3 PC program . . . 15
2.4.4 ATmega operations . . . 16
2.5 Computer vision . . . 18
2.5.1 OpenCV . . . 18
2.5.2 Conversion to real position . . . 18
2.5.3 Updating the position of the checkers . . . 20
2.6 Kinematics . . . 21
2.7 GUI . . . 23
2.7.1 GUI framework . . . 23
2.7.2 Base code . . . 23
2.7.3 Compatibility with the game code . . . 24

3 Hardware **26**
3.1 Gripper . . . 26
3.1.1 Gearbox . . . 26
3.1.2 Fingers . . . 28
3.1.3 Gripper Assembly . . . 30
3.2 Motor . . . 31

4 Discussion and conclusions **32**
4.1 AI methods . . . 32
4.1.1 MinMax algorithms shortcomings . . . 32
4.1.2 Evolution of the perfect algorithm . . . 32
4.1.3 Improvement of database . . . 33
4.2 Improving the computer vision algorithm . . . 33
4.3 Hardware Improvements . . . 35
4.4 Conclusion . . . 36

A Appendix **37**
A.1 Source Code . . . 37
A.2 GUI Flowchart . . . 38
A.3 Atmega . . . 39
A.4 Test of finger strength . . . 42
A.5 Gripper 3D files . . . 42
A.6 Silicone Grease Datasheet . . . 42

B Bibliography **43**

Introduction 1

The objective of this project is to develop a system utilizing a Universal Robot 5 CB3 (UR5 CB3) for playing a game of checkers and improving its performance iteratively. This entails:

- Establishing communication with the robot.
- Detecting a playing surface via computer vision.
- Designing a gripper capable of handling checkers pieces.
- Constructing an artificial intelligence (AI) system capable of playing the game and enhancing its gameplay through iterative analysis of previous games played.

To efficiently achieve these goals, project management methods will be utilized.

Software 2

2.1 Agile Software Development

During this project, agile software development is used, which is an iterative development method. This includes both continuous releases and feedback of the product. The feedback involves both internal dialogue between group members and with the project supervisor.

A Kanban board is set up through GitHub, featuring seven stages: Backlog, To-do, In progress, In review, Done, Main topics, and Done topics. Firstly, any ideas are placed in the backlog, where they are reviewed. If deemed realistic and effective solutions to achieve a functional checkers robot, they are moved to the To-do list. From there, group individuals assign work from the To-do list to themselves. When finished, the work is reviewed by others, and feedback is given. This process is repeated until done. When all work in a specific topic is completed, the main topic is considered done as well.

This method necessitates independent thinking from group members, as the design process at the start of the project is much shorter than when using the waterfall method, as an example. Since no clear end product is defined, measuring progress will also be difficult. To mitigate this issue, smaller subgoals were set throughout the project.

Pros	Cons
Easy to change end-goal	Difficult to measure progress
Group individuals are highly autonomous	Constant interaction between members is required
Results are produced from early stages of the project	
Effective when going through trial and error	

Table 2.1: Pros and cons of using agile project management

2.2 Checkers: The game

The main focus of the project is on the game of checkers. Therefore, a comprehensive explanation of the rules used in this project is necessary, since different rule sets exist:

1. Player checkers can only be moved diagonally on the dark squares.
2. Player checkers can only move toward the opposing player's backline.
3. Only one move can be performed per turn.
4. A jump can be performed, if the opposing player has a checker in one of the squares diagonal to the checker and has a free space to hop to on the other side of the jumped checker.
   Note: A jump can only be performed over one checker.
5. If a checker can jump, a jump must be performed.
6. If a jump has been performed and the moved checker can jump again, then a jump must be performed, overriding rule 3.
7. If a player's checker stands on the opposing player's backline, then it is promoted to a king.
8. If a promotion has occurred, then the player's turn ends immediately, overriding rule 6
9. A king can move toward both the player's and opposing player's backline, and is therefore also able to perform a jump in all directions.
10. A player wins if the opposing player has no available moves.

Taking these rules into account, a simple checkers game can be programmed. By letting the player give an input based on a starting board, as can be seen at the top of figure 2.1, it is possible to determine the validity of the move and update the board based on the given input. Following each move, there is a check, whether the move was an ordinary move or a jump. If the move is a jump, a check is made to see if the same piece can jump again. This game loop can be seen in figure 2.2, where the steps of determining validity and player turn are described. The code for this can be found in A.1.1

```
Player 1's turn:
  1  2  3  4  5  6  7  8
A B     B     B     B
B    B     B     B     B
C B     B     B     B
D    1     1     1     1
E 1     1     1     1
F    R     R     R     R
G R     R     R     R
H    R     R     R     R
Player 1, please enter the coordinates of the piece you would like to move (e.g. A3): c3
Player 1, please enter the coordinates of the square you would like to move to (e.g. B4): d2

Player 2's turn:
  1  2  3  4  5  6  7  8
A B     B     B     B
B    B     B     B     B
C B     1     B     B
D    B     1     1     1
E 1     1     1     1
F    R     R     R     R
G R     R     R     R
H    R     R     R     R
Player 2, please enter the coordinates of the piece you would like to move (e.g. A3): f4
Player 2, please enter the coordinates of the square you would like to move to (e.g. B4): e3

Player 1's turn:
  1  2  3  4  5  6  7  8
A B     B     B     B
B    B     B     B     B
C B     1     B     B
D    B     1     1     1
E 1     R     1     1
F    R     1     R     R
G R     R     R     R
H    R     R     R     R
```

Figure 2.1: The checkers game as seen in the terminal

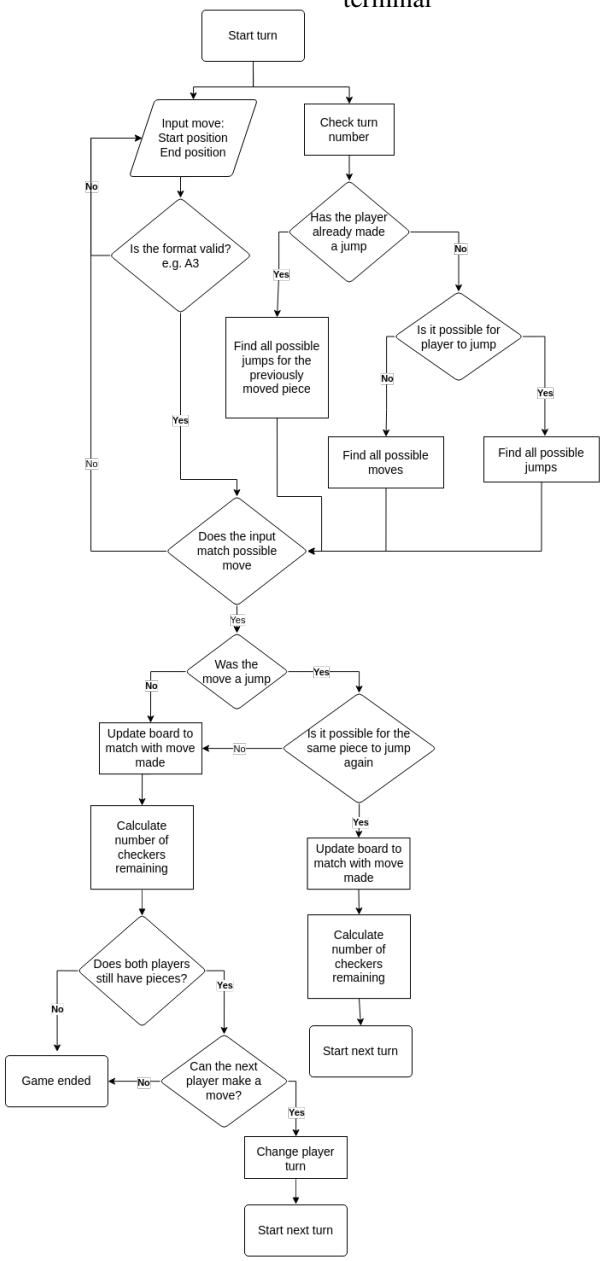

Figure 2.2: Flowchart of the process to check the move made

2.2.1 AI

In order to have an opponent to play against, an "AI" was implemented. Generating games using the AI can induce bugs, which is subsequently used for debugging the program. The following iterations of the AI are as follows:

Random

The random game mode is primarily used to debug the code. By running the code and setting both players to random, a game can be played in 3 milliseconds. This makes it effective to stress test the code, provoking undefined behaviors.

The program uses a function that checks the validity of a player's move by finding all possible moves for the current player. By putting in a delay after every move, it is possible to manually check the validity of a move to find errors linked to this function. Since no input is given when using the random AI, a random number is generated that matches with an index of one of the possible moves.

```
-----------------------
  1  2  3  4  5  6  7  8
A 1     RK    1     1
B    1     1     1     1
C 1     1     1     R
D    1     1     R     1
E 1     1     1     1
F    RK    1     1     R
G 1     1     1     1
H    1     1     1     R
-----------------------
Game took: 3 milliseconds
```

Figure 2.3: Console output from a Timed game

Minimum Maximum

The primary AI opponent of the player is the minimum maximum algorithm, referred to as the MinMax algorithm. The code for this section can be found in A.1.2 This works by recursively calling the MinMax function until a certain depth has been achieved. If a game of tic-tac-toe is considered where an AI uses a MinMax algorithm, it can be illustrated how it functions:

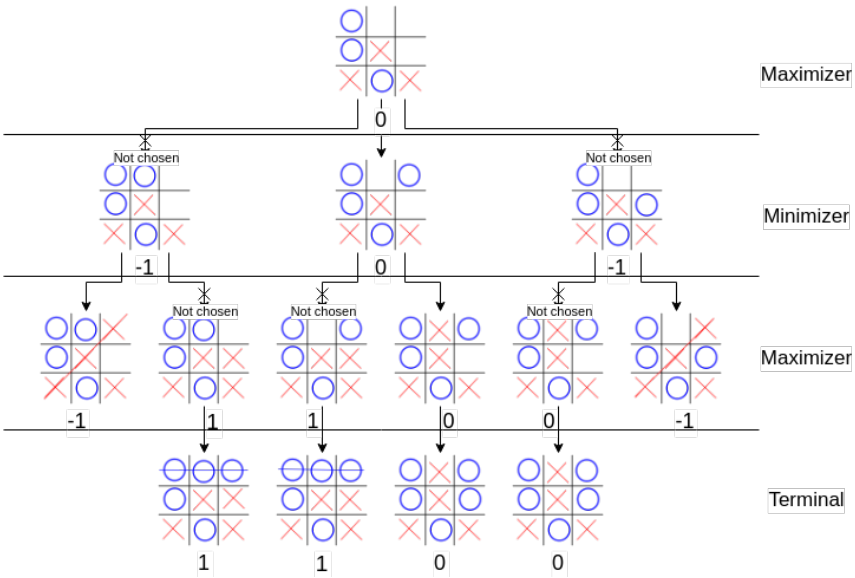

Figure 2.4: Flow of the MinMax algorithm on a game of tic-tac-toe

At the end of each branch in the tree, a value is assigned. A win for the maximizing player, in this case, the blue player, is awarded one point, a draw is awarded zero points, and a win for the minimizing player, the red player, is awarded negative one point. This assignment typically happens at the terminal node, but if a game-ending state is encountered sooner, then the branch ends at its current depth.

After points have been assigned, and depending on the player's turn, a path from the node below is chosen. If it is the maximizing player's turn, then the path with the highest amount of points will be chosen.

In the example showcased in figure 2.4 there is only one possible path to choose for the maximizing player at the bottom, therefore the points are directly transferred up. The minimizing player then chooses the path that awards the minimal amount of points. On the left side of the figure, the best path is to make a winning move and be awarded negative one point, and as can be seen the other path has been crossed out. In the middle, the best path is to prevent a loss by choosing the branch that awards zero points, and on the right a winning move is available. Finally, the current move can be chosen from the three paths.

As the algorithm runs until a game-ending state has been found, and the maximum number of points that can be awarded is zero, it means that the blue player can't win but only draw. The reason this occurs is the algorithm will always choose a path where both players have played the optimal move, therefore it will never choose a path where it can win if this can cause the other player to win as well.
Since tic-tac-toe is a simple game, where there are at most:

$$9! = 362880 \text{ possible game states} \tag{2.1}$$

Which is a low number compared to the number of possible game states in checkers, that has been calculated to be over five hundred quintillion [500,995,484,682,338,672,639] [1].

Therefore, running the algorithm until it has iterated through an entire game isn't an option. This makes it necessary to limit the search to a certain depth to reduce the time needed for the algorithm to calculate a move. Although the search depth needs to be limited, a certain depth is still needed to get a usable result.

Having the algorithm calculate seven moves ahead will make it appear more sophisticated than an algorithm that searches at a depth of three. Furthermore, because of the algorithm's branching of different game states, there will be an exponentially bigger difference in the time needed to calculate a move at larger depths:

```
  1  2  3  4  5  6  7  8
A 1     1     1     1
B    1     1     1     1
C 1     1     1     1
D    1     1     1     1
E 1     1     1     1
F    1    RK    1    RK
G R     1     1     1
H    1     1     1     1
-------------------------
Game played at depth: 3
Game took: 623 milliseconds
```

```
  1  2  3  4  5  6  7  8
A 1     1     1     1
B    1     1    RK    R
C 1     1     1     1
D    1     R     1     R
E 1     1     1     1
F    1     R     1     1
G 1     1     1     1
H    R     1     1     1
-------------------------
Game played at depth: 7
Game took: 124446 milliseconds
```

Figure 2.5: Console output with the time for a full game of two AIs at two different depths

To minimize the time between every move made by the AI, an addition to the MinMax algorithm will be used. This addition is called alpha-beta pruning. The technique stores the value of the best move for the maximizing player as α, and the minimizing player as β. By comparing these two values, the branches can be pruned if:

$$\alpha >= \beta \tag{2.2}$$

The α-value is set only on the maximizing player's turn, and the β-value on the minimizing player's turn. The value set at a node is transferred down and if the condition is met, all branches after the current branch are pruned, meaning the algorithm doesn't have to calculate the values of those branches:

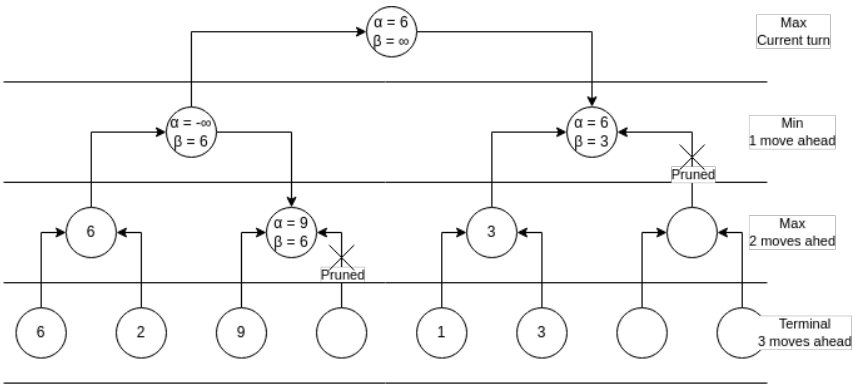

Figure 2.6: Alpha-beta pruning on a MinMax tree

Figure 2.6 shows two branches being pruned because α has a higher value than β. Using this pruning, it is possible to calculate the correct path at a significantly higher rate:

```
  1  2  3  4  5  6  7  8
A 1     1     1     1
B    1     1     1     1
C 1     1     1     1
D    1     1     1     1
E 1     1     R     1
F    1     R     1     1
G 1     1     1     1
H    1     1     1     1
-------------------------
Game played at depth: 5
Game took: 2042 milliseconds
```

```
  1  2  3  4  5  6  7  8
A 1     1     1     RK
B    RK    1     1     1
C 1     1     1     1
D    1     1     1     1
E R     1     1     1
F    1     1     1     1
G R     1     1     1
H    1     1     1     1
-------------------------
Game played at depth: 7
Game took: 79725 milliseconds
```

```
  1  2  3  4  5  6  7  8
A 1     RK    1     1
B    1     1     R     1
C 1     1     1     1
D    1     1     1     1
E R     1     1     1
F    1     1     RK    1
G 1     1     1     1
H    1     1     1     1
-------------------------
Game played at depth: 9
Game took: 694591 milliseconds
```

Figure 2.7: Elapsed time for a complete game between two AIs using the same depth

Evolution

To enable the MinMax algorithm to function properly, a scoring system must be devised. A straightforward approach to such a scoring system involves tallying the number of pieces owned by each player.

Each piece controlled by the maximizing player earns one point, while those controlled by the minimizing player incur a deduction of one point. This system can be refined by assigning the king a higher value than a regular pawn.

Further understanding of the game makes it possible to determine other positions that can be advantageous, such as positions where the player can jump a piece. In this project, ten parameters are weighted. They are as follows:

1. Number of normal pieces.
2. Number of king pieces.
3. Positions where a normal piece locks another up at the bottom, forcing a kill if the enemy piece moves.
4. Positions where a king piece can lock pieces against the edge, forcing a kill if the enemy piece moves.
5. Advancement on the board.
6. Possibility for the enemy player to jump.
7. If the enemy player can't jump, points are awarded if the player can jump.
8. Possibility for the player to jump.
9. Points are further deducted if the enemy player can jump.
10. Depth where the game state has been calculated.

Using an initial guess for the points awarded for each factor, a simulation has been set up to find the optimal value for the parameters. The simulation works by having a tournament of 32 AIs, where four of these are inherited directly from the previous generation without any modification. The 28 other AIs have a 10% chance for each of the parameters to be randomized up to ±30%.

The tournament puts all 32 AIs up against the four best-performing AIs from previous generations. The first round is played against four AIs with no modifications to the points. The win rate is then calculated by awarding one point for a win, one-third for a draw, and zero for a loss. These points are tallied up when all 32 candidates have played four matches against the top four.

To be added to the database containing the best-performing AIs, the win rate has to be at least 10 points. This database contains the AI parameters and the win rate. If the win rate is in the top four of all AIs, then the next generation will play against this newly added AI. The process is repeated for the leading AIs in each generation, once these four have been identified, 28 new AIs are created using the leading AIs as base parameters. The mutation rate, which at first starts at 30%, is lowered by 10% every generation. This means that the second generation will have a:

$$\pm 30\% \cdot 0.9 = \pm 27\% \text{ mutation rate} \tag{2.3}$$

The number, randomly generated, can be up to a 27% change in the parameters from the previous generation. After the evolution program has run for multiple generations, it is possible to check whether the newly evolved AI is doing better than the initial one, by having them play against each other and logging the victories for each of the AIs:

```
  1  2  3  4  5  6  7  8
A 1     1     1     1
B    1     1     1     1
C B     1     1     1
D    1     1     1     1
E 1     1     1     B
F    B     B     BK    1
G 1     1     1     1
H    1     1     1     1
------------------------
Game played at depth: 7
Game took: 13715 milliseconds
The evolved AI wins: 36
The initial AI wins: 21
Draws: 43
```

Figure 2.8: Game score for the AI with new parameters against the initial guess

The parameters that have been found by using the evolution algorithm win a majority of the games when it is playing against the AI that uses the initial guess for the different parameters.

Database

To complement the MinMax algorithm, a new AI is constructed using a MySQL database storing some of all the unique board states and moves between these, thereby avoiding having to compute all moves between turns. The code for the database teaching, training phase, and playing phase can be found in A.1.3.

The information is saved into three tables: one holding all the unique board states with an ID, and two with all the moves attached to a board ID, one for each player, which minimizes query time when searching for a move. In addition to these three tables, two more are created for the AI to function when learning from a game. The first table takes in all the unique board states and checks for duplicate input. The other table stores all the moves made by each player and whether they won or not. An Entity Relationship (ER) diagram of this can be seen to provide an overview of the tables and their contents:

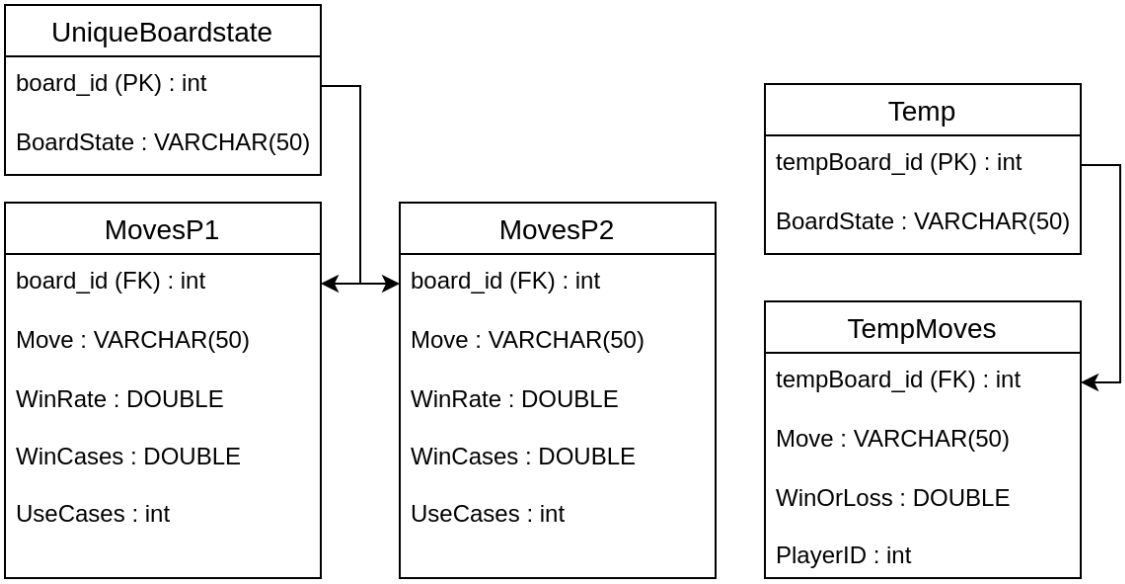

Figure 2.9: Entity relationship diagram of database

Teaching the database The foundation of teaching a database AI involves gathering a significant amount of information about boards and moves. Three different approaches have been taken to obtain the maximum amount of information as efficiently as possible.

1. Using the MinMax discovering feature with the alpha-beta turned off to log all the moves it iterates through in Temp, see figure 2.9, allowing it to iterate through numerous moves in a single game and knowing all moves for each board state logged [1].
2. Using the Random AI to identify all possible moves in the current state and randomly select one, enabling rapid move selection and ensuring that the same moves are less likely to be repeated in each game [2].
3. Creating an additional 2 tables called SimMoves1 and 2, each initially containing 200 rows with zeros. During gameplay, the AI always selects the first move in the first game. Once the game concludes, it increments that choice by one, effectively repeating all the same moves except the last one in the subsequent game. This process continues, moving up the chain with each game [3].

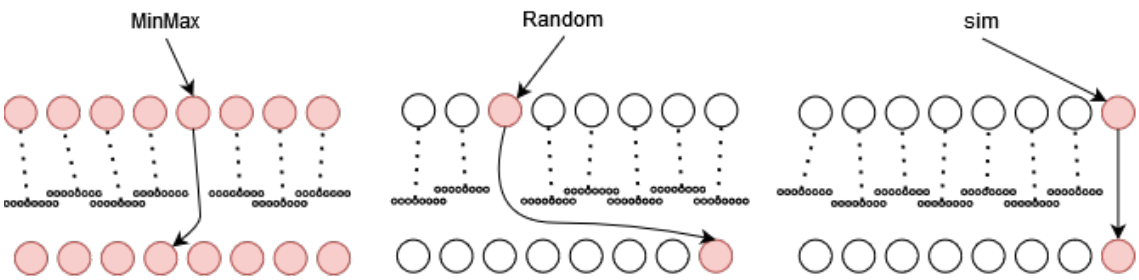

Figure 2.10: AI teaching methods: [1] MinMax method, [2] Random method, and [3] Simulation method

In figure 2.10 it can be seen how the different method iterates through moves. The arrows point to the chosen move, and the red circles are the moves logged into the database. The method ultimately chosen for teaching the AI was the MinMax algorithm. A flowchart illustrating the sequence of events following a game can be viewed for further clarity.

The MinMax has allowed the most optimal learning phase in the time frame of this project. It gives a broad knowledge of board states and all the moves on them. The random method had the downside of only learning 1 move at a time on a board state. While the simulation method, which examines all moves for each board state, would have been ideal, it would have been impractical due to the sheer number of possible board states. Despite that, the MinMax was chosen with the downside being, that it only selects "good moves" according to its algorithm. Thus, it would never go down a path of suboptimal moves and therefore never learn those board states.

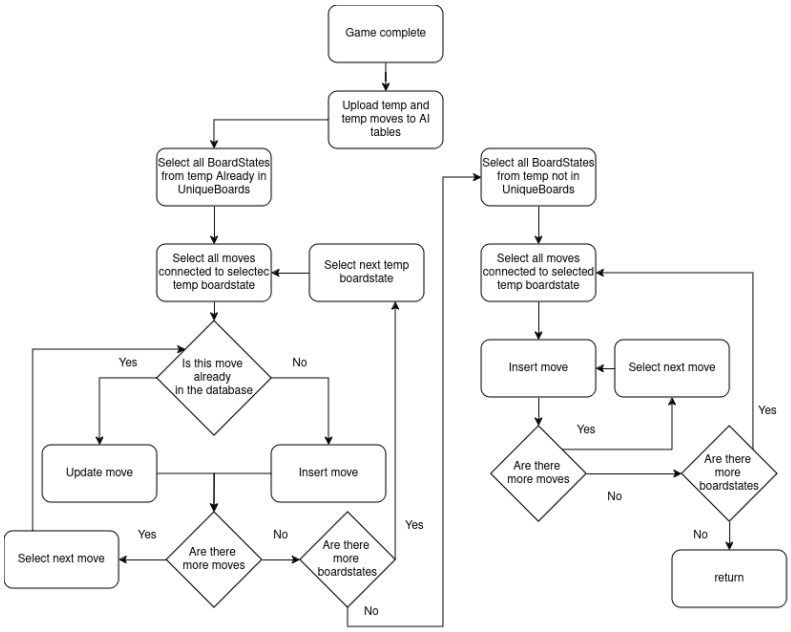

Figure 2.11: Flowchart of the post-processing of game data and transfer to the database

Training phase Since the MinMax feeds the database with moves that were not made in the game, a training phase is required. In this project, three ways of training have been considered:

1. Playing against MinMax, letting it play until it wins the majority of the time.
2. Playing against itself, updating the win rate of moves, making it discover different paths to wins.
3. Playing against a human until it wins the majority of the games.

The third method was not selected, due to its time inefficiency, but is used to test the capabilities of the database AI after training. Between the first and second method, the MinMax is picked since it gives better general training.

Playing phase When the AI is required to make a move, it checks if the current board state is already in the database. Following this, the database is scanned for all moves made by the current player, sorting them based on the win rate in descending order, and picking the move with the highest win rate. If one of the following scenarios occurs, a move will instead be made by the MinMax:

1. The database has no knowledge of this boardstate.
2. The database has no moves for this player on this boardstate.
3. The has with the highest win rate has a win rate under 15%.

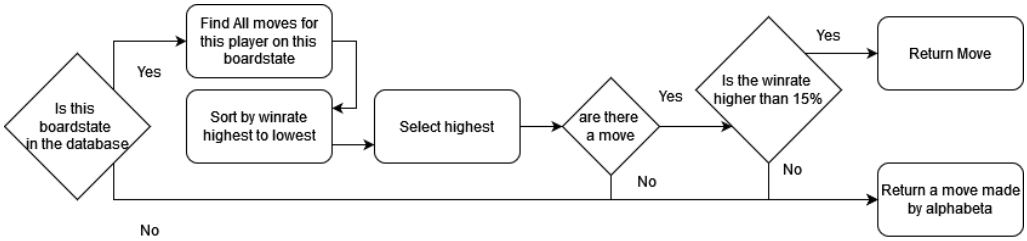

Figure 2.12: Flowchart of the playing phase for the database AI

Pros and cons

Database AI		MinMax AI	
Pros	Cons	Pros	Cons
Quick moves	Long learning phase	No setup is needed after implementation	Large depths = slow moves
Capable of knowing every board state and move		Easy to change parameters	Uses approximated point system
Capable of improvement through learning			Inaccurate during endgame
Compatibility with other algorithms			

2.3 Communication with the robot

The chosen method of communication with the robot is Real-Time Data Exchange (RTDE). This has been selected based on the requirements for the movements and data needed from the robot. The project utilizes the library made by Universal Robots for RTDE communication [2], making it possible to get the force the robot experiences in different directions and integrate this directly into the code. A table with the pros and cons of the two communication methods can be seen below:

UR RTDE		MODBUS	
Pros	Cons	Pros	Cons
Ease of use when set up	Complex setup needing real-time kernel	Standard communication method	Limited information on registers
Fast communication	Resource-intensive for both controller and computer	Already implemented on CB3 controller	Slower communication
Smooth incorporation into C++ code	Less compatible with the older CB3 robot		Needs comprehensive setup on controller
Communication in real-time			
Able to receive data from the robot			

2.4 ATmega

To pick up and relocate a checkers piece, a gripper has been designed, as described in section 3.1. The motor in the gripper is controlled with an ATmega644PA microcontroller. The microcontroller requires programming to establish communication with the PC, allowing commands to be sent from the PC to control the motor. Subsequently, a program is integrated into the main code on the PC to enable communication with the microcontroller. This chapter outlines the program run on the microcontroller and the PC program.

2.4.1 Communication

To facilitate communication, the project has adopted the UART communication protocol, a Full-Duplex asynchronous serial protocol. Full-Duplex enables simultaneous sending and receiving of data, facilitated by separate pins. Asynchronous operation means that there is no shared clock cycle, necessitating the definition of a frame, which will be elaborated on later. In serial protocol, data is transmitted one bit at a time. Contrarily, parallel communication would entail setting up eight wires between sender and receiver for sending eight bits simultaneously.

UART communication

When sending messages without a common clock cycle, there is no way of reading a message, since a message consists of logical low and high impulses called bits that can have a value of either 1 or 0. Therefore, a protocol must be set up to define the speed at which the message is sent and how the UART frame is constructed.

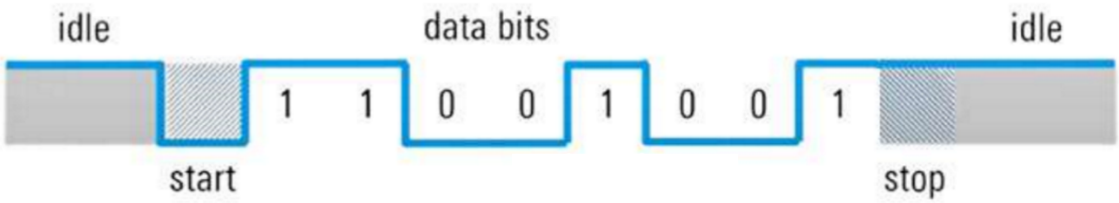

Figure 2.13: Illustration of a UART frame [3]

A UART frame consists of 3 elements. The first element involves defining the start and stop bits. When the controller is idle, the line maintains a logical high. Therefore, to initiate a signal, the start bit transitions to a logical low. The signal concludes when it returns to a logical high. The second element pertains to the frame size, indicating the number of bits within the frame. In this project, it will be set to 8, making each message an 8-bit package. The third element is the optional parity bit, which is used for error detection. However, for this project, parity bit usage is unnecessary and will be disabled.

With the frame constructed, the next step is to establish the baudrate, which determines the rate of bits transmitted per second. A baudrate of 9600 will be employed in this project, as it is commonly used when speed is not critical, as is the case here. With a baudrate of 9600 and frames consisting of 10 bits (including start and stop bits), messages will be sent approximately every 1 ms.

2.4.2 ATmega644PA program

The ATmega has been programmed in C-language using Microchip Studio. Microchip Studio is an IDE for programming microcontrollers. Besides that, the utility program AVRDUDE has been used to upload and test communication with the microcontroller. The program can be found in A.1.4.

The ATmega has two registers for reading and writing: Registers 0 and 1. These registers are connected to separate ports on the PCB. A USB-B port on the PCB is connected to register 1 for reading and writing. Register 1 will be utilized because of the provided USB-A to USB-B adapter.

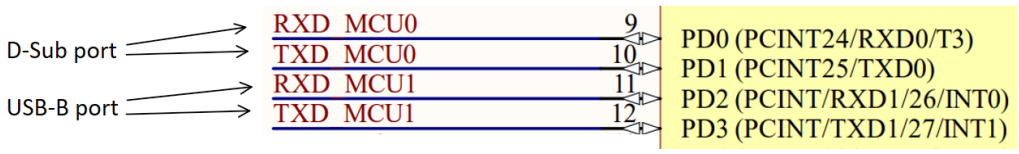

Figure 2.14: Ports connected to read and write pins

This register will be configured according to the frame and baud rate explained above, which can be found in A.3.1.

In addition to configuring the frame and baud rate, reading, and writing functionalities have been enabled, along with reading interrupts. This allows the program to instantaneously react upon receiving a message. The writing function is set in polling mode within a function, as it is only needed at specific times and should only be executed when required.

PWM and ADC

PWM signals regulate motor speed via a motor driver, while ADC signals enable utilization of the board's current sensor, allowing for monitoring of motor power consumption.

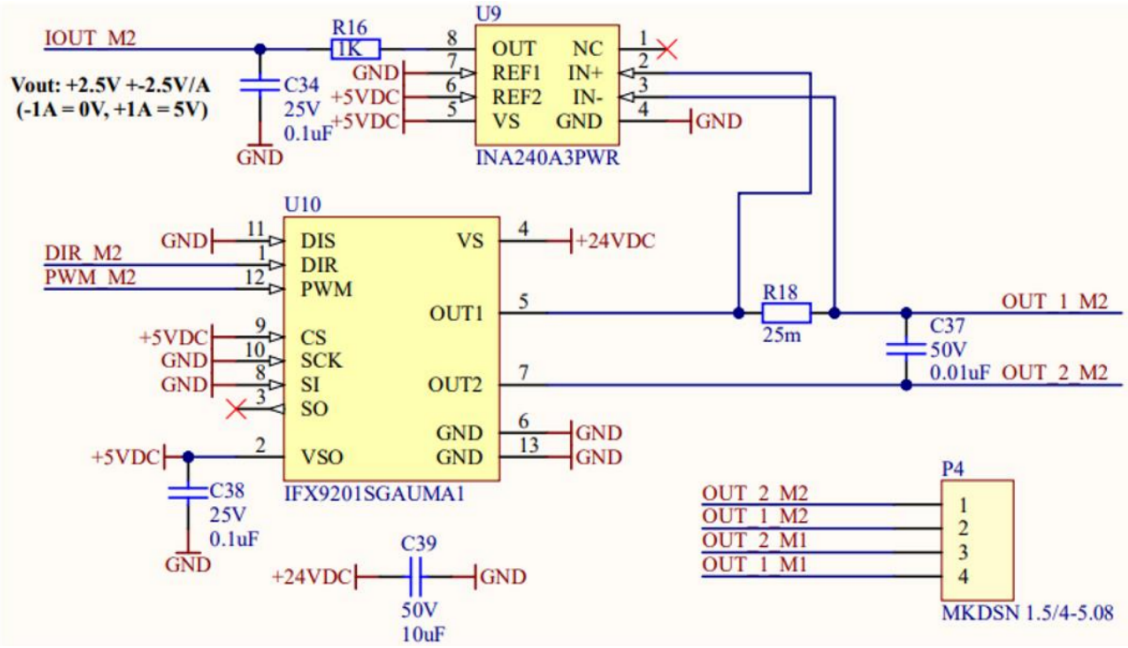

Figure 2.15: Motor driver and current sensor circuit diagram

To control the motor, both direction (DIR) and PWM input are utilized. The PWM signal is connected to pin PD4 of the ATmega. The direction pin, linked to PIN PC3, switches motor direction when needed, such as transitioning from open to closed, and vice versa.

During motor operation, the current sensor monitors the current supplied to the motor, generating a voltage signal proportional to the current input. The voltage output scales are shown in the top-left corner of the figure. This output is directed to the ADC channel on pin PA1, where the voltage level is analyzed and a successful piece pickup or full opening is determined.

PWM

Pulse Width Modulation (PWM), modulates a digital output to produce an average analog signal between the highest and lowest voltage levels. The average analog signal, the duty cycle, is a percentage. For instance, a 50% duty cycle means the output pin alternates between logical high and low at equal intervals, resulting in a signal that is half of the maximum voltage. This technique can be employed to operate the motor at 50% speed.

The pins connected to the motor drivers are configured using timer register 1. Therefore, to generate a PWM signal for the motor drivers, register 1 will be utilized. It is configured as a non-inverting fast PWM with a top value of 100. Details on setting this up can be found in the A.3.2.

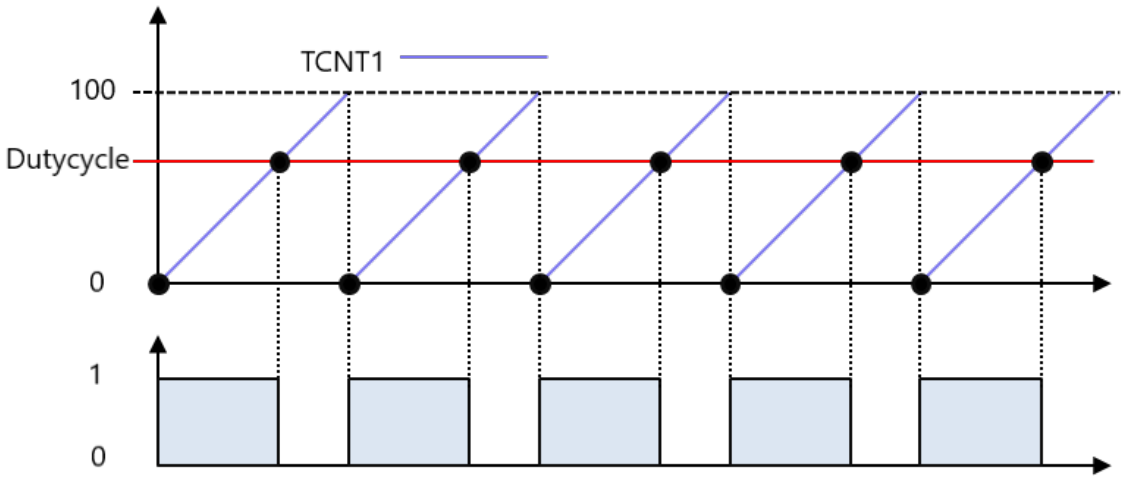

Figure 2.16: Illustration of the PWM signal

A non-inverting fast PWM signal raises the pin when the compare value exceeds the timer counter and resets it when the timer counter surpasses the maximum value.

During piece gripping, the duty cycle is set to 60% to avoid difficulties in stopping when running at full speed and detecting a piece. However, during opening, the duty cycle is set to 100% motor speed, as it relies on a resistant wall to stop it.

ADC

Analog to Digital Converter (ADC), is a crucial tool for the gripper's ability to control the motor. This conversion is necessary because the output voltage from the current sensor is an analog signal ranging from 0V to 5V, whereas a digital signal is either 0V or 5V. This conversion is achieved by representing the analog signal's voltage level as a binary value. The ADC register is a 10-bit register, with a reference voltage of 5V. Therefore, the voltage can be divided as follows:

$$5V/2^{10} \approx 0.0049V = 4.9mV \tag{2.4}$$

Each increment in the binary value of the ADC register corresponds to a change of approximately 4.9 mV.

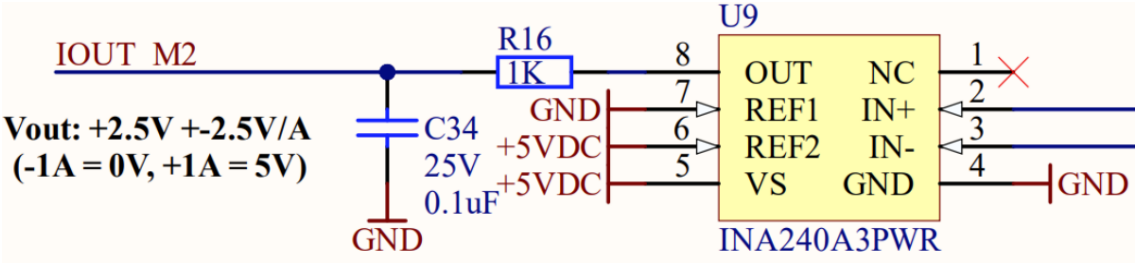

Figure 2.17: Current sensor circuit diagram

The current sensor maintains a base value of 2.5V when the motor consumes 0A. Depending on the motor's direction, the voltage either increases or decreases, ranging from -1A (outputting 0V) to 1A (outputting 5V). Thus, each 1A change corresponds to a 2.5V change in output. Converting this to mA, each 1mA change corresponds to a 2.5mV change. The ADC signal can detect 5mV changes, allowing for a resolution of about 2mA on the output before detection.

Since motor driver 2 is utilized, ADC readings will be taken from ADC1. This configuration is set up in the ADMUX register along with a reference pin. The ADC is then activated in the ADCSRA register, and the clock frequency is adjusted to ensure F_{ADC} falls within the recommended range of 50kHz to 200kHz. Details of this setup can be found in A.3.3.

Once a conversion is completed, the two registers ADCH and ADCL, comprising the 10-bit ADC register, store the voltage reading. This information detects significant current increases not associated with normal operation.

2.4.3 PC program

Termios is used for PC to ATmega communication. This is a Linux terminal I/O interface, that facilitates connection to a device via USB, which allows for specifying the frame, speed, and data reception method.

The PC program can be found in A.1.5. In the program, a class named ATmegaCom has been created. When an instance of ATmegaCom is declared, its constructor establishes a connection at the designated USB port. This configures certain flags of the termios structure to ensure that the received data is interpreted as raw data, without any modifications. Additionally, the receiver is configured to wait for at least 1 byte before proceeding after being called upon. This ensures that the computer waits for the ATmega to send a message, indicating that it has either grasped a piece or fully opened, before continuing execution. Finally, a baud rate of 9600, matching that of the ATmega, is specified. Details of the complete setup can be found in A.3.4.

```
class AtmegaCom
{
private:

    int _serial_port;

public:
    AtmegaCom();
    void sendmsg(char InputNumber);
};
```

Figure 2.18: ATmegaCom class

```
void AtmegaCom::sendmsg(char InputNumber) {
    char msg[] = {InputNumber};
    write(_serial_port, msg, sizeof(msg));

    char read_buf [1];

    int n = read(_serial_port, &read_buf, sizeof(read_buf));

}
```

Figure 2.19: The sendmsg method from the ATmega class

The sendmsg method is utilized when the PC needs to communicate with the ATmega. Initially, the input is loaded into a character array which is sent to the ATmega. The program remains suspended until an answer is received, allowing the main program to resume.

2.4.4 ATmega operations

When sending a message to the ATmega, two functions can be invoked based on separate values.
If the ATmega receives an ASCII character of 6, a decimal value of 54, it will close the gripper. If the value is the ASCII character of 8, a decimal value of 56, it will open the gripper.
Because these functions appear almost identical, only the close gripper function will be elaborated upon:

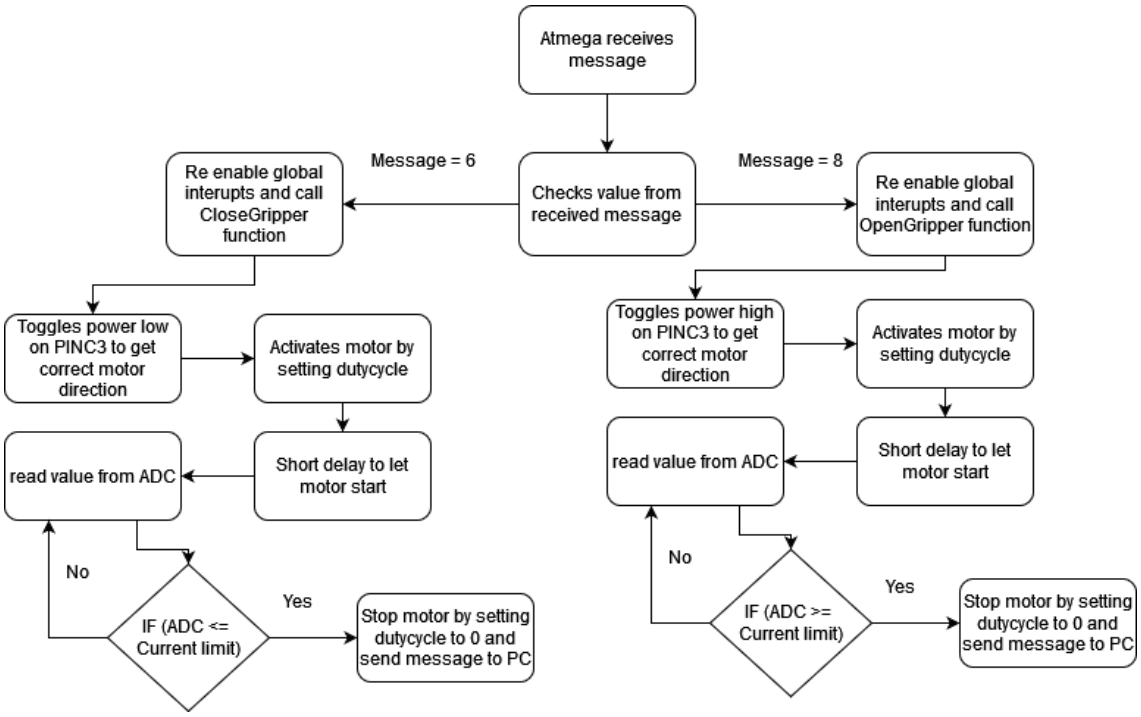

Figure 2.20: Flowchart of the ATmega program

```
void CloseGripper(){
    PORTC = (0 << PORTC3);
    DDRC = (0 << PORTC3);
    dutycycle = 60;
    _delay_ms(300);
    int ObjectHit = 0;
    while (ObjectHit == 0) {
        ADCSRA |= (1 << ADSC);
        while ((ADCSRA & (1 << ADIF)) == 0){}
        if(ADC <= 461){
            ObjectHit = 1;
        }
    }
    dutycycle = 0;
    swrite('7');
}
```

Figure 2.21: Close gripper function from the ATmega program

When calling CloseGripper the motor's direction is set. Then, the motor is activated with a duty cycle of 60%. Due to the motor's increased power demand during startup compared to normal operation, a short delay is necessary. Subsequently, a loop continuously compares the ADC register against a specified threshold. When the ADC value falls below the specified threshold, the loop breaks, deactivating the motor and sending a message to the PC to notify it to resume normal operation. Since the ADC value is a 10-bit register ranging from 0 to 1023 in decimal, and the current sensor outputs 2.5 volts at 0A, the default ADC value for 0A is 511. Each bit corresponds to approximately 4.9mV and 2mA. The designated value of 461 was determined by initially logging voltage readings through software. These readings ranged from 462 to 470, so 461 was chosen just below normal operating usage. The value of 461 corresponds to a mA consumption of:

$$\Delta bits = 511 - 461 = 50 \tag{2.5}$$

$$I_{consumed} = \Delta bits * I_{bit} \approx 50 * 2mA = 100mA \tag{2.6}$$

This corresponds with the motor datasheet, stating a nominal current of 90 mA. Therefore, the current consumption of 100mA from the motor must mean it has met an increased resistance, i.e. an endstop.

The only distinction between CloseGripper and OpenGripper is that the motor direction is set oppositely, and it operates on a 100% duty cycle instead. The motor direction is reversed by switching the polarity of the motor. This means the current switches direction as well, and the ADC readings are higher than 511.

2.5 Computer vision

To make the playing experience intuitive for the user, computer vision is used to detect the checkerboard. Therefore, the user is not required to teach the robot where to play, as this is automatically calibrated via the camera. The player must place the checkerboard on the established playing surface, place the camera in the marked position, and place the three calibration pieces in their assigned positions. Afterward, the player can move their pieces on the board, and the robot will be able to play accordingly. The code for this process can be found in A.1.6.

2.5.1 OpenCV

To implement computer vision, the C++ library "OpenCV (Open Computer Vision)" was used [4]. This includes built-in functions to help with image processing. Although OpenCV would support detection in 3D, the playing surface will be considered a 2D surface when regarding computer vision during this project.

Circle detection

The initial function of the program is to find every circle in the given image. This is done through OpenCV's built-in function called "HoughCircles()". For every circle detected, its coordinates are stored. All circles are then blacked out. The playing pieces are blacked out because they will disrupt the pattern that the program recognizes as the checkerboard. Since the table surface by the robot has holes, they will also be detected as circles, which is unwanted. Therefore, a test is later run to check which circles are playing pieces. This is done by comparing the location of the circle with the expected location of a piece.

Checkerboard detection

The next step comes in detecting the checkerboard. A built-in function of OpenCV called "findChessboardCorners()" is utilized. This function searches the image for any chessboard pattern. The inner corners of the board are then detected, and their coordinates are saved. These coordinates are saved in a vector with 49 inputs. The corners of the chessboard are always stored at the index of 1, 7, 42, and 49. By having these four corners always be at the same index in the vector, it is possible to easily calculate the location and orientation of the checkerboard.

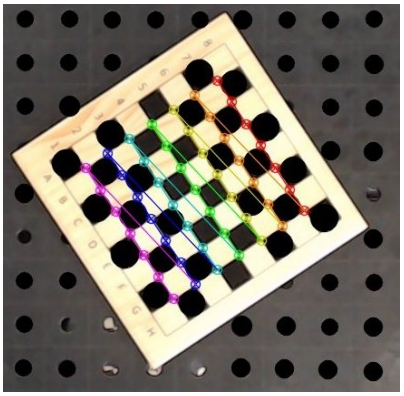

Figure 2.22: Detected chessboard as seen from the camera.

2.5.2 Conversion to real position

At this point, functions to find playing pieces and the checkerboard have been established. However, the coordinates calculated by these functions give a pixel on the image at which it is located. Therefore, to give the UR5 robot the correct input, the pixels have to be converted to meters. This can be done since the size of a square on the board is known and the pixel values of the corners are also known. By dividing the length between corners by the amount of pixels in between, the relation between pixels and real-life distance can be found. This factor is used to recalculate all values from a pixel value to a real position.

Furthermore, a frame is created by placing different colored circles at three edges on the table. This frame can be seen in figure 2.23. When choosing colors, emphasis was put on making them as close to the three primary colors as possible. This ensures the biggest color contrast between the circles.

After detecting these colored circles, their coordinates are used to define the x- and y-axis of the frame in the C++ code, which is used in section 2.6.

With the coordinates from the checkerboard corners, a frame is calculated here as well. Although the coordinates of the four outer corners are known, it is to be determined which corner is the origin of the frame on the board. To define this frame, an initial guess of the relevant corners is made. It is then checked if a checker is placed in the origin, the x-axis, or the y-axis. If a checker is placed in origin, then the initial guess is correct and is used to calculate the frame used in section 2.6. If a checker isn't found in origin but rather in the x- or y-axis, another guess is made for a new origin.

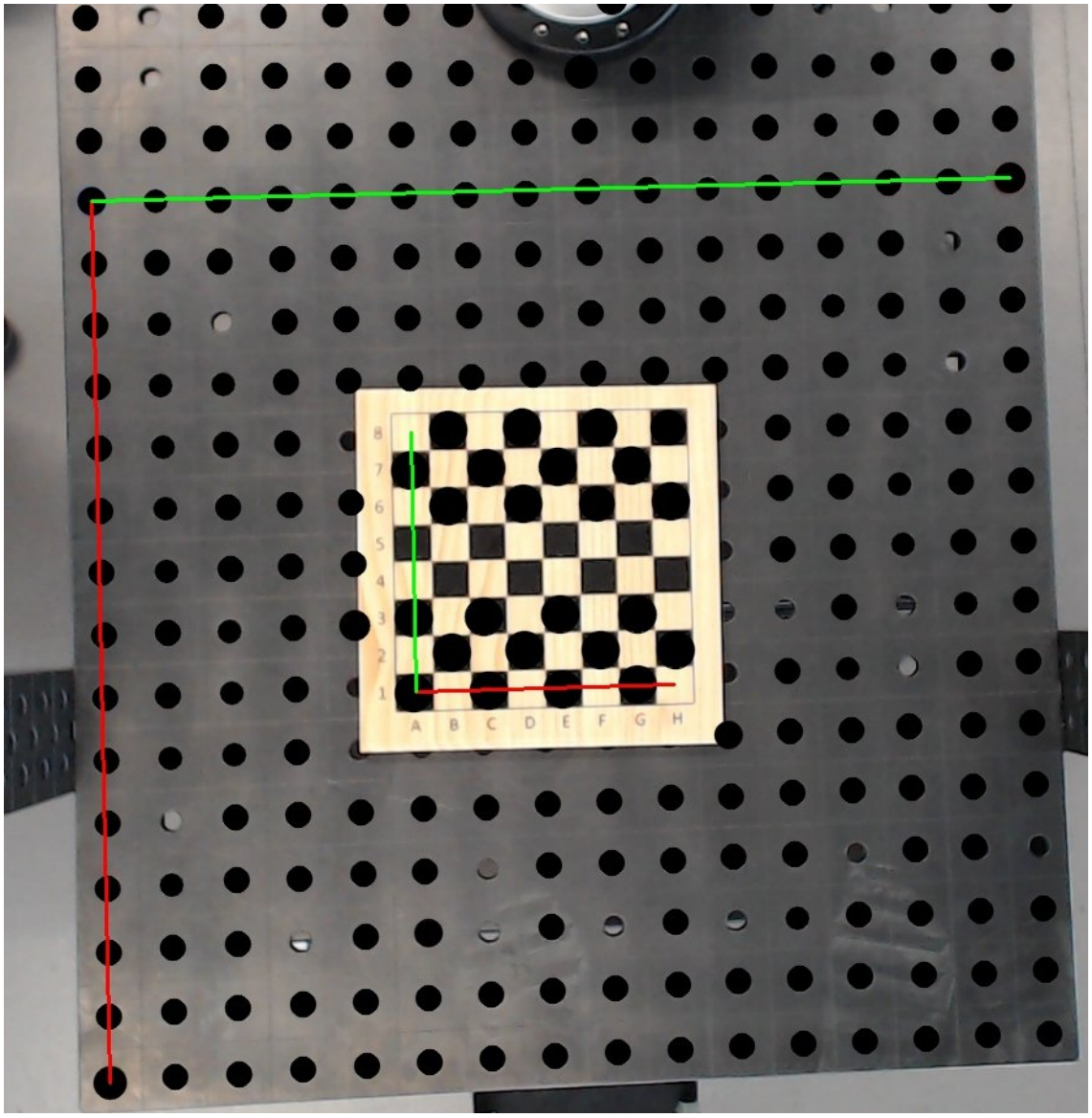

Figure 2.23: Frames on playing surface and board as seen from the camera.

This means the first player isn't always the same color, depending on how the camera detects the frame. After the correct origin has been determined, the color of the checkers on each side is determined and used for detecting the checkers.

2.5.3 Updating the position of the checkers

In the previous section, the starting position of the board and the color of each side's checkers were calculated, and a new origin was placed at the edge of the board, from which positions will now be calculated. The next step is to update the position of the board every time the player physically moves a piece. Every time the player makes a move, a button is pressed on the computer to continue. Afterward, a picture of the board is taken, to which the circle detection is used again. When the pieces on the board are found and matched with their color, it is possible to match the new board with the previous board to detect where a change has occurred and thereby find the move.

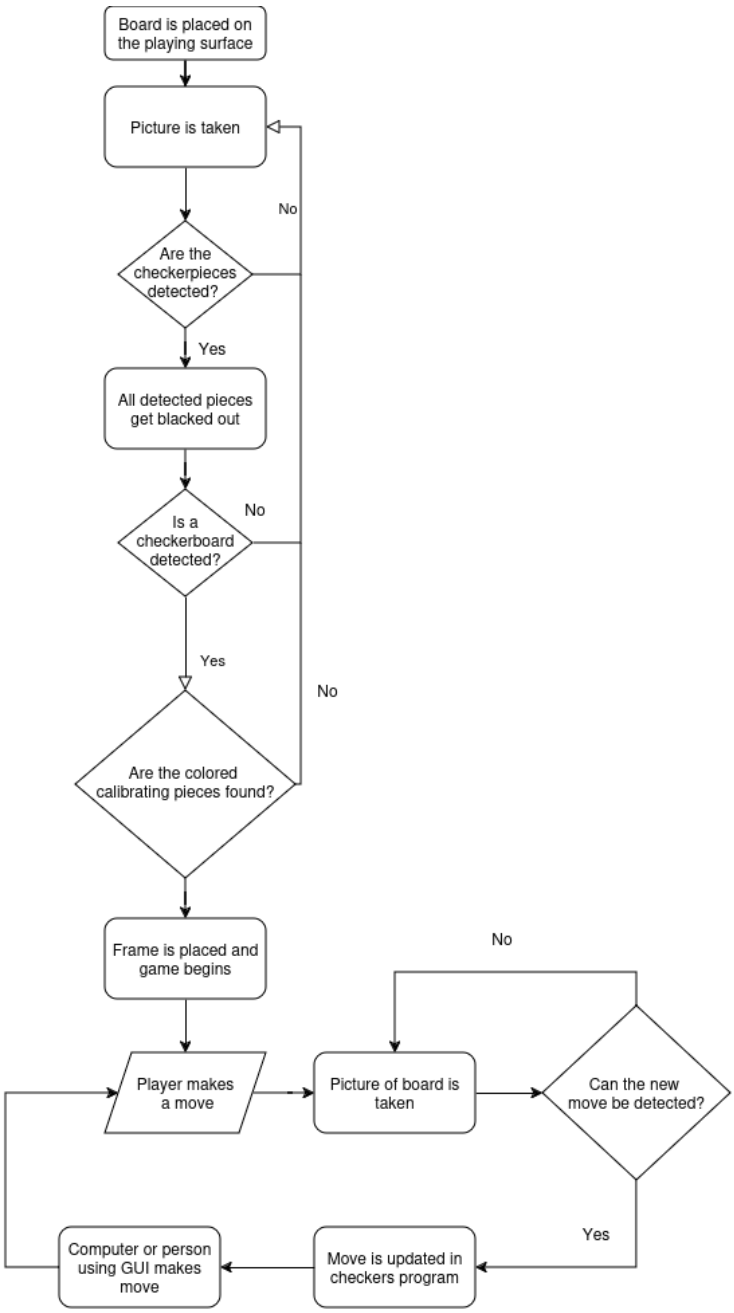

Figure 2.24: Flowchart of computer vision setup and playing process

2.6 Kinematics

Integration of a camera requires a method to convert the image into positions that are possible for the robot to interpret. This is achieved by establishing a common world frame on the table, visible to both the robot and the camera. The frame, predetermined within the robot's programming, is identified by the camera using the calibration circles. The relationship between these elements is illustrated below:

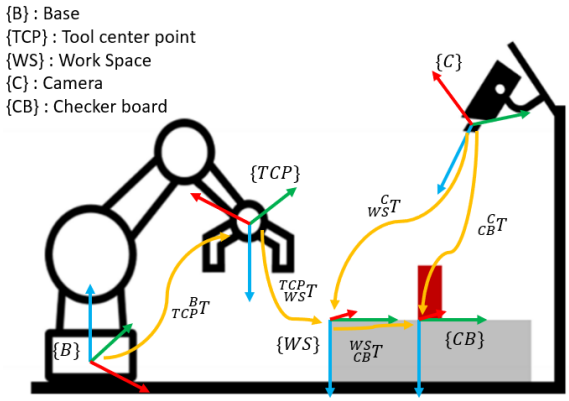

Figure 2.25: Illustration of all kinematics frames and transformations

Initially, three points are generated in the robot base frame using the robot's teach mode to construct the WS frame. These coordinates are then incorporated into the code, eliminating the need for teaching mode during setup and reducing the user's workload. The points establish an origin, a point in the x-direction, and a point in the y-direction.

By normalizing the vectors, a rotation matrix can be constructed. A translation vector can be made that is identical to the origin point, representing the distance between the base of the robot and the new frame. Combining that rotation and translation, a transformation matrix can be constructed as follows:

$$ {}^{B}_{WS}\mathbf{T} = \begin{bmatrix} {}^{B}_{WS}\mathbf{R} & {}^{B}\mathbf{P}_{WS_{org}} \\ 0x3 & 1 \end{bmatrix} \tag{2.7} $$

A similar process is undertaken for the camera. The points that are used for this, have been found as explained in the openCV section 2.5. From this we can get two matrices:

$$ {}^{C}_{WS}\mathbf{T} \quad \& \quad {}^{C}_{CB}\mathbf{T} \tag{2.8} $$

The aim is to determine the matrix ${}^{WS}_{CB}\mathbf{T}$ which can be calculated by:

$$ {}^{WS}_{CB}\mathbf{T} = {}^{C}_{WS}\mathbf{T}^{-1} \cdot {}^{C}_{CB}\mathbf{T} \tag{2.9} $$

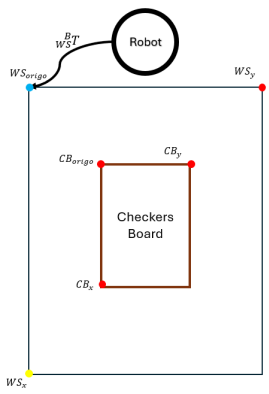

Figure 2.26: Illustration of robot work surface relative to robot base frame.

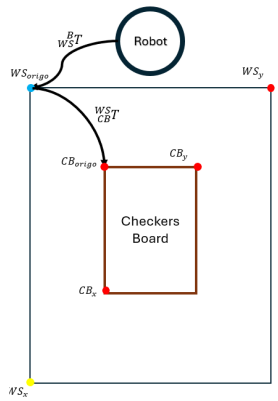

Figure 2.27: Illustration of checkerboard relative work surface frame.

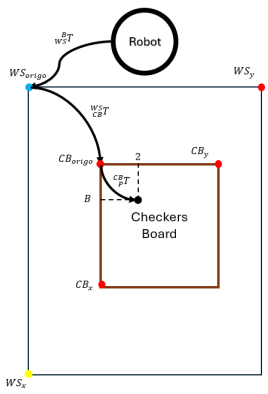

Figure 2.28: Illustration of a checkers piece relative to checkerboard frame.

After that, a rotation of the camera's world frame is needed to make it identical to the robots. This is caused by the camera's frames having the z-axis pointing downwards, whereas the robot points upwards. Therefore, a rotation of 180° will be made around the y-axes and then 90° around the z-axes, as can be seen in figure 2.29.

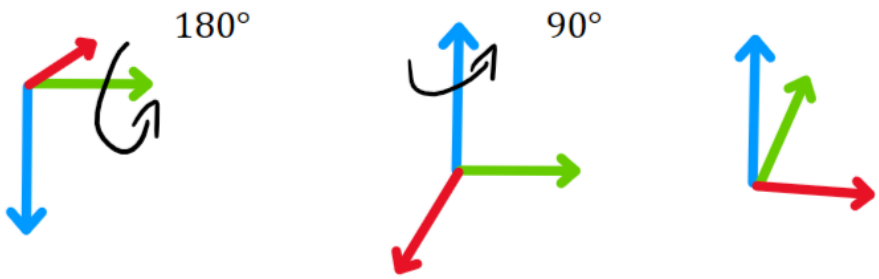

Figure 2.29: Rotation of camera frames

After rotating the matrix, the orientation matches the orientation of the world frame seen from the robot, with the z-axis upwards from the table. This allows for a coupling between the robot's and the camera's work surface, since the frames are made of points placed in the same physical positions. This allows for the following equation to determine the board seen by the robot:

$$ {}^{B}_{CB}\mathbf{T} = {}^{B}_{WS}\mathbf{T} \cdot {}^{WS}_{CB}\mathbf{T} \tag{2.10} $$

This transformation matrix tells the robot the origin of the checkerboard. A Transformation matrix from CB to P is created using the translation of the piece as seen from the checkerboard to provide the robot with the position of a checker. The translation is calculated at the checkerboard's origin and moves the required amount of checker squares. For example, a move to B2 would result in moving 2 squares in the x direction and 2 squares in the y direction.

$$ {}^{WS}_{P}\mathbf{T} = {}^{WS}_{CB}\mathbf{T} \cdot {}^{CB}_{P}\mathbf{T} \tag{2.11} $$

Then, using this matrix, it is possible to calculate the position as seen from the base of the robot:

$$ {}^{B}_{P}\mathbf{T} = {}^{B}_{WS}\mathbf{T} \cdot {}^{WS}_{P}\mathbf{T} \tag{2.12} $$

After the coordinates of the checkers have been determined, it is possible to find relevant z-coordinates by having the robot check when it collides with a checker, the board, and the designated graveyard. This is done by continually measuring the force that the robot experiences in the z-axis by using the built-in function "getActualTCPForce". When this force exceeds a given value, the current position of the TCP is logged and the value can then be used as the height of the checker, board, or graveyard. The checker's size is calculated by subtracting the height of the checker from the height of the board.

The relevant code for this section can be found in A.1.8

2.7 GUI

In order to play the game in an intuitive and manageable way, a graphical user interface (GUI) was implemented. The purpose of which will be to make it easier to:

1. Declare a move in the software.
2. Represent the current game state in an easily interpretable manner.
3. Showcase relevant information, regarding move possibilities and earlier moves.

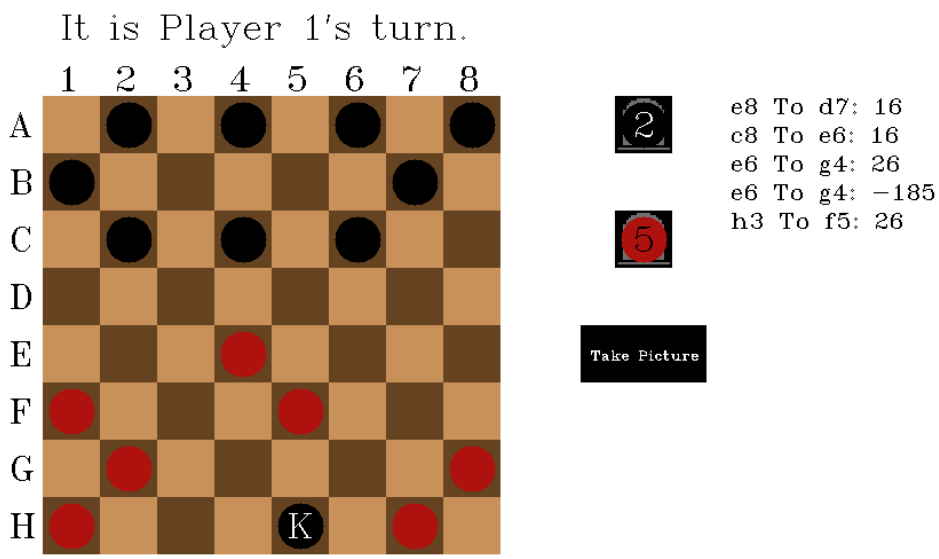

Figure 2.30: Screenshot of GUI

2.7.1 GUI framework

To create an interactive interface, a GUI framework was needed. Despite its lacking user-friendliness and graphical capabilities, the "HighGUI" framework is used, as it is included in the "OpenCV" library, and would therefore ensure compatibility between the various functions and data templates used in the library. The code can be seen in A.1.7

2.7.2 Base code

The interface is built upon three elements: a window, that acts as the interface, an image (contained within the window), which can be altered to represent the saved information in the code, ex. checkers piece locations, and lastly a mouse-callback function, that takes the mouse position and actions as input. By combining these three elements and the information stored in the code, it is possible to make an interactable interface.

The image is initialized as a part of the "Mat" class and works like a large matrix. After initialization, it is possible to copy information from other "Mat" class instances with the "copyTo()" function. Functions like "circle()" and "rectangle()" can also be called in order to draw a circle or rectangle respectively. The rectangle function has a wide variety of uses but has primarily been used to generate the board surface, through an implicit call with the overloaded parentheses operator from the "Mat" class, and giving an instance of the "Rect" class as an argument. By doing so, a rectangle, corresponding to the information

saved in the “Rect” class instance, is drawn on the image. After pushing these instances of the class into a vector, it is also possible to easily reference and access the information in a specific board piece by iterating through the vector, given a specific coordinate set. The same method has been used to reference the checkers pieces, though with the class “Circle” which inherits from “Rect”.

As mentioned earlier, a mouse-callback function is used to track the mouse and its actions, called “events”. The position of the mouse cursor is continuously recorded, but by adding a conditional that depends on the event argument of the function, it is possible to lock in a specific value when the user wishes to do so. The locked-in value can then be used to check whether the click was within the bounds of the constructed objects. This is determined using the “contains()” function, which returns true if a coordinate set is within the bounds of the class instance. If the click is within a board piece, the program checks if there is a checkers piece in the same location. If so, the piece is highlighted, and its ID is saved for later use. If a jump is possible for the piece, the jump location will then be colored red, automatically, as shown in figure 2.31. The flow of the program can be seen on the flowchart in A.2

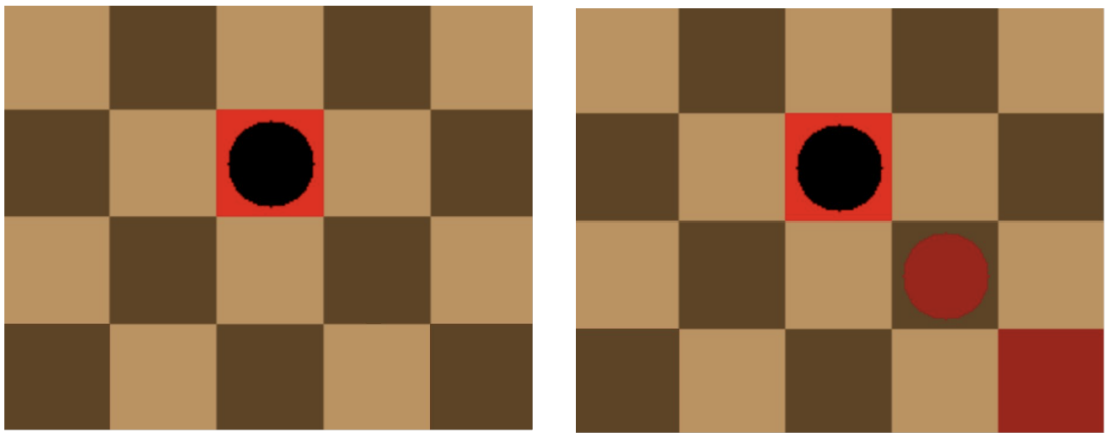

Figure 2.31: Screenshot of select and target location highlighting

2.7.3 Compatibility with the game code

To ensure all moves made during the game are in accordance with the game rules, compatibility with the “game code” has been built. The game code references specific checkers pieces and spaces according to the algebraic notation, ex. “A8”, as one would when playing the game in real life, while the GUI uses a position composed of an x- and y-position. To solve this issue, two conversion functions have been implemented, one for converting each way.

To convert from an algebraic notation set to a position, the numeric position of the first coordinate in the alphabet is needed, after which the value can be multiplied by the height of the board surface pieces plus the board offset along the x-axis. The same is done for the second coordinate but along the y-axis.

To convert from a position to algebraic notation, the opposite is done, i.e. subtract offset and divide by the height or width of the playing surface pieces. The functions are taken into use when a player makes a move and the move’s validity has to be verified. This is done by converting the coordinates of the start and end position, and then comparing the converted information using the “move()” function.

When playing against the algorithm and database, it is necessary to convert from the algebraic notation to a valid position, to update the game state. Hereafter, it is possible to follow the same procedure as one would for a PvP move.

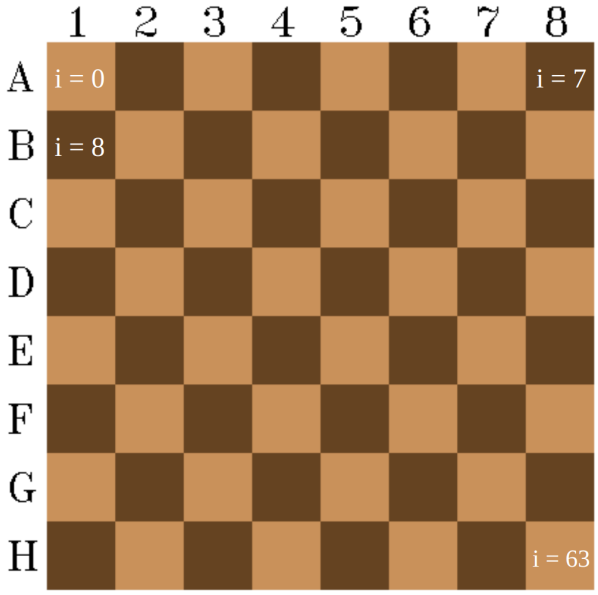

Figure 2.32: Illustration of board ID representation

Before checking whether a move is valid, the user should determine what move they would like to make, which is restricted by the rules of the game. To help visualize the rules, it was determined it would be helpful to highlight locations where it is possible to jump to, when a kill is possible. To do this, a lot of checks are performed where there is taken advantage of the way the playing surface is generated and stored. Each surface piece has a specific value that can be used to reference it, starting from "0" in the top left corner and ending at "63" in the bottom right corner, as seen in the figure 2.32.

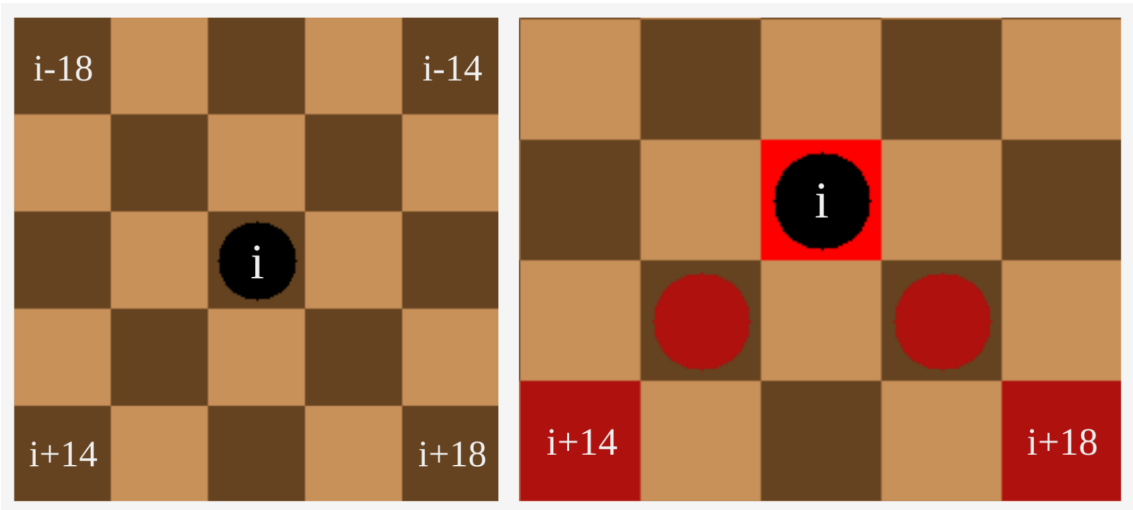

Figure 2.33: Illustration of killing move possibilities on the GUI

If a black checkers piece is selected while on an arbitrary surface piece with the reference ID "i", it is checked if a jump is possible.

If a jump is possible, it is checked whether there is an enemy checkers piece at the possible locations, if there is, the corresponding reference ID, "i+14" or "i+18", is used to highlight the location, as illustrated in figure 2.33. If the checkers piece is red, instead of black, the same method is used but in the opposite direction and therefore a subtraction from the current position "i". If the checkers piece is a "king", then the piece can move in both directions, and both positive and negative reference points are considered.

Hardware 3

3.1 Gripper

To make the robot interact with the world, it is crucial to design an applicable end-effector. In the case of playing checkers, there is a need for a gripper, that is capable of picking up thin cylindrical disks.

To accomplish this, a motor can be used to actuate the gripper fingers that will close around a checkers piece.

The goal of the design is to have a simple and robust gripper that can withstand light impacts with the work environment and is capable of picking up multiple checkers pieces at once. To optimize the design and designing process, the main manufacturing technique of this project is chosen to be additive manufacturing, also known as 3D printing. This process is cheap, it allows fast prototyping and advanced design features such as complex geometry and embedded hardware. The 3D printers used for the project were made available through the DevLab student organization.

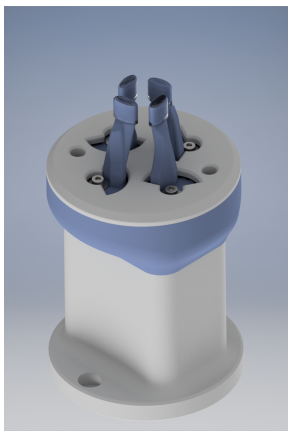

Figure 3.1: Rendered illustration of the assembled gripper

There are many possible ways to actuate the fingers, for example, a "rack & pinion" or similarly "spindle drive" that translates rotational motion into linear motion. These methods can be useful but can complicate the overall design. They require tight tolerances and can be difficult to produce with additive manufacturing.

In the case of grabbing cylindrical objects, it is possible to take advantage of the symmetric geometry and instead use a simpler and more robust mechanism. The mechanism is described in the section 3.1.1.

3.1.1 Gearbox

To convert the motor's rotation into a movement capable of gripping a cylindrical object, another rotational mechanism can be employed. All rotational motion can be described by the combination of two linear movements, a sine and cosine projection of the rotational motion. See figure 3.2.

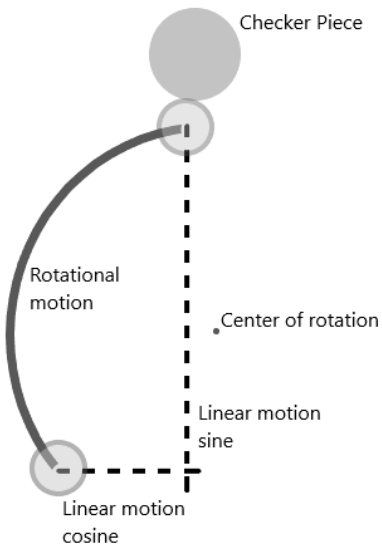

Figure 3.2: Illustration of rotational motion as a grabbing motion

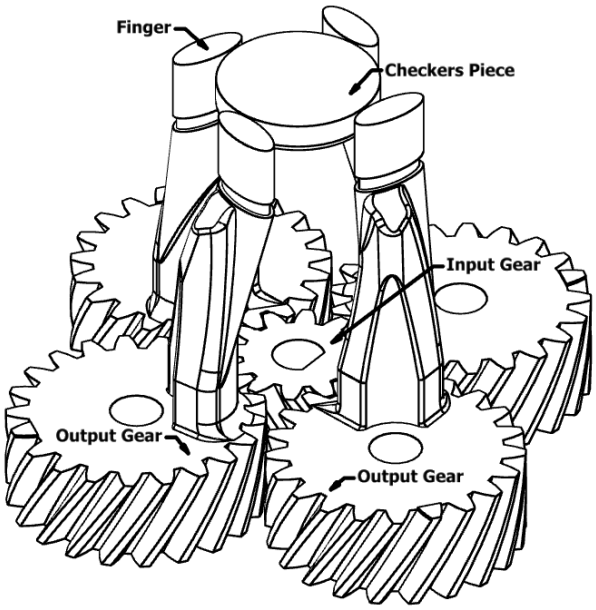

Figure 3.3: Illustration of the complete gearbox gripping a checkers piece

Instead of trying to translate to a linear motion, which is complicated, the linear part of a rotational motion can be used. The linear movement perpendicular to the object works to close in and move out, while the linear movement tangential to the object works to move around the perimeter, which can be considered irrelevant, because of the symmetric features of cylinders.

The transmission of movement from the motor to the fingers is done by attaching the fingers to the output gears and arranging the output gears around the input gear. The input will be attached directly to the motor shaft and function similarly to a “sun gear”, and the output gears like “planet gears”, in a planetary gearbox. The difference between this gearbox and a planetary gearbox is the “planet gears” are fixed instead of orbiting the “sun gear”. This means the output of the gearbox is the rotation of each “planet gear”. The size and teeth count of each gear are mostly irrelevant. The important features of the gears are:

1. Having a large module. The size of the gear teeth dependent on diameter and pitch.
2. The input gear being as small as possible while still leaving space for the surrounding gears.

This was achieved using the CAD software Autodesk Inventor and the built-in gear generator. The finger is then modeled on top of the output gear, making it one solid part.

The gears are 3D-printed with a high perimeter count, meaning the walls of the print are multiple layers thick, and the teeth of the gears are effectively solid plastic. This means the gear will remain strong even with infill relatively low, ~15%. Because of the size and strength required of the input gear, it is printed solid.

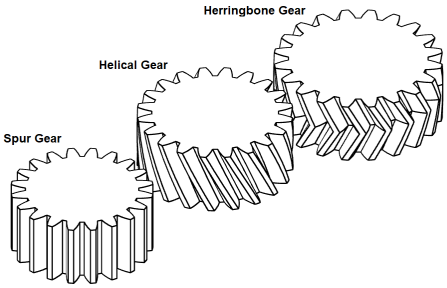

Figure 3.4: Illustration of different types of gears

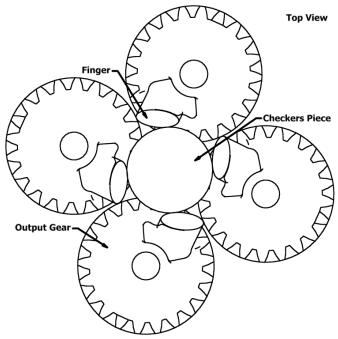

Figure 3.5: Illustration of the complete gearbox as seen from the top

Gear type

The simplest possible translation of motion is with the use of two spur gears meshing together. One gear is attached to the input axle, and the other gear to an output axle. When the input rotates, the output will rotate in the opposite direction with a speed and power proportional to the ratio of teeth of the gears. Spur gears have a rough transition from teeth to teeth. Teeth suddenly meet at a line contact across their entire width, causing stress and noise. Making the characteristic whine of spur gears at high speeds[5].

A type of gear with a smoother transition exists, called helical gears. The teeth are not parallel to the axis of rotation, like the spur gears, but set at an angle. This means the teeth of the gear first make contact at a single point. The contact moves and grows gradually across the tooth face to a maximum, then recedes to a single point until the teeth break contact. This reduces the sudden impact forces and wear on the gear.

The disadvantage of helical gears is the thrust along the axis of the gear resulting from the angle of engagement. However, this issue can be circumvented by using a herringbone gear, two helical gears that are mirrored and joined together. These gears have the advantages of helical gears and no axial thrust, they also provide self-alignment. Herringbone gears are also preferred when using additive manufacturing, but a disadvantage is the interlocking pattern sometimes requires simultaneous assembly. This will require the complete disassembly of the gripper gearbox in case one gear needs to be replaced. This can be a big time sink, especially when prototyping or testing. Therefore, helical gears have been chosen for their advantages over both spur gears and herringbone gears.

Lubricant

Even with the gradual contact of the helical gears, the plastic of 3D-printed gear will still be worn away quickly due to friction. This can be minimized by using a lubricant to reduce friction and wear on both the gear teeth and the contact surfaces such as axles and washers.

The lubricant to be used must be compatible with the 3D-printed plastic of the gears and housings. The plastic in question is Polylactic acid (PLA), a biodegradable polyester made from fermented plant starch, and therefore not very resilient to most chemicals[6]. This means the lubricant needs to be quite inert or at least non-oil-based.

Many cheap chemically inert lubricants exist. Examples are PTFE grease (aka. Teflon™ grease), silicone grease, lithium grease, paraffin wax, or graphite dust[7].

Out of these, silicone grease was chosen for its adhesion, chemically inert qualities, and low cost. The specific choice was sourced from RS components and the datasheet can be found in A.6.

3.1.2 Fingers

In the design of the fingers for the 3D-printed gripper, several features are considered. The following list highlights the main feature considerations:

- Finger quantity
- Finger strength
- Fingertips

Quantity

The first gripper was designed with three fingers, the minimum needed to securely hold checkers pieces. However, practical testing revealed a problem: the three-finger design risked colliding with surrounding pieces during pickup, complicating the gripper's orientation. Increasing the number of fingers to four solved this issue. This adjustment ensured the robot consistently positioned itself at the adjacent white squares, reducing the risk of collision since pieces are only on dark squares.

Strength simulation & test

A certain finger strength is required to withstand the force to pick up a checkers piece. By using software to simulate the force exerted on the fingers, and putting the manufactured finger to a test, it will ensure they can endure the necessary strength to pick up a checkers piece throughout a full game.

Having used Autodesk Inventor to design the gripper and gearbox, it was possible to use simulation to aid in the design of the fingers. This also made it possible to insert the parameters of the DC brush motor and simulate the peak forces from the output of the gearbox during typical operation.

With the use of Inventor's Dynamic simulation environment, two simulations were produced. One where the motor would spin only a single finger to collide with a fixed object at the fingertip, so a worst-case scenario. A second simulation, where the motor would close all four fingers around a checkers piece, and distribute the forces evenly between the fingertips. The forces on the points of contact were then graphed over time, and the outputs of the simulations can be seen in figure 3.6 and 3.7. A video render of the simulation has been uploaded to YouTube and can be found here: https://youtu.be/VRue0xD46xk

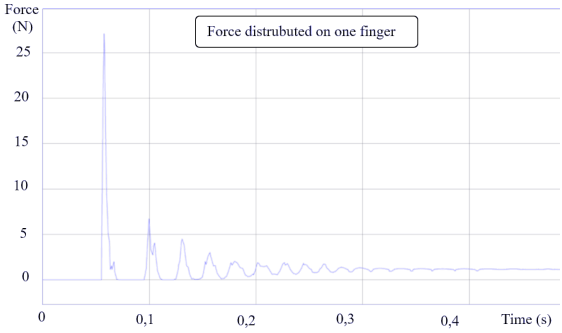

Figure 3.6: Simulation of the force exerted on one finger colliding with a fixed object

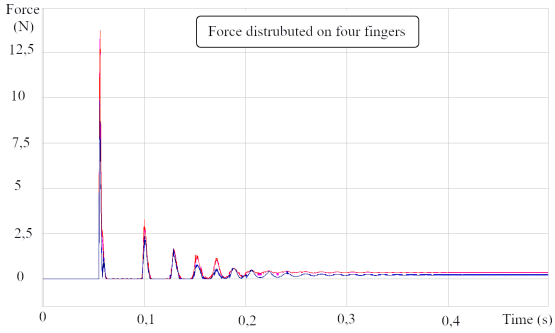

Figure 3.7: Simulation of the force exerted on four fingers colliding with a checkers piece

The peak forces calculated were a little over 27 N [Newtons] on a single finger. With the four fingers, each represented by a color, each finger can experience up to approximately 13 N. These are the maximum numbers that the motor and gearbox can output, so the fingers should be designed to withstand at least 27 N.

To further evaluate the integrity of the 3D-printed fingers, different tests were performed to ensure the fingers could support forces applied by the motor and the robot. The test can be seen in A.4.

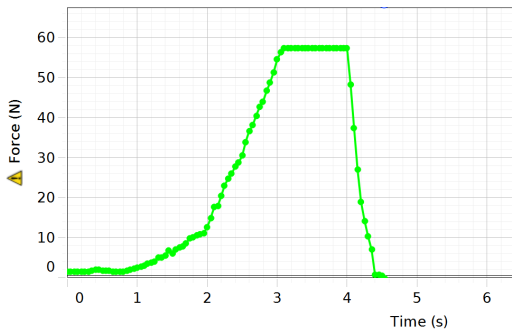

Figure 3.8: Collision with a bigger diameter than the checkers piece.

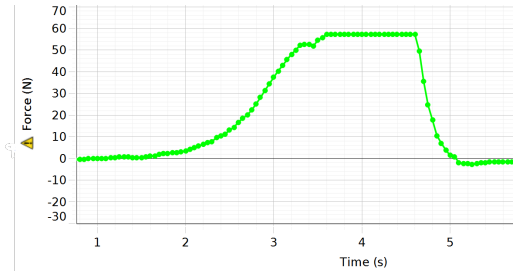

Figure 3.9: Collision when opening the gripper

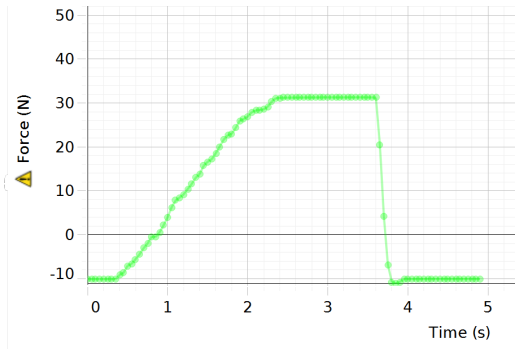

Figure 3.10: Collision with work surface

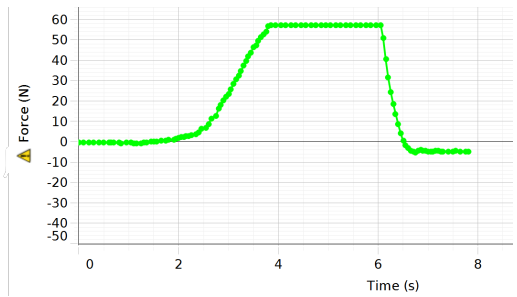

Figure 3.11: Collision with standard checkers piece

Due to equipment limitations, forces up to 50 N could be measured, causing the clipping in the graphs. All tests showed the fingers withstanding 50 N without damage. Notably, the collision test with the board measured only 35 N, a discrepancy attributed to the Pasco force sensor's limitations.

The fingers could withstand forces far greater than the worst-case simulation, and therefore the design can be considered a success.

Fingertips

The checkers pieces were constructed based on standard measurements: 4 mm in height and 20 mm in diameter. These dimensions guided the finger design. If the fingertips were too short, grasping multiple pieces would be difficult. If too long, the structural integrity could be compromised, risking breakage at the junction between the finger and fingertip.

The fingers and gears were designed as one homogeneous part, leveraging the advantages of 3D printing for added strength and simplicity. Creating them separately was possible but would have unnecessarily complicated the design and assembly process.

Since the fingers were most susceptible to breakage, maintaining simplicity ensured that replacing a broken finger would be quick and straightforward.

Additionally, rubber tips were designed for the plastic fingers to assist in picking up multiple pieces, such as a king checker.

3.1.3 Gripper Assembly

The gripper is constructed of two main segments. The motor housing and the gearbox housing. Within each 3D-printed segment, there is included steel hardware for assembly and reinforcement. The segments were designed to interlock together for extra strength and rigidity.

The gripper was also designed with cobot functionality in mind, therefore avoiding sharp edges and small surfaces as much as possible. Even designing a rubber fingertip to put over the plastic finger. All this helps uphold the cooperative functionality of the robot used, a UR cobot, and the use case of playing checkers with a human, interacting with the robot.

The complete assembly contains the motor, 8 3D-printed parts, and a small assortment of machine screws, washers, and nuts.

A video render of the exploded assembly view can be found on YouTube here:
https://youtu.be/uJsoO2M3YDw

Motor housing

The motor housing is the largest piece of the entire assembly and is also the part mounted to the robot flange. The body of the housing is rectangular, allowing for access to the two mounting bolts that fit the adapter plate used by the SDU Robotics lab. The housing is designed with thick walls and substantial strength in case of impacts with the work environment.

The housing is hollowed out to a loose fit with the motor, so the motor can be inserted into the housing and be protected. The motor is mounted with its flange to the gearbox housing and then inserted into the motor housing, where the wires are routed through a small hole at the bottom.

The gearbox housing is bolted onto the motor housing using two M3 bolts, that screw into some embedded nuts in the motor housing.

The embedded nuts are completely enclosed in the 3D-print. This is done by modeling a friction-fit hole of the nuts in the model, then slicing the model for 3D printing and adding a pause on the last layer before the hole is covered. If everything works correctly, the nuts will be covered by the 3D-printed plastic and will ensure an increased strength to the bolted joint compared to cut plastic or 3D-printed threads.

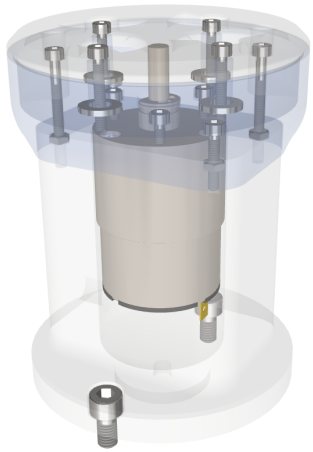

Figure 3.12: Illustration of the embedded hardware and motor of the assembled gripper

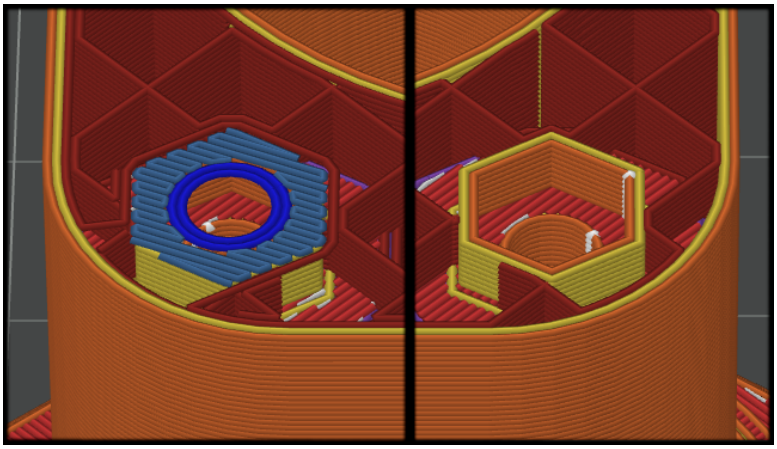

Figure 3.13: Illustration from BambuLab Slicer of a hole for an embedded nut

Gearbox housing

The gearbox housing is the most complex of all the parts, having the most geometries. It interfaces with the motor housing with a socket feature that is a slight friction-fit, which is used for locating and reinforcing the connection with the motor housing.

The output gears are mounted on small and fragile 3D-printed shafts, which is not ideal. This is remedied by extending the M3 mounting screws that are threaded into the plastic and are used for holding the output gears down, so they also function as a shaft reinforcement. Throughout the entire prototyping and testing process, a gear shaft has never failed. There are also metal washers at both ends of the shaft, used for reducing friction and wear on the gears.

The entire gearbox is also filled with lubricant and therefore should be covered with a lid. The lid is mounted to the embedded nuts in the motor housing with socket bolts that span through the gearbox housing. Completing the assembly process.

The lid also functions as an endstop for the gripper fingers. Making it possible to determine if the fingers are in a closed or open position with nothing more than current sensing. The range of motion of the gripper fingers can be modified by changing the gearbox lid.

3.2 Motor

The motor used in the gripper is a brushed direct current (DC) motor. These motors function by having the DC voltage energize a coil in the stator and a pairing coil in the rotor to generate a magnetic field that pulls on the rotor. When the rotor has rotated enough that the magnetic field is no longer effectively pulling, a couple of contacts on the rotor disconnect from the previous rotor coil and switch over to the next set.

This is done automatically by the internal carbon brushes of the motor, that provide electrical connection to the contacts on the rotor. Reversing the polarity of the DC voltage will also reverse the rotational direction of the motor. Pulsing the DC on and off will affect the speed at which the motor turns, slowing it down.

These motors are quite cheap and functional, but not very accurate or precise. With the motor's integrated gearbox slowing down the speed of the output, it is possible to have a lot of torque and some positional control over the motor.

Using a simple H-brigde driver circuit, it is possible to control the motor and also the direction. This is already integrated with the microcontroller board and can be controlled using PWM from the microcontroller. This is described in section 2.4.2.

Discussion and conclusions 4

4.1 AI methods

4.1.1 MinMax algorithms shortcomings

Since there exist 500,995,484,682,338,672,639 possible game states, it isn't viable to search through every possibility as mentioned in section 2.2.1. This means that to calculate a move within an acceptable time limit, the search depth has to be decreased, and because of this, the algorithm has to use the mentioned parameters. These parameters can be hard to determine without sufficient knowledge of the game that is being played. Therefore, an additional method to decrease the computations required has been utilized, called alpha-beta pruning, making it possible to explore further within the same time frame.

Although the alpha-beta pruning makes the algorithm faster, it still isn't viable to use the AI at high depths. An improvement to the alpha-beta pruning would be to go through the moves in a sorted order, starting with the move that is presumed to be the best. The algorithm that is used in this project finds all possible moves starting at the top-left of the checkerboard, checking all possible moves for the squares to the right before going to the next line.

Assuming moves in the middle of the board are generally better than those at the corners, a more effective method for iterating through moves in the MinMax algorithm is to start from the center of the move vector or to iterate through the board in a spiral pattern beginning from the center.

Even with improvements to the alpha-beta pruning, it will still be necessary to set up parameters to award different game states with points. Ten parameters have been utilized, but to improve the algorithm in accurately determining the best course of action, even further parameters might be necessary. Such parameters could be:

1. Amount of moves possible.
2. Game phase. The different parameters count for a different amount of points depending on the beginning, middle, or end phase.
3. Piece location. The middle two spaces of the backline can block an enemy from getting a king piece.

Another problem that occurs with the algorithm is often found at the end phase of the game, where a winning move might take more than the depth allows the algorithm to search. If the depth of the algorithm is set to 7, then it means that the algorithm will calculate 4 moves ahead for the current player, which can result in a problem if a winning setup requires any more than that. This is often the case when there are few pieces left, and they are at opposite ends of the board.

4.1.2 Evolution of the perfect algorithm

Since running the algorithm is demanding and takes a long time, the AI has been trained to search at a depth of five. This depth has been chosen since it was the highest depth that could play numerous games within a reasonable time. Because the search depth is set to five, the AI will be playing less optimally, than if it was set to a higher number.

The points given for the parameters used in this project have been found by letting the evolution program iterate through 55 generations. This has been trained by having a computer run calculations for 72 hours non-stop. Therefore, to perfect the AI would require a lot of time.

Since the mutation rate decreases with every generation, it means a major change, like the one that can happen in the first generations, would take multiple generations later on.

The time and power needed to run this simulation have been a constraint for further testing of different chances to evolve, evolution rates, amount of children, etc. Another aspect that hasn't been tested, is the tournament's way of evolving a model that is optimized against a player. The way the tournament is set up, an AI is awarded for having evolved in a way that makes it optimized to play against another AI. Since the MinMax algorithm plays differently than a normal player, by calculating ahead with predefined parameters, this optimization might not be the best against a human.

4.1.3 Improvement of database

The database component of the project ended up being a minor version of what could significantly enhance gameplay. By the project's conclusion, the database contained only a small amount of data on boards and moves, which restricted the AI's development and performance. A study, working on solving checkers, was discovered. The study ran dozens of computers working around the clock trying to solve for all moves in checkers. They had been working from 1989 to 1997, and again from 2001 to 2007 completing their goal. Meaning it took them 14 years with multiple computers [1].

```
Game played at depth: 7
Game took: 494028 milliseconds
The Database AI wins: 23
The MinMax AI wins: 0
Draws: 2
Player 1 wins! No more red pieces
Game Ended
```

Figure 4.1: Database win rate over MinMax

Another factor to improve is how a move is chosen. The method of choosing the move with the highest win rate should ensure that the AI picks the best move. However, the move may result in multiple counter moves, which the AI hasn't experienced yet. A separate algorithm could be made to check for the opponent's next move, to not risk setting up a good move from the opponent. Another possibility would be to calculate further ahead, finding the path that yields the highest chance of winning, like the MinMax.

4.2 Improving the computer vision algorithm

As briefly mentioned in section 2.5.1, the playing surface is regarded as a 2D surface as seen from the camera. Although this choice simplifies the process, it leads to minor deviations in the real positions calculated from computer vision. This happens because different places of the playing surface, all have different distances to the camera, which are not accounted for. Furthermore, the deviations worsen the further away from the camera the board is placed. When the camera locates the playing pieces and the board, the size of the board, as seen in 4.2, is relative to the distance between the board and the camera.

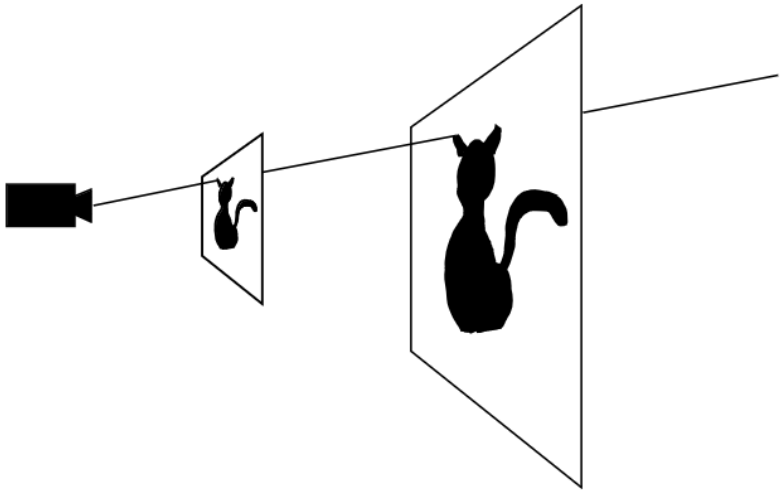

Figure 4.2: Distance from camera to object [8]

The camera and playing surface will never be perfectly aligned. When looking at the board from an angle, the squares will no longer appear quadratic, which again contributes to further deviations 4.3.

This can be fixed with more OpenCV functions. The process includes taking multiple pictures from different angles and using the findChessboardCorners() function on all of them. That way a reference point is created, which can be put into the calibrateCamera() function. This does however distort the final picture, which has to be rectified as well [8].

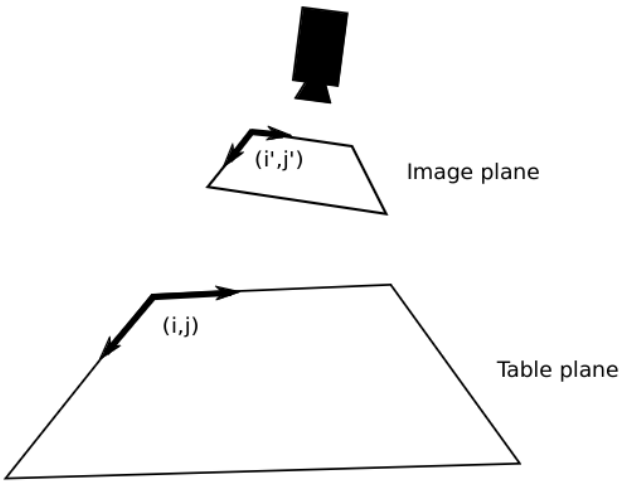

Figure 4.3: Camera with non-direct angle to playing surface [8]

Implementing both methods would require further research in computer vision and the OpenCV library, which for the project's deadline was not possible. It does also pose a problem in the setup phase, since many cameras would be needed, or the user would have to move the camera, to take the pictures from different angles. Therefore, the priority between precision and ease of use should be reevaluated.

4.3 Hardware Improvements

The motor used to complete the objective has a lot of torque and a large amount of inertia, therefore does not stop instantaneously. This causes two problems: Detecting when a checkers piece is picked up and opening the gripper.

Detecting piece quicker

Not detecting a piece properly can potentially cause the gripper to skip past the piece, twisting and bending the fingers enough to drop the piece.

This is caused by the material being flexible and the assembly not being constrained tightly enough. The gears have some "backlash"[9], allowing for possible movement at the tip of the finger. This adds up to causing a delay in the detection of a checkers piece, as the large spike in current consumption is not immediate. The current consumption gradually grows until the fingers and gears have no leeway.

Remaking the design with tighter tolerance gears and with shorter and more rigid fingers, decreasing flexibility, could improve the piece detection. The tip of the fingers could also have a different profile with a better fit when gripping a checkers piece.

Stopping the motor faster

The fingers meet the endstop and detection is not a problem. However, the large amount of inertia in the motor combined with the fragility of the input gear's D-key could hinder the functionality of the gripper.

Because of the small metal axle of the motor and its D-key, there is only a limited surface area to engage with. The metal axle easily deforms the soft plastic of the 3D print. Causing the motor axle to freely rotate in the gear's damaged D-key.

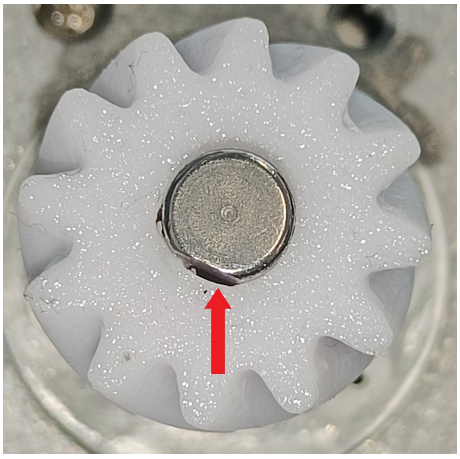

Figure 4.4: Broken D-key on motor shaft

A solution could be integrating a metal plate in the 3D-print as the contact face of the D-key, or getting an entirely metal input gear. This would greatly extend the lifetime of the input gear and the rigidity of the gearbox, solving some of the previously mentioned problems.

Another solution could be switching the motor direction at collision, and rotating the motor in the opposite direction as a method of braking. This idea however was abandoned due to the possibility of it causing critical damage to the gears in the motor.

The solution currently used to work around the issue is lowering the duty cycle, causing the motor to not have as much inertia when stopping. This, however, comes at the cost of time, making each operation of the gripper slower.

In order to work at higher speeds, the input gear needs reinforcement and the motor needs a means of braking.

4.4 Conclusion

The objective of this project was to develop a system utilizing a Universal Robot 5 CB3 (UR5 CB3) for playing checkers and to iteratively improve its performance. All outlined goals were successfully achieved.

Firstly, communication with the UR5 CB3 was established, allowing precise positioning and control. This enabled the integration of the robot with the subsequent components of the system.

Secondly, the detection of the playing surface was accomplished using computer vision techniques. The system consistently recognized the checkerboard, ensuring reliable gameplay with minimal human intervention.

The design and implementation of a gripper capable of handling checkers pieces was created. The gripper was engineered to grasp and move the pieces, be fast to manufacture, and be easy to assemble. It demonstrated both the durability and precision required for smooth gameplay.

The AI was programmed to learn the game of checkers and improve its strategies iteratively. By analyzing previous games, the AI system continuously enhanced its gameplay, showing significant improvements over time. This iterative learning process allowed the AI to develop strategies.

Overall, the project successfully met all its objectives. The UR5 CB3 robot was effectively transformed into an autonomous checkers player, capable of learning and improving its performance. The integration of communication protocols, surface detection, gripper design, and AI learning systems collectively contributed to the project's success. This project demonstrates some of the potential of robotics and artificial intelligence in recreational activities.

A video showcasing the final product can be found on YouTube with this link:
https://youtu.be/80n-h7Ir8S8

Appendix A

A.1 Source Code

All code written for this project can be found in the attached zip-folder in GameCode folder

A GitHub repository for the project can also be found at:
https://github.com/akmun23/Autonome_Robotter/

A.1.1 Checkers Game

See validmoves.cpp in attached files

A.1.2 Minimum Maximum Algorithm

See alphabeta.cpp in attached files

A.1.3 Database Algorithm

See checkersdatabase.cpp in attached files

A.1.4 Microcontroller Program

See Atmega644PACom.c in attached files

A.1.5 Communication with the Microcontroller

See AtmegaCom.cpp in attached files

A.1.6 Computer Vision Program

See vision.cpp in attached files

A.1.7 GUI Program

See DrawFunctions.h in attached files

A.1.8 Robot Code

See robot.cpp in attached files

A.2 GUI Flowchart

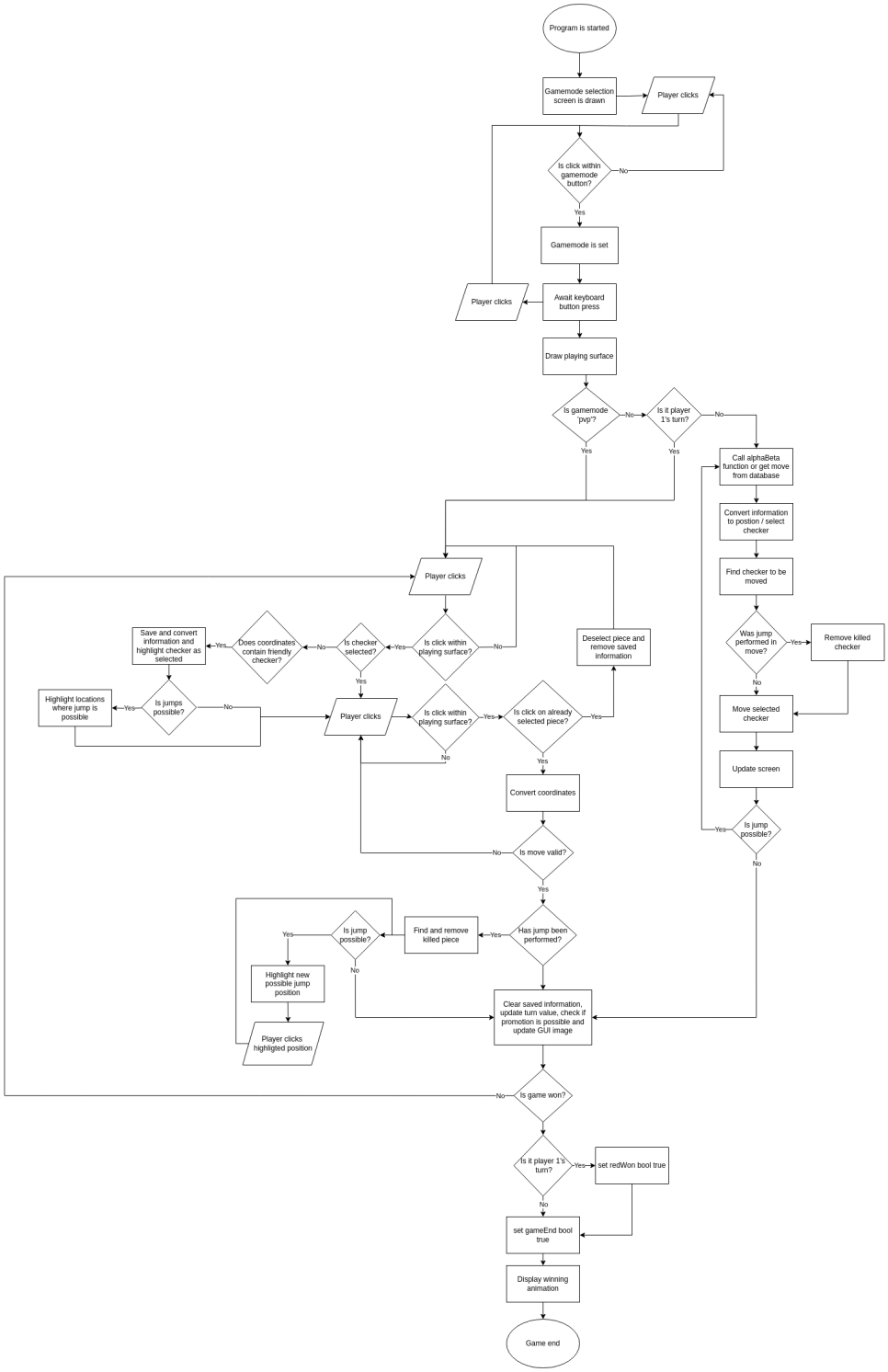

Figure A.1: Flowchart of GUI program

A.3 Atmega

A.3.1 Reading and Writing

```
UBRR1 = 118;
UCSR1B = (1 << RXEN1) | (1 << TXEN1) | (1 << RXCIE1);
UCSR1C = (1 << UCSZ11) | (1 << UCSZ10);
```

Figure A.2: ATmega reading and writing setup

The first value to be written is to the UBRR1 register, which sets the baud rate for the UART communication. The value of 118 has been calculated for the baud rate of 9600 with the following formula.

$$UBRR1 = \frac{F_{CPU}}{16 * Baudrate} - 1 \tag{A.1}$$

where $F_{CPU} = 18278281Hz$ and baudrate = 9600.

Then the 3 registers used for reading and writing from the USB-B port, UCSR1A, UCSR1B, and UCSR1C.

Register UCSR1A
The only bit in use is bit 5 called UDRE1. This bit has the purpose of being set high when the transmit buffer is empty. This is used in the write function, so it waits for the transmit buffer to be clear and then transmits its message.

Register UCSR1B
Bit 7 is set to 1 and enables interrupts when a message is received.
Bit 6 is set to 0 it's the same as bit 7 but for writing and will not be needed.
Bit 5 is set to 0 to disable interrupts caused by an empty data register.
Bit 4 is set to 1 and enables reading.
Bit 3 is set to 1 and enables writing.
Bit 2 is set to 0 and is combined with bits 2 and 1 in UCSR1C to make the 8-bit frame.
Bit 1 is set to 0, which is the 9-bit of a received message, but since 8-bit messages are utilized this won't be needed.
Bit 0 is set to 0 since it is used to send 9 bits of data.

Register UCSR1C
Bits 7 and 6 are set to 0 setting it to Asynchronous UART.
Bits 5 and 4 are set to 0 disabling parity mode.
Bit 3 is set to 0 defining 1 stop bit.
Bits 2 and 1 are set to 1 combined with bit 2 in UCSR1B defining the UART frame to 8-bit.
Bit 0 is not relevant since it is only used in synchronous mode.

A.3.2 PWM

```
TCCR1A = (1 << COM1B1) | (1 << WGM11) | (1 << WGM10);
TCCR1B = (1 << CS10) | (1 << WGM13) | (1 << WGM12);
OCR1A = 100;
TIMSK1 = (1 << TOIE1);
```

Figure A.3: ATmega PWM setup

To set up the timer controlling the PWM, 3 registers are used. TCCR1A, TCCR1B and TCCR1C. For control, another 2 registers are used and a couple of interrupts handle output. The OCR1A register is used as the top of the counter. This is set to 100. This makes generating the PWM signal easier since the OCR1B register is used for output control and can just be set to 60 and that equates to a 60% duty cycle output signal. An illustration of the functionality can be found in figure A.4.

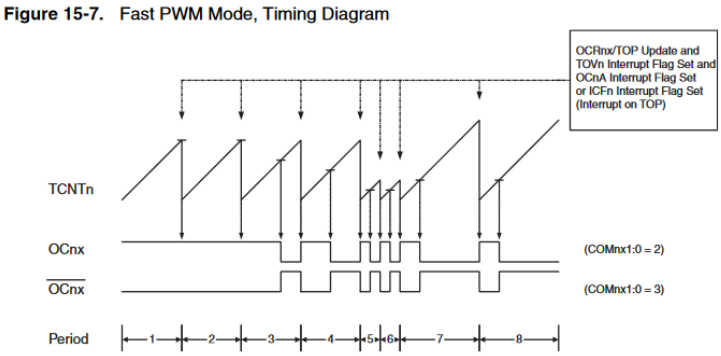

Figure A.4: ATmega PWM Function

Register TCCR1A.

Bits 7 and 6 are set to 0 since the pin for OC1A is not in use.

Bit 5 is set to 1. It's combined with bit 4.

Bit 4 is set to 0 and combined with bit 5 defines the pin OC1B to work in non-inverting mode.

Bits 3 and 2 are reserved.

Bits 1 and 0 are set to 1 and will be combined with bits 4 and 3 of TCCR1B and explained there.

Register TCCR1B

Bit 7 is set to 0 since the ICP pin is not used.

Bit 6 is set to 0 for the same reason as bit 7.

Bit 5 is reserved.

Bits 4 and 3 are set to 1 combined with bits 1 and 0 of TCCR1A making the timer mode of fast PWM with a top value of OCR1A. The output is set high when under OCR1B, and the TOVN flag is set high at the top.

Bits 2 and 1 are set to 0 and will be combined with bit 0.

Bit 0 is set to 1 and combined with Bits 2 and 1 setting the clock to have no prescaler.

Register TCCR1C

This register only has bits 7 and 6 that are not reserved. These are used for force output which will not be used and therefore not used.

Register TIMSK1

Bits 7, 6, 4, and 3 are reserved

The rest of the bits enable different interrupt flags for the timer, and here the use of bit 0 will be implemented.

Bit 0 sends an interrupt when the timer overflows.

A.3.3 ADC

```
ADMUX = (1 << REFS0) | (1 << MUX0);
ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
```

Figure A.5: ATmega ADC setup

To set up the ADC two registers will be used the ADMUX register and ADCSRA register which will be set up as follows

Register ADMUX

Bit 7 is set to 0 and will be combined with bit 6

Bit 6 is set to 1 and combined with bit 7 this defines the voltage reference to be at AVCC where there are 5V and an external capacitor at the AREF pin

Bit 5 is set to 0 since the result does not need to be left adjusted

Bits 3, 2, and 1 are set to 0 and will be combined with bit 0

Bit 0 is set to 1 and combined with bits 3 to 1 sets the ADC1 pin to be the single-ended input

Register ADCSRA

Bit 7 is set to 1 Enabling the ADC

Bit 6 is set to 1 when a reading is to be made

Bit 5 is set to 0 to disable auto triggering

Bit 4 is a flag that will be set when a conversion is complete.

Bit 3 is set to 0 since interrupts will not be used for the ADC

Bit 2, 1, and 0 are set to 1 setting the division factor of the ATmega's frequency

This division factor is set so that the frequency of the ADC signal lies between 200kHz and 50kHz

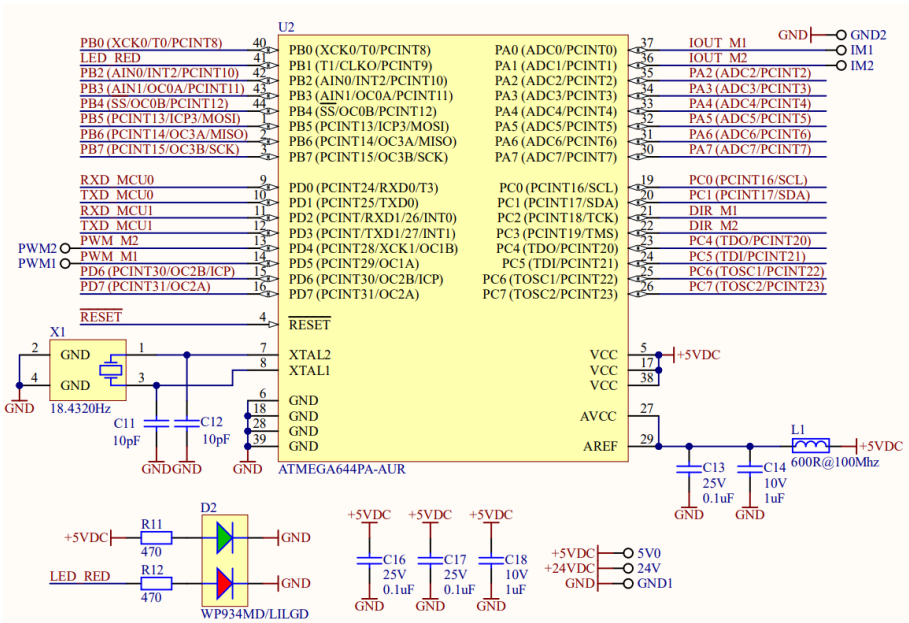

Figure A.6: ATmega pin overview

A.3.4 PC Script

```
tty.c_cflag &= ~PARENB;

tty.c_cflag &= ~CSTOPB;

tty.c_cflag &= ~CSIZE;
tty.c_cflag |= CS8;

tty.c_cflag &= ~CRTSCTS;

tty.c_cflag |= CREAD | CLOCAL;
tty.c_lflag &= ~ICANON;
tty.c_lflag &= ~ECHO;
tty.c_lflag &= ~ECHOE;
tty.c_lflag &= ~ECHONL;

tty.c_lflag &= ~ISIG;

tty.c_iflag &= ~(IXON | IXOFF | IXANY);

tty.c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL);

tty.c_oflag &= ~OPOST;
tty.c_oflag &= ~ONLCR;

tty.c_cc[VTIME] = 0;

tty.c_cc[VMIN] = 1;

cfsetispeed(&tty, B9600);
cfsetospeed(&tty, B9600);
```

Figure A.7: PC script for communicating with ATmega [10]

A.4 Test of finger strength

See file “GripperTest.pdf” in the 3D-files / Test folder.

A.5 Gripper 3D files

All 3D-files used to print the gripper can be found in attached files as standard .stl files.

A.6 Silicone Grease Datasheet

Hosted by RS components and can be found at:
https://docs.rs-online.com/3b37/0900766b8156d7ff.pdf Accesed on 10-05-2024

Bibliography B

[1] The Chinook project. Accesed on 21-05-2024. [Online]. Available: http://webdocs.cs.ualberta.ca/~chinook/databases/checker_positions.html

[2] UR RTDE documentation. Accesed on 22-05-2024. [Online]. Available: https://sdurobotics.gitlab.io/ur_rtde/api/api.html/

[3] "Data teknik with Ali Sahafi lecture 8."

[4] OpenCV documentation. [Online]. Available: https://opencv.org/

[5] Gear - Wikipedia. Accesed on 22-05-2024. [Online]. Available: https://en.wikipedia.org/wiki/Gear

[6] Polylactic acid - Wikipedia. Accesed on 22-05-2024. [Online]. Available: https://en.wikipedia.org/wiki/Polylactic_acid

[7] ABS, PLA best compatible lubrications - SeeMeCNC. Accesed on 15-05-2024. [Online]. Available: http://download.seemecnc.com/forum/viewtopic.php?t=1975

[8] T. M. Iversen, "Calibration of the vision setup in robolab," *See ComputerVision.pdf in attached files*, 2019.

[9] Backlash - Wikipedia. Accesed on 22-05-2024. [Online]. Available: https://en.wikipedia.org/wiki/Backlash_(engineering)

[10] Linux Serial Ports Using C/C++. Accesed on 24-05-2024. [Online]. Available: https://blog.mbedded.ninja/programming/operating-systems/linux/linux-serial-ports-using-c-cpp/

[11] Minimax - Wikipedia. Accesed on 23-05-2024. [Online]. Available: https://en.wikipedia.org/wiki/Minimax

[12] Alpha-beta pruning - Wikipedia. Accesed on 23-05-2024. [Online]. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning